SEPRAN

SEPRA ANALYSIS

Lab manual

GUUS SEGAL

Manual SEPRAN for the numerical analysis lab at TUD

August 2014

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

Contents

1   Introduction

The aim of this lab 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.
In Section (1.1) we give a summary of the tasks the student has to perform during a lab session. Read this section carefully and reread it before going to a next step.
Chapter 2 Gives an example of a complete SEPRAN run. It is advised to do this first, in order to get acquainted with the package.
Chapter 3 gives an introduction to the mesh generation, the computational part is treated in Chapter 4.

How to use this manual?

In order to get a quick start it is recommended to proceed as follows:

- Read Section 1.1.

- Read Chapter 2 to have an impression of what is required.
  Check if your problem is similar to one of these examples.

- If you need extra information about the mesh generation consult Chapter 3.

- If you need extra information concerning the computational part, consult Chapter 4.

The complete set of SEPRAN manuals can be found in// `http://ta.twi.tudelft.nl/sepran`

1.1   Important: Summary of a lab session

The student has to perform the following tasks:

1. Create an input file for the mesh generator, like in the Example (3.2). Information about the mesh generator can be found in Chapter 3.1.

2. Next run program sepmesh in the following way:

   ```
   sepmesh inputfile.msh
   ```

   where `inputfile.msh` is the name of the input file you created.

3. If sepmesh does not fail, view the mesh by

   ```
   sepview sepplot.001
   ```

4. If the mesh is correct then you have to create a subroutine `elemsubr.f90` in which you program the element matrix and element vector for each element group.
   The rules for this subroutine can be found in Section 4.7.1. Examples are in Chapter 4.
   Before programming the subroutine you have to copy a template into your local directory by the command:

   ```
   sepgetlab elemsubr
   ```

   Program your element matrix and element vector in this template. If you retype the source of the subroutine you might make errors, which are generally hard to find.
   Read Section 2.3.3 carefully.
   Remark
   Although SEPRAN contains preprogrammed elements for almost all lab exercises, the student is not allowed to use these. One has to program element matrix and vector himself!

5. Next you have to get the main program into your local directory by the command:

   ```
   sepgetlab sepcompexe
   ```

6. After that you have to create an input file for the program sepcompexe. See for example Sections (4.2) and (4.6).

7. If all three files are available perform the following steps:

   ```
   compile elemsubr.f90
   seplink sepcompexe
   sepcompexe < inputfile.prb > outputfile
   ```

   Here `inputfile.prb` is the name of the just created input file and `outputfile` the name of an output file. Each step must be checked first before continuing.
   Also check your output file for error messages.
   Again you can view the results by

   ```
   sepview sepplot.001
   ```

8. To export pictures for your report, use the following command:

   ```
   sepplot2pdf
   ```

   sepplot2pdf translates all files named sepplot.xxx into pdf files with the names sepposc.xxx.pdf.

Remark: you may always view this manual by the command `sepman labmanual`.

Do not read the whole manual, but study the examples first and then read only those parts necessary to understand the example.

2   Running a simple example

2.1   General information

SEPRAN is a general FEM package consisting of 2 parts:

- preprocessing

- computation and postprocesssing

In the preprocessing part (called `sepmesh`) we create a mesh. This is done by describing the boundary. The mesh generator subdivides the region into elements, for example triangles. The result is written to a file called `meshoutput`.
Pictures of the mesh may be viewed by program `sepview`.

In the computational part (called `sepcomp`), the problem is solved. The user provides information about the differential equation to be solved, the boundary conditions and so on. The file `meshoutput` is read as input. The system of equations is created and solved. One can also print or plot the solution. The pictures can again be viewed by `sepview`.

2.2   Running an example

Before explaining how SEPRAN works it is best to consider an example. We start by getting a standard example into our directory and just run.
Perform the following steps

1. make a directory for example by
   `mkdir example`

2. go to this directory for example by
   `cd example`

3. `sepgetex labexam`
   This command puts 3 files called `labexam.msh`, `labexam.prb` and `labexam.f90` into your directory.

4. `sepmesh labexam.msh`
   This runs sepmesh with as input: labexam.msh.

5. `sepview sepplot.001`
   This starts the viewer, which gives you the opportunity to see all the pictures made.

6. `seplink  labexam.f90`
   This compiles and links the Fortran main program labexam.f90.

7. `labexam < labexam.prb`
   This runs the computational program labexam with as input the file labexam.prb.

8. `sepview sepplot.001`
   View results.

2.3   Explanation of the example

The example concerns the solution of a potential problem in an 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
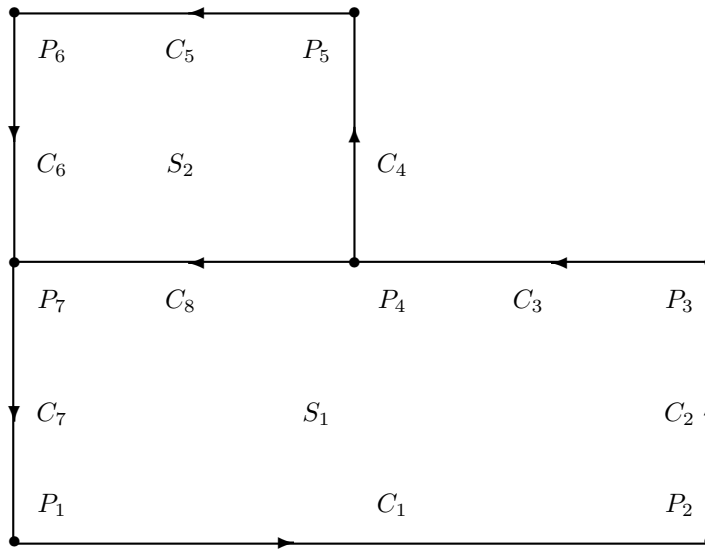
Figure 2.3.1: Definition of the L-shaped region with corresponding geometrical quantities

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:

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

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

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. So in fact this boundary condition is not given explicitly, but by not prescribing anything it is 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.

For the creation of the mesh the definition of Figure 2.3.1 is followed exactly. For both surfaces the surface generator triangle is used.

