SEPRAN

SEPRA ANALYSIS

INTRODUCTION

GUUS SEGAL

SEPRAN INTRODUCTION

January 2015

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

0 Contents

1 Introduction

The aim of this introductory manual is to make it possible for an inexperienced user to run simple SEPRAN programs without the necessity of studying all the possibilities SEPRAN provides.
The installation of SEPRAN is described in Section 1.1.
The global structure of a SEPRAN-session is described in Section 1.2.
Sections 1.3 to 1.6 describe the use of the main sepran programs.
Finally in Section 1.7 we repeat the main sepran commands.
Chapter 2 gives a number of examples. The user is advised to study this manual using a suitable example as starting point. An extended number of examples can be found in the manual examples.
Chapter 3 gives an introduction to mesh generation.
Chapter 4 describes a number of input options for the computational program.

1.1 Installation of SEPRAN

To install SEPRAN in a linux environment, perform the following tasks:

- Create a directory where you put sepran, for example `$HOME/sepran`
- Adapt the file `.profile` in your home directory `$HOME` by adding the following lines:

  ```
  export SPHOME=sepran_dir
  export PATH=$SPHOME/bin:${PATH}
  ```

  where `sepran_dir` is the name of the sepran home directory, for example `export SPHOME=$HOME/sepran`.
  After this change you may address the sepran home directory by `$SPHOME`. It may be necessary to open a new window or to login again before the change is effective.
  The statement `export PATH=$SPHOME/bin:${PATH}` makes sure the `$SPHOME/bin` is in your path and therefore you can use all sepran commands. If `.` is not in your path, it is advised to use: `export PATH=.:$SPHOME/bin:${PATH}`. Otherwise you have to use `./` for all local commands.
- Put the file SEPunixxxxx.zip, with xxxx the sequence number into `$SPHOME`.
- Carry out the following statements

  ```
  cd $SPHOME
  unzip SEPunixxxxx.zip
  cd bin/update
  sepinstall linux64
  ```

  If `.` is not in your path, use `./sepinstall linux64` The option linux64 implies that you use a 64 bits computer and the standard gfortran compiler
- If you want to use system installed blas subroutines use

  ```
  touch nosepranblas
  ```

  before installing sepran.
  Perhaps you have to update the file `$SPHOME/SEP_options` to make sure that the correct blas or laplack directory is used.
- For other options consider the file `$SPHOME/bin/update/sepinstall`.

On some computers, for example Ubuntu, installation of sepview is not possible. In that case you have to use jsepview for viewing.

1.2 The global structure of a SEPRAN-session

Except for complicated problems, the SEPRAN-session consists of four parts, which can be run separately. The four parts are

- Preprocessing (creation of mesh)
  This is done by program `sepmesh`.
- Viewing (plotting of the mesh)
  Program `sepview` or `jsepview`.
- Computation (definition and solution of the problem including postprocessing)
  This step is done by program `sepcomp`.
- Viewing (plotting of the results)
- If necessary postprocessing may be done separately.
  Program `seppost`.

The sequence of the session is always:

preprocessing followed by computation followed by postprocessing

1.3 The preprocessing part of SEPRAN

The preprocessing part of SEPRAN consists of the creation of the mesh. In this manual only one and two dimensional meshes are treated; for three dimensional regions the reader is referred to the Users Manual.

The generation of the mesh is performed by the program SEPMESH. At this moment only a batch version is available, which requires a file with data. This file must be created by the user for example with a text editor.

Program SEPMESH creates output in two ways:

- SEPMESH writes to the standard output device (usually the display from which you start the program, or a standard output file). This output consists of a copy of the input file, error messages if the input is incorrect and some messages from sub mesh generators.
- If the input is error-free and a mesh has been generated, then this mesh is written to a file named meshoutput.

Chapter 3 describes how a 2D-mesh may be generated, for a 3D-mesh the user is referred to the Users Manual.

SEPMESH must be used as follows:

```
sepmesh  inputfile
```

or

```
sepmesh  inputfile  >  outputfile
```

The input file is the file created by the user using the text-editor. If no output file is specified all information (including error messages) is written directly to the screen.
The output file may have any name except meshoutput and sepplot.$*\,*\,*$, where $*$ is any number.

Example: sepmesh mesh.dat > mesh.out.

Remark: besides the file meshoutput sepmesh also creates files sepplot.0001, sepplot.0002, etc. which contain plot information.

So sepmesh creates output in 3 ways:

- a file meshoutput containing the complete description of the mesh;
- files sepplot.0001, sepplot.0002, etc. containing information of the plots to be made;
- output written to the screen or the output file (for example mesh.out). This output contains a hard copy of the input, error messages (if any) and some information about the mesh.

1.4 The computational part of SEPRAN

In the computational part of SEPRAN first the mesh created by the preprocessing part is read, then the type of problem is defined, the system of equations is built (including boundary conditions) and finally the problem is solved. The result of this part is written to a file which can be used at the output part. SEPRAN is developed to solve very complicated problems. However, in this introduction only very simple standard problems are treated, which require only a minimum of input.

For simple problems SEPRAN provides a standard program: SEPCOMP. If the problem to be solved fits within the frame-work of SEPCOMP, there is no need to create a main program. In that case it is sufficient to run SEPCOMP itself. The input required for SEPCOMP is described in Chapter 4.

If the program SEPCOMP does not offer all the possibilities, that are needed for the solution of a specific problem, it is necessary to write a main program. SEPRAN provides a large number of subroutines in an increasing sequence of detail, which can be used to construct such a main program. The main subroutines are treated in the SEPRAN users manual, for an extended description the reader is referred to the programmers guide.

The computational part of a simple problem consists of the following components:

- In the starting part the mesh is read from the file created by SEPMESH, the definition of the problem, and the type of solver is read.
- In the next phase the essential boundary conditions are read, i.e. the prescribed unknowns.
- In the third phase the coefficients (material properties) are read, the system of equations is built and the problem is solved.
- Finally some derived quantities (like for example gradient, pressure, stream function) may be computed, and the solution as well as these derived quantities are written to a file.

SEPCOMP must be used as follows:

```
sepcomp  inputfile
```

or

```
sepcomp  inputfile  >  outputfile
```

The input file is the file created by the user using the text-editor. If no output file is specified all information (including error messages) is written directly to the screen.
Besides the user provided input file, sepcomp also needs the file meshoutput created by sepmesh. The output file may have any name except meshoutput, sepcomp.out and sepplot.$***\ast$, where $\ast$ is any digit.

Example: sepcomp comp.dat > comp.out.

Remark: sepcomp creates one file sepcomp.out, which contains information for the postprocessing.

So sepcomp creates output in 2 ways:

- the file sepcomp.out containing the complete description of the solution computed;
- output written to the screen or the output file (for example comp.out). This output contains a hard copy of the input, error messages (if any) and some information about the problem solved.

1.5 The postprocessing part of SEPRAN

Most postprocessing of SEPRAN can be done in the computational part.
Sometimes it may be more suitable to do the postprocessing separately. In that case one can use program SEPPOST.
In the postprocessing part of SEPRAN, the mesh is read as well as the solution, as created by the computational part. In this section the results are produced for the user in a more suitable form: prints, plots, integrals etc. The postprocessing part is performed by the program SEPPOST. For a description of its possibilities the reader is referred to the Users Manual.

SEPPOST is used in the same way as SEPMESH, i.e. the user creates an input file by the text-editor and then runs SEPPOST.
SEPPOST is used as follows:

```
   seppost  inputfile
```

or

```
   seppost  inputfile  >  outputfile
```

The input file is the file created by the user using the text-editor. If no output file is specified all information (including error messages) is written directly to the screen.
Besides the input file SEPPOST uses also the files created by SEPMESH (meshoutput) and the SEPRAN computational program (sepcomp.out).
SEPPOST does not produce plots directly but produces files named sepplot.0001, sepplot.0002, etc. containing plot information. Since this name is the same as for SEPMESH the plot information of SEPMESH is destroyed. To display the plot exactly the same procedure as for SEPMESH must be used.

Example:

```
   seppost  post.dat > post.out
```

1.6 Display of SEPRAN plots

The SEPRAN mesh generation part or the postprocessing part may generate plot files named sepplot.0001, sepplot.0002, sepplot.0003, etc.

In SEPRAN there are two ways of displaying these plots: you can make a picture at the screen, or you make a plot onto a laser printer or plotter.

To display plots on the screen, the following the command `jsepview` is available:
The command jsepview is activated by typing:

```
jsepview
```

or

```
jsepview sepplot.xxxx
```

where sepplot.xxxx is the file to be plotted.
If sepview is used without file name, the file may be selected by the option file. Once a file is selected all files with the same base name and extension .0001, .0002, ... may be viewed. The first file is the file selected.
jsepview has the following options:

Zooming in Press the left mouse button down and move the cursor upwards while pressing the button. Release the mouse button if the created rectangle is large enough. The picture within the rectangle will be drawn in the full window.

Zooming out Zooming out means displaying the previous window. Zooming out is done by moving the cursor downwards while creating a rectangle.

Play / Stop / Previous / Next At the upper right corner of the plot window, there are three buttons, a left arrow, a right arrow and a push button labeled ’Play’ or ’Stop’.

The name of a SEPRAN plot file is of the form nnnplot.xxx, where nnn is an arbitrary name, usually sep and xxx is a number. This number can be used to select a previous/next plot file of the same set with a higher/lower number, using the right and left arrow. If there is no plot file with a higher/lower number, the right/left arrow is disabled.
To show a set of plot files as an animation, you can use the ’Play’ button. As soon as SEPVIEW has started playing, the label on the button is changed to ’Stop’ to stop the animation. As soon as the last file in the set is shown, the animation is reversed.

To make hard copies of the files sepplot.xxxx you can either use

```
sepplot2eps
```

which creates eps files from the files sepplot.xxxx with names sepposc.xxxx.eps, or

```
sepplot2pdf
```

which creates pdf files from the files sepplot.xxxx with names sepposc.xxxx.pdf.

1.7 An overview of the simple SEPRAN commands

In this section we give an overview of some of the available SEPRAN commands. These commands are available both in a UNIX environment as well as in a MSDOS environment if you have the appropriate FORTRAN compiler at your proposal.

The following SEPRAN commands are available:

sepmesh (creates a SEPRAN mesh, see Section 1.3)

sepcomp (performs the computational part of SEPRAN, see Section 1.4)

seppost (performs the SEPRAN postprocessing, see Section 1.5)

jsepview (Plot SEPRAN files, see Section 1.6)

seplink (Link a SEPRAN main program and subroutines with the SEPRAN libraries, see Users Manual

sepplot2eps convert all sepplot.xxxx files to sepposc.xxxx.eps files.

sepplot2pdf convert all sepplot.xxxx files to sepposc.xxxx.pdf files.

sepman views a specific sepran manual. Options are

- **sepman intro** shows this manual
- **sepman um** Users Manual
- **sepman sp** Standard Problems
- **sepman exams** Examples
- **sepman pg** Programmer's Guide

If you want to use the SEPRAN commands in a UNIX operating system, it is necessary to add the SEPRAN bin directory to your path. Consult your local system officer on this issue or consult Section 1.1.

2 Some examples of complete runs

In this chapter we describe a number of complete examples in which we explain the structure of the input. A more formal description of the input or at least a part of it is given in the Chapters 3 (sepmesh) and 4 (sepcomp).
The following examples are treated:

2.1 treats the most simple example: a Laplace equation on a square.

2.2 is more complicated. Now we a source term, hence the Poisson equation and a non-homogeneous mixed boundary condition.

2.3 gives an example of the diffusion equation with different diffusion parameters in two parts of the domain.

2.4 shows how to work with a convection-diffusion equation

2.5 solves a 3D Laplace equation with iterative methods and also uses a 2d axi-symmetric case to compare the solutions.

2.6 solves a simple development of a flow in a channel. Both the solution of a linear Stokes equation as iteration of a non-linear Navier-Stokes equation is demonstrated.

2.7 shows the solution of a stationary isothermal laminar Newtonian flow in a T-shaped region by the Navier-Stokes equations.

2.8 is a linear elasticity problem (plane stress)

2.1 A simple example: the Laplace equation on a square

In this Section we describe the solution of an artificial mathematical example: the solution of a Laplace equation on a square.
Consider the square with edges C1 to C4 as sketched in Figure 2.1.1. in this region we solve the

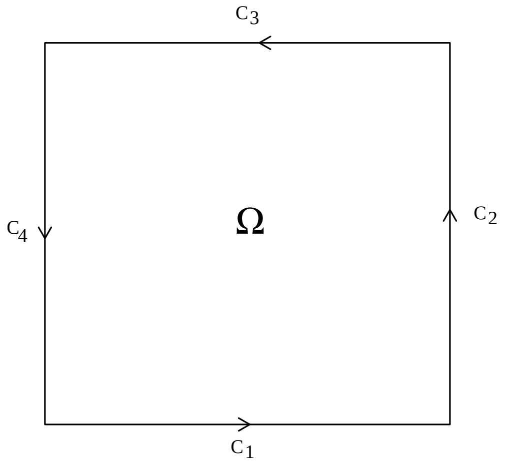


Figure 2.1.1: Definition of region for artificial mathematical example

Laplace equation with Dirichlet (i.e. prescribed) boundary conditions on the under (C1) and upper edge (C3), as well as on the right-hand-side (C2). At the left-hand side we have a no-flux condition (natural boundary condition).
On C1 and C2 the potential ϕ has value 1 and C3 $\phi = x$.
So we have to solve

$$-\Delta\phi = 0 \tag{2.1.1}$$

with boundary conditions:

$$\begin{aligned} \phi &= 0 && \text{at C1 and C2} \\ \phi &= x && \text{at C3} \\ \frac{\partial\phi}{\partial n} &= 0 && \text{at C4} \end{aligned} \tag{2.1.2}$$

The natural boundary condition does not require any input in the finite element method.

