



SEPRAN

SEPRAN ANALYSIS

USERS MANUAL

GUUS SEGAL

USERS MANUAL

January 2013

Ingenieursbureau SEPRA
Park Nabij 3
2491 EG Den Haag
The Netherlands
Tel. 31 - 70 3871309

Copyright ©1993-2013 Ingenieursbureau SEPRA.

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means; electronic, electrostatic, magnetic tape, mechanical, photocopying, recording or otherwise, without permission in writing from the author.

Contents

1 Introduction

- 1.1 Some definitions used in SEPRAN
- 1.2 Boundary conditions
- 1.3 The solution of coupled problems at one mesh
- 1.4 The standard SEPRAN input file
- 1.5 The SEPRAN environment file
- 1.6 Manipulation of constants and variables (subroutine COMPCONS)
- 1.7 Definition of general constants
- 1.8 Overview of the SEPRAN commands

2 The pre-processing part of SEPRAN

- 2.1 Introduction
- 2.2 Input for program SEPMESH from the standard input file
 - 2.2.1 Subroutine FUNCCOOR
- 2.3 Curve generators
 - 2.3.1 Subroutine FUNCCV
 - 2.3.2 Subroutine OWN_CURVE
- 2.4 Surface generators
 - 2.4.1 Surface generator GENERAL
 - 2.4.2 Surface generator RECTANGLE
 - 2.4.3 Surface generator QUADRILATERAL
 - 2.4.4 Surface generator COONS
 - 2.4.5 Surface generator PIPESURFACE
 - 2.4.6 Surface generator MESHUS
 - 2.4.7 Surface generator TRIANGLE
 - 2.4.8 Surface generator PARSURF
 - 2.4.9 Surface generator ISOPAR
 - 2.4.10 Surface generator PAVER
 - 2.4.11 Surface generator SPHERE
 - 2.4.12 Surface generator FRAMESURF
- 2.5 Volume generators
 - 2.5.1 Volume generator BRICK
 - 2.5.2 Volume generator PIPE
 - 2.5.3 Volume generator CHANNEL
 - 2.5.4 Volume generator GENERAL
- 2.6 Some examples of meshes generated by SEPRAN
- 2.7 Special input for program SEPMESH from the standard input file

3 The computational part of SEPRAN

- 3.1 Introduction
- 3.2 Description of the input for program SEPCOMP
 - 3.2.1 The main keyword START
 - 3.2.2 The main keyword PROBLEM

- 3.2.3** The main keyword STRUCTURE
- 3.2.4** The main keyword MATRIX
- 3.2.5** The main keywords ESSENTIAL BOUNDARY CONDITIONS
- 3.2.6** The main keyword COEFFICIENTS
- 3.2.7** The main keywords CHANGE COEFFICIENTS
- 3.2.8** The main keyword SOLVE
- 3.2.9** The main keyword NONLINEAR_EQUATIONS
- 3.2.10** The main keyword CREATE
- 3.2.11** The main keyword DERIVATIVES
- 3.2.12** The main keyword INTEGRALS
- 3.2.13** The main keyword OUTPUT
- 3.2.14** The main keyword BOUNDARY_INTEGRAL
- 3.2.15** The main keyword TIME_INTEGRATION
- 3.2.16** The main keyword CONTACT
- 3.2.17** The main keyword LOOP_INPUT
- 3.2.18** The main keyword EIGENVALUES
- 3.2.19** The main keyword CAPACITY
- 3.2.20** The main keyword INVERSE_PROBLEM
- 3.2.21** The main keyword REFINE
- 3.2.22** The main keyword NAVIER_STOKES
- 3.2.23** The main keyword PRESSURE_CORRECTION
- 3.2.24** The main keyword BEARING
- 3.3** Description of some function subroutines to be used
 - 3.3.1** Subroutine FUNALG
 - 3.3.2** Function subroutine FUNCSCAL
 - 3.3.3** Subroutine FUNCFL
 - 3.3.4** Subroutines FUNC1B and CFUN1B
 - 3.3.5** Subroutines FUNCOL and CFUNOL
 - 3.3.6** Function subroutines FUNCC1 and FUNCC3
 - 3.3.7** Function subroutine FUNCTR
 - 3.3.8** Function subroutine USERBOOL
 - 3.3.9** Subroutine FUNCCR
 - 3.3.10** Subroutine FUNCC2
 - 3.3.11** Subroutine FUNCVECT
 - 3.3.12** Function subroutines to get the values of constants and variables
 - 3.3.13** Subroutines to put the values of constants and variables in common CUSCONS
 - 3.3.14** Subroutines to get the positions of variables, constants and vectors in common CUSCONS
 - 3.3.15** Subroutine FUNCSOLCR
- 3.4** Description of the input for program SEPFREE
 - 3.4.1** Introduction
 - 3.4.2** Extra possibilities for the main keyword STRUCTURE
 - 3.4.3** The main keyword ADAPT_MESH
 - 3.4.4** The main keyword ADAPT_BOUNDARY
 - 3.4.5** The main keyword STATIONARY_FREE_BOUNDARY
 - 3.4.6** The main keyword INSTATIONARY_FREE_BOUNDARY
- 3.5** Description of some special files that may be used

- 3.5.1** Description of the file with the nodal point numbers
- 3.5.2** Description of the file with the nodal point numbers and corresponding values
- 3.5.3** Description of the file with the element numbers and corresponding values
- 3.5.4** Description of the file with the electrode pairs and corresponding capacities
- 3.6** Parallel computing
 - 3.6.1** The command `sepmakeparmesh`

4 How to program your own element subroutines

- 4.1** Introduction
- 4.2** Subroutine ELEM
- 4.3** Subroutine ELEM1
- 4.4** Subroutine ELEM2
- 4.5** Subroutine ELDERV
- 4.6** Subroutine ELCERV
- 4.7** Function subroutine ELINT
- 4.8** Subroutine ELSTRM

5 The postprocessing part of SEPRAN

- 5.1** Introduction
- 5.2** General input for program SEPPOST
- 5.3** Print commands for program SEPPOST
- 5.4** PLOT commands for program SEPPOST
- 5.5** Special commands for time-dependent problems with respect to program SEPPOST

6 Some examples of complete SEPRAN runs

- 6.1** Introduction
- 6.2** Examples of elliptic equations with one degree of freedom per point.
 - 6.2.1** An example of a simple potential problem.
 - 6.2.2** An example of a simple potential problem with a user defined structure.
 - 6.2.3** An example of a simple potential problem with a user defined element subroutine.
 - 6.2.4** An example of a simple potential problem with a refinement of the mesh.
 - 6.2.5** An example of how to compute derived quantities and integrals in combination with a simple potential problem.
 - 6.2.6** An example of how to use periodical boundary conditions in R^2
 - 6.2.7** An example of how to use periodical boundary conditions in R^3
 - 6.2.8** An example of the manipulation of scalars
 - 6.2.9** An example of the use of the for loop
 - 6.2.10** An example of the computation of capacities
 - 6.2.11** An example of the solution of an inverse problem
 - 6.2.12** An example of the use of arrays in the input block constants
- 6.3** Examples of non-linear problems.
 - 6.3.1** An example of a simple Navier-Stokes problem.
 - 6.3.2** An example of a simple Navier-Stokes problem with a user defined structure.
 - 6.3.3** An example of a simple Navier-Stokes problem showing the use of the WHILE option in the user defined structure.
- 6.4** Examples of time-dependent problems.

6.4.1 An example of a simple heat equation.

6.4.2 An example of a simple heat equation with a user defined structure.

6.4.3 An example of the solution of a coupled set of time-dependent equations.

6.4.4 An example of a stationary equation solved by the limit of a time-dependent problem.

6.4.5 An example of a time-dependent equation coupled with a stationary equation

6.6 Examples of instationary free boundary problems.

6.6.1 An example of a simple Stefan problem

6.6.2 The dissolution of a disk-like particle in a disk-shape environment

6.6.3 The dissolution of two particles

6.7 Auxiliary examples.

6.7.1 An example of reading own data for postprocessing purposes.

6.8 Examples of eigenvalue computations.

6.8.1 Eigenvalues and eigenvectors for a potential problem in an L-shape.

7 References

8 Index

1 Introduction

In the SEPRAN USERS MANUAL it is described how problems with SEPRAN can be solved. It is supposed that the reader is already familiar with the SEPRAN INTRODUCTION.

The USERS MANUAL describes the complete input and usage of the main SEPRAN programs like SEPMESH, SEPCOMP and SEPPOST. Besides that, it is described how function subroutines and user element subroutines may be attached to these programs.

Besides the standard possibilities SEPRAN allows the user also to construct his own main program using the standard subroutines SEPRAN provides. For the description of such usage the reader is referred to the SEPRAN PROGRAMMERS GUIDE.

Finally the standard elements available as well as some examples of the usage of these standard elements can be found in the manual STANDARD PROBLEMS.

In the next sections some remarks concerning the usage of SEPRAN, the treatment of boundary conditions and so on is given.

Section 1.1 gives some general definitions used throughout the SEPRAN manuals. It is advised to read these definitions before consulting the rest of the manuals.

In Section 1.2 the various types of boundary conditions that may be used in SEPRAN are treated. This section also describes which actions should be taken in order to prescribe these types of boundary conditions.

Section 1.3 treats how to couple problems on one mesh. For example if one wants to solve the Navier-Stokes equations coupled with the temperature equation it is possible to solve the coupled Boussinesq equations containing velocities and temperature as unknowns. But it is also possible to solve first the velocity by the standard Navier-Stokes equations, and then the temperature from the heat equation. This process may be used repeatedly. In that case we have two (coupled) problems at one mesh.

In Section 1.4 some general remarks concerning the input file are given.

Section 1.5 treats the SEPRAN environment file. This file permits you to define defaults with respect to SEPRAN globally or in a subdirectory only.

Chapter 2 describes the complete input for the program SEPMESH including the three-dimensional input.

In Chapter 3 the complete input for program SEPCOMP is described.

How to program and add your own elements is the subject of Chapter 4.

Chapter 5 is devoted to the input for program SEPPOST.

Finally Chapter 6 treats some examples.

1.1 Some definitions used in SEPRAN

Although it is supposed that the reader is familiar with the finite element method (FEM), for the sake of completeness, a definition of what will be a FEM in this manual will be given.

General definition

- We consider a problem defined over a finite region Ω in R^n ($n = 1, 2, 3, \dots$), for example an interval in R^1 , a rectangle in R^2 .
 Ω is divided into subregions which will be called elements.
- In each element a number of nodal points is selected. In each nodal point some degrees of freedom are chosen, not necessarily the same for each point. The number of degrees of freedom is problem dependent.
- Characteristic is the definition of an approximate solution of the problem, that will be expressed in the selected degrees of freedom. In that way a system of equations arises. Depending on the problem this system can be linear, non-linear, or of differential type.
- An essential part of the FEM is the storage of the contributions of each element to this system of equations into an element matrix respectively an element vector. Using a well known "adding process", the system of equations is computed.

Now we shall give some of the definitions frequently used throughout the SEPRAN manuals.

Some definitions

Standard Element The region is divided into NELGRP different types of elements (see the introduction). Each group of elements is represented by a standard element. This is not an element of the mesh, but gives the structure of the group of elements it represents. For example it defines the shape of the elements, the number of nodal points, and in most cases also the number of degrees of freedom, the type of approximation, and the differential equation that is discretized.

Degrees of freedom degrees of freedom refer to the unknowns. Sometimes the degrees of freedom refer to all unknowns in the mesh, in other cases only the unknowns in a nodal point are meant.

Prescribed degrees of freedom Some boundary conditions explicitly prescribe the values of the unknowns. For example the temperature given in a part of the boundary is such a prescribed boundary condition. Boundary conditions of this type are called essential boundary conditions. The corresponding unknowns are referred to as prescribed degrees of freedom. The number of unknowns is reduced by these boundary values.

Arrays of the structure of the solution vector The vector that contains the degrees of freedom in the nodal points is called the solution vector. Vectors that have exactly the same degrees of freedom in the nodal points, are called vectors with the structure of the solution vector.

Arrays of special structure Not all quantities that are computed, will be computed in exactly the same points as the solution vector and with the same number of degrees of freedom in the nodal points. Suppose for example that the solution ϕ of a problem is computed in the three nodal points of a linear triangle. Let furthermore the gradient of ϕ : $\mathbf{u} = \nabla\phi$ be computed in the same nodal points. Since the gradient has two components, it can not be stored in an array with the structure of the solution vector. Therefore an extra type of arrays is introduced, the so-called arrays of special structure. These arrays are defined in exactly the same way as solution vectors, with the only difference that the number of degrees of freedom in the nodal points differs from that of the solution vector. See for example Figure 1.1.1

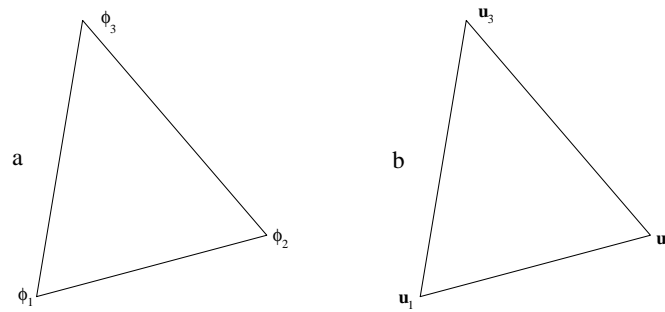


Figure 1.1.1: Standard elements: type a corresponds to solution vector, type b to array of special structure

Various types of arrays of special structure are permitted in one program. These arrays are defined in the input of program SEPCOMP, input block "PROBLEM", and different types get a different sequence number. For standard problems, as defined in the manual "Standard Problems", the possible arrays of special structure are defined, and no extension is possible.

Arrays of special structure can be used in most SEPRAN subroutines, however, they can never be the result of solution subroutines, nor is the large matrix based on a structure corresponding to such a vector of special structure.

Arrays that are defined per element Besides the arrays of special structure, where degrees of freedom are defined in nodal points, it is also possible to define quantities related to elements. For example when one stores the pressure in the centroid of an element, in fact this pressure is related to the element, and not to some nodal point. Degrees of freedom of this type are stored in a vector of the third category, the arrays that are defined per element. These arrays do not need any definition in the input, they are defined element-wise and it is supposed that the number of quantities is constant in each element. When an element has less than the maximal number of quantities related to elements, the remaining positions are supplied with zeros. Boundary elements, as defined in the input block "PROBLEM", are excluded from these arrays.

Boundary Elements Boundary elements are only used for special types of boundary conditions, the so-called natural boundary conditions. These elements do not influence the mesh, and hence also not the renumbering process. It is not permitted that a boundary element, belongs to more than one element, except when the boundary element is common for all the elements it belongs to. When a user wants to define special boundary conditions that require boundary elements connected to more elements, he must define these elements as regular elements (i.e. to be generated by the mesh generator), for example line elements, and not as boundary elements. These elements influence for example the renumbering process.

Coupled problems In some cases it is necessary to solve more than one equation separately on the same mesh. In many occasions the solutions of the separate equations influence the coefficients of the other equations and some iteration procedure may be necessary. If more than one problem must be defined on the same mesh we speak about coupled problems. To each of these problems a so-called problem sequence number is assigned. For each problem number a new problem may be defined. In Section 1.3 more about coupled problems is mentioned.

Element groups Each element with a different property is represented by a different standard element. Each of these elements are clustered in a series of elements, the so-called element groups.

Hence element groups are used to distinguish between elements with different properties.

Boundary element groups Each boundary element with a different property is represented by a different standard element. In the same way as element groups, for these elements boundary element groups are defined.

Remarks

1. Abrupt refinements in a mesh are suboptimal; ensure that neighboring elements do not differ too much in size.
2. *Compatibility of nodal points*

It is not necessary for the elements of neighboring submeshes to be of the same type. Although the package allows for nodal points on common boundaries of elements to belong to only one or few of them, the use of this facility is not recommended. In general nodal points on common boundaries must belong to all the elements that share these boundaries.

3. *Compatibility of degrees of freedom*

It is not necessary that common nodal points in different elements, have the same number of degrees of freedom. However, SEPRAN supposes that the common number of degrees of freedom, corresponds to exactly the same degrees of freedom. Hence it is allowed to have three degrees of freedom in a nodal point of an element, and to have only one degree of freedom in the same nodal point of an another element, but in that case the first degree of freedom in that nodal point is supposed to be the same for both elements. The user must be very careful when using this facility.

1.2 Boundary conditions

In this section the various possible boundary conditions in SEPRAN are considered. It is treated how these boundary conditions may be introduced in the program.

1.2.1 Essential boundary conditions

Boundary conditions are called essential if they prescribe the values of degrees of freedom at a boundary. The user must indicate in the input block "PROBLEM" on which boundaries which degrees of freedom are prescribed. The values of the boundary conditions must be filled with the aid of the input block "ESSENTIAL BOUNDARY CONDITIONS" or alternatively with the input block "CREATE".

1.2.2 Natural boundary conditions

Some boundary conditions give rise to the evaluation of boundary integrals. These boundary conditions are called natural. For a definition of which boundary conditions are natural for a specific problem the user is referred to the manual "Standard Problems". For standard natural boundary conditions, boundary elements must be introduced in the input block "PROBLEM". In some special cases, for example when the finite element method is coupled with an integral equation method in the outer region, the elements necessary for the evaluation of the boundary integrals overlap more than one inner element. In that case the user must use line elements or surface elements instead of boundary elements. These line elements and surface elements must be created by the mesh generator (SEPMESH). The actual evaluation of the boundary integrals is carried out in program SEPCOMP in the part building of matrices and right-hand sides (subroutine BUILD).

1.2.3 Periodical boundary conditions

In some problems periodical boundary conditions are prescribed on opposite boundaries. For example in Figure 1.2.1, the boundaries I and III may have periodical boundary conditions. In that case the corresponding degrees of freedom must be identified. This is done in the problem file.

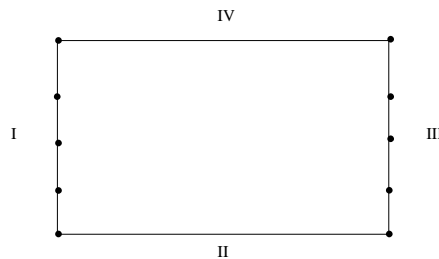


Figure 1.2.1: Periodical boundary conditions on sides I and III.

For an example of the use of periodical boundary conditions the user is referred to the Sections 6.2.6 (2D) and 6.2.7 (3D).

1.2.4 Essential boundary conditions not connected with degrees of freedom

Sometimes essential boundary conditions are prescribed that are not connected directly with the degrees of freedom of the problem. For example in Figure 1.2.2 the condition $\mathbf{u} \cdot \mathbf{n} = 0$ along the

skew boundary II (free surface condition) is an essential boundary condition.

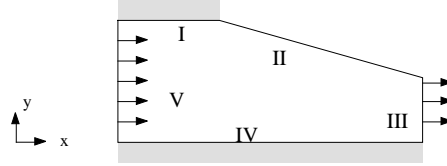


Figure 1.2.2: Free surface condition along boundary II

However, this condition does not prescribe the degrees of freedom u and v of the problem, but a linear combination of them. In order to incorporate this boundary condition it is necessary to make a local transformation of unknowns along boundary II and to introduce new unknowns u_n ($\mathbf{u} \cdot \mathbf{n}$) and u_t ($\mathbf{u} \cdot \mathbf{t}$) with \mathbf{n} the outward normal and \mathbf{t} the unit tangential vector. It is then possible to prescribe the unknown $\mathbf{u} \cdot \mathbf{n}$ without fixing $\mathbf{u} \cdot \mathbf{t}$.

The local transformations must be defined by the user in the input block "PROBLEM". See "local transformations". Essential boundary conditions must be introduced in the input block "PROBLEM" and the input block "ESSENTIAL BOUNDARY CONDITIONS" or "CREATE" for the transformed degrees of freedom. For output purposes as well as the parts of SEPCOMP that use solution vectors (for example solution of linear and non-linear equations and computation of derivatives) the original degrees of freedom are submitted to the user.

1.2.5 Boundary conditions of the type u is constant along a part of the boundary

When the constant is known this condition is an essential boundary condition and hence reduces to the boundary conditions given in 1.2.1 or un-chap-1.2.4 However, for some problems we have the boundary condition u is constant along a part of the boundary, with the constant unknown. This may be for example the case when we consider the flow in a channel with an obstacle in it. See Figure 1.2.3.

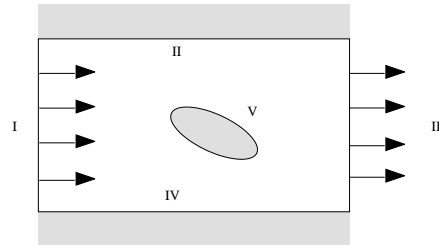


Figure 1.2.3: Flow in a channel with obstacle.

Suppose we want to compute the stream function, then in general we have given boundary conditions along the boundaries I, II, III and IV, but along boundary V we have the boundary condition Ψ is unknown constant. Boundary conditions of this type must be defined by the user in the input block "PROBLEM". See "UNKNOWNCONSTANT". (3.2.2.6)

1.2.6 Boundary conditions of the type $\psi_r = c_2\psi_l + c_1$

An extension of the notion of periodical boundary conditions are boundary conditions of the type $\psi_r = c_2\psi_l + c_1$, combined with the continuity equation $\frac{\partial\psi}{\partial n}|_l = \frac{\partial\psi}{\partial n}|_r$, where r is the so-called "right" boundary and l the "left" boundary. See Figure 1.2.4 for a definition of these boundaries.



Figure 1.2.4: Boundary conditions of the type $\psi_r = c_2\psi_l + c_1$ on the sides r and l

If $c_1 = 0$, $c_2 = 1$ this boundary condition reduces to a standard periodical boundary condition. To apply boundary conditions of this type, the user must define them as periodical boundary conditions with a factor and a constant in the input block "PROBLEM". All degrees of freedom in nodal points of boundary r are identified with the corresponding degrees of freedom on boundary l times c_2 plus the constant c_1 .

A special application of boundary conditions of this type is when separate regions with different coefficients are connected. In that case the equation $\frac{\partial\psi}{\partial n}|_l = \frac{\partial\psi}{\partial n}|_r$, changes into flux left = - flux right $\times c_2$.

Examples of the use of these boundary conditions can be found in the manual Standard Problems, Sections 3.1.9, 3.1.10 and 3.5.2.

1.3 The solution of coupled problems at one mesh

For some problems it is necessary to solve more than one equation at a time on the same mesh. In the simple case first one equation is solved and next the second one using the solution of the first one to create an initial estimate or to compute the coefficients of the differential equation.

In general, however, these combinations may be much more complex. For example it is possible that a time-dependent temperature equation is solved and that at certain times the stresses in the material are computed as function of the time. Or one may solve a system of time-dependent partial differential equations where the solution of the other equations is used to compute the coefficients of the next one.

Another typical example is formed by the stationary non-linear Boussinesq equations, which describe temperature dependent flow. In these equations the unknowns are velocities, pressure and temperature. The Boussinesq equations consist of a set of 2 (2D) or 3 (3D) momentum equations, a continuity equation and a temperature equation. Consult the manual Standard problems for the exact formulation.

Now there are two possible ways to solve these equations. In the first one, the coupled approach all equations are solved simultaneously. The advantage of such an approach is the faster convergence. However, since the number of unknowns may be large in practice, the computation time may be large due to the necessity of solving large linear systems per iteration step. An alternative, commonly used approach is first to solve the momentum equations coupled with the continuity equation using an estimate of the temperature in order to compute the right-hand side and then to compute the temperature by solving a convection-diffusion equation based on the just computed velocity. This may result in a better estimate of the temperature and the process may be repeated using this new temperature. Although the convergence of this process may be slower, it allows the solution of smaller systems of equations.

It is clear that in the last approach we have to solve two different problems on the same mesh. For that purpose in SEPRAN the notion problem sequence number or abbreviated problem number has been introduced. Each of these problems is provided with a problem sequence number. Problem sequence numbers are numbered from 1 to NPROB and are essential to distinguish between the various problems. The user must introduce these problem sequence numbers in the input part "PROBLEM". In this part also the type of problem connected to these problem sequence numbers is described. In many other input parts it is possible to connect quantities to specific problem sequence numbers. Vectors corresponding to a problem contain the problem sequence number in their administration and hence SEPRAN itself recognizes to which problem they belong. Of course the first time they are created this information must be made available.

1.4 The standard SEPRAN input file

The input from the standard input file is organized in records. A record is a line in the input file. Records must always be at most 240 positions long. SEPRAN requires a special form of free format input.

The input file may be subdivided into several parts:

Part CONSTANTS In this part constants are defined and given a name. These constants may be used in the rest of the input file.

If this part is used it must always be the first part of the file.

Part COMMANDS Contains the actual input

Part SET COMMANDS may be used anywhere in the input file.

Part INCLUDE may be used anywhere in the input file.

Part CONSTANTS The standard input file may start with a so-called constants part, provided one of the standard SEPRAN programs is called (SEPMESH, SEPCOMP or SEPPOST). For the use in user programs consult the Programmers Guide.

This part itself consists of three possible subparts that may be used in arbitrary order. Each of these parts must end with an END line, and each subpart itself may be used only once.

It is always necessary that this part is read as first input by SEPRAN.

The following subparts are available:

```
CONSTANTS
DEBUG_PARAMETERS
GENERAL_CONSTANTS
```

Each of these subparts have the following meaning:

Part CONSTANTS Part to define user defined constants, vector names and variable names.

This part has the following layout:

```
CONSTANTS
  INTEGERS
    name = value
    name
    name = value
    name = value1, value2, value3, ...
  REALS
    name = value
    name = value
    name = value
    name = value1, value2, value3, ...
  VARIABLES or SCALARS
    name = value
    name = value
    name
    name = value1, value2, value3, ...
  VECTOR_NAMES
    name
    name
```

```
STRINGS
  name = value
  name
END
```

These records have the following meaning

CONSTANTS (mandatory). This keyword indicates that constants will be defined.

If this keyword is not present as first keyword in the file it is not possible to define constants.

INTEGERS This keyword indicates that some integer constants will be defined.

It must be followed by the integers to be defined.

The layout of the integers is:

```
[sequence number] name_of_constant [value]
```

The square brackets ([and]) indicate that these parts are optional. The brackets itself have no meaning in the input and are not required.

The sequence number defines a sequence number with respect to the integer constant. Only sequence numbers between 1 and 1000 may be used. If a sequence number appears twice the last presence is used and a warning is issued.

The sequence numbers indicate how the integer constants are stored internally. The user can address these constants internally by their sequence number and in that case he must know the relation between sequence number and integer constant.

If no sequence number is given, the next one is used.

name_of_constant is mandatory and defines the name of the constant. The name must start with a letter and may consist of letters, digits and underscore signs only. All other signs are treated as separation sign, including the blank space. The name of the constant may be used in the rest of the input file as reference to the constant.

value must be a number according to standard FORTRAN rules. Spaces in the number are treated as separation character. If value is given the constant gets an initial value.

If a series of values is given the integer is considered as an integer array with length equal to the number of values. In that case length sequence numbers are used. Array values may be addressed as name_of_constant(*j*), with *j* the position. At this moment an array with index may only be referred to in the input block STRUCTURE by using:

```
SCALAR j = name_of_constant(3)
```

REALS This keyword indicates that some real constants will be defined.

It must be followed by the reals to be defined according to exactly the same rules as for the integers. Reals and integers have their own sequence numbers. Names of reals must be different from the names of the integers.

If a series of values is given the real is considered as a real array with length equal to the number of values. In that case length sequence numbers are used. Array values may be addressed as name_of_constant(*j*), with *j* the position. The same limitations as for integers are valid.

The following constants are available without declaration in the constants block.

pi represents π .

VARIABLES This keyword indicates that some variables will be defined.

It must be followed by the variables to be defined according to exactly the same rules as for the integers.

The difference between a variable and a real or integer constant is the following:

Constants that are used in the input file will be interpreted at the moment they are read. Then the value of the constant is substituted instead of the name of the constant. Hence if the constant changes later on this has no effect anymore.

Since all input is read at the start of the input, this means that there is hardly any possibility to change the constant, except as described in Section 1.6.

On the other hand variables are connected to the scalars as defined in the input file. See Section 3.2.3.

Scalars are evaluated at the moment they are used and may be recomputed during the execution of the program. Hence they allow a larger flexibility to manipulate. Internally this means that the value of the variable is not substituted during reading, but that a reference to the variable is made. For more details about the storage of the variables, the reader is referred to Section 1.6.

At this moment variables can only be used in combination with the keyword `STRUCTURE` as defined in Section 3.2.3.

Instead of `VARIABLES` also `SCALARS` may be used

The following variables are available without declaration in the constants block.

time represents the time during a time integration.

In general it is not necessary to define scalars or variables explicitly in this sub-block. If by the context it is clear that a variable is defined, for example by `var = value` or `var = expression` the variable is declared implicitly. Mark that this is only possible in the structure block.

VECTOR_NAMES This part is used to define vector names explicitly. In this way also the sequence of these vectors is prescribed.

In general it is not necessary to define vectors explicitly in this sub-block. If by the context it is clear that a vector is defined, for example by `vector = vector expression` or by other statements in the structure block, the vector is declared implicitly.

If a vector name is defined in `sepcomp`, this name is also known in the postprocessing program `seppost`.

Just as for integers, vector names may be preceded by a sequence number.

The following vector names are available without declaration.

COOR defines the vector of coordinates (ndim values per point).

X_COOR defines the vector of x coordinates (1 value per point).

Y_COOR defines the vector of y coordinates (1 value per point).

Z_COOR defines the vector of z coordinates (1 value per point).

STRINGS opens a set of strings that is stored. These strings may be used for example for print purposes.

END (mandatory), defines the end of the "CONSTANT" block.

The block `CONSTANTS` must always be read as first block. Simple expressions are allowed in the `SEPRAN` input file, however, if the user wants to manipulate with constants in a complex way he may also add a user written subroutine `COMPCONS` as described in Section 1.6. This subroutine allows the user to change constants and in this way use complex expressions based on `FORTRAN`.

Subroutine `COMPCONS` is called only once, immediately after the reading of the constants block.

The user may get the values of all the constants and variables in each user written subroutine, using the function subroutines:

GETINT 3.3.12.1,

GETCONST 3.3.12.2 and

GETVAR 3.3.12.3.

The positions of the variables or constants in common block CUSCONS can be found using the subroutines

GETNAMEVAR [3.3.14.3](#)

GETNAMEINT [3.3.14.1](#) and

GETNAMEREAL [3.3.14.2](#).

To get the sequence number of solution array use subroutine **PRGETNAME** [3.3.14.4](#).

To put values in the commons the user may use the subroutines:

PUTINT [3.3.13.1](#),

PUTREAL [3.3.13.2](#) and

PUTVAR [3.3.13.3](#).

Part DEBUG_PARAMETERS This part is used to define some debug options. It is meant for the experienced user and activates printing and plotting of intermediate quantities.

This part has the following layout:

```
DEBUG_PARAMETERS
  parameter_1
  parameter_2
  .
  .
END
```

For a definition of the possible parameters the user is referred to Section 1.3 of the Programmers Guide.

Part GENERAL_CONSTANTS This part is used to define some general constants that are used in the whole program.

This part has the following layout:

```
GENERAL_CONSTANTS
  parameter_1 = ...
  parameter_2 = ...
  .
  .
END
```

For a list of general constants that can be set, see Section [1.7](#).

Part COMMANDS We distinguish between so-called **COMMAND** records, data records and comment records.

A COMMAND record consists of a name, which may not be abbreviated and may not contain any spaces. Sometimes this name is followed by extra information.

A keyword is defined as a set of characters consisting of letters and the underscore sign (-) only. The SEPRAN input is in general case insensitive, which means that there is no difference between capitals and small letters. The only exception is the input of texts to be used for output purposes.

A data record contains additional information. Each data item that is described must be put in **one** record, unless stated otherwise.

Data records are generally of two forms:

```
VARIABLE = ( data1, data2, . . . )
or
VARIABLE = FUNCTION ( data1, data2, . . . )
```

where VARIABLE is some name and data1, data2 are names consisting of letters and digits or digits only.

FUNCTION is some function of the data.

The equal sign, parentheses and colons are special characters.

For example when the user must give the co-ordinates of point 1 (P1), which are for example (0 , 0), a typical data record would be:

```
P1 = ( 0 , 0 )
```

and when curve 1 (C1) is a straight line from point P1 to point P2, consisting of 4 equidistant elements of 2 nodal points each, the data record would be:

```
C1 = LINE 1 ( P1 , P2 , NELM = 4 )
```

There are no restrictions on the place of data in a data record, as long as its length is not longer than 80 positions. Leading and trailing spaces between names, numbers and special characters are allowed in data records. Spaces and commas are always treated as separators of information.

When a data record needs more than 240 characters, the user may end the record with the character &, and continue in the next record. The character & is treated as a space.

Names in data records (not in command records!) may be abbreviated by using as much characters as necessary to distinguish between the possible forms of input, however, this does not improve the readability of the data and should therefore be avoided. Moreover, since the number of allowed input forms is generally increasing, the abbreviation may become ambiguous in the future. Colons in data records are essential and may not be removed!

Items in data records must always be given in the order as indicated in the manual. However, if the format NAME = EXPRESSION for the variable is used, the order is not important.

When parameters are omitted in a data record, default values are used.

Numbers in records are represented as constants, for example: 1.0, 1.0D0, 1.0E-1, 1, .01 ; they may not contain any spaces. The first character of a number must be a digit, minus sign or a point.

Besides numbers the user may also make use of the constants (reals and integers only) defined in the block CONSTANTS.

This is done by using the name of the variable. When reading this constant the value of the constant is immediately substituted. Mark that substitution takes place when reading, so if the constants get a different value in the program this has no effect unless the reading of the input is done afterwards. How to do that is only described in the Programmers Guide.

expressions The SEPRAN input file allows for simple expressions.

Within the expressions you may use standard brackets (and), and the operators +, -, * (multiplication), / division and ** or ^ as power symbol. Furthermore you may use any of representations of the numbers given before and also previously defined constants may be used.

Special mathematical symbols

At this moment only the number *pi*, hence given as **pi** is available.

The following mathematical functions are available *sqrt*, *exp*, *cos*, *sin*, *tan*, *arcsin*, *arccos*, *arctan* and *log* (natural logarithm). In the input file they are represented by

```
sqrt
exp
sin
cos
```

```
tan
asin
acos
atan
log
```

A typical example might be:

```
a = exp(sin(2+cos(3))+5)
```

Variables or vectors may not be used in an expression, except stated otherwise. The standard FORTRAN priority rules for operators are applicable.

comments There are three ways of providing comments to the input file. The first one defines a complete record as a comment line: An asterisk (*) in the first column of a record means that the line is considered as a complete comment record, all other positions are free. Comment records may be placed anywhere in the input, except between continuation records. Blank records are also allowed.

The next possibility is to use comments within other types of lines. All input after a hash (#) or an exclamation mark (!) is interpreted as comment for that line. So data followed by # may contain explaining comments after that #.

An extra option that is available, is that of the so-called set commands.

SET COMMANDS The user may place the following SET COMMANDS anywhere in the standard input, provided a read command is given by one of the standard SEPRAN subroutines. So the best place to give the set commands is immediately before or after the first record to be read by a subroutine, or before the END record. If placed after the END record, it is read by the next subroutine requiring input from the standard input file.

The next SET COMMANDS are available:

```
SET WARN ON (default)
SET WARN OFF
SET TIME OFF (default)
SET TIME ON
SET OUTPUT OFF (default)
SET OUTPUT ON
SET OUTPUT NONE
SET OUTPUT LEVEL=1
SET MAX ERROR=e
SET MAX WARN=w
SET SKIP ON
SET SKIP OFF
```

The meaning of these commands is as follows:

If WARN is set off, then no warnings are printed from that moment. WARN ON reactivates the printing of warnings.

From the moment TIME ON is read, the actual CPU time from the start of the program is printed for each next SEPRAN subroutine, TIME OFF suppresses this printing.

OUTPUT ON activates the printing of the length used in the BUFFER array, as well as the length of variable length user arrays. OUTPUT OFF deactivates this printing. OUTPUT NONE deactivates all printing of SEPRAN messages, except for error messages and the printing explicitly required by output or print subroutines. So even the echoing of input records is

stopped.

SET OUTPUT LEVEL= l defines the level of output produced by SEPRAN. At this moment the following values of l are available:

- 1 (equivalent to SET OUTPUT NONE)
- 0 (equivalent to SET OUTPUT OFF)
- 1 (equivalent to SET OUTPUT ON)
- 2 gives the same output as $l=1$, but besides that also gives a message for each writing to or reading from the SEPRAN direct access file (files 1 to 4)
- 3 gives the same output as $l=2$, however, also each call of a memory management subroutine is recorded to the output file.

SET MAX WARN= w defines the maximal number of warnings that is printed before the printing of warnings is suppressed. The default value is $w=10$.

SET MAX ERROR= e defines the maximal number of errors that is printed before the program is terminated. The default value is $e=10$.

SET SKIP ON may be used in the input file to skip the next lines until a SET SKIP OFF command is found. The lines between are read but not printed nor interpreted. In this way several options may be available in the input file from which only one is used. An extra option that is available, is that of the so-called set commands.

INCLUDE Anywhere in the input the user may include another input file using:

```
include 'file_name'
```

Here `file_name` must be a file name between quotes. This file itself is included in the input file and therefore may also contain include files.

If the first 7 letters of the file name are SPHOME/ (capitals) then this name is replaced by the name of the sepran home directory followed by a /.

Example of the use of constants.

Suppose that the user wants to generate a mesh for a rectangle with size $a \times b$, where a and b may have different values for different experiments, then the following input file for SEPMESH might be used.

```
constants
  integers
    nelm1 = 10
    nelm2 = 15
  reals
    a = 2
    b = 3
end
mesh2d
  points
    p1 = ( 0, 0 )
    p2 = ( a, 0 )
    p3 = ( a, b )
    p4 = ( 0, b )
  curves
    c1 = line1(p1,p2,nelm=nelm1)
    c2 = line1(p2,p3,nelm=nelm2)
    c3 = line1(p3,p4,nelm=nelm1)
    c4 = line1(p4,p1,nelm=nelm2)
  surfaces
    s1 = general3 ( c1,c2,c3,c4 )
  plot
end
```

In this example it is also allowed to use an expression like:

```
constants
  integers
    nelm1 = 10
    nelm2 = 1.5 * nelm1
  reals
    a = 2
    b = 3
end
mesh2d
  points
    p1 = ( 0, 0 )
    p2 = ( a, 0 )
    p3 = ( a, b )
    p4 = ( 0, b )
  curves
    c1 = line1(p1,p2,nelm=nelm1)
    c2 = line1(p2,p3,nelm=nelm2)
    c3 = line1(p3,p4,nelm=nelm1)
    c4 = line1(p4,p1,nelm=nelm2)
  surfaces
    s1 = general3 ( c1,c2,c3,c4 )
  plot
end
```

Both examples produce the same result.

1.5 The SEPRAN environment file

The SEPRAN environment file is used to define some general constants, file reference numbers, names of files and types of plotting devices. The environment file has the name `sepran.env` and is defined as a standard ASCII file. This file consists of comment records and records with a definition.

The standard `sepran.env` file can be found in the SEPRAN subdirectory `instal`. This file can only be changed by the SEPRAN installation officer. However, it is also possible to make a copy of this file and put it in your own directory. This copy may be changed according to your own personal preferences. A disadvantage is that this copy is only valid in the directory where it is available, and programs running from other directories do not read this file.

Each SEPRAN program that is started checks if there is a file named `sepran.env` in the local directory. If this file is available the file is read and the contents are used by the program. If this file does not exist the general `sepran.env` file from the directory `SEPRAN/instal` is used, where SEPRAN is a substitute for the SEPRAN home directory.

The `sepran.env` file is a standard file with comment records and data records. A comment record starts with an asterisk, the data records start with an explanation followed by a colon (:), followed by the actual data. The `sepran.env` file has, in contrast to most other input files, a fixed sequence. Data records may not be interchanged. If this is done error messages may be the result, or more severely, the program may produce incomprehensible messages.

Contents of the `sepran.env` file

The present version of the `sepran.env` file contains the following data records (in that sequence):

- name of SEPRAN home directory
- version number of environment file
- type of operating system
- name of computer
- type of computer
- version name of SEPRAN
- name of executable
- number of characters in a word
- approximation of infinity
- machine accuracy
- accuracy of a half real
- Unit number for reading of SEPRAN input
- Unit number for writing of SEPRAN output
- Unit number for error messages
- Unit number for file 1
- Unit number for SEPRAN backing storage file
- Unit number for temporary file
- Unit number for file containing menus
- Unit number for default SEPRAN plot file
- Unit number for mesh output file
- Unit number for file `sepcomp.inf`
- Unit number for file `sepcomp.out`
- Default SEPRAN plot file (formatted/unformatted)
- file 10 (formatted/unformatted)
- file 74 (formatted/unformatted)
- append mode (yes/no)
- Name of file 1
- Name of file 4
- Name of file for menus

Name of temporary file
Name of SEPRAN input file
Name of SEPRAN output file
Name of (Binary mesh output file)
Name of sepcomp.inf (73)
Name of sepcomp.out (74)
Name of SEPRAN backing storage file
lenwor
Record length for file 1 (at least 1000)
Record length SEPRAN backing storage file
Record length for scratch file
defplo
cm
small
wide
aleng
posdev
nameplt
default plotter for interactive mode
default plotter for hard copies
carriage
isite
tem2
tem3
tem4
tem5
tem6
tem7
tem8
tem9
tem10

explanation of the data records

name of SEPRAN home directory

The complete path name of the SEPRAN home directory must be given.

version number of environment file

Do not change this number.

type of operating system

Answers that are recognized are:

msdos

unix

vax/vms

name of computer

Host name of the computer.

type of computer

Give type number from table below.

- | | |
|---|---------------------|
| 1 | IBM |
| 2 | Cyber (NOS/VE) |
| 3 | Apollo (Unis/Aegis) |
| 4 | HP 9000 (HP/UX) |
| 5 | Unix |
| 6 | VAX VMS |
| 7 | Harris |

- 8 Cray
 - 9 IBM PC 386 + MSDOS + FTN77/386
 - 10 CONVEX
 - 13 Alliant
 - 14 unknown
- version name of SEPRAN
This item is generally not used.
- name of executable
This item is generally not used.
- number of characters in a word
Give an integer number.
- approximation of infinity
The largest real (or double precision) number on the computer.
- machine accuracy
The machine accuracy must be given as real number.
For example at a 32 bits computer with double precision arithmetic: 1d-16
- accuracy of a half real
The machine accuracy for single precision reals.
- Unit number for reading of SEPRAN input
if 5 is given usually standard input is meant, without file name.
- Unit number for writing of SEPRAN output
if 6 is given usually standard output is meant, without file name.
- Unit number for error messages
- Unit number for file 1
- Unit number for SEPRAN backing storage file
- Unit number for temporary file
- Unit number for file containing menus
- Unit number for default SEPRAN plot file
Except the unit number iref_plot for the default plot file, SEPRAN uses two extra numbers for plotting files. These numbers are:
iref_plot + 1 for HPGL and Postscript files,
iref_plot + 3 for Tektronix (screen).
- Unit number for mesh output file
- Unit number for file sepcomp.inf
- Unit number for file sepcomp.out
- Default SEPRAN plot file (formatted/unformatted)
Indicates if the neutral SEPRAN plot files must be written in formatted or unformatted form.
- Name of file 10 (formatted/unformatted)
Indicates if the mesh output file must be written in formatted or unformatted form.
- Name of file 74 (formatted/unformatted)
Indicates if the sepcomp.out file must be written in formatted or unformatted form.
- append mode (yes/no)
Indicates if the output must be appended to an existing file (yes) or not (no).
This possibility is usually not available.
- Name of file 1
This name should not be changed.
- Name of file 4
This name should not be changed.
- Name of file for menus
This name should not be changed.
- Name of temporary file
If no name is given a scratch file is used.
- Name of SEPRAN input file
If no name is given standard input is used.
- Name of SEPRAN output file

If no name is given standard output is used.

Name of (Binary mesh output file)

Name of sepcomp.inf (73)

Name of sepcomp.out (74)

Name of SEPRAN backing storage file

lenwor
 Record length of some direct access files:
 Record length in words.
 In the open statement it is multiplied by lenwor

Record length for file 1 (at least 1000)
 Do not change this length

Record length SEPRAN backing storage file

Record length for scratch file

defplo
 Default plot size for pictures in cm's.

cm
 Multiplication factor to transform all quantities into cm's.
 If the plot package is based on cm's, cm should be 1, is it
 based on inches cm should be .3937.

small
 Width of small paper in cm's.

wide
 Width of wide paper in cm's.

aleng
 Maximal length of plot paper in cm's.

posdev
 Default plot package, is used in non-interactive mode
 Available packages:

- fb Write input parameters in binary form to default SEPRAN plot files, with names nameplt.001, nameplt.002 ...
- fa Write input parameters in ascii form to default SEPRAN plot files, with names nameplt.001, nameplt.002 ...

Each of these files contains information about one picture
 The files may be plotted later on, for example by SEPDISPLAY

f1 Write formatted, make only 1 output file (nameplt)

The next packages are meant for immediate plotting, without the intermediate step of a plot file

- a Apollo code
- c CalComp code
- g GKS code
- cg CGI code
- hh StarBase code for HP-display, Color HiRes (1280x1024)
- hl HP-display, Color LowRes (1024x768)
- hm Same, Monochrome LowRes (1024x768)
- hx StarBase code under X-Windows
- t Tektronix 4010 code
- tk Tektronix 4010 code, using MS-Kermit emulation
- tp Tektronix 4010 code, using Plot-10 library
- v PC, running FTN77/386 with VGA or EGA card

Possible hard copy devices:

p HP-GL plotter A4 format (file output)
pb HP-GL plotter A3
p0 HP-GL plotter A0
ps Postscript file (Portrait mode)
pl Postscript file (Landscape mode)

nameplt

Name of neutral SEPRAN plot file without extensions.
default plotter for interactive mode

default plotter for hard copies

carriage

Information about carriage control.

The parameter carriage indicates if the first character in a write statement to a file is used as carriage control parameter (yes) or not (no).

For most unix computers no carriage control is used, however, for example vax/vms uses carriage control.

isite

This parameter is only used at Eindhoven University.

The parameters tem2 to tem10 are meant for future purposes. They must be set equal to 0.

sample sepran.env file for DOS computer

The following sepran.env file is standard imposed on DOS computers. It gives an example of how a sepran.env file looks like.

```
*
* Version 2.1   Date 28-12-92
* Extension with respect to version 2.0:
* - Replace first dummy by isite
* Version 2.0   Date 17-10-92
* Extension with respect to version 1.1:
* - Addition of version number (second line)
* - Introduction of file for menus (part unit numbers and files)
* - Introduction of hard copy device identification (plots)
* Version 1.1   Date 27-12-91
* Extension with respect to version 1.0:
* - Addition of choice between carriage control or not
* - Addition of ten dummies for future purposes
*
* This file contains most of the machine-dependent quantities
* with respect to SEPRAN
* The file is read by each SEPRAN program and the information is stored
* in-core in some common blocks
* The sequence of the information is essential
* The file may be updated for your local computer
* Lines starting with * are treated as comments
* Only the information after the colon is interpreted
*
```

```
*   PC version using FTN77/386
*
*   Start with the SEPRAN home directory
*
SEPRAN home directory      (SPHOME)           :c:\sepran
version number of environment file          :2
*
*   Type of operating system
*   Recognised are unix, vms, cms, msdos, unknown, nos/ve
*
type of operating system           :msdos
name of computer                   :hp-vectra
*
*   Give type of computer system. Possible values:
*
*           1   IBM
*           2   Cyber   (NOS/VE)
*           3   Apollo  (Unis/Aegis)
*           4   HP 9000 (HP/UX)
*           5   .....  Unix
*           6   VAX     VMS
*           7   Harris
*           8   Cray
*           9   IBM PC 386 + MSDOS + FTN77/386
*          10   CONVEX
*          13   Alliant
*          14   unknown
*
type of computer (number from list)       :9
version name of SEPRAN                    :
name of executable                        :
*
*   Some machine dependent constants
*
number of characters in a word             : 4
approximation of infinity                  :1e307
machine accuracy                           :1e-15
accuracy of a half real                    :1e-6
*
*   Unit numbers for files
*
Unit number for reading of SEPRAN input    : 15
Unit number for writing of SEPRAN output   : 16
Unit number for error messages            : 14
Unit number for file 1                     : 11
Unit number for SEPRAN backing storage file: 12
Unit number for temporary file             : 13
Unit number for file containing menus      : 17
Unit number for default SEPRAN plot file  : 8
Unit number for mesh output file          : 10
Unit number for file sepcomp.inf          : 73
Unit number for file sepcomp.out          : 74
*
*   Remark:
*
```

```
* Except the unit number iref_plot for the default plot file, SEPRAN uses two
* extra numbers for plotting files. These numbers are:
* iref_plot + 1 for HPGL and Postscript files
* iref_plot + 3 for Tektronix (screen)
*
*
* Indication whether the files 10 and 74 are binary (unformatted) or formatted
* Recognised are formatted and unformatted
*
Default SEPRAN plot file           :formatted
file 10                           :unformatted
file 74                           :unformatted
*
* Indication whether the output file must be appended to an existing output
* file (yes) or not (no)
*
append mode                        :no
*
* Standard and default names for files:
* If no name is given the file is supposed to be without name
* and only the unit number is used in the open statement
* The files 5 and 6 are not opened
* If the reading and writing SEPRAN files have other unit numbers than
* 5 and 6, they must have a name
* The temporary file may have no name in which case it is opened as
* a scratch file or it may have the name
* tmp
* In the last case (only meant for UNIX computers) the temporary file is
* opened as a temporary file at the directory /tmp
*
file 1                             :SPHOME\bin\stanelm
file 4                             :SPHOME\bin\errormsg
file for menus                     :SPHOME\bin\menumsg
temporary file                     :
SEPRAN input file                  :sepran.dat
SEPRAN output file                 :sepran.out
(Binary mesh output file)          :meshout.put
sepcomp.inf (73)                   :sepcomp.inf
sepcomp.out (74)                   :sepcomp.out
SEPRAN backing storage file        :file2.sep
*
* Record length of some direct access files:
* Record length in words
* In the open statement it is multiplied by lenwor
*
lenwor                             :4
Record length for file 1 (at least 1000) :1000
Record length SEPRAN backing storage file :1157
Record length for scratch file      :1024
*
* Information about plotting
*
* Meaning of the constants:
*
* defplo: Default plot size for pictures in cm's
```

```

*   cm:      Multiplication factor to transform all quantities into cm's
*             If the plot package is based on cm's, cm should be 1, is it
*             based on inches cm should be .3937
*   small:   Width of small paper in cm's
*   wide:    Width of small paper in cm's
*   aleng:   Maximal length of plot paper in cm's
*
*           These parameters have only effect for some special plot packages
*
*   posdev:  Default plot package, is used in non-interactive mode
*           Available packages:
*
*   fb       Write input parameters in binary form to default SEPRAN plot
*           files, with names nameplt.001, nameplt.002 ...
*   fa       Write input parameters in ascii form to default SEPRAN plot
*           files, with names nameplt.001, nameplt.002 ...
*
*           Each of these files contains information about one picture
*           The files may be plotted later on, for example by SEPDISPLAY
*
*   f1       Write formatted, make only 1 output file (nameplt)
*
*           The next packages are meant for immediate plotting, without
*           the intermediate step of a plot file
*
*   a        Apollo code
*   c        Calcomp code
*   g        GKS code
*   cg       CGI code
*   hh       StarBase code for HP-display, Color HiRes (1280x1024)
*   hl       HP-display, Color LowRes (1024x768)
*   hm       Same, Monochrome LowRes (1024x768)
*   t        Tektronix 4010 code
*   tk       Tektronix 4010 code, using MS-Kermit emulation
*   tp       Tektronix 4010 code, using Plot-10 library
*   v        PC, running FTN77/386 with VGA or EGA card
*
*           Possible hard copy devices:
*
*   p        HP-GL plotter A4 format (file output)
*   pb       HP-GL plotter A3
*   p0       HP-GL plotter A0
*   ps       Postscript file
*   nameplt: Name of default SEPRAN plot file
*
defplo      :15
cm          :.3937
small      :20
wide       :65
aleng      :3600
posdev     :fa
nameplt    :sepplot
default plotter for interactive mode :v
default plotter for hard copies     :ps
*

```

```
* The next part is extended at 27-12-91:
*
* Information about carriage control
* The parameter carriage indicates if the first character in a write statement
* to a file is used as carriage control parameter (yes) or not (no)
* For most unix computers no carriage control is used, however, for example
* vax/vms uses carriage control
*
carriage                                :no
*
* Extension of 19-12-92: Read site number for special calls
*
* Allowed numbers:
*
* 0:   general
* 1:   TUE
*
isite                                    :0
*
* The next 9 positions are reserved for later use:
*
tem2                                     :0
tem3                                     :0
tem4                                     :0
tem5                                     :0
tem6                                     :0
tem7                                     :0
tem8                                     :0
tem9                                     :0
tem10                                    :0
```

1.6 Manipulation of constants and variables(subroutine COMPCONS)

Description

As described in Section 1.4 the user may define constants to be used in the input file by their name. Variables in the input file are also indicated by their name. So the names of constants and variables must be unique. Usually they are referred to as scalars.

In some cases the constants are functions of other constants and it is easier to express these constants as some function than to compute them before by hand.

Also one might consider to recompute the scalars at the start of the program.

For that purpose the user may provide the subroutine COMPCONS. This subroutine is called by the SEPRAN main programs immediately after the reading of the constants, but before the rest of the input file is read.

If the user provides this subroutine he has to link the main program with this subroutine as described in the Sections 2.1, 3.1 and 5.1

Heading

```
subroutine compcons
```

Parameters

Subroutine COMPCONS has no parameters itself but information of the constants is available through the common block CUSCONS. The source of this common block can be found in the directory SEPRAN/common, where SEPRAN is the main SEPRAN directory.

CUSCONS has the following shape:

```
integer maxints, maxreals, maxcls, maxvec
parameter ( maxints=1000, maxreals=1000, maxcls=1000, maxvec=100 )
double precision rlcons(maxreals), scalars(maxcls)
integer inconss(maxints), numints, numreals, numscals
common /cuscons/ rlcons, scalars, inconss, numints, numreals,
+               numscals
save /cuscons/
```

To include this common block into your program use:

```
include 'SPcommon/comcons1'
include 'SPcommon/cuscons'
```

Mark that the name in the include statement is case sensitive, hence the first two letters must be capitals and all other ones must be in lower case.

The parameters have the following meaning:

MAXINTS Maximum number of integer constants that may be used. At this moment this maximum is fixed to 1000 and the user may not change this number himself.

MAXREALS Maximum number of real constants that may be used. At this moment this maximum is fixed to 1000 and the user may not change this number himself.

MAXCLS Maximum number of scalar variables that may be used. At this moment this maximum is fixed to 1000 and the user may not change this number himself. This parameter is only used in SEPCOMP.

MAXVEC Maximum number of solution type vectors that may be used. At this moment this maximum is fixed to 100 and the user may not change this number himself.

This parameter is only used in SEPCOMP.

RLCONS Array of length MAXREALS in which the real constants are stored in the sequence as defined in the input block "CONSTANTS" (Section 1.4).

INCONS Array of length MAXINTS in which the integer constants are stored in the sequence as defined in the input block "CONSTANTS" (Section 1.4).

See RLCONS.

SCALARS Array containing the scalar variables as defined in program SEPCOMP, input part STRUCTURE. All the variables are stored as scalars.

NUMINTS Highest sequence number of integers that have been defined in INCONS. Integers that are skipped are automatically initialized to zero and their name to 10 blank spaces.

NUMREALS Highest sequence number of reals that have been defined in INCONS. Reals that are skipped are automatically initialized to zero and their name to 10 blank spaces.

NUMSCALS Actual number of scalars that have been defined. This number may be increased during the computations in program SEPCOMP.

Input

The common block CUSCONS and also the common block CUSNAME have been initialized by SEPRAN. The values read in the input block CONSTANTS have been substituted.

Output

The user may change the contents of the common block, provided he does not change the parameter statements.

The user itself is responsible for the change and he must do it in a consistent way.

Layout

Subroutine COMPCONS must be programmed as follows:

```
subroutine compcons
implicit none
include 'SPcommon/comcons1'
include 'SPcommon/cuscons'

    statements to adapt the parameters in the common blocks

end
```

Remark 1

If the user wants to extract and change the constants nelm1, nelm2, a and b as described in the example in Section 1.4 from cuscons and change these values, the following method is recommended:

```
subroutine compcons
implicit none
include 'SPcommon/comcons1'
include 'SPcommon/cuscons'
integer nelm1, nelm2
double precision a, b
nelm1 = incons(1)
nelm2 = incons(2)
a = rlcons(1)
b = rlcons(2)

... statements to change these parameters

incons(1) = nelm1
incons(2) = nelm2
rlcons(1) = a
rlcons(2) = b
end
```

Remark 2

A more simple way of using COMPCONS is with the aid of the subroutines GETINT [3.3.12.1](#), GETCONST [3.3.12.2](#), PUTINT [3.3.13.1](#) and PUTREAL [3.3.13.2](#).

We show the previous example using these subroutines.

```
subroutine compcons
implicit none
integer nelm1, nelm2, getint
double precision a, b, getconst
nelm1 = getint ( 'nelm1' )
nelm2 = getint ( 'nelm2' )
a = getconst ( 'a' )
b = getconst ( 'b' )

... statements to change these parameters

call putint ( 'nelm1', nelm1 )
call putint ( 'nelm2', nelm2 )
call putreal ( 'a', a )
call putreal ( 'b', b )
end
```

A clear advantage of this last approach is that one does not have to include the common block, and more important that if the sequence of the parameters in the input block CONSTANTS are changed, this does not effect the program. There is no need anymore to know the sequence number of a constant in the common block.

For an example of the use of COMPCONS the user is referred to Section [6.3.1](#).

1.7 Definition of general constants

In the input file the user may start with input defining `CONSTANTS`, `GENERAL_CONSTANTS` and `DEBUG_PARAMETERS`.

In this section we describe the contents of the part `GENERAL_CONSTANTS`.

This part has the following layout

```
GENERAL_CONSTANTS
  accuracy_obstacle = eps
END
```

At this moment the following general constants may be defined:

accuracy_obstacle This constant is used in those cases where an obstacle is present in the mesh and the user needs to distinguish between points and elements that are in or out of the obstacle.

All points that are really inside the obstacle as well as all points that are precisely on the boundary of the obstacle are marked as internal points. However, from a computational points of view points that are very close to the obstacle may also be considered as part of the obstacle.

For that purpose the parameter `accuracy_obstacle` is used. All nodes that are closer to the obstacle than $\varepsilon \times \Delta s$ are considered as part of the obstacle. Δs is defined as the smallest element width at the beginning or end of all curves. So the finer the mesh, the closer a node must be to the obstacle in order to be part of it.

The default value for ε is 0.3.

These constants are stored in common block `cgenconst` as described in the Programmers Guide Section 21.21.

1.8 Overview of the SEPRAN commands

Most of the commands to run SEPRAN have already been treated in the SEPRAN Introduction (Chapter 3). Here we give a full overview of all the commands that are available to create and run programs.

It concerns the following commands

`sepmesh`
`sepcomp`
`sepfree`
`seppost`
`jsepview`

`sepget`
`compile`
`compiledbg`
`seplink`
`sepdbg`

`sepgetex`

`sempi`
`sepmakeparamesh`
`sepcombineout`
`sepcompoutdiff`

`sepman`

These commands have the following meaning

sepmesh is used to create a SEPRAN mesh.

For a description see Section (2)

sepcomp is used to perform the finite element computations.

For a description see Section (3)

sepfree is used to perform the finite element computations in case of free or moving boundary.

For a description see Section (3.4)

seppost performs the post processing.

For a description see Section (5)

jsepview displays the plots created by one of the other programs.

For a description see Section (3.5) of the SEPRAN introduction.

sepget copies a standard SEPRAN subroutine or program into your local directory.

See for example Section (2.3.2)

compile compiles one or more fortran subroutines.

This can also be done by `seplink`, however, `compile` only compiles the subroutines indicated and does not make an executable.

Usage: `compile file1.f file2.f ...`

compiledbg has exactly the same meaning as `compile`.

The only difference is that subroutines are compiled with the debug option on, which means that they may be used in a debugger.

seplink compiles and links a main SEPRAN program with the given set of `.f` files and also all `.o` files in the directory.

It creates an executable and uses the SEPRAN libraries.

For a description see for example Section (2.1)

sepdbg has the same meaning as `seplink`.

In this case the program is compiled and linked with the debug option activated. Hence the program may run in a `sepdbg` environment.

sepgetexe copies the necessary files corresponding to a standard example into your local directory.

See for example Section (6.2.1)

sepmpi is used to compile and run a SEPRAN program in a parallel environment.

For a description see Section (3.6)

sepcombineout combines the local files `sepcomp_par.xxx` files into a global `sepcomp.out` file in a parallel environment.

For a description see Section (3.6)

sepcompoutdiff Computes the difference between two `sepcomp.out` files with equal length but different names and stores the result in a new file of the same shape.

Only the real numbers are subtracted.

Usage: `sepcompoutdiff file1 file output_file`

with `file1` and `file2` the two input files and `output_file` the file with the difference vectors

sepman activates `acoread` to display the manual required.

Usage: `sepman <name_of_manual>`

The following values of `<name_of_manual>` are available:

um shows the Users Manual.

sp manual Standard Problems

intro Introduction manual

exams manual with examples corresponding to the Standard Problems

pg Programmer's Guide

userexams Demonstrates some user examples

th theoretical manual

2 The pre-processing part of SEPRAN

2.1 Introduction

In the pre-processing part of SEPRAN the mesh is created. In most problems this may be done before the actual computational part. The only exception is for free boundary problems in which the boundaries and as a consequence also the mesh are adapted during each step of the computational process. In the introduction it has been described how the program SEPMESH may be used for simple one and two-dimensional problems. In this chapter the complete possibilities of SEPMESH are described. If you need to solve free boundary problems, then it is necessary to consult the SEPRAN PROGRAMMERS GUIDE in order to construct your own main program using the tools SEPRAN provides.

Sometimes it may be necessary to provide SEPMESH with one or more function subroutines which define curves or surfaces as functions defined by the user. In that case it is not possible to use the standard program SEPMESH immediately, but the user must create a simple main program consisting of 3 lines only and also provide the FORTRAN sources for the function subroutines.

The main program has the following structure:

```
program examplmesh
call startsepmesh
end
```

Subroutine STARTSEPMESH is in fact the body of program SEPMESH. The name of the program (in this case EXAMPLEMESH) may be chosen freely.

If one or more function subroutines are provided the easiest way is to put these subroutines immediately behind the main program. So in that case we get something like:

```
program examplmesh
call startsepmesh
end

subroutine funcsf ( ... )
.
.
.
end
```

In this example the parameters and the body of the function subroutines have intentionally been skipped, they are treated in the next sections.

The main program and the subroutines must be created by a text editor and put into a file. This file must have the extension .f or .f90 in a UNIX environment and .for in a MSDOS environment. The user input must be stored in a separate file.

Once the file containing main program and subroutines has been created, this file must be translated (compiled) and the program must be linked with the SEPRAN libraries. Both actions may be

performed in one step by the command seplink:

```
seplink file
```

where *file* is the name of the file containing the program without the extension .f.

For example the command:

```
seplink example
```

compiles and links the file example.f.

In the first step of seplink the fortran code is checked and translated. Fortran error-messages appear on the screen. If the number of errors exceeds the size of one screen it is wise to redirect the output to a file for example `out.put`. This file `out.put` can always be inspected with a standard text editor. If the compilation has been carried out correctly, seplink links the program and subroutines with the SEPRAN library. seplink automatically links all subroutines in the directory that have been precompiled, i.e. all files with the extension `.o` in unix or `.obj` in msdos. So the user must remove these files if they should not be included in the executable. If one or more subroutines are missing seplink reacts with the message `undefined symbol`, followed by the name of the subroutine(s) provided with an underscore at the end of the name. For example if you did provide a subroutine `funbs` instead of `funcs` you get the message

```
undefined symbol
```

```
funcs_
```

The error message "subroutines missing" usually results from an incorrectly spelled subroutine name or from the omission to declare an array.

Error messages of the linking phase are written directly to the screen. If both compilation and linking have been carried out successfully seplink produces a file with the name of the seplink parameter (that is without the extension .f). So `seplink example` produces a file `example`. To run the program `example` you type:

```
example < inputfile > outputfile
```

```
or
```

```
example < inputfile
```

In `outputfile` the results of program `example` are written. These may be error messages of SEPRAN or output written by the user. If `outputfile` is omitted all information is written to the screen.

The main program creates the file `meshoutput`.

Remark: the `outputfile` may have any name except `meshoutput`, or `sepplot.*.*`.

In Section 2.2 the complete input for program SEPMESH is described. Detailed information about curve generation can be found in Section 2.3. Section 2.4 is devoted to surface generation and Section 2.5 to volume generation. Finally in Section 2.6 some examples of input files for SEPMESH are given.

2.2 Input for program SEPMESH from the standard input file

The input for SEPMESH must be opened with the COMMAND MESH1D, MESH2D or MESH3D, depending on whether the problem is one-, two- or three-dimensional, and must be closed with the COMMAND END.

Besides the standard mesh generation SEPRAN has also the possibility to define a mesh by giving the co-ordinates of all nodal points. In that case SEPRAN creates a mesh based on all these points. This possibility is meant for example in the case that a user has made some measurements and wants to use the SEPRAN or AVS postprocessing to show the results. The input for this special case is described in Section 2.7.

The mesh generator recognizes the following input:

```
MESHnD
  coarse (unit=u, maxratio=m)
  scaling = s, type_scaling = t
  maxpoints = mp
  maxcurves = mc
  maxsurfaces = ms
  maxvolumes = mv
  points
    data records
  curves
    data records
  surfaces
    data records
  volumes
    data records
  meshline
    data records
  meshsurf
    data records
  meshvolume
    data records
  meshconnect
    data records
  interface_elements
    data records
  meshdummy
    data records
  renumber, options
  norenumber
  notopology
  intermediate points
    data records
  change_coordinates
    data records
  obstacles
    data records
  plot, options
  refine, options
  transform, options
  check_level = i
  parallel, options
END
```

Description of the COMMAND and DATA records.

The records must be given in the order as specified. An option is indicated like this: [option].

MESHnD (mandatory)

COMMAND record: opens the input for subroutine MESH, and defines the dimension of the space NDIM. (NDIM = n).

After the COMMAND MESHnD a number of optional commands may be given. These commands must be given between MESHnD and the mandatory command POINTS. Their mutual sequence is arbitrary.

COARSE (UNIT= u , MAXRATIO= m) (optional)

COMMAND record: defines that coarseness is used, u defines the unit length. The parameter MAXRATIO= m defines the maximum ratio between two successive edges on the boundary.

If this ratio is exceeded a warning is given. If $m = 0$, the maximum ratio is not checked.

The notion of coarseness is defined in the Introduction Section 4.1.2.

Default value: $u = 1$, $m = 0$.

SCALING = s (optional)

Defines if the coordinates of the boundary must be scaled before a submesh is created.

This option is only used for the volume generator GENERAL3D and will be used in the future for the surface generators GENERAL and TRIANGLE.

Possible values for s are:

NO No scaling is applied.

ALWAYS The coordinates of the boundary are scaled and after the creation of the submesh the coordinates are scaled back.

In this way it is possible to get a nice mesh even when the region is stretched.

DEPENDENT The coordinates of a selected set of surfaces and volumes are scaled.

This option has not yet been implemented.

Default value: No scaling.

TYPE_SCALING = t (optional)

Defines the type of scaling applied. This keyword makes only sense if scaling is applied.

Possible values for t are:

ALL The scaling is applied in all Cartesian directions. This means that before creating the mesh the Cartesian coordinates are mapped onto the region (0,1).

This option is only useful if a stretching in Cartesian coordinate directions is present.

The subdivision in each of the directions must be comparable, i.e. the number of elements in all Cartesian directions must be of the same order.

X_DIR The scaling is applied in the x-direction only.

Y_DIR The scaling is applied in the y-direction only.

Z_DIR The scaling is applied in the z-direction only.

DETECT The program detects the direction of the mesh and estimates a mean value of the element size in that direction. Scaling is adapted to these values.

Default value: ALL.

Remark: at this moment only the option ALL has been implemented. If you want to use one of the other options, please contact SEPRA.

MAXPOINTS = mp (optional)

mp defines the maximum number of user points that are allowed in the mesh. The default value is 1000. If a smaller value is used, some arrays may be smaller, but in general this has no effect on the total space. In fact, this option is meant for those cases that 1000 user points do not suffice.

MAXCURVES = *mc* (optional)

mc defines the maximum number of curves that are allowed in the mesh. The default value is 1000. Compare with MAXPOINTS.

MAXSURFACES = *ms* (optional)

ms defines the maximum number of surfaces that are allowed in the mesh. The default value is 1000. Compare with MAXPOINTS.

MAXVOLUMES = *mv* (optional)

mv defines the maximum number of volumes that are allowed in the mesh. The default value is 500. Compare with MAXPOINTS.

POINTS (mandatory)

COMMAND record: defines the points. Must be followed by data records of the type:

```
P1 = ( x1, y1, z1 [, c] )
P2 = ( x2, y2, z2 [, c] )
.
.
.
Pi = ( xi, yi, zi [, c] )
```

with *i* the point number and *x_i*, *y_i* and *z_i* the coordinates of point *i*. For one-dimensional problems only *x_i* is required, etc. Default values for the co-ordinates: 0.

c must only be used when the command COARSE has been read. It defines the coarseness of the elements in the neighborhood of the point *P_i*; default value: 1.

Remark: The sequence in which the points are given is arbitrary. If points are skipped, they get the co-ordinates (0,0,0) automatically. The largest number *i* used in *P_i* = ... defines the maximal number of user points.

If the user wants he may also give the co-ordinates in polar co-ordinates instead of Cartesian co-ordinates. In that case the input is

PD_{*i*} = (*r_i*, *φ_i*, *z_i*), with *φ* in degrees or

PR_{*i*} = (*r_i*, *φ_i*, *z_i*), with *φ* in radians

instead of *P_i* = (*x_i*, *y_i*, *z_i*).

These co-ordinates are automatically transformed into Cartesian co-ordinates.

CURVES (mandatory)

COMMAND record: defines the curve. Must be followed by data records of the type:

```
Ci = LINEj ( P1, P2, NELM=n [, RATIO=r, FACTOR=f ] )
Ci = ARCj ( P1, P2, P3, NELM=n [, RATIO=r, FACTOR=f ] )
Ci = USERj ( P1, P2, P3, . . . , Pn )
Ci = CLINEj ( P1, P2 [,NODD=o] )
Ci = CARCj ( P1, P2, P3 [,NODD=o] )
Ci = PARAM j ( P1, P2, NELM=n [,INIT=t_0] [,END=t_1] [, RATIO=r, FACTOR=f ] )
Ci = CPARAM j ( P1, P2 [,NODD=o [,INIT=t_0] [,END=t_1]])
Ci = PROFILE j ( P1, P2, NELM=n ,shape=s [,INIT=t_0] [,END=t_1] [, RATIO=r, FACTOR=f ] )
Ci = CPROFILE j ( P1, P2, NELM=n ,shape=s [,NODD=o [,INIT=t_0] [,END=t_1]])
Ci = SPLINE j ( P1, P2, . . . ,Pm, NELM=n [, RATIO=r, FACTOR=f ] [ ALPHA = a ] //
  [,TYPE=t [,tang=Pk, tang=P1] ] )
Ci = CSPLINE j ( P1, P2, . . . ,Pm [, NODD=o ] [ ALPHA = \alpha ] //
  [,TYPE=t [,tang=Pk, tang=P1] ] )
Ci = CURVES ( Ck, C1, Cm, . . )
Ci = TRANSLATE Cj ( P1 [,P2, P3, ... ] )
```

```

Ci = ROTATE Cj ( P1, P2, P3 [,P4, P5, ...] )
Ci = REFLECT Cj ( AXIS = P1, P2; P3 [,P4, P5, ...] )
Ci = SPCURVE ( Ck, C1, Cm, . . )
Ci = CIRCLE (P1, P2, P3, NELM=n [, RATIO=r, FACTOR=f ] )
Ci = OWN_CURVE j ( P1, P2, NELM=n [, IFUNC=i ] )
Ci = ELL_ARC j ( P1, P2, P3, NELM=n [, RATIO=r, FACTOR=f ] )
Ci = CELL_ARC j ( P1, P2, P3, [,NODD=o] )

```

Curves that are not defined explicitly are treated as non-existing curves. These curves are not available in the SEPRAN programs.

For an explanation of the various possibilities, see Section 2.3.

SURFACES (optional)

COMMAND record: defines the surfaces. Must be followed by data records of the type:

```

Si = GENERAL j ( C1, C2, C3, C4, . . . )
Si = TRIANGLEj ( C1, C2, C3, C4, . . . )
Si = QUADRILATERAL j (C1, C2, C3, C4 [,BLEND = b, CURVATURE = cu])
Si = RECTANGLE j ([N = n, M = m], C1, C2, . . .[, SMOOTH =i] )
Si = USER j ( NELEM = k, NPOINT = 1, C1, C2, . . . )
Si = SURFACES ( Sk, S1, Sm, . . )
Si = ORDERED SURFACE ( (Sk_1, Sk_2, ...), (S1_1, S1_2, . . .), (Sm_1, Sm_2, . . .), . . )
Si = TRANSLATE Sj ( C1 [,C2, C3, ...] )
Si = ROTATE Sj ( C1 [, C2, C3, ...] )
Si = REFLECT Sj ( C1 [, C2, C3, ...] )
Si = SIMILAR Sj ( C1 [, C2, C3, ...] )
Si = PIPESURFACE j ( C1, C2, C3, [C4] )
Si = COONS j (C1, C2, C3, C4[,BLEND = b, CURVATURE = cu])
Si = ISOPAR j (C1, C2, C3 )
Si = PARSURF j (C1, C2, C3, C4[,umin=u1, umax=u2, vmin=v1, vmax=v2])
Si = SPHERE j (C1 [,CENTRE=Pi] [,type=t])

```

with Si the surface number.

The value of j gives the shape number of the elements to be created, see Table 2.2.1 ($3 \leq j \leq 6$). Possibilities:

- 3 Linear triangle with 3 points
- 4 Isoparametric triangle with 6 points
- 5 Quadrilateral with 4 points
- 6 Isoparametric quadrilateral with 9 points
- 7 Isoparametric triangle with 7 points
- 8 Isoparametric triangle with 10 points
- 9 Isoparametric quadrilateral with 5 points
- 10 Triangle with 4 points

Surfaces that are not defined explicitly are treated as non-existing surfaces. These surfaces are not available in the SEPRAN programs.

For an explanation of the various possibilities see Section 2.4.

VOLUMES (optional)

COMMAND record: defines the volumes. Must be followed by data records of the type:

```

Vi = BRICK j ( N = n, M = m, L = 1, S1, S2, . . . )
Vi = USER j ( NELEM = n, NPOINT = k, S1, S2, . . . )
Vi = PIPE j ( S1, S2, S3 )

```

```

Vi = CHANNEL j ( S1, S2, S3 )
Vi = GENERAL j ( S1 )
Vi = TRANSLATE Vj ( Sk )
Vi = ROTATE Vj ( Sk )
Vi = REFLECT Vj ( Sk )

```

with V_i the volume number.

The value of j gives the shape number of the elements to be created, see Table 2.2.1. Possibilities:

- 11 Tetrahedral element with 4 points
- 12 Isoparametric tetrahedral element with 10 points
- 13 Hexahedral element with 8 points
- 14 Isoparametric hexahedral element with 27 points
- 15 Isoparametric tetrahedral element with 14 points
- 16 Isoparametric tetrahedral element with 15 points
- 17 Hexahedral element with 9 points
- 18 Tetrahedral element with 5 points

Volumes that are not defined explicitly are treated as non-existing volumes. These volumes are not available in the SEPRAN programs.

For an explanation of the various possibilities, see Section 2.5.

MESHLINE (optional)

COMMAND card; defines the one-dimensional elements, point or line elements in R^2 and R^3 . Must be followed by data records of the type:

```

LELM i = (shape = j, C1, C2 )[, int_property j = k, real_property l = r]
LELM i = (shape = 0, P1, P2 )[, int_property j = k, real_property l = r]

```

These parameters have the following meaning:

i defines the element group number.

Standard elements must be generated with increasing element group number, first all line elements, then all surface elements, and finally the volume elements.

SHAPE = j defines the shape number of the standard element. At this moment the following values are available:

$j=0$ point elements

$j=1$ linear elements

$j=2$ quadratic elements

$j=-1$ the nodal points on the curves $C1$ to $C2$ are considered as the nodal points of one super-element. Hence one element is created containing all these points.

C1,C2 line elements are generated along the curves $C1$ to $C2$; when $C2$ is not given, only curve $C1$ is used.

P1,P2 the points $P1$ to $P2$ are used as point elements; when $P2$ is not given, only point $P1$ is used.

int_property $j = k$ with this option the user may connect one or more integer number connected to the element group. The sequence number j defines the sequence number of the property, k its value. Several or none (default) integer properties may be connected to each element group.

These properties may later on be reused in the computational program, for example to distinguish types of element groups.

real_property $j = k$ has the same meaning as `int_property`, however, in this case it concerns real properties.

These properties may later on be reused in the computational program.

MESHSURF (optional)

COMMAND record: defines the two-dimensional elements in R^2 or surface elements in R^3 . Must be followed by data records of the type:

```
SELM i = ( S1, S2 )[, int_property j = k, real_property l = r]
```

These parameters have the following meaning:

i defines the element group number.

Standard elements must be generated with increasing element group number, first all line elements, then all surface elements, and finally the volume elements.

S1, S2 the elements generated on the surfaces S1, S1 + 1, . . . , S2 are appended to the mesh.

When S2 is not given only surface S1 is used.

int_property $j = k$ See the block MESHLINE.

real_property $j = k$ See the block MESHLINE.

MESHVOLUME (optional)

COMMAND record: defines the three-dimensional elements. Must be followed by data records of the type:

```
VELM i = ( V1, V2 )[, int_property j = k, real_property l = r]
```

These parameters have the following meaning:

i defines the element group number.

Standard elements must be generated with increasing element group number, first all line elements, then all surface elements, and finally the volume elements.

V1, V2 the elements generated on the volumes V1, V1 + 1, . . . , V2 are appended to the mesh. When V2 is not given, only volume V1 is used.

int_property $j = k$ See the block MESHLINE.

real_property $j = k$ See the block MESHLINE.

Remarks

- When MESHLINE, MESHSURF nor MESHVOLUME are given, it is assumed that there is only one type of internal element, with element group number 1. This element is a line element when no surfaces nor volumes are defined, and a surface element when no volumes are defined. Otherwise it is a volume element. Of course the shapes of elements in different sub meshes must be equal.
- The element group sequence numbers must be given in increasing sequence, however, it is allowed to make a jump in these sequence numbers. So it is possible that an element group sequence number is skipped. In that case the number of elements corresponding to that element group is zero and the type number of the elements as defined in the computational part of the program is set equal to zero.

Special purpose elements

MESHCONNECT (optional)

COMMAND record: defines elements that connect user points or nodal points in curves or surfaces.

Also it is possible to connect elements at curves or surfaces. In this way higher dimensional connection elements arise.

Must be followed by data records of the type:

```

CELMi = POINTS ( P1, P2 )
CELMi = CURVES 1 ( C1, C2 )
CELMi = SURFACES [j] (S1, S2 ) [, EXCLUDE ( [ALL] [, C3, C6] )

```

with i the element group number. Standard elements must be generated with increasing element group number, first all line, surface and volume elements, and then the connected elements.

When $CELMi = POINTS (P1,P2)$ is defined, an element is created from user point $P1$ to user point $P2$.

When $CELMi = CURVESj (C1,C2)$ is defined, elements are defined from nodal points on curve $|C1|$ to nodal points on curve $|C2|$, or from elements at curve $|C1|$ to elements on curve $|C2|$. When $C1$ is positive the elements are created in forward direction starting from the first position, when $C1$ is negative the elements are created in reversed order. The same rules are valid for $C2$.

The parameter l consists of two parts: $l = j + 100 \times EXCLUDE$. If $j=0$ all corresponding nodal points on the curves are connected. The statement $CELMi = CURVES0 (C1,C2)$ connects all nodal points on the curves $C1$ and $C2$. When this statement is followed by for example

$$CELMi = CURVES0 (C3,C4),$$

where $C3$ and $C4$ have a point in common with $C1$ respectively $C2$, then two elements are created connecting these common points. This situation may be avoided by defining new curves consisting of the curves $C1$ and $C3$ respectively. $C2$ and $C4$. When the initial point and end point of $C1$ and also of $C2$ coincide, then only one element is defined connecting the initial points.

For $EXCLUDE$ the following possibilities are available:

$EXCLUDE = 0$ All points of the curves $C1$ and $C2$ are used as indicated by i .

$EXCLUDE = 1$ All points except the begin point of $C1$ and $C2$ are used as indicated by i .

$EXCLUDE = 2$ All points except the end point of $C1$ and $C2$ are used as indicated by i .

$EXCLUDE = 3$ All points except the begin and end points of $C1$ and $C2$ are used as indicated by i .

If $j > 0$, the parameter $IEXCLUDE$ must be zero. In this case elements at the curves $C1$ and $C2$ are connected in the direction indicated by the signs of $C1$ and $C2$. $j=1$ corresponds to linear elements to be connected, resulting in 4 node elements and $j=2$ to quadratic elements, resulting in 6 point elements.

The option $CELMi = SURFACES [j] (S1,S2)$ connects all nodes or elements on the surfaces $S1$ with corresponding nodes on $S2$. So $S1$ and $S2$ must have exactly the same topology. If $j=0$ (default value) all points of the elements are connected.

The option $EXCLUDE = (..)$, excludes the boundaries indicated between the brackets from the connection. If ALL is chosen the complete outer boundary is excluded. If curves are given explicitly then only those curves are excluded. Only the curves at the "left" surface $S1$ must be given, those at surface $S2$ are excluded in exactly the same way, since the surfaces have the same topology.

If $j=1$, the part $EXCLUDE$ may not be used. In this case elements at the surfaces $S1$ and $S2$ are connected. This means of course that the surfaces and elements at these surfaces must be similar. The number of points at the connection elements are two times the number of points at the elements in the surface $S1$.

These connection elements are treated as special elements, which also implies that they are skipped in the postprocessing part. In this way it is for example possible to define double

points on a line, that are connected by these connection elements but introduce a discontinuity.

The connection elements get special shape numbers. In case of points to be connected the shape number is -10001, in case of elements to be connected -ishape-10001, where ishape is the shape number at the curve or surface.

Auxiliary commands (The sequence of these commands is arbitrary)

INTERFACE_ELEMENTS (optional)

COMMAND record: indicates that interface elements must be defined.

The shape of the records for the interface elements is the following:

```
INTERFACE_ELEMENTS
  IELM ielgrp_1 = CURVES ( Ci_1, Cj_1, SHAPE = s )
  IELM ielgrp_2 = CURVES ( -Ci_2, Cj_2, SHAPE = s )
  .
  .
  .
  IELM ielgrp_j = SURFACES ( Sk, Si )
  .
  .
```

Curve interface elements consist of a double row of nodal points as shown in Figure 2.2.1.

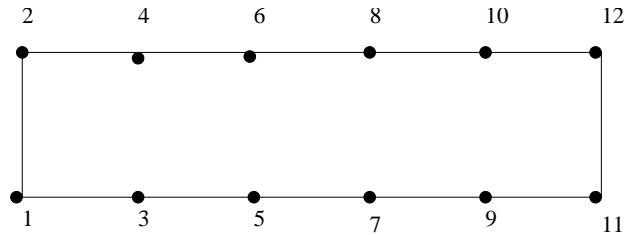


Figure 2.2.1: Example of an interface element and its local numbering

The distance between the two rows may be zero, but both rows have different nodal point numbers. The local numbering of the nodal points is as indicated in Figure 2.2.1. Hence at one side all nodes are even and at the other side they have odd sequence numbers.

At this moment interface elements may only be used in R^2 .

Curve interface elements are defined between two curves. The sign of the curves defines the direction of the interface element.

Surface interface elements are defined between two surfaces. Both surfaces must have positive sign and must be identical in the sense that they have the same subdivision in elements and the nodes must have the same coordinates. Such surfaces can for example be created by `copy_surface` (2.4).

The interface elements are created by taking the nodes of an element in the first surface followed by the nodes of the corresponding element in the second surface.

The parameters in the data records have the following meaning:

IELM *ielgrp_1* defines the element group to which the interface elements will belong. The element group number *ielgrp_1* must satisfy the same requirements as for the other standard elements, i.e. it must be equal to the preceding element group number or equal to this number plus one. The first interface element group number must be one larger than the last element group created before, for example by MESH SURF or MESHCONNECT.

CURVES (C_i, C_j) indicates that the interface element connects the curves C_i and C_j . The sign before the curve defines the direction of the element. The first curve is considered in the direction of the curve (taking the sign into account), the second curve is considered in opposite direction. For example if we must put interface elements with element group sequence number 3 between the two curves C12 and C13 in Figure 2.2.2 from left to right, than the two possible records could be:

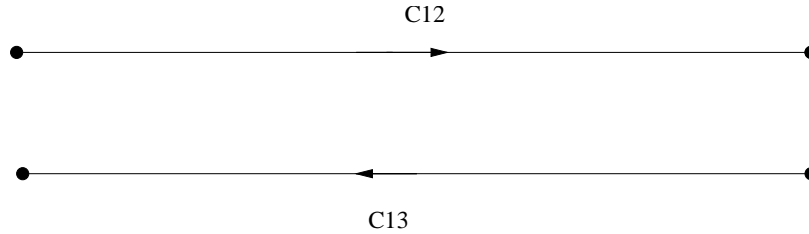


Figure 2.2.2: Example of two curves that are used to create interface elements

```
ielm3 = curves ( c12, -c13 )
```

or

```
ielm3 = curves ( -c13, c12 )
```

In the first case the odd local nodal point numbers of the interface elements are positioned at curve c12 in the second case at curve c13. In both cases numbering is from left to right.

SHAPE= s defines the shape number of the interface elements. This shape number is equal to $48 + n$, where n is the number of nodes at one side of the interface element. Hence a "linear" interface element, consisting of 4 points, two at each side has shape number 50, and a quadratic one shape number 51. The maximum shape number allowed is 59, which implies 10 nodes along each side. If **SHAPE**= s is omitted, the shape of the interface elements is detected by SEPRAN in the following way:

- If all surface elements are linear and no intermediate points are defined **SHAPE** = 50 is used.
- If intermediate points are used the shape is equal to $50 + \text{number of side points}$.
- If all surface elements are quadratic **SHAPE** = 51 is used.
- In all other cases an error message is given.

SURFACES (S_i, S_j) indicates that the interface element connects the surfaces S_i and S_j . Both surfaces must be identical.

MESHDUMMY (optional)

This special option is meant to create new element groups without actually combining this with elements. The only information required is the number of nodes in each element group. The reason to build such element groups is for example that in the computational program one creates the corresponding elements during the computations as for example sometimes using fictitious domain elements. However, for the description of the problem one does not want to know whether these new elements are already available or not.

The input for these dummy element groups is as follows:

```
delm i ( npelm = j )
```

where i refers to the new element group and j defines the number of nodes in this element group.

Remember all element groups must be ordered in a natural sequence without gaps.

RENUMBER (optional)

COMMAND record: indicates that the nodal points must be renumbered internally, in order to get an optimal "profile". In general renumbering is performed automatically except in some special cases. Renumbering has only effect on the sequence of the degrees of freedom and is not visible to the user.

This command may contain data. In that case it has the following shape:

```
RENUMBER method_part, preference_part, sequence_part, start_part, PRINT, NOT
```

with

method_part defines the type of renumbering method to be used. *method_part* consists of one of the following keywords:

```
Cuthill_McKee
Sloan
Best
```

Cuthill_McKee The classical Cuthill-McKee algorithm (1969) is used in order to optimize the profile or band width.

Sloan An algorithm due to Sloan (1986) is used in order to optimize the profile or band width.

Best Both the Sloan and Cuthill-McKee algorithm are used. The best of the original numbering and that of the two renumbering algorithms is used. Of course this option is the most expensive.

preference_part indicates how it is decided whether the original numbering or the renumbered one is used. *preference_part* consists of one of the following keywords:

profile indicates that the best of the renumbered sequence and the original sequence is used, where best is defined as the one with the smallest profile of the matrix. This is the most suitable choice for a direct solution of the linear solvers.

band indicates that the best of the renumbered sequence and the original sequence is used, where best is defined as the one with the smallest Band width of the matrix.

always indicates that the renumbering is used regardless of the fact that the profile or band width may increase. Especially for iterative linear solvers the band width nor the profile are of interest. In that case the new sequence may influence the speed of convergence and *always* may be preferable.

sequence_part makes it possible to distinguish between types of nodes in the renumbering algorithm. This option is only useful if an iterative solver is used. For a direct solver the computation time and size of the matrix is strongly increased if nodes are distinguished. *sequence_part* consists of the following keywords:

```
vertex_seq = i
mid_point_seq = i
centroid_seq = i
levels
```

Nodal points are subdivided into vertices, mid-points and centroids. All internal nodes in an element are considered as centroids, all nodes at the boundary of an element, vertices excluded, are considered as mid-side points. The action corresponding to *sequence_part* is applied after the renumbering algorithm has been applied and choices have been made. The user may give each of the series of nodes a priority number *i* between 1 and 3. The priority number is the value *i* behind the keyword *xxx_seq*.

The already renumbered nodes (possibly the old numbering) are rearranged in such a way that first all nodes with priority 1 are used, then with priority 2 and finally with priority 3. If a type of node is not indicated it gets automatically the priority number 3.

If two types of nodes have the same priority number, then their mutual sequence remains the old one.

The keyword *levels* indicates that this special renumbering of node types is applied per level rather than for the complete mesh. The notion level originates from the Cuthill-McKee algorithm. In order to make it independent of the type of renumbering scheme we define a level in the following way:

Find the neighbor of nodal point 1 with the highest nodal point number. If the nodal points are renumbered internally, the renumbered sequence is used. All nodes with sequence number at least equal to this neighbor belong to level 1.

Find all neighbors of the present level that do not belong to a level itself. All nodes with sequence number at least equal to the neighbor with maximal number belong to the next level. This process is repeated until no nodes are left.

start_part defines how the renumbering algorithm must be started. If omitted the start nodes are detected automatically using some suitable not too expensive algorithm. Otherwise this part must have one of the following shapes:

```
start = Pj
start = Ci
start = Si
```

which means the starting point is either user point P_j , or the starting nodes are all nodes at curve C_j or surface S_j .

PRINT indicates that the reduction in profile and band width of the renumbering algorithm is printed before the application of *sequence_part*, regardless of the general output level. If maximal output is required by the command SET OUTPUT ON this information will always be printed.

NOT indicates that no renumbering may be applied.

METHOD=i Remark: In previous version of SEPMESH it was only possible to use *method = i* as type of renumbering scheme. This option is still available but not longer recommended.

The value of i indicates the type of renumbering to be used according to the following rules:

```
method = 1 is equivalent to Cuthill-McKee profile
method = 2 is equivalent to Sloan profile
method = 3 is equivalent to Best profile
method = 4 is equivalent to Cuthill-McKee band
method = 5 is equivalent to Sloan band
method = 6 is equivalent to Best band
method = 7 is equivalent to Cuthill-McKee always
method = 8 is equivalent to Sloan always
```

Warning: each renumbering takes time, and the advantage compared to the other renumberings may be small, especially for small problems. Therefore, one is advised to find an optimal numbering only in the case of large problems with many unknowns.

In case of a direct linear solver (Gaussian elimination; see input block "SOLVE" of SEPCOMP) renumbering directly influences the computation time and size of the large matrix. A number of test examples indicates that the Sloan algorithm generally reduces the profile better than Cuthill-McKee, whereas Cuthill-McKee gives better results with respect to the band width. However, there are also examples with opposite outcomes. The linear solver applied to a matrix constructed with METHOD = 1,2,3 or 4 in the input block "MATRIX" in SEPCOMP, uses a profile method, so in that case Sloan may be the best choice.

In case of an iterative linear solver the ordering may influence the convergence speed. This is certainly the case if a preconditioner is used.

NORENUMBER (optional)

COMMAND record: indicates that the renumbering of nodal points must be suppressed. This keyword is in fact the same as RENUMBER NOT.

NOTOPOLOGY (optional)

COMMAND record: indicates that calling of the topology subroutine is suppressed. This means that no renumbering takes place and that no information about neighbors will be computed. This option may only be used if the mesh is created and immediately written to the file `meshoutput` and no renumbering is required. Also if spectral elements are used in combination with an iterative solution technique and possibly a finite element preconditioning.

INTERMEDIATE POINTS (optional)

With this command record the user may extend linear elements with extra points. It is possible to create new elements defining

- extra internal points on the sides of the elements (for each side of an element the same number of extra points)
- extra internal points on the faces of the elements (for each triangle or quadrilateral the same number of extra points)
- extra internal points in each volume element (tetrahedron or hexahedron).

This command expects data records of the following type:

definition of points, type of subdivision

where,

definition of points defines the number of intermediate points that must be added. This command has the shape:

sidepoints = ns , facepoints = nf , volmpoints = nv (No spaces allowed in the keywords)

Sidepoints = ns defines the number of points on a side of an element, the two end points excluded. With a side we always mean a one-dimensional quantity, i.e. for a line element this is the element itself, for a two-dimensional element it is the side of an element and for a three-dimensional element it is an edge. So ns should be at least 1. Points per side may not be omitted.

Facepoints = nf defines the extra number of points in a surface element, the boundary points excluded. The number of facepoints nf in a triangle is restricted to the choices 0 and 1 and in a quadrilateral it is restricted to 0 and $ns \times ns$ if ns is the number of sidepoints given.

Volmpoints = nv defines how many internal points in the volumes have to be created, boundary points excluded. At this moment the number of volume- points nv in a tetrahedron is restricted to the choices 0 and 1 and in a hexahedron it is restricted to 0 and $ns \times ns \times ns$ if ns is the number of sidepoints given.

In stead of facepoints and volmpoints the user may give the command

midpoints = filled

In that case the program automatically takes the filled option for all triangles ($nf = 1$), quadrilaterals ($nf = ns \times ns$), tetrahedrons ($nv = 1$) and hexahedrons ($nv = ns \times ns \times ns$) See for an example Figure 2.2.3. **type of subdivision** defines how the extra points must be positioned on the sides (and as consequence internally). The following possibilities are available:

subdivision = equidistant

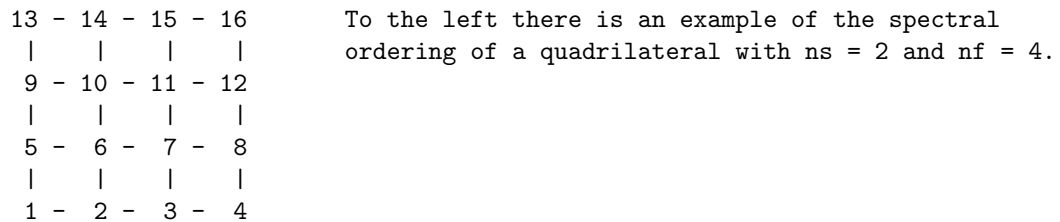
subdivision = Legendre

If subdivision is equidistant (which is the default value) is used, the intermediate points are positioned at equidistant step sizes, if Legendre is used the division of the nodes is according to the characteristic values of a Gauss-Legendre polynomial.

Remark: After the use of the phrase intermediate points spectral elements have been created. The ordering of spectral elements based on quadrilaterals and hexahedrons differs from the ordering as given for the standard SEPRAN elements in Table 2.2.1. For an example see Figure 2.2.3.

The following ordering rules for spectral elements are used in SEPRAN:

Quadrilateral - based : Ordering of the nodes is line following line. Example:



Hexahedron - based: Ordering of the nodes is line following line and plane following plane.

As a consequence of these rules a spectral quadrilateral with four nodes and a spectral hexahedron with eight nodes has an ordering of nodes that differs from the ordering used in Table 2.2.1. The spectral ordering is respectively :

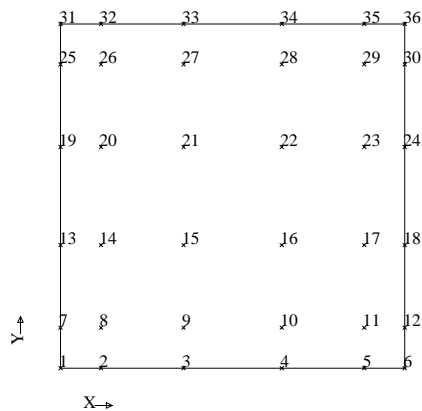


Figure 2.2.3: Example of a filled quadrilateral with corresponding numbering.

CHANGE_COORDINATES indicates that the user wants to change the co-ordinates that have been computed by the mesh generator in a function subroutine. This option might for example be used, if in first instance a simple mesh is generated, but afterwards the coordinates must be adapted in order to get a more complex mesh. A typical example is a brick in 3D that is in first instance straight, but where the user wants a slightly curved surface given by some given function. First the brick and co-ordinates are generated and afterwards the co-ordinates are changed.

If this keyword is used, the user must add a subroutine FUNCCKOOR (2.2.1) as described in Section 2.2.1, in which he describes how the co-ordinates must be adapted.

This keyword must be followed by data records describing how the co-ordinates must be adapted. These data records consist of two parts: the position part and the function description like:

```
position_part, function_description, when_part
```

The position part may have the following form:

```
all
POINT Pi to Pj
CURVE Ci to Cj
SURFACE Si to Sj
VOLUME Vi to Vj
```

all means that the co-ordinates of all points must be changed.

POINT Pi to Pj means that the co-ordinates of the user points P_i to P_j must be changed.

If only P_i is given, of course only P_i is changed.

The coordinates of the user points are changed before the curves are generated.

CURVE Ci to Cj means that the co-ordinates of the curves C_i to C_j must be changed. If only C_i is given, of course only C_i is changed.

SURFACE Si to Sj means that the co-ordinates of the surfaces S_i to S_j must be changed.

If only S_i is given, of course only S_i is changed.

VOLUME Vi to Vj means that the co-ordinates of the volumes V_i to V_j must be changed.

If only V_i is given, of course only V_i is changed.

The default value is all.

The function description must have the following form:

```
FUNC_CHAN = i
```

The parameter i defines the choice parameter ICHOICE_CHANGE in subroutine FUNCCKOOR (2.2.1).

If this part is omitted $i = 1$ is assumed.

Hence if no data records are given all nodes may be changed by a call to FUNCCKOOR (2.2.1) with ICHOICE_CHANGE = 1.

The when part must have the following form:

```
IMMEDIATELY
AT_END
```

These parameters decide when the coordinates are changed.

IMMEDIATELY The coordinates are changed immediately after the creation of the specific coordinates, i.e.

The coordinates of the user points are changed before the curves are generated.

The coordinates of the curves are changed before the surfaces are generated.

The coordinates of each surface is changed immediately after each surface has been generated.

The coordinates of each volume is changed immediately after each volume has been generated.

The coordinates as described by ALL are changed after all coordinates have been created.

AT_END In this case the coordinates are changed after all coordinates have been created.

Default value: AT_END

OBSTACLES (optional) COMMAND record: indicates that obstacles are defined. These obstacles may be used in free surface problems to indicate that the free boundary is not allowed to cross these obstacles. At this moment only obstacle curves and surfaces are allowed. These obstacles must always be closed. The obstacles are always plotted when a mesh or a boundary is plotted. However, as long as there are no nodal points of the mesh connected to the obstacle, the obstacle has no special meaning. The OBSTACLES record must be followed by data records describing the obstacles. These records have the following shape:

COBS1 = Cj

COBS2 = Ck

SOBSi = Sj

COBS i = C j defines obstacle i as a curve obstacle. The boundary of the obstacle is given by the curve Cj .

The boundary must consist of one curve only and this curve must be closed, i.e. the begin and end point must be the same. Using the option curves of curves it always possible to create such a curve from an ensemble of other curves.

SOBS i = S j defines obstacle i as a surface obstacle. The boundary of the obstacle is given by the surface Sj .

The boundary must consist of one surface only and this surface must be closed. Using the option surfaces of surfaces it always possible to create such a surface from an ensemble of other surfaces.

Obstacles may become active in the computational part. In that case the intersection with the standard mesh is considered.

We distinguish between the following types of elements

- Elements that are completely inside the obstacle. All elements with all nodes inside the obstacle or on the boundary of the obstacle belong to this group. If there are elements with all nodes on the boundary, these elements are considered to be inside the obstacle, even if these elements are actually outside the obstacle. This may be the case around sharp corners.
- Elements that are partly inside the obstacle. All elements with at least one point inside the obstacle (not on the boundary) and one point outside the obstacle (also not on the boundary) are considered to be partly inside the obstacle.
- The rest of the elements is considered to be outside the obstacle.

Once the obstacle is made active in the computational part we also distinguish between the various nodes in the mesh:

- **IN_ALL_OBSTACLE** All nodes that are inside or on the boundary of an obstacle belong to this group.

- **IN_INNER_OBSTACLE** In this case it concerns all nodes that are in elements that are completely inside an obstacle. If an element is completely inside the obstacle but has some points on the boundary of the obstacle then all nodes of this element except those on the boundary belong to this group
Compared to **IN_ALL_OBSTACLE**, this means that nodes on the boundary of the obstacle are excluded, as well as nodes of elements that are partly outside the obstacle even if they are inside the obstacle.
- **IN_BOUN_OBSTACLE** This refers to the nodes in the obstacle of those elements of the mesh that are partly outside the obstacle.
So all the points that are excluded in `in_inner_obstacle` but are part of `in_all_obstacle` belong to `in_boun_obstacle`.
- **ON_BOUN_OBSTACLE** refers to the nodes of those elements of the mesh that are on the boundary of the obstacle.
- **INON_BOUN_OBSTACLE** refers to a subset of the class of nodes in `ON_BOUN_OBSTACLE`. It concerns those nodes of the boundary of the obstacle that are only in volume (R^3) or surface (R^2) elements that itself are part of the obstacle. So in fact these are nodes that are on the boundary of the obstacle and possibly on the outer boundary of the region but not in elements that are outside the obstacle.

PLOT (optional) COMMAND record: indicates that the points, curves, the surfaces and the mesh must be plotted, each on a new picture. This COMMAND record may contain data. In that case it has the following shape:

```
PLOT ( LENGTH=l, YFACT=y, JMARK=j, NUMSUB=n, CURVE=c, NODES=no, USERPOINTS=u,
      COLOUR=c1, SUPPRESS=su, ROTATE=r, NOMESH, NOSUBMESH, REN_PLOT,
      EYEPOINT=(x_e,y_e,z_e)), ORIENTATION=i
```

LENGTH = l defines the length of the plot in centimeters.

Default value: depending on the computer installation, usually 15 or 20.

In order to be compatible with old SEPRAN version instead of **LENGTH** also **PLOTFM** may be used.

SCALE = s may be used instead of **PLOTFM** = l . In that case the size of the plot of the mesh and sub meshes is not fixed, but determined by the co-ordinates of the mesh and sub meshes.

Hence the length in the x-direction is given by $s dx$ and the length in the y-direction by $s dy$, where dx is the maximal difference of the x-co-ordinates in the mesh or sub mesh, and dy the same for the y-co-ordinates.

YFACT = y : Scale factor; all y-co-ordinates are multiplied by y before plotting the mesh. $y \neq 1$ should be used when the co-ordinates in x and y direction are of different scales, and hence the picture becomes too small. Default value: 1.

JMARK = j : Indication of how the plot of the mesh must be made. Possibilities:

0,3 Each nodal point is marked with a star and its nodal point number.

1,4 Each nodal point is marked with a star. Nodal point numbers are **not** plotted.

2,5 Nodal points are not marked, nor are nodal point numbers plotted.

When $JMARK < 3$ all element numbers are plotted in the centroid of the elements, when $JMARK \geq 3$ no element numbers are plotted.

Default value: 5

NUMSUB = n : The sub meshes with numbers \leq NUMSUB are not plotted. Default value: 0.

CURVE = c : indicates if the curves must be plotted in a separate picture. Possible values:

0 Curves are not plotted.

- 1 Curves are plotted without curve number.
- 2 Curves are plotted provided with curve number.

Default value: 2.

NODES = *no* : indicates if nodes along the curves must be plotted in the picture containing the curves. So this parameter makes only sense for $c > 0$. Possible values:

- 0 Nodes are not plotted.
- 1 Each node is indicated with a CROSS-symbol.
- > 1 Each node is indicated with a symbol from the symbol table. The sequence number on the symbol table is equal to $no - 1$.

Which symbols are stored in the symbol table depends on your plotting package.

Default value: 0.

USERPOINTS = *u* : indicates if and how user points must be plotted. Possible values:

- 0 User points are not plotted
- 1 Plot user points without numbers.
- 2 Plot user points with numbers.

Default value: 0.

COLOUR = *cl* : indicates if colors must be used to distinguish certain quantities. The most important object of the parameter is to plot each element group with a different color in the final plot of the mesh. Possible values:

- 0 Only one color (the standard color) is used.
- 1 The default colors are used. Each element group gets a different color.
- > 1 Colors are used. Each element group gets a different color. The first element group gets color cl , the next one $cl + 1$, etc.

Default value: 0.

SUPPRESS = *su* : indicates if pictures must be provided with texts ($su = 1$) or not ($su = 0$).

Default value: 1.

ROTATE = *r* : indicates whether plots must be rotated over 90 degrees or not. Possibilities:

- 0 The plots are made such that the plotting paper used is minimal.
- 1 The plot is not rotated.
- 2 The plot is always rotated over 90 degrees.

Default value: 0.

EYEPOINT = (x_e, y_e, z_e) : makes only sense in case of a 3D region. The use of EYEPOINT indicates that a final 3D mesh is plotted with hidden lines. In the case that there are many elements this plot may take much time. (x_e, y_e, z_e) defines the point where the observer is positioned.

Remark: EYEPOINT must be written as one word.

In case of a 3D mesh only the sub meshes are plotted, without the removal of hidden lines, except when EYEPOINT is given. For a final plot of the complete region with hidden lines removed one may either use program PLOTMESH or the subroutines PLOTM3 or PLOT3M, see the Programmers Guide.

NOMESH means that no separate plot of the mesh is made. Hence only the other plots (plots of curves and submeshes) are made.

NOSUBMESH : indicates that no separate plots of surfaces and volumes are made. This keyword makes the keyword *NUMSUB* superfluous.

REN_PLOT makes a separate plot of the mesh, where all nodes are provided with their renumbered node numbers. Each node is provided with a mark. Remember that the renumbering of the nodal points is only internal, the original numbering is used for output and input purposes.

In order to plot the original numbering use *JMARK* = 0 or 3

ORIENTATION = i defines the orientation of the base vectors in case of a three-dimensional plot with hidden lines. Hence this option is only applicable in combination with EYE-POINT. The following values of i are allowed:

- 1 The standard orientation of the axis (x-y-z) is used.
- 2 The orientation is defined as z-x-y
- 3 The orientation is defined as y-z-x

REFINE [n TIMES] (optional)

COMMAND record: indicates that the mesh created must be refined n times, i.e., all edges must be subdivided into 2^n pieces. If n times is omitted $n = 1$ is assumed.

TRANSFORM [TYPE= t] (optional)

COMMAND record: indicates that a mesh consisting of linear triangles and tetrahedrons only must be transformed to a mesh consisting of elements of type t only. The original mesh may contain linear elements only.

The parameter t has the following meaning:

- $t = 4$ means that all linear triangles are transformed into quadratic triangles and each linear tetrahedron into quadratic tetrahedrons.
- $t = 5$ means that all linear triangles are subdivided into bi-linear quadrilaterals and all linear tetrahedrons into trilinear hexahedrons.
- $t = 6$ means that all linear triangles are subdivided into bi-quadratic quadrilaterals and each linear tetrahedron into triquadratic hexahedrons.
- $t = 7$ means that all linear triangles are transformed into extended quadratic triangles, i.e. quadratic triangles with an extra point (7) in the center.

If TYPE= t is omitted, $t = 5$ is assumed.

CHECK_LEVEL= i (optional)

With this command the quality of the mesh may be checked. The value of i defines which checking level is available. Possible values:

- 0 No special checks. This is the default.
- 1 The volume of all elements is checked. If some element is too distorted to be acceptable for an element subroutine, an error message is given.
- 2 See level 1. In this case not only the elements are checked but also the minimum and maximum value of the volumes of all elements is printed. The ration of these two number gives an indication of the quality of the mesh. A small ratio of volume largest element divided by volume smallest element implies a smooth mesh. A large ratio may indicate that the mesh has a poor quality and the user should inspect the mesh plot carefully.
- 3 See level 2. In this case also a selected part of the subarrays of KMESH is printed. This option is meant for debugging purposes.
- 4 See level 2. In this case also array KMESH and all the subarrays of KMESH are printed. This option is meant for debugging purposes.

PARALLEL, options (optional)

If this keyword is used, it means that the user wants to use parallel computations, i.e. the process is carried on a number of computers at the same time. Of course this is only possible if you have a parallel cluster at your disposal.

The following options are available: (all in one line)

```
method = m, num_processors = i
```

These subkeywords have the following meaning:

method = m defines the method that is used to subdivide the mesh into submeshes. These submeshes act as a block in a multi-block approach, also called domain decomposition. The following options for *m* are available:

```
surfaces
volumes
layers
blocks
elements_serial
threed_blocks
```

These methods have the following meaning

surfaces means that each block coincides with one surface in case surfaces are the entities of highest dimension. If volumes form the entities of highest dimension, each block coincides with one volume.

volumes has exactly the same meaning as surfaces

layers means that the region is subdivided in exactly `num_processors` layers of elements. Each such layer forms a block. These elements are created in the following way:

- First the nodes are subdivided into groups consisting of approximately `npoint / num_processors` nodes, where the (possibly renumbered) sequence of the nodes is used.
- Next all elements are added to layer *i* if they do not belong to a previous layer and if all nodes of the element belongs to the set of points in the groups 1 to *i*.

elements_serial is specially meant for test purposes. The number of elements in each region is approximately the same.

Elements are not renumbered and are subdivided in natural sequence

threed_blocks is specially meant for test purposes. A 3d block of equal number of elements (hexahedrons) in each direction is subdivided into subblocks of equal size (block subdivision). The number of blocks must be a power of 3.

At this moment the default value is layers. This may be changed in the future.

num_processors = i defines the number of processors or blocks. If **method = surfaces** or **method = volumes** is used, this keyword is neglected.

Default value: 8

At this moment we have the following restrictions:

1. MPI must run on the parallel computer, since our implementation is based on mpi.

If the option `parallel` is found, `sepmesh` creates, besides the standard file `meshoutput`, also the files `meshoutput_par.000`, `meshoutput_par.001` to `meshoutput_par.xxx`, where `xxx` is the number of processors.

If the keyword `parallel` is found, `sepcomp` (or your own program) will solve the problem in parallel on a number of processors defined by the number of surfaces or volumes.

For the computational program you have to use the command `sempi` in order to link and run or the command

END (mandatory)

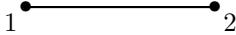
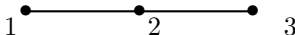
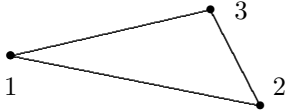
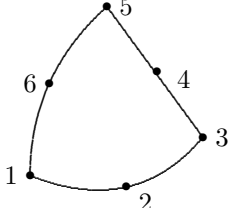

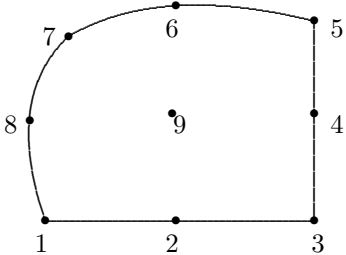
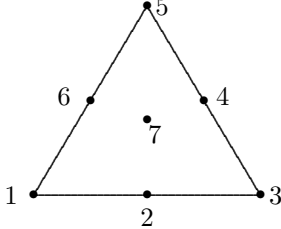
End of the input for program SEPMESH.

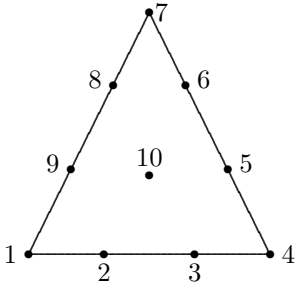
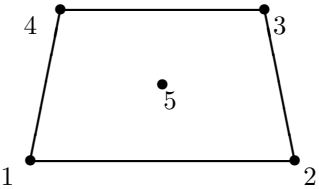
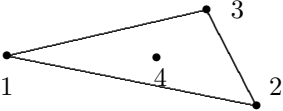
Remark:

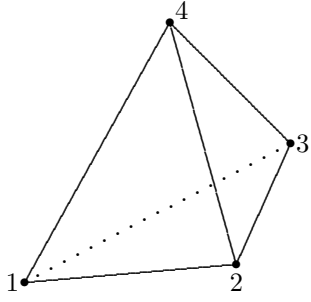
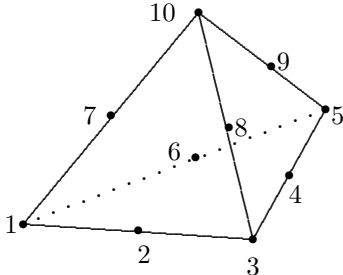
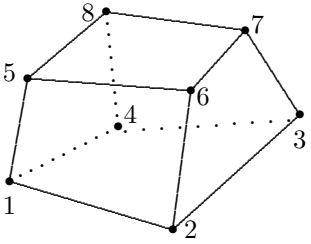
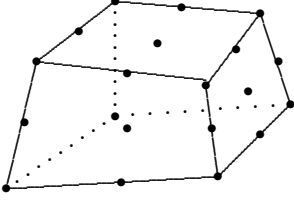
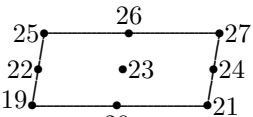
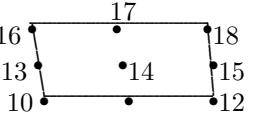
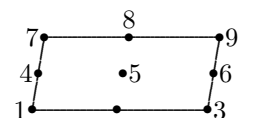
The input must be given in the sequence:

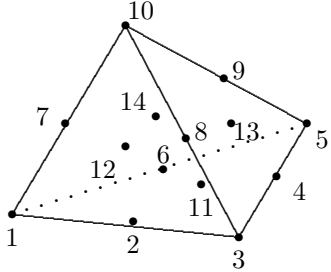
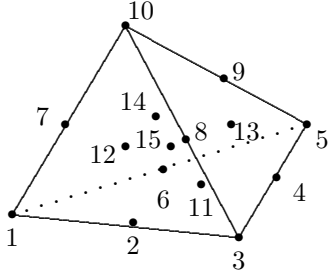
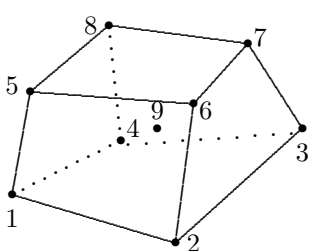
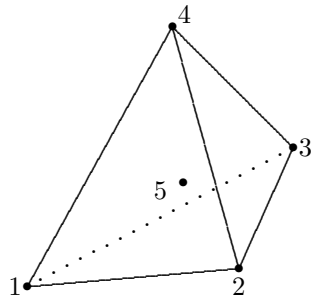
MESH record
optional commands
POINTS
CURVES
SURFACES
VOLUMES
MESHLINE
MESHSURF
MESHVOLUME
MESHCONNECT
INTERFACE_ELEMENTS
MESHDUMMY
special purpose elements
auxiliary commands
END

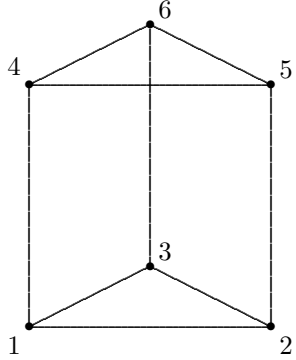
Table 2.2.1 Standard elements for mesh generation

shape number	shape	name
1		line element with 2 points
2		line element with 3 points
3		triangle with 3 points
4		isoparametric triangle with 6 points
5		quadrilateral with 4 points
6		isoparametric quadrilateral with 9 points
7		isoparametric triangle with 7 points

shape number	shape	name
8		isoparametric triangle with 10 points not available in the mesh generator
9		quadrilateral with 5 points
10		triangle with 4 points

shape number	shape	name
11		tetrahedral element with 4 points
12		isoparametric tetrahedral element with 10 points
13		hexahedral element with 8 points
14	 <p>numbering:</p>  <p>upper face</p>  <p>mid face</p>  <p>bottom face</p>	isoparametric hexahedral element with 27 points

shape number	shape	name
15		isoparametric tetrahedron with 14 points comp. with 12, extra points are the centroids of the faces
16		isoparametric tetrahedron with 15 points comp. with 15, extra point is the centroid of the tetrahedron
17		hexahedral element with 9 points point 9 is centroid
18		tetrahedral element with 5 points

shape number	shape	name
21		isoparametric prism with 6 points

2.2.1 Subroutine FUNCCOOR

Description

Subroutine FUNCCOOR is used to modify the co-ordinates that have been generated by the SEPRAN mesh generator. The call to this subroutine is activated by the input block CHANGE_COORDINATES.

FUNCCOOR must be written by the user.

Heading

```
subroutine funccoor ( icoicechange, ndim, coor, nodes, numnodes )
```

Parameters

DOUBLE PRECISION COOR(NDIM,*)

INTEGER ICHOICECHANGE, NDIM, NODES(NUMNODES), NUMNODES

ICHOICECHANGE is identical to the parameter *i* corresponding to FUNC_CHAN = *i* in the input block CHANGE_COORDINATES. This parameter may be used by the user to distinguish between several cases.

NDIM defines the dimension of the space.

COOR is a two-dimensional array of size NDIM × NPOINT (number of nodal points), containing all co-ordinates of the mesh.

At input the co-ordinates as generated by the mesh generator are stored in this array. At output the user may have changed some or all of these co-ordinates.

NODES is an integer array of length NUMNODES containing the nodal point numbers of all nodes that are given by the position part in the input block.

NUMNODES length of array NODES.

Input

The parameters ICHOICECHANGE, NDIM and NUMNODES have been given a value by the mesh generator before the call of FUNCCOOR.

The arrays NODES and COOR have been filled by the mesh generator before the call of FUNCCOOR.

Output

The user is supposed to have changed the values of the co-ordinates in array COOR.

Interface

Subroutine FUNCCOOR must be programmed as follows:

```
SUBROUTINE FUNCCOOR ( ICHOICECHANGE, NDIM, COOR, NODES, NUMNODES )
  IMPLICIT NONE
  INTEGER ICHOICECHANGE, NDIM, NUMNODES, NODES(NUMNODES)
  DOUBLE PRECISION COOR(NDIM,*)
  INTEGER i, nodal_point_number

  if ( ICHOICECHANGE==1 ) then

    do i = 1, NUMNODES
      nodal_point_number = NODES(i)
      COOR(1,nodal_point_number) = ...
      COOR(2,nodal_point_number) = ...
      COOR(3,nodal_point_number) = ...
    end do

  else
    .
    .
    .
    .
  end if

END
```

2.3 Curve generators

In this section the various curve generators will be treated. For the definition of the curves the user may specify the number of nodal points on a curve as well as the distribution of these points. Another possibility is to define an approximate length of the elements in the end points of the curves. Elements in between are defined such that the mesh size increases or decreases monotone and smoothly from one end to the other. When the user wants to utilize this possibility he must give the command COARSE, and give a unit length (UNIT). Furthermore each user point must be provided with a so-called coarseness (c). Then the approximate length of the elements in the surroundings of these points is equal to $c \times \text{UNIT}$, depending on the type of function that is used for the creation of the curve.

These curve generators are activated by the command CURVES in the input for the program SEPMESH. In Section 2.2 the following types of curve generators have been defined:

LINE
CLINE
ARC
CARC
USER
PARAM
CPARAM
SPLINE
CSPLINE
CURVES
TRANSLATE
ROTATE
REFLECT
SPCURVE
PROFILE
CPROFILE
CIRCLE
OWN_CURVE
ELL_ARC
CELL_ARC

These curve generators have the following global functions:

LINE generates a straight line between two end points.

CLINE generates a straight line between two end points using the concept of coarseness.

ARC generates an arc from begin point to end point; the centroid must be given.

CARC generates an arc from begin point to end point; the centroid must be given. The division of elements is based on the concept of coarseness.

USERi the user gives all coordinates of the nodal points on the line.

SPLINE A curve is defined by a spline through a number of points.

CSPLINE A curve is defined by a spline through a number of points. The division of elements is based on the concept of coarseness.

PARAM The user defines a curve by a function subroutine FUNCCV (2.3.1) using a parameter representation.

CPARAM The user defines a curve by a function subroutine FUNCCV (2.3.1) using a parameter representation. The division of elements is based on the concept of coarseness.

TRANSLATE Copy a curve and translate it over a fixed distance.

ROTATE Copy a curve and translate and rotate this new curve.

REFLECT Make a reflection of a curve with respect to a given line.

CURVES Create a new curve by combining old curves.

SPCURVE Construct a curve through a number of old curves. The old curves are made obsolete.

PROFILE Define a curve with a specified profile.

CPROFILE Define a curve with a specified profile. The division of elements is based on the concept of coarseness.

CIRCLE Define a circle in R^2 or R^3 with center and begin point.

OWN_CURVE The user defines a curve by a user written subroutine OWN_CURVE (2.3.2).

ELL_ARC generates an arc along an ellipse from begin point to end point; the centroid of the ellipse must be given. This option is only available in R^2 . So this almost identical to ARC, except that the radius does not have to be constant.

CELL_ARC is the coarse variant of ELL_ARC, comparable to CARC.

The following options have a global meaning for each of the curves, where they may be used:

NELM= n gives the number of elements that must be created along the curve (linear or quadratic depending on the value of j).

RATIO= r indicates the options for distribution of the nodal points. Possibilities:

$r=0$: equidistant mesh size (default)

$r=1$: the last element is f times the first one.

$r=2$: each next element is f times the preceding one.

$r=3$: the last element is $1/f$ times the first one.

$r=4$: each next element is $1/f$ times the preceding one.

$r=5$: the subdivision of the elements is symmetric with respect to midpoint of the curve.
The last element of each half is f times the first one.

$r=6$: See $r=5$.

Each next element of each half is f times the preceding one.

$r=7$: See $r=5$, but now with a factor $1/f$.

$r=8$: See $r=6$, but now with a factor $1/f$.

FACTOR= f the factor to be used when $r > 1$. Default: $f=1$.

NODD= o The value of o defines whether the number of end points of the elements on the curve is free, odd or even. Possibilities:

$o=0,1$: free

$o=2$: number of end points odd, which means that an even number of elements is generated along the curve.

$o=3$: number of end points even, which means that an odd number of elements is generated along the curve.

INIT $=t_0$ gives the starting value of the parameter t along the curve. This parameter t is used in the call of subroutine FUNCCV (2.3.1).

Default value: $t_0 = 0$.

END $=t_1$ gives the end value of the parameter t along the curve.

Default value: $t_1 = 1$.

Extended description of the various curve generators

LINE The input for LINE must be defined in the following way:

$$Ci = \text{LINE}j (P1, P2, \text{NELM}=n [, \text{RATIO}=r, \text{FACTOR}=f])$$

This means that a straight line is generated from point P1 to point P2, with a division of elements as indicated by the options.

CLINE The input for CLINE must be defined in the following way:

$$Ci = \text{CLINE}j (P1, P2 [, \text{NODD}=o])$$

When CLINE is used, a straight line is generated from P1 to P2, where the coarseness as given in the points P1 and P2 is used to define the elements. The value of o indicates whether the number of end points of the elements is free, odd or even.

ARC The input for ARC must be defined in the following way:

$$Ci = \text{ARC}j (P1, P2, P3, \text{NELM}=n [, \text{RATIO}=r, \text{FACTOR}=f])$$

When ARC is used an arc is generated from point P1 to P2 with center P3.

In R^2 the sign of P3 indicates the direction of the arc. When P3 is given the arc is created counter clockwise, when -P3 is given it is created clockwise.

In R^3 the smallest arc from point P1 to point P2 is chosen. If the angle is exactly 180° , that is if the points P1, P2 and P3 are positioned on a straight line, then the direction of the arc is undefined. The arc is positioned in the plane through P1, P2 and P3.

Remark: The user may give the centroid of an arc in an inaccurate way. The centroid is computed as the projection of the centroid given by the user on the line orthogonal to the line through the two end points of the arc and going through the midpoint of these two points. See Figure 2.3.1. Of course this is only possible when initial point and end point of the arc are essentially different points.

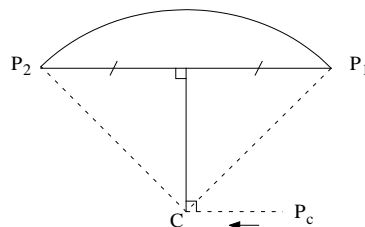


Figure 2.3.1: Computation of the centroid of an arc

CARC The input for CARC must be defined in the following way:

$$C_i = \text{CARC}_j (P_1, P_2, P_3 [, \text{NODD}=\alpha])$$

When CARC is used, an arc is generated from point P1 to P2 with center P3, where the coarseness as given in the points P1 and P2 is used to define the elements. For the value of α , see CLINE.

In R^2 the sign of P3 indicates the direction of the arc. When P3 is given the arc is created counter clockwise, when -P3 is given it is created clockwise.

In R^3 the smallest arc from point P1 to point P2 is chosen. If the angle is exactly 180° , that is if the points P1, P2 and P3 are positioned on a straight line, then the direction of the arc is undefined. The arc is positioned in the plane through P1, P2 and P3.

See also the remarks in ARC concerning the accuracy of the mid point.

USER The input for USER must be defined in the following way

$$C_i = \text{USER}_j (P_1, P_2, P_3, \dots, P_n)$$

When USER1 is used, the curve is defined by the points P1, P2, P3, . . . , Pn in that sequence, when USER2 is used the curve is defined by the same points, but also the midpoints are generated exactly in the middle of the lines (P1, P2), (P2, P3), . . . , (Pn-1 , Pn).

SPLINE: The input for SPLINE must be defined in the following way:

$$C_i = \text{SPLINE } j (P_1, P_2, \dots, P_m, \text{NELM}=\alpha [, \text{RATIO}=\beta, \text{FACTOR}=\gamma] \& \\ [, \text{TYPE}=\delta [, \text{tang}=\text{Pk}, \text{tang}=\text{Pl}]] [\text{ALPHA} = \epsilon])$$

When SPLINE_i is used, the curve is defined by a cubic spline through the points P1, P2, P3, . . . , P_m. At least 3 points are required. The curve passes through all points P1, P2, . . . P_m and has continuous derivative and curvature. None of the internal points P2, P3, . . . P_{m-1} is necessarily a nodal point. These points are not connected to nodal points either, so they can not be used in later stages of SEPRAN programs. In each sub interval [P_i, P_{i+1}] the curve is a polynomial of the third degree. In SEPRAN the following SPLINE types are accessible by the option TYPE=*t*:

t=1 The tangent of the curve is zero in the end points P1 and P_m.

t=2 In [P1,P2] and [P_{m-1},P_m] the curve is a polynomial of degree 2.

t=3 The spline is a closed curve, i.e. there is no begin or end point: P1 and P_m must be the same point!

t=4 The spline is defined by the points P1, P2, . . . , P_n with prescribed derivatives in both begin and end point of the curve. The user must give the direction and magnitude of these derivatives by the phrase tang = P_k, tang = P_l. It is assumed that the vector P_k - P1 is the derivative in the initial point P1, and the vector P_l - P_m in the end point P_m. Hence the points P_k and P_l must have been defined already in the section points. Note that P_k is connected with the starting point and P_l with the end point.

If TYPE = *t* is omitted, *t*=1 is assumed.

The division of elements on the curve is defined by the parameter *i*, NELM=*n*, and (optional) RATIO=*r*, FACTOR=*f*.

The factor α defines the type of spline to be used. $\alpha = 1$, the default value, gives the so-called cord-spline, $\alpha = 0.5$, gives the so-called centripetal-spline and $\alpha = 0$, so-called uniform-spline. Any value of α between 0 and 1 is permitted. Which value of α is most suited depends on the taste of the user.

CSPLINE The input for CSPLINE must be defined in the following way:

```
Ci = CSPLINE j ( P1, P2, ... ,Pm [, NODD=o ] [,TYPE=t [,tang=Pk, tang=P1] ] &
               [ ALPHA = a ] )
```

When CSPLINE i is used, the curve is defined by a cubic spline through the points P1, P2, P3, . . . , P m . At least 3 points are required. The curve passes through all points P1, P2, . . . P m and has continuous derivative and curvature. None of the internal points P2, P3, . . . P m is necessarily a nodal point. These points are not connected to nodal points either, so they can not be used in later stages of SEPRAN programs. In each sub interval [P i , P i + 1] the curve is a polynomial of the third degree. In SEPRAN the following SPLINE types are accessible by the option TYPE= t :

$t=1$ The tangent of the curve is zero in the end points P1 and P m .

$t=2$ In [P1,P2] and [P m - 1,P m] the curve is a polynomial of degree 2.

$t=3$ The spline is a closed curve, i.e. there is no begin or end point: P1 and P m must be the same point!

$t=4$ The spline is defined by the points P1, P2, . . . , P n with prescribed derivatives in both begin and end point of the curve. The user must give the direction and magnitude of these derivatives by the phrase tang = P k , tang = P l . It is assumed that the vector P k - P1 is the derivative in the initial point P1, and the vector P l - P m in the end point P m . Hence the points P k and P l must have been defined already in the section points. Note that P k is connected with the starting point and P l with the end point.

If TYPE = t is omitted, $t=1$ is assumed.

The division of elements on the curve is defined by the parameter i , the coarseness and (optional) NODD= o .

The parameter α has the same meaning as for SPLINE.

PARAM The input for PARAM must be given in the following way:

```
Ci = PARAM j ( P1, P2, NELM=n [,INIT=t_0] [,END=t_1] [, RATIO=r, FACTOR=f ] )
```

PARAM generates a user defined curve. The user must give the co-ordinates x , y and z as function of a parameter t with the aid of a user written subroutine *FUNCCV* (2.3.1). The parameter t goes from t_0 to t_1 . The initial point is given by P1 and the end point by P2.

In this case as well in the case of the function CPARAM, the length of the elements is created according to the rules defined by the parameters NELM, RATIO and FACTOR (PARAM), or the coarseness in the end points and the parameter NODD (CPARAM). As a consequence, the parameter t does not have to be distributed according to these same rules. Therefore, during the generation of the curves it is necessary for the program to compute the length of the curve and hence the function FUNCCV is called far more times than may be expected beforehand.

In the case of the function PARAM it is also permitted to give the distribution of the t -values over the interval. To that end 5 negative values of RATIO = r are permitted, with the following meaning:

$r=-1$: equidistant distribution of t , compare with $r=0$.

$r=-2$: the last t -value is f times the first one, compare with $r=1$.

$r=-3$: each next t -value is f times the preceding one, compare with $r=2$.

$r=-4$: the last t -value is $1/f$ times the first one, compare with $r=3$.

$r=-5$: each next t -value is $1/f$ times the preceding one, compare with $r=4$.

CPARAM The input for CPARAM must be given in the following way:

```
Ci = CPARAM j ( P1, P2 [,NODD=o [,INIT=t_0] [,END=t_1])
```


When CPARAM is used, a user defined curve is created as and the parameter t , where t goes from t_0 to t_1 . The initial point is given by P1 and the end point by P2. The coarseness as given in the points P1 and P2, is used for the definition of the curves.

TRANSLATE The input for TRANSLATE must be defined in the following way

$$C_i = \text{TRANSLATE } C_j (P_i [, P_j, P_k, \dots])$$

When TRANSLATE is used, the curve C_i is a copy of curve C_j translated over a distance $d = ((P_{1_i} - P_{1_j})_x, (P_{1_i} - P_{1_j})_y, (P_{1_i} - P_{1_j})_z)$ with P_{1_i} the first point on C_i and P_{1_j} the first point on C_j .

If the points P_i, P_j, P_k, \dots are given, these points correspond to the second, third etc. user points on C_j in that sequence. When these user points have co-ordinates (0,0,0), they get the new co-ordinates as computed by the translation, otherwise it will be checked whether these points have the correct co-ordinates, that is if these points are in fact positioned on C_i . The point numbers i of P_i may not exceed the maximal number of user points.

For most applications it is necessary that both the initial and end point of a curve are identified with user points. However, if the curve to be copied consists of many user points, defining the end point of the new curve requires a large number of (possibly unnecessary) user points on this new curve. For that reason the user may identify the last user point at the new curve by preceding the point number by a minus sign. This is for example the case if the curve must be connected to another curve.

So

$$\text{TRANSLATE } C_j (P_1, -P_5)$$

indicates that the begin point on curve C_i is the user point P_1 and the end point is user point P_5 . If more user points are defined on the new curve, then the point with the minus sign must always be the last one in the row.

ROTATE The input for ROTATE must be defined in the following way:

$$C_i = \text{ROTATE } C_j (P_1, P_2, P_3 [, P_4, P_5, \dots])$$

When ROTATE is used, the curve C_i is a copy of curve C_j translated and rotated, such that the first three user points at C_i (P_1, P_2 and P_3) are the copies of the corresponding user points at C_j . For two-dimensional meshes it suffices to give two points only. If a straight line has to be translated and rotated in 3D it also suffices to give two user points on the curve C_i . These user points define the translation as well as the rotation. ROTATE may only be used for curves in a plane. j must be smaller than i .

If the points P_4, P_5, P_6, \dots are given, these points correspond to the fourth, fifth etc. user points on C_j in that sequence. When these user points have co-ordinates (0,0,0), they get the new co-ordinates as computed by the rotation, otherwise it will be checked whether these points have the correct co-ordinates, that is if these points are in fact positioned on C_i . The point numbers i of P_i may not exceed the maximal number of user points.

In the same way as for TRANSLATE the last user point in the case of ROTATE may be identified by a minus sign.

For an example of the use of ROTATE see 2.6.

REFLECT The input for REFLECT must be given in the following way:

$$C_i = \text{REFLECT } C_j (\text{AXIS} = P_1, P_2; P_3, P_4 [, P_5, \dots])$$

When REFLECT is used, curve C_i is a reflection of C_j with respect to the reflection line $P_1 - P_2$. At least two user points P_3 and P_4 have to be given. If the points P_5, P_6, \dots are given, these points correspond to the third, fourth etcetera user points on C_j in that sequence.

When these user points have co-ordinates (0,0,0), they get the new co-ordinates as computed by the reflection, otherwise it is checked whether these points have the correct co-ordinates, that is if these points are indeed positioned on C_i . The point numbers i of P_i may not exceed the maximal number of user points.

In the case that a curve has to be reflected in R^3 instead of R^2 $AXIS = P1, P2$; should be replaced by $RPLANE = P1, P2, P3$; since in the three-dimensional space a reflection plane is required.

In the same way as for TRANSLATE the last user point in the case of REFLECT may be identified by a negative sign.

CURVES The input for CURVES must be defined in the following way:

```
Ci = CURVES ( Ck, Cl, Cm, . . )
```

When CURVES is used, a curve is defined by the subsequent curves Ck, Cl, Cm, \dots . When the sign of the curve number is positive, the positive direction will be used, otherwise (negative sign), the reversed direction of the curve will be used. The curve number C_i must be larger than Ck, Cl , and Cm .

SPCURVE The input for SPCURVE must be defined in the following way:

```
Ci = SPCURVE ( Ck, Cl, Cm, . . )
```

In this case C_i is defined as a spline defined by the points created at the subsequent curves Ck, Cl, Cm, \dots . When the sign of a curve number is positive, the positive direction will be used, otherwise (negative sign), the reversed direction of the curve will be used. The curve number C_i must be larger than Ck, Cl , and Cm .

The number of nodes at the curve C_i is the same as those on the composite curve defined by Ck, Cl, \dots , however, the distribution of the points is only defined by the length of the first and the last element at the original compound curve. After creating the new curve, the old ones are obsolete and can not be used anymore in SEPRAN. Their only task is to define the shape of the curve C_i .

Mark that there are no nodes defined on the curves Ck, Cl, \dots , and that these curve may not be considered as subcurves if C_i .

PROFILE The input for PROFILE must be defined in the following way:

```
Ci = PROFILE j ( P1, P2, NELM=n ,shape=s, INIT=t_0, END=t_1, RATIO=r, FACTOR=f )
```

C_i is defined as a profile defined by the shape parameter s . The number of elements n is defined by $NELM=n$ and the initial and end point by respectively P1 and P2.

The parameters RATIO and FACTOR have their usual meaning and are meant to define the distribution of nodes.

The parameters t_0 and t_1 have exactly the same meaning as in PARAM. The only difference is that both must be between 0 and 1, and that t_1 must always be larger than t_0 . If omitted the default values 0 and 1 are used, which correspond to a complete lower or upper profile.

The following shapes s of the profile are available:

```
upper_naca0012
lower_naca0012
```

upper_naca0012 defines the upper part of a naca0012 profile where the leading edge is positioned in P1 and the trailing edge in P2.

lower_naca0012 defines the lower part of a naca0012 profile where the leading edge is positioned in P1 and the trailing edge in P2.

CPROFILE The input for CPROFILE must be defined in the following way:

```
Ci = CPROFILE j ( P1, P2, shape=s, INIT=t_0, END=t_1, NODD=o)
```

C_i is defined as a profile defined by the shape parameter s . The number of elements n is defined by the coarseness in the end points and the value of o in $\text{NODD} = o$.

The initial and end point are defined by respectively P1 and P2.

The shapes s of the profile are exactly the same as for PROFILE and also the parameters t_0 and t_1 have the same meaning.

CIRCLE The input for CIRCLE must be defined in the following way:

```
Ci = CIRCLE j ( P1, P2, P3, NELM=n [, RATIO=r, FACTOR=f ] )
```

C_i is defined as a complete circle with center P1 and starting point P2.

The center P1 is not coupled to a node, P2 is always a nodal point.

P3 is only used in case of R^3 and may be skipped in R^2 . It is merely used to define the plane in which the circle is lying. Hence P3 must be positioned such that it is in the plane of the circle and not on the line through P1 and P2. There are no nodes connected to P3.

The number of elements n is defined by NELM, and FACTOR and RATIO have their usual meaning.

OWN_CURVE The input for OWN_CURVE must be defined in the following way:

```
Ci = OWN_CURVE j ( P1, P2, NELM=n [, IFUNC=i ] )
```

C_i is defined as a user curve defined between the user points P1 and P2. The user must define the curve himself through the user written subroutine OWN_CURVE (2.3.2).

Points P1 and P2 form the start and end of the curve and it is checked if the user provided coordinates are indeed equal to the ones stored in P1 and P2, unless P1 and P2 have not been filled before.

$\text{NELM} = n$, defines the number of elements along the curve.

$\text{IFUNC} = i$, defines a choice parameter that is passed undisturbed to subroutine OWN_CURVE, and may be used to distinguish between various calls of OWN_CURVE.

2.3.1 Subroutine FUNCCV

Description

Subroutine FUNCCV is used when curves must be generated using the PARAM or CPARAM mechanism. With this subroutine the user may define a curve as function of a parameter t . FUNCCV must be written by the user.

Call

```
CALL FUNCCV ( ICURVE, T, X, Y, Z)
```

Parameters

DOUBLE PRECISION T, X, Y, Z

INTEGER ICURVE

ICURVE Curve number. Subroutine MESH gives ICURVE the sequence number of the curve to be generated.

T Parameter t for the definition of the curve. Program SEPMESH gives t values between t_0 and t_1 .

X,Y,Z the user must give X, Y and Z the values of the co-ordinates as function of the parameter t and the curve number ICURVE.

Input

Program SEPMESH gives ICURVE and T a value

Output

The user must fill the co-ordinates X, Y and Z.

Interface

Subroutine FUNCCV must be programmed as follows:

```
SUBROUTINE FUNCCV ( ICURVE, T, X, Y, Z)
  IMPLICIT NONE
  INTEGER ICURVE
  DOUBLE PRECISION T, X, Y, Z
  .
  .
  .           statements to give x,y and z a value as function
  .           of t and ICURVE
  .
END
```

2.3.2 Subroutine OWN_CURVE

Description

Subroutine OWN_CURVE is used when the user wants to define a curve himself through the use of a own written subroutine. OWN_CURVE must be written by the user.

Heading

```
subroutine own_curve ( ifunc, icurnr, coor, npoints, ndim )
```

Parameters

INTEGER IFUNC, ICURNR, NPOINTS, NDIM

DOUBLE PRECISION COOR(NDIM,NPOINTS)

IFUNC This parameter, which may not be changed by the user, gets the value of IFUNC in the input file. So it is passed undisturbed to this subroutine, and the user may utilize this parameter to distinguish between several calls.

ICURNR Curve number. Subroutine MESH gives ICURNR the sequence number of the curve to be generated.

COOR Array of size $NDIM \times NPOINTS$ that must be filled by the user with the coordinates of all NPOINTS nodes along the curve in the sequence from P1 to P2, with P1 and P2 defined in the input file (2.3). If the user points P1 and P2 have been filled before, the coordinates of P1 and P2 must coincide with the coordinates of the initial and point of the curve stored in COOR.

NPOINTS Number of nodes along the curve. This parameter is computed by SEPMESH from the parameter NELM, and may not be changed by the user.

NDIM Defines the dimension of the space.

Input

Program SEPMESH gives IFUNC, ICURNR, NPOINTS and NDIM a value.

Output

The user must fill array COOR completely with the coordinates of the nodes along the curve.

Interface

Subroutine OWN_CURVE must be programmed as follows:

```
subroutine own_curve ( ifunc, icurnr, coor, npoints, ndim )
integer ifunc, icurnr, npoints, ndim
double precision coor(ndim,*)
.
.
.
.      statements to fill coor as function of ifunc
.      and possibly ICURNR
.
END
```

In order to get a sample subroutine OWN_CURVE in your local directory, use the command

```
sepget own_curve
```

In this sample subroutine one straight line is formed from the point (0,0) to (*width*,0), where *width* is defined in the input file. The value of *width* is placed in the subroutine by use of the function subroutine `getconst` (3.3.12).

2.4 Surface generators

Description

In this section the various surface generators are treated. These surface generators are activated by the command SURFACES in the input for the program SEPMESH. In 2.2 the following types of surface generators are available

GENERAL
TRIANGLE
QUADRILATERAL
RECTANGLE
USER
SURFACES
ORDERED SURFACE
TRANSLATE
COPY_SURFACE
ROTATE
REFLECT
SIMILAR
PIPESURFACE
COONS
ISOPAR
PARSURF
PAVER
SPHERE
FRAMESURF
COPY_SURFACE

These surface generators have the following global function:

GENERAL Creates a grid for a very general region in a plane (triangles and quadrilaterals). Elements try to take the ideal shape, i.e. as close as possible to equilateral triangles and squares. Sudden refinements are not allowed, the generator is relatively expensive. See 2.4.1 for a description of GENERAL.

TRIANGLE Creates a grid for a very general region in a plane (triangles only). The essential difference with GENERAL is that TRIANGLE allows a much larger ratio from coarse to fine elements, and therefore generates fewer elements in regions where coarse and fine elements are used. Furthermore TRIANGLE requires less computing time than GENERAL, especially for coarse grids. The use of TRIANGLE requires a special license. See 2.4.7 for a description of TRIANGLE.