2.3.1   The file labexam.msh

First we explain the file labexam.msh

```
constants              # See Labmanual Section 3.3
   integers
      n = 5            # Number of elements along short side
      m = 2*n          # Number of elements along long side
   reals
      width = 2        # width of the region
      height = 2       # height of the region
      half_height = 1  # height of the lower part
      upper_right = 1  # x-coordinate of upper part right-hand side
end
```

The previous lines are not mandatory. They make it easier to define constants in the input file. These constants may be referred to by by the name of the constant. The value defined in the block constants is substituted. In this example four reals and two integers are defined.

```
#
#  Define the mesh
#
mesh2d                 # See Labmanual Section 3.4
```

The previous line is mandatory and tells sepmesh that the region is two-dimensional.

```
#
#  user points
#
   points              # See Labmanual Section 3.4
      p1=(0,0)
      p2=(width,0)
      p3=(width,half_height)
      p4=(upper_right,half_height)
      p5=(upper_right,height)
      p6=(0,height)
      p7=(0,half_height)
```

The lines following the keyword `points`, which must be used, define the coordinates of the points p1 to p7.
The syntax of these lines is relatively free: the =-sign and the brackets have no meaning at all except that they are used as separators. They just increase the readability.

```
#
#  curves
#
   curves              # See Labmanual Section 3.4
      c1 = line (p1,p2,nelm=m)
      c2 = line (p2,p3,nelm=n)
      c3 = line (p3,p4,nelm=m)
      c4 = line (p4,p5,nelm=m)
      c5 = line (p5,p6,nelm=n)
      c6 = line (p6,p7,nelm=n)
      c7 = line (p7,p1,nelm=n)
      c8 = line (p4,p7,nelm=m)
```

The lines following the keyword `curves`, define the various curves c1 to c8. These curves are all straight lines. The initial and end point are given as well as the number of elements along the lines.

```
#
#  surfaces
#
   surfaces          # See Labmanual Section 3.4
      s1 = triangle (c1,c2,c3,c8,c7)
      s2 = triangle (-c8,c4,c5,c6)
```

The lines following the keyword `surfaces`, define the two surfaces s1 and s2.
`triangle3` means that the surface generator `triangle` is used. Since the type of elements is not specified, the region is subdivided into triangles with three nodes.
The curves surrounding the surfaces are given in counterclockwise direction. The minus sign for c8 in s2 indicates that curve c8 must be followed in opposite direction.

```
#
#  Couple each surface to a different element group in order to provide
#  different properties to the coefficients
#
   meshsurf            # See See Labmanual Section 3.4
      selm1 = s1
      selm2 = s2
```

The lines following meshsurf are necessary because we want to introduce two element groups. Group 1 is coupled to S1 and refers to the region with $\mu = \mu_1$, group 2 is coupled to S2 and refers to the region with $\mu = \mu_2$.

```
   plot                          # make a plot of the mesh
                                 # See Labmanual Section 3.4
```

This line activates the plotting of the mesh and submeshes.

```
end
```

The last line closes the input and is therefore mandatory.

2.3.2   The file labexam.prb

Next we explain the file labexam.prb

```
# labexam.prb
#
#  problem file for the example
#
#  To run this file use:
#      sepcomp labexam.prb
#
#  Reads the file meshoutput
#  Creates the file sepcomp.out
#
```

Again some constants are defined, but also the name of the solution vector, following the subkeyword
`potential`. Mark that all constants in the file labexam.msh are not valid anymore in this file.

```
problem                       # See Labmanual Section 4.8.1

   types                          # Define types of elements,
                                  # See User Manual Section 3.2.2
      elgrp1 = (type=1)        # Type number for surface 1
                                  # See Standard problems Section 3.1
      elgrp2 = (type=2)        # Type number for surface 2
   essbouncond                    # Define where essential boundary conditions are
                                  # given (not the value)
                                  # See User Manual Section 3.2.2
      curves (c1)               # lower boundary
      curves (c5)               # upper boundary
end
```

Here we have a complicated part of the input file. It starts with the mandatory keyword `problem`
and ends with `end`.
In between is the keyword `types` followed by `elgrp1 = (type=1)` and `elgrp2 = (type=2)`. These
statements say that elements in element groups 1 and 2 have type numbers 1 and 2. These type
numbers are used in the element subroutine elemsubr that creates the element matrix and element
vector for each element.
The keyword `essbouncond` defines that essential boundary conditions are given and the following
statements where they are given. These statements do not define the values of the essential boundary
conditions.

Next the structure of the main program is defined.

```
#
#   Define the main structure of the program
#   See Labmanual Section 4.8.3
#
structure
   matrix_structure: symmetric   #  a symmetric profile matrix is used
   prescribe_boundary_conditions potential = 1, curves(c5)
   solve_linear_system potential
   print potential
   plot_contour potential
   plot_coloured_levels potential
end
```

In this example it is very simple, first the matrix structure is defined and implicitly the type of solver to be used. In this case a symmetrical profile matrix, implying that a direct method (LDL^T-decomposition) is used.
Next the essential boundary conditions are filled. This is done by the statement
`prescribe_boundary_conditions potential`.
The part = 1, curves(c5) sets the potential equal to one at curve $c5$. On all the curves with essential boundary conditions not mentioned, the value is set equal to zero.
The results are printed and plotted

```
print potential
```

Makes a print of the computed potential to the screen.

```
plot contour potential
plot coloured contour potential
```

makes a contour plot (equi-potential lines) and a colored plot of the potential.

```
end_of_sepran_input
```

This statement ends the input.

2.3.3   The file labexam.f90

The file labexam.f90 contains a Fortran program (the main program), consisting of the 3 statements:

```
program labexam
call startsepcomp
end
```

The first statement defines the program name, the second calls in fact the whole sepran package which consists of thousands of subroutines. The final statement ends the program.

These statements are followed by the subroutine elemsubr, which must be programmed by the user. In this subroutine the element matrix and element vector are computed. This subroutine is called internally for each individual element.
The first part of the subroutine contains the heading

```
subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                      elem_vec, elem_mass, prevsolution, itype )
```

This statement consists of `subroutine` followed by the name of the subroutine and all parameters. Mark that the sequence of the parameters is fixed and none of them may be removed! The `&` symbol indicates that the statement continues on the next line.
The input parameters are `npelm, x, y, nunk_pel, prevsolution, itype`. They have the followed meaning

npelm Number of points in element.

x,y contain the x and y-coordinates of the element nodes.

nunk_pel contains the number of unknowns in the element. For most problems this is equal to npelm.

prevsolution is only used in non-linear problems. It contains the solution at the previous iteration.

itype contains the type number defined in the input block problem.

The output parameters are `elem_mat, elem_vec, elem_mass`. The first two must be filled by this subroutine the last one only in time-dependent problems.

elem_mat is a two-dimensional array that must be filled with the element matrix.

elem_vec is a one-dimensional array that must be filled with the element vector.

All statements with an exclamation mark (`!`) are comments.
The statements

```
implicit none
integer, intent(in) :: npelm, nunk_pel, itype
double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                prevsolution(1:nunk_pel)
double precision, intent(out) :: elem_mat(1:nunk_pel,1:nunk_pel),  &
                                 elem_vec(1:nunk_pel),  &
                                 elem_mass(1:nunk_pel)
```

are declaration statements for the parameters in the parameter list. The first one implies that all variables must be declared. The intent parts indicate that the parameters are input or output. All reals are declared as double precision.

```
double precision :: mu, beta(1:3), gamma(1:3), delta
integer :: i, j
```

form the declarations of the local parameters.
The statements

```
if ( itype == 1 ) then
   mu = 1d0
else
   mu = 2d0
end if
```

define the parameter μ as function of the type number and hence of the element group.
The `d0` after 1 and 2, indicates that these numbers should be considered as double precision with exponent 0. It is always save to define your real numbers in this way to avoid problems in computations.
For example subdivision of 2 integers results in an integer and hence `1/2` is equal to 0, whereas `1d0/2d0` is equal to `0.5d0`.

```
delta = (x(2)-x(1))*(y(3)-y(1))-(y(2)-y(1))*(x(3)-x(1))
beta(1) = (y(2)-y(3))/delta
beta(2) = (y(3)-y(1))/delta
beta(3) = (y(1)-y(2))/delta
gamma(1) = (x(3)-x(2))/delta
gamma(2) = (x(1)-x(3))/delta
gamma(3) = (x(2)-x(1))/delta
```

is a series of statements to compute Δ, β and γ as defined in the lecture notes.

```
do j = 1, 3
   do i = 1, 3
      elem_mat(i,j) = mu * 0.5d0 * abs(delta) *   &
         ( beta(i)*beta(j) + gamma(i)*gamma(j) )
   end do
end do
elem_vec(1:3) = 0d0
```

are statements to fill the element matrix and vector. Exactly the formulas given in the book Numerical Methods in Scientific Computing are used.