This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex laplace
```

To run the example use:

```
sepmesh laplace.msh
sepview sepplot.001
sepcomp laplace.prb
sepview sepplot.001
```

We subdivide the region into linear triangles. This is done by program `sepmesh`. This program requires an input file `laplace.msh` with the following contents

```
#  laplace.msh
#
#  example file for the generation of elements in a square
#  See the SEPRAN Introduction Section 2.1
#
#  To run this example use:
#
#  sepmesh square.msh
#
constants
   reals
      height = 1                    # height of the square
      width  = 1                    # width of the square
   integers
      nelm_hor  = 10                # number of elements in horizontal direction
      nelm_vert = 10                # number of elements in vertical direction
end
#
#    Actual definition of the mesh
#
mesh2d

   #   Definition of the coordinates the user points

   points
      p1=(0,0)
      p2=(width,0)
      p3=(width, height)
      p4=(0, height)

   #   Definition of the curves

   curves
      c1 = line ( p1,p2,nelm= nelm_hor )
      c2 = line ( p2,p3,nelm= nelm_vert )
      c3 = line ( p3,p4,nelm= nelm_hor )
      c4 = line ( p4,p1,nelm= nelm_vert )

   #   Definition of the surface

   surfaces
      s1 = general ( c1, c2, c3, c4 )

   #   Plot the mesh

   plot
end
```

Explanation:

All information following a hash (`#`) or an exclamation mark (`!`) and all lines with a star (`*`) in column 1 are considered to be comment and are not relevant for the program.
The input consists of two input blocks: `constants` and `mesh2d`.
The block `constants` defines a series of integers and reals to be used in the rest of the input.

The block `mesh2d` contains the actual input.
It consists of a number of subparts `points`, `curves` and `surfaces` to define the mesh as well as an auxiliary command `plot`.

mesh2d starts the block and indicates that the mesh is 2d.

points defines the user points, in this case the vertices of the square with its coordinates. These points are used to defines the curves.

curves defines all the curves used. These curves are needed to define the surface but also to define the boundary conditions.
The statement `c1 = line ( p1,p2,nelm= nelm_hor )` defines curve C1 as a straight line from user point P1 to user point P2.
Along this line we use `nelm_hor` elements.
Note that the direction of the curve is defined by the sequence of its user points.

surfaces defines the subdivision of the region as well as the boundary defines by the curves C1 to C4 in that direction. Should a curve be used in the opposite direction than use a minus sign before the C.
There are several types of surface generators, GENERAL is one of them.

plot indicates that a plot must be made of the mesh.

end ends the block `mesh2d`.

In this example we use the default type of elements, i.e. linear triangles.
Figure 2.1.2 shows the mesh created by this input block.

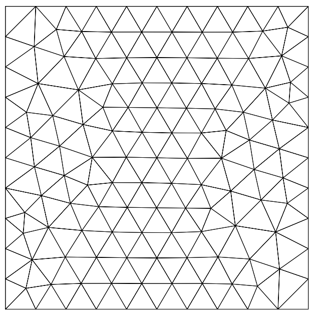


Figure 2.1.2: Mesh created by GENERAL

After creating and viewing the mesh we run the program sepcomp with the following input file

```
#
#  laplace.prb
#
#  example file for the solution of the Laplace equation on a square
#  See the SEPRAN Introduction Section 2.1
#
# Define the type of problem to be solved

problem
```

```
   laplace
   essential_boundary_conditions
      curves=(c1 to c3)
end

# Define which actions must be performed

structure

  #  Define the structure of the matrix

   matrix_structure: symmetric       # the matrix is symmetric

  #  Fill essential boundary conditions

   prescribe_boundary_conditions potential = 1, curves(c1, c2)
   prescribe_boundary_conditions potential = x_coor, curves(c3)

  # Build matrix and right-hand side and solve system of equations

   solve_linear_system potential

  #  Plot results

   plot_colored_levels potential
   plot_contour potential

  # Compute gradient of potential

   gradp = grad ( potential )
   plot_vector gradp

  # Compute volume integral of potential

   volint = integral(potential)
   print volint

end
end_of_sepran_input
```

2[ex] This input file consists of two blocks `problem` and `structure`. These blocks are standard for each input file for sepcomp. For complicated problems more blocks may be required, like for example a `constants` block as in the input for sepmesh. Each block ends with end. The statement `end_of_sepran_input` defines the end of the sepran input. This is only necessary if one uses extra input in own subroutines.

Explanation:

problem This block defines the type of problem to be solved including the boundary conditions.
The first statement is the type of problem, in this case `laplace`. For an overview of the known types, the user is referred to the manual Standard Problems.
The statement `essential_boundary_conditions` is followed by all curves where essential boundary conditions are given in this case `curves=(c1 to c3)` but also `curves=(c1, c2, c3)` could be used or:

```
   curves = (c1)
   curves = (c2, c3)
```

These statements only define where essential boundary conditions are given, not its actual values. These are defined in the structure block.
Since on C4 we have a homogeneous natural boundary condition, no input for C4 is necessary.

structure might be considered as the main program. It defines the steps to be performed and in which sequence.
In this example is starts with

```
   matrix_structure: symmetric
```

This indicates that a direct solver is used (default) and that the matrix is symmetric, so that only one half has to be stored. If we omit this statement the matrix is supposed to asymmetric.

```
   prescribe_boundary_conditions potential = 1, curves(c1, c2)
   prescribe_boundary_conditions potential = x_coor, curves(c3)
```

defines the values of the essential boundary conditions. It defines the potential implicitly as a vector defined over the mesh with one degree of freedom (unknown) per point.
`x_coor` is the default definition for the vector of x coordinates. In the same way the user may use `y_coor` and in R^3 `z_coor`.
`coor` defines the whole coordinate vector and has 2 components per point in R^2.

```
   solve_linear_system potential
```

indicates that the matrix and right-hand side must be build and the linear system of equations must be solved. The result is stored in the vector `potential`. For this specific problem no extra information is needed, but it is necessary that the essential boundary conditions have been filled before.

```
   plot_colored_levels potential
   plot_contour potential
```

These commands make a colored plot and a contour plot respectively of the potential.

```
   gradp = grad ( potential )
   plot_vector gradp
```

computes the gradient of the potential, stores the result in gradp and with the last statement we make a vector plot of the gradient.

```
   volint = integral(potential)
   print volint
```

is for demonstration only. `volint` is the integral over the volume.

2.2 Heat conduction in a plate with hole

In this Section we consider a square plate with a hole like sketched in Figure 2.2.1. In this plate

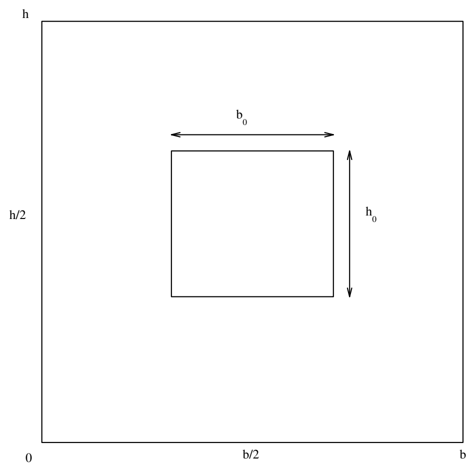


Figure 2.2.1: Square plate with hole

we want to solve a stationary heat conduction problem, which is described by the two-dimensional Poisson equation

$$-div k\nabla T \;=\; \phi. \tag{2.2.3}$$

T(x,y) denotes the unknown temperature , k the heat conduction coefficient and ϕ the production of heat per unit of volume and time.
In order to solve this problem uniquely, it is necessary to prescribe boundary conditions on the whole boundary. Types of boundary conditions that can be applied are for example:

- Temperature T prescribed at the wall.
- Heat flux prescribed at the wall.
  $q_n = 0$: insulated wall
  $q_n = q_0$: prescribed heat flux
  with q_n normal component of the heat flux $\mathbf{q} \;=\; k\nabla T$.
- Heat transfer relation of Newton
  $q_n = \alpha(T_w - T_\infty)$
  with α the heat transfer coefficient,
  T_w, T_∞: the temperature on the wall and far away from the wall respectively.

We assume that in the case of the plate in Figure 2.2.1, the left and right-hand wall are kept at a constant temperature T = 0. The lower wall is kept at temperature T = 1. The heat production ϕ is assumed to be constant.
On the boundary of the hole we assume that $q_n = 0$ (insulated) or that $q_n = q_0$ (constant heat flux).
In first instance we assume that the upper wall is insulated (q_n=0) and in second instance we use the heat transfer relation $\alpha T + k\nabla T \cdot \mathbf{n} = 0$.
This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex poisson1
```

for the first case or

```
sepgetex poisson2
```

for the second case.
Running this problem is the same as in Section 2.1.

2.2.1 insulated top wall

First we consider the case with the insulated wall. The mesh input file is:

```
# poisson1.msh
#
# mesh for plate with hole
# See the SEPRAN Introduction Section 2.2
# linear triangles are used
# Definition of constants:
#
constants
   reals
      b_2       = 0.5              # half width
      b0_2      = 0.2              # half width hole
      h         = 1                # height
      hole_low  = 0.3              # lower side hole
      hole_upp  = 0.7              # upper side hole
end

set warn off   # suppress warnings

# Mesh input:

mesh2d
   coarse (unit=0.05)
   points
      p1 = (0  ,0)             # origin
      p2 = (b_2, 0)            # point on the right under side
      p3 = (b_2, h)            # point on the right upper side
      p4 = (0, h)              # point on the top of the symmetry axis
      p5 = (0, hole_upp)       # point on the top of the hole on symmetry axis
      p6 = (b0_2, hole_upp)    # point on the right upper side of hole
      p7 = (b0_2, hole_low)    # point on the right lower side of hole
      p8 = (0, hole_low)       # point on the bottom of the hole on symmetry axis
   curves
      c1 = line(p1,p2)         # lower side
      c2 = line(p2,p3)         # right-hand side
      c3 = line(p3,p4,p5,p6,p7,p8,p1)  # upper side and left-hand side
   surfaces
      s1 = general ( c1,c2,c3 )
   plot
end
```

Compared to the example in Section 2.1 we see two new aspects:

```
   coarse (unit=0.05)
```

defines the global length of the elements along the curves. So in this case it is not necessary to define the number of elements along a line.

```
   c3 = line(p3,p4,p5,p6,p7,p8,p1)
```

constructs a line consisting of straight parts from user point to user point. This is only possible in combination with a given coarseness of the elements.

The input file for sepcomp reads

```
# poisson1.prb
#
# Poisson equation on plate with hole
# See the SEPRAN Introduction Section 2.2
#
# Definition of constants:

constants
   reals
      phi = 1      # source term
end

# Definition of problem:

problem
   poisson              # poisson equation
   essential_boundary_conditions
      curves(c1 to c2)      # Essential boundary condition on C1 and C2
end

#  structure of the main program

structure

  # Define structure of the matrix

   matrix_structure: symmetric     # matrix is symmetric

  # Define the essential boundary conditions on c1 (T=1)
  # The values in the rest of the region are automatically set to 0

   prescribe_boundary_conditions temperature = 1, curves (c1)

  # Create matrix and solve system of linear equations

   source = phi  ! define source term
   solve_linear_system temperature

   heat_flux = flux(temperature)

  # output of results

   print temperature
   print heat_flux

   plot_contour temperature
   plot_coloured_levels temperature
   plot_vector heat_flux

   no_output
```

```
end
end_of_sepran_input
```

In this example we have to use `poisson` instead of `laplace` since there is a right-hand side.
The following items are new to Section 2.1

```
   source = phi  ! define source term
```

this statement defines the source term (right-hand side) for the poisson equation as described in the manual Standard Problems.

```
   heat_flux = flux(temperature)
```

computes the heat flux defined by $flux = -k\nabla T$, where k is the diffusion constant (in this example 1).

```
   no_output
```

suppresses the writing of the solution to the files `sepcomp.out`. Writing is only necessary if `seppost` is used for postprocessing.

2.2.2 mixed boundary condition at top wall

For example poisson2 we need four different curves since the top wall must be identified. See the file `poisson2.msh`.
New in the file `poisson2.prb` are the following statements:

```
problem
   poisson              # poisson equation
   boundary_elements
      belm1 = curves(c3)     # Natural boundary condition on C3 (upper side)
```

Since at C3 we have a mixed boundary condition we need boundary elements. The line `belm1 = curves(c3)` defines such elements along curve C3.

```
   diff_sigma = alpha  ! radiation
```

defines the radiation parameter α in $\alpha T + k\nabla T \cdot \mathbf{n} = 0$.

2.2.3 mixed boundary condition at top wall combined with given heat_flux through the hole

Suppose we want to combine the mixed boundary condition with a given heat_flux through the hole, for example $q_n = 1$. In that case we need boundary elements along the top wall and boundary elements along the hole. These boundary elements need different input since the boundary conditions are different. To achieve this we need two boundary groups and it is not possible anymore to give all coefficients in the structure block. This case requires an extra block `coefficients`.

This example is called `poisson3` and can be downloaded in the usual manner.
The mesh file is slightly adapted since the hole must be addresses as a sepatate curve (c5).