RECTANGLE The submesh generator RECTANGLE, creates sub meshes in a plane that can be mapped onto a "rectangle". RECTANGLE is restricted in the sense that the region to be subdivided must be topological equivalent to a rectangle, i.e. it must have some resemblance with a "curved" rectangle. Furthermore the number of points on opposite sides of this rectangle must be equal. RECTANGLE is much faster than GENERAL, and allows for oblong elements. See 2.4.2 for a description of RECTANGLE.

QUADRILATERAL the submesh generator QUADRILATERAL creates sub meshes in a plane that can be mapped onto a "rectangle". Also this region must be topological equivalent to a rectangle, but unlike RECTANGLE, there are no restrictions with respect to the number of points situated on opposite sides. See 2.4.3 for a description of QUADRILATERAL.

USER a user provided mesh generator called MESHUS is called. See 2.4.6.

TRANSLATE Copy and translate a surface.

COPY_SURFACE Copy a surface (no translation).

ROTATE Copy, translate and rotate a surface.

REFLECT Make a reflection of a surface with respect to a reflection surface.

PIPESURFACE When PIPESURFACE is used, a "cylinder" type surface in R^3 is generated using the curves $C1, C2, C3$ and $C4$ as generating curves. See 2.4.5 for a description of PIPESURFACE.

COONS The submesh generator COONS creates a mesh on a curved surface in 3D space, where the region is topologically equivalent to a rectangle. The surface is mapped onto a rectangle and this rectangle is subdivided into elements using the submesh generator QUADRILATERAL. See 2.4.4 for a description of COONS.

ISOPAR The submesh generator ISOPAR creates a mesh on a curved surface in R^3 , or a surface in R^2 , where the region is topologically equivalent to a triangle. The surface is mapped onto a triangle and this triangle is subdivided into triangular elements only. See Section 2.4.9 for a description of ISOPAR.

PARSURF The submesh generator PARSURF creates a mesh on a curved surface in 3D space, where the region is topologically equivalent to a rectangle. The surface is mapped onto a rectangle and this rectangle is subdivided into elements using the submesh generator QUADRILATERAL. The difference with the submesh generator COONS is that the surface is given by the user in parameter form, whereas COONS defines the surface itself. See 2.4.8 for a description of PARSURF.

PAVER The submesh generator PAVER creates a two-dimensional mesh. This generator is especially meant for elements with large aspect ratios. The boundary may be very irregular. It is supposed that the elements follow the boundary closely, and that the width of the elements orthogonal to the boundary is small compared to the length in the boundary direction. Such meshes are common for boundary layers like in aircraft industry. See 2.4.10 for a description of PAVER.

SPHERE The submesh generator SPHERE creates a mesh on a sphere or a half sphere. All the user has to give is one circle that defines the (half) sphere implicitly. In fact SPHERE is not a separate mesh generator, the use of it merely decreases the amount of input for generating a sphere. Internally the sphere is build by a number of calls to other submesh generators. To that end extra user points, curves and surfaces are defined that are not visible to the user. See 2.4.11 for a description of SPHERE.

SURFACES Create a surface consisting of surfaces, without ordering this surface.

ORDERED SURFACE Create a surface consisting of surfaces in an ordered way.

Extended description of some of the surface generators

SURFACES The input for SURFACES must be defined in the following way:

$$S_i = \text{SURFACES} (S_k, S_l, S_m, \dots)$$

When SURFACES is used, a surface is defined consisting of the surfaces S_k, S_l, S_m, \dots . The surface is unordered and may only be used at those places where a surface created by GENERAL is allowed.

All elements in these surfaces must have the same shape number.

ORDERED SURFACE The input for ORDERED SURFACE must be defined in the following way:

$S_i = \text{ORDERED SURFACE} ((S_{k_1}, S_{k_2}, \dots), (S_{l_1}, S_{l_2}, \dots), (S_{m_1}, S_{m_2}, \dots), \dots)$

When ORDERED SURFACE is used, a surface is defined consisting of the subsurfaces $S_{k_1}, S_{k_2}, \dots, S_{l_1}, S_{l_2}, \dots, S_{m_1}, S_{m_2}, \dots$. In this case the surface is ordered in a fixed way. The subsurfaces must all have been created by calls of submesh generator RECTANGLE or by translation or rotation of such calls, or they must all be created by the generator PIPE SURFACE. The subsurfaces can not be chosen arbitrarily, but must consist of rows of subsurfaces each with the same number of elements in the "column" direction per "row". See Figure 2.4.1 for an example.

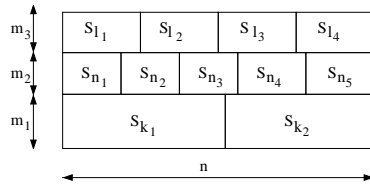


Figure 2.4.1: Example of an ordered surface consisting of subsurfaces

Surfaces corresponding to one row must be placed between brackets, and must be given in the same sequence (i.e. in Figure 2.4.1 from right to left or left to right). The surfaces S_{k_1}, S_{k_2}, \dots must have the same number of elements in the "m"-direction (m_1). The same is true for the subsurfaces S_{l_1}, S_{l_2}, \dots and for the subsurfaces S_{m_1}, S_{m_2}, \dots . The number of elements in the "n"-direction of each row must be constant (n). The statement

$S_i = \text{ORDERED SURFACE} (S_j, S_k, S_l, \dots)$ is treated in the same way as $S_i = \text{ORDERED SURFACE} ((S_j, S_k, S_l, \dots))$, hence using single brackets instead of double ones implies that only one row of surfaces is used.

The subsurfaces must all start in the same relative position, in Figure 2.4.1 left under, and follow the same direction. See Figure 2.4.2. When a subsurface has been generated from right to left, then minus the subsurface number must be used. The upward direction, however, must always be the same.

All elements in these surfaces must have the same shape number.

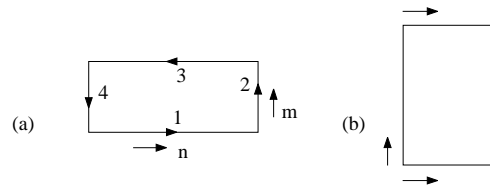


Figure 2.4.2: Standard direction for subsurface. (a) RECTANGLE (b) PIPE SURFACE

TRANSLATE The input for TRANSLATE must be defined in the following way:

$S_i = \text{TRANSLATE } S_j (C1 [C2, C3, \dots])$

When TRANSLATE is used, surface S_i is a copy of surface S_j translated over a distance $\Delta \mathbf{x} = ((P1_i - P1_j)_x, (P1_i - P1_j)_y, (P1_i - P1_j)_z)$ with $P1_i$ the first point on the first curve $C1$ of S_i and $P1_j$ the first point on the first curve $C1$ of S_j . j must be smaller than i .

When the curves $C2, C3, C4, \dots$ are given, these curves correspond to the curves on surface

S_j . The curves must have been created in the section CURVES, for example by the command TRANSLATE or ROTATE. It is checked whether the co-ordinates on C_2, C_3, C_4, \dots are indeed positioned on the surface S_i .

Remark: The curves C_1, C_2, C_3 must completely enclose the surface S_j . It is not permitted to use a half open boundary.

COPY_SURFACE The input for COPY_SURFACE is very simple:

$$S_i = \text{COPY_SURFACE } S_j$$

The result is a surface S_i that is a copy of surface S_j . S_i is completely identical to S_j , so it has the same coordinates. Only the nodal point numbers in the final mesh will be different.

ROTATE The input for ROTATE must be defined in the following way:

$$S_i = \text{ROTATE } S_j (C_1 [C_2, C_3, \dots])$$

When ROTATE is used, surface S_i is a copy of surface S_j translated and rotated such that the curve C_1 on S_i is a copy of the curve C_1 on S_j . j must be smaller than i . ROTATE may only be applied to surfaces lying in a plane. When C_1 forms a straight line, then the user must give at least as many curves such that these curves do specify a plane, i.e. not all points on these curves are positioned on a straight line.

When the curves C_2, C_3, C_4, \dots are given, these curves correspond to the curves on surface S_j . The curves must have been created in the section CURVES, for example by the command TRANSLATE or ROTATE. It is checked whether the co-ordinates on C_2, C_3, C_4, \dots are indeed positioned on the surface S_i .

Remark: The curves C_1, C_2, C_3 must completely enclose the surface S_j . It is not permitted to use a half open boundary.

REFLECT The input for REFLECT must be defined in the following way:

$$S_i = \text{REFLECT } S_j (C_1 [C_2, C_3, \dots])$$

When REFLECT is used, surface S_i is a reflection of surface S_j . The reflection is defined by comparing the nodes in the curves C_1, C_2, \dots, C_n with the corresponding nodes in the curves of S_j . j must be smaller than i . The first node of the boundary of S_j and the first node of the boundary of S_i define the reflection axis or reflection plane. All other boundary points are checked by SEPRAN to be reflections of each other for that same axis or plane. If this is not the case an error message will be given.

Remark: The curves C_1, C_2, C_3 must completely enclose the surface S_j . It is not permitted to use an half open boundary.

SIMILAR The input for SIMILAR must be defined in the following way:

$$S_i = \text{SIMILAR } S_j (C_1 [C_2, C_3, \dots])$$

When SIMILAR is used, the element topology of the nodes in surface S_i is a copy of the topology of the elements in surface S_j (j must be smaller than i). The number of points on the boundary as specified by the curves C_1, C_2, C_3, \dots must be the same as the number of points on the boundary of surface S_i . Contrary to ROTATE the whole boundary has to be specified by the user. SIMILAR may only be used for 3-D problems. Although not required it is recommended to use SIMILAR for surfaces that are lying in a plane.

FRAMESURF Creates a triangular grid in R^3 on a curved surface. The user has to specify the boundary of the surface in the usual way. Besides that it is assumed that the user has a rough description of the surface in terms of triangles. These triangles are used to define points on the surface, but will not be present in the triangulation. So they are used as a frame work for

the surface. The coarseness of the surface mesh is completely defined by the boundary. See Section (2.4.12)

COPY_SURFACE The input for copy surface is very simple:

`Si = copy_surface Sj`

Surface S_i is an exact copy of surface S_j with the same coordinates and topology. The boundary of the surface is not defined explicitly and hence can not be referred to.

2.4.1 Surface generator GENERAL

The surface generator GENERAL is called by the program SEPMESH. The user may activate GENERAL by data records of the type:

$$S_i = \text{GENERAL } j (C_1, C_2, C_3, \dots)$$

with S_i the surface number, j the shape number of the elements created in this surface, and C_1, C_2, \dots the curves enclosing S_i . It is necessary that the curves are created in a direction such that the surface is at the left side of the curves.

GENERAL has the following characteristics:

- A fine division of nodal points on a part of the boundary causes a fine mesh in the neighborhood of this boundary; a coarse division, a coarse mesh. When the user wants to create a local fine mesh inside the surface, he may use extra curves to force GENERAL to create such a mesh. See for example Figure 2.4.1.1. In order to create a fine mesh in the center of S_1 and a coarse one at the boundaries, the user may introduce the inner point P_3 and the inner curve C_3 .

A coarse division of nodal points on the curves C_1, C_2, C_4, C_5, C_6 and a coarse to fine grid on curve C_3 (i.e. fine in the neighborhood of P_3 , coarse in the neighborhood of P_2), will result in the mesh required.

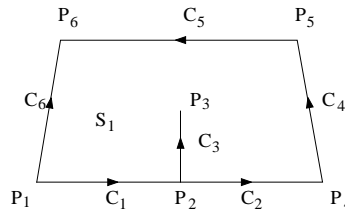


Figure 2.4.1.1: Surface S_1 defined by $S_1 = (C_1, C_3, -C_3, C_2, C_4, C_5, C_6)$

- The mesh generator can not generate elements when a sudden refinement of the nodal points of the boundary is present. Hence when the user wants to create elements on a long small pipe (see Figure 2.4.1.2) GENERAL can not be used, or the user must transform his co-ordinates such that the length/width ratio is not too large. Otherwise the user may use one of the other available mesh generators, like TRIANGLE, RECTANGLE or QUADRILATERAL.

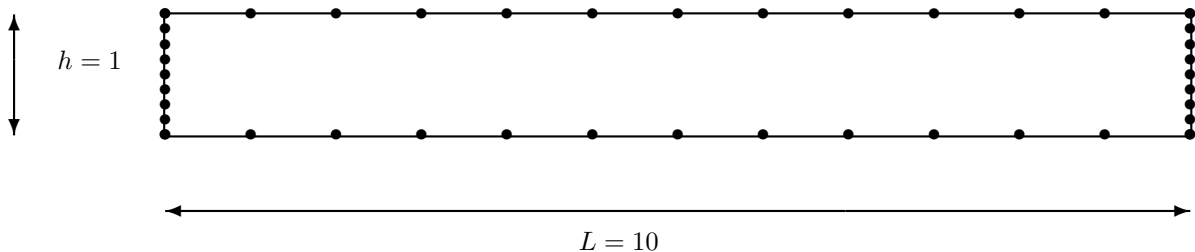


Figure 2.4.1.2: Example of a region that can not be subdivided by GENERAL

- If the boundary is too random (Christmas tree), a subdivision into submeshes may be necessary.

- When quadrilateral elements are generated by GENERAL it is necessary that the number of nodal points on the boundary of the surface is even in the case of linear elements, and the number of vertices of elements on the boundary is even in the case of quadratic elements. The user must take care of this demand.
- Quadrilateral elements require a smoother division of elements than triangular elements. Therefore it may be necessary to define more submeshes when quadrilateral elements are used. The user is advised to consider his plot and decide whether extra submeshes must be created.

Figure 2.4.1.3 shows some meshes that are created by GENERAL.

Remark

In R^3 submesh generator GENERAL may only be applied in a plane.

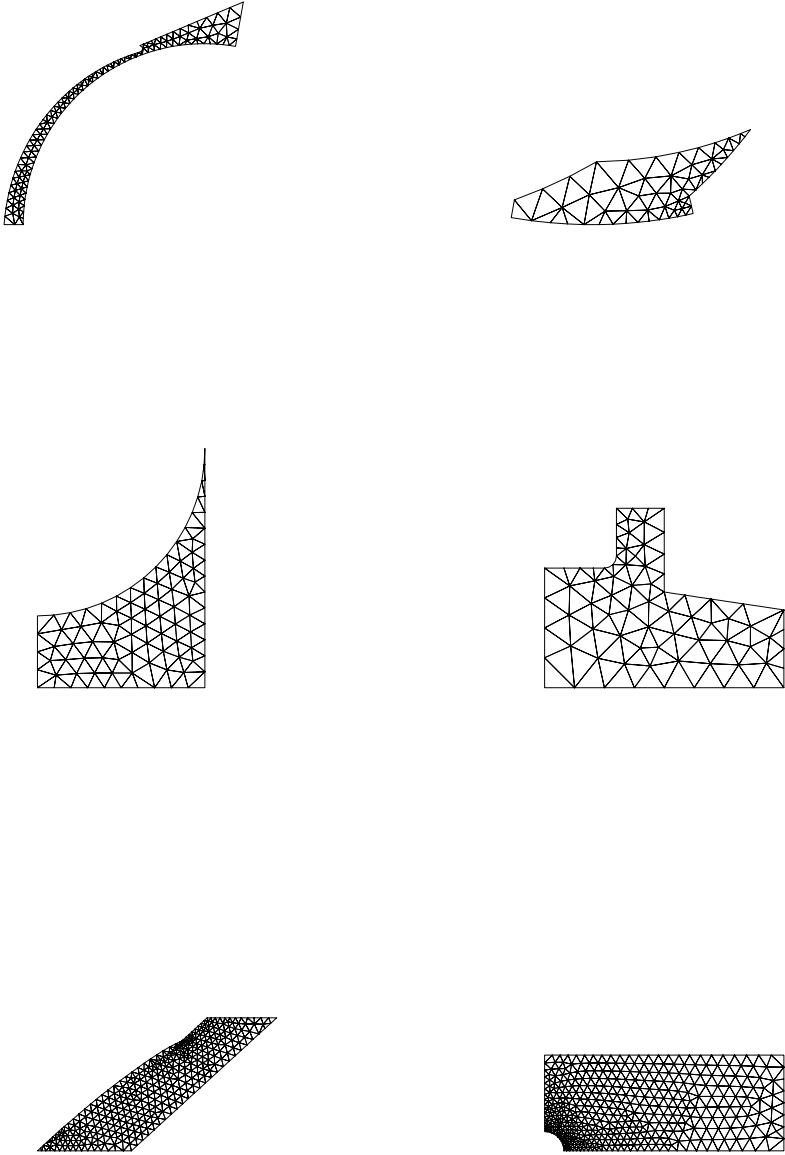


Figure 2.4.1.3: Example of meshes created by GENERAL.

2.4.2 Surface generator RECTANGLE

The surface generator RECTANGLE is called by the program SEPMESH. The user may activate RECTANGLE by data records of the type:

```
Si = RECTANGLE j ( [N = n, M = m,] C1, C2, C3, . . . [, SMOOTH = i]
  [, diagonal_dir = d] )
```

S_i defines the surface number

j defines the shape number of the elements created in this surface

C1, C2, C3,... denote the curves enclosing S_i .

N, M are necessary for the definition of the surface, except when the number of curves is exactly four, see below.

SMOOTH = i With the parameter SMOOTH it is possible to define a kind of smoothing in the grid. First the mesh is generated by a algebraic method and then, if $i > 0$, the mesh is smoothed by a so-called potential smoother. The smoothing is stopped if the relative difference between two steps in the smoothing process is less than 10^{-i} . If SMOOTH is not given $i = 0$ is assumed and no smoothing takes place.

Smoothing may be especially useful if the grid is used for a boundary fitted finite volume or finite difference program.

diagonal_dir = d may be used to prescribe the direction of the diagonals in case of triangular elements. The triangles are generated by subdividing the quadrilaterals into 2 triangles. Possible values:

0, 1 Standard case. If volume elements are generated, all diagonals are pointed in the same direction.

If no volume elements are present the direction of the diagonal is chosen such that the largest angle is subdivided. If all angles are approximately 90 degrees, a standard direction is chosen.

2 Alternating case, the diagonals have alternating directions in subsequent quadrilaterals.

Default value: 0

Characteristics of RECTANGLE:

Generates a submesh that can be mapped onto a rectangular grid. It is therefore necessary to define four "vertices" on the boundary of the submesh. The first vertex coincides with the first nodal point (and hence also the last one) of the boundary of the submesh. The next points are chosen such that the number of elements in the first direction is equal to N and in the second one is equal to M . Hence for linear elements (shape numbers 3 and 5 in Table 2.2.1) the "vertices" are the points:

$1 = (2N + 2M + 1) , N + 1 , N + M + 1 ,$ and $2N + M + 1$

along the boundary, where the points on the boundary are numbered from 1 to $2N + 2M + 1$. For quadratic elements (shape numbers 4 and 6 in Table 2.2.1), these are the points:

$1 = (4N + 4M + 1) , 2N + 1, 2N + 2M + 1,$ and $4N + 2M + 1$.

Therefore the number of nodal points along the boundary must be equal to

$2 (N + M) + 1$ for linear and $4 (N + M) + 1$ for quadratic elements, including coinciding nodal points. The so-called "vertices" do not have to be "physical" vertices, however, the more the submesh to be generated resembles a "rectangle", the better the subdivision will be.

So it is clear that the number of curves defined for RECTANGLE is arbitrary as long as the

number of points is in accordance with the parameters N and M. These parameters subdivide the outer boundary into 4 parts.

A more general version of RECTANGLE, which does not demand equal number of points on opposite points is the mesh generator QUADRILATERAL (2.4.3).

If the number of curves is exactly equal to four and the parameters N and M are not given in the input, it is assumed that N is equal to the number of elements at the first curve and M equal to the number of elements at the second one. Of course it is in that case necessary that the number of points at the third curve is equal to the number of points at the first curve and the number of elements at the fourth curve is also equal to M.

Figure 2.4.2.1 shows the grid in a l-shaped region using the non-smooth grid generated by the following input:

```
mesh2d
  points
    p1=(0,0)
    p2=(1,0)
    p3=(1,3)
    p4=(4,3)
    p5=(4,4)
    p6=(0,4)
  curves
    c1 = line1(p1,p2,nelm=16)
    c2 = line1(p2,p3,nelm=16)
    c3 = line1(p3,p4,nelm=16)
    c4 = line1(p4,p5,nelm=16)
    c5 = line1(p5,p6,nelm=16)
    c6 = line1(p6,p1,nelm=16)
    c7 = curves(c2,c3)
    c8 = curves(c5,c6)
  surfaces
    s1 = rectangle5(c1,c7,c4,c8)
  plot
end
```

In Figure 2.4.2.2 the result is shown once the record with RECTANGLE is replaced by:

```
mesh2d
  points
    p1=(0,0)
    p2=(1,0)
    p3=(1,3)
    p4=(4,3)
    p5=(4,4)
    p6=(0,4)
  curves
    c1 = line1(p1,p2,nelm=16)
    c2 = line1(p2,p3,nelm=16)
    c3 = line1(p3,p4,nelm=16)
    c4 = line1(p4,p5,nelm=16)
    c5 = line1(p5,p6,nelm=16)
```

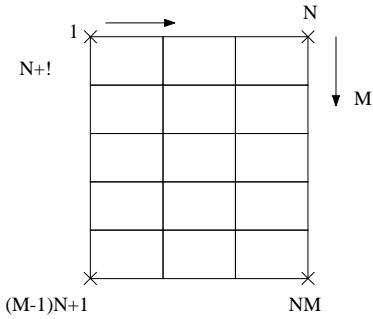


```
c6 = line1(p6,p1,nelm=16)
c7 = curves(c2,c3)
c8 = curves(c5,c6)
surfaces
  s1 = rectangle5(c1,c7,c4,c8,smooth=2)
plot
end
```

Remark

In R^3 submesh generator RECTANGLE may only be applied in a plane.

Examples



Computational grid

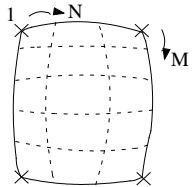
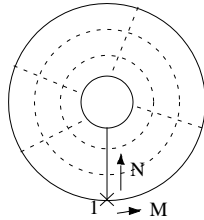
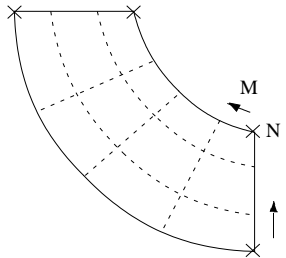
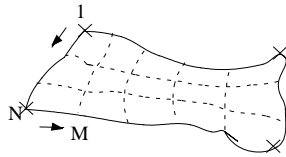
$N = 3, \quad M = 5$

Number of elements along the boundary:

$2(N + M)$

Both triangles and quadrilaterals are permitted.

Actual grids



Coinciding first and last boundary

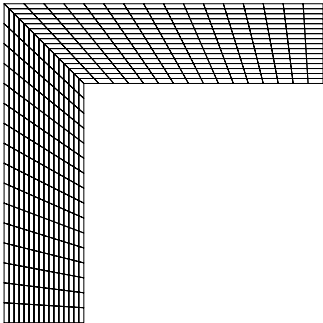


Figure 2.4.2.1: Mesh in L-shaped region without smoothing.

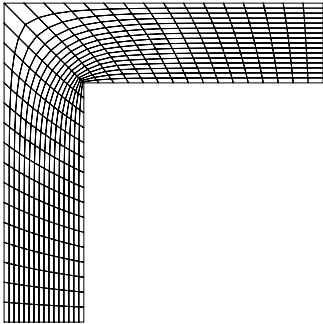


Figure 2.4.2.2: Mesh in L-shaped region with the effect of smoothing.

2.4.3 Surface generator QUADRILATERAL

The surface generator QUADRILATERAL is called by the program SEPMESH. The user may activate QUADRILATERAL by data records of the type:

Si = QUADRILATERAL j (C1, C2, C3, C4 [, BLEND=*b*, CURVATURE=*cu*)

The submesh generator QUADRILATERAL creates a mesh for regions that can be mapped onto a rectangle. Besides that, the region must be topological equivalent to a rectangle. Topological equivalent to a rectangle means that a mapping onto a rectangle must be possible. The sides of the region may be curved, but the curvature may be not so extreme that there is no resemblance with a rectangle. QUADRILATERAL works almost as fast as RECTANGLE while there are no restrictions with respect to the number of points situated on opposite sides. When quadrilaterals are required the number of points on the four curves together has to be even. The user has to take care of this himself. The four curves *C1*, *C2*, *C3*, *C4* must form the four "sides" of the "rectangle". If some of these sides consist of subcurves the user must combine these curves into one curve using the option CURVES of curves.

The parameter BLEND defines the internal mapping of the nodal points. *b* may take the values 0 to 3, where for triangles 2 and 3 are equivalent to 0 and 1 respectively. *b* = 0 corresponds to some local mappings, whereas *b* = 1 corresponds to global mappings. Which of the choices is the best must be found out in practice for each specific region. *b* = 2 or 3 has the same meaning as *b* = 0 or 1, but a easy topology is used for quadrilateral elements. This solution may not be suited for long strips where the number of elements on opposite sides differ too much.

Default value: *b* = 2

The parameter CURVATURE defines the curvature of quadratic and higher order elements. The value of *cu* has the following meaning:

- 0 All element sides are straight, even on the boundary.
 - 1 All element sides are straight, except those on the boundary. These sides may be curved if the boundary is curved.
 - 2 All element sides may be curved in correspondence with the mapping defined by BLEND.
- Default value: *cu* = 1

Figure 2.4.3.1 shows some meshes created by QUADRILATERAL.

Remark

In R^3 submesh generator QUADRILATERAL may only be applied in a plane. For curved surfaces submesh generator COONS may be used.

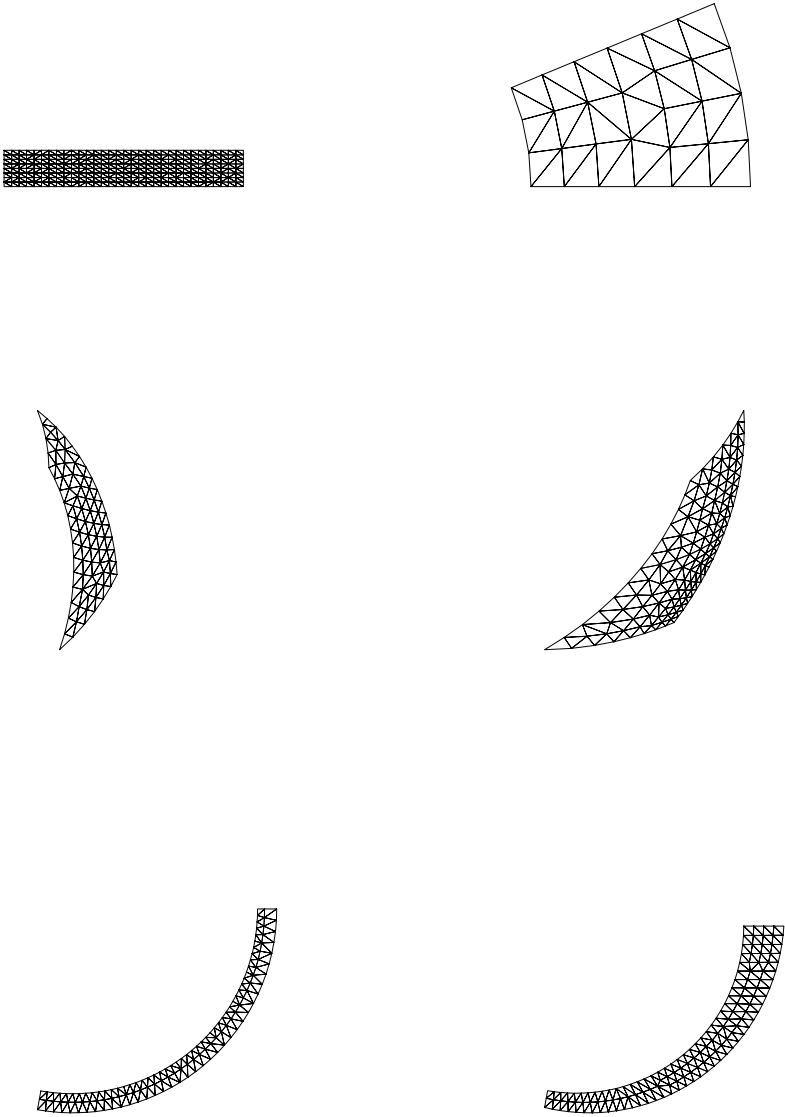


Figure 2.4.3.1: Example of meshes created by QUADRILATERAL.

2.4.4 Surface generator COONS

The surface generator is COONS called by the program SEPMESH. The user may activate COONS by data records of the type:

```
Si = COONS j (C1, C2, C3, C4 [, BLEND=b, CURVATURE=cu, diagonal_dir = d ])
```

S_i defines the surface number

j defines the shape number of the elements created in this surface

C1, C2, C3, C4 define the four curves that enclose the surface.

BLEND = b has exactly the same meaning as in QUADRILATERAL (Section 2.4.3). It is only used if elements in the plane are created by QUADRILATERAL, see below.

CURVATURE = cu has exactly the same meaning as in QUADRILATERAL (Section 2.4.3). It is only used if elements in the plane are created by QUADRILATERAL, see below.

diagonal_dir = d has exactly the same meaning as in RECTANGLE (Section 2.4.2). It is only used if elements in the plane are created by RECTANGLE, see below.

The submesh generator COONS creates a mesh on a curved surface in 3D space which is defined by exactly four generating boundary curves. There are no restrictions with respect to the number of points on opposite sides and it is not required that the four curves are in a plane as is the case for the generators RECTANGLE (Section 2.4.2), GENERAL (Section 2.4.1), TRIANGLE (Section 2.4.7) and QUADRILATERAL (Section 2.4.3). In fact COONS is based on a simple Coon's formula, which maps the curved surface onto a "rectangular" surface in a plane. To create elements in this plane submesh generator QUADRILATERAL is used. A typical example of the use of COONS is given in example 2.4.4.1.

Remark

If the number of nodes at the curves C1 and C3 are equal and the number of nodes at the curves C2 and C4 are equal, submesh generator RECTANGLE is used for the subdivision of the region in the mapped plane. In all other cases QUADRILATERAL is used. In the specific case that RECTANGLE is used, it is possible to apply the volume generator BRICK with curved surface boundaries.

Example 2.4.4.1

In this example a "bottle" in 3-D is created by the curve generators SPLINE and ARC, the surface generator COONS and the volume generator GENERAL. The definition of the curves and surfaces is given in Figure 2.4.4.1.

The input for this mesh is defined by:

```
*bottle.msh
mesh3d
  points
    p1 = ( 0.0, 0.0, 0.0 )
    p2 = ( 1.0, 0.0, 0.0 )
    p3 = ( 1.0, 1.0, 0.0 )
    p4 = ( 0.0, 1.0, 0.0 )
    p5 = ( 0.0, 0.0, 3.0 )
    p6 = ( 1.0, 0.0, 3.0 )
```

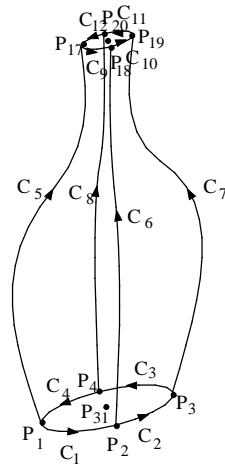


Figure 2.4.4.1: Definition of the points, curves and surfaces for the bottle

```

p7 = ( 1.0, 1.0, 3.0 )
p8 = ( 0.0, 1.0, 3.0 )
p9 = ( 0.2, 0.2, 3.4 )
p10 = ( 0.8, 0.2, 3.4 )
p11 = ( 0.8, 0.8, 3.4 )
p12 = ( 0.2, 0.8, 3.4 )
p13 = ( 0.3, 0.3, 3.8 )
p14 = ( 0.7, 0.3, 3.8 )
p15 = ( 0.7, 0.7, 3.8 )
p16 = ( 0.3, 0.7, 3.8 )
p17 = ( 0.3, 0.3, 5.0 )
p18 = ( 0.7, 0.3, 5.0 )
p19 = ( 0.7, 0.7, 5.0 )
p20 = ( 0.3, 0.7, 5.0 )
p21 = ( 0.5, 0.5, 0.0 )
p22 = ( 0.5, 0.5, 5.0 )

```

curves

```

c1=arc1(p1,p2,p21,nelm=4)
c2=arc1(p2,p3,p21,nelm=4)
c3=arc1(p3,p4,p21,nelm=4)
c4=arc1(p4,p1,p21,nelm=4)
c5=spline1(p1,p5,p9,p13,p17,nelm=19,type=1)
c6=spline1(p2,p6,p10,p14,p18,nelm=19,type=1)
c7=spline1(p3,p7,p11,p15,p19,nelm=19,type=1)
c8=spline1(p4,p8,p12,p16,p20,nelm=19,type=1)
c9=arc1(p17,p18,p22,nelm=4)
c10=arc1(p18,p19,p22,nelm=4)
c11=arc1(p19,p20,p22,nelm=4)
c12=arc1(p20,p17,p22,nelm=4)

```

surfaces

```

s1=coons3(c1,c2,c3,c4)
s2=coons3(c1,c6,-c9,-c5)
s3=coons3(c2,c7,-c10,-c6)
s4=coons3(-c3,c7,c11,-c8)
s5=coons3(-c4,c8,c12,-c5)
s6=coons3(c9,c10,c11,c12)

```

```
s7=surfaces(s1,s2,s3,s4,s5,s6)
volumes
v1=general111(s7)
plot
end
```

Figure 2.4.4.2 shows the mesh created by the program SEPMESH. Since hidden-line removal is applied only a part of the surface is visible.

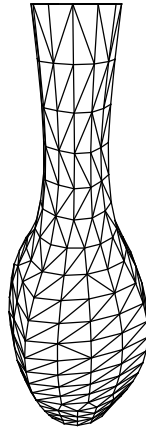


Figure 2.4.4.2: Bottle defined in example 2.4.4.1

2.4.5 Surface generator PIPESURFACE

The surface generator PIPESURFACE is called by program SEPMESH. The user may activate PIPESURFACE by data records of the type:

```
Si = PIPESURFACE j ( C1, C2, C3 [, C4], [diagonal_dir = d], [interpolation = g] )
```

Si defines the surface number

j defines the shape number of the elements created in this surface

C1, C2, C3, C4 denote the three or four generating curves. See Figure 2.4.5.1 for an explanation.

diagonal_dir = d may be used to prescribe the direction of the diagonals in case of triangular elements. The triangles are generated by subdividing the quadrilaterals into 2 triangles.

Possible values:

- 0 Standard case, the diagonals have exactly the same direction as they would have in case of the use of the submesh generator RECTANGLE.
- 1 Reversed case, the diagonals have the opposite direction as they would have in case of the use of the submesh generator RECTANGLE.
- 2 Alternating case, the diagonals have alternating directions in subsequent quadrilaterals.

Default value: 0

interpolation = g may be used to define the type of interpolation that is used to generate nodes in the pipe surface based on the boundary nodes. Note that pipesurface is a algebraic mesh generator which means that the number of points is the same in each line from one side to the opposite side of the boundary. Possible values for *g* are

```
default
hor_straight
vert_straight
```

default means that the standard interpolation is applied. This means that coordinates are computed using a so-called Coon's interpolation. The result is a set of lines that tries to follow the boundary as good as possible. However, the result may be that all the lines in the pipesurface may be curved.

hor_straight prevents this "curved" behavior of the internal grid lines. The grid lines are forced to be straight lines from curve C1 to C2, where the subdivision along these internal lines is based on the subdivision along C3 and C4. Since these lines are straight it is possible that points are computed outside the original pipesurface. This might be the case if C1 and C2 are very curved.

vert_straight is the same as **vert_straight**, but now the straight lines are drawn between C3 and C4.

A pipe surface is generated by three or four curves. At the "bottom" and the "top" exactly one generating curve must be defined, whereas from "bottom" to "top" in general two curves are required. Only when the "top" and "bottom" curves are closed, there is only one curve needed from "bottom" to "top". Figure 2.4.5.1 shows the situation of a closed pipe surface (3 curves), an open pipe surface (4 curves) is given in Figure 2.4.5.2.

The curves C1 and C2 may be open or closed curves. Each of these curves may consist of more than one curve, however, in that case these curves must be combined into one curve by the command $C_i = \text{CURVES} (C_j, C_k, C_l, \dots)$. Furthermore C1 and C2 must be congruent curves, that is C2 may be mapped onto C1 only by a translation and a rotation, so when C1 is closed then C2 is also closed etc.

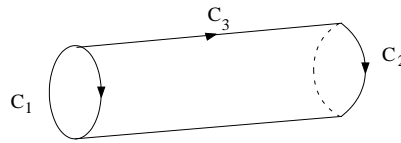


Figure 2.4.5.1: Closed PIPESURFACE with 3 generating curves

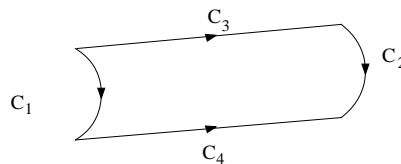


Figure 2.4.5.2: Open PIPESURFACE with 4 generating curves

The generating curves C_3 and C_4 must connect two corresponding points on the curves C_1 and C_2 . All other points on the curves C_1 and C_2 are connected by curves parallel to C_3 and C_4 , thus generating the surface. The distribution of points on C_3 and C_4 defines the position of elements in the C_3 direction.

The nodal points are numbered parallel to C_1 from the first point of C_1 to the last point of C_1 etc. until the last point of C_2 is reached. C_3 starts at the first point of C_1 and ends at the first point of C_2 , C_4 starts at the last point of C_1 and ends at the last point of C_2 .

To show the effect of the interpolation we show the various results of the interpolation of a pipe where the upper part consists of a circle and the lower part of two perpendicular straight lines. In order to get a set of curved lines in the standard case it was necessary to make an asymmetric subdivision of elements along one of the straight lines. Figure 2.4.5.3 shows the standard (default) interpolation and Figure 2.4.5.4 the horizontal straight interpolation with straight lines from top to bottom. To show that you have to be careful with the option straight lines is shown in Figure 2.4.5.5

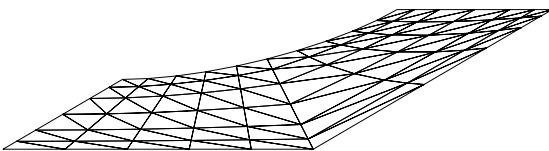


Figure 2.4.5.3: Default interpolation

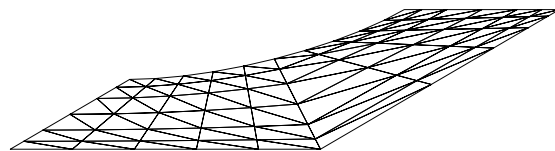


Figure 2.4.5.4: Horizontal straight

where a vertical straight interpolation is used and lines are drawn outside the original region. These three examples can be copied into your local directory by the command `sepgetex pipesurface0x`, with x 1, 2 or 3.

Remark

A pipe surface consisting of subpipe surfaces may contain double subpipe surfaces. See example 2.6.1. In that case it is necessary that the ordered surface starts with the double surface.

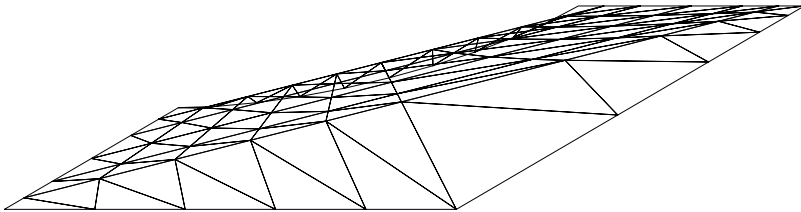


Figure 2.4.5.5: Vertical straight interpolation

2.4.6 Surface generator MESHUS

The surface generator MESHUS is called by the program SEPMESH. The user may activate MESHUS by data records of the type:

$$S_i = \text{USER } j \text{ (NELEM=k, NPOINT=l, C1, C2, C3, ...)}$$

where S_i is the surface number, j the shape number of the elements to be created in this surface, and C_1, C_2, \dots the curves enclosing S_i . The parameters NELEM and NPOINT must be overestimates of the number of elements respectively number of nodal points in the submesh. These estimates are used to define the working space necessary to long as the space needed in IBUFFR is available. Subroutine MESHUS must be written by the user.

Call

```
CALL MESHUS ( ISURF, NEW, COOR, KMESH, INPELM, NBNDPT, KBNDPT, BCORD,
             NPOINT, NELEM )
```

Parameters

INTEGER ISURF, NEW, KMESH(*), INPELM, NBNDPT, KBNDPT(*), NPOINT, NELEM

LOGICAL NEW

DOUBLE PRECISION COOR(*), BCORD(*)

ISURF Surface number i of the data record S_i .

NEW Indication whether it concerns a new mesh (NEW = true) or a mesh of which only the co-ordinates must be changed (NEW = false).

COORD At output the co-ordinates of the nodal points of the submesh must be stored sequentially in COOR from position 1.

KMESH At output the local nodal point numbers of the elements of the submesh must be stored sequentially in KMESH from position 1.

INPELM Number of nodal points in an element. INPELM is computed by subroutine MESH with the aid of the parameter j in the data record USER j .

NBNDPT Number of nodal points in the boundary of the submesh.

KBNDPT At output array KBNDPT of length NBNDPT must be filled with the local boundary nodal points of the submesh sequentially in the sequence as given by C_1, C_2, \dots in the data record $S_i = \text{USER } j \text{ (...)}$.

BCORD At input the co-ordinates of the boundary nodal points are stored in array BCORD in the sequence as is given by C_1, C_2, \dots in the data record $S_i = \text{USER } j$.

NPOINT At input NPOINT contains the overestimated value of NPOINT in the data record. At output NPOINT must have exactly the number of nodal points in the submesh.

NELEM At input NELEM contains the overestimated value of NELEM in the data record. At output NELEM must have exactly the number of elements in the submesh.

Input

Subroutine MESH gives ISURF, NEW, INPELM, NBNDPT, NPOINT and NELEM a value.

Array BCORD has been filled by MESH.

When NEW = false (only the co-ordinates must be changed), the arrays COOR and KMESH have been filled with the co-ordinates and the elements of the preceding mesh.

Output

The user must give NPOINT and NELEM a value when NEW = true.

When NEW = true, the arrays KBNDPT (positions 1, . . . , NBNDPT), COOR (positions 1, . . . , NPOINT×NDIM) and KMESH (positions 1, . . . , INPELM×NELEM) must be filled by the user. When NEW = false only array COOR must be changed by the user.

Interface

Only one submesh may be created in each call of subroutine MESHUS. The nodal points and elements in the mesh have local numbers from 1 to NPOINT respectively 1 to NELEM. Subroutine MESH generates the global numbering. The arrays KBNDPT and BCORD are filled in the sequence as given by the user in the data record.

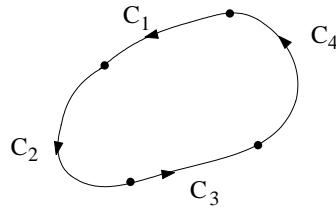


Figure 2.4.6.1: Submesh with boundary

Hence for example for the boundary of the submesh in Figure 2.4.6.1. First all nodal points of C1 then of C2, C3 and finally of C4 are stored. These curves must be subsequent curves. Common nodal points of curves are counted only once.

Array COOR must be filled in the sequence:

$x_1, y_1, z_1, x_2, y_2, z_2, \dots, x_{NPOINT}, y_{NPOINT}, z_{NPOINT}$
 hence x_i must be stored in COOR (NDIM × (i-1) + 1)
 y_i must be stored in COOR (NDIM × (i-1) + 2) etc.

NDIM denotes the dimension of the space.

The easiest way to fill COOR is to define it as a two-dimensional array of size NDIM × NPOINT, with NDIM the actual dimension of the space. Since NPOINT may not be the actual value at input the undefined size should be used for the second component, hence COOR(NDIM,*).

Array KMESH must be filled in the sequence:

$i_1, i_2, i_3, \dots, i_{INPELM}$: nodal points of element 1, followed by:

$j_1, j_2, j_3, \dots, j_{INPELM}$: nodal points of element 2, etc.

Hence the nodal points in element k are stored in KMESH positions INPELM × (k-1) + 1, . . . , INPELM × (k-1) + INPELM.

The easiest way to fill KMESH is to define it as a two-dimensional array of size INPELM × NELEM. Since NELEM may not be the actual value at input the undefined size should be used for the second component, hence KMESH(INPELM,*).

Subroutine MESHUS must be programmed by the user in the following way:

```

SUBROUTINE MESHUS ( ISURF, NEW, COOR, KMESH, INPELM, NBNDPT, KBNDPT,
+                 BCORD, NPOINT, NELEM )
  IMPLICIT NONE
  INTEGER NDIM
  PARAMETER ( NDIM=2 )
  INTEGER ISURF, INPELM, KMESH(inpelm,*), NBNDPT, KBNDPT(*),
+         NPOINT, NELEM
  DOUBLE PRECISION COOR(ndim,*), BCORD(ndim,nbndpt)
  LOGICAL NEW

  IF(NEW) THEN
    statements to fill COOR, KMESH and KBNDPT
    statements to compute NPOINT and NELEM
  ELSE
    statements to change COOR
  END IF

END
```

2.4.7 Surface generator TRIANGLE

The surface generator TRIANGLE is called by program SEPMESH. The user may activate TRIANGLE by data records of the type:

```
Si = TRIANGLE j ( C1, C2, C3, ... [internal_points = p1,...,pm], [internal_curves = c1,...,cm])
```

with S_i the surface number, j the shape of the elements created in this surface, and C_1, C_2, \dots the curves enclosing S_i .

The option `internal_points = p1, ..., pm` forces nodal points to coincide with the user points p_1, \dots, p_m . The local coarseness in these points are used for the coarseness of the mesh locally. So by defining these points the user can create fixed points in the mesh, with a given accuracy, without having to define curves containing these points.

With `internal_curves = c1, ..., cm` the user can define curves inside the domain. The mesh is adapted to these curves, which means that elements may have a common side with an element of these curves, but never intersects these curves. All edges on the internal curves are present in the final mesh, so the size of these edges also defines the local coarseness.

Unlike GENERAL, TRIANGLE may contain inner regions which are not filled with elements (See Figure 2.4.7.1). The boundaries of these inner regions must be closed in itself. This means that the user must first give the boundaries of the outer region consecutively such that a closed boundary arises, and then the boundaries of the inner regions. At this moment no more than 5 inner regions are allowed. Another difference with GENERAL is that double boundaries like boundary C3 in Figure 2.4.4.1 (Section 2.4.1) are not permitted.

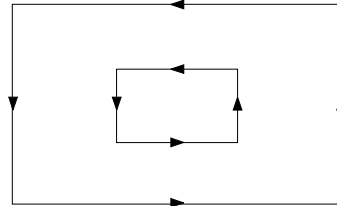


Figure 2.4.7.1: Example of a region with an inner region not to be filled with elements

TRIANGLE has the following characteristics:

- A fine division of nodal points on a part of the boundary causes a fine mesh in the neighborhood of this boundary; a coarse division, a coarse mesh. When the user wants to create a local fine mesh inside the surface, he may use extra curves to force TRIANGLE to create such a mesh.
- There is no essential restriction with respect to the coarseness of nodal points along the boundary. This means that TRIANGLE is able to treat more general meshes than GENERAL.
- A curve in the boundary of TRIANGLE may only be used once.
- TRIANGLE is able to treat at most five "holes" in the region. These "holes" are recognized by the closed boundaries given in the COMMAND `Si = TRIANGLE j (C1, C2, C3, . . .)`. No line should be drawn to connect inner and outer boundaries. See Figure 2.4.7.1.

- At this moment TRIANGLE is limited to 3-node linear triangles. Extensions will be made at request.
- TRIANGLE is not a part of the standard SEPRAN package. A separate license is needed to use TRIANGLE.

Remark

In R^3 submesh generator TRIANGLE may only be applied in a plane.

2.4.8 Surface generator PARSURF

The surface generator is PARSURF called by the program SEPMESH. The user may activate PARSURF by data records of the type:

```
Si = PARSURF j (C1, C2, C3, C4 [, umin=u1, umax=u2, vmin=v1, vmax=v2 ] )
```

The submesh generator PARSURF creates a mesh on a curved surface in 3D space which is defined by exactly four generating boundary curves as well as a parameter representation of the surface. There are no restrictions with respect to the number of points on opposite sides and it is not required that the four curves are in a plane as is the case for the generators RECTANGLE, GENERAL, TRIANGLE and QUADRILATERAL.

In first instance PARSURF defines a grid in the so-called u-v plane, where $umin \leq u \leq umax$ and $vmin \leq v \leq vmax$. The default values for $umin$ and $vmin$ are 0, for $umax$ and $vmax$: 1. The grid in the u-v plane is defined by the distances along the curves $C1$ to $C4$, where $C1$ coincides with $v = vmin$, $C2$ with $u = umax$, $C3$ with $v = vmax$ and $C4$ with $u = umin$. The co-ordinates in the uv-plane are used as input for a user subroutine FUNCSF. It is the task of the user to transform these co-ordinates to the xyz-plane, in this way defining the actual surface.

Subroutine FUNCSF must be written by the user in the following way:

```
subroutine FUNCSF ( ISURF, COOR_UV, COOR_XYZ, NPOINT )
implicit none
integer ISURF, NPOINT
double precision COOR_UV(2,NPOINT), COOR_XYZ(3,NPOINT)

    statements to fill array COOR_XYZ

end
```

In this subroutine ISURF denotes the surface number and is filled by subroutine MESH or program SEPMESH.

NPOINT denotes the number of nodal points in the submesh and is also filled by MESH.

Array COOR_UV contains the u-v co-ordinates of the nodal points in the uv-plane, according to:

$$u(\text{node } i) = \text{COOR_UV}(1,i); v(\text{node } i) = \text{COOR_UV}(2,i)$$

Array COOR_XYZ must be filled by the user for all nodal points. It must contain the xyz co-ordinates corresponding to the uv co-ordinates in the following way:

$$x(\text{node } i) = \text{COOR_XYZ}(1,i); y(\text{node } i) = \text{COOR_XYZ}(2,i); z(\text{node } i) = \text{COOR_XYZ}(3,i)$$

A typical example of the use of PARSURF is given in example [2.4.8.1](#)

Remark

If the number of nodes at the curves $C1$ and $C3$ are equal and the number of nodes at the curves $C2$ and $C4$ are equal, submesh generator RECTANGLE is used for the subdivision of the region in the mapped plane. In all other cases QUADRILATERAL is used. In the specific case that

RECTANGLE is used, it is possible to apply the volume generator BRICK with curved surface boundaries.

Example 2.4.8.1

In this example a sphere surface is created by the surface generator PARSURF. To that end the sphere is subdivided into 6 surfaces, each surrounded by 4 arcs. Each of the 6 surfaces is generated by PARSURF.

Figure 2.4.8.1 shows the definition of the curves on the sphere.

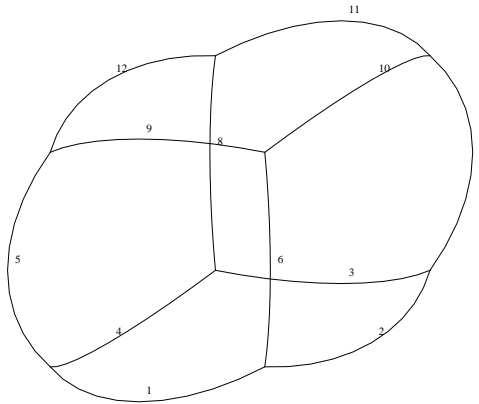


Figure 2.4.8.1: Definition of the curves on the sphere

The input for this mesh is defined by:

```
* sphere_parsurf.msh
mesh3d

points

p1 = ( 0.0, 0.0, 0.0 )
p2 = ( 1.0, 0.0, 0.0 )
p3 = ( 1.0, 1.0, 0.0 )
p4 = ( 0.0, 1.0, 0.0 )
p5 = ( 0.0, 0.0, 1.0 )
p6 = ( 1.0, 0.0, 1.0 )
p7 = ( 1.0, 1.0, 1.0 )
p8 = ( 0.0, 1.0, 1.0 )
p9 = ( 0.5, 0.5, 0.5 )

curves

c1 = arc1 ( p1, p2, p9, nelm = 10)
```

```

c2 = arc1 ( p2, p3, p9, nelm = 10)
c3 = arc1 ( p3, p4, p9, nelm = 10)
c4 = arc1 ( p4, p1, p9, nelm = 10)
c5 = arc1 ( p1, p5, p9, nelm = 10)
c6 = arc1 ( p2, p6, p9, nelm = 10)
c7 = arc1 ( p3, p7, p9, nelm = 10)
c8 = arc1 ( p4, p8, p9, nelm = 10)
c9 = arc1 ( p5, p6, p9, nelm = 10)
c10 = arc1 ( p6, p7, p9, nelm = 10)
c11 = arc1 ( p7, p8, p9, nelm = 10)
c12 = arc1 ( p8, p5, p9, nelm = 10)

```

surfaces

```

s1 = parsurf3 ( c1 , c2 , c3 , c4 , umin=0.0, umax=1.0, vmin=0,vmax=1.0)
s2 = parsurf3 ( c1 , c6 , -c9 , -c5 )
s3 = parsurf3 ( c2 , c7 , -c10, -c6 )
s4 = parsurf3 (-c3 , c7 , c11, -c8 )
s5 = parsurf3 (-c4 , c8 , c12, -c5 )
s6 = parsurf3 ( c9 , c10, c11, c12, umin=0, umax=1. )

```

meshsurf

```

selm1 = (s1,s6)

```

```

plot ( plotfm=15 )

```

end

Since the user subroutine FUNC3F must be written, the standard program SEPMESH may not be used. Below we give an example of the program MAKEMESH that is used to create the triangles on the sphere.

```

program makemesh
call startsepmesh
end

subroutine func3f ( isurf, cooruv, coor3d, npoint )
! =====
!
!
! programmer      Niek Praagman
!
! version 1.0     date    08-10-93 Example for manual
! *****
!
! KEYWORDS
!
! mesh_generation
! 3d
! surfaces
!
! *****
!
!

```

```
! DESCRIPTION
!
! Example of user written routine for parameter-surface :
! Determine 3-D coordinates of generated (u,v)-points in surfaces
!
! *****
!
! INPUT / OUTPUT PARAMETERS
!
! implicit none
! integer isurf, npoint
! double precision cooruv(2,*), coor3d(3,*)
!
!
! cooruv i Cooruv contains the (u,v)-coordinates
!
! coor3d o coor3d contains at output the (x,y,z)-coordinates
!
! isurf i number of surface to be considered
!
! npoint i number of points
!
! *****
!
! SUBROUTINES CALLED
!
! *****
!
! LOCAL PARAMETERS
!
! integer i
! double precision dx, dy, dz, t, u, v, xp2, yp2, zp2
!
! dx step in x-direction
! dy step in y-direction
! i general loop variable
! t line-parameter
! u local coordinate in plane
! v local coordinate in plane
! xp2 x-coordinate local point
! yp2 y-coordinate local point
! zp2 z-coordinate local point
!
! *****
!
! I/O
!
! none
!
! *****
!
! ERROR MESSAGES
!
! *****
!
```

```
! PSEUDO CODE
!
! Check which surface
!
! Read or compute the parameter description of the surface
!
! Determine for each point the coordinates in the surface after
! that the local (u,v) coordinates have been determined
!
! =====
!
! Determine values in surface dependent on surf number :
!
! Determine the coordinates of all points in coor by direct
! parametrization :
!
! do i = 1, npoint
!
!     u = cooruv(1,i)
!     v = cooruv(2,i)
!
!     Check to which surface block this point belongs :
!
!     if ( isurf==1 .or. isurf==6 ) then
!
!         Determine the x,y,z coordinates :
!
!         xp2 = u
!         yp2 = v
!
!         if ( isurf==1 ) then
!
!             zp2 = 0d0
!
!         else
!
!             zp2 = 1d0
!
!         end if ! ( isurf==1 )
!
!     else if ( isurf==2 .or. isurf==4 ) then
!
!         Determine the x,y,z coordinates :
!
!         xp2 = u
!         zp2 = v
!
!         if ( isurf==2 ) then
!
!             yp2 = 0d0
!
!         else
!
!             yp2 = 1d0
```

```
        end if ! ( isurf==2 )

        else if ( isurf==3 .or. isurf==5 ) then

!           Determine the x,y,z coordinates :

            yp2 = u
            zp2 = v

            if ( isurf==3 ) then

                xp2 = 1d0

            else

                xp2 = 0d0

            end if ! ( isurf==3 )

        end if ! ( isurf==1 .or. isurf==6 )

        dx = xp2 - 0.5d0
        dy = yp2 - 0.5d0
        dz = zp2 - 0.5d0

        t = sqrt ( 0.75 / ( dx*dx + dy*dy + dz*dz ) )

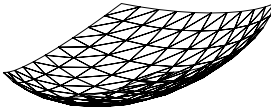
!           Compute 3D coordinates :

            coor3d( 1,i ) = 0.5d0 + t * dx
            coor3d( 2,i ) = 0.5d0 + t * dy
            coor3d( 3,i ) = 0.5d0 + t * dz

        end do ! i = 1, npoint

    end
```

Figure 2.4.8.2 shows the first surface created by PARSURF and Figure 2.4.8.3 the final mesh generated by the program MAKEMESH. Since hidden-line removal is applied only a part of the surface is visible.



SURFACE 1

Figure 2.4.8.2: Surface 1 generated by PARSURF

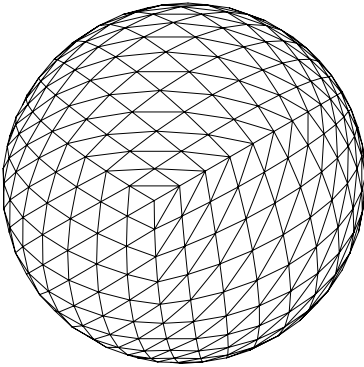


Figure 2.4.8.3: Triangles on sphere generated by MAKEMESH

2.4.9 Surface generator ISOPAR

The surface generator ISOPAR is called by the program SEPMESH. The user may activate ISOPAR by data records of the type:

`Si = ISOPAR j (C1, C2, C3, [mapping = m, centre = pi, subdivision = s]`

Si defines the surface number

j defines the shape number of the elements created in this surface

C1, C2, C3 define the three curves that enclose the surface.

mapping = m means that the coordinates computed are mapped onto a three dimensional surface of special shape. Hence this option is only used in R^3

Possible values for *m* are

default
sphere

default means that no special mapping is applied. The coordinates are the ones computed by ISOPAR.

sphere means that the coordinates are mapped on a sphere with center *Pi*. if no center is given it is checked if there are generating curves that are of the type arc. If so the center of the first arc is used as center of the sphere. The mapping takes place by multiplying the distance between the center and the computed coordinates such that the new points are exactly on the sphere.

centre = Pi is only used in case of a mapping of the type sphere. It defines user point *Pi* as the centre of the sphere.

subdivision = s is a special option that defines the type of subdivision in case of triangles. For quadrilaterals this keyword has no influence.

Possible values for *s* are

default
regular

default means that the default subdivision is applied without special restrictions.

regular means that the subdivision into triangles is made in exactly the same way as for quadrilaterals. In fact it is a subdivision in quadrilaterals, where each quadrilateral is subdivided into 2 triangles.

This means also that we have the same restriction as for quadrilaterals, i.e. the number of elements along each side must be equal and the number of nodes along a side must be odd.

A reason to use regular instead of default is that the number of elements is fixed, which means that if the boundaries are similar also the subdivision into triangles is topologically equivalent.

Due to rounding errors, this does not have to be the case when default is used.

For example if the surface is used to generate a **pipe** or a **channel** this option might be necessary.

The sub mesh generator ISOPAR creates a mesh on a curved surface in R^3 or a surface in R^2 , which is defined by exactly three generating boundary curves, which should not intersect itself. There are no restrictions with respect to the number of points on opposite sides and it is not required that the three curves are in a plane.

However, it is assumed that the surface is topological equivalent to a triangle, which means that it can be mapped onto a triangle. ISOPAR may be considered as the extension of QUADRILATERAL and COONS to regions which resemble a "triangle" rather than a "rectangle".

ISOPAR may be used to create triangles and quadrilaterals. However, in the case of quadrilaterals there are two restrictions:

- the number of elements on each of the three sides must be equal
- the number of elements on each side must be even

If a more general quadrilateral mesh must be created the user is advised to use submesh generator COONS 2.4.4.

Example 2.4.9.1 demonstrates the use of ISOPAR in R^2 and example 2.4.9.2 in R^3 . Mark that in both examples `check_level = 2` is used in order to check the quality of the mesh.

Example 2.4.9.1

In this example a triangle is subdivided into linear triangles. The definition of the curves is given in Figure 2.4.9.1.

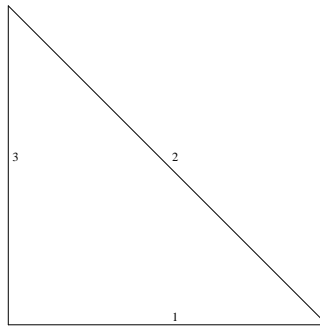


Figure 2.4.9.1: Definition of the curves for the triangle

The input for this mesh is defined by:

* Example of the use of ISOPAR in R2

mesh2d

points

p1 = (.0 , .0)

p2 = (2. , .0)

p3 = (.0 , 2.)

curves

c1 = line1(p1, p2, nelm = 4)

c2 = line1(p2, p3, nelm = 4)

c3 = line1(p3, p1, nelm = 4)

surfaces

s1 = isopar3(c1, c2, c3)

```

plot
check_level = 2

end

```

Figure 2.4.9.2 shows the mesh created by the program SEPMESH.

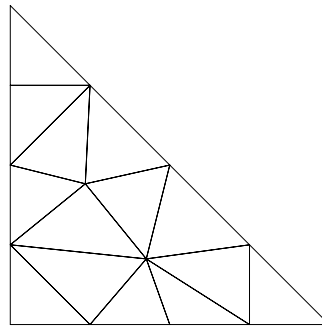


Figure 2.4.9.2: Triangle defined in example 2.4.9.1

Example 2.4.9.2

In this example one-eighth of a sphere is subdivided into tetrahedra. To that end the outer surface is subdivided into triangles using the grid generator ISOPAR and the volume is subdivided into tetrahedra by the volume generator GENERAL. The definition of the curves is given in Figure 2.4.9.3.

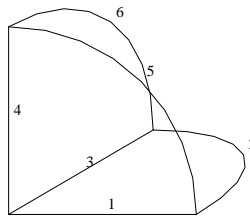


Figure 2.4.9.3: Definition of the curves for the $\frac{1}{8}$ sphere

The input for this mesh is defined by:

```

#
# partsphere.msh
# example of the use of ISOPAR in 3D
# one-eighth of a sphere is subdivided into elements
#
# To get this example into your local directory use:
#

```

```

# sepgetex partsphere
#
# To run this file use:
#   sepmesh partsphere.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    radius = 1      # radius of the sphere
  integers
    nelm = 8        # number of elements along each of the sides
end
#
# Define the mesh
#
mesh3d             # See Users Manual Section 2.2
#
# user points
#
points            # See Users Manual Section 2.2
  p1 = ( 0,        0,        0 ) # centre of the sphere
  p2 = ( radius,   0,        0 ) # corner points on sphere
  p3 = ( 0,        radius,   0 )
  p4 = ( 0,        0,        radius)
#
# curves
#
curves            # See Users Manual Section 2.3
  c1 = line1 ( p1, p2, nelm = nelm ) # line form centre to corner
  c2 = arc1 ( p2, p3, p1, nelm = nelm ) # arc from corner to corner
  c3 = line1 ( p3, p1, nelm = nelm ) # line form centre to corner
  c4 = line1 ( p1, p4, nelm = nelm ) # line form centre to corner
  c5 = arc1 ( p2, p4, p1, nelm = nelm ) # arc from corner to corner
  c6 = arc1 ( p3, p4, p1, nelm = nelm ) # arc from corner to corner
#
# surfaces
#
surfaces          # See Users Manual Section 2.4
  s1 = isopar3 ( c1 , c2 , c3 ) # pie point
  s2 = isopar3 ( c1 , c5 , -c4 ) # pie point
  s3 = isopar3 ( -c3 , c6 , -c4 ) # pie point
  s4 = isopar3 ( c2 , c6 , -c5, mapping = sphere, centre = p1 )
# sphere surface
  s5 = surfaces ( s1, s2, s3, s4 ) # complete surface
#
# volumes
#
volumes           # See Users Manual Section 2.4
  v1 = general11 ( s5 ) # complete region

plot, eyepoint = (-1,-2,3) # make a plot of the mesh
# See Users Manual Section 2.2

```

```
check_level = 2                # check the volumes of each element  
  
end
```

Figure 2.4.9.4 shows the mesh created by the program SEPMESH. Since hidden line removal is applied only the outer part of the sphere is visible.

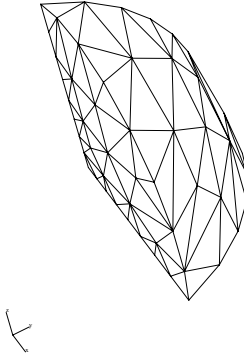


Figure 2.4.9.4: $\frac{1}{8}$ sphere defined in example 2.4.9.2

2.5.3 The user is referred to Section for an example using quadrilaterals.

2.4.10 Surface generator PAVER

The surface generator PAVER is called by the program SEPMESH. The user may activate PAVER by data records of the type:

```
Si = PAVER j ( C1, [t=a, f=b,] C2,[t=a, f=b,], ... )
```

The sub mesh generator PAVER creates a mesh along the curves C1, C2, ...

It is allowed to have a long aspect ratio of the elements, where the width of the elements (perpendicular to the boundary) is much smaller than the length (along the boundary). Such elements currently appear in boundary layers (airplane body-wing configurations), but they may also be used in other applications.

The general idea of the generator is as follows. Start at the closed boundary defined by C1, C2, ... Cn. Create a row of elements of width $t f^{i-1}$ along the present boundary. t defines the thickness of the first layer and may vary per curve. The user may define this thickness himself. i is the sequence number of the layer and f is user defined factor, which may be defined per curve. The region is filled with such layers until it is fully filled. Of course overlapping elements are avoided and a kind of smoothing is applied, in order to get nice elements.

Hence the thickness of the layers is equal to $t, f t, f^2 t, f^3 t \dots$

If $t = 0$ at a part of the boundary this means that no layer is formed along that part, but that the layers grow from other parts towards that boundary part. t must be non-zero for at least one of the curves.

The parameters t and f define the thickness of the layers per curve. If they are not given for a specific curve, the value of the previous curve at the boundary is used. The default values for the first curve are $f = 1$ and $t = 1$.

A negative value of f means that the thickness t will be considered as the mean value of the length of the adjacent elements. For example

```
s1 = paver3 ( c1, t=0.1, f =1 )
```

means that the thickness of the layer is 0.1 everywhere. However,

```
s1 = paver3 ( c1, t=0.5, f =-1 )
```

means that the thickness of the layer in node i is equal to one half of the length of the two line elements adjacent to node i . In case of a uniform grid size of the nodes at C1, this implies that the aspect ratio is equal to 0.5.

The boundary of the region must be created counterclockwise in order that the elements are created on the inner side. If the region contains a hole the boundary of that hole must be created clockwise since then the elements are created on the outer side of the curve.

At this moment only linear triangular elements are implemented in PAVER.

Example 2.4.10.1

As a very simple example of the use of paver we consider a rectangular region where the number of elements at the opposite sides in the length direction is different and where there is reasonable aspect ratio between length and width of the elements.

The input file is defined by

```
# paver01.msh
# Simple example of the use of paver
mesh2d
  points
    p1 = (0d0, 0d0)
```

```

    p2 = (10d0, 0d0)
    p3 = (10d0, 3d0)
    p4 = (0d0, 3d0)
  curves
    c1 = line1(p1,p2,nelm=9)
    c2 = line1(p2,p3,nelm=1)
    c3 = line1(p3,p4,nelm=4)
    c4 = line1(p4,p1,nelm=1)
  surfaces
    s1 = paver3(c1,t=0.1,f=1.0,c2,c3,t=0.2,c4)
  plot
end

```

Figure 2.4.10.1 shows the mesh created by SEPMESH.

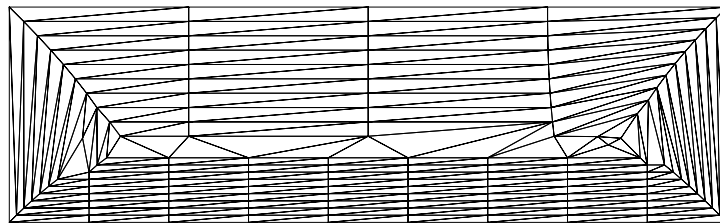


Figure 2.4.10.1: Subdivision of mesh in example 2.4.10.1

Example of a circle 2.4.10.2

The next example is also very simple. It concerns the subdivision of a circle in triangles by paver. In this example we show the effect of the factor f . The input file is defined by

```

# paver07.msh
# Simple example of the use of paver
# In this case a circle is created and a layer of elements inside is made
#
# See Users Manual Section 2.4.10
#
# To run this file use:
#   sepmesh paver07.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4

```

```

reals
  radius = 1      # radius of circle
  thickness = 0.1 # thickness of elements
  factor = 1.3    # factor to be used to decrease or increase element
                  # thickness when far away from curves
                  # factor < 1: decrease thickness
                  # factor > 1: increase thickness

integers
  n = 8           # number of elements along a half circle
  shape_cur = 1   # shape number of elements along curve
  shape_sur = 3   # shape number of elements inside surface
end
#
# Define the mesh
#
mesh2d           # See Users Manual Section 2.2
#
# user points
#
points          # See Users Manual Section 2.2
  p1=(0,0)      # Centroid of circle
  p2=( radius,0) # Utmost right point
  p3=(- radius,0) # Utmost left point
#
# curves
#
curves          # See Users Manual Section 2.3
  c1 = arc shape_cur (p2,p3,p1,nelm= n) # half circle upper part
  c2 = arc shape_cur (p3,p2,p1,nelm= n) # half circle lower part
  c3 = curves ( c1, c2 )                # Complete circle
#
# surfaces
#
surfaces       # See Users Manual Section 2.4
  s1 = paver shape_sur (c3, t = thickness, f = factor )

plot           # make a plot of the mesh
              # See Users Manual Section 2.2
end

```

Figure 2.4.10.2 shows the mesh created by SEPMESH using factor $f = 1.3$ and 2.4.10.3 using factor $f = 0.95$. Mark that making f a little bit smaller results in a huge increase of elements.

Example of a circle with a hole 2.4.10.3

In this example we extend the previous example by taking an extra circle around the previous one and creating elements between those circles only. So the curve in the inner circle must be considered clockwise, which is achieved by providing it with a minus sign in the call to paver.

The input file is defined by

```

# paver08.msh
# Simple example of the use of paver
# In this case a circle with an inner circle is created and a layer of elements
# is made between those circles
#
# See Users Manual Section 2.4.10

```

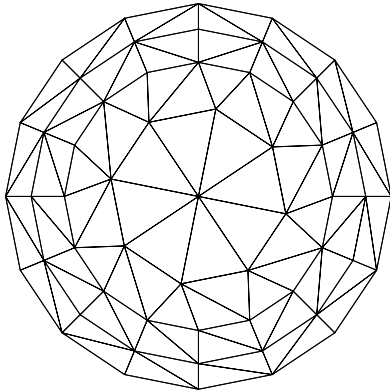


Figure 2.4.10.2: Subdivision of circle (2.4.10.2) with factor = 1.3

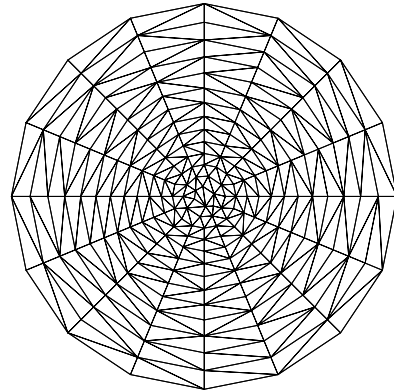


Figure 2.4.10.3: Subdivision of circle (2.4.10.2) with factor = 0.95

```

#
# To run this file use:
#   sepmesh paver08.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    radiusin = 1    # radius of inner circle
    radiusout = 2   # radius of outer circle
    thickness = 0.1 # thickness of elements
    factor = 1.3    # factor to be used to decrease or increase element
                    # thickness when far away from curves
                    # factor < 1: decrease thickness
                    # factor > 1: increase thickness

  integers
    n = 8           # number of elements along a half circle
    shape_cur = 1   # shape number of elements along curve
    shape_sur = 3   # shape number of elements inside surface
end
#
# Define the mesh
#
mesh2d             # See Users Manual Section 2.2
#
# user points
#
points            # See Users Manual Section 2.2
  p1=(0,0)        # Centroid of both circles
  p2=( radiusin,0) # Utmost right point of inner circle
  p3=(-radiusin,0) # Utmost left point of inner circle
  p12=( radiusout,0) # Utmost right point of outer circle
  p13=(-radiusout,0) # Utmost left point of outer circle
#

```



```

# curves
#
curves          # See Users Manual Section 2.3
  c1 = arc shape_cur (p2,p3,p1,nelm= n)   # half inner circle upper part
  c2 = arc shape_cur (p3,p2,p1,nelm= n)   # half inner circle lower part
  c3 = curves ( c1, c2 )                  # Complete inner circle
  c11= arc shape_cur (p12,p13,p1,nelm= n) # half outer circle upper part
  c12= arc shape_cur (p13,p12,p1,nelm= n) # half outer circle lower part
  c13= curves ( c11, c12 )                # Complete outer circle
#
# surfaces
#
surfaces        # See Users Manual Section 2.4
  # outer circle is used in counter clockwise direction
  # inner circle is used in clockwise direction
  s1 = paver shape_sur (c13, t = thickness, f = factor //
                        -c3, t = thickness, f = factor )

plot            # make a plot of the mesh
               # See Users Manual Section 2.2
end

```

Figure 2.4.10.4 shows the mesh created by SEPMESH using factor $f = 1.3$.

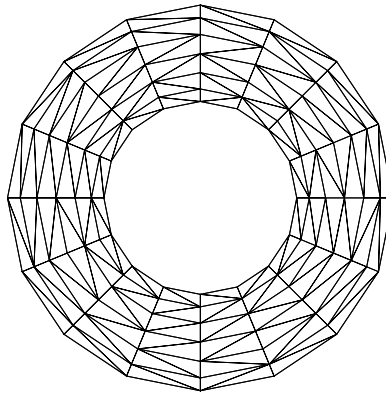


Figure 2.4.10.4: Subdivision of region between two circles (2.4.10.3)

Example 2.4.10.4

A more complex region is given in Figure 2.4.10.5. This figure shows the curves used. The input file is defined by

```

# paver02.msh
# Example of the use of paver at a relatively simple region
mesh2d
  points
    p1 = (0d0, 0d0)
    p2 = (2d0, 0d0)
    p3 = (2d0, -2d0)
    p4 = (1.5d0, -2d0)
    p5 = (2d0, -2.5d0)

```

```

p6 = (2.5d0, -2.5d0)
p7 = (2.5d0, 0.3d0)
p8 = (0d0, 0.3d0)
curves
  c1 = line1(p1,p2,nelm=3)
  c2 = line1(p2,p3,nelm=4)
  c3 = line1(p3,p4,nelm=1)
  c4 = line1(p4,p5,nelm=1)
  c5 = line1(p5,p6,nelm=1)
  c6 = line1(p6,p7,nelm=5)
  c7 = line1(p7,p8,nelm=4)
  c8 = line1(p8,p1,nelm=1)
  c9 = curves(c1,c2,c3,c4,c5,c6,c7,c8)
surfaces
  s1=paver3(c9,t=0.1d0,f=1d0)
plot
end

```

Figure 2.4.10.6 shows the mesh created by SEPMESH.

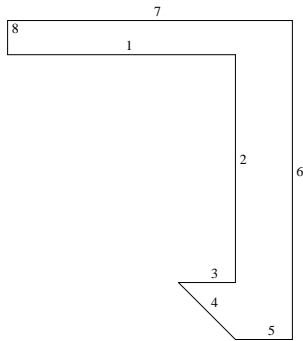


Figure 2.4.10.5: Definition of curves in example 2.4.10.4

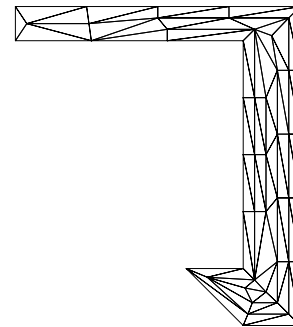


Figure 2.4.10.6: Subdivision of mesh in example 2.4.10.4

Example 2.4.10.5

An extension of the region of example 2.4.10.4 is given in Figure 2.4.10.7. An extra surface has been added in this example
The input file is defined by

```

# paver03.msh
# Example of the use of paver for two surfaces
mesh2d
  points
    p1 = (0d0, 0d0)
    p2 = (2d0, 0d0)
    p3 = (2d0, -2d0)
    p4 = (1.5d0, -2d0)
    p5 = (2d0, -2.5d0)
    p6 = (2.5d0, -2.5d0)

```

```

p7 = (2.5d0, 0.3d0)
p8 = (0d0, 0.3d0)
curves
  c1 = line1(p1,p2,nelm=3)
  c2 = line1(p2,p3,nelm=4)
  c3 = line1(p3,p4,nelm=1)
  c4 = line1(p4,p5,nelm=1)
  c5 = line1(p5,p6,nelm=1)
  c6 = line1(p6,p7,nelm=5)
  c7 = line1(p7,p8,nelm=4)
  c8 = line1(p8,p1,nelm=1)
  c9 = curves(c1,c2,c3,c4,c5,c6,c7,c8)
  c10 = line1(p1,p4,nelm=4)
surfaces
  s1=paver3(c10,t=0.15d0,f=1d0,-c3,t=0.10d0,f=1d0,-c2,-c1)
  s2=paver3(c9,t=0.1d0,f=1d0)
plot
end

```

Figure 2.4.10.8 shows the mesh created by SEPMESH.

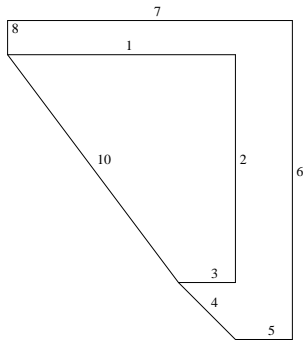


Figure 2.4.10.7: Definition of curves in example 2.4.10.5

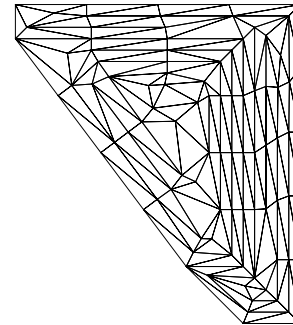


Figure 2.4.10.8: Subdivision of mesh in example 2.4.10.5

Example 2.4.10.6

A complex region corresponding to a more practical problem is given in Figure 2.4.10.9. The input file is defined by

```

* paver04.msh
mesh2d
  coarse (unit=1)
  points
    #> Fider Pattern <#
    p01 = ( 0.2900D+02, 0.0000D+00, 0.1000D+02)
    p02 = ( 0.2900D+02, 0.1200D+02, 0.1000D+02)
    p03 = ( 0.2400D+02, 0.1700D+02, 0.1000D+02)
    p04 = ( 0.2400D+02, 0.1200D+02, 0.1000D+02)
    p05 = ( -0.2400D+02, 0.1700D+02, 0.1000D+02)
    p06 = ( -0.2900D+02, 0.1200D+02, 0.1000D+02)

```

```
p07 = ( -0.2400D+02,  0.1200D+02,  0.1000D+02)
p08 = ( -0.2900D+02, -0.1750D+02,  0.1000D+02)
p09 = ( -0.2400D+02, -0.2250D+02,  0.1000D+02)
p10 = ( -0.2400D+02, -0.1750D+02,  0.1000D+02)
p11 = ( -0.2100D+02, -0.2250D+02,  0.1000D+02)
p12 = ( -0.1600D+02, -0.1750D+02,  0.1000D+02)
p13 = ( -0.2100D+02, -0.1750D+02,  0.1000D+02)
p14 = ( -0.1600D+02,  0.8000D+01,  0.1000D+02)
p15 = (  0.1600D+02,  0.8000D+01,  0.1000D+02)
p16 = (  0.1600D+02, -0.1750D+02,  0.1000D+02)
p17 = (  0.2100D+02, -0.2250D+02,  0.1000D+02)
p18 = (  0.2100D+02, -0.1750D+02,  0.1000D+02)
p19 = (  0.2400D+02, -0.2250D+02,  0.1000D+02)
p20 = (  0.2900D+02, -0.1750D+02,  0.1000D+02)
p21 = (  0.2400D+02, -0.1750D+02,  0.1000D+02)
#> Computed centroids of fider pattern <#
#> Cylinder Pattern <#
p22 = (  0.2275D+02,  0.0000D+00,  0.1000D+02)
p23 = (  0.2275D+02,  0.1400D+02,  0.1000D+02)
p24 = (  0.2216D+02,  0.1541D+02,  0.1000D+02)
p25 = (  0.2075D+02,  0.1400D+02,  0.1000D+02)
p26 = (  0.2075D+02,  0.1600D+02,  0.1000D+02)
p27 = ( -0.2075D+02,  0.1600D+02,  0.1000D+02)
p28 = ( -0.2216D+02,  0.1541D+02,  0.1000D+02)
p29 = ( -0.2075D+02,  0.1400D+02,  0.1000D+02)
p30 = ( -0.2275D+02,  0.1400D+02,  0.1000D+02)
p31 = ( -0.2275D+02, -0.1550D+02,  0.1000D+02)
p32 = ( -0.2225D+02, -0.1600D+02,  0.1000D+02)
p33 = ( -0.2225D+02, -0.1550D+02,  0.1000D+02)
p34 = ( -0.2133D+02, -0.1600D+02,  0.1000D+02)
p35 = ( -0.2075D+02, -0.1500D+02,  0.1000D+02)
p36 = ( -0.2175D+02, -0.1400D+02,  0.1000D+02)
p37 = ( -0.2175D+02,  0.1300D+02,  0.1000D+02)
p38 = ( -0.2116D+02,  0.1441D+02,  0.1000D+02)
p39 = ( -0.1975D+02,  0.1300D+02,  0.1000D+02)
p40 = ( -0.1975D+02,  0.1500D+02,  0.1000D+02)
p41 = (  0.1975D+02,  0.1500D+02,  0.1000D+02)
p42 = (  0.2116D+02,  0.1441D+02,  0.1000D+02)
p43 = (  0.1975D+02,  0.1300D+02,  0.1000D+02)
p44 = (  0.2175D+02,  0.1300D+02,  0.1000D+02)
p45 = (  0.2175D+02, -0.1400D+02,  0.1000D+02)
p46 = (  0.2075D+02, -0.1500D+02,  0.1000D+02)
p47 = (  0.2133D+02, -0.1600D+02,  0.1000D+02)
p48 = (  0.2225D+02, -0.1600D+02,  0.1000D+02)
p49 = (  0.2275D+02, -0.1550D+02,  0.1000D+02)
p50 = (  0.2225D+02, -0.1550D+02,  0.1000D+02)
#> Computed centroids of cylinder pattern <#
```

```
curves
```

```
#> Fider Pattern <#
c01 = cline1( p01,p02 )
c02 = carc1( p02,p03,p04 )
c03 = cline1( p03,p05 )
c04 = carc1( p05,p06,p07 )
c05 = cline1( p06,p08 )
```

```

c06 = carc1( p08,p09,p10 )
c07 = cline1( p09,p11 )
c08 = carc1( p11,p12,p13 )
c09 = cline1( p12,p14 )
c10 = cline1( p14,p15 )
c11 = cline1( p15,p16 )
c12 = carc1( p16,p17,p18 )
c13 = cline1( p17,p19 )
c14 = carc1( p19,p20,p21 )
c15 = cline1( p20,p01 )
c16 = curves( c05,c06,c07,c08,c09,c10 )
c17 = curves( c16,c11,c12,c13,c14,c15 )
#> Cylinder Pattern <#
c18 = cline1( p22,p23 )
c19 = carc1( p23,p24,p25 )
c20 = carc1( p24,p26,p25 )
c21 = cline1( p26,p27 )
c22 = carc1( p27,p28,p29 )
c23 = carc1( p28,p30,p29 )
c24 = cline1( p30,p31 )
c25 = carc1( p31,p32,p33 )
c26 = cline1( p32,p34 )
c27 = cline1( p34,p35 )
c28 = cline1( p35,p36 )
c29 = cline1( p36,p37 )
c30 = carc1( p37,p38,-p39 )
c31 = carc1( p38,p40,-p39 )
c32 = cline1( p40,p41 )
c33 = carc1( p41,p42,-p43 )
c34 = carc1( p42,p44,-p43 )
c35 = cline1( p44,p45 )
c36 = cline1( p45,p46 )
c37 = cline1( p46,p47 )
c38 = cline1( p47,p48 )
c39 = carc1( p48,p49,p50 )
c40 = cline1( p49,p22 )
c41 = curves( c18,c19,c20,c21,c22,c23,c24,c25,c26,c27 )
c42 = curves( c41,c28,c29,c30,c31,c32,c33,c34,c35,c36 )
c43 = curves( c42,c37,c38,c39,c40 )
#> Cylinder pattern - Fider Pattern Connection
c44 = cline1( p01,p22 )
c45 = curves( c17,c44,-c43,-c44 )

surfaces
#> Surface inside Cylinder Pattern <#
s01 = paver3(c43,t=0.2d0,f=1d0)
#> Surface between Cylinder Patt. & Fider Patt. <#
s02 = paver3(c1,t=1,f=1,c2,t=0.4,f=1.1,c3,t=0.2,f=1.2,c4,t=0.4,f=1.1,//
            c17,t=1,f=1,-c43,t=0.2,f=1.7)

plot
end

```

Figure 2.4.10.10 shows the mesh created by SEPMESH.

Example 2.4.10.7

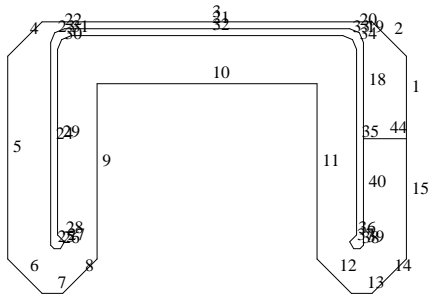


Figure 2.4.10.9: Definition of curves in example 2.4.10.6

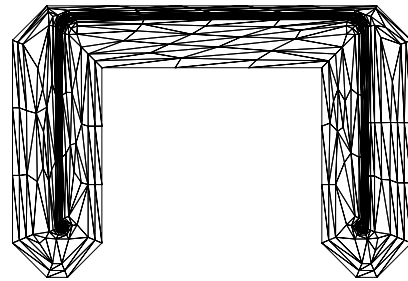


Figure 2.4.10.10: Subdivision of mesh in example 2.4.10.6

This example corresponds also to a more practical problem. The boundary is given in Figure 2.4.10.11.

The input file is defined by

```
*paver05.msh
mesh2d
  points
    p1 = ( -26.025430, -18.081430 )
    p2 = ( -26.519253, -17.199243 )
    p3 = ( -25.969575, -18.047753 )
    p4 = ( 10.554302, 5.612942 )
    p5 = ( 10.829140, 5.188684 )
    p6 = ( 11.268208, 5.439185 )
    p7 = ( 24.615004, -17.954437 )
    p8 = ( 25.402135, -18.140416 )
    p9 = ( 25.448632, -17.943637 )
    p10 = ( 25.604989, -17.815433 )
    p11 = ( 26.973145, -16.693648 )
    p12 = ( 26.702484, -14.945223 )
    p13 = ( 26.973145, -16.693648 )
    p14 = ( 28.616441, -17.349250 )
    p15 = ( 28.991046, -17.498700 )
    p16 = ( 29.210329, -17.837196 )
    p17 = ( 32.252042, -15.866732 )
    p18 = ( 31.427526, -14.593963 )
    p19 = ( 32.744722, -13.842457 )
    p20 = ( 32.141943, -12.787421 )
    p21 = ( 30.824743, -13.538926 )
    p22 = ( 30.284834, -12.121790 )
    p23 = ( 30.356826, -12.310742 )
    p24 = ( 30.181196, -12.410942 )
    p25 = ( 31.332826, -14.427961 )
```

p26 = (31.157202, -14.528163)
p27 = (31.267135, -14.697865)
p28 = (29.907344, -15.578756)
p29 = (29.797408, -15.409055)
p30 = (29.621782, -15.509254)
p31 = (28.205769, -13.027344)
p32 = (27.766702, -13.277847)
p33 = (27.491862, -12.853590)
p34 = (24.990164, -14.474226)
p35 = (24.715325, -14.049971)
p36 = (24.276259, -14.300470)
p37 = (12.514733, 6.314567)
p38 = (12.953801, 6.565067)
p39 = (12.678959, 6.989323)
p40 = (15.276425, 8.671999)
p41 = (15.001587, 9.096254)
p42 = (15.440651, 9.346756)
p43 = (13.764923, 12.283890)
p44 = (13.940551, 12.384091)
p45 = (13.830613, 12.553794)
p46 = (15.190393, 13.434679)
p47 = (15.300329, 13.264976)
p48 = (15.475956, 13.365177)
p49 = (16.869688, 10.922313)
p50 = (17.045314, 11.022514)
p51 = (17.171344, 10.864396)
p52 = (16.226131, 12.050288)
p53 = (17.543328, 12.801793)
p54 = (14.930833, 17.380844)
p55 = (13.613633, 16.629339)
p56 = (12.789116, 17.902110)
p57 = (8.364518, 15.035792)
p58 = (9.189037, 13.763022)
p59 = (7.804570, 13.144132)
p60 = (7.989167, 13.226649)
p61 = (8.099102, 13.056946)
p62 = (13.520614, 16.569123)
p63 = (13.630550, 16.399421)
p64 = (13.806177, 16.499621)
p65 = (14.609058, 15.092369)
p66 = (14.433434, 14.992167)
p67 = (14.543370, 14.822467)
p68 = (12.043718, 13.203119)
p69 = (12.318559, 12.778862)
p70 = (11.879490, 12.528362)
p71 = (13.649922, 9.425232)
p72 = (-24.738383, -15.443259)
p73 = (-25.013223, -15.019004)
p74 = (-25.452289, -15.269504)
p75 = (-26.929417, -12.680467)
p76 = (-27.368484, -12.930969)
p77 = (-27.643322, -12.506711)
p78 = (-30.329368, -14.246771)
p79 = (-30.439304, -14.077070)
p80 = (-30.614930, -14.177269)

```
p81 = ( -31.417811, -12.770021 )
p82 = ( -31.242183, -12.669821 )
p83 = ( -31.352120, -12.500118 )
p84 = ( -29.162505, -11.081652 )
p85 = ( -29.272438, -10.911949 )
p86 = ( -29.121670, -10.777215 )
p87 = ( -30.252438, -11.787730 )
p88 = ( -31.076956, -10.514958 )
p89 = ( -32.252047, -11.276199 )
p90 = ( -31.427527, -12.548968 )
p91 = ( -32.744727, -13.300471 )
p92 = ( -30.574314, -17.104666 )
p93 = ( -30.223059, -16.904268 )
p94 = ( -30.067754, -17.277658 )
p95 = ( -30.223059, -16.904268 )
p96 = ( -29.833296, -16.796453 )
p97 = ( -28.128081, -16.324770 )
p98 = ( -27.822810, -14.582056 )
p99 = ( -28.128081, -16.324770 )
p100 = ( -26.684728, -17.347987 )
p101 = ( -26.272343, -17.640337 )
```

curves

```
c1 = arc1( 1, 3, 2 , nelms = 1)
c2 = line1( 3, 4, nelms = 9 )
c3 = arc1( 4, 6, -5 , nelms = 2)
c4 = line1( 6, 7, nelms = 7 )
c5 = line1( 7, 8, nelms = 2 )
c6 = arc1( 8, 10, 9 , nelms = 1)
c7 = arc1( 10, 12, -11 , nelms = 3)
c8 = arc1( 12, 14, -13 , nelms = 3)
c9 = arc1( 14, 16, 15 , nelms = 2)
c10 = line1( 16, 17, nelms = 3 )
c11 = arc1( 17, 19, 18 , nelms = 2)
c12 = line1( 19, 20, nelms = 2 )
c13 = arc1( 20, 22, 21 , nelms = 2)
c14 = arc1( 22, 24, 23 , nelms = 1)
c15 = line1( 24, 25, nelms = 2 )
c16 = arc1( 25, 27, -26 , nelms = 1)
c17 = line1( 27, 28, nelms = 2 )
c18 = arc1( 28, 30, -29 , nelms = 1)
c19 = line1( 30, 31, nelms = 3 )
c20 = arc1( 31, 33, 32 , nelms = 2)
c21 = line1( 33, 34, nelms = 3 )
c22 = arc1( 34, 36, -35 , nelms = 2)
c23 = line1( 36, 37, nelms = 7 )
c24 = arc1( 37, 39, -38 , nelms = 2)
c25 = line1( 39, 40, nelms = 3 )
c26 = arc1( 40, 42, 41 , nelms = 2)
c27 = line1( 42, 43, nelms = 3 )
c28 = arc1( 43, 45, -44 , nelms = 1)
c29 = line1( 45, 46, nelms = 2 )
c30 = arc1( 46, 48, -47 , nelms = 1)
c31 = line1( 48, 49, nelms = 3 )
c32 = arc1( 49, 51, 50 , nelms = 1)
c33 = arc1( 51, 53, 52 , nelms = 2)
```



```

c34 = line1( 53, 54, nelm = 3 )
c35 = arc1( 54, 56, 55 , nelm = 2)
c36 = line1( 56, 57, nelm = 3 )
c37 = arc1( 57, 59, 58 , nelm = 2)
c38 = arc1( 59, 61, 60 , nelm = 1)
c39 = line1( 61, 62, nelm = 4 )
c40 = arc1( 62, 64, -63 , nelm = 1)
c41 = line1( 64, 65, nelm = 2 )
c42 = arc1( 65, 67, -66 , nelm = 1)
c43 = line1( 67, 68, nelm = 3 )
c44 = arc1( 68, 70, 69 , nelm = 2)
c45 = line1( 70, 71, nelm = 3 )
c46 = line1( 71, 72, nelm = 9 )
c47 = arc1( 72, 74, -73 , nelm = 2)
c48 = line1( 74, 75, nelm = 3 )
c49 = arc1( 75, 77, 76 , nelm = 2)
c50 = line1( 77, 78, nelm = 3 )
c51 = arc1( 78, 80, -79 , nelm = 1)
c52 = line1( 80, 81, nelm = 2 )
c53 = arc1( 81, 83, -82 , nelm = 1)
c54 = line1( 83, 84, nelm = 2 )
c55 = arc1( 84, 86, 85 , nelm = 1)
c56 = arc1( 86, 88, 87 , nelm = 2)
c57 = line1( 88, 89, nelm = 2 )
c58 = arc1( 89, 91, 90 , nelm = 2)
c59 = line1( 91, 92, nelm = 3 )
c60 = arc1( 92, 94, 93 , nelm = 1)
c61 = arc1( 94, 96, 95 , nelm = 1)
c62 = arc1( 96, 98, -97 , nelm = 3)
c63 = arc1( 98, 100, -99 , nelm = 3)
c64 = arc1( 100, 1, 101 , nelm = 2)
surfaces
s1=paver3(c1, t=0.2d0, c2, c3, c4, c5, c6, c7, c8, c9, c10 //
          c11, c12, c13, c14, c15, c16, c17, c18, c19, c20 //
          c21, c22, c23, c24, c25, c26, c27, c28, c29, c30 //
          c31, c32, c33, c34, c35, c36, c37, c38, c39, c40 //
          c41, c42, c43, c44, c45, c46, c47, c48, c49, c50 //
          c51, c52, c53, c54, c55, c56, c57, c58, c59, c60 //
          c61, c62, c63, c64 )
plot(nodes=1,curve=1)
end

```

Figure 2.4.10.12 shows the mesh created by SEPMESH.

Example 2.4.10.8

Next we consider a three-dimensional example of the use of paver. The boundary is given in Figure 2.4.10.13.

The input file is defined by

```

*paver06.msh
mesh3d
points
p1 = (0d0, 0d0, 0d0)
p2 = (2d0, 2d0, 0d0)
p3 = (-2d0, 2d0, 0d0)

```

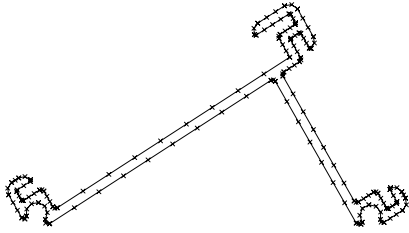


Figure 2.4.10.11: Definition of curves in example 2.4.10.7

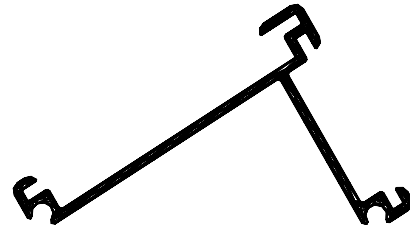


Figure 2.4.10.12: Subdivision of mesh in example 2.4.10.7

```

p4 = (-2d0, -2d0, 0d0)
p5 = (2d0, -2d0, 0d0)
p6 = (1d0, 0d0, 0d0)
p7 = (0d0, 1d0, 0d0)
p8 = (-1d0, 0d0, 0d0)
p9 = (0d0, -1d0, 0d0)
p10 = (2d0, 2d0, 2d0)
curves
  c1 = line1(p2,p3,nelm=4)
  c2 = line1(p3,p4,nelm=4)
  c3 = line1(p4,p5,nelm=4)
  c4 = line1(p5,p2,nelm=4)
  c5 = arc1(p6,p7,p1,nelm=4)
  c6 = arc1(p7,p8,p1,nelm=4)
  c7 = arc1(p8,p9,p1,nelm=4)
  c8 = arc1(p9,p6,p1,nelm=4)
  c9 = curves(c1,c2,c3,c4)
  c10 = curves(c5,c6,c7,c8)
  c11 = translate c9(p10)
  c12 = line1(p2,p10,nelm=3)
surfaces
  s1 = paver3(c9,t=0.0d0,f=1d0,-c10,t=0.40)
  s2 = paver3(c10,t=0.2d0,f=1d0)
  s3 = surfaces(s1,s2)
  s4 = translate s3(c11)
  s5 = pipesurface3(c9,c11,c12)
volumes
  v1=pipe11(s3,s4,s5)
plot(curve=1,nodes=1,eyepoint(1000,1000,700))
end

```

Figures 2.4.10.14 to 2.4.10.16 show the surfaces 1, 2 and 4 generated by SEPMESH. Figure 2.4.10.17 shows the created mesh.

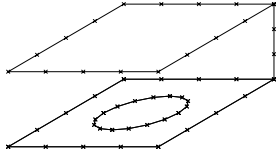


Figure 2.4.10.13: Definition of curves in example 2.4.10.8



Figure 2.4.10.14: Surface 1 in example 2.4.10.8



Figure 2.4.10.15: Surface 2 in example 2.4.10.8



Figure 2.4.10.16: Surface 4 in example 2.4.10.8

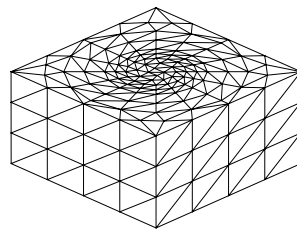


Figure 2.4.10.17: Subdivision of mesh in example 2.4.10.8

2.4.11 Surface generator SPHERE

The surface generator SPHERE is called by the program SEPMESH. The user may activate SPHERE by data records of the type:

```
Si = SPHERE j ( Ck, [,CENTRE=Pi] [,type=t] [,subsurfaces = (S1,S2)] )
```

The sub mesh generator SPHERE creates a mesh along the surface of a sphere or a half sphere. This surface generator itself is not a new surface generator, it is merely meant to reduce the amount of input. Internally extra user points, curves and surfaces are defined so that actual other surface generators are created.

The only curve that is needed is a complete circle on the sphere

Si defines the surface number.

j defines the shape of the elements created on the surface. At this moment only triangular elements are allowed.

Ck defines the generating curve for the sphere. This curve may be either of the type CIRCLE or a combination of arcs, clustered by curves of curves. In fact the user may decide himself how to create C1 as long as it is a complete circle on the surface of the sphere. All points on this circle will be nodal points on the sphere, and the distribution of the nodes on the circle defines the distribution of elements on the sphere.

CENTRE=Pi defines the user point that is used as center of the sphere. If the curve **Ck** is of the type CIRCLE or created by a combination of ARCs it is not necessary to define the center. In that case the center of the circle or the arcs is used.

type=t defines the type of sphere to be created.

Possible values are

```
SPHERE
UPPER_HALF_SPHERE
LOWER_HALF_SPHERE
```

sphere means that a complete sphere is created

upper_half_sphere means that a half sphere is created. The plane through the circle is used as part of the surface as well as the part of the sphere positioned above the circle. What is above or below is defined by the direction in which the points on the circle are defined. The left-handed screw rule is used to define the part that is above the circle.

lower_half_sphere has exactly the same meaning as upper_half_sphere, but of course now the part below the circle is used.

Default value: SPHERE

subsurfaces=(Sk,Sl) connects a subsurface number to two parts of the surface, depending on the type of surface to be created.

If type = **sphere**, the first surface corresponds to the upper half sphere and the second one to the lower half sphere.

If type = **upper_half_sphere** or **lower_half_sphere**, the first surface corresponds to the sphere and the second surface number to the plane through the circle.

If no subsurfaces are given, internally these subsurfaces are created by sequence numbers not known to the user. If, however, the subsurfaces are given, then the sequence numbers must be unique. They can not be used to create another surface. These surface numbers may be used throughout the rest of the input, for example to prescribe boundary conditions or to create a volume.

Example 2.4.11.0

As a very simple example of the use of SPHERE we create a sphere with radius 1 and center (0,0,0). For a more complex example the user is referred to Section 2.6, example 2.6.4. The input file is defined by

```
#
# sphere.msh
# example of the use of the submesh generator sphere
# The outer surface of a sphere is subdivided into triangles
#
# To get this example into your local directory use:
#
# sepgetex sphere
#
# To run this file use:
#   sepmesh sphere.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    radius = 1      # radius of the sphere
  integers
    nelm = 16       # number of elements along the circle
end
#
# Define the mesh
#
mesh3d              # See Users Manual Section 2.2
#
# user points
#
points              # See Users Manual Section 2.2
  p1 = ( 0,        0,        0 ) # centre of the sphere
  p2 = ( radius,   0,        0 ) # point on sphere
  p3 = ( 0,        radius,   0 ) # Extra point needed to define the
                                # plane through the sphere
#
# curves
#
curves              # See Users Manual Section 2.3
  c1 = circle1 ( p1, p2, p3, nelm = nelm ) # Circle with centre p1
                                           # First point is p2
                                           # p3 is needed to define the
                                           # plane
#
# surfaces
#
surfaces            # See Users Manual Section 2.4
  s1 = sphere 3 ( c1 )                # sphere defined by the circle

plot, eyepoint(-1,-2,3)                # make a plot of the mesh
                                         # in order to make a 3d-hiddenline plot
```

```
# of the final mesh, eyepoint must be  
# given  
# See Users Manual Section 2.2
```

```
end
```

Figure 2.4.11.1 shows the mesh created by SEPMESH.

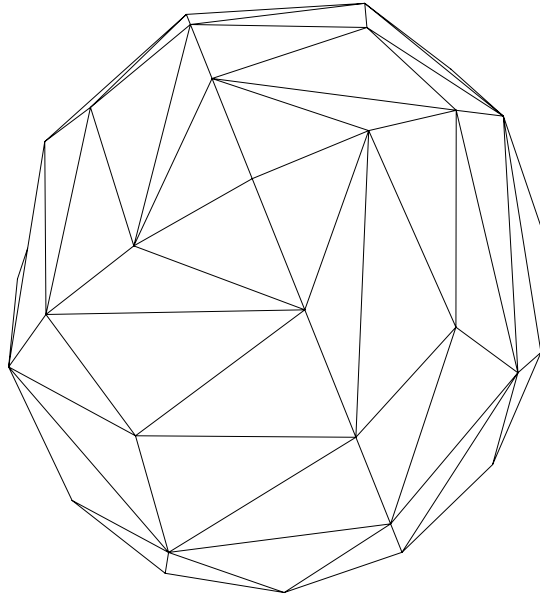


Figure 2.4.11.1: Subdivision of mesh in example 2.4.11

2.4.12 Surface generator FRAMESURF

The surface generator FRAMESURF is called by the program SEPMESH. The user may activate FRAMESURF by data records of the type:

```
Si = FRAMESURF j ( C1, C2, C3, ... )
```

The special purpose sub mesh generator generates a triangular mesh in R^3 , with the boundary given between the brackets. The coarseness of the surface mesh is completely defined by the surrounding curves. The set of curves C1, C2, must form a closed curve. Holes in the boundary are not allowed.

The user must give a rough triangulation of the surface himself. This triangulation is only used to define the curvature of the surface. So these triangles may be ill-shaped, with very small or large angles. The number of triangles that is connected to a point has no influence to the coarseness. In fact the triangulation is made on basis of the boundary and afterwards points are mapped onto the triangle that is closest to that point.

The triangulation must be provided by the user in a file with name `framesurf_x`, where `x` is the sequence number i of the surface. No spaces or leading zeros are allowed in `x`.

The file must have the following contents:

- line 1: `npoints, ntriangles`
i.e. number of points and number of triangles
- lines 2 to 1+`npoints`, for each line:
`inode, xinode, yinode, zinode`
with `inode` the sequence number of the point (between 1 and `npoints`), and `xinode, yinode, zinode` the coordinates of this point.
- lines 2+`npoints` to 1+`npoints`+`ntriangles`, for each line:
`node1, node2, node3`
where the nodes refer to the points given previously. So these lines define the triangles.

2.5 Volume generators

Description

In this section the various volume generators are treated. These volume generators are activated by the command VOLUMES in the input for the program SEPMESH. In 2.2 the following types of volume generators have been defined.

BRICK
 USER
 PIPE
 CHANNEL
 TRANSLATE
 ROTATE
 REFLECT
 GENERAL

These volume generators have the following global function:

BRICK the submesh generator BRICK, creating a mesh that can be mapped onto a "BRICK" is called. BRICK may be considered as the generalization of RECTANGLE. See 2.5.1.

USER A user provided mesh generator called MESHUS is called. See 2.4.6.

PIPE the submesh generator PIPE, creating a mesh in a pipe is called, see 2.5.2. In fact PIPE may be considered as an extension of BRICK.

CHANNEL the submesh generator CHANNEL, creating a mesh between two topologically equivalent faces is called. In fact this generator is completely the same as PIPE. The only difference is that a different interpolation is used to connect points from "upper" surface to "lower" surface. In fact these surfaces may be extremely curved, as long as it is possible to connect each point in both surfaces by a straight line. Such an approach may be used for example for a small channel. See 2.5.3.

TRANSLATE, ROTATE and REFLECT The volume generators TRANSLATE, ROTATE and REFLECT are the three-dimensional extensions of the corresponding surface generators with the same name. These generators expect exactly one surface surrounding the volume to be subdivided, see the input for GENERAL3D. The user activates these generators by data records of the type :

$$Vi = \text{Translate } Vj (Sk)$$

$$Vi = \text{Rotate } Vj (Sk)$$

$$Vi = \text{Reflect } Vj (Sk)$$

The new volume Vi is a translation (rotation, reflection) of Vj . Also the surface that is used, in the example Sk , should be a translation (rotation, reflection) of the surface used for the generation of Vj . The main advantage of Translate, Rotate and Reflect is that they are very quick compared to the time-consuming generator GENERAL3D.

At this moment Vj must have been created by GENERAL3D, volumes created by BRICK or PIPE may not be translated, rotated or reflected by TRANSLATE, ROTATE and REFLECT.

GENERAL the submesh generator GENERAL, creating a general mesh in a three-dimensional region is called. GENERAL forms the 3D extension of the surface generator TRIANGLE. GENERAL may only be applied if your institute has the corresponding license. At this moment GENERAL may only be used to create tetrahedral elements.

GENERAL expects exactly one surface surrounding the region to be subdivided. Such a

surface may always be created by the option SURFACES of SURFACES. This surface of course must be subdivided into triangles. Due to its generality GENERAL is more time-consuming than the regular surface generators PIPE and BRICK. It is cheaper to use GENERAL on several small regions, than once on the connected region. See [2.5.4](#)

2.5.1 Volume generator BRICK

The volume generator BRICK is called by program SEPMESH. The user may activate BRICK by data records of the type:

```
Vi = BRICK j ( [N = n, M = m, L = l,] S1, S2, . . . , S6 [,orientation = o] )
```

with V_i the volume number, j the shape number of the elements created in this volume, and S_1, S_2, \dots the surfaces enclosing V_i . The parameters N, M and L are superfluous, but are kept in order that old input files may be used. The parameter *orientation* may be used in case the surfaces have another orientation than the standard one.

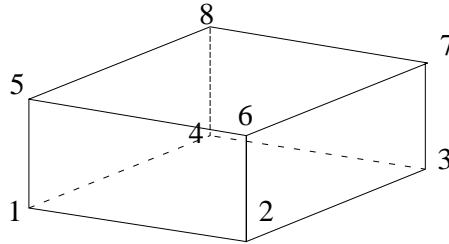


Figure 2.5.1.1: Rectangular region with 6 sides

Characteristics of BRICK:

Generates a sub mesh that can be mapped onto a rectangular (three-dimensional) grid. This rectangular region is plotted in Figure 2.5.1.1.

The region is defined by the 6 surfaces. These surfaces must each be triangulated by the surface generator RECTANGLE (2.4.2). The surfaces must be given in the sequence S_1, S_2, \dots, S_6 , where

S_1 is defined by the face 1,2,3,4,

S_2 is defined by the face 1,2,6,5,

S_3 is defined by the face 2,3,7,6,

S_4 is defined by the face 4,3,7,8,

S_5 is defined by the face 1,4,8,5,

S_6 is defined by the face 5,6,7,8.

The numbers refer to the points in Figure 2.5.1.1. These surfaces must each be generated in the way as indicated above, hence S_1 must start with point 1, then point 2, point 3 and point 4, etc.

The parameters N, M and L correspond to the number of elements to be created along the surfaces in the following way:

Curves (1,2), (5,6), (4,3), (8,7): n elements.

Curves (1,4), (2,3), (5,8), (6,7): m elements.

Curves (1,5), (2,6), (3,7), (4,8): l elements.

The surfaces may be curved and do not have to be part of a plane. The number of elements to be created is equal to αnml where $\alpha=1$ for hexahedral elements and $\alpha=6$ for tetrahedral elements.

The parameter j indicates the type of elements to be generated, see Table 2.2.1.

If the parameters N , M and L are not given in the input, it is assumed that N is equal to the number of elements at the curve (1,2), M equal to the number of elements at curve (1,4) and L equal to the number of elements at curve (1,5).

The orientation of the surfaces for the volume generator BRICK is fixed, which means that the nodes and elements along the surfaces must be generated in the standard sequence. However, in some problems (for example if several "BRICKS" are coupled), it is impossible to define all surfaces in the required sequence. For that reason the parameter orientation has been submitted. The standard orientation of the surfaces is as described before. However, 7 alternative orientations are allowed as shown in Figure 2.5.1.2. Orientation 1 refers to the standard orientation. The parameter

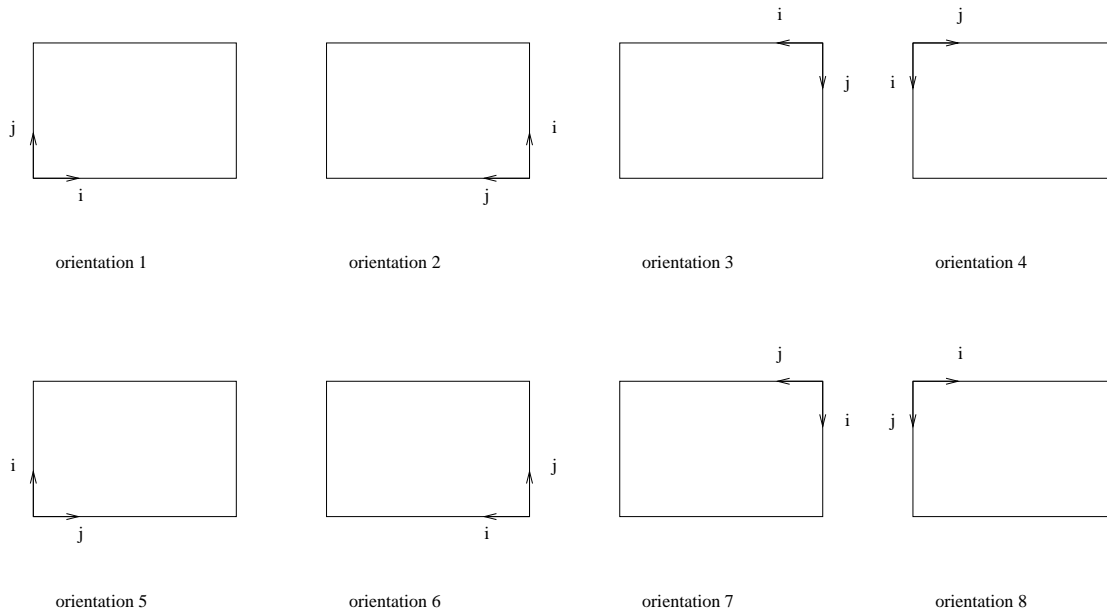


Figure 2.5.1.2: Possible orientations of the surfaces

o in orientation = o , must consists of 6 digits, each digit in the range 1 to 8. Each digit refers to one of the surfaces in the natural sequence, where the most left digit refers to the first surface, and the most right digit to the sixth surface. The value of the digit gives the value of the orientation as defined in Figure 2.5.1.2. If the orientation field is omitted, the default orientation $o = 111111$ is assumed.

Different orientations are always possible with quadrilaterals, but in case of triangles it is necessary that the orientation of the "diagonals" of corresponding quadrilaterals at opposite faces, are in the same direction. In the case that only one brick is defined the volume generator BRICK automatically takes care of this property. However, if multiple bricks are coupled, this may be a problem.

Remark

Instead of the surface generator RECTANGLE also the surface generators COONS and PARSURF may be used to generate the surfaces, provided the number of nodes is the same at opposite curves. In fact in these cases the underlying sub mesh generator is RECTANGLE.

Examples

Example 2.5.1.1 Simple brick

In this example we consider a brick of length 2, width 1 and height 0.5. The number of elements in length direction is defined by nelml, in the width direction by nelmw and in the height direction by nelmh.

The brick has exactly the structure of Figure 2.5.1.1. At the faces we use triangular elements (shape 3) and in the interior tetrahedrons (shape 11). The input file is given by:

```
*cube1.msh
constants
  integers
    nelml = 5
    nelmw = 4
    nelmh = 3
  reals
    length = 2
    width = 1
    height = 0.5
end
mesh3d
  points
    p1=(0,0,0)
    p2=( length,0,0)
    p3=( length, width,0)
    p4=(0, width,0)
    p5=(0,0, height)
    p8=(0,0,0)
  curves
    c1 =line1(p1,p2,nelm= nelml)
    c2 =line1(p2,p3,nelm= nelmw)
    c3 =line1(p3,p4,nelm= nelml)
    c4 =line1(p4,p1,nelm= nelmw)
    c5 =line1(p1,p5,nelm= nelmh)
    c6 =translate c5 ( p2, p6 )
    c7 =translate c5 ( p3, p7 )
    c8 =translate c5 ( p4, p8 )
    c9 =translate c1 ( p5, p6 )
    c10=translate c2 ( p6, p7 )
    c11=translate c3 ( p7, p8 )
    c12=translate c4 ( p8, p5 )
  surfaces
    s1=rectangle3(c1,c2,c3,c4)
    s2=rectangle3(c1,c6,-c9,-c5)
    s3=rectangle3(c2,c7,-c10,-c6)
    s4=rectangle3(-c3,c7,c11,-c8)
    s5=rectangle3(-c4,c8,c12,-c5)
    s6=rectangle3(c9,c10,c11,c12)
  volumes
    v1=brick11(s1,s2,s3,s4,s5,s6)
end
```

Example 2.5.1.2 Brick with different orientations

In this example we consider the same brick as in example 2.5.1.1, however, to demonstrate the use of the orientations each surface i has orientation $i+1$.

```
*cube2.msh
constants
  integers
    nelml = 5
    nelmw = 4
```

```
        nelmh = 3
    reals
        length = 2
        width = 1
        height = 0.5
end
mesh3d
    points
        p1=(0,0,0)
        p2=( length,0,0)
        p3=( length, width,0)
        p4=(0, width,0)
        p5=(0,0, height)
        p8=(0,0,0)
    curves
        c1 =line1(p1,p2,nelm= nelml)
        c2 =line1(p2,p3,nelm= nelmw)
        c3 =line1(p3,p4,nelm= nelml)
        c4 =line1(p4,p1,nelm= nelmw)
        c5 =line1(p1,p5,nelm= nelmh)
        c6 =translate c5 ( p2, p6 )
        c7 =translate c5 ( p3, p7 )
        c8 =translate c5 ( p4, p8 )
        c9 =translate c1 ( p5, p6 )
        c10=translate c2 ( p6, p7 )
        c11=translate c3 ( p7, p8 )
        c12=translate c4 ( p8, p5 )
    surfaces

#    Each surface has a different orientation

        s1=rectangle5(c2,c3,c4,c1)
        s2=rectangle5(-c9,-c5,c1,c6)
        s3=rectangle5(-c6,c2,c7,-c10)
        s4=rectangle5(c8,-c11,-c7,c3)
        s5=rectangle5(c4,c5,-c12,-c8)
        s6=rectangle5(-c10,-c9,-c12,-c11)
    volumes
        v1=brick13(s1,s2,s3,s4,s5,s6, orientation=234567)
    plot
end
```

The result of this input file is a standard brick file comparable with the one of example [2.5.1.1](#), however, with the triangles and tetrahedra replaced by quadrilaterals and hexahedrons.

2.5.2 Volume generator PIPE

The volume generator PIPE is called by program SEPMESH. The user may activate PIPE by data records of the type:

$$V_i = \text{PIPE } j \text{ (} S_1, S_2, S_3 \text{)}$$

with V_i the volume number, j the shape number of the elements created in this volume, and S_1, S_2, S_3 the three surfaces enclosing V_i . See Figure 2.5.2.1 for an explanation.

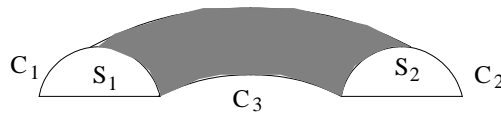


Figure 2.5.2.1: PIPE with generating surfaces

The surfaces S_1 and S_2 must have exactly the same topology, for example translated over some distance and rotated over some angle. For example S_2 may be created by the command TRANSLATE or SIMILAR. Surface S_3 must be generated by the command PIPESURFACE.

A simple example of the use of pipe is given in the following example.

Example 2.5.2.1 Construction of a simple pipe

Consider the straight pipe in Figure 2.5.2.2. The user points, curves and surfaces have already been placed in this figure.

In order to subdivide this pipe we have to generate three surfaces: the bottom surface, the top surface and a pipe surface.

The bottom surface is generated by GENERAL. Its boundary is given by a circle. In R^3 a circle needs at least three generating points.

The top surface is copied from the bottom surface.

The pipe surface is defined by the two circles and one extra generating curve, which is a straight line.

The following input file may be used to generate the mesh:

```
# simplepipe.msh
#
# Example of a simple pipe
# See users Manual Section 2.5.2
#
# To run this file use:
#   sepmesh simplepipe.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  integers
    nelmh =12      # Number of elements along the circles in bottom and
```

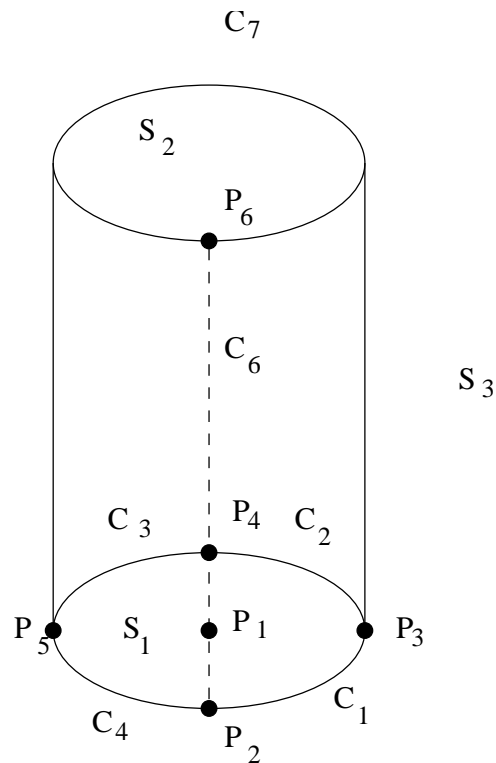


Figure 2.5.2.2: Example of a simple pipe

```

# top surface
nelmv = 3      # Number of elements in the vertical direction
              # (pipe surface)
shape_cur = 1  # Shape number of curve elements
shape_sur = 3  # Shape number of surface elements in horizontal dir
shape_vol = 11 # Shape number of volume elements
reals
radius = 1    # Radius of a circle in the bottom surface
height = 1    # Height of the pipe
end
#
# Define the mesh
#
mesh3d        # See Users Manual Section 2.2
#
# user points
#
points        # See Users Manual Section 2.2
# First points for bottom circle (3 points)
p1 = (0,0,0)  # centroid of circle in bottom surface
p2 = ( radius,0,0) # point on circle
p3 = (0,1,0)  # point is not used, just to define the
              # plane the circle is lying in
# Next points for top circle (3 points)
p11 = (0,0, height) # centroid of circle in top surface
p12 = ( radius,0, height) # point on circle
p13 = (0,1, height) # point is not used, just to define the
                   # plane the circle is lying in

```

```
#
# curves
#
curves          # See Users Manual Section 2.3
  c1 = circle  shape_cur (p1,p2,p3,nelm= nelmh)    # circle in bottom surface
  c2 = circle  shape_cur (p11,p12,p13,nelm= nelmh) # circle in top surface
  c3 = line    shape_cur (p2,p12,nelm= nelmv)      # straight line from p2 to p12
#
# surfaces
#
surfaces        # See Users Manual Section 2.4
  s1 = general  shape_sur (c1 )                    # bottom surface (circle)
  s2 = translate s1 (c2 )                          # top surface (circle)
                                                    # This surface must be topological
                                                    # equivalent to s1, hence translate
  s3 = pipesurface shape_sur (c1,c2,c3)           # pipe surface
#
# volumes
#
volumes         # See Users Manual Section 2.5
  v1 = pipe    shape_vol (s1,s2,s3)                # Complete pipe
plot, eyepoint = (1.0, 0.5, 1.5)                 # make a plot of all parts
                                                    # and also of the final mesh
                                                    # See Users Manual Section 2.2

end
```

Figure 2.5.2.3 shows the mesh generated by these statements.

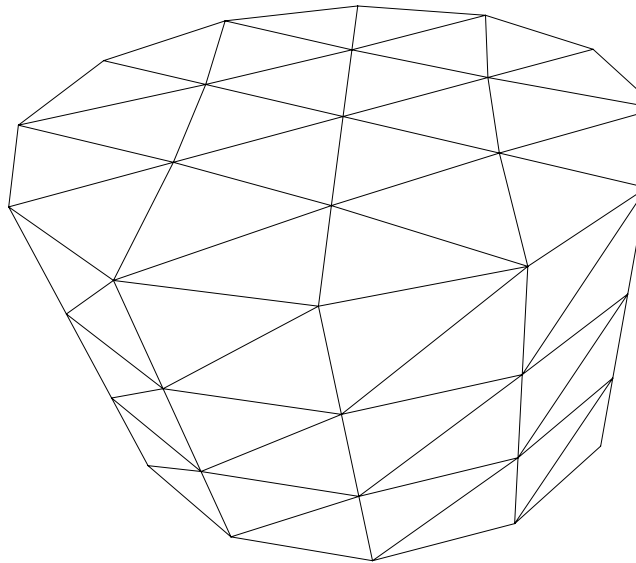


Figure 2.5.2.3: Pipe generated by sepmesh

Example 2.5.2.2 A straight pipe with a hole

In this example we consider a straight pipe, where the bottom and top surface consist of a rectangle with a hole. This example has been supplied by Bas van Rens of the technical University Eindhoven. Figure 2.5.2.4 shows the mesh as generated by `sepmesh`. In this example we show how a pipe with

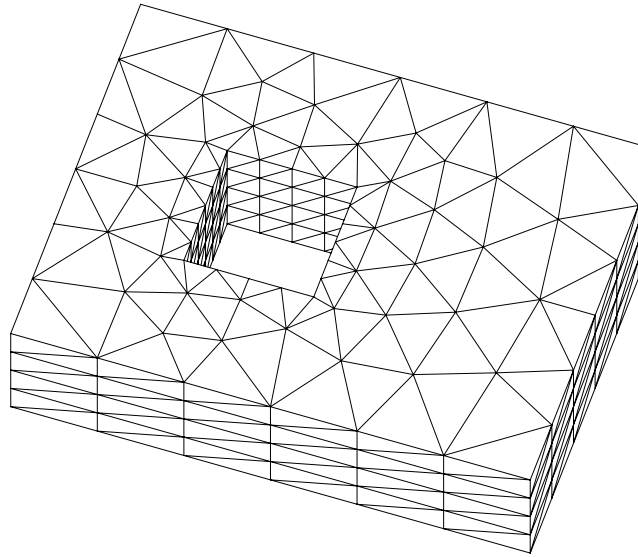


Figure 2.5.2.4: straight pipe with hole generated by `sepmesh`

a hole can be generated, without constructing extra lines from outer boundary to hole. To that end submesh generator `triangle` is used to generate the bottom surface, since `general` can not produce a hole without extra connection lines. the pipe surfaces for the outer boundary and the hole are created separately. They require each only one generating curve, a bottom curve and a top curve. These two pipe surfaces, although not connected, can be made to one pipe surface by using the ordered surface statement. Once that has been done and the top surface is generated as a translation of the bottom surface, the pipe can be generated in one statement using only three surfaces.

The corresponding input is given in the following lines:

```
* pipehole.msh
*
* Example of a straight pipe with a hole
*
*
* Define some general constants
*
constants          # See Users Manual Section 1.4
  integers
    nelml = 6      # Number of elements along the length direction of
                  # the bottom surface
    nelmw = 6      # Number of elements along the width direction of
                  # the bottom surface
```



```

*
surfaces          # See Users Manual Section 2.4
  s1=triangle3(c9,-c12)          # Bottom surface
  s2=translate s1(c10,-c13)      # Top surface
  s3=pipesurface3(c9,c10,c11)   # Outer pipe surface
  s4=pipesurface3(-c12,-c13,c14) # Pipe surface along hole
  s5=ordered surfaces(s3,s4)     # The complete pipe surface consists
                                # of two separate parts
*
* volumes
*
volumes          # See Users Manual Section 2.5
  v1=pipe11(s1,s2,s5)           # The complete pipe is generated
plot, eyepoint=(1000,-3000,5000) # make a plot of all parts
                                # and also of the final mesh
                                # See Users Manual Section 2.2
end

```

Example 2.5.2.3 A conical pipe

This example is almost identical to the first one, with the exception that the top surface has a radius that is different from the bottom surface. This implies that the surface can not be copied by translate, but that we have to use the command similar, which creates a surface that is topologically equivalent to the bottom surface. the input file is given by the following statements:

```

# conepipe.msh
*
* Example of a simple conical pipe
*
* Define some general constants
*
constants        # See Users Manual Section 1.4
  integers
    nelmh =12     # Number of elements along the circles in bottom and
                  # top surface
    nelmv = 3     # Number of elements in the vertical direction
                  # (pipe surface)
  reals
    radius_bot = 1 # Radius of a circle in the bottom surface
    radius_top = 2 # Radius of a circle in the top surface
    height = 1    # Height of the pipe
end
*
* Define the mesh
*
mesh3d           # See Users Manual Section 2.2
*
* user points
*
points          # See Users Manual Section 2.2
  # First points for bottom circle (3 points)
  p1 = (0,0,0)  # centroid of circle in bottom surface
  p2 = ( radius_bot,0,0) # point on circle
  p3 = (0,1,0)  # point is not used, just to define the

```

```

# plane the circle is lying in
# Next points for top circle (3 points)
p11 = (0,0, height) # centroid of circle in top surface
p12 = ( radius_top,0, height) # point on circle
p13 = (0,1, height) # point is not used, just to define the
# plane the circle is lying in
*
* curves
*
curves # See Users Manual Section 2.3
c1 = circle(p1,p2,p3,nelm= nelmh) # circle in bottom surface
c2 = circle(p11,p12,p13,nelm= nelmh) # circle in top surface
c3 = line1(p2,p12,nelm= nelmv) # straight line from p2 to p12
*
* surfaces
*
surfaces # See Users Manual Section 2.4
s1 = general3 (c1 ) # bottom surface (circle)
s2 = similar s1 (c2 ) # top surface (circle)
# This surface must be topological
# equivalent to s1, hence similar
s3 = pipesurface3(c1,c2,c3) # pipe surface
*
* volumes
*
volumes # See Users Manual Section 2.5
v1 = pipe11(s1,s2,s3) # Complete pipe
plot, eyepoint = (10, 5, -5) # make a plot of all parts
# and also of the final mesh
# See Users Manual Section 2.2
end
```

Figure 2.5.2.5 shows the mesh generated by these statements.

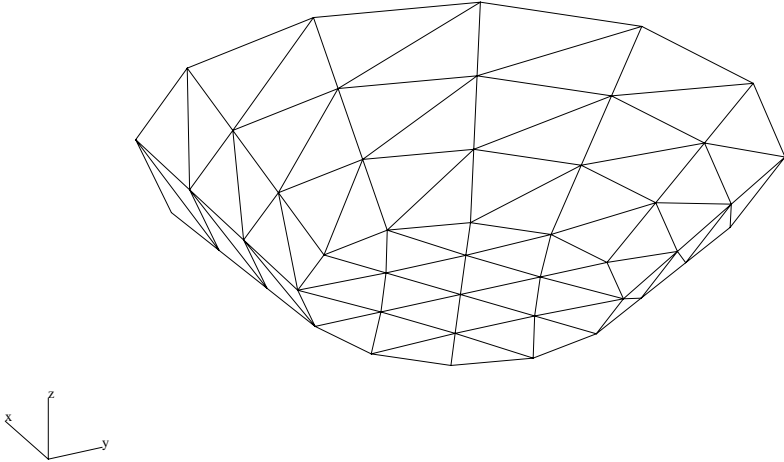


Figure 2.5.2.5: Conical pipe generated by sepmesh

2.5.3 Volume generator CHANNEL

The volume generator CHANNEL is called by program SEPMESH. The user may activate CHANNEL by data records of the type:

```
Vi = CHANNEL j ( S1, S2, S3 )
```

with V_i the volume number, j the shape number of the elements created in this volume, and S_1 , S_2 , S_3 the three surfaces enclosing V_i . See Figure 2.5.2.1 for an explanation.

In fact CHANNEL is completely identical to PIPE (2.5.2), with the exception of the computation of the coordinates.

The surfaces S_1 and S_2 must have exactly the same topology, which means that corresponding points in one surface are connected to the same points as the points in the other surface.

Surface S_3 must be generated by the command PIPESURFACE using only one generating curve. In fact it is assumed that the pipe surface connecting S_1 and S_2 consists of straight lines only. The subdivision of elements along these lines is assumed to be the same for all lines.

The new points in the grid are all created by connecting corresponding points on S_1 and S_2 by straight lines and subdividing these lines in exactly the same way as on S_3 . Although this may seem a limitation compared to PIPE, it allows the surfaces S_1 and S_2 to be much more curved than in the case of PIPE.

An example of the use of channel is given in the following example.

Example 2.5.3.1 Meshing a quarter of a TV tube

In this example we create a mesh that is to be used to simulate the construction of a TV tube from liquid glass. In order to simulate the filling of the glass it is necessary to create a mesh between inner and outer surface of the tube. For our computation it is sufficient to consider only one quarter of the tube.

To get this example into your local directory use the command sepgetex:

```
sepgetex raytube
```

To run the example use:

```
seplink raytube
raytube < raytube.msh
sepview sepplot.001
```

Figure 2.5.3.1 shows the curves that are created in this example. It also shows the contours of the construction. The most important curve numbers on the outer surface are plotted in Figures 2.5.3.2 and 2.5.3.3

In order to subdivide this tube we have to generate three surfaces: the outer surface, the inner surface and a pipe surface connecting the two.

The outer surface is split into four parts:

1. the back plane together with the lower plane and the curved part between the two, see Figure 2.5.3.4
2. the right-hand side plane together with the curved part to the lower plane, see Figure 2.5.3.5
3. The circular part between right-hand side plane and back plane, see Figure 2.5.3.6
4. The sphere part connecting all planes, see Figure 2.5.3.7

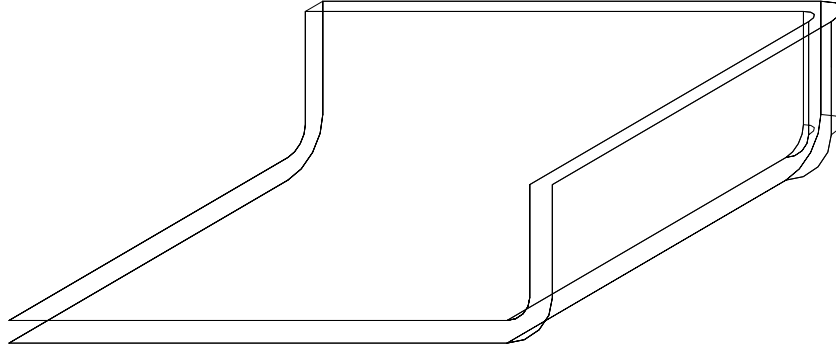


Figure 2.5.3.1: curve in TV tube

The inner surface is constructed completely identical.

The pipesurface needs one generating curve and the two outer curves of outer and inner surface. The outer curve must be constructed starting with the first curve of the first subsurface of the complete surface. Otherwise outer and inner surface are not compatible with the pipe surface.

Figure 2.5.3.8 shows the pipe surface.

The following input file may be used to generate the mesh:

```
# raytube.msh
#
# Input file for the quarter of a TV tube as described in Section 2.5.3
# of the Users Manual
#
# First some general constants are defined
#
constants          # See Users Manual Section 1.4
  reals
    l_x = 0.22      # length tube in x direction
    l_y = 0.16      # length tube in y direction
    l_z = 0.05      # length tube in z direction
    b = 0.01        # thickness of tube tube
    R_w = 0.02      # radius of circles at outer surface (wide)
    R_s = 0.01      # radius of circles at inner surface (small)
    y_w = l_y+ R_w  # y coordinate of outer back plane
    y_s = l_y+ R_s  # y coordinate of inner back plane
    z_w = l_z+ R_w  # z coordinate of top of back plane
    x_w = l_x+ R_w  # x coordinate of outer plane at the right
    x_s = l_x+ R_s  # x coordinate of inner plane at the right
  integers
    nelmx = 5       # number of elements in x direction
    nelmy = 5       # number of elements in y direction
    nelmz = 5       # number of elements in z direction
    nelmb = 3       # number of elements in width direction
```

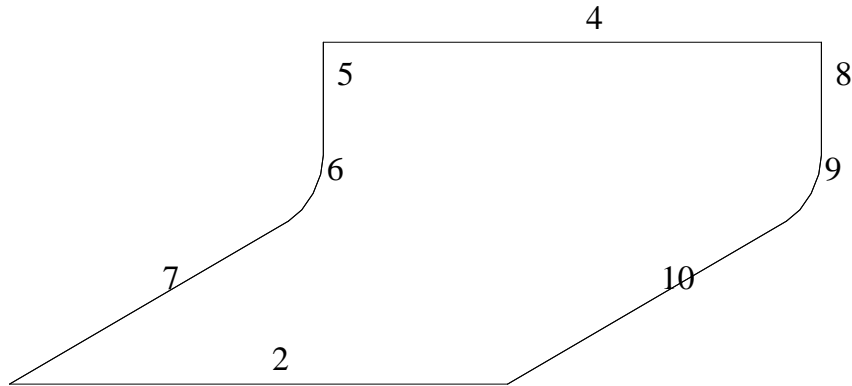


Figure 2.5.3.2: curve numbers in back plane and lower plane in TV tube

```

nelmc = 4      # number of elements along the circles
shape_cur = 1  # linear elements along the curves
shape_sur = 5  # bilinear quadrilaterals along the surfaces
shape_vol = 13 # trilinear hexahedrons in the volume
end
#
# Define the mesh
#
mesh3d          # See Users Manual Section 2.2
#
# user points
#
points          # See Users Manual Section 2.2

# First the user points on the outer surface are defined

# lower plane (z=0)

p4 = (0,0,0)      # left under
p8 = ( l_x, 0, 0 ) # right under
p3 = (0, l_y,0)   # left upper
p7 = ( l_x, l_y,0) # right upper

# back plane (y=y_w)

p2 = (0, y_w, R_w) # left under
p6 = ( l_x, y_w, R_w) # right under
p1 = (0, y_w, z_w) # left upper
p5 = ( l_x, y_w, z_w) # right upper

# circular part between back plane and lower plane

```

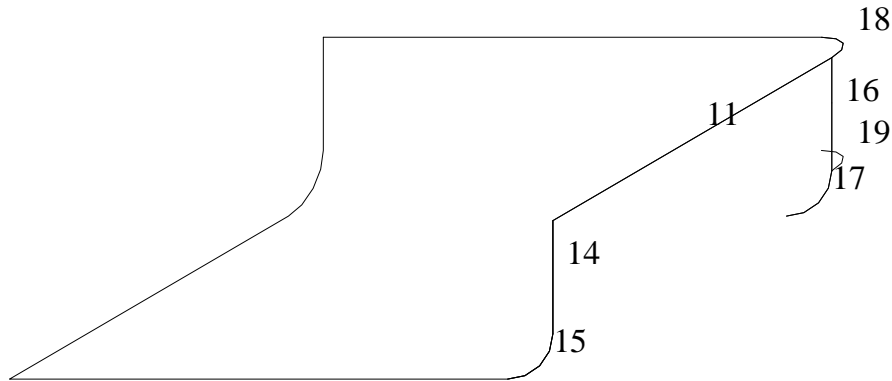



Figure 2.5.3.3: curve numbers in other planes in TV tube

```

p13 = ( 0, l_y, R_w)      # Centre of outer circle left
p14 = ( l_x, l_y, R_w)   # Centre of outer circle right

# Right-hand plane (x=x_w)

p10 = ( x_w, l_y, R_w)   # left under
p11 = ( x_w, 0, R_w)     # right under
p9  = ( x_w, l_y, z_w)   # left upper
p12 = ( x_w, 0, z_w)     # right upper

# inner surface
# the same numbering as for the outer surface is used increased by 20

# lower plane (z=b)

p24 = ( 0, 0, b )        # left under
p28 = ( l_x, 0, b )      # right under
p23 = ( 0, l_y, b )      # left upper
p27 = ( l_x, l_y, b )    # right upper

# back plane (y=y_s)

p22 = ( 0, y_s, R_w)     # left under
p26 = ( l_x, y_s, R_w)   # right under
p21 = ( 0, y_s, z_w)     # left upper
p25 = ( l_x, y_s, z_w)   # right upper

# Right-hand plane (x=x_s)

p30 = ( x_s, l_y, R_w)   # left under
p31 = ( x_s, 0, R_w)     # right under

```

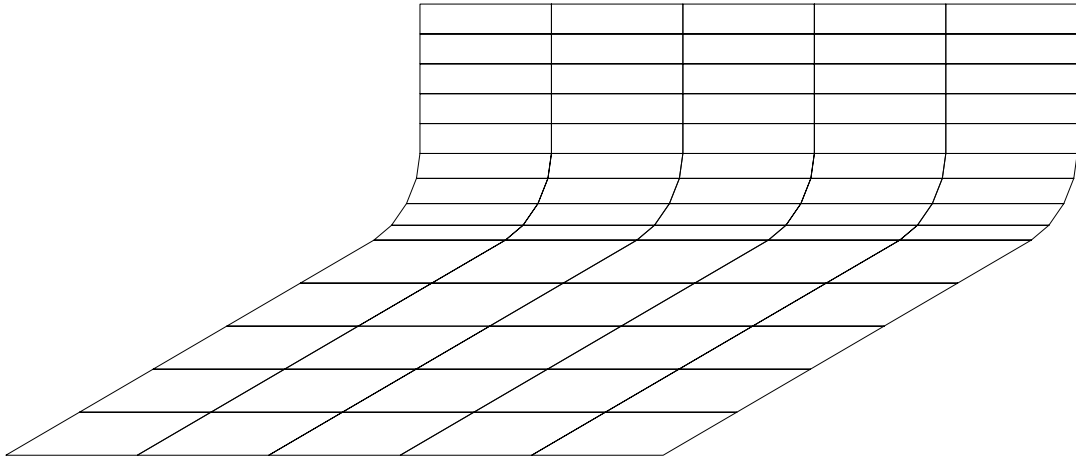


Figure 2.5.3.4: Back plane and lower plane

```

p29 = ( x_s, l_y, z_w) # left upper
p32 = ( x_s,0, z_w)    # right upper
#
# curves
#
curves          # See Users Manual Section 2.3

# outer surface

# Back plane and lower plane together
# first the four outer curves

c1 = curves ( c5, c6, c7 )           # left-hand curve
c2 = line  shape_cur (p4,p8,nelm= nelmx) # lower curve
c3 = curves ( c8, c9, c10 )         # right-hand curve
c4 = translate c2 ( p1, p5 )        # upper curve

# Next the subcurves of c1 and c3
# c1:

c5 = line  shape_cur (p1,p2,nelm= nelmz) # back plane part
c6 = arc  shape_cur (p2,p3, p13, nelmc ) # circular part
c7 = line  shape_cur (p3,p4,nelm= nelmy) # lower face

# c3:

c8 = translate c5 ( p5, p6 )         # back plane part
c9 = translate c6 ( p6, p7 )         # circular part
c10= translate c7 ( p7, p8 )         # lower face

# Right-hand plane and circular part to lower plane
# First outer curves (c10 has already been created!)