3 Mesh generation

3.1 General remarks

The definition of the elements is performed in two stages:

- in the first stage the user defines geometrical quantities as points, curves and surfaces, 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 and surfaces

For the generation of meshes we define the following quantities:

Points, Curves and Surfaces

Points form the basis for all other components. The user must define the main points necessary for the generation of curves. 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, but only if they are part of a surface.

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 boundaries of a surface 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 3.1.1). Surfaces may contain holes. In that case the holes must be enclosed by a closed set of curves. The first set of curves refers to the outer boundary, and it must be followed by curves for each hole separately.

Examples

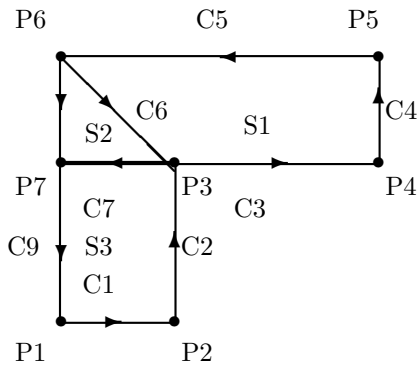In Figure 3.1.1 the points, curves and surfaces for some regions are defined. Points are indicated by Pk (k=1 ,2 ,,,), curves by Cl (l=1 ,2 ,,,) and surfaces by Sm (m=1 ,2 ,,,). The corresponding commands are POINTS, CURVES and SURFACES.
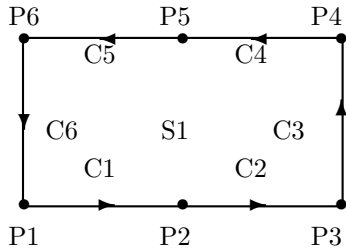
C1 = (P1,P2)   C2 = (P2,P3)
C3 = (P3,P4)   C4 = (P4,P5)
C5 = (P5,P6)   C6 = (P6,P1)
S1:(C1,C2,C3,C4,C5,C6)

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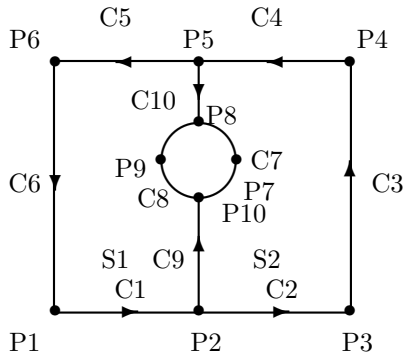
C1 = (P1,P2)   C2 = (P2,P3)
C3 = (P3,P4)   C4 = (P4,P5)
C5 = (P5,P6)   C6 = (P6,P1)
C7 = (P8,P7,P10)   C8 = (P10,P9,P8)
C9 = (P2,P10)   C10 = (P5,P8)
S1:(C1,C9,C8,-C10,C5,C6)
S2:(C2,C3,C4,C10,C7,-C9)

Outer boundaries part 1: C1, C2, C3, C4, C5, C6
Outer boundaries part 2: C7,C8
Inner boundaries: C9, C10
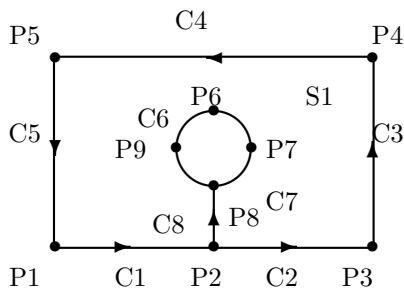
C1 = (P1,P2)   C2 = (P2,P3)
C3 = (P3,P4)   C4 = (P4,P5)
C5 = (P5,P1)   C6 = (P6,P9,P8)
C7 = (P8,P7,P6)   C8 = (P2,P8)
S1:(C8,-C6,-C7,-C8,C2,C3,C4,C5,C1)

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

Figure 3.1.1: Points, Curves and Surfaces
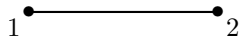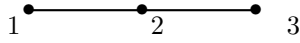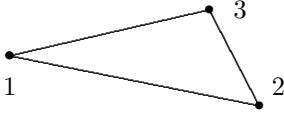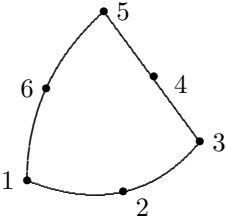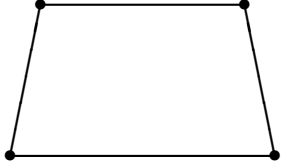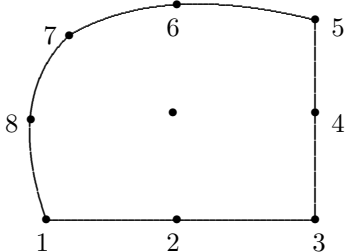
shape number	shape	name
1		line element with 2 points
2		line element with 3 points
3		triangle with 3 points
4		isoparametric triangle with 6 points
5		quadrilateral with 4 points
6		isoparametric quadrilateral with 9 points

Table 3.1.1: Standard elements for mesh generation

3.1.2   Generation of curves

First the user must define the points, secondly the curves and finally the surfaces.
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.
For the definition of the curves the following FUNCTIONS are available:

LINE : generates a straight line from point Pi to Pj.

ARC : generates an arc from point Pi to Pj; the centroid Pc must be given.

ELL_ARC : generates a part of an ellipse from point Pi to Pj; the centroid Pc must be given. The difference with ARC is that Pi and Pj do not have to lie on a circle.

CURVES : generates a curve consisting of the subsequent curves Ck, Cl, Cm.

PARAM The user defines a curve by a function subroutine FUNCCV (3.5.1) using a parameter representation.

For other functions the reader is referred to the Users Manual. is an integer which defines the type of elements along the curves to be created.

The FUNCTIONS LINE, ARC, USER, CURVES and PARAM have the following shape:

```
C1 = LINE ( P1, P2, NELM = n )
C2 = ARC  ( P1, P2, Pc, NELM = n )
C6 = CURVES ( Ck, Cl, Cm, . . )
C7 = PARAM ( P1, P2, NELM=n ,INIT=t_0,END=t_1 )
C9 = ELL_ARC  ( P1, P2, Pc, NELM = n )
```

with n the number of elements in the curve.