Changes in the prb file are given below:

```
problem
   poisson              # poisson equation
   boundary_elements
      belm1 = curves(c3)     # Natural boundary condition on C3 (upper side)
      belm2 = curves(c5)     # Natural boundary condition on C5 (hole)
```

Since the boundary conditions at both parts are different we need two boundary groups marked by belm1 and belm2.
In this example the coefficients for the poisson equation remain unchanged and can be given in the structure block. However, this is not longer the case for the two non-homogeneous natural boundary conditions. Therefore it is necessary to add the following coefficients block:

```
   bngrp 1      # First boundary group (curve 3)
      diff_sigma = alpha         # alpha = 500
   bngrp 2      # Second boundary group (curves 5)
      diff_flux = h              # h = 1
```

The natural boundary condition requires two reals at input: `diff_sigma` giving the coefficient before the T in the mixed boundary condition and `diff_flux` which is the corresponding right-hand side.
So in this case we have $\alpha T + k\nabla T \cdot \mathbf{n} = h$, where for the first boundary group $h = 0$ and for the second one $\alpha = 0$.
Coefficients that are not given get the value 0, hence the fact that we give one coefficient for each group.

2.3 A diffusion problem in a L-shaped region

As an example we consider the solution of a diffusion problem in a L-shaped region, consisting of two regions S_1 and S_2 with different permeability constants $\mu(S_1)$ and $\mu(S_2)$. At the upper boundary C_5 the potential is equal to 1, at the lower boundary C_1 the potential is equal to 0. The other outer boundaries may be considered as insulators. The fluxes at the intersection of the region S_1 to S_2 must be continuous.
For a definition of the region as well as its corresponding geometrical quantities, see Figure 2.3.1

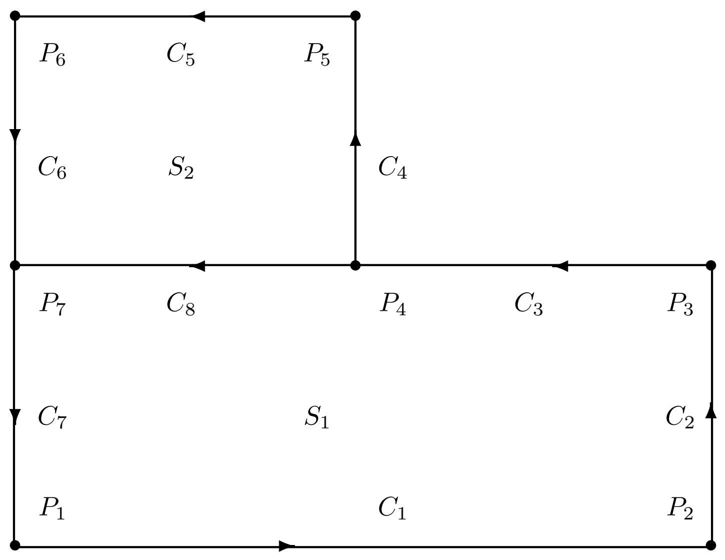


Figure 2.3.1: Definition of the L-shaped region with corresponding geometrical quantities

The mathematical formulation of this problem may be described as follows:

The potential problem is defined by

$$-\operatorname{div} \mu \bigtriangledown \phi = 0$$

with

$$\mu(S_1) = 1, \mu(S_2) = 2.$$

The boundary conditions are given by:

$$\phi(C_1) = 0, \quad \phi(C_5) = 1$$

$\mu \frac{\partial \phi}{\partial n} = 0$ along the curves C_2, C_3, C_4, C_6 and C_7.

The interface condition at boundary C_8 is given by:

$$\mu(S_1) \frac{\partial \phi}{\partial n}(S_1) = -\mu(S_2) \frac{\partial \phi}{\partial n}(S_2)$$

n denotes the outward normal
The boundary condition $\mu \frac{\partial \phi}{\partial n} = 0$ is a so-called natural boundary condition requiring no special arrangements in the finite element method. The same is the case for the coupling condition at C_8. So in fact these boundary conditions are not given explicitly, but by not prescribing anything they are satisfied automatically.

The easiest way to define the two values of μ in the regions S_1 and S_2 is to define two different element groups. So each element group is connected to a different value of the permeability.

This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex diffusion
```

Running this problem is the same as in Section 2.1.
The mesh input file is:

```
# diffusion.msh
#
# mesh for diffusion problem in L-shaped region
# See the SEPRAN Introduction Section 2.3
# Definition of constants:
#
#  To run this file use:
#     sepmesh diffusion.msh
#
#  Creates the file meshoutput
#
#  Define some general constants
#
constants
   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
      u = 0.1          # unit length of elements
end
set warn off   ! suppress warnings
#
#  Define the mesh
#
mesh2d
  coarse(unit=u)
#
#  user points
#
   points
      p1=(0,0)                        # left-hand-side point (lower edge)
      p2=(width,0)                    # right-hand-side point (lower edge)
      p3=(width, half_heigth)         # right-hand-side point (middle part)
      p4=(upper_right, half_heigth,0.5)# right-hand intersection point
                                      # 0.5 indicates that the elements have
                                      # length 0.5*u around this point
      p5=(upper_right, heigth)        # right-hand-side point (upper edge)
      p6=(0, heigth)                  # left-hand-side point (upper edge)
      p7=(0, half_heigth)             # left-hand intersection point
#
#  curves
#
   curves
      c1 = line (p1,p2)      # lower edge
      c2 = line (p2,p3,p4)   # right-hand-side edge and rhs upper edge of
                             # lower block
      c4 = line (p4,p5)      # right-hand-side edge of upper block
      c5 = line (p5,p6)      # upper edge
      c6 = line (p6,p7)      # left-hand-side edge of upper block
      c7 = line (p7,p1)      # left-hand-side edge of lower block
      c8 = line (p4,p7)      # intersection line
#
#  surfaces
#
   surfaces
      s1 = general (c1,c2,c8,c7)  # lower block
      s2 = general (-c8,c4,c5,c6)    # upper block
#
#  Couple each surface to a different element group in order to provide
#  different properties to the coefficients
#
   meshsurf
      selm1 = s1   # element group 1
      selm2 = s2   # element group 2

   plot                           # make a plot of the mesh

end
```

Compared to the examples in Seection 2.2 we see the following new items:

```
      p4=(upper_right, half_heigth,0.5)
```

User point P4 is provided with the coordinates, plus an extra number 0.5 which implies that the local coarseness is equal to 0.5*u. Omitting this number means that the coarseness is equal to u. So we have a local refinement around point P4.
An extra subblock is introduced:

```
    meshsurf
      selm1 = s1   # element group 1
      selm2 = s2   # element group 2
```

If omitted we have only one element group, but in this case we have two element groups (1 and 2) indicated by selm1 and selm2. This is necessary since the properties of the region in both surfaces differ.
Figure 2.3.2 shows the mesh created by SEPMESH.

The problem input file is defined by

```
#  diffusion.prb
#
#  problem file diffusion problem in L-shaped region
#  See the SEPRAN Introduction Section 2.3
#
#  To run this file use:
#     sepcomp diffusion.prb
#
#  Reads the file meshoutput
#  Creates the file sepcomp.out
#
#
#
```

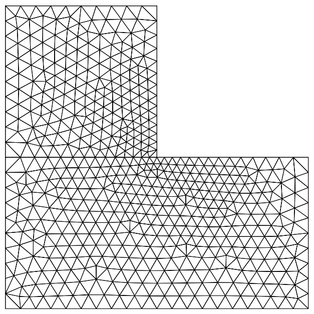


Figure 2.3.2: Mesh created by GENERAL

```
#  Define some general constants
#
constants
   reals
      mu_1      = 1                  # permeability in surface 1
      mu_2      = 2                  # permeability in surface 2
end

# Definition of problem:

problem
      diffusion                 # diffusion equation for all element groups
   essential_boundary_conditions  # Define where essential boundary conditions
                                  # are given (not the value)
      curves (c1)               # lower boundary
      curves (c5)               # upper boundary
end

#  structure of the main program

structure

  # Define structure of the matrix

   matrix_structure: symmetric     # matrix is symmetric

  # Define the essential boundary conditions on c1 (T=1)
  # The values in the rest of the region are automatically set to 0

   prescribe_boundary_conditions potential = 1, curves(c5)

  # Create matrix and solve system of linear equations

   solve_linear_system potential

  # output of results
```

```
   plot_contour potential
   plot_colored_levels potential
   print potential, curves=(c8)
end
#
#  The coefficients for the differential equation
#  All parameters not mentioned are zero
#
coefficients
   elgrp1
      diffusion =  mu_1     # Constant permeability
   elgrp2
      diffusion =  mu_2     # Constant permeability
end
end_of_sepran_input
```

Again we use a coefficients block to indicated the value of the diffusion per element group, just like in Section 2.2.3. In that example we used boundary groups because the coefficients where related to the natural boundary elements.

2.4 Heat transfer and temperature in a Poisseuille flow

The goal of this example is to study the influence of outstream boundary conditions on the convection-diffusion equation as well as to study the influence of the streamline Petrov Galerkin upwinding.

Consider the flow in a pipe as sketched in Figure 2.4.1. We suppose that the flow is a stationary

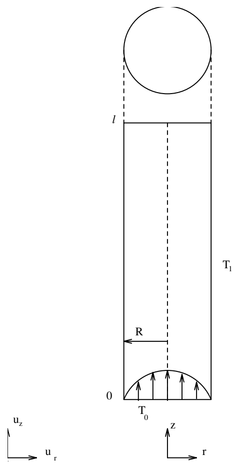


Figure 2.4.1: Pipe with Poisseuille flow

Poisseuille flow. The heat dissipation is neglected and there are no other heat sources. The temperature equation is considered stationary. Changes of the velocity due to temperature differences are neglected.

We consider the flow in a straight, cylindrical tube with radius R. For z = 0 the walls have a temperature T_0 and for z > 0 a temperature T_1. We consider the region $0 \leq$ z $\leq l$ and assume that the liquid on z = l has a homogeneous temperature. Since flow and temperature are axi-symmetrical it is sufficient to solve the problem in a cross-section with axi-symmetrical coordinates.
Define the inlet length l_{in} as the minimal length for which temperature various at most 1% through the cross section. For z $\geq l_{in}$, it can be shown that

$$l_{in} = 0.6\frac{Pe}{R},$$

$$Pe = \frac{U_{max}R}{a},$$

with Pe the Peclet number , U_{max} the maximum velocity and a the so-called temperature adjustment coefficient.

The convection-diffusion equation for the temperature can be written in dimensionless form as:

$$-\frac{1}{Pe}\Delta T + \mathbf{u}\cdot\nabla T = 0. \tag{2.4.1}$$

In our example we use the next data:

$T_0 = 0$

$T_1 = 1$

$u_r = 0$, $u_z = 1 - r^2$

$R = 1$ $Pe = 700$

On the lower boundary we have $T = T_0$, on the side wall $T = T_1$. On the symmetry axis we have

of course the symmetry boundary condition $\frac{\partial T}{\partial n} = 0$.
From these data it follows that the inlet length is equal to 420.
We compute the temperature for various choices of the outflow boundary and various boundary conditions on the outflow boundary.
We consider three cases:
`convdiffusion1` with length 500 and without upwind.
`convdiffusion2` with length 250 with upwind
`convdiffusion3` with length 250 without upwind but with Neumann conditions at outflow.
These examples can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex convdiffusionx
```

where x = 1, 2 or 3. Running this problem is the same as in Section 2.1.

2.4.1 Length 500 without upwind

The mesh file in this example is almost the same as in the previous examples. The only extra item is:

```
   type_elements = quadrilateral  # linear quadrilaterals are used
```

This statement ensures that bi-linear quadrilaterals are used instead of the default linear triangles.
The input file for sepcomp reads:

```
# convdiffusion1.prb
#
# Convection-diffusion equation in pipe problem (axisymmetric)
#

# Definition of constants:

constants
   reals
      k = 1/700        # diffusion constant
end

# Definition of problem:

problem
   convection_diffusion

   essential_boundary_conditions
      curves(c1 to c3)         # Essential boundary condition on c1 to c3
end

structure

  # The default matrix structure (non-symmetric) is used

  # Define the essential boundary conditions on c2 to c3 (T=1)

   prescribe_boundary_conditions temperature = 1, curves (c2 to c3)

  # Create matrix and solve system of linear equations
```

```
   coordinate_system = 'axi_symmetric'
   diffusion = k
   v_velocity = 1 - x_coor^2  ! uz = 1 -r^2

   solve_linear_system temperature

  # Plots of temperature

   plot_contour temperature
   plot_colored_levels temperature

   no_output

end
end_of_sepran_input
```

We see that simple expressions are allowed within the constants block as well as the structure block. Simple standard functions like `sin`, `cos` and `exp` may be applied. A power of a number or vector is denoted in the matlab way.
New in this problem file are:

```
   convection_diffusion
```

indicating that the convection-diffusion equation must be solved.

```
   coordinate_system = 'axi_symmetric'
```

denotes that we use axi-symmetric coordinates (r, z), which internally are called x and y. Since `axi_symmetric` is a string it must be placed between quotes.
Other options are

```
   coordinate_system = 'cartesian'
   coordinate_system = 'polar'
```

with obvious meaning. The first one is the default and may be omitted.

```
   diffusion = k
   v_velocity = 1 - x_coor^2  ! uz = 1 -r^2
```

defines the scalar diffusion k and the vector velocity in z direction. The r-velocity component is zero and may be skipped. We see that the r-velocity is a function of r which corresponds to `x_coor`. The power 2 is applied for each node in the vector `v_velocity`.
After computing this example one can see from the corresponding picture that there no oscillations near the outlet. However, if we reduce the inlet length to 250, there is clearly an oscillatory behavior.