```

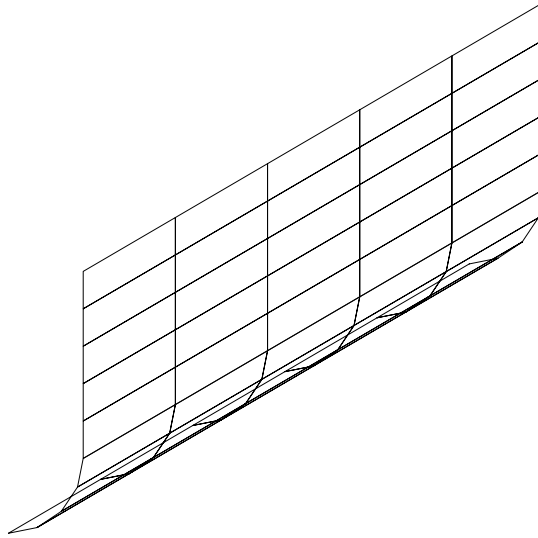


Figure 2.5.3.5: right-hand side plane

```

c11 = translate c10 ( p9, p12 )           # upper curve
c12 = curves ( c14, c15 )                # front curve
c13 = curves ( c16, c17 )                # curve at the back

# Next the subcurves of c12 and c13

c14 = translate c5 ( p12, p11 )           # right-hand plane (c12)
c17 = arc shape_cur (p10,p7, p14, nelmc) # circular part on c13
c16 = translate c5 ( p9, p10 )           # right-hand plane (c13)
c15 = translate c17 ( p11, p8 )          # circular part on c12

# curved part between Right-hand plane and back plane

c19 = arc shape_cur (p6,p10, p14, nelmc) # lower circle
c18 = translate c19 ( p5, p9 )           # upper circle

# The complete outer curve:

c20 = curves ( c1, c2, -c15, -c14, -c11, -c18, -c4 )

# inner surface
# Exactly the same construction as for the outer surface
# curve numbers and point numbers are increased by 20

# Back plane and lower plane together
# first the four outer curves

c21 = curves ( c25, c26, c27 )           # left-hand curve
c22 = line shape_cur (p24,p28,nelmx)     # lower curve
c23 = curves ( c28, c29, c30 )           # right-hand curve

```



Figure 2.5.3.6: curved part

```

c24 = translate c22 ( p21, p25 )           # upper curve

# Next the subcurves of c21 and c23
# c21:

c25 = line  shape_cur (p21,p22,nelm= nelmz)   # back plane part
c26 = arc   shape_cur (p22,p23, p13, nelmc )  # circular part
c27 = line  shape_cur (p23,p24,nelm= nelmy)   # lower face

# c23:

c28 = translate c25 ( p25, p26 )           # back plane part
c29 = translate c26 ( p26, p27 )           # circular part
c30 = translate c27 ( p27, p28 )           # lower face

# Right-hand plane and circular part to lower plane
# First outer curves (c30 has already been created!)

c31 = translate c10 ( p29, p32 )           # upper curve
c32 = curves ( c34, c35 )                 # front curve
c33 = curves ( c36, c37 )                 # curve at the back

# Next the subcurves of c32 and c33

c34 = translate c25 ( p32, p31 )           # right=hand plane (c32)
c37 = arc   shape_cur (p30,p27, p14, nelmc ) # circular part on c33
c36 = translate c25 ( p29, p30 )           # right=hand plane (c33)
c35 = translate c37 ( p31, p28 )           # circular part on c32

# curved part between Right-hand plane and back plane

c39 = arc   shape_cur (p26,p30, p14, nelmc ) # lower circle

```

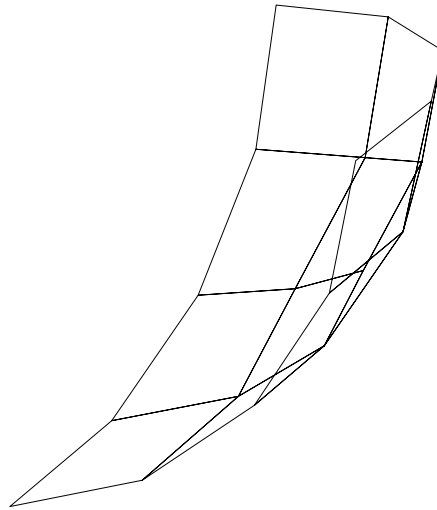


Figure 2.5.3.7: Sphere

```

c38 = translate c39 ( p25, p29 )           # upper circle

# The complete outer curve:

c40 = curves ( c21, c22, -c35, -c34, -c31, -c38, -c24 )

# One curve over the thickness

c41 = line shape_cur ( p1, p21, nelm= nelmb )

surfaces

# outer surface consisting of

# back + lower plane
s1 = pipesurface shape_sur ( c1, c3, c4, c2 )
# Right-hand plane
s2 = pipesurface shape_sur ( c11, c10, c13, c12 )
# curved part between back plane and Right-hand plane
s3 = pipesurface shape_sur ( c18, c19, c8, c16 )
# 1/8 sphere
s4 = isopar shape_sur ( c9, -c17, -c19, mapping=sphere, centre=p14 )
# These surfaces together
s5 = surfaces ( s1, s2, s3, s4 )

# inner surface consisting of

# back + lower plane
s6 = pipesurface shape_sur ( c21, c23, c24, c22 )
# Right-hand plane
s7 = pipesurface shape_sur ( c31, c30, c33, c32 )

```

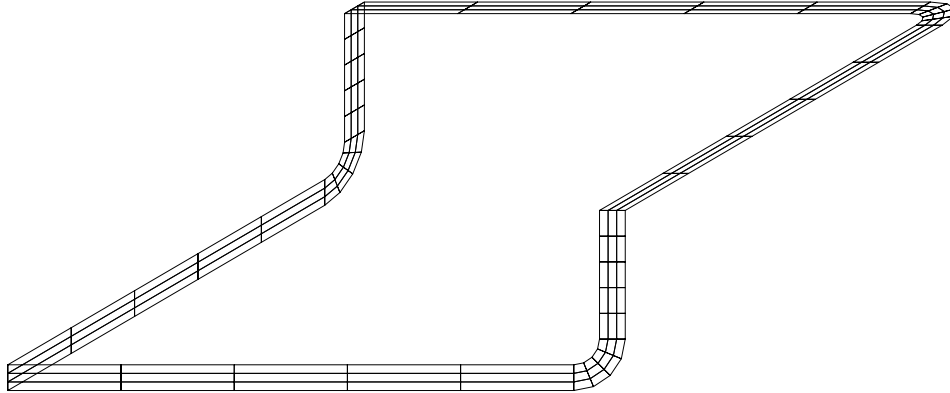


Figure 2.5.3.8: Pipe surface

```

# curved part between back plane and Right-hand plane
s8 = pipesurface shape_sur ( c38, c39, c28, c36 )
# 1/8 sphere
s9 = isopar shape_sur ( c29, -c37, -c39, mapping=sphere, centre=p14 )
# These surfaces together
s10 = surfaces ( s6, s7, s8, s9 )

# Boundary surface between inner and outer surface

s11 = pipesurface shape_sur ( c20, c40, c41 )

#
# volumes
#
volumes          # See Users Manual Section 2.5

v1 = channel shape_vol ( s5, s10, s11 )          # Complete region

plot, eyepoint = ( -1, 5, -2)          # make a plot of all parts
                                          # and also of the final mesh
                                          # See Users Manual Section 2.2

end

```

Figure 2.5.3.9 shows the mesh generated by these statements.

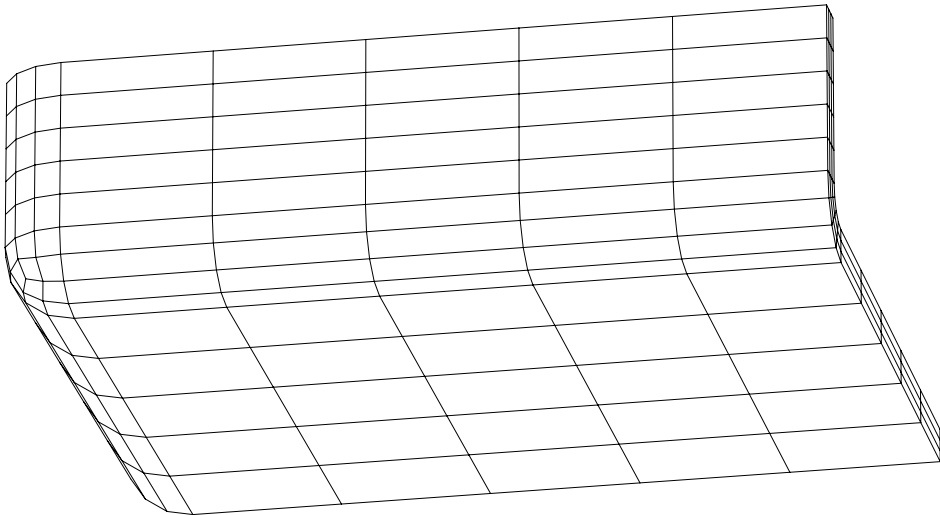


Figure 2.5.3.9: Final mesh

2.5.4 Volume generator GENERAL

The volume generator GENERAL is called by program SEPMESH. The user may activate GENERAL by data records of the type:

```
Vi = GENERAL j ( Sk [internal_points = p1,...,pm] &
  [internal_curves = c1,...,cm] [internal_surfaces = s1,...,sm] )
```

with V_i the volume number, j the shape number of the elements created in this volume, and S_k the surfaces enclosing V_i .

At this moment GENERAL is only able to generate tetrahedrons. hexahedrons are not allowed. The volume to be generated must be surrounded by one surface only, which of course may consists of several subsurfaces, connected to each other by the command:

```
Sk = SURFACES ( Sj, S1, Sm, ... )
```

Usually S_k must be a closed surface, but it may also consist of a number of closed surfaces, as long as it is clear which region is meant. For example it is allowed to generate a region with a hole in it by letting the surface S_k consist of the outer surface and the inner surface together. In that case GENERAL creates a mesh in the region between these two closed surfaces.

Due to its generality GENERAL takes more computer time than the other structured mesh generators, especially if the the number of elements is large. This computer time may be reduced by starting with a coarse mesh and to refine it afterwards by the command **refine**. The size of the elements is determined by the size of the elements in the surrounding surface.

The option `internal_points = p1, ..., pm` forces nodal points to coincide with the user points `p1, ..., pm`. The local coarseness in these points are used for the coarseness of the mesh locally. So by defining these points the user can create fixed points in the mesh, with a given accuracy, without having to define curves containing these points.

With `internal_curves = c1, ..., cm` the user can define curves inside the domain. The mesh is adapted to these curves, which means that elements may have a common side with an element of these curves, but never intersects these curves. All edges on the internal curves are present in the final mesh, so the size of these edges also defines the local coarseness. internal curves must not be part of one of the surfaces.

With `internal_surfaces = s1, ..., sm` the user can define surfaces inside the domain. The mesh is adapted to these surfaces, which means that elements may have a common side with an element of these surfaces, but never intersects these surfaces. All edges on the internal surfaces are present in the final mesh, so the size of these edges also defines the local coarseness.

Remark: in some extreme cases GENERAL may fail to create a mesh.

It is then possible to replace `general` by `old_general`, which corresponds to the previous version of GENERAL. If GENERAL fails, sometimes OLD_GENERAL may be able to create a mesh.

Below we give a number of examples of the use of GENERAL.

Example 2.5.4.1 Subdivision of a tetrahedron

Consider the tetrahedron in Figure 2.5.4.1. The curve numbers have also been plotted in this figure. In order to create a mesh in this tetrahedron we use the mesh generator GENERAL. As a consequence the outer surface must be joined into one surface. The following input file may be used to create elements in this case. In order to reduce the computing time a coarse mesh is created that is refined afterwards.

```
# tetrahedron.msh
```

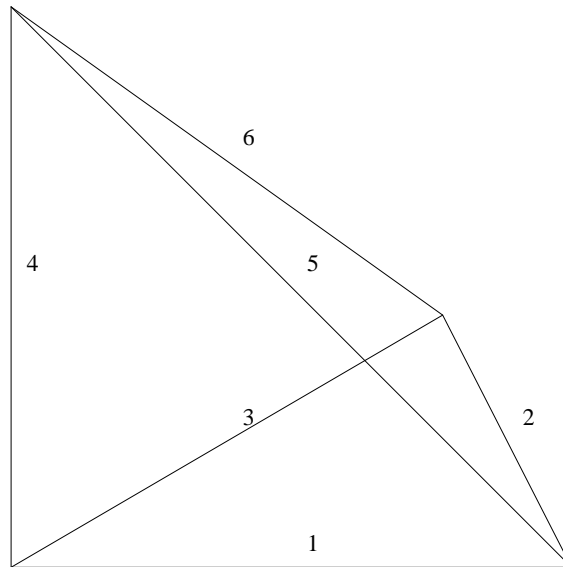



Figure 2.5.4.1: Tetrahedron and corresponding curves

```

#
# Example of a mesh in a tetrahedron created by general
#
# See users manual Section 2.5.4
#
#
# To run this file use:
#   sepmesh tetrahedron.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    size = 1       # Size of the tetrahedron
  integers
    nxy = 2        # Number of elements along a curve in the xy-plane
    nz = 3         # Number of elements along other curves
    shape_cur = 1  # Shape number of elements along curves
                   # 1 = linear elements
    shape_sur = 3  # Shape number of elements along surfaces
                   # 3 = linear triangles
    shape_vol = 11 # Shape number of elements along volumes
                   # 11 = linear tetrahedrons
end
#
# Define the mesh

```

```

#
mesh3d          # See Users Manual Section 2.2
#
# user points
#
  points        # See Users Manual Section 2.2

    p1 = ( 0,    0,    0 ) # 4 vertices of tetrahedron
    p2 = ( size,  0,    0 )
    p3 = ( 0, size,    0 )
    p4 = ( 0,    0, size )
#
# curves
#
  curves        # See Users Manual Section 2.3

    c1 = line shape_cur ( p1, p2, nelm = nxy )
    c2 = line shape_cur ( p2, p3, nelm = nxy )
    c3 = line shape_cur ( p3, p1, nelm = nxy )
    c4 = line shape_cur ( p1, p4, nelm = nz )
    c5 = line shape_cur ( p2, p4, nelm = nz )
    c6 = line shape_cur ( p3, p4, nelm = nz )
#
# surfaces
#
  surfaces      # See Users Manual Section 2.4

    s1 = triangle shape_sur ( c1, c2, c3 ) # 4 surfaces defining the
    s2 = triangle shape_sur ( c1, c5, -c4 ) # tetrahedron
    s3 = triangle shape_sur ( c2, c6, -c5 )
    s4 = triangle shape_sur ( -c4, -c3, c6 )

    s5 = surfaces ( s1, s2, s3, s4 )        # all surfaces combined to 1
#
# volumes
#
  volumes      # See Users Manual Section 2.5

    v1 = general shape_vol (s5)
#
# Refine the mesh 2 times, i.e. the number of elements along each side
# is multiplied by 4
#
  refine 2 times

  plot, eyepoint=(5,10,-2)      # make a plot of the mesh
                                # See Users Manual Section 2.2

end

```

Figure [2.5.4.2](#) shows the mesh created in this example.

Example 2.5.4.2 Subdivision of a block with a hole

Consider the block in Figure [2.5.4.2](#) provided with a conical hole inside. The curve numbers have also been plotted in this figure. We want to create a mesh in the block, but outside the hole. This

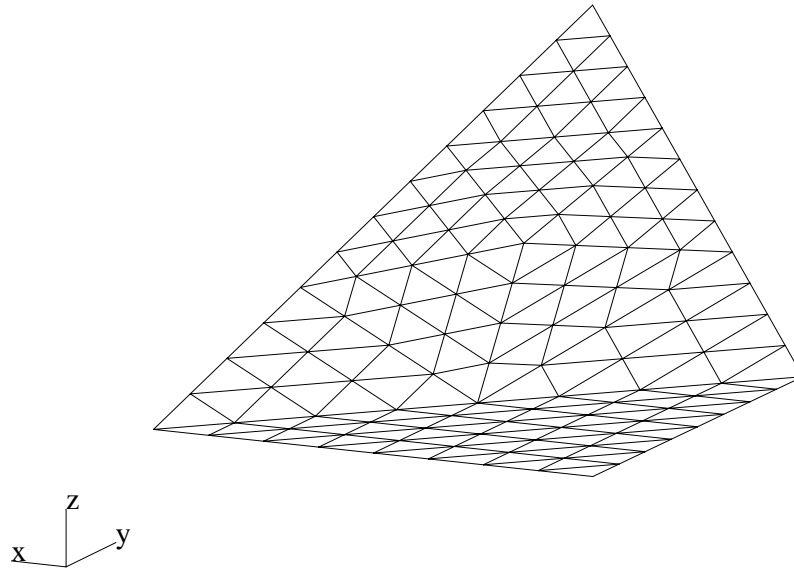


Figure 2.5.4.2: Mesh created in tetrahedron

may be done by GENERAL provided the complete outer surface including the surface of the hole is joined into one surface. GENERAL automatically detects the region between the two surfaces. The following input file may be used to create elements in this case. In order to reduce the computing time a coarse mesh is created that is refined afterwards.

```
# generalhole.msh
#
# Example of a mesh in a block with an internal hole
#
# See users manual Section 2.5.4
#
#
# To run this file use:
#   sepmesh tetrahedron.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  integers
    nelmh = 2      # Number of elements in the horizontal direction
    nelmv = 2      # Number of elements in the vertical direction
  reals
    z_block_bottom = -2    # Z-coordinate of the block bottom
    z_block_top    = 2     # Z-coordinate of the block top
    z_hole_bottom  = -1    # Z-coordinate of the hole bottom
    z_hole_top     = 1     # Z-coordinate of the hole top
```

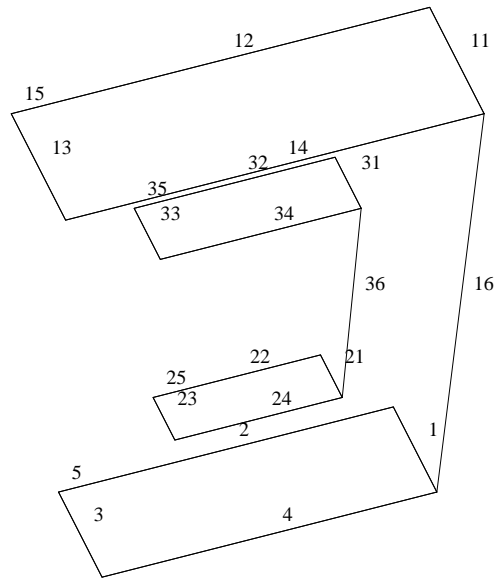


Figure 2.5.4.3: Block with hole and corresponding curves

```

end
#
# Define the mesh
#
mesh3d          # See Users Manual Section 2.2
#
# user points
#
points          # See Users Manual Section 2.2

p1 = ( 2 , 0 ,z_block_bottom)
p2 = ( 0 , 2 ,z_block_bottom)
p3 = (-2 , 0 ,z_block_bottom)
p4 = ( 0 ,-2 ,z_block_bottom)

p11 = ( 2.5 , 0 ,z_block_top)
p12 = ( 0 , 2.5 ,z_block_top)
p13 = (-2.5 , 0 ,z_block_top)
p14 = ( 0 ,-2.5 ,z_block_top)

p21 = ( 1 , 0 ,z_hole_bottom)
p22 = ( 0 , 1 ,z_hole_bottom)
p23 = (-1 , 0 ,z_hole_bottom)
p24 = ( 0 ,-1 ,z_hole_bottom)

p31 = ( 1.2 , 0 ,z_hole_top)
p32 = ( 0 , 1.2 ,z_hole_top)
p33 = (-1.2 , 0 ,z_hole_top)

```

```

    p34 = ( 0 , -1.2 , z_hole_top)
#
# curves
#
curves          # See Users Manual Section 2.3
# bottom face
c1 = line1(p1,p2,nelm=nelmh)
c2 = line1(p2,p3,nelm=nelmh)
c3 = line1(p3,p4,nelm=nelmh)
c4 = line1(p4,p1,nelm=nelmh)
c5 = curves(c1,c2,c3,c4)          # block bottom curve

# top face
c11 = line1(p11,p12,nelm=nelmh)
c12 = line1(p12,p13,nelm=nelmh)
c13 = line1(p13,p14,nelm=nelmh)
c14 = line1(p14,p11,nelm=nelmh)
c15 = curves(c11,c12,c13,c14)    # block top curve

# outer face
c16 = line1(p1,p11,nelm=nelmv)   # First edge of block surface

# bottom face of hole
c21 = line1(p21,p22,nelm=nelmh)
c22 = line1(p22,p23,nelm=nelmh)
c23 = line1(p23,p24,nelm=nelmh)
c24 = line1(p24,p21,nelm=nelmh)
c25 = curves(c21,c22,c23,c24)    # hole bottom curve

# top face of hole
c31 = line1(p31,p32,nelm=nelmh)
c32 = line1(p32,p33,nelm=nelmh)
c33 = line1(p33,p34,nelm=nelmh)
c34 = line1(p34,p31,nelm=nelmh)
c35 = curves(c31,c32,c33,c34)    # hole top curve

# outer face of hole
c36 = line1(p21,p31,nelm=nelmv)   # First edge of hole conical surface
#
# surfaces
#
surfaces        # See Users Manual Section 2.4
s1 = general3(c5)          # block bottom surface
s2 = general3(c15)         # block top surface
s3 = pipesurface3(c5,c15,c16) # block outer surface
s4 = surface(s1,s2,s3)     # block boundary

s5 = general3(c25)         # hole bottom surface
s6 = general3(c35)         # hole top surface
s7 = pipesurface3(c25,c35,c36) # hole outer surface
s8 = surface(s5,s6,s7)    # hole boundary

s9 = surfaces(s4,s8)       # All surfaces together
#
# volumes

```

```
#
volumes          # See Users Manual Section 2.5
  v1 = general11(s9)      # block with internal hole

plot, eyepoint = (10, 5, 10)      # make a plot of all parts
                                   # and also of the final mesh
                                   # See Users Manual Section 2.2

end
```

Figure 2.5.4.4 is a plot of the complete outer surface including the surface of the hole, Figure 2.5.4.5 shows the mesh created in this example.

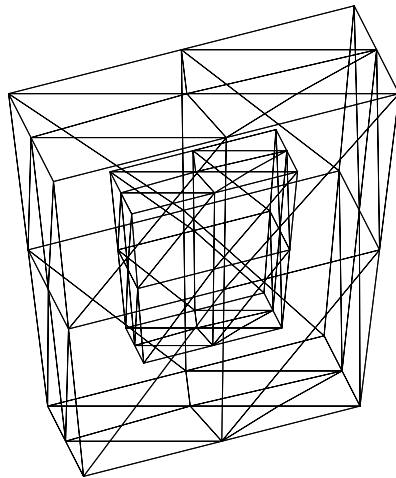


Figure 2.5.4.4: Surface of block with hole

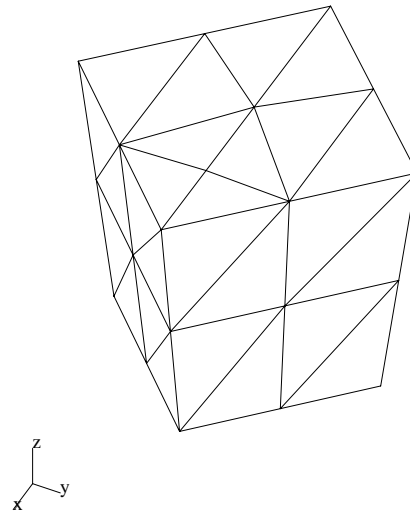


Figure 2.5.4.5: Mesh created in block with hole

2.6 Some examples of meshes generated by SEPRAN

In this section we give some examples of meshes generated by SEPRAN. The complete input and definition of these meshes is described.

Example 2.6.1 Three coupled pipes

In this example we consider three pipes with the same axis, but with different radius. Figure 2.6.1 shows a sketch of the situation. Each of the pipes has a separate volume number (V_1, V_2 and V_3).

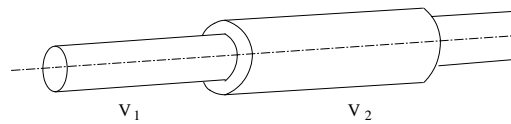


Figure 2.6.1: Definition of the three coupled pipes (V_1, V_2 and V_3)

The pipes will be generated by the volume generator PIPE. To that end exactly three surfaces are needed to define each pipe. In Figures 2.6.2 and 2.6.3 the surfaces $S_1, S_2,$ and S_3 defining V_1 , S_5, S_8 and S_9 defining V_2 and S_6, S_{10} and S_{11} defining V_3 are indicated. The definition of the curves and points defining these surfaces are given in Figures 2.6.4, 2.6.5 and 2.6.6. Since S_2 and S_6 are parts of the surfaces S_5 respectively S_8 , the surfaces S_5 and S_8 must be defined as surfaces of surfaces.

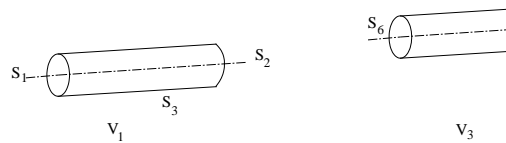


Figure 2.6.2: Definition of the surfaces at the volumes V_1 and V_3

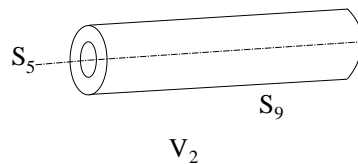


Figure 2.6.3: Definition of the surfaces at the volume V_2

Surface S_1 is the *bottom* surface of pipe V_1 , and is defined by the curve C_5 which consists of the subcurves C_1, C_2, C_3 and C_4 . The surface is generated by the surface generator GENERAL. Linear triangles are used in the surfaces. The curves C_1, \dots, C_4 each define one quarter of the generating circle C_5 .

Surface S_2 is the *top* surface of pipe V_1 , and is defined by translation of S_1 , since PIPE requires congruent *top* and *bottom* surfaces. The generating curve C_6 is defined by translation of C_5 over the distance $P_6 - P_1$. The third surface is generated by PIPESURFACE with generating curves

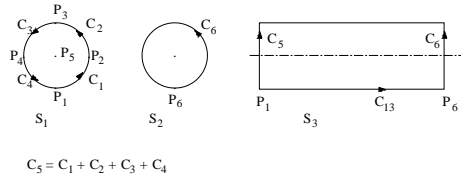


Figure 2.6.4: Definition of curves and points in surfaces of volume V_1

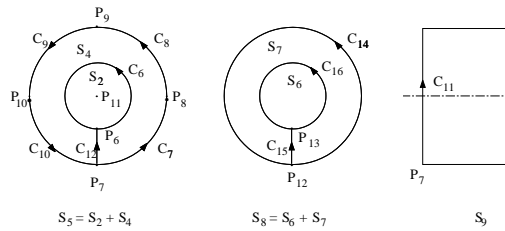


Figure 2.6.5: Definition of curves and points in surfaces of volume V_2

C_5, C_6 and C_{13} .

Surface S_5 is the *bottom* surface of pipe V_2 . Since S_2 is a part of S_5 , S_5 must be a surface of surfaces. The rest of S_5 is the subsurface S_4 defined by the curves C_{11} ($C_{11} = C_7 + C_8 + C_9 + C_{10}$), $C_{12}, -C_6, -C_{12}$. This subsurface is also generated by GENERAL.

Surface S_8 is the *top* surface of pipe V_2 . S_8 consists of the subsurfaces S_6 and S_7 where S_6 is the result of translation of S_2 and S_7 of translation of S_4 . The generating curve of S_6 is C_{16} which is created by translation of C_6 ; S_7 has generating curves $C_{14}, C_{15}, -C_{16}, -C_{15}$ each of which is translated from C_{11}, C_{12}, C_6 and C_{12} respectively.

Surface S_{10} is translated from S_6 with curve C_{17} as translation from C_{16} .

The surfaces S_8 and S_{11} are standard PIPE SURFACES.

The following input file may be used to generate the pipe:

```

*
* Example of a mesh for three connected pipes
*
mesh3d
  coarse(unit=.5)
  points
    p1 = ( 0, -1, 0 )
    
```

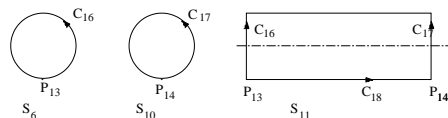


Figure 2.6.6: Definition of curves and points in surfaces of volume V_3


```
p2 = ( 1, 0, 0 )
p3 = ( 0, 1, 0 )
p4 = ( -1, 0, 0 )
p5 = ( 0, 0, 0 )
p6 = ( 0, -1, 1 )
p7 = ( 0, -2, 1 )
p8 = ( 2, 0, 1 )
p9 = ( 0, 2, 1 )
p10= ( -2, 0, 1 )
p11= ( 0, 0, 1 )
p12= ( 0, -2, 2 )
p13= ( 0, -1, 2 )
p14= ( 0, -1, 3 )
curves
c1 = carc1(p1,p2,p5)
c2 = carc1(p2,p3,p5)
c3 = carc1(p3,p4,p5)
c4 = carc1(p4,p1,p5)
c5 = curves(c1,c2,c3,c4)
c6 = translate c5(p6)
c7 = carc1(p7,p8,p11)
c8 = carc1(p8,p9,p11)
c9 = carc1(p9,p10,p11)
c10= carc1(p10,p7,p11)
c11 = curves(c7,c8,c9,c10)
c12=cline1(p7,p6)
c13=cline1(p1,p6)
c14=translate c11 (p12)
c15=translate c12 (p12,p13)
c16=translate c6 (p13)
c17=translate c16 (p14)
c18 = cline1(p13,p14)
c19 = cline1(p7,p12)
surfaces
s1 = general3 (c5)
s2 = translate s1(c6)
s3 = pipesurface 3(c5,c6,c13)
s4 = general 3 ( c11,c12,-c6,-c12 )
s5 = surfaces (s2,s4)
s6 = translate s2(c16)
s7 = translate s4 (c14,c15,-c16,-c15)
s8 = surfaces(s6,s7)
s9 = pipesurface 3 (c11,c14,c19)
s10 = translate s6(c17)
s11 = pipesurface 3 (c16,c17,c18)
volumes
v1 = pipe11(s1,s2,s3)
v2 = pipe11(s5,s8,s9)
v3 = pipe11(s6,s10,s11)
plot, eyepoint=(0,-5,1.5), rotate=2
end
```

The resulting mesh is plotted in Figure [2.6.7](#).

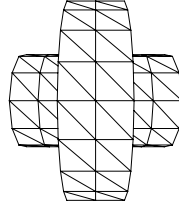


Figure 2.6.7: Hidden surface plot of mesh corresponding to three coupled pipes.

Example 2.6.2 Two concentric pipes with different material properties

In this example we consider two concentric pipes with common axis. The inner pipe has material properties which are different from the outer pipe. Therefore different element groups must be connected with each of the pipes. The pipes are generated by the volume generator PIPE (volumes V_1 and V_2). For each pipe exactly three surfaces are needed. Figure 2.6.8 defines the two concentric pipes; the corresponding surfaces are sketched in Figure 2.6.9.

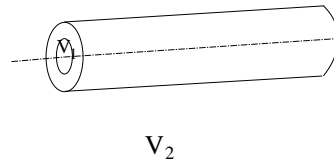


Figure 2.6.8: Definition of concentric pipes (V_1 and V_2)

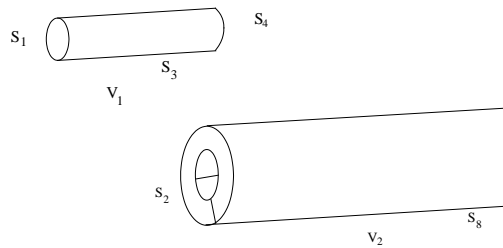


Figure 2.6.9: Definition of the surfaces at the volumes V_1 and V_2

Volume V_1 has *bottom* surface S_1 , *top* surface S_4 and pipe surface S_3 ; V_2 has *bottom* surface S_2 , *top* surface S_5 and pipe surface S_8 . The pipe surface S_8 is rather complicated because of the fact that V_2 is a hollow pipe. The definition of the curves and points of each surface is given in Figures 2.6.10 and 2.6.11.

Surface S_1 is the *bottom* surface of pipe V_1 , and is defined by the curve C_{10} which consists of the subcurves C_1, C_2, C_3 and C_4 . The surface is generated by the surface generator GENERAL. Linear triangles are used in the surfaces.

Surface S_4 is the *top* surface of pipe V_1 , and is defined by translation of S_1 , since PIPE requires congruent *top* and *bottom* surfaces. The generating curve C_{13} is defined by translation of C_{10} over the distance $P_2 - P_{10}$.

Surface S_3 is the pipe surface for the pipe. It has three generating curves: C_{10} (bottom), C_{13} (top)

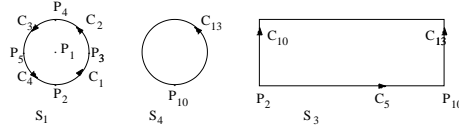


Figure 2.6.10: Definition of curves and points in surfaces of volume V_1

and C_5 (vertical curve). Since the pipe surface is closed, there is only one curve in vertical direction.

Surface S_2 is the *bottom* surface of pipe V_2 . It contains a hole enclosed by curve C_{10} . The curves defining S_2 are $C_5, C_6, -C_5$ and $-C_{10}$. The surface is generated by GENERAL.

Surface S_5 is created by translation of S_2 with defining curves: $C_{16}, C_{15}, -C_{16}$ and $-C_{13}$.

Surface S_8 is the most complicated one, since it contains surface S_3 as subsurface. Therefore S_8 must be a surface of surfaces, and because the pipe generator PIPE requires a surface generated by PIPE SURFACE, the surface of surfaces must be an ordered surface. The bottom line of the pipe surface is defined by the curves $C_5, C_{11}, -C_5$ and $-C_{10}$, where C_{11} consists of the subcurves C_6, C_7, C_8 and C_9 . We have already defined the pipe surface S_3 corresponding to curve C_{10} . The curve C_5 is a double curve in the definition of the bottom curve of S_8 . Therefore, it is necessary to define a pipe surface based on this bottom curve, and furthermore the ordered surface must start with this subsurface. Surface S_7 is the pipe surface defined with bottom curve C_5 , top curve C_{16} and vertical curves C_{14} and C_{17} . The envelop surface S_6 is defined by the curves C_{11}, C_{15} and C_{17} . Finally S_8 is an ordered surface defined by the subsurfaces $S_7, S_6, -S_7$ and $-S_3$. In this example there is only one row of subsurfaces.

Volume V_1 is connected to element group 1, V_2 to element group 2.

The following input file may be used to generate the concentric pipes:

```
*
*   mesh input for concentric pipes
*
mesh3d
  coarse ( unit=.5 )
  points
    p1 = ( 0, 0, 0 )
    p2 = ( 0, -1, 0 )
    p3 = ( 1, 0, 0 )
    p4 = ( 0, 1, 0 )
    p5 = ( -1, 0, 0 )
    p6 = ( 0, -2, 0 )
    p7 = ( 2, 0, 0 )
    p8 = ( 0, 2, 0 )
    p9 = ( -2, 0, 0 )
    p10= ( 0, -1, 1 )
    p11= ( 0, -2, 1 )
  curves
    c1 = carc1(p2,p3,p1)
    c2 = carc1(p3,p4,p1)
```

```

c3 = carc1(p4,p5,p1)
c4 = carc1(p5,p2,p1)
c5 = cline1(p2,p6)
c6 = carc1(p6,p7,p1)
c7 = carc1(p7,p8,p1)
c8 = carc1(p8,p9,p1)
c9 = carc1(p9,p6,p1)
c10 = curves(c1,c2,c3,c4)
c11 = curves(c6,c7,c8,c9)
c12 = curves(c5,c11,-c5,-c10)
c13= translate c10(p10)
c14= line1(p2,p10,nelm=2)
c15=translate c11 (p11)
c16=translate c5 (p10,p11)
c17=translate c14 (p6,p11)
surfaces
  s1 = general3 (c10)
  s2 = general3 (c12)
  s3 = pipesurface 3(c10,c13,c14)
  s4 = translate s1(c13)
  s5 = translate s2 (c16,c15,-c16,-c13)
  s6 = pipesurface 3(c11,c15,c17)
  s7 = pipesurface 3(c5,c16,c14,c17)
  s8 = ordered surface( (s7,s6,-s7,-s3) )
volumes
  v1 = pipe11(s1,s4,s3)
  v2 = pipe11(s2,s5,s8)
plot, eyepoint=(-3,-3,-5),rotate=2
end

```

The resulting mesh is plotted in Figure 2.6.12.

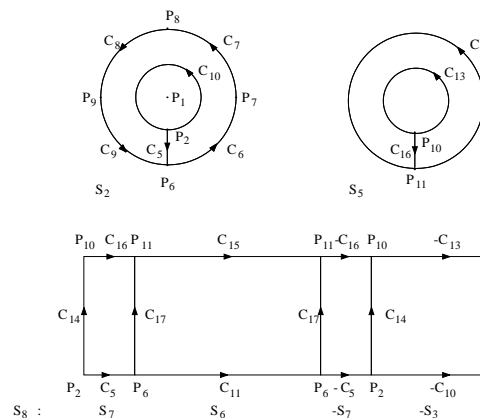


Figure 2.6.11: Definition of curves and points in surfaces of volume V_2


```

# is only used in order to connect
# the profile with outer boundary
# This is necessary since general
# is used to create the surface
# it is an internal curve
c4 = line1(p3,p4,nelm= nelm4) # upper part of outflow boundary
c5 = line1(p4,p5,nelm= nelm5) # upper side of bounding box
c6 = line1(p5,p6,nelm= nelm6) # inflow boundary
c7 = line1(p6,p7,nelm= nelm7) # lower side of bounding box
c8 = line1(p7,p3,nelm= nelm8) # lower part of outflow boundary
# Contraction of curves
c9 = curves(c8,c4) # profile
c10= curves(c1,c3,c4,c5,c6,c7,c8,-c3,-c2) # Complete boundary for the
#surface generator
#
# surfaces
#
surfaces # See Users Manual Section 2.4
# linear triangles are used
s1 = general3(c10)
plot # make a plot of the mesh
end

```

Figure 2.6.13 shows the curves for this mesh with corresponding curve numbers.

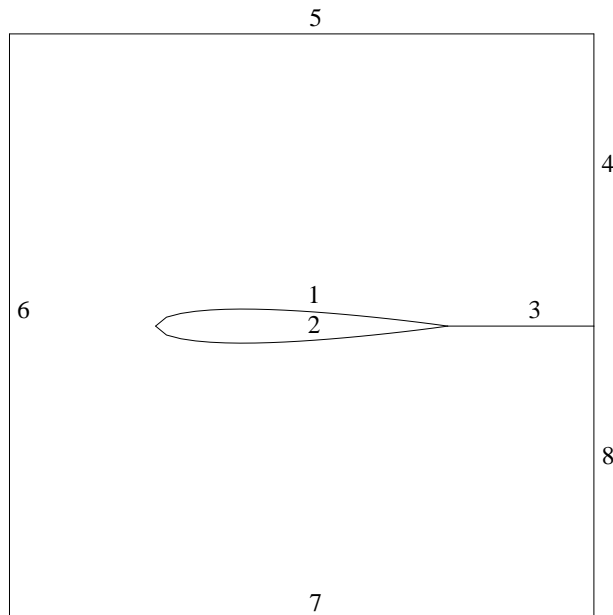


Figure 2.6.13: curves for naca0012 mesh

The mesh created is shown in Figure 2.6.14

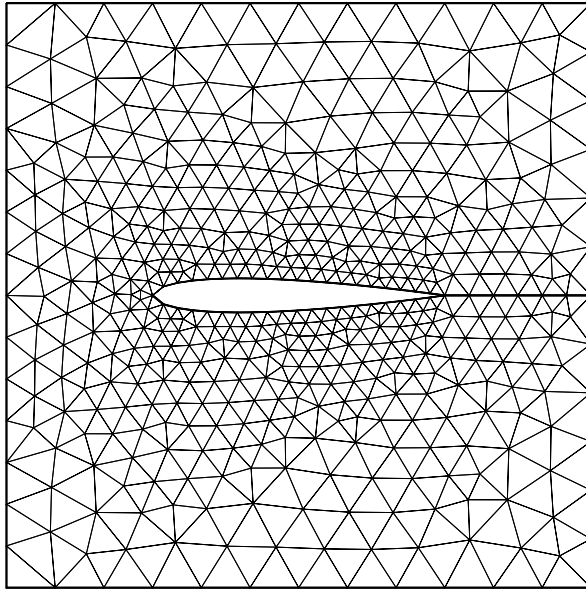


Figure 2.6.14: Plot of naca0012 mesh

Example 2.6.4 Sphere in a pipe

In this example we consider the situation of a sphere that is positioned within a circular pipe. In the sphere we have a different type of material than in the surrounding pipe. As a consequence 2 element groups must be used.

Consider a sphere with radius 1 and centre $(0,0,0)$. This sphere is surrounded by a circular pipe with radius 2 and height 4. The upper surface of the pipe is at $z=2$ and the lower surface at $z=-2$. Since the 3d generator `general` does not allow holes in the mesh, it is necessary to split the pipe into two parts. To that end the pipe is cut into two parts by the plane $z=0$.

Elements in the sphere are created by the submesh generator `GENERAL (3D)`.

The pipe parts can not be generated by the submesh generator `PIPE` since upper surface and lower surface of the pipe parts are different of shape. For that reason also the pipe is triangulated by `GENERAL`.

However, in the case of a long pipe, one might consider to split the pipe into four parts, two in the neighborhood of the sphere and two for the rest. The last other two might be triangulated by `PIPE`.

This example shows how to create a sphere and to fill it by tetrahedrons. Furthermore it shows how submesh generator `triangle` can be used in combination with holes in the surface.

In order to get this example into your local directory, use:

```
sepgetex sphere_in_pipe
```

The input file is given by:

```
#  
# sphere_in_pipe.msh  
# example of the use of the submesh generator sphere in combination with  
# other submesh generators
```



```

# plane
c2 = circle1 ( p1, p4, p3, nelm = Pnelm ) # Outer circle with centre p1
c3 = circle1 ( p5, p6, p7, nelm = Pnelm ) # Circle with centre p5
# Lower plane
c4 = circle1 ( p8, p9, p10, nelm = Pnelm ) # Circle with centre p8
# Upper plane
c5 = line1 ( p6, p4, nelm = PHnelm ) # Line from lower circle to
# outer circle in middle plane
c6 = line1 ( p9, p4, nelm = PHnelm ) # Line from upper circle to
# outer circle in middle plane

#
# surfaces
#
surfaces # See Users Manual Section 2.4
s1 = sphere 3 ( c1, subsurfaces(S2,S3) ) # sphere defined by the circle
# The upper half is called S2
# The lower half is called S3

s4 = triangle 3 ( c1, c2 ) # Part of disk outside the sphere in the
# middle plane
# Mark that this surface contains a hole
# and that the boundary consists of two
# closed parts

s5 = triangle 3 ( c3 ) # Lower disk
s6 = pipesurface 3 ( c3, c2, c5 ) # Lower half of pipe surface
s7 = surfaces ( s3, s4, s5, s6 ) # Lower half of pipe
s8 = triangle 3 ( c4 ) # Upper disk
s9 = pipesurface 3 ( c4, c2, c6 ) # Upper half of pipe surface
s10 = surfaces ( s2, s4, s8, s9 ) # Upper half of pipe

#
# volumes
#
volumes # See Users Manual Section 2.4
v1 = general11 ( s1 ) # sphere
v2 = general11 ( s7 ) # Lower half of pipe
v3 = general11 ( s10 ) # Upper half of pipe

#
# Connect elements to element groups
#
meshvolume # See Users Manual Section 2.1
velm1 = v1 # element group 1 is the sphere
velm2 = (v2,v3) # element group 2 is the pipe outside the sphere

plot, eyepoint(5,0,0) # make a plot of the mesh
# in order to make a 3d-hiddenline plot
# of the final mesh, eyepoint must be
# given
# See Users Manual Section 2.2

end

```

Figure 2.6.15 shows the curves for this mesh with corresponding curve numbers.

The mesh created is shown in Figure 2.6.16

Figure 2.6.17, shows the submesh (with hole) in the cutting plane, generated by triangle.

Figure 2.6.18, shows the submesh generated for the sphere.

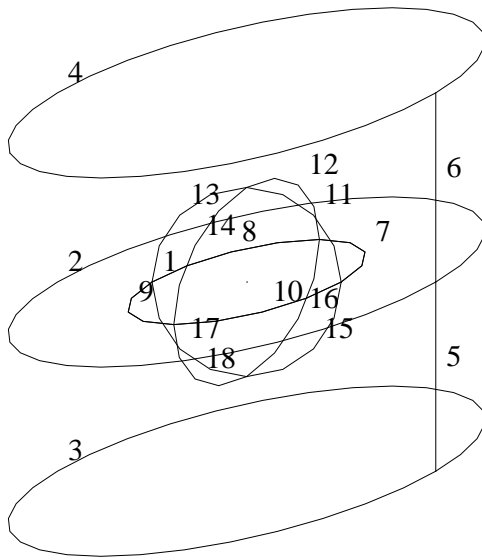


Figure 2.6.15: curves for sphere_in_pipe mesh

Figures 2.6.19 and 2.6.20, show the submeshes generated for the lower and the upper sphere.

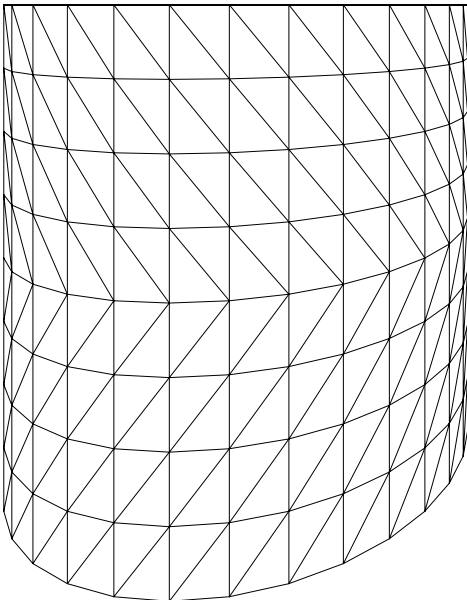


Figure 2.6.16: mesh for sphere_in_pipe mesh

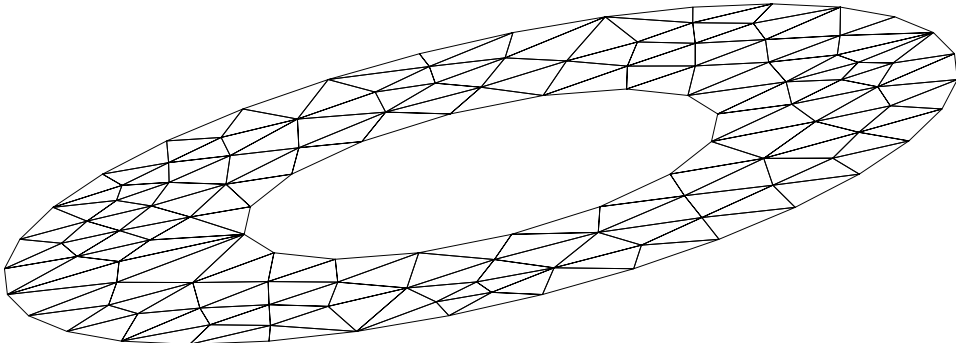


Figure 2.6.17: submesh in cutting plane for sphere_in_pipe mesh

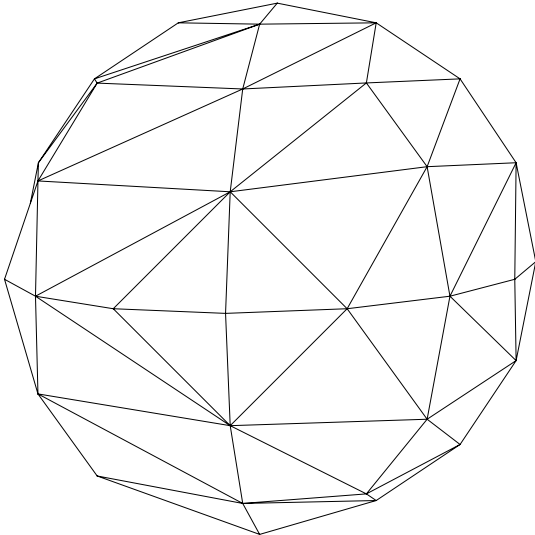


Figure 2.6.18: sphere submesh for sphere_in_pipe mesh

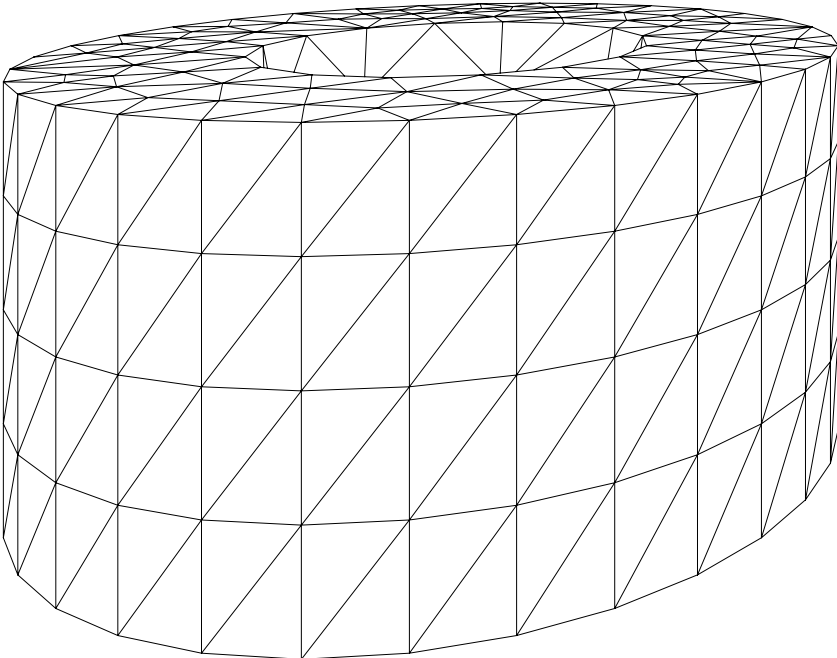


Figure 2.6.19: submesh of lower pipe for sphere_in_pipe mesh

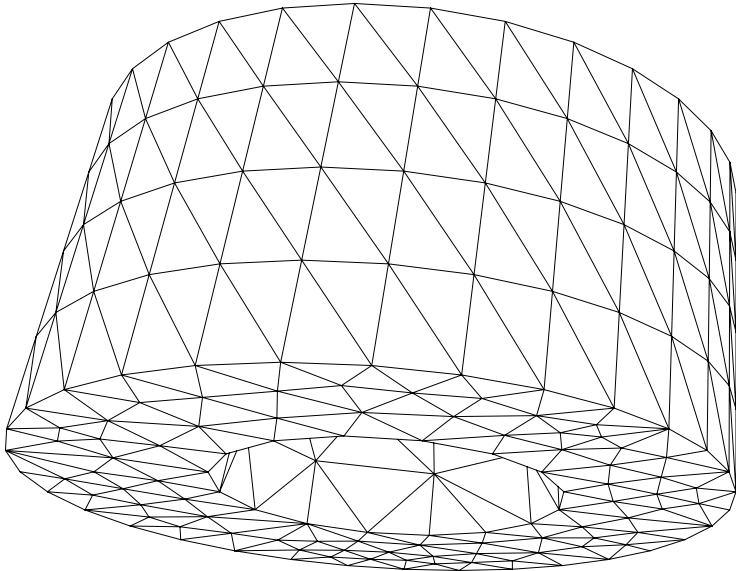


Figure 2.6.20: submesh of upper pipe for sphere_in_pipe mesh

2.7 Special input for program SEPMESH from the standard input file

In the case that the user gives the co-ordinates of all nodal points, and he wishes that SEPRAN generates a mesh based upon these points; the input for program SEPMESH is completely different from the standard input. In this section this special input is described.

The special input has the following structure:

```
MESHnD
  NODAL_POINTS, options
  BOUNDARY_POINTS
    co-ordinates
  INTERNAL_POINTS
    co-ordinates
  PLOT, options
END
```

COMMAND and DATA records.

The records must be given in the order as specified. An option is indicated like this: [option].

MESHnD (mandatory)

COMMAND record: opens the input for subroutine MESH, and defines the dimension of the space NDIM. (NDIM = n).

At his moment only $n = 2$ and $n = 3$ is allowed.

After this command it is necessary to give the command NODAL_POINTS instead of POINTS, indicating that the special mesh generation must be carried out instead of the standard one.

NODAL_POINTS, options (mandatory)

COMMAND record: defines the special situation. The options must be given at the same line, where the standard SEPRAN rule for continuation may be applied.

The following options are available:

```
UNSTRUCTURED
STRUCTURED ( NX=n, NY=m, NZ=k )
RECTANGULAR ( NX=n, NY=m, NZ=k, DX=dx, DY=dy, DZ=dz, ORIGIN = (Ox,Oy,Oz) )
ELEMENT_SHAPE = i
```

The options UNSTRUCTURED, STRUCTURED and RECTANGULAR are mutually exclusive. Meaning of the options:

UNSTRUCTURED indicates that the mesh to be created through all the nodal points is unstructured. The nodes may be positioned anywhere in the domain.

This is the default.

STRUCTURED indicates that the nodal points are positioned in a structured grid. Hence the nodal points may be mapped onto a rectangular (computational) grid. In this "computational" grid the number of points in the x-direction is given by $NX = n$, in the y-direction by $NY = m$, and in the z-direction by $NZ = k$. Hence exactly $n \times m$ (2D) or $n \times m \times k$ (3D) nodal points are present.

The default values for m, n and k are 1.

RECTANGULAR is a special case of structured. In this case not only the computational grid is rectangular, also the original grid. Hence it is not necessary for the user to give the co-ordinates of all points, but it suffices to give the origin and the spacing.

ORIGIN = (O_x, O_y, O_z) defines the origin and $DX = \Delta x$, $DY = \Delta y$, $DZ = \Delta z$ the spacing. The default origin is $(0, 0, 0)$ and the default spacing is $(1, 1, 1)$

ELEMENT_SHAPE = i defines the element shape to be used in the generation of the elements. These element shapes refer to Table 2.2.1 in Section 2.2. In 2D the shape numbers 3 to 6 may be used, in 3D the shape numbers 11 to 14.

The default shape number in 2D is 3 for an unstructured grid and 5 for a structured one. In 3D the default shape number is 11 for an unstructured grid and 13 for a structured one.

BOUNDARY_POINTS In the case of an unstructured 2D grid, the user has the option to define the boundary of the region explicitly. This is done by the command **BOUNDARY_POINTS** followed by data records containing the co-ordinates of the boundary points. The sequence of these co-ordinates define the sequence of the nodes along the boundary. The node numbers are set to 1, 2, .. *number_of_boundary_points*.

If the option **BOUNDARY_POINTS** is not present, SEPRAN defines the boundary itself assuming a convex region.

The co-ordinates of the boundary points must be given on the lines following the command **BOUNDARY_POINTS**.

Of course the sequence is always $x_1, y_1, z_1, x_2, y_2, z_2, \dots$. In R^2 the z-coordinate must not be given. These co-ordinates must be given as numbers defined in the standard SEPRAN sense. Newline, parenthesis and commas are used as separator between two numbers. A text indicates the end of the input. This text is interpreted as a keyword, hence it is not allowed to put any text between the numbers. Of course comment is allowed in the usual way.

INTERNAL_POINTS defines the rest of the co-ordinates in the mesh. Also in this case the sequence of the input defines the sequence of the nodes. The boundary nodes given before are always used as first nodes. Mark that it is necessary to give the boundary points first and then the internal nodes.

The same rules with respect to the input, as for the boundary points are valid in this case.

PLOT (options) indicates that the user wants a plot of the mesh. The following options (all in one line) are available:

```

LENGTH = l
SCALE = s
YFACT = y
JMARK = j
ROTATE = r
SUPPRESS = su
EYEPOINT = (x_e, y_e, z_e)
ORIENTATION = i

```

The meaning of these options is exactly the same as those given in Section 2.2.

3 The computational part of SEPRAN

3.1 Introduction

In the computational part of SEPRAN the solution is computed. At this moment there are two main programs for the computational part available: SEPCOMP and SEPFREE.

Program SEPCOMP is meant for the standard problems. It assumes that a mesh has been made before and computes the solution corresponding to that mesh. The output is written to the file sepcomp.out so that postprocessing may be performed by program SEPPOST.

Program SEPFREE is developed for free surface and moving boundary problems. For these problems the boundary of the mesh may change in each step of the process. As a consequence it is necessary that SEPFREE creates its mesh itself and does not use the output of program SEPMESH. Besides that, SEPFREE has exactly the same possibilities as SEPCOMP, extended with some extra options that are meant for adapting the boundary and the mesh for free surface problems.

In the introduction it has been described how the program SEPCOMP may be used for simple linear or non linear problems. In this chapter the complete possibilities of SEPCOMP are described. Should SEPCOMP be insufficient for your purposes, then it is necessary to consult the SEPRAN PROGRAMMERS GUIDE in order to construct your own main program using the tools SEPRAN provides.

How to call program SEPCOMP, or how to link your program if it is a version of SEPCOMP extended with your own subroutines, is already described in the SEPRAN INTRODUCTION Sections 3.2 and 5.2.

Furthermore we refer to Section 5.3 of the SEPRAN INTRODUCTION with respect to some programming considerations.

In Section 3.2 the complete input for program SEPCOMP is described.

Program SEPFREE may be used in exactly the same way as program SEPCOMP. An important difference is of course that the body of SEPFREE is another one than that of SEPCOMP.

The body of SEPCOMP is called startsepcomp, that of SEPFREE is called startsepfree. So when a user needs function subroutines he has to make his own main program in the same way as described in Section 5.2 of the INTRODUCTION, however, with sepcom replaced by freebsub anywhere in the text.

The complete input description of program SEPFREE or actually subroutine startsepfree is given in Section 3.4.

3.2 Description of the input for program SEPCOMP

The input for program SEPCOMP is subdivided into a number of blocks. Some of these blocks must be given in a fixed sequence; all others are free. Each block starts with a specific main keyword and ends with the keyword END. Unless stated otherwise all commands in a block must be given on a new line. It is advised to indent the input between main keyword and the keyword END to make the block more visible. The same is advised for subblocks. The end of the input is indicated by the physical end of file or by the keyword END_OF_SEPRAN_INPUT. This last keyword may be necessary if the user reads his own input in the standard SEPRAN input file.

SEPCOMP starts with reading all SEPRAN input before carrying out the necessary computations. In this way input errors are checked immediately. The present version of SEPCOMP recognizes the following blocks at least the following main keywords indicating the beginning of a block:

- START
- PROBLEM
- STRUCTURE
- MATRIX
- ESSENTIAL BOUNDARY CONDITIONS (3 keywords)
- COEFFICIENTS
- CHANGE COEFFICIENTS (2 keywords)
- SOLVE
- NONLINEAR_EQUATIONS
- TIME_INTEGRATION
- CREATE
- DERIVATIVES
- INTEGRALS
- BOUNDARY_INTEGRAL
- OUTPUT

If the block START is used, it must always be the first block. This block is, however, optional.

The block PROBLEM must be given as first or second block, it may only be preceded by the block START.

All other blocks may be given in any sequence. The information of a block, however, must always be positioned between the main keyword and the keyword END. The block PROBLEM is mandatory, all other blocks are optional.

If no input for a block is given default values are used.

Except the blocks START, PROBLEM and STRUCTURE all blocks may be used more than once. This makes, however, only sense if these multiple inputs can also be addressed. Usually this is the case in combination with STRUCTURE, but also in the case of NONLINEAR EQUATIONS it is possible to address more inputs.

In order to distinguish between the various inputs for one type of block, each block of a kind is provided by a sequence number. This sequence number may be explicitly given by the user by providing the main keyword by the option SEQUENCE_NUMBER = *i*. If the user does not give the sequence number the sequence number is computed implicitly by the sequence the input is read.

The sequence number given is always the next one compared to a previous one given for that type of block. If no previous one consists the sequence number is automatically equal to 1.

Hence, if the following blocks are given in that sequence:

```
Block 1    COEFFICIENTS
           END
Block 2    COEFFICIENTS, SEQUENCE_NUMBER = 5
           END
Block 3    INTEGRALS, SEQUENCE_NUMBER = 2
           END
Block 4    COEFFICIENTS
           END
Block 5    INTEGRALS
           END
```

then block 1 is of type COEFFICIENTS and gets sequence number 1, block 2 is of the same type with sequence number 5 and block 4 with sequence number 6. Block 3 is of type INTEGRALS and gets sequence number 2, block 5 is of the same type and gets sequence number 3.

It is advised to use the implicit way of numbering only in the case that there is only one block of the specific type. Explicit numbering avoids unwanted effects because of miscalculation.

The main blocks have the following meaning:

START In this block some information about defaults to be used may be given. In fact START is used to change defaults of SEPRAN for example the name of files and so on.

PROBLEM Defines the type of problems to be solved, i.e. the type of differential equations to be solved, the type of boundary conditions, at which boundaries these boundary conditions are given etcetera PROBLEM only defines types not values. So in the part PROBLEM it is fixed at which boundaries essential boundary conditions must be prescribed, but not what the values of these boundary conditions are.

STRUCTURE The part STRUCTURE is only necessary if the user wants to perform more actions than provided by the standard possibilities. For example if the user wants to solve a linear or non-linear problem, without any extras there is no need to define the block STRUCTURE.

However, if the standard possibilities do not suffice and the user wants to define his own structure of the main program then this block should be used. Another reason why the block STRUCTURE may be used is that it allows the user to define extra output.

In fact the block structure may be seen as a very simple way of defining your own main program.

If the block STRUCTURE is not in the input file, SEPCOMP checks if he finds the block NONLINEAR_EQUATIONS. If so he expects to solve a non-linear problem, otherwise a standard linear problem is solved.

MATRIX Defines the type of storage to be used for the large matrix. In this part it is given whether the large matrix is symmetrical, complex, etcetera. But also the user defines whether the storage scheme corresponds to a direct method or a compact method. Implicitly this defines the type of solver that will be used to solve the systems of linear equations. If a direct storage is used, a profile solver will be called (direct method), if a compact storage is used, the linear system is solved by an iterative method.

If the part **MATRIX** is skipped it is assumed that the matrix is real, non-symmetric and that a direct method is used.

ESSENTIAL BOUNDARY CONDITIONS Defines the values of the essential boundary conditions. It is only necessary to define the non-zero essential boundary conditions, all other essential boundary conditions are made equal to zero automatically.

If the part **ESSENTIAL BOUNDARY CONDITIONS** is skipped all essential boundary conditions are set equal to zero.

COEFFICIENTS Defines the values of the coefficients in the partial differential equations and the natural boundary conditions. In some cases this part defines also extra information of how to compute matrices and vectors. For example this part may be used to define the numerical integration rule, the type of co-ordinate system to be used and so on.

The part **COEFFICIENTS** is mandatory if standard elements as described in the manual **STANDARD PROBLEMS** are used. It may only be skipped if all element subroutines are written by the user.

CHANGE COEFFICIENTS Makes only sense if a non-linear problem is used. In that case it is possible to change some of the coefficients during the iteration process. This may be for example useful if after some iterations the user wants to change to another iteration scheme, but also if he wants to use a kind of continuation process in which a specific parameter is increased or decreased during the iteration process in order to get a better convergence.

Whether the part **CHANGE COEFFICIENTS** must be used, depends on the part **NONLINEAR EQUATIONS**.

SOLVE Gives information with respect to the linear solver to be used. For example in the case of a direct method, it is possible to tell the solver that the matrix is positive definite. In the case of an iterative solver, the user may give extra information about the type of linear solver etcetera

If the part **SOLVE** is skipped, the default values are used. This means that in the case of a storage scheme corresponding to a direct solver, a profile method is used and it is not assumed that the matrix is positive definite. In the case of an iterative solver this means that the default iterative solver, with the default accuracy and the default set of parameters is used.

NONLINEAR EQUATIONS Indicates that the partial differential equation to be solved is stationary and non-linear. In that case an iterative procedure is necessary to solve the non-linear problem. In each step of the non-linear iteration a linear system of equations is solved. In this part the user gives some information about the iteration process.

If the keyword **NONLINEAR EQUATIONS** is skipped it is assumed that the partial differential equation to be solved is linear and no iteration is carried out.

TIME INTEGRATION Indicates that a time-dependent problem must be solved. In this case a time integration method is applied and the solution is computed during a number of time-steps from initial time to end time. If the keyword **TIME INTEGRATION** is found and not the keywords **STRUCTURE** or **NONLINEAR EQUATIONS** automatically the time dependent problem will be solved. If the keyword **NONLINEAR EQUATIONS** is present and also the keyword **TIME INTEGRATION** without the keyword **STRUCTURE** the input for **TIME INTEGRATION** is read, but no action is taken.

CREATE Indicates that a vector must be created. This vector may be used for example as starting value of an iteration process, but also as vector to be used to compute coefficients for

the differential equations. Another application of CREATE is that it can be used to define essential boundary conditions for the solution. In fact the body of the subroutine activated by CREATE is the same as the one activated by ESSENTIAL BOUNDARY CONDITIONS.

CREATE is only activated if the block STRUCTURE is used. Otherwise it is read but no action is taken.

DERIVATIVES This block is used to define derived quantities. DERIVATIVES is only activated if the block STRUCTURE is used. Otherwise it is read but no action is taken.

INTEGRALS This block is used to define integrals to be computed. INTEGRALS is only activated if the block STRUCTURE is used. Otherwise it is read but no action is taken.

BOUNDARY INTEGRALS This block is used to define boundary integrals to be computed. BOUNDARY_INTEGRALS is only activated if the block STRUCTURE is used. Otherwise it is read but no action is taken.

OUTPUT Defines which output is written to the file sepcomp.out. This output may be used in the post-processing part of SEPRAN.

If the keyword OUTPUT is skipped only the computed solution is written to the output file. Otherwise it is also possible to compute derivatives or other derived quantities and to write these to the file sepcomp.out

SEPCOMP may compute both vectors and scalars. At this moment each vector and each scalar is distinguished by a sequence number. The number of available vectors and scalar is limited. At this moment this limit is set equal to 25, which means that only vectors and scalars with sequence numbers 1 to 25 may be used. Of course it is possible to overwrite the vectors and scalars during the computation process.

In the next subsections the input of each of the blocks is described.

Remark: at this moment program SEPCOMP is not yet provided with time-dependent options. If you want to solve a time-dependent problem it is necessary to use the more complicated way of using SEPRAN, i.e. you have to write the main program yourself.

3.2.1 The main keyword START

The block defined by the main keyword START may be used to influence some default parameters in SEPCOMP but also in SEPPOST and programs written by the user itself. If this part is omitted, the standard defaults are used.

The block defined by the main keyword START has the following structure (options are indicated between the square brackets "[" and "] "):

```
START (optional)
  DATABASE = d
  ROTATE
  NOROTATE
  RENUMBER r
  SEPCOMP = s
  NAME_BACK = 'name_backing_storage_file'
  NAME_PLOT = 'name_plot_files'
  NAME_SEPINF = 'name_sepcomp.inf'
  NAME_SEPOUT = 'name_sepcomp.out'
  NAME_MESH = 'name_mesh_file'
  NAME_SEPCOMP_IN = 'name_sepcomp.in'
  MAXPLOTS = m
  ITIMEFIRST = i
  ITIMELAST = i
  CPU_TIME
  WALL_CLOCK_TIME
  NO_WRITE_SEP
  INPUT_SEPCOMP_OUT
END
```

The sequence of the keywords between START and END is arbitrary, necessary is that they start with START and end with END.

Meaning of the keywords:

START This keyword is optional. It must be followed by subkeywords indicating which default setups must be changed.

DATABASE = d Indicates whether the permanent file 2 is used or created.

Possible values for *d* are

not File 2 is not used.

new File 2 is used, all preceding information on this file is destroyed, or the file is a new one.

old File 2 is used, all preceding information on this file is kept. This possibility may only be used if file 2 has been created before by a SEPRAN program.

Default: DATABASE = not

NOROTATE means that PLOTS are not rotated.

Default: depending on the size of the picture.

ROTATE means that the plots are rotated over an angle of 90°.

The options ROTATE and NOROTATE are mutually exclusive.

Default: depending on the size of the picture.

Remark: in general SEPCOMP does not produce pictures. However, in the case of non-linear equations it is possible to trace the iteration process with some simple pictures. These pictures are effected by the options ROTATE/NOROTATE. Furthermore START may also be used in SEPPOST in which case the option ROTATE/NOROTATE may be useful.

RENUMBER r Indicates the type of renumbering to be used for the mesh. Possible values for r are:

Cuthill band
 Cuthill profile
 Cuthill always
 Sloan band
 Sloan profile
 Sloan always
 Best band
 Best profile
 not

not indicates that no renumbering is performed. Remember that SEPMESH does not renumber the nodes.

Cuthill band/profile/always means that the Cuthill-McKee renumbering is used. The renumbered sequence is compared with the original numbering.

If profile is used the best of the original numbering and the Cuthill-McKee is used with respect to the size of the profile.

Band refers to the optimal band width, and

always means that no comparison is made, but that always the Cuthill-McKee numbering is used.

Sloan band/profile/always has the same meaning, however, with respect to the Sloan renumbering.

Best has the same effects, but uses both Cuthill-McKee and Sloan to compare with the original numbering in order to find an optimal choice.

Default: Sloan profile

Remark: in general renumbering is essential in order to decrease the computation time of the linear solver except in some exceptional cases as for example a rectangular mesh numbered in smallest direction. In the case of iterative methods renumbering is less important than for direct methods but still a gain a computation time may be possible.

SEPCOMP = s indicates how the file sepcomp.out should be used.

Possible values for s are:

not
 formatted
 unformatted

not sepcomp.out is not used.

formatted both files are used as formatted files.

unformatted sepcomp.out is treated as an unformatted file.

Default: unformatted

NAME_BACK = '*name_backing_storage_file*' defines the name of the backing storage file (file 2). This file gets the name **name_backing_storage_file**.

Do not forget the quotes " around the file name.

Default name: the name stored in the file sepran.env (usually **sepranback**).

NAME_PLOT = '*name_plot_files*' defines the names of the default plot files. These files get the name **name_plot_files.001**, **name_plot_files.002**,...

Do not forget the quotes " around the file name.

Default name: the name stored in the file sepran.env (usually **sepplot**).

NAME_SEPINF = *'name_sepcomp.inf'* defines the name of the so-called sepcomp.inf file (file 73). This file gets the name **name_sepcomp.inf**.

Do not forget the quotes " around the file name.

Default name: the name stored in the file sepran.env (usually **sepcomp.inf**).

NAME_SEPOUT = *'name_sepcomp.out'* defines the name of the so-called sepcomp.out file (file 74). This file gets the name **name_sepcomp.out**.

Do not forget the quotes " around the file name.

Default name: the name stored in the file sepran.env (usually **sepcomp.out**).

NAME_MESH = *'name_mesh_file'* defines the name of the mesh output file (file 10). This file gets the name **name_mesh_file**.

Do not forget the quotes " around the file name.

Default name: the name stored in the file sepran.env (usually **meshoutput**).

NAME_SEPCOMP_IN = *'name_sepcomp.in'* defines the name of the **sepcomp.out** file to be used for input.

If also output is written to a sepcomp.out file, both names must be different.

Default name: **sepcomp.in**

NO_WRITE_SEP indicates that no output is written to the **sepcomp.out** file. So this file is not opened.

INPUT_SEPCOMP_OUT indicates that the file indicated by **name_sepcomp_in** (or the default name) is used as input file. This is meant for example to do some postprocessing or for example for a restart.

If this keyword is found, the problem definition is read from the sepcomp.out file, and there should be no input block **PROBLEM**.

It also reads all vector names that have been defined in the sepcomp.out file.

The keyword itself does not activate the reading of the solution vectors. To do that you need the keyword **READ_SOLUTIONS** in the **STRUCTURE** block.

MAXPLOTS = **m** defines the number of plot files that may be created.

At this moment only two values are allowed: $m = 1000$ (default) or $m = 10000$. In the last case a plot file uses a number with 4 digits instead of 3.

ITIMEFIRST = **i** is only used in the program SEPPOST.

If **ITIMEFIRST** > 1, the first **ITIMEFIRST**-1 solutions in the file **sepcomp.out** are skipped and not stored.

ITIMELAST = **i** is only used in the program SEPPOST.

It defines the last solution that is read for postprocessing. All the rest of the arrays are skipped.

CPU_TIME Forces the printing of time in terms of CPU time. This is the default, except for parallel computing.

WALL_CLOCK_TIME Forces the printing of time in terms of wall clock time. This is the default in case of parallel computing.

END (mandatory). Indicates the end of the input block.

3.2.2 The main keyword PROBLEM

The block defined by the main keyword PROBLEM defines which problem is to be solved by program SEPCOMP. For each element group defined in SEPMESH the user must indicate what type of problem has to be solved. Problems are indicated by so-called type numbers. Each type number corresponds uniquely to a type of partial differential equations. To know which type number corresponds to a specific partial differential equation it is necessary to consult the manual STANDARD PROBLEMS.

SEPRAN also allows for the definition of your own elements. For that reason the group of element numbers between 1 and 99 is strictly reserved for user defined elements, whereas type numbers larger than 99 correspond to SEPRAN standard elements. Type numbers smaller than 1 have a special meaning.

For each differential equation it is necessary to give boundary conditions. SEPRAN distinguishes between so-called essential boundary conditions and natural boundary conditions. An essential boundary condition is a boundary condition that prescribes unknowns at the boundary explicitly, natural boundary conditions in general give some information about derivatives or combinations of unknowns and derivatives at the boundary. Which type of boundary conditions are essential and which are natural for a specific partial differential equation, can be found in the manual STANDARD PROBLEMS.

Natural boundary conditions require extra elements, the so-called boundary elements. These elements may be defined in the mesh generator as line elements (R^2) or surface elements (R^3), but they may also be defined in the part PROBLEM as boundary elements. The essential difference between line elements defined in the mesh generator and boundary elements, is that line elements are always used in the post-processing part as separate elements, whereas boundary elements are skipped. With post-processing also computation of derivatives and integrals is meant. In general there is no reason why boundary conditions have anything to do with post-processing, so it is advised, if possible, to use boundary elements instead of line elements. Besides that the use of line elements in post-processing makes things more complicated and may even result in unwanted error messages.

The block defined by the main keyword PROBLEM has the following structure:

```
PROBLEM [,SEQUENCE_NUMBER = s]
  TYPES
    data corresponding to TYPES
  NATBOUNCOND
    data corresponding to NATBOUNCOND
  BOUNELEMENTS
    data corresponding to BOUNELEMENTS
  ESSBOUNDCOND
    data corresponding to ESSBOUNDCOND
  PERIODICAL_BOUNDARY_CONDITIONS
    data corresponding to PERIODICAL_BOUNDARY_CONDITIONS
  LOCALTRANSFORM
    data corresponding to LOCALTRANSFORM
  UNKNOWNCONSTANT
    data corresponding to UNKNOWNCONSTANT
  GLOBAL_UNKNOWNNS
    data corresponding to GLOBAL_UNKNOWNNS
  GLOBAL_ELEMENTS
    data corresponding to GLOBAL_ELEMENTS
  GLOBAL_RENUMBERING
    data corresponding to GLOBAL_RENUMBERING
  FICTITIOUS_UNKNOWNNS
```

```

    data corresponding to FICTITIOUS_UNKNOWNNS
FICTITIOUS_ELEMENTS
    data corresponding to FICTITIOUS_ELEMENTS
SKIP_ELEMENTS
    data corresponding to SKIP_ELEMENTS
NUM_LEVELSET = i
LEVELSET i, data
REORDER
    data corresponding to REORDER
PRINT_LEVEL = k
END

```

The keywords PROBLEM, END and TYPES are mandatory. All subkeywords may be given in arbitrary order as long as they appear only once. The data corresponding to these subkeywords must be given immediately after the keywords themselves.

The sequence number is optional. It has only effect if more than one input block problem is found in the input file.

The first input block problem must always be read as first input block, following the constants part. The other input blocks problem may be put at any place in the input block before the `end_of_sepran_input` keyword.

In first instance the first input block read is made active.

To go to another input block you have to give the command `NEW_PROBLEM_DESCRIPTION` in the structure block [3.2.3.20](#).

If the keyword `NATBOUNCOND` is given then also the keyword `BOUNELEMENTS` must be present and `NATBOUNCOND` must always precede `BOUNELEMENTS`.

If the keyword `GLOBAL_UNKNOWNNS` is given then also the keyword `GLOBAL_ELEMENTS` must be present and `GLOBAL_UNKNOWNNS` must always precede `GLOBAL_ELEMENTS`.

If the keyword `GLOBAL_RENUMBERING` is given then also the keyword `GLOBAL_UNKNOWNNS` must be present and `GLOBAL_UNKNOWNNS` must always precede `GLOBAL_RENUMBERING`.

If the keyword `FICTITIOUS_UNKNOWNNS` is given then also the keyword `FICTITIOUS_ELEMENTS` must be present and `FICTITIOUS_UNKNOWNNS` must always precede `FICTITIOUS_ELEMENTS`.

A special possibility is the use of the keyword PROBLEM followed by DEFAULT:

```
PROBLEM DEFAULT
```

This possibility is actually meant for the special case that the user does not want to compute a solution by SEPRAN, but instead provides the solution in some other way, for example by reading it from a file. (See the input block "STRUCTURE" for this possibility). Once the keyword DEFAULT is found immediately behind the keyword PROBLEM, no extra input with respect to the input block "PROBLEM" is expected. In this case the default problem description is assumed. This implies that for all element groups elements of type 800 are used (general second order elliptic equation with one unknown per point). Furthermore no boundary conditions are present, i.e. neither essential nor natural. None of the special possibilities in this section can be applied.

Explanation of the subkeywords and description of the data records (options are indicated between the square brackets "[" and "]"):

PROBLEM (mandatory)

COMMAND record: opens the input this block.

TYPES (mandatory)

defines the problem definition numbers of the standard elements.

See [\(3.2.2.1\)](#).

NATBOUNCOND (optional)

indicates that standard boundary elements are used.

See [\(3.2.2.2\)](#). If used, it must be followed by:

BOUNELEMENTS (must only be used when NATBOUNCOND is used)

indicates that boundary elements are created.

See (3.2.2.3).

PERIODICAL_BOUNDARY_CONDITIONS (optional)

indicates that periodical boundary conditions will be prescribed and where.

See (3.2.2.15).

ESSBOUNCOND (optional)

indicates that essential boundary conditions will be prescribed and where.

See (3.2.2.4).

LOCALTRANSFORM (optional)

indicates that local transformations must be defined in the points created on the curves and surfaces defined by the data records.

See (3.2.2.5).

UNKNOWNCONSTANT (optional)

COMMAND record: indicates that along one or more parts of the boundary we have the boundary condition u equals unknown constant.

See (3.2.2.6).

GLOBAL_UNKNOWNNS This command is used when the user wants to define special unknowns that are not coupled to specific nodal points but have a more global character.

See (3.2.2.7).

GLOBAL_ELEMENTS must be used to specify the region to which the global unknown is coupled. The corresponding elements are used to compute the extra rows and columns in the matrix and right-hand side.

See (3.2.2.8).

GLOBAL_RENUMBERING If global unknowns are introduced one usually uses less boundary conditions than in the case without global unknowns.

See (3.2.2.9).

FICTITIOUS_UNKNOWNNS This command is meant for the use of the fictitious domain method. This is a special free surface method.

See (3.2.2.10).

FICTITIOUS_ELEMENTS must be used to specify the region to which the fictitious unknown is coupled.

See (3.2.2.11).

SKIP_ELEMENTS This record indicates that certain elements as indicated by the data records must be skipped while creating the large matrix and vector.

See (3.2.2.12).

REORDER [LEVELS] $i1, i2, (i3, i4, i5), i6$ (optional)

With this command the user can influence the internal numbering of the unknowns.

See (3.2.2.13).

PRINT_LEVEL = k (optional)

Defines the amount of extra information that is printed.

This option is only meant for debugging purposes.

The following values of k are permitted:

- 0 (Default) No extra information besides the standard is printed.
- 1 A selected part of the sub arrays of KPROB are printed.
- 2 The complete array KPROB with all its sub parts are printed.

If omitted $k=0$ is assumed.

NUM_LEVELSET = k (optional)

Defines the number of level set functions, see (3.2.2.14).

LEVELSET = k (optional)

Defines information about the level set, see (3.2.2.14).

END (mandatory)

end of input for the input block "PROBLEM".

In the case of coupled problems to be solved uncoupled as described in Section 1.3 the user must define more problems. In that case the input in the input block "PROBLEM" is as follows:

```
PROBLEM 1, sequence_number = 1
  TYPES
  .
  .
  .
PROBLEM 2
  TYPES
  .
  .
  .
PROBLEM 3
  .
  .
  .
END
```

PROBLEM i refers to the i^{th} problem. Only one END record must be used; this record closes the input of the input block "PROBLEM".

In case of more problem descriptions, for example to change the type of boundary conditions, the input would look like:

```
PROBLEM 1, sequence_number = 1
  TYPES
  .
  .
  .
PROBLEM 2
  TYPES
  .
  .
  .
PROBLEM 3
  .
  .
  .
END

PROBLEM 1, sequence_number = 2
  TYPES
  .
  .
  .
PROBLEM 2
  TYPES
  .
  .
  .
PROBLEM 3
  .
  .
  .
END
```

·
·
·

Warning in case of user built SEPRAN programs

If Local Transformations are used, the user must be very careful with respect to boundary conditions and solution. SEPRAN supposes always that boundary conditions are filled for the transformed degrees of freedom (for example normal and/or tangential velocity). The solution subroutines, however, transform the solution BACK, and hence these boundary conditions into the original form (for example Cartesian velocities). As a consequence, if the computed solution must be used in a new call of one of the matrix building subroutines, and if this solution must contain the boundary conditions, then either the boundary conditions must be refilled, or the non-transformed array should be used. This concerns primarily array ISOL in the call of the matrix builder. In almost all applications, however, array ISLOLD in these subroutines should be available in original (i.e. back-transformed) form. The linear solver provides the possibility to suppress the back-transformation, or to carry out the back-transformation only.

3.2.2.1 The subkeyword TYPES

The subkeyword TYPES defines the problem definition numbers of the standard elements. Must be followed by data records of the type:

```
ELGRP 1 = (type = n1 [, n2, n3, . . . ])
ELGRP 2 = (type = n1 [, n2, n3, . . . ])
ELGRP i = (type = n1 [, n2, n3, . . . ])
```

with i the element group number; exactly NELGRP (is number of element groups) data records are necessary.

Instead of ELGRP i one may also use ELGRP i to j if all element groups i to j have exactly the same type number(s).

$n1$ is the problem definition number of the i^{th} element group.

The element group number refers to the element group number defined in the mesh generation part. The number of element groups to be defined in this part TYPES must be exactly equal to the number of element groups defined in the mesh generation.

The type numbers $n1, n2, \dots$ define which types of problems are to be solved. These type numbers have the following meaning:

- 1 Type number for periodical boundary conditions. See also Section 1.2.3. This type number may only be used for elements defined by MESHCONNECT as described in Section 2.2.
- 0 Type number 0 means that all elements for this group must be skipped. For all these elements the number of unknowns per point is equal to zero. This type number is especially meant for test procedures, in which case it may be useful to skip element groups not yet programmed.
- 1-99 When the user wants to define his own problems (standard elements), he has to use problem definition numbers between 1 and 99. SEPRAN does not distinguish between the various type numbers in the range 1 to 99. However, these type numbers are submitted to the element subroutines as parameter ITYPE in common block CACTL (See Chapter 4) and may be used to distinguish between various types of differential equations or boundary conditions.
- >99 corresponds to the standard problems provided by SEPRAN. For a description which type numbers correspond to what problems, the user is referred to the manual Standard Problems. For example type number 800 refers to potential problems and type number 900 to incompressible Navier-Stokes.

When the user wants to use other problem definition numbers to construct the large matrix and vector, however, with exactly the same mesh and the same type of boundary conditions, alternative problem definition numbers may be given ($n2, n3, \dots$). This is for example the case when the user wants to build a stiffness matrix and a mass matrix. The problem definition numbers $n1$ are used throughout the program until subroutine CHTYPE is used to change to another group of problem definition numbers. See the Programmers Guide for details. At this moment SEPCOMP does not offer this possibility.

When all problem definition numbers $n1, n2, \dots$ of an element group are in the range 1 to 100, the data record ELGRP i , (type = . . .) may be followed by the next data record:

```
NUMDEGFD = ( d1, d2, . . . , dn )
```

with d_i the number of degrees of freedom in the i^{th} nodal point of the standard element. When only $d1$ is given, then the number of degrees of freedom in each nodal point of the standard element is equal to $d1$.

If this record is omitted it is assumed that all nodes contain exactly one degree of freedom.

When vectors of special structure are used (see Section 1.1), the next data records must be given:

$$\begin{aligned} \text{VEC1} &= (d1, d2, . . . , dn) \\ \text{VEC2} &= (d1, d2, . . . , dn) \\ &\vdots \\ \text{VECi} &= (d1, d2, . . . , dn) \end{aligned}$$

where VEC_i corresponds to the i^{th} vector of special structure. d_j gives the number of degrees of freedom in the j^{th} nodal point of the standard element, corresponding to the i^{th} vector of special structure. When only d_1 is given, the number of degrees of freedom in each nodal point of the standard element, corresponding to the i^{th} vector of special structure, is equal to d_1 .

In some cases the i^{th} degree of freedom in a nodal point has a different physical meaning for different nodes. Consider for example the element in Figure 3.2.2.1, with degrees of freedom ψ , u and v in the vertices and u_t in the mid-side points.

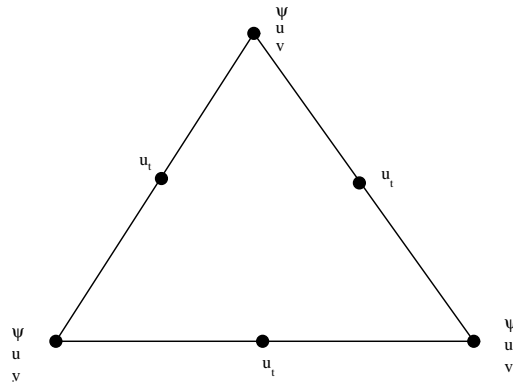


Figure 3.2.2.1: Example of element with different physical meaning of first degree of freedom in different nodes.

In such a case we wish to distinguish between the first degree of freedom ψ in the vertices and the first degree of freedom u_t in the mid-side points. For such problems the next data records must be given to couple a physical unknown with a specific degree of freedom

$$\begin{aligned} \text{NUMBER 1} &= (d1, d2, . . . , dn) \\ \text{NUMBER 2} &= (d1, d2, . . . , dn) \\ &\vdots \\ \text{NUMBER i} &= (d1, d2, . . . , dn) \end{aligned}$$

where $\text{NUMBER } i$ corresponds to the i^{th} physical unknown in local nodal point j . When $d_j = 0$, it means that the physical unknown i is not present in node j .

If for example we couple the physical unknown ψ in Figure 3.2.2.1 with number 1, u with number 2, v with number 3 and u_t with number 4 then the following records are necessary:

$$\begin{aligned} \text{NUMBER 1} &= (1, 0, 1, 0, 1, 0) \\ \text{NUMBER 2} &= (2, 0, 2, 0, 2, 0) \\ \text{NUMBER 3} &= (3, 0, 3, 0, 3, 0) \\ \text{NUMBER 4} &= (0, 1, 0, 1, 0, 1) \end{aligned}$$

Where, in a subroutine a specific degree of freedom is asked, the number corresponding to the physical unknown is meant. So if we prescribe the degree of freedom 1 in the input block "ESSENTIAL BOUNDARY CONDITIONS" or "CREATE", for this example only ψ is prescribed.

These records are not necessary if the i^{th} degree of freedom in a node always corresponds to the

i^{th} physical unknown, even if a nodal point does not contain i degrees of freedom. So in most applications there is no need to give the data records with NUMBER = ...

In some cases the user might want to extend the number of vectors of special structure in case of standard elements (type number ≥ 100). This is possible by replacing, the first type number ITYPE for the element groups it concerns, by ITYPE+10000×IEXTRA, where IEXTRA denotes the extra number of vectors of special structure, the user wants to add. These vectors of special structure are only coupled to nodes, never to elements.

For each of the IEXTRA vectors a record must be supplied immediately after the ELGRP $i = (\text{type} = \dots \text{record}$. These records must have the following shape:

```
n_vertex = n1, n_midside_points = n2, n_centroid = n3, n_face = n4
```

n_vertex = $n1$ defines the number of degrees of freedom in the vertices of the elements corresponding to the element group. If omitted $n1 = 0$ is used.

n_midside_points = $n2$ defines the number of degrees of freedom in the midside points of the elements corresponding to the element group. If omitted $n2 = 0$ is used.

n_centroid = $n3$ defines the number of degrees of freedom in the centroid of the elements corresponding to the element group. If omitted $n3 = 0$ is used.

n_face = $n4$ defines the number of degrees of freedom in the centroids of the faces of the elements corresponding to the element group. If omitted $n4 = 0$ is used.

3.2.2.2 The subkeyword NATBOUNCOND

The subkeyword NATBOUNCOND indicates that standard boundary elements are used. Must be followed by data records of the type:

```
BNGRP i = ( type = n1 [, n2, n3, . . . ] )
```

with i the boundary element group number.

$n1$ is the boundary problem number of the i^{th} boundary element group. $n2, n3$ etc. have the same meaning as under TYPES.

The boundary element groups must be defined sequentially from 1. No boundary element group numbers may be skipped. The largest boundary element group number defines NUMNATBND, the number of boundary element groups.

When the user wants to define his own boundary elements, he has to use problem definition numbers between 1 and 99.

Problem definition numbers above 99 correspond to SEPRAN standard problems.

When all problem definition numbers $n1, n2, \dots$ of a boundary element group are smaller than 100, then the data record BNGRP i may be followed by a data records of the type:

```
NPELM = j
NUMDEGFD = ( d1, d2, ... , dn )
VEC 1 = ( d1, d2, ... , dn )
.
.
.
VEC i = ( d1, d2, ... , dn )
NUMBER 1 = ( d1, d2, . . . , dn )
NUMBER 2 = ( d1, d2, . . . , dn )
.
.
NUMBER i = ( d1, d2, . . . , dn )
```

If $ITYPE > 10000$, $ITYPE$ consists of two subparts $ITYPE_{act}$ and $IEXTRA$, according to $ITYPE_{act} + 10000 \times IEXTRA$. $ITYPE_{act}$ denotes the actual type number and $IEXTRA$ denotes the extra number of vectors of special structure, the user wants to add. These vectors of special structure are only coupled to nodes, never to elements.

For each of the $IEXTRA$ vectors a record must be supplied immediately after the BNGRP $i = (type = \dots$ record. These records must have the following shape:

```
n_vertex = n1, n_midside_points = n2, n_centroid = n3, n_face = n4
```

NPELM = j defines the number of nodes j in the boundary element. If this record is omitted the number of nodes in the elements is detected from the corresponding boundary elements given in the input part BOUNELEMENTS.

NUMDEGFD = ($d1, d2, \dots, dn$) has exactly the same meaning as in the case of standard element groups.

If this record is not given it is supposed that the number of unknowns in each point is the same as the maximum of unknowns in that point as defined by the internal elements.

VEC $i = (d1, d2, \dots, dn)$ has exactly the same meaning as in the case of standard element groups.

If omitted, information is copied from the internal elements.

NUMBER i = (d1, d2, ... , dn) has exactly the same meaning as in the case of standard element groups.

If omitted, information is copied from the internal elements.

Remark: If boundary elements are lying within more than one internal element, without being the common side of these elements, the user must define line elements instead of boundary elements. Furthermore it is assumed that the number of unknowns in the boundary element per point is defined by the number of unknowns present in the nodal point. If the user wants to use a different number he has to use line elements.

3.2.2.3 The subkeyword **BOUNELEMENTS**

The subkeyword **BOUNELEMENTS** indicates that boundary elements are created. Must be followed by data records of the following type:

```

BELMi = POINTS ( P3, P6, P8, . . . )
BELMi = CURVES ( SHAPE = 1, C1 to C2 )
BELMi = SURFACES ( S1 to S2 )
BELMi = CURVES ( SHAPE = 2, C5 )
BELMi = OBSTACLE k, SHAPE = 2
BELMi = CONTACT_ELEMENTS j
BELMi = NO_CONTACT_ELEMENTS j
BELMi = CONTACT_POINTS j
BELMi = NO_CONTACT_POINTS j
BELMi = CURV_POINTS ( C1 to C2 )
BELMi = SURF_POINTS ( S1 to S2 )
BELMi = CROSSSECTION_OBSTACLE i, EXCLUDE_TYPE = k, EXCLUDE_CURVES (C1 to C2 )
BELMi = ZERO_LEVELSET i, EXCLUDE_CURVES (c1, c2, c3, ... )

```

These records take care of the generation of the boundary elements.
meaning of the various parameters:

i is the boundary element group number. More than one boundary group number may be used. Also a specific number **i** may be used more than once, provided the boundary elements correspond to the same group.

If boundary element group numbers are not used, the number of elements for that group is equal to zero. The boundary elements must be created with increasing boundary element group number.

POINTS defines boundary elements that consist of user points only. **POINTS** must be followed by a series of user points. Only points that coincide with nodal points may be used.

P_i, P_j, ... define the user point numbers. It is also possible to use **P_i TO p_j** in which case all user points **P_i, P_i + 1, ... , P_j** are used. Both possibilities may be used in combination, like

```
POINTS P3, P5, P7 TO P10, p15 TO p23
```

CURVES defines boundary elements that are defined on curves. **CURVES** must be followed by the shape number and the curve numbers.

SHAPE = .. defines the shape number of the standard elements. See Table 2.2.1.

If **SHAPE = ..** is omitted the shape of the boundary elements is derived from the internal elements. If all internal elements are quadratic, quadratic boundary elements are assumed, otherwise linear ones.

If **INPELM = ..** has been given, this may also define the shape number.

C1 to C2 boundary elements are generated along the curves **C1 to C2**, when **C2** is not given only curve **C1** is used. When **C2** is given, the curves **C1 to C2** must be subsequent curves with coinciding initial and end point, i.e. the end point of **C1** must be equal to the initial point of **C1 + 1** etc.

The boundary elements must always be created counter-clockwise with respect to the inner region. Hence the corresponding curves must also be generated counter-clockwise.

SURFACES defines boundary elements that are defined on surfaces. **SURFACES** must be followed by the surface numbers. The shape number is copied from the surfaces **S1 to S2**. Hence these surfaces must have been created with the same shape number.

S1 to S2 surface elements from the surfaces S1 to S2 are used. When S2 is not given only S1 is used.

OBSTACLE j defines boundary elements that are defined for all active elements along **OBSTACLE j** . An active element is an element consisting of nodal points of the mesh that all lie on the obstacle. During the computation, elements may become active or may be deactivated. Hence the number of obstacle boundary elements varies during the computations. The shape number may be 1 (linear elements) or 2 (quadratic elements). The default value is 1.

CONTACT_ELEMENTS j defines boundary elements that are defined for all elements along **CONTACT j** that make contact. An element makes contact if all nodes of that element make contact. The shape of the elements is defined by the surface at which the contact takes place.

NO_CONTACT_ELEMENTS j defines boundary elements that are defined for all elements along the surface corresponding to **CONTACT j** that make no contact. An element makes contact if all nodes of that element make contact. The shape of the elements is defined by the surface at which the contact takes place. In this case all other elements along that surface are used.

CONTACT_POINTS j defines boundary elements that are defined for all points along **CONTACT j** that make contact. In this case point elements are defined (for example for concentrated loads) and the shape number is set equal to 0

NO_CONTACT_POINTS j defines boundary elements that are defined for all points along the surface corresponding to **CONTACT j** that make no contact. In this case point elements are defined (for example for concentrated loads) and the shape number is set equal to 0

CURV_POINTS C_i to C_j defines point boundary elements along the curves C_i to C_j . These curves must form a contiguous set of curves.

SURF_POINTS S_i to S_j defines point boundary elements along the surfaces S_i to S_j .

CROSSECTION_OBSTACLE i defines a very special boundary element.

This element can only be used in combination with elements of type 922 as described in the manual Standard problems Section xxx.

The element is meant for a fluid problem in a fixed mesh where an obstacle is present. i is the obstacle sequence number.

The obstacle may be situated anywhere in the mesh and does not have to coincide with the mesh itself. In order to make the velocity of the points on the boundary of the obstacle equal to the obstacle velocity these elements must be used. The intersection of the fixed mesh with the obstacle is computed and line elements are defined that correspond with the edges of the fixed grid that intersect the obstacle. The cross-section points are situated in these elements. These new line elements are used to express the given velocity in the intersection point of the obstacle into the velocity of the two end points of the line element. This condition is formulated as a constraint and as a consequence two (2D) or three (3D) extra Lagrangian multipliers are introduced.

The new line elements are only introduced for edges with a point completely inside the obstacle and a point outside the obstacle. If a point is on the boundary of the obstacle (or very close to it), no cross-section element is introduced.

In order to avoid too many constraints the option `exclude_type = k` may be used.

The parameter k has the following meaning:

1. To each point outside the obstacle only one line element may be connected. Which of the possibilities is chosen in case of more cross-section elements connected to that point, is arbitrary.
2. To each point inside or outside the obstacle only one line element is connected. This is more restrictive than $k = 1$.

The default value is $k = 1$.

The user may exclude curves of the obstacle to be part of the cross-section by using the option `exclude_curves (c1 to c2)`, in which case intersections with the curves C1 until C2 are excluded.

The default value is that no curves are excluded.

ZERO_LEVELSET *i* defines boundary elements in case of the level set method. The boundary elements are defined along the line (2D) or face (3d) with level set zero. This is in general the interface we are interested in.

i refers to the level set sequence number.

In 3D the option `exclude_curves` defines the set of curves where we do not define the boundary elements.

3.2.2.4 The subkeyword ESSBOUNCOND

The subkeyword ESSBOUNCOND indicates that essential boundary conditions will be prescribed. Must be followed by data records of the type:

```
[degrees of freedom] [location part]
```

in arbitrary order. The part degrees of freedom has the following shape:

```
DEGFD k [,DEGFD 1 [, DEGFD m ....] ]
```

DEGFDj indicates that the j^{th} degree of freedom will be prescribed (the value of these degrees of freedom must be given by the input block "ESSENTIAL BOUNDARY CONDITIONS" or "CREATE". Hence DEGFD1, DEGFD3 indicates that the first and third degree of freedom in the corresponding nodal points are prescribed. At most 20 degrees of freedom are permitted in one record.

When DEGFDj = is omitted, all degrees of freedom are supposed to be prescribed in the corresponding nodal points. This is identical to DEGFD0.

The location part has the following shape

```
POINTS ( Pk, P1, . . . , Pm)
CURVES [l] (Cj [to Cm] )
SURFACES [i1, i2, i3, ... ] (Sj [to Sm] ), [SKIP_BOUNDARY] &
    [SKIP_CURVES (Ci, Cj, Ck, ... ),] [INCLUDE_CURVES (Ci, Cj, Ck, ... )]
VOLUMES [i1, i2, i3, ... ] (Vj [to Vm] )
NODES (Nj [to Nm] )
ELEMENTS i [to j] (RN1)
GROUP ielgrp [(RN1, RN2, RN3, ... )]
OBSTACLE i
CONTACT i
NO_CONTACT i
NODAL_POINTS = ( i1, i2, i3, ... )
FILE_NODAL_POINTS = 'name_of_file'
IN_ALL_OBSTACLE i
IN_INNER_OBSTACLE i
IN_BOUN_OBSTACLE i
ON_BOUN_OBSTACLE i
INON_BOUN_OBSTACLE i
OUTER_CURVES
OUTER_SURFACES
ZERO_LEVELSET i
CAVITATION i
INTERSECTION (Si, Sj, Sk, ... )
```

POINTS indicates that essential boundary conditions are given in user defined points, POINTS must be followed by a series of user points. Only points that coincide with nodal points may be used.

P_i, P_j, \dots define the user point numbers. It is also possible to use P_i TO p_j in which case all user points $P_i, P_i + 1, \dots, P_j$ are used. Both possibilities may be used in combination, like

```
POINTS P3, P5, P7 TO P10, p15 TO p23
```

CURVES indicates that essential boundary conditions are given on curves.

C_j to C_m essential boundary conditions of this type are defined on the curves C_j to C_m, or C_j only when C_m is omitted. When C5 is given, the curves C_j to C_m must be subsequent curves with coinciding initial and end point, i.e. the end point of C_j must be equal to the initial point of C_{j+1} etc. Only if the step to be used for the curves is equal to 1, which means that all nodes are used, then it is not necessary that all curves are contiguous.

l consists of two parts: $l = i + 100 \times \text{EXCLUDE}$

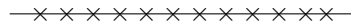
i may take one of the following values:

i=0 all nodal points on the curves are prescribed as indicated by DEGF_{Dj}.

i>0 only the nodal points 1, 1+(i+1), 1+2×(i+1), ... on these curves are prescribed as indicated by DEGF_{Dj}.

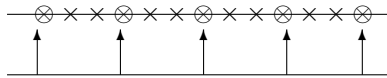
i<0 all nodal points except the points 1, 1-(i-1), 1-2×(i-1), ... on these curves are prescribed as indicated by DEGF_{Dj}.

Hence **i=0**



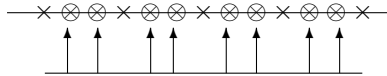
degrees of freedom DEGF_{Dj} are prescribed in all nodal points.

i=2



degrees of freedom DEGF_{Dj} are prescribed in the nodal points indicated by ⊗.

i=-2



degrees of freedom DEGF_{Dj} are prescribed in the nodal points indicated by ⊗.

Remark: **i** must be so that the last nodal point of the curves C₁ to C₅ is equal to 1 + **k i** with **k** integer (**> 0** or **< 0**).

EXCLUDE may take one of the following values:

0 All points of the curves C₁ to C₅ are used as indicated by **i**.

1 All points except the begin point of C₁ are used as indicated by **i**.

2 All points except the end point of C₅ (or C₁ if C₅ is omitted) are used as indicated by **i**.

3 All points except the begin point of C₁ and the end point of C₅ (or C₁ if C₅ is omitted) are used as indicated by **i**.

Remark: **EXCLUDE > 0** may only be used in combination with **i ≥ 0**.

SURFACES must be used to define essential boundary conditions on surfaces.

i1, i2, i3, have the following meaning:

When omitted, all nodal points on the surface get the prescribed degrees of freedom as indicated by DEGF_{D1}, DEGF_{D2}, Otherwise, the degrees of freedom are only prescribed in the *i*^{1th}, *i*^{2th}, *i*^{3th}, ... nodal point of each element of the surfaces. When degrees of freedom in the other points must be prescribed, a new record is necessary.

(S1 to S2) define the surfaces on which the essential boundary conditions are prescribed; when S2 is omitted, only S1 is used, otherwise the surfaces S1 to S2 are used. The brackets around S1 to S2 may **not** be removed.

SKIP_BOUNDARY indicates that only the internal points of the set of surfaces are transformed. If this keyword is not given all points, including the points on the curves surrounding the set of surfaces, are transformed.

INCLUDE_CURVES has exactly the same meaning as **skip_boundary**, except that the curves given in the list are included in the part of the surface to be transformed.

Also the end points of the curves are included.

SKIP_CURVES has exactly the same meaning as **skip_boundary**, except that of the set of outer curves only those given in the list are skipped.

Also the end points of the curves are skipped. Hence there is a slight difference between including a curve, or skipping the other ones.

Mark that the keywords **skip_boundary**, **include_curves** and **skip_curves** are mutually exclusive.

VOLUMES must be used to define essential boundary conditions on surfaces.

i1, i2, i3, have the following meaning:

When omitted, all nodal points on the volume get the prescribed degrees of freedom as indicated by **DEGFD1, DEGFD2,** Otherwise, the degrees of freedom are only prescribed in the $i1^{th}, i2^{th}, i3^{th}, \dots$ nodal point of each element of the volumes. When degrees of freedom in the other points must be prescribed, a new record is necessary.

(V1 to V2) define the volumes on which the essential boundary conditions are prescribed; when **V2** is omitted, only **V1** is used, otherwise the volumes **V1 to V2** are used. The brackets around **V1 to V2** may **not** be removed.

NODES must be used to define essential boundary conditions in all nodal points with absolute nodal point numbers (not user point numbers) **N1 to N2**. When **N2** is omitted, only **N1** is used.

ELEMENT must be used to define essential boundary conditions for all elements.

RN1 defines the relative nodal point. Hence the essential boundary condition is defined in the $RN1^{th}$ node of all elements.

Let, for example, in Figure 3.2.2.2 nodal point 17 be the start point of the element, and let the direction of the nodal points be counter clockwise, then point 30 has relative nodal point number 4.

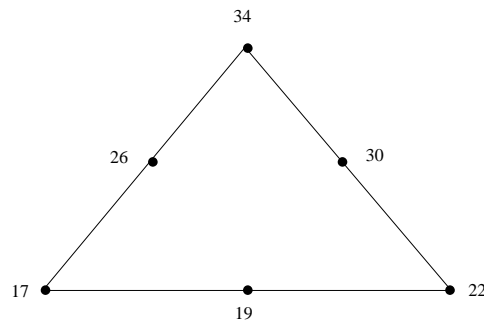


Figure 3.2.2.2: Nodal points in element.

i, j defines the range of the element numbers for which the essential boundary condition is prescribed. If j is omitted, only i is used.

GROUP must be used to define essential boundary conditions for a complete group of elements.

RN1, RN2, ... , RNl define the relative nodal points **RN1, RN2, ... , RNl** of these elements. When the relative nodal points are omitted, all nodal points in the elements are used.

ielgrp defines the element group number.

OBSTACLE must be used to define essential boundary conditions along active obstacle points. It may only be used in combination with free or moving boundary problems. If in the boundary adapting subroutine obstacles are defined it makes sense to use this option, otherwise it is neglected.

If this option is given, all active points along **OBSTACLE** j get the prescribed boundary conditions as indicated by **DEGFD**. Active points are all nodes of the mesh that lie on the obstacle. During the computation, nodes may become active or may be deactivated. Hence the number of obstacle boundary conditions varies during the computations.

CONTACT indicates that all nodal points at the contact surface corresponding to contact i , that make contact, have essential boundary conditions of the prescribed type.

NO_CONTACT indicates that all nodal points at the contact surface corresponding to contact i , that make no contact, have essential boundary conditions of the prescribed type.

NODAL_POINTS = (...) indicates that a series of nodal points is given by the user in which essential boundary conditions are valid.

FILE_NODAL_POINTS='file_name' indicates that a series of nodal points is given by the user in which essential boundary conditions are valid.

`file_name` indicates the name of a file in which the nodal point numbers are stored. This name must be positioned between two quotes. If this option is used, the user must provide a file of this name as described in Section 3.5.1.

At the moment the file is used, it is opened with reference number 75, the contents are read and the file is closed again. This means that the user may not have opened a file with reference number 75 at the same time, and moreover that if the file is reused again reading starts from the first record.

IN_ALL_OBSTACLE i Essential boundary conditions are prescribed in all nodes of the mesh that are situated within the obstacle with obstacle sequence number i

IN_INNER_OBSTACLE i Essential boundary conditions are prescribed in the nodes of those elements of the mesh that are completely within the obstacle with obstacle sequence number i . If an element is completely inside the obstacle but has some points on the boundary of the obstacle then all nodes of this element except those on the boundary belong to this group. Compared to **IN_ALL_OBSTACLE**, this means that nodes on the boundary of the obstacle are excluded, as well as nodes of elements that are partly outside the obstacle even if they are inside the obstacle.

IN_BOUN_OBSTACLE i Essential boundary conditions are prescribed in the nodes of those elements of the mesh that are partly outside the obstacle with obstacle sequence number i . So all the points that are excluded in `in_inner_obstacle` but are part of `in_all_obstacle` belong to `in_boun_obstacle`.

ON_BOUN_OBSTACLE i Essential boundary conditions are prescribed in the nodes of those elements of the mesh that are on the boundary of the obstacle with obstacle sequence number i .

INON_BOUN_OBSTACLE i Essential boundary conditions are prescribed in the nodes of those elements of the mesh that belong to `on_boun_obstacle`, but besides that are only inside elements that are completely inside the obstacle and not in elements that are outside the obstacle.

OUTER_CURVES Essential boundary conditions are prescribed in all the nodes of the mesh that are on the outer boundary of a 2D mesh. At this moment it is only possible to prescribe all the unknowns in these curves.

OUTER_SURFACES Essential boundary conditions are prescribed in all the nodes of the mesh that are on the outer boundary of a 3D mesh. At this moment it is only possible to prescribe all the unknowns in these surfaces.

ZERO_LEVELSET i Essential boundary conditions are prescribed in all the nodes of the mesh where the level set function ϕ_i has the value 0. This is exactly the boundary made by the command `make_levelset_mesh` in the structure block. See (3.2.3.18).

CAVITATION i Essential boundary conditions are prescribed in all the nodes in a bearing with pressures below the cavitation pressure.
The parameter i in this case must always be 1.
See Section 3.2.24 and 3.2.3.21

INTERSECTION (S_i, S_j, S_k, \dots) Essential boundary conditions are prescribed in all the nodes in the intersection of each couple of surfaces given in the corresponding list.
Hence in case of 3 surfaces S1, S5 and S7 it concerns the common nodes of S1 and S5, S1 and S7 and S5 and S7.

3.2.2.5 The subkeyword LOCALTRANSFORM

The subkeyword LOCALTRANSFORM indicates that local transformations must be defined in the points created on the curves and surfaces defined by the data records. These records consist of the following parts

[degrees of freedom] [location part] [transformation_information]

in arbitrary order. These parts itself must be stored in one record, but for each new definition a new record must be used.

The part degrees of freedom has the following shape

DEGFD k [,DEGFD 1 [, DEGFD m]]

If omitted DEGFD 0 is assumed

The location part has one of the following structures:

CURVES [l] (Cj [to Cm])
 SURFACES [i1, i2, i3, ...] (Sj [to Sm]), [SKIP_BOUNDARY] &
 [SKIP_CURVES (Ci, Cj, Ck, ...),] [INCLUDE_CURVES (Ci, Cj, Ck, ...)]
 OBSTACLE i, JSTEP = j
 CONTACT i, JSTEP = j

The brackets may **not** be removed from these records.

Transformation information has the following shape:

TRANSFORMATION = s, MATRIXR = (r1,r2,r3,r4), MATRIXV = (v1,v2,v3,v4), &
 TANG_DIR = t, NORMAL = n, TANG_CURVE = (k_1, k_2, ...)

degrees of freedom

DEGFD1,DEGFD2 indicates that the $DEGFD1^{th}$ and the $DEGFD2^{th}$ degree of freedom must be transformed (2D only). If omitted the first and the second degree of freedom are transformed.

DEGFD1,DEGFD2,DEGFD3 indicates that the $DEGFD1^{th}$, $DEGFD2^{th}$ and the $DEGFD3^{th}$ degree of freedom must be transformed (3D only). If omitted the first, the second and the third degree of freedom are transformed.

location part

CURVES When local transformations must be defined on curves the function CURVES must be used followed by l and the curve numbers C_k and C_j , indicating that local transformations must be defined on the curves C_k to C_j , or C_k when C_j is omitted. When C_j is given, the curves C_k to C_j must be subsequent curves with coinciding initial and end point, i.e. the end point of C_k must be equal to the initial point of C_{k+1} etc.

l consists of two parts: $l = i + 100 \times \text{EXCLUDE}$

i has the following meaning:

If i is omitted, $i=0$ is assumed.

When $i=0$ all nodal points on the curves are transformed as indicated by DEGFD1 and DEGFD2.

When $i>0$ only the points $1, 1 + (i+1), 1 + 2 \times (i+1), \dots$ on these curves are transformed as indicated by DEGFD1 and DEGFD2.

When $i < 0$ all nodal points except the points 1, 1 - (i-1), 1 - 2×(i-1), . . . on these curves are transformed as indicated by DEGFD1 and DEGFD2.

Compare with the essential boundary conditions.

For EXCLUDE the following possibilities are available:

EXCLUDE=0 All points of the curves C1 to C5 are used as indicated by i.

EXCLUDE=1 All points except the begin point of C1 are used as indicated by i.

EXCLUDE=2 All points except the end point of C5 (or C1 if C5 is omitted) are used as indicated by i.

EXCLUDE=3 All points except the begin point of C1 and the end point of C5 (or C1 if C5 is omitted) are used as indicated by i.

Remarks: EXCLUDE > 0 may only be used in combination with $i \geq 0$.

Curves may also be transformed in R^3 , however, if the standard transformation is applied, only the tangential direction is defined uniquely. The first two new local coordinates, referring to 2 normals are defined in some arbitrary way, which means that this option makes only sense if the essential boundary conditions corresponding to these normal directions are both zero.

SURFACES is used to define local transformation along surfaces.

i1, i2, i3, . . . have the following meaning:

When omitted, all nodal points on the surface are transformed. Otherwise, only the $i1^{th}, i2^{th}, i3^{th}, \dots$ nodal point of each element of the surfaces are transformed.

(S1 to S2) define the surfaces on which the transformation must take place; when S2 is omitted, only S1 is used, otherwise the surfaces S1 to S2 are used. The brackets around S1 to S2 may **not** be removed.

SKIP_BOUNDARY indicates that only the internal points of the set of surfaces are transformed. If this keyword is not given all points, including the points on the curves surrounding the set of surfaces, are transformed.

INCLUDE_CURVES has exactly the same meaning as **skip_boundary**, except that the curves given in the list are included in the part of the surface to be transformed. Also the end points of the curves are included.

SKIP_CURVES has exactly the same meaning as **skip_boundary**, except that of the set of outer curves only those given in the list are skipped. Also the end points of the curves are skipped. Hence there is a slight difference between including a curve, or skipping the other ones.

Mark that the keywords **skip_boundary**, **include_curves** and **skip_curves** are mutually exclusive.

OBSTACLE indicates that the local transformation is defined along obstacle i . That means that all points at the obstacle that are active will be transformed. The complete obstacle is used to define the tangential vector.

If JSTEP > 0, only the points 1 1+JSTEP, 1+2×JSTEP along the obstacle curve are used.

CONTACT has not yet been implemented.

transformation_information

The parameter s behind the keyword TRANSFORMATION defines how the transformation must be carried out. The following values for s are available:

STANDARD
 SYMMETRIC
 NON_SYMMETRIC
 MIN_UN_UT

If this part is omitted, STANDARD is assumed.

STANDARD implies that a standard symmetric transformation is applied. The unknowns are redefined in the sequence (u_n, u_t) according to the following definition:

$$\begin{bmatrix} u_{DEGFD1} \\ u_{DEGFD2} \end{bmatrix} = \mathbf{R} \begin{bmatrix} u_n \\ u_t \end{bmatrix} \quad (3.2.2.1)$$

with (u_{DEGFD1}, u_{DEGFD2}) the old, and (u_n, u_t) the new degrees of freedom (in that sequence). The tangential vector is computed and has the direction of the curves from the first nodal point on C_i to the last point on C_j , the normal is taken clockwise from the tangential vector. See Figure 3.2.2.3.

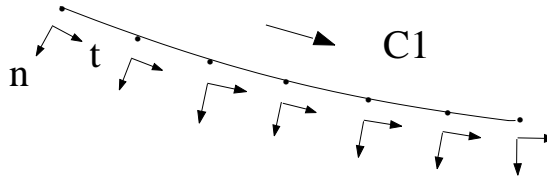


Figure 3.2.2.3: Normal and tangential vectors on a curve.

When the user wants to start at the end of a curve he must use negative values for the curve numbers.

Hence $DEGFD1, DEGFD2 = \text{CURVES } i(-C_i, -C_j)$

In that case all curves from C_i to C_j are considered from end point to begin point.

Warning

The direction of the tangential vectors is computed by taking the line between the two neighboring points, except for the two end points where the two last points are connected. As a consequence in a sharp corner an average tangential vector is defined. At the end of the curves with local transformations one must be very careful with the prescription of boundary conditions, since in the endpoints the degrees of freedom may be rotated. Especially if such an endpoint is a corner of a region this may introduce severe complications. For example let in Figure 3.2.2.4 a local transform be along curve C1 are (u_n, u_t) , which is equal to $(-u_y, u_x)$ whereas the untransformed degrees of freedom at C2 are (u_x, u_y) . In point P we therefore have also the degrees of freedom $(-u_y, u_x)$ and in that respect P differs from the rest of curve C2.

Remark

If the curves are in R^3 , then the standard transformation is defined as follows:

Let \mathbf{t} be the tangential vector along the curve. Let i be the smallest component of \mathbf{t} . Then the i^{th} element of the vector \mathbf{n}_1 is set equal to 0, and the other two components are created

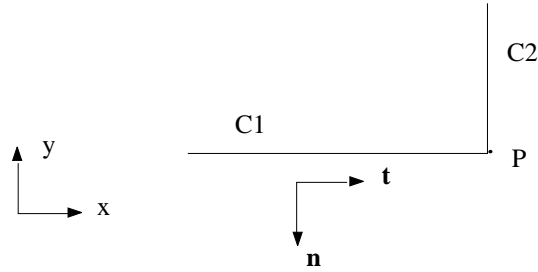


Figure 3.2.2.4: Corner of a region where local transformations have been defined along one curve.

such that $\mathbf{t} \cdot \mathbf{n}_1 = 0$ and $\|\mathbf{n}_1\| = 1$. The second normal vector \mathbf{n}_2 is made orthogonal to \mathbf{t} and \mathbf{n}_1 , by computing the outer product and normalizing.

The transformation matrix \mathbf{R} is defined as: $\mathbf{R} = (\mathbf{n}_1, \mathbf{n}_2, \mathbf{t})$.

Since the directions \mathbf{n}_1 and \mathbf{n}_2 are more or less arbitrary, this transformation makes only sense, if one wants to make both normal components equal to 0.

MIN_UN_UT is identical to standard, however, the new degrees of freedom are not (u_n, u_t) but $(-u_n, u_t)$. This may be for example necessary in case of anti-symmetric periodical boundary conditions. See the manual Standard Problems Section 7.1.10 for an example.

SYMMETRIC is identical to STANDARD, however, the user must define the transformation matrix R himself. In this case it is necessary to give the option **MATRIXR** explicitly according to:

MATRIXR = (**R_11**, **R_12**, **R_21**, **R_22**)

These numbers define the transformation matrix by:

$$\mathbf{R} = \begin{bmatrix} R_{11} & -R_{12} \\ R_{21} & R_{22} \end{bmatrix} \quad (3.2.2.2)$$

The matrix must be symmetrical hence $R_{12} = R_{21}$ In the case of STANDARD this transformation is

$$\mathbf{R} = \begin{bmatrix} n_x & -n_y \\ n_y & n_x \end{bmatrix} \quad (3.2.2.3)$$

with $n_x^2 + n_y^2 = 1$.

R_{ij} may have one of the following shapes:

v
VALUE = **v**
FUNC = **i**

v or **VALUE** = v defines the value explicitly.

FUNC = i indicates that the matrix element is a function of space. The user written function subroutine **FUNC**TR (See Section 3.3.7) must be provided by the user. The parameter i is used to distinguish between various functions.

The matrix R defines the relation between old and new degrees of freedom according to

$$\begin{bmatrix} u_{DEGFD1} \\ u_{DEGFD2} \end{bmatrix} = R \begin{bmatrix} u_{new1} \\ u_{new2} \end{bmatrix} \quad (3.2.2.4)$$

For an example of the use of this option see the manual Standard Problems Section 7.1.10.

NON_SYMMETRIC This special option not only transforms the unknowns but also the so-called test functions. This option is only necessary for very special boundary conditions.

In this case the unknowns are transformed with a matrix \mathbf{R} as is the case with the option SYMMETRIC and the test functions are transformed with a matrix \mathbf{V} . Both matrices may be unsymmetrical. The effect of this transformation is:

$$\begin{bmatrix} u_{DEGFD1} \\ u_{DEGFD2} \end{bmatrix} = \mathbf{R} \begin{bmatrix} u_{new1} \\ u_{new2} \end{bmatrix} \quad (3.2.2.5)$$

Furthermore in each transformation point the part of the large matrix corresponding to this point and the indicated degrees of freedom and the part of the right-hand side are transformed according to the relations:

$$\mathbf{S}_{transform} = \mathbf{V}^T \mathbf{S} \mathbf{R} \quad \mathbf{F}_{transform} = \mathbf{V}^T \mathbf{F} \quad (3.2.2.6)$$

with \mathbf{S} the matrix and \mathbf{F} the right-hand side.

The option NON_SYMMETRIC requires the matrix \mathbf{R} defined by MATRIXR:

MATRIXR = (R_11, R_12, R_21, R_22)

and also the matrix \mathbf{V} defined by

MATRIXV = (V_11, V_12, V_21, V_22)

V_{ij} may be of the same shape as R_{ij} given above.

TANG_DIR = t defines the direction of the first tangential component. This keyword is only used in combination with SURFACES. The normal on a surface is uniquely defined, however, to define the tangential directions extra input to define the direction is needed. If this keyword is given the first tangential direction is defined by the subkeyword t . The following options for t are available:

LINE(P1,P2)
XDIR
YDIR
ZDIR

Meaning of these subkeywords:

LINE(P1,P2) The first tangential direction (\mathbf{t}_1) is given along the line P1 to P2. P1 and P2 must be standard user points.

XDIR $\mathbf{t}_1 = (1,0,0)$

YDIR $\mathbf{t}_1 = (0,1,0)$

ZDIR $\mathbf{t}_1 = (0,0,1)$

The second tangential direction is defined in the direction given by the outer product of the normal and the first tangential direction.

If the tangential direction is not given, some local tangential directions are defined, which are unknown to the user, and certainly not uniquely defined.

NORMAL = n defines the direction of the normal component. This keyword is only used in combination with SURFACES.

The following values of n are permitted:

outward
inward

If outward is used, the normal component in the outward direction is used, in the case of inward the inward directed normal is used.

if omitted the default value: outward is used.

TANG_CURVE = k_1, k_2, \dots defines the direction of the second tangential vector along curves k_1, k_2, \dots . This option is meant to make a special transformation along a set of curves in a surface. The normal vector is defined in the standard way, perpendicular to the surface. The second tangential vector is defined tangent to the curves and the first tangential vector is defined perpendicular to both other vectors.

So after this option the first component along the curve is perpendicular to the surface, the second in the surface but perpendicular to the curve and the third one in the direction of the curve. Such an option can for example be used to force a vector in the direction of the curve, by setting the first and second degree of freedom equal to 0.

3.2.2.6 The subkeyword UNKNOWNCONSTANT

The subkeyword UNKNOWNCONSTANT indicates that along one or more parts of the boundary we have the boundary condition u equals unknown constant. The parts of the boundaries are defined by data records consisting of the following parts:

[degree of freedom] [location part]

in arbitrary order. These parts itself must be stored in one record, but for each new definition a new record must be used.

The part degrees of freedom has the following shape

DEGFD k

The location part has one of the following structures:

CURVES [i] (C j [to C m])
SURFACES [i_1, i_2, i_3, \dots] (S j [to S m])

The brackets may **not** be removed from these records.

degrees of freedom

DEGFD k indicates that the k^{th} degree of freedom is constant along the part of the boundary indicated by the location part. If omitted the $k = 1$ is assumed.

location part

When the boundary condition is defined on curves the function CURVES must be used followed by i and the curve numbers C k and C j , indicating that the constant value is defined on the curves C k to C j , or C k when C j is omitted. When C j is given, the curves C k to C j must be subsequent curves with coinciding initial and end point, i.e. the end point of C k must be equal to the initial point of C $k+1$ etc.

i has the following meaning:

If i is omitted, $i=0$ is assumed.

If $i=0$ all nodal points on the curves are assumed to contain the degree of freedom DEGFD l that has constant value along C k to C j .

If $i>0$ only the points 1, $1 + (i+1)$, $1 + 2 \times (i+1)$, . . . on these curves have the constant value.

If $i<0$ all nodal points except the points 1, $1 - (i-1)$, $1 - 2 \times (i-1)$, . . . on these curves have the constant value.

Compare with the essential boundary conditions.

If the boundary condition is given along surfaces the same rules as for the essential boundary conditions apply for the location part.

3.2.2.7 The subkeyword GLOBAL_UNKNOWNNS

The subkeyword GLOBAL_UNKNOWNNS is used when the user wants to define special unknowns that are not coupled to specific nodal points but have a more global character. For example if we have a flow in a straight channel with periodical boundary conditions on instream and outstream boundary and an unknown pressure jump over the inflow and outflow boundary. In order to compute this pressure jump it is necessary to prescribe also the amount of fluid that flows into the channel. The pressure jump is not coupled to one specific point but for example to the complete inflow boundary (or outflow boundary). For this specific unknown one may define a global unknown, which is treated as an extra unknown with an extra equation.

For each global unknown (or in case it is a vector of unknowns: for each vector of global unknowns) a new so-called global group must be introduced. The record GLOBAL_UNKNOWNNS must be followed by extra records defining the type numbers corresponding to the global unknowns according to:

```
glgrp i = type = t
      if types number t positive and < 100 possibly followed by:
          numdegfd = n
```

glgrp i type = t , defines the sequence number i of the global group and the corresponding type number t of the elements corresponding to this group. Only one type number is permitted.

numdegfd = n defines the number of unknowns n coupled to this global group. This statement may only be used if the type number t is between 1 and 99. If omitted and the type number is in that range $n = 1$ is assumed.

If GLOBAL_UNKNOWNNS are defined it is necessary to introduce also the keyword GLOBAL_ELEMENTS to specify where the global unknowns are defined.

For an example of the use of global unknowns the user is referred to the manual Standard Problems, Sections 7.1.9 and 7.1.11.

3.2.2.8 The subkeyword GLOBAL_ELEMENTS

The subkeyword GLOBAL_ELEMENTS must be used to specify the region to which the global unknown is coupled. The corresponding elements are used to compute the extra rows and columns in the matrix and right-hand side. The following extra records are required

```
gelm2 = curves ( [shape = k,] c1 to c2 )
gelm3 = surfaces ( s1 to s2 )
gelm4 = volumes ( v1 to v2 )
gelm5 = all
```

gelm i = curves (shape = k, c1 to c2) Means that the unknown(s) corresponding to global element group i are defined on the curves $c1$ to $c2$. The shape number indicates what type of elements are used in the building of the matrix or right-hand side. $k=1$ refers to linear line elements, $k=2$ to quadratic line elements. These elements are necessary to evaluate the integrals that are used to compute rows and columns in the matrix corresponding to the global unknowns. This option is at this moment only available for two and three-dimensional regions containing surface or volume elements.

gelm i = surfaces (s1 to s2) Means that the unknown(s) corresponding to global element group i are defined on the surfaces $s1$ to $s2$. This option is at this moment only available for three-dimensional regions containing volume elements.

gelm i = volumes (v1 to v2) Means that the unknown(s) corresponding to global element group i are defined on the volumes $v1$ to $v2$. This option has not yet been implemented

all Means that the unknown(s) corresponding to global element group i are defined on the complete domain. This option has not yet been implemented

3.2.2.9 The subkeyword GLOBAL_RENUMBERING

If global unknowns are introduced one usually uses less boundary conditions than in the case without global unknowns. It is common practice that the global unknowns are introduced to prevent prescribing unknown boundary conditions. In the standard case the unknowns are numbered in the sequence: standard unknowns, followed by global unknowns. However, in some cases it may be possible that the sub-matrix created by the standard unknowns only, is itself singular. The global unknowns are used to make the matrix non-singular. Unfortunately, the linear solver does not use pivoting, and when using the profile solver first the standard matrix is decomposed. If the corresponding sub-matrix is singular this leads to an error message about a small pivot and either the process stops or the result is very inaccurate. In order to solve this problem it is necessary to perform a kind of renumbering. In general it is sufficient to replace each row and column corresponding to a global unknown by a row and column of a corresponding standard internal unknown, preferably the last one corresponding to the global unknown.

In order to activate such a renumbering it is necessary to introduce the keyword GLOBAL_RENUMBERING. It must be followed by data records prescribing which unknowns must be interchanged in sequence with the global unknowns.

The data records have the following structure:

```
DEGFDi, DEGFDj, ...
```

This means that the i^{th} physical unknown with the largest sequence number is interchanged with the first global unknown and the j^{th} physical unknown with the largest sequence number is interchanged with the second global unknown and so on. Hence of all the physical unknowns with sequence number i , the one with the highest number (after renumbering) is the one that is interchanged with the first global unknown.

For an example of the use of GLOBAL_RENUMBERING see the manual Standard Problems Section 7.4 and 7.4.1.

3.2.2.10 The subkeyword FICTITIOUS_UNKNOWNS

The subkeyword FICTITIOUS_UNKNOWNS is meant for the use of the fictitious domain method. This is a special free surface method.

In this case it is necessary to define extra unknowns on a special curve of surface. This curve or surface is supposed to be part of a structure, which moves inside a fluid.

On the curve or surface, points are defined with corresponding extra unknowns. These unknowns act as so-called Lagrangian multipliers and are meant to connect the velocities of structure and fluid. These Lagrangian multipliers can not be regarded as usual unknowns, since for each multiplier a constraint is given. The constraint is of course that the velocity of structure and fluid are equal. Such a constraint can not be implemented by standard finite elements, but requires a special approach. For that reason it is necessary to define fictitious unknowns and corresponding elements. For each fictitious unknown (or in case it is a vector of unknowns: for each vector of fictitious unknowns) a new so-called fictitious group must be introduced. The record FICTITIOUS_UNKNOWNS must be followed by extra records defining the type numbers corresponding to the fictitious unknowns according to:

```
fictgrp i = type = t
    if types number t positive and < 100 possibly followed by:
        numdegfd = n
```

fictgrp i type = t , defines the sequence number i of the fictitious group and the corresponding type number t of the elements corresponding to this group. Only one type number is permitted.

numdegfd = n defines the number of unknowns n coupled to this fictitious group. This statement may only be used if the type number t is between 1 and 99. If omitted and the type number is in that range $n = 1$ is assumed.

If FICTITIOUS_UNKNOWNS are defined it is necessary to introduce also the keyword FICTITIOUS_ELEMENTS to specify where the fictitious unknowns are defined.

For an example of the use of fictitious unknowns the user is referred to the manual Standard Problems, Sections 7.4 and 7.4.1.

3.2.2.11 The subkeyword FICTITIOUS_ELEMENTS

The subkeyword FICTITIOUS_ELEMENTS must be used to specify the region to which the fictitious unknown is coupled. The corresponding elements are used to compute the extra rows and columns in the matrix and right-hand side. The following extra records are required

```
felm2 = curves ( c1 to c2 ), description
felm3 = surfaces ( s1 to s2 ), description
```

felm i = curves ($c1$ to $c2$) Means that the unknown(s) corresponding to fictitious element group i are defined on the curves $c1$ to $c2$. **description** defines how the Lagrange multipliers must be positioned and to what structural and fluid groups they are related.

felm i = surfaces ($s1$ to $s2$) Means that the unknown(s) corresponding to fictitious element group i are defined on the surfaces $s1$ to $s2$. **description** has exactly the same meaning as for curves.

The following options for the description part are available:

```
multiplier_shape = i, structure_group = j, fluid_groups = i1 to i2
```

These keywords have the following meaning:

multiplier_shape = i defines the shape of the position of the multipliers in the curves or surfaces. At this moment it is assumed that the multipliers points are always Gaussian integration points. i defines the number of multiplier points per structural element per direction. Hence if $i=2$ this means for curves that there are 2 points per element and for surfaces that there are 4 points per element. Elements on a curve or surface is implicitly defined by the elements on the structural part. If the structural part consists of surfaces, then we use curves, if they consist of volume element, we use surfaces.
Default value: 1

structure_group = j defines the corresponding structural elements.
This part must always be given

fluid_group = $i1$ to $i2$ defines the fluid elements in which the structural elements are positioned.
The precise location may vary in each time step.
This part must always be given

3.2.2.12 The subkeyword SKIP_ELEMENTS

The subkeyword SKIP_ELEMENTS indicates that certain elements as indicated by the data records must be skipped while creating the large matrix and vector. In order to avoid singular matrices automatically all unknowns that are only positioned in elements to be skipped will be considered as prescribed. Hence these unknowns are added to the list of essential boundary conditions. The following data records are available to describe which elements must be skipped:

```
inner_obstacle i
on_obstacle i and curves (cj to ck)
```

These records have the following meaning:

inner_obstacle i All elements of the fixed mesh that are completely within the obstacle with sequence number i are skipped.

on_obstacle i and curves (c_j to c_k) All elements of the fixed mesh that consists of nodes that are either partly within the obstacle (or on the boundary of the obstacle) or are on the curves with sequence numbers between i and j are skipped. Moreover, at least one of the nodes must be in the obstacle and one of the nodes must be on the curves.

So actually it concerns the elements that are partly within the obstacle but with all points outside the obstacle on one of the curves c_j to c_k .

These elements bridge the space between obstacle and boundary.

3.2.2.13 The subkeyword REORDER

The subkeyword REORDER has the shape

REORDER [LEVELS] $i1, i2, (i3, i4, i5), i6$ (optional)

or

REORDER `plast`.

With this command the user can influence the internal numbering of the unknowns. In the standard case all unknowns are ordered internally in the sequence of the (possibly reordered) nodal points. Hence, first all unknowns of nodal point 1, then all unknowns of nodal point 2. etcetera.

Using the command REORDER the user may change this sequence.

If the numbering $i1, i2, (i3, i4, i5), i6$ is used this means that first all unknowns $i1$ are numbered for all nodes, then all unknowns $i2$ for all nodes, then the unknowns $i3, i4$ and $i5$ in that sequence for all nodes, followed by all unknowns $i6$ and finally all other unknowns.

So we get the sequence:

$i1$ (node 1), $i1$ (node 2), ... , $i1$ (node n point),
 $i2$ (node 1), $i2$ (node 2), ... , $i2$ (node n point),
 $i3$ (node 1), $i4$ (node 1), $i5$ (node 1), $i3$ (node 2),
 $i4$ (node 2), $i5$ (node 2), ... , $i5$ (node n point),
 $i6$ (node 1), $i6$ (node 2), ... , $i6$ (node n point),
 $i7$ (node 1), $i8$ (node 1), etc.

So the sequence of the unknowns given after REORDER is used, where unknowns given between brackets are treated as one cluster. All unknowns not given are also treated as a cluster. With unknowns we mean physical unknowns if defined and otherwise degrees of freedom.

Remark: This numbering is only used for the internal unknowns in the solution array and the large matrix. The numbering of other vectors, the element subroutines or the output is not influenced by this statement.

The option LEVELS performs the above numbering per level, where level is a SEPRAN defined cluster of nodal points. So first all unknowns for the first level are numbered, then for the second level and so on.

The definition of LEVEL in SEPRAN is as follows:

Find the neighbor of nodal point 1 with the highest nodal point number. If the nodal points are renumbered internally, the renumbered sequence is used. All nodes with sequence number at least equal to this neighbor belong to level 1.

Find all neighbors of the present level that do not belong to a level itself. All nodes with sequence number at least equal to the neighbor with maximal number belong to the next level. This process is repeated until no nodes are left.

In case the mesh consists of linear elements levels 1 and 2 are clustered to one new level 1.

The option `reorder levels` is meant for so called mixed problems that must be solved in the integrated form, this is without applying special algorithms to delete the mixed character like for example penalty methods, pressure correction or Uzawa schemes. In case a direct solver is applied REORDER LEVELS is more or less necessary, see the manual STANDARD PROBLEMS. For an iterative solution method, both LEVELS and REORDERED LEVELS may be applied. Experiments indicate that in this case to REORDERED LEVELS may give the best performance.

REORDER `plast` is a special option, that is meant for fluid problems, with coupled velocity and pressure degrees of freedom. If this option is used, the unknowns are ordered in the sequence, first all velocity degrees of freedom and then all pressure degrees of freedom.

In case of simple-type methods, this option is mandatory.

3.2.2.14 The subkeywords NUM_LEVELSET and LEVELSET

With respect to using the level set method there are a number of keywords available in the PROBLEM input block. It concerns the main keywords treated in this subsection as well as the subkeywords in the parts referring to boundary conditions.

Description of the main keywords:

NUM_LEVELSET = k (optional)

Defines the number of level set functions that are used to define boundary conditions and so on. The level set functions itself are defined as solution vectors, the corresponding sets get sequence numbers 1 to `num_levelset`. The combination of level set sequence number and level set function is defined in the command `make_levelset_mesh` in the structure block (3.2.3.18).

At this moment only the values 0 and 1 for k have been implemented.

Default value: $k = 0$

LEVELSET i = data (optional)

defines which part of the mesh corresponding to level set sequence number i is taken into account.

data may consist of one of the following keywords:

ALL

POSITIVE_PART

NEGATIVE_PART

Meaning of these keywords

ALL means that all nodes are used in the mesh. In fact this keyword is superfluous.

POSITIVE_PART means that only points where the corresponding level set function ϕ_i has non-negative values are taken as computational domain. All points with negative value of ϕ_i are not used in the solution of equations. Therefore the values in the corresponding nodes are unchanged.

NEGATIVE_PART has the same meaning as **POSITIVE_PART** but now for the opposite sign.

Default value: **ALL**

3.2.2.15 The subkeyword PERIODICAL_BOUNDARY_CONDITIONS

The subkeyword PERIODICAL_BOUNDARY_CONDITIONS indicates that periodical boundary conditions will be prescribed. Must be followed by data records of the type:

```
[location part] [definition part]
```

in arbitrary order. The location part has the following shape

```
POINTS ( Pk, Pl)
CURVES [l] (Cj Cm) )
SURFACES (Sj Sm) ), [EXCLUDE (Ci, Cj, Ck, ... )] or
[EXCLUDE all]
```

These options have the following meaning

POINTS (Pk, Pl) indicates that periodical boundary conditions are defined in user points P_k and P_l .

CURVES l (Cj Cm)] indicates that periodical boundary conditions are defined on the curves C_j and C_m . Both curves must have the same number of nodes. In case a curve must be considered in reversed order a minus sign must be used. Periodical bc's are defines on opposite nodes. The parameter l has the same meaning as in Section (2.2) under the heading MESHCONNECT. Usually this parameter may be neglected.

SURFACES (Sj Sm)] connects nodes on the surfaces S_j and S_m . It is necessary that both surfaces have the same number of nodes and elements and exactly the same topology. The option EXCLUDE = (. .), excludes the boundaries indicated between the brackets from the connection. If ALL is chosen the complete outer boundary is excluded. If curves are given explicitly then only those curves are excluded. Only the curves at the "left" surface S_j must be given, those at surface S_m are excluded in exactly the same way, since the surfaces have the same topology.

The definition part has the following shape:

```
DEGFD k [,DEGFD l [, DEGFD m ...] ], [CONSTANT = c], [FACTOR = f]
```

This part may be repeated several times on the next lines with different degrees of freedom, factors or constants. In that case it refers to the last location part used.

For example

```
degfd = 1, constant = 0, factor = 1
degfd = 2, constant = 0, factor = 2
```

DEGFDj indicates that the j^{th} degree of freedom will be coupled. Hence DEGFD1, DEGFD3 indicates that the first and third degree of freedom in the corresponding nodal points are coupled, hence we have periodical boundary conditions for these unknowns.

constant = c defines the constant c in the case of boundary conditions of the type $\psi_r = f\psi_l + c$. Default value $c = 0$.

factor = f defines the constant f . Default value $f = 1$.

If constant and factor are omitted we are dealing with pure periodical boundary conditions.

Remarks: $f \neq 1$ is only permitted for real solution vectors. The complex case has not yet been implemented!

Examples of the use of these boundary conditions can be found in the manual Standard Problems, Sections 3.1.9, 3.1.10 and 3.5.2.

3.2.3 The main keyword STRUCTURE

The block defined by the main keyword STRUCTURE defines which actions should be performed by program SEPCOMP. In fact this block defines the complete structure of the main program.

STRUCTURE should only be used if the standard options for the solution of a linear problem or non-linear problem do not suffice. In the block STRUCTURE it is precisely described which vectors and scalars are created, how they are created and in which sequence. STRUCTURE contains a number of commands which internally refer to separate subroutines. Each of these subroutines requires input. The input for these specific subroutines is defined in separate input blocks. Each of these blocks may be provided with a local sequence number as described in Section 3.2. The commands in STRUCTURE may refer to these sequence numbers. The block defined by the main keyword STRUCTURE starts with the command STRUCTURE at a separate record and ends with the keyword END on another separate record. In between commands may be given in any sequence and on separate records. However, the commands itself are carried out in exactly the sequence as given in this block. This means that the user himself is responsible for the correctness of the sequence of the commands. The only check that is performed is that vectors and scalars that are used as input have already been filled before.

STRUCTURE makes it possible with a number (100) of vectors (solutions and so on) as well as a number (1000) of scalars.

Each of them has a sequence number.

In the sequel the vector with sequence number i will be denoted by V_i and the scalar with sequence number j by S_j .

The block STRUCTURE consists of a series of commands that may be repeated. Besides the commands STRUCTURE recognizes the a number of structures which are treated at the end of this section. The number of command types in the block STRUCTURE is relatively large. For that reason we have split these commands in this manual in a number of groups, however, commands of the various groups may be interchanged without any problem. The block itself has the following structure

(options are indicated between the square brackets "[" and "] "):

STRUCTURE

a list of commands, each starting on a new line and carried out in the
sequence given

END

Commands may be repeated and given in any order. However, they are executed in exactly the sequence given in the block which means that this sequence defines the complete program and hence must be logical. So it is for example necessary to prescribe the boundary conditions first and then to solve the system of linear equations, since otherwise the effect of the essential boundary conditions to the solution is not present and the solution may be undefined.

The following types of commands are available:

- commands to prescribe boundary conditions or to create a vector. (3.2.3.1)
- commands to solve systems of equations (linear, non-linear or time-dependent). (3.2.3.3)
- commands to compute quantities that can be derived from previously computed vectors, for example compute derivatives or integrals. (3.2.3.6)
- commands for special computations, like the computation of eigenvalues or the computation of a contact problem or the solution of an inverse problem. (3.2.3.7)
- commands to control the output to the output files used by a postprocessor (3.2.3.8)

- commands to manipulate vectors or scalars, including copying and mapping, and changing the problem. (3.2.3.9)
- print commands. (3.2.3.12)
- plot commands. (3.2.3.13)
- commands to read vectors from or write to a file.
This may be a user file or the standard sepran backing storage files. (3.2.3.14)
- commands for mesh manipulation, like refine the mesh, or writing of a mesh. (3.2.3.15)
- commands for interpolation. (3.2.3.16)
- commands to manipulate obstacles. (3.2.3.17)
- commands to use the level set method. (3.2.3.18)
- A special command to give the user the opportunity to carry his own fortran statements.
This may be for example for special output, but also to perform a complete computation. (3.2.3.19)
- auxiliary commands, like changing the structure of the matrix or changing the coefficients and so on. (3.2.3.20)
- Special commands related to certain types of equations. Examples are special commands for the time-dependent Navier-Stokes equations. (3.2.3.21)

Besides these specific commands, also a number of loop commands as for example `for`, `time_loop` and `while` can be used. (3.2.3.23) In Subsection (3.2.3.22) the defaults are described, which are used if no structure block can be found.

A typical input for structure may have the following shape:

```

structure
  prescribe_boundary_conditions, potential
  solve_nonlinear_system, potential
  output
end

```

It always starts with the keyword `structure` and ends with the keyword `end`.

This input corresponds to a standard non-linear potential problem and is strictly speaking superfluous. It assumes that in the constants input block (See Section 1.4 the name `potential` has been added to the list of `vector_names`. Instead of `%potential` also a sequence number (for example 1) may be used, but this option is not recommended because it is less readable. Vector names defined in the computational program are also available in the postprocessing.

The sequence numbers in this example refer to the sequence numbers of the input blocks, where the input for this specific part can be found. This is meant for the case that there are more statements with the same type of action, however, with different input blocks. If omitted the default sequence number is used, usually 1.

This example carries out the following actions:

1. The essential boundary conditions as described in the input block `essential boundary conditions` are applied to the vector with name `potential`. The rest of the vector is set equal to 0.
2. The system of non-linear equations as described in the input block `,nonlinear_equations` is solved. The vector with name `potential` is used as start vector, and the final solution is stored in that same vector.

3. The resulting vector is written to the file `sepcomp.out` in order to be used by the program `SEPPOST`. If there is extra information to be written, this is described in the input block `output`.

Next we shall describe each of these groups in detail

3.2.3.1 commands to prescribe boundary conditions or to create a vector

The following commands are available:

```

PRESCRIBE_BOUNDARY_CONDITIONS vector_name [options]
CREATE_VECTOR vector_name [options]
CREATE_FORCE_VECTOR vector_name [options]
VECTOR vector_name = [options]

```

Mark that the input file is case insensitive except for texts between quotes. Hence the use of capitals in the previous part is only to emphasize the commands. Meaning of the various commands:

PRESCRIBE_BOUNDARY_CONDITIONS vector_name [options]

With this command the vector with name `vector_name` is provided with essential boundary conditions.