$element\_type = 1$ means linear elements, consisting of 2 points (Default value)
$element\_type = 2$ means quadratic elements, consisting of 3 points, with the second point in the center of the first and the last one.
INIT $= t_0$ and END $= t_1$, define the range of the parameter t. The default values are: $t_0 = 0$ and $t_1 = 1$. If these values are omitted the default values are used.

3.1.3   Generation of surfaces

Each surface must coincide with a submesh (in two-dimensional problems). For generation of nodal points and elements in the surface a number of so-called surface generators are available. Of these surface generators only two are treated in this manual. For the other ones the user is referred to the Users Manual.

The surface generators described in this manual are TRIANGLE and QUADRILATERAL.

TRIANGLE has the following characteristics:

1. A fine division of nodal points on a part of the boundary causes a fine mesh in the neighborhood of this boundary; a coarse division, a coarse mesh.

2. The mesh generator can generate elements when a sudden refinement of the nodal points of the boundary is present, however, the quality of the mesh may be poor in that case. Hence when the user wants to create elements on a long small pipe (see Figure 3.1.2) TRIANGLE is not suited, or the user must transform his coordinates such that the length/width ratio is not too large. For that type of meshes use QUADRILATERAL.

3. TRIANGLE allows holes in the mesh.

4. In case the user needs quadrilaterals instead of triangles, he should use the generator GENERAL. Consult the Users Manual Section 2.4.1 for its use and restrictions.

$$h = 1$$

$$L = 10$$
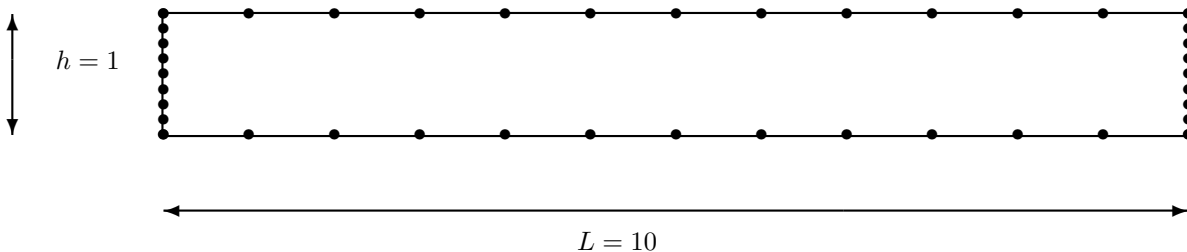
Figure 3.1.2: Example of a region that should not be subdivided by TRIANGLE

QUADRILATERAL has the following characteristics:

1. 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 not be so extreme that there is no resemblance with a rectangle.

2. QUADRILATERAL expects exactly four curves, each one representing one "side" of the transformed "rectangle". If some of these sides consist of subcurves the user must combine these curves into one curve using the option CURVES (of curves).

3. 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.

4. QUADRILATERAL has no problem with oblong elements.

The functions TRIANGLE and QUADRILATERAL have the following shape:

```
S1 = TRIANGLE ( C1, C2, C3, C4 . . . )
S2 = QUADRILATERAL ( C1, C2, C3, C4)
```

is an integer which defines the type of elements in the surfaces to be created. In Table 3.1.1 a survey of the available standard elements for mesh generators is given. The element types 3 and 4 can be generated by TRIANGLE, type 3 to 6 by QUADRILATERAL.

TRIANGLE has an extra option to include stand-alone user points in the mesh, where they appear as nodal point. This is done by adding

```
internal_points = pi, pj, ...
```

before the closing bracket in the line `TRIANGLE ( C1,... )`

3.1.4   Coupling of geometrical quantities with element groups

The points, curves and surfaces as defined in 3.1.1 to 3.1.3 are necessary to generate elements. However, not all of them may be necessary for the finite element problem. Those elements that are necessary must be identified with a standard element by means of an element group number (see 2.1).
The coupling of the geometrical elements with the standard elements is done using the command MESHSURFACES defining the two dimensional elements. These commands must be followed by function records of the type:
       SELM i = (S1, S2)
with `SELM` corresponding to the surface elements.
i is the element group number.

If all surface elements have the same properties the part with MESHSURFACES may be skipped.

3.2   A simple example

Before we describe the input for the mesh generator in detail we shall first give an example to show how a simple mesh may be created.

To that end we consider a rectangular region as sketched in Figure 3.2.1. The user points and
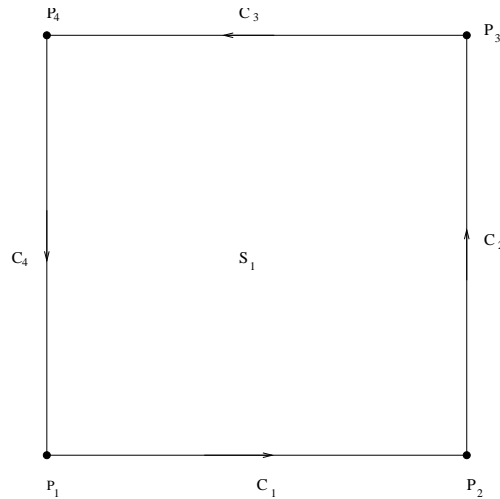
Figure 3.2.1: Example of a region to be divided in elements

curves are indicated in the region. Suppose that the height is 1 and the width is also 1.

In order to create a mesh by SEPRAN we first have to make an input file by a text editor. Suppose this input file is called `square.msh`.

In order to create the mesh we have to call the program sepmesh in the following way:

```
sepmesh square.msh
```

If the input file is incorrect, sepmesh produces error messages, which are self-explaining. Sometimes, however, the number of errors is so large that more than one screen is needed. In that case it might be wise to redirect the file to an output file, for example:

```
sepmesh square.msh > square.out
```

Never use the name `meshoutput` for this output file.

The output file may be inspected by a text editor.

Then sepmesh creates a mesh and puts information in a file called `meshoutput`. Depending on the contents of the file square.msh a series of files `sepplot.001`, `sepplot.002` ... may be created which contain plots related to the mesh. These plots may be viewed by `sepview`.

The input file square.msh may for example look like:

```
#  square.msh
#
#  example file for the generation of elements in a square
#
#  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 = triangle ( c1, c2, c3, c4 )

    #     Plot the mesh

    plot
end
```

Explanation:

- In the part `constants ... end`, some general constants with respect to the mesh are defined. In this case, the width and the height of the mesh and the number of elements in horizontal and vertical direction. In this way it is easy to change these numbers later on.

- The part `mesh2d ... end` is meant for the actual mesh generation.
  First all user points are defined, next the curves as straight lines with linear elements (`line`), begin point, and point and number of elements.
  After that, the surface is defined using the submesh generator triangle with triangular elements (`triangle`), and corresponding curves and finally a plot command is given.
  The constants in the mesh definition, may be used in the remainder of the file.
  Everything after the hash symbol is treated as comment.