2.4.2 Length 250 with upwind

To avoid the oscillations one may use upwind as is done in `convdiffusion2`. The mesh file is identical to the one in Section 2.4.1 except for the length.
In the sepcomp input file there is one extra statement before the solution of the linear system

```
   supg = 'critical'
```

indicating that stream line Petrov Galerkin upwind is used. `critical` is one of the options. See the manual Standard Problem Section 2.4.4 for all possibilities.
In this case the oscillations are more or less suppressed.

2.4.3 Length 250 without upwind, but Neumann at outflow

An alternative in case of a short length is to change the outlet boundary condition from Dirichlet to Neumann. This means that no essential boundary conditions are given at the top wall C3. The Neumann condition does not require extra input.
Also in this case the oscillations are gone.

2.5 3D Laplace equation and iterative methods

In this example we show a very simple 3d Laplace example and demonstrate how we can use iterative methods for the solution of the system of linear equations. We compute the temperature in a straight pipe. The lower wall and the upper wall of the pipe consist of parallel circles with radius 1. The outer wall is perpendicular to these circles and has height 5. The temperature in the pipe satisfies the Laplace equation

$$-divk\nabla T \ = \ 0. \qquad (2.5.1)$$

The pipe is insulated on the upper wall and on the under wall we have a constant temperature T_0.
On the outer wall we assume the heat transfer relation: $q_n = \alpha(T_w - T_\infty)$
To solve this problem, we can use the axi-symmetry or we can solve the problem as a three-dimensional one.
First we carry out the axi-symmetric approach.
The following 2 examples are available in the directory `$SPHOME/sourceexam/intro`.

```
laplace_axi
laplace_3d
```

They can be copied with sepgetex and run in the usual way.

2.5.1 Axi-symmetric approach

Since the problem is axi-symmetric there is no need to use the 3d approach. However, this is just a demonstration of how 3d examples work and how to use iterative methods. The iterative method is only shown in the 3d case, since for not too large 2d problems direct methods are usually faster than iterative ones.
The mesh input file and the prb file are standard and will not be repeated.
In this case the matrix is positive definite. For the direct solver the solver this means that it is possible to use a Cholesky decomposition instead of the standard LDL^T decomposition.
In order to give the linear solver this information an extra input block solve is required.

```
solve
   positive_definite          # Use Cholesky instead of L D L^T
end
```

2.5.2 3D case

In the 3d case we have to create a 3d mesh. The mesh input file is:

```
# laplace_3d.msh
#
# mesh for pipe (3D)
# Definition of constants:
#

set warn off   ! suppress warnings

constants
   reals
      height = 5       # height of the pipe
      radius = 1       # radius of pipe
   integers
      nelm_r = 20      # number of elements in phi-direction
```

```
                       # along the circle
      nelm_h = 25      # number of elements in height direction
end

# Mesh input:

mesh3d                        # Region is 3d
#
#  user points
#
   points

     # pd means point with first two coordinates given in radius and angle
     # in degrees (hence cylindrical coordinates)

      pd1 = (0,0,0 )               # Center of circle
      pd2 = (radius,0,0)           # (r,phi,z) for first point at bottom circle
      pd3 = (radius,180,0)         # (r,phi,z) for second point at bottom circle
      pd4 = (radius,0, height)     # (r,phi,z) for first point at top circle
#
#  curves
#
   curves
      # line means straight line, with given number of elements
      # circle means an arc with first point as center, second point as starting
      # point and third point to define the plane through the circle
      # number of elements is given
      # translate makes a direct copy of a curve

      c1 = circle(p1, p2, p3,nelm = nelm_r)   # bottom circle
      c2 = translate c1 (p4)                  # top circle
      c3 = line (p2,p4,nelm = nelm_h)     # generating curve along cylinder
#
#  surfaces
#
   surfaces         # Linear triangles are used
                    # General is a general 2d surface generator
                    # pipesurface generates a surface along a pipe
                    # It is an algebraic generator
                    # The first curve refers to the bottom
                    # The second curve to the top and the third one
                    # defines a line from bottom to top
                    # translate gives copy of the other surface

      s1 = general  (c1)                # bottom surface
      s2 = translate s1 (c2)            # top surface
      s3 = pipesurface (c1,c2,c3)       # cylinder
#
#  volumes
#
   volumes         # Linear tetrahedrons are used (11)
                   # The volume generator pipe is algebraic
                   # The first surface is the bottom surface
                   # The second one the top surface,
                   # it must have exactly the same topology as the bottom
```

```
                  # the third one a pipesurface

      v1 = pipe (s1,s2,s3)             # pipe

   plot, eyepoint = (50,50,50)    # make a plot of the mesh
end
```

There are several new aspects in this file that will be treated separately.

```
      pd1 = (0,0,0 )              # Center of circle
      pd2 = (radius,0,0)          # (r,phi,z) for first point at bottom circle
      pd3 = (radius,180,0)        # (r,phi,z) for second point at bottom circle
      pd4 = (radius,0, height)    # (r,phi,z) for first point at top circle
```

In stead of P1 to P4 we use Pd1 to Pd4. The d indicates that the coordinates are given in (r, ϕ, z) coordinates where ϕ is given in degrees. Internally this is translated into (x, y, z) coordinates. Furthermore a pipe is used. A pipe consists of a lower surface an upper surface and a so-called pipe surface connecting both surfaces. The top surface and bottom surface must have an identical subdivision in elements if the 3d generator pipe is used. pipe is a so-called algebraic generator and therefore much faster than general3d, which can handle complex geometries.

```
      s1 = general  (c1)            # bottom surface
      s2 = translate s1 (c2)        # top surface
      s3 = pipesurface (c1,c2,c3)   # cylinder
```

The bottom surface is made in the usual way. To make sure that the top has exactly the same topology as the bottom we use a translate. pipesurface makes a structured mesh between top and bottom.

```
      v1 = pipe (s1,s2,s3)             # pipe
```

creates the 3d elements. They are by default linear tetrahedrons.

```
   plot, eyepoint = (50,50,50)    # make a plot of the mesh
```

The eyepoint is necessary to make a hidden line 3d plot of the mesh. The coordinates of the eye point define the position of th viewer.

The prb file is almost identical to the axi-symmetric case. Of course boundary conditions are given along surfaces instead of curves. Since we use an iterative solver we have to inform the program that the matrix structure is no longer a profile storage but that only non-zero elements are stored (compact storage). This is done by

```
   matrix_structure: storage_scheme = compact, symmetric
```

The default iterative solver (Conjugate Gradients) is used with the default accuracy of 10^{-3}. To enlarge the iteration speed a preconditioner is used, in this case of type ILU (default).
The progress of the iteration is printed by the statement:

```
   sol_print = 2       ! print level for iterative solver
```

Keywords for the linear solver in the structure block always start with sol_. Another option is to add an input block SOLVE. In Section (4.5) this input block is partially described together with some options for the linear solver that can be used in the structure block.

2.6 Development of a flow in a straight pipe (Navier-Stokes flow)

Consider the flow between two flat plates. At the inflow we prescribe a constant velocity. After some time the flow will have been converted to a fully developed flow . In this exercise we shall simulate this numerically. Consider the configuration as sketched in Figure 2.6.1. Due to symmetry

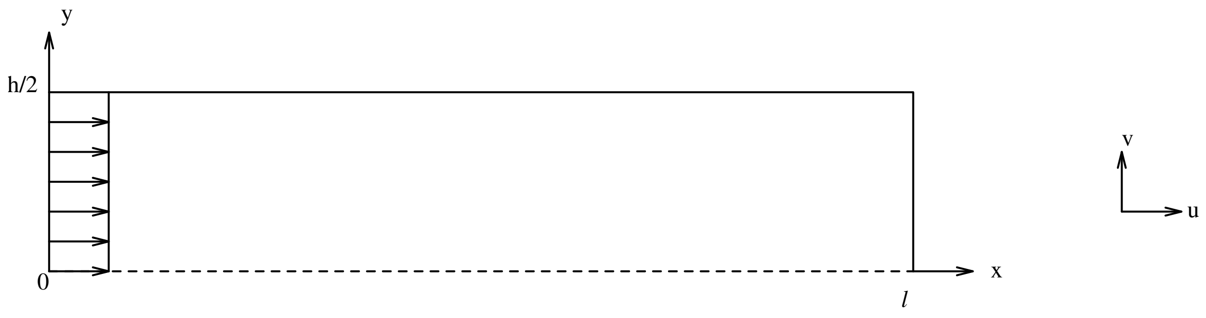


Figure 2.6.1: Straight pipe

it is sufficient to consider a half channel only. The liquid satisfies the Navier-Stokes equations. At the inflow we prescribe a constant velocity field parallel to the flat plates. On the fixed walls we have a no-slip condition. At the outflow we assume that the flow is parallel to the plates. Furthermore we suppose that the pressure is equal to 0.
In first instance we consider the linear Stokes equations, after that the non-linear Navier-stokes equations are solved. In both case Crouzeix Raviart elements with discontinuous pressure are considered. The pressure and velocity are solved separately by the penalty function method as described n the manual Standard Problems.
This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex stokes
```

for the first case or

```
sepgetex navstokes
```

for the second case.
Running this problem is the same as in Section 2.1.

2.6.1 Stokes problem

The mesh file for this problem is trivial and will not be repeated here. The only special thing is that we use quadratic triangles, which is activated by the statement

```
   type_elements = quadratic     # quadratic triangles are used
```

The input file for sepcomp reads

```
#  stokes.prb
#
#  Flow for straight channel (linear stokes equation)
#
# Definition of constants:

set warn off   ! suppress warnings
```

```
constants
   reals
      eta   = 0.01  # viscosity
      rho   = 1     # density
      eps   = 1e-6  # penalty parameter
end

#  problem definition

problem

    navstokes_penalty       # Type number for Navier-Stokes, without swirl
                            # 6-point triangle
                            # Approximation 7-point extended triangle
                            # Penalty function method
   essential_boundary_conditions
      degfd2,curves (c1, c2) # Symmetry boundary and Outflow (v-component 0)
      curves (c3, c4)        # Fixed upper wall and Inflow

end

#  structure of the main program

structure

  # Define structure of the matrix

   matrix_structure: symmetric     # matrix is symmetric

  # Define the essential boundary conditions on c4 (only the u-component)

   prescribe_boundary_conditions velocity = 1, curves (c4), degfd1

  # Create matrix and solve system of linear equations

   penalty = eps
   density = rho
   viscosity = eta

   solve_linear_system velocity

   # compute pressure and stream function

   pressure = derivatives(velocity,icheld=7)
   psi = stream_function (velocity)

   # output

   plot_vector velocity
   plot_contour pressure          # Contour plot of pressure
   plot_contour psi               # Contour plot of stream function
   no_output

end
end_of_sepran_input
```

New in this example are the problem block:

```
   navstokes_penalty
   essential_boundary_conditions
      degfd2,curves (c1, c2) # Symmetry boundary and Outflow (v-component 0)
      curves (c3, c4)        # Fixed upper wall and Inflow
```

The first item indicates that the (Navier-) Stokes equations are solved with a penalty function formulation. Hence we have 2 degrees of freedom per node (the u and v velocity components).
For the curves C1 and C2 only the second component is prescribed, hence `degfd2`. For the curves C3 and C4 both components are prescribed.
The part

```
   penalty = eps
   density = rho
   viscosity = eta
```

defines the 3 coefficients that are required. Since no model is given, automatically the Stokes equations are solved.

```
   pressure = derivatives(velocity,icheld=7)
   psi = stream_function (velocity)
```

Thes statements compute the pressure as a derived quantity (see manual Standard Problems) and the stream function.

2.6.2 Navier-Stokes problem

The mesh file is identical to the Stokes problem.
The input file for sepcomp reads:

```
#  navstokes.prb
#
#  Flow for straight channel (non-linear stokes equation)
#
# Definition of constants:

set warn off   ! suppress warnings
constants
   reals
      eta   = 0.01  # viscosity
      rho   = 1     # density
      eps   = 1e-6  # penalty parameter
   int_variables
      iter          # define integer only for printing
end

#  problem definition

problem

    navstokes_penalty       # Type number for Navier-Stokes, without swirl
                            # 6-point triangle
                            # Approximation 7-point extended triangle
                            # Penalty function method
   essential_boundary_conditions
      degfd2,curves (c1, c2) # Symmetry boundary and Outflow (v-component 0)
      curves (c3, c4)        # Fixed upper wall and Inflow

end

#  structure of the main program

structure

  # Default matrix structure (asymmetric)

  # Define the essential boundary conditions on c4 (only the u-component)

   prescribe_boundary_conditions velocity = 1, curves (c4), degfd1

  # Create matrix and solve system of linear equations

   penalty = eps
   density = rho
   viscosity = eta
   linearization = 'none'   ! first iteration: Stokes

#  Non-linear iteration

   velocity1 = velocity     ! copy of velocity (contains boundary conditions)

   iter = 0
   diff = 1
   print 'iteration  ||u_n-u_n-1||'
   while ( diff>1e-4 and iter<10 ) do

      iter = iter+1

      solve_linear_system velocity1        ! compute velocity1

      diff = inf_norm(velocity1-velocity)  ! compute difference
      print iter, diff

      velocity = velocity1                 ! copy result in velocity

     # adapt iteration method for next iterations

      if ( iter==1 ) then
         linearization = 'picard'  ! second iteration: Picard
      end_if  ! ( iter==1 )

      if ( iter==2 ) then
         linearization = 'newton'  ! rest of iterations: Newton
      end_if  ! ( iter==1 )

   end_while  ! ( diff> 1e-4 and iter<10 )

   # compute pressure and stream function
   pressure = derivatives(velocity,icheld=7)
   psi = stream_function (velocity)

   # output

   plot_vector velocity
   plot_contour pressure          # Contour plot of pressure
   plot_contour psi               # Contour plot of stream function
   no_output

end
end_of_sepran_input
```

A new item in the constants block is

```
   int_variables
      iter          # define integer only for printing
```

This defines `iter` as an integer variable. If these statements are omitted iter is considered to be real. This is not a problem but for the printing it is nicer to have an integer.
The coefficients are extended by

```
   linearization = 'none'   ! first iteration: Stokes
```

which means that we start without linearization. In other words the first iteration is Stokes. The quotes around none are necessary, since this is a string variable.
The non-linear iteration could be performed with

```
   solve_nonlinear_system velocity
```

but this requires an extra input block. Also the method used here gives more insight in the nonlinear iteration.