The result of this operation is that vector `vector_name` has been filled or changed.

For a description of the options, see [3.2.3.2](#)

CREATE_VECTOR vector_name [options]

is used to create the vector `vector_name` explicitly. This vector may be used as initial estimate for a non-linear problem, just to prescribe the essential boundary conditions, but also to be used in the definition of coefficients for a differential equation.

Exactly the same options as for `PRESCRIBE_BOUNDARY_CONDITIONS` are available. The only differences are that `FUNC = k` refers to function `FUNC` as described in the INTRODUCTION Section 5.5.4. and that `sequence_number s` refers to the input block "CREATE" ([3.2.10](#)) with sequence number `s`.

VECTOR vector_name [options]

is similar to `CREATE_VECTOR`. However, it must have the form:

```
vector vector_name = xxx, options
```

where `xxx` is either a constant or an expression, which can be a scalar or a vector expression. The vector expression may only contain vectors that have been defined before including `x_coor`, `y_coor`, `z_coor` and `coor`.

The following options are available:

```

PROBLEM = p
TYPE = t
CURVES = (Ci, Cj, Ck )
SKIP_ELEMENT_GROUPS = (s1, s2, ... )
DEGFD j

```

Meaning of the various options:

PROBLEM = p defines the problem number.

Default value: 1

TYPE = t indicates that the vector is a vector of special structure, with sequence number t .

Default value: solution vector

CURVES = (Ci, Cj, Ck) defines the curves for which the vector is defined.

Default value: the whole region

SKIP_ELEMENT_GROUPS = (s1, s2, ...) defines element groups to be skipped.

Default value: none

DEGFD j means that only the j -th degree of freedom is filled.

Default value: the complete vector

CREATE_FORCE_VECTOR *vector_name* [options]

The command `create_force_vector` creates a right-hand side vector without the effect of the essential boundary conditions.

One can use this command for example if one wants to translate a distributed load to a nodal point load. Since the result is a vector with the same structure as the solution and right-hand side, this vector may be added to or subtracted from other vectors.

The following options are available

```
SEQ_COEF = s
PROBLEM = p
LINEAR_SUBELEMENTS
```

Information about the coefficients for the differential equation and natural boundary conditions is read in the input block "COEFFICIENTS" (3.2.6) with sequence number c as indicated by `seq_coef = c`.

After that the right-hand-side vector is built, using the coefficients as described in the first step. The effect of essential boundary conditions is not taken into account.

The result of the operation is that vector `vector_name` has been filled with the right-hand side vector.

The various options have the following meaning:

seq_coef = c defines the input block for the coefficients.

Default value: 1.

problem = p defines the problem sequence number to be used for creation of right-hand side.

Default value: 1.

linear_subelements ensures that quadratic elements are treated as a cluster of linear elements. For example a 6-node triangle is locally subdivided into 4 3-node triangles. The right-hand side is built with these linear elements.

Default value: no subdivision.

3.2.3.2 Options corresponding to the keyword `prescribe_boundary_conditions`

The following options are available

```
VALUE = v
FUNC = k
POINTS = (Pi, Pj, Pk )
CURVES = (Ci, Cj, Ck )
SURFACES = (Si, Sj, Sk )
PROBLEM = s (Default 1)
SEQUENCE_NUMBER = s (Default the next one)
QUADRATIC, MAX = a, MEAN_VALUE = v, BOUNDARY_LAYER_WIDTH = b,
  DIFFERENCE = delta_T
HALF_QUADRATIC, MAX = a
POINTS = (Pi, Pj, Pk )
CURVES = (Ci, Cj, Ck )
SURFACES = (Si, Sj, Sk )
ZERO_LEVELSET i
OLD_VECTOR = m
SEQ_VECTORS = V1, V2, ...
```


Meaning of the various options:

SEQUENCE_NUMBER = s may only be used if none of the other options is used. This is meant for complicated cases. The boundary conditions are prescribed as described in the input block "ESSENTIAL BOUNDARY CONDITIONS" (3.2.5) with sequence number s . If the vector already exists the values of the vector are changed, otherwise the vector is set equal to zero before applying the essential boundary conditions.

If `sequence_number = s` is omitted implicitly the next sequence number is assumed. Hence in the first "call" of `prescribe_boundary_conditions` sequence number 1 and so on.

All other options For all other options the user is referred to Section (3.2.10).

Besides the options mentioned in this section, also the options provided in Section (3.2.10) under `functional description`, `degrees of freedom`, and `location part` are allowed.

3.2.3.3 commands to solve systems of equations

The following commands are available:

```
SOLVE_LINEAR_SYSTEM vector_name [options]
SOLVE_NONLINEAR_SYSTEM vector_name [options]
SOLVE_TIME_DEPENDENT_PROBLEM vector_name [sequence_number = s]
```

Meaning of the various commands:

SOLVE_LINEAR_SYSTEM options

The command `solve_linear_system` performs actually three independent steps.

Firstly information about the coefficients for the differential equation and natural boundary conditions is read in the input block "COEFFICIENTS" (3.2.6).

In the next step the matrix and right-hand-side vector is built, using the coefficients as described in the first step. Finally the system of linear equations is solved by the linear solver. Information about the solution process is read in the input block "SOLVE" (3.2.8) with sequence number `s` as indicated by `seq_solve = s`.

Before applying the command `solve_linear_system` it is necessary that the essential boundary conditions have already been filled into the solution vector `vector_name`. This may be done in several ways:

- By applying the command `prescribe_boundary_conditions` to `vector_name`
- By applying the command `create_vector` to `vector_name`
- By creating `vector_name` through another operation like a previous solve.

Of course the vector `vector_name` at input must correspond to problem `p`. `vector_name` must be of the type solution vector.

The result of the total operation is that `vector_name` has been filled with the solution of a linear differential equation.

For a description of the options, see 3.2.3.4

SOLVE_NONLINEAR_SYSTEM vector_name [options]

The command `solve_nonlinear_system` is comparable to the command `solve_linear_system`. However, in this case a non-linear system of equations is solved by an iteration process. In each step of the iteration process coefficients are filled, systems of equations are built and a system of linear equations is solved.

Before applying the command `solve_nonlinear_system` it is necessary that at least the essential boundary conditions have already been filled into the solution vector `vector_name`. Usually the iteration process expects that a complete initial estimate has been filled in `vector_name`. `vector_name` may be filled in the same way as described for the linear problems.

The result of this operation is that `vector_name` has been filled with the solution of a non-linear differential equation.

For a description of the options, see 3.2.3.5

SOLVE_TIME_DEPENDENT_PROBLEM vector_name [sequence_number = s]

The command `solve_time_dependent_problem` is comparable to the command `solve_linear_system`. However, in this case a time dependent problem is solved, which implies that a time integration is applied. Vector `vector_name` must contain the initial condition at $t = t_0$ and the solution is computed at $t = t_{end}$. This computed solution is again stored in the vector and hence replaces the initial condition. It is possible to write intermediate results to the file `sepcomp.out` for postprocessing purposes.

Information about the time dependent problem to be solved, as well as how this problem must be solved is given in the input block `TIME_INTEGRATION` (3.2.15) with sequence number

s. This block may not be omitted.

Vector `vector_name` must have been filled before with the initial condition and will be filled afterwards with the solution at the end time given in the input block `TIME_INTEGRATION`. If n coupled time-dependent equations are solved it is assumed that the corresponding vectors are the vectors `vector_name` and the next $n - 1$ vectors that are defined in the list of vector names.

3.2.3.4 Options corresponding to the keyword `solve_linear_system`

The following options are available

```

PROBLEM
SEQ_SOLVE
SEQ_COEF
DEFECT_CORRECTION
LINEAR_SUBELEMENTS
REACTION_FORCE
FEM_VECTOR
FEM_PRECONDITIONING
KEEP_MATRIX
DESTROY_MATRIX
REUSE_MATRIX
KEEP_VECTOR
DESTROY_VECTOR
REUSE_VECTOR
ISEQ_RHS
TRANSFORM_BOUNDARY_CONDITIONS

```

The various options have the following meaning:

vector_name defines the name of the solution vector.

seq_coef = c defines the input block for the coefficients.

If omitted implicitly the next sequence number is assumed.

problem = p defines the problem sequence number to be used for creation of right-hand side and matrix.

If `problem = p` is omitted implicitly the next sequence number is assumed.

seq_solve = s defines the input block for the linear solver.

If omitted implicitly the next sequence number is assumed.

defect_correction indicates that a defect correction method as described in Section 3.2.8 is applied.

linear_subelements ensures that quadratic elements are treated as a cluster of linear elements.

For example a 6-node triangle is locally subdivided into 4 3-node triangles. The matrix is built with these linear elements. Of course this option influences the type of approximation and hence the accuracy.

Mark that this option can only be applied if the number of degrees of freedom per point is constant.

reaction_force = Vi indicates that reaction forces must be computed. These reaction forces are stored in a vector with name `Vi`.

The reaction force is defined in the following way:

suppose that the degrees of freedom are split in a part of prescribed degrees of freedom \mathbf{u}_p and a part of free degrees of freedom \mathbf{u}_f . Then the system of equations can be written as:

$$\begin{bmatrix} \mathbf{S}_{ff} & \mathbf{S}_{fp} \\ \mathbf{S}_{pf} & \mathbf{S}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u}_f \\ \mathbf{u}_p \end{bmatrix} = \begin{bmatrix} \mathbf{r}_f \\ \mathbf{r}_p \end{bmatrix} \quad (3.2.3.1)$$

where \mathbf{S} is the matrix and \mathbf{r} the right-hand side.

Since the essential boundary conditions are prescribed only the part $\mathbf{S}_{ff}\mathbf{u}_f = \mathbf{r}_f - \mathbf{S}_{fp}\mathbf{u}_p$ is solved. The part $\mathbf{S}_{pf}\mathbf{u}_f + \mathbf{S}_{pp}\mathbf{u}_p = \mathbf{r}_p$ is usually not satisfied. This would only be the case if

we have a no-flux boundary condition instead of an essential boundary condition. The difference between left-hand side and right-hand side represents the flux through the boundary of the region where we have essential boundary conditions. This flux is denoted by the term reaction force and is defined as the vector

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{S}_{pf}\mathbf{u}_f + \mathbf{S}_{pp}\mathbf{u}_p - \mathbf{r}_p \end{bmatrix} \quad (3.2.3.2)$$

Hence the reaction force is only non-zero in points with prescribed boundary conditions. For those points it defines the flux through the boundary. For example in case of a Reynolds element for lubrication this is exactly the flow.

If the user is only interested in the part $\mathbf{S}_{pf}\mathbf{u}_f + \mathbf{S}_{pp}\mathbf{u}_p$, i.e. without subtraction of the right-hand side he must use `reaction_force = -Vi`, hence the vector name `Vi` is provided with a minus sign.

Remark: another possibility to compute the reaction forces is as a derived quantity. See the input for derivatives (3.2.11).

fem_vector = Vi This option makes only sense in the case that a spectral element mesh is used as well as the option `fem_preconditioning`. In that case the iterative solver starts by solving the finite element problem on the corresponding finite element mesh consisting of linear elements and using the same nodes as in the spectral mesh. This solution is used as initial approximation for the conjugate gradient solver. If the option `fem_vector = Vi` is given, the result of finite element solver is stored in the vector with name `Vi`, otherwise this result is destroyed.

fem_preconditioning This option makes only sense in the case that a spectral element mesh is used. It indicates that the spectral problem is solved iteratively by a conjugate gradient method using a so-called finite element preconditioner. This preconditioner is based on the matrix corresponding to the finite element mesh consisting of linear elements and using the same nodes as in the spectral mesh. Of course the number of elements in the finite element mesh is much larger than in the spectral mesh.

The use of the fem preconditioner is only possible if the matrix structure is based on the finite element grid and not on the spectral element grid. In order to get this structure, it is necessary to define `mesh=fem_mesh` in the input block `MATRIX`, see 3.2.4.

Mark that the iterative solution of the spectral problem is much more efficient both with respect to time as memory than the direct solution, especially for three-dimensional problems. For such problems the fem preconditioning is more or less necessary.

iseq_rhs = j If this option is used the right-hand side vector is stored in the vector with name `j`. Hence this vector may be used throughout the program.

keep_matrix Indicates that the matrix must be kept instead of destroyed after solving the system of equations.
Default value: destroy matrix.

destroy_matrix Indicates that the matrix must be destroyed after solving the system of equations.
Default value: destroy matrix.

reuse_matrix Indicates that the prior matrix must be reused. In case of a direct solver, the LU-decomposition of this matrix is reused.
Of course this option makes only sense if the matrix is kept at a prior call.
Default value: do not reuse.

keep_vector Indicates that the vector must be kept instead of destroyed after solving the system of equations.
Default value: destroy vector.

destroy_vector Indicates that the vector must be destroyed after solving the system of equations.
Default value: destroy vector.

reuse_vector Indicates that the prior vector must be reused. In case of a direct solver, the LU-decomposition of this vector is reused.

Of course this option makes only sense if the vector is kept at a prior call.

Default value: do not reuse.

iseq_rhs = Vi Indicates that the right-hand side vector already exists. The vector with name Vi is used as rhs vector.

transform_boundary_conditions or `transform_bc` makes only sense in the case of local transformations.

Suppose that the boundary conditions are filled in the original (for example Cartesian) form. In the linear solver we expect them to be given in the transformed form. With this statement the boundary conditions are transformed to the local coordinate system.

This statement may be of use inside a time loop, since after a solve step the solution is always transformed back into the global coordinate system.

Default: no transformation.

3.2.3.5 Options corresponding to the keyword `solve_nonlinear_system`

The following options are available

```
sequence_number
problem
reaction_force
maxiter
print_level
at_error
miniter
iteration_method
accuracy
seq_coef
criterion
```

The various options have the following meaning:

sequence_number = s is used, in complicated situations, which means that the standard options are not suitable. It refers to information about the coefficients for the differential equation and natural boundary conditions. This information is given in the input block "NONLINEAR EQUATIONS" (3.2.9) with sequence number s . This block also contains information about the linear solver to be applied. If `sequence_number` = s is omitted implicitly the next one is assumed.

`sequence_number` may only be used in combination with `reaction_force` and `problem`. All other keywords exclude the use of `sequence_number`.

problem = p defines the problem sequence number for which the non-linear system of equations must be created and solved. If omitted, the next one is assumed.

reaction_force = V_i has exactly the same meaning as in the command `solve_linear_system`. This reaction force is computed in each step of the iteration process. However, if a correction is computed per step, like for example if `newton` is applied, then the reaction force does not make sense and should not be computed.

The minus sign before V_i is defined in the same way as in the case of `solve_linear_system`. If this option is omitted, no reaction force is computed.

maxiter = m defines the maximum number of iterations that may be performed. If the number of iterations reaches this maximum value and the accuracy has not been reached, an error message is given and the program is terminated.

print_level = p gives the user the opportunity to indicate the amount of output information he wants from the iteration process. p may take the values 0, 1 or 2. The amount of output increases for increasing value of p .

at_error = e defines which action should be taken if the iteration process terminates because no convergence could be found. Possible values are:

```
stop
resume
```

If `stop` is used the iteration process is stopped if no convergence is found, otherwise (`resume`) means that control is given back to the main program and the result of the last iteration is used as solution.

miniter = m defines the minimum number of iterations that have to be carried out.

iteration_method = m defines the type of non-linear iteration method that is applied. Possible values for m are:

standard
newton

standard means that a standard iteration method is applied: The process starts with a given start vector \mathbf{u}^0 containing the boundary conditions. In each iteration $\mathbf{S}^k \mathbf{u}^{k+1} = \mathbf{f}^k$ is solved, where the solution vector \mathbf{u}^{k+1} also contains the given boundary conditions. The matrix \mathbf{S}^k and the right-hand-side vector \mathbf{f}^k may vary in each iteration step.

newton corresponds to the standard Newton (Raphson) method. This process is as follows:

```

start:   given start vector  $\mathbf{u}^0$ 
While not converged
  Solve correction  $\mathbf{S}^k \delta \mathbf{u} = \mathbf{f}^k$ 
  Correct          $\mathbf{u}^{k+1} = \mathbf{u}^k + \delta \mathbf{u}$ 

```

The correction in each step must satisfy homogeneous essential boundary conditions, since otherwise the essential boundary conditions are changed in the correction step.

accuracy = ϵ defines the accuracy at which the iteration terminates, provided the minimum number of iterations has been performed. Accuracy has been reached if the difference between two succeeding iterations is less than ϵ .

seq_coef = s refers to the input block s for the coefficients. This defines the coefficients for the non-linear differential equation.

Default: $s = 1$.

criterion = c defines the various types of criteria that can be used to terminate the iteration process. For a description of the possible values of c , see Section [3.2.9](#)

Default: $c = \text{abs}$.

3.2.3.6 commands to compute quantities that can be derived from previously computed vectors

The following commands are available:

```

DERIVATIVES vector_name [seq_coef = c] [seq_deriv = s] [problem=p]
    [scalar_name] [icheld = i] [input_vector = v] [ix = j]
    [type_output = j] [points (pi, pk, ...)] [curves (ci, cj, ...)]
    [surfaces ( sk, sl, ... ) ] [zero_level_set i]
INTEGRAL scalar_name [seq_coef = c] [seq_integral = i] [vector_name]
    [scal_min = m ] [scal_max = n] [icheli = i] [active_level_set i]
    [non_active_level_set i] [degfd i ]
BOUNDARY_INTEGRAL, [seq_boun_integral = i] [vector_name] [scalar1= j ]
    [scalar2 = m ] [scalar3 = n] [ curves = (ci, cj, ck ) ]
    [ surfaces = (si, sj, sk ) ] [ ichint = i ]

```

Meaning of the various commands:

DERIVATIVES vector_name [seq_coef = c] [seq_deriv = s] [problem=p] [scalar_name]

For simple cases see (3.2.3.10).

The command derivatives may be used to create the vector `vector_name` as derived quantity of previously constructed vectors. For many derived quantities it is necessary to define coefficients which are used in the computation process. Consult the manual STANDARD PROBLEMS to check if and which coefficients are required for a specific derived quantity.

These coefficients are defined by the input block "COEFFICIENTS" (3.2.6) with sequence number `c`. If `seq_coef = c` is omitted it is assumed that no coefficients are needed.

Problem = `p` defines the problem sequence number that is used to compute the derived quantities. If omitted the next one is assumed.

`scalar_name` must be used to indicate that the boundary integral to be computed, provided one has to be computed, must be stored in the variable with name `scalar_name`

If this option is used, no vector name must be given.

Input concerning the derived quantities to be computed is defined in the input block "DERIVATIVES" (3.2.11) with sequence number `s`. If `s` is omitted the next one is assumed.

The result of this operation is that a vector `vector_name` has been created.

For an example, see Section 6.2.5.

All other options are described in Section (3.2.11).

INTEGRAL scalar_name [seq_coef = c] [seq_integral = i] [vector_name] [scal_min = m] [scal_max = n] [icheli = i] [active_level_set i] [non_active_level_set i] [degfd i]

For simple cases see (3.2.3.11).

The command integral may be used to compute scalar `scalar_name` as integral over vector `vector_name`. The sequence of integral and `scalar_name` may be interchanged, hence `scalar_name = integral (options)`.

For some integrals it is necessary to define coefficients which are used in the integration process. Consult the manual STANDARD PROBLEMS to check if and which coefficients are required for a specific integral. These coefficients are defined by the input block "COEFFICIENTS" (3.2.6) with sequence number `c`. If `seq_coef = c` is omitted it is assumed that no coefficients are needed.

Input concerning the integral to be computed is defined in the input block "INTEGRALS" (3.2.12) with sequence number `s`. If `s` is omitted the next one is assumed.

`scal_min = m` defines that if the minimum value over the element integrals must be computed, then this minimum value should be stored in the scalar with name `m`. Whether the minimum is computed is defined in the input block INTEGRALS.

`scal_max = n` defines that if the maximum value over the element integrals must be computed, then this maximum value should be stored in the scalar with name `n`. Whether the maximum

is computed is defined in the input block "INTEGRALS".

The result of this operation is that `scalar_name` has got a value and possibly the scalars m and n too.

For `icheli = i` we refer to (3.2.12).

The options `active_level_set i` and `non_active_level_set i` indicate that the integration is carried out over the region where either the level set function is positive or negative.

For an example, see Section 6.2.5.

BOUNDARY INTEGRAL ,[seq.boun.integral = i] [vector_name] [scalar1= j] [scalar2 = m]
[scalar3 = n]

For simple cases see (3.2.3.11).

The command `boundary_integral` may be used to compute SCALAR j as an integral of `vector_name` over (a part of) the boundary.

Input concerning the boundary integral to be computed is defined in the input block "BOUNDARY INTEGRAL" (3.2.14) with sequence number s . If s is omitted the next one is assumed.

If the integral to be computed is a vector then the second component is stored in SCALAR m . In the same way the third component is stored in SCALAR n .

The result of this operation is that the SCALAR j has got a value and possibly the scalars m and n too.

For all other parameters the user is referred to Section (3.2.14). For an example, see Section 6.2.5.

3.2.3.7 commands for special computations

The following commands are available:

```

COMPUTE_EIGENVALUES [options]
COMPUTE_CAPACITY vector_name [, sequence_number = k]
SOLVE_INVERSE_PROBLEM vector_name [, sequence_number = k]
COMPUTE_CONTACT_SURFACE [, sequence_number = k]
COMPUTE_PRINCIPAL_STRESSES [options]
COMPUTE_BUBBLE [options]

```

Meaning of the various commands:

COMPUTE_EIGENVALUES , options

This command activates the computation of eigenvalues and eigenvectors.

The following options are available:

```
vector_name, scalar_name, sequence_number = k, problem = s, num_eigval = s
```

These options must be given on the same line.

The options have the following meaning:

sequence_number = k The sequence number k refers to the input block EIGENVALUES (3.2.18) in which it is defined how the eigenvalues and possibly eigenvectors must be computed.

problem = s defines the problem number that must be used for the computation of the eigenvalues.

Default value: 1

vector_name defines the vector in which the first eigenvector must be stored. All other eigenvectors are stored sequentially in the next vectors from the list defined in the part **vector_names**.

Default value: 1

scalar_name defines the scalar in which the first eigenvalue must be stored. All other eigenvalues are stored sequentially in the next scalars from the list defined in the part **variables**.

Default value: 1

num_eigval = s defines the number of eigenvalues that must be computed.

Default value: 1

COMPUTE_CAPACITY , options

This command activates the computation of the capacities of electrodes placed around an object.

The following options are available:

```
vector_name, sequence_number = k
```

These options must be given on the same line.

The options have the following meaning:

sequence_number = k The sequence number k refers to the input block CAPACITY (3.2.19) in which it is defined how the capacities of the electrodes must be computed.

Default value: 1

vector_name defines the vector in which the capacities must be stored.

SOLVE_INVERSE_PROBLEM , options

This command activates the computation of an inverse problem, which is a problem in which a coefficient are unknown and must be computed using measured values.

The following options are available:

`vector_name, sequence_number = k`

These options must be given on the same line.

The options have the following meaning:

sequence_number = k The sequence number k refers to the input block `INVERSE_PROBLEM` (3.2.20) in which it is defined how the inverse problem must be solved.

Default value: 1

vector_name defines the vector in which the unknown coefficient must be stored.

COMPUTE_CONTACT_SURFACE , options

This command activates the contact algorithm.

The following options are available:

`sequence_number = k`

The sequence number k refers to the input block `CONTACT` (3.2.16) in which the contact algorithm is defined.

COMPUTE_PRINCIPAL_STRESSES , options

This command activates the computation of the principal stresses from an already computed stress tensor.

The following options are available:

`STRESS_VECTOR_name`
`EIGENVALUES_name`
`EIGENVECTORS_name`
`SCALING`

These options must be given on the same line.

The options have the following meaning:

STRESS_VECTOR_name defines the stress vector that is used to compute the principal stresses. This stress tensor must have been filled before.

EIGENVALUES = $j1$ If the user wants to store the eigenvalues of the stress tensor per node, vertex or element, depending on the storage in the stress tensor, he must add this statement. The vector of eigenvalues is stored in the solution vector as a vector of special structure with 2 or 3 degrees of freedom per node, vertex or element. **EIGENVALUES_name** defines the vector name.

Default value: 0

EIGENVECTORS_name If the user wants to store the eigenvectors of the stress tensor per node, vertex or element, depending on the storage in the stress tensor, he must add this statement. The vector of eigenvalues is stored in the solution vector as a vector of special structure with 4 or 9 degrees of freedom per node, vertex or element. **EIGENVECTORS_name** defines the vector name.

Default value: no eigenvectors are stored

SCALING If scaling is given in combination with **EIGENVECTORS_name**, the eigenvectors are multiplied by the corresponding eigenvalues. In fact this gives the usual set of principal stresses.

Default value: no

Mark that at least one of the integers $j1$ and $j2$ must be unequal to zero.

COMPUTE_BUBBLE , options

With this command you can compute the regions where the solution extends a given threshold. The number of these bubbles is computed as well as the area of all these bubbles. If required a plot of the bubbles can be made.

Available options (all in one line):

```
VECTOR_name  
THRESHOLD = t  
PLOT  
COLOR = c  
YFACT = y  
TEXTX = '..'  
TEXTY = '..'  
DEGFD = d  
FILENAME = '...'
```

The options have the following meaning:

VECTOR_name defines the vector that is considered.

Default value: first vector,

THRESHOLD = t defines the threshold value t . All values above this value belong to a bubble.

Default value: 0

PLOT if used all bubbles are plotted with one color.

Default value: no plot

COLOR = c defines the sequence number of the color to be used in case a plot is made (device dependent).

Default value: 10

YFACT = y defines the scaling of the y-coordinates in the plot.

Default value: 1

TEXTX = '..' gives the text to be plotted along the x-axis.

Default value: x

TEXTY = '..' gives the text to be plotted along the y-axis.

Default value: y

DEGFD = d defines the degree of freedom in the vector that is used to compute the bubble.

Default value: 1

FILENAME = '..' defines an output file containing for each output time a line containing the number of bubbles, the time and the area of all these bubbles. This file is meant as input file for matlab.

Default value: bubble.dat

3.2.3.8 commands to control the output to the output files used by a postprocessor

The following commands are available:

```
OUTPUT vector_name [sequence_number = s] ['file_name']  
NO_OUTPUT
```

Meaning of the various commands:

OUTPUT Vi [sequence_number = s] [file = 'file_name']

The command **OUTPUT** forces the vector Vi and, depending on the definition in the input block "OUTPUT", the next vectors, to be written to the file `sepcomp.out` for post-processing purposes.

Information about what output should be written must be stored in the input block "OUTPUT" (3.2.13) with sequence number s. If s is omitted the next one is assumed.

With the option `file = 'file_name'`, the user may indicate the name of the files to which the information is written. For example if `file_name = 'userfile'`, then the information is written to two files: `userfile.inf` and `userfile.out`.

These files can be used in program `SEPPOST`.

Default value: `sepcomp`

NO_OUTPUT If the keyword **OUTPUT** is not present still some default output is written to the file `sepcomp.out`. However, in case the keyword **NO_OUTPUT** is found, this output is suppressed and no file `sepcomp.out` is filled.

3.2.3.9 commands to manipulate vectors or scalars, including copying and mapping

The following commands are available:

```

Vi [options]
Sj [options]
STRING = ...
COPY Vj Vk, degfd1 = l, degfd2 = m
COPY_COOR Vj, problem = p
CHANGE_PROBLEM Vj, problem = p

```

Meaning of the various commands:

Vi [options]

The command `Vi` computes the vector with name `Vi` by manipulating other vectors. Which vectors are manipulated and how is defined by the options.

For a description of the options, see [3.2.3.10](#)

Si options]

The command `Si = ..` computes `Si` by manipulation of vectors or scalars. Which vectors are manipulated and how is defined by the options.

For a description of the options, see [3.2.3.11](#)

STRING = makes the string at the left-hand side equal to the one on the right-hand side. The string on the left-hand side must be a string from the set of strings given the input block `CONSTANTS`. (See Section [3.4](#)).

The right-hand side may consist of a number of strings but also reals, scalars or integers separated by spaces. The values of the reals and so on are substituted at the moment the string is created.

Example:

```
stringa = 'the value of alpha is ' alpha
```

COPY Vj Vk, degfd1 = l, degfd2 = m

The command `COPY` copies the `Vj` into `Vk`.

If `degfd1 = l, degfd2 = m` is omitted the complete `Vj` is copied into `Vk` and `Vk` gets exactly the same structure as `Vj`. `Vj` must have been filled, `Vk` may have been filled before, in which case the contents are overwritten.

If at least one of `degfd1` or `degfd2` is given, then `Vj` must have been filled before and `Vk` must have been created before. This means that actually `Vk` has also been filled.

In this case the l^{th} degree of freedom in each point of `Vj` is copied in the m^{th} degree of freedom in each point of `Vk`. The structure of array `Vk` is kept.

Default values for `l` and `m` are 1, provided at least one of the two is given.

COPY_COOR Vj problem p

creates the vector `Vj` with exactly `ndim` degrees of freedom, for each point in the domain defined by problem `p`, with `ndim` the dimension of space. In this vector the coordinates of the mesh are stored.

If problem `p` is neglected, problem 1 is used. Usually this problem refers to the whole domain, but for example if type number 0 is used for some submeshes, these parts of the domain are skipped.

CHANGE_PROBLEM Vj problem p

With this command you may change the problem number of `Vj` into number `p`. Use of this command must be done with great care, since it may give unwanted effects.

3.2.3.10 Options corresponding to the keyword `Vector_name =`

The following options are available:

```

MODULUS Vj [degfd k] MULTIPLY = [constant c]
PHASE Vj [degfd k] MULTIPLY = [constant c]
LENGTH Vj
SUBTRACT [constant c] [Sj]
MULTIPLY m
c1 Vj + c2 Vk
Vj * Vk
sign(Vj)
sign(Vk) * Vj
Vk / Vj
EXTRACT Vj DEGFD k
REAL Vj
IMAGINARY Vj
CONJUGATE Vj
FUNC Vj1 Vj2
INNER_PRODUCT Vj1 Vj2, degfd = i
MAP Vj, PROBLEM = p, TYPE = t
ELEMENT_GROUPS = (s1, s2, ...)
SKIP_ELEMENT_GROUPS = (s1, s2, ...)
Vj
MESH_VELOCITY ( VECTOR1, VECTOR2 )
INTEGRATED_TO_NODAL Vj
TRANSFORM_TO_NORMALDIR Vj, CURVES = (Ci, Cj, ... ), SMOOTH_END_POINTS
STRESS_VECTOR Vj, CURVES = (Ci, Cj, ... )
DERIVATIVES ( V1, S1, icheld = i, problem = p, seq_coef = s1,
              seq_deriv = s2, type_output = j, points (pi, pk, ...),
              curves (ci, cj, ...), surfaces ( sk, sl, ... ) )
POLAR (Vj)
GRADient (Vj)
STREAM_FUNCTION ( Vj, start_node = s, start_value = f )
RIGHT_HAND_SIDE, problem = p, seq_coef = i

```

These options have the following meaning:

MODULUS computes V_i as modulus of the complex vector V_j . If `MULTIPLY` is given V_i is multiplied by the constant c . If `DEGFD k` is given only the $DEGFD^{th}$ degree of freedom per point is taken into account.

PHASE computes V_i as phase of the complex vector V_j . If `MULTIPLY` is given V_i is multiplied by the constant c . `DEGFD k` is given only the $DEGFD^{th}$ degree of freedom per point is taken into account.

LENGTH computes the component-wise length of the vector V_i . For two-dimensional problems it is supposed that the first two components of V_i per point must be used, for three-dimensional problems the first three ones.

The result of this operation is given by

$$u_{out}(i) = (|u_1(i)|^2 + |u_2(i)|^2 + |u_3(i)|^2)^{\frac{1}{2}}$$

per nodal point i , i.e. the Euclidean vector

SUBTRACT subtracts either the constant c or the scalar S_j from the vector V_i . If `DEGFD k` is given only the $DEGFD^{th}$ degree of freedom per point is taken into account.

c1 Vj + c2 Vk computes $V_i = c_1 V_j + c_2 V_k$, where V_j and V_k are vectors and c_1 and c_2 are either constants or scalars. One of the constants may be zero or not present and instead of a plus sign also a minus sign may be used.

Both vectors V_j and V_k must be real or both must be complex. If DEGF D k is given only the DEGF D th degree of freedom per point is taken into account.

Typical examples are

```

Vi = 10 * Vj - 3.5 * Vk
Vi = Vj - Vk
Vi = Vj + Vk
potential = 2 * potential + 3* temperature
Vi = Vj

```

The last example is identical to copying of a vector.

Vj * Vk creates a new vector with the same length as V_j and V_k defined by $V_i(l) = V_j(l)*V_k(l)$ for all l .

sign(Vj) creates a new vector with the same length as V_j defined by $V_i(l) = \text{sign}(V_j(l))$, where sign is 0 if $V_j(l) = 0$.

sign(Vj) * Vk creates a new vector with the same length as V_j and V_k defined by $V_i(l) = \text{sign}(V_j(l))*V_k(l)$ for all l .

Vk / Vj creates a new vector with the same length as V_j and V_k defined by $V_i(l) = V_k(l)/V_j(l)$ for all l .

EXTRACT Vj DEGF D k puts degree of freedom k of V_j into a V_i consisting of one unknown in each point. It is supposed that the original vector has the degree of freedom in each point.

REAL Vj computes V_i as real part of the complex V_j .

IMAGINARY Vj computes V_i as imaginary part of the complex V_j .

CONJUGATE Vj computes V_i as the complex conjugate of the complex V_j .

INNER_PRODUCT Vj1 Vj2 computes the V_i as point-wise dot product of the vectors V_{j1} and V_{j2} . This means that V_{j1} and V_{j2} must have the same number of degrees of freedom per point. The result is a vector with one degree of freedom per point.

If $\text{degfd} = i$ is given only the components 1 to i are used to compute the dot product per point, if in that point there are more than i degrees of freedom.

FUNC Vj1 Vj2 computes the V_i as function of the vectors V_{j1} and V_{j2} . The function itself must be defined by the user through the user subroutine FUNALG (real case) or FUNALC (complex case) described in Section 3.3.1.

MAP Vj, PROBLEM = p , TYPE = t maps the V_i onto the V_j . The V_j may have a different structure than V_i . The structure of V_j is defined by the problem number p (default value: 1) and the type t (default value: 0).

$t = 0$ means that V_j is of the type of a solution vector,

$t = k (> 0)$ means that V_j is a vector of special structure of type k .

If the number of degrees of freedom in a point are different in both vectors, only the common ones are copied.

$t = k (< 0)$ means that V_j is a vector of special structure defined per element. The number of degrees of freedom per element is equal to $-k$. These degrees of freedom are computed by computing the mean value per element, of the $-k^{\text{th}}$ degree of freedom in V_i .

ELEMENT_GROUPS = (s1, s2, ...) defines which element groups must be used to compute the vector. Only element groups defined in the mesh generation part are used.

The default value for **ELEMENT_GROUPS** is all element groups.

Of course this option can only be used in combination with other options. The keyword may not be used in combination with **SKIP_ELEMENT_GROUPS**

SKIP_ELEMENT_GROUPS = (s1, s2, ...) defines which element groups must be skipped when the vector is computed. The default value for **SKIP_ELEMENT_GROUPS** is skip no element groups.

Of course this option can only be used in combination with other options. The keyword may not be used in combination with **SKIP_ELEMENT_GROUPS**

V_j , means that V_i is a copy of V_j .

For example `u_old = u`.

MESH_VELOCITY (V1, V2) creates the mesh velocity vector defined by $\frac{\mathbf{x}^{n+1}-\mathbf{x}^n}{\Delta t}$, with \mathbf{x}^n the coordinates at time level n and Δt the time step.

V1 must be a vector containing the coordinates of the mesh at the new time level $n + 1$ and **V2** at the prior time level n . It is sufficient if this vector is defined over the region where the mesh velocity is required. In other words usually these vectors have to be defined over the fluid domain only. The mesh velocity V_i gets the same structure as the coordinate vectors **V1** and **V2**.

INTEGRATED_TO_NODAL Vj replaces the vector V_j which is supposed to be an integrated quantity along the outer boundary to a vector V_i which is supposed to be defined per node.

A typical example is the reaction force, which is defined as a flux through the boundary and hence as an integral over the derivative. After applying this command you get the nodal point values of the flux along the boundary. This is done by multiplying by the inverse of the boundary mass matrix.

This option may be combined with `zero_level_set i` if it should be carried out for all nodes with level set function equal to zero. Otherwise the whole outer boundary is used.

TRANSFORM_TO_NORMALDIR Vj transforms the vector V_j which is defined in the usual coordinates (usually Cartesian) into normal and tangential components along the curves defined in the part **CURVES**. So in R^2 the result is a vector of two components, the first one the normal component the second one the tangential component, defined along these curves. The value of V_i in other points is the same as the values of V_j .

If the option `smooth_end_points` is given it is checked if the first and last point of the curve have values that differ much from the neighboring nodes. If so an extrapolation is used from two neighboring points to compute the values in the end points. This option makes sense if the value in the end point is due to an unwanted computation. For example if the the vector is a reaction force it is possible that the value in an end point depends on two different curves, both with essential boundary conditions. It is possible that the reaction force in that case contains an influence of two different fluxes. Then smoothing may be an option.

POLAR Vj transforms the vector V_j from Cartesian coordinates to polar coordinates and stores the result in V_i .

GRADient Vj computes the gradient of V_j in all nodes and stores the result in the vector V_i .

The first 4 letters of the keyword are significant, hence also **GRAD** may be used.

STREAM_FUNCTION (Vj, options) computes the stream function of the incompressible vector V_j .

the following options are available:

```
start_node = s
start_value = f
```

start_node = s defines the starting node in which the start value is given.
Default value: 1

start_value = f defines the corresponding start value.
Default value: 0

STRESS_VECTOR Vj transforms the stress tensor V_j with 6 components per node to a vector V_i with two (or three) components per node. The stress tensor is defined over the whole region. The stress vector is zero everywhere except in the curves defined by the part **CURVES**. The stress vector is defined as the dot product of the stress and the normal along the curves. In order to get the normal and shear stress it is necessary to apply the transformation **transform_to_normaldir**. Mark that in case of Navier-Stokes the stress tensor is defined without the pressure contribution.

DERIVATIVES options computes V_i as derivative of vector V_j . The following options are available:

```
Vj
Sk
icheld = i
problem = p
seq_coef = s1
seq_deriv = s2
ix = j
TYPE_OUTPUT = j
POINTS (P1, P2, ...)
CURVES (C1, C2, ...)
SURFACES (S1, S2, ...)
ZERO_LEVEL_SET i
```

Meaning of these options:

Vj defines the vector from which the derivative must be computed.

Sk must be used to indicate that the boundary integral to be computed, provided one has to be computed, must be stored in the variable with name **Sk**
If this option is used, no vector name must be given.

problem = p defines the problem number for the output vector V_i .

seq_coef = s1 For some derivatives it is necessary to define coefficients which are used in the computation process. Consult the manual **STANDARD PROBLEMS** to check if and which coefficients are required for a specific derivative. These coefficients are defined by the input block "COEFFICIENTS" (3.2.6) with sequence number c . If **seq_coef = c** is omitted it is assumed that no coefficients are needed.

seq_deriv = s2 is only used in complex situations. It defines the input block for derivatives as described in Section (3.2.11).

This parameter can not be used if **icheld** is given.

For all other options see Section (3.2.11).

RIGHT_HAND_SIDE, options creates a right-hand-side vector defined by the options. This vector may be used for example in time dependent problems to add to an existing right-hand side.

the following options are available:

```
problem = p
seq_coef = i
```

problem = p defines the problem number.
Default value: 1

seq_coef = i defines the sequence number for the input block coefficients.
Default value: 0

3.2.3.11 Options corresponding to the keyword `Scalar_name =`

The following options are available:

```

value
(func=k)
min ( f1, f2, f3, ... )
max ( f1, f2, f3, ... )
constant(i)
constant(sj)
sk(j)
expression

integral vi, options
boundary_integral vj, options
boundary_sum vj, options

xxx_norm (vk1, vk2) [degfd k]
average vj [degfd k]
dot_product, vector1 = vk1, vector2 = vk2 [degfd k]
min_max vj, [scal_max = i1] [coor_min = i2] [coor_max = i3] &
  [abs_value] [degfd k]
extract_value vj, [node = i1], [user_point =i1], degfd k
mean_area
min_area
volume
point_number ( zero_levelset i )
mean_value vi
standard_deviation vi
intersection_integral vi, degfdi, origin = (0_x,0_y), angle = a&
  end_point=(e_x,e_y), length = 1

element_groups = (s1, s2, ...)
skip_element_groups = (s1, s2, ... )

get_time

```

These options have the following meaning:

value If a value is given explicitly, the scalar gets this value.

FUNC=k gives the scalar the value `FUNCSCAL (k, SCALARS)`, which means that it may be a function of the other scalars.

See Section 3.3.2 for a description of `FUNCSCAL`.

min or max (F1,F2,F3,...) means that the scalar gets the minimum respectively maximum value of `F1, F2, F3` and so on. `F1, F2, F3, ..` may be either numbers or of the shape `Sj`, referring to scalar `j`. Of course scalar `j` must have been given a value before.

constant(*i*) gives the scalar the value of the i^{th} entry of the real or integer array `constant` which has been declared in the block constants. i must be an integer value in the appropriate range. Instead of i also `Si` may be used, where `Si` is a variable (scalar). The value of the `Si` is evaluated at the moment the statement is carried out.

This construction is for example meant for a combination with a for loop.

See Section 6.2.12 for an example.

`S(i)` has exactly the same meaning as `constant(i)`, however, instead of a real or integer array, a scalar (variable) array is used.

expression means that the Scalar gets the value of the expression. This expression must satisfy the rules of Section 1.4.

In this case not only constants but also scalars may be used in the expression. These scalars are evaluated at the moment the statement is reached.

Examples are:

```
si = 3* b
sj1 = cos( Si + v)
```

INTEGRAL Vi, options computes S_i as integral over the vector with name V_i .

The following options are available

```
seq_coef
seq_integral
scal_min
scal_max
icheli
active_level_set i
non_active_level_set i
degfd i
```

Meaning of these options

seq_coef For some integrals it is necessary to define coefficients which are used in the integration process. Consult the manual STANDARD PROBLEMS to check if and which coefficients are required for a specific integral. These coefficients are defined by the input block "COEFFICIENTS" (3.2.6) with sequence number c . If `seq_coef = c` is omitted it is assumed that no coefficients are needed.

seq_integral In complicated cases the input concerning the integral to be computed must be defined in the input block "INTEGRALS" (3.2.12) with sequence number s . Keywords of that input block may only be used if `seq_integral` is omitted.

scal_min = m defines that if the minimum value over the element integrals must be computed, then this minimum value should be stored in the scalar with name m .

scal_max = n defines that if the maximum value over the element integrals must be computed, then this maximum value should be stored in the scalar with name n .

icheli defines the type of integral to be computed. This parameter is passed to the element subroutines which decide which integral corresponds to the value s of ICHELI. With respect to the standard elements provided by SEPRAN, it is necessary to consult the manual Standard problems for the meaning of ICHELI in specific cases. If user elements are defined (type numbers between 1 and 99), the parameter ICHELI is passed undisturbed to the element subroutine.

The default value for ICHELI is 1.

active_level_set i reduces the computation of the integral to the part where levelset i is active as defined in the problem input part. This makes only sense in case a levelset method is used.

non_active_level_set i reduces the computation of the integral to the part where levelset i is not active.

degfd i implies that the integral is computed for degree of freedom i only.

Default value: 0 (i.e. all degrees of freedom).

boundary_integral Vj, options computes S_i as boundary integral over the vector with name V_i .

The following options are available

```

seq_boun_integral = i
scalar1 = s
scalar2 = s
curves = (Ci, Cj, Ck )
surfaces = (Si, Sj, Sck )
degfd j
ichint = k

```

Meaning of these options

seq_boun_integral = s In complicated cases the input concerning the boundary integral to be computed is defined in the input block "BOUNDARY_INTEGRAL" (3.2.14) with sequence number s . Keywords of that input block may only be used if seq_boun_integral is omitted.

scalar2 = m If the integral to be computed is a vector then the second component is stored in SCALAR m .

scalar3 = n In the same way the third component is stored in SCALAR n .

curves = (Ci, Cj, Ck) defines over which curves the integral must be computed. This subkeyword may only be used in R^2 .

surfaces = (Si, Sj, Sk) defines over which surfaces the integral must be computed.

degfd j defines the degree of freedom per point that must be used to compute the integral over the vector.
Default value: 0

ichint = k defines the type of integral to be computed. For a description see: (3.2.14).

boundary_sum Vj, options computes Si as sum over the values of the vector with name Vi over the boundary indicated. The same options as for **boundary_integral** may be applied, except for ichint which has always the value 8.

This option is useful in the case of a reaction force. The sum of the reaction forces along a boundary is equal to the flux through that boundary.

xxx_NORM computes Si as norm of the difference $Vk1 - Vk2$. If only one vector $Vk1$ is given, the norm of this vector is computed. The value of xxx defines the type of norm to be used. Possible values:

ONE_NORM $\| \mathbf{u} \|_1 = \sum_{i=1}^N | u_i |$, where \mathbf{u} is either the vector $Vk1$ or $Vk1 - Vk2$.

This is equivalent to NORM1.

TWO_NORM $\| \mathbf{u} \|_2 = \left(\sum_{i=1}^N u_i^2 \right)^{\frac{1}{2}}$

This is equivalent to NORM2.

INF_NORM $\| \mathbf{u} \|_{\infty} = \max_{1 \leq i \leq N} | u_i |$

This is equivalent to NORM3.

MEAN_ONE_NORM $\| \mathbf{u} \| = \frac{\| \mathbf{u} \|_1}{N}$

This is equivalent to NORM4.

MEAN_TWO_NORM $\| \mathbf{u} \| = \frac{\| \mathbf{u} \|_2}{N}$

This is equivalent to NORM5.

If DEGFD k is given only the $DEGFD^{th}$ degree of freedom per point is taken into account.

AVERAGE computes Si as the average value of the Vj . If DEGFD k is given only the $DEGFD^{th}$ degree of freedom per point is taken into account.

DOT_PRODUCT computes Si as the dot product of the vectors $Vk1$ and $Vk2$.

MIN_MAX computes the minimum and maximum value of the V_j . The minimum is stored in the scalar i .

If **SCAL_MAX** = $i1$ is given the maximum is stored in the scalar $Si1$. If **ABS_VALUE** is given the minimum and maximum of the absolute value of the entries of the vector is computed.

If **COOR_MIN** = $i2$ is given, not only the minimum is computed but also the co-ordinates for which this minimum is reached. These co-ordinates are stored in the scalars with sequence number $i2, i2 + 1, \dots, i2 + NDIM - 1$, with $NDIM$ the dimension of the space.

If **COOR_MAX** = $i3$ is given, not only the maximum is computed but also the co-ordinates for which this maximum is reached. These co-ordinates are stored in the scalars $Si3$ and the next $NDIM-1$ scalars in the list of scalars, with $NDIM$ the dimension of the space.

EXTRACT_VALUE extracts the value of k^{th} degree of freedom of the V_j in the node $i1$ or the user point $i1$ and stores this in the scalar Si .

Either node or user_point must be used.

MEAN_AREA computes the mean area of all elements.

MIN_AREA returns the smallest area of all elements.

VOLUME returns the area of all elements.

POINT_NUMBER (zero_levelset i) returns with the node number of the node with zero level set. This option makes only sense if the levelset method is used in R^1 .

MEAN_VALUE Vi returns with the mean value of vector Vi . This may be combined with degree of freedom given and zero_levelset.

STANDARD_DEVIATION Vi returns with the standard deviation of vector Vi . The same possibilities as for mean_value are present.

INTERSECTION_INTEGRAL Vi options, computes the integral off the vector Vi over the intersection of a 2d mesh with an intersection line defined by the options.

The following options are available:

```
degfdi
origin = (0_x,0_y)
end_point = (E_x,E_y)
angle = a
length = l
```

Meaning of these options:

degfd i defines the degree of freedom of vector Vi that must be used.

Default value: 1

origin = (O_x, O_y) defines the origin of the intersection line.

Default values: if 0_x and 0_y are known as constants or variables their values are used, otherwise $(0,0)$.

angle = a defines the angle of the line with respect to the positive x-axis.

Default value: if **angle** is known as constant or variable its value is used, otherwise 0.

end_point = (E_x, E_y) defines the end point of the intersection line. If omitted the end point is the final intersection of the line with the 2d mesh.

Default values: if E_x and E_y are known as constants or variables their values are used, otherwise no end points are used.

length = l is an alternative for **end_point**. If available it defines the length of the intersection line.

This is an alternative for prescribing the end point.

Both keywords are mutually exclusive.

ELEMENT_GROUPS = (s1, s2, ...) defines which element groups must be used to compute the scalar. Only element groups defined in the mesh generation part are used.

The default value for **ELEMENT_GROUPS** is all element groups.

Of course this option can only be used in combination with other options. The keyword may not be used in combination with **SKIP_ELEMENT_GROUPS**

SKIP_ELEMENT_GROUPS = (s1, s2, ...) defines which element groups must be skipped when the scalar is computed. The default value for **SKIP_ELEMENT_GROUPS** is skip no element groups.

Of course this option can only be used in combination with other options. The keyword may not be used in combination with **SKIP_ELEMENT_GROUPS**

GET_TIME gives the scalar the value of the present time.

3.2.3.12 print commands

The following commands are available:

```

PRINT Sj1, Sj2, ... [text='some text'], FILE = 'name_of_file', APPEND
PRINT Vj [options]
PRINT 'text between quotes', FILE = 'name_of_file', APPEND
PRINT TIME
PRINT MESH [options]
WRITE_TO_FILE, APPEND
CLOSE_FILE
PRINT zero_levelset i, FILE = 'name_of_file', APPEND
PRINT TIME_HISTORY Vj, FILE = 'name_of_file', APPEND
PRINT_INTERSECTION V1, options
APPEND

```

Meaning of the various commands:

PRINT Sj1, Sj2, ... [text='some text']

The command PRINT Sj1, Sj2, ... prints the value of Sj1, Sj2, ... to the output file. If text is given it should be followed by some text between quotes. This text is used to identify the scalar to be printed in the following way:

```
text = Sj1, Sj2, ...
```

where Sj1 denotes the value of Sj1. If FILE = 'name_of_file' is given the output is written to the file with the name indicated by 'name_of_file'.

If append is given the output is appended to this file, otherwise writing starts at the beginning of the file and prior contents are lost.

PRINT Vj [options]

The command PRINT Vj prints the value of Vj to the output file. The following options are available:

```

text = 't'
region = (xmin,xmax,ymin,ymax,zmin,zmax)
points = p1, p2, p3, ...
curves = c1, c2, cn, ...
surfaces = s1, s3, s5, ...
volumes = v1, v2, v6, ...
zero_levelset i
nodes = (i,j,k,...)
min_value = ..
max_value = ..
degfd = k
normal_component
tangential_component
suppress_coordinates
suppress_header
suppress_zeros
suppress_nodes
sequence = (y)
type_coordinates = c
type_format = i
equidistant_grid, distance = ( dx, dy, dz )
file = 'name_file'

```

```
file_format = form
append
```

These options have the following meaning:

text should be followed by some text between quotes. This text is used to identify the vector to be printed.

If omitted the name of the vector as defined in the block constants (vector_names) is used.

region If this option is used only the points within the region $xmin \leq x \leq xmax, ymin \leq y \leq ymax, zmin \leq z \leq zmax$ are printed.

points followed by P_i, P_j, P_k, \dots ensures that the printing of the solution is restricted to the user points given in the list.

curves followed by C_i, C_j, C_k, \dots ensures that the printing of the solution is restricted to the curves given in the list.

surfaces followed by S_i, S_j, S_k, \dots ensures that the printing of the solution is restricted to the surfaces given in the list.

volumes followed by V_i, V_j, V_k, \dots ensures that the printing of the solution is restricted to the volumes given in the list.

zero_levelset i means that only nodes in the interface with level set function value 0 are printed.

nodes = (i, j, k, ...) prints values in the given nodal points only.

min_value = c prints only the values larger than or equal to c .

max_value = c prints only the values which are at most equal to c .

degfd defines which degree of freedom must be printed. If omitted all degrees of freedom in the points requested are printed.

normal_component may only be used if curves or surfaces is given. Furthermore there must be at least $ndim$ unknowns in each point at the boundary to be printed, where $ndim$ is the dimension of space. In that case the normal component at the boundary is computed and printed. The vector from which the normal component is computed consists of the degrees of freedom 1, 2 and 3 (or 1 and 2 in R^2) in each point, except if degfd is given, in which case the degrees of freedom degfd, degfd+1 and degfd+2 are used.

tangential_component has the same meaning as normal_component, however, now with respect to the tangential component.

suppress_coordinates suppresses the printing of the co-ordinates in the output.

suppress_header suppresses the printing of the header.

suppress_zeros prints only in the points where the solution is non-zero.

suppress_nodes suppresses the printing of the node numbers.

sequence defines the ordering of the nodal points.

If no sequence is given the co-ordinates are ordered in increasing x-sequence and for constant x-value in increasing y-sequence. If sequence = (y) is given, then first increasing y-sequence and then increasing x-sequence is used (2D) or the sequence y, z, x in 3D. Sequence = (z) creates the sequence z, y, x (3D only).

Sequence = (n) uses the sequence of the nodal point numbers rather than sorting the output.

type_coordinates defines the type of coordinate system to be used, when printing the co-ordinates.

Possible values for c are

Cartesian a Cartesian coordinate system is used.

Polar the coordinates are translated to a polar system (r, ϕ) .

This option can only be used in R^2 .

Default value: Cartesian

type_format defines the type of format to be used when printing.

If $i = 1$ (default value), all reals are printed in 5 decimal digits, if $i = 2$ 8 decimal digits are used.

$i = 3, 4$ has the same meaning as $i = 1, 2$ respectively, however, in this case each node starts on a new line.

equidistant_grid is only available for print commands that are defined on the whole region, not for the boundary.

When this command is used, the solution is interpolated to an equidistant grid defined by $(x_{\min}, x_{\max}) \times (y_{\min}, y_{\max}) \times (z_{\min}, z_{\max})$ and step size (dx, dy, dz) .

The interpolated function is printed.

Hence in this case both the options `REGION = (xmin, xmax, ymin, ymax, zmin, zmax)` and `DISTANCE = (dx, dy, dz)` are obligatory.

file = 'name_file' if this option is found, the output is written to a file named `name_file`.

This name must be given between quotes.

append has only meaning in case the output is written to a file. In that case the contents of the file are not destroyed but the new output is appended to the existing output.

file_format = form defines the format in which the file is written. There are two options for *form*:

standard

matlab

standard the default value, gives the standard type of output.

matlab gives the output in a format that makes it suitable for reading in matlab.

Remarks:

The options `points`, `curves` and `surfaces` are mutually exclusive.

The options `normal_component` and `tangential_component` are also mutually exclusive. They may only be used in combination with the option `curves` or `surfaces`.

PRINT 'text between quotes'

The command `PRINT` prints the text between the quotes to the output file.

Also in this case the options `file = 'file_name'` and `append` may be used.

PRINT TIME May be used to print the time in a time-dependent problem. In this way it is clear at what time level the scalars, vectors and so on are printed.

PRINT MESH options

With this command you can print information of mesh related quantities. This requires knowledge of the data structure as described in the Programmer's Guide.

Possible options (one at a time):

TOPOLOGY prints the topology of the mesh.

NODE_ELEMENTS prints all elements connected to each node.

NODE_NODES prints all node connected to each node.

NEW_NUMBERING prints the new nodal point numbering.

PART_H prints the contents of `kmesh part h` (see Programmer's Guide).

POIN_CUR_SUR prints the node numbers of user points, curves and surfaces.

ELEMENT_ELEMENTS prints all elements connected to each element.

LEVELS prints the nodes in each level.

SUB_CUR_SUR prints all subcurves of curves and all subsurfaces of composite surfaces.

IINPUT_RINPUT prints the arrays `iinput` and `rinput`.

PART_T prints the contents of `kmesh` part `t` (see Programmer's Guide).

PART_U prints the contents of `kmesh` part `u` (see Programmer's Guide).

PART_V prints the contents of `kmesh` part `v` (see Programmer's Guide).

PART_W prints the contents of `kmesh` part `w` (see Programmer's Guide).

PART_X prints the contents of `kmesh` part `x` (see Programmer's Guide).

PART_Z prints the contents of `kmesh` part `y` (see Programmer's Guide).

PART_Y prints the contents of `kmesh` part `z` (see Programmer's Guide).

COOR prints the coordinates of the nodes.

EDGE_NODES prints all nodes of each edge.

ELEMENT_EDGES prints all edges of each element.

NODE_EDGES prints all edges connected to each node.

VOLUME_SURFS prints all surfaces connected to each volume.

EDGE_ELEMENTS prints all elements connected to each edge.

ELEMENT_FACES prints all faces connected to each element.

FACE_EDGES prints all edges of each face.

FACE_ELEMENTS prints all elements connected to each face.

FACE_NODES prints all nodes of each face.

NODE_FACES prints all faces connected to each node.

BLOCKINFO prints the contents of array `blockinfo`.

OUTER_CUR_SUR prints all outer curves or surfaces respectively.

SURF_NODES prints all nodes of each surface.

NORMALS prints the outer normals.

OUTER_BOUN prints the outer boundary.

GROUP_NODES prints all nodes of each element group.

GROUP_INFO prints element group information.

EDGE_FACES prints all faces connected to each edge.

QUANTITIES print special quantities stored for the mesh.

WRITE_TO_FILE `'name_of_file'`, indicates that all print output is printed to the file `name_of_file` (name between quotes) from the moment this statement is reached.

CLOSE_FILE Once this statement is reached the active file is closed and the standard output is active again.

PRINT time_history V_i Prints the time history of the vector with name V_i . If `file = name_of_file` is given the output is written to the file `name_of_file` (name between quotes). The positions in which the time history are printed are must be given in the statement `TIME_HISTORY` (3.2.3.23).

PRINT zero_levelset i Prints the nodal points followed by the coordinates of the nodes at the interface defined by level set function `zero`. If `FILE` is given, the output is printed to the file with name `name_of_file` extended with a sequence number.

PRINT INTERSECTION V_i computes the intersection of a 2d mesh with a straight line and interpolates the vector V_i to the intersection points. The following options are available:

```
degfd i
origin = (0_x, 0_y)
end_point = (E_x,E_y)
angle = a
length = l
file = 'file_name'
append
```

The options `file = 'file_name'` and `append` have the standard meaning. The other options mean:

degfd i defines the degree of freedom of vector V_i that must be used.

Default value: 0, i.e. all degrees of freedom are printed.

origin = (O_x, O_y) defines the origin of the intersection line.

Default values: if `0_x` and `0_y` are known as constants or variables their values are used, otherwise $(0,0)$.

angle = a defines the angle of the line with respect to the positive x-axis.

Default value: if `angle` is known as constant or variable its value is used, otherwise 0.

end_point = (E_x, E_y) defines the end point of the intersection line. If omitted the end point is the final intersection of the line with the 2d mesh.

Default values: if `E_x` and `E_y` are known as constants or variables their values are used, otherwise no end points are used.

length = l is an alternative for `end_point`. If available it defines the length of the intersection line.

This is an alternative for prescribing the end point.

APPEND has only meaning if printing to a file is used, either by using `file = ...` or `write_to_file`. if present the contents of the file are kept and the new output is written at the end of the existing file.

3.2.3.13 plot commands

The following commands are available:

```

PLOT_VECTOR [options]
PLOT_TENSOR [options]
PLOT_CONTOUR [options]
PLOT_COLOURED_LEVELS [options]
PLOT_FUNCTION [options]
PLOT_BOUNDARY [options]
PLOT_MESH [options]
PLOT_INTERSECTION [options]
PLOT_TIME_HISTORY [options]
PLOT_3D [options]
OPEN_PLOT
CLOSE_PLOT
PLOT zero_levelset i
EXTEND_AXI_SYMMETRIC_MESH [options]

```

Meaning of the various commands:

PLOT_VECTOR , options (all in one line) Indicates that a vector plot of the solution must be made. This is only possible in 2D and if the solution vector contains at least 2 degrees of freedom per point. The following options are available:

```

Vi
REGION = ( xmin, xmax, ymin, ymax )
YFACT = y
FACTOR = f
DEGFD1 = i1, DEGFD2 = i2
ZERO_LEVEL_SET

```

Meaning of these options:

Vi defines the name of the solution vector to be plotted.

Default value: the first vector in the list of vector names.

region = (*xmin, xmax, ymin, ymax*) If this option is given, it restricts the area to be plotted to the region defined by (*xmin, xmax*) × (*ymin, ymax*).

Default value: the whole region is plotted.

yfact = *y* defines the multiplication factor to be used in y-direction of the plots. This parameter should only be used in case the length-width ratio is far from 1.

Default value: yfact = 1.

factor = *f* defines the multiplication factor to be used to multiply the length of the arrows.

If omitted, the program computes this length itself, however, in some applications this is not the the one you would like to have. If factor=0, the factor is computed by the program.

Default value: factor = 0.

degfd1 = *i1*, **degfd2** = *i2* define the sequence numbers of the degrees of freedom to be used for the vector plot.

Default value: degfd1 = 1, degfd2 = degfd1+1.

ZERO_LEVEL_SET indicates that the zero level set curve is plotted. This makes only sense if the level set method is applied.

PLOT_TENSOR , options (all in one line) Indicates that a tensor plot of the solution must be made. This is only possible in 2D and if the tensor contains at least 4 degrees of freedom per point.

The tensor is plotted by making a vector plot of the first two components and a vector plot of the third and fourth component in the same picture. This option makes sense in combination with **COMPUTE_PRINCIPAL_STRESSES**, since then the principal stresses are plotted.

The following options are available:

```

Vi
REGION = ( xmin, xmax, ymin, ymax )
YFACT = y
FACTOR = f
DEGFD1 = i1, DEGFD2 = i2, DEGFD3 = i3, DEGFD4 = i4

```

Meaning of these options:

DEGFD3 = i3, DEGFD4 = i4 define the components of the second vector to be plotted.

All other parameters have exactly the same meaning as in **PLOT_VECTOR**.

PLOT_CONTOUR , options (all in one line) Indicates that a contour plot of one of the components of the solution must be made. The following options are available:

```

Vi
REGION = ( xmin, xmax, ymin, ymax, zmin, zmax )
YFACT = y
DEGFD=i
MINLEVEL = m1
MAXLEVEL = m2
NLEVELS = n
TEXT = '...'
ZERO_LEVEL_SET
levels = (l1, l2, ... )
color = j

```

Meaning of these options:

Vi defines the name of the solution vector to be plotted.

Default value: the first vector in the list of vector names.

region = (xmin, xmax, ymin, ymax, zmin, zmax) If this option is given, it restricts the area to be plotted to the region defined by $(xmin, xmax) \times (ymin, ymax) \times (zmin, zmax)$.

Default value: the whole region is plotted.

yfact = y defines the multiplication factor to be used in y-direction of the plots. This parameter should only be used in case the length-width ratio is far from 1.

Default value: $yfact = 1$.

degfd = i defines the sequence number of the degree of freedom from which a contour plot must be made.

Default value: $degfd = 1$.

MINLEVEL = m1 defines the minimum level of the contour lines.

Default value: computed automatically.

MAXLEVEL = m2 defines the maximum level of the contour lines.

Default value: computed automatically.

NLEVELS = n defines the number of contour lines.

Default value: 10.

ZERO_LEVEL_SET indicates that the zero level set curve is plotted. This makes only sense if the level set method is applied.

LEVELS = (l_1, l_2, \dots, l_n) defines the contour levels.

color = j defines the sequence number of the color to be used for the contour lines.

TEXT = 'text' defines the text to be plotted below the picture. The text must be put between quotes.

Default value: name of vector.

PLOT_COLOURED_LEVELS , options (all in one line) Indicates that a colored levels plot of one of the components of the solution must be made.

The options are exactly the same as for **PLOT_CONTOUR**

PLOT_FUNCTION , options

A 1d function along a curve or set of curves is plotted. The following options are available:

```

Vi
curves ( cj1, cj2, ... )
degfd i
textx
texty

```

Meaning of these options:

Vi defines the name of the solution vector to be plotted.

Default value: the first vector in the list of vector names.

curves (cj1, cj2, ...) defines the set of curves along which the solution vector must be plotted. The curves are used as x-axis, the solution as y-axis.

Default value: c1.

degfd i defines the point-wise, degree of freedom, of the solution vector.

Default value: 1.

textx Defines the text to be plotted below the x-axis.

Default value: actual curve numbers.

texty Defines the text to be plotted along the y-axis.

Default value: name of the vector.

PLOT_BOUNDARY , options

Indicates that the boundary of the region must be plotted. The following options are available:

```

REGION = ( xmin, xmax, ymin, ymax )
YFACT = y

```

Meaning of these options:

region = $(xmin, xmax, ymin, ymax)$ If this option is given, it restricts the area to be plotted to the region defined by $(xmin, xmax) \times (ymin, ymax)$.

Default value: the whole region is plotted.

yfact = y defines the multiplication factor to be used in y-direction of the plots. This parameter should only be used in case the length-width ratio is far from 1.

Default value: yfact = 1.

PLOT_MESH , options

Indicates that the mesh must be plotted. The following options are available:

```

REGION = ( xmin, xmax, ymin, ymax )
YFACT = y
EYEPOINT = (e_1, e_2, e_3)
ORIENTATION = i

```

```

MARK_NODES
NODE_NUMBERS
ELEMENT_NUMBERS
INNER_CURVES
OUTER_CURVES
NO_CURVES

```

Meaning of these options:

region = $(xmin, xmax, ymin, ymax)$ If this option is given, it restricts the area to be plotted to the region defined by $(xmin, xmax) \times (ymin, ymax)$.

Default value: the whole region is plotted.

yfact = y defines the multiplication factor to be used in y-direction of the plots. This parameter should only be used in case the length-width ratio is far from 1.

Default value: $yfact = 1$.

EYEPOINT = (e_1, e_2, e_3) Has the same meaning as in the plot command for mesh generation. See Section (2.2).

ORIENTATION = i Has the same meaning as in the plot command for mesh generation. See Section (2.2).

MARK_NODES If used all nodal points are marked with a cross.

NODE_NUMBERS If used all nodal points are provided with the node number.

ELEMENT_NUMBERS If used all nodal elements are provided with the element number.

INNER_CURVES If used only the inner curves are plotted.

OUTER_CURVES If used only the outer curves are plotted.

NO_CURVES If used only the no curves are plotted.

Default: all curves are plotted.

PLOT_INTERSECTION vector_name options makes an intersection of the two-dimension region with a straight line and interpolates the vector with name `vector_name` onto this intersection. A one-dimensional plot of the interpolated function is made. The following options are available:

```

degfd=i
textx = '..'
texty = '..'
origin = (0_x,0_y)
angle = a

```

degfd=i defines the degree of freedom that is used for plotting.

Default value: 1.

textx = '..', texty = '..' defines the text that is plotted along the x-axis and the y-axis respectively.

Default values: 'x' and 'y'.

origin = (O_x, O_y) , angle = a defines the intersection line. The starting point is defined by the coordinates O_x, O_y and the angle with respect to the x-axis is defined by a .

Default values: $(O_x, O_y) = (0,0)$ and $a=0$.

PLOT_TIME_HISTORY vector_name, options makes a one-dimensional plot of the time history of the vector with name `vector_name`. The positions in which the time history are plotted are must be given in the statement `TIME_HISTORY (3.2.3.23)`. The following options are available:

```

colors
legenda

```

Meaning of these options

colors = i1, i2, ... defines the set of colors to be used for plotting of the various curves.

legenda = (x1,y1), (x2,y2), ... defines a set of coordinates to be used in the legenda to couple a color to a position.

Default: no legenda.

PLOT_3D vector_name options makes a 3d plot of two-dimensional function defined by the vector `vector_name`. The following options are available:

```
degfd=i
text = 't'
```

Meaning of these options

degfd=i defines the degree of freedom of the function that is used for plotting.

Default value: 1.

text = 't' defines the text that is used as bottom text.

Default value: name of the vector.

PLOT_zero_levelset i plots the curve defined by level set function zero (R^2 only).

The options with respect to curves as in `plot_mesh` are also applicable.

OPEN_PLOT All plots between the commands `open_plot` and `close_plot` are drawn in one picture.

CLOSE_PLOT See `open_plot`.

EXTEND_AXI_SYMMETRIC_MESH, options may be used in case of an axi-symmetric (2d) problem. This command creates an extra mesh with nodes and elements that are the same as the original mesh plus ones that are mirrored around the symmetry axis.

This command is meant to make it possible to plot a symmetric picture on both sides of the symmetry axis. The following options are available:

```
input_mesh = i
output_mesh = j
```

Meaning of these options

input_mesh = i defines the mesh number of the mesh to be extended.

Default value: 1

output_mesh = i defines the mesh number of the extended mesh.

Default value: 2

The mesh numbers for input and output mesh must be different, but in standard applications there is no need to give either of them.

Once the command `extend_axi_symmetric_mesh` has been used, each plot may be extended to the complete domain by adding the subkeyword `axi_symmetric` to these plot commands. So you may get both pictures in the r-z domain for $r \geq 0$ as well as pictures on the the whole domain.

3.2.3.14 commands to read vectors from or write to a file

The following commands are available:

```
WRITE_VECTOR Vj [options]
READ_VECTOR Vj [options]
INPUT_VECTOR Vj, 'file_name', type_of_vector
READ_SOLUTIONS
```

Meaning of the various commands:

WRITE_VECTOR Vj, options

may be used to write vector V_j to backing storage (file 2). This command is meant for reusing the solution in another program (in general not for post-processing). If this command is used the block defined by the main keyword `START` must have been used. In this block the option

```
DATABASE = d
```

with `d` `new` or `old` must have been given.

The following options are available:

```
sequence_number = i
save_administration
```

sequence_number = i, defines the sequence number to which the array must be written. If omitted the vector is written to the next free sequence number and this number is printed.

save_administration forces the administration of the backing storage file to be renewed at the file. If program `sepcomp` finishes without an error message the administration is written automatically to the file. However, if some error occurs it may be possible that the administration is not saved. As a consequence the arrays written may be not recognized in a subsequent program. For that reason the option `save_administration` forms an extra security.

READ_VECTOR Vj, options

may be used to read the contents of an array from backing storage (file 2) into vector V_j . If this command is used the block defined by the main keyword `START` must have been used. In this block the option

```
DATABASE = old
```

must have been given and the array to be read must have been stored. Furthermore this array must correspond to the present mesh.

The following options are available:

```
sequence_number = i
```

sequence_number = i defines the sequence number from which the array must be read. If omitted the array with the highest sequence number in the backing storage file is read.

INPUT_VECTOR Vj, file = 'file_name' [type_of_vector]

Indicates that a vector with name V_j must be read from the file with name *file_name*. Mark that the quotes around the file name are essential in order to indicate that a literal text must be read.

type_of_vector indicates the type of the vector that is read.

The following values for *type_of_vector* are available:

scalar The vector contains one unknown per point

vector The vector contains exactly $ndim$ (= dimension of the space) unknowns per point.

tensor The vector is of the type symmetric tensor. In R^1 this means 1 unknown per point, in R^2 3 unknowns per point, and in R^3 6 unknowns per point.

number= n The vector contains exactly n degrees of freedom in each point.

element.wise The vector contains exactly one degree of freedom per element.

The default value is *scalar*.

The file from which the data must be read must be an ASCII file containing the vector to be read in the sequence:

degrees of freedom of node 1, degrees of freedom of node 2 and so on.

Hence first all degrees of freedom of the first node (in natural sequence) then of second node etc. The nodes are read in the sequence of the mesh, hence the input must be given in the sequence of the mesh nodes.

The numbers are read according to standard FORTRAN free format, i.e. the number of numbers per line is arbitrary. No texts are allowed in this file, not even comments.

READ_SOLUTIONS This keyword may only be used if the keyword `INPUT_SEPCOMP_OUT` is present in the input block `START`. See Section (3.2.1).

It reads a set of solution vectors at a certain time, from the input `sepcomp.out` file.

If this keyword is found, the next set of solution vectors corresponding to one time level is read. If the time parameter t corresponding to this set is smaller than the actual time, this set is skipped and a new set is read. This process is repeated until the actual time is not larger than the time in the input file. After reading the actual time is set to the time found in the input file `sepcomp.out`.

3.2.3.15 commands for mesh manipulation

The following commands are available:

```

DEFORM_MESH [options]
CHANGE_COORDINATES [, sequence_number = k]
REFINE_MESH [options]
PRESENT_MESH = i
INTERCHANGE_MESH = (i,j)
WRITE_MESH [options]
READ_MESH i [options]
COPY_MESH i to j
COMPARE_MESH [options]
INTERSECTION LINE [options]
INTERSECT_MESH [options]
MOVE_BOUNDARY [options]

```

Meaning of the various commands:

DEFORM_MESH , options

If this command is given the coordinates of the mesh are changed using the displacement stored in the solution vector, possibly multiplied by a scaling factor.

The following options are available:

Vk , `scale = s`

These options must be given on the same line.

The options have the following meaning:

Vk defines the name of the vector in which the displacement vector is stored.

Default value: the first vector in the list of vector names.

`scale = s` defines the multiplication vector to be used.

Hence the new coordinates are given by:

$$\mathbf{x} = \mathbf{x} + s \mathbf{u} \quad (3.2.3.3)$$

with \mathbf{x} the coordinate vector, \mathbf{u} the displacement vector and s the scaling factor.

Default value: 1

CHANGE_COORDINATES sequence_number = k .

This command may be used to change the coordinates of the mesh. The default sequence number is 1.

It requires an input block of the form:

```

change_coordinates, sequence_number = k
...
end

```

`sequencenumber = k` is optional. If omitted the next sequence number is used.

At the place of the dots the actual input is expected. This input is completely identical as the input described in Section 2.2

If no input block is found all coordinates are used and the parameter `FUNC_CHAN` is set equal to 1.

Besides the extra input block also a subroutine `FUNCCOOR` as described in Section 2.2.1 must be provided to the main program.

REFINE_MESH , options If this option is used the mesh must be refined globally.

At most 5 different meshes may be present in SEPCOMP.

The following actions are performed when REFINE_MESH is found:

- The mesh is refined and stored as a mesh with sequence number MESH_OUT
- All solution vectors and vectors of special structure defined per point are interpolated from the mesh with sequence number MESH_IN to the mesh with sequence number MESH_OUT.
- Not only the mesh is refined, also the corresponding problem description and matrix structures are recomputed.
- If MESH_IN and MESH_OUT have different numbers, all information of both meshes is stored. If these numbers are identical the information of the coarse mesh is destroyed. This means that each vector does not only have a sequence number but also a mesh sequence number. Two vectors V1 with different mesh sequence numbers are stored as separate vectors and can both be addressed.
- The new mesh sequence number is set equal to MESH_OUT. This means that all next computations are carried out at the new mesh, until the mesh sequence number is changed.

The following options are available:

```
TIMES = n
MESH_IN = m1
MESH_OUT = m2
SEQUENCE_NUMBER = i
```

These options have the following meaning:

TIMES = n indicates that the mesh must be refined n times. Each refinement implies doubling of the number of points in each direction.

Default value 1.

MESH_IN = $m1$ $m1$ defines the sequence number of the mesh to be refined. This number must be between 1 and 5.

Default value 1.

MESH_OUT = $m2$ $m2$ defines the sequence number of the refined mesh. This number must be between 1 and 5.

$m1$ and $m2$ may be equal, in which case the information about the coarse mesh is lost.

Default value 1.

SEQUENCE_NUMBER = i refers to the sequence number of the input block REFINE [3.2.21](#).

If this block is used it is also possible to define the other parameters in the REFINE block, rather than in the STRUCTURE block.

Default value 1.

PRESENT_MESH = i sets the mesh sequence number equal to i . i must be in the range 1 to 5.

All next next computations are carried out at the new mesh, until the mesh sequence number is changed. Of course the mesh corresponding to i must exist.

INTERCHANGE_MESH = (i, j) means that the sequence numbers of the meshes with sequence number i and j are switched. If for example mesh 1 is the coarse mesh and mesh 2 the fine one, then `interchange_mesh = (1,2)` gives the coarse mesh sequence number 1 and the fine mesh sequence number 2. This operation does not take computation time, nor does it use any storage. It is meant to make programming more easy.

An example of the use of this option can be found in Example [6.2.4](#).

Default value (1,2).

WRITE_MESH , options

indicates that the present mesh must be written to a file in the standard `meshoutput` format. The following options are available:

```
FILE = 'file_name'
```

`FILE = 'file_name'` defines the name of the file to which the information must be written. If the file already exists the file is overwritten. After writing the mesh, the file is closed.

Default value: `meshoutput`

READ_MESH *i*, options

indicates that a new mesh must be read from a file in the standard `meshoutput` format. The mesh is stored as mesh number *i*.

The default value for *i* is 1.

The following options are available:

```
FILE = 'file_name'  
problem = p
```

Meaning of these options:

FILE = 'file_name' defines the name of the file from which the information must be read. After reading the mesh, the file is closed.

Default value: `meshoutput`

j defines the sequence number of the mesh in which the mesh is read.

problem = p defines how the problem definition for the new mesh must be defined.

Possibilities:

```
default  
old  
none
```

Meaning of these options:

none indicates that no problem definition is coupled to the mesh read.

default indicates that a default problem definition is used.

old Copies the problem definition from the old mesh to the new mesh read.

The default value is `problem = old`.

COPY_MESH *i to j* copies mesh with sequence number *i* into sequence number *j*.

COMPARE_MESH [options] compares two meshes and checks if they are identical.

The following options are available:

```
mesh1 = i  
mesh2 = j
```

These quantities define the mesh sequence numbers of the meshes to be compared.

Default values: `mesh1 = 1`, `mesh2 = 2`.

INTERSECTION LINE [options] computes the intersection of a line with a curve and stores the result in a scalar.

The following options are available:

```
curve Ci  
origin = (o_x,o_y)  
angle = alpha  
scalar_name
```


The curve is defined by C_i and the line by the origin as well as the angle in radians. The distance from intersection point to origin along the line is stored in the scalar.

Default values: origin = (0,0); angle = 0.

INTERSECTION MESH [options] computes the intersection of a 3d mesh with a given plane and stores the result in a 2-dimensional mesh.

The following options are available:

```
MESH_OUT
PLANE (ax+by+cz=d)
```

MESH_OUT defines the sequence number of the output (2d) mesh and PLANE the plane that intersects the mesh by the parameters a , b , c and d .

Default values: mesh.out = meshnr+1, where meshnr is the present mesh sequence number.

MOVE_BOUNDARY [options] defines how the boundary of a mesh may be moved and defines the displacement vector for the movement of this boundary.

The following options are available:

```
DISPLACEMENT_VECTOR = ...
VELOCITY_VECTOR = ...
SEQ_BOUN_INPUT = ...
PROB_DISPLACEMENT = ...
STREAM_FUNCTION = ...
```

meaning of these keywords

DISPLACEMENT_VECTOR = ... Defines the name of the displacement vector.

Default value: displacement, if not existing first vector in list of vectors.

VELOCITY_VECTOR = ... Defines the name of the velocity vector.

Default value: velocity, if not existing first vector in list of vectors.

SEQ_BOUN_INPUT defines the input for moving the boundary. The sequence number refers to the input of the input block ADAPT_BOUNDARY (see Section 3.4.4).

Default value: 1.

PROB_DISPLACEMENT = ... defines the problem number for the displacement vector.

Default value: 1.

STREAM_FUNCTION = ... Defines the name of the stream function.

Default value: psi, if not existing first vector in list of vectors.

3.2.3.16 commands for interpolation

The following commands are available:

```
INTERPOLATE [options]
```

Meaning of the various commands:

INTERPOLATE , options

Interpolates a solution vector or a vector of special structure defined per point from one mesh to another one. The following options are available:

```
 $V_i$   
MESH_IN = m1  
MESH_OUT = m2
```

These options have the following meaning:

V_i Defines the vector to be interpolated.

Default value: the first vector in the list of vector names.

MESH_IN = $m1$ $m1$ defines the sequence number of the mesh from which V_i must be interpolated. This mesh must exist.

Default value 1.

MESH_OUT = $m2$ $m2$ defines the sequence number of the mesh to which V_i must be interpolated. This mesh must exist.

Default value 2.

3.2.3.17 commands to manipulate obstacles

Obstacles are defined in the mesh generation part. They are defined as a closed set of curves (2D) or a closed set of surfaces (3D). For the computation they are only of interest when they intersect the actual mesh.

The following commands are available to manipulate the obstacles:

MOVE_OBSTACLE *i* , options

With this command the obstacle with sequence number *i* is moved over a distance $\mathbf{u}\Delta t$, where \mathbf{u} is the velocity of the obstacle and Δt the present time step as defined in a time integration block.

The following options are available:

```
velocity = ( u_1, u_2, u_3)
rotation_plane = xz
rotation_velocity = u_phi
displacement = ( u_1, u_2, u_3)
origin = (o_x, o_y, o_z)
```

These options must be given on the same line.

The options have the following meaning:

velocity = (u_1, u_2, u_3) defines the velocity vector \mathbf{u} .

The number of components must be equal to the dimension of the space.

The displacement of the obstacle is defined by $\Delta t \times \mathbf{u}$.

rotation_plane = *xy* If the obstacle also rotates we need to know in which plane the rotation takes place. At this moment only rotations in x-y plane, are implemented.

rotation_velocity = u_ϕ defines the rotation velocity. Is only used in combination with rotation_plane.

This option is only available for obstacles in R^3 .

Default value: 0

displacement = (u_1, u_2, u_3) has the same meaning as velocity = (u_1, u_2, u_3), except that the displacement is given rather than the velocity. Both options are mutually exclusive.

origin = (o_x, o_y, o_z) Defines the origin of the axis of rotation. Together with rotation_plane the rotation axis is defined in this way.

Default value: (0,0,0)

MAKE_OBSTACLE_MESH , options

With this command a new mesh is created, such that the old mesh is kept but that elements that are intersected by the obstacle are subdivided into smaller elements, such that these elements are either within the obstacle or outside the obstacle. So the class of elements partly inside the obstacle is empty after this command.

If an edge of the element is intersected close to one of its nodes (less than 10% related to the edge length), it is assumed that the intersection is exactly in the corresponding node. Furthermore, if an edge is intersected more than once, only one intersection is used.

This option allows you to use a fixed mesh, but to create a temporary mesh adapted to the obstacle. It should be used in combination with the keyword REMOVE_OBSTACLE_MESH.

If already some solution vectors have been created all these vectors are interpolated to the new mesh automatically. If you want to give boundary conditions adapted to the obstacle these boundary conditions should be applied to the new mesh.

At this moment this option is only available for linear triangles.

The following options are available:

```
MESH_ORIG = i
MESH_OBST = j
SEQ_STRUCTURE = s
```

These options must be given on the same line.
The options have the following meaning:

MESH_ORIG = i defines the sequence number of the original mesh (usually 1).
Default value: 1

MESH_OBST = j defines the sequence number of the adapted mesh. This number must be different from the original mesh.
Default value: 2

SEQ_STRUCTURE = s defines the structure of the matrices corresponding to the new mesh. The value s has the same meaning as for the keyword **MATRIX** 3.2.4.
 $s = -1$ means that the same matrix structure as for the original mesh is used.
Default value: -1

After the creation of the new mesh the present mesh sequence number is made equal to j .

REMOVE_OBSTACLE_MESH , options

This command may only be used if already **MAKE_OBSTACLE_MESH** has been called. This command does the opposite: the obstacle mesh is removed and all solution vectors are mapped onto the original mesh.

At this moment this option is only available for linear triangles.

The following options are available:

```
MESH_ORIG = i
MESH_OBST = j
```

These options must be given on the same line.
The options have the following meaning:

MESH_ORIG = i defines the sequence number of the original mesh (usually 1). This mesh must exist.
Default value: 1

MESH_OBST = j defines the sequence number of the adapted mesh. This mesh must exist and correspond to the original mesh.
Default value: 2

After the creation of the new mesh the present mesh sequence number is made equal to i .

3.2.3.18 commands to use the level set method

A level set function ϕ is a marker function to distinguish between certain regions of interest especially in case of free or moving boundary problems. In general $\phi = 0$ defines the interface between two parts of the region and in that way the free boundary. For example $\phi > 0$ may be define the fluid part of some material, whereas $\phi < 0$ defines the solid part.

The following commands are available:

MAKE_LEVELSET_MESH , options

REMOVE_LEVELSET_MESH , options

LEVELSET_MESH_VELOCITY Vector_name, options

ADAPT_INTERFACE_BOUNDARY Vector_name, options

MAKE_DISTANCE_FUNCTION Vector_name, options

CREATE_LEVELSET_VECTOR Vector_name, options

Explanation

MAKE_LEVELSET_MESH , options

With this command a new mesh is created, such that the old mesh is kept but that elements that are intersected by $\phi = 0$ are subdivided into smaller elements, such that these elements are either within the region $\phi > 0$ or in the region $\phi < 0$. So the class of elements that have both $\phi > 0$ and $\phi < 0$ is empty after this command.

If an edge of the element is intersected close to one of its nodes (less than 10% related to the edge length), it is assumed that the intersection is exactly in the corresponding node. Furthermore, if an edge is intersected more than once, only one intersection is used.

This option allows you to use a fixed mesh, but to create a temporary mesh adapted to the $\phi = 0$ level. It should be used in combination with the keyword **REMOVE_LEVELSET_MESH**.

If already some solution vectors have been created all these vectors are interpolated to the new mesh automatically. If you want to give boundary conditions adapted to the $\phi = 0$ level these boundary conditions should be applied to the new mesh.

At this moment this option is only available for linear triangles.

The following options are available:

```
MESH_ORIG = i
MESH_SUBDIVIDE = j
SEQ_STRUCTURE = s
LEVELSET_VECTOR = 1
INTERPOLATE ( Vecx, Vecy, ... )
NO_INTERPOLATION
EPS_MOVE = e
EPS_MIN = e
```

These options must be given on the same line.

The options have the following meaning:

MESH_ORIG = i defines the sequence number of the original mesh (usually 1).

Default value: 1

MESH_SUBDIVIDE = j defines the sequence number of the adapted mesh. This number must be different from the original mesh.

Default value: 2

SEQ_STRUCTURE = s defines the structure of the matrices corresponding to the new mesh. The value s has the same meaning as for the keyword **MATRIX** 3.2.4.

$s = -1$ means that the same matrix structure as for the original mesh is used.

Default value: -1

LEVELSET_VECTOR = l defines the sequence number of the level set function ϕ with respect to the set of solution vectors.

Default value: 2

INTERPOLATE (Vecx, Vecy, ...) Interpolate the vectors defined between brackets from old to new mesh.

Default value: fill all vectors

NO_INTERPOLATION Do not interpolate vectors from old to new mesh.

Default value: fill all vectors

EPS_MOVE = e defines under which circumstances a node is moved or an edge is intersected in the new mesh.

Let the length of an edge that is intersected by the level set 0 contour, be l . If the distance of one of the nodes to the intersection is less than $e \times l$ ($0 \leq e < 0.5$) then this node is moved to the intersection. This is not the case if 2 levelset 0 lines are too close to this node. Otherwise the intersection point is a new node in the new mesh.

Default value: eps_move if this is constant or scalar (variable).

If eps_move is not available the default value is equal to the general_constant *eps_accuracy*, which itself has a default value of 0.3.

EPS_MIN = e identifies an intersection of an edge with the zero levelset function with a node if the distance to this node is less than $l \times e$, under the condition that the nodes has not been moved before.

Default value: eps_min if this is constant or scalar (variable).

If eps_min is not available the default value is equal to the $0.01 \times \text{eps_accuracy}$.

After the creation of the new mesh the present mesh sequence number is made equal to j .

REMOVE_LEVELSET_MESH , options

This command may only be used if already **MAKE_LEVELSET_MESH** has been called. This command does the opposite: the level set mesh is removed and all solution vectors are mapped onto the original mesh.

At this moment this option is only available for linear triangles.

The following options are available:

```
MESH_ORIG = i
MESH_SUBDIVIDE = j
KEEP_PART
```

These options must be given on the same line.

The options have the following meaning:

MESH_ORIG = i defines the sequence number of the original mesh (usually 1). This mesh must exist.

Default value: 1

MESH_SUBDIVIDE = j defines the sequence number of the adapted mesh. This mesh must exist and correspond to the original mesh.

Default value: 2

KEEP_PART indicates that a part of the mesh must be kept in order to compute the mesh velocity later on. After having computed the mesh velocity this part is also removed.

After the creation of the new mesh the present mesh sequence number is made equal to i .

LEVELSET_MESH_VELOCITY , Vector_name options

With this command the mesh velocity in case of a levelset method is computed. Also the concentration at the interface is adapted as well as the concentration in points that are in the new levelset region but not in the old one.

The following options are available:

```
VECTOR_NAME
BASIS_MESH = i
MESH_IN = i
MESH_OUT = i
SOLUTION = Vj
LEVELSET = Vk
OLD_LEVELSET = Vm
TIME_STEP = dt
```

These options must be given on the same line.

The options have the following meaning:

VECTOR_NAME defines the name of the vector containing the mesh velocity.

Default value: mesh_vel, hence if this vector exists vector_name may be skipped

BASIS_MESH = i , defines the sequence number of the basis mesh.

Default value: 1

MESH_IN = i , defines the sequence number of the previous levelset mesh.

Default value: 2

MESH_OUT = i , defines the sequence number of the new levelset mesh.

Note that these values do not have to be changed if the command `interchange_mesh (2, 3)` is used in the time loop. See subsection [3.2.3.15](#).

Default value: 3

SOLUTION = V_j , defines the name of the solution vector. The values at the new interface are computed by taking the values at the nearest points of the old interface. Points that correspond to the new levelset region but not to the old one are treated in the same way. The mesh velocity is computed such that a correct new solution will be computed by solving the convection-diffusion equation.

Default value: concentration

LEVELSET = V_k , defines the name of the vector that defines the new levelset region. It must correspond to the mesh with sequence number `mesh_out`.

Default value: phi

OLD_LEVELSET = V_m , defines the name of the vector that defines the old levelset region.

It must correspond to the mesh with sequence number `mesh_in`.

Default value: phiold

TIME_STEP = dt , defines the time step, which is needed to compute the mesh velocity.

Default value: dt

ADAPT_INTERFACE_BOUNDARY , Vector_name options

is used in 2D in case the levelset interface is open and intersects the outer boundary. It is used to set the values in the end points equal to the values in the adjacent points in order to prevent unnecessary wiggles.

The following options are available:

```
VECTOR_NAME
LEVELSET = i
```

These options must be given on the same line.

The options have the following meaning:

VECTOR_NAME defines the name of the vector containing the solution.

Default value: `concentration`, hence if this vector exists `vector_name` may be skipped

LEVELSET = i defines the sequence number of the levelset.

Default value: 1

MAKE_DISTANCE_FUNCTION , `Vector_name` options

is used to make the levelset function ϕ a signed distance function. This may be done by solving the following non-linear convection equation:

$$\frac{\partial \phi}{\partial t} + \text{sign}(\phi) \frac{\text{grad } \phi}{\|\text{grad } \phi\|} \cdot \text{grad } \phi = \text{sign}(\phi), \quad (3.2.3.4)$$

which converges to $\|\text{grad } \phi\| = 1$ if ϕ goes to infinity. This problem is solved by a pseudo time integration and can be considered as a kind of Picard iteration to solve the non-linear equation.

An alternative option is to use an exact computation of the distance by computing the distance to the nearest points, edges and faces of the interface $\phi = 0$.

The following options are available:

```
time_step = dt
upwind / no_upwind
accuracy = eps
maxiter = m
print_level = p
integration_method = m
seq_solve = s
phi_barrier = p
at_error = e
type_norm = e
diffusion = e
number_smoothing_steps = i
distance_method = d
step_size = h
```

These options must be given on the same line.

The options have the following meaning:

time_step = dt defines the (pseudo) time step to be used for the integration of the convection equation. To get a good convergence this term should be not too small or too large.

If an implicit method is used a good choice might be between $4h$ and $32h$, where h is some representative measure for the length of the elements.

Default value: `dtpseudo`, where `dtpseudo` is scalar with name `dtpseudo`. This parameter should have a value.

upwind / no_upwind indicates if upwind must be used to integrate the equation.

Default value: `upwind`

accuracy = eps defines the accuracy used to terminate the iteration.

Default value: `eps` or 0.01. `eps` is used if a scalar with the name `eps` exists.

maxiter = m defines the maximum number of iterations.

Default value: 100

print_level = p defines the amount of output created during iteration.

$p = 0$ gives no information, $p = 2$ maximal information.

Default value: 0

integration_method = m defines the type of time-integration used. At this moment only implicit euler is available.

Default value: `euler_implicit`

seq_solve = s defines the sequence number of the linear solver input block.

Default value: 1

phi_barrier = p defines a band around the interface in which accuracy is checked. Only in a small band of 1 or 2 elements the gradient of ϕ is required, hence only there we need an accurate ϕ . Usually $p = 2.5h$ is a good measure.

Default value: epsdist or 1. epsdist is used if a scalar with the name **epsdist** exists.

at_error = e defines what the iteration method should do if no convergence occurs.

e may have one of the value: **stop** or **resume**.

Default value: stop

type_norm = e Defines the type of norm that is used to compute the accuracy.

The following options for e are available.

distance
max_distance
difference
max_difference

Meaning of these options

distance The iteration stops when the difference between $|\nabla\phi|$ and 1 is less than ϵ . The L^2 norm is used.

max_distance See distance, however the maximum norm is used.

difference The iteration stops when the difference between 2 succeeding iterations is less than ϵ . The L^2 norm is used.

max_difference See difference, however the maximum norm is used.

Default value: distance

diffusion = e It is possible to add some diffusion to the differential equation in order to get a better convergence.

The diffusion is only applied for a number of smoothing steps, after that the diffusion is made equal to 0.

Default value: $e = 0$

number_smoothing_steps = i Defines the number of smoothing steps, where the diffusion is unequal to 0.

Default value: 0

distance_method = d Defines the type of method used to correct the distance function.

Possible values for d :

pseudo_time The differential equation is solved with a pseudo time integration method.

distance The exact distance is computed.

At this moment this method is only applied for the nodes that are direct neighbors of the interface and its direct neighbors. This is sufficient to compute the gradient of ϕ (and hence the normal), as well as the curvature.

Default value: distance

step_size = h To compute the shortest distance in case of **distance_method = distance** the domain is overlapped with a rectangular grid with step size h . This makes the method much more efficient. An optimal choice for h is the mean step size of the basis mesh.

Default value: h if this variable is defined

CREATE_LEVELSET_VECTOR , vector_name options

can be used to create the level set vector ϕ provided an obstacle is defined in the mesh input.

In first instance the mesh adapted to the obstacle in the same way as in `make_levelset_mesh`.

The vector is created on this mesh by setting it to 0 on the intersection of obstacle and mesh.

The values in the other nodes are defined by the distance to the zero levelset. Finally the vector is interpolated to the original mesh.

The following options are available.

```
mesh_orig
mesh_subdivide (default mesh_orig+1)
vector_name (default phi or 2)
obstacle i (default 1)
sign_inobstacle = i (default 1)
problem = i (default 2)
eps_move = e (default accuracy_obstacle)
eps_min = e (default 0.01 * accuracy_obstacle)
```

These options must be given on the same line.

The options have the following meaning:

mesh_orig defines the original mesh in which the function is defined.

Default: present mesh

mesh_subdivide defines the temporary mesh that is created and removed at the end.

Default: mesh_orig+1

vector_name defines the vector in which the levelset function is stored.

Default: phi and if this vector does not exist the second vector in the list of vectors.

obstacle i defines the sequence number of the obstacle to be used.

Default: 1

sign_inobstacle = i defines the sign of the levelset function inside the obstacle.

Default: 1

problem = i defines the problem number corresponding to the vector.

Default: 2

eps_move = e see make_levelset_mesh.

eps_min = e see make_levelset_mesh.

3.2.3.19 A special command to give the user the opportunity to execute his own fortran statements

The following commands are available:

```
USER_OUTPUT [options]
```

Meaning of the various commands:

USER_OUTPUT options,

may be used to perform some user defined output but also to manipulate the solution type arrays or scalars.

If **USER_OUTPUT** is given a user written subroutine **USEROUT** is called in which the user may manipulate the solution vectors in his own way. The use of this option requires some knowledge of the main **SEPRAN** subroutines and is therefore only possible if the **SEPRAN** Programmers Guide is studied.

A description of how the subroutine **USEROUT** must be programmed can be found in the Programmers Guide Section 19.5.1.

The following options are available:

```
sequence_number = i
extra_integers = (i_1, i_2, ... )
extra_scalars = (s_1, s_2, ... )
```

sequence_number = i , defines a sequence number to be used in the call of **USEROUT**.

This sequence number may be used to distinguish between various calls.

If omitted the array with the highest sequence number in the backing storage file is read.

extra_integers = (i₁, i₂, ...) , defines a number of extra integers that may be used in subroutine **USEROUT**.

extra_scalars = (s₁, s₂, ...) , defines a number of extra scalars (reals) that may be used in subroutine **USEROUTS**.

Mark that these scalars must be defined in the block constants under the part **VARIABLES** or **SCALARS**. (Section 1.4) The users must refer to these scalars either by their sequence number (not recommended) or by a reference to their names preceded by a % hence something like %name_of_scalar. This last option is recommended. The consequence is that s_i must always be an integer or a reference to a scalar.

If **extra_scalars** is present instead of **USEROUT** subroutine **USEROUTS** is called. See the Programmers Guide Section 19.5.2.

The values of the scalars may be changed during the computations. In the call of **USEROUTS** always the actual value is given.

3.2.3.20 auxiliary commands

The following commands are available:

```
MATRIX_STRUCTURE [options]
CHANGE_STRUCTURE_OF_MATRIX, seq_structure = s, seq_storage = i
CHANGE_COEFFICIENTS [options]
NEW_PROBLEM_DESCRIPTION, SEQUENCE_NUMBER = s
SET_TIME t
```

Meaning of the various commands:

MATRIX_STRUCTURE , options, defines the structure of the matrix.

The following options behind **MATRIX_STRUCTURE** may be used

```
STORAGE_SCHEME
SYMMETRIC
COMPLEX
REAL
UNSYMMETRIC
INCOMPRESSIBILITY_SYM
INCOMPRESSIBILITY_UNSYM
ASSEMBLE_PRECON
REACTION_FORCE
SHIFTED_LAPLACE
SEQUENCE_NUMBER
PRINT_LEVEL
MUMPS
POSITIVE_DEFINITE
```

The meaning of these keywords can be found in Section (3.2.4), in the input block matrix.

CHANGE_STRUCTURE_OF_MATRIX , seq_structure = s, seq_storage = i

The command **CHANGE_STRUCTURE_OF_MATRIX** may be used to define a new structure of the matrix. For example if the matrix is stored as profile matrix a new matrix may be created stored as a compact matrix. This does not mean that old matrices are transformed into a new form, but only that all newly created matrices get this new structure. Which structure these matrices get is described in the input block "MATRIX" (3.2.4) with sequence number *s*.

Normally the structure of the standard matrix is overwritten, but the user may store the new storage at a new place with sequence number *i* by giving the keyword **seq_storage = i**. The standard storage has sequence number 1, so sequence number 2 is quite obvious. However, if **nprob** problems are used, the standard sequence numbers are 1 to **nprob**, and then the new structure should start at **nprob+1**.

CHANGE_COEFFICIENTS seq_coef = *k*, seq_change = *j* at_iteration = *i*, problem = *p*.

This command may be used to change the coefficients that are active. The coefficients are defined according to the sequence number *k* and then changed according to the sequence number *j* of the input block "CHANGE COEFFICIENTS" (3.2.7).

The rule to be applied is as follows:

each option **seq_coef = k**, wherever found defines a specific series of coefficients. The last one defined is active. A command **change.coefficients**, whether it is in the structure block or for example in the input block **NONLINEAR_EQUATIONS** replaces these actual series by new coefficients. These coefficients remain active until a new command **seq_coef** is found with a sequence number that differs from the actual one. In that case the old definition of **seq_coef** is used and the changes are not longer applied. So if for example the sequence:

```
solve_linear_system, seq_coef = 1
change_coefficients, seq_coef = 1, seq_change = 2
solve_linear_system, seq_coef = 1
solve_linear_system, seq_coef = 2
solve_linear_system, seq_coef = 1
```

is used, then the first linear system is solved with the original series of coefficients corresponding to sequence number 1. The second linear system is solved with the same system, however, with the original series updated according to the input of change coefficients with sequence number 2. The third linear system is solved with the coefficients according to the input block with sequence number 2 and finally the last linear system is solved using the original coefficients with sequence number 1, i.e. without the effect of the change coefficients.

NEW_PROBLEM_DESCRIPTION , SEQUENCE_NUMBER = s .

With this command the user may change to a new problem description as read by the input block PROBLEM with sequence number s . To switch back to the original problem description reuse this command with the corresponding sequence number (usually 1).

This option is meant for example to change the type of the boundary conditions, for example if you switch from essential to natural boundary conditions.

If executed also the structure of the large matrix is recomputed automatically.

SET_TIME t .

Sets the time equal to the constant t . Instead of a constant also a scalar (variable) may be used.

This command may be used for example to reset the time to the initial time after performing a time loop.

A more simple option is:

```
time = ...
```

3.2.3.21 Special commands related to certain types of equations

Besides the more general commands treated in this Section, there are also a number of commands related to special equations.

A part of the information with respect to these commands can be found in this manual, another part in the manual Standard Problems corresponding to the specific equation.

At present it only concerns commands related to the (time-dependent) Navier-Stokes equations and the Reynolds equations for bearings.

The following commands are available:

```
NAVIER_STOKES [ sequence_number = s ]  
SOLVE_BEARING [ sequence_number = s ]
```

Meaning of the various commands:

NAVIER_STOKES [sequence_number = s] This command is used to solve the time-dependent Navier Stokes equation by one of the available methods. This command can only be used as part of a time loop (3.2.3.23). The input for the Navier-Stokes equations must be stored in an input block `Navier-Stokes` as described in Section (3.2.22).

SOLVE_BEARING [sequence_number = s] This command is used to solve the incompressible Reynolds equations for bearings. At this moment it is only meant to perform one complete iteration of the mass-conserving method of Kumar. The input for the bearing must be stored in an input block `BEARING` as described in Section (3.2.24).

If `SOLVE_BEARING` is used it is necessary that the parameter `IBCMAT` in input block is equal to 1, so `method = 20x`, with `x` the storage method used.

Furthermore the problem block must be extended with the essential boundary condition `cavitation = 1`

3.2.3.22 Defaults

If the block STRUCTURE is omitted SEPCOMP checks for the presence of the block NONLINEAR EQUATIONS. If this block is available SEPCOMP reacts as if the block STRUCTURE is available with the following contents:

```
structure
  prescribe_boundary_conditions V1
  solve_nonlinear_system V1
  output V1
end
```

V1 is the name of the first vector in the set of vector names.

Otherwise it is supposed that a linear system must be solved and the structure is:

```
structure
  prescribe_boundary_conditions V1
  solve_linear_system V1
  output V1
end
```

3.2.3.23 Loop commands

Structure recognizes a number of loop commands. At this moment the following loop commands are available:

```
WHILE loop
FOR loop
START_LOOP
START_TIME_LOOP
IF statement
BREAK
```

Besides that you can also stop the program half way the structure block for example for testing purposes using the command:

```
STOP
```

If this command is reached the program stops computation and does not produce output anymore. These loops have the following shape:

```
WHILE ( boolean ) DO
    command_1
    command_2
    command_3
END_WHILE

IF ( boolean ) THEN
    command_1
    command_2
    command_3
END_IF

START_LOOP, sequence_number = k
    ....
END_LOOP

FOR variable = a to b step c
    ....
END_FOR

TIME_HISTORY [options]

START_TIME_LOOP
    ....
    TIME_INTEGRATION, sequence_number = s, vector = i
    ....
END_TIME_LOOP
```

The IF, WHILE, FOR and the LOOP structures may be nested, as long as the level of nestings remains less than 10. The TIME LOOP may not be nested. It may contain, however other kinds of loops.

Hence we may have:

```
WHILE ( boolean1 ) DO
    command_1
```



```

command_2
WHILE ( boolean2 ) DO
  command_1
  WHILE ( boolean3 ) DO
    command_4
  END_WHILE
END_WHILE
command_3
END_WHILE

```

or

```

START_LOOP, sequence_number = 1
  command_1
  command_2
START_LOOP, sequence_number = 2
  command_1
  WHILE ( boolean3 ) DO
    command_4
  END_WHILE
END_LOOP
command_3
END_LOOP

```

WHILE executes all commands following it till the end_while statement. This is repeated as long as boolean returns with the value *true*.

boolean may be a standard boolean, like $a > b$ or $c == d$, but may also be of the shape `boolean_expr(k)`. It defines whether the while command must be executed or not.

boolean_expr(k) refers to a user written subroutine USERBOOL with one parameter (k), which may be used to identify the call.

USERBOOL must be written by the user as described in Section 3.3.8.

END_WHILE indicates the end of the while loop.

IF executes all commands following it till the end_if statement provided boolean returns with the value *true*. Also in this case the boolean may be of type `boolean_expr(k)`.

FOR executes all commands following it till the end_for statement. The loop variable must be declared as a variable in the block constants (See Section 1.4). This loop variable starts with the value a . After the `end_for` is reached the value of the loop variable is increased by c until it is larger than b . If `step c` is omitted, step 1 is assumed.

a , b and c must be integers.

The loop variable may be used for computations, but it may not be changed within the loop.

For an example of the use of the for loop see 6.2.9.

END_FOR indicates the end of the for loop.

START_LOOP , sequence_number = k .

All commands that are given from this command until `end_loop` is found will be considered as part of the loop. The sequence number k refers to the input block "LOOP_INPUT". This blocks describes under what conditions the process has been converged. See Section 3.2.17.

END_LOOP Ends the loop.

START_TIME_LOOP All commands that are given from this command until `end_time_loop` is found will be considered as part of the computation of a time-dependent problem.

TIME_INTEGRATION , `sequence_number = s`, `vector = i`, defines what time integration must be carried out in the computation of the time loop. This statement may be preceded by some statements to make preparations for the time integration, and succeeded by statements manipulating the computed solution. Only one time-step is carried out.

The sequence number s refers to the input block `TIME_INTEGRATION`, which defines the parameters of the time integration process. See Section 3.2.15 for a description. At this moment only one fixed time step may be used. So only one end time and one time step may be given. Furthermore the only available time integration at this moment is Euler implicit. If you need extra time integrations please contact SEpra.

The options with respect to `TOUT`, like `TOUTINIT`, have at the present moment no effect at all.

Of course the options referring to the stationary accuracy make also no sense in this case.

The sequence number i defines the vector to be integrated in time.

The Loop may be preceded by the command

TIME_HISTORY V_i This command defines in which points a vector V_i is stored during each time step. Later on this time history can be printed by `PRINT time_history V_i` (Section 3.2.3.12) or plotted by `PLOT_TIME_HISTORY V_i` (Section 3.2.3.13).

The following options are available:

```
coordinates ((x1,y1,z1),(x2,y2,z2),... (xn,yn,zn))
points ( p1, p2, ... )
zero_level_set
```

Meaning of these options:

coordinates ((x_1,y_1,z_1),(x_2,y_2,z_2),...(x_n,y_n,z_n)) defines for which coordinates the function must be stored. At this moment the nodal point that is the closest to each coordinate is used. This nodal point is kept fixed even if the coordinates change during computation. Each set or coordinates must be enclosed by brackets.

points (p_1, p_2, \dots) defines for which user points the function must be stored.

zero_level_set stores the time history for all nodes in the zero levelset.

END_TIME_LOOP Ends the loop for the time integration.

For an example of the use of `TIME_LOOP` see Section 6.4.5.

BREAK inside one of these loops causes the program to jump out of the actual loop. So usually `break` is used in combination of an `if` statement.

3.2.4 The main keyword MATRIX

The block defined by the main keyword MATRIX defines the structure of the large matrix and hence implicitly the linear solver to be used. SEPRAN distinguishes between symmetric and non-symmetric, real and complex matrices. Furthermore storage schemes for direct methods differ from the storage scheme for iterative solvers.

Whether the large matrix is symmetrical or not depends on the type of problem to be solved. In the manual STANDARD PROBLEMS for each problem it is given whether the matrix is symmetrical or not. This is also the case for real and complex matrices. Each symmetrical matrix may of course be stored as a non-symmetrical matrix, however, the storage needed doubles in general and also the computation time may increase. A real matrix may in general not be stored as a complex matrix. The choice between a direct linear solver and an iterative linear solver is not so easy to make. In general a direct solver is the most robust and most simple to use. However, for large problems in R^2 and smaller problems in R^3 iterative solvers use much less memory and often also less computation time. However, for some problems iterative solvers converge very slowly or even diverge. Unfortunately no hard criterion can be formulated when one method is preferred above the other one. An important remark is that in the case of time-dependent problems and sometimes also stationary non-linear problems in general a good initial estimate of the solution is available. In combination with a not too strict termination criterion this makes the iterative solvers more favorable.

The block defined by the main keyword MATRIX has the following structure (options are indicated between the square brackets "[" and "]"):

MATRIX [,SEQUENCE_NUMBER = *s*] (mandatory)

COMMAND record: indicates that information of the structure of the large matrix will be given.

The sequence number *s* may be used to distinguish between various input blocks with respect to the matrix structure.

This record must be followed by DATA records giving information of the structure of the large matrix.

```
PROBLEM = 1 [options] (mandatory)
PROBLEM = 2 [options]
.
.
.
```

The following options are available (all in one line)

```
STORAGE_SCHEME = s
NOSPLIT, PRINT_LEVEL = p
EXTRA_FILLIN = f
PHYS = (iphys1 TO iphys2 ),
MESH = m
DECOUPLED_DEGFD = (i, j, ...)
SKIP_BOUNDARY_CONDITIONS
SYMMETRIC
COMPLEX
REAL
UNSYMMETRIC
INCOMPRESSIBILITY_SYM
INCOMPRESSIBILITY_UNSYM
ASSEMBLE_PRECON
REACTION_FORCE
```

SHIFTED_LAPLACE
 MUMPS
 POSITIVE_DEFINITE

For each problem as given in the input block "PROBLEM" such a DATA record may be defined. If the data record is omitted for a PROBLEM, METHOD = 2 is assumed. The DATA records must be given in the order of the problems. If only one problem is solved PROBLEM = 1 may be omitted.

STORAGE_SCHEME = *s* defines the storage scheme of the large matrix and hence the structure. This storage scheme is coupled to the linear solver. In fact the storage scheme defines which solvers can be applied.

Possible values for *s* are:

PROFILE
 COMPACT
 SIMPLE
 ROW_COMPACT
 VECTOR_COMPUTER
 MUMPS
 PETSC

Meaning of these keywords:

PROFILE means that the matrix is stored as profile matrix, which is the most optimal storage for LU decomposition. Hence if this keyword is used, a direct linear solver will be used.

For middle sized 2d and small 3d problems this is the most efficient type of solver.

COMPACT the matrix is stored as compact matrix. Only the non-zero elements are stored. This means that an iterative solver like for example CG or overrelaxation will be used.

The symmetry of the data structure is utilized.

Iterative solvers are necessary for large 2d and middle sized 3d problems, but also for problems where the solution is not unique, for example because the solution is determined except for an additive constant.

In the case of time dependent problems or non-linear problems, iterative methods may be efficient, since a good initial estimate is available from the prior iteration or time step.

ROW_COMPACT is the same as compact, except that the symmetry of the data structure is not used. Hence the storage of the matrix structure requires approximately twice the space as for **COMPACT**.

This storage must only be used for special iterative solvers.

VECTOR_COMPUTER Combined row/column compressed storage scheme, designed for iterative solvers on vector processors. Moreover, an internal permutation of rows and columns is used. It stores the whole off-diagonal part of the matrix. It has no effect on scalar processors.

At this moment there is no reason to use this kind of storage.

SIMPLE is a very special storage scheme corresponding to a set of particular solvers for the Navier-Stokes equations. Only when these solvers are used, the keyword **SIMPLE** should be used.

MUMPS stores the matrix in row compact form as described in row_compact. In combination with symmetric only the lower part of the column numbers are stored. If the matrix is positive definite this may also be combined with the keyword **positive_definite**.

If this storage is used the linear solver calls an interface to the mumps package, which makes only sense if you have the mumps package installed.

PETSC stores the matrix in row compact form as described in `row_compact`.

If this storage is used the linear solver calls an interface to the `petsc` package, which makes only sense if you have the `petsc` package installed.

If the keyword `STORAGE_SCHEME` is skipped, the default `PROFILE` is used.

REAL indicates that the matrix is real (Default value).

COMPLEX indicates that the matrix is complex.

SYMMETRIC indicates that the matrix is symmetric and hence only the lower part will be stored. In case of a simple method, this refers to the velocity part of the matrix only.

UNSYMMETRIC indicates that the matrix is un-symmetric and hence the whole matrix must be stored (Default value).

INCOMPRESSIBILITY_SYM can only be used in combination with the storage scheme `SIMPLE`. If used the gradient pressure matrix is supposed to be the transpose of the divergence velocity matrix. Since this is usually the case this is the default value.

INCOMPRESSIBILITY_UNSYM has the same meaning as the previous keyword, but in this case the gradient and divergence matrix are different.

ASSEMBLE_PRECON in this case not only the standard matrix must be assembled, but also a matrix that is assembled from element matrices that are created by computing the LU decomposition of the element matrix. This last matrix may be used as preconditioner for for example Conjugate Gradient solvers.

REACTION_FORCE if this keyword is used an extra part of the matrix corresponding to essential boundary conditions is stored. This part is only needed in case one needs to compute the reaction forces. The sign of `IBCMAT` must be equal to the sign of `JMETHOD`.

NOSPLIT indicates that the large matrix may not be split into parts if this matrix does not fit into memory. So instead of splitting an error message is given.

An advantage of this option may be that instead of the expensive splitting of the matrix, a larger `BUFFER` array is defined which may avoid the splitting.

This option is only activated for direct methods.

SHIFTED LAPLACE makes only sense in combination with `storage_scheme = compact`.

It is used in case the shifted Laplace preconditioner is used, which is meant for Helmholtz type equations. Using this option means that extra space is reserved to store the diagonal of the discretization of the *zeroth order* part of the equations.

PRINT_LEVEL= p defines if the type of output that is produced. The following values of p are available:

0 No extra output is printed.

1 The contents of the matrix structure array corresponding to the internal unknowns is printed.

2 The complete contents of array matrix structure array is printed including the relation with the boundary conditions.

10-12 The structure of the large matrix is printed formally. This possibility is only available in combination with `JMETHOD = 1, 2, 3 or 4`.

Default value: $p = 0$.

EXTRA_FILLIN= f defines if extra fill-in must be created or not. Extra fill-in makes only sense if an iterative solution method is used (hence a compact storage: `JMETHOD = 5,6,7 or 8`) in combination with an ILU preconditioner. The extra fill-in may produce a more suitable preconditioner, however, also requires extra memory. The following values of f are available:

0 No extra fill-in is produced.

1 Special case for discontinuous Navier-Stokes elements. The connections of the centroids of the elements with the centroids of neighboring elements are also filled in the structure for all physical variables `iphys1` to `iphys2`

- 2 All unknowns in the neighbor points of the unknowns in a row of the matrix are also supposed to be part of the non-zero structure. This enlarges the space used by the matrix and the preconditioner, but does not change the matrix at all.
By adding extra fill-in, the ILU preconditioner may be a better approximation for the real LU. As a consequence the number of iterations may decrease at the cost of extra operations (time) per iteration and also extra memory.
- 3 Combination of 2 and 3.
- 4 Special case: only rows corresponding to nodes with more than $ndim+1$ unknowns are used for the extra fill-in as defined in 2. So this reduces the fill-in compared to 2. This option makes only sense in the case that the number of degrees of freedom in some points is more than $ndim+1$.
- 5 See 4, of the extra nodes only those with more. than $ndim+1$ unknowns are used.
- 6 Special case: only rows corresponding to unknowns with physical number $> ndim+1$ are used for the extra fill-in as defined in 2.
- 7 See 6, of the extra unknowns only those with physical degree of freedom $> ndim+1$ are used.

Mark that in the range 3 to 7, the extra fill-in is decreasing.

Default value: $f = 0$.

PHYS = (**iphys1 TO iphys2**) is used in combination with EXTRA_FILLIN.

Default values: $iphys1 = ndim + 1$, $iphys2 = ndim + 1$ for linear elements and $iphys2 = 2ndim + 1$ otherwise.

DECOUPLED_DEGFD = (i_1, i_2, i_3, \dots) indicates that the degrees of freedom i_1, i_2, i_3, \dots in each nodal point are not coupled. This means that the corresponding elements in the element matrix are equal to 0. For example, suppose we have two velocity degrees of freedom in each node, u_1 and u_2 , the system of equations will have the structure:

$$\begin{bmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} \\ \mathbf{S}_{21} & \mathbf{S}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{bmatrix} \quad (3.2.4.1)$$

Suppose we know that the matrices \mathbf{S}_{12} and \mathbf{S}_{21} are equal to zero. Then the components u_1 and u_2 are not coupled and we can use `decoupled_degfd = (1,2)`. The advantage is that less memory is necessary.

In the sequel we shall speak about a reduced storage scheme.

This option is only effective in case of a compact storage. In case of a profile method the option is neglected.

SKIP_BOUNDARY_CONDITIONS is used in the case that we do not need the effect of the essential boundary conditions in the matrix. This option is meant in case the structure of the matrix is only meant for a preconditioner and for solving the system of equations. In that case we need at least two input blocks "MATRIX".

MESH = m is used to define which mesh must be used for the definition of the matrix structure. This option makes only sense if spectral elements are used, otherwise it is neglected.

Possible values for m are

FEM_MESH
SEM_MESH

Meaning of these subkeywords:

FEM_MESH The matrix structure is based on the finite element mesh consisting of linear elements that is constructed from the spectral element mesh. This mesh has the same number of nodes as the spectral mesh but a lot more elements.

This option is necessary if the spectral problem solved by an iterative conjugate gradient method, using the finite element matrix corresponding to linear elements as preconditioner.

SEM_MESH The matrix structure is based on the spectral element mesh. This means that no finite element preconditioning is applied, but only the standard linear solver.

The default value is: **SEM_MESH**

END (mandatory)

COMMAND record: end of the input of the block MATRIX.

Remarks:

- if the block corresponding to MATRIX is skipped **METHOD = 2** is assumed for all problems.
- In previous versions of SEPRAN the keywords **STORAGE_SCHEME**, **SYMMETRIC**, **COMPLEX**, **REAL**, **UNSYMMETRIC**, **INCOMPRESSIBILITY_SYM**, **INCOMPRESSIBILITY_UNSYM**, **ASSEMBLE_PRECON**, and **REACTION_FORCE**, where replace by one keyword **METHOD = i** consists of three parts **JMETHOD**, **IBCMAT** and **INVMAT** according to $|i_k| = |JMETHOD + 100 \times (INVMAT + 2 \times IBCMAT)|$ with

JMETHOD Parameter to indicate which solution method is chosen.

Possibilities:

- 1 Symmetric real profile matrix
- 2 Non-symmetric real profile matrix
- 3 Symmetric complex profile matrix
- 4 Non-symmetric complex profile matrix
- 5 Symmetric real compact matrix
- 6 Non-symmetric real compact matrix
- 7 Symmetric complex compact matrix
- 8 Non-symmetric complex compact matrix
- 9 Non-symmetric real compact matrix using the row compact storage
- 10 Non-symmetric complex compact matrix using the row compact storage
- 11 Non-symmetric real compact matrix using the vector compact storage
- 12 Non-symmetric complex compact matrix using the vector compact storage
- 13 **STORAGE_SCHEME = Simple, SYMMETRIC, INCOMPRESSIBILITY_SYM**
- 14 **STORAGE_SCHEME = Simple, UNSYMMETRIC, INCOMPRESSIBILITY_SYM**
- 15 **STORAGE_SCHEME = Simple, SYMMETRIC, INCOMPRESSIBILITY_UNSYM**
- 16 **STORAGE_SCHEME = Simple, UNSYMMETRIC, INCOMPRESSIBILITY_UNSYM**
- 19 The matrix is complex (hermitian), profile method.

24-27 See 5 to 8, however, the shifted Laplace storage is used.

INVMAT is 1 if **ASSEMBLE_PRECON** is used, 0 otherwise.

IBCMAT is 1 if **REACTION_FORCE** is used, 0 otherwise.

3.2.5 The main keywords ESSENTIAL BOUNDARY CONDITIONS

The block defined by the main keywords ESSENTIAL BOUNDARY CONDITIONS defines whether the solution vector is real or complex and also defines the values of the essential boundary conditions. At which boundaries essential boundary conditions are given has already been described in the part PROBLEM. In fact all essential boundary conditions that are not explicitly given in this part are set equal to zero.

The block defined by the main keywords ESSENTIAL BOUNDARY CONDITIONS has the following structure (options are indicated between the square brackets "[" and "]"):

ESSENTIAL [COMPLEX] BOUNDARY CONDITIONS [SEQUENCE_NUMBER = s]
 [PROBLEM = p] [VECTOR= i] (mandatory)
 COMMAND record: opens the input.

The various options have the following meaning:

COMPLEX indicates that the solution vector is a complex vector.

SEQUENCE_NUMBER = s may be used to distinguish between various input blocks with respect to the essential boundary conditions.

VECTOR = i is used to define the i^{th} vector in a row of vectors. If omitted the first vector in this block has sequence number 1, and all other vectors have the sequence number of the preceding one plus 1.

If the block STRUCTURE is used and in this block the command PRESCRIBE_BOUNDARY_CONDITIONS, sequence_number = s , vector = v is given, then the actual vector sequence number is $v - 1 + i$.

PROBLEM = p is used to define the problem sequence number corresponding to the solution vector to be filled. If omitted $p = i$ is assumed, with i the sequence number of the vector.

Must be followed by data records defining the essential boundary conditions. Only the non-zero essential boundary conditions must be specified in this part. If all essential boundary conditions are zero, no DATA records are necessary.

The data records are exactly the same as the data records for the input block "CREATE" (Section 3.2.10) defined by the part

[functional description] [degrees of freedom] [location part]

END (mandatory)

COMMAND record: end of the block ESSENTIAL BOUNDARY CONDITIONS.

It is allowed to define boundary conditions for more than one vector in one block ESSENTIAL BOUNDARY CONDITIONS. In that case the following type of input may be used:

```
ESSENTIAL BOUNDARY CONDITIONS PROBLEM 1, VECTOR = 2, SEQUENCE_NUMBER 1
.
.
.
.
ESSENTIAL BOUNDARY CONDITIONS PROBLEM 2, VECTOR = 5
.
.
.
.
ESSENTIAL BOUNDARY CONDITIONS PROBLEM 3
.
.
.
.
END
```

In this case only one END record must be used for this sequence number.

Remarks:

In fact the input block "ESSENTIAL BOUNDARY CONDITIONS" is completely identical to the input block "CREATE". However, there is one essential difference. If in the part "functional description" FUNC is used then the function subroutine FUNCBC (INTRODUCTION 5.5.1) is used to define the function in the real case and CFUNCB (INTRODUCTION 5.5.2) in the complex case. This is in contrast to the block CREATE where the function subroutines FUNC respectively CFUNC (INTRODUCTION 5.5.4) are used.

If the block corresponding to ESSENTIAL BOUNDARY CONDITIONS is skipped it is assumed that all essential boundary conditions have the value 0 and moreover, that the solution vector is real. So in case of a complex problem always the part ESSENTIAL BOUNDARY CONDITIONS must be given.

3.2.6 The main keyword COEFFICIENTS

The block defined by the main keyword COEFFICIENTS defines the values of the coefficients for the differential equations to be solved as well as the coefficients to be used in the natural boundary conditions. Furthermore in this block extra information may be given which defines how the matrix and right-hand side must be built. This information may for example be the type of numerical integration rule to be applied in the computation of the element matrices and element vectors, the type of co-ordinate system to be applied (Cartesian, axi-symmetric etc.) or information about the type of linearization to be used in a non-linear problem.

Coefficients may be of real type or of integer type. Real coefficients are in general the coefficients in the differential equation itself. Integer coefficients correspond to types of methods to be used (integration, co-ordinate system, linearization and so on).

Each coefficient corresponding to a differential equation, both real and integer gets a unique sequence number.

Which coefficients must be given by the user for a specific partial differential equation and which corresponding sequence number must be used is given in the manual STANDARD PROBLEMS. There is no general rule in this case. So it is always necessary to consult the manual STANDARD PROBLEMS.

The block defined by the main keyword COEFFICIENTS has the following structure (options are indicated between the square brackets "[" and "]"):

```
COMPLEX COEFFICIENTS [,SEQUENCE_NUMBER = s] [, PROBLEM = p]
(mandatory if type numbers larger than 99 are used)
COMMAND record: opens the input block.
```

The option COMPLEX indicates that the coefficients for a complex problem are defined. The sequence number *s* may be used to distinguish between various input blocks with respect to the coefficients.

The problem sequence number *p* may be used to define the problem number corresponding to the matrix and vector to be filled. If omitted the next problem number is assumed.

Must be followed by data records defining the coefficients of the problems to be solved. For each standard element corresponding to an element group the user must specify some coefficients. For a definition of the specific coefficients corresponding to a particular problem, the reader is referred to the manual STANDARD PROBLEMS.

Coefficients must be defined in the following way

```
ELGRP 1
  COEF i_1 = ( VALUE = 3.5 )
  COEF i_2 = ( FUNC = 2 )
  ICOEF i_3 = 11
  COEF i_4 = (POINTS, IREF = r_1)
  COEF i_5 = (ELEMENTS, IREF = r_2)
  COEF i_6 = COEF i_5
  COEF i_7 = 3.5
  COEF J_8 = (SOL_FUNC= k)
  COEF J_9 = %name_scalar
ELGRP 2
  COEF j_1 = ( FUNC = 6 )
  COEF j_2 = ( VALUE = 2.956D0 )
  COEF j_3 = ( OLD SOLUTION j [, DEGREE OF FREEDOM d] )
  COEF j_4 = ( OLD SOLUTION j [, DEGREE OF FREEDOM d] [,COEF (VALUE=1.2)] )
  COEF j_4 = ( OLD SOLUTION j [, DEGREE OF FREEDOM d] [,COEF = 1.2] )
  COEF j_5 = ( OLD SOLUTION j [, DEGREE OF FREEDOM d] [,COEF (FUNC=1)] )
```

The element groups (ELGRP 1, ELGRP 2, . . .) must be given in a natural sequence. Only those element groups corresponding to type numbers larger than 99 must be used. Instead of ELGRP i one may also use ELGRP i to j , if all element groups in the range i to j have the same input for the coefficients.

COEF $i = (\text{VALUE} = 3.5)$ means that coefficient i is a real coefficient with value 3.5.

COEF $i = \%name_scalar$ means that coefficient i is a real coefficient with value of the scalar with name `name_scalar`.

This option can only be used when scalars are used and the name of the scalars have been defined in the input block `CONSTANTS` as treated in Section 1.4.

See also the input block `STRUCTURE` in Section 3.2.3.

COEF $i = 3$ means that coefficient i is a real coefficient with value 3.

COEF $j = (\text{FUNC} = k)$ implicates that the j^{th} coefficient is a function of the co-ordinates.

In that case the user must submit a function subroutine `FUNCCF` as described in the `SEPRAN INTRODUCTION 5.5.3`. The value k following the equals sign corresponds to the parameter `IFUNC` in `FUNCCF`, hence in this example `IFUNC` in the call of `FUNCCF` is equal to k .

`IFUNC` must be in the range $[1, 1000]$. Values of $k > 1000$ have a special meaning.

If $1000 < k < 2000$, the function subroutine `FUNCC1` is called instead of `FUNCCF` with parameter `IFUNC-1000`. For a description of `FUNCC1` see Section 3.3.6. In this case the coefficient is a function of the co-ordinates and the previous solution.

if $k > 10000$, the function subroutine `FUNCC3` is called instead of `FUNCCF` with parameter `IFUNC-10000`. In this case the coefficient is a function of the co-ordinates and all the vectors that are defined and present in `SEPCOMP`.

Effectively this is the same as using `SOL_FUNC JFUNC`, with `JFUNC = IFUNC-10000`. For some problems the user must define an integer choice parameter instead of an actual coefficient. In that case `ICOEF $k = 11$` , defines the value of the k^{th} coefficient to be equal to 11.

COEF $i_4 = (\text{POINTS}, \text{IREF} = r_1)$, means that coefficient i_4 is given in all nodal points.

If $r_1 > 0$, then the coefficients are read from a file with reference number `IREF= r_1` . If $r_1 < 0$, the user must provide a subroutine `FUNCCFL` as described in 3.3.3.

COEF $i_5 = (\text{ELEMENTS}, \text{IREF} = r_2)$, means that coefficient i_5 is given per element.

If $r_2 > 0$, the coefficients are read from a file with reference number `IREF= r_2` .

COEF $i_6 = \text{COEF } i_5$ means that coefficient 6 is equal to coefficient 5. Of course coefficient 5 must have been defined before.

COEF $i_7 = \text{SOL_FUNC} = k$ means that coefficient i_7 is given as function of the co-ordinates and the previous computed solutions or otherwise defined vectors. In that case the user must provide a user function `FUNCC3` as described in Section 3.3.6. The value of k corresponds to the parameter `IFUNC` in `FUNCC3`.

COEF $j_3 = (\text{OLD SOLUTION } j, \text{DEGREE OF FREEDOM } d)$, means that coefficient j_3 is equal to the d^{th} degree of freedom of the vector `Vj`. Of course instead of j one may use the name of the vector.

If `DEGREE OF FREEDOM d` is omitted the first degree of freedom in each nodal point is used.

COEF $j_4 = (\text{OLD SOLUTION } j, \text{DEGREE OF FREEDOM } d, \text{COEF (VALUE= v))$

, or **COEF $j_4 = (\text{OLD SOLUTION } j [, \text{DEGREE OF FREEDOM } d] [, \text{COEF} = v])$** , means that coefficient j_4 is equal to the d^{th} degree of freedom of the vector `Vj` multiplied by v .

If `DEGREE OF FREEDOM d` is omitted the first degree of freedom in each nodal point is used.

COEF $j_5 = (\text{OLD SOLUTION } j, \text{DEGREE OF FREEDOM } d, \text{COEF (FUNC= f))$

, means that coefficient j_5 is equal to the d^{th} degree of freedom of the vector `Vj` multiplied by the function described by f . This means that the user function `FUNCCF` is

called with parameter ICHOIS equal to f in order to compute the multiplication factor. If DEGREE OF FREEDOM d is omitted the first degree of freedom in each nodal point is used.

When the coefficients for all element groups have been defined, then the coefficients for all boundary element groups must be defined in a natural sequence. These boundary element groups define the so-called natural boundary conditions. See also the manual STANDARD PROBLEMS for an exact definition of what the natural boundary conditions are in a specific case. The input very much resembles the input for the standard problems:

```
BNGRP 1
  COEF i_1 = 2
  COEF i_2 = ( FUNC = 1 )
BNGRP 2
  COEF j_1 = ( FUNC = 3 )
```

After the definition of the coefficients for all element groups and boundary element groups, the coefficients for all global element groups must be defined in a natural sequence. These global element groups define the so-called global unknowns. See 3.2.3. See also the manual STANDARD PROBLEMS for an exact definition of what the global unknowns are in a specific case. The input very much resembles the input for the standard problems:

```
GLGRP 1
  COEF i_1 = 2
  COEF i_2 = ( FUNC = 1 )
GLGRP 2
  COEF j_1 = ( FUNC = 3 )
```

END (mandatory)

COMMAND record: end of input for the block COEFFICIENTS.

3.2.7 The main keywords CHANGE COEFFICIENTS

The block defined by the main keywords CHANGE COEFFICIENTS defines which coefficients (integer or reals must be changed). So the use of this block assumes that already coefficients have been defined. Only those coefficients that must be changed with respect to the previous definition have to be given. CHANGE COEFFICIENTS is only used in combination with NON-LINEAR PROBLEM. Since more than one section CHANGE COEFFICIENTS is allowed, CHANGE COEFFICIENTS must be provided with a sequence number.

The block defined by the main keyword CHANGE COEFFICIENTS has the following structure (options are indicated between the square brackets "[" and "]"):

```
CHANGE [COMPLEX] COEFFICIENTS [,SEQUENCE_NUMBER = k]
```

COMMAND record: opens the input for change coefficients.

must be followed by data records defining the coefficients to be changed of the problems to be solved. For each standard element corresponding to an element group the user may specify some coefficients. The keyword COMPLEX indicates that the coefficients may be complex. All real coefficients are treated as if they are complex. The sequence number k may be used in NONLINEAR EQUATIONS to identify this input.

Coefficients may be defined in exactly the same as for COEFFICIENTS, except that the number of parameters does not have to be given, since this number is copied from COEFFICIENTS. Example:

```
ELGRP 1
  COEF i1 = ( VALUE = 3.5 ) or COEF i1 = 3.5
  COEF i2 = ( FUNC = 2 )
  ICOEF i3 = 11
  COEF i6 = COEF i5
  .
  .
ELGRP 2
  COEF j1 = ( FUNC = 6 )
  COEF j2 = ( VALUE = 2.956D0 )
  .
  .
```

When the coefficients for all element groups have been defined, the coefficients for all boundary element groups must be defined in a natural sequence. For example:

```
BNGRP 1
  COEF i1 = 2
  COEF i2 = ( FUNC = 1 )
BNGRP 2
  COEF j1 = ( FUNC = 3 )
  .
  .
END (mandatory)
```

COMMAND record: end of input for change coefficients.

3.2.8 The main keyword SOLVE

The block defined by the main keyword SOLVE gives information with respect to the linear solver to be used. Even in a non-linear problem a series of linear problems is solved, and hence this block makes also sense in that case.

The type of linear solver to be used (direct or iterative) has already been defined in the block MATRIX. In this part some extra information for the solver may be defined.

In the remainder of this section we denote the system of equations to be solved by:

$$\mathbf{Su} = \mathbf{f}.$$

The block defined by the main keyword SOLVE has the following structure (An option is indicated like this [*option*]):

```
SOLVE [,SEQUENCE_NUMBER = s]
  POSITIVE_DEFINITE
  ITERATION_METHOD = iter_method [,options]
  DIRECT_SOLVER = method
  RESIDUAL = res
  SYMMETRIC
  DEFECT_CORRECTION [,options]
  SPECTRAL [,options]
  PROJECTION_METHOD = p [,options]
  ISEQ_EXACT = i
  ISEQ_RES_EXACT = i
  ISEQ_START_RES = i
  ISEQ_END_RES = i
  LIMIT_SOLUTION
  OVER_PRESSURES [,options]
  SUB_EQUATION i
END
```

SOLVE opens the input for the linear solver.

The sequence number *s* may be used to distinguish between various input blocks with respect to the linear solver. The sequence of the subkeywords is arbitrary and none of these subkeywords is mandatory.

POSITIVE_DEFINITE indicates that the matrix to be solved is not only symmetrical, but also positive definite. This is only used in case of a direct profile solver. This option may not only improve the computation time, it also offers an extra check on the correctness of the input.

ITERATION_METHOD defines the type of iteration method to be used as well as the options to be applied. If the structure of the matrix as defined in the block MATRIX ... END by `storage_method = profile` the input about the iteration method is neglected, since then always a direct solver will be used.

iter_method may take one of the following values

```
CG
CGS
BICGSTAB
IDR
SYMMLQ
GMRES
GMRESR
OVERRELAXATION
GCR
```

```

SIMPLE_GCR
SIMPLER_GCR
HSIMPLER_GCR
MSIMPLER_GCR
AL_GCR
LSC_GCR
QLSC_GCR
SCHUR
BLOCK_TRIANGULAR

```

If `storage_method = compact`, `symetric` (symmetrical real or complex matrix) is used in combination with `iteration_method` always the conjugate gradient method (CG) (See Golub and van Loan [6]) is used. If the matrix is non-symmetrical real or complex, the method is defined by `iter_method`.

For vector computers, `storage_method = row_compact` or `vector_computer` may be used for the CG-type solvers. However, at the moment no preconditioning is available for this type of storage.

CG means, use the conjugate gradient method (cg) in case of a symmetric matrix or the bi-cgstab(1) method of Sonneveld and van der Vorst [2] in case of a unsymmetric matrix.

CGS activates the conjugate gradients squared method of Sonneveld [5].

BICGSTAB is identical to CG.

IDR uses the IDR(s) method of Sonneveld and van Gijzen [].

GMRES uses the so-called GMRES method with restart (See Saad [3]).

GMRESR (v.d.Vorst and Vuik [4]) is a type of GMRES method with as inner loop the GMRES method and as outer loop the GCR method using a variable polynomial preconditioner. If the number of iterations for GMRES is large and a super linear convergence is visible, GMRESR may improve the convergence speed compared to GMRES.

GCR The GCR method is used.

Actually this is the same as GMRESR with `NINNER = 1`.

OVERRELAXATION activates the overrelaxation process. However, for this type of iteration the storage scheme may not be `compact`,, but must be `row_compact`.

SIMPLE_GCR implies that the *simple* method is used to solve coupled equations. At this moment this option is only available for velocity and pressure coupling.

The system of equations as described in Section 3.2.4, The system of equations is considered as a system consisting of pressure and velocity blocks. This complete system is solved iteratively using a block preconditioner. In each step of the process, first the momentum equations are solved, then the pressure correction, and finally the velocity is adapted. To perform this overall iteration, the GCR method is used. In fact the *simple* step is used as preconditioner for GCR.

The system of equations in each substep are solved by the methods following the keywords, `SUB_EQUATION 1`, and `SUB_EQUATION 2`, respectively.

`SUB_EQUATION 1`, refers to the velocity (momentum equations), and `SUB_EQUATION 2`, to the pressure equations.

So if `SIMPLE_GCR` is used, one has to satisfy the following requirements:

- The problem to be solved, needs both velocity and pressure unknowns.
- The storage scheme in the input block `MATRIX`, must have the value `SIMPLE`.
- The keywords, `SUB_EQUATION 1`, and `SUB_EQUATION 2`, must be present in this `SOLVE` block.

The present version of *simple* is far from optimal, in fact it is rather slow, but in the future, faster methods will be implemented.

For an example of the use of *simple* see the manual EXAMPLES, Sections 7.1.20 and 7.1.21.

SIMPLER_GCR implies that the *simpler* method is used as preconditioner instead of the *simple* method. In some cases this method performs better, especially in case of convection (Navier-Stokes).

HSIMPLER_GCR is a hybrid form of *simple* and *simpler*. In the first iteration *simple* is used, in all other iterations *simpler*. This method behaves better than *simpler* in case of the Stokes equations.

MSIMPLER_GCR is the same as `simpler_gcr`. The only difference is that the scaling matrix Q' is replaced by the diagonal of the velocity mass matrix instead of the diagonal of the velocity matrix.

AL_GCR implies that the *AL* method (Augmented-Lagrangian method) of Benzi et al is used. In this method the system of equations to be solved is updated by the product of divergence and gradient matrix multiplied by a parameter. In some cases this can be done on element level, (Crouzeix-Raviart elements), in other cases (Taylor-Hood elements), an adaptation of the method is required. For an example of the use of *AL* see the manual EXAMPLES, Sections 7.1.22 and 7.1.23.

This method requires the storage scheme **SIMPLE** in the input block **MATRIX**.

LSC_GCR uses the LSC method of Elman et al. to solve the system of equations arising from the Navier-Stokes equations. The diagonal of the mass velocity matrix is used as scaling matrix.

This method requires the storage scheme **SIMPLE** in the input block **MATRIX**.

QLSC_GCR is the same as `lsc_gcr`, but the diagonal of the velocity mass matrix is replaced by the diagonal of the velocity matrix.

SCHUR A schur complement iteration is used to solve the discretized Navier-Stokes equations. The intermediate steps are solved with an iterative method.

BLOCK_TRIANGULAR A block triangular preconditioner in combination with GCR is used to solve the discretized Navier-Stokes equations. The intermediate steps are solved with an iterative method.

The options following *iter_method* may be given in any sequence. They must be given in the same record. If it is not possible to give all options within 80 columns it is necessary to proceed on the next line. In that case the first line must be closed with the continuation mark // i.e slash immediately followed by another slash. This process may be used recursively. The following options are available:

```

PRECONDITIONING = prec
MAX_ITER = m
ACCURACY = epsilon
ABS_ACCURACY = eps1, REL_ACCURACY = eps2
PRINT_LEVEL = p
KRYLOV_SPACE = k
NTRUNC = n1
NINNER = n2
TERMINATION_CRIT = crit
KEEP_PRECONDITIONING
START = st
ADAPT_DIAGONAL = s
FACTOR = f
AT_ERROR = t
ISEQ_DIAG = i

```



```

ISEQ_PRECDIAG = i
PREC_LUMPING
TYPE_SCALING = type
SDIMENSION = s
LDIMENSION = s
STRUCTURE_PREMAT = i
OMEGA = om
LAPLACE_SHIFT = b

```

PRECONDITIONING defines the type of preconditioner to be used. The following values for *prec* are available:

```

none
diagonal
ilu
eisenstat
Gauss_Seidel
mod_eisenstat
old
block_ssor
block_ilu

```

none no preconditioner is used.

diagonal diagonal scaling of the matrix is used as preconditioner.

ilu the preconditioner is a so-called incomplete LU decomposition.

eisenstat incomplete LU decomposition where only the diagonal is changed. Efficient implementation of Eisenstat.

Gauss_Seidel preconditioning with a Gauss Seidel iteration.

mod_eisenstat modified incomplete LU decomposition according to Axelsson. Efficient implementation of Eisenstat. The factor for the modification may be defined by `factor = f`.

old a previously computed pre-conditioner is used.

block_ssor preconditioning with a Gauss Seidel iteration. The main difference with **Gauss_Seidel** is that all unknowns in one point are considered as a coupled block in the matrix. This means that the Gauss Seidel process is carried out with respect to these blocks instead of for all unknowns. This may improve the convergence when there are more unknowns per point.

Of course if there is only one unknown per point this is exactly the same as **Gauss_Seidel**.

block_ilu is the same as the **ilu** preconditioning. However, in this case a block incomplete LU decomposition is carried out. This may improve the convergence when there are more unknowns per point.

Of course if there is only one unknown per point this is exactly the same as **ilu**.

The default value is **eisenstat**.

PREC_LUMPING is only active if a preconditioning is applied.

The effect of this keyword is that if the off-diagonal elements of the matrix have the same sign as the diagonal of the matrix, these elements are added to the diagonal of the preconditioning matrix and made zero themselves (lumping), before the preconditioning is applied. This makes only sense if all diagonal elements have the same sign. For many physical problems this is indeed the case.

For some problems lumping may improve the convergence.

The default is not lumping, except when the preconditioning matrix does not exist. In that case automatically lumping is applied in order to make the solver more robust.

TYPE_SCALING = type defines the type of scaling that must be applied to the system of equations before solving.

This option is only available in case of a non-symmetric matrix in combination with a conjugate type solver.

At this moment the option is not implemented in combination with the Eisenstat preconditioner.

The main goal of the scaling is to get a more balanced system of equations. This makes the termination criterion more reliable and the convergence more smooth. It is especially useful in case some diagonal elements are large, for example because a boundary condition is implemented by imposing a large number on the main diagonal.

The scalings matrix is always a diagonal matrix D with elements equal to the sum of the absolute values of the row.

The following options for *type* are available:

```

none
row
symmetric
column

```

none means that no scaling takes place.

row means that the matrix and right-hand side are premultiplied by the matrix D :
 $D^{-1}Su = D^{-1}F$.