3.3 Some remarks concerning the input files

In the previous section we have seen an example of a simple input file. In the next section we shall treat a part of the input for the mesh generator. But before doing so we consider some general rules that are valid for all SEPRAN input files that are defined in the SEPRAN manuals, unless otherwise stated.
First of all it must be noted that the input file is not a FORTRAN file, hence rules that apply for FORTRAN files are not generally applicable to the SEPRAN input files. In fact each input file is interpreted, character for character.
The following rules are generally applicable for the input files:

- At most 240 characters in each line of the input file are read, all characters that are present after column 240 are neglected.

- If an input line requires more than 240 columns continuation of this line may be defined by putting the character & after the last text on a line, but of course within the columns 1 to 240. This means that the line is continued on the next line.
  For example the next two lines are considered as one line:

  ```
  c4 = line ( p1, p2,&
              nelm = nelm_hor  )
  ```

- You may put comment in the input file in two ways:

  1. By putting a * in column 1. The whole line is treated as comment.
  2. By putting a hash (#) or an exclamation mark (!) in the text. All characters behind these marks are treated as comment.

- SEPRAN does not distinguish between capitals and lower case, except in character strings.

- Numbers must satisfy the standard FORTRAN rules. However, they may not contain spaces. Examples are 1   1.0   1d0 1.0d0 1e0 1.0e0 0.01 .01 -0.01

- Spaces and end of lines are treated as separation symbols. Also special characters as , = : ; may be used as separator.

- The input file may start with a part CONSTANTS to define some general constants. This part has the following layout:

  ```
  CONSTANTS
     INTEGERS
        name1 = value
        name2
        name3 = value
     REALS
        rname1 = value
        rname6 = value
        rname3 = value
  END
  ```

  These records have the following meaning

  CONSTANTS (mandatory). This keyword indicates that constants will be defined.
  If this keyword is not present as first keyword in the file it is not possible to define constants. This keyword may be followed by the subkeywords (always on a new line):

  INTEGERS This keyword indicates that some integer constants will be defined.
  It must be followed by the integers to be defined.
  The layout of the integers is:

`name_of_constant value`

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

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

REALS This keyword indicates that some real constants will be defined.
It must be followed by the reals to be defined according to exactly the same rules as for the integers. Names of reals must be different from the names of the integers.

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

The block CONSTANTS must always be read as first block.

3.4 Input for the mesh generator

The input for the mesh generator must be opened with MESH1D or MESH2D, depending on whether the problem is one– or two-dimensional, and must be closed with END.

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

MESHnD (mandatory)
      opens the input for SEPMESH, and defines the dimension of the space NDIM. (NDIM = n).

type_elements = xxx (optional). Defines the type of elements that are used.
      The following options for xxx are available:

   LINEAR All elements are linear, hence linear line elements, linear triangles and linear tetra-hedrons. (Default value)

   QUADRATIC All elements are quadratic.

   QUADRILATERAL The elements are bi-linear quadrilaterals.

   QUADRATIC_QUADRILATERAL The elements are bi-quadratic quadrilaterals.

POINTS (mandatory)
      defines the points. Must be followed by records of the type:

```
P1 = ( x_1 , y_1 )
P2 = ( x_2 , y_2 )
             .
             .
             .
Pi = ( x_i , y_i )
```

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

CURVES (mandatory)
      defines the curve. Must be followed by records of the type:

```
C1 = LINE ( P1, P2, NELM=4  )
C2 = ARC ( P1, P2, P3, NELM = 3 )
```

   etc.
   with Ci the curve number.
   The names LINE, ARC are names that may **not** be removed.

   See Section 3.1.2.

SURFACES (optional)
      defines the surfaces. Must be followed by records of the type:

```
S1 = TRIANGLE ( C1, C2, C3, C4, . . . )
S2 = TRIANGLE ( -C5, C6, -C9, C5, . . . )
S3 = QUADRILATERAL ( C4, -C6, C8, C2)
```

   etc.
   with Si the surface number.
   The names TRIANGLE and QUADRILATERAL are names that may **not** be removed.
   If the user wants to define several element groups, for example since he uses different values of a specific coefficient for different parts of the region, he has to use the option MESHSURF.

MESHSURF (optional)

defines the two-dimensional elements in R^2. Must be followed by records of the type:

```
SELM i = ( S1 to S2 )
```

with i the element group number. Standard elements must be generated with increasing element group number, first all line elements, then all surface elements. Elements of the same element group must have exactly the same shape, i.e. they must be all linear triangles or all bi-quadratic quadrilaterals and so on.

S1 to S2: the elements generated on the surfaces S1, S1 + 1,..., S2 are appended to the mesh. When `to S2` is not given only surface S1 is used.

Auxiliary commands

PLOT (optional)

indicates that the points, curves, the surfaces and the mesh must be plotted, each on a new picture.

END (mandatory)

End of the input for subroutine MESH.

Remark:

The input must be given in the sequence:

  MESH record
  TYPE_ELEMENTS record
  POINTS
  CURVES
  SURFACES
  MESHSURF
  PLOT
  END

When MESHSURF are given, it is supposed that there is only one type of internal element, with element group number 1.

Examples

Simple square

Consider the region in Figure 3.4.1

Let the number of elements along each side be equal to 10, with equidistant mesh sizes. Then the following input can be used:

```
mesh2d
   points
      p1=(0,0)
      p2=(1,0)
      p3=(1,1)
      p4=(0,1)
   curves
      c1=line(p1,p2,nelm=10)
      c2=line(p2,p3,nelm=10)
      c3=line(p3,p4,nelm=10)
```
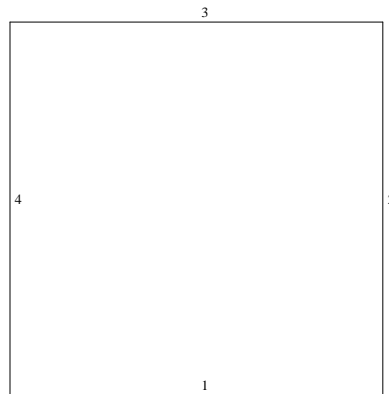
Figure 3.4.1: Example of a region to be divided in elements

```
      c4=line(p4,p1,nelm=10)
   surfaces
      s1 = triangle(c1,c2,c3,c4)
   plot
end
```

Figure 3.4.2 shows the result of the mesh generation. Figure 3.4.3 shows the result of the same region with TRIANGLE replaced by QUADRILATERAL.
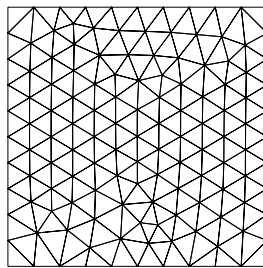
Figure 3.4.2: The result of the mesh generation.