```
   iter = 0
   diff = 1
   print 'iteration  ||u_n-u_n-1||'
   while ( diff>1e-4 and iter<10 ) do

      iter = iter+1

      solve_linear_system velocity1        ! compute velocity1

      diff = inf_norm(velocity1-velocity)  ! compute difference
      print iter, diff

      velocity = velocity1                 ! copy result in velocity

     # adapt iteration method for next iterations

      if ( iter==1 ) then
         linearization = 'picard'  ! second iteration: Picard
      end_if  ! ( iter==1 )

      if ( iter==2 ) then
         linearization = 'newton'  ! rest of iterations: Newton
      end_if  ! ( iter==1 )

   end_while  ! ( diff> 1e-4 and iter<10 )
```

We start by setting the variables `iter` and `diff` and use a while statement for the non-linear iteration. The porcess stops when either `iter` is more than the maximum of 10 or if the accuracy of 10^{-4} is reached. The solution of the linear system is stored in velocity1 to make it possible to compare the result of two succeeding iterations. The difference is computed in the inf-norm, which is the maximum over all components. After iteration 1 the linearization is set to Picard and after the next iteration to Newton, which is used in all other iterations. here we have used a `if then` construction. `else` has not yet been implemented. The first step is just to be sure that the iteration is close enough to the final solution, so that Newton converges. In fact for this specific problem this is superfluous.

2.7 Stationary isothermal laminar Newtonian flow in a T-shaped region

In this example we consider the Newtonian flow in a channel in a t-configuration (Cartesian coordinates). See Figure 2.7.1 for a definition of the domain.

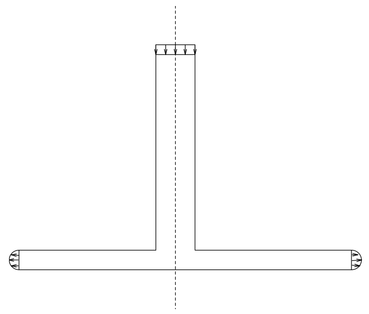


Figure 2.7.1: T-shaped region for the flow problem

The instream velocity is supposed to be uniform. Also the points at the fixed walls in the instream region get the same uniform velocity. For the fixed walls the no-slip condition is assumed. At the outflow we suppose a zero tangential velocity component and a zero normal stress tensor, implying a zero pressure at the outstream boundary.

Because of the symmetry of the configuration only one half of the region is considered. The discretization is performed with quadratic triangles (type number 900). The geometry is described by the points P_1 to P_8, the curves C_1 to C_{10} and the surfaces S_1 to S_3, see Figure 2.7.2.

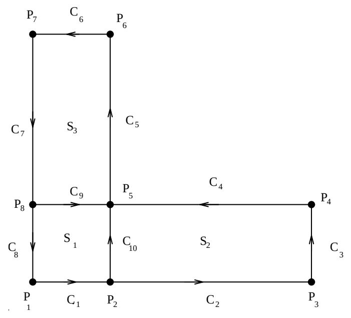


Figure 2.7.2: Geometry definition for the flow problem

The boundary conditions can be formulated as:

fixed walls: $C_1,\ C_2,\ C_4,\ C_5$: $\mathbf{v}\ =\ \mathbf{0}$

symmetry axis: $C_7,\ C_8$: $v_n\ =\ 0,\quad \sigma_t\ =\ 0$

instream flow: C_6: $v_n\ =\ -1,\quad v_t\ =\ 0$

outstream flow: C_3: $v_t\ =\ 0,\quad \sigma_n\ =\ \mathbf{0}$

The viscosity η is chosen equal to 0.01; the density ρ to 1 and the penalty parameter ϵ equal to 10^{-6}.
This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex navstokes1
```

The mesh file for this problem is standard and will not be shown here.

In SEPCOMP we use `solve_nonlinear_system velocity` instead of the while loop of example of 2.6.2. This implies that we have to add several extra input blocks. The first one describes the input for the non-linear solver given by

```
nonlinear_equations
   global_options, maxiter=10, accuracy=1e-4,print_level=1
   equation 1
      change_coefficients
         at_iteration 2, sequence_number 1
         at_iteration 3, sequence_number 2
end
```

The global options speak for them selves.
If you want to use different linearization methods like in Section 2.6.2, it is necessary to use the combination `equation 1` followed by `change_coefficients`, although in this case there is only 1 equation to be solved. In iteration 2 we use the coefficients that are indicated by change coefficients with sequence number 1, and from iteration 3 those indicated by change coefficients with sequence number 2. Using change coefficients implies that we have to give an input block coefficients for the first iteration. The extra blocks are given by:

```
coefficients   # Input for iteration 1 (Stokes)
   penalty  = eps           # 6:  Penalty function parameter eps
   density  = rho           # 7:  Density
   viscosity = etha         #12:  Value of etha (viscosity)
end

change coefficients, sequence_number = 1   # Input for iteration 2
      linearization = picard   # 5:  Type of linearization (Picard iteration)
end

change coefficients, sequence_number = 2   # Input for iteration 3
      linearization = Newton   # 5:  Type of linearization (Newton iteration)
end
```

2.8 The hole-in-plate problem (example of plane stress)

Consider the plate in Figure 2.8.1.

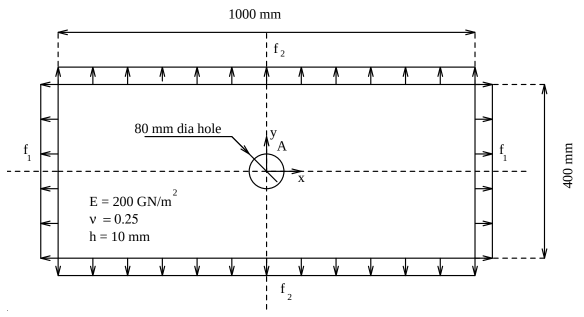


Figure 2.8.1: The hole-in-plate problem

For symmetry reasons it is sufficient to discretize only one quarter of the plate. The problem is solved by linear elements. For the generation of the mesh we define the 6 user points, and 5 curves. The definition of user points and curves is given in Figure 2.8.2

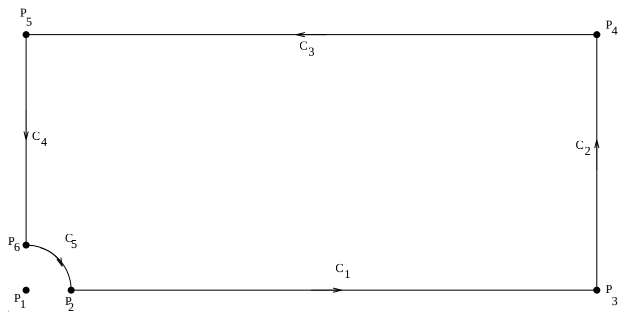


Figure 2.8.2: Definition of user points and curves

The following parameters are used:

Young's modulus is equal to 2×10^{11} N/m^2, Poisson's ratio is equal to 0.25 and the plate thickness is 0.01m.

The boundary loads f_1 and f_2 are given by:

$f_1 \;=\; 2 \times 10^6$ N/m^2 $f_2 \;=\; 10^6$ N/m^2

Essential boundary conditions:

symmetry axis: C_1: $v = 0$ C_4: $u = 0$

This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex plathole
```

The input file for sepmesh is quite standard. New is the use of an arc (in this case a quarter of a circle)

```
      c5=arc (p6,p2, -p1)    ! hole (circle with center p1)
```

This concerns a circle with center P1, starting at user point P6 and ending in P1. The minus sign before P1 indicates that the direction is clockwise.

The following parts are new in the input file for sepcomp

```
problem

   elasticity         # Linear elasticity problem
```

This part of problem indicates that a linear elastic problem must be solved.

```
   elastic_modulus = E
   poissons_ratio = nu
   Thickness = h
   solve_linear_system displacement
```

defines the parameters for the elasticity equation. Since the natural boundary conditions are different on both sides we need two boundary groups and also a separate input block coefficients:

```
coefficients
   bngrp1
     x_load = Tx
   bngrp2
     y_load = Ty
end
```

So in group 1 we have an `y_load` zero but in group 2 `x_load` is zero.

2.9 Natural convection in a square cavity

Consider the same square as in Figure 2.1.1. Inside we have a fluid with constant temperature that in first instance is at rest. Next we heat one of the sides, so we get a temperature difference. Due to the temperature gradient a circular flow will arise. In this example we consider the stationary case. Under certain physical conditions the velocity, temperature and pressure satisfy the Boussinesq equations. It is possible to solve these coupled equations as one set of non-linear equations, but in this example we shall solve the momentum equations (Navier-Stokes) and the temperature equation in a decoupled way in a non-linear iteration process.
So we have an iteration process, where we first solve the Navier-Stokes equations with a body force defined by $\rho g \beta (T - T_0)$, with ρ the density g the acceleration of gravity, β the volume expansion coefficient and T_0 a reference temperature, which we define to be zero in this example.
Next we solve the convection diffusion equation for the temperature using the just computed velocity. This process is repeated until convergence.
The whole problem is made dimensionless with two dimensionless numbers Prandtl and Rayleigh. The left and right walls are kept at a constant temperature. The left-hand wall gets the temperature T=1, the right-hand side wall: T=0. The lower wall and upper wall are assumed to be insulated. The velocity satisfies no-slip conditions.

This example can be found in the directory `$SPHOME/sourceexam/intro` and it can be copied to a local directory by

```
sepgetex boussdecop
```

The mesh is finer at the left- and right-hand sides. This is done using the concept of coarseness. In the vertical direction we use an equidistant subdivision by defining the number of elements. The mesh input file will not be repeated here.
The input file for sepcomp is given completely:

```
#  boussdecop.prb
#
#  Square cavity, natural convection, decoupled approach
#
# Definition of constants:

set warn off      ! suppress warnings

constants
   reals
      eta  = 1            # viscosity
      rho   = 1           # density
      eps   = 1e-8        # penalty parameter
      g  = 9.81           # acceleration of gravity
      rayleigh = 1e3      # rayleigh number
      prandtl  = .71      # Prandtl number
      beta  = rayleigh/(g*prandtl)  # Volume expansion coefficient
      fy = rho*g*beta     # coefficient in rhs
      cp    = 1           # Heat capacity
      kappa = 1/prandtl   # thermal conductivity
   int_variables
      iter          # define integer only for printing
end
#
#  Definition of the problem
#
problem 1                            # Velocity
```

```
   navstokes_penalty                # Type number for Navier-Stokes
   essential_boundary_conditions
      curves (c1 to c4)             # All velocities are prescribed
problem 2                           # Temperature
   convection_diffusion             # Type number for convection-diffusion
   essential_boundary_conditions
      curves (c2,c4)                # The temperature at walls c2 and c4
                                    # is prescribed
end

# Define structure of main program

structure

#  Initial conditions

   create_vector velocity, problem = 1
   create_vector temperature = 1 - x_coor, problem = 2


#  Non-linear iteration

   velocity1 = velocity        ! copy of velocity (contains boundary conditions)
   temperature1 = temperature

   # coefficients

   model = 'newtonian'          # type of constitutive equation
   linearization = 'newton'     # type of linearization of convection
   penalty = eps                # penalty function parameter
                                # The pressure is of order 1000
   density = rho                # density of fluid
   viscosity = eta
   diffusion = kappa            # thermal conductivity (1/prandtl)

   iter = 0
   diff = 1
   print 'iteration  ||u_n-u_n-1||  ||T_n-T_n-1||'
   while ( diff>1e-3 and iter<10 ) do

      iter = iter+1

      y_force = fy*temperature
      solve_linear_system velocity1        ! compute velocity1

      diff1 = inf_norm(velocity1-velocity) ! compute difference
      velocity = velocity1                 ! copy result in velocity

      solve_linear_system temperature1     ! compute temperature1

      diff2 = inf_norm(temperature1-temperature) ! compute difference

      diff = max(diff1,diff2)
      print iter, diff1, diff2
      temperature = temperature1

   end_while  ! ( diff> 1e-3 )

   y_force = fy*temperature
   pressure = derivatives(velocity,icheld=7)

   # compute stream function

   stream_function = stream_function (velocity)