This option is the recommended to improve the termination criterion.

symmetric in this case the matrix is pre- and post multiplied by the square root of D . This introduces a symmetric type of scaling but it does not necessarily means that the termination criterion is improved in an optimal way: $D^{-\frac{1}{2}}SD^{-\frac{1}{2}}u = D^{-\frac{1}{2}}F$

column This option post multiplies the matrix by the diagonal matrix D . In fact it scales the solution vector, but it does not influence the residual and hence the termination criterion based on the residual: $SD^{-1}y = F$, with $y = Du$.

Default value: **none**.

Remark: if the termination criterion is based on the residual, it is recommended to use the row scaling.

For an example see the manual Standard Elements Section 7.1.17.

MAX_ITER defines the maximum number of iterations is restricted to m . If the number of iterations exceeds this maximum an error message is given and the program is halted.

The default value for m is the number of unknowns, but usually the process should be finished much earlier.

ACCURACY defines when the iteration process is terminated. If the error is less than ϵ the iteration is stopped.

The default value is $\epsilon = 10^{-3}$

ABS_ACCURACY may be used instead of accuracy or together with REL_ACCURACY.

If only ABS_ACCURACY is used it has the same meaning as ACCURACY, but the default termination criterion is set to absolute.

REL_ACCURACY may be used instead of accuracy or together with ABS_ACCURACY.

Used stand-alone there is no difference with ACCURACY, however, in combination with ABS_ACCURACY, both accuracy tests will be applied. If one of the tests is satisfied the process is assumed to be converged.

Remark: the combination **abs_accuracy** and **rel_accuracy** given is only active in case the keyword **termination_crit** is not given explicitly.

print_level = p defines the amount of output that is produced during the iteration process.

The lower the value of p the less output is produced. Possible values for p are:

-1 No print output

0 Only error messages are printed

1 A little amount of information of the iteration method is printed

2 A maximal amount of information of the iteration method is printed

- 3** See 2, however, in this case not only information about the iteration process is printed, but also the matrix and right-hand side are printed. Since in general the matrix may become very large, use of this option is not recommended. It should only be applied for research activities, and only in the case of a very small grid.

The default value is 0.

KRYLOV_SPACE = k defines the dimension of Krylov space for GMRES, i.e. number of iterations at which a restart is done. In case of GMRESR it concerns the outer loop. The default value is 20 for GMRES and 100 for GMRESR.

NTRUNC = $n1$ may be used in combination with GMRESR or CG for a symmetric matrix. In combination with GMRESR it defines the maximum number of search directions. $n1$ should be much smaller than k .

In combination with CG for a symmetric matrix it defines the maximum number of iterations to estimate the smallest eigenvalue. This eigenvalue is used to correct the termination criterion for the condition of the matrix. Once the number of iterations equals NTRUNC the then computed eigenvalue is used for the rest of the process. If the matrix is very ill-conditioned it may be necessary to choose NTRUNC larger than the default value.

The default value is 20.

NINNER = $n2$ defines the maximum number of iterations in the inner loop of the GMRESR process. The default value is 5.

termination_crit = *crit* defines the type of termination criterion. The iteration method is stopped if the accuracy of the solution is at least ϵ with respect to this criterion. Possible values for *crit* are:

absolute The process is stopped if $\|\mathbf{Res}_k\| < \epsilon$.

rel_residual The process is stopped if $\|\mathbf{Res}_k\| < \epsilon\|Res_0\|$.

rel_right_hand_side The process is stopped if $\|\mathbf{Res}_k\| < \epsilon\|\mathbf{f}\|$.

rel_solution The process is stopped if $\|\mathbf{Res}_k\| < \epsilon\|\mathbf{u}\|$.

The default value is rel_residual.

Here \mathbf{Res}_k denotes the residual $\mathbf{S}\mathbf{u}_k - \mathbf{f}$ during the k^{th} iteration, \mathbf{u}_k the solution at the k^{th} iteration and \mathbf{f} the right-hand side of the system of equations to be solved.

KEEP_PRECONDITIONING enforces the preconditioning matrix that has been computed to be kept for the next system of linear equations to be solved. This preconditioner is reused if preconditioning = old is set in the next system.

START = st defines how the initial vector must be chosen. Possible values for st are:

zero
old_solution
random

zero The start vector is the zero vector.

old_solution enforces the iteration process to use the old solution available. item[random]
The start vector is a random vector.

Default value: zero

AT_ERROR = t defines what the process must do if an error has been found, for example if the maximum number of iterations has been reached. Possible values for t are

stop
resume

stop means that the process is stopped as soon as an error has been detected.

resume means that the iteration process is left with a warning. The program is resumed with the last computed iteration.

Default value: **stop**

ADAPT_DIAGONAL = s may be used to change the diagonal of the preconditioning matrix. If this option is used, the diagonal of the preconditioning matrix is made equal to the minimum of the computed value and s .

Default value: $s=0$.

factor = f Defines the factor to be used in the modified Eisenstat preconditioning. This value must be between 0 and 1.

Default value: $f=0.95$.

ISEQ_DIAG = i If this option is given, the diagonal of the matrix is stored in the i^{th} solution vector. vector

ISEQ_PRECDIAG = i If this option is given, the diagonal of the preconditioning matrix is stored in the i^{th} solution vector. vector

SDIMENSION = s defines the parameter s in the IDR(s) method.

The default value of s is 1, which makes the method identical to bicg-stab.

LDIMENSION = l defines the parameter l in the Bi_CGSTAB(l) method.

The default value of l is 1.

STRUCTURE_PREMAT = is defines the matrix structure of the preconditioning matrix. Usually this structure is exactly the same as for the matrix to be solved. However, if the user wants, he may use a different structure. This structure must be given in a separate input block **MATRIX**. The value of is refers to the sequence number of the matrix structure. This sequence number is defined by the **change_structure_of_matrix** command in the **STRUCTURE** block, by the keyword **seq_storage** = is .

Hence **change_structure_of_matrix** must be called before the solution of the system of equations.

A reason to use **structure_premat** is for example if one needs extra fill in to get the system of equations converging or a reduced fill in, for example by decoupling of unknowns. in that case the option **skip_boundary_conditions** in the matrix block is recommended.

OMEGA = ω defines a relaxation parameter ω in the SIMPLE iteration. In some cases it might help to accelerate the convergence.

LAPLACE_SHIFT = β makes only sense in combination with **preconditioner** = **ilu**. It is only activated if the option **SHIFTED_LAPLACE** is used in the input block **MATRIX**, see (3.2.4).

The shifted Laplace preconditioner is meant for Helmholtz-type equations. Such an equation has the shape

$$-div \alpha \nabla c - k^2 c = f \quad (3.2.8.1)$$

An ILU preconditioner is based on the complete discretization matrix. In case of a shifted Laplace preconditioner the ILU matrix is constructed from the matrix corresponding to

$$-div \alpha \nabla c + \beta k^2 c \quad (3.2.8.2)$$

So the preconditioning matrix is made positive definite instead of indefinite. The value of β defines the shift. At this moment only the diagonal of the matrix corresponding to the part $k^2 c$ is used, so the extra storage is equal to the number of unknowns. In case of a complex matrix **beta** is supposed to be complex and **beta** should be given by (**beta1**, **beta2**) or if β is real by **beta**.

For some experiments with the shifted Laplace preconditioner, we refer to the manual Examples, Section .

SUB_EQUATION i describes, how the i^{th} sub-equation in the simple method must be solved. This keyword must be followed by data records describing, which iteration

method must be used for the sub-equation. The option in these data records are exactly the same as for the standard iterative method.

So we might have for example

```
solve
  iteration_method = simple_gcr, preconditioning = ilu, print_level = 2 &
    start = old_solution, accuracy = 1d-9
  sub_equation 1
    iteration_method = cg, preconditioning = ilu, print_level = 0, eps = 0.1
  sub_equation 2
    iteration_method = cg, preconditioning = ilu, print_level = 0, eps = 0.1
end
```

If overrelaxation is used to solve the system of linear equations then it is necessary that `storage_scheme = row_compact` is set in the input block MATRIX. both for symmetric and non-symmetric matrices. In fact a possible symmetry is not utilized.

An extra option available for overrelaxation is that of constrained optimization. It is not only possible to solve a system of linear equations, but also the solution $u(i)$ may be computed such that it satisfies $a \leq u(i) \leq b$.

With respect to overrelaxation the following options are available:

```
MAX_ITER = m
ACCURACY= epsilon
START = OLD_SOLUTION
ALPHA = alpha
BETA = beta
OMEGA = omega
NITER1 = n1
LAMBDA = lambda
minimum = m1 or (value=m1) or (func=i1)
maximum = m2 or (value=m2) or (func=i2)
degfd = i
```

The options `max_iter`, `accuracy` and `start = old_solution` have exactly the same meaning as for the conjugate gradient type solvers.

The options `minimum`, `maximum` and `degfd` are used for constrained optimization only. They define the restrictions on the solution.

If `minimum = m1` or `minimum = (value=m1)` is given the solution is computed under the restriction $u(i) \geq m1$. The option `func = i1` has not yet been implemented.

If `maximum = m2` or `maximum = (value=m2)` is given the solution is computed under the restriction $u(i) \leq m2$. The option `func = i2` has not yet been implemented.

The options `alpha`, `beta`, `omega`, `niter1`, `niter2` and `lambda` may be used to influence the convergence of the overrelaxation process. See the part "How to influence the overrelaxation process" at the end of this section.

DIRECT_SOLVER = *method* , options

defines the type of direct solver to be used. At this moment there are two values for *method* available:

```
PROFILE [,options]
COMPACT [,options]
MUMPS [,options]
```

profile activates the standard direct profile solver. There is no need to give the command `direct_solver = profile` unless one of the options will be used. This direct solver may only be used if the storage scheme of the matrix corresponds to `storage_scheme = profile`. The profile solver does not use any form of pivoting. As a consequence in some cases the solution found is less accurate than possible for the system of equations to be solved. Although this situation is rare for systems of equations originating of the discretization of partial differential equations, SEPRAN offers the possibility to improve the solution by iterative improvement. As a check it is also possible to print the norm of the residual by the option `RESIDUAL = res`. In case of iterative improvement this option prints the residual during each step of the iteration process.

Iterative improvement is activated with the extra option

$$\text{NUM_ITER_REF} = n,$$

which defines the maximum number of extra iterative improvements. Iterative improvement stops if the residual is smaller than the accuracy ϵ defined by

$$\text{ACCURACY} = \epsilon$$

mumps makes only sense if you have the mumps package installed.

It may only be used in combination with the keyword MUMPS either in the input block MATRIX or in the command `matrix.structure` in the input block STRUCTURE.

The options give you the opportunity to give extra information to the MUMPS package. At this moment teh following options are available:

$$\text{renum_mumps} = i$$

`renum_mumps = i` defines the type of renumbering scheme as stored in ICNTL(7) in the mumps manual.

compact activates a very special direct solver based on a compact storage. To use this solver it is necessary to define `storage_scheme = compact`. This direct solver is in fact the method Y12M of Zlatev [1]. This method can only be used if the Y12M solver is available in your institute. Y12M is a direct solver, which uses pivoting. However, it is based on a compact storage. Extra space needed for the Gaussian elimination process is created during the process itself. For large systems of equations the storage required may be less than for the profile solver. At this moment the use of Y12M is not recommended unless the system of equations can only be solved if pivoting is applied.

With respect to Y12M the following options are available:

$$\begin{aligned} \text{droptolerance} &= d \\ \text{num_pivot_rows} &= m \\ \text{num_iter_ref} &= n \\ \text{accuracy} &= \text{epsilon} \\ \text{print_level} &= p \\ \text{lu_storage} &= n \end{aligned}$$

droptolerance = d indicates that all elements that are created during the Gaussian elimination process with a value smaller than d are set equal to zero and hence not stored.

The default value is 10^{-12} .

num_pivot_rows = m defines the number of rows to be used in the pivoting process.

The larger the value the more stable and expensive the method.

The default value is 3.

num_iter_ref = n defines the maximum number of iterative improvements to be performed after the Y12M method is applied. This option makes only sense if a large drop tolerance or no pivot rows are applied.

The default value is 0.

accuracy = ϵ defines the accuracy for the iterative improvement.

The default value is 10^{-3} .

print_level = p defines the amount of output produced by this method. Values between 0 and 2 are allowed, where 0 means that only error messages are printed and 2 gives the maximal amount of output.
The default value is 0.

lu_storage = n defines the amount of storage that must be defined for the LU-decomposition. n defines the multiplication factor, i.e. the space used for the LU-decomposition is $n \times$ the storage needed for the original matrix. If n is too large, possibly no space is available. If n is too small the process may become very slow or even may terminate without finding the solution.
The default value is 15.

RESIDUAL indicates if the norm of the residual $\|\mathbf{S}\mathbf{u} - \mathbf{f}\|$ must be printed, where \mathbf{S} defines the matrix, \mathbf{u} the solution and \mathbf{f} the right-hand-side vector. This option is only available for direct methods.

res defines the way the residual is computed. Possible values:

none The residual is not printed.

absolute The 2-norm of the residual is computed and printed.

relative The 2-norm of the residual divided by the 2-norm of the right-hand side is computed and printed.

The default value is none.

SYMMETRIC makes only sense in the case that a storage type corresponding to `storage_scheme = row_compact` or `storage_scheme = vector_computer` is used. It indicates that the matrix is symmetric although a non-symmetric storage is used.

DEFECT_CORRECTION implies that a defect correction method must be applied. This keyword makes only sense if in the building of the matrix already provisions for a defect correction process have been taken.

The defect correction process may be described as follows:

Suppose we want to solve $\mathbf{Ax} = \mathbf{b}$, but the matrix \mathbf{A} is not suited for iterative solution and direct solution is too expensive. A possible alternative is to solve an approximate equation $\tilde{\mathbf{A}}\mathbf{x}_1 = \mathbf{b}$ and to iterate to \mathbf{x} by the iteration process:

$$\tilde{\mathbf{A}}\mathbf{x}_k = \mathbf{b} - \mathbf{Ax}_{k-1} \quad (3.2.8.3)$$

So in this case we need two matrices \mathbf{A} and $\tilde{\mathbf{A}}$. A typical example is the solution of a convection equation by central differences, where the matrix may such that the iterative solution is hardly possible. An upwind matrix may be very suitable for the iterative solution but can lead to inaccurate solutions. In that case the upwind matrix may be used as approximate matrix in order to solve the central difference scheme. For some applications only one iteration of the defect correction scheme is sufficient to achieve an accurate solution in a very limited number of inner iterations.

Whether the defect correction method may be applied depends on the type of equation to be solved. Consult the manual standard problems if a type of equation allows for defect correction. This means that the matrix builder constructs two matrices at the same time.

The following options are available:

```
accuracy = eps
max_iter = m
print_level = p
solve_accuracy = eps
```

with

accuracy = ε defines the accuracy of the defect correction method. The iteration is stopped as soon as the difference between two succeeding solutions is less than the required accuracy.

max_iter = m defines the maximum number of iterations to be performed in the defect correction process.

print_level = p defines the amount of output to be produced by the defect correction method.

solve_accuracy = ε defines the accuracy of the linear solver in each step of the defect correction method.

SPECTRAL is used in case of spectral elements. It is assumed that the system of equations is solved by a CG type method with FEM preconditioner. The standard keywords in the block SOLVE refer to the FEM preconditioner.

The keyword SPECTRAL itself refers to the CG type method for the spectral solver.

The following options are available:

```
accuracy = eps
max_iter = m
print_level = p
```

with

accuracy = ε defines the accuracy of the spectral method. The iteration is stopped as soon as the residual is less than the required accuracy.

max_iter = m defines the maximum number of iterations to be performed in the defect correction process.

print_level = p defines the amount of output to be produced by the spectral CG solver. Values of p may vary from 0 (no output) to 2 (output in each iteration).

PROJECTION_METHOD = p implies that a projection method will be used to accelerate the convergence of the Krylov subspace method. There are the following reasons to apply a projection method:

- it can be used in combination with a traditional preconditioner to give a faster convergence,
- problems with large contrasts can only be solved if a projection method is used,
- in parallel computing, a projection method combined with a block preconditioner leads to a scalable parallel method.

For an example of the use of the projection method see the manual Standard Problems Section 3.1.6.

At this moment the method has only been implemented for symmetrical positive definite matrices that are solved by CG.

The following values for p are available:

```
none
deflation
coarse
```

These parameters have the following meaning:

none The projection method is not applied.

deflation For this choice a deflation method is used to project a number of "bad" eigenvalues to zero. The resulting singular problems has a better effective condition number.

coarse This choice leads to a coarse grid correction of the preconditioners. The "bad" eigenvalues are now shifted. In general deflation leads to a faster convergence than the coarse grid correction.

The default value is `none`

The following options may be used to define the projection vectors:

```
projection_vectors = v
```

The following values for v are available:

```
approximate_eigenvectors
algebraic_restricted
algebraic
```

These parameters have the following meaning:

approximate_eigenvectors Approximate eigenvectors corresponding to the smallest eigenvalues are used to define the projection method. These approximate eigenvectors are computed as defined by the options. Use of this choice is only possible if integer properties are defined in the input for the mesh generator under the parts MESH SURF or MESH VOLUME (See Section 2.2). The first integer property is used, and it may have either the value 1 or 0.

Integer property 1 means that the coefficient in that region is assumed to be relatively large, whereas integer property 0 means that a small coefficient is valid in this region. This property is used to define the approximate eigenvectors.

algebraic_restricted This choice is also only possible if integer properties are defined (see above). The number of projection vectors is equal to the number of small eigenvalues. The projection vectors are chosen such that they have the value 1 in a large coefficient domain including the boundary and the value 0 in all other domains.

algebraic This is the most general choice. The number of projection vectors is equal to the number of domains. The projection vectors have the value 1 in one domain and the value 0 in all other domains. This is the only choice, which can be combined with parallel computing.

The default value is `algebraic`

The following sub-options may be used to define the approximate eigenvectors:

```
projection_accuracy = e
projection_ignore = t
projection_keep = k
projection_print_level = i
```

These options have the following meaning:

projection_accuracy = ε The projection vectors are computed by solving the equations for each region with the small coefficient separately with suitable values on the boundaries of these regions. The accuracy to which these vectors are computed are defined by ε . The default value is: 10^{-2} .

projection_ignore = t defines a threshold value with respect to the projection vector. All elements in the projection vectors in absolute value smaller than t are set equal to zero, thus saving memory and computation time. The default value is: 0

projection_keep = k defines if the projection vectors must be kept, that previously computed projection vectors must be used or that they are computed and destroyed afterwards.

The following values for k are available:

```
destroy
keep
old
```

destroy means that the eigenvectors are computed and destroyed afterwards.

keep means that the eigenvectors are computed and stored so that they can be used in a new call to the linear solver.

old means that previously computed eigenvectors are reused.

The default value is: **destroy**.

projection_print_level = i defines the amount of extra output that is produced while computing the projection eigenvectors.

i must have a value between -1 and 2. The larger the value of i the more extra output is produced.

-1 means no extra output at all.

The default value is: 0

OVER_PRESSESURES means that a very special boundary condition is applied that is especially meant for computing over-pressures.

This boundary condition may only be combined with a CG type of iterative solver.

Suppose we have a sandstone layer with a shale layer on top of it and above it again a sandstone layer. If the over-pressure in the lowest sand-stone layer exceeds a threshold value, fluid leaks through the shale layer into the sandstone layer above, yielding a larger pressure in that layer. To simulate that process the pressure in the lower sandstone layer is fixed to the threshold value and the rest of the fluid is added to the right-hand side of the upper sandstone layer as a source term.

This process is carried out in two steps:

1. The system of linear equations is solved without the special boundary condition
2. If the pressure exceeds the threshold value the matrix and right-hand side are adapted and the system of equations is solved again.

In case of more layers this process may be repeated.

The following options are available for this boundary condition (all in one record):

```
user_points = ( i1, i2, j1, j2 , ... )
nodes = ( i1, i2, j1, j2 , ... )
thresholds = ( t1, t2, ... )
```

USER_POINTS defines pairs of user points for each layer where we have a break through.

The first user point number defines the user point in the lowest sandstone layer the second one in the sandstone layer above this layer. Hence the number of points must always be even.

NODES has exactly the same meaning as **user_points**, however, now the node numbers must be given instead of the user point numbers.

The options **user_points** and **nodes** are mutually exclusive.

THRESHOLDS defines the threshold values in the lowest of the two points of a pair. The number of values must be exactly equal to the number of pair defined by **user_points** or **nodes**.

ISEQ_EXACT = i If this item is given the exact solution must be stored by the user in the i^{th} solution vector. This vector may be used to compute the error or to compute the residual of the exact solution with respect to the system of equations.

ISEQ_RES_EXACT = i can only be used in combination with **ISEQ_EXACT** = i . If used the residual of the exact solution with respect to the system of equations is stored in the i^{th} solution vector.

This option is meant for debugging purposes to find in which points the error of the discretization is large.

ISEQ_START_RES = i If used the residual of the starting solution with respect to the system of equations is stored in the i^{th} solution vector.

ISEQ_END_RES = i If used the residual of the final solution with respect to the system of equations is stored in the i^{th} solution vector.

This residual must be small, otherwise the system of equations has a very bad condition, or an error in the solution process has occurred.

LIMIT_SOLUTION This keyword is a special one that should be used only in exceptional cases.

It limits the solution between the minimum and maximum values at the start of the solver. Usually this means that the solution will be restricted to the minimum and maximum values of the boundary conditions. If these conditions are always positive and the vector in the inner region has not been set, then 0 will be the minimum since the vector is initialized to 0, before boundary conditions are set.

The limiting is very crude: all values above the maximum value are reset to the maximum value and all all values below the minimum value are reset to the minimum value.

How to influence the overrelaxation process

The overrelaxation process to solve $\mathbf{Su} = \mathbf{f}$ is defined as:

$$u_i^{k+\frac{1}{2}} = \frac{f_i - \sum_{j=1}^{i-1} s_{ij}u_j^{k+1} - \sum_{j=i+1}^N s_{ij}u_j^k}{s_{ii}} \quad (3.2.8.4)$$

$$u_i^{k+1} = x_i^k = \omega(u_i^{k+\frac{1}{2}} - u_i^k) \quad 0 < \omega < 2 \quad (3.2.8.5)$$

- a** Estimate ω_0 (Default $\omega_0 = 1$).

The starting value of ω_0 is equal to the parameter OMEGA. The default value is 1, however, once this parameter is changed in SEPCOMP the new value is kept.

- b** $n1$ iterations with $\omega = \omega_0$ are carried out.

When $n1 = 0$, $\omega_1 = \omega_0$, go to e.

When $n1 = -1$, $n1$ is made equal to n^{ndim} , with n the size of the matrix (i.e. number of unknowns), and $ndim$ the dimension of the space.

$n1$ is equal to the parameter NITER1 at the start. The default value is -1, however, once this parameter is changed in SEPCOMP the new value is kept.

- c** λ_{max} is computed using the power method and ω_{opt} estimated:

$$\omega_{opt} = \frac{2}{1+\sqrt{1-\mu}}, \quad \mu = \frac{1}{\lambda_{max}} \frac{\lambda_{max}-1+\omega^2}{\omega} \quad (*).$$

The parameter LAMBDA gets the value λ_{max} .

- d** A new value of ω is computed by

$$\omega_1 = 1 + \alpha(\omega_{opt} - 1) \quad 0 \leq \alpha \leq 1$$

The starting value of α is equal to the parameter ALPHA. The default value is $\frac{2}{3}$, however, once this parameter is changed in SEPCOMP the new value is kept.

- e** $n2$ iterations with $\omega = \omega_1$ are carried out.

When $n1 = 0$, $\omega_2 = \omega_1$, go to h.

When $n2 = -1$, $n2$ is made equal to $n1$. $n2$ is equal to the parameter NITER2 at the start. The default value is -1, however, once this parameter is changed in SEPCOMP the new value is kept.

- f** λ_{max} is computed using the power method, and ω_{opt} estimated with formula (*).

- g** A new value of ω is computed by

$$\omega_2 = 2 + \beta(\omega_{opt} - 2) \quad 0 \leq \beta \leq 1$$

The starting value of β is equal to the parameter BETA. The default value is $\frac{4}{5}$, however, once this parameter is changed in SEPCOMP the new value is kept.

When $n1=0$ and $n2 = 0$, λ_{max} is estimated in each step.

When $n1=-2$, λ_{max} is not estimated. The value of LAMBDA is used.

- h** Until the process is terminated, iterations are performed using the new value of ω .

Remark

The values of α and β may not be the best choice. A good method to get an acceptable estimation of α and β is the following:

Compute the problem on a small scale, that is with a few elements, with various values of α and β (especially β). Compare the values of ω and λ and the number of iterations to make the best choice. Use these values in the original problem with a lot of elements.

3.2.9 The main keyword `NONLINEAR_EQUATIONS`

The block defined by the main keyword `NONLINEAR_EQUATIONS` indicates that a non-linear stationary problem has to be solved. In this block information concerning the iteration process must be defined.

The block defined by the main keyword `NONLINEAR_EQUATIONS` has the following structure (options are indicated between the square brackets "[" and "]"):

```
NONLINEAR_EQUATIONS [SEQUENCE_NUMBER = s] [ PROBLEM = p]
  (optional): opens the input for the non-linear solver.
```

```
  NUMBER_OF_COUPLED_EQUATIONS = n
  GLOBAL_OPTIONS, options (optional)
  EQUATION i (mandatory)
    followed by the subsubkeywords:
      LOCAL_OPTIONS, options (optional)
      FILL_COEFFICIENTS = f (optional)
      CHANGE_COEFFICIENTS (optional)
        followed by the subsubsubkeywords:
          AT_ITERATION j, SEQUENCE_NUMBER = k
```

```
END (mandatory)
```

The sequence number s may be used to distinguish between various input blocks with respect to the nonlinear solvers.

The problem sequence number p may be used to define the problem number corresponding to the matrix and vector to be filled. If omitted the next problem number is assumed.

The sequence of the subkeywords, subsubkeywords and subsubsubkeywords is arbitrary. However, subsubkeywords corresponding to a subkeyword must all be grouped under the subkeyword and so on. All sub, subsub and subsubsub keywords given above must start at a new line in the input file.

Meaning of the subkeywords:

NUMBER_OF_COUPLED_EQUATIONS defines the number of equations that must be solved simultaneously. If omitted n is equal to `NPROB`, i.e. the number of problems defined.

This option is meant for the case of decoupled problems. Hence if the Boussinesq equations are solved in a coupled way, which means temperature, velocity and if necessary pressure in one element type, then the number of coupled equations is one. If, however, alternatively the velocity (including pressure) and temperature are solved, then the equations are decoupled and the number of coupled equations must be two.

GLOBAL_OPTIONS define some global choices with respect to the linear solver. The options itself should be put on the same line as the keyword `GLOBAL_OPTIONS`. If this line exceeds position 80, continuation at the next line is necessary. This is activated by closing the line with `//` (before column 81) and putting the rest of the information on the next line. This process may be indefinitely repeated. The following options are available:

```
maxiter = m                (Default 20)
miniter = m                (Default 2)
accuracy = eps             (Default 1d-3)
print_level = p           (Default 0)
relaxation = omega        (Default 0)
criterion = c             (Default abs)
lin_solver = i            (Default value 1)
```

```

at_error = e           (Default stop)
output = i             (Default 0)
seqtotal_vector = i   (Default 1)
iteration_method = m   (Default standard)
abs_residual_accuracy = eps (Default: not)
rel_residual_accuracy = eps (Default: not)
limit_solution        (Default: not)
linear_subelement     (Default: not)

```

Meaning of the various options:

maxiter = m defines the maximum number of iterations that may be performed. If the number of iterations reaches this maximum value and the accuracy has not been reached, an error message is given and the program is terminated.

miniter = m defines the minimum number of iterations that have to be carried out.

accuracy = ϵ defines the accuracy at which the iteration terminates, provided the minimum number of iterations has been performed. Accuracy has been reached if the difference between two succeeding iterations is less than ϵ .

print_level = p gives the user the opportunity to indicate the amount of output information he wants from the iteration process. p may take the values 0, 1 or 2. The amount of output increases for increasing value of p .

relaxation = ω defines a relaxation parameter for the non-linear iteration process ($0 \leq \omega \leq 2$). If $\omega = 0$ or 1, the solution of the previous iteration (\mathbf{u}^k) is taken as new estimate of the old solution in the iteration process.

If $\omega \neq 0$ or 1, the old solution \mathbf{u}_{old} is defined as:

$$\mathbf{u}_{old} = \omega \mathbf{u}^k + (1 - \omega) \mathbf{u}^{k-1}, \quad \mathbf{u}^{-1} = \mathbf{u}^0.$$

criterion = c defines the type of termination criterion to be used. Possible values are:

```

absolute
relative
rhs_absolute
rhs_relative
energy_absolute
energy_relative
res_absolute
res_relative

```

If **absolute** is used (default value) the process is stopped if $\|\mathbf{u}^{k+1} - \mathbf{u}^k\| \leq \epsilon$.

If **relative** is used the process is stopped if $\frac{\|\mathbf{u}^{k+1} - \mathbf{u}^k\|}{\|\mathbf{u}^{k+1}\|} \leq \epsilon$.

If **rhs_absolute** is used the process is stopped if $\|\mathbf{f}^k\| \leq \epsilon$, where \mathbf{f}^k is the right-hand side in the k^{th} iteration.

If **rhs_relative** is used the process is stopped if $\frac{\|\mathbf{f}^k\|}{\|\mathbf{u}^{k+1}\|} \leq \epsilon$, where \mathbf{f}^k is the right-hand side in the k^{th} iteration.

If **res_absolute** is used the process is stopped if $\|\mathbf{r}^k\| \leq \epsilon$, where \mathbf{r}^k is the residual in the k^{th} iteration.

If **res_relative** is used the process is stopped if $\frac{\|\mathbf{r}^k\|}{\|\mathbf{u}^{k+1}\|} \leq \epsilon$, where \mathbf{r}^k is the residual in the k^{th} iteration.

If **energy_absolute** is used the process is stopped if $\|(\mathbf{f}^k, \delta \mathbf{u})\| \leq \epsilon$, where \mathbf{f}^k is the right-hand side in the k^{th} iteration.

The inner product $(\mathbf{f}, \delta\mathbf{u})$ is known as the energy increment in case of an incremental Newton iteration.

If `energy_relative` is used the process is stopped if $\frac{\|(\mathbf{f}^k, \delta\mathbf{u})\|}{\|\mathbf{u}^{k+1}\|} \leq \epsilon$.

Mark that the criteria `rhs_absolute`, `rhs_relative`, `energy_absolute` and `energy_relative` make only sense in combination with a Newton or incremental Newton iteration.

`lin_solver = i` refers to the sequence number of the input block for the linear solver.

`at_error = e` defines which action should be taken if the iteration process terminates because no convergence could be found. Possible values are:

```
stop
resume
```

If `stop` is used the iteration process is stopped if no convergence is found, otherwise (`resume`) means that control is given back to the main program and the result of the last iteration is used as solution.

`output = i` indicates if during the iteration process the iteration vectors must be written to the file `sepcomp.out` ($i > 0$) or not ($i = 0$). If $i > 0$ the value of i refers to the sequence number of the input block "OUTPUT" which describes which results must be written. This option offers the possibility to follow the iteration process with SEPPOST.

`iteration_method = m` defines the type of non-linear iteration method that is applied. Possible values for m are:

```
standard
newton
incremental_newton
```

standard means that a standard iteration method is applied: The process starts with a given start vector \mathbf{u}^0 containing the boundary conditions. In each iteration $\mathbf{S}^k \mathbf{u}^{k+1} = \mathbf{f}^k$ is solved, where the solution vector \mathbf{u}^{k+1} also contains the given boundary conditions. The matrix \mathbf{S}^k and the right-hand-side vector \mathbf{f}^k may vary in each iteration step.

newton corresponds to the standard Newton (Raphson) method. This process is as follows:

```
start:   given start vector  $\mathbf{u}^0$ 
While not converged
  Solve correction  $\mathbf{S}^k \delta\mathbf{u} = \mathbf{f}^k$ 
  Correct       $\mathbf{u}^{k+1} = \mathbf{u}^k + \delta\mathbf{u}$ 
```

The correction in each step must satisfy homogeneous essential boundary conditions, since otherwise the essential boundary conditions are changed in the correction step.

incremental_newton The incremental Newton method is a variant to the standard Newton process especially meant for non-linear solid mechanics. It is supposed that the initial vector contains the solution of a previous situation. In the new situation the essential boundary conditions may be changed. In this process we start with an initially given so-called total vector and a so-called incremental vector. In the first step of the iteration process the incremental vector must contain the change in essential boundary conditions compared to the initial total vector. In the subsequent steps the essential boundary conditions for the incremental vector are set equal to zero. Hence the process becomes:

```
start:   given start vector  $\mathbf{u}^0$  and an incremental vector  $\delta\mathbf{u}$  containing the change
         in essential boundary conditions
While not converged
  Solve correction  $\mathbf{S}^k \delta\mathbf{u} = \mathbf{f}^k$ 
```

Correct $\mathbf{u}^{k+1} = \mathbf{u}^k + \delta\mathbf{u}$
 Make essential boundary conditions of correction vector 0

seqtotal_vector = i is only used in the case of an incremental Newton method. In that case the vector to be solved is incremental vector which has to be provided with the essential boundary conditions. Besides the solution vector we need the total solution vector containing the actual solution vector \mathbf{u} . This vector is updated in each iteration step. i refers to the sequence number of this total vector. It is necessary that the total solution vector and the incremental vector are different vectors.

abs_residual_accuracy = ε_{res} If this option is found, the residual vector $\mathbf{S}^k\mathbf{u}^k - \mathbf{f}^k$ is computed. Hence the iteration matrix multiplied by the old solution minus the right-hand side. This vector may be considered as some measure for the accuracy. Depending on the print level the norm of the residual is printed. Moreover, in order to reach convergence, not only the difference between two succeeding solutions must be small be also: $\|Residual\| < \varepsilon_{res}$. If you are only interested in the norm of the residual and do not want to use the extra demand on convergence; make ε_{res} large.

rel_residual_accuracy = ε_{res} has the same meaning as **abs_residual_accuracy**, however, in this case the norm is divided by the norm of the computed solution. Mark that **abs_residual_accuracy** and **rel_residual_accuracy** are mutually exclusive.

limit_solution This keyword is a special one that should be used only in exceptional cases. It limits the final solution between the minimum and maximum values at the start of the iterations.

The limiting is very crude: all values above the maximum value are reset to the maximum value and all all values below the minimum value are reset to the minimum value.

The iterations itself are not limited, only the end result. If limiting of the intermediate values is required, **limit_solution** should be used in the linear solver.

linear_subelement ensures that quadratic elements are treated as a cluster of linear elements. For example a 6-node triangle is locally subdivided into 4 3-node triangles. The matrix is built with these linear elements. Of course this option influences the type of approximation and hence the accuracy.

Mark that this option can only be applied if the number of degrees of freedom per point is constant.

EQUATION i If the user wants to define coefficients (which is necessary in the case of the standard elements described in the manual STANDARD PROBLEMS), he has to use the keyword **EQUATION** followed by the sequence number i , for each separate equation, where $1 \leq i \leq n$.

LOCAL_OPTIONS defines the options that are used for equation i only. The following options are available:

```
accuracy = eps
relaxation = omega
criterion = c
lin_solver = i
abs_residual_accuracy = eps
rel_residual_accuracy = eps
seqtotal_vector = i
iteration_method = m
limit_solution
linear_subelement
```

These options have exactly the same meaning as in the case of the global options. The only difference is that they are only applied to this specific equation. As default values, the values defined in **global_options** are used.

FILL_COEFFICIENTS followed by the sequence number f indicates that the coefficients must be filled and that the input is given by the main block defined by the main keyword COEFFICIENTS with sequence number f .

CHANGE_COEFFICIENTS indicates that at certain iteration numbers some of the coefficients must be changed. This may for example be the case if a switch to a new linearization, for example from Picard to Newton must be performed. In the records following CHANGE_COEFFICIENTS it is indicated at which iteration sequence number the coefficients must be changed. j defines the iteration number, the sequence number k refers to the main block CHANGE COEFFICIENTS provided with sequence number k .

3.2.10 The main keyword CREATE

The block defined by the main keyword CREATE is used to create a SEPRAN vector, which may be a solution vector or a vector of special structure. If this block is available it is always read and interpreted. However, the actual creation of the vector takes only place if the option CREATE_VECTOR is used in the input block "STRUCTURE".

The block defined by the main keyword CREATE has the following structure (options are indicated between the square brackets "[" and "]"):

```
CREATE [COMPLEX] VECTOR = [i] [,PROBLEM = p] [,SEQUENCE_NUMBER = s]
    Record defining the type of output vector
    Records defining the output vector
END
```

The various options in the CREATE record have the following meaning:

CREATE mandatory, means that a vector must be created.

COMPLEX indicates that the solution vector is a complex vector.

SEQUENCE_NUMBER = s may be used to distinguish between various input blocks with respect to the vector.

VECTOR = i is used to define the i^{th} vector in a row of vectors. If omitted the first vector in this block has sequence number 1, and all other vectors have the sequence number of the preceding one plus 1. If the block STRUCTURE is used and in this block the command CREATE_VECTOR, sequence_number = s, vector = v is given, then the actual vector sequence number is $v - 1 + i$.

PROBLEM = p is used to define the problem sequence number corresponding to the solution vector to be filled. If omitted $p = i$ is assumed, with i the sequence number of the vector.

The record defining the type of output vector may take one of the following shapes:

```
TYPE = SOLUTION VECTOR
TYPE = VECTOR OF SPECIAL STRUCTURE [Vj]
TYPE = VECTOR defined per element [N_alpha]
TYPE = SPECIAL_VECTOR_PER_NODE_PER_ELEMENT [N_alpha]
TYPE = CAPACITY_VECTOR, NELECTRODES = n [NPHYS]
```

SOLUTION VECTOR means that the output vector has exactly the same structure as the solution vector, which means that renumbering of unknowns may take place.

VECTOR OF SPECIAL STRUCTURE V_j means that the output vector is a vector of special structure as defined in the PROBLEM block. For standard problems the available structures are defined in the manual Standard Problems.
 V_j defines the sequence number of the special structure. If omitted $V_j=1$ is assumed.

VECTOR defined per element N_α means that the output vector is a vector of special structure where the number of parameters is constant per element. Hence this vector contains element related quantities as described in the input block PROBLEM.
 N_α defines the number of parameters defined per element. If omitted $N_\alpha=1$ is assumed.

SPECIAL_VECTOR_PER_NODE_PER_ELEMENT N_α means that the output vector is a vector of special structure where the number of parameters is constant per node per element. So for each node and element a number of quantities is defined. These quantities may have different values for the same node in different elements.

$N\alpha$ defines the number of parameters defined per node. If omitted $N\alpha=1$ is assumed.
 The sequence in which the unknowns are stored is: first all unknowns for the relative first node of element 1, then for relative second node and so on, followed by element 2, 3 ...

CAPACITY_VECTOR, NELECTRODES = n [*nphys*] means that the output vector is a vector of capacities corresponding to electrodes as described in Section 3.2.19. n defines the number of electrodes and *nphys* the number of degrees of freedom per entry in the capacity vector. The default value for *nphys* is 1.

The sequence in which the capacities are stored is: first all unknowns for the combination (1,1) then for (1,2), (1,3), ...

Only one type may be defined. If no type is defined SEPRAN checks if the vector has been filled before. If the vector has been filled before corresponding to the same problem j , the preceding type is used, otherwise the type SOLUTION_VECTOR is assumed.

After the type definition, records defining the vector must be given. The vector is created by applying the definitions sequentially, so for example first a vector may be set to a constant, then the curves may be changed into other values and finally the user points may be changed. The sequence of the commands defines the sequence in which the vector is filled. This sequence may be essential for the final value in a specific node. The records defining the computation have the following shape:

[functional description] [degrees of freedom] [location part] in arbitrary order.

The functional description may be of one of the following shapes:

```
VALUE = alpha
VALUE = (alpha , beta)
FUNC[TION] = k
SPECIAL_FUNC[TION] = l
OLD_FUNC[TION] = m
OLD_VECTOR = n
SEQ_VECTORS = V1, V2, ...
QUADRATIC, MAX = a
HALF_QUADRATIC, MAX = a
VECTOR = V1
```

with

VALUE = α sets the required degrees of freedom equal to the constant value α . If the vector is complex, also a complex value may be given like (α, β) . In that case $(\alpha, 0)$ and α are identical.

FUNCTION = k defines the degrees of freedom as a function of the co-ordinates. In the case of a real vector the function is defined by the function subroutine FUNC:

```
function FUNC ( k, X, Y, Z )
```

see INTRODUCTION 5.5.4.

In the case of a complex vector the function is defined by the function subroutine CFUNC:

```
function CFUNC ( k, X, Y, Z )
```

see INTRODUCTION 5.5.4.

SPECIAL FUNCTION = l also defines the degrees of freedom as a function of the co-ordinates, however, in this case more complicate subroutines FUNC1B (real case) and CFUN1B (complex case) must be used:

```
subroutine FUNC1B(1,INDEX1,INDEX2,USOL,COOR)
```

or

```
subroutine CFUN1B(1,INDEX1,INDEX2,USOL,COOR)
```

See Section 3.3.4.

OLD_FUNCTION = m defines the degrees of freedom as a function of the co-ordinates and the previous solution which must be stored in the array to be created. In fact this option is meant to update an existing vector as function of this vector.

It is assumed that old vector and the new vector to be created take the same positions. Hence the vector to be created must have been filled before and is changed as function of the old values.

Such a possibility is especially meant for non-linear boundary conditions. In the case of a real vector the function is defined by the function subroutine FUNCOL:

```
function FUNCOL ( m, X, Y, Z, UOLD )
```

see Section 3.3.5.

In the case of a complex vector the function is defined by the function subroutine CFUNOL:

```
function CFUNOL ( m, X, Y, Z, UOLD )
```

see Section 3.3.5.

VECTOR = **V1** copies the degrees of freedom from the corresponding degrees of freedom in vector V1.

OLD_VECTOR = m defines the degrees of freedom as a function of the co-ordinates and a number of predefined vectors. The user is supposed to provide the subroutine FUNCVECT as described in Section 3.3.11.

```
subroutine funcvect( m, ndim, coor, numnodes, uold,
+                  nuold, result, nphys)
```

SEQ_VECTORS = **V1**, **V2**, ... makes only sense in combination with **OLD_VECTOR** = m . It defines the sequence numbers of the vectors to be used in subroutine FUNCVECT, as well as the number of vectors.

If omitted vector V1 is used.

QUADRATIC, **MAX** = α defines the degrees of freedom as a quadratic function along a (series of) curve(s), such that the quadratic function is equal to 0 in the end points and has the value α in the mid point. If **MAX** = α is omitted, $\alpha = 1$ is assumed.

This boundary condition, which is meant to prescribe a quadratic velocity profile, for example in a fully developed flow, may only be used for one specific degree of freedom. Furthermore the boundary condition is only allowed along curves.

HALF_QUADRATIC, **MAX** = α defines the degrees of freedom as a quadratic function along a (series of) curve(s), such that the quadratic function is equal to 0 in the begin point and has the value α in the end point. If **MAX** = α is omitted, $\alpha = 1$ is assumed.

This boundary condition, which is meant to prescribe a quadratic velocity profile in case of a symmetry axis, for example in a fully developed axi-symmetric flow, may only be used for one specific degree of freedom. Furthermore the boundary condition is only allowed along curves.

If the functional description is omitted, the default: **VALUE**=0 is assumed.

The degrees of freedom part may have one of the following structures:

```
DEGFD2
DEGFD3, DEGFD1, DEGFD6
```

which indicates that in the nodes to be created only physical unknown 2 or the physical unknowns 1, 2 and 6 are filled.

If this part is omitted all degrees of freedom in the nodes to be created are filled.

The location part may have one of the following structures:

```
POINTS ( Pk, P1, . . . , Pm)
USER POINTS ( Pk to P1 [,step = s] )
CURVES [l] (Cj [to Cm] )
SURFACES [i1, i2, i3, ... ] (Sj [to Sm] )
VOLUMES [i1, i2, i3, ... ] (Vj [to Vm] )
OUTER_CURVES
OUTER_SURFACES
NODES (Nj [to Nm [, step = s ] ] )
ELEMENTS i [to j] [(RN1, RN2, RN3, ... )]
GROUP ielgrp [(RN1, RN2, RN3, ... )]
GLOBAL_GROUP iglgrp
CONTACT i
NO_CONTACT i
OBSTACLE i
IN_ALL_OBSTACLE i
IN_INNER_OBSTACLE i
IN_BOUN_OBSTACLE i
ON_BOUN_OBSTACLE i
ZERO_LEVELSET i
FILE_NODAL_VALUES = 'file_name'
FILE_ELEMENT_VALUES = 'file_name'
FILE_CAPACITY_VALUES = 'file_name'
```

These records have the following meaning:

POINTS (P_{i1}, P_{i2}, \dots) defines all user points between the brackets.

CURVES [l] (C_j [to C_m]) defines only the part of the vector in the curves C_j to C_m (or C_j if C_m is omitted).

l has the following meaning: $l = i + 100 \times \text{IEXCLUDE}$.

If omitted $l = 0$ is assumed.

i has the following meaning:

$i=0$ means that all nodal points on the curves are prescribed as indicated by DEGFDj.

$i>0$ means that only the nodal points 1, $1+(i+1)$, $1+2 \times (i+1)$, ... on these curves are prescribed as indicated by DEGFDj.

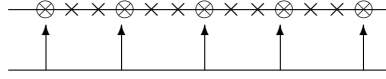
$i<0$ means that all nodal points except the points 1, $1-(i-1)$, $1-2 \times (i-1)$, ... on these curves are prescribed as indicated by DEGFDj.

Hence $i=0$

—XXXXXXXXXXXXXXXXXX—

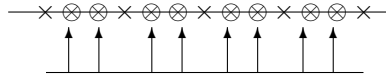
degrees of freedom DEGF D_j are prescribed in all nodal points.

$i=2$



degrees of freedom DEGF D_j are prescribed in the nodal points indicated by x.

$i=-2$



degrees of freedom DEGF D_j are prescribed in the nodal points indicated by x.

Remark: i must be so that the last nodal point of the curves C_1 to C_5 is equal to $1 + k$ i with k integer (> 0 or < 0).

EXCLUDE may have one of the following values:

- 0 All points of the curves C_1 to C_5 are used as indicated by i .
- 1 All points except the begin point of C_1 are used as indicated by i .
- 2 All points except the end point of C_5 (or C_1 if C_5 is omitted) are used as indicated by i .
- 3 All points except the begin point of C_1 and the end point of C_5 (or C_1 if C_5 is omitted) are used as indicated by i .

Remark: **EXCLUDE** > 0 may only be used in combination with $i \geq 0$.
If omitted $l = 0$ is assumed.

The curves C_j to C_m must be subsequent curves!

USER POINTS (P_{l_1} [to P_{l_2} [,step j]]) defines the part of the vector in the user points P_{l_1} to P_{l_2} (or P_{l_1} if P_{l_2} is omitted).

The parameter j after step has the following meaning (default: 0):

$j = 0$ all user points between P_{l_1} and P_{l_2} are used

$j > 0$ the user points $P_{l_1}, P_{l_1+j}, P_{l_1+2j}, \dots$ are used

$j < 0$ all user points between P_{l_1} and P_{l_2} are used except the points $P_{l_1}, P_{l_1-(j-1)}, P_{l_1-2(j-1)}, \dots$

SURFACES [i_1, i_2, i_3, \dots] (S_j [to S_m]) has the same meaning as **CURVES** (C_j [to C_m]) but now with respect to the surfaces S_j to S_m or S_j if S_m is omitted.

The parameters i_1, i_2, \dots, i_p have the following meaning: when omitted, all nodal points in the surface are used. Otherwise, the degrees of freedom are only prescribed in the $i_1^{th}, i_2^{th}, i_3^{th}, \dots$ nodal point of each element of the surfaces.

A typical example is **SURFACES** (1,3,5) for a quadratic triangle, indicating that only the vertices in the triangle are used. See Figure 3.2.10.1. The rest of the record has the same meaning as in the points record.

All surfaces in this statement must have the same number of nodes per element.

VOLUMES [i_1, i_2, i_3, \dots] (V_j [to V_m]) has the same meaning as **SURFACES** [i_1, i_2, i_3, \dots] (S_j [to S_m]) but now with respect to the volumes V_j to V_m or V_j if V_m is omitted.

The parameters i_1, i_2, \dots, i_p have the following meaning: when omitted, all nodal points in the

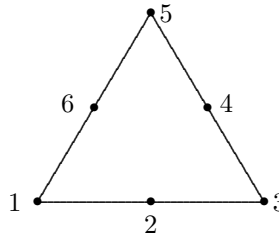


Figure 3.2.10.1: Quadratic triangle, with corresponding nodal point numbering.

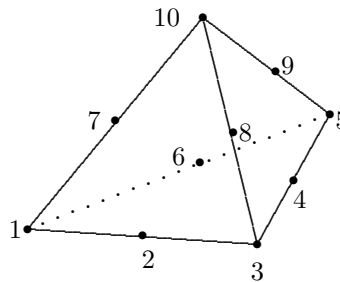


Figure 3.2.10.2: Quadratic tetrahedron, with corresponding nodal point numbering.

volume are used. Otherwise, the degrees of freedom are only prescribed in the $i_1^{th}, i_2^{th}, i_3^{th}, \dots$ nodal point of each element of the volumes.

A typical example is VOLUMES (1,3,5,10) for a quadratic tetrahedron, indicating that only the vertices in the tetrahedron are used. See Figure 3.2.10.2. The rest of the record has the same meaning as in the points record.

All volumes in this statement must have the same number of nodes per element.

OUTER_CURVES All the nodes of the mesh that are on the outer boundary of a 2D mesh are used to define the quantities.

OUTER_SURFACES All the nodes of the mesh that are on the outer boundary of a 3D mesh are used to define the quantities.

NODES (N_1 [to N_2 [,step= j]]) defines the components in the nodal points N_1 to N_2 or N_1 if N_2 is omitted. The absolute nodal point numbers are used, not the user points.

If $j > 1$ only the points $N_1, N_1 + j, N_1 + 2j, \dots, N_2$ are used.

All volumes in this statement must have the same number of nodes per element.

ELEMENTS i [to j] [(RN_1, RN_2, \dots, RN_l)] defines the components in the relative nodal points (RN_1, RN_2, \dots, RN_l) of the elements i to j (or i if j is omitted). The absolute numbers are used.

All elements in this statement must have the same number of nodes per element.

GROUP $ielgrp$ [(RN_1, RN_2, \dots, RN_l)] defines the components in the relative nodal points RN_1, RN_2, \dots, RN_l of all elements with element group number $ielgrp$. When the relative nodal points are omitted, all nodal points in the element are used.

All element groups in this statement must have the same number of nodes per element.

GLOBAL_GROUP $iglgrp$ defines the components in the unknowns corresponding to the global group $iglgrp$.

OBSTACLE i makes only sense in combination with free or moving boundary problems. In that case the user may define obstacles in the region. The free or moving boundary may not cross these obstacles. Once a boundary crosses an obstacle, the corresponding nodes are projected on the obstacle and are made active. Later on they may be deactivated during the computation. Only active nodes along the boundary are used for the location part. Hence this set of nodes may be empty.

IN_ALL_OBSTACLE i Values are prescribed in all nodes of the mesh that are situated within the obstacle with obstacle sequence number i .

IN_INNER_OBSTACLE i Values are prescribed in the nodes of those elements of the mesh that are completely within the obstacle with obstacle sequence number i .
Compared to **IN_ALL_OBSTACLE**, this means that nodes near the boundary of the obstacle are excluded.

IN_BOUN_OBSTACLE i Values are prescribed in the nodes of those elements of the mesh that are partly outside the obstacle with obstacle sequence number i .
So all the points that are excluded in **in_boun_obstacle** but are part of **in_all_obstacle** belong to **in_boun_obstacle**.

ON_BOUN_OBSTACLE i Values are prescribed in the nodes of those elements of the mesh that are on the boundary of the obstacle with obstacle sequence number i .

CONTACT i makes only sense if the user has defined a contact surface and activated a contact algorithm. Only nodes that make contact and are at the contact surface with sequence number i are used for the location part. Hence this set of nodes may be empty.

NO_CONTACT i makes only sense if the user has defined a contact surface and activated a contact algorithm. Only nodes that make no contact and are at the contact surface with sequence number i are used for the location part. Hence this set of nodes may be empty.

ZERO_LEVELSET i Values are prescribed in all the nodes of the mesh where the level set function ϕ_i has the value 0. This is exactly the boundary made by the command **make_levelset_mesh** in the structure block. See (3.2.3.18).

FILE_NODAL_VALUES 'file_name' may be used to read nodal point numbers and corresponding values of the vector to be created (or the boundary conditions) from the file with the name given between the two quotes. If this option is used, the user must provide a file of this name as described in Section 3.5.2.

At the moment the file is used, it is opened with reference number 75, the contents are read and the file is closed again. This means that the user may not have opened a file with reference number 75 at the same time, and moreover that if the file is reused again reading starts from the first record.

This option is only available for solution vectors or vectors of special structure (not defined per element).

FILE_ELEMENT_VALUES 'file_name' may be used to read element numbers and corresponding values of the vector to be created (or the boundary conditions) from the file with the name given between the two quotes. If this option is used, the user must provide a file of this name as described in Section 3.5.3.

At the moment the file is used, it is opened with reference number 75, the contents are read and the file is closed again. This means that the user may not have opened a file with reference number 75 at the same time, and moreover that if the file is reused again reading starts from the first record.

This option is only available for vectors of special structure defined per element.

The number of values to be read per element (**NDEGFD**) is equal to the number of quantities stored per element in this type of vector.

FILE_CAPACITY_VALUES 'file_name' may be used to read electrode pairs and corresponding values of the capacity vector to be created from the file with the name given between the two quotes. If this option is used, the user must provide a file of this name as described in Section 3.5.4.

At the moment the file is used, it is opened with reference number 75, the contents are read and the file is closed again. This means that the user may not have opened a file with reference number 75 at the same time, and moreover that if the file is reused again reading starts from the first record.

This option is only available for vectors of the type capacity vector.

If this part is omitted, all nodes are used.

Typical examples are:

```

VALUE=3
FUNC=6
VALUE=(2,0.5)
SPECIAL FUNCTION=k

DEGFD1, VALUE=5
FUNC=6, DEGFD1, DEGFD3

POINTS (P1, P2, P6), DEGFD2, FUNC=5
POINTS (P1), FUNC=3, DEGFD1
USER POINTS (P3 to P6)
DEGFD3, USER POINTS (P3 to P7, step=2), VALUE=0.5
FUNC=2, DEGFD2, DEGFD6, CURVES 3 (C1 to C3)
SURFACES 1,3,5 (S1 to S5), VALUE=7d6
VOLUMES (V2 to V3), VALUE=(0,27.345)
NODES (1 to 16, step=2), VALUE=(2,5), DEGFD1
ELEMENTS (3 to 253, step=7), DEGFD2, DEGFD3, FUNC=7
GROUP 7 (2,4,6)

```

Remarks:

- Each vector in the input block CREATE may be defined by at most one command CREATE VECTOR so the parameter i in this COMMAND must be unique per sequence number s .
- If no data records (except perhaps the type record) are given after the CREATE command, the complete vector is set equal to zero.
However, as soon as at least one data record defining the vector or a part of it is given, the vector is not initialized. That means that degrees of freedom that are not defined in the data records are not changed or initialized. The user is responsible for the correct filling of the vector.
- The vector is filled in the order given in the input file. Hence, the statements

```

VALUE=0
DEGFD2 = (FUNC=3)

```

set first the vector equal to zero and then replace the second component by the function defined by FUNC=3.

On the other hand the statements

```

DEGFD2=(FUNC=3)

```

VALUE=0

have as final effect that the vector is set equal to zero. In this case the first command is useless and only consumes computing time.

A typical input block "CREATE" might be:

```
CREATE VECTOR 3, problem 1, sequence_number = 2
  type = vector of special structure 2
  value = 0
  degfd1 = func = 3
CREATE VECTOR 1, problem 2
  surface (s1 to s3), degfd2 = func = 4
CREATE COMPLEX VECTOR 2
  type = solution vector
  value = (0,0)
  degfd2 = func = 6
  curves (c1 to c5), degfd1 = value = (3,.5)
  curves (c2 to c3), degfd3 = func = 7
END
```

3.2.11 The main keyword DERIVATIVES

The block defined by the main keyword DERIVATIVES gives information with respect to the derived quantities (usually derivatives) to be computed. If this block is available it is always read and interpreted. However, the actual computation of derivatives takes only place if the option DERIVATIVES is used in the input block "STRUCTURE".

The block defined by the main keyword DERIVATIVES has the following structure (options are indicated between the square brackets "[" and "]"):

```
DERIVATIVES [,SEQUENCE_NUMBER = s] [,PROBLEM = p]
  data records
END
```

The first record opens the input for the computation of derived quantities.

The sequence number s may be used to distinguish between various input blocks with respect to the derivatives. The problem sequence number p is used to define the problem number corresponding to the vector to be filled. If omitted $p = 1$ is assumed.

The last record closes the input Data records defining the type of output vector have the following shape

```
TYPE_OUTPUT = t, options (at the same line)
ICHELD = s
ELEMENT_GROUPS = (s1, s2, ...)
SKIP_ELEMENT_GROUPS = (s1, s2, ...)
IX = s
DEGREE_OF_FREEDOM = d
SEQ_INPUT_VECTOR [k] = i
NUMVEC = n
CLEAR
NO_CLEAR
BOUNDARY_INTEGRAL (C1, C4, ...)
CURVES (C1, C4, ...)
POINTS (P3, P1, ...)
SURFACES (S2, S8, ...)
ZERO_LEVEL_SET i
```

These data records have the following meaning:

TYPE_OUTPUT = t ,options

defines the type of output vector. This option must be used if user elements with type numbers between 1 and 99 are used. The SEPRAN standard elements detect itself the type of output vector. In case of a mixture of user elements and SEPRAN elements, the output is defined by the standard elements.

The following values of t are allowed:

```
SOLUTION
SPECIAL
ELEMENTWISE
INTERPOLATE
ELEMENT_NODES
ELEMENT_INTEGRATION_POINTS
REACTION_FORCE
```

and with respect to options we have the following choices

COMPLEX
REAL
IVEC = s

Meaning of all these values:

SOLUTION means that the vector to be created is of the type of a solution vector.

SPECIAL the vector is a vector of special structure with sequence number *iv* defined per node. See the input block "PROBLEM" how to define these vectors. The default value for *iv* is 1.

ELEMENTWISE the vector is a vector of special structure defined per element with *iv* degrees of freedom per element.

INTERPOLATE the vector to be created is a vector of special structure with sequence number *iv*. However, in contrast to the option **SPECIAL** the vector is not created by an averaging process, but merely by adding quantities in common nodes. In other words the vector of special structure is built as if it is a right-hand-side vector.

ELEMENT_NODES The output vector is a vector of special structure defined per element. Each quantity is stored for each node per element No averaging takes place.

ELEMENT_INTEGRATION_POINTS The output vector is a vector of special structure defined per element. Each quantity is stored for each integration point per element

REACTION_FORCE is a very special option. It creates a reaction force along a set of given curves. (surfaces have not yet been implemented). In fact it defines the flux through these curves, provided there are no other elements on the other side of the curves. This is for example the case if the curve is a curve on the outer boundary. But also if we exclude element groups the result may be that there are no elements left on the other side of the curves. Mark that this option is an extension of the possibilities provided by the linear and non-linear solver.

In order that the correct flux is computed, it is necessary that the coefficients in the input are exactly the coefficients used to solve the problem. So in a non-linear problem these must be the coefficients last used (might be defined by **change_coefficients**). The options on this input line are not used.

For an example see Section [6.2.5](#)

COMPLEX defines the output vector as a complex vector and

REAL defines the output vector as a real vector.

Default value: **SOLUTION**

ICHELD = s defines the type of derived quantity to be computed. This parameter is passed to the element subroutines which decide which derivative corresponds to the value *s* of **ICHELD**. With respect to the standard elements provided by SEPRAN, it is necessary to consult the manual Standard problems for the meaning of **ICHELD** in specific cases. If user elements are defined (type numbers between 1 and 99), the parameter **ICHELD** is passed undisturbed to the element subroutine.

The default value for **ICHELD** is 1.

ELEMENT_GROUPS = (s1, s2, ...) defines which element groups must be used to compute the derivatives. Only element groups defined in the mesh generation part are used.

The default value for **ELEMENT_GROUPS** is all element groups.

SKIP_ELEMENT_GROUPS = (s1, s2, ...) defines which element groups must be skipped when derivatives are computed. The keyword **ELEMENT_GROUPS** excludes the use of the keyword **SKIP_ELEMENT_GROUPS**.

The default value for **SKIP_ELEMENT_GROUPS** is skip no element groups.

IX = ix defines the parameter IX which is used by the element subroutines. Usually IX is meant to indicate which derivative must be computed. For example if $ix=1$, the x-derivative is computed and if $ix=2$, the y-derivative is computed. Whether this parameter is used by the standard elements can be found in the manual Standard Problems.

If user elements are defined (type numbers between 1 and 99), the parameter IX is passed undisturbed to the element subroutine.

The default value for IX is 1.

DEGREE_OF_FREEDOM = d defines the parameter JDEGFD which is used by the element subroutines. Usually JDEGFD is meant to indicate which degree of freedom per node must be used to compute the derivative. So in general the derivative $\frac{\partial u_{JDEGFD}}{\partial IX}$ is computed. Whether this parameter is used by the standard elements can be found in the manual Standard Problems.

If user elements are defined (type numbers between 1 and 99), the parameter JDEGFD is passed undisturbed to the element subroutine.

The default value for JDEGFD is 1.

SEQ_INPUT_VECTOR [$k = i$] defines the parameter from which input vectors the derivatives must be computed. If only one input vector is required explicitly k may be skipped. If the element subroutine requires two separate input vectors, the parameter k must be used to distinguish between the first and second one. The default value for k is 1.

The parameter i refers to the parameter i in vector V_i .

Hence if the pressure in the Navier-Stokes equations must be computed, with the velocity vector stored as V_3 as input, then **SEQ_INPUT_VECTOR** = 3 must be used.

The default value for i is 1.

NUMVEC = n defines the number of subsequent vectors to be used as input vector. For example if the input vector is defined as input vector V_3 and $n=5$, it is supposed that the vectors V_3 to V_7 may be used as input vectors for the computation of derivatives.

The default value is $n = 1$.

CLEAR indicates that the vector to be created is set equal to zero before building the vector.

This is also the default value.

NO_CLEAR indicates that the vector to be created is not set equal to zero before building the vector. This means that this vector must have been created before and that the result of the computation is added to this vector. The default value is CLEAR.

BOUNDARY_INTEGRAL (**C1**, **C4**, ...) indicates that a boundary integral must be computed along the curves C_1 , C_4 , ... This boundary integral must be of the type $\int \mathbf{u} \cdot \mathbf{n} d\Gamma$, with \mathbf{u} a vector quantity.

It is of course necessary that \mathbf{u} has been defined by the other options. So \mathbf{u} must be a derived quantity defined in the same input block.

CURVES (**C1**, **C4**, ...) Is only used in case reaction forces must be computed. It defines along which curves these reaction forces must be computed.

POINTS (**P3**, **P1**, ...) Same as curves but now with respect to user points.

SURFACES (**S2**, **S8**, ...) Same as curves but now with respect to surfaces.

ZERO_LEVEL_SET i Same as curves but now with respect to the zero levelset with sequence number i .

3.2.12 The main keyword INTEGRALS

The block defined by the main keyword INTEGRALS gives information with respect to the integrals to be computed. If this block is available it is always read and interpreted. However, the actual computation of integrals takes only place if the option INTEGRALS is used in the input block "STRUCTURE".

The block defined by the main keyword INTEGRALS has the following structure (options are indicated between the square brackets "[and]"):

INTEGRALS [,SEQUENCE_NUMBER = *s*] (optional)

COMMAND record: opens the input for the computation of integrals.

The sequence number *s* may be used to distinguish between various input blocks with respect to the integrals.

This COMMAND record must be followed by data records defining the type of output vector:

ICHELI = *s*

ELEMENT_GROUPS = (*s*₁, *s*₂, ...)

SKIP_ELEMENT_GROUPS = (*s*₁, *s*₂, ...)

DEGREE_OF_FREEDOM = *d*

MAXMIN_ELEMENTS = choice

END (mandatory)

The sequence of the subkeywords is arbitrary.

The subkeyword ICHELI defines the type of integral to be computed. This parameter is passed to the element subroutines which decide which integral corresponds to the value *s* of ICHELI. With respect to the standard elements provided by SEPRAN, it is necessary to consult the manual Standard problems for the meaning of ICHELI in specific cases. If user elements are defined (type numbers between 1 and 99), the parameter ICHELI is passed undisturbed to the element subroutine.

The default value for ICHELI is 1.

The subkeyword ELEMENT_GROUPS = (*s*₁, *s*₂, ...) defines which element groups must be used to compute the integrals. Only element groups defined in the mesh generation part are used.

The default value for ELEMENT_GROUPS is all element groups.

The subkeyword SKIP_ELEMENT_GROUPS = (*s*₁, *s*₂, ...) defines which element groups must be skipped when integrals are computed. The keyword ELEMENT_GROUPS excludes the use of the keyword SKIP_ELEMENT_GROUPS.

The default value for SKIP_ELEMENT_GROUPS is skip no element groups.

The subkeyword DEGREE_OF_FREEDOM = *d* defines the parameter JDEGFD which is used by the element subroutines. Usually JDEGFD is meant to indicate which degree of freedom per node must be used to compute the integral. So in general the integral over u_{JDEGFD} is computed. Whether this parameter is used by the standard elements can be found in the manual Standard Problems.

If user elements are defined (type numbers between 1 and 99), the parameter JDEGFD is passed undisturbed to the element subroutine.

The default value for JDEGFD is 1.

The subkeyword MAXMIN_ELEMENTS = choice defines whether only the integral must be com-

puted or also the minimum and the maximum value over the elements. The following values for choice are allowed:

NO

YES

ABS

The option NO means that only the integral is computed and YES means that both the integral and the minima and maxima over the elements are computed. If ABS is used the minima and maxima of the absolute values of the element integrals are computed.

3.2.13 The main keyword OUTPUT

The block defined by the main keyword OUTPUT gives information with respect to the vectors to be written to the file sepcomp.out. These vectors may be visualized by program SEPPOST. An extra possibility offered by this part of the program is that derived quantities may be computed and written to the file sepcomp.out. In contrast to the part of the program corresponding to the input block "DERIVATIVES", these derived quantities are not stored in the program itself.

The block defined by the main keyword OUTPUT has the following structure

```
OUTPUT, sequence_number = i
  NOT_SEPPOST
  TO_AVS
  TO_OPEN_DX
  TO_TECPLOT
  TO_PARAVIEW
  WRITE j SOLUTIONS
  AVS_FILE_NAME = 'file_name'
  OPENDX_FILE_NAME = 'file_name'
  TECPLOT_FILE_NAME = 'file name'
  PARAVIEW_FILE_NAME = 'file name'
  FILE_PER_TIME_STEP
  ALL_TIME_STEPS_IN_ONE_FILE
  SEPARATE_ELGRPS
  APPEND
  Vi = ICHELD=k1, options
  Vj = ICHELD=k2, options
  .
  .
END
```

Meaning of these keywords:

OUTPUT ,SEQUENCE_NUMBER = *s* COMMAND record: opens the input for the definition of the quantities to be written to the file sepcomp.out.

The sequence number *s* may be used to distinguish between various input blocks with respect to the output. If omitted the next number is used, compared to the last one read in the input file.

NOT_SEPPOST indicates that the output vector is not written to the file sepcomp.out. This option is only useful if the output vector is written to another file for post-processing purposes. At this moment only one alternative for SEPPOST is available: AVS.

If this keyword is omitted the output is always written to sepcomp.out.

TO_AVS indicates that the output vector must be written to a file with AVS format. This file may be read by AVS for post-processing purposes. Use of this option does not suppress the writing of the output to sepcomp.out.

If this keyword is omitted no output is written to AVS files.

AVS_FILE_NAME defines the name of the AVS file. This name is followed by the suffix _xxx.yyy, where xxx and yyy are sequence numbers related to time and iteration number respectively.

If omitted the default name **sepavs** is used.

TO_OPEN_DX indicates that the output vector must be written to a file with OPEN_DX format. This file may be read by OPEN_DX for post-processing purposes. Use of this option does not suppress the writing of the output to sepcomp.out.

If this keyword is omitted no output is written to OPEN_DX files.

OPENDX_FILE_NAME defines the name of the OPEN_DX file. This name is followed by the suffix `_xxx.yyy`, where `xxx` and `yyy` are sequence numbers related to time and iteration number respectively.

If omitted the default name `sepdx` is used.

TO_TECPLOT indicates that the output vector must be written to a file with TECPLOT format. This file may be read by TECPLOT for post-processing purposes. Use of this option does not suppress the writing of the output to `sepcomp.out`.

If this keyword is omitted no output is written to TECPLOT files.

TECPLOT_FILE_NAME defines the name of the TECPLOT file. This name is followed by the suffix `_xxx.yyy`, where `xxx` and `yyy` are sequence numbers related to time and iteration number respectively.

If omitted the default name `septec` is used.

TO_PARAVIEW indicates that the output vector must be written to a file with AVS(PARAVIEW) format. This file may be read by PARAVIEW for post-processing purposes. Use of this option does not suppress the writing of the output to `sepcomp.out`.

If this keyword is omitted no output is written to PARAVIEW files.

This option is only available for 3d problems.

PARAVIEW_FILE_NAME defines the name of the PARAVIEW file. This name is followed by the suffix `_yyy.inp`, where `yyy` is a sequence number related to time or iteration number.

If omitted the default name `sepparaview_xxx.inp` is used.

FILE_PER_TIME_STEP means that for each time step a separate file is made. This is the default value for all output possibilities except for the file `sepcomp.out`, where this option does not have any effect at all.

ALL_TIME_STEPS_IN_ONE_FILE means that output for all time steps is stored in one large file. This is standard for the file `sepcomp.out`. In the case of a TECPLOT file this is an option, in all other cases it has not been implemented yet.

WRITE j SOLUTIONS indicates that not only the output vector V_1 (or V_i if i is given explicitly in the command OUTPUT in the input block "STRUCTURE"), but also the vectors V_2 , V_3 , ... , V_{1+j-1} (respectively V_i , V_{i+1} , ... , V_{i+j-1}) must be written to the output file. Of course these vectors must have been filled before.

The vectors to be written get the new sequence numbers V_0 , V_1 , ... , V_{j-1} , which must be used by program SEPPOST. These numbers are only meant for SEPPOST not for the use in SEPCOMP.

SEPARATE_ELGRPS is only used in combination with TO_PARAVIEW. In this case the output for each element group is written to separate files `name_of_file_xxx.inp`, where `xxx` is the element group sequence number. For each element group a separate mesh is made consisting of this element group only and the solutions are written for this specific mesh.

APPEND is only used in combination with TO_TECPLOT or TO_AVS. If this keyword is present it is checked if there are already tecplot or avs files with the name given. If so the old files are not destroyed but numbering continues from the last sequence number used.

$V_i = \text{ICHELD} = k1$ (**options**) may be used repeatedly for various values of i . i must be larger than 0 and if the option write j solutions is used, i must be larger than $j - 1$. This option defines a new vector V_i for program SEPPOST that is constructed as derived quantity in the same way as may be done by the input block "DERIVATIVES".

ICHELD = $k1$ defines the type of derived quantity to be computed. See the input block "DERIVATIVES" for an explanation.

This option is not available for OPEN_DX.

The following options are recognized:

```
IX      = k2
JDEGFD = k3
INPVC0 = k4
INPVC1 = k5
IVEC   = k6
SEQ_COEFFICIENTS = s
```

These options have the following meaning:

IX = k2 defines the parameter IX, see the input block "DERIVATIVES".

JDEGFD = k3 defines the parameter JDEGFD, see the input block "DERIVATIVES".

INPVC0 = k4 defines the first input vector to be used, and the option **INPVC1 = k5** the second one, see the input block "DERIVATIVES".

IVEC = k6 may be used to identify an array of special structure with sequence number k6. This option is only used if element type numbers in the user range 1 to 99 are used. Otherwise SEPRAN decides itself what type of output vector must be created.// The default value is k6=0, which means that the output vector has the structure of a solution vector.

SEQ_COEFFICIENTS = s indicates which input block for the coefficients must be used to compute the derivatives.

If omitted, the last coefficients read are used. In practical applications it is always save to give this number explicitly.

In the case of time-dependent problems it is always necessary to give this parameter, since SEPRAN does not keep the last coefficient in the time-dependent subroutines.

END (mandatory), closes the output.

The sequence of the subkeywords is arbitrary.

3.2.14 The main keyword BOUNDARY_INTEGRAL

The block defined by the main keyword BOUNDARY_INTEGRAL gives information with respect to the boundary integrals to be computed. If this block is available it is always read and interpreted. However, the actual computation of integrals takes only place if the option BOUNDARY_INTEGRAL is used in the input block "STRUCTURE".

The block defined by the main keyword BOUNDARY_INTEGRAL has the following structure (options are indicated between the square brackets "[" and "]"):

BOUNDARY_INTEGRAL [,SEQUENCE_NUMBER = *s*] (optional) COMMAND record: opens the input for the computation of boundary integrals.

The sequence number *s* may be used to distinguish between various input blocks with respect to the boundary integrals.

This COMMAND record must be followed by data records defining the type of output vector:

ICHINT = *i*

ICHFUN = *j*

IRULE = *k*

CURVES (C1, C2, C3, ...)

SURFACES (S1, S2, S3, ...)

DEGREE_OF_FREEDOM = *d*

END (mandatory)

The sequence of the subkeywords is arbitrary.

The subkeyword ICHINT defines the type of boundary integral to be computed. The following values for ICHINT are available:

1. $\int_{\partial\Omega} f u ds$, where *u* denotes the solution defined by the input vector (VECTOR *i* in the input block STRUCTURE) and *f* a function defined by ICHFUN.
2. $\int_{\partial\Omega} f \mathbf{u} \cdot \mathbf{n} ds$, where \mathbf{u} denotes the solution defined by the input vector (VECTOR *i* in the input block STRUCTURE) and \mathbf{n} the normal defined at the boundary. It is supposed that the solution can be considered as a vector, which means that there are at least NDIM (dimension of space) degrees of freedom per point to be integrated.
3. $\int_{\partial\Omega} f \mathbf{u} \cdot \mathbf{t} ds$, where \mathbf{t} defines the tangential vector.
4. $\int_{\partial\Omega} f \mathbf{n} \cdot \boldsymbol{\sigma} \cdot \mathbf{n} ds$, where \mathbf{n} denotes the normal and $\boldsymbol{\sigma}$ defines a tensor given by the input vector.
5. $\int_{\partial\Omega} f \mathbf{n} \cdot \boldsymbol{\sigma} \cdot \mathbf{t} ds$, where \mathbf{n} denotes the normal, \mathbf{t} the tangential vector and $\boldsymbol{\sigma}$ defines a tensor.
6. $\int_{\partial\Omega} f \mathbf{n} \cdot \boldsymbol{\sigma} ds$, where \mathbf{n} denotes the normal and $\boldsymbol{\sigma}$ defines a tensor. The result of this operation is a vector instead of a scalar.
7. $\int_{\partial\Omega} f u \mathbf{n} ds$, where \mathbf{n} denotes the normal and *u* defines a scalar. The result of this operation is a vector instead of a scalar.
8. sum over the points at the curves of the function. The result is a scalar. This option is meant in the combination with reaction forces, which are already integrals. The sum of the reaction forces along a boundary defines exactly the boundary integral.

9. $\int_{\partial\Omega} f ds$, where f is a function defined by ICHFUN. So the difference with ICHINT=1 is, that no solution vector is integrated. For example if $f = 1$, the computed value is equal to the area of the boundary.

The default value for ICHINT is 1.

At this moment the options 3 to 7 are only implemented for two-dimensional regions, the options 1 and 2 are available both for R^2 and R^3 .

In the two-dimensional case the normal is defined as the outward normal if the curve is defined such that it is a part of a counterclockwise boundary of the region, otherwise it is the inward normal. The tangential vector is defined in the direction of the curve.

The subkeyword ICHFUN defines how the function f must be computed. The following values for ICHFUN are permitted:

0 The function f is identical to 1.

>0 The function f must be computed by a function subroutine FUNC or CFUNC as described in the SEPRAN introduction Section 5.5.4. If the solution vector is complex CFUNC is used otherwise FUNC should be used. The value of ICHFUN is used as parameter ICHOIS in the input of the function subroutines.

At this moment ICHFUN>0 is only permitted in combination with ICHINT=1.

The default value for ICHFUN = 1.

The subkeyword IRULE defines the type of numerical integration rule to be applied. The following values of IRULE are available:

1. Trapezoid rule (Integration based upon two points)
2. Trapezoid rule with axi-symmetric co-ordinates, i.e. $ds = 2\pi rdst$.
3. Simpson rule (Integration based upon three points)
4. Simpson rule with axi-symmetric co-ordinates, i.e. $ds = 2\pi rdst$.

The default value for IRULE = 1.

If the approximation of the integrand is linear (i.e. two points in each element at the boundary), the trapezoid rule should be applied. For higher order approximations (at least three points in each element at the boundary) Simpson's rule is recommended.

In the case of three-dimensional boundary integrals over surfaces only IRULE=1 is available. The integration to be applied is based upon the type of approximation used to construct the solution. Hence in the case of linear elements a linear type integration rule is applied, whereas for quadratic elements a quadratic integration rule is utilized.

The subkeyword DEGREE_OF_FREEDOM = d defines which unknown in each point from the solution vector (VECTOR i) is used.

When \mathbf{u} is a vector (ICHINT>1), the degrees of freedom u_1, u_2 and u_3 in each nodal point are supposed to be the degrees of freedom $d, d+1$ and $d+2$ respectively.

When σ is a tensor in R^2 σ^{11} corresponds to d, σ^{12} to $d+1$ and σ^{22} to $d+2$.

The default value for d is 1.

The subkeyword CURVES (C1, C2, C3, ...), defines over which curves the integral must be computed. This subkeyword may only be used in R^2 . If a curve must be integrated in reversed direction, the curve number must be provided with a minus sign. Of course this possibility makes only sense for ICHINT > 1.

The subkeyword SURFACES (S1, S2, s3, ...), defines over which surfaces the integral must be computed. This subkeyword may only be used in R^3 .

Mark that the subkeywords curves and surfaces are mutually exclusive.

3.2.15 The main keyword TIME_INTEGRATION

The block defined by the main keyword TIME_INTEGRATION indicates that an instationary problem has to be solved. Unless stated otherwise the time integration corresponds to a problem that contains first order time derivatives only, like for example the heat equation or the instationary Navier-Stokes equation.

In this block information concerning the time integration process must be defined.

The block defined by the main keyword TIME_INTEGRATION has the following structure (options are indicated between the square brackets "[" and "]"):

```

TIME_INTEGRATION, SEQUENCE_NUMBER = s
  METHOD = type1
  TINIT = t0
  TEND = (t1, t2, t3, ... ,tN)
  THETA = (theta1, theta2, ... , thetaM)
  TSTEP = (dt1, dt2, dt3, ... ,dtN)
  TOUTINIT = t
  TOUTEND = t
  TOUTSTEP = t
  ACCURACY = eps
  ABS_STATIONARY_ACCURACY = eps
  REL_STATIONARY_ACCURACY = eps
  BETA = beta
  GAMMA = gamma
  ALPHA_F = alpha_f
  ALPHA_M = alpha_m
  MASS_MATRIX = typem
  STIFFNESS_MATRIX = types
  RIGHT_HAND_SIDE = typer
  SEQ_OUTPUT = iseq
  SEQ_SOLUTION_METHOD = iseq
  SEQ_COEFFICIENTS = iseq_1, iseq_2, ...
  SEQ_BOUNDARY_CONDITIONS = iseq
  NUMBER_OF_COUPLED_EQUATIONS = n
  BOUNDARY_CONDITIONS = typeb
  LINEAR_SUBELEMENT
  DIAGONAL_MASS_MATRIX
  CONSISTENT_MASS_MATRIX
  PRINT_LEVEL = p
  THRESHOLD_TIME = t
  AT_ERROR = type_err
  USE_CTIMEN
  KEEP_T
  REUSE_TIME_PARAMETERS
  PRINT_TIME_HISTORY = ((x1,y1,z1), (x2,y2,z2), ... (xn,yn,zn))
  NON_LINEAR_ITERATION
  UPDATED_LAGRANGE
  SKIP_MESH_DEFORMATION
  ABS_ITERATION_ACCURACY = a
  REL_ITERATION_ACCURACY = a
  MAX_ITER = m
  NO_COMPUTATION
  SEQ_ADD_RHSD = iseq
  EQUATION i
  LOCAL_OPTIONS followed by:

```

```

MASS_MATRIX = typem
STIFFNESS_MATRIX = types
RIGHT_HAND_SIDE = typer
SEQ_SOLUTION_METHOD = iseq
SEQ_COEFFICIENTS = iseq
SEQ_LOCAL_TIME_STEP = iseq
SEQ_VELOCITY = iseq
SEQ_ACCELERATION = iseq
SEQ_MATRIX = i
LINEAR_SUBELEMENT
DIAGONAL_MASS_MATRIX
CONSISTENT_MASS_MATRIX
VECTOR = i
LIMIT = type_limit
REACTION_FORCE = j
SEQ_ADD_RHSD = iseq
DERIVATIVES, options (all in one line)
  where options may have the following contents
    SEQ_COEF = s
    SEQ_DERIV = s
    PROBLEM = s
    VECTOR = s
END

```

The block starts with the mandatory keyword `TIME_INTEGRATION` and ends with the mandatory keyword `END`. The main keyword must start at a new line, the `END` keyword may not be followed by other keywords in the same line. The data keywords in between may be placed each at a new line, however, there is no necessity in doing so, since in this block a newline character is treated as any other separation character.

The keywords have the following meaning:

TIME_INTEGRATION opens the input for the time integration.

The sequence number s may be used to distinguish between various input blocks with respect to time integration. The sequence of the subkeywords is arbitrary and none of the subkeywords is mandatory.

Default value: $s = 1$.

METHOD defines the type of time integration that is applied.

The following values of *type1* are allowed:

```

STATIONARY
EULER_TRAPEZOID
ROZENBROCK_WANNER
RUNGE_KUTTA_12
RUNGE_KUTTA_34
EULER_IMPLICIT
CRANK_NICOLSON
EULER_EXPLICIT
RUNGE_KUTTA_4
THETA
FRACTIONAL_STEP
CENTRAL_DIFFERENCES
GENERALIZED_THETA
SECOND_ORDER_GEAR
NEWMARK
GENERALIZED_ALPHA

```

STATIONARY means that in each time step a stationary problem is solved. In this case the time stepping is only used to change the data, like boundary conditions, position of the boundary, coefficients.

EULER_TRAPEZOID means that a Crank-Nicolson type integration with self-selecting step-size is used. In order to make an estimation of the error in each step the solution is predicted with an implicit Euler step and corrected with an implicit Crank-Nicolson step. This is a second order accurate implicit method.

ROZENBROCK_WANNER is not yet available.

RUNGE_KUTTA_12 means that a Runge-Kutta-Fehlberg type integration is applied with self-selecting step-size. This is a second order accurate explicit method.

RUNGE_KUTTA_34 is not yet available.

EULER_IMPLICIT means that an implicit first order Euler integration is applied. The step-size is selected by the user.

CRANK_NICOLSON means that an implicit second order Crank-Nicolson integration is applied. The step-size is selected by the user.

EULER_EXPLICIT means that an implicit first order Euler integration is applied. The step-size is selected by the user.

RUNGE_KUTTA_4 is not yet available.

THETA means that an implicit θ method is applied. The step-size is selected by the user. Suppose one wants to solve the ordinary differential equation:

$$\frac{dc}{dt} = f(x, t), \quad (3.2.15.1)$$

then the θ method can be written as:

$$\frac{c^{n+1} - c^n}{\Delta t} = \theta f(x, t^{n+1}) + (1 - \theta) f(x, t^n) \quad (3.2.15.2)$$

where n denotes the time level and Δt the time-step. θ must be chosen in the range $0 \leq \theta \leq 1$.

For $\theta = 0$ the method reduces to the classical explicit Euler method, for $\theta = 1$ to the implicit Euler method. $\theta = \frac{1}{2}$ corresponds to the Crank Nicolson scheme.

In our code θ must be in the interval $(0.5, 1)$, for which the method is unconditionally stable.

The Crank Nicolson scheme is second order accurate and is preferred if a time-accurate method is required. However, a clear disadvantage of Crank Nicolson is that high frequency perturbations are not damped. For that reason a transient may be always visible if Crank Nicolson is applied. In order to damp the effect of the transient frequently values of θ greater than 0.5 are used. For example $\theta = 0.55$ is very popular.

The Euler implicit scheme may be less accurate but disturbances are damped very rapidly. Hence, if one is only interested in a stationary solution, this method is the one to use.

FRACTIONAL_STEP A disadvantage of the θ -method is the fixed θ . It could be advantageous to combine a number of different θ 's per time step in such a way that second order accuracy is accomplished, and some damping is ensured as well. Two methods that offer this opportunity are the fractional θ -method and the generalized θ -method. The latter is a generalization of the fractional θ -method, so we will restrict ourselves to the description of the generalized θ -method. We rewrite equation (3.2.15.2) as follows, letting $\Sigma_k = \sum_{i=1}^k \theta_i$:

$$\begin{aligned} c^{n+\Sigma_2} &= c^n + \Delta t (\theta_1 f(x, t^n) + \theta_2 f(x, t^{n+\Sigma_2})) \\ c^{n+\Sigma_4} &= c^{n+\Sigma_2} + \Delta t (\theta_3 f(x, t^{n+\Sigma_2}) + \theta_4 f(x, t^{n+\Sigma_4})) \end{aligned} \quad (3.2.15.3)$$

$$\begin{array}{c} \vdots \\ \vdots \\ c^{n+\Sigma_{2k}} = c^{n+\Sigma_{2k-2}} + \Delta t (\theta_{2k-1} f(x, t^{n+\Sigma_{2k-2}}) + \theta_{2k} f(x, t^{n+\Sigma_{2k}})) \end{array}$$

There are two necessary conditions:

1. $\Sigma_{2k} = 1$ for a k -stage method. This gives a first order method, and is only a scaling requirement.
2. $\sum_{i=1}^k \theta_{2i-1}^2 = \sum_{i=1}^k \theta_{2i}^2$ to guarantee second order accuracy.

A third condition is optional, but guarantees some damping:

1. $\theta_{2i-1} = 0$ for at least one $i \in 1, \dots, k$.

This condition includes at least one Implicit Euler step per time step.

GENERALIZED_THETA The generalized θ -method is a 3-stage method, and is therefore 3 times as expensive as the Crank-Nicolson method. However, one may choose $\Delta t_{gen\theta} = 3 \cdot \Delta t_{CN}$ to accomplish similar results for both methods. A common choice for the generalized θ -method is the following ‘optimum’ for $k = 3$:

$$\begin{aligned} \theta_1 = \theta_5 = \frac{\alpha}{2}, \quad \theta_3 = 0, \\ \theta_2 = \theta_6 = \alpha \frac{\sqrt{3}}{6}, \quad \theta_4 = \alpha \frac{\sqrt{3}}{3} \\ \alpha = \left(1 + \frac{2}{\sqrt{3}}\right)^{-1}. \end{aligned} \tag{3.2.15.4}$$

A common choice for the fractional θ -method is the following:

$$\begin{aligned} \theta_1 = \theta_5 = \beta\theta, \quad \theta_3 = \alpha(1 - 2\theta), \\ \theta_2 = \theta_6 = \alpha\theta, \quad \theta_4 = \beta(1 - 2\theta), \\ \alpha = \frac{1 - 2\theta}{1 - \theta}, \quad \beta = \frac{\theta}{1 - \theta}, \\ \theta = 1 - \frac{1}{2}\sqrt{2}. \end{aligned} \tag{3.2.15.5}$$

SECOND_ORDER_GEAR The first order equation is solved by the implicit second order gear method.

CENTRAL_DIFFERENCES The system of time-dependent equations to be solved is supposed to contain only second derivatives of time and no first order time-derivatives. Hence:

$$\mathbf{M} \frac{\partial^2 \mathbf{u}}{\partial t^2} + \mathbf{S} \mathbf{u} = \mathbf{F} \tag{3.2.15.6}$$

The time-derivatives are discretized by a central difference scheme. The other terms are treated explicitly hence:

$$\mathbf{M} \frac{\mathbf{u}^{n+1} - 2\mathbf{u}^n + \mathbf{u}^{n-1}}{\Delta t^2} + \mathbf{S} \mathbf{u}^n = \mathbf{F}^n \tag{3.2.15.7}$$

The step size must be given by the user.

This problem requires both an initial condition and the initial derivative. These solutions must be stored in two consecutive solution vectors. After the time integration the first vector contains the solution at $t = \text{TEND}$, and the second vector the solution at the prior time step.

NEWMARK is used for the same type of problems as `central_differences`. It is an implicit scheme for the solution of 3.2.15.6, which can be written as:

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \Delta t \mathbf{v}^n + \frac{\Delta t^2}{2} ((1 - 2\beta) \mathbf{a}^n + 2\beta \mathbf{a}^{n+1}) \tag{3.2.15.8}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \Delta t((1 - \gamma)\mathbf{a}^n + \gamma\mathbf{a}^{n+1}) \quad (3.2.15.9)$$

$$\mathbf{M}\mathbf{a}^{n+1} + \mathbf{S}\mathbf{u}^{n+1} = \mathbf{F}^{n+1} \quad (3.2.15.10)$$

The step size must be given by the user.

This problem requires both an initial condition and the initial derivative.

The user must define three vectors for the displacement u , the velocity v and the acceleration a .

The sequence number of the displacement vector in the list of solution vectors is defined in the structure input block by:

`time_integration, sequence_number = i, vector = %u,`

with `u` the name of the displacement vector. If omitted the first vector is used.

The sequence numbers of the velocity and acceleration vectors can be defined by the keywords `seq_velocity` and `seq_acceleration`, which must be defined under `EQUATION 1`. If omitted they get the sequence number of the velocity plus one and two respectively.

The parameters β and γ must be defined by the keywords `beta` and `gamma`. If omitted, the default values $\beta = 0.25$ and $\gamma = 0.5$ are used, which ensure second order accuracy in time.

GENERALIZED_ALPHA is an extension of Newmark, which allows damping of higher order frequencies.

The formulas (3.2.15.8) and (3.2.15.9) are reused, but (3.2.15.10) is replaced by:

$$\mathbf{M}\mathbf{a}^{n+1-\alpha_m} + \mathbf{S}\mathbf{u}^{n+1-\alpha_f} = \mathbf{F}^{n+1-\alpha_f}, \quad (3.2.15.11)$$

with

$$\mathbf{a}^{n+1-\alpha_m} = (1 - \alpha_m)\mathbf{a}^{n+1} + \alpha_m\mathbf{a}^n, \quad (3.2.15.12)$$

and

$$\mathbf{u}^{n+1-\alpha_f} = (1 - \alpha_f)\mathbf{u}^{n+1} + \alpha_f\mathbf{u}^n. \quad (3.2.15.13)$$

The parameters α_m and α_f may be defined by the keywords `ALPHA_F` and `ALPHA_M`.

The default values for generalized alpha are:

$\beta = 1$, $\gamma = 1.5$, $\alpha_m = -1$ and $\alpha_f = 0$. This combination is second order accurate in time and damps higher order frequencies.

The default value for `METHOD` is `EULER_IMPLICIT`.

TINIT defines the initial time t_0 .

Default value: 0

TEND defines a series of end times $t_1, t_2, t_3, \dots, t_N$. The process is restarted at each of these end times. In each of the steps a new time step may be chosen.

At most 10 end times are permitted. Of course it is necessary that $t_{i+1} > t_i$.

Default value: 1, except when `STATIONARY_ACCURACY` is given in which case it is 1000.

THETA defines a series of θ values either for the θ method or for the fractional or generalized θ methods.

In case of a θ method only one θ value may be given. If omitted the default value 1 is used.

In case of a fractional θ method the default values of θ are given by the series:

$$\begin{aligned} \theta_1 = \theta_5 = \beta\theta, \quad \theta_3 = \alpha(1 - 2\theta), \\ \theta_2 = \theta_6 = \alpha\theta, \quad \theta_4 = \beta(1 - 2\theta), \\ \alpha = \frac{1 - 2\theta}{1 - \theta}, \quad \beta = \frac{\theta}{1 - \theta}, \\ \theta = 1 - \frac{1}{2}\sqrt{2}. \end{aligned} \quad (3.2.15.14)$$

In case of a generalized θ method the default values of θ are given by the series:

$$\begin{aligned} \theta_1 = \theta_5 &= \frac{\alpha}{2}, & \theta_3 &= 0, \\ \theta_2 = \theta_6 &= \alpha \frac{\sqrt{3}}{6}, & \theta_4 &= \alpha \frac{\sqrt{3}}{3} \\ \alpha &= \left(1 + \frac{2}{\sqrt{3}}\right)^{-1}. \end{aligned} \tag{3.2.15.15}$$

TSTEP defines a series of time steps (Δt). The number of time-steps given must be equal to the number of end times given.

It is only necessary to give a time step if no self-selecting time-step mechanism is present.

Default value: 0.1 ($t_N - t_0$)

TOUTINIT defines the first time at which the solution is written to the file sepcomp.out for output purposes. If the time t is almost equal to TOUTINIT output is performed at that time, otherwise the first time that is larger than TOUTINIT is used.

Default value: no output is written at all.

TOUTEND defines the last time for which output is written.

Default value: if toutinit given then t_N

TOUTSTEP defines the frequency for which output is written.

Default value: if toutinit given then TOUTEND - TOUTINIT

ACCURACY is only necessary for self-selecting step-sizes. It defines the accuracy with which the time-stepping process is computed.

Default value: 10^{-2} .

ABS_STATIONARY_ACCURACY is only necessary if a stationary solution of the time-dependent equations is required.

The process stops if the difference between end solution and present solution is less than ϵ . The difference is measured in the norm of the vector, and the linear convergence properties of the process are accounted for. If the end time is reached, without finding a converged solution, a warning is issued.

If the process clearly diverges an error message is given.

Default value: 0.

REL_STATIONARY_ACCURACY has exactly the same meaning as ABS_STATIONARY_ACCURACY, however, in this case the accuracy is measured with respect to the norm of the final solution. REL_STATIONARY_ACCURACY may be combined with ABS_STATIONARY_ACCURACY, in which case the process is stopped if one of the criteria is fulfilled.

If both ABS_STATIONARY_ACCURACY and REL_STATIONARY_ACCURACY are omitted the process is assumed to be instationary.

Default value: 0.

MASS_MATRIX gives information about the mass matrix. The following values of *typem* are allowed:

CONSTANT means that the mass matrix does not depend on time. This is the case if the coefficient for the time-derivative is time-independent and no upwinding is applied.

VARIABLE means that the mass matrix may vary in time.

Default value: VARIABLE

STIFFNESS_MATRIX gives information about the stiffness matrix. The following values of *types* are allowed:

CONSTANT means that the stiffness matrix does not depend on time. This is the case if the coefficients in the equation are time-independent and no upwinding is applied. The coefficient for the time derivative nor the right-hand side play a role in this respect.

VARIABLE means that the stiffness matrix may vary in time.

Default value: VARIABLE

RIGHT_HAND_SIDE gives information about the right-hand side vector. The following values of *typer* are allowed:

CONSTANT means that the right-hand side vector does not depend on time.

VARIABLE means that the right-hand side vector may vary in time.

ZERO means that the right-hand side vector is equal to zero. This is the case if no source term is present. The effect of boundary conditions does not play a role in this respect.

Default value: VARIABLE

SEQ_OUTPUT defines the sequence number of the information of the output. This sequence number is only used if TOUTINIT is given.

The sequence number refers to the sequence number of the block corresponding to the main keyword OUTPUT.

If no block OUTPUT is available the default values for the output block are used.

Default value: 1

SEQ_SOLUTION_METHOD defines the sequence number of the information of the linear solver.

The sequence number refers to the sequence number of the block corresponding to the main keyword SOLVE.

If no block SOLVE is available the default values for the linear solver are used.

It is in general advised to use a compact storage instead of a profile storage since time-dependent problems have usually a very good condition and a good starting value for the iteration process.

If a compact storage is used, always the solution at the preceding time level is used as start for the iteration process. Besides that, if the mass matrix, the stiffness matrix and the time-step are constant, the preconditioning matrix is kept after the first time-step and reused in the next time steps.

Default value: 1

SEQ_COEFFICIENTS defines the sequence numbers of the information of the coefficients.

These sequence number refer to the sequence number of the block corresponding to the main keyword COEFFICIENTS. For each coupled equation a separate sequence number is necessary.

Default values: 1, 2, ... , n , with n the number of coupled equations.

SEQ_BOUNDARY_CONDITIONS defines the sequence numbers of the information of the essential boundary conditions.

This sequence number refers to the sequence number of the block corresponding to the main keywords ESSENTIAL BOUNDARY CONDITIONS.

If no block ESSENTIAL BOUNDARY CONDITIONS is available the essential boundary conditions are set equal to zero.

In combination with the option `boundary_conditions = old_vector` this keyword has a different meaning. See below.

Default value: 1.

NUMBER_OF_COUPLED_EQUATIONS defines the number of coupled equations n . The equations to be solved correspond to the sequence numbers 1, 2, ... , n , except when in the block STRUCTURE the command:

`SOLVE_TIME_DEPENDENT_PROBLEM`, `vector = i`

is given. In that case the sequence numbers are $i, i+1, \dots, i+n-1$.

Default value: 1.

BOUNDARY_CONDITIONS gives extra information about the boundary conditions.

The following values for *typeb* are available:

```
constant
variable
initial_field
old_vector
```

Meaning of these parameters:

CONSTANT means that the essential boundary conditions are constant in time and hence have to be computed only once.

VARIABLE means that the essential boundary conditions are time-dependent and must be evaluated in each time-step.

INITIAL_FIELD means that the essential boundary conditions are constant in time and equal to the boundary conditions of the initial condition. hence the values are copied from the initial condition.

OLD_VECTOR means that the essential boundary conditions are time-dependent and must be stored in a solution vector. The sequence number i of this solution vector in the set of solution vectors must be given by `seq_boundary_conditions = i`. If the number of coupled vectors is equal to 1, this implies that the boundary conditions are stored in the solution vector with sequence number i , if there are more than one coupled equations, they must be stored in $i, i+1, i+2, \dots$.

Mark that the boundary conditions must be filled at the new time level, i.e. $t + \Delta t$. So this option can only be used in those methods that have a fixed time step and no intermediate time steps. If the boundary condition is time dependent, this also implies that the user can not use the standard solution vector for this option, because that solution vector must contain the boundary condition at the old time level.

For an example of the use of this option see Section [6.4.3](#)

Default value: VARIABLE

LINEAR_SUBELEMENT indicates that quadratic elements are treated as a cluster of linear elements. For example a 6-node triangle is locally subdivided into 4 3-node triangles. The matrix is built with these linear elements. Of course this option influences the type of approximation and hence the accuracy.

Mark that this option can only be applied if the number of degrees of freedom per point is constant.

Default value: no subdivision

DIAGONAL_MASS_MATRIX indicates that the mass matrix must be built as diagonal matrix, i.e. lumping must be applied. Whether or not this option makes sense depends on the application. Consult the manual Standard Problems for this matter.

Default value: full (consistent) matrix.

CONSISTENT_MASS_MATRIX indicates that the mass matrix must be built as a full (consistent) matrix with the same structure as the stiffness matrix.

The options `DIAGONAL_MASS_MATRIX` and `CONSISTENT_MASS_MATRIX` are mutually exclusive.

Default value: full (consistent) matrix.

PRINT_LEVEL = p defines the amount of output produced by the time integration subroutine in case a stationary solution is required. The following values for p are allowed:

- 1 All output information is suppressed, even the message of no-convergence.
- 0 No information about the iteration process is printed except in the case of an error.
- 1 The number of iterations performed is printed.
- 2 Some extra information about the iteration process is printed for each iteration.
- 3 See 2, in this case also the ratio of two succeeding iterations is printed.
- 100 Contains the maximum of preceding values and also prints the solution in each iteration.

Mark that in this case with iteration a time-step is meant, since time-stepping is used as iteration process.

Default value: 0

THRESHOLD_TIME = t_{thresh} is only used in case a stationary solution is wanted. The convergence of the time-stepping process considered as an iteration method is checked in each step, but the divergence of the iteration process is only checked as soon as $t > t_{thresh}$.

This value may be necessary since due to the transient it may seem as if the process diverges, whereas it converges after some time.

Default value: $t_0 + 10\Delta t$.

AT_ERROR = $type_err$ is only used in case a stationary solution is wanted. The following values for $type_err$ are allowed:

STOP The program halts if divergence of the iteration process has been found and an error message is given.

RETURN If divergence of the iteration process is found control is given back to the main program and the last computed value is kept as solution.

Default value: STOP

PRINT_TIME_HISTORY = $((x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n))$ defines in which points the time-history must be printed. The co-ordinates of the points must be given in pairs surrounded by brackets. The number of co-ordinates per point is defined by the dimension of space.

The nodal points closest to the co-ordinates given are detected and a time history is printed for all unknowns and all equations in these nodal points, including the corresponding time. The advantage above writing all vectors to the SEPRAN output file `sepcomp.out` is that the number of time steps is not restricted and that the amount of output is relatively small. Program SEPPOST is at this moment to a maximum of 100 time steps and might therefore give less information for printing time histories.

USE_CTIMEN Indicates if the time step, initial time and end time are given in the input file or by the user in common block CTIMEN as described in the Programmers Guide Section 11.3.

KEEP_T Corresponds to `use_ctimen`. If this parameter is present the time is not changed during the time step and the parameters in common CTIMEN are used.

REUSE_TIME_PARAMETERS Is very similar to `keep_t` but it corresponds to `no_computation`.

It is supposed that in the call corresponding to NO_COMPUTATION all input with respect to initial time, end time, time step and output times have been read. Furthermore the time has been raised before in that part.

In this part no time input is read. During the computation first the time is reset to the previous time level and afterwards the time is raised to the original value. This option is meant for the use in combination with `time_loop` in the STRUCTURE block.

For an example of the use of this option see Section [6.4.3](#)

NON_LINEAR_ITERATION indicates that a non-linear time-dependent problem is solved. In each time step an iteration method is applied until convergence is achieved.

UPDATED_LAGRANGE is a variant of `non_linear_iteration`. Instead of a standard, Picard type, iteration, the updated Lagrange method is used. This method is meant for non-linear elasticity problems with large deformations, as described in the manual Standard Problems, Section 5.3.2.

This method requires two solution vectors u and un , which, at this moment, must be stored as first and second solution vector. un contains the total displacement, u the displacement during the time step.

At the end of the time-step the mesh is deformed.

This option is only available in combination with Newmark or Generalized alpha, in other works only for second order time derivatives.

SKIP_MESH_DEFORMATION is only effective in combination with `updated_lagrange`. If used the deformation of the mesh is skipped.

ABS_ITERATION_ACCURACY = ϵ , defines the absolute accuracy corresponding to the non-linear iteration process activated by the keyword `non_linear_iteration`. The process is halted if the difference between two succeeding iterations is less than ϵ .
Default value: $\epsilon = 10^{-2}$.

REL_ITERATION_ACCURACY = ϵ defines the relative accuracy corresponding to the non-linear iteration process activated by the keyword `non_linear_iteration`. The process is halted if the difference between two succeeding iterations subdivided by the norm of the solution is less than ϵ .
Default value: not used.

MAX_ITER = m defines the maximum number of iterations allowed in the non-linear iteration process activated by the keyword `non_linear_iteration`.
Default value: 1.

NO_COMPUTATION indicates that no computation is carried out. All that is done is setting the time step, initial time, end time and the output times.

Furthermore in each call the time is raised with the time step. This option is meant for a time loop where from a special moment the time must be increased to a new level. Only after that a computation takes place.

For an example of the use of this option see Section [6.4.3](#)

SEQ_ADD_RHSD = *vecname* indicates that the vector with name *vecname*, must be added to the right-hand side of the equations.

This vector may be created by:

```
vecname = right_hand_side, problem = p, seq_coef = i
```

See Section [3.2.3.10](#).

Default: no adding of vector

EQUATION i indicates that separate information about the i^{th} coupled equation is given. All information read in this part until a new subkeyword `EQUATION` or the keyword `END` is found, corresponds to this specific equation. Local options are only valid for the local equation. They overrule the global options given before.

Mark that first the global options must be given and then the specific information about equations. The keyword `EQUATION` may be followed by the subkeywords:

```
LOCAL_OPTIONS followed by subkeywords
DERIVATIVES, options (all in one line)
```

LOCAL_OPTIONS indicate that local options are defined for this equation that overrule the global options given before. The following local options are available

```

MASS_MATRIX = typem
STIFFNESS_MATRIX = types
RIGHT_HAND_SIDE = typer
SEQ_SOLUTION_METHOD = iseq
SEQ_COEFFICIENTS = iseq
SEQ_LOCAL_TIME_STEP = iseq
LINEAR_SUBELEMENT
DIAGONAL_MASS_MATRIX
CONSISTENT_MASS_MATRIX
VECTOR = i
LIMIT = type_limit
REACTION_FORCE = j
SEQ_VELOCITY = k
SEQ_ACCELERATION = l

```

Most of these options have exactly the same meaning as the global options, however, they are restricted to the specific equation.

The default values are always the global options.

The following options are local:

```

SEQ_LOCAL_TIME_STEP = iseq
REACTION_FORCE = j

```

Meaning of these subkeywords:

LIMIT = type_limit indicates that the solution must be limited between two values.

Possible values for `type_limit` are:

```

none
minmax

```

none (default value) means no limiting

minmax means that the solution is limited between the minimum and maximum value at the previous time level, where the boundary conditions are taken at the new time level. This option may be used to prevent a solution to be less than the smallest value in the previous time level (for example zero) or more than the highest value (overshoot).

This limiter must only be used if all other possibilities like upwind fail.

SEQ_LOCAL_TIME_STEP = j indicates that the time step is space dependent. For each node a time step must be provided by the user. This time step must be stored in the set of solution vectors, with sequence number *j*.

SEQ_VELOCITY = j defines the sequence number of the velocity vector (time derivative of the unknown) in the set of solution vectors. This option is only used in the Newmark and generalized alpha method.

If omitted the sequence number is set equal to the sequence number of the unknown vector (usually the displacement) plus 1.

SEQ_ACCELERATION = j defines the sequence number of the acceleration vector (second order time derivative of the unknown) in the set of solution vectors. This option is only used in the Newmark and generalized alpha method.

If omitted the sequence number is set equal to the sequence number of the unknown vector (usually the displacement) plus 2.

SEQ_MATRIX = j defines the sequence number of the stiffness and mass matrices as well as the right-hand side, to be used in the Newmark or generalized alpha method in the set of matrices.

This option makes only sense if you overwrite the matrix by another time integration method or solve step. Since the default value is 2, you only have to prescribe the sequence number if you solve another time integration method with more than one coupled equation. In that case you have to raise the sequence number.

The reason that we are not allowed to overwrite the matrices and right-hand side, is that they are reused in the next time step.

Default value: 2

REACTION_FORCE = j implies that the reaction force is computed and stored in the set of solution vectors, with sequence number j .

DERIVATIVES must be followed by its options. These options must be given in the same line, where the general rule for continuation as described in Section 1.4 must be applied if the line does not fit in 80 columns.

If **DERIVATIVES** is given, derived quantities can be computed that depend on the just computed quantities. These derived quantities can be used for further computations. The possible options for the keyword are:

```
SEQ_COEF = s
SEQ_DERIV = s
PROBLEM = s
VECTOR = i
```

For many derived quantities it is necessary to define coefficients which are used in the computation process. Consult the manual STANDARD PROBLEMS to check if and which coefficients are required for a specific derived quantity.

These coefficients are defined by the input block "COEFFICIENTS" with sequence number c . If `seq_coef = c` is omitted it is assumed that no coefficients are needed.

`Problem = p` defines the problem sequence number that is used to compute the derived quantities.

If `vector = i` is omitted i gets the value `nprob+1`, which is equal to the number of different problems defined plus one.

Input concerning the derived quantities to be computed is defined in the input block "DERIVATIVES" with sequence number s . If s is omitted the next one is assumed.

The result of this operation is that a vector V_i has been created.

END (mandatory)

Closes the input block TIME_INTEGRATION.

Boundary conditions and coefficients in time-dependent problems may depend on time. If so it is necessary to have the time t available at the moment these functions must be evaluated. For example if the essential boundary conditions depend on time it is necessary to have the time t at ones disposal in the boundary condition function subroutine FUNCBC.

For that reason SEPRAN provides a common block named CTIMEN which contains a number of parameters with respect to the time integration. CTIMEN is updated in the time-integration subroutines and may be used in each local subroutine.

Common CTIMEN has the following shape:

```
double precision t, tout, tstep, tend, t0, theta, rtime
integer iflag, icons, ktime
common /ctimen/ t, tout, tstep, tend, t0, theta, rtime(4), iflag,
+          icons, ktime(8)
```

The parameters in CTIMEN have the following meaning

t Actual time. This is the most important parameter for the user.

tout The integration is carried out from initial time to tout.

tstep Time step for the numerical integration.

tend End time for the numerical integration.

t0 Initial time for the integration.

theta Parameter θ for the θ method.

rtime Dummy array that is not yet defined.

iflag Return code from the time integration subroutines.

Possible values:

3 Successfully return

4 The process has stopped because of errors or inaccuracy

icons Indication if the matrices and time-step are constant (1) or not (0)

ktime Dummy array that is not yet defined.

For an example of the use of CTIMEN the reader is referred to Section [6.4.1](#).

3.2.16 The main keyword CONTACT

The block defined by the main keyword CONTACT indicates that a contact algorithm must be applied to identify points at a surface as contact points or no-contact points. After application of the contact algorithm all points at the given surface are identified and this identification may be used to define boundary conditions, to create vectors or to compute special boundary integrals. At this moment the contact algorithm is restricted to three dimensional problems. Contact may only be defined along a surface.

A contact algorithm is needed if some solid is pressed to a surface and is not able to move through that surface. So at the position where the contact is made (the so-called contact surface) it is necessary to prescribe an essential boundary condition, i.e. that the displacement is equal to 0. The computation of the contact surface is essentially non-linear. Once contact is established, the boundary conditions are changed and this means that if we solve the mechanical problem with the new boundary conditions, also the solution is changed. The effect of the new solution is that the contact surface may be changed. Possibly the contact region may be extended, in which case the region where the essential boundary conditions are prescribed is extended, but it is also possible that the displacement moves away from the contact surface. This situation can not be deducted from the displacement itself, since the prescribed displacement prevents a displacement at all. However, if there are essential boundary conditions we have also a non zero reaction force. If the normal component of the reaction force in a point, points away from the contact surface we release this point by not prescribing the displacement anymore. Effectively this is performed by removing the point from the contact surface. This process is repeated until convergence is established. Convergence may be checked by computing the contact distance i.e. the distance between solid and contact surface and comparing the corresponding contact distance vector in successive iterations. Another option is to check if the contact surface in two successive iterations is the same. This can be done for example by a while loop in a structure block (3.2.3) in combination with the user function subroutine userbool (3.3.8). The information if the contact surface is changed is stored in the common block ccontact, which can be included in your program by `include 'SPcommon/ccontact'`. This common block looks like:

```
integer ninpcp
parameter ( ninpcp=...)
logical contact_changed(ninpcp)
integer contact_ncontct(ninpcp)
common /ccontact/ contact_changed, contact_ncontct
```

If `contact_changed(i) = true` then the i^{th} contact surface has been changed, if false it has not been changed.

For an example of the use of a contact algorithm see the manual SEPRAN EXAMPLES, Chapter 5.5.

In this block information concerning the contact algorithm must be defined.

The block defined by the main keyword CONTACT has the following structure (options are indicated between the square brackets "[" and "]"):

```
CONTACT, SEQUENCE_NUMBER = k
CONTACT_SURFACE = S3
CONTACT_DISTANCE = V5
CONTACT_FORCE = V1
CONTACT_METHOD = type_1
CONTACT_DISABLE_METHOD = type_2
END
```

The block starts with the mandatory keyword CONTACT and ends with the mandatory keyword END. The main keyword must start at a new line, the END keyword may not be followed by other

keywords in the same line. The data keywords in between may be placed each at a newline, however, there is no necessity in doing so, since in this block a newline character is treated as any other separation character.

The keywords have the following meaning:

CONTACT opens the input for the definition of the contact algorithm.

The sequence_number k may be used to distinguish between various input blocks with respect to contact algorithms. The sequence of the subkeywords is arbitrary and none of the subkeywords is mandatory.

Default value: $k = 1$.

CONTACT_METHOD defines the type of contact algorithm that is applied.

The following values of $type_1$ are allowed:

NEG_DISTANCE

NEG_DISTANCE means that all points in the surface that have a negative distance as defined by the CONTACT_DISTANCE vector are marked as contact points.

The default value for $type_1$ is NEG_DISTANCE.

CONTACT_DISABLE_METHOD defines which contact points are unmarked as contact points.

The algorithm applied is as follows:

First it is checked if already points have been marked as contact points. If that is the case it is checked if some of these points must be unmarked as contact points.

Next for all other points it is checked if points must be defined as contact points. Hence points that are unmarked in this step can not be marked in the same step. In order to mark them again as contact points the contact algorithm must be called again.

The following values of $type_2$ are allowed:

CONTACT_FORCE

CONTACT_FORCE means that all points with a negative value of the CONTACT_FORCE vector in that point are unmarked as contact points, provided they have been marked before as contact points.

The default value for $type_2$ is CONTACT_FORCE.

CONTACT_SURFACE = S_j defines the surface along which contact may be made. For each surface only one contact is allowed and for each contact only one surface is allowed.

The default value is S1.

CONTACT_DISTANCE = V_j defines the vector that is used to define the contact distance in the algorithm. The vector V_j must contain exactly one unknown in each point of the contact surface.

The default value is V1.

CONTACT_FORCE = V_j defines the vector that is used to define the contact force in the algorithm. The vector V_j must contain exactly one unknown in each point of the contact surface.

The default value is V2.

END (mandatory)

Closes the input block CONTACT.

3.2.17 The main keyword LOOP_INPUT

The block defined by the main keyword LOOP_INPUT defines when convergence is reached and the iteration must be stopped.

The block defined by the main keyword LOOP_INPUT has the following structure (options are indicated between the square brackets "[" and "] "):

```
LOOP_INPUT [,SEQUENCE_NUMBER = s]
  (mandatory): opens the input for the loop.

    maxiter = m
    miniter = m
    accuracy = eps
    print_level = p
    criterion = c
    at_error = e
    seq_vector = i

END
(mandatory): end of input
```

The sequence number s may be used to distinguish between various input blocks with respect to loops.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords LOOP_INPUT and END however, must be placed at a new record.

Meaning of the subkeywords:

maxiter = m , defines the maximum number of iterations that may be performed.

The default value is 20.

miniter = m , defines the minimum number of iterations that must be performed.

The default value is 2.

accuracy = ε , defines the accuracy. If the difference between succeeding solutions is less than ε the process is considered converged and the iteration is halted.

The default value is $\varepsilon = 10^{-3}$

print_level = p gives the user the opportunity to indicate the amount of output information he wants from the iteration process. p may take the values -1, 0, 1 or 2. The amount of output increases for increasing value of p . If $p = -1$ no output at all is produced.

The default value is $p = 0$

criterion = c defines the type of termination criterion to be used. Possible values are:

```
absolute
relative
```

If **absolute** is used (default value) the process is stopped if $\|\mathbf{u}^{k+1} - \mathbf{u}^k\| \leq \varepsilon$.

If **relative** is used the process is stopped if $\frac{\|\mathbf{u}^{k+1} - \mathbf{u}^k\|}{\|\mathbf{u}^{k+1}\|} \leq \varepsilon$.

Here u^k means the solution at the k^{th} iteration.

The default value is **absolute**

at_error = e defines which action should be taken if the iteration process terminates because no convergence could be found. Possible values are:

stop
return

If **stop** is used the iteration process is stopped if no convergence is found, otherwise (**return**) means that control is given back to the main program and the result of the last iteration is used as solution.

The default value is **stop**.

seq_vector = s , defines the sequence number of the vector that is used to check the convergence. The default value is $s = 1$.

3.2.18 The main keyword EIGENVALUES

The block defined by the main keyword EIGENVALUES defines how eigenvectors and eigenvalues must be computed and to what problem definition these eigenvalues belong.

At this moment only eigenvalues for a symmetric problem can be computed. The method used is an iterative method based on the Lanczos algorithm.

The block defined by the main keyword EIGENVALUES has the following structure (options are indicated between the square brackets "[" and "] "):

```
EIGENVALUES [,SEQUENCE_NUMBER = s]
  eigenvectors
  number_of_eigenvalues = n
  accuracy = eps
  print_level = p
  maxiter = m
  maxinterval = m
  maxpoints = m
  seq_coef = m
  mass_matrix = d
  start_vector = s
  randvalue = i
END
```

The keywords EIGENVALUES and END are mandatory all others are optional.

EIGENVALUES opens the input for the computation of the eigenvectors The sequence number s may be used to distinguish between various input blocks with respect to eigenvalues.

END closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords EIGENVALUES and END however, must be placed at a new record.

Meaning of the subkeywords:

eigenvectors If this keyword is given, not only the eigenvalues, but also the eigenvectors must be computed.

Default value: no eigenvectors

number_of_eigenvalues = n defines the number of eigenvalues and, in case of the presence of the keyword eigenvectors, also the number of eigenvectors.

Default value: 1

accuracy = ε defines the accuracy of the process, i.e. when the iteration is halted.

Default value: 0, i.e. machine accuracy.

print_level = p defines the output with respect to the iteration process.

Possible values:

- 0 No intermediate output is produced
- 1 A moderate amount of extra output is created showing the convergence behavior.
- 2 A maximal amount of extra output is created.

Default value: 0

maxiter = m defines the maximum number of iterations that may be performed.

Default value: 0, which means that the subroutine defines it himself based on the number of unknowns.

maxinterval = m defines the maximum number of intervals used. This parameter makes only sense for specialists.

Default value: 0, which means that the subroutine defines it himself based on the number of unknowns.

maxpoints = m defines the maximum number of intervals used. This parameter makes only sense for specialists.

Default value: 0, which means that the subroutine defines it himself based

seq_coef = m the sequence number m refers to the coefficients that must be used to compute the stiffness matrix. In fact this parameter defines the coefficients of equation for which the eigenvalues must be computed.

Default value: 1

mass_matrix = d defines how the mass matrix must be constructed. Possible values for d are:

diagonal
consistent

meaning of these subsubkeywords

diagonal the mass matrix is lumped to a diagonal matrix. This method is only accurate if linear (or bi/tri linear) elements are used.

consistent the mass matrix is computed in an accurate way and is therefore a full matrix. As a consequence the process is more time-consuming and requires an extra amount of memory.

Default value: diagonal

start_vector = s defines how the start vector must be created. Possible values for s are:

structured
random

meaning of these subsubkeywords

structured the starting vector is a smooth vector.

random the starting vector is randomly defined.

Default value: structured

randvalue = i defines a parameter to create the random vector.

Default value: 1

for an example of the computation of eigenvalues and eigenvectors the reader is referred to Section [6.8.1](#)

3.2.19 The main keyword CAPACITIES

The block defined by the main keyword CAPACITIES defines how capacities between electrodes must be computed.

Consider a region containing some material with a given permittivity ε . On the boundary of this region a number of electrodes (sensors) are placed. See for example Figure 3.2.19.1.

All electrodes are kept at zero potential except one for which the potential is set equal to 1. Because

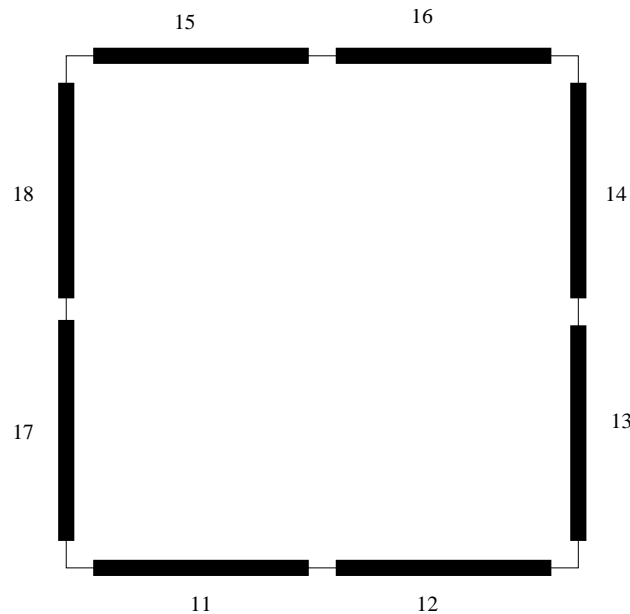


Figure 3.2.19.1: Definition of region and electrodes for square 2d sensor

this process is usually used to compute the permittivity of the material (which is a so-called inverse problem), a loop is made in order that all the electrodes will become the electrode with potential 1. The potential in the inner region satisfies the potential equation: $\text{div } \varepsilon \nabla V = 0$.

The capacity c_{ij} is defined as: $c_{ij} = \int_{E_j} \varepsilon V d\Gamma$, where i refers to the electrode where the potential V

is equal to 1, and j refers to the j^{th} electrode E_j over which the integral must be computed. Due to symmetry, it is sufficient to compute c_{ij} only for $j < i$.

In order to compute the integrals accurately a special property of the finite element method is used. These integrals are exactly the sum of the reaction forces over the electrodes. As a consequence the user must fill the reaction forces. Actually this means that the user must set the value of IBCMAT in the input block MATRIX equal to 1. See 3.2.4. The reaction forces itself are computed automatically in this case.

The block CAPACITIES is used to perform the loops over all the electrodes and to compute all the capacities and store them in an array. The capacities are stored in the sequence (1,2), (1,3), ... (1, n), (2,3), (2,4), ... , where n is the number of electrodes. Hence the length of this vector is $\frac{n \times (n-1)}{2}$.

The block defined by the main keyword CAPACITIES has the following structure (options are indicated between the square brackets "[" and "]"):

```
CAPACITIES [,SEQUENCE_NUMBER = s]
  lin_solver = i
  curve_begin = j
  curve_end = j
  solution_vector = s
  capacity_vector = s
```

```
    reaction_vector = s
    seq_coef = m
END
```

The keywords CAPACITIES and END are mandatory all others are optional.

CAPACITIES opens the input for the computation of the capacities The sequence number s may be used to distinguish between various input blocks with respect to capacities.

END closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords CAPACITIES and END however, must be placed at a new record.

Meaning of the subkeywords:

lin_solver = i defines the sequence number of the input for the linear solver that is used to solve the potential problem.

Default value: 1

seq_coef = m defines the sequence number of the input for the coefficients that are used to solve the potential problem.

Default value: 1

curve_begin = i It is required that all electrodes coincide with one curve each. Furthermore these numbers must be numbered in increasing sequence without gaps. i defines the curve number of the first electrode.

Default value: 1

curve_end = j defines the curve number of the last electrode. Hence the number of electrodes is equal to $j - i + 1$.

Default value: 2

solution_vector = s defines the sequence number of the solution vector. Mark that all essential boundary conditions must have been filled in this vector. Only the varying potential 1 on the various sensors is filled by the program itself.

Default value: 1

capacity_vector = s defines the sequence number of the capacity vector. This vector contains the result of the computations. It is a vector with a special adapted structure with length $\frac{N(N-1)}{2}$, with N the number of electrodes.

Default value: 3

reaction_vector = s defines the sequence number of the reaction force vector. See the introduction of this section.

Default value: 2

For an example of the computation of CAPACITIES the reader is referred to Section [6.2.10](#)

3.2.20 The main keyword INVERSE_PROBLEM

The block defined by the main keyword INVERSE_PROBLEM defines how inverse problems may be solved.

An inverse problem is a problem where the coefficients of the differential equation are unknown. However, instead of the coefficients one has a number of measurements under different conditions. Using these measurements, one tries to estimate the coefficients. In general an inverse problem is non-linear, difficult to solve and the solution may be non-unique.

In this section we try to solve the inverse problem. At this moment this is only possible for one specific case. Moreover there is only one solution method available. It is our purpose to extend the number of solution methods in the future.

The inverse problem we are able to solve at this moment is that of a medium with unknown permittivities. The medium is present in some closed domain.

On the boundary of this region a number of electrodes (sensors) are placed. See for example Figure 3.2.19.1.

All electrodes are kept at zero potential, except one, for which the potential is set equal to 1. Now it is possible to measure the capacity of all electrodes. By varying the sensor for which the potential is set equal to 1, we can measure $\frac{N(N-1)}{2}$ independent capacities.

Assuming a given set of permittivities one can compute the corresponding capacities for the various given potentials at the sensors. By changing the permittivities one can try to find that permittivity in the domain that fits the measurements as close as possible.

The block INVERSE_PROBLEM is used to prescribe how the inverse problem must be solved.

The block defined by the main keyword INVERSE_PROBLEM has the following structure (options are indicated between the square brackets "[" and "]"):

```
INVERSE_PROBLEM [,SEQUENCE_NUMBER = s]
  lin_solver = i
  curve_begin = j
  curve_end = j
  solution_vector = s
  capacity_vector = s
  reaction_vector = s
  epsilon_vector = s
  seq_coef = m
  element_group = i
  method = m
  regular_parm = r
  delta_eps = e
  eps_ref = e
END
```

The keywords INVERSE_PROBLEM and END are mandatory all others are optional.

INVERSE_PROBLEM opens the input for the computation of the INVERSE_PROBLEM The sequence number *s* may be used to distinguish between various input blocks with respect to INVERSE_PROBLEM.

END closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords INVERSE_PROBLEM and END however, must be placed at a new record.

Meaning of the subkeywords:

lin_solver = i defines the sequence number of the input for the linear solver that is used to solve the potential problem.

Default value: 1

seq_coef = m defines the sequence number of the input for the coefficients that are used to solve the potential problem with the computed permittivity (epsilon) values.

Since the permittivity changes during the computation it is necessary that the permittivity is stored in a vector. At this moment this vector must be a vector of special structure defined per element. In the coefficients block the corresponding coefficient must refer to this permittivity vector. See for an example Section 3.2.20

Default value: 1

curve_begin = i It is required that all electrodes coincide with one curve each. Furthermore these numbers must be numbered in increasing sequence without gaps. i defines the curve number of the first electrode.

Default value: 1

curve_end = j defines the curve number of the last electrode. Hence the number of electrodes is equal to $j - i + 1$.

Default value: 2

solution_vector = s defines the sequence number of the solution vector. In this case the solution vector is the potential vector. Mark that all essential boundary conditions must have been filled in this vector. Only the varying potential 1 on the various sensors is filled by the program itself.

Default value: 1

capacity_vector = s defines the sequence number of the capacity vector. This vector must contain the measurements. It is a vector with a special adapted structure with length $\frac{N(N-1)}{2}$, with N the number of electrodes.

This vector must be filled in the following sequence: (1,2), (1,3), ... (1, N), (2,3), (2,4), ...

Here (i, j) refers to the situation that electrode i has potential 1, and the capacity is measured at electrode j .

Default value: 3

reaction_vector = s defines the sequence number of the reaction force vector. In Section 3.2.20 it is explained why a reaction force vector is required.

Default value: 2

epsilon_vector = s defines the sequence number of the permittivity vector. During the computation this vector is used for the coefficients of the problem. At the end of the computation it contains the computed permittivities.

Default value: 4

element_group = i defines the element group for which one wants to compute the permittivities. Sometimes the region is surrounded by another region for which the permittivity is already known.

Default value: 1

method = m defines the type of solution method used to solve the inverse problem. The following values for m are available

`capacity_simple`

Meaning of these subkeywords

capacity_simple If this keyword is found the measured values are supposed to be a vector of capacities and one wants to compute the permittivities.

The method that is used is the following one:
Define the sensitivity matrix \mathbf{S} of size $L \times NELEM$ by

$$S_{ij} = \frac{A_{max}}{A_j} \frac{C_i^{ielem} - C_i^0}{C_i^1 - C_i^0}. \quad (3.2.20.1)$$

L is the number of measurements: $L = \frac{N(N-1)}{2}$,

$NELEM$ is the number of elements,

A_j is the area of element j ,

A_{max} is the area of element with maximal area,

i refers to the component of the capacity vector,

\mathbf{C}^0 is the reference capacity vector computed by setting the permittivity vector completely equal to ε_{ref}

\mathbf{C}^1 is the capacity vector computed by setting the permittivity vector completely equal to $\varepsilon_{ref} + \delta\varepsilon$, and

\mathbf{C}^{ielem} is the capacity vector computed by setting the permittivity vector completely equal to ε_{ref} , except in element $ielem$ ($=j$) where it gets the value $\varepsilon_{ref} + \delta\varepsilon$.

Hence the sensitivity matrix consists of a set of finite differences.

Now the new value of the permittivities is computed by solving:

$$(\mathbf{S}^T \mathbf{S} + \mu \mathbf{I}) \varepsilon = \mathbf{S}^T \frac{\mathbf{C}^m - \mathbf{C}^0}{\mathbf{C}^1 - \mathbf{C}^0}. \quad (3.2.20.2)$$

The last division must be considered component-wise.

\mathbf{I} is the unity matrix,

\mathbf{C}^m is the vector of measured capacities and

μ is a regularization parameter.

The matrix $\mathbf{S}^T \mathbf{S}$ has dimension $NELEM \times NELEM$ but has rank L . So without the regularization matrix this system of equations is singular. The matrix $\mathbf{S}^T \mathbf{S} + \mu \mathbf{I}$ is not constructed explicitly but instead the so-called Sherman-Morrison-Woodbury formula: (See Gene H. Golub and Charles F. van Loan; Matrix Computations, page 3) is used, which requires only a full matrix of size $L \times L$ to be inverted.

Default value: capacity_simple

regular_parm = μ defines the regularization parameter μ defined in the method.

Default value: 0.01

delta_eps = \mathbf{e} defines the parameter $\delta\varepsilon$ defined in the method.

Default value: 0.1

eps_ref = \mathbf{e} defines the parameter ε_{ref} defined in the method.

Default value: 1

For an example of the computation of INVERSE_PROBLEM the reader is referred to Section [6.2.11](#)

3.2.21 The main keyword **REFINE**

The block defined by the main keyword **REFINE** defines the various options in case of refinement of the mesh.

Some of the options may also be given in the structure block. However, if an option is given in the structure block, as well as in the **REFINE** input block, these options must be the same.

The block defined by the main keyword **REFINE** has the following structure (options are indicated between the square brackets "[" and "] "):

```
REFINE [ ,SEQUENCE_NUMBER = s]
  TIMES = n
  MESH_IN = m1
  MESH_OUT = m2
  LOCAL_REFINEMENT
  TYPE_CRITERION = t
  ISEQ_REFARRAY = i
END
```

The keywords **REFINE** and **END** are mandatory all others are optional.

REFINE opens the input for the refine input block. The sequence number *s* may be used to distinguish between various input blocks with respect to **REFINE**.

END closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords **REFINE** and **END** however, must be placed at a new record.

Meaning of the subkeywords:

TIMES = *n* indicates that the mesh must be refined *n* times. Each refinement implies doubling of the number of points in each direction.
Default value 1.

MESH_IN = *m1* *m1* defines the sequence number of the mesh to be refined. This number must be between 1 and 5.
Default value 1.

MESH_OUT = *m2* *m2* defines the sequence number of the refined mesh. This number must be between 1 and 5.
m1 and *m2* may be equal, in which case the information about the coarse mesh is lost.
Default value 1.

LOCAL_REFINEMENT If `local_refinement` is found, the mesh is not refined globally, but only locally. This is only possible if some type of criterion for mesh refinement is available. This type of refinement is defined by the keyword `type_criterion`
Default value: global refinement

TYPE_CRITERION = *t* defines the criterion under which the elements are refined. This option is only used in case of local refinement.
The following options for the parameter *t* are available:

MARKED_ELEMENTS

Meaning of these parameters

MARKED_ELEMENTS The user has to supply a vector defined per element with one degree of freedom per element.

Although the entries in this vector must be real-valued (double precision) the actual value should be 0 or 1. If zero the element is not refined, if > 0 , it must be refined.

The array that is used to store these values is referred to in `iseq_refarray`. The user is himself responsible for the creation of this array.

In case of the enthalpy method for phase change problems, this vector can be created by a call to the derivatives subroutine, using `icheld` equal to 45.

Default value `MARKED_ELEMENTS`.

ISEQ_REFARRAY = *i* defines the sequence number of the vector in which the criterion for element refinement is stored, as described in `MARKED_ELEMENTS`.

So a typical call would be `iseq_refarray = %Refine`, where `Refine` is a vector name defined before. Of course this vector must have been filled before.

Default value 1.

3.2.22 The main keyword Navier_Stokes

The block defined by the main keyword Navier_Stokes defines the various options to solve the time dependent Navier-Stokes equations.

For a description of these methods, the reader is referred to the manual standard problems Section 7.1.10.

The block defined by the main keyword has the following structure NAVIER_STOKES (options are indicated between the square brackets "[" and "] "):

```
NAVIER_STOKES [ SEQUENCE_NUMBER = s]
    METHOD = m
    SEQ_TIME_INTEGRATION
    SEQ_VELOCITY
    SEQ_PRESSURE_CORRECTION
    NUMBER_OF_SUBSTEPS
END
```

The keywords NAVIER_STOKES and END are mandatory all others are optional.

NAVIER_STOKES opens the input for the Navier-Stokes input block. The sequence number *s* may be used to distinguish between various input blocks with respect to NAVIER_STOKES.

END closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords NAVIER_STOKES and END however, must be placed at a new record.

Meaning of the subkeywords:

METHOD = *m* Defines how the time-dependent Navier-Stokes equations must be solved.

Possible values for *m* are

```
standard
pressure_correction
convection_substepping
```

These subkeywords have the following meaning

standard The time-dependent Navier-Stokes equations are solved by a standard implicit time-stepping technique. Elements that are used to solve the stationary Navier-Stokes equations are combined with standard time-dependent discretization techniques.

pressure_correction The Navier-Stokes equations are solved by the *pressure – correction* technique.

For a description of the pressure correction method, the reader is referred to the manual standard problems Section 7.1.10.

Using this method also means that you have to add an input block PRESSURE_CORRECTION (3.2.23).

All other keywords, except of course SEQ_PRESSURE_CORRECTION, in the input block NAVIER_STOKES are skipped in case of pressure correction.

convection_substepping The time-dependent Navier-Stokes equations are solved by a operator splitting technique as described in Section 7.1.10 of the manual standard problems. The equations are solved by an implicit (Euler) technique, but the convective terms are treated in a special way. For the convection we use explicit substeps, where more substeps per time step are possible to satisfy stability requirements.

So the method consists of a number of explicit substeps for the convection only, followed

by an overall implicit Stokes step.

Consequence of this approach is that we need two input blocks for the coefficients: one for the incompressible Stokes equations and one for the explicit convection part. See manual Standard Problems Section 7.1.10.

The sequence number for the input block of the coefficients for the Stokes part must be given in the input block `time_integration` (3.2.15). The sequence number for the coefficients for the convective part must be one larger than that of the Stokes block. So we always need two consecutive sequence numbers for the coefficients input blocks.

The sequence number for the linear solver must also be given in `time_integration` block.

Default value: standard.

SEQ_TIME_INTEGRATION = s defines the sequence number of the time integration input block that should be used to define parameters with respect to time integration.

This parameter is not used in case of pressure correction.

Default value 1.

SEQ_VELOCITY = s Defines the sequence number s of the velocity vector in the set of solution vectors.

This parameter is not used in case of pressure correction.

Default value 1.

SEQ_PRESSURE_CORRECTION = s defines the sequence number of the input for the pressure correction block.

This parameter is only used in case of pressure correction.

Default value 1.

NUMBER_OF_SUBSTEPS = n defines the number of substeps performed in the operator splitting method.

It concerns the number of substeps for the explicit integration of the convection.

This parameter is only used in case of substepping.

Default value 1.

3.2.23 The main keyword `PRESSURE_CORRECTION`

The block defined by the main keyword `PRESSURE_CORRECTION` defines the various options in case of pressure correction to solve the Navier-Stokes equations.

For a description of the pressure correction method, the reader is referred to the manual standard problems Section 7.1.10.

The block defined by the main keyword has the following structure `PRESSURE_CORRECTION` (options are indicated between the square brackets "[" and "]"):

```

PRESSURE_CORRECTION [ , SEQUENCE_NUMBER = s]
  SEQ_VELOCITY = s
  SEQ_PRESSURE = s
  SEQ_VEL_COEFFICIENTS = s
  SEQ_PRESS_COEFFICIENTS = s
  SEQ_TIME_INTEGRATION = s
  SEQ_VEL_SOLVER = s
  SEQ_PRESS_SOLVER = s
  SEQ_CONV_COEFFICIENTS = s
  SEQ_PRESS_BOUNCOND = s
  NUMBER_OF_SUBSTEPS = n
  CONVECTION_TREATMENT = m
END

```

The keywords `PRESSURE_CORRECTION` and `END` are mandatory all others are optional.

`PRESSURE_CORRECTION` opens the input for the pressure correction input block. The sequence number *s* may be used to distinguish between various input blocks with respect to `PRESSURE_CORRECTION`.

`END` closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords `PRESSURE_CORRECTION` and `END` however, must be placed at a new record.

Meaning of the subkeywords:

`SEQ_VELOCITY = s` Defines the sequence number of the velocity vector in the set of solution vectors.

So a natural choice would be `seq_velocity = %velocity`

Default value: If the vector with name `velocity` exists: `%velocity` otherwise 1.

`SEQ_PRESSURE` Defines the sequence number of the pressure vector in the set of solution vectors.

So a natural choice would be `seq_pressure = %pressure`

Default value: If the vector with name `pressure` exists: `%pressure` otherwise 2.

`SEQ_VEL_COEFFICIENTS = s` Defines from which input block coefficients the coefficients for the velocity equation must be read.

Default value 1.

`SEQ_PRESS_COEFFICIENTS = s` Defines from which input block coefficients the coefficients for the pressure equation must be read.

Default value 3.

SEQ_TIME_INTEGRATION = s Defines from which input block TIME_INTEGRATION the input for the time integration must be read.

Default value 1.

SEQ_VEL_SOLVER = s Defines from which input block SOLVE the input for the linear solver with respect to the velocity equations must be read.

Default value 1.

SEQ_PRESS_SOLVER = s Defines from which input block SOLVE the input for the linear solver with respect to the pressure equations must be read.

Default value SEQ_PRESS_SOLVER.

CONVECTION_TREATMENT = m defines how the convection terms are treated.

The parameter m may have one of the following values:

standard
substeps

These subkeywords have the following meaning:

standard the standard method is applied, which means that the convective terms are treated in the same way as the viscous part.

substeps implies that an operator splitting is applied. The convection is treated in an explicit way, using smaller substeps (if necessary) and after that the viscosity and incompressibility are used in an implicit step. This is the same method as treated in Section (3.2.22) when method = convection_substepping.

So here we have the combination of substepping and pressure correction.

Default value standard.

SEQ_CONV_COEFFICIENTS = s defines the coefficients input sequence number for the convective part.

In the same way as in Section (3.2.22) we need an input block for the coefficients for the Stokes equations (here with number `seq_vel_coefficients`) and for the convection part.

Default value 2.

SEQ_PRESS_BOUNCOND = s defines the sequence number of the input block for the pressure boundary conditions.

Default value 0.

NUMBER_OF_SUBSTEPS = n defines the number of substeps performed in the operator splitting method.

It concerns the number of substeps for the explicit integration of the convection.

This parameter is only used in case of substepping.

Default value 1.

3.2.24 The main keyword BEARING

The block defined by the main keyword **bearing** defines the various options to solve the incompressible Reynolds equations for bearings.

For a description of these methods, the reader is referred to the manual standard problems Section 4.1.

At this moment the block is only meant for Kumars mass conservation iteration scheme. In case this block is used, it is necessary that the parameter **IBCMAT** in input block is equal to 1, so method = 20x, with x the storage method used.

Furthermore the problem block must be extended with the essential boundary condition **cavitation** = 1. In this way all nodes with pressures below the cavity pressure are treated as prescribed with value equal to the cavitation pressure.

The block defined by the main keyword has the following structure **BEARING** (options are indicated between the square brackets "[" and "]"):

```
BEARING [ SEQUENCE_NUMBER = s]
  METHOD = m
  SEQ_COEFFICIENTS = i
  SEQ_PRESSURE = i
  SEQ_REACTION_FORCE = i
  SEQ_SOLVE = i
  MAX_ITER = i
  PROBLEM_NUMBER = i
  PRINT_LEVEL = i
  P_CAVITY = p
END
```

The keywords and **END** are mandatory all others are optional.

BEARING opens the input for the bearing input block. The sequence number *s* may be used to distinguish between various input blocks with respect to **BEARING**.

END closes the input.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords **BEARING** and **END** however, must be placed at a new record.

Meaning of the subkeywords:

METHOD = *m* Defines how the Reynolds equations must be solved.

Possible values for *m* are

kumar

These subkeywords have the following meaning

kumar the mass conservation scheme of Kumar is applied for one stationary step. See the manual standard problems Section 4.1.

Default value: **kumar**.

SEQ_COEFFICIENTS = *i* Defines the sequence number of the coefficients input block.

This block defines the coefficients for the Reynolds equation as described in Section 4.1 of the manual "standard problems".

Default value: 1.

SEQ_PRESSURE = i defines the sequence number of the pressure in the set of solution vectors.
Default value: if the vector with name `pressure` is defined: `%pressure`, otherwise 1.

SEQ_REACTION_FORCE = i defines the sequence number of the reaction force in the set of solution vectors.
Default value: if the vector with name `reac_force` is defined: `%reac_force`, otherwise 2.

SEQ_SOLVE = i Defines the sequence number of the input block for the linear solver.
Default value: 1.

MAX_ITER = i gives the maximum number of iterations allowed.
The process stops if the maximum number of iterations is reached, or if no nodes are swapped from one stage to the other one, as described in Kumars work.
If at the end of the iteration process still nodes are swapped, a warning is printed, but the process continues.
Default value: 10.

PROBLEM_NUMBER = i defines the problem number with respect to the Reynolds equation.
Default value: 1.

PRINT_LEVEL = i defines the amount of output. The larger the value of i the more output is printed. If i has some value, also all output corresponding to values less than i are printed
Possible values

- 0 No extra output is printed
- 1 For each iteration the number of nodes swapped from stage to the other is printed
- 2 In each iteration the pressure vector is printed.
- 3 In each iteration the reaction force vector is printed.

Default value: 0.

P_CAVITY = p defines the cavity pressure.
Default value: 0.

3.3 Description of some function subroutines to be used

As already described in the INTRODUCTION it is possible to define coefficients, boundary conditions and so on as functions of space and possibly time. In this manual also the possibility of vectors constructed as function of other vectors has been introduced.

For all these possibilities the user must provide SEPRAN with some (function) subroutines. Some of these subroutines have already been described in the INTRODUCTION. It concerns the subroutines FUNCBC (5.5.1), CFUNCB (5.5.2), FUNCCE (5.5.3), FUNC and CFUNC (5.5.4).

In this chapter some extra subroutines will be defined.