Square with a hole

Consider the region in Figure 3.4.4

The following input may be used:

```
#  hole.msh
#
constants
   reals
      height = 1                    # height of the square
      width  = 1                    # width of the square
      xhole_left = 0.25             # left x-coordinate of hole
      yhole_under = 0.25            # lower y-coordinate of hole
```
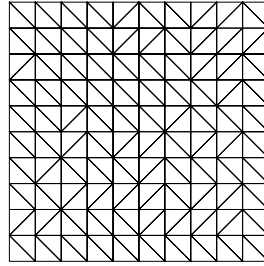
Figure 3.4.3: Result of the same region, with TRIANGLE replaced by QUADRILATERAL.
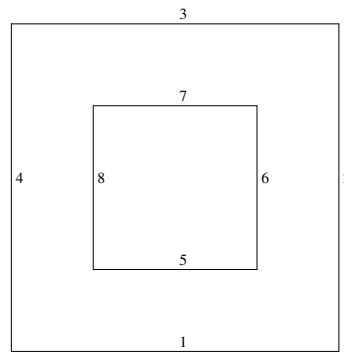
Figure 3.4.4: Example of square with hole

```
      width_hole = 0.5                    # width of the hole
      height_hole = 0.5                   # height of the hole
   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)
      p5=(xhole_left,yhole_under)
      p6=(xhole_left+width_hole,yhole_under)
      p7=(xhole_left+width_hole,yhole_under+height_hole)
      p8=(xhole_left,yhole_under+height_hole)
```

```
#    Definition of the curves

curves
#   Boundary of square

    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 )

#   Boundary of hole
    c5 = line ( p5,p6,nelm= nelm_hor )
    c6 = line ( p6,p7,nelm= nelm_vert )
    c7 = line ( p7,p8,nelm= nelm_hor )
    c8 = line ( p8,p5,nelm= nelm_vert )

#    Definition of the surface

surfaces
    s1 = triangle ( c1, c2, c3, c4, &
                    c5, c6, c7, c8 )

#    Plot the mesh

plot
end
```

Figure 3.4.5 shows the result of the mesh generation.
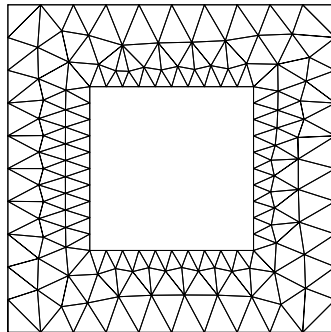
Figure 3.4.5: Mesh for square with hole

3.5   How to use the parameter curve

For some of the exercises it is necessary to define the boundary as a parameter curve. In that case we use an input like:

```
C7 = PARAM  ( P1, P2, NELM=n, INIT=t_0, END=t_1 )
```

This means that a parameter curve is defined by the user with a parameter t that is between t_0 and t_1. For $t = t_0$ the coordinates must coincide with P1 and for $t = t_1$. with P2. In order to define the parameter curve the user must define a FORTRAN subroutine called `funccv` that gives the coordinates x, y and z as function of t. Default values for t_0 and t_1 are 0 and 1 respectively. The function funccv itself is described in section 3.5.1.

You have to make the subroutine itself by a text editor and give it the name `funccv.f90`

To compile this subroutine and to run sepmesh with this subroutine, perform the following steps:

```
sepgetlab sepmeshexe
compile funccv.f90
seplink sepmeshexe
sepmeshexe < input.msh
```

The first step copies a file sepmeshexe.f into your directory.
The next step compiles the file funccv.f90, which much obey all Fortran 90 rules. If the compilation is without errors a file called `funccv.o` is created. Only then it makes sense to do the next step.
The following step compiles the Fortran program sepmeshexe.f links it with the SEPRAN library as well as the file funccv.o and creates an executable called `sepmeshexe`.
In the final step we run sepmeshexe with an input file, which in this case is called `input.msh`, but of course you may use any name you want. Mark the "<" sign which indicates that input is read from a file.
If everything is correct and no error messages are produced, this works exactly as sepmesh and creates files `meshoutput` and `sepplot.001` and so on.
In the subsection 3.5.1 we describe the subroutine FUNCCV, and in subsection 3.5.2 we give an example.

3.5.1   Subroutine FUNCCV

Description

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

Heading

```
        subroutine funccv ( icurve, t, x, y, z)
```

Parameters

> **DOUBLE PRECISION** T, X, Y, Z
>
> **INTEGER** ICURVE
>
> **ICURVE** Curve number. Subroutine MESH gives ICURVE the sequence number of the curve to be generated.

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

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

Input

Program SEPMESH gives ICURVE and T a value

Output

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

Interface

Subroutine FUNCCV must be programmed as follows:

```
subroutine funccv ( icurve, t, x, y, z)
implicit none
integer, intent(in) :: icurve
double precision, intent(in) :: t
double precision, intent(out) :: x, y, z
        .
        .
        .        statements to give x,y and z a value as function
        .        of t and icurve
        .
end subroutine funccv
```

3.5.2 An example of the use of FUNCCV

In this artificial example we consider a square, where we replace the upper straight line by the parameter curve defined by: $x = 1 - t, y = 1 + t(1 - t)$, where t is between 0 and 1.
So we have to make an input file for sepmesh, a Fortran file `funccv.f90` satisfying the Fortran rules and copy the file sepmeshexe.f to the local directory.
The mesh input file in this example looks like:

```
#  example_funccv.msh
#
#  mesh file to show the use of the param function
#
#
#  Define some general constants
#
constants              # See Users Manual Section 1.4
   reals
      width = 1         # width of the square
      length = 1        # length of the square
   integers
      n = 10            # number of elements in length direction
      m = 10            # number of elements in width direction
end
#
```

```
#  Define the mesh
#
mesh2d
type_elements = quadrilateral  # bi-linear quadrilaterals are used
#
#  user points
#
   points
      p1=(0,0)                # Left under point
      p2=(length,0)          # Right under point
      p3=(length,width)      # Right upper point
      p4=(0,width)            # Left upper point
#
#  curves
#
   curves             # See Users Manual Section 2.3
      c1=line (p1,p2,nelm = n)     # lower boundary
      c2=line (p2,p3,nelm = m)     # right-hand side boundary
      c3=param (p3,p4,nelm = n)    # upper boundary (parameter function)
      c4=line (p4,p1,nelm = m)     # left-hand side boundary
#
#  surfaces
#
   surfaces           # See Users Manual Section 2.4
                      # Linear quadrilaterals are used
      s1=quadrilateral(c1,c2,c3,c4)

   plot                         # make a plot of the mesh

end
```

The file funccv.f90 has the following shape

```
subroutine funccv ( icurve, t, x, y, z )
implicit none
integer, intent(in) :: icurve
double precision, intent(in) :: t
double precision, intent(out) :: x, y, z
if ( icurve == 3 ) then

!  icurve = 3, curve number 3

   x = 1-t
   y = 1+t*(1-t)

else

!  icurve # 3, wrong curve

   print *,'icurve is ', icurve, 'has not been programmed'

end if

end subroutine funccv
```