#  print the vectors

   print velocity, region = ( 0.4999, 0.5001, -1, 1)
   print temperature, curves (c1)
   print pressure, curves (c1)

   plot_vector velocity
   plot_contour pressure
   plot_contour stream_function
   plot_contour temperature

   plot_colored_levels stream_function
   plot_colored_levels temperature
   plot_colored_levels pressure

   no_output
end
end_of_sepran_input
```

In the block constants all (dimensionless) physical constants are given.
The problem block refers to 2 problems:
Problem 1: Navier-Stokes with penalty formulation (See Manual Standard Problems) and essential boundary conditions at all curves.
Problem 2: convection diffusion with essential boundary conditions on the vertical curves (c2 and c4).
With `create_vector` the velocity is initialized by zero and the temperature gets the value $1 - x$. Note that the problem is given to indicate what type of vector is required. This is necessary since the number of degrees of freedom differs for both problems, but also because the linear solver has to know which boundary conditions are essential.
The part coefficients initializes the coefficients that are kept constant during the iteration.
The iteration process is ruled by the while loop, like in Example 2.6.2. First the Navier-Stokes equation is solved with given force in y-direction. After that the convection diffusion equation is solved, using the just computed velocity. Iteration is stopped if the maximum difference in both velocity and temperature is less than 10^{-3} or the number of iterations is more than 10.

3 Mesh generation

3.1 General remarks

The generation of submeshes may be done by a standard submesh generator, by a user written submesh generator or by input from the standard input file. In this manual only the first possibility is treated, for the other cases see the Users Manual. Furthermore this manual is restricted to one- and two-dimensional meshes only.

The definition of the elements is performed in two stages:

- in the first stage the user defines geometrical quantities as points, curves, surfaces and volumes, and elements along these quantities,
- in the second stage elements created in the first stage are coupled to element groups. Only those elements necessary for the solution of the finite element problem must be identified with an element group.

3.1.1 Definition of points, curves, surfaces and volumes

For the generation of meshes we define the following quantities:

Points, Curves, Surfaces and Volumes

Points form the basis for all other components. The user must define the main points necessary for the generation of curves. These points must be numbered sequentially from 1 onwards. After the generation of the mesh they are connected to nodal point numbers. The corresponding nodal point numbers are generally not equal to the point numbers defined by the user.

Curves form the one-dimensional quantities of the meshes. For example lines and arcs are curves. The initial and end points of any curve must already have been defined as points. Curves have an orientation, defined by the initial and end points, hence line C3 = (P3, P4) is different from line C4 = (P4, P3).

Surfaces form the two-dimensional quantities of the mesh. The boundaries of the surfaces must already have been defined as curves. The boundary of a surface must be closed in itself, the internal part of the surface must be on the left-hand side of the curves. Hence the boundaries of a surface must be created counter clockwise and may not intersect itself. Whenever in a description of a surface a curve is needed in the opposite direction of which it was defined, then its number must be preceded by a minus sign. (See Figure 3.3). For "exotic" boundaries it may be wise to divide the region considered in a number of less "exotic" subregions since most of the SEPRAN generators will give better results in such a situation and besides also require less computation time.

Volumes form the three-dimensional quantities of the mesh. They are not defined in this manual. For three-dimensional problems the user is referred to the Users Manual.

All points, curves, surfaces and volumes must be numbered sequentially, each starting with number one. The outer and inner boundaries as defined in 2.2 must consist of points (in R^1), points and curves (in R^2), and points, curves and surfaces (in R^3).

The submeshes as defined in 2 must coincide with curves (in R^1), surfaces and sometimes curves (in R^2), or with volumes and sometimes curves and surfaces (in R^3).

Anywhere in the manuals where curves, points and surfaces are mentioned, the curves, points and surfaces generated by the mesh generator are meant. Nodal points of the mesh must be coupled with these points, curves and surfaces.

Examples

Consider the regions in Figure 3.1 and 3.2.

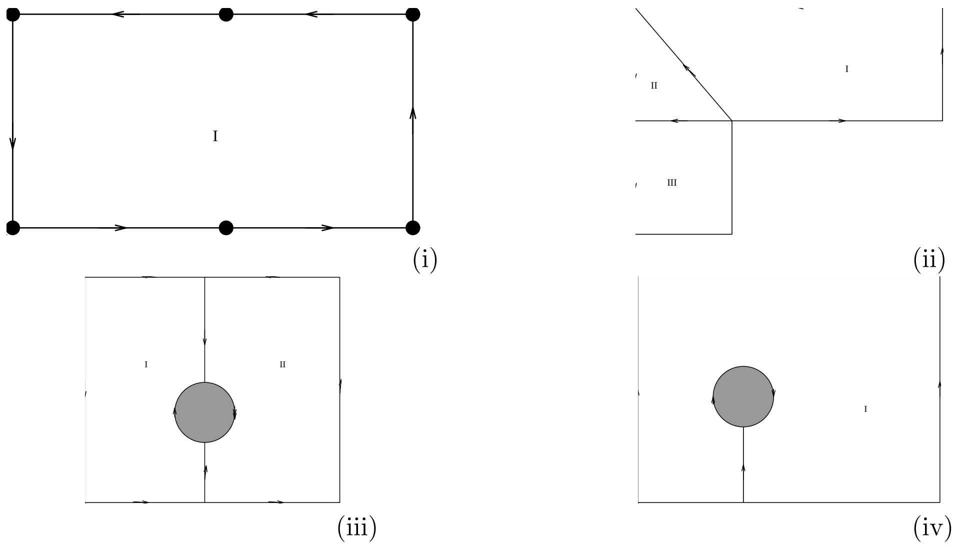


Figure 3.1: Examples of regions consisting of 1 (i),(iv) 3 (ii) and 2 (iii) subregions

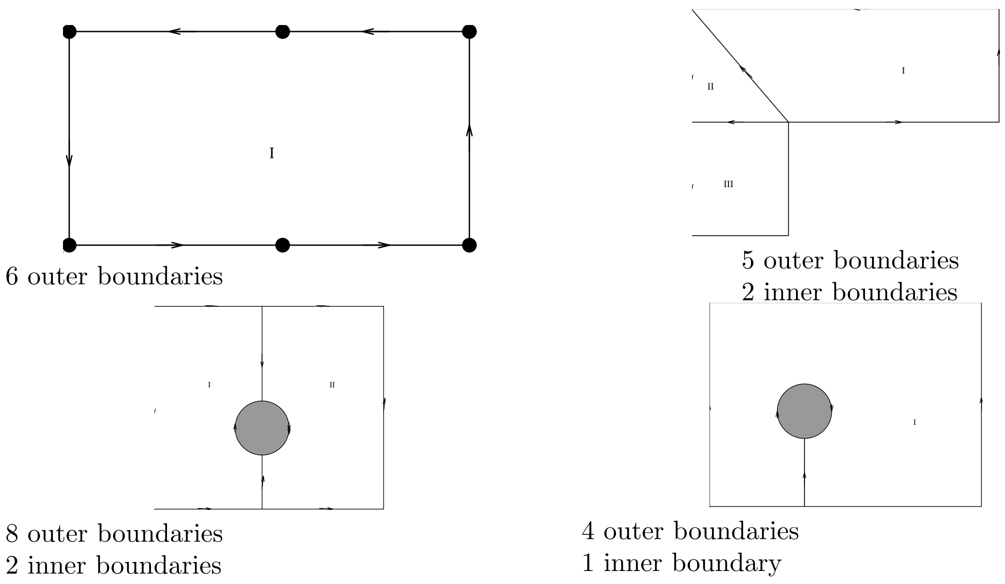
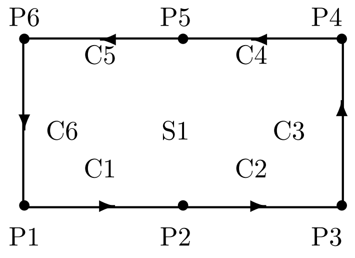


Figure 3.2: Examples of inner and outer boundaries each provided with a direction

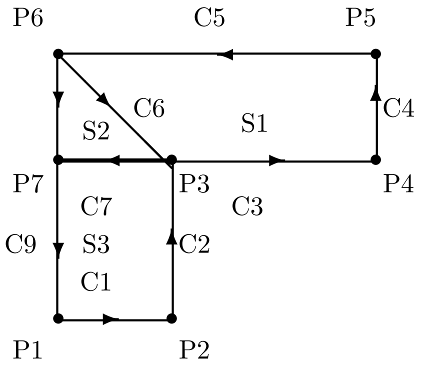
In Figure 3.3 the points, curves and surfaces for these regions are defined. Points are indicated by Pk (k=1 ,2 ,,,), curves by Cl (l=1 ,2 ,,,) and surfaces by Sm (m=1 ,2 ,,,). The corresponding commands are POINTS, CURVES and SURFACES.

Remark:
When the user wants to create double points on a line (for example for a crack), he has to introduce two curves with the same end points on this line. For example the outer boundaries 2 and 3 in Figure 3.4 must be created as 2 different curves.



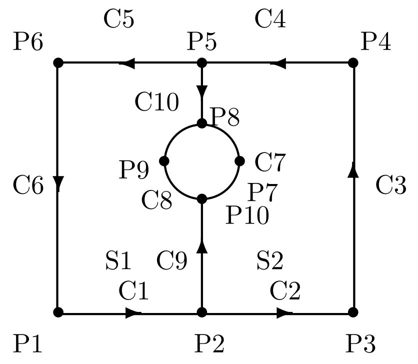
C1 = (P1,P2) C2 = (P2,P3)
C3 = (P3,P4) C4 = (P4,P5)
C5 = (P5,P6) C6 = (P6,P1)
S1:(C1,C2,C3,C4,C5,C6)

Outer boundaries: C1, C2, C3, C4, C5, C6



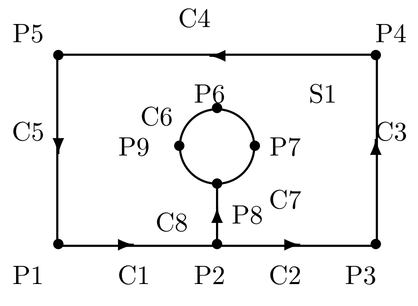
C1 = (P1,P2) C2 = (P2,P3)
C3 = (P3,P4) C4 = (P4,P5)
C5 = (P5,P6) C6 = (P6,P3)
C7 = (P3,P7) C8 = (P6,P7)
C9 = (P7,P1)
S1:(C3,C4,C5,C6)
S2:(-C7,-C6,C8)
S3:(C1,C2,C7,C9)

Outer boundaries: C1, C2, C3, C4, C5, C8, C9
Inner boundaries: C6, C7



C1 = (P1,P2) C2 = (P2,P3)
C3 = (P3,P4) C4 = (P4,P5)
C5 = (P5,P6) C6 = (P6,P1)
C7 = (P8,P7,P10) C8 = (P10,P9,P8)
C9 = (P2,P10) C10 = (P5,P8)
S1:(C1,C9,C8,-C10,C5,C6)
S2:(C2,C3,C4,C10,C7,-C9)

Outer boundaries part 1: C1, C2, C3, C4, C5, C6
Outer boundaries part 2: C7,C8
Inner boundaries: C9, C10



C1 = (P1,P2) C2 = (P2,P3)
C3 = (P3,P4) C4 = (P4,P5)
C5 = (P5,P1) C6 = (P6,P9,P8)
C7 = (P8,P7,P6) C8 = (P2,P8)
S1:(C8,-C6,-C7,-C8,C2,C3,C4,C5,C1)

Outer boundaries part 1: C1, C2, C3, C4, C5
Outer boundaries part 2: -C6, -C7
Inner boundaries: C8

Figure 3.3: Points, Curves and Surfaces

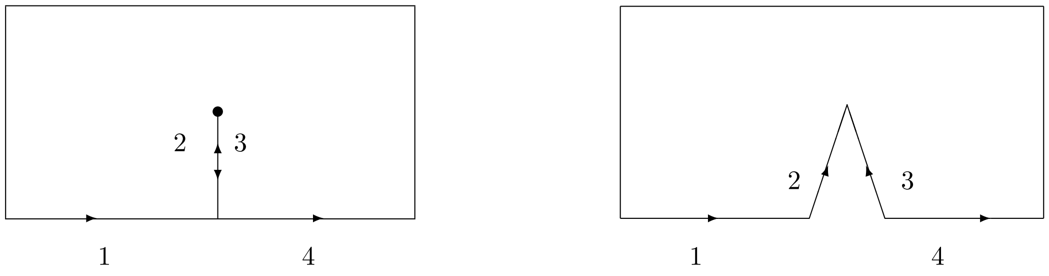


Figure 3.4: Treatment of a crack

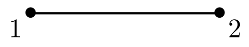
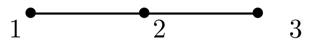
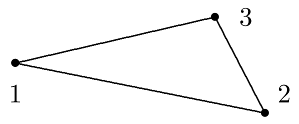
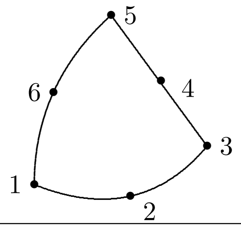
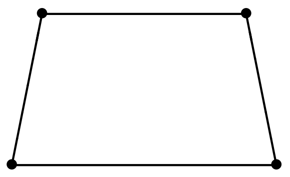
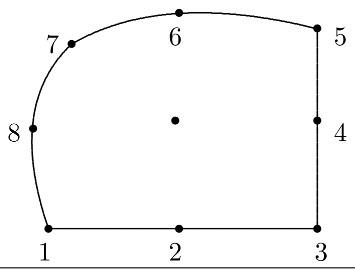
shape number	shape	name
1		line element with 2 points
2		line element with 3 points
3		triangle with 3 points
4		isoparametric triangle with 6 points
5		quadrilateral with 4 points
6		isoparametric quadrilateral with 9 points

Table 3.1: Standard elements for mesh generation

3.2 The mesh input file

In general the mesh input file consists of two input blocks:

CONSTANTS is used to define some constants that may be used in the second block. This block is not necessary but makes reading of the input more clear.

MESHxD is the actual input block for the mesh generation. The constants defined in the first block may be used in this block.

In all input files characters like `,`, `:`, `;` have no meaning and are only meant to make the text more readable. Internally they are treated as spaces.

3.2.1 The block CONSTANTS

This input block has the shape