- 3.3.1** describes the subroutines FUNALG and FUNALC. These subroutines define a new component of a vector to be created as function of the components of two old vectors.
- 3.3.2** treats function subroutine FUNCSCAL. This subroutine defines a scalar as function of already computed scalars.
- 3.3.3** deals with subroutine FUNCFL to fill a vector of corresponding to all nodes as function of the co-ordinates in an array USER.
- 3.3.4** describes the subroutines FUNC1B and CFUN1B. These subroutines are used to fill a vector element wise.
- 3.3.5** treats the subroutines FUNCOL and CFUNOL, which may be used to fill a vector as function of the co-ordinates and the previous value of the vector.
- 3.3.6** deals with the function subroutines FUNCC1 and FUNCC3, which are used to compute coefficients as function of previous computed solutions and otherwise created vectors.
- 3.3.7** gives a description of the function subroutine FUNCTR that may be used in combination with local transforms.
- 3.3.8** has subroutine USERBOOL as subject. This subroutine is used in combination with the while construction in the input block STRUCTURE.
- 3.3.9** treats subroutine FUNCCR, which may be used to adapt the boundary in a free surface or moving surface problem.
- 3.3.10** describes subroutine FUNCC2, which is a special subroutine to define coefficients that depend on the gradient of the solution.
- 3.3.11** describes subroutine FUNCVECT, which is a special subroutine to define a vector as function of the co-ordinates and a number of predefined vectors.
- 3.3.12** describes subroutines GETINT, GETCONST and GETVAR to extract values of constants and variables as defined in the CONSTANTS input block.
- 3.3.13** describes subroutines PUTINT, PUTREAL and PUTVAR to put values of constants and variables as defined in the CONSTANTS input block.
- 3.3.14** describes subroutines GETNAMEINT, GETNAMEREAL, GETNAMEVAR and PRGETNAME that may be utilized by the user to get the position of constants or variables in the common block cuscons by using their names.
- 3.3.15** describes subroutine FUNCSOLCR, which may be used to adapt the boundary in a free surface or moving surface problem.

3.3.1 Subroutines FUNALG and FUNALC

Description

The subroutines FUNALG and FUNALC are user written subroutines that must be provided if the option COMPUTE_VECTOR i, FUNC VECTOR j1 VECTOR j2 is used in the input block "STRUCTURE". FUNALG is used if the vectors j1 and j2 are both real, FUNALC if they are both complex. A function of a real and a complex vector is not yet available in SEPRAN.

The resulting vector is always of the same type as the input vectors.

FUNALG and FUNALC must be written by the user.

Call

```
CALL FUNALG ( VALUE1, VALUE2, VALUE3 )      (real case)
      or
CALL FUNALC ( VALUE1, VALUE2, VALUE3 )      (complex case)
```

Parameters

DOUBLE PRECISION VALUE1, VALUE2, VALUE3 (in case of FUNALG) or

DOUBLE COMPLEX VALUE1, VALUE2, VALUE3 (in case of FUNALC)

VALUE1 One component of the first vector (Vj1) to be manipulated. VALUE1 has got a value and may not be changed by the user.

VALUE2 Corresponding component of the second vector (Vj2) to be manipulated. VALUE2 has got a value and may not be changed by the user.

VALUE3 Corresponding component of the resulting vector (Vi) of the manipulation. VALUE3 must be given a value by the user.

Input

VALUE1 and VALUE2 have been given a value by SEPCOMP.

Output

VALUE3 must have a value

Remark

The subroutines FUNALG or FUNALC are called for each node in the mesh.

Example

Suppose that V3 must be created as function of V1 and V2 in the following way:

$$V3_i = \sin(V1_i)\cos(V2_i), \quad (3.3.1.1)$$

where i denotes the i^{th} component of each vector. The in the input block "STRUCTURE" the command

```
COMPUTE_VECTOR 3, FUNC VECTOR 1 VECTOR 2
```

must be given and a subroutine FUNALG of the following shape must be provided by the user:

```
SUBROUTINE FUNALG ( VALUE1, VALUE2, VALUE3 )
  IMPLICIT NONE
  DOUBLE PRECISION VALUE1, VALUE2, VALUE3
```

```
value3 = sin(value1) * cos(value2)
```

```
END
```


3.3.2 Function subroutine FUNCSCAL

Description

The function subroutine FUNCSCAL is a user written subroutine that must be provided if the option SCALAR j, FUNC = k is used in the input block "STRUCTURE". It is used to construct a scalar as function of previously computed scalars.

Heading

```
function funcscal ( k, arscalars )
```

Parameters

DOUBLE PRECISION FUNCSCAL, ARSCALARS(*)

INTEGER K

ARSCALARS In this array all scalars are stored that are defined in the part created by the input block "STRUCTURE". The scalars are stored in the sequence defined by the user, which means that S1 is stored in ARSCALARS(1), S2 in ARSCALARS(2) etcetera. The user must know himself which scalars are filled and which not.

The user is free to include the common block CUSCONS as described in Section 1.6. However, some care is needed since array ARSCALARS is identical to array SCALARS in CUSCONS, and a change in one of them results in an immediate change in the other one.

A disadvantage of the use of this array is the use of fixed positions. Hence if one adds a new scalar in the input block before one of the scalars used, the contents change. To prevent this it is better to use `getvar` as described in Section 3.3.12.3.

K This parameter may be used to distinguish between various cases. K is identical to the parameter k given in FUNC = k. K has got a value by program SEPCOMP.

FUNCSCAL Result of the computation. The user must give FUNCSCAL a value as function of the other scalars.

Input

K has been given a value by SEPCOMP.

Array ARSCALARS has been filled by program SEPCOMP, at least for those values that have been explicitly computed by the user in the part STRUCTURE before the call SCALAR j, FUNC = k.

Output

FUNCSCAL must have a value

Example Suppose that S3 must be created as function of S1 and S2 in the following way:

$$S3 = \sin(S1)\cos(S2). \quad (3.3.2.1)$$

In the input block "STRUCTURE" the command

SCALAR 3, FUNC = 1 must be given and a function subroutine FUNCSCAL of the following shape must be provided by the user:

```
FUNCTION FUNCSCAL ( K, ARSCALARS )
  IMPLICIT NONE
  DOUBLE PRECISION FUNCSCAL ARSCALARS(*)
  INTEGER K

  if ( k.eq.1 ) then

!    --- Compute the result for k = 1

      funcscal = sin(arscalars(1)) * cos(arscalars(2))

  else

!    --- Other values of k

      :
      :

  end if

END
```

A better approach is to use `getvar`. Suppose that S1 and S2 have the names S1 and S2. Then an alternative is:

```
FUNCTION FUNCSCAL ( K, ARSCALARS )
  IMPLICIT NONE
  DOUBLE PRECISION FUNCSCAL ARSCALARS(*)
  INTEGER K
  DOUBLE PRECISION GETVAR, S1, S2

  if ( k.eq.1 ) then

!    --- Compute the result for k = 1

      S1 = getvar ( 's1' )
      S2 = getvar ( 's2' )
      funcscal = sin(S1) * cos(S2)

  else

!    --- Other values of k

      :
      :

  end if

END
```

3.3.3 Subroutine FUNCFL

Description

The subroutine FUNCFL is a user written subroutine that must be provided if the option COEF $i = (\text{POINTS}, \text{ref}=r)$ with $r < 0$ is used in the input block "COEFFICIENTS". It is used to construct an array of length NPOINT, where NPOINT is the number of nodal points, with a function that depends on the space co-ordinates only. This array is stored in an array USER that is transported to the element subroutines and implicitly defines coefficients for these element subroutines.

Heading

```
subroutine funcfl ( coor, npoint, ichois, user )
```

Parameters

INTEGER NPOINT, ICHOIS

DOUBLE PRECISION COOR($ndim, NPOINT$), USER(NPOINT)

COOR In this two-dimensional array of size $ndim \times NPOINT$, the co-ordinates of the nodal points are stored. $ndim$ denotes the dimension of the space and should be a number not a parameter.

The i^{th} co-ordinate in point j is stored in COOR (i, j).

Array COOR has been filled by program SEPCOMP.

NPOINT Number of points in the mesh.

ICHOIS Choice parameter that may be used by the user, for example to distinguish between several coefficients. ICHOIS is equal to $-r$ in the command COEF $i = (\text{POINTS}, \text{ref}=r)$.

USER Array to be filled by the user from position 1. The user must fill exactly NPOINT values corresponding to the NPOINT nodes.

Input

NPOINT and ICHOIS have been given a value by SEPCOMP.

Array COOR has been filled by program SEPCOMP.

Output

Array USER must have been filled by the user.

Example

Suppose that the coefficient to be filled is defined in the following way:

$$COEFF(i) = \sin(x_1)\cos(x_2), \quad (3.3.3.1)$$

and suppose furthermore that the dimension of space is 2.

If the command COEF $i = (\text{POINTS}, \text{ref}=-1)$ has been given by the user, the following code may be used to fill USER.

```
SUBROUTINE FUNCFL ( COOR, NPOINT, ICHOIS, USER )
  IMPLICIT NONE
  INTEGER ndim
  PARAMETER ( ndim = 2 )
  INTEGER NPOINT, ICHOIS
  DOUBLE PRECISION COOR(ndim,NPOINT), USER(NPOINT)

  if ( ichois==1 ) then

!    --- Compute the vector for ichois = 1
      user(1:npoint) = sin(coor(1,1:npoint)) * cos(coor(2,1:npoint))

  else

!    --- Other values of ichois
      .
      .

  end if

END
```

3.3.4 Subroutines FUNC1B and CFUN1B

Description

The subroutines FUNC1B and CFUN1B are user written subroutines that must be provided if the option SPECIAL FUNCTION = l is used in the input block "CREATE" or "ESSENTIAL BOUNDARY CONDITIONS". With these subroutines the user may fill function values for one surface element in the solution vector.

FUNC1B and CFUN1B must be written by the user. FUNC1B is meant for real vectors and CFUN1B for complex vectors.

Call

```
CALL FUNC1B ( ICHOIS, INDEX1, INDEX2, USOL, COOR )
           or
CALL CFUN1B ( ICHOIS, INDEX1, INDEX2, USOL, COOR )
```

Parameters

INTEGER ICHOIS, INDEX1(*), INDEX2(*)

DOUBLE PRECISION COOR(*ndim*,*)

DOUBLE PRECISION USOL(*) (subroutine FUNC1B) or **DOUBLE COMPLEX** USOL(*) (subroutine CFUN1B)]

ICHOIS Choice parameter that may be used by the user, for example to distinguish between several coefficients. ICHOIS is equal to l in the command SPECIAL FUNCTION = l .

INDEX1 Array of length INPELM (number of nodal points in the element), containing the nodal point numbers of the surface element. Array INDEX1 may be used to find the co-ordinates of the nodes in the element.

INDEX2 Array of length ICOUNT (number of degrees of freedom in the element), containing the positions of the degrees of freedom in the solution vector USOL.

USOL Real or complex solution vector. The user must fill array USOL for the indicated degrees of freedom in the surface element. Array INDEX2 may be used to find the degrees of freedom in USOL, corresponding to the element. The i^{th} local degree of freedom in the element can be found from USOL(INDEX2(i)). A common way to extract the old solution in the nodal points of the element is to define a help array U of size *icount*. The following piece of code copies the old solution from array uold into array U:

```
u(1:icount) = usol(index2(1:icount))
```

COOR In this two-dimensional array of size *ndim* × NPOINT, the co-ordinates of the nodal points are stored. *ndim* denotes the dimension of the space and should be a number not a parameter. NPOINT denotes the number of nodal points. It is not necessary to know this number. The i^{th} co-ordinate in point j is stored in COOR (i, j).

Array COOR has been filled by program SEPCOMP. To find the co-ordinates of the nodes of the element, array INDEX1 must be used. The x -co-ordinate of the i^{th} local point in the element is given by COOR(1,INDEX1(i)), the y -co-ordinate by COOR(2,INDEX1(i)). A common way to extract the co-ordinates of the element is to define a help array X of size *ndim* × *npelm*, where *npelm* denotes the maximum number of nodes in the elements. The following piece of code copies the co-ordinates from array coor into array X:

```
x(1:ndim,1:inpelem) = coor(1:ndim,index1(1:inpelem))
```

Input

ICHOIS has got a value by program SEPCOMP.
 The arrays INDEX1 and INDEX2 have been filled for each element.
 Array COOR has been filled.
 Array USOL may be partly filled.

Output

The user must fill the required degrees of freedom of the element in array USOL.

Extra interface

Besides the parameters in the parameter list, SEPCOMP has another possibility to communicate with the subroutines FUNC1B and CFUN1B. This possibility is provided by the common block CACTL.

```
integer IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT, NOTVC,  
        IRELEM, NUSOL, NELEM, NPOINT  
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,  
              NOTVC, IRELEM, NUSOL, NELEM, NPOINT \emp
```

The following parameters may be useful:

IELEM Element number.

ITYPE Problem definition number. This number is defined in the input block "PROBLEM" by $ELGRPi = (\text{type} = ni)$.

IELGRP Standard element sequence number. Boundary elements get standard sequence numbers: $NELGRP + 1, NELGRP + 2, \dots, NELGRP + NUMNATBND$, where $NELGRP$ is the number of element groups and $NUMNATBND$ the number of boundary element groups.

INPELM Number of nodal points in element.

ICOUNT Number of degrees of freedom in element.

IFIRST This parameter indicates if the element subroutine is called for the first time for the specific element group ($IFIRST=0$) or not ($IFIRST=1$). This parameter may be of help for experienced FORTRAN programmers in order to initialize parameters and even local arrays only once. Since FORTRAN 77 does not save local parameters it is necessary to use the "SAVE" statement if this option is utilized.

NELEM Number of elements with standard element sequence number $IELGRP$ in the mesh.

NPOINT Number of nodal points in the mesh.

NUSOL Number of degrees of freedom in the solution vector.

IRELEM Relative element number with respect to standard element sequence number $IELGRP$.

Subroutine FUNC1B must be programmed as follows:

```
SUBROUTINE FUNC1B ( ICHOIS, INDEX1, INDEX2, USOL, COOR )  
IMPLICIT NONE  
integer ndim  
parameter ( ndim=..)
```

```

DOUBLE PRECISION COOR(ndim,*), USOL(*)
INTEGER ICHOIS, INDEX1(*), INDEX2(*)
INTEGER IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
+   NOTVC, IRELEM, NUSOL, NELEM, NPOINT
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
+   NOTVC, IRELEM, NUSOL, NELEM, NPOINT
.
.
.   statements to fill USOL for surface element IELEM
.
.
END

```

Subroutine CFUN1B must be programmed as follows:

```

SUBROUTINE CFUN1B ( ICHOIS, INDEX1, INDEX2, USOL, COOR )
IMPLICIT NONE
integer ndim
parameter ( ndim=..)
INTEGER ICHOIS, INDEX1(*), INDEX2(*)
DOUBLE COMPLEX USOL(*)
DOUBLE PRECISION COOR(ndim,*)
INTEGER IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
+   NOTVC, IRELEM, NUSOL, NELEM, NPOINT
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
+   NOTVC, IRELEM, NUSOL, NELEM, NPOINT
.
.
.   statements to fill USOL for surface element IELEM
.
.
END

```

See subroutine ELEM (Section 4.2) for more details with respect to the filling of USOL.

3.3.5 Function subroutines FUNCOL and CFUNOL

Description

The subroutines FUNCOL and CFUNOL are user written subroutines that must be provided if the option OLD_FUNCTION = m is used in the input block "CREATE" or "ESSENTIAL BOUNDARY CONDITIONS". With these function subroutines the user may define a function that depends on the vector to be changed itself. So in fact it is assumed that the vector has been filled before and that a new value must be created as function of the previous one.

FUNCOL and CFUNOL must be written by the user. FUNCOL is meant for real vectors and CFUNOL for complex vectors.

Call

```
VALUE = FUNCOL ( ICHOIS, X, Y, Z, UOLD )  
      or  
VALUE = CFUNOL ( ICHOIS, X, Y, Z, UOLD )
```

Parameters

INTEGER ICHOIS

DOUBLE PRECISION FUNCOL, X, Y, Z

DOUBLE COMPLEX CFUNOL

DOUBLE PRECISION UOLD (in case of FUNCOL) or **DOUBLE COMPLEX** UOLD (in case of CFUNOL)

ICHOIS Choice parameter. This parameter enables the user to distinguish between several cases. ICHOIS gets the value m as given by OLD_FUNCTION = m . This value is filled by program SEPCOMP.

X, Y, Z X, Y and Z co-ordinates of the nodal point.
For each nodal point this subroutine is called.

FUNCOL FUNCOL should get the computed real value of the function in the nodal point.

CFUNOL CFUNOL should get the computed complex value of the function in the nodal point.

UOLD UOLD contains the value of the vector in the specified point at input, for the specified degree of freedom.

Input

Program SEPCOMP has given ICHOIS, UOLD, X, Y and Z a value depending on the dimension of the space.

Output

FUNCOL or CFUNOL must have been given a value by the user.

Function subroutine FUNCOL must be programmed as follows:

```
FUNCTION FUNCOL ( ICHOIS, X, Y, Z, UOLD )
  IMPLICIT NONE
  DOUBLE PRECISION X, Y, Z, UOLD, FUNCOL
  INTEGER ICHOIS
  .
  .
  .      statements to give alpha a value as function of the
  .      parameters
  .
  FUNCOL = alpha
  END
```

Function subroutine CFUNOL must be programmed as follows:

```
FUNCTION CFUNOL ( ICHOIS, X, Y, Z, UOLD )
  IMPLICIT NONE
  DOUBLE PRECISION X, Y, Z
  DOUBLE COMPLEX UOLD, CFUNOL
  INTEGER ICHOIS
  .
  .
  .      statements to give alpha a value as function of the
  .      parameters
  .
  CFUNOL = alpha
  END
```

3.3.6 Function subroutines FUNCC1 and FUNCC3

Description

With these function subroutine a function may be defined, for the creation of real coefficients. These function subroutines have exactly the same task as function subroutine FUNCCF, however they have extra parameters in order to allow for function subroutines that are functions of previously computed solutions and vectors.

Both FUNCC1 and FUNCC3 must be written by the user.

Function subroutine FUNCC1

Function subroutine FUNCC1 contains one extra parameter in comparison to FUNCCF.

Call

```
VALUE = FUNCC1 ( IFUNC, X, Y, Z, UOLD )
```

Parameters

DOUBLE PRECISION FUNCC1, X, Y, Z, UOLD(*)

INTEGER IFUNC

IFUNC Choice parameter. This parameter enables the user to distinguish between several cases. IFUNC is defined by the user in the input part COEFFICIENTS. Its value is equal to $i-1000$, where i is the parameter in $\text{FUNC}=i$.

X,Y,Z X, y and z-coordinates of the nodal point. For each nodal point this subroutine is called.

UOLD In this array of length of the number of degrees of freedom in the nodal point, the values of the solution at the previous time-step or iteration are stored. UOLD(1) contains the value of the first unknown in that point, UOLD(2) of the second one, and so on.

FUNCC1 FUNCC1 should get the computed value of the function in the nodal point.

Input

IFUNC, UOLD, X, Y, and Z have been filled by SEPCOMP depending on the dimension of the space.

Output

FUNCC1 must have a value

Example

Suppose that there is one degree of freedom per point (u), and suppose furthermore that for IFUNC = 1 ($k=1001$) the function $f(x,y,u) = xy * u$, and for IFUNC = 2 the function $f(x,y) = \sin(x) * u$ is required.

Then FUNCC1 can be programmed as follows:

```
FUNCTION FUNCC1 ( IFUNC, X, Y, Z, UOLD )
  IMPLICIT NONE
  DOUBLE PRECISION FUNCC1, X, Y, Z, UOLD(*)
  INTEGER IFUNC

  IF ( IFUNC==1 ) THEN

C    --- IFUNC = 1    F = X Y * U
```

```

      FUNCC1 = X * Y * UOLD(1)

      ELSE

C      --- IFUNC = 2    F = SIN ( X ) * U

      FUNCC1 = SIN(X) * UOLD(1)

      ENDIF
      END

```

Function subroutine FUNCC3

Function subroutine FUNCC3 contains some extra parameters compared to FUNCC1.

Call

VALUE = FUNCC3 (IFUNC, X, Y, Z, NUMOLD, MAXUNK, UOLD)

Parameters

INTEGER IFUNC, NUMOLD, MAXUNK

DOUBLE PRECISION FUNCC3, X, Y, Z, UOLD(NUMOLD,MAXUNK)

IFUNC Choice parameter. This parameter enables the user to distinguish between several cases.

IFUNC is defined by the user in the input part COEFFICIENTS. Its value is equal to $i-10000$, where i is the parameter in $\text{FUNC}=i$.

If $\text{SOL_FUNC} = k$ is used in the input part of coefficients, $\text{IFUNC} = k$.

X,Y,Z X, y and z-coordinates of the nodal point. For each nodal point this subroutine is called.

NUMOLD Dimension parameter for array UOLD. It indicates the number of vectors $V_1, V_2, \dots, V_{\text{NUMOLD}}$ stored in UOLD. These vectors are defined in the input part of SEPCOMP, either by defining a coupled problem or by using STRUCTURE and defining some vectors explicitly.

MAXUNK Dimension parameter for array UOLD. It indicates the maximum number of degrees of freedom stored in the vectors corresponding to UOLD.

UOLD In this two-dimensional array of length $\text{NUMOLD} \times \text{MAXUNK}$ positions, the values of all the vectors $V_1, V_2, \dots, V_{\text{NUMOLD}}$ in the nodal point are stored. UOLD(1,1) contains the first degree of freedom of V_1 in the node, UOLD(1,2) the second degree of freedom and so on. UOLD(2,1) corresponds to the first degree of freedom of V_2 in the node.

FUNCC3 FUNCC3 should get the computed value of the function in the nodal point.

Input

IFUNC, NUMOLD, MAXUNK UOLD, X, Y, and Z have been filled by SEPCOMP depending on the dimension of the space.

Output

FUNCC3 must have a value

Example

Suppose that V_1 contains two unknowns per point (u and v), that V_2 contains the derivatives $\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial v}{\partial x}$ and $\frac{\partial v}{\partial y}$ and that V_3 contains the unknown p . suppose furthermore that for $\text{IFUNC} = 1$ the function $f(x, y) = u^2 + v^2$, and for $\text{IFUNC} = 2$ the function $f(x, y) = p + \frac{\partial u}{\partial x}^2 + \frac{\partial v}{\partial y}^2$ is required. Then $\text{NUMOLD}=3$ and $\text{MAXUNK}=4$ and FUNCC3 can be programmed as follows:

```
FUNCTION FUNCC3 ( IFUNC, X, Y, Z, NUMOLD, MAXUNK, UOLD )
  IMPLICIT NONE
  INTEGER IFUNC, NUMOLD, MAXUNK
  DOUBLE PRECISION FUNCC1, X, Y, Z, UOLD(NUMOLD, MAXUNK)

  IF ( IFUNC==1 ) THEN

C   --- IFUNC = 1    F = U^2 + V^2

      FUNCC3 = UOLD(1,1)**2 + UOLD(1,2)**2

  ELSE

C   --- IFUNC = 2    F = p + (du/dx)^2 + (dv/dy)^2

      FUNCC3 = uold(3,1) + uold(2,1)**2 + UOLD(2,4)**2

  ENDIF
END
```

3.3.7 Function subroutine FUNC_{TR}

Description

The subroutine FUNC_{TR} is a user written subroutine that must be provided if the option MA_{TR}IXR or MA_{TR}IXV is given with local transformation in combination with FUNC =.

See the input block "PROBLEM", Section 3.2.2.

With this option the user defines the entries of the transformation matrices **R** and **V** as function of the co-ordinates.

Call

```
VALUE = FUNCTR ( ICHOICE_TRANS, X, Y, Z )
```

INTEGER ICHOICE_TRANS

DOUBLE PRECISION FUNC_{TR}, X, Y, Z

FUNC_{TR} The user must give FUNC_{TR} the value of the matrix element in the specific point.

ICHOICE_TRANS Choice parameter that is made equal to the parameter *i* in FUNC = *i*. This parameter may be used by the user to distinguish between various possibilities.

X, Y, Z co-ordinates of the point in which the function must be evaluated.

Input

The parameters ICHOICE_TRANS, X, Y and Z have been given a value by the calling subroutine.

Output

The user must have been given FUNC_{TR} a value.

Layout

The function subroutine FUNC_{TR} must have the following shape:

```
function functr ( ichois_trans, x, y, z )
integer ichois_trans
double precision functr, x, y, z )
if ( ichois_trans==1 ) then

    statements to give FUNCTR a value for choice 1

else if ( ichois_trans==2 ) then

    statements to give FUNCTR a value for choice 2

else ...
.
.
.

end
```

3.3.8 Function subroutine USERBOOL

Description

The subroutine USERBOOL is a user written subroutine that must be provided if the option `boolean_expr(k)` is used in the while loop in the input block STRUCTURE, Section 3.2.3.

With this option the user defines the value of a logical operator as function of the parameter k and some internal variables, for example the ones stored in common block CUSCONS (Section 1.6).

Call

```
VALUE = USERBOOL ( K )
```

LOGICAL USERBOOL

INTEGER K

USERBOOL The user must give USERBOOL give the value `.true.` or `.false.` as function of k and possibly some internal variables.

K Choice parameter that is made equal to the parameter k in `boolean_expr(k)`. This parameter may be used by the user to distinguish between various possibilities.

Input

The parameter K has been given a value by the calling subroutine.

Output

The user must have been given USERBOOL a value.

Layout

The function subroutine USERBOOL must have the following shape:

```
function userbool ( k )
logical userbool
integer k
if ( k==1 ) then

    statements to give USERBOOL a value for choice 1

else if ( k==2 ) then

    statements to give USERBOOL a value for choice 2

else ...
.
.
.
end
```

For an example of the use of USERBOOL, the reader is referred to Section 6.3.3.

3.3.9 Subroutine FUNCCR

Description

Subroutine FUNCCR is used to adapt the boundary in case of a moving boundary or free surface problem.

It is used in the case that the adaptation method in the input block `adapt_boundary` (3.4.4) is equal to `funccr` (i.e. the parameter `IADAPT` is equal to 2).

It must be written by the user and is needed to compute the new co-ordinates of the boundary to be adapted. FUNCCR is called for each node separately.

Heading

```
subroutine funccr ( coorol, coornw, uold, inodp, ilocal, anorm, n )
```

Parameters

DOUBLE PRECISION COOROL(2,*), COORNW(2,*), UOLD(*), ANORM(2,*)

INTEGER INODP, ILOCAL, N

COOROL Two-dimensional array of length $2 \times N$ containing the co-ordinates of the boundary. The co-ordinates are stored sequentially in the direction of the boundary.

COORNW In this two-dimensional array of length $2 \times N$ the user must store the co-ordinates of the new boundary in the same sequence as for COOROL.

UOLD In this array of length NUNKP the solution is stored for the *ILOCAL*th node at the boundary.

The solution is defined by the option `seq_vectors = (i1, i2, ...)` in the input blocks `stationary_free_boundary` (3.4.5) or `instationary_free_boundary` (3.4.6). Only the first vector in the row is used.

In case FUNCCR is activated by a user call to a SEPRAN subroutine as described in the Programmers Guide, USOL corresponds to the first vector in array ISOL.

NUNKP denotes the number of degrees of freedom in that node.

INODP Global number of the *ILOCAL*th node at the boundary.

ILOCAL Local nodal point number corresponding to node at the boundary. $1 \leq ILOCAL \leq N$.

ANORM In this two-dimensional array of length $2 \times N$ the components of the outward normal are stored in the same sequence as in COOROL.

N Number of points at the boundary.

Input

The arrays COOROL, UOLD and ANORM have been filled by program SEPFREE or the subroutine called by the user (CHANBN, ADAPBN, ...).

INODP, ILOCAL and N have got a value from program SEPFREE.

Output

Array COORNW positions (1,ILOCAL) and (2,ILOCAL) must have been filled by the user.

Layout

The subroutine FUNCCR must have the following shape:

```
subroutine funcr ( coorol, coornw, uold, inodp, ilocal, anorm, n )
implicit none
double precision coorol(2,*), coornw(2,*), uold(*), anorm(2,*)

xold = coorol(1,ilocal)
yold = coorol(2,ilocal)
xnorm = anorm(1,ilocal)
ynorm = anorm(2,ilocal)
u1 = uold(1)
u2 = uold(2)
.
.
.   statements to compute xnew and ynew
.
.
coornw(1,ilocal) = xnew
coornw(2,ilocal) = ynew

end
```


3.3.10 Subroutine FUNCC2

Description

Subroutine FUNCC2 is used to define a coefficient and possibly its derivative in case the coefficient depends on the gradient of the solution in each point. FUNCC2 is called for each integration point separately.

Call

```
CALL FUNCC2 ( ICHOICE, X, Y, Z, GRADC, G, DGDGRAD )
```

Parameters

DOUBLE PRECISION X, Y, Z, GRADC(*), G, DGDGRAD(*)

INTEGER ICHOICE

ICHOICE Parameter that may be used to distinguish between possibilities. ICHOICE has been given a value by the calling subroutine.

X, Y, Z Co-ordinates of the point in which the function g must be evaluated.

GRADC In this array of length NDIM, with NDIM the dimension of the space, the gradient of the old solution in that point is stored by the calling subroutine.

G This parameter must get the value of the function g in the particular point.

DGDGRAD Depending on the application this array of length NDIM, or possibly an array of length 1 (scalar) must be filled with the gradient of the function g , with respect to ∇c .

Input

Array GRADC has been filled by the calling subroutine.

ICHOICE, X, Y, and Z have been filled by the calling subroutine.

Output

G must have got a value.

Depending on the application DGDGRAD must have been filled by the user.

Layout

The subroutine FUNCC2 must have the following shape:

```
subroutine funcc2 ( icoice, x, y, z, gradc, g, dgdgrad )
implicit none
integer icoice
double precision x, y, z, gradc(*), g, dgdgrad(*)

      .
      .
      .  statements to compute g and possibly dgdgrad
      .
      .

end
```

3.3.11 Subroutine FUNCVECT

Description

Subroutine FUNCVECT is used to define a vector as function of the co-ordinates and a number of predefined vectors.

FUNCVECT is called if in the block CREATE of the block ESSENTIAL BOUNDARY CONDITIONS, the option

```
OLD_VECTOR = m
```

usually in combination with

```
SEQ_VECTORS = V1, V2, ...
```

is used.

Heading

```
subroutine funcvect( icoice, ndim, coor, numnodes, uold,
+                   nuold, result, nphys)
```

Parameters

INTEGER ICHOICE, NDIM, NUMNODES, NUOLD, NPHYS

DOUBLE PRECISION COOR(NDIM,NUMNODES), UOLD(NUMNODES,NPHYS,NUOLD),
RESULT(NUMNODES,*)

ICHOICE Parameter that may be used to distinguish between possibilities. ICHOICE is identical to the parameter m in `OLD_VECTOR = m`.

NDIM Dimension of the space.

NUMNODES Number of nodal points for which the vector must be filled. This number is implicitly defined by the user in the location part.

NUOLD Number of predefined vectors that are used to create the new vector. This number is implicitly defined by the user in `SEQ_VECTORS = V1, V2, ...`

NPHYS Maximal number of degrees of freedom that are available in the old vectors uold per point.

COOR in this array the co-ordinates of the nodes to be used are stored. COOR is a two-dimensional array of size $NDIM \times NUMNODES$.

$COOR(k, i)$ contains the x_k -th co-ordinate of the i -th point in the set.

UOLD contains the values of the previously defined vectors for all available degrees of freedom in the set of points. The user himself knows how much degrees of freedom per point are available for each vector.

$UOLD(i, j, k)$ refers to the j -th degree of freedom in the i -th point in the set with respect to the k -th vector.

RESULT must be filled by the user with the values of the vector. RESULT is a two-dimensional array of size $NUMNODES \times NDEG$, where NDEG is the number of degrees of freedom per point to be filled. The user himself knows how many degrees of freedom he has to fill. $RESULT(i, j)$ refers to the j -th degree of freedom in the i -th point in the set.

Input

The arrays COOR and UOLD have been filled before the call of FUNCVECT.
The parameters ICHOICE, NDIM, NUMNODES, NUOLD and NPHYS have been filled
by the calling subroutine.

Output

Array RESULT must be filled by the user for all nodes in the set.

Layout

The subroutine FUNCVECT must have the following shape:

```
subroutine funcvect( icoice, ndim, coor, numnodes, uold,
+                   nuold, result, nphys)

implicit none
integer icoice, ndim, numnodes, nuold, nphys
double precision coor(ndim,numnodes),
+ uold(numnodes,nphys,nuold), result(numnodes,*)

      .
      .
      .   statements to compute and fill array RESULT
      .
      .
      .

end
```

3.3.12 Function subroutines to get the values of constants and variables.

In this section we describe three function subroutines that may be utilized by the user to get the values of constants or variables from the common block constants by using their names. It concerns the following functions subroutines:

GETINT 3.3.12.1 Returns with value of integer.

GETCONST 3.3.12.2 Returns with value of constant. The result is always real.

GETVAR 3.3.12.3 Returns with value of variable.

3.3.12.1 Function subroutine GETINT

Description

With this function subroutine the user may extract the value of an integer such as it is given in the input file in the block CONSTANTS, sub-block integers. This function subroutine may be called in each user written subroutine.

Heading

```
function getint ( intname )
```

Parameters

INTEGER GETINT

CHARACTER (len = *) INTNAME

GETINT Contains the value of the integer with name INTNAME at output.

If INTNAME does not exist an error message is given.

For that reason it is not allowed to use subroutine GETINT in a print or write statement.

INTNAME Name of the integer of which the value must be returned.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter INTNAME must have a value.

Output

GETINT has got the value of the corresponding integer.

Layout

The use of GETINT in a subroutine is very simple:

```
subroutine xxxxxx

implicit none
integer getint, intfind...
```

```
      .  
      .  
      intfind = getint ( 'name_of_integer' )  
      .  
      .  
      .  
end
```

3.3.12.2 Function subroutine GETCONST

Description

With this function subroutine the user may extract the value of a constant such as it is given in the input file in the block CONSTANTS, sub-block integers or reals. The result is always real. This function subroutine may be called in each user written subroutine.

Heading

```
function getconst ( realname )
```

Parameters

DOUBLE PRECISION GETCONST

CHARACTER (len = *) REALNAME

GETCONST Contains the value of the constant with name realname at output.

If realname does not exist an error message is given.

For that reason it is not allowed to use subroutine GETCONST in a print or write statement.

REALNAME Name of the constant of which the value must be returned.

This constant may be both one of the integer block or the real block.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter REALNAME must have a value.

Output

GETCONST has got the value of the corresponding constant.

Layout

The use of GETCONST in a subroutine is very simple:

```
subroutine xxxxxx  
  
implicit none  
double precision getconst, realfind...  
      .  
      .
```

```
      .  
      realfind = getconst ( 'name_of_constant' )  
      .  
      .  
      .  
end
```

3.3.12.3 Function subroutine GETVAR

Description

With this function subroutine the user may extract the value of a variable such as it is given in the input file in the block CONSTANTS, sub-block variables. The result is always real. This function subroutine may be called in each user written subroutine.

Heading

```
function getvar ( VARNAME )
```

Parameters

DOUBLE PRECISION GETVAR

CHARACTER (len = *) VARNAME

GETVAR Contains the value of the variable with name varname at output.

If varname does not exist an error message is given.

For that reason it is not allowed to use subroutine GETVAR in a print or write statement.

VARNAME Name of the variable of which the value must be returned.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter VARNAME must have a value.

Output

GETVAR has got the value of the corresponding constant.

Layout

The use of GETVAR in a subroutine is very simple:

```
subroutine xxxxxx  
  
implicit none  
double precision getvar, realfind...  
      .  
      .  
      .  
      realfind = getvar ( 'name_of_constant' )  
      .  
      .  
      .  
end
```

3.3.13 Subroutines to put the values of constants and variables in common CUSCONS

In this section we describe three subroutines that may be utilized by the user to put the values of constants or variables into the common block `cuscons` by using their names. It concerns the following functions subroutines:

PUTINT 3.3.13.1 Puts the value of an integer in `cuscons`.

PUTREAL 3.3.13.2 Puts the value of a real in `cuscons`.

PUTVAR 3.3.13.3 Puts the value of a variable in `cuscons`.

Mark that the subroutines `PUTINT` and `PUTREAL` make only sense in subroutine `COMPCONS` (1.6), since that is the only place where constants may be changed.

3.3.13.1 Subroutine `PUTINT`

Description

With this subroutine the user may put the value of an integer such as it is given in the input file in the block `CONSTANTS`, sub-block integers into the common `CUSCONS`. The use of this subroutine makes only sense in subroutine `COMPCONS` (1.6). It is meant to put a changed value back in the block `CONSTANTS`.

Heading

```
subroutine putint ( intname, value )
```

Parameters

INTEGER VALUE

CHARACTER (len = *) INTNAME

VALUE Contains the value of the integer with name `INTNAME` that must be put into common `CUSCONS`.

If `INTNAME` does not exist an error message is given.

INTNAME Name of the integer of which the value must be substituted.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

The parameters `INTNAME` and `VALUE` must have a value.

Output

`VALUE` is substituted in common block `CUSCONS`, array `INCONS`.

Layout

The use of `PUTINT` in subroutine `COMPCONS` is very simple:


```
subroutine COMPCONS

implicit none
integer intval
.
.
.
intval = ....
call putint ( 'name_of_integer', intval )
.
.
.

end
```

3.3.13.2 Subroutine PUTREAL

Description

With this subroutine the user may put the value of a real such as it is given in the input file in the block CONSTANTS, sub-block reals into the common CUSCONS. The use of this subroutine makes only sense in subroutine COMPCONS (1.6). It is meant to put a changed value back in the block CONSTANTS.

Heading

```
subroutine putreal ( realname, value )
```

Parameters

INTEGER VALUE

CHARACTER (len = *) REALNAME

VALUE Contains the value of the real with name realname that must be put into common CUSCONS.

If realname does not exist an error message is given.

realname Name of the real of which the value must be substituted.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

The parameters REALNAME and VALUE must have a value.

Output

VALUE is substituted in common block CUSCONS, array RLCONS.

Layout

The use of PUTREAL in subroutine COMPCONS is very simple:

```
subroutine COMPCONS

implicit none
double precision value
      .
      .
      .
value = ....
call putreal ( 'name_of_real', value )
      .
      .
      .

end
```

3.3.13.3 Subroutine PUTVAR

Description

With this subroutine the user may put the value of a variable (scalar) such as it is given in the input file in the block VARIABLES, into the common CUSCONS. This subroutine may be called in each user written subroutine.

Heading

```
putvar ( varname, value )
```

Parameters

DOUBLE PRECISION VALUE

CHARACTER (len = *) VARNAME

VARNAME Contains the value of the variable with name varname that must be substituted.

If varname does not exist an error message is given.

VARNAME Name of the variable of which the value must be substituted.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

The parameter VARNAME and VALUE must have a value.

Output

VALUE is substituted in common block CUSCONS, array SCALARS.

Layout

The use of PUTVAR in a subroutine is very simple:

```
subroutine xxxxxx

implicit none
double precision value...
      .
      .
      .
value = ...
call putvar ( 'name_of_scalar', value )
      .
      .
      .

end
```

3.3.14 Subroutines to get the positions of variables, constants and vectors in common CUSCONS

In this section we describe three subroutines that may be utilized by the user to get the position of constants or variables in the common block cuscons by using their names.

Also a subroutine is described to find the sequence number of a solution array by giving its name. It concerns the following subroutines:

GETNAMEINT 3.3.14.1 returns with the position of an integer in array INCONS in common CUSCONS.

GETNAMEREAL 3.3.14.2. a real in array RLCONS in common CUSCONS.

GETNAMEVAR 3.3.14.3 returns with the position of a variable in array SCALARS in common CUSCONS.

PRGETNAME 3.3.14.4 gives the sequence number of a solution array with respect to the general solution array.

3.3.14.1 Function subroutine GETNAMEINT

Description

With this subroutine the user may get the position of an integer in array INCONS in common block CUSCONS. This subroutine may be called in each user written subroutine.

Heading

```
function getnameint ( intname )
```

Parameters

INTEGER GETNAMEINT

CHARACTER (len = *) INTNAME

GETNAMEINT Contains the position of the integer with name INTNAME at output.

INTNAME Name of the integer of which the position must be returned.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter INTNAME must have a value.

Output

GETNAMEINT has got the value of the corresponding integer position.

Layout

The use of GETNAMEINT in a subroutine is very simple:

```
subroutine xxxxxx

implicit none
integer getnameint, intpos, intval...
include 'SPcommon/comcons1'
include 'SPcommon/cuscons'
.
.
.
intpos = getnameint ( 'name_of_integer' )
intval = incons(intpos)
.
.
.

end
```

3.3.14.2 Function subroutine GETNAMEREAL

Description

With this subroutine the user may get the position of a real in array RLCONS in common block CUSCONS. This subroutine may be called in each user written subroutine.

Heading

```
function getnamereal ( realname )
```

Parameters

INTEGER GETNAMEREAL

CHARACTER (len = *) REALNAME

GETNAMEREAL Contains the position of the real with name REALNAME at output.

REALNAME Name of the real of which the position must be returned.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter REALNAME must have a value.

Output

GETNAMEREAL has got the value of the corresponding real position.

Layout

The use of GETNAMEREAL in a subroutine is very simple:

```
subroutine xxxxxx

implicit none
integer getnamereal, realpos...
double precision realval
include 'SPcommon/comcons1'
include 'SPcommon/cuscons'
.
.
.
realpos = getnamereal ( 'name_of_real' )
realval = rlcons(realpos)
.
.
.

end
```

3.3.14.3 Function subroutine GETNAMEVAR

Description

With this subroutine the user may get the position of a real in array SCALARS in common block CUSCONS. This subroutine may be called in each user written subroutine.

Heading

```
function getnamevar ( varname )
```

Parameters

INTEGER GETNAMEVAR

CHARACTER (len = *) VARNAME

GETNAMEVAR Contains the position of the real with name VARNAME at output.

VARNAME Name of the variable (scalar) of which the position must be returned.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter VARNAME must have a value.

Output

GETNAMEVAR has got the value of the corresponding scalar position.

Layout

The use of GETNAMEVAR in a subroutine is very simple:

```
subroutine xxxxxx

implicit none
integer getnamevar, varpos...
double precision value
include 'SPcommon/comcons1'
include 'SPcommon/cuscons'
.
.
.
varpos = getnamevar ( 'name_of_scalar' )
value = rlcons(varpos)
.
.
.

end
```

3.3.14.4 Subroutine PRGETNAME

Description

With this subroutine the user may get the position of a vector in array of solution arrays. In fact it returns the sequence number of the solution array with a specific name. This subroutine may be called in each user written subroutine.

Heading

```
subroutine prgetname ( vectorname, ivecnm )
```

Parameters

INTEGER IVECNM

CHARACTER (len = *) VECTORNAME

IVECNM Contains the position of the vector with name vectorname at output.

vectorname Name of the vector of which the position must be returned.

This name must be a string, which means that it is a text between quotes.

The name itself is case-insensitive, which means that the user is free to use capitals or lower-case letters, they are all considered the same.

Input

Parameter VECTORNAME must have a value.

Output

PRGETNAME has got the value of the corresponding vector position.

Layout

The use of PRGETNAME in a subroutine is very simple:

```
subroutine xxxxxx ( isol, ... )

implicit none
integer isol(5,*), ivecnm, ...
.
.
.
call prgetname ( 'name_of_vector', ivecnm )
call yyy ( isol(1,ivecnm), .... )
.
.
.

end
```


3.3.15 Subroutine FUNCSOLCR

Description

Subroutine FUNCSOLCR is used to adapt the boundary in case of a moving boundary or free surface problem.

It is used in the case that the adaptation method in the input block `adapt_boundary` (3.4.4) is equal to `funcsolcr` (i.e. the parameter `IADAPT` is equal to 9).

It must be written by the user and is needed to compute the new co-ordinates of the boundary to be adapted. FUNCSOLCR is called for the complete boundary.

Heading

```
subroutine funcsolcr ( coorol, coornw, uold, anorm, n, nphys, nuold )
```

Parameters

INTEGER N, NPHYS, NUOLD

DOUBLE PRECISION COOROL(2,*), COORNW(2,*), UOLD(N,NPHYS,NUOLD), ANORM(2,*)

COOROL Two-dimensional array of length $2 \times N$ containing the co-ordinates of the boundary. The co-ordinates are stored sequentially in the direction of the boundary.

COORNW In this two-dimensional array of length $2 \times N$ the user must store the co-ordinates of the new boundary in the same sequence as for COOROL.

UOLD In this array of size $N \times NPHYS \times NUOLD$, NUOLD solution vectors are stored.

The vector is stored as follows:

UOLD(i,j,k) refers to the i^{th} node along the boundary, and the j^{th} degree of freedom of k^{th} vector.

The solution is defined by the option `seq_vectors = (i1, i2, ...)` in the input blocks `stationary_free_boundary` (3.4.5) or `instationary_free_boundary` (3.4.6).

In case FUNCSOLCR is activated by a user call to a SEPRAN subroutine as described in the Programmers Guide, USOL corresponds to the first NUOLD vectors in array ISOL.

NUNKP denotes the number of degrees of freedom in that node.

NPHYS Maximum number of degrees of freedom per point in the vectors UOLD.

NUOLD Number of vectors stored in UOLD. This number is defined by `seq_vectors = (i1, i2, ...)`

ANORM In this two-dimensional array of length $2 \times N$ the components of the outward normal are stored in the same sequence as in COOROL.

N Number of points at the boundary.

Input

The arrays COOROL, UOLD and ANORM have been filled by program SEPFREE or the subroutine called by the user (CHANBN, ADAPBN, ...).

NPHYS, NUOLD and N have got a value from program SEPFREE.

Output

Array COORNW must have been filled by the user.

Layout

The subroutine FUNCSOLCR must have the following shape:

```
subroutine funcsolcr ( coorol, coornw, uold, anorm, numnodes,
+                    nphys, nuold )
implicit none
integer numnodes, nuold, nphys
double precision coorol(2,numnodes), uold(numnodes,nphys,nuold),
+                coornw(2,numnodes), anorm(2,numnodes)

integer ilocal
double precision xold, yold, xnorm, ynorm, u1, u2, xnew, ynew
do ilocal = 1, n

    xold = coorol(1,ilocal)
    yold = coorol(2,ilocal)
    xnorm = anorm(1,ilocal)
    ynorm = anorm(2,ilocal)
    u1 = uold(ilocal,1,1)
    u2 = uold(ilocal,2,1)
    .
    .
    .   statements to compute xnew and ynew
    .
    .
    coornw(1,ilocal) = xnew
    coornw(2,ilocal) = ynew

end do

end
```

3.4 Description of the input for program SEPFREE

3.4.1 Introduction

Program SEPFREE is meant for free surface and moving boundary problems. It has exactly the same possibilities as program SEPCOMP, however, there are some differences:

- SEPFREE may use a mesh that has been created before by program SEPMESH. However, it is also possible to create the mesh in SEPFREE itself. In that case SEPFREE reads all mesh input itself. No file `meshoutput` is created unless stated explicitly.
- SEPFREE is able to deal with free surface problems. Such problems are essentially non-linear and need some kind of iteration loop (in stationary cases) or alternatively some time-stepping algorithm in case of time-dependent problems. In each of these steps the boundary of the region and hence the mesh may be adapted to the newly computed solution. So besides the standard input as described for program SEPCOMP, SEPFREE needs some extra input blocks describing how the boundary and the mesh must be adapted. Besides that, extra information with respect to the iteration process or time-stepping algorithm is required.
- At this moment there is no default STRUCTURE for the program SEPFREE. As a consequence the user must always explicitly define the input block STRUCTURE describing which steps must be carried out.

The input for program SEPFREE is organized in exactly the same way as for program SEPCOMP. Besides the input blocks described in Section 3.2 for program SEPCOMP the following extra blocks may be used.

- MESH
- ADAPT_MESH
- ADAPT_BOUNDARY
- STATIONARY_FREE_BOUNDARY
- INSTATIONARY_FREE_BOUNDARY

The blocks MESH, PROBLEM, MATRIX and STRUCTURE must all be given. All other blocks are optional.

The sequence of the input blocks is always:

```
START (optional)
.
.
END
MESH (optional)
.
.
END
READ MESH (optional)
PROBLEM (mandatory)
.
.
END
```

rest of the blocks in arbitrary sequence

Except the blocks `START`, `MESH`, `PROBLEM` and `STRUCTURE` all blocks may be used more than once. The new main blocks have the following meaning:

MESH This block opens the input for the mesh description. This is exactly the same block as the complete input for program `SEPMESH`. At this moment free surface problems have been restricted to two-dimensional problems. Furthermore the options `refine` and `transform` may not be used in combination with free surface problems. See Section 2.2.

If the block `MESH` is not present before the block `PROBLEM` it is assumed that the mesh has been created before and that the information concerning the mesh is stored in the file `meshoutput`. All information from this file is read.

READ MESH may be used to indicate that a mesh must be read from the file `meshoutput`. Mark that this keyword is superfluous, since skipping the keyword `MESH` has exactly the same result.

STRUCTURE This block has already been described in Sections 3.2 and 3.2.3. However, in case of free surface problems some extra options are available which are described in Section 3.4.2.

ADAPT_MESH Describes how the mesh must be adapted during each step of the iteration process (stationary case) or time stepping process (time-dependent case). See Section 3.4.3.

ADAPT_BOUNDARY Describes how the boundary of the mesh must be adapted during each step of the iteration process (stationary case) or time stepping process (time-dependent case). This block itself is activated by `ADAPT_MESH`. See Section 3.4.4.

STATIONARY_FREE_BOUNDARY Gives information concerning the iteration process in case of a stationary free boundary problem. See Section 3.4.5.

INSTATIONARY_FREE_BOUNDARY Gives information concerning the time stepping algorithm in case of a time-dependent free boundary problem. See Section 3.4.6.

In the next subsections the input of each of the blocks is described.

3.4.2 Extra possibilities for the main keyword STRUCTURE

The block defined by the main keyword STRUCTURE has already been described in Section 3.2.3. However, with respect to free boundary problems STRUCTURE has been provided with some extra options that are described in this section. In fact it concerns extra commands that may be given in the block as well as an extra option for the command `change_coefficients`. The following two extra commands may be used for stationary free boundaries:

```
START_STATIONARY_FREE_BOUNDARY_LOOP, sequence_number = k
...
END_STATIONARY_FREE_BOUNDARY_LOOP
```

In case of instationary free boundaries at least three commands are needed:

```
START_INSTATIONARY_FREE_BOUNDARY_LOOP, sequence_number = k
....
TIME_INTEGRATION, sequence_number = s, vector = i
....
END_INSTATIONARY_FREE_BOUNDARY_LOOP
```

These commands have the following meaning:

START_STATIONARY_FREE_BOUNDARY_LOOP , sequence_number = k .

All commands that are given from this command until `end_stationary_free_boundary_loop` is found will be considered as part of the computation of a steady state free boundary. The sequence number k refers to the input block "STATIONARY_FREE_BOUNDARY". This block describes how the boundary and the mesh must be adapted, and under what conditions the process has been converged. See Section 3.4.5.

END_STATIONARY_FREE_BOUNDARY_LOOP Ends the loop for the computation of the free boundary.

START_INSTATIONARY_FREE_BOUNDARY_LOOP , sequence_number = k .

All commands that are given from this command until `end_instationary_free_boundary_loop` is found will be considered as part of the computation of a time-dependent free boundary. The sequence number k refers to the input block "INSTATIONARY_FREE_BOUNDARY". This block describes how the boundary and the mesh must be adapted. The updating itself is performed at the position of the `end_instationary_free_boundary`. See Section 3.4.6 for a description of the input block "INSTATIONARY_FREE_BOUNDARY".

TIME_INTEGRATION , sequence_number = s , vector = i , defines what time integration must be carried out in the computation of the free boundary. This statement may be preceded by some statements to make preparations for the time integration, and succeeded by statements manipulating the computed solution. Only one time-step is carried out. The computation takes place at the mesh generated in the old time step.

The sequence number s refers to the input block TIME_INTEGRATION, which defines the parameters of the time integration process. See Section 3.2.15 for a description. At this moment only one fixed time step may be used. So only one end time and one time step may be given. Furthermore the only available time integration at this moment is Euler implicit. If you need extra time integrations please contact SEPRA.

The options with respect to TOUT, like TOUTINIT, have at the present moment no effect at all.

Of course the options referring to the stationary accuracy make also no sense in this case.

The sequence number i defines the vector to be integrated in time.

END_INSTATIONARY_FREE_BOUNDARY_LOOP Ends the loop for the computation of the free boundary. In this step the free boundary is adapted and a new mesh is created according to the description in the input block "INSTATIONARY_FREE_BOUNDARY".

So in the case of a stationary free boundary a typical part of the input block STRUCTURE might be:

```
start_stationary_free_boundary_loop, sequence_number = 1
  solve_linear_system, seq_coef = 1, seq_solve = 1
  other statements corresponding to the loop
end_stationary_free_boundary_loop
```

In the program this implemented as:

```
niter := 1
ready := false
while ( not ready ) do
  solve_linear_system, seq_coef = 1, seq_solve = 1
  other statements corresponding to the loop
  if ( niter>1 ) then
    compute difference between old and new solution
    ready := difference < required accuracy
  end if
  if ( not ready ) then
    adapt boundary and mesh
    interpolate solution if necessary
    niter := niter+1
  end if
end while
```

This loop option, which may **not** be nested, gives the user the maximum flexibility to carry out whatever computation he wants within the loop.

Besides these extra commands the following option is available for the command `change_coefficients`:

```
iteration = i
```

iteration = *i* defines at which iteration in the loop defined by the commands

`start_stationary_free_boundary_loop` and `end_stationary_free_boundary_loop` the coefficients must be changed as described in Section [3.2.3](#).

In the case of a time-dependent free boundary problem, a typical part of the input STRUCTURE looks like:

```
start_instationary_free_boundary_loop, sequence_number = 1
  statements to make some preparations
  time_integration, sequence_number = 1, vector = 1
  other statements to update the solution
end_instationary_free_boundary_loop
```

In the program this is implemented as:

```
ready := false
while ( not ready ) do
  make preparations
  if ( first call ) then
    set t, t0, dt and tend
```

```
end if
t := t + dt
solve one time step
if ( t > tend ) then
  ready := true
end if
adapt boundary and mesh
interpolate solution if necessary
end while
```

3.4.3 The main keyword ADAPT_MESH

The block defined by the main keyword ADAPT_MESH defines how a new mesh must be generated in case of a free surface or moving boundary problem.

In each step of the iteration or in each time-step a solution is computed. This solution implicitly defines a new boundary for the free boundary problem, and hence the mesh must be adapted to this new boundary.

Adaptation of boundary and mesh is performed by the subroutines activated by the main keyword ADAPT_MESH. Adapting the boundary means that the mesh is changed but not necessarily the topology. Sometimes it is sufficient to update the co-ordinates of the mesh. This is much more cheaper than remeshing of the entire mesh. However, changing the co-ordinates only may result in very distorted elements, in which case remeshing must be applied. Whether or not the topology is adapted is the responsibility of the user.

The block defined by the main keyword ADAPT_MESH has the following structure (options are indicated between the square brackets "[" and "]"):

```
ADAPT_MESH [,SEQUENCE_NUMBER = s]
  (mandatory): opens the input for the adaptation of the mesh

  adapt_boundary = (i1, i2, i3,... )
  change_topology = c
  plot_mesh
  interpolate_solution = ( v1, v2, ... , vn )

END
(mandatory): end of input
```

The sequence number s may be used to distinguish between various input blocks with respect to the adaptation of meshes.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords ADAPT_MESH and END however, must be placed at a new record.

Meaning of the subkeywords:

adapt_boundary= ($i1, i2, i3, \dots$) defines how the boundary should be adapted. In fact the boundary is adapted by a series of calls to the subroutines corresponding to the main keyword adapt_boundary. These calls are carried out in the sequence of the sequence numbers $i1, i2$ and so on as found after the subkeyword adapt_boundary.

Default value: 1

change_topology= c indicates if the topology must be changed or that the old topology is kept, in which case only the co-ordinates are changed.

The following options for c are available:

```
not
always
dependent
```

These options have the following meaning:

not The topology is not adapted. Only the co-ordinates are changed.

always The topology is always adapted, hence remeshing is always applied.

dependent Adapting the topology depends on the quality of the newly created mesh. This option has not yet been implemented.

Default value: not

plot_mesh If this keyword is present the newly created mesh is plotted.

interpolate_solution = (v_1, v_2, \dots, v_n) indicates that the solution vectors with sequence numbers v_1, v_2 etc. must be interpolated from the old to new mesh. This option is only effected if the topology of the mesh is changed. So if the mesh is changed without changing the topology, no interpolation is carried out.

3.4.4 The main keyword ADAPT_BOUNDARY

The block defined by the main keyword ADAPT_BOUNDARY defines how the boundary of a free surface or moving surface problem must be adapted during the various iterations. This block describes the adaptation for a part of the boundary that consists of a set of subsequent curves. hence at this moment adaptation is restricted to two-dimensional problems only.

Adaptation of the complete mesh may contain several "calls" to ADAPT_BOUNDARY. The most simple way to do so, is the use of the main keyword ADAPT_MESH in combination with one or more calls to ADAPT_BOUNDARY.

The block defined by the main keyword ADAPT_BOUNDARY has the following structure (options are indicated between the square brackets "[" and "]"):

```
ADAPT_BOUNDARY [,SEQUENCE_NUMBER = s]
(mandatory): opens the input for the adaptation of boundaries.
```

```

curves = ( c1, c5, -c8, ... )
adaptation_method = a
direction = i
number = n
linear/quadratic
coordinate_system = s
exclude_begin = i
exclude_end = i
multiply = m
factor = f
threshold_value = t
angle_1 = alpha1
angle_2 = alpha2
accuracy = eps
omega = w
maxiter = m
redistribute_nodes
change_number_of_elements
plot_boundary
region = ( xmin, xmax, ymin, ymax )
yfact = y
distribute_curves = ( c3, c8, ... )
alpha_redistribute = alpha
move_begin = ci
move_end = ci
obstacle (i1, i2, .. )
velocity=(u_1,u_2,u_3)
check_direction = c
```

```
END
```

```
(mandatory): end of input
```

The sequence number s may be used to distinguish between various input blocks with respect to the adaptation of boundaries.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords ADAPT_BOUNDARY and END however, must be placed at a new record.

Meaning of the subkeywords:

curves = (*c1, c5, -c8, ...*) defines the series of contiguous curves of which the part of the boundary consists that must be adapted. The last point of each curve must coincide with the first point of the next curve. If a negative curve number is used (like in *-c8*) the curve is used in reversed order.

If the subkeyword **curves** is not present, curve *c1* is used.

adaptation_method = *a* Defines the type of adaptation method that is used to update the boundary. Which of the methods is used strongly depends on the the type of problem. The reader is advised to consult the manual standard problems for specific cases. The following values for *a* may be used:

```

standard
funcctr
funcsolcr
film_method
velocity
normal_velocity
normal_gradient
stefan
constant_velocity

```

These subkeywords have the following meaning:

standard The curves are adapted by:

$\mathbf{x}_{new}(i) = \mathbf{x}_{old}(i) + \alpha(i)\mathbf{n}(i)$ with $\mathbf{x}_{old}(i)$ the old co-ordinates of the i^{th} node and $\mathbf{x}_{new}(i)$ the new ones.

$\mathbf{n}(i)$ is the normal at the boundary of the i^{th} node and $\alpha(i)$ the *numberth* degree of freedom in this node.

The normal is computed assuming a counter clockwise direction of the boundary.

funcctr The curves are adapted by the user provided subroutine FUNCCR as described in Section 3.3.9.

In this subroutine the user may adapt the boundary point-wise using point-wise information and at most one solution vector.

The solution vector is activated by **seq_vectors** = *i* in the input blocks

stationary_free_boundary (3.4.5) or **instationary_free_boundary** (3.4.6).

Only the first vector is used (Default $i = 1$).

funcsolcr The curves are adapted by the user provided subroutine FUNCSOLCR as described in Section 3.3.15.

The main difference with FUNCCR is that the user must adapt the boundary as a whole. All information he gets relates to the complete boundary and moreover he may use more than one old solution vector.

The solution vectors must be activated **seq_vectors** = (*i1, i2, ...*) in the input blocks **stationary_free_boundary** (3.4.5) or **instationary_free_boundary** (3.4.6).

At most 10 vectors are allowed.

film_method See **standard**, however, now α is computed according to the so-called 'film method' as described in Caswell and Viriyayuthakorn (1983).

$$\alpha(i) = \frac{factor (\psi(i) - \psi(0))}{|\mathbf{u} \cdot \mathbf{t}(i)|} \quad (3.4.4.1)$$

$$\psi(i) - \psi(0) = \int_{x(0)}^{x(i)} \mathbf{u} \cdot \mathbf{n} dx \quad (3.4.4.2)$$

The velocity \mathbf{u} must be stored in the vector activated by **seq_vectors** = *i* in the input blocks **stationary_free_boundary** (3.4.5) or **instationary_free_boundary** (3.4.6).

Only the first vector is used (Default $i = 1$).

It is supposed that the degrees of freedom number and number+1 per point correspond to \mathbf{u} .

The flow rate in the first point of the curves to be adapted is assumed to be zero. This is used as start for the integration of the stream function ψ .

With respect to factor: see the subkeyword **factor**

velocity The curves are adapted by:

$\mathbf{x}_{new}(i) = \mathbf{x}_{old}(i) + factor \mathbf{u}(i)$ (Cartesian case) or

$(r, z)_{new}(i) = (r, z)_{old}(i) + \frac{factor}{2\pi}(u_r, u_z)(i)$ (Axisymmetric case)

With respect to factor: see the subkeyword **factor**.

normal_velocity See **standard**. Now α is computed according to $\alpha = factor \mathbf{u} \cdot \mathbf{n}$.

With respect to factor: see the subkeyword **factor**.

With respect to the velocity see the subsubkeyword **film_method**.

normal_gradient The curves are adapted by:

$\mathbf{x}_{new}(i) = \mathbf{x}_{old}(i) + factor \mathbf{u} \cdot \mathbf{n}(i)$ with $\mathbf{x}_{old}(i)$ the old co-ordinates of the i^{th} node and $\mathbf{x}_{new}(i)$ the new ones.

$\mathbf{n}(i)$ is the normal at the boundary of the i^{th} node.

$\mathbf{u}(i)$ is the velocity in point i .

With respect to *factor*, see mult (INPADA(11)).

In fact this is the same as **normal_velocity**, however, there is an essential difference.

In this particular case the velocity is assumed to be stored in the array corresponding to ISOL as a vector of special structure defined per elements, hence it is discontinuous. The velocity in the centroid of the elements at the curves are multiplied by the local normal and *factor*. In this way the displacement of the boundary in all mid points of the curve elements is defined. In order to compute the displacement in the vertices of the curve elements we proceed as follows:

The tangential vector \mathbf{t} in vertex i is defined by: $\mathbf{t} = \mathbf{x}_{i+1} - \mathbf{x}_{i-1}$. The normal \mathbf{n} is perpendicular to this tangent vector, assuming a counter clockwise direction of the boundary.

The displacement of the vertex between two mid-side points is defined such that the area occupied by the region defined by the displacement of both mid-side points is equal to the area occupied by the region defined by the displacement of the vertex in between these two mid-side points. See Figures 3.4.4.1 and 3.4.4.2 for an explanation.

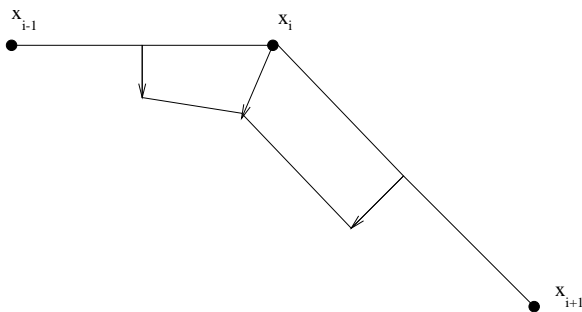


Figure 3.4.4.1: Area occupied by the region defined by the displacement of the vertex

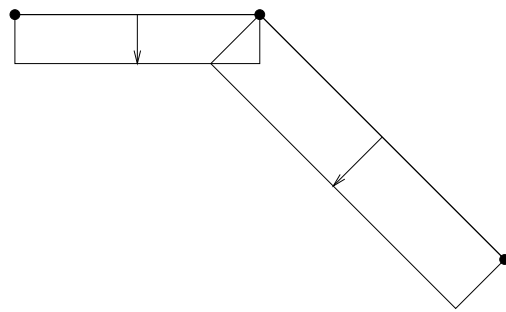


Figure 3.4.4.2: Area occupied by the region defined by the displacement of both mid-side points

This definition of the displacement in the vertices can not be applied for the two end points of the curves. For these two points we use the displacement computed in the midside points.

The reason for using this quite complicated definition is that it provides a kind of smoothing compared to **normal_velocity**. This smoothing is such that if the displacement in the midpoints is equal to zero, also the displacement in the vertices is equal to zero.

At this moment `normal_gradient` is only implemented for linear elements.
 With respect to the velocity see the subkeyword `film_method`.

stefan This type of method is developed such that the Stefan boundary condition in integral form is satisfied exactly. This condition implies that the amount of diffused material is equal to the amount of dissolved material. In fact `stefan` is identical to `normal_gradient`. The main difference is that due to the displacement of the vertices also the mid points are displaced. As a consequence the area defined by the mid-point displacement is not longer equal to the area defined by the vertex displacement. In order to satisfy this requirement a non-linear algorithm must be applied. This method is described in Segal et al (1997).

The non-linear algorithm starts with the linear algorithm defined by the option `normal_gradient` as first iteration and than adapts the boundary such that vertices and mid points move such that the Stefan boundary condition is satisfied. This process needs a relaxation parameter ω . The maximum number of iterations is defined by `maxiter` and the accuracy by the option `accuracy = eps`.

constant_velocity The curves are adapted by:

$$\mathbf{x}_{new}(i) = \mathbf{x}_{old}(i) + factor \mathbf{u}, \text{ where } \mathbf{u} \text{ is a constant velocity.}$$

The value of the velocity is given by the keyword `velocity`.

With respect to factor: see the subkeyword `factor`.

Default value: `standard`

direction = i Indicates the direction of the computation If `direction` = 1 the computation is performed from first to last point, if `direction` = -1 from last to first point. It is sometimes required to alternate between 1 and -1 during iteration.

This option is only used for in case of the film method.

Default value: 1

number= n the $number^{th}$ degree of freedom of the solution vector (and higher) are used for the computation of α . See `adaptation_method`.

Default value: 1

linear/quadratic Indicates if the elements along the boundary must be treated as linear or as quadratic elements. Both options are mutually exclusive.

Default value: to be computed by the subroutine in the following way: If all internal elements are quadratic, the boundary elements are assumed quadratic, otherwise linear.

coordinate_system = s defines the type of co-ordinate system to be used. This is of importance in case the `adaptation_method` is of the type: `film_method`, `velocity` or `normal_velocity`. The following values of s may be used:

Cartesian
 Axi_symmetric

Default value: `Cartesian`

exclude_begin = i indicates which of the co-ordinates of the first point of the first curve must be excluded from adaptation.

The following values of i may be used:

none
 first
 second
 both

These options have the following meaning:

none Both co-ordinates must be adapted

first Only the second co-ordinate must be adapted, the first one remains unchanged

second Only the first co-ordinate must be adapted, the second one remains unchanged

both Both co-ordinates remain unchanged

Default value: **none**

exclude_end = i indicates which of the co-ordinates of the last point of the last curve must be excluded from adaptation.

The same options as for **exclude_begin** are available.

Default value: **none**

multiply = m indicates how the value of *factor* must be computed. This value is used in case the *adaptation_method* is of the type: **film_method**, **velocity** or **normal_velocity**.

The following values of m may be used:

none

dt

These options have the following meaning:

none Multiplication factor is equal to the factor f .

dt Multiplication factor is the parameter $f \times \Delta t$. Here δt denotes the time step *tstep* as stored in common *ctimen*. This time-step is set by the time-integration methods.

Default value: none

factor = f defines the multiplication factor f with respect to the *adaptation_method*. Mark that the combination of **factor** and **multiply** defines the actual multiplication factor.

threshold_value = t defines the threshold value in Caswell's method.

If $|\mathbf{u} \cdot \mathbf{t}(i)|$ larger than the threshold value the boundary is adapted, otherwise the old co-ordinates are used.

If no threshold value is given the machine accuracy (usually 10^{-15}) is used.

Default value: machine accuracy.

angle_1 = α_1 defines the angle of the free surface at the first point.

Default value: no angle given.

angle_2 = α_2 defines the angle of the free surface at the last point.

Default value: no angle given.

accuracy = ϵ defines the accuracy of the iteration process in case of the adaptation method of type **stefan**.

Default value: $\epsilon = 10^{-2}$.

omega = ω defines the relaxation parameter ω of the iteration process in case of the adaptation method of type **stefan**.

Default value: $\omega = 0.5$.

maxiter = m Defines the maximum number of iterations of the iteration process in case of the adaptation method of type **stefan**.

Default value: **maxiter** = 10.

redistribute_nodes If this keyword is given then, once the new boundary is computed, this boundary is approximated by a spline and the nodes along this new boundary are redistributed according to the coarseness defined in the user points. The number of nodes remains the same.

Default value: no redistribution of nodes

change_number_of_elements See `redistribute_nodes` however, in this case the number of elements may be changed. This option may only be used if the topology of the mesh is changed when creating a new mesh.

Mark that the keywords `redistribute_nodes` and `change_elements` are mutually exclusive.

Default value: do not change the number of elements.

plot_boundary If this keyword is found the newly created boundary is plotted.

Default value: no plot

region = $(x_{min}, x_{max}, y_{min}, y_{max})$ This keyword makes only sense in combination with `plot_boundary`.

If given, it restricts the area to be plotted to the region defined by $(x_{min}, x_{max}) \times (y_{min}, y_{max})$.

Default value: the whole region is plotted.

yfact = y defines the multiplication factor to be used in y-direction of the plots. This parameter should only be used in case the length-width ratio is far from 1.

Default value: `yfact` = 1.

yfact = y defines the multiplication factor to be used in y-direction of the plots. This parameter should only be used in case the length-width ratio is far from 1.

Default value: `yfact` = 1.

distribute_curves = $(c3, c8, \dots)$ If this keyword is used, redistribution of nodes is only applied along the curves indicated by $(c3, c8, \dots)$. Of course this keyword is only effective in combination with `redistribute_nodes` or `change_elements`.

Default value: all curves.

alpha_redistribute = α Defines the parameter α corresponding to the splines to be generated in case of `redistribute_nodes`. See Section 2.3 for the definition of α .

Default value: 0.5

move_begin = ci Indicates that the first point of the set of curves must move along curve ci .

Effectively this means that first the displacement of the point is computed and after that, the new point is projected onto the curve ci .

Default value: no projection onto a curve.

move_end = ci Indicates that the last point of the set of curves must move along curve ci .

Effectively this means that first the displacement of the point is computed and after that, the new point is projected onto the curve ci .

Default value: no projection onto a curve.

obstacle (i_1, i_2, \dots) Indicates that the update of the boundary must take into account the presence of the obstacles i_1, i_2, \dots . The obstacles itself have been defined by the mesh generator.

If the boundary crosses an obstacle, all nodes that cross the free boundary are projected onto the obstacle. Hence the obstacle is treated as a physical obstacle in the mesh.

All obstacles must consist of one closed curve. The option curves of curves may be used in the mesh generation, in order to create an obstacle.

It is possible to define boundary conditions and boundary elements along the obstacle, which are only activated as soon as the corresponding nodes in the mesh are positioned at the obstacle. See Sections 3.2.2 and 3.2.5 for the details.

At most 10 obstacles are permitted.

Default value: no obstacles

velocity = (u_1, u_2, u_3) defines the constant velocity in case `constant_velocity` is used as adaptation method.

Default value: $\mathbf{u} = (0, 0, 0)$.

check_direction = c makes only sense in combination with `adaptation_method` = `stefan`. If this keyword is used it is checked if the update of the boundary is in the direction of the outward or inward pointed normal depending on the value of c .

If the update of the boundary has the wrong sign an error message is given and the process halted.

Possible values for c are:

none
positive
negative

Meaning of these sub keywords:

none No check is carried out.

positive It is check if the change of the boundary is in the same direction as the outwards pointed normal.

negative It is check if the change of the boundary is in the same direction as the inwards pointed normal.

Default value: $c = \mathbf{none}$

3.4.5 The main keyword STATIONARY_FREE_BOUNDARY

The block defined by the main keyword STATIONARY_FREE_BOUNDARY defines how the free boundary for a stationary case must be adapted as well as how the corresponding mesh must be adapted. Furthermore this block defines when convergence is reached and the iteration must be stopped.

The block defined by the main keyword STATIONARY_FREE_BOUNDARY has the following structure (options are indicated between the square brackets "[" and "]"):

```
STATIONARY_FREE_BOUNDARY [,SEQUENCE_NUMBER = s]
(mandatory): opens the input for the stationary free boundary problem.

    maxiter = m
    miniter = m
    accuracy = eps
    print_level = p
    criterion = c
    adapt_mesh = i
    at_error = e
    seq_vectors = (i1, i2, ... )
    write_mesh
    no_write_mesh

END
(mandatory): end of input
```

The sequence number s may be used to distinguish between various input blocks with respect to stationary free boundary problems.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords STATIONARY_FREE_BOUNDARY and END however, must be placed at a new record.

Meaning of the subkeywords:

maxiter = m , defines the maximum number of iterations that may be performed.

The default value is 20.

miniter = m , defines the minimum number of iterations that must be performed.

The default value is 2.

accuracy = ε , defines the accuracy. If the difference between succeeding solutions is less than ε the process is considered converged and the iteration is halted.

The default value is $\varepsilon = 10^{-3}$

print_level = p gives the user the opportunity to indicate the amount of output information he wants from the iteration process. p may take the values -1, 0, 1 or 2. The amount of output increases for increasing value of p . If $p = -1$ no output at all is produced.

The default value is $p = 0$

criterion = c defines the type of termination criterion to be used. Possible values are:

```
absolute
relative
```

If **absolute** is used (default value) the process is stopped if $\|\mathbf{u}^{k+1} - \mathbf{u}^k\| \leq \epsilon$.

If **relative** is used the process is stopped if $\frac{\|\mathbf{u}^{k+1} - \mathbf{u}^k\|}{\|\mathbf{u}^{k+1}\|} \leq \epsilon$.

Here u^k means the solution at the k^{th} iteration.

The default value is **absolute**

adapt_mesh = i defines how the mesh must be adapted. i refers to the sequence number of the input block "ADAPT_MESH" as defined in Section 3.4.3.

The default value is $i = 1$

at_error = e defines which action should be taken if the iteration process terminates because no convergence could be found. Possible values are:

stop
return

If **stop** is used the iteration process is stopped if no convergence is found, otherwise (**return**) means that control is given back to the main program and the result of the last iteration is used as solution.

The default value is **stop**.

seq_vectors = i_1, i_2, \dots , defines the sequence numbers of a set of vectors that may be used to update the free boundary.

In general only the first one in the row is used, except when the boundary is adapted by subroutine FUNCSOLCR as described in Section 3.3.15.

See Section 3.4.4 how to activate the use of FUNCSOLCR.

The first vector in the row is used to check the convergence to steady state.

The default value is $i_1 = 1$.

write_mesh indicates that the mesh must be written to the file **meshoutput** as soon as convergence has been reached. This step is necessary for postprocessing purposes.

The default value is **write_mesh**

no_write_mesh indicates that no mesh must be written.

3.4.6 The main keyword INSTATIONARY_FREE_BOUNDARY

The block defined by the main keyword INSTATIONARY_FREE_BOUNDARY defines how the free boundary for a time-dependent case must be adapted as well as how the corresponding mesh must be adapted. Furthermore in this block it is possible to define the mesh velocity.

The block defined by the main keyword INSTATIONARY_FREE_BOUNDARY has the following structure (options are indicated between the square brackets "[" and "] "):

```
INSTATIONARY_FREE_BOUNDARY [,SEQUENCE_NUMBER = s]
```

(mandatory): opens the input for the instationary free boundary problem.

```

print_level = p
adapt_mesh = i
seq_vectors = (i1, i2, ... )
mesh_velocity = i
check_boundary = mb
check_mesh = mm
alpha_boun = a
angle_min = a1
angle_max = a2
interpolate_solution = ( v1, v2, ... , vn )
write_mesh
no_write_mesh

```

```
END
```

(mandatory): end of input

The sequence number s may be used to distinguish between various input blocks with respect to instationary free boundary problems.

The sequence of the subkeywords is arbitrary. They may be put at several lines, but it is also allowed to put a series of subkeywords in one line.

The main keywords INSTATIONARY_FREE_BOUNDARY and END however, must be placed at a new record.

Meaning of the subkeywords:

print_level = p gives the user the opportunity to indicate the amount of output information he wants from the iteration process. At this moment only $p = 0$ is available.

The default value is $p = 0$

adapt_mesh = i defines how the mesh must be adapted. i refers to the sequence number of the input block "ADAPT_MESH" as defined in Section 3.4.3.

The default value is $i = 1$

seq_vectors = i_1, i_2, \dots , defines the sequence numbers of a set of vectors that may be used to update the free boundary.

In general only the first one in the row is used, except when the boundary is adapted by subroutine FUNCSOLCR as described in Section 3.3.15.

See Section 3.4.4 how to activate the use of FUNCSOLCR.

The default value is $i_1 = 1$.

mesh_velocity = i If this statement is found, the mesh velocity is computed, and stored in the vector with sequence number i . If this statement is not given, the mesh velocity is not computed.

The mesh velocity \mathbf{u} is defined by $\mathbf{u} = \frac{\mathbf{x}^{n+1} - \mathbf{x}^n}{\Delta t}$, with \mathbf{x}^n the co-ordinates at time level t , \mathbf{x}^{n+1} the co-ordinates at time level $t + \Delta t$, and Δt the time step.

Of course the mesh velocity can only be computed if the mesh topology is preserved. This means that in the block "ADAPT_MESH" only `change_topology = not` is allowed. Remeshing may only take place after the computation of the mesh velocity.

check_boundary = *mb* indicates if, and how the boundary must be checked after the mesh has been adapted. Possible values for *mb*:

0 The boundary is not checked.

1 The boundary is checked in the following way:

- If there are some subsurfaces generated by the submesh generator **general** it is checked if the begin and end elements of subsequent boundary curves satisfy the requirement $0.3 < \frac{\Delta x_1}{\Delta x_2} < 3.3$. Here Δx_1 and Δx_2 are the lengths of two subsequent elements. This requirement is essential for the submesh generator **general**.
- For all curves in the mesh it is checked if the ratio of the begin and end elements of the original curves generated by SEPMESH and the present elements is not larger than α .

If one of the criteria is not satisfied, remeshing will be applied. In that case also the number of nodes along the boundaries may be adapted.

Default value: 0

check_mesh = *mm* indicates if, and how the mesh must be checked after the mesh has been adapted.

If also `check_boundary` is active and the mesh must be adapted because of the boundary, `check_mesh` is not applied. Possible values for *mm*:

0 The mesh is not checked.

1 The mesh is checked in the following way:

All angles β of the triangles and quadrilaterals in the mesh must satisfy $\beta_{min} < \beta < \beta_{max}$. If this criterion is not satisfied, remeshing will be applied.

In this case the number of nodes along the boundary is not changed.

Default value: 0

Remark: if `check_mesh = 1` is used and remeshing is applied, it is necessary to interpolate the solution vectors to the new mesh by `interpolate_solution = (...)`, even if `interpolate_solution` is used in the block `adapt_mesh`. If you do not interpolate it is possible that the number of unknowns corresponding to the mesh differs from that in the solution vector.

alpha_boun = α gives the value of the parameter α to be used for `check_boundary`.

Default value: 2

angle_min = β_{min} gives the value (in degrees) of the parameter β_{min} to be used for `check_mesh`.

Default value: 10

angle_max = β_{max} gives the value (in degrees) of the parameter β_{max} to be used for `check_mesh`.

Default value: 120

interpolate_solution = (*v1, v2, ..., vn*) indicates that the solution vectors with sequence numbers *v1, v2* etc. must be interpolated from the old to new mesh. This option is only effected if the topology of the mesh is changed. So if the mesh is changed without changing the topology, no interpolation is carried out.

If the mesh velocity is computed, this velocity is automatically interpolated to the mesh if necessary.

write_mesh Indicates that the mesh at $t = tend$ is written to the file `meshoutput`. Hence the old file `meshoutput` is destroyed.

Default value: the mesh is not written.

no_write_mesh Is the opposite of `write_mesh`. This is also the default value.

3.5 Description of some special files that may be used

In some cases the user wants to provide data via separate files. This is possible if in the input a reference to such a file is present. In the previous sections in some cases such a possibility has been provided. In this Section we describe the formats to be used for these files.

3.5.1 describes the file with nodal point numbers that may be used to define in which nodes essential boundary conditions are given.

3.5.2 describes the file with nodal point numbers and corresponding values that may be used to fill values in nodal points.

3.5.3 describes the file with element numbers and corresponding values that may be used to fill values in elements.

3.5.4 describes the file with pairs of electrode numbers and corresponding values that may be used to fill values in a capacity vector.

3.5.1 Description of the file with the nodal point numbers

Description

Sometimes the user prefers to give nodal point numbers in a separate file instead of via the standard input file. This is for example possible if in the standard input file some reference to this special file has been made. In the previous sections it can be found under which conditions such a reference is possible.

In this section we describe the file with nodal point numbers that may be for example used in case the user want to prescribe essential boundary conditions in the points given in this file.

The file itself must be an ASCII file with the name as referred to in the standard input file. reading is performed using the FORTRAN free format option: `read(75,*)`. This means that each read starts at a new record and that extra information in a record is skipped. The first record must contain the number of nodal points (N) stored in the file. In the next N records the nodal point numbers must be stored. Each nodal point number must start at a new record (line). At least N+1 records are required.

An example of such a file is:

```
5   # number of nodes
3   # node numbers
6
7
9
1
```

But also

```
5   # number of nodes
3   3.5 # node numbers + values which are skipped
6   3.4
7   3.3
9   3.0
1   1.2
```

3.5.2 Description of the file with the nodal point numbers and corresponding values

Description

Sometimes the user prefers to give nodal point numbers and corresponding values in a separate file instead of via the standard input file. This is for example possible if in the standard input file some reference to this special file has been made. In the previous sections it can be found under which conditions such a reference is possible.

In this section we describe the file with nodal point numbers and values that may be for example used in case the user want to fill essential boundary conditions in the points given in this file.

The file itself must be an ASCII file with the name as referred to in the standard input file. reading is performed using the FORTRAN free format option: `read(75,*)`. This means that each read starts at a new record and that extra information in a record is skipped. The first record must contain the number of nodal points (N) stored in the file. In the next N records the nodal point numbers must be stored followed by the corresponding value. Node number and value are read by the same read statement. Each nodal point number must start at a new record (line). At least N+1 records are required.

An example of such a file is:

```
5    # number of nodes
3    3.5 # node numbers + values
6    3.4
7    3.3
9    3.0
1    1.2
```

Mark that this is the same file as the one described in Section [3.5.1](#).

3.5.3 Description of the file with the element numbers and corresponding values

Description

Sometimes the user prefers to give element numbers and corresponding values in a separate file instead of via the standard input file. This is for example possible if in the standard input file some reference to this special file has been made. In the previous sections it can be found under which conditions such a reference is possible.

In this section we describe the file with element numbers and values that may be for example used in case the user want to fill a vector of special structure defined per element. Such a vector may for example be used to define coefficients.

The file itself must be an ASCII file with the name as referred to in the standard input file. reading is performed using the FORTRAN free format option: `read(75,*)`. This means that each read starts at a new record and that extra information in a record is skipped. The first record must contain the number of elements (N) stored in the file. In the next N records the element numbers must be stored followed by NDEGFD corresponding values. Element number and values are read by the same read statement.

Each element number must start at a new record (line). At least N+1 records are required.

NDEGFD is defined by the input part that activates the reading of this file, for example the record `FILE_ELEMENT_VALUES 'file_name'` in the create block (3.2.10).

An example of such a file is:

```
5      # number of elements
3      3.5 3.4 3.3 3.2 # element numbers + values
6      3.4 4.4 4.3 2.2
7      3.3 5.4 3.3 4.2
9      3.0 1.4 1.3 6.2
1      1.2 2.4 2.3 3.2
```

3.5.4 Description of the file with the electrode pairs and corresponding capacities

Description

Besides filling values in standard vectors it is also possible to fill values in a vector of the structure of a capacity vector. This is especially meant for the inverse problem as described in Section 3.2.20. The reason is that one wants to read measured values and use them as right-hand side.

The file itself must be an ASCII file with the name as referred to in the standard input file. reading is performed using the FORTRAN free format option: `read(75,*)`. This means that each read starts at a new record and that extra information in a record is skipped. The first record must contain the number of lines (N) following the first line stored in the file. In the next N records pairs of electrode numbers must be stored followed by NDEGFD corresponding values. Pairs of capacity numbers and values are read by the same read statement.

Each new pair must start at a new record (line). The pair of electrode numbers refers to the electrode that has potential 1 and the electrode on which the capacity is measured. The sequence of the pair is not important.

Precisely N+1 records are required.

NDEGFD is defined by the input part that activates the reading of this file, for example the record `FILE_CAPACITY_VALUES 'file_name'` in the create block (3.2.10).

An example of such a file is:

```
5      # number of lines
1 2    3.52 # pair + value
1 3    3.43
2 5    3.35
9 3    3.02
1 6    1.232
```

For an example see Section 6.2.11.

3.6 Parallel computing

If you want to use a parallel environment of SEPRAN, it is necessary that a library with MPI subroutines is at your disposal.

In order to run a parallel program it is necessary to subdivide the mesh into sub-blocks each corresponding to one processor. Hence the number of sub-blocks defines the number of processors used. For the subdivision into sub-blocks you may either introduce the keyword `parallel` into your mesh input file (see 2.2) or you may use the command `sepmakeparmesh`, which uses an existing mesh to create sub-meshes (see 3.6.1). The sub-meshes are all written to files with names `meshoutput_par.000`, `meshoutput_par.001` to `meshoutput_par.xxx`, where `xxx` is the number of processors.

`meshoutput_par.000` contains global information of the block structure, the other files contain the mesh input for each block separately.

Due to the existence of the files `meshoutput_par.xxx`, the computational program `sepcomp` knows that a parallel computation must be performed. However, this is not sufficient. It is also necessary to use the command `sempi` to run the program. This command requires a main program, which may be either a user written program or the main program `sepcompexe` which corresponds to `sepcomp`.

In case of a user written program the command is:

```
sempi name_of_user_program input_file
```

with `name_of_user_program` the name of the user written program. This may be followed by the standard fortran extension, but that is not necessary. `input_file` is the name of the standard input file that is used for the user program.

In case of program `sepcomp`, one has to get program `sepcompexe` locally before running `sempi`, hence

```
sepget sepcompexe  
sempi sepcompexe input_file
```

Note that it is necessary that main program and input file are all in a directory that can be reached on all nodes.

In case of extra subroutines, just compile these subroutines, before executing `sempi`.

The main program produces two kinds of files. The first set of files have names `sepran_out.xxx` and the second one `sepcomp_par.xxx`, with `xxx` the same extension as for the `meshoutput_par` files (000 excluded).

The files `sepran_out.xxx` contain the standard output of the runs on each processor, including error messages if there are any.

The files `sepcomp_par.xxx` have the same meaning as `sepcomp.out`, but now for each processor. If one wants to do any postprocessing on the complete mesh, it is necessary to combine all these files into one large file `sepcomp.out`. That can be done by the command `sepcombineout`, which needs no parameters.

So a typical parallel run would be something like:

```
sepmesh mesh_input_file (with keyword parallel)  
or  
sepmesh mesh_input_file  
sepmakeparmesh
```

followed by

```
sepget sepcompexe  
sempi sepcompexe input_file  
sepcombineout
```

3.6.1 The command `sepmakeparmesh`

If you have an existing SEPRAN mesh, either made by SEPMESH or some third party mesh generator, it is possible to subdivide this mesh into blocks by the command `sepmakeparmesh`. This command requires an input file.

To run the command use:

```
sepmakeparmesh input_file
```

with `input_file` the name of the input file.

The input file has the usual syntax for a sepran input file. At this moment it has only one input line with the keywords:

```
method = xxx, num_processors = n
```

Meaning of these keywords:

method = xxx defines the way the sub-blocks are defined. The same options as in Section (2.2) are available. However, at this moment only the option

layers

is recommended. The program makes the layers itself by using a Cuthill-McKee renumbering scheme, regardless whether the original mesh has been renumbered or not. The layers are constructed such that there are no disjoint parts in a block.

num_processors = n defines the number of processors to be used, and therefor also the number of sub-blocks

4 How to program your own element subroutines

4.1 Introduction

In this chapter it is described, how the user may add his own element subroutines to SEPRAN. In order to use your own elements it is necessary to define type numbers between 1 and 99. All other type numbers refer either to standard SEPRAN elements or to non-existing elements.

As soon as type numbers in the user range 1 to 99 are given, a number of input parts expect that user written element subroutines are provided.

The following element subroutines are expected by input blocks for program SEPCOMP.

The input block SOLVE and the input block NONLINEAR_EQUATIONS expect that an element subroutine to build element matrix and element right-hand side is provided. Depending on the input this means that one of the subroutines ELEM, ELEM1 or ELEM2 is expected. Which of these three subroutines is expected is described in Section 4.2.

The input blocks DERIVATIVES or OUTPUT expect that subroutines ELDERV or ELCERV are submitted. See Section 4.5 in order to decide which subroutine is expected.

Finally the input block INTEGRALS expects a function subroutine ELINT.

In Section 4.2 subroutine ELEM is described. Section 4.3 describes ELEM1 and Section 4.4 ELEM2. The derivative element subroutines ELDERV and ELCERV are described in Sections 4.5 and 4.6. Finally in Section 4.7 the function subroutine ELINT is described.

4.2 Subroutine ELEM

Description

The subroutines ELEM, ELEM1 and ELEM2 are called by program SEPCOMP in the case that a large matrix or right-hand side must be constructed and type numbers between 1 and 99 are used. A large matrix or right-hand side is constructed if the input block "SOLVE" or the input block "NONLINEAR.EQUATIONS" is used in the input of SEPCOMP. Another possibility is that the user has programmed his own main program and calls subroutine BUILD as described in the Programmer's Guide.

All three element subroutines ELEM, ELEM1 and ELEM2 are used to build an element matrix and an element vector. However, there are some differences. In fact ELEM is the simplest of the three. ELEM is used to construct an element matrix and an element vector, where the matrix and vector may depend on the old solution but not on other SEPRAN vectors.

ELEM1 is the first extension. In ELEM1 not only the element matrix and element vector may be built but also the element mass matrix. Furthermore the same restrictions as for ELEM are valid. ELEM1 is typically meant for time-dependent problems and eigenvalue problems.

Subroutine ELEM2 is the most extensive element subroutine available. It may be used to construct the element stiffness matrix, the element vector and the element mass matrix. The difference with ELEM1 is that the element matrices and vectors all may depend on all already constructed SEPRAN vectors. Such a possibility is essential for coupled programs.

Before describing subroutine ELEM in more detail, it is necessary to know which of these subroutines is called by program SEPCOMP and under what conditions. The decision if subroutine ELEM, ELEM1 or subroutine ELEM2 is called is made in a very special way. In fact the only reason why this is so complicated is that SEPRAN is upwards compatible and programs made in the past must always be running in future versions.

If only one vector V1 is created by program SEPCOMP, the situation is simple. In that case always subroutine ELEM is called instead of ELEM2. In case of a time-dependent problem (marked by time loop or time integration), subroutine ELEM1 is called in case of one vector V1.

However, if there are more vectors to be created then the situation becomes complicated. This situation can only occur if the input block "STRUCTURE" is used. If, at the moment that a matrix and right-hand side is built, only the vector V1 has been filled, then subroutine ELEM is called. In all other cases ELEM2 is called. Hence, if the input block "STRUCTURE" has the following contents:

```
structure
  prescribe_boundary_conditions, vector = 1
  solve_linear_system, vector = 1
  create vector 2
  solve_linear_system, vector = 2
end
```

then the first call of solve_linear_system activates the call of ELEM and the second one the call of ELEM2. If, however, the block is defined by:

```
structure
  prescribe_boundary_conditions, vector = 2
  solve_linear_system, vector = 2
end
```

then the call of solve_linear_system activates the call of ELEM2.

Subroutines ELEM1 and ELEM2 are described in the next sections; in this section we restrict ourselves to subroutine ELEM.

Subroutine ELEM is called by program SEPCOMP for each element with type number > 0 and ≤ 99 . So in program SEPCOMP a subroutine BUILD is called which creates the large matrix and right-hand side. This subroutine BUILD contains a loop over the element groups. For each element group it contains a loop over all elements in this group. If the element group corresponds to an element with type number 1 to 99, subroutine ELEM is called in the inner loop. Hence, subroutine ELEM is called for each element separately.

Subroutine ELEM must be written by the user.

Call

```
CALL ELEM ( COOR, ELEMMT, ELEMVC, IUSER, USER, UOLD, MATRIX,
           VECTOR, INDEX1, INDEX2 )
```

Parameters

INTEGER IUSER(*), INDEX1(*), INDEX2(*)

DOUBLE PRECISION COOR(*ndim*,*), ELEMMT(*icount*,*icount*), ELEMVC(*), UOLD(*)

LOGICAL MATRIX, VECTOR

COOR Double precision two-dimensional array of size $\text{NDIM} \times$ number of points, where NDIM is the dimension of the space. So for a two-dimensional problem COOR must be declared as COOR(2,*).

To find the co-ordinates of the nodes of the element, array INDEX1 must be used. The x -co-ordinate of the i^{th} local point in the element is given by COOR(1,INDEX1(i)), the y -co-ordinate by COOR(2,INDEX1(i))

A common way to extract the co-ordinates of the element is to define a help array X of size $\text{ndim} \times \text{npelm}$, where *npelm* denotes the maximum number of nodes in the elements. The following piece of code copies the co-ordinates from array coor into array X:

```
x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))
```

ELEMMT In this double precision two-dimensional array the user must store the element matrix, if the large matrix must be computed, in the following way:

$$\text{ELEMMT}(j,i) = s_{ij} ; i,j = 1(1)\text{ICOUNT},$$

where ICOUNT is the number of degrees of freedom in the element (prescribed or not). The degrees of freedom in an element are stored sequentially, first all degrees of freedom corresponding to the first point, then to the second, etcetera.

The reason that in fact the transpose of the element matrix must be stored in ELEMMT is due to the way FORTRAN stores more-dimensional arrays. Internally ELEMMT is stored as one-dimensional array in the sequence s_{11}, s_{12}, \dots . However, a FORTRAN two-dimensional array is stored as s_{11}, s_{21}, \dots

ELEMVC In this double precision array the user must store the element vector, if the large vector must be computed, in the following way:

$$\text{ELEMVC}(i) = f_i ; i = 1(1)\text{ICOUNT}.$$

IUSER,USER These arrays are used by SEPCOMP to store information of the coefficients for the differential equation. The storage of IUSER and USER is described in the manual Standard Problems. For simple problems using SEPCOMP, the storage in IUSER and USER may be too complicated to be used.

If the user calls subroutine BUILD in his own SEPRAN program, he may fill IUSER and USER in his own way, since these arrays are passed undisturbed from main program to element subroutine.

UOLD In this array the old solution, as indicated by V1, is stored. This solution may contain the boundary conditions only, if the array has been created by prescribe_boundary_conditions, but also a starting vector if V1 has been created by create or even the previous solution in an iteration process if nonlinear_equations are used. Array INDEX2 may be used to find the degrees of freedom in UOLD, corresponding to the element. The i^{th} local degree of freedom in the element can be found from UOLD(INDEX2(i)).

A common way to extract the old solution in the nodal points of the element is to define a help array U of size icount. The following piece of code copies the old solution from array uold into array U:

```
u(1:icount) = uold(index2(1:icount))
```

MATRIX Logical variable. When MATRIX is true the element matrix must be computed, when MATRIX is false the element matrix is not used.

MATRIX has got a value at the input of ELEM and may not be changed by the user.

VECTOR Logical variable. When VECTOR is true the element vector must be computed, when VECTOR is false the element vector is not used.

VECTOR has got a value at the input of ELEM and may not be changed by the user.

INDEX1 In this integer array of length INPELM, the point numbers of the nodal points in the element are stored. The user needs these numbers to compute the co-ordinates of the nodal points of the element. INPELM is the number of nodal points in the element. See COOR.

INDEX2 In this integer array of length ICOUNT, the positions of the degrees of freedom in the element with respect to array UOLD, are stored sequentially. The user needs this information in order to compute the preceding solution in the nodal points.

Examples:

Suppose we have a triangle with 3 nodes and in each node there is exactly one unknown. Suppose we want to store the three unknowns corresponding to the element in an array U of length 3. Then the following statements may be used:

```
U(1:3) = UOLD (INDEX2(1:3))
```

Suppose for the same triangle we have two unknowns per point and we want to store the first unknown in an array U and the second in an array V. Then the statements become:

```
do i = 1 ,3
  U(i) = UOLD (INDEX2(2*i-1))
  V(i) = UOLD (INDEX2(2*i))
end do ! i = 1, 3
```

Besides the parameters in the parameter list program SEPCOMP, (actually subroutine BUILD) communicates also with ELEM by the common block CACTL:

```
integer IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT, NOTVC,
      IRELEM, NUSOL, NELEM, NPOINT
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
      NOTVC, IRELEM, NUSOL, NELEM, NPOINT \emp
```


In a program it is better to include the common from the sepran commons directory: (free format .f90 file)

```
include 'SPCOMMON/cactl'
```

or in case of a fixed format .f file

```
include 'SPcommon/cactl'
```

Note that the capitals as well as the lower case letters in the expression between the quotes are mandatory, since everything between quotes is case sensitive. The following parameters may be useful:

IELEM Element number.

ITYPE Problem definition number. This number is defined in the input block "PROBLEM" by $ELGRPi = (\text{type} = ni)$.

IELGRP Standard element sequence number. Boundary elements get standard sequence numbers: $NELGRP + 1, NELGRP + 2, \dots, NELGRP + NUMNATBND$, where $NELGRP$ is the number of element groups and $NUMNATBND$ the number of boundary element groups.

INPELM Number of nodal points in element.

ICOUNT Number of degrees of freedom in element.

IFIRST This parameter indicates if the element subroutine is called for the first time for the specific element group ($IFIRST=0$) or not ($IFIRST=1$). This parameter may be of help for experienced FORTRAN programmers in order to initialize parameters and even local arrays only once. Since FORTRAN does not save local parameters it is necessary to use the "SAVE" statement if this option is utilized.

NOTMAT This parameter indicates if an element matrix is identical to zero ($NOTMAT=1$) or not ($NOTMAT=0$) for all elements with standard element sequence number $IELGRP$. This parameter is one of the two parameters in common block CACTL the user is allowed to change in subroutine ELEM.

If the element matrix is identical zero for the complete element group, the user may indicate this by setting $NOTMAT = 1$ in subroutine ELEM.

NOTVEC This parameter indicates if an element vector is identical to zero ($NOTVEC=1$) or not ($NOTVEC=0$) for all elements with standard element sequence number $IELGRP$. This parameter is one of the two parameters in common block CACTL the user is allowed to change in subroutine ELEM.

If the element vector is identical zero for the complete element group, the user may indicate this by setting $NOTVEC = 1$ in subroutine ELEM.

NELEM Number of elements with standard element sequence number $IELGRP$ in the mesh.

NPOINT Number of nodal points in the mesh.

NUSOL Number of degrees of freedom in the solution vector.

IRELEM Relative element number with respect to standard element sequence number $IELGRP$.

The parameters in CACTL are given a value by program SEPCOMP. These values may change from element to element and must not be changed by the user.

In order to distinguish between different element groups both the parameters $IELGRP$ and $ITYPE$ may be used.

Input

Program SEPCOMP fills the arrays COOR, INDEX1, INDEX2 and array UOLD before the call of ELEM.

BUILD gives MATRIX and VECTOR a value. All parameters in common block CACTL have got a value by program SEPCOMP.

Output

The arrays ELEMNT and ELEMVC must have been filled by the user, depending on the values of MATRIX, VECTOR, NOTMAT and NOTVC.

Interface

Subroutine elem.f90 must be programmed as follows:

```

subroutine elem ( coor, elemnt, elemvc, iuser, user, uold, &
                 matrix, vector, index1, index2 )
implicit none

include 'SPCOMMON/cactl'

integer ndim
parameter ( ndim = 2 )
double precision coor(ndim,*), elemnt(icount,icount), elemvc(icount), &
                 user(*), uold(*)
integer iuser(*), index1(inpelm), index2(icount)
logical matrix, vector

!      declarations of local variables
!      for example:

integer npelm, i, k
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm)

if ( itype==1 ) then

!      --- possibly statements to fill x and u for type number 1
!      for example:

x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))

u(1:icount) = uold(index2(1:icount))

!      --- statements to fill the arrays elemnt and elemvc

do k = 1, icount
  do i = 1, icount
    elemnt(i,k) = "s(ki)"
  end do
  elemvc(k) = "f(k)"
end do

else if ( itype==2 ) then

```

```
! --- the same type of statements for itype = 2, etcetera

end if
end
```

Remarks:

This is the free format .f90 version.

For problems in complex variables (like the Helmholtz equation) one may declare ELEMNT and ELEMVC as double complex arrays, which means that they are treated as double precision complex arrays.

If the number of unknowns per point is not 1 in each nodal point, the user must be very careful with respect to the sequence in which the element matrix and element vector must be filled.

The degrees of freedom are always filled in the sequence of the local nodal point numbering of the elements. Consider for example the quadratic triangle in Figure 4.2.1. The local numbering of the nodes starts always in a vertex and follows the boundary until the last point has been numbered. There is no guarantee that the numbering is counter-clockwise, although it usually is. Suppose that the first local number coincides with the vertex with global nodal point number 13 and that the

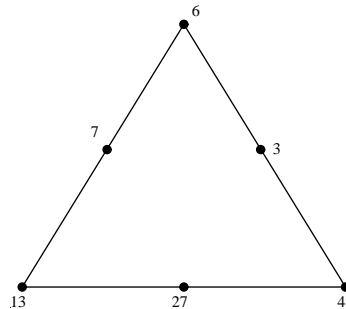


Figure 4.2.1: Element with global nodal point numbers

direction of the numbering is counter clockwise. Suppose that furthermore that in the vertices there are 3 unknowns (ψ, u, v) and in the mid-side points there is one unknown (u_t). Then the unknowns in the element are numbered in the sequence:

$$\psi_{13}, u_{13}, v_{13}, u_{t27}, \psi_{41}, u_{41}, v_{41}, u_{t3}, \psi_6, u_6, v_6, u_{t7}$$

So the rows and columns of the element matrix must be numbered in the same sequence. Array `index2` has been filled in exactly the same sequence.

Shifted Laplace preconditioner

If the shifted Laplace preconditioner is used as described in Sections (3.2.4) and (3.2.8), the element matrix consists of two parts. Part 1 of length `icount` \times `icount` refers to the standard element matrix, whereas the second part of length `icount` refers to the diagonal of the *zeroth order* part. Of course in this case ELEMNT can no longer be defined as a two-dimensional array.

The user must fill the second part of the matrix too.

A simple method to program this case is to use a help subroutine: `elemhelp` with one extra parameter `DIAG`, like:

```
SUBROUTINE ELEMHELP ( COOR, ELEMNT, ELEMVC, IUSER, USER, UOLD, MATRIX,
VECTOR, INDEX1, INDEX2, DIAG )
```

`DIAG` is a double precision of double complex array of length `ICOUNT`, that should be filled with the diagonal matrix. In this subroutine ELEMNT can be used as two-dimension array.

ELEMHELP can be called from ELEM by

```
CALL ELEMHELP ( COOR, ELEMNT, ELEMVC, IUSER, USER, UOLD, MATRIX,  
                VECTOR, INDEX1, INDEX2, ELEMNT(ICOUNT**2+1) )
```

Inside ELEM, ELEMNT is of course one-dimensional.

4.3 Subroutine ELEM1

Description

Subroutine ELEM1 is called by program SEPCOMP in the case that a large matrix or right-hand side must be constructed and type numbers between 1 and 99 are used. ELEM1 has exactly the same task as ELEM, but besides that, ELEM1 also constructs an element mass matrix. This mass matrix may only be a diagonal (i.e. lumped) mass matrix.

At this moment it is not possible to activate ELEM1 directly from SEPCOMP. The only way to address ELEM1 is by explicitly calling the subroutine BUILD in your own SEPRAN main program. See the Programmer's Guide for details.

Subroutine BUILD contains a loop over the element groups. For each element group it contains a loop over all elements in this group. If the element group corresponds to an element with type number 1 to 99, subroutine ELEM1 is called in the inner loop. Hence, subroutine ELEM1 is called for each element separately.

Subroutine ELEM1 must be written by the user.

Call

```
CALL ELEM1 ( COOR, ELEMMT, ELEMVC, ELEMMS, IUSER, USER, UOLD,
            MATRIX, VECTOR, INDEX1, INDEX2, NOTMAS )
```

Parameters

INTEGER IUSER(*), INDEX1(*), INDEX2(*), NOTMAS

DOUBLE PRECISION COOR(*ndim*,*), ELEMMT(icount,icount), ELEMVC(icount), UOLD(*), ELEMMS(icount)

LOGICAL MATRIX, VECTOR

COOR, ELEMMT, ELEMVC, IUSER, USER, UOLD, MATRIX, VECTOR, INDEX1, INDEX2
see subroutine ELEM (4.2)

ELEMMS In this array the user must store the element vector corresponding to the vector MASSMT in the call of subroutine BUILD, when this vector must be computed, in the following way:

$$\text{ELEMMS}(i) = f_i; i = 1 \text{ (1) ICOUNT}$$

In the sequel this array will be referred to as the element mass matrix.

NOTMAS This parameter indicates whether an element mass matrix is equal to zero (NOTMAS=1) or not (NOTMAS=0) for all elements with standard element sequence number IELGRP. When the user sets NOTMAS equal to 1, this means that the element matrices with element sequence number IELGRP are not added to the mass matrix. Mark that NOTMAS has the opposite meaning of NOTMAT.

Parameters from the common block CACTL: see subroutine ELEM (4.2).

Input

Program SEPCOMP fills the arrays COOR, INDEX1, INDEX2 and array UOLD before the call of ELEM1.

BUILD gives MATRIX, VECTOR and NOTMAS a value. All parameters in common block CACTL have got a value by program SEPCOMP.

Output

The arrays ELEMMT, ELEMMS and ELEMVC must have been filled by the user, depending on the values of MATRIX, VECTOR, NOTMAT, NOTMAS and NOTVC. Hence when the user sets NOTMAT equal to one, he does not have to fill the element matrices for this standard element group.

Interface

Subroutine elem1.f90 must be programmed as follows:

```

subroutine elem1 ( coor, elemmt, elemvc, elemms, iuser, user, uold, &
                  matrix, vector, index1, index2, notmas )
implicit none

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision coor(ndim,*), elemmt(icount,icount), elemvc(icount), &
                 user(*), uold(*), elemms(icount)
integer iuser(*), index1(inpelm), index2(icount), notmas
logical matrix, vector

!      declarations of local variables
!      for example:

integer npelm, i, k
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm)

if ( itype==1 ) then

!      --- possibly statements to fill x and u for type number 1
!      for example:

x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))

u(1:icount) = uold(index2(1:icount))

!      --- statements to fill the arrays elemmt, elemms and elemvc

do k = 1, icount
  do i = 1, icount
    elemmt(i,k) = "s(ki)"
  end do
  elemvc(k) = "f(k)"
  elemms(k) = "m(kk)"
end do

else if ( itype==2 ) then

!      --- the same type of statements for itype = 2, etcetera

end if
end

```

In this case we used the free format .f90 version.

4.4 Subroutine ELEM2

Description

Subroutine ELEM2 is called by program SEPCOMP in the case that a large matrix or right-hand side must be constructed and type numbers between 1 and 99 are used. ELEM2 must be considered as an extension of ELEM1, because it has the extra possibility of using one or more result arrays of preceding computations. These arrays may correspond to different problems, i.e. different problem numbers, and may also be of special type.

In Section 4.2 it has been described in which case ELEM2 will be called instead of ELEM.

Subroutine BUILD contains a loop over the element groups. For each element group it contains a loop over all elements in this group. If the element group corresponds to an element with type number 1 to 99, subroutine ELEM2 is called in the inner loop. Hence, subroutine ELEM2 is called for each element separately.

Subroutine ELEM2 must be written by the user.

Call

```
CALL ELEM2 ( COOR, ELEMNT, ELEMVC, ELEMMS, IUSER, USER, UOLD, MATRIX,
            VECTOR, INDEX1, INDEX2, INDEX3, INDEX4, NOTMAS, NUMOLD )
```

Parameters

INTEGER NOTMAS, NUMOLD, IUSER(*), INDEX1(*inpel*), INDEX2(*icount*), INDEX4(NUMOLD,*inpel*)

INTEGER (kind=8) INDEX3(NUMOLD,*)

DOUBLE PRECISION COOR(*ndim*,*), ELEMNT(*icount*,*icount*), ELEMVC(*icount*), UOLD(*),
ELEMMS(*icount*)

LOGICAL MATRIX, VECTOR

COOR, ELEMNT, ELEMVC, IUSER, USER, UOLD, MATRIX, VECTOR, INDEX1, INDEX2
see subroutine ELEM (4.2)

NOTMAS, ELEMMS see subroutine ELEM1 (4.3)

UOLD In this array all preceding solutions are stored, i.e. all solutions that have been computed before. These solutions correspond to the vectors V1, V2 etcetera.

These solutions do not have to be stored consecutively, neither do they have to start at position 1 of UOLD. In order to find the positions of the respective arrays in UOLD the arrays INDEX3 and if necessary INDEX4 should be used.

INDEX3 Two-dimensional integer array of length NUMOLD x NINDEX containing the positions of the "old" solutions in array UOLD with respect to the present element.

For example UOLD(INDEX3(*i*,*j*)) contains the *j*th unknown in the element with respect to the *i*th old solution vector. The number *i* refers to the *i*th vector corresponding to IVCOLD in the call of BUILD.

Mark that it is an integer array of integers of length 8 bytes and therefore it is necessary to define this array as **integer (kind=8)**. The reason is that it contains pointers and all pointers to positions in the buffer array of SEPRAN are long integers.

INDEX4 Two-dimensional integer array of length NUMOLD x INPELM containing the number of unknowns per point accumulated in array UOLD with respect to the present element.

For example INDEX4(*i*,1) contains the number of unknowns in the first point with respect to the *i*th vector stored in UOLD. The number of unknowns in the *j*th point with respect to *i*th vector in UOLD is equal to INDEX4(*i*,*j*) - INDEX4(*i*,*j* - 1) (*j* > 1)
i is always meant as relative with respect to the present element.

Parameters from the common block CACTL: see subroutine ELEM (4.2).

Input

Program SEPCOMP fills the arrays COOR, INDEX1, INDEX2, INDEX3, INDEX4 and if necessary array UOLD before the call of ELEM2.

BUILD gives MATRIX, VECTOR and NOTMAS a value. All parameters in common block CACTL have got a value by program SEPCOMP.

Output

The arrays ELEMMT, ELEMMS and ELEMVC must have been filled by the user, depending on the values of MATRIX, VECTOR, NOTMAT, NOTMAS and NOTVC. Hence when the user sets NOTMAT equal to one, he does not have to fill the element matrices for this standard element group.

Interface

Subroutine elem2.f90 must be programmed as follows:

```

subroutine elem2 ( coor, elemmt, elemvc, elemms, iuser, user, uold, &
                  matrix, vector, index1, index2, index3, index4, &
                  notmas, numold )

implicit none
include 'SPCOMMON/cactl'
integer ndim
parameter ( ndim = 2 )
logical matrix, vector
integer numold, iuser(*), index1(inpelm), index2(icount), &
        index4(numold,inpelm), notmas
integer (kind=8) index3(numold,*)
double precision coor(ndim,*), elemmt(icount,icount), elemvc(icount), &
        user(*), uold(*), elemms(icount)

!      declarations of local variables
!      for example:

integer npelm, i, k
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm), v(npelm)

if ( itype==1 ) then

!      --- Possibly statements to fill X and U for type number 1
!      for example:

x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))

u(1:icount) = uold(index2(1:icount))

u(1:inpelm) = uold(index3(1,1:inpelm))
v(1:inpelm) = uold(index3(2,1:inpelm))

!      --- statements to fill the arrays elemmt, elemms and elemvc

do k = 1, icount

```

```
        do i = 1, icount
            elemmt(i,k) = "s(ki)"
        end do
        elemvc(k) = "f(k)"
        elemms(k) = "m(kk)"
    end do

    else if ( itype==2 ) then

!       --- the same type of statements for itype = 2, etcetera

        end if
    end
```

In this case we used the free format .f90 version.

4.5 Subroutine ELDERV

Description

The user-written subroutine ELDERV is called by program SEPCOMP in the case that an element vector of derived quantities must be constructed and type numbers between 1 and 99 are used. This subroutine is called if the input block "STRUCTURE" is used in program SEPCOMP and in this block the command derivatives is given. If in the input block "DERIVATIVES" the option ELEMENT-WISE is used, instead of ELDERV the subroutine ELCERV is called. ELDERV is also called if the user runs his own main program and calls one of the derivative subroutines.

Communication with SEPCOMP is performed with the parameters in the heading of the subroutine, as well as the parameters in common block CACTL. In program SEPCOMP a subroutine DERIV is called which creates the vector of derived quantities and a vector with weights for the averaging procedure described at the end of this section.

This subroutine DERIV contains a loop over the element groups. For each element group it contains a loop over all elements in this group. If the element group corresponds to an element with type number 1 to 99, subroutine ELDERV is called in the inner loop. Hence, subroutine ELDERV is called for each element separately.

Heading

```
subroutine elderv (  icheld, ix, jdegfd, coor, elemvc, elemwg, &
                  iuser, user, vector1, vector2, index1, index2 )
```

Parameters

INTEGER ICHELD, IX, JDEGFD, IUSER(*), INDEX1(*), INDEX2(*)

DOUBLE PRECISION COOR(*ndim*,*), ELEMVC(ICOUNT), ELEMWG(ICOUNT),
USER(*), VECTOR1(*), VECTOR2(*)

ICHELD This parameter has got the value *s* from the command ICHELD = *s* in the input block "DERIVATIVES". ICHELD has been given this value by program SEPCOMP. The user is not allowed to change this value.

IX This parameter has got the value *ix* from the command IX = *ix* in the input block "DERIVATIVES". IX has been given this value by program SEPCOMP. The user is not allowed to change this value.

JDEGFD This parameter has got the value *d* from the command DEGREE_OF_FREEDOM = *d* in the input block "DERIVATIVES". JDEGFD has been given this value by program SEPCOMP. The user is not allowed to change this value.

COOR Double precision two-dimensional array of size NDIM × number of points, where NDIM is the dimension of the space. So for a two-dimensional problem COOR must be declared as COOR(2,*).

To find the co-ordinates of the nodes of the element, array INDEX1 must be used. The *x*-co-ordinate of the *i*th local point in the element is given by COOR(1,INDEX1(i)), the *y*-co-ordinate by COOR(2,INDEX1(i))

A common way to extract the co-ordinates of the element is to define a help array X of size *ndim* × *npelm*, where *npelm* denotes the maximum number of nodes in the elements. The following piece of code copies the co-ordinates from array coor into array X:

```
x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))
```

ELEMVC In this array, the user must store the derived quantities multiplied by its corresponding weights in the same order as the degrees of freedom corresponding to the output vector to be filled. Hence, first all degrees of freedom in the first nodal point, then all degrees of freedom in the second nodal point, etc.

ELEMWG In this array the user must store the weight factors for the averaging procedure (See "Method"). ELEMWG must be filled in exactly the same sequence as ELEMVC.

IUSER,USER These arrays are used by SEPCOMP to store information of the coefficients for the computation. The storage of IUSER and USER is described in the manual Standard Problems. For simple problems using SEPCOMP, the storage in IUSER and USER may be too complicated to be used.

If the user calls subroutine DERIV in his own SEPRAN program, he may fill IUSER and USER in his own way, since these arrays are passed undisturbed from main program to element subroutine.

VECTOR1,VECTOR2 In these arrays the vectors are stored from which the derived quantities have to be computed. VECTOR1 corresponds to the vector i indicated by SEQ_INPUT_VECTOR 1 = i in the input block "DERIVATIVES" and VECTOR2 to the vector j indicated by SEQ_INPUT_VECTOR 2 = j . If this last option is not used array VECTOR2 may not be filled. Array INDEX2 may be used to find the degrees of freedom in VECTOR j , corresponding to the element. The i^{th} local degree of freedom in the element can be found from VECTOR j (INDEX2(i)).

A common way to extract the solution in the nodal points of the element is to define a help array U of size *icount*. The following piece of code copies the old solution from array VECTOR1 into array U:

```
u(1:icount) = vector1(index2(1:icount))
```

INDEX1 In this integer array of length INPELM, the point numbers of the nodal points in the element are stored. The user needs these numbers to compute the co-ordinates of the nodal points of the element. INPELM is the number of nodal points in the element. See COOR.

INDEX2 In this integer array of length ICOUNT, the positions of the degrees of freedom in the element with respect to the arrays VECTOR1 and VECTOR2, are stored sequentially. The user needs this information in order to extract the values of these vectors in the nodal points.

Besides the parameters in the parameter list program SEPCOMP, (actually subroutine DERIV) communicates also with ELDERV by the common block CACTL:

```
integer ielem, itype, ielgrp, inpelm, icount, ifirst,
+       notmat, notvec, irelem, nusol, nelem, npoint
common /cactl/ ielem, itype, ielgrp, inpelm, icount, ifirst,
+             notmat, notvec, irelem, nusol, nelem, npoint
```

The following parameters may be useful:

IELEM Element number.

ITYPE Problem definition number. This number is defined in the input block "PROBLEM" by ELGRP i = (type = n_i).

IELGRP Standard element sequence number. Boundary elements are skipped in the call of the element subroutines.

INPELM Number of nodal points in element.

ICOUNT Number of degrees of freedom in the vector VECTOR1 (and VECTOR2 if it exists) for this element. Mark that this does not have to be the number of degrees of freedom in the element vector. The user does know this number himself and for that reason there is no specific parameter indicating the length of the element vector.

IFIRST This parameter indicates if the element subroutine is called for the first time for the specific element group (IFIRST=0) or not (IFIRST=1). This parameter may be of help for experienced FORTRAN programmers in order to initialize parameters and even local arrays only once. Since FORTRAN 77 does not save local parameters it is necessary to use the "SAVE" statement if this option is utilized.

NOTMAT This parameter is not used.

NOTVC This parameter indicates if an element vector is identical to zero (NOTVC=1) or not (NOTVC=0) for all elements with standard element sequence number IELGRP. This parameter is the only parameter in common block CACTL the user is allowed to change in subroutine ELDERV.

If the element vector is identical zero for the complete element group, the user may indicate this by setting NOTVC = 1 in subroutine ELDERV.

NELEM Number of elements with standard element sequence number IELGRP in the mesh.

NPOINT Number of nodal points in the mesh.

NUSOL Number of degrees of freedom in the solution vector.

IRELEM Relative element number with respect to standard element sequence number IELGRP.

The parameters in CACTL are given a value by program SEPCOMP. These values may change from element to element and must not be changed by the user.

In order to distinguish between different element groups both the parameters IELGRP and ITYPE may be used.

Input

Program SEPCOMP gives the parameters ICHELD, IX and JDEGFD a value.

Program SEPCOMP fills the arrays COOR, INDEX1, INDEX2 and array VECTOR1 and VECTOR2 before the call of ELDERV.

All parameters in common block CACTL have got a value by program SEPCOMP.

Output

The arrays ELEMVC and ELEMWG must have been filled by the user, depending on the value of NOTVC.

Interface

Subroutine elderv.f90 must be programmed as follows:

```
subroutine elderv (  icheld, ix, jdegfd, coor, elemvc, elemwg, iuser, user, &
                  vector1, vector2, index1, index2 )
implicit none
```

```

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision coor(ndim,*), elemvc(*), elemwg(*),
+             user(*), vector1(*), vector2(*)
integer icheld, ix, jdegfd, iuser(*), index1(inpelm), index2(icount)

!       declarations of local variables
!       for example:

integer npelm, i, k, jcount
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm)

if ( itype==1 ) then

!       --- possibly statements to fill x and u for type number 1
!       for example:

x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))

u(1:icount) = vector1(index2(1:icount))

!       --- statements to fill the arrays elemvc and elemwg

jcount = ...
do i = 1, jcount
  elemwg(i) = "w(i)"
  elemvc(i) = "f(i)*w(i)"
end do

else if ( itype==2 ) then

!       --- the same type of statements for itype = 2, etcetera

end if
end

```

Remark

For problems in complex variables (like the Helmholtz equation) one may declare `EL-EMVC` and possibly `VECTOR1` as double complex arrays, which means that they are treated as double precision complex arrays.

Method

The averaging procedure in program `SEPCOMP` is as follows. Suppose that nodal point j is lying in K different elements. Let the quantity q be given in nodal point j , with a different value in each element. In order to compute an averaged value of q in j , weights w_i ($i = 1, 2, \dots, K$) for each element corresponding to nodal point j must be defined.

The averaged value of q in nodal point j is computed by the following formula:

$$\bar{q}(x^j) = \frac{\sum_{i=1}^K q_i(x^j) w_i}{\sum_{i=1}^K w_i} w_i \geq 0; \quad \sum_{i=1}^K w_i > 0 \quad (4.5.1)$$

with

$\bar{q}(x^j)$ the averaged value of q in nodal point j ,

$q_i(x^j)$ the value of q in nodal point j with respect to element i ,

w_i the weight corresponding to nodal point j with respect to element i .

Simple choices are for example:

$w_i = 1$ or

$w_i = \text{area of element } i$.

The adding process over the various elements is carried out by program SEPCOMP, it is sufficient to compute the derived quantities and weights with respect to each nodal element separately with the aid of subroutine ELDERV.

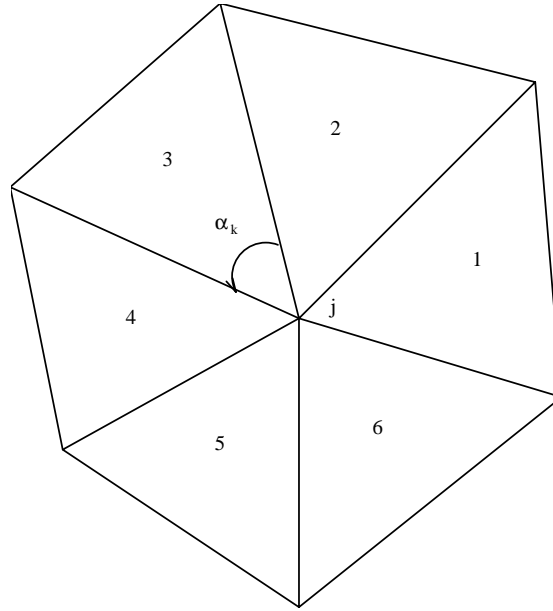


Figure 4.5.1: nodal point j in different elements

4.6 Subroutine ELCERV

Description

The user-written subroutine ELCERV is called by program SEPCOMP in the case that an element vector of derived quantities must be constructed and type numbers between 1 and 99 are used. This subroutine is called if the input block "STRUCTURE" is used in program SEPCOMP and in this block the command derivatives is given. Besides that in the input block "DERIVATIVES" the option ELEMENTWISE must be used to activate ELCERV. Hence ELCERV is used in the case that the derived quantities are stored in a vector of special structure defined per element. No averaging over the elements takes place. ELCERV is also called if the user runs his own main program and calls one of the derivative subroutines.

Communication with SEPCOMP is performed with the parameters in the heading of the subroutine, as well as the parameters in common block CACTL. In program SEPCOMP a subroutine DERIV is called which creates the vector of derived quantities and a vector with weights for the averaging procedure described at the end of this section.

This subroutine DERIV contains a loop over the element groups. For each element group it contains a loop over all elements in this group. If the element group corresponds to an element with type number 1 to 99, subroutine ELCERV is called in the inner loop. Hence, subroutine ELCERV is called for each element separately.

Call

```
CALL ELCERV ( ICHELD, IX, JDEGFD, COOR, ELEMVC, IUSER, USER,
             VECTOR1, VECTOR2, INDEX1, INDEX2 )
```

Parameters

INTEGER ICHELD, IX, JDEGFD, IUSER(*), INDEX1(*), INDEX2(*)

DOUBLE PRECISION COOR(*ndim*,*), ELEMVC(ICOUNT), USER(*), VECTOR1(*), VECTOR2(*)

ICHELD This parameter has got the value *s* from the command ICHELD = *s* in the input block "DERIVATIVES". ICHELD has been given this value by program SEPCOMP. The user is not allowed to change this value.

IX This parameter has got the value *ix* from the command IX = *ix* in the input block "DERIVATIVES". IX has been given this value by program SEPCOMP. The user is not allowed to change this value.

JDEGFD This parameter has got the value *d* from the command DEGREE_OF_FREEDOM = *d* in the input block "DERIVATIVES". JDEGFD has been given this value by program SEPCOMP. The user is not allowed to change this value.

COOR Double precision two-dimensional array of size NDIM × number of points, where NDIM is the dimension of the space. So for a two-dimensional problem COOR must be declared as COOR(2,*).

To find the co-ordinates of the nodes of the element, array INDEX1 must be used. The *x*-co-ordinate of the *i*th local point in the element is given by COOR(1,INDEX1(i)), the *y*-co-ordinate by COOR(2,INDEX1(i))

A common way to extract the co-ordinates of the element is to define a help array X of size *ndim* × *npelm*, where *npelm* denotes the maximum number of nodes in the elements. The following piece of code copies the co-ordinates from array coor into array X:

```
x(1:ndim,1:inpelem) = coor(1:ndim,index1(1:inpelem))
```


ELEMVC In this array, the user must store the derived quantities in the same order as the degrees of freedom corresponding to the output vector to be filled. Hence, first all degrees of freedom in the first nodal point, then all degrees of freedom in the second nodal point, etc.

IUSER,USER These arrays are used by SEPCOMP to store information of the coefficients for the computation. The storage of IUSER and USER is described in the manual Standard Problems. For simple problems using SEPCOMP, the storage in IUSER and USER may be too complicated to be used.

If the user calls subroutine DERIV in his own SEPRAN program, he may fill IUSER and USER in his own way, since these arrays are passed undisturbed from main program to element subroutine.

VECTOR1,VECTOR2 In these arrays the vectors are stored from which the derived quantities have to be computed. VECTOR1 corresponds to the vector i indicated by SEQ_INPUT_VECTOR $1 = i$ in the input block "DERIVATIVES" and VECTOR2 to the vector j indicated by SEQ_INPUT_VECTOR $2 = j$. If this last option is not used array VECTOR2 may not be filled. Array INDEX2 may be used to find the degrees of freedom in VECTOR j , corresponding to the element. The i^{th} local degree of freedom in the element can be found from VECTOR j (INDEX2(i)).

A common way to extract the solution in the nodal points of the element is to define a help array U of size *icount*. The following piece of code copies the old solution from array VECTOR1 into array U:

```
u(1:icount) = vector1(index2(1:icount))
```

INDEX1 In this integer array of length INPELM, the point numbers of the nodal points in the element are stored. The user needs these numbers to compute the co-ordinates of the nodal points of the element. INPELM is the number of nodal points in the element. See COOR.

INDEX2 In this integer array of length ICOUNT, the positions of the degrees of freedom in the element with respect to the arrays VECTOR1 and VECTOR2, are stored sequentially. The user needs this information in order to extract the values of these vectors in the nodal points.

Besides the parameters in the parameter list program SEPCOMP, (actually subroutine DERIV) communicates also with ELCERV by the common block CACTL:

```
INTEGER IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
+       NOTVC, IRELEM, NUSOL, NELEM, NPOINT
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
+       NOTVC, IRELEM, NUSOL, NELEM, NPOINT
```

The following parameters may be useful:

IELEM Element number.

ITYPE Problem definition number. This number is defined in the input block "PROBLEM" by ELGRP i = (type = n_i).

IELGRP Standard element sequence number. Boundary elements are skipped in the call of the element subroutines.

INPELM Number of nodal points in element.

ICOUNT Number of degrees of freedom in element.

IFIRST This parameter indicates if the element subroutine is called for the first time for the specific element group (IFIRST=0) or not (IFIRST=1). This parameter may be of help for experienced FORTRAN programmers in order to initialize parameters and even local arrays only once. Since FORTRAN 77 does not save local parameters it is necessary to use the "SAVE" statement if this option is utilized.

NOTMAT This parameter is not used.

NOTVC This parameter indicates if an element vector is identical to zero (NOTVC=1) or not (NOTVC=0) for all elements with standard element sequence number IELGRP. This parameter is the only parameter in common block CACTL the user is allowed to change in subroutine ELCERV.

If the element vector is identical zero for the complete element group, the user may indicate this by setting NOTVC = 1 in subroutine ELCERV.

NELEM Number of elements with standard element sequence number IELGRP in the mesh.

NPOINT Number of nodal points in the mesh.

NUSOL Number of degrees of freedom in the solution vector.

IRELEM Relative element number with respect to standard element sequence number IELGRP.

The parameters in CACTL are given a value by program SEPCOMP. These values may change from element to element and must not be changed by the user.

In order to distinguish between different element groups both the parameters IELGRP and ITYPE may be used.

Input

Program SEPCOMP gives the parameters ICHELD, IX and JDEGFD a value.

Program SEPCOMP fills the arrays COOR, INDEX1, INDEX2 and array VECTOR1 and VECTOR2 before the call of ELCERV.

All parameters in common block CACTL have got a value by program SEPCOMP.

Output

Array ELEMVC must have been filled by the user, depending on the value of NOTVC.

Interface

Subroutine `elcerv.f90` must be programmed as follows:

```

subroutine elcerv (  icheld, ix, jdegfd, coor, elemvc, iuser, user, &
                   vector1, vector2, index1, index2 )
implicit none

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision coor(ndim,*), elemvc(*), &
                user(*), vector1(*), vector2(*)
integer icheld, ix, jdegfd, iuser(*), index1(inpelm), index2(icount)

!       declarations of local variables
!       for example:

integer npelm, i, k, jcount
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm)

if ( itype==1 ) then

! --- possibly statements to fill x and u for type number 1
!   for example:

x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))

u(1:icount) = vector1(index2(1:icount))

! --- statements to fill array elemvc

jcount = ...
do i = 1, jcount
  elemvc(i) = "f(i)"
end do

else if ( itype==2 ) then

! --- the same type of statements for itype = 2, etcetera

end if
end

```

Remark:

For problems in complex variables (like the Helmholtz equation) one may declare `ELEMVC` and possibly `VECTOR1` as double complex arrays, which means that they are treated as double precision complex arrays.

4.7 Function subroutine ELINT

Description

The user-written subroutine ELINT is called by program SEPCOMP in the case that a volume integral must be computed and type numbers between 1 and 99 are used. This subroutine is called if the input block "STRUCTURE" is used in program SEPCOMP and in this block the command integral is given. ELINT is also called if the user runs his own main program and calls one of the volume integration subroutines.

Communication with SEPCOMP is performed with the parameters in the heading of the subroutine, as well as the parameters in common block CACTL. In program SEPCOMP a subroutine INTEGR is called which computes the actual integral as sum over all element integrals.

This subroutine INTEGR contains a loop over the element groups. For each element group it contains a loop over all elements in this group. If the element group corresponds to an element with type number 1 to 99, subroutine ELINT is called in the inner loop. Hence, subroutine ELINT is called for each element separately.

Call

```
VALUE = ELINT ( ICHELI, JDEGFD, COOR, IUSER, USER,
               VECTOR, INDEX1, INDEX2 )
```

Parameters

INTEGER ICHELI, JDEGFD, IUSER(*), INDEX1(*), INDEX2(*)

DOUBLE PRECISION COOR(ndim,*), USER(*), VECTOR(*)

ICHELI This parameter has got the value s from the command $ICHELI = s$ in the input block "INTEGRALS". ICHELI has been given this value by program SEPCOMP. The user is not allowed to change this value.

JDEGFD This parameter has got the value d from the command $DEGREE.OF.FREEDOM = d$ in the input block "INTEGRALS". JDEGFD has been given this value by program SEPCOMP. The user is not allowed to change this value.

COOR Double precision two-dimensional array of size $NDIM \times$ number of points, where $NDIM$ is the dimension of the space. So for a two-dimensional problem COOR must be declared as COOR(2,*).

To find the co-ordinates of the nodes of the element, array INDEX1 must be used. The x -co-ordinate of the i^{th} local point in the element is given by COOR(1,INDEX1(i)), the y -co-ordinate by COOR(2,INDEX1(i))

A common way to extract the co-ordinates of the element is to define a help array X of size $ndim \times npelm$, where $npelm$ denotes the maximum number of nodes in the elements. The following piece of code copies the co-ordinates from array coor into array X:

```
x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))
```

ELINT The user must give ELINT the value of the integral computed over the element.

IUSER,USER These arrays are used by SEPCOMP to store information of the coefficients for the computation. The storage of IUSER and USER is described in the manual Standard Problems. For simple problems using SEPCOMP, the storage in IUSER and USER may be too complicated to be used.

If the user calls subroutine INTEGR in his own SEPRAN program, he may fill IUSER and USER in his own way, since these arrays are passed undisturbed from main program to element subroutine.

VECTOR In this array the vector to be integrated as given in the command INTEGRAL, vector = j has been stored. Array INDEX2 may be used to find the degrees of freedom in VECTOR, corresponding to the element. The i^{th} local degree of freedom in the element can be found from VECTOR(INDEX2(i)).

A common way to extract the solution in the nodal points of the element is to define a help array U of size *icount*. The following piece of code copies the old solution from array VECTOR into array U:

```
u(1:icount) = vector(index2(1:icount))
```

INDEX1 In this integer array of length INPELM, the point numbers of the nodal points in the element are stored. The user needs these numbers to compute the co-ordinates of the nodal points of the element. INPELM is the number of nodal points in the element. See COOR.

INDEX2 In this integer array of length ICOUNT, the positions of the degrees of freedom in the element with respect to array VECTOR are stored sequentially. The user needs this information in order to extract the values of these vectors in the nodal points.

Besides the parameters in the parameter list program SEPCOMP, (actually subroutine INTEGR) communicates also with ELINT by the common block CACTL:

```
INTEGER IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
      NOTVC, IRELEM, NUSOL, NELEM, NPOINT
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
      NOTVC, IRELEM, NUSOL, NELEM, NPOINT
```

The following parameters may be useful:

IELEM Element number.

ITYPE Problem definition number. This number is defined in the input block "PROBLEM" by $ELGRP_i = (\text{type} = n_i)$.

IELGRP Standard element sequence number. Boundary elements are skipped in the call of the element subroutines.

INPELM Number of nodal points in element.

ICOUNT Number of degrees of freedom in element.

IFIRST This parameter indicates if the element subroutine is called for the first time for the specific element group (IFIRST=0) or not (IFIRST=1). This parameter may be of help for experienced FORTRAN programmers in order to initialize parameters and even local arrays only once. Since FORTRAN 77 does not save local parameters it is necessary to use the "SAVE" statement if this option is utilized.

NOTMAT This parameter is not used.

NOTVC This parameter indicates if an element integral is identical to zero (NOTVC=1) or not (NOTVC=0) for all elements with standard element sequence number IELGRP. This parameter is the only parameter in common block CACTL the user is allowed to change in subroutine ELINT.

If the element integral is identical zero for the complete element group, the user may indicate this by setting NOTVC = 1 in subroutine ELINT.

NELEM Number of elements with standard element sequence number IELGRP in the mesh.

NPOINT Number of nodal points in the mesh.

NUSOL Number of degrees of freedom in the solution vector.

IRELEM Relative element number with respect to standard element sequence number IELGRP.

The parameters in CACTL are given a value by program SEPCOMP. These values may change from element to element and must not be changed by the user.

In order to distinguish between different element groups both the parameters IELGRP and ITYPE may be used.

Input

Program SEPCOMP gives the parameters ICHELI and JDEGFD a value.

Program SEPCOMP fills the arrays COOR, INDEX1, INDEX2 and array VECTOR before the call of ELINT.

All parameters in common block CACTL have got a value by program SEPCOMP.

Output

ELINT must have been given a value by the user, depending on the value of NOTVC.

Interface

Subroutine elint.f90 must be programmed as follows:

```

function elint ( icheli, jdegfd, coor, iuser, user, &
                vector, index1, index2 )
implicit none

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision coor(ndim,*) user(*), vector(*)
integer icheli, jdegfd, iuser(*), index1(inpelm), index2(icount)

!      declarations of local variables
!      for example:

integer npelm, i, k
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm)

if ( itype==1 ) then

!      --- possibly statements to fill x and u for type number 1
!      for example:

x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))

u(1:icount) = vector(index2(1:icount))

```

```
!      --- statements to compute the integral and to store it in elint
      elint = integral to be computed
      else if ( itype==2 ) then
!      --- the same type of statements for itype = 2, etcetera
      end if
      end
```

4.8 Subroutine ELSTRM

Description

The user-written subroutine ELSTRM is called by program SEPPOST in the case that the stream function must be computed and type numbers between 1 and 99 are used.

Communication with SEPPOST is performed with the parameters in the heading of the subroutine, as well as the parameters in common block CACTL. In program SEPPOST a subroutine STREAM is called which computes the stream function in the following way.

STREAM starts in a node. An element in which this node is found and subroutine ELSTRM is called for this element. It is supposed that ELSTRM returns with the values of the stream function in the other nodes computed from the value in the given node. This can be done by integrating the velocity. Next this process is repeated from the following nodes until all nodes are considered. Hence, subroutine ELSTRM is called for each element separately.

Call

```
CALL ELSTRM ( ICHELS, JNODP, STRMJ, ELEMVC, COOR, INDEX1, VECTOR, INDEX2, INDEX3 )
```

Parameters

INTEGER ICHELS, JNODP, INDEX1(*), INDEX2(*), INDEX3(*)

DOUBLE PRECISION STRMJ, ELEMVC(*), COOR(*ndim*,*), VECTOR(*),

ICHELS Choice parameter in the call of subroutine STREAM. This parameter is transmitted undisturbed by subroutine STREAM and can be used at the users choice.

JNODP Nodal point number of the vertex where the stream function or potential is known.

STRMJ Value of the stream function or potential in nodal point JNODP.

ELEMVC In this array the user must store the value of the stream function or potential in the nodal points. The length of ELEMVC is equal to the number of nodal points in the element (INPELM).

COOR Double precision two-dimensional array of size NDIM \times number of points, where NDIM is the dimension of the space. So for a two-dimensional problem COOR must be declared as COOR(2,*).

To find the co-ordinates of the nodes of the element, array INDEX1 must be used. The x -co-ordinate of the i^{th} local point in the element is given by COOR(1,INDEX1(i)), the y -co-ordinate by COOR(2,INDEX1(i))

A common way to extract the co-ordinates of the element is to define a help array X of size $ndim \times npelm$, where $npelm$ denotes the maximum number of nodes in the elements. The following piece of code copies the co-ordinates from array coor into array X:

```
x(1:ndim,1:inpelm) = coor(1:ndim,index1(1:inpelm))
```

INDEX1 In this integer array of length INPELM, the point numbers of the nodal points in the element are stored. The user needs these numbers to compute the co-ordinates of the nodal points of the element. INPELM is the number of nodal points in the element. See COOR.

VECTOR In this array the vector, from which the potential or stream function must be computed, has been stored. Array INDEX2 may be used to find the degrees of freedom in VECTOR, corresponding to the element. The i^{th} local degree of freedom in the element can be found from VECTOR(INDEX2(i)).

A common way to extract the solution in the nodal points of the element is to define a help array U of size *icount*. The following piece of code copies the old solution from array VECTOR into array U:

```
u(1:icount) = vector(index2(1:icount))
```

INDEX2 In this integer array of length ICOUNT, the positions of the degrees of freedom in the element with respect to array VECTOR are stored sequentially. The user needs this information in order to extract the values of these vectors in the nodal points.

INDEX3 Integer array with, cumulatively, the number of degrees of freedom in the nodal points of the actual element IELEM. The length of INDEX3 is equal to INPELM+1. INDEX3 is filled as follows: INDEX3(1) = 0 INDEX3(INPELM+1) = ICOUNT the number of degrees of freedom in point *j* of the actual element is equal to INDEX3(*j* + 1) - INDEX3(*j*), *j*=1, ... , INPELM.

Besides the parameters in the parameter list program SEPPOST, (actually subroutine STREAM) communicates also with ELSTRM by the common block CACTL:

```
INTEGER IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
      NOTVC, IRELEM, NUSOL, NELEM, NPOINT
COMMON /CACTL/ IELEM, ITYPE, IELGRP, INPELM, ICOUNT, IFIRST, NOTMAT,
      NOTVC, IRELEM, NUSOL, NELEM, NPOINT
```

The following parameters may be useful:

IELEM Element number.

ITYPE Problem definition number. This number is defined in the input block "PROBLEM" by ELGRPi = (type = ni).

IELGRP Standard element sequence number. Boundary elements are skipped in the call of the element subroutines.

INPELM Number of nodal points in element.

ICOUNT Number of degrees of freedom in element.

IFIRST This parameter indicates if the element subroutine is called for the first time for the specific element group (IFIRST=0) or not (IFIRST=1). This parameter may be of help for experienced FORTRAN programmers in order to initialize parameters and even local arrays only once. Since FORTRAN 77 does not save local parameters it is necessary to use the "SAVE" statement if this option is utilized.

NOTMAT This parameter is not used.

NOTVC This parameter is not used.

NELEM Number of elements with standard element sequence number IELGRP in the mesh.

NPOINT Number of nodal points in the mesh.

NUSOL Number of degrees of freedom in the solution vector.

IRELEM Relative element number with respect to standard element sequence number IELGRP.

The parameters in CACTL are given a value by program SEPPOST. These values may change from element to element and must not be changed by the user.

In order to distinguish between different element groups both the parameters IELGRP and ITYPE may be used.

Input

Program SEPPOST gives the parameters ICHELS, JNODP and STRMJ a value.

Program SEPPOST fills the arrays COOR, INDEX1, INDEX2, INDEX3 and array VECTOR before the call of ELSTRM.

All parameters in common block CACTL have got a value by program SEPPOST.

Output

Array ELEMVC must have been filled by the user in the order of the nodal points as stored in array INDEX1. Hence $ELEMVC(n) = \psi(n)$, $n = 1, \dots, INPELM$.

Method

The stream function or potential can be computed by integration of the velocity along the boundaries of the elements. To that end the relations: $\frac{\partial \psi}{\partial x} = -v_2$, $\frac{\partial \psi}{\partial y} = v_1$ in the case of cartesian co-ordinates and stream function computation and $\mathbf{v} = \nabla \phi$ in the case of the potential. From these relations it follows that: $\psi_2 - \psi_1 = \int_{\mathbf{x}_1}^{\mathbf{x}_2} \mathbf{v} \cdot \mathbf{n} d\Gamma$ and $\phi_2 - \phi_1 = \int_{\mathbf{x}_1}^{\mathbf{x}_2} \mathbf{v} \cdot \mathbf{t} d\Gamma$. For a 3-point triangle or a 4-point quadrilateral \mathbf{v} is (bi)linear, and therefore a trapezoid rule is sufficient to integrate the equations over the element. For a quadratic element Simpson's rule must be used. In the case of axi-symmetric co-ordinates a more accurate integration rule is necessary, for example Simpson's rule in the linear case and a 3-point Gauss rule in the quadratic case. This is also true for isoparametric elements.