Figure 3.5.1 shows the curves created by sepmeshexe and Figure 3.5.2 the corresponding mesh.
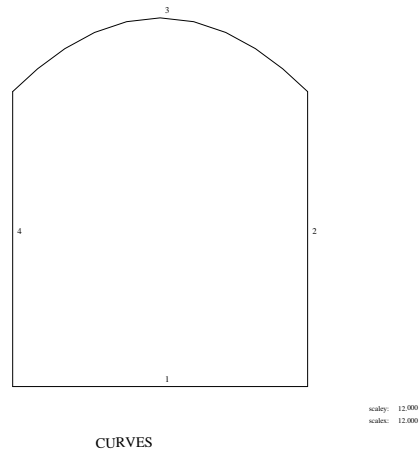
Figure 3.5.1: Curves created by sepmeshexe
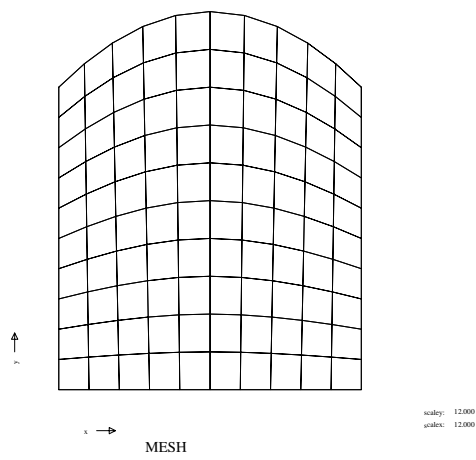
Figure 3.5.2: Mesh created by sepmeshexe

4   The computational part of SEPRAN

4.1   Introduction

The computational part of SEPRAN is the heart of the exercise. The participant has to use a standard main program called `sepcompexe` and add one or more element subroutines, which he has to program himself. In order to avoid unnecessary errors the templates of these subroutines have already been preprogrammed. These subroutines must be written in FORTRAN, hence all standard FORTRAN 90 rules must be satisfied. Besides that a SEPRAN input file must be written, obeying all SEPRAN rules. A simple example can be found in Section 2.3.

Section 4.2 contains an example of the use of boundary elements.

In Section 4.3 an example of a non-linear problem is given.

Section 4.4 explains how to compute integrals and derivatives of the solution.

Finally an example with 2 unknowns per point is treated in Section 4.5.

4.2   A mathematical test example showing the use of boundary elements

Consider the pure artificial problem:
$\Delta\phi = 0$ $x \in (0,1) \times (0,1)$
with boundary conditions:
$\phi = xy$ on curves c1, c2 and c4
$\frac{\partial\phi}{\partial n} = x$ on curve c3.
The region is shown in Figure 4.2.1 The student must create 4 files in this particular case:

C_3

C_4

C_2

C_1
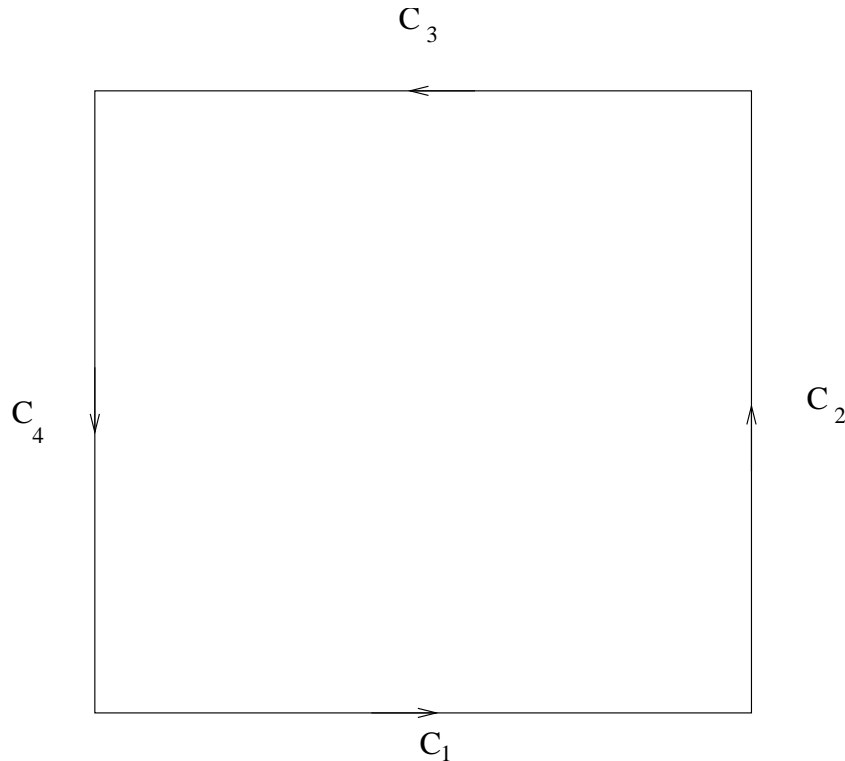
Figure 4.2.1: Region corresponding to artificial test example

```
lab4-2.msh
elemsubr.f90
funcbc.f90
lab4-2.prb
```

The file `lab4-2.msh` contains the mesh input. The user can create this file himself with the information of Chapter 3.
You have to get sepcompexe into your local directory by `sepgetlab sepcompexe`.
Use the command `sepgetlab elemsubr` to get the template for `elemsubr.f90` into your local directory. The file `lab4-2.prb` contains the input for the computational program.
The commands to be carried out are:

```
sepmesh lab4-2.msh
compile elemsubr.f90
seplink sepcompexe
sepcompexe < lab4-2.prb > outputfile
```

Mark that the next command may only be carried out if you are sure that the previous one has been finished successfully.

In this particular case we have to supply both an element subroutine `elemsubr`, which is used to define the boundary conditions that depend on space.

The file elemsubr.f90 has the following shape

```fortran
      subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                            elem_vec, elem_mass, prevsolution, itype )
!
!                     INPUT / OUTPUT PARAMETERS
!
      implicit none
      integer, intent(in) :: npelm, nunk_pel, itype
      double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                      prevsolution(1:nunk_pel)
      double precision, intent(out) :: elem_mat(1:nunk_pel,1:nunk_pel),  &
                                       elem_vec(1:nunk_pel),  &
                                       elem_mass(1:nunk_pel)
! **********************************************************************
!
!                      LOCAL PARAMETERS
!
      double precision :: beta(1:3), gamma(1:3), delta, h
      integer :: i, j

! ======================================================================
!
      if ( itype==1 ) then

!     --- Type = 1: internal element
!          Compute the factors beta, gamma and delta as defined in the NMSC

          delta = (x(2)-x(1))*(y(3)-y(1))-(y(2)-y(1))*(x(3)-x(1))
          beta(1) = (y(2)-y(3))/delta
          beta(2) = (y(3)-y(1))/delta
          beta(3) = (y(1)-y(2))/delta
          gamma(1) = (x(3)-x(2))/delta
          gamma(2) = (x(1)-x(3))/delta
          gamma(3) = (x(2)-x(1))/delta

!         --- Fill the element matrix as defined in the Lecture Notes

          do j = 1, 3
             do i = 1, 3
                elem_mat(i,j) = 0.5d0 * abs(delta) *   &
                   ( beta(i)*beta(j) + gamma(i)*gamma(j) )
             end do
          end do

!         --- The element vector is zero

          elem_vec(1:3) = 0d0

      else

!     --- Type = 2: boundary element
!          Compute Jacobian h
```

```
        h = sqrt ( (x(2)-x(1))**2 + (y(2)-y(1))**2 )

!       --- The element matrix is zero

        elem_mat = 0d0

!       --- Fill the element vector

        elem_vec(1:2) = h * 0.5d0 * x(1:2)

     end if

     end subroutine elemsubr
```