```
CONSTANTS
   INTEGERS
      name_integer i = value
         .
         .
         .
      name_integer j = value
   REALS
      name_real i = value
         .
         .
         .
      name_real j = value
END
```

If this block is used, the initial line `CONSTANTS` and the end line `END` are obligatory. The blocks `INTEGERS` and `REALS` are meant to define integer and real constants.
The syntax for these variables is always:

```
    name = value
```

where `name` is unique name.
`value` may be a value, but also an expression.

3.2.2 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 *abs*, *sqrt*, *exp*, *cos*, *sin*, *tan*, *arcsin*, *arccos*, *arctan* and *log* (natural logarithm). In the input file the are represented by

```
  sqrt
  exp
  sin
  cos
  tan
  asin
  acos
  atan
  log
  abs
```

A typical example might be:

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

The standard FORTRAN priority rules for operators are applicable. If an expression should be continued on the next line it should be ended with the continuation character `&`.

3.2.3 Comments

Every text following a hash symbol `#` or an exclamation mark `!` is considered as comment and has no meaning for SEPRAN. Also all blank lines or lines starting with a star `*` in column 1 is treated as comment line.

3.2.4 MESHxD

The input block MESHxD consists of the following parts:

```
MESH2D
   coarse ( unix = u)
   type_elements = xxx

   points

      p i = ( x_i, y_i, c_i )
         .
         .
         .

   curves
      c i = ....
         .
         .
   surfaces
      s i = ....
         .
         .
   meshsurf

      selm 1 = ...
         .
         .
   plot

END
```

coarse (unix = u) The user may give the number of elements along a curve, but in general it is easier to use the concept of coarseness. The value of u gives the standard element length in each user point.

type_elements = xxx defines the type of elements that will be used. If different type of elements are required it is always possible to provide each curve, surface of volume with its own type. The following types are available (the numbers between brackets denote the internal shape number used)

LINEAR (1,3) All elements are linear, hence linear line elements, linear triangles and linear tetrahedrons.

QUADRATIC (2,4) All elements are quadratic.

TRIANGLE (1,3) is identical to `linear`

QUADRILATERAL (1,5) The elements are bi-linear quadrilaterals.

LINEAR_TRIANGLE (1,3) is identical to `linear`

QUADRATIC_TRIANGLE (2,4) is identical to `quadratic`

LINEAR_QUADRILATERAL (1,5) is identical to `quadrilateral`

QUADRATIC_QUADRILATERAL (2,6) The elements are bi-quadratic quadrilaterals.

The default value is linear.

points is the start of the definition of the user points. See Section 3.3

curves is the start of the definition of the curves. See Section 3.4

surfaces is the start of the definition of the surfaces. See Section 3.5

meshsurf is only necessary if more than one element group. is required. It connects physical quantities like curves and surfaces to element groups. See Section 3.6

plot

3.3 The subkeyword points

The keyword points must be followed by records defining the various user points. The general structure is

```
p i = ( x_i, y_i, c_i )         (2D)
   or
p i = ( x_i, y_i, z_i, c_i )    (3D)
```

where i is the user point sequence number (> 0), x_i and y_i are the coordinates.
c_i is only used in case of coarseness. It defines a multiplication factor to define the local element width in the neighborhood of the user point as $u \times c_i$, where u is the global coarseness defined in the coarse statement.
The default value for c_i is 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 $Pi = \dots$ 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

PDi $= (r_i, \phi_i, z_i)$, with ϕ in degrees or

PRi $= (r_i, \phi_i, z_i)$, with ϕ in radians

instead of Pi $= (x_i, y_i, z_i)$.

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

3.4 The subkeyword curves

SEPRAN contains a number of curve generators, but in this section only a few are described. For a complete list the reader is referred to the User's Manual Section 2.3.
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 $c\times$ 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.
The following curve generators are treated here:

```
line shape_cur ( pi, pj, pk, pl, options )
arc shape_cur ( p1, p2, p3, options )
translate ck ( P1, ... )
curves ( ci, cj, ck, ... )
```

`shape_cur` defines the shape of the elements along the curve. If omitted the value given by `type_elements` is used or the default value.
`shape_cur` may have the value `linear` (1) or `quadratic` (2), generating linear or quadratic elements. The digits between brackets are the alternatives for the texts and are used internally.

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.

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.

The curve generators have the following global functions:

LINE generates a straight line between two end points. If the option COARSE is used, more than two user points may be given. In that case the curve consists of a series of straight lines between successive points. The user points are always end points of elements and the local element length around these points is defined by the local coarseness.

ARC generates an arc from begin point (P1) to end point (P2). The centroid is defined by 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 3.1. Of course this is only possible when initial point and end point of the arc are essentially different points.

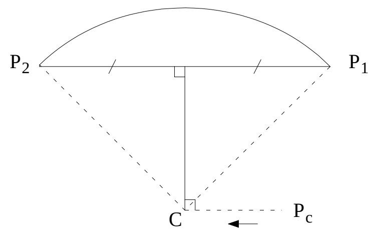


Figure 3.1: Computation of the centroid of an arc

TRANSLATE Copy a curve (ck) and translate it over a fixed distance. When TRANSLATE is used, the curve Ci is a copy of curve Cj translated over a distance d = $((P1_i - P1_j)_x, (P1_i - P1_j)_y, (P1_i - P1_j)_z)$ with $P1_i$ the first point on Ci and $P1_j$ the first point on Cj.
If the points $Pi, Pj, Pk, ...$ are given, these points correspond to the second, third etc. user points on Cj 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 Ci. The point numbers i of Pi 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

TRANSLATE Cj ($P1, -P5$)

indicates that the begin point on curve Ci is the user point $P1$ and the end point is user point

$P5$. 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.

CURVES Create a new curve by combining old 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, $Cl, Cm, ...$ 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 Ci must be larger than Ck, Cl, and Cm.

3.5 The subkeyword surfaces

SEPRAN contains a number of curve generators, but in this section only a few are described. For a complete list the reader is referred to the Users Manual Section 2.4.
The following is a list of generators described in this manual.

```
  Si = GENERAL shape_sur ( C1, C2, C3, . . . )
  Si = TRIANGLE shape_sur ( C1, C2, C3, . . . )
  Si = QUADRILATERAL shape_sur (C1, C2, C3, C4 )
```

`shape_sur` defines the shape of the elements along the curve. If omitted the value given by `type_elements` is used or the default value. The options for `shape_sur` are exactly the same as described for `type_elements` in Section 3.2.4 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.
Si is the surface number, and C1, C2, . . . the curves enclosing Si.
For more information the user is referred to the Users Manual Section 2.4.1.

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.
Another advantage of triangle is that it allows holes in the surface. This is done by describing first the curves enclosing the surface and then those enclosing the first hole and so on.
For more information the user is referred to the Users Manual Section 2.4.7.

QUADRILATERAL 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 has 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.
For more information the user is referred to the Users Manual Section 2.4.3.

3.6 The subkeyword meshsurf

The keyword MESHSURF connects surfaces to element groups. In the same way MESHLINE connects curves to element groups. The syntax is:

```
meshsurf
   selm1 = surfaces ( si to sj)
   selm2 = surfaces ( sk to sl)
      .
      .
      .
```

So element group 1 corresponds to all elements in the surfaces Si to Sj. If only Si is meant to Sj may be skipped. In the same way element group 2 corresponds to all elements in the surfaces Sk to Sl and so on.
If for example element group 3 corresponds to surfaces S1 and S4 one should use:

```
   selm3 = surfaces (s1)
   selm3 = surfaces (s4)
```

Hence the same element group number may be repeated.

4 The computational part of SEPRAN

In the computational part of SEPRAN the solution is computed. In fact SEPRAN has been developed such that the computational part may contain also the preprocessing parts as well as post-processing. In simple cases (at this moment the straight-forward solution of a linear or non-linear problem) the computational part may be carried out by program SEPCOMP. If SEPCOMP has insufficient flexibility then it is necessary to create your own main program. SEPRAN offers a large number of tools for this purpose. The most simple ones are treated in the SEPRAN users manual, for the extended case the reader is referred to the programmers guide.

In Section 1.4 it has been described how to call SEPCOMP. In this chapter the input and output possibilities of SEPCOMP for the most simple case are described. For more complex possibilities of SEPCOMP the reader is referred to the users manual.
Section 4.1 describes the input for program SEPCOMP.

4.1 The sepcomp input file

The input file for sepcomp consists of a number of input blocks. In this manual we describe only a few of the possible blocks and for each block only a limited part of the possibilities. For a complete description the user is referred to the Users Manual. It concerns the following input blocks:

CONSTANTS is used to define some constants that may be used in the second block. This block is not necessary but makes reading of the input more clear.

PROBLEM defines the problem to be solved i.e. the type of differential equation per element group and the type of boundary conditions.
See Section 4.2.

STRUCTURE may be considered as the main program. It contains a list of statements that are executed consecutively.
See Section 4.3.

COEFFICIENTS is an optional block to define coefficients (parameters) for the differential equation and the natural boundary conditions.
See Section 4.4.

SOLVE is an optional block with extra information for the linear solver.
See Section 4.5.

NONLINEAR_EQUATIONS is an optional block with extra information for the non-linear solver.
See Section 4.6.

4.1.1 The block CONSTANTS

This input block has the shape

```
CONSTANTS
   INTEGERS
      name_integer i = value
         .
         .
         .
      name_integer j = value
   REALS
      name_real i = value
         .
         .
         .
      name_real j = value
   VECTOR_NAMES
      name_vector i
   VARIABLES
      name_variable i
   INT_VARIABLES
      name_variable i
END
```

If this block is used, the initial line `CONSTANTS` and the end line `END` are obligatory. The blocks `INTEGERS` and `REALS` are meant to define integer and real constants. See Section 3.2.1
The block `vector_names` gives the opportunity to declare some names as vectors. This is usually

not necessary since SEPRAN tries to detect from the context in the structure block, whether a name is a scalar or a vector. However, in case of doubt, for example if you prescribe a vector by a constant this offers you the possibility to force the name to be a vector.
In the same way the block `variables` declares a name as a scalar. In practice there is no need to this is.
The block `int_variables` is used to declare a scalar as an integer one, which means that it can only take integer values. This is only important for printing.
By default all other scalars are real.

4.2 The keyword PROBLEM

This keyword must be used before all other input blocks, but after the CONSTANTS block, see 2.1. It defines what types of differential equations should be solved and what types of boundary conditions are given at which part of the boundary.
In this manual we give only a selection of the possible input for this keyword. For a complete description the user is referred to the Users Manual.
The general structure of the input is:

```
problem 1

   types part
   boundary_elements
      boundary elements part
   essential_boundary_conditions
      essential boundary conditions part
   periodical_boundary_conditions
      periodical boundary conditions part
problem 2
   .
   .
end
```

The keyword PROBLEM starts the block and END denotes the end of the input block. Both are obligatory. The sequence number 1 is only required if the user wants to solve more than 1 problem at a time. For the second problem we start with a set of statements starting with sequence number 2 and so on.
`end` is used only once.

types part is obligatory. It describes the types of differential equations to be solved. The syntax is `elgrp i to j = type_name`. This may be repeated for each element group.
If all element groups are described by the same type of differential equation the part `elgrp i to j` may be skipped. If the differential equation relates to one element group only it is sufficient to use `elgrp i`.

`type_name` may be either the name of a known differential equation, like for example `laplace` or `type = i`, with i a number.
For available names and numbers the user is referred to the manual Standard Problems.
So one may have something like

```
convection_diffusion
```

or

```
elgrp 1 = convection_diffusion
elgrp 2 = stokes
elgrp 3 = poisson
```

or

```
elgrp1 to 2 = convection_diffusion
elgrp 3 = stokes
elgrp 4 to 5 = poisson
```

boundary_elements is only necessary in case one has non-homogeneous natural boundary conditions. In that case boundary elements are required. The type of boundary elements is detected from the corresponding differential equation. Should that not be possible because there is more than one type of differential equation, then it is necessary to add an extra part `natbouncond` as described in the Users Manual. The part `boundary elements part` gives a description of the place where we have these boundary elements. This part has the following shape

```
belm1 = points (p1, p2, ...)
belm2 to 3 = curves ( c3 to c7)
belm3 = curves( c9, c11, c15)
```

So `belm` must be followed by the boundary element group number. The construction i to j is allowed and also a boundary element group number may be repeated.
`points (p1, p2, ...)` defines point elements in the user points indicated.
`curves( c9, c11, c15)` is used for boundary elements along curves.

essential_boundary_conditions must be used if essential boundary conditions must be prescribed. It defines where there these boundary conditions are prescribed and for which degrees of freedom. It does not define the actual values. However, prescribing the values without using this sub-block has no effect at all.
The `essential boundary conditions part` has the shape

```
points (p1, p2, ...)
degfd1, curves ( c3 to c7)
degfd1, degfd3, curves( c9, c11, c15)
```

So we give the degrees of freedom (per point), followed by the position. If the degrees of freedom are omitted all degrees of freedom are prescribed.

periodical_boundary_conditions corresponds to periodical boundary conditions. The user must define the degrees of freedom in opposite points that have common values.
The `periodical boundary conditions part` has the shape

```
points (p1, p2)
degfd1, curves ( c3, c7)
degfd1, degfd3, curves( c9, -c11)
```

So always two items (user points or curves) are given. If curves have opposite directions, one of the two must be provided with a minus sign.

4.3 The keyword STRUCTURE

This block describes the sequence in which the program is executed. Although it is possible to run a program without this block, its use is strongly recommended since it ensures flexibility. Within this block the user gives to commands to carry out certain actions, like prescribing essential boundary conditions or solving a system of equations. It is also possible to give scalars and vectors a value.
In this manual only a limited number of the possible options are described. In the Users Manual a complete overview can be found.
Globally we may subdivide the contents of the structure block in the following parts

```
definition of matrix structure
prescribing of boundary conditions or creation of vectors
solving systems of equations
print and plot options
commands to construct quantities from other vectors.
commands to give scalars and vectors a value
flow control statements
```

scalars and vectors may be declared explicitly in the CONSTANTS block (3.2), but also implicitly in this block. If a quantity is a scalar it remains a scalar forever and the same is true for vectors. Whether a quantity is a scalar or a vector depends on the context. If it is obvious like the result of prescribing boundary conditions or solving a system of equations it is a vector. If a name is given a value through an expression, it depends on the right-hand side whether the result is a scalar or a vector. If the right-hand side contains vectors, the left-hand side is also a vector; otherwise it is a scalar.

4.3.1 definition of matrix structure

The definition of matrix structure is only necessary if the matrix structure differs from the default structure. If nothing is given the matrix is treated as an asymmetric matrix and a profile (envelope) storage is used, which implies that systems of equations are solved by direct methods.
Otherwise the following syntax is used

```
matrix_structure options
```

The following options are available

```
symmetric
storage_scheme = s
```

symmetric defines the matrix as symmetric matrix, which implies that approximately one half of the matrix is stored.
The default is asymmetric.

storage_scheme = *s* defines the storage scheme. *s* may be either `profile` (default) or `compact`. In the first case a direct linear solver is used, in the second case an iterative solver. Only non-zero elements of the matrix are stored for a compact matrix.

4.3.2 prescribing boundary conditions and creation of vectors

For the creation of vectors and prescribing essential boundary conditions, the following possibilities are available.