Interface

Subroutine `elstrm.f90` must be programmed as follows:

```
subroutine elstrm ( ichels, jnodp, strmj, elemvc, coor, index1, &
                  vector, index2, index3 )
implicit none

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision strmj, elemvc(*), coor(ndim,*) vector(*),
integer ichels, jnodp, index1(inpelm), index2(icount), index3(*)

!      declarations of local variables
!      for example:

integer npelm, i, k
parameter ( npelm = 4 )
double precision x(ndim,npelm), u(npelm)

if ( itype==1 ) then

!      --- possibly statements to fill x and u for type number 1
!      for example:

      do k = 1, ndim
        do i = 1, inpelm
          x(k,i) = coor(k,index1(i))
        end do
      end do

      do k = 1, icount, 2
        u(k) = vector1(index2(k)*2-1)
        v(k) = vector1(index2(k)*2)
      end do

!      --- statements to compute the stream function and fill it in elemvc

else if ( itype==2 ) then

!      --- the same type of statements for itype = 2, etcetera

end if
end
```

Example

As a simple example consider a subroutine ELSTRM that can handle 3-point triangles and 4-point quadrilaterals. It is supposed that the vector $\mathbf{v} = (v_1, v_2)$ is composed of the first two degrees of freedom in the nodal points.

```

subroutine elstrm ( ichels, jnodp, strmj, elemvc, coor, index1, &
                  vector, index2, index3 )
implicit none

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision strmj, elemvc(*), coor(ndim,*) vector(*),
integer ichels, jnodp, index1(inpelm), index2(icount), index3(*)

!       declarations of local variables

integer npelm, i, k
parameter ( npelm = 4 )
double precision x(npelm), y(npelm), u(npelm), v(npelm)
integer ind(2*inpelm)

x(1:inpelm) = coor(1,index1(1:inpelm))
y(1:inpelm) = coor(2,index1(1:inpelm))
u(1:inpelm) = vector(1,index2(1:inpelm))
v(1:inpelm) = vector(2,index2(1:inpelm))

!       --- find the vertex where the stream function is given

do i = 1, inpelm
  if ( jnodp == index1(i) ) ivert = i
end do

!       --- prepare an index array in order to be able to program a do loop
!       for the computation of the stream function or potential in
!       vertices that are not given

do i = 1, inpelm
  ind(i) = i
  ind(inpelm+i) = i
end do

!       --- compute the stream function or potential for cartesian co-ordinates
!       with help of the trapezoid rule starting from ivert

elemvc(ivert) = strmj
do i = ivert, ivert + inpelm - 1
  if (ichels==1) then

!       --- when ichels=1 the stream function is computed

      elemvc(ind(i+1)) = elemvc(ind(i)) - &
        (y(ind(i))-y(ind(i+1))) * (u(ind(i))+u(ind(i+1))) - &
        (x(ind(i))-x(ind(i+1))) * (v(ind(i))+v(ind(i+1))) / 2d0

```

```
    else
!      --- when ichels=2 the potential is computed
        elemvc(ind(i+1)) = elemvc(ind(i)) - &
            (x(ind(i))-x(ind(i+1))) * (u(ind(i))+u(ind(i+1))) + &
            (y(ind(i))-y(ind(i+1))) * (v(ind(i))+v(ind(i+1))) / 2d0
    endif
end do ! i = ivert, ivert + inpelm - 1
end
```

Remarks

- The vector \mathbf{t} is defined as $(\frac{(x_{i+1}-x_i)}{l_i}, \frac{(y_{i+1}-y_i)}{l_i})$ with l_i the length of the side of the element between the the nodal points i and $i + 1$ (internal numbering!). The unit normal is taken in the outward direction. Therefore it is necessary to know whether the internal numbering of the nodal points is counter-clockwise or not. The sign s of the normal can be determined in the following way: $\Delta = (y_1 - y_2) \times (x_3 - x_2) - (y_2 - y_3) \times (x_2 - x_1)$ $S = \text{sign } \Delta$
Then \mathbf{n} is defined by $S(\frac{(y_{i+1}-y_i)}{l_i}, -\frac{(x_{i+1}-x_i)}{l_i})$
- SEPRAN contains the standard subroutine ELS400 for the computation of the stream function or potential for 3, 6 or 7-point isoparametric triangles, 4, 8 or 9-point isoparametric quadrilaterals for standard problems with e.g. type numbers 400, 402 and 404. If the user has a problem for which the v_1, v_2 to be integrated are the first two degrees of freedom in a nodal point, subroutine ELS400 can be used in the following way:

```

subroutine elstrm ( ichels, jnodp, strmj, elemvc, coor, index1, &
                  vector, index2, index3 )
implicit none

include 'SPCOMMON/cact1'

integer ndim
parameter ( ndim = 2 )
double precision strmj, elemvc(*), coor(ndim,*) vector(*),
integer ichels, jnodp, index1(inpelm), index2(icount), index3(*)

! --- do not disturb the common area /cact1/, but change itype temporarily
!
!   itype = 400 cartesian co-ordinates
!   itype = 402 polar co-ordinates
!   itype = 404 axi-symmetric co-ordinates
!
!   ichels = 1: stream function
!   ichels = 2: potential

jtype = itype
itype = 400
call els400 ( ichels, jnodp, strmj, elemvc, coor, index1, vector, &
             index2,index3 )

! --- reset itype

itype = jtype
end

```

5 The postprocessing part of SEPRAN

5.1 Introduction

In the post processing part of SEPRAN, the output of the solution and derived quantities is produced in a readable (visible) form. Integrals over quantities, integrals over boundaries etc. may be computed and printed or plotted. The output generated may be produced in either print or plot form. Print output is both written to the screen or to a file for later reproducing on a printer.

The post processing is performed by the main program SEPPOST. It requires two types of input.

First it uses some files produced by the mesh generation part (meshoutput) and the computational part (sepcomp.out).

If the file sepcomp.out is not available it is only possible to plot the mesh with SEPPOST. All other commands are suppressed.

Secondly it requires input from the standard input file. (An interactive version will be available in due course).

The input of SEPPOST is described in the next paragraphs.

5.2 describes the general shape of the input, including the so-called compute commands, the define commands and the reset commands,

5.3 treats the various print commands,

5.4 the plot commands and

5.5 is devoted to some special commands with respect to time-dependence.

5.2 General input for program SEPPOST

The input for the post processing part must be opened with the COMMAND POSTPROCESSING and must be closed with the COMMAND END.

COMMAND and DATA records.

Options are indicated between the square brackets [and].

POSTPROCESSING (mandatory)

COMMAND record: opens the input for program SEPPOST.

May be followed by DATA records of the shape:

```
NAME V0 = potential
NAME V1 = gradient of potential
.
.
```

These records identify the vectors V0, V1 etc. as defined by the input block "OUTPUT" in the input file of SEPCOMP, with the names potential, gradient of potential etc. These names are used in the output subroutines, for example in the heading of the prints. If no names are given, the names defined in sepcomp are used. So in general there is no need to define the names.

The actual post processing records have the following shape:

```
PRINT Vi . . .
PLOT . . .
COMPUTE Vi . . .
Vi = alpha * Vi + beta
DEFINE . . .
RESET . . .
TIME = . . .
TIME HISTORY . . .
READ_MESH . . .
PRESENT_MESH . . .
INTERPOLATE . . .
FILL COEFFICIENTS
PUT_IN_AVS_FILE . . .
ADD_TO_COOR Vi
INTERSECT_MESH . . .
```

The input must be ended by:

END (mandatory)

End of the input for program SEPPOST.

The actual post processing commands may be given in any order, with the restriction that vectors V_i to be printed or plotted must have been defined before, for example by a compute statement.

The SET commands, as treated in Section 1.4, may be used anywhere in the input. They become activated from the moment they have been read.

PRINT commands are treated in 5.3.

PLOT commands in 5.4.

DEFINE and RESET commands

The DEFINE and RESET commands are used to set or reset of some defaults for printing or plotting. Their general syntax is:

```
define plot parameters = . . .
define colour table = . . .
reset plot parameters
reset colour table
```

With the define plot parameters statement, the user defines new defaults for the plot parameters. These defaults remain valid until the user resets plot parameters with the reset command, or a new define plot parameters is read. For a description of the plot parameters the user is referred to 5.4.

Remark: one of the plot parameters: region = (xmin, xmax, ymin, ymax) is also used for the print commands. So if this parameter is also given in the define plot parameters, it affects the print output.

The statement define color table defines the color numbers for colored plots. See 5.4.

COMPUTE commands

The COMPUTE command is used to define a vector V_i as function of an already available vector V_j . Using the same number i in a new COMPUTE statement redefines vector V_i .

Instead of V_i or V_j , the user may also use `name_of_vector` provided that name already exists. With respect to the V_i immediately following `compute` the name does not have to exist. If the name does not exist the name is added to the list of vectors.

The general syntax for the compute statements is:

```
compute [Vi =] stream function Vj [ start node = s ] //
    [ stream function value = f]//
    [skip element groups ( g_1, g_2, ... )]
compute Vi = velocity profile Vj [degfd=k] origin=(0_x, 0_y)] [angle = a ]
compute Vi = intersection Vj [degfd=k] [origin=(0_x, 0_y)] [angle = a ]
compute Vi = intersection Vj [degfd=k] [numbunknowns=n] plane(ax+by+cz=d)//
    [transformation = tr] [origin = (0_x, 0_y, 0_z)] //
    [tang_x = (tx_1, tx_2, tx_3), tang_y = (ty_1, ty_2, ty_3)] //
    [type_tang = t]
compute [Vi =] mach_number, [velocity_vector = vj,] [enthalpy_vector = vj,]//
    gamma = g
compute [Vi =] total_enthalpy
compute [Vi =] stagnation_pressure
compute [Vi =] scaled_pressure
compute [Vi =] entropy
compute Vi = function_of Vj, type_func = f, [degfd=k]
```

The parameter i following `compute V` may be a number, but it is preferred to use a name `compute entropy entropy1`. If the name following `compute xxx` has not yet been defined it is added to the list of vector names. If no name is given the default name is used if available.

The following names are available:

```
stream_function
mach_number
total_enthalpy
stagnation_pressure
scaled_pressure
entropy
normal_stress
```

Meaning of these commands:

compute $V_i = \text{stream function } V_j$ means that vector V_i must be computed as stream function from the velocity vector V_j . It is advised to use a name instead of a number or no name at all.

If V_i is omitted the result vector is stored in a vector with name `stream_function`. If this name does not exist, it is added to list of vector names.

V_j is supposed to be a vector with the two x and y velocity components as first and second component in each nodal point.

If start node = s is given, the value of the stream function is set in nodal point s (default $s=1$).

stream function value = f defines the value in the start node s (default $f=0$).

With the option `skip element groups (g_1, g_2, ...)` the user may skip the corresponding element groups. That means that the values in the corresponding part of the mesh are not used for the computation of the stream function.

compute $V_i = \text{velocity profile } V_j$ defines vector V_i as a function given by one of the velocity components (`degfd=k`, default $k=1$) along the line with origin (O_x, O_y) (default $(0,0)$) under an angle of a degrees (default $a=0$). This possibility is only permitted for two-dimensional vector fields. The intersection of the line with the mesh is computed and the solution is interpolated onto this line.

Remark: at this moment the method is sensitive to round off, which means that if a line coincides with the boundary of the mesh, only some parts or no part at all may be found in the intersection. In that case it is recommended to shift the line over a small distance.

compute $V_i = \text{intersection } V_j$ If the mesh corresponding to V_j is a 2D mesh, solution V_j along the line with origin (O_x, O_y) (default $(0,0)$) under an angle of a degrees (default $a=0$). k is defined by `degfd=k` (default $k=1$). This possibility is only available for functions defined on a two-dimensional mesh. Furthermore this possibility is completely identical to the preceding one, including the remark given before.

If the mesh corresponding to V_j is a 3D mesh, solution V_j in the plane defined by $ax+by+cz=d$. The 3D region is intersected by the plane and a new 2D mesh consisting of linear triangles is created. The vector V_j is interpolated on this new 2D mesh, and the interpolation is called V_i . With the function V_i all standard postprocessing commands may be executed including the intersection with a line. Default values for a, b, c and d are zero. SEPPOST also recognizes planes equal to 1 or -1, depending on the preceding sign. All values a, b, c and d equal to zero is not allowed.

Note that $x-y=0$ is not allowed but must be replaced by $x -1 y = 0$.

k is defined by `degfd=k` (default $k=1$).

`numbunknowns = n` defines the number of degrees of freedom that are interpolated. Hence the degrees of freedom $k, k+1, \dots, K+n-1$ are interpolated. The default value is $n = 1$.

`transformation = tr` may only be used in combination with `numbunknowns ≥ 3` . It defines how an interpolated vector must be transformed. The following values for tr are available:

`Cartesian` The Cartesian vector components are kept, i.e. no transformation is applied.

`plane_oriented` The first 3 components of the vector are transformed such that the third component is perpendicular to the plane and the first two components are orthogonal components within the plane. All other components are not transformed.

The options `tang_x = (tx_1, tx_2, tx_3)`, `tang_y = (ty_1, ty_2, ty_3)` and `type_tang = t` may be used to define the axis in the intersection plane explicitly. Both possibilities are mutually exclusive.

By giving `tang_x = (tx_1, tx_2, tx_3)`, `tang_y = (ty_1, ty_2, ty_3)` the user explicitly defines the unit vectors in the intersection plane and hence defines the axis. If `tang_x` is given also `tang_y` must be given. Furthermore both unit vectors must be

perpendicular and be in the plane.

An alternative is to use the option `type_tang = t`, where for t the following values are available.

xy The axis are in the x-y plane. The first axis is the x-axis the second one the y-axis. This is only allowed if the plane is defined as $z = \text{constant}$.

yz The axis are in the y-z plane. The first axis is the y-axis the second one the z-axis. This is only allowed if the plane is defined as $x = \text{constant}$.

xz The axis are in the x-z plane. The first axis is the x-axis the second one the z-axis. This is only allowed if the plane is defined as $y = \text{constant}$.

proj_xy The axis are formed by projecting the x-axis and y-axis onto the given plane in that sequence. This is only possible if the coefficient for z in the plane is unequal to 0.

proj_yz The axis are formed by projecting the y-axis and z-axis onto the given plane in that sequence. This is only possible if the coefficient for x in the plane is unequal to 0.

proj_xz The axis are formed by projecting the x-axis and z-axis onto the given plane in that sequence. This is only possible if the coefficient for y in the plane is unequal to 0.

If neither one of these options is given the default is used, which implies that the tangential vectors are computed by the program from the definition of the plane. The consequence may be that the direction of the axis is different from what one expects.

`origin = (0_x, 0_y, 0_z)` defines the origin of the axis. The directions of the unit vectors are defined by the tangential vectors.

If omitted the default origin is used, which means that the origin is computed by the program from the definition of the plane.

compute $V_i = \text{MACH_NUMBER}$ indicates that the Mach number must be computed in each point of the mesh, using the formula:

$$M = \frac{u}{a} = \frac{\sqrt{u_x^2 + u_y^2}}{\sqrt{(\gamma - 1)h}} \quad (5.2.2)$$

where u is the length of the velocity vector in a point, γ a gas constant, h is the enthalpy, and $\mathbf{u} = (u_x, u_y)^T$ is the velocity vector.

`gamma = γ` defines the gas constant.

The default value is 1.4.

`enthalpy_vector = Vj` defines in which vector Vj the enthalpy has been stored.

The default value is the vector with name `enthalpy` and if this name does not exist `j=3`.

`velocity_vector = Vj` defines in which vector Vj the velocity has been stored.

The default value is the vector with name `velocity` and if this name does not exist `j=5`.

`Vi` defines the name or number of the result vector.

If V_i is omitted the result vector is stored in a vector with name `mach_number`. If this name does not exist, it is added to list of vector names.

compute $V_i = \text{TOTAL_ENTHALPY}$ indicates that the total enthalpy must be computed in each point of the mesh, using the formula:

$$H = h + \frac{1}{2}u^2, \quad (5.2.3)$$

where h is the enthalpy and $u^2 = (u_x^2 + u_y^2)$ is the square of the length of the velocity vector.

The velocity vector is either the vector with name `velocity` or if this name does not exist `j=5`.

The enthalpy vector is either the vector with name `enthalpy` or if this name does not exist `j=3`.

`Vi` defines the name or number of the result vector.

If `Vi` is omitted the result vector is stored in a vector with name `total_enthalpy`. If this name does not exist, it is added to list of vector names.

compute `Vi = STAGNATION_PRESSURE` indicates that the stagnation pressure (also called: total pressure) must be computed in each point of the mesh, using the formula:

$$p_t = \left(1 + \frac{\gamma - 1}{2} M^2\right)^{\gamma/\gamma-1} p, \quad (5.2.4)$$

where M is the Mach number, $u^2 = (u_x^2 + u_y^2)$ the square of the length of the velocity vector and p the pressure.

The pressure vector is either the vector with name `pressure` or if this name does not exist `j=2`.

The Mach number vector is either the vector with name `mach-number` or if this name does not exist `j=10`.

`Vi` defines the name or number of the result vector.

If `Vi` is omitted the result vector is stored in a vector with name `stagnation_pressure`.

If this name does not exist, it is added to list of vector names.

compute `Vi = SCALED_PRESSURE` indicates that the pressure coefficient must be computed in each point of the mesh, using the formula:

$$c_p = \frac{p - p_\infty}{\frac{1}{2} \rho_\infty u_\infty^2}, \quad (5.2.5)$$

where p is the pressure, and the symbol ∞ indicates the free-stream values (usually the values at the inlet).

The pressure vector is either the vector with name `pressure` or if this name does not exist `j=2`.

`Vi` defines the name or number of the result vector.

If `Vi` is omitted the result vector is stored in a vector with name `scaled_pressure`. If this name does not exist, it is added to list of vector names.

compute `Vi = ENTROPY` indicates that the entropy must be computed in each point of the mesh, using the formula:

$$S = \ln(p/\rho^\gamma), \quad (5.2.6)$$

where p is the pressure and ρ the density.

The pressure vector is either the vector with name `pressure` or if this name does not exist `j=2`.

The density vector is either the vector with name `density` or if this name does not exist `j=4`.

`Vi` defines the name or number of the result vector.

If `Vi` is omitted the result vector is stored in a vector with name `entropy`. If this name does not exist, it is added to list of vector names.

compute `Vi = function_of Vj` indicates that the vector `Vi` is computed as a component-wise function of the the vector `Vj`. The vector `Vi` gets the same structure and size as `Vj`.

`degfd = k` defines the degree of freedom per point to be considered, if 0 or omitted all degrees of freedom are used.

`type_func = f` defines the type of function that must be used to compute a vector as function of another vector.

Possible values

`log` the function is a logarithm with base 10

Compute statements only define the vector V_i , which means that the actual computation is performed only if necessary. At most 26 vectors V_i , including V_0 are allowed in SEPPOST.

$$\mathbf{V}_i = \alpha \mathbf{V}_i + \beta$$

This command is meant for scaling of solutions.

The vector indicated by sequence number i is multiplied by α and the constant β is added to the vector. This command works on all components at a time, so individual multiplication of a special degree of freedom is impossible. The vectors on left and right-hand side must be the same. The sequence of the individual items in this command can not be changed, however, the plus sign may be replaced by a minus sign.

Default value for *alpha* is 1, and for *beta* is 0.

Example, suppose the temperature vector **Temperature** is scaled by subtracting a constant T_0 and multiplying the result by a factor ΔT . So the scaled temperature is equal to $T_{scaled} = \Delta T(T - T_0)$.

If we want to print or plot the un-scaled temperature, we have to use the command:

```
Temperature = {1/delta_T} Temperature + T_0
```

where **delta_T** and **T_0** are constants stored in the constants block.

To scale this vector again we need two steps:

```
Temperature = Temperature - T_0
Temperature = delta_T * Temperature
```

The ***** sign is optional and has no meaning.

Special commands

Except the commands mentioned before there are some other special commands that may be used.

These commands have the following shape:

```
READ_MESH i, file = 'name_of_file'
PRESENT_MESH = j
INTERPOLATE Vj, mesh_in = k, mesh_out = l
FILL COEFFICIENTS
PUT_IN_AVS_FILE Vi
ADD_TO_COOR Vi, factor = f
```

These commands have the following meaning:

READ_MESH i indicates that a mesh is read with sequence number i . The standard mesh always gets sequence number 1, and in case of intersections also new meshes are created, with new numbers.

The sequence number i may not have been used before.

This keyword must be followed by the command **file = 'name_of_file'**, where **name_of_file** is the name of the new mesh file. If the standard mesh file **meshoutput** is formatted, the new one must also be formatted. Alternatively if one of them is unformatted then also the other one must be unformatted.

Besides that there is an extra restriction for the new mesh file: the number of element groups must be the same as for the first mesh. The reason is that the problem description corresponding to the first mesh (except for boundary conditions), is reused for the second mesh.

PRESENT_MESH = j sets the actual mesh sequence number to j . All commands following this statement are carried out on the second mesh.

mark that this makes only sense if the vectors to be operated are defined on this second mesh.

INTERPOLATE V_j interpolates the vector V_j from one mesh to another.

The mesh from which interpolation must take place is defined by `mesh_in = k`, where k must correspond to an existing mesh.

The mesh to which the vector is interpolated is defined by `mesh_out = 1`.

FILL COEFFICIENTS is a very special command that is only needed in the rare case that coefficients are needed for the postprocessing. This may be for example the case to define a porosity in particle tracking.

If this command is used, it must be followed immediately by a input block for the coefficients as defined in Section 3.2.6.

PUT_IN_AVS_FILE V_i write the vector V_i to a file in avs input format in the same way as defined in Section 3.2.13.

ADD_TO_COOR V_i , `factor = f` With this command you may change the coordinates of the mesh in the following way: $\mathbf{x} = \mathbf{x} + f\mathbf{u}$.

Here \mathbf{u} is the displacement vector corresponding to V_i . Of course this vector must have exactly NDIM degrees of freedom per point, with NDIM the dimension of the space.

So this option is meant to construct a distorted mesh. The factor f can be used as stretching factor.

In order to get the original coordinates, use the same command with a factor $-f$.

INTERSECT_MESH, `plane = (...)`, `file_mesh_out = '...'`, `file_solut_out = '...'` This command is meant to intersect the 3d mesh with a plane, just as in compute $V_i =$ intersection V_j ...

The plane must be defined in exactly the same way, for example ($z=0$).

The main difference is that in this particular case all solutions are interpolated to the new mesh. Besides that the new 2d mesh is written to a file defined by the name following `file_mesh_out` and the interpolated solutions to a file defined by the name following `file_solut_out`. Both file names must be given between quotes. The mesh file has exactly the same structure as the file `meshoutput` and the other file as `sepcomp.out`. Of course in this case for a 2d mesh. So viewing of the interpolated results is only possible by a new call to `seppost`.

After writing the 2d mesh and solutions, the interpolation is removed.

5.3 Print commands for program SEPPOST

The general input for the program SEPPOST is described in 5.2. This paragraph is devoted to the available print commands.

At this moment only two print commands are available. The syntax of the print commands is:

Options are indicated between the square brackets [and].

```
PRINT Vi, options
PRINT BOUNDARY FUNCTION Vi [,options] [,boundary_description]
```

The following options are available:

```
region = (xmin,xmax,ymin,ymax,zmin,zmax)
points = p1, p2, p3, ...
curves = c1, c2, cn, ...
surfaces = s1, s3, s5, ...
degfd = k
normal_component
tangential_component
suppress_coordinates
suppress_header
suppress_nodes
sequence = (y)
polar_system
equidistant_grid, distance = ( dx, dy, dz )
```

These options have the following meaning:

- region** If this option is used only the points within the region $xmin \leq x \leq xmax, ymin \leq y \leq ymax, zmin \leq z \leq zmax$ are printed.
- points** followed by P_i, P_j, P_k, \dots ensures that the printing of the solution is restricted to the user points given in the list.
- curves** followed by C_i, C_j, C_k, \dots ensures that the printing of the solution is restricted to the curves given in the list.
- surfaces** followed by S_i, S_j, S_k, \dots ensures that the printing of the solution is restricted to the surfaces given in the list.
- degfd** defines which degree of freedom must be printed. If omitted all degrees of freedom in the points requested are printed.
- normal_component** may only be used if curves or surfaces is given. Furthermore there must be at least $ndim$ unknowns in each point at the boundary to be printed, where $ndim$ is the dimension of space. In that case the normal component at the boundary is computed and printed. The vector from which the normal component is computed consists of the degrees of freedom 1, 2 and 3 (or 1 and 2 in R^2) in each point, except if degfd is given, in which case the degrees of freedom degfd, degfd+1 and degfd+2 are used.
- tangential_component** has the same meaning as normal_component, however, now with respect to the tangential component.
- suppress_coordinates** suppresses the printing of the co-ordinates in the output.
- suppress_header** suppresses the printing of the header.
- suppress_nodes** suppresses the printing of the node numbers.

sequence defines the ordering of the nodal points.

If no sequence is given the co-ordinates are ordered in increasing x-sequence and for constant x-value in increasing y-sequence. If sequence = (y) is given, then first increasing y-sequence and then increasing x-sequence is used (2D) or the sequence y, z, x in 3D. Sequence = (z) creates the sequence z, y, x (3D only).

polar_system is only available in 2D. The co-ordinates that are printed are translated from Cartesian to polar coordinates and the r and ϕ coordinates are printed.

equidistant_grid is only available for the command **print**, not for **print boundary function**.

When this command is used, the solution is interpolated to an equidistant grid defined by $(x_{min}, x_{max}) \times (y_{min}, y_{max}) \times (z_{min}, z_{max})$ and step size (dx, dy, dz) .

The interpolated function is printed.

Hence in this case both the options **REGION = (xmin, xmax, ymin, ymax, zmin, zmax)** and **DISTANCE = (dx, dy, dz)** are obligatory.

distance = (dx, dy, dz) may only be used in combination with the keyword **equidistant_grid**. It defines the step sizes for the equidistant grid.

Remarks:

The options **points**, **curves** and **surfaces** are mutually exclusive.

The options **normal_component** and **tangential_component** are also mutually exclusive. They may only be used in combination with the option **curves** or **surfaces**.

When **PRINT Vi** is given the complete vector is printed, together with the corresponding nodal point numbers and the co-ordinates. However, by giving the region or the option **points**, **curves** or **surfaces** the region may be limited.

The region to be printed may also be defined with the statement **DEFINE PLOT PARAMETERS region = (.....)** which affects both plots and prints.

PRINT BOUNDARY FUNCTION Vi, may be used to print a function defined along curves (2D and 3D) or surfaces(3d). The *boundary_description* may take one of the forms:

```
CURVES ( C1, C2, C3, C5, ... )
SURFACES ( S1, S2, S3, S5, ... )
```

The option **surfaces** may only be used if volume elements are present, the option **curves** if surface or volume elements exist.

The print along the curves is done in the direction of the curve and in the sequence given by the user. If negative curve numbers are used, the corresponding curve is used in reversed direction.

Furthermore the options are the same as for **print**. In fact since **PRINT Vi** allows for printing along curves or surfaces there is no need to use **PRINT BOUNDARY FUNCTION** anymore.

5.4 PLOT commands for program SEPPOST

The general input for the program SEPPOST is described in 5.2. This paragraph is devoted to the available plot commands.

The syntax of the plot commands is:

Options are indicated like this: [*options*].

Contour plots:

```
PLOT CONTOUR Vi [,degfd = k] [,plot parameters] [,nlevel = n] //
  [,levels = (q1,q2,...)] [,minlevel = min] [,maxlevel = max] //
  [,smoothing factor = s]
PLOT COLOURED LEVELS Vi [,degfd = k] [,plot parameters] [,nlevel = n] //
  [,levels = (q1,q2,...)] [,minlevel = min] [,maxlevel = max]
```

Vector plots:

```
PLOT VECTOR Vi [,degfd1 = k_1 ,degfd2 = k_2] [,plot parameters]
```

Function plots:

```
PLOT FUNCTION Vi [,plot parameters]
PLOT VELOCITY PROFILE Vi [degfd=k] [,plot parameters]//
  [origin = (0_x , 0_y)] [, angle = a]
PLOT INTERSECTION Vi [degfd=k] [,plot parameters] [origin = (0_x , 0_y)]//
  [, angle = a]
PLOT BOUNDARY FUNCTION Vi, curves (C1, C2, C3, C5, . . . ,Cn ) //
  [,plot parameters] [,degfd=k], [arc_scales = (smin, smax)]
```

3D plots:

```
3D PLOT Vi [,plot parameters] [,lindirec=1] [block_mode=m]//
  [intersect_angle=a] [,ground_value=g] [,nstep=n] [,transparent]
3D COLOURED PLOT Vi [,plot parameters]
```

Mesh plots:

```
PLOT MESH [,skip element groups ( g_1, g_2,... )] [,plot parameters] //
  [,renumbered nodes]
PLOT COLOURED_MESH [,skip element groups ( g_1, g_2,... )] [,plot parameters]
PLOT Vj MESH [,plot parameters]
PLOT CURVES [,plot parameters]
PLOT Vj CURVES [,plot parameters]
PLOT POINTS [,plot parameters]
```

User plot commands:

```
PLOT TEXT [,plot parameters]
PLOT POLYGON, coordinates( (x_1, y_1), (x_2, y_2), . . . , (x_n , y_n) )//
  [,plot parameters]
```

Other plot commands:

```

PLOT FIELD Vi [, plot parameters] [,PSTART = (x,y)] [,BNDPART = (j,k)]//
  [,FLUX = f] [, FROM] [, TOWARDS]
PLOT TRACK Vi [, plot parameters] [,TMAX = t]//
  PSTART = (x_1,y_1[,z_1], x_2,y_2[,z_2] ... x_n,y_n[,z_n])//
  [,NMARK=m] [,NVIEW=v] [,MESH] [,PRINT TRACK]//
  [TSTEP_PRINT = t] [,VALUES = (Vi, Vj, ...)], [EPS_TIME_STEP = eps],//
  [TYPE_TIME_INTEGRATION = h]

```

Special plot commands:

```

OPEN PLOT
CLOSE PLOT
PLOT IDENTIFICATION, TEXT = ' text to be plotted '

```

Meaning of these commands:

Contour plots:

PLOT CONTOUR means plot contour lines (lines with constant function value) for the given function.

degfd= k indicates that the k^{th} degree of freedom in each node is used as definition of the function, otherwise the first degree of freedom is used.

nlevel = n defines the number of contour levels. The default value is 11.

levels = (q_1, q_2, \dots) defines the contour levels explicitly. In this case there is no need to give the number of levels since this number is computed from the levels given.

minlevel defines the minimum contour level and

maxlevel defines the maximum contour level. These options should only be used if the contour levels are not given explicitly. If omitted, they are computed by the program.

smoothing factor defines the kind of smoothing that must be applied to the contour lines. $s = 0$ (default), means no smoothing, the contour lines are piece-wise linear. $s = 1$, computes a mean value between three succeeding values to filter some of the possible wiggles (Shuman filtering). For $s = 2, 3, 4$ and 5 a smooth spline is used to plot the contour lines. The higher the value of s , the smoother the spline. Although these pictures are much nicer for publication, the actual plot is in no way better than that of the non-smooth contours. Values larger than 5 are not permitted for s .

To suppress the legenda and the numbers along the contour lines use the option **nonumber** in case **nlevel** is given and the options **nonumber** and **noplot_legenda** if the levels are given explicitly.

PLOT COLOURED LEVELS V_i makes a colored contour plot of the array V_i , where the region between two levels is colored.

The colors used for the plotting are the standard colors defined for your system. These colors may be changed by the statement **define color table**. See **color table**.

degfd= k indicates that the k^{th} degree of freedom in each node is used as definition of the function, otherwise the first degree of freedom is used.

nlevel = n defines the number of contour levels. The default value is 11.

levels = (q_1, q_2, \dots) defines the contour levels explicitly. In this case there is no need to give the number of levels since this number is computed from the levels given.

minlevel defines the minimum contour level and

maxlevel defines the maximum contour level. These options should only be used if the contour levels are not given explicitly. If omitted, they are computed by the program.

Vector plots:

PLOT VECTOR V_i makes a vector plot of two of the degrees of freedom in each point. These components may be defined by $\text{degfd1} = k_1$, $\text{degfd2} = k_2$ respectively. If omitted $\text{degfd1} = 1$, and $\text{degfd2} = 2$ is assumed.

Function plots:

PLOT FUNCTION V_i , makes a plot of a one dimensional function. At this moment only vectors defined by `COMPUTE V_i = velocity profile` or `COMPUTE V_i = intersection`, (See 5.2) may be plotted by this command. If the solution corresponds to a one-dimensional mesh, the complete solution is plotted.

PLOT VELOCITY PROFILE V_i combines the commands `COMPUTE ... = velocity profile V_i ...` as described in 5.2 and the command `PLOT FUNCTION V_i` .

PLOT INTERSECTION V_i combines the commands `COMPUTE ... = intersection V_i ` as described in 5.2 and the command `PLOT FUNCTION V_i` .

In fact there is no difference between `PLOT VELOCITY PROFILE` and `PLOT INTERSECTION`

PLOT BOUNDARY FUNCTION V_i , `CURVES (C1,..., Cn)` may be used to plot a function defined along the curves C1 to Cn, where it is necessary that the end point of the i^{th} curve, is identical to the initial point of the $i + 1^{\text{th}}$ curve. If negative curve numbers are used, the corresponding curve is used in reversed direction.

If $\text{degfd} = k$ is given, the k^{th} degree of freedom is plotted; otherwise the first one is plotted.

The option `arc_scales=(smin,smax)` defines the lower and upper bound of the arc length.

If this option is not used, the arc length starts with the value 0 and the thus computed arc length is plotted as "x"-coordinate.

3D plots:

3D PLOT V_i makes a three-dimensional plot with hidden lines of a function defined on a two-dimensional mesh.

LINDIREC indicates whether the picture is build of one set of lines (1) or of two orthogonal sets (2).

The default value is 2.

NSTEP= n indicates how many grid lines are used for the 3D-plot. n is equal to the number of lines per cell minus one. A cell has width dx defines by

$$N_{point} = \left(\frac{a}{dx} + 1\right)\left(\frac{b}{dx} + 1\right) \quad (5.4.7)$$

in which a and b are the smallest rectangle sides parallel to the x and y axis that wholly contain the region. This is perfect for a rectangle with equidistant steps.

BLOCK_MODE = m defines in which mode the picture is made. Possible values for m are:

1. sheet mode. This mode sees both sides of the function surface. It requires a bit more work than block mode and is only guaranteed on convex regions. On non-convex regions it may or may not work, depending on the geometry and viewing angle.
2. block mode. This mode only sees the upper side of the function and displays it as if it were cast in concrete. That is, at the boundary of the region, lines are drawn to the ground, that hide the lower side of the function surface. In order to make this work properly the function is lifted if its lowest value is negative. If all function values are non-negative, nothing is changed. Block mode works well in all types of geometries.

intersect_angle= a defines the orientation of the primary intersection lines. The default value is 0° .

ground_value= g is only used in block mode. All function values at the boundary are connected to this value.

If f_{min} is the smallest function value, take care that $ground_value \leq f_{min}$.

If $ground_value > 0$ a value is calculated.

If $f_{min} > 0$: $ground_value = 0$.

If $f_{min} \leq 0$: $ground_value = f_{min}$.

TRANSPARENT indicates that the plot is not a hidden line plot but a type of transparent plot. In this case not all options are available.

3D COLOURED PLOT V_i makes a three-dimensional plot with colored faces of a function defined on a two-dimensional mesh. This possibility may be used only on a display. The three-dimensional surface is plotted from infinity towards the viewer, and because of that, a hidden line (surface) picture arises automatically. The color of the surfaces indicate their distance with respect to the viewer. On a black and white screen, this option produces a classical hidden line plot, however, due to the fact that all faces are plotted and filled with a color (or black) this option is much faster than the standard hidden line procedure. As a consequence, 3D COLOURED PLOT can not be used on a plotter. For a plot you need a hard-copy unit.

The position of the viewer may be given by EYE POINT, the position to which the user looks is given by PROJECTION POINT. See PLOT PARAMETERS. By changing these parameter the observer is able to view of the picture from different angles.

Mesh plots:

PLOT MESH is used to plot a mesh. With the option SKIP ELEMENT GROUPS (g_1, g_2, \dots) the element groups g_1, g_2, \dots are excluded from plotting. The brackets around (g_1, g_2, \dots) are essential.

With the option renumbered nodes, the renumbered nodal point numbers are plotted in stead of the standard nodal point numbers.

PLOT COLOURED_MESH is used to plot a mesh with colors.

The use of this option is actually meant for 3D.

Plotting of a 3D mesh using colors is a lot cheaper than without colors, since it paints all elements at the outer surfaces from behind to the front. This means that it is not necessary to compute hidden lines.

This option is not suited for plotting on paper, but only to a screen, for example by sepview.

PLOT V_j MESH is used to plot a 2D mesh corresponding to the intersection of a 3D mesh with a plane, defined by COMPUTE $V_j = \text{INTERSECTION } V_i, \text{ PLANE}(ax+by+cz=d)$. In this case there is only one element group and no renumbering takes place.

PLOT CURVES is used to plot the curves in the mesh. These curves are defined by the user during the mesh generation.

PLOT V_j CURVES is used to plot the curves in the 2D mesh corresponding to the intersection of a 3D mesh with a plane, defined by COMPUTE $V_j = \text{INTERSECTION } V_i, \text{ PLANE}(ax+by+cz=d)$. These curves are created by the intersection program and define the outer boundary of the intersection.

PLOT POINTS is used to plot the user points in the mesh. These user points are defined by the user during the mesh generation.

User plot commands:

The user plot commands offer the user the possibility to add extra information to the standard

SEPRAN plots. These commands can only be used in combination with the commands OPEN PLOT and CLOSE PLOT. A typical application is:

```
OPEN PLOT
PLOT CONTOUR VO
PLOT TEXT, TEXT = '.....', ORIGIN = (o_x , o_y) [,plot parameters]
PLOT BOX_TEXT, TEXT = '.....', ORIGIN = (o_x , o_y) [,plot parameters]
CLOSE PLOT
```

PLOT TEXT gives the user the possibility to plot a text anywhere in the picture, provided this command has been preceded by an OPEN PLOT command and at least one of the standard SEPRAN plot commands. The command must be succeeded by (if necessary) extra plot commands and finally the command CLOSE PLOT. PLOT TEXT may never be given before a SEPRAN plot command is given. The plot parameters TEXT = 'text to be plotted' and ORIGIN = (o_x, o_y) are mandatory. The height of the letters is defined by the parameter `text_size` (Default: 0.25 cm). The origin must be given in user co-ordinates unless the option `plot_coordinates` is used in which case absolute plot coordinates in centimeters are used. The size of the plot is usually 25×20 cm.

PLOT BOX_TEXT gives the user the possibility to plot a text in the little box below the picture. This box is only plotted if the option `plot_box is` is used. So this command is also only used in an OPEN PLOT ... CLOSE PLOT environment. The origin is given in centimeters, with respect to the small block of size 25×5 cm.

PLOT POLYGON gives the user the possibility to plot a polygon anywhere in the picture. This command is subject to the same restrictions as the command PLOT TEXT. Combinations of OPEN PLOT, CLOSE PLOT, several SEPRAN plot commands and several USER plot commands (like PLOT TEXT and PLOT POLYGON) are allowed. The polygon to be plotted is defined by the data coordinates $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. This defines a polygon from x_1, y_1 to x_2, y_2, \dots , until x_n, y_n , where n is at least 2. If a closed polygon should be plotted, it is necessary to make the first and last point identical. The co-ordinates must be given in user co-ordinates. The brackets in the data statement coordinates are essential and may not be omitted.

If the user wants to use absolute coordinates (in centimeters) instead of user coordinates, he should use the option: `plot_coordinates`.

Other plot commands:

PLOT FIELD makes a two-dimensional plot of (electric) field lines. PSTART= (x, y) defines the co-ordinates of the starting or end point. In that case the option FROM defines that the point PSTART is the initial point and TOWARDS that it is the end point. If both are omitted, the field line is computed in both directions. If PSTART is not given, FLUX = f , must be given. f defines the flux-cut between two points on consecutive field lines. In that case FROM means that the field lines starting on (a part of) the boundary are calculated. TOWARDS means that the field lines ending on (a part of) the boundary are calculated and both omitted means that either one is chosen depending on the direction of the electric field in the first point of (a part of) the boundary. With BNDPART the part of the boundary between user points P_j and P_k is defined as the part of the boundary where field lines must start or end.

If PLOT FIELD is used, it is necessary that the dielectric constant ϵ is stored as first coefficient (see 9.17). For that reason the command PLOT FIELD must be preceded by FILL COEFFICIENTS in order to fill the first coefficient ϵ .

PLOT TRACK computes and plots a particle trace in a velocity field.

The following subkeywords may be used:

TMAX indicates the end time for the tracing of particles.

NMARK indicates the number of markers to be placed along a track.

Markers are positioned at time intervals $TMAX/NMARK$.

PSTART = ... defines the starting points of the particle trajectories.

MESH means that not only the trajectories are plotted, but also the mesh.

PRINT TRACK prints the co-ordinates of the trajectories to the standard output file.

NVIEW defines the type of parallel projection in the three-dimensional case.

Possible values are:

1. projection on (x,z) plane from $y = \infty$
2. projection on (y,z) plane from $z = \infty$
3. projection on (x,z) plane from $y = -\infty$
4. projection on (y,z) plane from $z = -\infty$

TSTEP_PRINT = Δt means that the trajectories are printed to the standard output file, regardless of the presence of the command **PRINT TRACK**. The trajectories are only printed for the values of t equal to 0, Δt , $2\Delta t$, ... $TMAX$.

VALUES = (**Vi**, **Vj**, ...) means that not only the trajectories are printed, but that also the values of the vectors V_i , V_j , ... are computed in the corresponding points by interpolation. These values are also printed.

The use of this subkeyword makes the command **PRINT TRACK** superfluous.

If this subkeyword is used in combination with **TSTEP_PRINT** = **t**, the corresponding time steps are used, otherwise the time step is the arbitrary result of the computation of the particle trajectories.

TYPE_TIME_INTEGRATION = t defines the type of time integration applied to compute the trajectories.

Possible values:

heun The explicit second order Heun method is applied.

runge_kutta The explicit fourth order Runge Kutta method is applied.

trapezoid The implicit second order trapezoid rule is applied.

Default value: heun

EPS_TIME_STEP = ε defines the accuracy that is used to implicitly defining the time step.

A smaller value of ε may result in more accurate results.

Default value: $\varepsilon = 10^{-2}$

For an example of the use of **PLOT TRACK**, see the manual Standard Problems, Section 7.1.1.

Special plot commands:

OPEN PLOT is a necessary command if the user wants to plot more than picture in one plot.

All plot commands after this statement are plotted in the same plot until a **CLOSE PLOT** is given. **OPEN PLOT** may be used to plot several SEPRAN plot commands or to provide SEPRAN plots with extra information, for example by using **PLOT TEXT** and/or **PLOT POLYGON**. So in this way it is for example possible to get a contour plot and a vector plot in one picture.

CLOSE PLOT is necessary to close the plot opened by **OPEN PLOT**.

PLOT IDENTIFICATION may be used to provide all succeeding plots with the same IDENTIFICATION. This command must always be given together with the data command:

TEXT = '...'. This text is plotted on each succeeding picture until a new **PLOT IDENTIFICATION** command is read. To suppress the effect of **PLOT IDENTIFICATION**, use a blank text: ' '. The position of the start of the plot identification must be given by the user by the function **ORIGIN** (o_x, o_y). In this special case the **ORIGIN** is given in centimeters counted from the origin of the plot.

Plot parameters

The following plot parameters may be used at the place formally indicated by `[,plot parameters]`:

```
angle = alpha
axis
bold
boundaries
colour = c
contract
curve_colour = i
dir_rotate = i
elements
equidistant_grid, distance = ( dx, dy, dz )
eye point = ( x_e, y_e, z_e )
factor = f
height = h
inner
istep_colour
istep_symbol
length = l
lev_text_size = l
mark
maxcolour = m_2
mincolour = m_1
ncolour = n
negpos_levels
nneglev = n
noaxis
nobold
noboundaries
nocontract
node
noelements
noinner
nomark
nonode
nonumber
noplot_legenda
noplot_scales
norotate
nposlev = n
number
number format = ( n_x, m_x, n_y, m_y )
num_rotations = n
num_text_size = t
one_picture
pict i of n
plot_box
plot_coordinates
plot_legenda
plot_rows
plot_scales
reference = refval
region = (xmin, xmax, ymin, ymax)
rotate
```

```

rounded_levels
scales = ( x_under , x_upper , y_under , y_upper )
start_surface = (si to sj)
steps = ( stepx, stepy )
symbol = s
text = ' .... '
text_levels = ' .... '
text_size = t
textx = ' .... '
texty = ' .... '
type_func = f
type_scales = t
volumes = ( Vi, Vj, ... )
xscale = x
xy_angle = a
user_coordinates
yfact = y
yscale = y
zscale = z

```

These options may be separated by commas.

angle = α This parameter gives the angle under which the observer sees the plot.
 $0 \leq \alpha \leq 360$

axis This parameter is used to indicate that the plot must be provided with an axis with scale. It makes only sense for those pictures that do not plot axis themselves, i.e. all pictures except those indicated by PLOT FUNCTION type commands. If an OPEN PLOT command is given, the axis are plotted only once.

bold indicates that outer boundaries to be plotted are plotted by double lines. **bold** may be suppressed by **nobold**. Default: **nobold**.

boundaries is used in combination with 3D COLOURED PLOT. It indicates that the boundaries of each face must be plotted in the standard color. For example on a black and white terminal, the combination boundaries and a black color gives a classical hidden line plot. Boundaries may be suppressed by **noboundaries**. Default: **boundaries**.

colour = c Defines the color number to be used for line plotting.

contract is used in combination with PLOT MESH. It indicates that the elements are contracted by a factor of .8 before plotting. As a result all common boundaries of elements are plotted twice. **contract** may be suppressed by **nocontract**. Default: **nocontract**.

curve_colour = i can only be combined with a contour plot or a colored level plot in R^3 .
 If this option is used the visible curves are plotted in the contour plot with colour sequence number i ($1 \leq i \leq 100$).

dir_rotate = i is only used in combination with the plotting of a three-dimensional mesh and the subkeyword **num_rotations**.

It defines in which direction the rotations of the mesh must be carried out.

Possible values:

1. Rotation in x-y plane (z is fixed)
2. Rotation in z-x plane (y is fixed)
3. Rotation in y-z plane (x is fixed)

Default value: 1

element indicates that during the plotting of the mesh also the element numbers are plotted. element may be suppressed by `noelement`. Default: `noelement`.

equidistant_grid is only available for print or plot commands that are defined on the whole region, not for the boundary.

When this command is used, the solution is interpolated to an equidistant grid defined by $(x_{\min}, x_{\max}) \times (y_{\min}, y_{\max}) \times (z_{\min}, z_{\max})$ and step size (dx, dy, dz) .

The interpolated function is printed or plotted.

Hence in this case both the options `REGION = (xmin, xmax, ymin, ymax, zmin, zmax)` and `DISTANCE = (dx, dy, dz)` are obligatory.

eye point = (x_e, y_e, z_e) defines the point where the observer is positioned.

This point is only used in combination with the option `3D COLOURED PLOT`. The default value is $(0, -10, 0)$.

factor = f defines a multiplication factor. In the case of `PLOT VECTOR` it defines the multiplication factor of each vector before plotting.

In the case of a function plot, the function is multiplied by f .

Default $f=1$ in the case of a function plot and automatically scaling in the case of a vector plot. If `factor = 0` (default value), this factor is automatically computed, otherwise the length of each vector is multiplied by f before plotting. For the length of the vectors, the physical units are used, where the unit length is made equal to the geometrical unit length as indicated by the co-ordinates.

height = h gives the height of the texts to be plotted by commands involving `TEXT = '...'` in centimeters. In the case of `3D PLOT` height gives the height of the picture.

inner indicates that for plots where the boundary of the region is plotted, not only the outer boundaries are plotted, but also the inner boundaries. `inner` may be suppressed by `noinner`. Default `noinner`.

istep_colour = n makes only sense in combination with the option `one_picture`. If $n > 0$ each function to be plotted gets a new colour. The colour sequence number is defined as $n \times i$, with i the function sequence number.

Default value 0, meaning that all functions get the same colour.

istep_symbol = n makes only sense in combination with the option `one_picture`. If $n > 0$ each function to be plotted gets a new symbol. The symbol sequence number is defined as $n \times i$, with i the function sequence number.

Default value 0, meaning that all functions get the same symbol (or none).

length = l gives the length of the plot in centimeters. Instead of `length` also `plotfm` may be used. The default length is machine dependent but usual values are 20 cm or 15 cm.

lev_text_size = l defines the size of the Levels text.

mark indicates that during the plotting of the mesh also the node points are marked with a star. `mark` may be suppressed by `nomark`. Default: `nomark`.

maxcolour = m_2 gives the last value to be used in the colour table for a specific plot. The default value is `mincolour + ncolour`.

mincolour = m_1 gives the first value to be used in the colour table for a specific plot. The default value is 1.

ncolour = n This parameter defines the number of colors that is used in the colored plots. In fact this has the same meaning as `nlevel = n`, and both may be interchanged without having any influence. If both are given the first one read is used.

The default value is 20 for colored levels, and 10 for plotted lines.

negpos_levels makes sense for contour plots only. If used the range for negative values and the range for positive values are considered separately. The number of levels for the positive and the negative range are the same except when **nneglev** and **nposlev** are given explicitly.

nneglev must be used in combination with **negpos_levels**. It defines the number of levels in the negative range.

Default value: 10

noaxis suppresses the option axis.

nobold suppresses the option bold.

noboundaries suppresses the option boundaries.

nocontract suppresses the option contract.

noelement suppresses the option element.

node indicates that during the plotting of the mesh also the node numbers are plotted. **node** may be suppressed by **nonode**. Default: **nonode**.

noinner suppresses the option inner.

nomark suppresses the option mark.

nonode suppresses the option node.

noplot_legenda suppresses the option **plot_legenda**.

noplot_scales suppresses the option **plot_scales**.

nonumber suppresses the option number.

This means for example that the numbers corresponding to the contour lines are skipped.

norotate means that the picture is not rotated.

Default: depending on the size of the picture.

nposlev must be used in combination with **negpos_levels**. It defines the number of levels in the positive range.

Default value: 10

number makes only sense in combination with **PLOT CURVES** or **PLOT POINTS**. It indicates that the curves and user points must be provided with numbers. The default is **nonumber**.

If contour lines are plotted, the option **number** indicates that the contour lines must be supplied with a number. Hence in that case **nonumber** is used to suppress the labels corresponding to the contours. In combination with **PLOT CONTOUR** the default is **number**.

number format = (n_x, m_x, n_y, m_y) defines the number of digits of the numbers to be printed along the axis, where n_x, n_y define the number of digits in front of the decimal point (zero means floating format) and m_x, m_y the number of digits behind the decimal point. Default: if **scales** is given (0,2,0,2) otherwise computed by the program.

num_rotations = n is only used if a three-dimensional mesh is plotted.

It defines the number of plots that are made of the mesh by rotating over $360^\circ / \text{num_rotations}$. Hence when **num_rotations** = 1, only one plot is made and this keyword has no effect.

The direction of the rotation is defined by the keyword **dir_rotate**.

Default value: 1

num.text_size = t defines the size of the numbers in the numbers in the contour legend.

one_picture makes only sense if a function plot is made for various time steps. In that case all functions are plotted in the same picture for all the time levels. If omitted each function is plotted in a different picture.

pict = i of n May be used in combination with the records PLOT FUNCTION, PLOT VELOCITY PROFILE, or TIME HISTORY PLOT. If this statement is used, more than one one-dimensional plot is made in one picture with axes. Statements of this type must be placed consecutively, without other type of statements between. The number i must be given in increasing order from 1 to n . n gives the number of curves to be plotted in one picture.

For example the syntax in the case of $n = 3$ should be:

```
PLOT FUNCTION  $V_{k_1}$ , ... , pict 1 of 3
PLOT FUNCTION  $V_{k_2}$ , ... , pict 2 of 3
PLOT FUNCTION  $V_{k_3}$ , ... , pict 3 of 3
```

plot_box If this option is given, the picture is enclosed by a box. Also a small box as sketched in Figure 16.2.1 of the Programmers Guide Section 16.2 is plotted below this box. The large box has dimensions 25×20 cm, the small box 25×5 cm. The small box may be used to plot extra information.

plot_coordinates is used in combination with special commands like **plot text** and **plot polygon**. If used, the coordinates that are given (like the origin) are not in user coordinates but in absolute plot coordinates (centimeters). The size of the plot is usually 25×20 cm.

plot_legenda It ensures the plotting of the legenda and also the plot of the standard text below a picture.
plot_legenda is the default value, hence actually only noplot_legenda is a useful option.

plot_rows This option is only active in combination with plot mesh.

The mesh must be three dimensional.

The result of this option is that first a row of elements is created, either by the explicit option **start_surface = (si to sj)** or by finding a set of points as far as possible from the view point.

Then all elements containing this start set is plotted.

In the next step all neighbor nodes not yet considered are used as starting set. All elements corresponding to it. that not have been plotted before are now plotted. This process is repeated until all elements have been plotted. So actually the mesh is plotted layer by layer. This is typically meant as inspection tool.

plot_scales If this option is used in combination with a plot, the value of the x and y-scales and in case of a vector plot the scaling factor for the vector length are plotted.
plot_scales is the default value, hence actually only noplot_scales is a useful option

reference = refval is used to define a reference value for the 3D plot of a function. This reference value defines the reference value for the z-height. If reference is not given, the maximal function value is used as reference value.

region = (xmin, xmax, ymin, ymax) is used to define a cut of a two-dimensional region.

rotate means that the picture is rotated over an angle of 90° .

rounded_levels makes only sense in combination with contour plots. If used the contour levels are rounded to "nice" values.

scales = (xunder, xupper, yunder, yupper) define the range of the scales along the axis of a one-dimensional plot, See Figure 5.4.1. (Default: computed by the program).

start_surface = (Si to Sj) is used in combination with **plot_rows**. It is meant to define the starting row of points.

steps = (stepx, stepy) defines the number of steps to be used along the axis.
(default: (10,10))

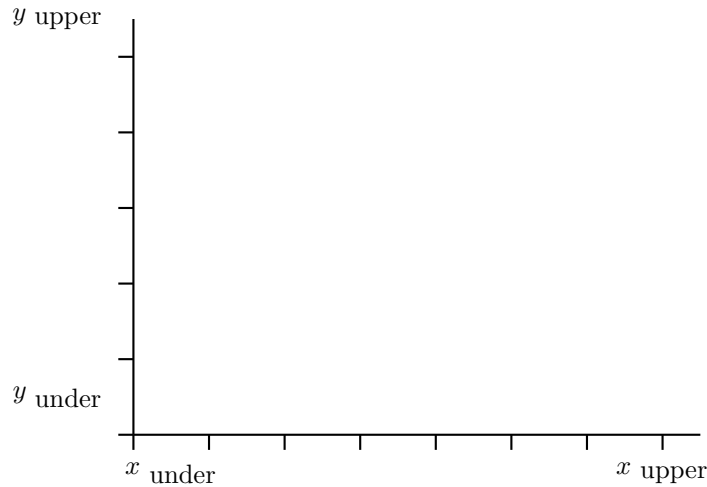


Figure 5.4.1: Definition of x_{under} etc.

symbol = s defines the number of the symbol to be used for plotting a one-dimensional function (installation dependent).

text = ' ... ' defines a text to be plotted. For the commands PLOT IDENTIFICATION and PLOT TEXT, this is the text as described before. For the other SEPRAN plot commands, except the commands corresponding to the PLOT FUNCTION type, this text is always plotted at the bottom of the picture. Use of this option is independent of OPEN PLOT and CLOSE PLOT. If OPEN PLOT is given the text is plotted only once.

text.levels = ' ... ' defines the text to be plotted above the legenda in case of a contour plot or a colored levels plot.
If omitted, the default value LEVELS is used.

text.size = t defines the size of a text to be plotted in centimeters. Default value 0.25 cm.

textx = ' ... ', **texty** = ' ... ' define the texts to be plotted along the axes (default x and y). The part between the quotes is used as text.

type.func = f defines the type of function that must be used to compute a vector as function of another vector.
Possible values

log the function is a logarithm with base 10

type.scales = t defines the type of scaling that must be used along the axis in a function plot.
Possible value for t are:

linlin means that linear scales are used along the x and y-axis.

loglin means that a logarithmic (base 10) scale is used along the x-axis and a linear scale along the y-axis.

linlog means that a logarithmic (base 10) scale is used along the y-axis and a linear scale along the x-axis.

loglog means that logarithmic (base 10) scales are used along the x and y-axis.

Default value: linlin

user_coordinates is used in combination with special commands like `plot text` and `plot polygon`.

If used, the coordinates that are given (like the origin) are not in absolute coordinates but in user coordinates. Since this is the default, this option is superfluous.

volumes = (V_i, V_j, \dots) may be used in combination with the plotting of a 3D mesh. In that case only the volumes mentioned are plotted.

xscale Scale factor of x-axis. Default value 1

This parameter is only used in the case of a function plot. Scaling along the y-axis must be done with factor.

xy_angle = a angle ϕ of projected y-axis with respect to x-axis. ($10 < \phi < 170$)

Default value 60° .

This parameter is only used in the case of 3D PLOT.

yfact = y Scale factor; all y-coordinates are multiplied by y before plotting the mesh. $y \neq 1$ should be used when the co-ordinates in x and y direction are of different scales, and hence the picture becomes too small. Default value: 1.

$y < 0$ defines the *absolute* height of the picture to be $|y|$ cm.

yscale Scale factor of y-axis. Default value 0.5

This parameter is only used in the case of 3D PLOT.

zscale Scale factor of z-axis.

Default value 1.

This parameter is only used in the case of 3D PLOT.

The plot parameters defined in a plot record are only valid for that specific plot record. They overwrite defaults locally. Parameters defined by the `DEFINE plot parameters` command are used for all records.

Colour table

The numbers in the colour table define the colors to be used for the plotting. Which colors are connected with these numbers depends on your local installation.

The default colour table is defined by the numbers 1, 2, 3,...

By the command `DEFINE COLOUR TABLE = (C1, C2, C3, ...)` the user may connect new numbers to the colors 1, 2, 3 etc.

5.5 Special commands for time-dependent problems with respect to program SEPPOST

The general input for the program SEPPOST is described in 5.2. This paragraph is devoted to the available time commands.

The syntax of the time commands is:

Options are indicated between the square brackets [and].

```

TIME =  $t_0$ 
TIME = ( $t_0, t_1$ )
TIME = ( $t_0, t_1, \text{istep}$ )
TIME HISTORY [ $(t_0, t_1)$ ] print min  $V_i$ 
TIME HISTORY [ $(t_0, t_1)$ ] print max  $V_i$ 
TIME HISTORY [ $(t_0, t_1)$ ] print min abs ( $V_i$ )
TIME HISTORY [ $(t_0, t_1)$ ] print max abs ( $V_i$ )
TIME HISTORY [ $(t_0, t_1)$ ] print point(x,y,z)  $V_i$  [,degfd=k]
TIME HISTORY [ $(t_0, t_1)$ ] print position value =  $c V_i$  [,degfd=k]
TIME HISTORY [ $(t_0, t_1)$ ] plot min  $V_i$ 
TIME HISTORY [ $(t_0, t_1)$ ] plot max  $V_i$ 
TIME HISTORY [ $(t_0, t_1)$ ] plot min abs ( $V_i$ )
TIME HISTORY [ $(t_0, t_1)$ ] plot max abs ( $V_i$ )
TIME HISTORY [ $(t_0, t_1)$ ] plot point(x,y,z)  $V_i$  [,degfd=k]
TIME HISTORY
[ $(t_0, t_1)$ ] plot position value =  $c V_i$  [,degfd=k]

```

TIME = (t_0, t_1, istep) is meant for time-dependent problems. All commands after this COMMAND are carried out for the actual times t_0 to t_1 with integer steps istep . If t_0 and /or t_1 do not coincide with times at which the solution is actually computed, the times closest to t_0 and t_1 are chosen. If t_1 is omitted only $t = t_0$ is used. istep gives the number of time steps minus one between succeeding times (default 1).

TIME HISTORY (t_0, t_1) makes a time history of the quantity from time t_0 to t_1 . If (t_0, t_1) is omitted, the complete time interval is used.

plot / print min/max V_i plots or prints the minimum, maximum value of V_i respectively, $\text{abs}(V_i)$ does the same for the absolute value of V_i .

plot / print point (x,y,z) V_i makes a time history of the value of V_i in point (x,y,z). At this moment the node closest to (x,y,z) is used instead of the point itself.

plot / print position value = $c V_i$ is a special command. It assumes that v_i is the result of an intersection of a line with a 2D mesh. The position of the coordinate along this line, where the function reaches the value c is plotted or printed as function of time.

6 Some examples of complete SEPRAN runs

6.1 Introduction

In this chapter we shall treat some examples of problems that may be solved by SEPRAN. All these examples require only knowledge of the SEPRAN Users Manual and with respect to the standard elements sometimes a little bit of the manual Standard Problems. No information of the Programmers Guide is required.

Most of the examples treated have a strict artificial character, they are meant to explain the structure of SEPRAN sessions more clearly, rather than solving actual problems. More realistic problems may be found in the manual Standard Problems.

This chapter is subdivided into the following main sections.

Section 6.2 treats the solution of simple elliptic problems with only one degree of freedom. Both the use of *structure* and of user written elements is dealt with.

Section 6.3 deals with the solution of some simple non-linear equations.

Section 6.4 is devoted to the solution of some time-dependent problems and how to use the input block *time_integration*

6.2 Examples of elliptic equations with one degree of freedom per point

In this section we treat some examples of (artificial) elliptic problems to show some of the possibilities to solve these equations.

The following examples will be treated:

6.2.1 An example of a simple potential problem.

This example shows how a simple artificial potential problem may be solved by SEPRAN. No special input is required, nor is the structure of the program defined by the user.

6.2.2 An example of a simple potential problem with a user defined structure.

This example is exactly the same as the one in 6.2.1, however in this case the user defines the structure of the program himself.

6.2.3 An example of a simple potential problem with a user defined element subroutine.

This example is exactly the same as the one in 6.2.1, however in this case the user uses his own element subroutine.

6.2.4 An example of a simple potential problem with a refinement of the mesh.

This example is exactly the same as the one in Section 7.1 of the SEPRAN INTRODUCTION. Extra compared to the introduction is that the mesh is refined after the solution is computed and the solution is also computed at the refined mesh. The difference between both solutions is computed and printed.

6.2.5 An example of how to compute derived quantities and integrals in combination with a simple potential problem

This example is in fact identical to the one in Section 6.2.1, but as an extra it is shown how the user may compute derived quantities like the gradient and how integrals may be computed.

6.2.6 An example of how to use periodical boundary conditions in R^2

6.2.7 An example of how to use periodical boundary conditions in R^3

6.2.8 An example of the manipulation of scalars

This example shows how one can manipulate scalars and use them as coefficients for the differential equation. Besides that it treats the importance of the compatibility condition, in case one wants to solve a singular problem.

6.2.1 An example of a simple potential problem

In this section we will consider the solution of a simple Laplacian equation defined on a unit square by SEPRAN.

To get this example into your local directory give the command:

```
sepgetex examu621
```

To run the example use:

```
sepmesh examu621.msh
sepcomp examu621.prb
seppost examu621.pst
jsepview sepplot.001
```

You may combine sepgetex and running by using:

```
sepexam examu621
```

This example is meant to demonstrate the most simple case of a SEPRAN program. No special options are used.

Consider the Laplace equation

$$\Delta T = 0 \quad (6.2.1.1)$$

with Δ the Laplacian operator. We assume that the region at which this equation is defined is the unit square $(0, 1) \times (0, 1)$ defined in Figure 6.2.1.1

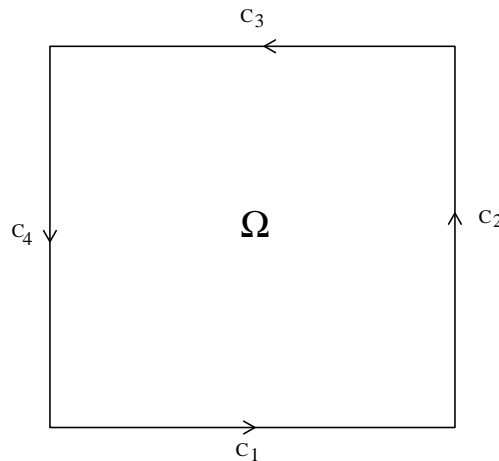


Figure 6.2.1.1: Definition of region for simple Laplacian equation

We suppose that the boundary conditions are given by

$$C1, C2, C4 : T = 0 \quad (6.2.1.2)$$

$$C3 : T = 1 \quad (6.2.1.3)$$

The mesh for this problem may be created by program SEPMESH.

A sample input file based upon the submesh generator quadrilateral and a 10×10 grid is the following one using quadrilateral elements:

```
! examu621.msh
! mesh for the unit square (0,1) x (0,1)
!
```

```

! define some standard constants
constants
  reals
    width = 1      # width of the region
    height = 1    # height of the region
  integers
    n = 10        # number of elements in horizontal direction
    m = 10        # number of elements in vertical direction
    shape_cur = 1 # type of elements along curves (linear)
    shape_sur = 5 # type of elements in surfaces (bi-linear quadrilaterals)
end
#
# Define the mesh
#
mesh2d          # See Users Manual Section 2.2
#
# user points
#
points          # See Users Manual Section 2.2
  p1=( 0, 0)
  p2=( width, 0)
  p3=( width, height)
  p4=( 0, height)
#
# curves
#
curves          # See Users Manual Section 2.3
  c1=line shape_cur (p1,p2,nelm=n)
  c2=line shape_cur (p2,p3,nelm=m)
  c3=line shape_cur (p3,p4,nelm=n)
  c4=line shape_cur (p4,p1,nelm=m)
#
# surfaces
#
surfaces        # See Users Manual Section 2.4
  s1=quadrilateral shape_sur (c1,c2,c3,c4)
plot            # make a plot of the mesh
               # See Users Manual Section 2.2
end

```

This problem can easily be solved by program sepcomp. Since no function subroutines are required the standard version of sepcomp may be utilized.

Sepcomp requires input from the standard input file. To solve this problem elements of type 800 are used as described in the manual Standard problems Section 3.1.1. The following sample input file is sufficient to solve the problem.

The system of equations is solved by an iterative solver (Conjugate gradients) and for that reason the matrix must be stored as a compact matrix.

```

# examu621.prb
#
# problem file for the simple Laplacian problem as described
#           in the SEPRAN Users Manual Section 6.2.1
#
# To run this file use:
#   sepcomp temperature.prb

```

```
#
# Define some general constants
#
constants
  vector_names
  temperature
end
#
# Define the type of problem to be solved
#
problem          # See Users Manual Section 3.2.2

  types          # Define types of elements,
                # See Users Manual Section 3.2.2
  elgrp1 = (type=800) # Type number for Poisson equation
                # See Standard problems Section 3.1
  essbouncond    # Define where essential boundary conditions are
                # given (not the value)
                # See Users Manual Section 3.2.2
  curves (c1 to c4) # all boundaries c1 to c4
end
#
# Define the structure of the large matrix
# See Users Manual Section 3.2.4
#
matrix
  storage_method = compact, symmetric # symmetric matrix, stored as compact one
                                     # hence an iterative method is used
end
#
# Define non-zero essential boundary conditions
# See Users Manual Section 3.2.5
#
essential boundary conditions
  curves (c3) value = 1 # At C3 T=1, at all other boundaries we
                       # have T=0, which does not require input
end
#
# The coefficients for the differential equation
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1
#
coefficients
  elgrp1
    coef6 = 1 # a11 = 1
    coef9 = coef 6 # a22 = 1
end
```

The solution may be visualized by seppost using the following sample input file:

```
* examu621.pst
* input for seppost
*
postprocessing
  plot contour temperature
end
```

6.2.2 An example of a simple potential problem with a user defined structure

In this section we consider exactly the same problem as in Section 6.2.1, however, in this case we use explicitly the block *structure* in the input file for sepcomp. This example shows the minimum structure block necessary to solve a simple problem. Since the plot is made in the structure block, sepplot is superfluous.

To get this example into your local directory give the command:

```
sepgetex examu622
```

To run the example use:

```
sepmesh examu622.msh
sepcomp examu622.prb
jsepview sepplot.001
```

The sample input file for sepcomp becomes:

```
# examu622.prb
#
# problem file for the example as described
#           in the SEPRAN Users Manual Section 6-2-2
#
# To run this file use:
#   sepcomp examu622.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
# Define some general constants
#
constants
  reals
    mu      = 1          # permeability
  vector_names
    temperature
end
#
# Define the type of problem to be solved
#
problem          # See Users Manual Section 3.2.2

  types          # Define types of elements,
                # See Users Manual Section 3.2.2
    elgrp1 = (type=800) # Type number for Poisson equation
                # See Standard problems Section 3.1
  essbouncond    # Define where essential boundary conditions are
                # given (not the value)
                # See Users Manual Section 3.2.2
    curves (c1 to c4) # all boundaries c1 to c4
end
#
```

```
# The coefficients for the differential equation
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1
#
coefficients
  elgrp1
    coef6 = mu           # a11 = mu
    coef9 = coef 6      # a22 = mu
  end
#
# Structure block, defines sequence in which statements are carried out
# See Users Manual Section 3.2.3
#
structure

# Define the structure of the large matrix, Section 3.2.3.20

matrix_structure, storage_scheme = compact, symmetric
# symmetric matrix, stored as compact one
# hence an iterative method is used

# Put essential boundary conditions into the solution vector, Section 3.2.3.1

prescribe_boundary_conditions temperature, curves (c3) value = 1
# At C3 T=1, at all other boundaries we
# have T=0, which does not require input

# Solve problem, Section 3.2.3.3

solve_linear_system temperature

# Make a contour plot of the solution, Section 3.2.3.13

plot_contour temperature

end
```

6.2.3 An example of a simple potential problem with a user defined element subroutine

Consider the pure artificial problem

$$\begin{aligned} \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} &= 0 \quad x \in (0, 1) \times (0, 1) \\ \varphi &= xy \text{ at boundaries 1, 2 and 4} \\ \frac{\partial \varphi}{\partial n} &= x \text{ at boundary 3,} \end{aligned} \quad (6.2.3.1)$$

where the region is shown in Figure 6.2.3.1

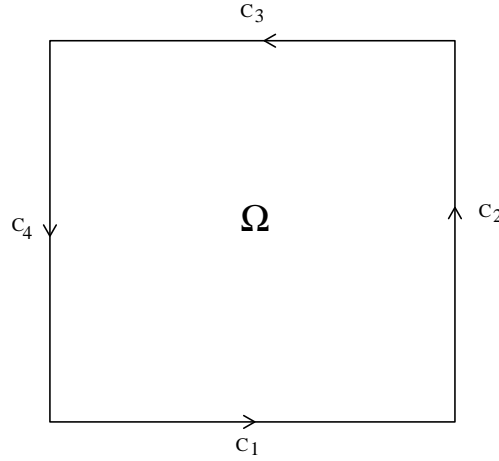


Figure 6.2.3.1: Definition of region for artificial problem

The weak formulation corresponding to 6.2.3.1 is:

$$\int_{\Omega} \nabla T \cdot \nabla v d\Omega = \int_{\Gamma_3} T d\Gamma \quad (6.2.3.2)$$

for all v satisfying $v = 0$ at boundaries C1, C2 and C4.

This leads to the Galerkin formulation:

$$\sum_j T_j \int_{\Omega} \nabla \varphi_j \cdot \nabla \varphi_i d\Omega = \int_{\Gamma_3} \varphi_i d\Gamma \quad (6.2.3.3)$$

For the internal elements this leads to an element matrix with elements:

$$S_{ij}^{e^k} = \int_{e^k} \nabla \varphi_j \cdot \nabla \varphi_i d\Omega \quad (6.2.3.4)$$

and a zero element vector.

For a linear triangle the gradient of the basis functions φ_i is a constant given by the following sequence of formulae:

$$\begin{aligned} e_{11} &= y_2 - y_3 & e_{12} &= x_3 - x_2 \\ e_{21} &= y_3 - y_1 & e_{22} &= x_1 - x_3 \\ e_{31} &= y_1 - y_2 & e_{32} &= x_2 - x_1 \end{aligned} \quad (6.2.3.5)$$

$$\Delta = e_{31}e_{12} - e_{32}e_{11} \quad (6.2.3.6)$$

$$\frac{\partial \varphi_i}{\partial x} = \frac{e_{i1}}{\Delta}, \quad \frac{\partial \varphi_i}{\partial y} = \frac{e_{i2}}{\Delta} \quad (6.2.3.7)$$

The boundary condition $\frac{\partial \varphi}{\partial n} = 1$ introduces a boundary integral $\int_{\Gamma_3} 1 \varphi_i d\Gamma$ with φ_i a test function.

The element matrix for the boundary element is equal to zero, the element vector is given by

$$\frac{h}{2} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

where x_i is the x -coordinate of the i^{th} point in the element.

To get this example into your local directory use:

```
sepgetex examu623
```

To run the various parts use:

```
sepmesh examu623.msh
jsepview sepplot.001
seplink examu623.f90
examu623 < examu623.prb
seppost examu623.pst
jsepview sepplot.001
```

The program sepmesh may be run with the following input file:

```
*examu623.msh
*
* Mesh for artificial test example (rectangle)
*
constants          # See Users Manual Section 1.4
  reals
    width = 1       # width of the region
    height = 1      # height of the region
  integers
    n = 10          # number of elements in horizontal direction
    m = 10          # number of elements in vertical direction
    shape_cur = 1   # type of elements along curves (linear)
    shape_sur = 3   # type of elements in surfaces (linear triangles)
end
#
# Define the mesh
#
mesh2d              # See Users Manual Section 2.2
#
# user points
#
points              # See Users Manual Section 2.2
  p1=( 0, 0)
  p2=( width, 0)
  p3=( width, height)
  p4=( 0, height)
#
# curves
#
curves              # See Users Manual Section 2.3
```

```

      c1=line shape_cur (p1,p2,nelm=n)
      c2=line shape_cur (p2,p3,nelm=m)
      c3=line shape_cur (p3,p4,nelm=n)
      c4=line shape_cur (p4,p1,nelm=m)
#
# surfaces
#
surfaces      # See Users Manual Section 2.4
      s1=quadrilateral shape_sur (c1,c2,c3,c4)
plot          # make a plot of the mesh
              # See Users Manual Section 2.2
end

```

The SEPRAN main program consists of a call to startsepcomp only, subroutine ELEM contains the computation of element matrices and element vectors.

The following program may be used in this case:

```

program exam_um_623
call startsepcomp
end

! --- Subroutine ELEM for the computation of element matrix and element
!      vectors

subroutine elem ( coor, elemmt, elemvc, iuser, user, &
                 uold, matrix, vector, index1, index2 )
implicit none
double precision coor(2,*), elemmt(*), elemvc(*), user(*), uold(*)
integer iuser(*), index1(*), index2(*)
logical matrix, vector

include 'SPCOMMON/cact1'

double precision x(3,2), e(3,2), gradphi(3,2), delta, h
integer i, j

!
! delta   contains the Jacobian delta
!
!
!           ij
! e       contains the factors e      (=e(i,j))
!
! gradphi contains the gradient of phi
! d phi i /dx = gradphi(i,1)
! d phi i /dy = gradphi(i,2)
!
! h       Length of boundary element
!
! i,j     General loop variables
!
! x       contains the co-ordinates:  x(i,1) = x    x(i,2) = y
!                                     i              i
!
!
if ( itype==1 ) then

```



```
!   --- itype = 1,  internal element (triangle)
!
!       Store the co-ordinates into x
!
!       x(1:3,1) = coor(1,index1(1:3))
!       x(1:3,2) = coor(2,index1(1:3))
!
!       --- compute the differences eij and the Jacobian delta
!
!       e(1,1) = x(2,2) - x(3,2)
!       e(2,1) = x(3,2) - x(1,2)
!       e(3,1) = x(1,2) - x(2,2)
!
!       e(1,2) = x(3,1) - x(2,1)
!       e(2,2) = x(1,1) - x(3,1)
!       e(3,2) = x(2,1) - x(1,1)
!
!       delta = e(3,1) * e(1,2) - e(3,2) * e(1,1)
!
!       --- Fill the gradient of phi
!
!       gradphi = e / delta
!
!       --- Fill element matrix
!
!       do j = 1, 3
!         do i = 1, 3
!           elemmt( (i-1)*3+j ) = 0.5d0 * abs(delta) * &
!             ( gradphi(i,1)*gradphi(j,1) + gradphi(i,2)*gradphi(j,2) )
!         end do ! i = 1, 3
!       end do ! j = 1, 3
!
!       --- Set element vector equal to zero
!
!       elemvc(1:3) = 0d0
!
!     else
!
!       --- itype = 2,  boundary element
!
!       Store the co-ordinates into x
!
!       x(1:2,1) = coor(1,index1(1:2))
!       x(1:2,2) = coor(2,index1(1:2))
!
!       h = sqrt ( (x(2,1)-x(1,1))**2 + (x(2,2)-x(1,2))**2 )
!
!       --- Set element matrix equal to zero
!
!       elemmt(1:4) = 0d0
!
!       --- Fill element vector
!
!       elemvc(1:2) = h * 0.5d0 * x(1:2,1)
```

```

        end if

    end

!   Function FUNCBC is used to prescribe the boundary condition u = xy

    function funcbc ( icoice, x, y, z )
    implicit none
    integer icoice
    double precision x, y, z, funcbc

    if ( icoice==1 ) then

        funcbc = x * y

    end if

    end

```

This program requires input from the standard input file.

The following input is suitable for the solution of the potential problem:

```

*examu623.prb
*
*   Input file for main program:  Test artificial example
*
#
#   Define some general constants
#
constants
    vector_names
        potential
end
#
#   Define the type of problem to be solved
#
problem                # See Users Manual Section 3.2.2

    types                # Define types of elements,
                        # See Users Manual Section 3.2.2
        elgrp1 = (type=1) # User element itype = 1
    natbouncond          # Defines type for natural boundary conditions
        bnggrp1 = (type=2)
    bounelements        # Defines position of natural boundary conditions
        belm1 = curves ( c3 )
    essbouncond          # Define where essential boundary conditions are
                        # given (not the value)
                        # See Users Manual Section 3.2.2
        curves (c1 to c2)
        curves (c4)
end

* The matrix is symmetrical positive definite
*   A direct solver will be used

```

```
matrix
  symmetric
end
*
* Define non-zero essential boundary conditions
*
essential boundary conditions
  curves (c1,c2), (func=1)
  curves (c4), (func=1)
end
```

Program seppost allows us to print and plot the solution. It requires input from the standard input file.

If, for example, we want to print the solution, make a contour plot, a 3D plot and a separate plot of the function at boundary c3 then the following input file may be used:

```
*examu623.pst
postprocessing
  print potential
  plot contour potential (yfact=0.5)
  plot boundary function potential, curves = (c3)
  3d plot potential, angle = 135
end
```

6.2.4 An example of a simple potential problem with a refinement of the mesh

In this Section we consider the solution of the potential problem in the L-shaped region as treated in Section 7.1 of the Introduction. First we consider the case that we refine the mesh one times. This example is called potrefin and you can get it into your local directory by the command:

```
sepgetex potrefin
```

To run this example perform the following steps:

```
sepmesh potrefin.msh
  view mesh for example by: jsepview sepplot.001
sepcomp potrefin.prb
seppost potrefin.pst
  view plots for example by: jsepview sepplot.001
```

After that we consider the case of a mesh that is refined three times. That example is called potrefin1 and you can get it into your local directory by the command:

```
sepgetex potrefin1
```

To run this example perform the following steps:

```
sepmesh potrefin1.msh
  view mesh for example by: jsepview sepplot.001
seplink potrefin1
potrefin1 < potrefin1.prb
seppost potrefin1.pst
  view plots for example by: jsepview sepplot.001
```

First we consider example potrefin.

We start with the same mesh as in the introduction, hence the mesh input file is a copy of that file.

```
!
! potrefin.msh
!
! Mesh file for example 6.2.4 in users manual
!
# To run this file use:
#   sepmesh potrefin.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  integers
    n = 5           # number of elements along short sides
    m = 2*n        # number of elements along long sides
    shape_cur = 1   # linear line elements
    shape_sur = 3   # linear triangles
  reals
    width = 2       # width of the region
    heigth = 2     # heigth of the region
```

```

        half_heigth = 1 # heigth of the lower part
        upper_right = 1 # x-coordinate of upper part right-hand side
end
#
# Define the mesh
#
mesh2d          # See Users Manual Section 2.2
#
# user points
#
points          # See Users Manual Section 2.2
  p1=(0,0)
  p2=( width,0)
  p3=( width, half_heigth)
  p4=( upper_right, half_heigth)
  p5=( upper_right, heigth)
  p6=(0, heigth)
  p7=(0, half_heigth)
#
# curves
#
curves          # See Users Manual Section 2.3
  c1 = line shape_cur (p1,p2,nelm=m)
  c2 = line shape_cur (p2,p3,nelm=n)
  c3 = line shape_cur (p3,p4,nelm=m)
  c4 = line shape_cur (p4,p5,nelm=m)
  c5 = line shape_cur (p5,p6,nelm=n)
  c6 = line shape_cur (p6,p7,nelm=n)
  c7 = line shape_cur (p7,p1,nelm=n)
  c8 = line shape_cur (p4,p7,nelm=m)
#
# surfaces
#
surfaces        # See Users Manual Section 2.4
  s1 = general shape_sur (c1,c2,c3,c8,c7)
  s2 = general shape_sur (-c8,c4,c5,c6)
#
# Couple each surface to a different element group in order to provide
# different properties to the coefficients
#
meshsurf        # See Users Manual Section 2.2
  selm1 = s1
  selm2 = s2

plot            # make a plot of the mesh
                # See Users Manual Section 2.2

end

```

In program SEPCOMP we start by solving the linear problem, but once we have computed the solution we refine the mesh 1 times. In this way we end up with two meshes: mesh_1, the original mesh and mesh_2 the refined mesh. Next the problem is solved on the fine mesh.

In order to compare the accuracy of both solutions, the potential V1 at the fine mesh is first copied to a new vector V2 and the vector V2 is interpolated to the coarse mesh. Next the vectors V1 and V2 at the coarse mesh are compared, and the difference is printed. Finally the result at the fine

mesh is written together with the fine mesh. hence the results at the fine mesh will be plotted and printed by SEPPOST.

Hence the program consists of the following steps:

```

Solve problem at coarse grid; store result in vector 1 (coarse grid)
Refine mesh
Solve problem at fine grid; store result in vector 1 (fine grid)
Copy vector 1 fine grid to vector 2 fine grid
Interpolate vector 2 to coarse grid
Compute the difference of both vectors at the coarse grid

```

Mark that vector 1 at the fine grid is another vector than vector 1 at the coarse grid, since each vector has two sequence numbers: one indicating the vector sequence number and another indicating the mesh sequence number. The input file for program SEPCOMP is given by:

```

!
! potrefin.prb
!
! Problem file for example 6.2.4 in users manual
!
constants
  vector_names
    potential
    potential_ref
  variables
    difference          ! norm of difference between 2 vectors
end
problem
  types
    elgrp1 = (type=800)
    elgrp2 = (type=800)
  essboundcond
    curves (c1)
    curves (c5)
end
!
! Define structure of main program
!
structure

# Define structure of matrix

matrix_structure, storage_scheme=profile, symmetric
  ! The matrix is symmetrical, a direct solver will be used

# First the problem is solved at a coarse grid
# At this moment the mesh sequence number is 1

prescribe_boundary_conditions, potential, curves (c5), value=1
solve_linear_system, potential

# next the mesh is refined

refine_mesh, mesh_out=2
plot_mesh

```

```

# Now the mesh sequence number is set to 2
# The problem is solved at the fine grid

  prescribe_boundary_conditions, potential_ref, curves (c5), value=1
  solve_linear_system, potential_ref
  plot_contour potential_ref yfact=0.5
  plot_coloured_levels potential_ref yfact=0.5
  plot_3d potential_ref
  plot_function potential_ref, curves = (c3), &
    textx=' x along curve 3', texty='potential'

# In order to compare the results, the solution on the fine grid
# is copied to the coarse grid

  interpolate potential_ref, mesh_in=2, mesh_out=1

# The difference at the coarse grid is computed and printed
# The mesh sequence number is reset to 1

  present_mesh = 1
  difference = inf_norm( potential, potential_ref)
  print difference, text='difference between mesh 1 and mesh 2 is '

# The new mesh and solution are written for postprocessing purposes
# The mesh sequence number is again set to 2

  present_mesh = 2
  write_mesh
  output
end
!
! The coefficients for the differential equation
!
coefficients
  elgrp1
    coef 6 = (value=1)
    coef 9 = (value=1)
  elgrp2
    coef 6 = (value=2)
    coef 9 = (value=2)
end
!
! The matrix to be solved is positive definite
!
solve
  positive definite
end
end_of_sepran_input

```

The potential at the fine mesh may be printed and plotted by program SEPPOST. In fact the same input as in the Introduction may be used.

In this example we used the extra option to plot the mesh. The SEPPOST input file has the following structure:

```
!
```

```
! potrefin.pst
!
! post file for example 6.2.4 in users manual
!
postprocessing

plot mesh
print potential_ref
plot identification, text='Example of potential problem', origin=(3,18)
open plot
  plot contour potential_ref (yfact=0.5,plotfm=10)
  plot text, text='Contour plot potential', origin = (0.4,-.2)
close plot
open plot
  plot contour potential_ref (yfact=0.5),(levels=0.1,0.2,0.3,0.4)
  plot text, text='Contour plot potential, 4 levels given'//
  origin=(0.4,-.2)
close plot
open plot
  plot contour potential_ref (yfact=0.5,smoothing factor = 1)
  plot text, text='Contour plot potential, smoothing 1'//
  origin = (0.4,-.2)
close plot
open plot
  plot contour potential_ref (yfact=0.5,smoothing factor = 2)
  plot text, text='Contour plot potential, smoothing 2'//
  origin = (.4,-.2)
close plot
open plot
  plot coloured contour potential_ref (yfact=0.5)
  plot text, text='Contour plot potential', origin = (.4,-.2)
close plot
  plot boundary function potential_ref, curves = (c3),//
  textx=' x along curve 3', texty='potential'
  plot boundary function potential_ref, curves = (c3),//
  textx=' x along curve 3', texty='potential', symbol=3
  plot boundary function potential_ref, curves = (c3),//
  textx=' x along curve 3', texty='potential', symbol=14
open plot
  3d plot potential_ref, angle = 135
  plot text, text='3D plot potential (angle 135 degrees)'//
  origin = (.4,-.2)
close plot
end
```

Figure 6.2.4.1 shows the refined mesh.

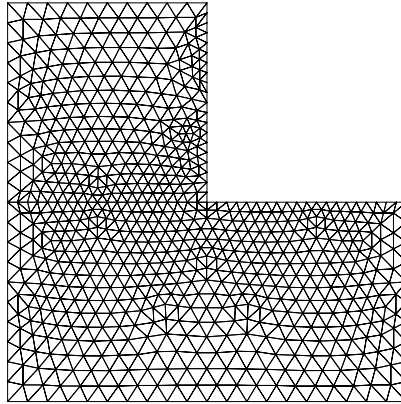


Figure 6.2.4.1: Refined mesh for potential problem

In example `potrefin1` the mesh is refined 3 times. To program this efficiently we use the while loop in the structure block. The while is also able to perform a loop counting. The program has the following structure:

```
Solve problem at coarse mesh and store result in vector 1
icount := 0
While icount < 3 do
  icount := icount+1
  Copy vector 1 to vector 2 at coarse mesh (1)
  Refine mesh to mesh 2
  Interpolate vector 2 to fine mesh (2)
  Copy vector 2 to vector 1 at fine mesh (creates start vector)
  Solve problem at fine mesh and store result in vector 1
  Compute the difference between vectors 1 and 2 at fine mesh
  Interchange the sequence numbers of both meshes so that for the
  new computation 1 is again the coarse mesh
End_while
```

This all leads to the following input file:

```
!
! potrefin1.prb
!
! Problem file for example 6.2.4 in users manual
!
constants
  vector_names
    potential
```

```
        potential_ref
variables
    difference          ! norm of difference between 2 vectors
    icount = 0         ! counter
end

problem
    types
        elgrp1 = (type=800)
        elgrp2 = (type=800)
    essboundcond
        curves (c1)
        curves (c5)
end
!
! Define structure of main program
!
structure

# Define structure of matrix

    matrix_structure, storage_scheme=compact, symmetric
        ! The matrix is symmetrical, a direct solver will be used

# First the problem is solved at a coarse grid
# At this moment the mesh sequence number is 1

    prescribe_boundary_conditions, potential, curves (c5), value=1
    solve_linear_system, potential

# Next a while loop is carried out, in which the mesh is refined 3 times
# In subroutine USERBOOL the counting is performed

    while ( icount<3 ) do

#     the mesh is refined
#     From now on the present mesh is the fine mesh

        potential_ref = potential
        refine_mesh, mesh_out=2
        icount = icount+1    ! raise icount

#     In order to compare the results, the solution on the coarse grid
#     is copied to vector potential_ref and this vector is interpolated
#     to the fine grid

        interpolate potential_ref, mesh_in=1, mesh_out=2

#     vector potential_ref is copied to vector potential, since it is a good
#     starting value for the linear solver

        potential_ref = potential

#     The problem is solved at the fine grid
```

```

    prescribe_boundary_conditions, potential, curves (c5), value=1
    solve_linear_system, potential
    plot_contour potential_ref yfact=0.5
    plot_coloured_levels potential_ref yfact=0.5
    plot_3d potential_ref
    plot_function potential_ref, curves = (c3),&
        textx=' x along curve 3', texty='potential'

#   The difference at the coarse grid is computed and printed

    difference = norm_dif=3, vector1 = potential, vector2 = potential_ref
    print difference, text='difference between mesh 1 and mesh 2 is '

#   In the next steps the roles of mesh 1 and mesh 2 are interchanged
#   Hence we are working with two meshes only
#   This implies that sequence number 1 refers to the present fine mesh
#   and sequence number 2 to the previous coarse mesh
#   The refine step at the start of the while loop puts the newly refined
#   mesh again in sequence number 2

    interchange_mesh ( 1, 2 )

end_while

#   The new mesh and solution are written for postprocessing purposes

    present_mesh = 1
    write_mesh
    output
end
!
!   The coefficients for the differential equation
!
coefficients
    elgrp1
        coef 6 = (value=1)
        coef 9 = (value=1)
    elgrp2
        coef 6 = (value=2)
        coef 9 = (value=2)
end
!
!   The matrix to be solved is positive definite
!
solve
    iteration_method=cg, accuracy=1d-4, start=old_solution, print_level=2
end
end_of_sepran_input

```

The post processing file is also identical to that of potrefin.

6.2.5 An example of how to compute derived quantities and integrals in combination with a simple potential problem

In this section we consider exactly the same problem as in Section 6.2.1. The user defined structure as given in Section 6.2.2 will be used and it is shown how this structure can be extended in order to compute derived quantities like the gradient of the solution as well as some integrals.

The mesh input of Section 6.2.1 is used.

The starting point of the input for program SEPCOMP is the problem file defined in Section 6.2.2. However, in this case we show how one can compute the following quantities:

- ∇T
- $\int_{\Omega} T(x) d\Omega$
- $\int_{C3} T(x) d\Omega$
- The reaction force along a curve

According to the manual Standard Problems Section 3.1, the computation of the gradient of T may be performed by the option derivative. The gradient is coupled to ICHELD=2.

The input block STRUCTURE must be extended with a command indicating that derivatives must be computed and that these derivatives are stored in vector 2.

In order to compute the integral of the temperature T over the complete region the option integral is required. According to the manual Standard Problems ICHELI must be set equal to 2 and a function f must be defined by defining coefficients for the integral. These coefficients are stored in input block COEFFICIENTS with sequence number 2. The input in this block is simple, since it is only used to define the fourth coefficient (f) equal to 1. A separate command in the input block STRUCTURE is necessary to activate the computation.

The integral of the temperature over curve C3 is computed by the command boundary integral in the input block STRUCTURE.

To compute the reaction forces a separate call of derivative is required.

Mark that the reaction force is equal to the effect of a distributed flux $\frac{\partial T}{\partial n}$ in all nodes. This means that in each node on the boundary we have a reaction force equal to $\int_{\Gamma} \frac{\partial T}{\partial n} \phi_i d\Gamma$, where ϕ_i is the corresponding basis function for that node. In this example this means that the reaction force is equal to $\frac{\partial T}{\partial n}$ multiplied by the element length.

The postprocessing is incorporated in the structure block.

So the input for program SEPCOMP has the following form:

```
# examu625.prb
#
# problem file for simple 2d Laplacian problem with computation of
# derived quantities
# See Users Manual Section 6.2.5
#
# To run this file use:
#   sepcomp examu625.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
```

```
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals             # Define real parameters for the problem
    kappa = 1      # diffusion parameter
  vector_names     # Define the names of all vectors that are to be used
    temperature    # Solution vector temperature (T)
    gradient       # Gradient of the temperature (grad T)
    reaction_force
  variables        # Define the names of all scalars that are to be used
    volint         # Volume integral of temperature T
    bounint       # Boundary integral of temperature along curve
end
#
# Define the type of problem to be solved
#
problem            # See Users Manual Section 3.2.2

  types            # Define types of elements,
                  # See Users Manual Section 3.2.2
    elgrp1=800    # Type number for second order elliptic equation
                  # See Standard problems Section 3.1
  essbouncond     # Define where essential boundary conditions are
                  # given (not the value)
                  # See Users Manual Section 3.2.2
    curves(c1 to c4) # Essential boundary conditions on all boundaries
end

# Define the coefficients for Laplacian equation
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 1
  elgrp1
    coef6 = kappa      # a11 = kappa
    coef9 = coef 6     # a22 = kappa
end

# Definition of coefficients for the volume integration

coefficients, sequence_number = 2
  elgrp1
    coef4 = 1          # f = 1
end

# Define which linear solver must be used and what accuracy is required
# In fact all these parameters are equal to the default values and so
# this complete block may be omitted
# See Users Manual Section 3.2.8

solve, sequence_number = 1
  iteration_method = cg, accuracy = 1e-3
end
```

```
# Define the structure of the problem
# In this part it is described how the problem must be solved
# This part is superfluous, but if you want to solve a more sophisticated
# problem this is a good start
#
structure                # See Users Manual Section 3.2.3

# Define the structure of the large matrix
matrix_structure storage_scheme=compact, symmetric ! symmetric compact matrix

# Compute the temperature
# First prescribe the essential boundary conditions

prescribe_boundary_conditions, temperature, curves(c3) value = 1
    # At C3 T=1, at all other boundaries we
    # have T=0, which does not require input

# Next solve the system of equations
# The sequence number seq_coef refers to the sequence number of the
# input block coefficients and
# the sequence number seq_solve refers to the sequence number of the
# input block solve

solve_linear_system, temperature, seq_coef = 1, seq_solve=1

# Compute the gradient of the temperature as a derivative and store in gradient
# The sequence number seq_coef refers to the sequence number of the
# input block coefficients. This is superfluous since it is the default
# icheld = 2, indicates that the gradient must be computed

gradient = derivatives(temperature, icheld = 2, seq_coef = 1)

# Plot the results

plot_contour temperature # make a contour plot of the temperature
plot_vector gradient     # make a vector plot of the gradient

# Compute the volume integral and the boundary integral and print both
# The sequence number seq_coef refers to the sequence number of the
# input block coefficients
# icheli = 2 indicates that a function f times the temperature must be
# integrated

volint = integral (temperature, icheli = 2, seq_coef = 2 )
bounint = boundary_integral ( temperature, ichint = 1, curves(c3) )
print volint, text = 'integral of temperature over volume'
print bounint, text = 'integral of temperature over curve 3'

# Print dT/dn along curve 1 by taking the normal component of the gradient

print gradient, curve = c1, normal_component

# Compute the integrated flux dT/dn through curve 1 by computing reaction forces
# along curves 1 to 4 and print
# Mark that this flux is equal to the length of the element multiplied by dT/dn
```

```
# hence in this case exactly 0.1 dT/dn

reaction_force = derivatives ( temperature, type_output = reaction_force &
    curves ( c1 to c4 ), seq_coef = 1 )
print reaction_force, curve = c1

# Write the results to a file
# Since no extra information is used, we have omitted an input block

output

end

end_of_sepran_input
```

6.2.6 An example of how to use periodical boundary conditions in R^2

In this section we give an example of how periodical boundary conditions must be used in R^2 . To get this example into your local directory give the command:

```
sepgetex periodic2d
```

As example we consider a very simple temperature problem defined on a unit square $(0, 1) \times (0, 1)$. Figure 6.2.6.1 shows the definition of the curves. At the upper side (curve $C_{21} = C_4 + C_5$) the

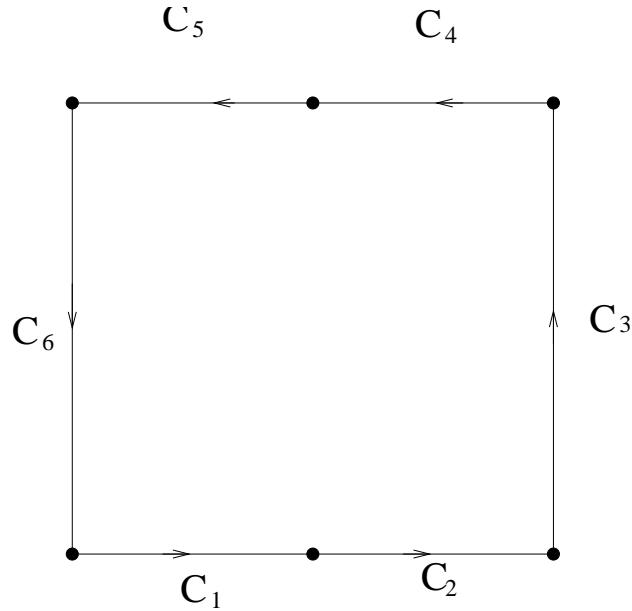


Figure 6.2.6.1: Definition of curves for 2D periodical example

temperature is given by $T = x$ ($0 \leq x \leq 0.5$), $T = 1 - x$ ($0.5 \leq x \leq 1$) At the lower side (curve $C_{20} = C_1 + C_2$) the temperature is equal to 0. At the left-hand side (curve C_6) and the right-hand side (curve C_3) we have periodical boundary conditions.

The mesh input is given by the following file `periodic2d.msh`.

```
*
* periodic2d.msh
* Mesh for example with 2d periodic boundary conditions
*
constants
  integers
    n_half = 10
    m      = 8
  reals
    x_low = 0
    x_upp = 1
    x_mid = 0.5
    y_low = 0
    y_upp = 1
end
mesh2d
  points
    p1=( x_low, y_low )
    p2=( x_mid, y_low )
```



```

    p3=( x_upp, y_low )
    p4=( x_upp, y_upp )
    p5=( x_mid, y_upp )
    p6=( x_low, y_upp )
  curves
    c1 = line1(p1,p2,nelm= n_half)
    c2 = line1(p2,p3,nelm= n_half)
    c3 = line1(p3,p4,nelm= m)
    c4 = line1(p4,p5,nelm= n_half)
    c5 = line1(p5,p6,nelm= n_half)
    c6 = line1(p6,p1,nelm= m)
    c20 = curves(c1,c2)
    c21 = curves(c4,c5)
  surfaces
    s1 = rectangle5 (c20,c3,c21,c6)
  plot
end

```

To create the mesh give the command:

```
sepmesh periodic2d.msh
```

In the problem file it is also necessary to define the periodical boundary conditions. This is done by using type number -1 for the connection elements (element group 2).

Since the boundary conditions depend on space, it is necessary to supply a user written function subroutine FUNCBC. This is done in the following program (file periodic2d.f):

```

program periodic2d
call startsepcomp
end
function funcbc ( ichois, x, y, z )
implicit none
integer ichois
double precision x, y, z, funcbc
if ( ichois==1 ) then
!   --- Curve c4: T = 1-x
    funcbc = 1d0 - x
else
!   --- Curve c5: T = x
    funcbc = x
end if
end

```

This program must be linked by seplink:

```
seplink periodic2d
```

The corresponding input file periodic2d.prb is given by:

```
*
*   periodic2d.prb
*
*   Example of periodical boundary conditions in 2D
*
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  vector_names
  potential
end
*
* Problem definition
*
problem
  types
    elgrp1 = (type=800)          # Standard laplacian equation
  periodical_boundary_conditions
    curves(c3, -c6)
  essboundcond
    curves(c20)                 # lower side
    curves(c21)                 # upper side
end

! Define the structure of the main program

structure
  matrix_structure symmetric
  prescribe_boundary_conditions potential, curves(c4), (func=1) # T = x
  prescribe_boundary_conditions potential, curves(c5), (func=2) # T = 1-x
  solve_linear_system potential
  print potential
  plot_contour potential
  plot_colored_levels potential
end
*
* The coefficients for the differential equation
*
coefficients
  elgrp1
    coef 6 = (value=1)          # Coefficient a_11
    coef 9 = (value=1)          # Coefficient a_22
end
*
* The matrix to be solved is positive definite
*
solve
  positive definite
end
end_of_sepran_input
```

To run the program with input use:

```
periodic2d < periodic2d.prb
```

Figure 6.2.6.2 shows the isotherms computed in this example and Figure 6.2.6.3 the corresponding colour plot.

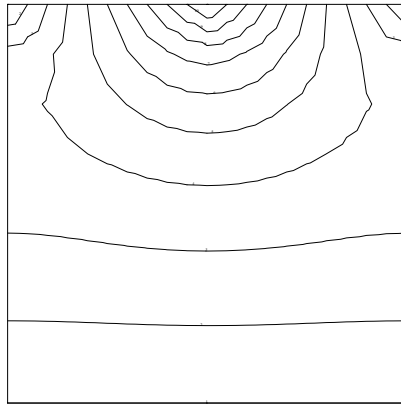


Figure 6.2.6.2: Isotherms in 2D periodical example

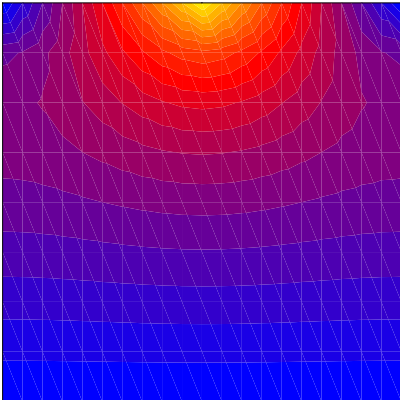


Figure 6.2.6.3: Temperature levels in 2D periodical example

6.2.7 An example of how to use periodical boundary conditions in R^3

In this section we give an example of how periodical boundary conditions must be used in R^3 . To get this example into your local directory give the command:

```
sepgetex periodic3d
```

As example we consider the natural extension of the 2D temperature problem defined in Section 6.2.6. As region we define a unit brick $(0, 1) \times (0, 1) \times (0, 1)$. Figure 6.2.7.1 shows the definition of the curves. The surfaces have the same numbering as in Section 2.5.1, i.e. the lower face S_1

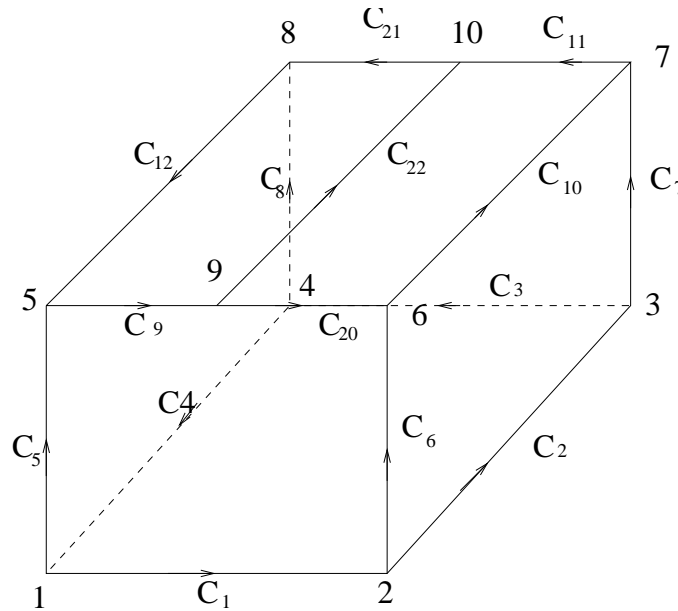


Figure 6.2.7.1: Definition of curves for 3D periodical example

is defined by the curves C_1 , C_2 , C_3 and C_4 . The front face S_2 by the curves C_1 , C_6 , $-C_9$ and $-C_5$. The left-hand face S_3 by the curves C_2 , C_7 , $-C_{10}$ and $-C_6$. The back face S_4 by the curves $-C_3$, C_7 , C_{11} and $-C_8$. The right-hand face S_5 by the curves $-C_4$, C_8 , C_{12} and $-C_5$. The upper surface S_{10} consists of two subfaces S_6 and S_7 . The reason to this is the incorporation of the essential boundary conditions. At the upper face (surface $S_{10} = S_6 + S_7$) the temperature is given by $T = x$ ($0 \leq x \leq 0.5$), $T = 1 - x$ ($0.5 \leq x \leq 1$). At the lower surface (face S_1) and the surfaces S_3 and S_5 the temperature is equal to 0. At the front surface (S_2) and the back surface (S_4) we have periodical boundary conditions.

The mesh input is given by the following file `periodic3d.msh`.

```
*
* periodic3d.msh
* Mesh for example with 3d periodic boundary conditions
*
constants
  integers
    n      = 10      # number of elements in x-direction
    n_half = 5      # half number of elements in x-direction
    m      = 8      # number of elements in y-direction
    k      = 6      # number of elements in z-direction
    edge_1 = 1      # sequence number of 1-st edge (numbering of 2.5.1)
    edge_2 = 2      # sequence number of 2-st edge (numbering of 2.5.1)
    edge_3 = 3      # sequence number of 3-st edge (numbering of 2.5.1)
```

```

edge_4 = 4      # sequence number of 4-st edge (numbering of 2.5.1)
edge_5 = 5      # sequence number of 5-st edge (numbering of 2.5.1)
edge_6 = 6      # sequence number of 6-st edge (numbering of 2.5.1)
edge_7 = 7      # sequence number of 7-st edge (numbering of 2.5.1)
edge_8 = 8      # sequence number of 8-st edge (numbering of 2.5.1)
edge_9 = 30     # sequence number of 9-st edge (numbering of 2.5.1)
edge_10= 10     # sequence number of 10-st edge (numbering of 2.5.1)
edge_11= 31     # sequence number of 11-st edge (numbering of 2.5.1)
edge_12= 12     # sequence number of 12-st edge (numbering of 2.5.1)
face_1 = 1      # sequence number of 1-st face (numbering of 2.5.1)
face_2 = 2      # sequence number of 2-st face (numbering of 2.5.1)
face_3 = 3      # sequence number of 3-st face (numbering of 2.5.1)
face_4 = 4      # sequence number of 4-st face (numbering of 2.5.1)
face_5 = 5      # sequence number of 5-st face (numbering of 2.5.1)
face_6 = 10     # sequence number of 6-st face (numbering of 2.5.1)
reals
  x_low = 0      # Co-ordinates in x, y and z-direction
  x_upp = 1
  x_mid = 0.5
  y_low = 0
  y_upp = 1
  z_low = 0
  z_upp = 1
end
mesh3d
  points
    p1 =( x_low, y_low, z_low )
    p2 =( x_upp, y_low, z_low )
    p3 =( x_upp, y_upp, z_low )
    p4 =( x_low, y_upp, z_low )
    p5 =( x_low, y_low, z_upp )
    p6 =( x_upp, y_low, z_upp )
    p7 =( x_upp, y_upp, z_upp )
    p8 =( x_low, y_upp, z_upp )
    p9 =( x_mid, y_low, z_upp )
    p10=( x_mid, y_upp, z_upp )
  curves
    c edge_1 = line1(p1,p2,nelm= n)
    c edge_2 = line1(p2,p3,nelm= m)
    c edge_3 = line1(p3,p4,nelm= n)
    c edge_4 = line1(p4,p1,nelm= m)
    c edge_5 = line1(p1,p5,nelm= k)
    c edge_6 = translate c edge_5 (p2,p6)
    c edge_7 = translate c edge_5 (p3,p7)
    c edge_8 = translate c edge_5 (p4,p8)
    c9      = line1(p5,p9,nelm= n_half)
    c20     = line1(p9,p6,nelm= n_half)
    c edge_9 = curves(c9,c20)
    c edge_10= translate c edge_2 (p6,p7)
    c11     = line1(p7,p10,nelm= n_half)
    c21     = line1(p10,p8,nelm= n_half)
    c edge_11= curves(c11,c21)
    c edge_12= translate c edge_4 (p8,p5)
    c22     = line1(p9,p10,nelm= m)
  surfaces

```

```

s face_1 = rectangle5 ( c edge_1, c edge_2 , c edge_3 , c edge_4 )
s face_2 = rectangle5 ( c edge_1, c edge_6 , -c edge_9 , -c edge_5 )
s face_3 = rectangle5 ( c edge_2, c edge_7 , -c edge_10, -c edge_6 )
s face_4 = rectangle5 ( -c edge_3, c edge_7 , c edge_11, -c edge_8 )
s face_5 = rectangle5 ( -c edge_4, c edge_8 , c edge_12, -c edge_5 )
s6       = rectangle5 ( c9, c22, c21, c edge_12)
s7       = rectangle5 ( c20, c edge_10, c11, -c22)
s face_6 = ordered surfaces ( (s6, s7) )
volumes
  v1 = brick13 ( s face_1, s face_2, s face_3, s face_4, s face_5, &
                s face_6 )
plot, eyepoint = ( -5, -5, -5 )
end

```

To create the mesh give the command:

```
sepmesh periodic3d.msh
```

The mesh produced by sepmesh is plotted in Figure 6.2.7.2.

In the problem file it is also necessary to define the periodical boundary conditions. This is done

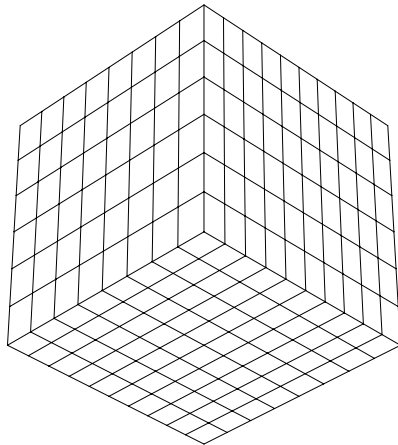


Figure 6.2.7.2: Mesh in 3D periodical example

by using type number -1 for the connection elements (element group 2).

Since the boundary conditions depend on space, it is necessary to supply a user written function subroutine FUNCBC. This is done in the following program (file periodic3d.f):

```

program periodic3d
call startsepcomp
end
function funcbc ( ichois, x, y, z )

```

```

    implicit none
    integer ichois
    double precision x, y, z, funcbc
    if ( ichois==1 ) then

!    --- Surface S6: T = x

        funcbc = x

    else

!    --- Surface S7: T = 1-x

        funcbc = 1d0-x

    end if

end

```

This program must be linked by seplink:

```
seplink periodic3d
```

The corresponding input file periodic3d.prb is given by:

```

*
*    periodic3d.prb
*
*    Example of periodical boundary conditions in 3D
*
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
    vector_names
    potential
end
*
* Problem definition
*
problem
    types
        elgrp1 = (type=800)          # Standard laplacian equation
    periodical_boundary_conditions
        surfaces(s2,s4)
    essboundcond
        surface(s1)                 # lower face
        surface(s3)                 # right-hand side face
        surface(s5)                 # left-hand side face
        surface(s10)                # upper face
end

! Define the structure of the main program

structure

```



```
matrix_structure storage_scheme = compact, symmetric
prescribe_boundary_conditions potential, surface(s6), (func=1) # T = x
prescribe_boundary_conditions potential, surface(s7), (func=2) # T = 1-x
solve_linear_system potential
print potential, curve(c1 to c22)
plot_contour potential
plot_colored_levels potential
end
*
* The coefficients for the differential equation
*
coefficients
  elgrp1
    coef 6 = (value=1)           # Coefficient a_11
    coef 9 = (value=1)           # Coefficient a_22
    coef11 = (value=1)          # Coefficient a_33
end
*
* The matrix is solved by the conjugate gradients method
*
solve
  iteration_method = cg
end
end_of_sepran_input
```

To run the program with input use:

```
periodic3d < periodic3d.prb
```

The pictures produced by seppost are plotted in Figures [6.2.7.3](#) and [6.2.7.4](#).

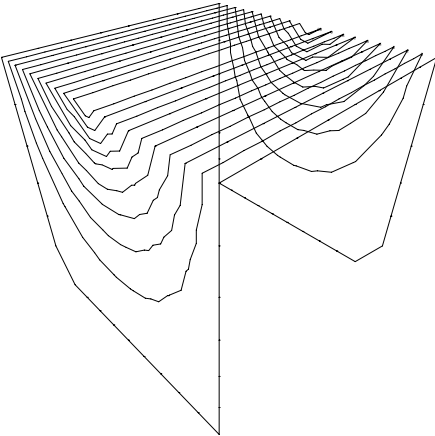


Figure 6.2.7.3: Isotherms in 3D periodical example

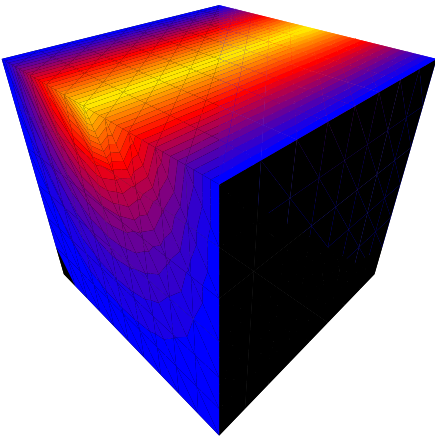


Figure 6.2.7.4: Colored temperature levels in 3D periodical example

6.2.8 An example of the manipulation of scalars

In this section we show how one can manipulate scalars.

In fact it concerns a very simplified model of 3D sensor model of microtomography.

To get this example into your local directory give the command:

```
sepgetex sensor
```

To run this example use:

```
sepmesh sensor.msh
jsepview sepplot.001
secomp sensor.prb
jsepview sepplot.001
```

As example we consider a pipe of height 2 and radius 1. The lower face of the pipe consists of one electrode through which a current of 1 goes.

The upper face consists of two concentric disks. The inner disk has radius 0.5. The outer disk is the electrode on the upper face and through it goes exactly the same current of 1. Figure 6.2.8.1 shows the definition of the curves. The potential in the pipe satisfies the Laplacian equation: $\delta V = 0$.

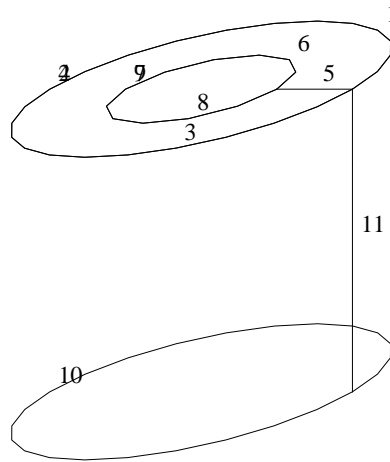


Figure 6.2.8.1: Definition of curves for 3d sensor

On the electrodes the current density i is defined by $\frac{\partial V}{\partial n} = i$, which is a natural boundary condition. The value of i is equal to the current divided by the area of the corresponding electrode. Of course the currents and hence the current densities have opposite signs. On the rest of the pipe there is no current, hence we have a natural boundary condition.

The mesh input is given by the following file.

```
# sensor.msh
#
# mesh file for 3D sensor model of microtomography
# See Users Manual Section 6.2.8
#
# To run this file use:
#   sepmesh sensor.msh
```

```

#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    radius          = 1          # radius of cross section of pipe
    inner_radius    = 0.5        # inner radius of electrode
    height          = 2          # height of the pipe
end
#
# Define the mesh
#
mesh3d              # 3d mesh; see Users Manual Section 2.2
  coarse(unit=0.25) # The concept of coarseness is used with a unit length
                    # of 0.25
#
# user points
#
points              # See Users Manual Section 2.2
  p1=(0,0, height) # centre of circles in upper face
  pd2=( radius,0, height) # first point on outer circle of upper
                          # face
  pd3=( radius,120, height) # second point on outer circle of upper
                          # face
  pd4=( radius,240, height) # third point on outer circle of upper
                          # face
  pd5=( inner_radius,0, height) # first point on inner circle of upper
                          # face
  pd6=( inner_radius,120, height) # second point on inner circle of upper
                          # face
  pd7=( inner_radius,240, height) # third point on inner circle of upper
                          # face
  pd8=( radius,0,0) # point on outer circle of lower face
                   # exactly below first point on upper face
#
# curves
#
curves              # See Users Manual Section 2.3
  c1=carc1(p2,p3,p1) # First arc of outer circle of upper face
  c2=carc1(p3,p4,p1) # Second arc of outer circle of upper face
  c3=carc1(p4,p2,p1) # Third arc of outer circle of upper face
                    # At least three curves are necessary to
                    # define a complete circle in R^3
  c4=curves(c1,c2,c3) # outer circle of upper face
  c5=cline1(p2,p5)   # Connection line between inner circle
                    # and outer circle.
                    # This line is necessary because general
                    # is used and the boundary of general must
                    # be closed
  c6=carc1(p5,p6,p1) # First arc of inner circle of upper face
  c7=carc1(p6,p7,p1) # Second arc of inner circle of upper face
  c8=carc1(p7,p5,p1) # Third arc of inner circle of upper face
  c9=curves(c6,c7,c8) # inner circle of upper face

```

```

    c10=translate c4(p8)      # outer circle of lower surface, is just a
                             # translation of the outer circle of upper
                             # face
    c11=cline1(p2,p8)       # generating curve for pipe surface
#
# surfaces
#
surfaces      # See Users Manual Section 2.4
  s1=general3(c4,c5,-c9,-c5) # region in upper face between inner and
                             # outer circle. This defines the electrode
                             # on the upper face
  s2=general3(c9)           # region in upper face enclosed by inner
                             # circle
  s3=surfaces(s2,s1)       # Complete upper face
  s4=translate s3(c10)     # lower face is translation of upper face
  s5=pipesurface3(c4,c10,c11) # pipe surface
#
# volumes
#
volumes      # See Users Manual Section 2.5
  v1=pipe11(s3,s4,s5)     # The complete pipe
  plot, eyepoint=(4,4,4)  # make a plot of the mesh
                             # also a hidden line plot of the pipe
                             # See Users Manual Section 2.2
end

```

Figure 6.2.8.2 shows the mesh created by sepmesh. The problem file in this case is special in

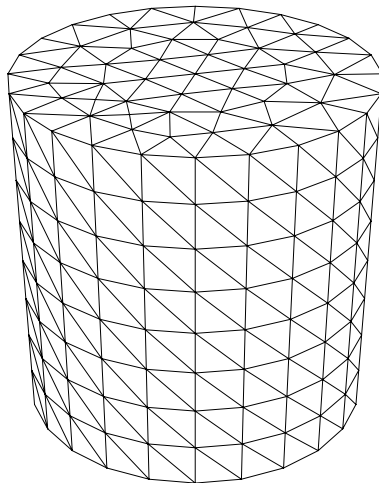


Figure 6.2.8.2: Mesh for 3d sensor

the sense that the current density must be computed by subdividing the current by the area of the electrodes. This area must be computed first. The area is stored in a scalar (variable) and the current in a constant. To compute the current density we have to divide the constant by a scalar, which can only be done through the help of subroutine FUNCSCAL. The computed scalar current_density is used as coefficient for the natural boundary conditions.

At first sight it might seem as if one could compute the area analytically and hence compute the

current density before. However, in this problem we have an extra complication. We are solving the Laplacian equation with natural boundary conditions. As a consequence the solution is fixed upon an additive constant. If we use an iterative solver like conjugate gradients this is no problem and there is no need to set the constant. However, it is necessary that the right-hand side corresponds to correct space. The right-hand side must satisfy a so-called compatibility condition, which means that the integral over the right-hand side must be equal to zero. Numerically this amounts to saying that the sum of the elements of the right-hand side vector must be zero. Due to the fact that the disk-like regions are not exact but approximated by piecewise straight lines the exact area of the disks is not exactly equal to the area of the approximated disks. As a consequence the analytically computed current density does not satisfy the compatibility constraint whereas the numerical computed current density does.

It might look not so important to satisfy the compatibility constraint exactly, however, experiments have shown that in that case the iterative solver does not converge. Only when the right-hand side is compatible a fast convergence is possible.

The input file for the computational program is given by:

```
# sensor.prb
#
# problem file for 3D sensor model of microtomography
# See Users Manual Section 6.2.8
#
# To run this file use:
#   sepcomp sensor.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
# Define some general constants
#
constants
  reals
    mu_1 = 0.001      # mu in volume V1
    current_1 = 1     # current through surface s1 (i.e. electrode
                      # in upper face)
    current_2 = 1     # current through surface s4 (i.e. electrode
                      # in lower face)

  variables
    currentdens_1    # Current density in surface s1, must be computed
    currentdens_2    # Current density in surface s4, must be computed
    area_1           # Area of surface s1, must be computed
    area_2           # Area of surface s4, must be computed

  vector_names
    potential        # Solution vector
    electric_field   # Derived quantity
end
#
# Define the type of problem to be solved
#
problem              # See Users Manual Section 3.2.2

types                # Define types of elements,
                    # See Users Manual Section 3.2.2
    elgrp1 = (type=800) # A Laplacian equation is solved, see manual
                    # Standard Problems Section 3.1
natboundcond        # Natural boundary conditions are necessary to
```

```

                                # define the current density
                                # In this part the type of boundary conditions
                                # is defined
    bngrp1=(type=801)             # Type 801 corresponds to the natural boundary
    bngrp2=(type=801)             # condition of type 800, see manual
                                # Standard Problems Section 3.1
    bounelements                 # Next the position of the boundary elements
                                # must be given
    belm1=surfaces(s1)           # electrode in upper face (surface S1)
    belm2=surfaces(s4)           # electrode in lower face (surface S4)
end

# Define the structure of the problem
# In this part it is described how the problem must be solved
# This is necessary because the area of the boundary must be determined
# in order to compute the current density of each of the electrodes
#
structure                        # See Users Manual Section 3.2.3

# Define the structure of the large matrix

matrix_structure storage_scheme=compact, symmetric ! symmetric compact matrix

# Compute the area of surface S1 (electrode on upper face):
# Store in scalar area_1
# ichint = 9 means: do not integrate over solution

area_1 = boundary_integral ( surfaces(s1), ichint = 9 )

# Compute the area of surface S4 (electrode on lower face):
# Store in scalar area_2

area_2 = boundary_integral ( surfaces(s4), ichint = 9 )

# Print both areas

print area_1, text = 'area of surface S3 is'
print area_2, text = 'area of surface S5 is'

# Compute the current density by subdividing the current by the area
# This can only be done in a function subroutine FUNCSCAL

currentdens_1 = current_1/area_1
currentdens_2 = -current_2/area_2

# Print both computed current densities

print currentdens_1, text = 'current density in surface S1 is'
print currentdens_2, text = 'current density in surface S4 is'

# solve the potential problem using the computed current densities as
# coefficients

solve_linear_system potential
```

```
# Compute the electric field
# icheld = 6 defines the derivative as - A grad(phi),
# where A is the diffusion matrix

electric_field = derivatives ( potential, icheld=6 )

# Post processing:

print potential

plot_contour potential
plot_colored_levels potential

end

# Define the coefficients for the problems (first iteration)
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients
  elgrp1
    coef 6=(value= mu_1)      # alpha 11=mu(S1)
    coef 9=coef 6             # alpha 22=alpha 11
    coef 11=coef 6           # alpha 33=alpha 11
  bngrp1                      # first boundary group (upper electrode)
    coef7= currentdens_1     # The current density has been computed in a
                             # scalar
  bngrp2                      # second boundary group (lower electrode)
    coef7= currentdens_2     # The current density has been computed in a
                             # scalar

end
# input for the linear solver
# See Users Manual Section 3.2.8

solve
  iteration_method=CG, preconditioning=mod_eisenstat, accuracy=1e-3&
  print_level=2
end
end_of_sepran_input
```

Figure 6.2.8.3 shows the equi-potential lines computed in this example and Figure 6.2.8.4 the corresponding colour plot.

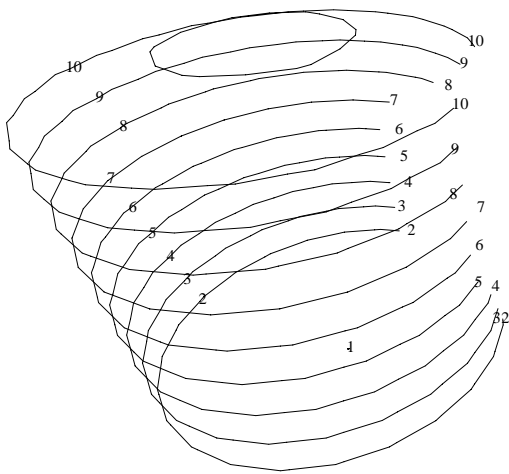


Figure 6.2.8.3: Equi-potential lines in 3d sensor

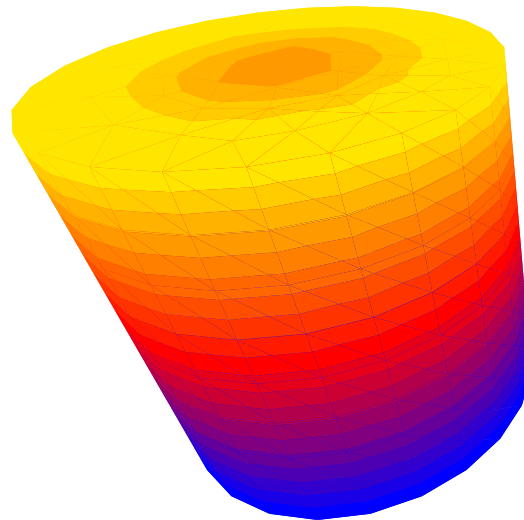


Figure 6.2.8.4: Colored potential levels in 3D sensor

6.2.9 An example of the use of the for loop

In this section we show how one can use a for loop in the structure block in combination with the use of variables (scalars). The same example without for loop can be found in [6.2.10](#). In fact it concerns a very simplified model of a square 2D sensor model of electrical capacitance tomography.

To get this example into your local directory give the command:

```
sepgetex squaresensor
```

To run this example use:

```
sepmesh squaresensor.msh
jsepview sepplot.001
sepcomp squaresensor.prb
jsepview sepplot.001
```

As example we consider a square region of size 16×16 . On each side there are 2 electrodes of length 14 at distance 1 of the vertex.

Figure [6.2.9.1](#) shows the definition of the region and the electrodes (fat). The electrodes are numbered 11 to 18. This is also the numbering used for the corresponding curves in the definition of the mesh. The potential in the square satisfies the diffusion equation: $\text{div } \varepsilon \nabla V = 0$.

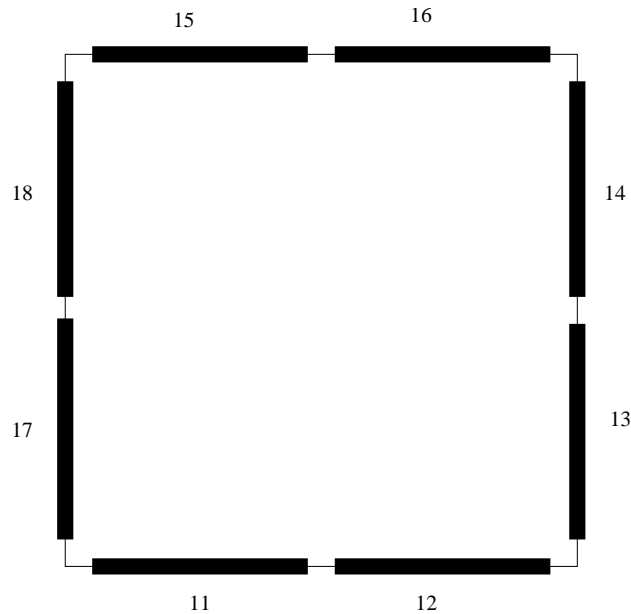


Figure 6.2.9.1: Definition of region and electrodes for square 2d sensor

On the electrodes the potential is given. This problem is a part of a so-called inverse problem, where one tries to find the value of ε by doing some measurements for different values of the potential on the electrodes.

It is the purpose of this exercise to put a zero potential on all electrodes, except one. In a loop over all electrodes the potential at one of the electrodes is set equal to 1. For the inverse problem one is interested in the so-called capacity of an electrode as function of the given boundary condition.

The capacity c_{ij} is defined as: $c_{ij} = \int_{E_j} \varepsilon V d\Gamma$, where i refers to the electrode where the potential V

is equal to 1, and j refers to the j^{th} electrode E_j over which the integral must be computed. Due to symmetry, it is sufficient to compute c_{ij} only for $j \leq i$.

The mesh input is given by the following file.

```
# squaresensor.msh
#
set warn off ! suppress warnings

# Mesh for example sensor
#
# Square grid with 8 electrodes
#
# This example shows how to use the for loop in a structure block
# See Users Manual Section 6.2.9
#
# To run this file use:
#   sepmesh squaresensor.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    x_left          = -16      # x-coordinate of left-hand side
    x_right         =  16      # x-coordinate of right-hand side
    x_left_elec_1   = -15      # left x-coordinate of left electrode on
                                # lower and upper side
    x_right_elec_1  = -1       # right x-coordinate of left electrode on
                                # lower and upper side
    x_left_elec_2   =  1       # left x-coordinate of right electrode on
                                # lower and upper side
    x_right_elec_2  = 15       # right x-coordinate of right electrode on
                                # lower and upper side
    y_low           = -16      # y-coordinate of lower side
    y_top           =  16      # y-coordinate of upper side
    y_low_elec_1    = -15      # lower y-coordinate of lower electrode on
                                # left and right side
    y_top_elec_1    = -1       # upper y-coordinate of lower electrode on
                                # left and right side
    y_low_elec_2    =  1       # lower y-coordinate of upper electrode on
                                # left and right side
    y_top_elec_2    = 15       # upper y-coordinate of upper electrode on
                                # left and right side
  integers
    nelm_elec       = 14       # number of elements along an electrode
    nelm_side       =  1       # number of elements between electrode and
                                # closest vertex of the square
    nelm_between    =  2       # number of elements between two electrodes
end
#
# Define the mesh
#
mesh2d             # See Users Manual Section 2.2
#
# user points
#
points            # See Users Manual Section 2.2
  p1 = ( x_left, y_low)      # left-under vertex
```

```

p2 = ( x_right, y_low)           # right-under vertex
p3 = ( x_left, y_top)           # left-upper vertex
p4 = ( x_right, y_top)          # right-upper vertex
p11 = ( x_left_elec_1, y_low)   # left point left electrode at bottom
p12 = ( x_right_elec_1, y_low)  # right point left electrode at bottom
p13 = ( x_left_elec_2, y_low)   # left point right electrode at bottom
p14 = ( x_right_elec_2, y_low)  # right point right electrode at bottom
p21 = ( x_right, y_low_elec_1)  # lowest point of lower electrode at
                                # right-hand side
p22 = ( x_right, y_top_elec_1)  # highest point of lower electrode at
                                # right-hand side
p23 = ( x_right, y_low_elec_2)  # lowest point of upper electrode at
                                # right-hand side
p24 = ( x_right, y_top_elec_2)  # highest point of upper electrode at
                                # right-hand side
p31 = ( x_left_elec_1, y_top)   # left point left electrode at top
p32 = ( x_right_elec_1, y_top)  # right point left electrode at top
p33 = ( x_left_elec_2, y_top)   # left point right electrode at top
p34 = ( x_right_elec_2, y_top)  # right point right electrode at top
p41 = ( x_left, y_low_elec_1)   # lowest point of lower electrode at
                                # left-hand side
p42 = ( x_left, y_top_elec_1)   # highest point of lower electrode at
                                # left-hand side
p43 = ( x_left, y_low_elec_2)   # lowest point of upper electrode at
                                # left-hand side
p44 = ( x_left, y_top_elec_2)   # highest point of upper electrode at
                                # left-hand side

#
# curves
#
curves          # See Users Manual Section 2.3

# Four sides of square

c1 = curves(c21,c11,c22,c12,c23) # bottom line
c2 = curves(c31,c13,c32,c14,c33) # right-hand side
c3 = curves(c41,c15,c42,c16,c43) # top line
c4 = curves(c51,c17,c52,c18,c53) # left-hand side

# electrodes on side 1

c11 = line1(p11,p12,nelm= nelm_elec) # left
c12 = line1(p13,p14,nelm= nelm_elec) # right

# electrodes on side 2

c13 = line1(p21,p22,nelm= nelm_elec) # bottom
c14 = line1(p23,p24,nelm= nelm_elec) # top

# electrodes on side 3

c15 = line1(p31,p32,nelm= nelm_elec) # left
c16 = line1(p33,p34,nelm= nelm_elec) # right

# electrodes on side 4

```

```

    c17 = line1(p41,p42,nelm= nelm_elec)    # bottom
    c18 = line1(p43,p44,nelm= nelm_elec)    # top

# parts between electrodes on side 1

    c21 = line1(p1,p11,nelm= nelm_side)
    c22 = line1(p12,p13,nelm= nelm_between)
    c23 = line1(p14,p2,nelm= nelm_side)

# parts between electrodes on side 2

    c31 = line1(p2,p21,nelm= nelm_side)
    c32 = line1(p22,p23,nelm= nelm_between)
    c33 = line1(p24,p4,nelm= nelm_side)

# parts between electrodes on side 3

    c41 = line1(p3,p31,nelm= nelm_side)
    c42 = line1(p32,p33,nelm= nelm_between)
    c43 = line1(p34,p4,nelm= nelm_side)

# parts between electrodes on side 1

    c51 = line1(p1,p41,nelm= nelm_side)
    c52 = line1(p42,p43,nelm= nelm_between)
    c53 = line1(p44,p3,nelm= nelm_side)
#
# surfaces
#
surfaces          # See Users Manual Section 2.4
    s1=rectangle5(c1,c2,-c3,-c4)

plot              # make a plot of the mesh
                  # See Users Manual Section 2.2

end

```

Figure 6.2.9.2 shows the curves and Figure 6.2.9.3 shows the mesh created by `sepmesh`. In the problem file we define a loop over the boundary conditions (loop i) and a loop over the integrals to be computed (loop j). The parameters i and j must be defined as variables in the input block constants.

The best way to compute the integral in this case is to use the concept of reaction forces, since the sum of the reaction forces over an electrode is exactly the integral we want.

The input file for the computational program is given by:

```

# squaresensor.prb
#
set warn off ! suppress warnings

# Problem for example sensor
#
# Square grid with 8 electrodes
#
# This example shows how to use the for loop in a structure block

```

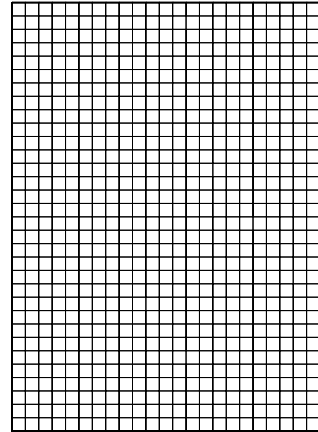
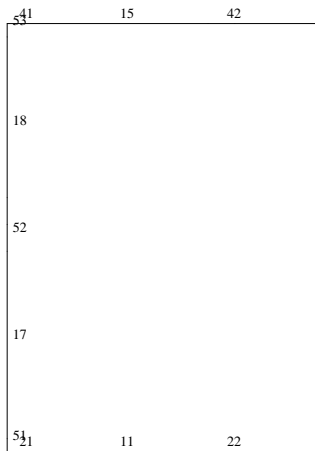


Figure 6.2.9.2: Curves for 2d square sensor

Figure 6.2.9.3: Mesh for 2d square sensor

```

# See Users Manual Section 6.2.9
#
# To run this file use:
#   sepcomp squaresensor.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    eps = 1          # diffusion parameter
  variables
    i                # i parameter in loop
    j                # j parameter in loop
    capacity         # computed capacity
  vector_names
    potential
    reaction_force
end
#
# Define the type of problem to be solved
#
problem            # See Users Manual Section 3.2.2

  types            # Define types of elements,
                  # See Users Manual Section 3.2.2
    elgrp1=(type=800) # Type number for diffusion problem
                  # See Standard problems Section 3.1
  essboundcond     # Define where essential boundary conditions are
                  # given (not the value)

```

```

                                # See Users Manual Section 3.2.2
    curves(c11 to c18)          # Essential boundary conditions on all electrodes
end

# Define the coefficients for the problems (first iteration)
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients
  elgrp1
    coef 6=(value= eps)        # alpha 11=eps
    coef 9=coef 6              # alpha 22=eps
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

structure

  # Define the structure of the large matrix

  matrix_structure symmetric, reaction_force ! symmetric matrix

  print_text '    i    j    capacity '    # Heading for the output

# Loop over i from 11 to 18, i.e. for all electrodes

for i = 11 to 18

  # for each i we use a different value of the boundary
  # conditions
  # The parameter i in the for-loop is used

  create_vector potential, value = 0 ! clear potential
  prescribe_boundary_conditions potential, curves(c i), value=1

  # Compute the potential as function of the boundary conditions
  # by solving a linear system of equations

  solve_linear_system, potential, reaction_force = reaction_force

# Loop over j from 11 to i, i.e. for all electrodes
# i.e. for all electrodes with number not larger than i

for j = 11 to i

  # Compute the capacity by summing the reaction forces over the
  # electrodes

  capacity = boundary_integral ( reaction_force, curves = c j, ichint=8 )

  # Print i, j and the computed capacity

  print i, j, capacity

```

```
    end_for  
  
end_for  
  
# post processing  
  
plot_contour potential  
plot_colored_levels potential  
  
output  
  
end  
end_of_sepran_input
```

Figure 6.2.9.4 shows the equi-potential lines computed in this example and Figure 6.2.9.5 the corresponding color plot.

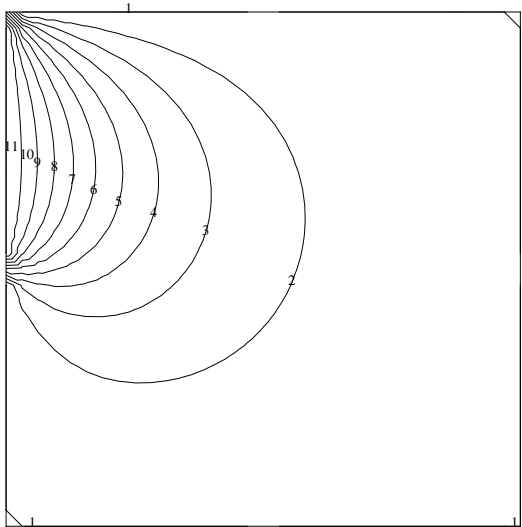


Figure 6.2.9.4: Equi-potential lines in square 2d sensor

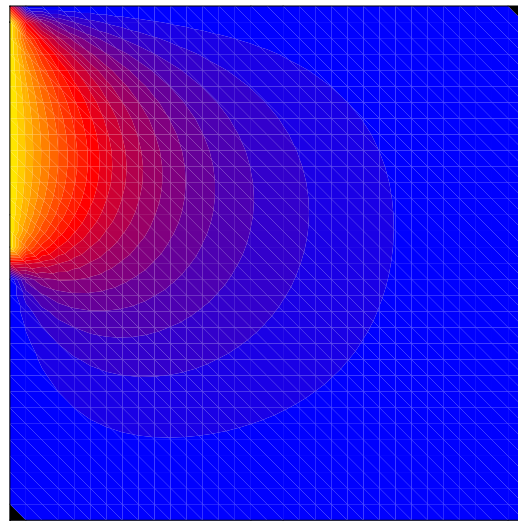


Figure 6.2.9.5: Colored potential levels in square 2D sensor

6.2.10 An example of the computation of capacities

In this section we compute exactly the same problem as in Section 6.2.9.

The only difference is that instead of using the for loop, the option COMPUTE_CAPACITY in the structure block is used, in combination with input concerning the CAPACITY.

To get this example into your local directory give the command:

```
sepgetex capacity
```

To run this example use:

```
sepmesh capacity.msh
jsepview sepplot.001
sepcomp capacity.prb
jsepview sepplot.001
```

Since the problem is exactly the same as in 6.2.9, only a different input file for sepcomp is used.

The input file for the computational program is given by:

```
# capacity.prb
#
# Problem for example sensor
#
# Square grid with 8 electrodes
#
# This example shows how to compute the capacities of the various electrodes
# See Users Manual Section 6.2.10
#
# To run this file use:
#   sepcomp capacity.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  integers
    sensor_first = 11      # curve number of first sensor
    sensor_last = 18      # curve number of last sensor
                          # all curves sensor_first to sensor_last
                          # correspond to sensors
  reals
    eps = 1               # the permittivity of the medium
  vector_names
    potential            # solution vector
    reaction_force       # help vector to store the reaction force
    capacity_vector      # Vector containing all capacities
end
#
# Define the type of problem to be solved
#
problem            # See Users Manual Section 3.2.2

types              # Define types of elements,
```

```

                                # See Users Manual Section 3.2.2
    elgrp1=(type=800)            # Type number for diffusion problem
                                # See Standard problems Section 3.1
    essboundcond                # Define where essential boundary conditions are
                                # given (not the value)
                                # See Users Manual Section 3.2.2
    curves(c sensor_first to c sensor_last) # Essential boundary conditions
                                # on all electrodes
end

# Define the coefficients for the problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number=1
    elgrp1
        coef 6=(value= eps)      # alpha 11=eps
        coef 9=coef 6            # alpha 22=eps
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

structure

# Define the structure of the large matrix

    matrix_structure, symmetric, reaction_force ! symmetric matrix

# Store the essential boundary conditions into the potential vector
# Only the zero bc's have to be filled.
# The boundary conditions 1 are filled in a loop over de electrodes

    prescribe_boundary_conditions, potential # all are zero

# Compute the vector of capacities

    compute_capacity, capacity_vector
    print capacity_vector

    plot_contour potential
    plot_colored_levels potential

end

# Input for the computation of the capacity
# See Users Manual Section 3.2.18

capacity, sequence_number=1
    lin_solver = 1                # Defines the type of linear solver
                                # This is the default value
    seq_coef = 1                 # Defines the coefficients (default value)
    curve_begin = sensor_first   # curve number of first sensor
    curve_end = sensor_last      # curve number of lasst sensor
    solution_vector = potential  # sequence number of solution vector
```

```
    reaction_vector = reaction_force # sequence number of reaction force
end
```

The results of this program are identical to the ones in Section [6.2.9](#).

6.2.11 An example of the solution of an inverse problem

In this section we show how an inverse problem might be solved. For that purpose we take the same problem as in Sections [6.2.10](#).

In this case, however, we assume that the permittivity is unknown. Instead of reading a vector of measured values, which can be done by `create_vector` we create a capacity vector by choosing a given permittivity field and try to reconstruct this field by solving the inverse problem. So this is just an demonstration of how an inverse problem could be solved, not a real example.

To get this example into your local directory give the command:

```
sepgetex inversesensor
```

To run this example use:

```
sepmesh inversesensor.msh
jsepview sepplot.001
sepcomp inversesensor.prb
jsepview sepplot.001
```

The mesh that is used is exactly the same as in [6.2.9](#).

The problem definition is of course completely different.

First we start with a reference permittivity field that is equal to 0.9 in the whole domain, except in element 1, where it gets the value 0.8.

Next the corresponding capacity vector is computed, in the same way as in Section [6.2.10](#). This vector is used as given vector of measured values for the solution of the inverse problem.

The input file for the computational program is given by:

```
# inversesensor.prb
#
set warn off ! suppress warnings

# Problem for example inverse problem for the sensor
#
# Square grid with 8 electrodes
#
# This example shows how to solve an inverse problem
# See Users Manual Section 6.2.11
#
# To run this file use:
#   sepcomp inversesensor.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  integers
    sensor_first = 11      # curve number of first sensor
    sensor_last  = 18      # curve number of last sensor
                        # all curves sensor_first to sensor_last
                        # correspond to sensors

  reals
    delta_eps = 0.1        # Step for computation of sensitivity matrix
    eps_ref   = 1          # Reference value for permittivity used
```

```

        regular_parm = 0.01      # for computation of sensitivity matrix
                                # Regularization parameter to be used
        eps = 0.9                # for computation of sensitivity matrix
                                # Value of the given permittivity field
        eps_1 = 0.8              # This value will be changed in one element
                                # Value of the given permittivity field
                                # in element 1
vector_names
  potential                      # solution vector
  reaction_force                 # help vector to store the reaction force
  capacity_vector               # Vector containing all capacities
  epsilon_vector                # vector containing the permittivities
end
#
# Define the type of problem to be solved
#
problem                          # See Users Manual Section 3.2.2

types                            # Define types of elements,
                                # See Users Manual Section 3.2.2
  elgrp1=(type=800)             # Type number for diffusion problem
                                # See Standard problems Section 3.1
  essboundcond                 # Define where essential boundary conditions are
                                # given (not the value)
                                # See Users Manual Section 3.2.2
  curves(c sensor_first to c sensor_last) # Essential boundary conditions
                                # on all electrodes
end

# Define the coefficients for the problem (first for the reference problem)
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients
  elgrp1
    coef 6= old_solution epsilon_vector # alpha 11=eps
                                        # The given permittivity field is stored
                                        # in vector epsilon_vector
    coef 9=coef 6                      # alpha 22=eps
end

coefficients, sequence_number=2
  elgrp1
    coef 6= old_solution epsilon_vector # alpha 11=eps
                                        # The permittivity field is stored
                                        # in vector epsilon_vector
                                        # This vector changes during the computation
    coef 9=coef 6                      # alpha 22=eps
end

# Define the structure of the main program
# See Users Manual Section 3.2.4

structure
```

```
# Define the structure of the large matrix

matrix_structure symmetric, reaction_force ! symmetric compact matrix

# Store the essential boundary conditions into the potential vector

prescribe_boundary_conditions, potential ! all are zero

# Solve the reference problem to compute the vector of measured values

create_vector, epsilon_vector
compute_capacity, capacity_vector
print capacity_vector

# Next solve inverse problem, using the just computed vector of
# measured values as right-hand side

solve_inverse_problem, epsilon_vector
print epsilon_vector

# post processing

plot_contour potential
plot_colored_levels potential

# Make colored_levels plot of permeability vector

plot_colored_levels epsilon_vector

# write result to output file

output

end

# Create the reference vector of permittivities
# See Users Manual Section 3.2.10

create
  type = vector defined per element 1
  value = eps # The reference value is set equal to eps
  elements 1, value = eps_1 # In element 1 it gets the value eps_1
end

# Input for the computation of the capacity
# See Users Manual Section 3.2.18

capacity
  lin_solver = 1 # Defines the type of linear solver
                # This is the default value
  seq_coef = 1 # Defines the coefficients (default value)
  curve_begin = sensor_first # curve number of first sensor
  curve_end = sensor_last # curve number of last sensor
  solution_vector = potential # sequence number of solution vector
  reaction_vector = reaction_force # sequence number of reaction force
```

```

end

# Input for the inverse problem
# See Users Manual Section 3.2.19

inverse_problem
  lin_solver = 1                # Defines the type of linear solver
                               # This is the default value
  seq_coef = 2                 # Defines the coefficients for the inverse
                               # problem
  curve_begin = sensor_first   # curve number of first sensor
  curve_end = sensor_last     # curve number of last sensor
  solution_vector = potential # sequence number of solution vector
  reaction_vector = reaction_force # sequence number of reaction force
  capacity_vector = capacity_vector # sequence number of capacity vector
                               # This vector must contain the measured
                               # values
  epsilon_vector = epsilon_vector # sequence number of permittivity vector
                               # This vector contains the result
                               # of the computations at output
                               # During the computations it is used to
                               # store various values of the permittivity
  element_group = 1           # The unknown medium corresponds to the
                               # first element group (default)
  method = capacity_simple    # Type of solution method (default)
  regular_parm = regular_parm # Regularization parameter
  eps_ref = eps_ref           # Reference permittivity
  delta_eps = delta_eps       # Step in permittivity
end
end_of_sepran_input

```

Figure 6.2.11.1 shows the equi-potential lines of the last potential computed in this example and Figure 6.2.11.2 the corresponding color plot. Figure 6.2.11.3 shows the computed permittivity field.

In general the inverse problem is solved in combination with reading a file of data. In that case the input file for program sepcomp is slightly different. Example inversesensorrd corresponds to that case.

To get this example into your local directory give the command:

```
sepgetex inversesensorrd
```

To run this example use:

```

sepmesh inversesensorrd.msh
jsepview sepplot.001
sepcomp inversesensorrd.prb
jsepview sepplot.001

```

The corresponding input file has the following structure:

```

# inversesensorrd.prb
# The capacities are read from a file
#

```

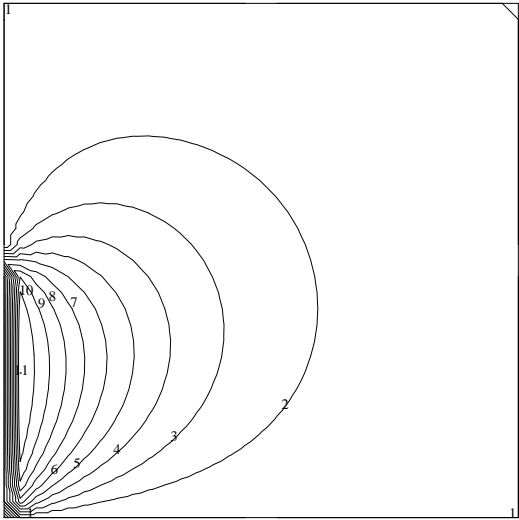


Figure 6.2.11.1: Equi-potential lines of last computed potential

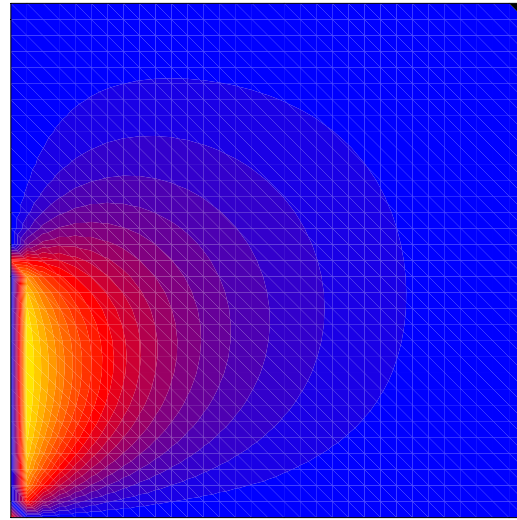


Figure 6.2.11.2: Colored potential levels of last computed potential

```

set warn off ! suppress warnings

# Problem for example inverse problem for the sensor
#
# Square grid with 8 electrodes
#
# This example shows how to solve an inverse problem
# See Users Manual Section 6.2.11
#
# To run this file use:
#   sepcomp inversesensor.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  integers
    sensor_first = 11      # curve number of first sensor
    sensor_last  = 18      # curve number of last sensor
                          # all curves sensor_first to sensor_last
                          # correspond to sensors
    nelectrodes = 8       # Number of electrodes (18-11+1)
  reals
    delta_eps = 0.1       # Step for computation of sensitivity matrix
    eps_ref   = 1         # Reference value for permeability used
                          # for computation of sensitivity matrix
    regular_parm = 0.01   # Regularization parameter to be used
                          # for computation of sensitivity matrix
  vector_names

```

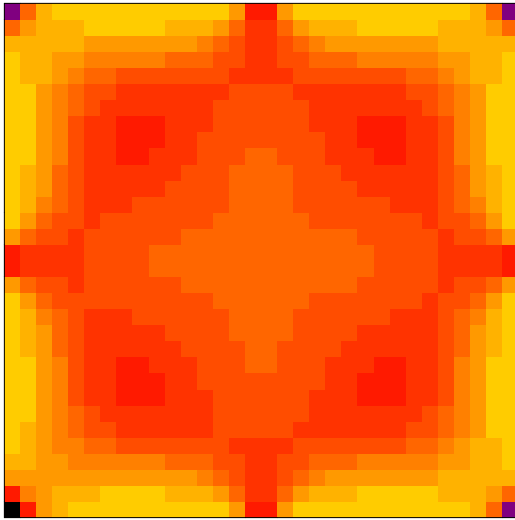



Figure 6.2.11.3: Colored levels of permittivity field

```

    potential          # solution vector
    reaction_force    # help vector to store the reaction force
    capacity_vector    # Vector containing all capacities
    epsilon_vector     # vector containing the permeabilities
end
#
# Define the type of problem to be solved
#
problem              # See Users Manual Section 3.2.2

    types            # Define types of elements,
                    # See Users Manual Section 3.2.2
    elgrp1=(type=800) # Type number for diffusion problem
                    # See Standard problems Section 3.1
    essboundcond     # Define where essential boundary conditions are
                    # given (not the value)
                    # See Users Manual Section 3.2.2
    curves(c sensor_first to c sensor_last) # Essential boundary conditions
                    # on all electrodes
end

# Define the coefficients for the problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients
    elgrp1
    coef 6= old_solution epsilon_vector # alpha 11=eps
                                        # The given permeability field is stored
                                        # in vector epsilon_vector
    coef 9=coef 6                       # alpha 22=eps
end

```

```
# Define the structure of the main program
# See Users Manual Section 3.2.4

structure

  # Define the structure of the large matrix

  matrix_structure symmetric, reaction_force ! symmetric compact matrix

  # Store the essential boundary conditions into the potential vector

  prescribe_boundary_conditions, potential ! all are zero

  # Read the measured values

  create_vector, capacity_vector
  print capacity_vector

  # Next solve inverse problem, using the just read vector of
  # measured values as right-hand side

  solve_inverse_problem, epsilon_vector
  print epsilon_vector

  # post processing

  plot_contour potential
  plot_colored_levels potential

  # Make colored_levels plot of permeability vector

  plot_colored_levels epsilon_vector

  # write result to output file

  output

end

# Create the reference vector of permeabilities
# See Users Manual Section 3.2.10

create
  type = capacity_vector, nelectrodes = nelectrodes # create a capacity
                                                    # vector
  file_capacity_values = 'capacity.file' # Read the data from the file
                                         # capacity file
end

# Input for the inverse problem
# See Users Manual Section 3.2.19

inverse_problem
  lin_solver = 1 # Defines the type of linear solver
```

```

seq_coef = 1 # This is the default value
              # Defines the coefficients for the inverse
              # problem
curve_begin = sensor_first # curve number of first sensor
curve_end = sensor_last # curve number of last sensor
solution_vector = potential # sequence number of solution vector
reaction_vector = reaction_force # sequence number of reaction force
capacity_vector = capacity_vector # sequence number of capacity vector
                                   # This vector must contain the measured
                                   # values
epsilon_vector = epsilon_vector # sequence number of permeability vector
                                   # This vector contains the result
                                   # of the computations at output
                                   # During the computations it is used to
                                   # store various values of the permeability
element_group = 1 # The unknown medium corresponds to the
                  # first element group (default)
method = capacity_simple # Type of solution method (default)
regular_parm = regular_parm # Regularization parameter
eps_ref = eps_ref # Reference permeability
delta_eps = delta_eps # Step in permeability
end
end_of_sepran_input

```

This input requires a user provided input file `capacity.file`. An example of such a file is:

```

28 #number of lines
1 2 7.84631E-01
1 3 9.90854E-02
1 4 5.36668E-02
1 5 5.41135E-02
1 6 4.53446E-02
1 7 1.71442E+00
1 8 9.90835E-02
2 3 1.79338E+00
2 4 9.90860E-02
2 5 4.53453E-02
2 6 5.41144E-02
2 7 9.90835E-02
2 8 5.36679E-02
3 4 7.84631E-01
3 5 5.36673E-02
3 6 9.90858E-02
3 7 5.41135E-02
3 8 4.53453E-02
4 5 9.90858E-02
4 6 1.79338E+00
4 7 4.53446E-02
4 8 5.41144E-02
5 6 7.84631E-01
5 7 9.90854E-02
5 8 1.79338E+00
6 7 5.36668E-02
6 8 9.90860E-02
7 8 7.84631E-01

```

6.2.12 An example of the use of arrays in the input block constants

In this section we will consider exactly the same problem as in Section 6.2.1. The only difference is that we perform a loop over three different κ values. Although the solution is not dependent on the actual value of κ , this demonstrates the use of arrays in the input block `constants` in combination with a for loop.

To get this example into your local directory give the command:

```
sepgetex array_example
```

To run the example use:

```
sepmesh array_example.msh
sepcomp array_example.prb
seppost array_example.pst
sepview sepplot.001
```

You may combine sepgetex and running by using:

```
sepexam array_example
```

The input files for sepmesh and seppost for this example are exactly the same as in Section 6.2.1. The only new part is the input file for program sepcomp, where an array is defined, a for loop is carried out and a variable coefficient is used.

```
# array_example.prb
#
# problem file for example of arrays in block constants
# See Users Manual Section 6.2.12
#
# To run this file use:
#   sepcomp array_example.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    kappa_array = 1, 10, 100  # array of diffusion parameters
  vector_names
    potential
  variables
    i                # Loop variable
    kappa            # diffusion parameter
end
#
# Define the type of problem to be solved
#
problem            # See Users Manual Section 3.2.2

  types            # Define types of elements,
```

```

                                # See Users Manual Section 3.2.2
    elgrp1=800                    # Type number for second order elliptic equation
                                # See Standard problems Section 3.1
    essbouncond                   # Define where essential boundary conditions are
                                # given (not the value)
                                # See Users Manual Section 3.2.2
    curves(c1 to c4)             # Essential boundary conditions on all boundaries
end
# Define the structure of the problem
# In this part it is described how the problem must be solved
# In this example a loop over the various kappa values is carried out
#
structure                        # See Users Manual Section 3.2.3
# Loop over all kappa values

    for i = 1 to 3

        # Prescribe the essential boundary conditions
        prescribe_boundary_conditions, potential

        # Copy the value of kappa from the ith position in kappa_array to kappa
        # This is necessary since only scalars may be used as coefficients
        kappa = kappa_array(i)

        # Build and solve the system of linear equations
        solve_linear_system, potential

        # Write the results to a file
        output
    end_for
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4
matrix
    storage_method = compact, symmetric    # Symmetrical compact matrix
                                           # So an iterative method will be applied
end

# Define the essential boundary conditions
# See Users Manual Section 3.2.5

essential boundary conditions
    curves(c3) value = 1                # At C3 T=1, at all other boundaries we
                                        # have T=0, which does not require input
end

# Define the coefficients for Laplacian equation
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients
    elgrp1 ( nparm=20 )                # The coefficients are defined by 20 parameters
    coef6 = kappa                      # a11 = kappa, in this case it is a scalar
                                        # variable
```

```
coef9 = coef 6          # a22 = kappa
end
```

6.3 Examples of non-linear problems

In this section we treat some examples of non-linear problems to show some of the possibilities to solve these equations.

The following examples will be treated:

6.3.1 An example of a simple Navier-Stokes equation.

This example shows how a simple laminar incompressible bend flow may be computed by SEPRAN. No special input is required, nor is the structure of the program defined by the user.

As a demonstration of the concept of constants the mesh utilizes the input block "CONSTANTS".

6.3.2 An example of a simple Navier-Stokes problem with a user defined structure.

This example is exactly the same as the one in 6.3.1, however in this case the user defines the structure of the program himself.

6.3.3 An example of a simple Navier-Stokes problem showing the use of the WHILE option in the user defined structure.

This example is exactly the same as the one in 6.3.1, however in this case the concept of non-linear equations is not used but instead the problem is solved as a series of linear equations. Convergence is tested using the WHILE concept in the user defined structure.

6.3.1 An example of a simple Navier-Stokes problem

In this section we will consider the laminar flow of an incompressible fluid through a bend as sketched in Figure 6.3.1.1.

In order to get the corresponding files into your local directory use:

```
sepgetex bend
```

To run the example use the following commands:

```
sepmesh bend.msh
sepview sepplot.001
sepcomp bend.prb
seppost bend.pst
sepview sepplot.001
```

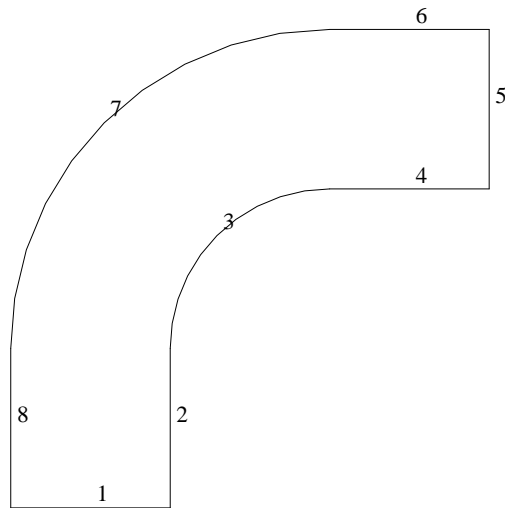


Figure 6.3.1.1: Definition region of the 90 degree bend

The instream velocity at curve C1 is assumed to be a quadratic velocity profile.

The curved boundaries C9 and C10 are fixed walls with a no-slip condition and at the outflow boundary C5, we do not impose boundary conditions. According to the Standard Problems Manual (Chapter 6.8.1) this is equivalent to a zero stress condition, implying that the pressure at outflow is equal to zero.

At the inflow and outflow of the bend straight pipes have been constructed in order to be able to prescribe more or less fully developed flow.

In order to create the mesh program SEPMESH may be used.

First we consider the mesh input file bend1.msh (The comments are sufficiently self explaining):

```
# bend.msh
#
# mesh file for bend problem
# See Users Manual Section 6.3.1
# Examples Section 7.1.12
```



```

#
# To run this file use:
#   sepmesh bend.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    r_in = 1      # inner radius of bend
    r_out = 2     # outer radius of bend
    width = 0     # width = r_out - r_in, see COMPCONS
    h_in = 1     # length of inflow pipe
    h_out = 1    # length of outflow pipe
    x_2 = -r_in  # right-hand x-coordinate of inlet
    y_2 = -h_in  # right-hand y-coordinate of inlet
    x_9 = -r_out # left-hand x-coordinate of inlet
  integers
    nelm_width = 5 # Number of elements in the cross section
    nelm_in = 3   # Number of elements in the length direction
                  # of the inflow pipe
    nelm_out = 3  # Number of elements in the length direction
                  # of the outflow pipe
    nelm_bend = 5 # Number of elements in the length direction
                  # of the bend
    shape_curve = 2 # Shape number of elements along curve
                   # (2 is quadratic)
    shape_surface = 4 # Shape number of elements in region
                     # (4 is quadratic triangle)
end
#
# Define the mesh
#
mesh2d          # See Users Manual Section 2.2
#
# user points
#
points
  p1 = ( 0, 0 ) # Center of arcs
  p2 = ( x_2, y_2 ) # right-hand point of inlet
  pd3 = ( r_in, 180 ) # lower point of circle at inner bend
                # Use polar co-ordinates
  pd4 = ( r_in, 90 ) # upper point of circle at inner bend
                # Use polar co-ordinates
  p5 = ( h_out, r_in ) # Point on outlet and inner bend
  p6 = ( h_out, r_out ) # Point on outlet and outer bend
  pd7 = ( r_out, 90 ) # upper point of circle at outer bend
                # Use polar co-ordinates
  pd8 = ( r_out, 180 ) # lower point of circle at outer bend
                # Use polar co-ordinates
  p9 = ( x_9, y_2 ) # left-hand point of inlet
#
# curves
#

```

```

curves          # See Users Manual Section 2.3
  c1 = line  shape_curve(p9,p2,nelm= nelm_width)    # inlet
  c2 = line  shape_curve(p2,p3,nelm= nelm_in)       # lower part of inner
                                                    # bend
  c3 = arc   shape_curve(p3,p4,-p1,nelm= nelm_bend) # circle part of inner
                                                    # bend
  c4 = line  shape_curve(p4,p5,nelm= nelm_out)      # upper part of inner
                                                    # bend
  c5 = line  shape_curve(p5,p6,nelm= nelm_width)    # outlet
  c6 = line  shape_curve(p6,p7,nelm= nelm_out)      # upper part of outer
                                                    # bend
  c7 = arc   shape_curve(p7,p8,p1,nelm= nelm_bend) # circle part of outer
                                                    # bend
  c8 = line  shape_curve(p8,p9,nelm= nelm_in)       # lower part of outer
                                                    # bend
  c9 = curves(c2,c3,c4)                            # inner bend
  c10= curves(c6,c7,c8)                            # outer bend

#
# surfaces
#
surfaces        # See Users Manual Section 2.4
  s1 = rectangle shape_surface (c1,c9,c5,c10)

plot            # make a plot of the mesh
               # See Users Manual Section 2.2
end

```

The flow problem is nonlinear and it is solved with the standard element of type 900 as described in the manual Standard Problems Section 7.1.

The physical parameters are density $\rho = 1$, viscosity $\eta = 0.01$ and the penalty parameter $\epsilon = 10^{-6}$. The inflow profile is parabolic with a maximum velocity of 1 *m/sec*, resulting in a Reynolds number of 100.

sepcomp requires input from the standard input file describing the problem definition, the boundary conditions and the non-linear solution process.

In the first iteration a Stokes problem is solved, the second iteration is based upon a Picard linearization and for all other iterations a Newton linearization is used.

The input file bend.prb can be used for this example:

```

# bend.prb
#
# problem file for 2d bend problem
# penalty function approach
# problem is stationary and non-linear
# See Users Manual Section 6.3.1
# Examples Section 7.1.12
#
# To run this file use:
#   sepcomp bend.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
set warn off    ! suppress warnings
#

```

```
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    rho            = 1                # density
    eta            = 0.01             # viscosity
  vector_names
    velocity
    pressure
end
#
# Define the type of problem to be solved
#
problem            # See Users Manual Section 3.2.2

  types            # Define types of elements,
                  # See Users Manual Section 3.2.2
    elgrp1=900    # Type number for Navier-Stokes, without swirl
                  # 6-point triangle
                  # Approximation 7-point extended triangle
                  # Penalty function method
                  # See Standard problems Section 7.1
    essbouncond   # Define where essential boundary conditions are
                  # given (not the value)
                  # See Users Manual Section 3.2.2
    curves(c1)    # Inflow
    curves(c9)    # Inner bend (noslip)
    curves(c10)   # Outer bend (noslip)
                  # All not prescribed boundary conditions
                  # satisfy corresponding stress is zero
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

matrix
  # Non-symmetrical profile matrix, So a direct method will be applied
end

# Create start vector and put the essential boundary conditions into this
# vector
# See Users Manual Section 3.2.5

essential boundary conditions

  curves(c1), degfd2, quadratic # The v-component of the velocity at
                                # inflow is quadratic

end

# Define the coefficients for the problems (first iteration)
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 7.1

coefficients
```

```

    elgrp1 ( nparm=20 )      # The coefficients are defined by 20 parameters
      icoef2 = 1            # 2: type of constitutive equation (1=Newton)
      icoef5 = 0            # 5: Type of linearization (0=Stokes flow)
      coef6 = 1d-6         # 6: Penalty function parameter eps
      coef7 = rho          # 7: Density
      coef12 = eta         #12: Value of eta (viscosity)
end

# Define the coefficients for the next iterations
# See Users Manual Section 3.2.7

change coefficients, sequence_number = 1 # Input for iteration 2
  elgrp1
    icoef5 = 1            # 5: Type of linearization (1=Picard iteration)
end

change coefficients, sequence_number = 2 # Input for iteration 3
  elgrp1
    icoef5 = 2            # 5: Type of linearization (2=Newton iteration)
end

# input for non-linear solver
# See Users Manual Section 3.2.9

nonlinear_equations
  global_options, maxiter=10, accuracy=1d-4, print_level=1, lin_solver=1
  equation 1
    fill_coefficients 1
    change_coefficients
      at_iteration 2, sequence_number 1
      at_iteration 3, sequence_number 2
end

# Define output, and compute pressure

output
  v1 = icheld=7 # pressure
end

end_of_sepran_input

```

As usual postprocessing may be performed by SEPPOST.
 We show the sample input file bend.pst:

```

# bend.pst
# Input file for postprocessing for channel problem
# See Users Manual Section 6.3.1
# Examples Section 7.1.12
#
#
# To run this file use:
#   seppost bend.pst > bend.out
#
# Reads the files meshoutput and sepcomp.out
#

```

```
postprocessing          # See Users Manual Section 5.2
#
# Print both vectors completely

    print velocity
    print pressure

# compute the stream function
# See Users Manual Section 5.2
# store in stream_function

    compute stream function velocity

# Plot the results
# See Users Manual Section 5.4

    plot vector velocity          # Vector plot of velocity
    plot contour pressure        # Contour plot of pressure
    plot coloured contour pressure
    plot contour stream_function # Contour plot of stream function
    plot coloured contour stream_function

end
```

6.3.2 An example of a simple Navier-Stokes problem with a user defined structure

In this section we will consider exactly the same problem as in Section 6.3.1.

In order to get the corresponding files into your local directory use:

```
sepgetex bend1
```

To run the example use the following commands:

```
sepmesh bend1.msh
sepview sepplot.001
sepcomp bend1.prb
seppost bend1.pst
sepview sepplot.001
```

The only difference is that we use explicitly the input block STRUCTURE to define the course of the program.

Since the files bend1.msh and bend1.pst are exactly the same as in Section 6.3.1 we do not repeat them here.

The file bend.prb however, is replaced by the file bend1.prb:

```
*bend1.prb
set warn off      ! suppress warnings
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  vector_names
    velocity
    pressure
end
problem
  # Define type of elements
  types
    elgrp1=900          # Type number for Navier-Stokes, without swirl
                      # 6-point triangle
                      # Approximation 7-point extended triangle
                      # Penalty function method
  # Define where essential boundary conditions are present
  essbouncond
    curves(c1)         # Inflow
    curves(c9)         # Inner boundary (noslip)
    curves(c10)        # Outer boundary (noslip)
end

* define type of matrix

matrix
  # Non-symmetrical profile matrix, So a direct method will be applied
end

# Define the structure of the main program

structure
```

```

    prescribe_boundary_conditions, velocity
    solve_nonlinear_system velocity
    print velocity
    output
end

* Create start vector and put the essential boundary conditions into this
* vector

essential boundary conditions
    value = 0                # First set vector equal to zero

    # Next fill all non-zero essential boundary conditions
    curves(c1), degfd2, quadratic # The v-component of the velocity at
                                # inflow is quadratic

end

* Define coefficients for the first iteration

coefficients
    elgrp1 ( nparm=20)      # The coefficients are defined by 8 parameters
    icoef2 = 1              # 2: type of constitutive equation (1=Newton)
    icoef5 = 0              # 5: Type of linearization (0=Stokes flow)
    coef6 = 1d-6           # 6: Penalty function parameter eps
    coef7 = 1              # 7: Density
                                # 8: angular velocity = 0
                                # 9: body force in x-direction = 0
                                #10: body force in y-direction = 0
    coef12 = 0.01         #12: Value of etha (viscosity)
end

* Define the coefficients for the next iterations

change coefficients, sequence_number = 1 # Input for iteration 2
    elgrp1
        icoef5 = 1          # 3: Type of linearization (1=Picard iteration)
    end

change coefficients, sequence_number = 2 # Input for iteration 3
    elgrp1
        icoef5 = 2          # 3: Type of linearization (2=Newton iteration)
    end

* Define the parameters for the non-linear solver

nonlinear_equations, sequence_number = 1
    global_options, maxiter=10, accuracy=1d-4, print_level=1, lin_solver=1
    equation 1
        fill_coefficients 1
        change_coefficients
            at_iteration 2, sequence_number 1
            at_iteration 3, sequence_number 2
    end
end

```

```
* Define output, and compute pressure
```

```
output
```

```
  v1 = icheld=7  # pressure
```

```
end
```

```
end_of_sepran_input
```


6.3.3 An example of a simple Navier-Stokes problem showing the use of the WHILE option in the user defined structure

In this section we will consider exactly the same problem as in Section 6.3.1.

The only difference is that we use explicitly the input block STRUCTURE in combination with the WHILE concept to define the course of the program.

Since the files bend.msh and bend.pst are exactly the same as in Section 6.3.1 we do not repeat them here.

In order to get the files into your local directory use:

```
sepgetex bend2
```

The file bend.prb however, is replaced by the file bend2.prb.

In this example we start with the solution of the Stokes equation as initial guess and then repeat the iterations with Newton linearization until the difference between two succeeding iterations is less than the required accuracy ϵ .

The norm of the difference of two succeeding iterations is stored in scalar 1, which gets the name max_differ in the input block CONSTANTS.

The WHILE statement requires a boolean expression.

The input file bend2.prb reads:

```
*bend2.prb
set warn off      ! suppress warnings
constants
  reals
    eps = 1d-4          # Accuracy for non-linear iteration
  variables
    max_differ = 1      # Norm of difference between two succeeding
                        # iterations
  vector_names
    velocity
    vel_prev
    pressure
end
problem
  # Define type of elements
  types
    elgrp1=900          # Type number for Navier-Stokes, without swirl
                        # 6-point triangle
                        # Approximation 7-point extended triangle
                        # Penalty function method
  # Define where essential boundary conditions are present
  essbouncond
    curves(c1)          # Inflow
    curves(c9)          # Inner boundary (noslip)
    curves(c10)         # Outer boundary (noslip)
end

* define type of matrix

matrix
  # Non-symmetrical profile matrix, So a direct method will be applied
end

# Define the structure of the main program
```

```

structure
  prescribe_boundary_conditions, velocity

  # First step: solve Stokes equation as start
  solve_linear_system, seq_coef=1, velocity

  # Next solve the Navier-Stokes equations using the while concept
  # The process is stopped when the difference between two iterations
  # is less than eps
  # Newton linearization is applied

  while ( max_differ>eps ) do
    vel_prev = velocity
    solve_linear_system, seq_coef=2, velocity
    max_differ = norm_dif=3, vector1 = velocity, vector2 = vel_prev
    print max_differ, text='Max difference = '
  end_while

  output
end

* Create start vector and put the essential boundary conditions into this
* vector

essential boundary conditions
  value = 0                # First set vector equal to zero

  # Next fill all non-zero essential boundary conditions
  curves(c1), degfd2, quadratic # The v-component of the velocity at
                                # inflow is quadratic

end

* Define coefficients for the first iteration

coefficients, sequence_number = 1
  elgrp1 ( nparm=20)      # The coefficients are defined by 8 parameters
  icoef2 = 1              # 2: type of constitutive equation (1=Newton)
  icoef5 = 0              # 5: Type of linearization (0=Stokes flow)
  coef6 = 1d-6           # 6: Penalty function parameter eps
  coef7 = 1               # 7: Density
                          # 8: angular velocity = 0
                          # 9: body force in x-direction = 0
                          #10: body force in y-direction = 0
  coef12 = 0.01         #12: Value of etha (viscosity)
end

* Define coefficients for the next iteration

coefficients, sequence_number = 2
  elgrp1 ( nparm=20)      # The coefficients are defined by 8 parameters
  icoef2 = 1              # 2: type of constitutive equation (1=Newton)
  icoef5 = 2              # 5: Type of linearization (2=Newton)
  coef6 = 1d-6           # 6: Penalty function parameter eps

```

```
coef7 = 1          # 7: Density
                  # 8: angular velocity = 0
                  # 9: body force in x-direction = 0
                  #10: body force in y-direction = 0
coef12 = 0.01     #12: Value of etha (viscosity)
end

* Define output, and compute pressure

output
  v2 = icheld=7    # pressure
end
end_of_sepran_input
```

6.4 Examples of time-dependent problems

In this section we treat some examples of (artificial) time-dependent problems to show some of the possibilities to solve these equations.

The following examples will be treated:

6.4.1 An example of a simple heat equation.

This example shows how a simple artificial heat equation may be solved by SEPRAN. No special input is required, nor is the structure of the program defined by the user.

6.4.2 An example of a simple heat equation with a user defined structure.

This example is exactly the same as the one in [6.4.1](#), however in this case the user defines the structure of the program himself.

6.4.3 An example of the solution of a coupled set of time-dependent equations.

This example is a non-linear artificial example, which shows how coupled non-linear time-dependent problems may be solved

6.4.4 An example of a stationary equation solved by the limit of a time-dependent problem.

In this example the time-dependent equations are used to iterate to the final stationary solution.

6.4.5 An example of a time-dependent equation coupled with a stationary equation.

In this example it is shown how a time-dependent problem may be coupled with a stationary problem that must be solved in each time step. The construction `time_loop` in the structure block is used.

6.4.1 An example of a simple heat equation

In this section we will consider the solution of a simple heat equation defined on a unit square by SEPRAN.

This example is just meant to demonstrate the use of the time integration in the case of a SEPRAN program. No special options are used.

Consider the heat equation

$$\frac{\partial T}{\partial t} - 0.5\Delta T = 0 \quad (6.4.1.1)$$

with Δ the Laplacian operator. We assume that the region at which this equation is defined is the unit square $(0, 1) \times (0, 1)$.

We suppose that the initial condition is given by

$$T(\mathbf{x}, 0) = \sin(\mathbf{x})\sin(\mathbf{y})$$

and the boundary conditions by

$$T(\mathbf{x}, \mathbf{t}) = \sin(\mathbf{x})\sin(\mathbf{y})\exp(-\mathbf{t}) \text{ at all four boundaries.}$$

It is easy to verify that the exact solution in this case is also equal to

$$T(\mathbf{x}, \mathbf{t}) = \sin(\mathbf{x})\sin(\mathbf{y})\exp(-\mathbf{t})$$

To get this example into your local directory use:

```
sepgetex heatequ1
```

In order to solve this problem a mesh is created by sepmesh using the submesh generator general. An example input file for sepmesh is the file heatequ1.msh:

```
* heatequ1.msh
*
* mesh for the unit square (0,1) x (0,1)
mesh2d
  coarse(unit=0.1)
  points
    p1=(0,0,1)
    p2=(1,0,1)
    p3=(1,1,1)
    p4=(0,1,1)
  curves
    c1=cline1(p1,p2)
    c2=cline1(p2,p3)
    c3=cline1(p3,p4)
    c4=cline1(p4,p1)
  surfaces
    s1=general3(c1,c2,c3,c4)
  plot (jmark=5, numsub=1)
end
```

In order to prescribe the initial conditions and the boundary conditions it is necessary to provide function subroutines since both depend on space and time. The boundary condition is time-dependent, which implies that the time t must be present. This time t can be found in the common block CTIMEN as described in Section 3.2.15.

The main program may have the following shape (file heatequ1.f)

```
program heatequ1
```

```

        implicit none
        call sepcom(0)
        end

! *****
!
!   function func for the initial condition
!
! *****
        function func ( icoice, x, y, z )
        implicit none
        double precision func, x, y, z
        integer icoice
        double precision t, tout, tstep, tend, t0, rtimdu
        integer iflag, icons, itimdu
        common /ctimen/ t, tout, tstep, tend, t0, rtimdu(5), iflag,
+             icons, itimdu(8)

        func = exp(-t)*sin(x)*sin(y)

        end

! *****
!
!   function for essential boundary conditions
!
! *****
        function funcbc ( icoice, x, y, z )
        implicit none
        double precision funcbc, x, y, z
        integer icoice
        double precision t, tout, tstep, tend, t0, rtimdu
        integer iflag, icons, itimdu
        common /ctimen/ t, tout, tstep, tend, t0, rtimdu(5), iflag,
+             icons, itimdu(8)

        if ( icoice==1 ) then
            funcbc = sin(x)*sin(y)*exp(-t)
        else if ( icoice==2 ) then
            funcbc = sin(x)*sin(y)*exp(-t)
        else if ( icoice==3 ) then
            funcbc = sin(x)*sin(y)*exp(-t)
        else if ( icoice==4 ) then
            funcbc = sin(x)*sin(y)*exp(-t)
        end if

        end

```

Mark that the boundary conditions at the four sides have the same structure and that one choice parameter would be sufficient.

The way the problem is solved is completely defined by the input file for the program `heatequation_1`. The following input file (`heatequ1.prb`) may for example be used:

```

* heatequ1.prb
*
set warn off ! suppress warnings

```

```
* problem definition for time-dependent heat equation
* linear triangles type number 800

constants          # See Users Manual Section 1.4
  vector_names
  potential
end

problem
  types
  elgrp1 = 800
  essbouncond
  curves(c1,c4)
end

*
* Definition of matrix structure
*
matrix
  symmetric
end

*
* Define initial conditions
*
create vector
  func = 1
end

*
* Essential boundary conditions
*
essential boundary conditions
  curves(c1),(func=1)
  curves(c2),(func=2)
  curves(c3),(func=3)
  curves(c4),(func=4)
end

*
* Definition of coefficients for the heat equation (t=0 only)
*
coefficients
  elgrp1(nparm=20)
  coef6 = 0.5          # a11 = 0.5
  coef9 = coef 6      # a22 = 0.5
  coef17 = 1          # rho = 1
end

time_integration, sequence_number = 1
  method = euler_implicit
  tinit = 0
  tend = 1
  timestep = 0.1
  toutinit = 0
  toutend = 1
  toutstep = 0.1
  seq_boundary_conditions = 1
  seq_coefficients = 1
```

```
diagonal_mass_matrix
stiffness_matrix = constant
mass_matrix = constant
right_hand_side = zero
end
```

In this input file the elements are defined by type number 800, which is the standard element for general second order elliptic and parabolic equations. A description of this element can be found in the manual Standard Problems Section 3.1.

At all four boundaries we have essential boundary conditions.

Since no convection is present in this example the matrix is symmetric and positive definite.

The only coefficients that have to be given in this particular case are the coefficients a_{11} and a_{22} , which are both equal to 0.5 and the parameter $\rho c p$ which is equal to 1.

Since the problem is time-dependent it is necessary to have a block `TIME_INTEGRATION` in the input file, otherwise only the stationary problem will be solved. In this block all necessary information for the time integration must be given. In this example it has been chosen to use the Euler implicit method, with the step size `TSTEP` equal to 0.1. The initial time for the integration is 0 and the end time 1. In each time step the solution is written to the file `sepcomp.out`.

In this example both the stiffness matrix and the mass matrix are independent of time. Since linear elements are used, we may apply a diagonal mass matrix. There is no force term present in the equation, hence the right-hand side is equal to zero.

The input block `CREATE` is necessary to create the initial condition, if omitted the initial condition is made equal to zero.

In order to run this program it is necessary to link the program `heatequation_1` with the `sepran` libraries with the command `seplink`:

```
seplink heatequ1
```

To run the program we use

```
heatequ1 < heatequ1.prb
```

The solution may be visualised by `seppost` using the file `heatequ1.pst` as input file:

```
* heatequ1.pst
*
set warn off ! suppress warnings
*
* input for seppost
*
postprocessing
  time = (0,1)
  print potential
  plot contour potential, minlevel = 0, maxlevel = 1
  time history plot point(.5,.5) potential
end
```


6.4.2 An example of a simple heat equation with a user defined structure

The example we will consider in this section is exactly the same as the one treated in Section 6.4.1. The only difference is that, since we know what the exact solution is, we also want to compute the accuracy of the solution. To that end we use exactly the same program and input files as in Section 6.4.1. The only difference is that the input file for the program `heatequ1` in this case must contain a block `STRUCTURE` which defines the structure of the main program and also computes the error. In order to make a slight difference the Euler implicit method has been replaced by Crank Nicolson. The input file (`heatequ2.prb`) for the program `heatequ1.prb` is now:

```
* heatequ2.prb
*
set warn off ! suppress warnings

* problem definition for time-dependent heat equation
* linear triangles type number 800

constants          # See Users Manual Section 1.4
  vector_names
    potential
    exact_potential
  variables
    error
end

problem
  types
    elgrp1 = 800
  essbouncond
    curves(c1,c4)
end

*
* Definition of matrix structure
*
matrix
  symmetric
end

structure
  create_vector, potential
  solve_time_dependent_problem
  create_vector, exact_potential
  error = norm_dif=3,vector1=potential, vector2=exact_potential
  print error, text = 'difference at time = 1'
end

*
* Define initial conditions
*
create vector
  func = 1
end

*
* Essential boundary conditions
*
essential boundary conditions
  curves(c1,c4),(func=1)
end
```

```
*
* Definition of coefficients for the heat equation (t=0 only)
*
coefficients
  elgrp1(nparm=20)
    coef6 = 0.5           # a11 = 0.5
    coef9 = coef 6       # a22 = 0.5
    coef17 = 1           # rho = 1
  end
time_integration, sequence_number = 1
  method = crank_nicolson
  tinit = 0
  tend = 1
  tstep = 0.1
  toutinit = 0
  toutend = 1
  toutstep = 0.1
  seq_boundary_conditions = 1
  seq_coefficients = 1
  diagonal_mass_matrix
  stiffness_matrix = constant
  mass_matrix = constant
  right_hand_side = zero
end
```

The block STRUCTURE defines the complete program.

First the initial condition is created by the command CREATE_VECTOR.

Next the time-dependent solution is solved.

After that the exact solution at time $t = 1$ is created and stored in the vector V2.

Finally the difference between exact and computed solution is computed and printed.

6.4.3 An example of the solution of a coupled set of time-dependent equations

In this section we consider another artificial example, the solution of two coupled non-linear time-dependent simple parabolic equations. This example is introduced in order to show how coupled equations may be handled. This example is available in 4 versions:

timedcop Standard case, only one call to a time dependent solver

timedcop_01 Special case, the time dependent solver is replaced by a time loop in the structure block.

Both equations are solved by a separate call to the time integration routine.

timedcop_02 See timedcop_01, however, in this case only one call to the time integration routine is made. Both equations are solved consecutively in this subroutine.

timedcop_03 See timedcop_01, however, now the boundary conditions are filled in extra vectors outside the integration subroutine.

To get one of these example in your local directory use:

```
sepgetex timedcopxx
```

with xx either nothing, _01, _02, or _03. To run this example use the following commands:

```
sepmesh timedcopxx.msh
view the plots
seplink timedcopxx
timedcop < timedcopxx.prb
seppost timedcopxx.pst
view the plots
```

Consider the following set of parabolic equations:

$$\frac{\partial p}{\partial t} - \Delta p + Tp = x + 3t^2x^2 \quad (6.4.3.1)$$

$$\frac{\partial T}{\partial t} - \Delta T + Tp = 3x + 3t^2x^2 \quad (6.4.3.2)$$

defined at the unit square $(0, 1) \times (0, 1)$.

At the boundaries we impose the following boundary conditions:

$$p = tx \quad (6.4.3.3)$$

$$T = 3tx \quad (6.4.3.4)$$

One easily verifies that the exact solution of this artificial problem is also given by equations [6.4.3.3](#) and [6.4.3.4](#).

As usual the mesh is created by sepmesh for example using the following input file:

```
# timedcop.msh
#
# mesh file for the solution of a coupled set of time-dependent equations
# See Users Manual Section 6-4-3
#
# To run this file use:
#   sepmesh timedcop.msh
#
```

```

# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    width = 1      # width of the region
    height = 1     # height of the region
  integers
    m = 5          # number of elements in height direction
    n = 5          # number of elements in width direction
    lin = 1        # linear elements
end
#
# Define the mesh
#
mesh2d             # See Users Manual Section 2.2
#
# user points
#
  points          # See Users Manual Section 2.2
    p1=(0,0)      # Left under point
    p2=( height,0) # Right under point
    p3=( height, width) # Right upper point
    p4=(0, width) # Left upper point
#
# curves
#
  curves          # See Users Manual Section 2.3
                 # Linear elements are used
    c1=line  lin (p1,p2,nelm= n)    # lower boundary
    c2=line  lin (p2,p3,nelm= m)    # right-hand side boundary
    c3=line  lin (p3,p4,nelm= n)    # upper boundary
    c4=line  lin (p4,p1,nelm= m)    # left-hand side boundary
#
# surfaces
#
  surfaces        # See Users Manual Section 2.4
                 # Linear triangles are used
    s1=rectangle3(c1,c2,c3,c4)

  plot           # make a plot of the mesh
                # See Users Manual Section 2.2

end

```

In order to prescribe the initial conditions and the boundary conditions it is necessary to provide function subroutines since both depend on space and time. The boundary condition is time-dependent, which implies that the time t must be present. This time t can be found in the common block CTIMEN as described in Section 3.2.15.

The main program may have the following shape (file timedcop.f)

```

program timedcop

!   --- Main program for for the solution of a coupled set of time-dependent

```

```

!      equations as described in the Users Manual Section 6-4-3
!      This program is only necessary since function subroutines must be
!      provided to define the time and space dependent coefficients
!      boundary conditions, initial condition and exact solution

      implicit none
      call sepcom(0)
      end

! *****
!
!      function func for the initial condition and the exact solution
!
! *****
      function func ( icoice, x, y, z )
      implicit none
      double precision func, x, y, z
      integer icoice

      include 'SPcommon/ctimen'

      if ( icoice==1 ) then
         func = t * x
      else if ( icoice==2 ) then
         func = 3d0 * t * x
      end if

      end

! *****
!
!      function funcfc for the coefficients
!
! *****
      function funcfc ( icoice, x, y, z )
      implicit none
      double precision funcfc, x, y, z
      integer icoice

      include 'SPcommon/ctimen'

      if ( icoice==3 ) then
         funcfc = x + 3d0 * ( t * x ) ** 2
      else if ( icoice==4 ) then
         funcfc = 3d0 * x + 3d0 * ( t * x ) ** 2
      end if

      end

! *****
!
!      function for essential boundary conditions :
!
! *****
      function funcbc ( icoice, x, y, z )
      implicit none
      double precision funcbc, x, y, z

```

```

integer ichoice

include 'SPcommon/ctimen'

if ( ichoice==1 ) then
  funcbc = t * x
else if ( ichoice==2 ) then
  funcbc = 3d0 * t * x
end if

end

```

In our example the unknown p is considered as first unknown and T as second one.

The input file for program `timedcop` (`timedcop.prb`) actually defines the complete structure of the program as well as what is solved.

The two equations are solved as separate equations by elements of the type 800 as described in the manual STANDARD PROBLEMS Section 3.1. In each time step first the p equation is solved using the value of T at the preceding time level and then the T equation is solved using the value of p at the present time level.

The following input file may be used for example:

```

# timedcop.prb
#
# problem file for the solution of a coupled set of time-dependent equations
# See Users Manual Section 6-4-3
#
# To run this file use:
#   sepcomp timedcop.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    kappa = 1          # diffusion coefficient
    rho   = 1          # density times heat capacity
    t0    = 0          # initial time
    t_end = 1          # end time
    dt    = 0.1        # time step
  vector_names
    Pressure
    Temperature
    P_exact
    T_exact
  variables
    p_error
    T_error
end
#
# Define the type of problems to be solved
#

```

```
problem 1          # See Users Manual Section 3.2.2
                  # This concerns the first problem with the
                  # pressure p as unknown
    types          # Define types of elements,
                  # See Users Manual Section 3.2.2
        elgrp1=800 # Type number for Laplacian equation
                  # See Standard problems Section 3.1
        essbouncond # Define where essential boundary conditions are
                  # given (not the value)
                  # See Users Manual Section 3.2.2
        curves(c1 to c4) # There are essential boundary conditions on
                  # all four boundaries

problem 2          # This concerns the second problem with the
                  # temperature T as unknown
    types          # Define types of elements,
                  # See Users Manual Section 3.2.2
        elgrp1=800 # Type number for Laplacian equation
                  # See Standard problems Section 3.1
        essbouncond # Define where essential boundary conditions are
                  # given (not the value)
                  # See Users Manual Section 3.2.2
        curves(c1 to c4) # There are essential boundary conditions on
                  # all four boundaries

end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

matrix
    symmetric, problem = 1    # Symmetrical profile matrix
                              # So a direct method will be applied
    symmetric, problem = 2    # The same for the second problem
end

# Create the start vector for the process, i.e. the solution at t=0
# See Users Manual Section 3.2.10
# By omitting the end between both creates pressure and temperature
# are created in one call.
# This option is not recommended and only works because pressure and
# temperature are consecutive solution vectors

create vector 1, problem 1
    function = 1              # The initial condition for the pressure is
                              # a function of x and y
                              # This function is defined in subroutine
                              # func, see the main program
                              # The parameter ichoice gets value 1

create vector 2, problem 2
    function = 2              # The initial condition for the temperature is
                              # a function of x and y
                              # This function is defined in subroutine
                              # func, see the main program
                              # The parameter ichoice gets value 2

end
```

```
# Define the essential boundary conditions for both vectors pressure
# and temperature
# See Users Manual Section 3.2.5
# Since we are using a coupled solution, no end between the two is allowed

essential boundary conditions 1
  func=1                # The boundary condition for the pressure is
                        # a function of t, x and y
                        # This function is defined in subroutine
                        # funcbc, see the main program
                        # The parameter ichoice gets value 1
essential boundary conditions 2
  func=2                # The boundary condition for the temperature is
                        # a function of t, x and y
                        # This function is defined in subroutine
                        # funcbc, see the main program
                        # The parameter ichoice gets value 2

end

# Define the coefficients for the pressure problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 1
  elgrp1(nparm=20)      # The coefficients are defined by 20 parameters
  coef6 = kappa         # a11 = kappa
  coef9 = coef 6        # a22 = kappa
  coef15 = old solution Temperature # beta = T
  coef16 = func=3       # The right-hand side is a
                        # a function of t and x defined by
                        #  $f = x + 3 t^2 x^2$ 
                        # This function is defined in subroutine
                        # funcfc, see the main program
                        # The parameter ichoice gets value 3
  coef17 = rho          # rho_cp = rho

end

# Define the coefficients for the temperature problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 2
  elgrp1(nparm=20)      # The coefficients are defined by 20 parameters
  coef6 = kappa         # a11 = kappa
  coef9 = coef 6        # a22 = kappa
  coef15 = old solution pressure   # beta = p
  coef16 = func=4       # The right-hand side is a
                        # a function of t and x defined by
                        #  $g = 3 x + 3 t^2 x^2$ 
                        # This function is defined in subroutine
                        # funcfc, see the main program
                        # The parameter ichoice gets value 4
  coef17 = rho          # rho_cp = rho

end
```



```

# Define the time integration process
# See Users Manual Section 3.2.15

time_integration, sequence_number = 1
  method = euler_implicit           # Time discretization algorithm
  number_of_coupled_equations = 2   # p and T are treated as a
                                     # coupled system
  tinit = t0                         # initial time
  tend = t_end                       # end time
  timestep = dt                      # time step
  toutinit = t0                     # initial time for output
  toutend = t_end                   # end time for output
  toutstep = dt                    # time step for output
  seq_boundary_conditions = 1        # defines which essential boundary
                                     # conditions must be used for
  seq_coefficients = 1, 2           # defines the coefficients for
                                     # both equations
  diagonal_mass_matrix              # The mass matrix is lumped
  mass_matrix = constant             # and constant for both problems
end

# Write only pressure and temperature to output file for postprocessing,
# not the exact solutions
# See Users Manual, Section 3.2.13

output
  write 2 solutions
end

# Define the structure of the problem
# In this part it is described how the problem must be solved
# This is necessary since some extra output is required
#

structure
  # Create initial condition for pressure and temperature
  create_vector, pressure
  # Solve the coupled time dependent problems in a decoupled way
  solve_time_dependent_problem

  # Create the exact solution for pressure and temperature and t = tend
  create_vector, p_exact

  # Compute and print the error of both solutions at t = tend
  p_error = norm_dif=3,vector1=pressure, vector2=p_exact
  print p_error, text = 'difference in p at time = 1'
  T_error = norm_dif=3,vector1=Temperature, vector2=t_exact
  print T_error, text = 'difference in T at time = 1'
end

```

The program starts with the creation of the initial conditions by the command *create_vector*. In this case both the *p* and *T* vectors are created.

Next the time-dependent equations are solved from $t = 0$ to $t = 1$ and in each time-step the solution is written to the file *sepcomp.out*.

Finally the exact solution is stored in vectors 3 and 4. The reason to use vectors 3 and 4 is that already p is stored in vector 1 and T in vector 2.

Mark that although in both cases only one vector is mentioned in the *structure* block actually in each case two vectors are created because of the input.

The difference between exact solution and numerical solution is computed and printed.

Post processing may be performed in the standard way by program seppost. See for example Section [6.4.1](#).

Instead of using the statement `solve_time_dependent_problem` in the structure block, it is also possible to perform an explicit time loop in that block. This offers the opportunity to perform all kinds of extra actions during the time stepping procedure. There are two alternatives to do this:

- By calling the integration subroutine for both equations separately
- By solving the equations in a "coupled" way.

To get the first option in your local directory use:

```
sepgetex timedcop_01
```

and for the second one:

```
sepgetex timedcop_02
```

The corresponding problem files are:

```
# timedcop_01.prb
#
# problem file for the solution of a coupled set of time-dependent equations
# In this case a time loop in the structure block is used
# See Users Manual Section 6-4-3
#
# To run this file use:
#   sepcomp timedcop_01.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    kappa = 1          # coefficient in diffusion term
    rho   = 1          # density times heat capacity
    t0    = 0          # initial time
    t_end = 1          # end time
    dt    = 0.1        # time step
  vector_names
    1: Pressure
    2: Temperature
    3: P_exact
    4: T_exact
  variables
    p_error
    T_error
end
#
# Define the type of problems to be solved
#
problem 1          # See Users Manual Section 3.2.2
                  # This concerns the first problem with the
                  # pressure p as unknown
```

```
types                                # Define types of elements,
                                     # See Users Manual Section 3.2.2
    elgrp1=800                        # Type number for Laplacian equation
                                     # See Standard problems Section 3.1
    essbouncond                       # Define where essential boundary conditions are
                                     # given (not the value)
                                     # See Users Manual Section 3.2.2
    curves(c1 to c4)                 # There are essential boundary conditions on
                                     # all four boundaries

problem 2

types                                # This concerns the second problem with the
                                     # temperature T as unknown
    elgrp1=800                        # Define types of elements,
                                     # See Users Manual Section 3.2.2
    essbouncond                       # Type number for Laplacian equation
                                     # See Standard problems Section 3.1
    curves(c1 to c4)                 # Define where essential boundary conditions are
                                     # given (not the value)
                                     # See Users Manual Section 3.2.2
                                     # There are essential boundary conditions on
                                     # all four boundaries

end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

matrix
    symmetric, problem = 1           # Symmetrical profile matrix
                                     # So a direct method will be applied
    symmetric, problem = 2           # The same for the second problem
end

# Create the start vector for the process, i.e. the solution at t=0
# See Users Manual Section 3.2.10

create vector, problem 1, sequence_number = 1
                                     # Mark that no sequence number is given after
                                     # create vector. The vector that is used is
                                     # defined in the structure block
    function = 1                     # The initial condition for the pressure is
                                     # a function of x and y
                                     # This function is defined in subroutine
                                     # func, see the main program
                                     # The parameter ichoice gets value 1

end
create vector, problem 2, sequence_number = 2
    function = 2                     # The initial condition for the temperature is
                                     # a function of x and y
                                     # This function is defined in subroutine
                                     # func, see the main program
                                     # The parameter ichoice gets value 2

end

# Define the essential boundary conditions for both vectors pressure
# and temperature
```

```

# See Users Manual Section 3.2.5

essential boundary conditions, sequence_number = 1
  func=1          # The boundary condition for the pressure is
                  # a function of t, x and y
                  # This function is defined in subroutine
                  # funcbc, see the main program
                  # The parameter ichoice gets value 1
end
essential boundary conditions, sequence_number = 2
  func=2          # The boundary condition for the temperature is
                  # a function of t, x and y
                  # This function is defined in subroutine
                  # funcbc, see the main program
                  # The parameter ichoice gets value 2
end

# Define the coefficients for the pressure problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 1
  elgrp1(nparm=20)          # The coefficients are defined by 20 parameters
  coef6 = kappa            # a11 = kappa
  coef9 = coef 6           # a22 = kappa
  coef15 = old solution Temperature
  coef16 = func=3          # beta = T
                           # The right-hand side is a
                           # a function of t and x defined by
                           #  $f = x + 3 t^2 x^2$ 
                           # This function is defined in subroutine
                           # funcfc, see the main program
                           # The parameter ichoice gets value 3
  coef17 = rho             # rho_cp = rho
end

# Define the coefficients for the temperature problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 2
  elgrp1(nparm=20)          # The coefficients are defined by 20 parameters
  coef6 = kappa            # a11 = kappa
  coef9 = coef 6           # a22 = kappa
  coef15 = old solution pressure
  coef16 = func=4          # beta = p
                           # The right-hand side is a
                           # a function of t and x defined by
                           #  $g = 3 x + 3 t^2 x^2$ 
                           # This function is defined in subroutine
                           # funcfc, see the main program
                           # The parameter ichoice gets value 4
  coef17 = rho             # rho_cp = rho
end

# Define the time integration process
# See Users Manual Section 3.2.15

```

```
# First with respect to the pressure problem

time_integration, sequence_number = 1
  method = euler_implicit           # Time discretization algorithm
  tinit = t0                        # initial time
  tend = t_end                      # end time
  timestep = dt                     # time step
  toutinit = t0                    # initial time for output
  toutend = t_end                   # end time for output
  toutstep = dt                    # time step for output
  seq_boundary_conditions = 1       # defines which essential boundary
                                   # conditions must be used
  seq_coefficients = 1             # defines the coefficients for
                                   # both equations
  diagonal_mass_matrix             # The mass matrix is lumped
  mass_matrix = constant           # and constant for both problems
end

# Next with respect to the temperature problem
# This part is used in the same time loop, so no method or time steps
# are given
# Only specific parts referring to the second equation

time_integration, sequence_number = 2
  seq_coefficients = 2
  seq_boundary_conditions = 2
  mass_matrix = constant
end

# Write only pressure and temperature to output file for postprocessing,
# not the exact solutions
# See Users Manual, Section 3.2.13

output
  write 2 solutions
end

# Define the structure of the problem
# In this part it is described how the problem must be solved
# This is necessary since some extra output is required
#

structure
  # Create initial condition for pressure and temperature
  create_vector, sequence_number=1, pressure
  create_vector, sequence_number=2, Temperature

  # Write the results in the first time step

  output

# Explicit time loop
# This offers the possibility to do more in each time step
start_time_loop
  time_integration, sequence_number = 1, Pressure
```

```
    print time
    time_integration, sequence_number = 2, Temperature
    output
end_time_loop

# Create the exact solution for pressure and temperature and t = tend
create_vector, sequence_number=1, p_exact
create_vector, sequence_number=2, T_exact

# Compute and print the error of both solutions at t = tend
p_error = norm_dif=3,vector1=pressure, vector2=p_exact
print p_error, text = 'difference in p at time = 1'
T_error = norm_dif=3,vector1=Temperature, vector2=t_exact
print T_error, text = 'difference in T at time = 1'

end

# timedcop_02.prb
#
# problem file for the solution of a coupled set of time-dependent equations
# In this case a time loop with a coupled problem in the structure block is used
# See Users Manual Section 6-4-3
#
# To run this file use:
#   sepcomp timedcop_02.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    kappa = 1          # coefficient in diffusion term
    rho   = 1          # density times heat capacity
    t0    = 0          # initial time
    t_end = 1          # end time
    dt    = 0.1       # time step
  vector_names
    Pressure
    Temperature
    P_exact
    T_exact
  variables
    p_error
    T_error
end
#
# Define the type of problems to be solved
#
problem 1          # See Users Manual Section 3.2.2
                  # This concerns the first problem with the
                  # pressure p as unknown
```

```

types                                # Define types of elements,
                                     # See Users Manual Section 3.2.2
    elgrp1=800                        # Type number for Laplacian equation
                                     # See Standard problems Section 3.1
    essbouncond                       # Define where essential boundary conditions are
                                     # given (not the value)
                                     # See Users Manual Section 3.2.2
    curves(c1 to c4)                 # There are essential boundary conditions on
                                     # all four boundaries
problem 2
                                     # This concerns the second problem with the
                                     # temperature T as unknown
types                                # Define types of elements,
                                     # See Users Manual Section 3.2.2
    elgrp1=800                        # Type number for Laplacian equation
                                     # See Standard problems Section 3.1
    essbouncond                       # Define where essential boundary conditions are
                                     # given (not the value)
                                     # See Users Manual Section 3.2.2
    curves(c1 to c4)                 # There are essential boundary conditions on
                                     # all four boundaries
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

matrix
    symmetric, problem = 1           # Symmetrical profile matrix
                                     # So a direct method will be applied
    symmetric, problem = 2           # The same for the second problem
end

# Create the start vector for the process, i.e. the solution at t=0
# See Users Manual Section 3.2.10

create vector, problem 1, sequence_number = 1
                                     # Mark that no sequence number is given after
                                     # create vector. The vector that is used is
                                     # defined in the structure block
    function = 1                     # The initial condition for the pressure is
                                     # a function of x and y
                                     # This function is defined in subroutine
                                     # func, see the main program
                                     # The parameter ichoice gets value 1
end
create vector, problem 2, sequence_number = 2
    function = 2                     # The initial condition for the temperature is
                                     # a function of x and y
                                     # This function is defined in subroutine
                                     # func, see the main program
                                     # The parameter ichoice gets value 2
end

# Define the essential boundary conditions for both vectors pressure
# and temperature

```



```

# See Users Manual Section 3.2.5
# Since we are using a coupled solution, no end between the two is allowed

essential boundary conditions 1
  func=1                # The boundary condition for the pressure is
                        # a function of t, x and y
                        # This function is defined in subroutine
                        # funcbc, see the main program
                        # The parameter ichoice gets value 1
essential boundary conditions 2
  func=2                # The boundary condition for the temperature is
                        # a function of t, x and y
                        # This function is defined in subroutine
                        # funcbc, see the main program
                        # The parameter ichoice gets value 2

end

# Define the coefficients for the pressure problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 1
  elgrp1(nparm=20)      # The coefficients are defined by 20 parameters
  coef6 = kappa         # a11 = kappa
  coef9 = coef 6        # a22 = kappa
  coef15 = old solution Temperature # beta = T
  coef16 = func=3       # The right-hand side is a
                        # a function of t and x defined by
                        #  $f = x + 3 t^2 x^2$ 
                        # This function is defined in subroutine
                        # funcfc, see the main program
                        # The parameter ichoice gets value 3
  coef17 = rho          # rho_cp = rho
end

# Define the coefficients for the temperature problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 2
  elgrp1(nparm=20)      # The coefficients are defined by 20 parameters
  coef6 = kappa         # a11 = kappa
  coef9 = coef 6        # a22 = kappa
  coef15 = old solution pressure # beta = p
  coef16 = func=4       # The right-hand side is a
                        # a function of t and x defined by
                        #  $g = 3 x + 3 t^2 x^2$ 
                        # This function is defined in subroutine
                        # funcfc, see the main program
                        # The parameter ichoice gets value 4
  coef17 = rho          # rho_cp = rho
end

# Define the time integration process
# See Users Manual Section 3.2.15

```

```
# First with respect to the pressure problem

time_integration
  method = euler_implicit           # Time discretization algorithm
  number_of_coupled_equations = 2   # p and T are treated as a
                                    # coupled system
  tinit = t0                        # initial time
  tend = t_end                      # end time
  timestep = dt                     # time step
  toutinit = t0                    # initial time for output
  toutend = t_end                  # end time for output
  toutstep = dt                    # time step for output
  diagonal_mass_matrix             # The mass matrix is lumped
  mass_matrix = constant           # and constant for both problems
  seq_boundary_conditions = 1      # defines which essential boundary
                                    # conditions must be used for the
                                    # two coupled problems

# Information for the pressure equation
equation 1
  local_options
  seq_coefficients = 1             # defines the coefficients for
                                    # both equations

# Information for the temperature equation
equation 2
  local_options
  seq_coefficients = 2             # defines the coefficients for
                                    # both equations

end

# Write only pressure and temperature to output file for postprocessing,
# not the exact solutions
# See Users Manual, Section 3.2.13

output
  write 2 solutions
end

# Define the structure of the problem
# In this part it is described how the problem must be solved
# This is necessary since some extra output is required
#

structure
  # Create initial condition for pressure and temperature
  create_vector, sequence_number=1, pressure
  create_vector, sequence_number=2, Temperature

# Write the results in the first time step

output

# Explicit time loop
```

```
# This offers the possibility to do more in each time step
start_time_loop
# Mark that a coupled system can only be solved if the vectors are
# stored consecutively in the solution vector
    time_integration, Pressure
    print time
    output, sequence_number=1
end_time_loop

# Create the exact solution for pressure and temperature and t = tend
create_vector, sequence_number=1, p_exact
create_vector, sequence_number=2, T_exact

# Compute and print the error of both solutions at t = tend
p_error = norm_dif=3,vector1=pressure, vector2=p_exact
print p_error, text = 'difference in p at time = 1'
T_error = norm_dif=3,vector1=Temperature, vector2=t_exact
print T_error, text = 'difference in T at time = 1'

end
```

Finally we demonstrate an option that for this problem has no practical use at all. It concerns the use of the options `no_computation`, `reuse_time_parameters` and `boundary_conditions = old_vector`. In this case first the time parameters are set and the time raised. Next the essential boundary conditions are filled in two separate vectors and after that the actual time integration is carried out using these two special vectors. The time is not raised in this step. To get this option in your local directory use:

```
sepgetex timedcop_03
```

The corresponding problem file is:

```
# timedcop_03.prb
#
# problem file for the solution of a coupled set of time-dependent equations
# In this case a time loop with a coupled problem in the structure block
# is used in combination with boundary conditions stored in a separate vector
# See Users Manual Section 6-4-3
#
# To run this file use:
#   sepcomp timedcop_03.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    kappa = 1          # coefficient in diffusion term
    rho   = 1          # density times heat capacity
    t0    = 0          # initial time
    t_end = 1          # end time
    dt    = 0.1        # time step
  vector_names
    Pressure          # Pressure
    Temperature       # Temperature
    P_exact           # Exact pressure
    T_exact           # Exact temperature
    Press_bc          # Special vector to store the boundary
                      # conditions for the pressure
    Temp_bc           # Special vector to store the boundary
                      # conditions for the temperature
  variables
    p_error
    T_error
end
#
# Define the type of problems to be solved
#
problem 1          # See Users Manual Section 3.2.2
                  # This concerns the first problem with the
                  # pressure p as unknown
  types            # Define types of elements,
```

```

        elgrp1=800          # See Users Manual Section 3.2.2
                          # Type number for Laplacian equation
        essbouncond       # See Standard problems Section 3.1
                          # Define where essential boundary conditions are
                          # given (not the value)
        curves(c1 to c4)  # See Users Manual Section 3.2.2
                          # There are essential boundary conditions on
                          # all four boundaries
problem 2
                          # This concerns the second problem with the
                          # temperature T as unknown
        types            # Define types of elements,
                          # See Users Manual Section 3.2.2
        elgrp1=800      # Type number for Laplacian equation
                          # See Standard problems Section 3.1
        essbouncond     # Define where essential boundary conditions are
                          # given (not the value)
                          # See Users Manual Section 3.2.2
        curves(c1 to c4) # There are essential boundary conditions on
                          # all four boundaries
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4

matrix
  symmetric, problem = 1  # Symmetrical profile matrix
                          # So a direct method will be applied
  symmetric, problem = 2  # The same for the second problem
end

# Create the start vector for the process, i.e. the solution at t=0
# See Users Manual Section 3.2.10

create vector, problem 1, sequence_number = 1
        function = 1      # Mark that no sequence number is given after
                          # create vector. The vector that is used is
                          # defined in the structure block
                          # The initial condition for the pressure is
                          # a function of x and y
                          # This function is defined in subroutine
                          # func, see the main program
                          # The parameter ichoice gets value 1
end
create vector, problem 2, sequence_number = 2
        function = 2      # The initial condition for the temperature is
                          # a function of x and y
                          # This function is defined in subroutine
                          # func, see the main program
                          # The parameter ichoice gets value 2
end

# Define the essential boundary conditions for both vectors pressure
# and temperature
# In this case the boundary conditions are filled in the vectors Press_bc and
```

```

# Temp_bc
#

create vector, problem 1, sequence_number = 3
  curves(c1 to c4), func=3      # The boundary condition for the pressure is
                                # a function of t, x and y
                                # The parameter icoice in func gets value 3
end
create vector, problem 2, sequence_number = 4
  curves(c1 to c4), func=4      # The boundary condition for the temperature is
                                # a function of t, x and y
                                # The parameter icoice in func gets value 4
end

# Define the coefficients for the pressure problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 1
  elgrp1(nparm=20)              # The coefficients are defined by 20 parameters
  coef6 = kappa                 # a11 = kappa
  coef9 = coef 6                # a22 = kappa
  coef15 = old solution Temperature # beta = T
  coef16 = func=3              # The right-hand side is a
                                # a function of t and x defined by
                                #  $f = x + 3 t^2 x^2$ 
                                # This function is defined in subroutine
                                # funcf, see the main program
                                # The parameter icoice gets value 3
  coef17 = rho                 # rho_cp = rho
end

# Define the coefficients for the temperature problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients, sequence_number = 2
  elgrp1(nparm=20)              # The coefficients are defined by 20 parameters
  coef6 = kappa                 # a11 = kappa
  coef9 = coef 6                # a22 = kappa
  coef15 = old solution pressure # beta = p
  coef16 = func=4              # The right-hand side is a
                                # a function of t and x defined by
                                #  $g = 3 x + 3 t^2 x^2$ 
                                # This function is defined in subroutine
                                # funcf, see the main program
                                # The parameter icoice gets value 4
  coef17 = rho                 # rho_cp = rho
end

# Define the time integration process
# In this case it is done in two separate input blocks
# See Users Manual Section 3.2.15

# First input block defines the initial time, time step and end time

```

```
# Both for computing and output
# It is also used to raise the actual time, but not to perform a time step

time_integration, sequence_number = 1
  tinit = t0                                # initial time
  tend = t_end                              # end time
  timestep = dt                             # time step
  toutinit = t0                             # initial time for output
  toutend = t_end                           # end time for output
  toutstep = dt                             # time step for output
  no_computation                            # In the first timestep no computation is carried
                                           # out, only the parameters are set and the time is
                                           # raised
end

# Second input block defines the process, and reuses the parameters defined
# in the first block
# It is assumed that the boundary conditions are stored in two separate
# vectors press_bc and its successor temp_bc
# Mark that this is only allowed if both vectors have numbered consecutively,
# press_bc with the lowest number

time_integration, sequence_number = 2
  method = euler_implicit                   # Time discretization algorithm
  number_of_coupled_equations = 2          # p and T are treated as a
                                           # coupled system

  reuse_time_parameters
  diagonal_mass_matrix                     # The mass matrix is lumped
  mass_matrix = constant                   # and constant for both problems
  seq_boundary_conditions = press_bc        # defines which essential boundary
                                           # conditions must be used for the
                                           # two coupled problems
                                           # These conditions are stored
  boundary_conditions = old_vector         # in the vectors press_bc and
                                           # temp_bc

# Information for the pressure equation
equation 1
  local_options
  seq_coefficients = 1                     # defines the coefficients for
                                           # both equations

# Information for the temperature equation
equation 2
  local_options
  seq_coefficients = 2                     # defines the coefficients for
                                           # both equations

end

# Write only pressure and temperature to output file for postprocessing,
# not the exact solutions, nor the boundary value vectors
# See Users Manual, Section 3.2.13

output
  write 2 solutions
```

```
end

# Define the structure of the problem
# In this part it is described how the problem must be solved
# This is necessary since some extra output is required
#

structure
  # Create initial condition for pressure and temperature
  create_vector, sequence_number=1, pressure
  create_vector, sequence_number=2, Temperature

  # Write the results in the first time step

  output

  # Explicit time loop
  # This offers the possibility to do more in each time step

  start_time_loop

  # Mark that a coupled system can only be solved if the vectors are
  # stored consecutively in the solution vector

  # First the time parameters are set and the time raised
  time_integration, sequence_number = 1, Pressure

  # Next the essential boundary conditions are stored in the arrays
  # press_bc and temp_bc
  # Since the time is raised, it concerns the new time level

  create_vector, sequence_number=3, press_bc
  create_vector, sequence_number=4, Temp_bc

  # The actual time integration is carries out
  time_integration, sequence_number = 2, Pressure

  # Some output
  print time
  output
  print Pressure
  print Temperature
end_time_loop

# Create the exact solution for pressure and temperature and t = tend
create_vector, sequence_number=1, p_exact
create_vector, sequence_number=2, T_exact

# Compute and print the error of both solutions at t = tend
p_error = norm_dif=3,vector1=pressure, vector2=p_exact
print p_error, text = 'difference in p at time = 1'
T_error = norm_dif=3,vector1=Temperature, vector2=t_exact
print T_error, text = 'difference in T at time = 1'

end
```


6.4.4 An example of a stationary equation solved by the limit of a time-dependent problem

In this section we consider exactly the same problem as in Section 6.2.1. The difference is that now we solve the heat equation and use the result for $t \rightarrow \infty$ as stationary solution. Hence a time-stepping algorithm is used to solve the Laplacian equation.

The parameter ρc_p is set equal to 1 and an Euler implicit method with time-step $\Delta t = 0.1$ is used for the time integration. The reason to use Euler implicit is that it has the best known damping properties in combination with a very simple scheme. To solve the system of linear equations in each time-step a conjugate gradient method is used in combination with a standard preconditioning.

The mesh input file is the same as for the example in Section 6.2.1:

```
*examu644.msh
* mesh for the unit square (0,1) x (0,1)

mesh2d
  points
    p1=(0,0)
    p2=(1,0)
    p3=(1,1)
    p4=(0,1)
  curves
    c1=line1(p1,p2,nelm=10)
    c2=line1(p2,p3,nelm=10)
    c3=line1(p3,p4,nelm=10)
    c4=line1(p4,p1,nelm=10)
  surfaces
    s1=quadrilateral5(c1,c2,c3,c4)
  plot
end
```

No special program is needed to solve these equations, it is sufficient to use program sepcomp. The following input file may be used for example:

```
*examu644.prb
* problem definition for simple Laplacian problem solved as a time-dependent
* problem
* type number 800

constants
  vector_names
    potential
end

problem
  types
    elgrp1 = 800
  essbouncond
    curves (c1,c4)
end
*
* Definition of matrix structure
# Symmetrical compact matrix
# So an iterative method will be applied
```

```
matrix
  storage_method = compact, symmetric
end
*
* Essential boundary conditions
*
essential boundary conditions
  curves (c3) value = 1      # At C3 T=1, at all other boundaries we
                              # have T=0, which does not require input
end
*
* Definition of coefficients for the Laplacian equation
*
coefficients
  elgrp1(nparm=20)
    coef6 = 1                # a11 = 1
    coef9 = coef 6           # a22 = 1
    coef17 = 1               # rho cp = 1
end
time_integration
  method = euler_implicit
  tinit = 0
  timestep = 0.1
  tend = 10
  abs_stationary_accuracy = 0.01
  mass_matrix = constant
  stiffness_matrix = constant
  right_hand_side = zero
  diagonal_mass_matrix
  seq_coefficients = 1
  boundary_conditions = constant
end
solve
  iteration_method = cg
end
```

In fact the block *solve* is not necessary at all, since due to the compact storage method *cg* is the default value.

Post processing may be performed in the standard way by program *seppost*. A sample input file is:

```
* examu644.pst
* input for seppost
*
postprocessing
  plot contour potential
end
```

6.4.5 An example of a time-dependent equation coupled with a stationary equation

In this section we consider an artificial time-dependent problem (the heat equation) that is solved using the option `time_integration`. Special in this example is that in each time-step we solve a stationary problem (elasticity equation) with an initial strain that depends on the just computed temperature. To get this example in your local directory use:

```
sepgetex stresstm
```

To run this example use the following commands:

```
sepmesh stresstm.msh
view the plots
seplink stresstm
stresstm < stresstm.prb
seppost stresstm.pst
view the plots
```

In order to solve this problem with program SEPCOMP, it is necessary to use the option `TIME_LOOP` in the input block `STRUCTURE`.

Consider the the cross-section of a long glass plate of length 2 m and height 0.02 m. Assuming symmetry it is sufficient to consider only one quarter of the plate. The domain Ω is sketched in Figure 6.4.5.1. On this region we assume that the temperature satisfies a heat equation with

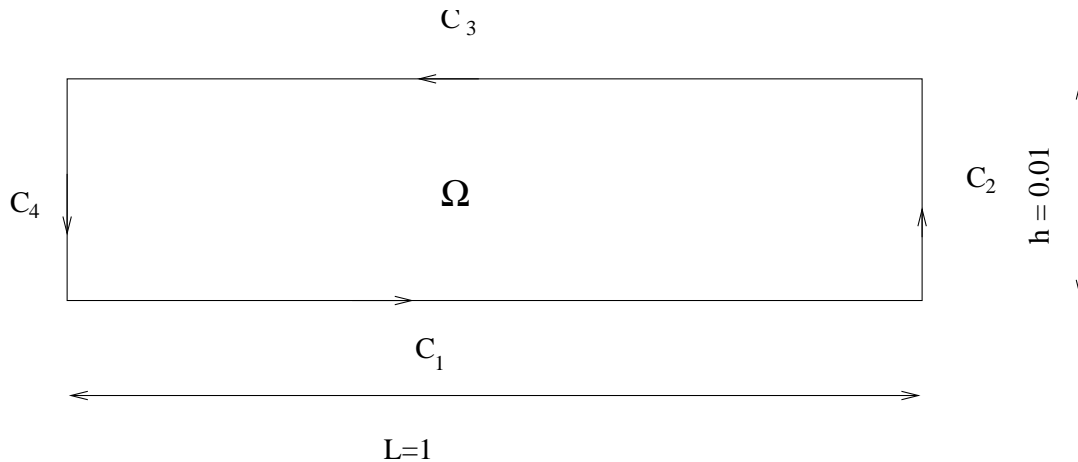


Figure 6.4.5.1: Definition of region for coupled heat equation and elasticity equation

$$\begin{aligned} \text{heat conductivity } \lambda &= 1 \\ \rho c_p &= 2.510^6 \end{aligned}$$

No convection nor any heat supply is assumed.

At the symmetry boundaries (C1 and C4) we have the natural boundary conditions $\frac{\partial T}{\partial n} = 0$.

At the two other boundaries we prescribe a time-dependent temperature $T = 700 - 20t$.

In the initial state the temperature is equal to $T_0 = 700$.

In each time step we want to compute the displacement due to initial strain. The plate is free at the boundaries, which implies that we have natural boundary conditions at the sides C2 and C3. At the symmetry boundaries C1 and C4 the normal displacement is equal to 0. The data for the elasticity equation are:

$$\begin{aligned} \text{Young's modulus } E &= 7.2 \cdot 10^{10} \\ \text{Poisson's ratio } \nu &= 0.2 \\ \text{Initial strain} &= \alpha(T - T_0) \end{aligned}$$

The mesh is refined in the neighborhood of the boundaries resulting in the following input file:

```

* stresstm.msh
*
* mesh for glass plate
* only one quarter is considered
*
constants
  integers
    n = 20
    n1 = 30
  reals
    length = 1      # plate has half length of 1 m
    width = 0.01   # plate has half width of 1 cm
end
mesh2d
  points
    p1=(0,0)
    p2=( length,0)
    p3=( length, width)
    p4=(0, width)
  curves
    c1=line 1(p1,p2, nelm= n1,ratio=3, factor=2) # refined in the direction
    c2=line 1(p2,p3, nelm= n,ratio=3, factor=2)  # of the outer boundary
    c3=line 1(p3,p4, nelm= n1,ratio=1, factor=2)
    c4=line 1(p4,p1, nelm= n,ratio=1, factor=2)
  surfaces
    s1=rectangle3(c1,c2,c3,c4)
  plot (jmark=5, numsub=1)
end

```

Since the boundary conditions for the velocity are time dependent, it is necessary to use a function subroutine FUNCBC. Hence one must provide the following main program stresstm.f:

```

program stresstm
  implicit none
  call sepcom(0)
  end

c *****
c
c   function for essential boundary conditions
c
c *****
c   function funcbc ( icoice, x, y, z )
c     implicit none
c     double precision funcbc, x, y, z
c     integer icoice
c     include 'SPcommon/ctimen'

c     if ( icoice==1 ) then
c       funcbc = 700-20*t
c     end if

c   end

```

Common block `ctimen` contains the parameter t , which gives the actual time.

For the solution of this problem we have to define 2 problems, one for the temperature and one for the displacement. In our example we are using four vectors:

$$\begin{aligned} V1 &= T \\ V2 &= \mathbf{u} \\ V3 &= \alpha(T - T_0) \\ V4 &= \sigma \text{ (the stress)} \end{aligned}$$

The third vector is used to define the initial strain, the last vector for output purposes. In the program we need the option `TIME_LOOP`, since during each time step we have to solve a linear problem for the displacement. As a consequence a `STRUCTURE` block is necessary. The structure of the program is as follows:

```

compute the start vector for the temperature $T$ (initial condition)
fill the boundary conditions for the displacement $\bf u$
For all time steps do
  Perform one time step to compute T
  Store alpha(T-T0)
  Compute the displacement
  Compute the stress tensor
  Write information to output file if necessary
end loop

```

The following input file may be used:

```

! stresstm.prb
!
! problem definition for time-dependent heat equation
! linear triangles type number 800
! In each time step the displacement due to initial strain is computed
!
! 1: temperature T
! 2: displacement u
! 3: alpha(T-T0)
! 4: stress tensor
!
constants
  reals
    T0 = 700
    alpha = 1d-5
    alphT0 = 7d-3
  vector_names
    temperature # T
    displacement # u
    alpha_T # alpha(T-T0)
    stress # stress tensor
end
problem 1
  types
    elgrp1 = 800 # standard for heat equation
  essbouncond
    curves(c2) # temperature is given at
    curves(c3) # sides C2 and C3
problem 2
  types
    elgrp1 = 250 # Elasticity equation
  essbouncond

```

```

        degfd1, curves(c4)      # normal displacement at sides
        degfd2, curves(c1)      # C1 and C4 is 0
end
!
! Definition of matrix structure
!
matrix,
    symmetric, problem = 1      # heat equation
    symmetric, problem = 2      # elasticity equation
end
structure
    create_vector, temperature
    prescribe_boundary_conditions, sequence_number=2, displacement
    scalar 1 = alpha
    start_time_loop
        time_integration, sequence_number = 1, temperature      # Compute T
        alpha_T = alpha      temperature
        alpha_T = subtract constant alphT0      # alpha_T = alpha(T-T0)
        solve_linear_system, seq_coef=2, problem=2, displacement # Compute u
        derivatives, seq_coef=2, seq_deriv=1, stress      # Compute stress
        output      # write results
    end_time_loop
end
!
! Define initial conditions for temperature
!
create vector
    value=700
end
!
! Essential boundary conditions
!
essential boundary conditions
    curves (c2,c3),(func=1)      # T at boundary is function of time
end
essential boundary conditions, sequence_number=2, problem=2
    # Displacement at symmetry is zero
end
!
! Definition of coefficients for the heat equation (t=0 only)
!
coefficients
    elgrp1(nparm=20)
        coef6 = 1      # a_11 = 1 (lambda)
        coef9 = coef 6      # a_22 = 1 (lambda)
        coef17 = 2.5d6      # rho_cp = 2.5d6
end
!
! Coefficients for the elasticity equation
!
coefficients, sequence_number=2
    elgrp1(nparm=45)
        coef6 = 7.2d10      # E = 7.2d10
        coef7 = 0.2      # nu = 0.3
        coef31 = old_solution alpha_T      # alpha(T0-T)

```

```

        coef32 = coef31                # alpha(T0-T)
    end
    !
    ! Derivatives block, to compute the stress tensor
    !
    derivatives
        icheld = 6
        seq_input_vector = displacement
    end
    !
    ! Definition of the time integration for the temperature
    !
    time_integration
        method = euler_implicit        # euler implicit method
        tinit = 0                      # T_0 = 0
        tend = 10                     # Tend = 10
        tstep = 1                      # dt = 1
        toutinit = 0                  # Each time step is written
        toutend = 10
        toutstep = 1
        seq_boundary_conditions = 1    # bc's for temperature
        seq_coefficients = 1          # coefficients for temperature
        diagonal_mass_matrix          # lumped mass matrix
        stiffness_matrix = constant    # stiffness matrix is time independent
        mass_matrix = constant        # mass matrix is time independent
        right_hand_side = zero        # no source term
    end
end

```

Post processing may be performed in the standard way by program seppost. The y-coordinate is multiplied by a factor of 30 in order to get a better view.

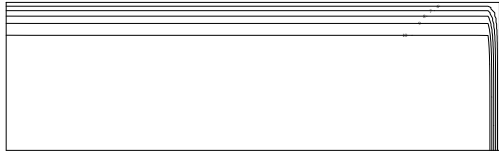
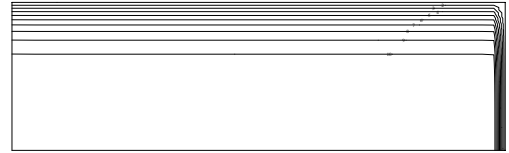
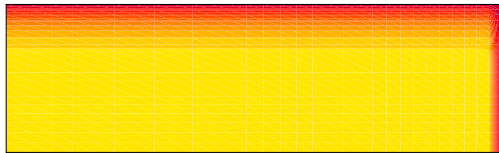
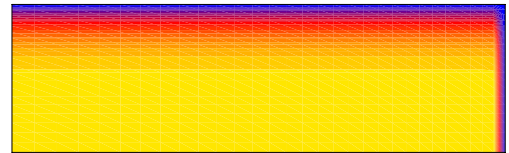
A sample input file is:

```

* stresstm.pst
*
*
* input for seppost
*
postprocessing
    time = (0,10)
    plot contour temperature, minlevel = 500, maxlevel = 700//
        yfact=30                                # temperature T
    plot coloured contour temperature, minlevel = 500 //
        maxlevel = 700, yfact=30
    plot vector displacement, factor=250 //
        yfact=30                                # displacement u
    plot contour stress, degfd1=1, yfact=30    # sigma_xx
    plot coloured contour stress, degfd1=1, yfact=30
    time history plot point(0.5,0.05) temperature # T in point(0.5,0.05)
end

```

Figures 6.4.5.2 and 6.4.5.3 show the isotherms at $t = 6$ and $t = 10$ respectively. Figures 6.4.5.4 and 6.4.5.5 show the colored temperature levels at $t = 6$ and $t = 10$ respectively. Figures 6.4.5.6 and 6.4.5.7 show a vector plot of the displacements at $t = 6$ and $t = 10$ respectively. Figures 6.4.5.8 and 6.4.5.9 show a contour plot of the xx-component of the stress tensor (σ_{xx}) at $t = 6$ and $t = 10$ respectively. Figures 6.4.5.10 and 6.4.5.11 show colored contour levels of the xx-component of the

Figure 6.4.5.2: Isotherms at $t=6$ Figure 6.4.5.3: Isotherms at $t=10$ Figure 6.4.5.4: Temperature levels at $t=6$ Figure 6.4.5.5: Temperature levels at $t=10$

stress tensor (σ_{xx} at $t = 6$ and $t = 10$ respectively. Finally in Figure 6.4.5.12 the temperature history in point $(0.5, 0.05)$ is shown.

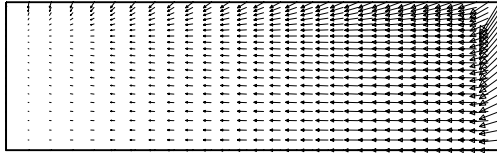


Figure 6.4.5.6: Vector plot of displacement at $t=6$

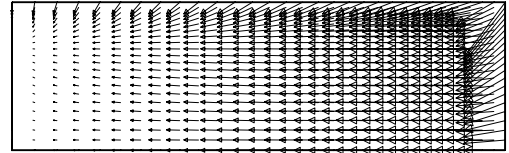


Figure 6.4.5.7: Vector plot of displacement at $t=10$

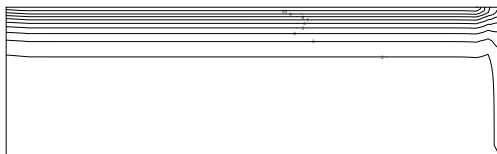


Figure 6.4.5.8: Contour plot of σ_{xx} at $t=6$

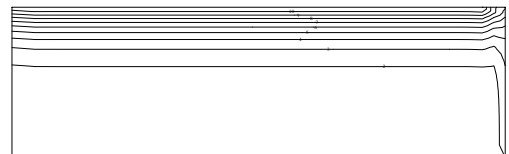


Figure 6.4.5.9: Contour plot of σ_{xx} at $t=10$

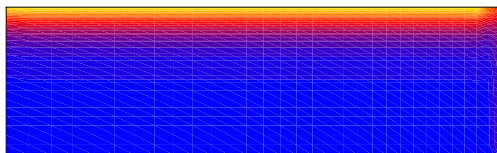


Figure 6.4.5.10: Contour plot of σ_{xx} at $t=6$

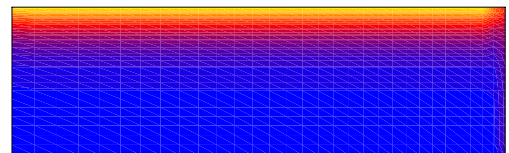


Figure 6.4.5.11: Contour plot of σ_{xx} at $t=10$

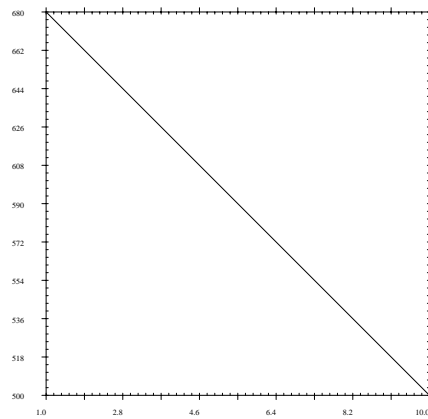


Figure 6.4.5.12: Temperature history in point (0.5,0.05)

6.6 Examples of instationary free boundary problems

In this section we treat some examples of time-dependent free boundary problems. It will be shown how such problems may be solved by means of program sepfree.

The following examples will be treated:

6.6.1 The solution of a Stefan problem.

This example shows how a typical Stefan problem may be solved. It concerns in this case the homogenization of an aluminum alloy.

6.6.2 The dissolution of a disk-like particle in a disk-shape environment.

This example shows how a more complex Stefan problem may be solved. It concerns in this case the homogenization of an aluminum alloy, consisting of a Al_2Cu disk-like particle dissolving in disk-like environment. It is demonstrated why the boundary must be adapted by the option `adaptation_method = stefan`.

6.6.3 The dissolution of a two particles.

This example is an extension of the previous one. In this case two circular particles of unequal size are dissolved.

6.6.1 An example of a simple Stefan problem

In this section we consider a free boundary problem of Stefan type. Typical technical applications, which are described by this problem are: melting or freezing of ice, etching of semi-conductor devices and homogenization of aluminum alloys.

In this particular example, we model the homogenization of an aluminum alloy.

We consider a rectangular region in which a zinc part at the left-hand side is in contact with aluminum at the right-hand side. See Figure 6.6.1.1 for the configuration.

At a certain temperature the zinc dissolves in the aluminum. The problem to be solved is the

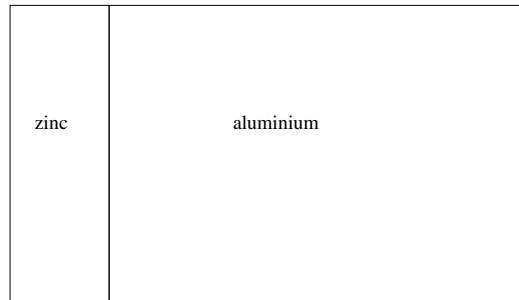


Figure 6.6.1.1: Zinc in contact with aluminum

concentration (c) of zinc in aluminum and the position of the interface between zinc and aluminum. This interface (S) moves slowly to the left as time increases.

All computations take place in the aluminum and none in the zinc.

If we make the equations dimensionless we may formulate the problem in the following way:

$$\frac{\partial c}{\partial t} - \operatorname{div}(k\nabla c) = 0 \quad (x, y) \in (\Omega), \quad t \in (0, T], \quad (6.6.1.1)$$

The region Ω is defined by the rectangle $\Omega = (S(y, t), 5) \times (0, 1)$, whereas $S(y, t)$ is the free surface, which has to be computed.

The initial conditions are:

$$S(y, 0) = 1, \quad y \in [0, 1], \quad (6.6.1.2)$$

$$c(x, y, 0) = 0, \quad (x, y) \in [1, 5] \times [0, 1], \quad (6.6.1.3)$$

and the boundary conditions

$$\frac{\partial c}{\partial n}(x, 0, t) = 0, \quad x \in [S(0, t), 5], \quad (6.6.1.4)$$

$$\frac{\partial c}{\partial n}(x, 1, t) = 0, \quad x \in [S(1, t), 5], \quad (6.6.1.5)$$

$$\frac{\partial c}{\partial n}(5, y, t) = 0, \quad y \in [S(0, t), 5]. \quad (6.6.1.6)$$

Finally at the free boundary we need 2 boundary conditions, one for the solution of the differential equation and one for the displacement of the free boundary.

In this example we use:

$$c(S(t), y, t) = 1, \quad \lambda \frac{\partial c}{\partial n}(S(t), y, t) = \mathbf{v} \cdot \mathbf{n} = v_n \quad (6.6.1.7)$$

In Equation 6.6.1.7, v_n denotes the velocity in which the free surface moves along the normal to the free surface. Note that this example is in fact one-dimensional.

This free boundary problem is solved by the method of Murray and Landis, (1959).

In the first time-step the diffusion equation together with the initial and boundary conditions is

solved. On the free boundary S only the Dirichlet condition is used.

In the next time steps first the boundary is moved using the boundary condition 6.6.1.7. This means that the co-ordinates of the free surface at time $t + \Delta t$ are given by:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(\mathbf{t}) + \Delta t v_n \mathbf{n} = \mathbf{x}(\mathbf{t}) + \lambda \frac{\partial c}{\partial n} \Delta t \mathbf{n} \quad (6.6.1.8)$$

Once the boundary is moved, the concentration c can be computed in the new region using equation 6.6.1.1. However, the computation of the concentration implies that we have to compute $\frac{c(t+\Delta t)-c(t)}{\Delta t}$. We do not know $c(t)$ in the nodal points, since due to the displacement of the boundary also all nodes have been moved. So either we have to interpolate the concentration to the new nodes, or we have to make a correction for the displacement. Interpolation is of course possible, but relatively expensive. The correction is much more simple. All we have to do is to subtract a convection term, where the velocity is equal to the so-called mesh velocity defined by $\frac{\mathbf{x}(t+\Delta t)-\mathbf{x}(t)}{\Delta t}$. Mark that due to the subtraction, the negative mesh velocity is used in the convective terms. This is just the other sign as in standard convection due to a material derivative. So in each step we have to solve:

$$\frac{\partial c}{\partial t} - \text{div}(k \nabla c) - \mathbf{u}_{\text{mesh}} \cdot \nabla c = 0 \quad (6.6.1.9)$$

The first step in the solution process is the creation of the initial mesh. This may be done by program SEPMESH, or immediately by program SEPFREE. In this example we have chosen for a start with SEPMESH.

The corresponding input file is given by:

```
*stefan.msh
constants
  integers
    nelm1 = 50                # number of grid points in x-direction
    nelm2 = 4                 # number of grid points in y-direction
end
mesh2d
  points
    p1 = (1,0)
    p2 = (5,0)
    p3 = (5,1)
    p4 = (1,1)
  curves
    c1 = line1(p1,p2,nelm= nelm1)
    c2 = line1(p2,p3,nelm= nelm2)
    c3 = line1(p3,p4,nelm= nelm1)
    c4 = line1(p4,p1,nelm= nelm2) # free surface
  surfaces
    s1 = quad3(c1,c2,c3,c4)
  plot
end
```

After mesh generation, program SEPFREE is used to perform the time integration, adapt the boundary and the mesh and produce some intermediate prints and plots.

The convection diffusion equation is solved with the standard elements described in Section 3.1 of the Manual Standard Problems. The corresponding type number is 800. The natural boundary conditions at the boundaries $c1$ to $c3$ are all homogeneous and do not require extra input. The boundary condition $c = 1$ along the free surface is an essential boundary condition.

The structure of the main program is defined in the input block STRUCTURE.

Three different vectors are used:

concentration contains the concentration in the zinc.

u_mesh contains the mesh velocity.

grad_c contains the gradient of the concentration.

First the initial vector for the concentration is created ($c = 0$) and stored in **concentration**.

Next the gradient of the concentration is computed and stored in **u_mesh**. This creates a vector with 2 unknowns per point. Since the concentration is constant, the initial velocity vector is equal to **0**. In this way we can start with the solution of the convection diffusion equation with initial velocity **0**.

After the velocity is computed, the essential boundary condition ($c = 1$) can be applied.

After these preparations, which may include printing of initial conditions, the time-dependent free surface problem may be solved.

We want to produce one picture, with the boundaries of the region at all output steps. This makes it possible to show the progress of the free boundary. In order to put all plots in one picture it is necessary to surround the computation with a **open_plot** and **close_plot** statement.

In order to plot the initial boundary we give the **plot_boundary** command before the time integration.

The instationary free boundary loop consists of the following steps:

1. The gradient of the concentration is computed and stored in **grad_c**
2. Some vectors are printed at the times where output is required
3. The boundary of the region is adapted using the option **normal_velocity** and multiplication by Δt . This is in fact exactly the method given by Equation 6.6.1.8. The velocity to compute the normal velocity is defined by vector **grad_c** multiplied by λ . In this example we use $\lambda = 0.0101$.
If output is required the boundary of the mesh is plotted.
4. Once the boundary is adapted the mesh is adapted without changing the topology and the mesh velocity is computed and stored in vector **u_mesh**.
5. Finally the concentration is computed by solving the convection diffusion equation with the negative mesh velocity as convection velocity.

The time integration applied is an implicit Euler method with time-step 1, for t in the range (0,10). At times 2, 4, 6, 8 and 10 output is required. This is indicated by the parameters **toutinit**, **toutstep** and **toutend**.

A diagonal (lumped) mass matrix is used, and the right-hand side vector is of course zero, because no source term is present.

The input file for program **sepfree** is given by:

```
*stefan.prb
*
constants
  reals
    k = 1                # diffusion coefficient
    lambda = 0.0101     # velocity multiplication factor
  vector_names
    concentration       # concentration of zinc
    u_mesh              # mesh velocity
    grad_c              # gradient of concentration
end
problem
  types
```

```

    elgrp 1 = 800                # convection diffusion equation
    essboundcond
    curves(c4)                  # essential boundary conditions at free
                                # surface
                                # at the other boundaries we use natural
                                # boundary conditions which do not
                                # need any input
end

structure
# initial condition, See input block create
create_vector, concentration

derivatives, u_mesh            # mesh velocity = 0
                                # See input block
                                # derivatives

# essential boundary conditions. See input block essential boundary cond
prescribe_boundary_conditions, concentration

print concentration
print u_mesh, text='mesh velocity'
open_plot                      # all pictures in one plot
    plot_boundary              # plot initial boundary
    start_instationary_free_boundary_loop
                                # See input block
                                # instationary free boun
                                # actual time integration
                                # See input block
                                # time_integration
                                # gradient of c
                                # See input block
                                # derivatives

    print concentration, text='concentration'
    print u_mesh, text='mesh velocity'
    print grad_c, text='gradient concentration'
    end_instationary_free_boundary_loop
close_plot                      # end plotting in one
                                # picture
end

create vector
    value = 0                  # initial concentration
end

instationary_free_boundary
    adapt_mesh = 1             # mesh adaptation,
                                # see input block adapt_mesh

    seq_vector = grad_c        # vector to be used for adaptation of
                                # boundary (gradient of concentration)

    mesh_velocity = u_mesh      # mesh velocity is computed and stored in V2
end

time_integration
    method = euler_implicit    # time integration method
    tinit = 0                  # initial time

```

```

tend = 10                # final time
tstep = 1                # time-step
toutinit = 0            # initial time for output
toutend = 10            # final time for output
toutstep = 2            # time-step for output
seq_solution_method = 1 # see input block solve
right_hand_side = zero  # no source term
seq_coefficients = 1    # see input block coefficients
seq_boundary_conditions = 1 # see input block essential boundary cond
diagonal_mass_matrix    # lumped mass matrix
end

solve                    # defines the linear solver to be used
  direct_solver = profile
end

essential boundary conditions, sequence_number = 1 # defines the values of the
  # essential boundary conditions
  curves(c4), value = 1 # value of the Dirichlet boundary
  # condition at the free surface
end

coefficients            # defines the coefficients for the
  # convection-diffusion equation
  # See manual standard problems 3.1

elgrp1, nparam=20
  coef 6 = k            # div k grad
  coef 9 = coef 6       # div k grad
  coef 12 old_solution u_mesh//
    degree_of_freedom 1, coef = -1 # u-velocity adaptation of the mesh
    # is equal to -u component of
    # mesh velocity (V2)
  coef 13 old_solution u_mesh//
    degree_of_freedom 2, coef = -1 # v-velocity adaptation of the mesh
    # is equal to -v component of
    # mesh velocity (V2)
  coef 17 = 1           # density rho
end

adapt_mesh              # adaptation of the mesh
  adapt_boundary = 1    # see input block adapt_boundary
  change_topology = not # topology is kept
end

adapt_boundary          # defines how the boundary is
  # adapted
  curves = c4           # free surface
  plot_boundary         # The boundary is plotted in
  # each output step
  adaptation_method = normal_velocity # the normal velocity is adapted by
  factor = lambda, multiply = dt     # lambda times dt times the
  # normal derivative of the
  # the concentration
end

```



```
derivatives                                # defines which derivative to be
                                           # computed
    icheld = 2, seq_input_vector = concentration # gradient of first input
                                           # vector (c)
                                           # See manual standard problems 3.1
end

end_of_sepran_input
```

6.6.2 The dissolution of a disk-like particle in a disk-shape environment

In this section we consider the same type of problem as in Section 6.6.1. The type of material in this case is different, since it concerns the dissolution of an Al_2Cu particle in an $Al - Cu$ alloy. The mathematical model applied is exactly the same as the one used in Section 6.6.1. The constants used are of course different and moreover, the shape of the particle is different. In this case it concerns a rectangular particle and it is this shape that causes the difficulties. Figure 6.6.2.1 shows the configuration.

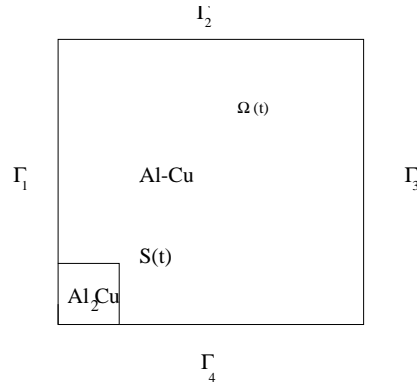


Figure 6.6.2.1: Dissolution of a Al_2Cu particle in an $Al - Cu$ alloy

The first step in the solution process is the creation of the initial mesh. This may be done by program SEPMESH, or immediately by program SEPFREE. In this example we have chosen for a start with SEPMESH.

The corresponding input file is given by:

```
*stefan1.msh
constants
  reals
    L_cell = 4          # Length of the cell
    L_edge = 1         # Initial length of an edge of the particle
end
mesh2d
  coarse(unit=1)
  points
    p1 = ( L_edge,0,0.1)
    p2 = ( L_cell,0,0.1)
    p3 = ( L_cell, L_cell,0.1)
    p4 = ( 0, L_cell,0.1)
    p5 = ( 0, L_edge,0.1)
    p6 = ( L_edge, L_edge,0.1)
    p7 = ( 0,0,0.1)
  curves
    c1 = cline1(p1,p2)
    c2 = cline1(p2,p3)
    c3 = cline1(p3,p4)
    c4 = cline1(p4,p5)
    c5 = cline1(p5,p6)
    c6 = cline1(p6,p1)
    c7 = curves(c2,c3)
    c8 = curves(c5,c6)
```

```

    c9 = cline1(p5,p7)
    c10 = cline1(p1,p7)
    c11 = cline1(p6,p3)
  surfaces
    s1 = general3(c1,c2,-c11,c6)
    s2 = general3(c3,c4,c5,c11)
  plot
end

```

If we apply the method used in Section 6.6.1 the free boundary shows a non-physical behavior as shown in Figure 6.6.2.2 For that reason the improved method indicated by `adaptation_method = stefan`

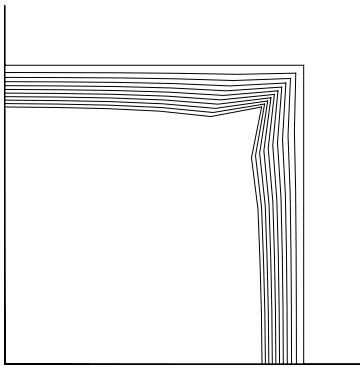


Figure 6.6.2.2: Position of free boundary at first 10 time-steps using the normal_velocity method

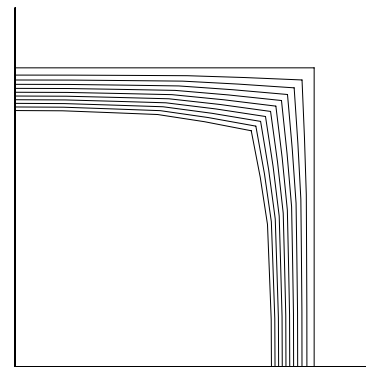


Figure 6.6.2.3: Position of free boundary at first 10 time-steps using the stefan method

is applied.

The input file for program sepfree is given by:

```

*stefan1.prb
constants
  reals
    k = 0.04858
    lambda = 0.000969
  vector_names
    concentration          # concentration of zinc
    u_mesh                 # mesh velocity
    grad_c                 # gradient of concentration
end
start
  norotate
end
problem

```

```
types
  elgrp 1 = 800          # convection diffusion equation
  essboundcond
  curves(c8)           # essential boundary conditions at free
                        # surface
end

structure
  create_vector, concentration
  derivatives, u_mesh
  prescribe_boundary_conditions, concentration
  open_plot             # all pictures in one plot
  plot_boundary, region = (0,1.1,0,1.1) # plot initial boundary
  start_instationary_free_boundary_loop
  time_integration
  derivatives, grad_c
  end_instationary_free_boundary_loop
  close_plot
end

matrix
  storage_scheme = compact # compact matrix
end

create vector
  value = 0.0011
end

instationary_free_boundary
  adapt_mesh = 1
  seq_vector = grad_c
  mesh_velocity = u_mesh
  interpolate_solution(concentration)
  check_boundary = 1
  check_mesh = 1
end

time_integration
  method = euler_implicit
  tinit = 0
  tend = 80
  tstep = 0.5
  toutinit = 0          # initial time for output
  toutend = 500        # final time for output
  toutstep = 10        # time-step for output
  seq_solution_method = 1
  right_hand_side = zero
  seq_coefficients = 1
  seq_boundary_conditions = 1
  diagonal_mass_matrix
end

solve
  iteration_method=cg, start=old_solution, print_level=1
end
```

```
essential boundary conditions
  curves(c8), value = 3.88
end

coefficients
  elgrp1, nparm=20
    coef 6 = k                                # div k grad
    coef 9 = coef 6                            # div k grad
    coef 12 old_solution u_mesh, degree_of_freedom 1, coef = -1 # u-velocity
    coef 13 old_solution u_mesh, degree_of_freedom 2 , coef = -1 # v-velocity
    coef 17 = 1                                # density rho
end

adapt_mesh
  adapt_boundary = 1, change_topology = not,# plot_mesh
end

adapt_boundary
  curves = c8
  plot_boundary
  adaptation_method = stefan
  factor = lambda, multiply = dt
  redistribute_nodes
  move_begin = c9
  move_end = c10
end

derivatives
  icheld = 12, seq_input_vector = concentration
end

end_of_sepran_input
```

Mark that in this case different parameters, boundary conditions and initial concentration are used:

```
initial concentration = 0.0011
concentration at interface = 3.88
diffusion coefficient  $k = 0.04858$ 
Stefan constant  $\lambda = 0.000969$ 
```

Furthermore the begin and end points are forced to move along given curves. Figure [6.6.2.3](#) shows the free boundary for several time-steps.

6.6.3 The dissolution of two particles

In fact this example is identical to the one in Section 6.6.2, except that we consider the case that two particles are dissolved.

To get this example into your local directory use:

```
sepgetex 2partsys
```

and to run it use:

```
sepmesh 2partsys.msh
sepfree 2partsys.prb
seppost 2partsys.pst
```

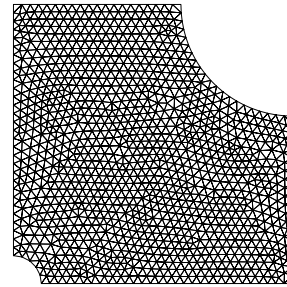
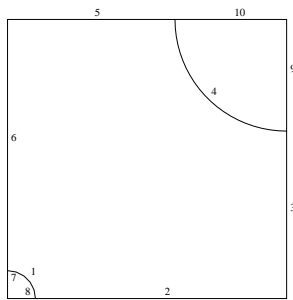


Figure 6.6.3.1: Boundaries of the mesh

Figure 6.6.3.2: Mesh for two particles

The mesh input file in this case is given by:

```
constants
  reals
    R_part_1 = 0.5          # Radius particle 1
    C_part_2 = 5           # Centroid particle 2
    S_part_2 = 3           # End point particle 2
  end
mesh2d
  coarse (unit = 0.5)
  points
    p1 = (0,0,0.3)
    p2 = ( R_part_1,0,0.3)
    p3 = ( C_part_2,0,0.3)
    p4 = ( C_part_2, S_part_2,0.3)
    p5 = ( C_part_2, C_part_2,0.3)
    p6 = ( S_part_2, C_part_2,0.3)
    p7 = (0, C_part_2,0.3)
    p8 = (0, R_part_1,0.3)
  curves
    c1 = carc1(p8,p2,-p1)
    c2 = cline1(p2,p3)
    c3 = cline1(p3,p4)
```



```
create vector
  value = 0.0011
end

instationary_free_boundary
  adapt_mesh = 1
  seq_vector = grad_c
  mesh_velocity = u_mesh
  interpolate_solution(v1)
  check_boundary = 1
  check_mesh = 1
  write_mesh
end

time_integration,sequence_number = 1
  method = euler_implicit
  tinit = 0
  tend = 50
  timestep = 0.5
  toutinit = 0           # initial time for output
  toutend = 400         # final time for output
  toutstep = 10         # time-step for output
  seq_solution_method = 1
  right_hand_side = zero
  seq_coefficients = 1
  seq_boundary_conditions = 1
  diagonal_mass_matrix
end

solve
  iteration_method=cg, start=old_solution, print_level=0
end

essential boundary conditions
  curves(c1), value = 3.88
  curves(c4), value = 3.88
end

coefficients
  elgrp1, nparm=20
  coef 6 = $k           # div k grad
  coef 9 = coef 6       # div k grad
  coef 12 old_solution u_mesh, degree_of_freedom 1, coef = -1 # u-velocity
  coef 13 old_solution u_mesh, degree_of_freedom 2 , coef = -1 # v-velocity
  coef 17 = 1           # density rho
end

adapt_mesh
  adapt_boundary = (1,2), change_topology = not # plot_mesh to be inserted for mesh plots
end

adapt_boundary, sequence_number = 1
  curves = (c1)
  adaptation_method = stefan
```



```

    factor = 0.000969, multiply = dt
    redistribute_nodes
    move_begin = c7
    move_end = c8
end

adapt_boundary, sequence_number = 2
    curves = c4
    adaptation_method = stefan
    factor = 0.000969, multiply = dt
    redistribute_nodes
    move_begin = c9
    move_end = c10
end

derivatives
    icheld = 12, seq_input_vector = concentration
end

end_of_sepran_input

```

In this case we need to describe the movement of both free boundaries. At the end of the process the final mesh is written to the file `meshoutput` so that the solution can be plotted by `seppost`. Figure 6.6.3.3 shows the free boundary during various time steps.

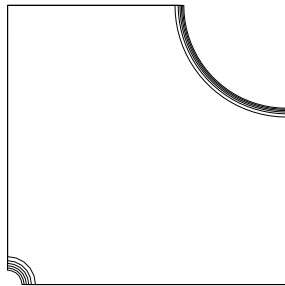


Figure 6.6.3.3: Position of free boundary during a number of time-steps

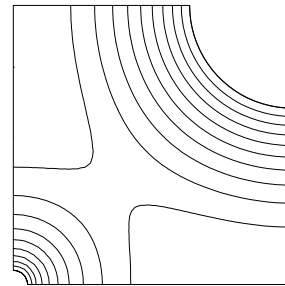


Figure 6.6.3.4: Concentration profiles for two-particle problem at $t=50$

A typical input file for `seppost` is given by

```

* 2partsys.pst
postprocessing
    plot mesh
    3d plot concentration
    plot contour concentration, noaxis, noplot_legenda, noplot_scales, nonumber
    plot coloured contour concentration
end

```

Figure 6.6.3.4 shows the computed concentration. The options nonumber and so on are used to suppress all information.

6.7 Auxiliary examples

In this section we treat some special examples not directly related to a special category. The following examples will be treated:

- 6.7.1** An example of reading own data for postprocessing purposes. This shows how one can read own data defined on some grid. A mesh is created automatically and the data may be processed by seppost.

6.7.1 An example of reading own data for postprocessings purposes

In this section we consider the use of SEPRAN for postprocessings purposes only.

Suppose the user has a field of co-ordinates and data and he wants to use SEPRAN or AVS (or both) for postprocessings purposes. Such data may be for example the result of measurements.

Unless the co-ordinates are positioned in a rectangular grid, AVS does not have the possibility to read and visualize the data, since AVS always expects an underlying structure.

With SEPRAN it is possible to read the co-ordinates and create an unstructured mesh using the option `NODAL_POINTS` as described in Section 2.7. The corresponding data may then be read by the option `INPUT_VECTOR` in the input block `STRUCTURE` (Section 3.2.3).

Since in this case SEPRAN is not used to solve a problem, the default problem description is used. With the option `OUTPUT` the result may be written to an AVS input file as well to the file `sepcomp.out`. Hence also program `SEPPOST` may be used to visualize the data.

In this example we show this process for some arbitrary data using an unstructured 2d grid. The boundary is defined by the user.

First we consider the input for program `sepmesh`:

```
*owndat2d.msh

# Example of creation of mesh in a set of points

mesh2d

#

nodal_points, unstructured, element_shape = 3

#

boundary_points

0.    , 0.
0.5   , 0.
1.    , 0.
1.    , 0.5
1.    , 1.
0.5   , 1.
0.    , 1.
0.    , 0.5

#

internal_points

0.25  , 0.25
0.50  , 0.50
0.75  , 0.25
0.25  , 0.75
0.75  , 0.75

#

plot
```

```
#
end
```

Figure 6.7.1.1 shows the mesh created by SEPRAN. Once the mesh has been created the data may

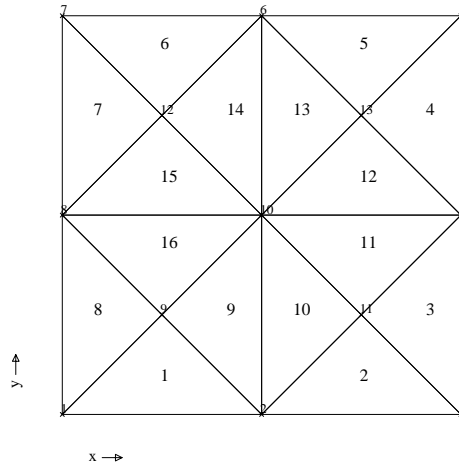


Figure 6.7.1.1: Mesh created by SEPRAN

be read by program sepcomp. The following input may be used for this purpose.

```
*owndat2d.prb
constants
  vector_names
    potential
end
problem default
structure
  input_vector potential, scalar, file = 'owndat2d.fil'
  print potential
  output
end
output
  to_avs
end
```

The data to be read are stored in the file owndat2d.fil. In this artificial case this file contains the x-coordinates as data. The file owndat2d.fil may have the following shape.

```
0.
0.5
1.
1.
1.
1.
0.5
0.
0.
```

0.25
0.50
0.75
0.25
0.75

SEPCOMP makes an AVS input file as well as the files necessary for program SEPPOST.
An example input file for SEPPOST is the following one:

```
*owndat2d.pst
postprocessing

  print potential
  plot contour potential
  plot coloured contour potential
  3d plot potential, angle = 135
end
```

Figure 6.7.1.2 shows the contour lines of the data and Figure 6.7.1.3 a 3D plot.

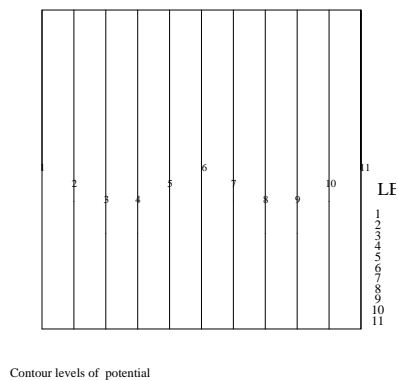
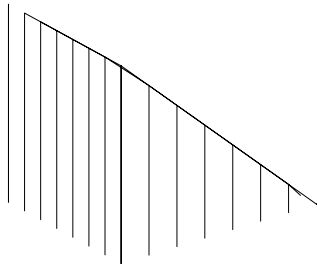


Figure 6.7.1.2: Contour plot of artificial data



3D plot of potential

Figure 6.7.1.3: 3D plot of artificial data

6.8 Examples of eigenvalue computations

In this section we treat examples of the computation of eigenvalues and eigenvectors. The following examples will be treated:

6.8.1 Eigenvalues and eigenvectors for a potential problem in an L-shape

6.8.1 Eigenvalues and eigenvectors for a potential problem in an L-shape

In this example we compute the 3 smallest eigenvalues and corresponding eigenvectors of the Laplace equation on a L-shaped region.

In order to get this example into your local directory use:

```
sepgetex eigenvalue
```

To run the example use the following commands:

```
sepmesh eigenvalue.msh
sepview sepplot.001
sepcomp eigenvalue.prb
seppost eigenvalue.pst
sepview sepplot.001
```

Consider the eigenvalue problem associated to the Laplace operator:

$$-\Delta u = \lambda u \quad x \in \Omega \quad (6.8.1.1)$$

with

$$u = 0 \quad \text{at the boundary } \Gamma \text{ of } \Omega. \quad (6.8.1.2)$$

Our purpose is to find the 3 smallest eigenvalues of the problem 6.8.1.1, 6.8.1.2 and their corresponding non-trivial eigenfunctions u . Discretization of 6.8.1.1, 6.8.1.2 by the finite element method yields the following generalized eigenvalue problem:

$$\mathbf{S} \mathbf{u} = \lambda \mathbf{M} \mathbf{u}, \quad (6.8.1.3)$$

with \mathbf{S} and \mathbf{M} positive definite. The region Ω is sketched in Figure 6.8.1.1 The discretization

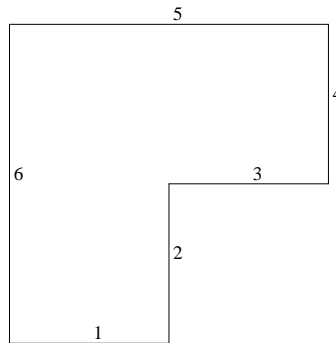


Figure 6.8.1.1: L-shaped region for the eigenvalue problem

is performed with linear triangles and the standard Laplacian equation described in the manual Standard Problems Section 3.1 (type number 800).

To create the mesh, program SEPMESH is used with input file eigenvalue.msh:

```

# eigenvalue.msh
#
# mesh file for eigenvalue problem in an L-shaped region
# See Users Manual Section 6.8.1
#
#
# The shape of the mesh is defined as follows:
#
#          P6          C5          P5
#          *-----*
#          |          |          |
#          |          |          | C4
#          |          |          |
#          C6|          P3          |
#          |          |          *-----*
#          |          |          | C3          P4
#          |          |          |
#          |          |          | C2
#          |          |          |
#          *-----*
#          P1          C1          P2
#
# To run this file use:
#   sepmesh eigenvalue.msh
#
# Creates the file meshoutput
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    width_lower = 1      # width of the lower part of the L-shape
    width       = 2      # width of the complete L-shape
    height      = 2      # height of the complete L-shape
    height_lower = 1      # height of the lower part of the L-shape
end
#
# Define the mesh
#
mesh2d              # See Users Manual Section 2.2
  coarse(unit=0.05) # The concept of coarseness is used with a unit length
                    # of 0.05
#
# user points
#
points              # See Users Manual Section 2.2
  p1=( 0           , 0           ,1) # Point left under
  p2=( width_lower, 0           ,1) # Point right under (lower part)
  p3=( width_lower, height_lower,1) # Point right upper (lower part)
  p4=( width       , height_lower,1) # Point right under (upper part)
  p5=( width       , height      ,1) # Point right upper
  p6=( 0           , height      ,1) # Point left upper
# In all these points we have a unit
# length of 0.05
#

```

```

# curves
#
curves          # See Users Manual Section 2.3
  c1=cline1(p1,p2)      # lower boundary
  c2=cline1(p2,p3)      # right boundary of lower part
  c3=cline1(p3,p4)      # lower boundary of upper part
  c4=cline1(p4,p5)      # right boundary of upper part
  c5=cline1(p5,p6)      # upper boundary
  c6=cline1(p6,p1)      # left boundary
#
# surfaces
#
surfaces        # See Users Manual Section 2.4
  s1=general3(c1,c2,c3,c4,c5,c6)
plot            # make a plot of the mesh
               # See Users Manual Section 2.2
end

```

The mesh created is shown in Figure 6.8.1.2 In the computational part (program SEPCOMP) 3

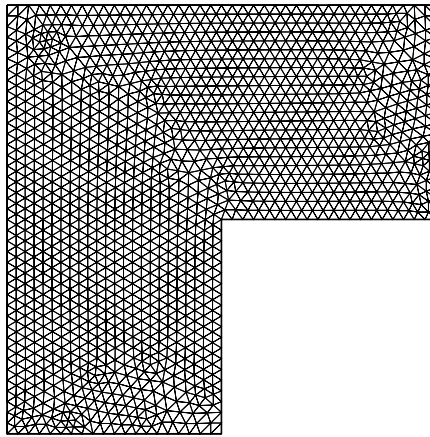


Figure 6.8.1.2: Mesh of L-shaped region generated by GENERAL

eigenvalues and eigenvectors are computed. The eigenvalues are stored as scalars and printed. The eigenvectors get the names `eigenvector_j` with $j = 1, 2, 3$. These names are reused in the postprocessing. The eigenvectors are written to the postprocessing file `sepcom.out`. Besides that also the integrals over the eigenvectors are computed and printed, in order to show how one can manipulate the vectors just computed. The input file is given below:

```

# eigenvalue.prb
#
# problem file for eigenvalue problem in an L-shaped region
# See Users Manual Section 6.8.1
#

```

```
# To run this file use:
#   sepcomp eigenvalue.prb
#
# Reads the file meshoutput
# Creates the file sepcomp.out
#
# Define some general constants
#
constants          # See Users Manual Section 1.4
  reals
    mu   = 1      # diffusion coefficient
    rho  = 1      # density
  vector_names
    eigenvector_1      # The first three vectors are the eigenvectors
    eigenvector_2
    eigenvector_3
  variables
    eigenvalue_1      # The first three scalars are the eigenvalues
    eigenvalue_2
    eigenvalue_3
end
#
# Define the type of problem to be solved
#
problem            # See Users Manual Section 3.2.2

  types            # Define types of elements,
                  # See Users Manual Section 3.2.2
    elgrp1 = (type=800) # A Laplacian equation is solved, see manual
                  # Standard Problems Section 3.1
  essbouncond      # Define where essential boundary conditions are
                  # given (not the value)
                  # See Users Manual Section 3.2.2
    curves(c1,c6)  # In all boundaries of the region
                  # essential boundary conditions are defined
                  # phi = 0
end

# Define the structure of the problem
# In this part it is described how the problem must be solved
# This is necessary because the integral of the pressure over the boundary
# is required
#
structure          # See Users Manual Section 3.2.3
# compute 3 eigenvalues and eigenvectors
compute_eigenvalues, num_eigval = 3, eigenvector_1, eigenvalue_1
print eigenvalue_1, text = 'eigenvalue 1' # Print the 3
print eigenvalue_2, text = 'eigenvalue 2' # eigenvalues
print eigenvalue_3, text = 'eigenvalue 3'
output            # write the three eigenvectors for postprocessing purposes
end

# Define the structure of the large matrix
# See Users Manual Section 3.2.4
```

```

matrix
  # The matrix is a compact symmetric matrix
  # This is necessary for the eigenvalue algorithm
  storage_method = compact, symmetric
end

# Define the coefficients for the problem
# All parameters not mentioned are zero
# See Users Manual Section 3.2.6 and Standard problems Section 3.1

coefficients
  elgrp1 ( nparm=20 )      # The coefficients are defined by 20 parameters
  icoef1 = 1              # diagonal mass matrix
  coef6 = mu              # a11 = mu (1)
  coef9 = coef 6          # a22 = a11
  coef17 = rho            # rho = 1 (mass matrix)
end

# Definition of the eigenvalue input
# See Users Manual Section 3.2.18

eigenvalues
  seq_coef = 1            # Coefficients for the differential equation
                          # Since a vector sequence number is given in
                          # structure, automatically eigenvectors
                          # are computed
                          # All other parameters are standard
end

end_of_sepran_input

```

To show the eigenvectors, program seppost is used with the following input:

```

# eigenvalue.pst
# Input file for postprocessing for eigenvalue problem in an L-shaped region
# See Users Manual Section 6.8.1
#
#
# To run this file use:
#   seppost eigenvalue.pst > eigenvalue.out
#
# Reads the files meshoutput and sepcomp.out
#
postprocessing          # See Users Manual Section 5.2

# Print the three eigen vectors
# See Users Manual Section 5.3
# The names of the vectors have already been defined in the input for sepcomp

print  eigenvector_1
print  eigenvector_2
print  eigenvector_3

# Define plot identification
# See Users Manual Section 5.4

```

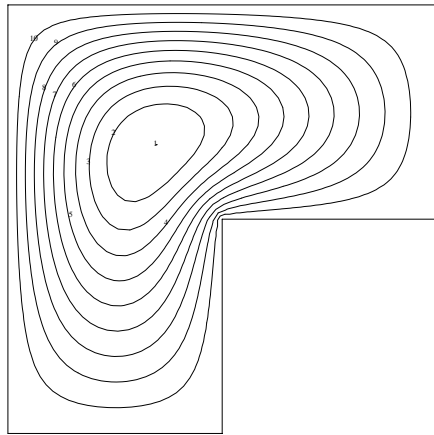
```
plot identification, text='Example of eigenvalue problem', origin=(3,18)

# Make contour plots and coloured level plots of all three eigenvectors
# See Users Manual Section 5.4

plot contour eigenvector_1
plot coloured contour eigenvector_1
plot contour eigenvector_2
plot coloured contour eigenvector_2
plot contour eigenvector_3
plot coloured contour eigenvector_3

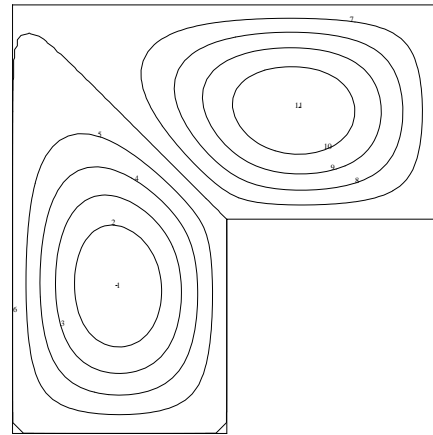
end
```

Figures 6.8.1.3 to 6.8.1.5 show the contour lines of the eigenvalues, Figures 6.8.1.6 to 6.8.1.8 show the colored contour levels.



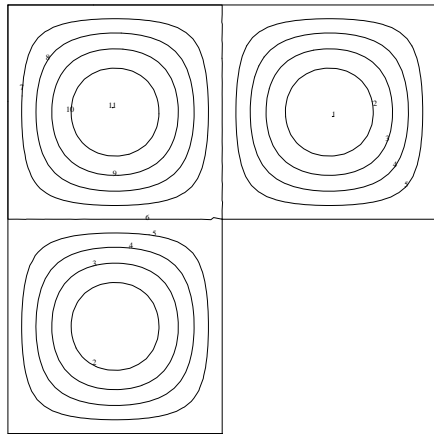
Contour levels of eigenvector_1

Figure 6.8.1.3: Contour lines of eigenvector 1



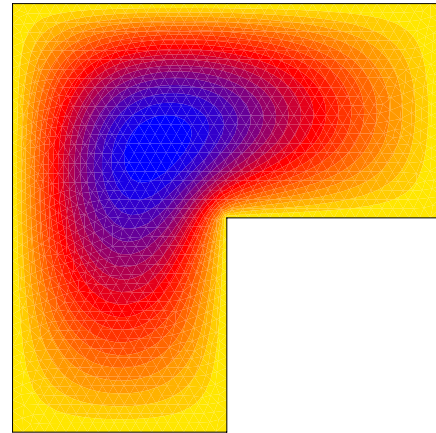
Contour levels of eigenvector_2

Figure 6.8.1.4: Contour lines of eigenvector 2



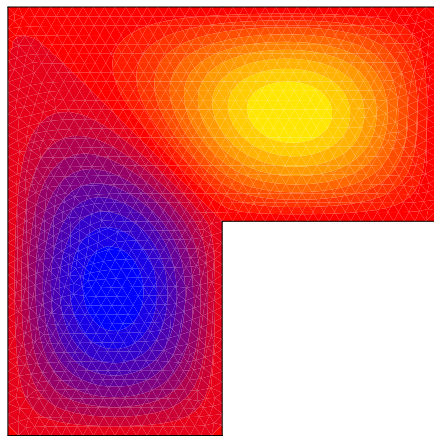
Contour levels of eigenvector_3

Figure 6.8.1.5: Contour lines of eigenvector 3



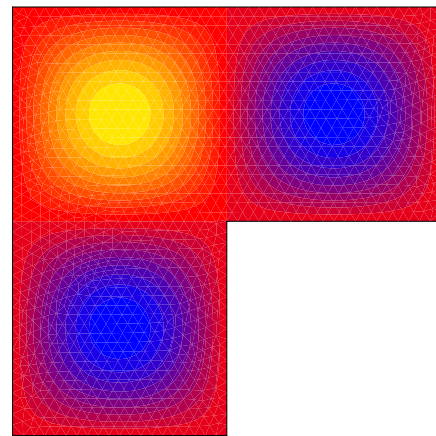
Contour levels of eigenvector_1

Figure 6.8.1.6: Colored contour levels of eigenvector 1



Contour levels of eigenvector_2

Figure 6.8.1.7: Colored contour levels of eigenvector 2



Contour levels of eigenvector_3

Figure 6.8.1.8: Colored contour levels of eigenvector 3

References

- Caswell and Viriyayuthakorn** (1983) Finite element simulation of die swell for a Maxwell fluid, J. of Non-Newt. Fluid Mech. Vol 12, 13-29.
- Cuthill, E. and J. McKee** (1969) Reducing the band width of sparse symmetric matrices. Proc. ACM Nat. Conf. Association of Computing Machinery, New York.
- W.D. Murray and F. Landis** (1959) Numerical and machine solutions of transient heat-conduction problems involving melting or freezing, Trans. ASME (C) J. Heat Transfer, 81, p. 106-112.
- Segal Guus, Kees Vuik and Fred Vermolen** (1997) A conservative discretization for the free boundary in a two-dimensional Stefan problem. Report
- Sloan, S.W.** (1986) An algorithm for profile and wavefront reduction of sparse matrices. Int. J. for Num. Meth. in Engng, 23, p. 239-251.

Index

approximate projection vector [3.2.8](#)
arc [2.3](#)
array defined per element [1.1](#), [3.2.2](#)
array of special structure [1.1](#), [3.2.2](#)
array of the structure of the solution vector [1.1](#), [3.2.2](#)
avs [3.2.13](#)
adapting boundary of mesh [3.4.3](#), [3.4.4](#)
adapting of mesh [3.4.3](#)
bearing [3.2.24](#)
band method [2.2](#)
bend problem [6.3](#), [6.3.1](#), [6.3.2](#), [6.3.3](#)
block_ilu [3.2.8](#)
block_ssr [3.2.8](#)
boolean expression [3.2.3](#)
bool_expr [3.2.3](#)
boundary conditions [1.2](#), [3.2.2](#)
boundary conditions of the type u constant [1.2.5](#), [3.2.2](#)
boundary conditions of the type $\psi_r = c_2\psi_l + c_l$ [1.2.5](#), [3.2.2](#)
boundary element [1.1](#), [3.2.2](#)
boundary element group [1.1](#), [3.2.2](#)
boundary integral [3.2](#), [3.2.3](#), [3.2.14](#), [6.2.5](#)
brick [2.5](#), [2.5.1](#)
bubble [3.2.3.7](#)
capacity [3.2.3](#), [3.2.19](#), [6.2.10](#), [6.2.11](#)
carc [2.3](#)
cavitation [3.2.2.4](#), [3.2.24](#)
CFUN1B [3.3](#), [3.3.4](#)
CFUNOL [3.3](#), [3.3.5](#)
cg [3.2.8](#)
cgs [3.2.8](#)
change coefficients [3.2](#), [3.2.7](#), [3.2.9](#)
change coordinates [2.2](#), [2.2.1](#), [3.2.3](#)
change structure [3.2.3](#)
change topology [3.4.3](#)
channel [2.5.3](#)
circle [2.3](#)
cline [2.3](#)
coefficients [3.2](#), [3.2.6](#), [3.2.9](#)
colored mesh [5.4](#)
command record [1.4](#)
compatibility [1.1](#), [6.2.8](#)
COMPCONS [1.4](#), [1.6](#)
compute bubble [3.2.3.7](#)
conjugate gradients [3.2.8](#)
conjugate vector [3.2.3](#)
connection elements [1.2.3](#), [1.2.6](#), [2.2](#)
constants [1.4](#), [1.6](#), [3.2.3](#)
constraint [3.2.8](#)
contact [3.2.2](#), [3.2.16](#), [3.2.2](#), [3.2.10](#), [3.2.14](#)
contact algorithm [3.2.16](#)
contact distance [3.2.16](#)
contact force [3.2.16](#)
contact surface [3.2.16](#)

copy 3.2.3
coupled problems 1.1, 1.3, 3.2.9, 3.2.15, 6.4.5
cparam 2.3
cprofile 2.3
create vector 3.2, 3.2.10
cross-section 3.2.2
cspline 2.3
CTIMEN 3.2.15, 6.4.1
curve 2.2, 2.3
curve generator 2.3
CUSCONS 1.6
CUSNAME 1.6
database 3.2.1, 3.2.3
data record 1.4
defect correction 3.2.8
deform mesh 3.2.3
degree of freedom 1.1, 3.2.2
derivatives 3.2, 3.2.3, 3.2.11, 3.2.15, 6.2.5
direct method 3.2.8
dissolution of particles 6.3.2, 6.3.3
eigenvalues 3.2.3, 3.2.18, 6.8.1
eigenvectors 3.2.3, 3.2.18, 6.8.1
elasticity equation 6.4.5
ELCERV 4.6
ELDERV 4.5
electrode 6.2.8, 6.2.9, 6.2.10
ELEM 4.2
ELEM1 4.3
ELEM2 4.4
element group 1.1, 3.2.2
element group properties 2.2
element subroutine 4.1, 4.2
ELINT 4.7
ELSTRM 4.8
environment file 1.5
essential boundary conditions 1.2.1, 3.2, 3.2.2, 3.2.5
essential boundary conditions not connected to degrees of freedom 1.2.4
extract vector 3.2.3
files 3.5
file with electrodes and capacities 3.5, 3.5.4
file with elements and values 3.5, 3.5.3
file with nodes 3.5, 3.5.1
file with nodes and values 3.5, 3.5.2
film method 3.4.4
for-loop 3.2.3, 6.2.9
FRAMESURF 2.4.12
free surface problems 3.1, 3.4, 3.4.5, 3.4.6, 6.3
FUNC1B 3.2.10, 3.3, 3.3.4
FUNALC 3.3, 3.3.1
FUNALG 3.3, 3.3.1
FUNCC1 3.3, 3.3.6
FUNCC3 3.3, 3.3.6
FUNCCOOR 2.2, 2.2.1
FUNCCR 3.3, 3.3.9, 3.4.4
FUNCCV 2.3, 2.3.1

FUNCFL 3.3, 3.3.3
FUNCOL 3.3, 3.3.5
FUNCSOLCR 3.3, 3.3.15, 3.4.4
FUNCTR 3.2.2, 3.3, 3.3.7
FUNCVECT 3.2.10, 3.3.11
FUNSCAL 3.3, 3.3.2, 6.2.8
Gaussian elimination 3.2.8
general 2.4, 2.4.1, 2.5, 2.5.4
general_constants 1.4, 1.7
generalized alpha 3.2.15
generalized theta 3.2.15
GETCONST 1.4, 3.3.12.2
GETINT 1.4, 3.3.12.1
GETNAMEINT 1.4, 3.3.14.1
GETNAMEREAL 1.4, 3.3.14.2
GETNAMEVAR 1.4, 3.3.14.3
GETVAR 1.4, 3.3.12.3
global_elements 3.2.2
global_unknowns 3.2.2
gmres 3.2.8
gmresr 3.2.8
heat equation 6.4, 6.4.1, 6.4.2, 6.4.5
ilu 3.2.8
imaginary vector 3.2.3
include 1.4
input file 1.4
instationary free boundaries 3.4, 3.4.2, 3.4.6, 6.3
integer properties 2.2, 3.2.8
integers 1.4, 1.6
integral 3.2, 3.2.3, 3.2.12, 6.2.5
intermediate points 2.2
interpolate solution 3.4.3
intersection 5.4
inverse problem 3.2.3, 3.2.20, 6.2.11
isopar 2.4, 2.4.9
iteration 3.2.8
iterative improvement 3.2.8
Krylov space 3.2.8
Kumar 3.2.24
length 3.2.3
level 3.2.2
level set 3.2.2, 3.2.2.4, 3.2.3.18, 3.2.10
line 2.3
linear combination 3.2.3
linking SEPRAN programs 2.1
local transformations 3.2.2
loop 3.2.3
lumping 3.2.8
mass conserving 3.2.24
matrix 3.2, 3.2.4
mesh deformation 3.2.3
mesh refinement 2.2, 3.2.3, 6.2.4
mesh velocity 3.4.6
MESHUS 2.4, 2.4.6
modulus 3.2.3

moving boundary problems 3.1, 3.4
natural boundary conditions 1.2.2, 3.2.2
Navier-Stokes 3.2.23, 6.3, Newmark 3.2.15
6.3.1, 6.3.2, 6.3.3
non-linear equations 3.2, 3.2.3, 3.2.9, 6.3
norm 3.2.3
NPROB 1.3
obstacle 2.2, 3.2.2, 3.2.10
open_dx 3.2.13
output 3.2, 3.2.3, 3.2.13
overrelaxation 3.2.8
parallel 2.2, 3.6, 3.6.1
PARAM 2.3
PARSURF 2.4, 2.4.8
particle trace 5.4
periodical boundary conditions 1.2.3, 2.2, 3.2.2, 6.2.6, 6.2.7
permittivity 3.2.19, 3.2.20, 6.2.9, 6.2.10, 6.2.11
phase 3.2.3
pipe 2.5, 2.5.2
pipe surface 2.4, 2.4.5
plot 5.1, 5.4
plot colored levels 3.2.3, 5.1, 5.4
plot colored mesh 5.4
plot contour 3.2.3, 5.1, 5.4
plot mesh 5.4
plot tensor 3.2.3
plot vector 5.1, 5.4, 3.2.3
positive definite 3.2.8
post processing 5.1
preconditioning 3.2.8
preprocessing 2.1
prescribe boundary condition 3.2.3
prescribed degree of freedom 1.1, 3.2.3
pressure correction 3.2.3.3, 3.2.22, 3.2.23
PRGETNAME 1.4, 3.3.14.4
principal stresses 3.2.3
print 5.1, 5.3
problem 3.2, 3.2.2
profile 2.3
profile method 2.2
projection method 3.2.8
projection vector 3.2.8
properties 2.2
PUTINT 1.4, 3.3.13.1
PUTREAL 1.4, 3.3.13.2
PUTVAR 1.4, 3.3.13.3
quadratic (velocity profile) 3.2.10
quadrilateral 2.4, 2.4.3
ray tube 2.5.3
reaction force 3.2.3, 3.2.11
read vector 3.2.3
real properties 2.2
real vector 3.2.3
reals 1.4, 1.6
rectangle 2.4, 2.4.2

reflect 2.3, 2.4, 2.5
renumber 2.2, 3.2.1, 3.2.2
refinement of the mesh 2.2, 3.2.3, 3.2.21, 6.2.4
residual 3.2.8
rotate 2.3, 2.4, 2.5
scalar 1.4, 1.6, 3.2.2, 6.2.8
scaling 3.2.8
sensor 6.2.8, 6.2.9, 6.2.10
sepcombineout 3.6
sepcomp 1, 3.1, 3.2
sepfree 3.1, 3.4
seplink 2.1
sepmakeparmesh 3.6, 3.6.1
sepmesh 1, 2.1, 2.2
sempi 3.6
seppost 1, 5.1, 5.2
sepran.env 1.5
set commands 1.4
similar 2.4
simple iteration 3.2.4, 3.2.2.13, 3.2.8
simple gcr 3.2.8
solve 3.2, 3.2.3, 3.2.7
solve linear system 3.2.3
solve non-linear system 3.2.3
solve time-dependent system 3.2.3
special files 3.5
sphere 2.4, 2.4.11
spline 2.3
spline curve 2.3
standard element 1.1
start 3.2, 3.2.1
stationary free surface problem 3.4, 3.4.5
stefan problem 6.3.1, 6.3.2, 6.3.3
stream function 4.8, 5.2
structure 3.2, 3.2.3, 3.4.2, 6.4.2
substepping 3.2.22
subtract 3.2.3
surface 2.2, 2.4
surface generator 2.4
tensor 3.2.3
time integration 3.2, 3.2.3, 3.2.15, 6.4
time history 5.5, 3.2.15
time_loop 3.2.3, 6.4.5
transformation 2.2
transformation matrix 3.2.2
translate 2.3, 2.4, 2.5
triangle 2.4, 2.4.7
type 3.2.2
USERBOOL 3.2.3, 3.3.8, 6.3.3
user curve 2.3, 2.3.1, 2.3.2
USEROUT 3.2.3
USEROUTS 3.2.3
user point 2.2
variables 1.4, 1.6, 3.2.2, 3.2.3, 6.2.9
vector 3.2.3

vector defined per element [1.1](#), [3.2.2](#)
vector names [1.4](#), [3.2.2](#)
vector of special structure [1.1](#), [3.2.2](#)
vector of the structure of the solution vector [1.1](#), [3.2.2](#)
velocity profile [5.4](#)
volume [2.2](#), [2.5](#)
volume generator [2.5](#)
while loop [3.2.3](#), [6.3.3](#)