```
prescribe_boundary_conditions name = value, options
create_vector name = value, options
vector name = value, options
name = vector_expression
```

In all these options `name` is the name of the vector to be created.
`value` is either a scalar expression or a vector expression, except in the last case where it must be a vector expression. Otherwise the result is scalar, unless `name` has already been defined or declared as vector. A vector expression is an expression containing at least one vector. All operations on the vectors in the expression are carried out component-wise.

prescribe_boundary_conditions name = value, options is meant to prescribe boundary conditions for the vector `name`. If the vector is new, it is first made equal to zero, before filling the boundary conditions according to value and options.
If the vector already exists only the boundary conditions that are mentioned are filled. All other values remain unchanged. This means that more than one command `prescribe_boundary_conditions` can be given for the same vector, in order to fill different boundary conditions on different parts. Note that the sequence of filling defines the final result.
It is important to realize that name = value must follow the prescribe part immediately and that no other characters (except spaces) may be put in between. Otherwise the statement is not recognized and complete other definitions may be the result.
Several options are possible but here we treat only the following ones:

```
points = (Pi, Pj, Pk, ... )
curves = (Ci, Cj, Ck, ... )
degfd i, degfd j, ...
problem = i
```

points = (Pi, Pj, Pk, ...) prescribes the boundary conditions in the user points mentioned.

curves = (Ci, Cj, Ck, ...) prescribes the boundary conditions in the curves Ci, Cj, Ck, ...
If no points or curves are given, the whole region is used.

degfd i, degfd j, ... restricts the filling to specific degrees of freedom.
Default: all degrees of freedom.

problem = i ensures that the vector corresponds to the problem with sequence number i. This is onl important if more than one problem is used.
Default: $i = 1$

create_vector name = value, options is meant to create a vector. Actually this is the same as prescribe boundary conditions. Also the syntax is the same.

vector name = value, options is an alternative for create_vector with the same options

name = vector_expression is another alternative. Only when the right-hand side is a vector expression, name becomes a vector. Furthermore the same options as above may be used.

4.3.3 solving systems of equations

The following options for solving systems of equations are available

```
solve_linear_system name, options
solve_nonlinear_system name, options
```

where `name` is the name of the vector to be created.

solve_linear_system name, options is used to build the matrix and right-hand side and to solve the resulting system of linear equations.
Coefficients for the building may be given by scalars, constants or vectors. If this is not sufficient one can use a coefficients block.
Among the options we have

```
problem = s
seq_coef = c
seq_solve = i
```

problem $= s$ defines the problem number to which the system corresponds. This is only necessary if there are more than one problems and if the vector with name `name` has not been created before. In the latter case the problem number is extracted from the vector.

seq_coef $= c$ is only needed in case more than one coefficients block is used. c refers to the sequence number of the coefficients block (4.4).

seq_solve $= i$ is used to refer to a solve input block (4.5). This is also only necessary in case of more than one block.

solve_nonlinear_system name, options has the same meaning as linear, but besides that, an iteration process is used to converge to the final solution.
The next set of options may be used.

```
problem = s
maxiter = m
seq_coef = i
accuracy = eps
sequence_number = c
```

For more options consult the Users Manual

problem $= s$ has the same meaning as for the linear system

maxiter $= m$ defines the maximum number of iterations

seq_coef $= i$ refers to the input block for the coefficients (4.4)

accuracy $= \epsilon$ the iteration is stopped when the difference between two successive iterations is less than ϵ.

sequence_number $= c$ refers to an input block nonlinear_equations (4.6)

An alternative for non-linear equations is to program the iteration yourself like in Example (2.6.2). This enhances the flexibility.

4.3.4 print and plot options

Below follows a selection of print and plot options.

```
print items
plot_vector name, options
plot_contour name, options
plot_colored_levels name, options
plot_function name, options
```

`name` is as usual the name of the vector to be plotted.

print items is used to print constants, scalars, vectors and strings. In case of scalars and constants a series of quantities may be printed.

plot_vector name, options makes a vector plot. Hence the vector `name` must have at least two degrees of freedom per node. If necessary degfdi and degfdj define the two components to be used fore the vector plot.

plot_contour name, options makes a contour plot. This means that lines with constant value are plotted. The following options may be of interest

```
degfd i
nlevels = n
minlevel = m1
maxlevel = m2
text = '..'
```

degfd i defines the degree of freedom to be used (default 1).

nlevels = n gives the number of contour lines (default 10).

minlevel = m1 minimum level.

maxlevel = m2 maximum level.

text = '..' identification text.

plot_colored_levels name, options fills the parts between lines with constant value with one specific color. Colors are changed gradually. The same options as for plot_contour are available.

plot_function name, options plots a one-dimensional vector or a vector along a set of curves. Options:

```
degfd i
curves ( cj1, cj2, ... )
textx
texty
```

degfd i defines the degree of freedom to be used (default 1).

curves (cj1, cj2, ...) defines the curves (2D) that are used to define the function. The arc length along the curves is used for the horizontal axis.

textx text to be plotted along the horizontal axis.
Default: either x (1D) or the curve numbers (2D).

texty text to be plotted along the vertictal axis.
Default: the name of the vector.

4.3.5 construct quantities from other vectors

Within the structure block there are a number of possibilities to compute quantities from vectors. Here we shall give a selection. A complete overview can be found in the Users Manual.

```
V = derivatives(name,options)
V = stream_function (name)
V = flux(name)
V = gradient(name)
V = stress_tensor ( name)
S = integral ( name )
```

with `name` the name of the vector from which the result must be computed. V denotes that the result is a vector and S that the it is scalar.

derivatives(name,options) computes a derived quantity from name. Which quantity is computed is defined by the option `icheld = i`, where the meaning of i depends on the type of differential equation. This can be found in the manual Standard Problems

stream_function (name) computes the stream function from an incompressible vector field in 2D. Note that the vector needs at least two degrees of freedom per node.

flux(name) computes the flux from a vector. The definition of flux depends on the differential equation.

gradient(name) gives the gradient of a vector.

stress_tensor (name) computes the stress tensor from a displacement vector.

integral (name) computes the integral over the complete domain of the vector `name`.

4.3.6 creating scalars and vectors

The creation of scalars and vectors has been shown in the previous sections. A scalar may be the result of an integral or some other specific function. Also it may set by `scalar_name = value`. Value may be a number, but also an expression containing other scalars or constants.
A vector may be the result of computation such as in Sections 4.3.2 to 4.3.5.

4.3.7 flow control statements

The following flow control statements may be used in the structure block.

```
while ( boolean_expression ) do
   statements
end_while

if ( boolean_expression ) then
   statements
end_if

for variable = a to b step c
   statements
end_for
```

`boolean_expression` is a standard boolean expression consisting of numbers, constants, variables and the operators `<`, `<=`, `>`, `>=`, `/=`, `==`, `and`, `or` and `not`.
The while, if and for statements have their usual meaning. If `variable` in the for statement is a non-existing name it is made an integer scalar. Otherwise it must be a scalar not a constant.

4.4 The keyword COEFFICIENTS

The input block COEFFICIENTS is only necessary if coefficients can not be given in the structure block. This is for example the case when coefficients are different for different element groups, or boundary groups in case of natural boundary conditions. Only coefficients that are not given in the structure block have to be given in this block.
The general structure of the input is:

```
coefficients, sequence_number = i
   elgrp i
      name_of_coefficient = value
         .
         .
   elgrp j
      ...
   bngrp k
      ...
   bngrp l
      ...
end
```

coefficients, sequence_number = i start of the coefficients block. The sequence number is only necessary in case more than one coefficients block is present. It is used to distinguish between the several blocks.

elgrp i defines the start of input for element group i. All input until a next elgrp, bngrp or end is considered to correspond to this group.

bngrp k has the same meaning as elgrp but now with respect to the boundary elements.

name_of_coefficient = value defines the value of `name_of_coefficient`. value may be a number, constant, scalar or vector. For other possibilities the user is referred to the Users Manual. Which names of coefficients are recognized, depends on the differential equation. Information can be found in the manual Standard Problems.

end is the obligatory end of the block.

4.5 The keyword SOLVE

In case the linear solver needs extra information an input block SOLVE is required or special commands in the structure block must be used.

Among the commands in the structure block we have the following:

```
   POSITIVE_DEFINITE
   SOL_ITER_METHOD = name_of_iterative_solver
   SOL_EPS = v
   SOL_PRINT = i
   SOL_PREC = name_of_preconditioner
   SOL_MAXITER = i
```

Note that the keywords are case insensitive.

positive_definite is only used in case of a direct solution method. The effect is that Cholesky decomposition is used and as a consequence it is checked if the matrix is positive definite.

SOL_ITER_METHOD = name_of_iterative_solver is only used in case of an iterative solver. It defines the type of iterative solver to be used. The value of `name_of_iterative_solver` must be a text between quotes.
The following values for text are possible:

cg The cg method is used (symmetric positive definite matrices)

cgs The cgs method is used

bicgstab The bi-cgstab method is applied

gmres gmres method

idr idr method

The default is `SOL_ITER_METHOD = 'CG'`

SOL_EPS = v defines the accuracy. The default value is v = 1e-3.

SOL_PRINT = i defines the amount of output to be printed by the solver. The larger the number the more output is given.
The default value is i = 0.

SOL_PREC = name_of_preconditioner defines the name of the preconditioner to be used, which must be given as text between quotes. Possible values are `'NONE'` and `'ILU'` for respectively no preconditioner or the ILU preconditioner. The last one is the default.

SOL_MAXITER = i defines the maximum number of iterations to be performed. If this number is reached the iteration halts.

An alternative for use of above commands is to add one or more input blocks solve.
The structure of the input block is

```
solve, sequence_number = i
   positive_definite
   iteration_method = j, options
end
```

solve, sequence_number = i start of the solve block. The sequence number is only necessary in case more than one solve block is present. It is used to distinguish between the several blocks.

positive_definite has the same meaning as above.
Possible values for j are the same as for the command `SOL\_ITER\_METHOD`, however, now the quotes must be omitted.
A selection of the options is

```
accuracy = eps
print_level = p
preconditioning = l
max_iter = m
```

with the following meaning

accuracy = ϵ defines the accuracy for which the iteration is stopped.

print_level = p defines the amount of output (minimum 0, maximum 2)

preconditioning = l gives the type of preconditioner. Possible values are `none` default and `ilu`.

max_iter = m maximum number of iterations.

end is the obligatory end of the block.

For a complete list of options consult the Users Manual.

4.6 The keyword NONLINEAR_EQUATIONS

In case the non-linear solver needs extra information an input block NONLINEAR_EQUATIONS is required. The structure of the input is

```
nonlinear_equations, sequence_number = i
   number_of_coupled_equations = n
   global_options, options
   equation 1
      local_options, options
      change_coefficients
         at_iteration i, sequence_number = k
   equation 2
      local_options, options
end
```

nonlinear_equations, sequence_number = i start of the block. The sequence number is only necessary in case more than one nonlinear equations block is present. It is used to distinguish between the several blocks.

number_of_coupled_equations = n defines the number of equations that is coupled. If omitted n is equal to the number of problems defined in the PROBLEM block.

global_options, options defines options that are used for all equations.
A selection of the list of options is

```
maxiter = m      (Default 20)
miniter = m      (Default 2)
accuracy = eps   (Default 1d-3)
print_level = p  (Default 0)
```

maxiter = m maximum number of iterations (Default 20)

miniter = m minimum number of iterations (Default 2)

accuracy = ϵ The iteration is stopped if the difference between two succeeding iterations is less than ϵ. (Default 1e-3)

print_level = p defines the amount of output (minimum 0, maximum 2).

equation j gives the start of a new equation. If there are local options, for each equation you have to define this line.

local_options, options options for the specific equation. The same input as for global options is allowed.

change_coefficients implies that at certain iterations the coefficients must be changed, like for example the type of linearization. This part must be followed by

at_iteration i**, sequence_number** k , which defines what input block for change coefficients is used from iteration i. See Section 2.7 for an example.
The input for change coefficients is the same as for coefficients, see 4.4.

end end of input block

5 Index