Explanation: The subroutine elemsubr is almost identical to the one in Section 2.3.2. The extra part concerns type 2 which refers to the boundary elements. The input file for this example may have the following shape

```
#******************************************************************************
#
#      File:  lab4-2.prb
#
#      Contents:  Input for computational part of the example as described
#                 in the SEPRAN Lab manual 4.2
#
#      Usage:    sepcompexe < lab4-2.prb
#
#      It has been supposed that the following actions have been carried out
#      with success:
#
#      sepmesh lab4-2.msh
#      compile elemsubr.f90
#      seplink sepcompexe
#******************************************************************************
#
#
#
#  Problem definition
#
problem
   types                            # Define type numbers per element group
      elgrp1 = (type=1)             # Element group 1: itype = 1
   natbouncond                      # Define types of natural boundary
                                    # conditions
      bngrp1 = (type=2)             # Boundary element group 1: itype = 2
   bounelements                     # Define where natural boundary
                                    # conditions are present
      belm1 = curves(c3)            # Boundary element group 1 is
                                    # defined on c3
   essboundcond                     # Define where essential boundary
                                    # conditions are defined (not there
                                    # value)
      curves(c1 to c2)              # The potential on c1 to c2 is given
      curves(c4)                    # The potential on c4 is given
```

```
end
#
#  Define the structure of the main program
#
structure
  # Define the structure of the large matrix
   matrix_structure: symmetric   #  a symmetric profile matrix is used
#  Define non-zero essential boundary conditions
#  # On c1, c2 and c4: phi = xy
   prescribe_boundary_conditions potential = x_coor*y_coor, curves(c1,c2,c4)
   solve_linear_system, potential
   print potential
   plot_contour potential
   plot_coloured_levels potential
end
end_of_sepran_input
```

Explanation:
Also this part is almost identical to the file in Section 2.3.2. There are two major differences:

- In the problem block we have added a part `natbouncond` and a part `bounelements`. The `natbouncond` part defines the type number corresponding to the boundary elements. Also in this case we may have several boundary element groups to distinguish between several *natural* boundary conditions.
  In the `bounelements` part we define the boundary elements corresponding to the boundary group.

The boundary conditions on curves c1, c2 and c4 are a function of x and y. You can use x_coor and y_coor to get the arrays that contain the x and y-coordinates respectively.
Note that the sequence `prescribe_boundary_conditions name_of_vector = ...` is essential.
The program checks on the equals sign and expects this to be the third item in the statement.

4.3 A non-linear potential problem

The student must create 3 files in this particular case:

```
lab4-3.msh
elemsubr.f90
lab4-3.prb
```

As an example we consider the solution of a non-linear potential problem in a unit square. So actually we are using the same region as in Section 4.2.

In this special case we want to solve the non-linear potential problem:

$$- \operatorname{div} \mu \bigtriangledown \phi = e^{-\phi} \tag{4.3.1}$$

with $\phi = 0$ on the whole boundary. Since the right-hand side depends on the solution we have to apply a non-linear iteration.

The following Picard type iteration could be applied:

$\phi^0 = 0$
$\varepsilon = 10^{-3}$
Diff $= 1$
$k = 1$
while Diff $> \varepsilon$ **do**
    Solve: $- \operatorname{div} \mu \bigtriangledown \phi^k = e^{-\phi^{k-1}}$
    Diff $= ||\phi^k - \phi^{k-1}||$
    $k = k + 1$
end while

In each step of the iteration the linear partial differential equation $- \operatorname{div} \mu \bigtriangledown \phi^k = e^{-\phi^{k-1}}$ must be solved. This equation requires the building of a matrix and right-hand side and hence a corresponding element subroutine.

The element matrix in this case is of the same shape as in Section 2.3.2. The only difference is the element vector, which is non zero, due to the presence of a non-vanishing right-hand side.

Following the book *Numerical Methods in Scientific Computing* the element vector is given by

$$f_i^e = \frac{|\Delta|}{6} \phi_i^{k-1}, \tag{4.3.2}$$

provided a Newton Cotes integration rule is applied. The file `lab4-3.msh` contains the mesh input. The user can create this file himself with the information of Chapter 3.

The file `elemsubr.f90` contains the element subroutine that must be compiled and linked, like in Section 2.3.2.

Use the command `sepgetlab elemsubr` to get the template of the element subroutine into your local directory. The file `lab4-3.prb` contains the input for the computational program.

The commands to be carried out are:

```
sepmesh lab4-3.msh
compile elemsubr.f90
seplink sepcompexe
sepcompexe < lab4-3.prb > outputfile
```

Mark that the next command may only be carried out if you are sure that the previous one has been finished successfully.

In this particular case we have to supply an element subroutine `elemsubr`.

The file elemsubr.f90 has the following shape

```
      subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                            elem_vec, elem_mass, prevsolution, itype )
!
!                     INPUT / OUTPUT PARAMETERS
!
      implicit none
      integer, intent(in) :: npelm, nunk_pel, itype
      double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                    prevsolution(1:nunk_pel)
      double precision, intent(out) :: elem_mat(1:nunk_pel,1:nunk_pel),  &
                                    elem_vec(1:nunk_pel),  &
                                    elem_mass(1:nunk_pel)
! ************************************************************************
!
!                     LOCAL PARAMETERS
!
      double precision :: beta(1:3), gamma(1:3), delta
      integer :: i, j
! ======================================================================
!
!     --- Compute the factors beta, gamma and delta as defined in NMSC

      delta = (x(2)-x(1))*(y(3)-y(1))-(y(2)-y(1))*(x(3)-x(1))
      beta(1) = (y(2)-y(3))/delta
      beta(2) = (y(3)-y(1))/delta
      beta(3) = (y(1)-y(2))/delta
      gamma(1) = (x(3)-x(2))/delta
      gamma(2) = (x(1)-x(3))/delta
      gamma(3) = (x(2)-x(1))/delta


!     --- Fill the element matrix as defined in the Lecture Notes

      do j = 1, 3
         do i = 1, 3
            elem_mat(i,j) = 0.5d0 * abs(delta) *   &
                ( beta(i)*beta(j) + gamma(i)*gamma(j) )
         end do
      end do


!     --- The element vector is defined by the previous solution

      elem_vec(1:3) = abs(delta)/6d0*exp(-prevsolution(1:3))

      end subroutine elemsubr
```

Explanation: The subroutine elemsubr is almost identical to the one in Section 2.3.2. The extra part concerns the element vector that depends on the solution in the previous iteration. This solution is stored in array uold.

The input file for this example may have the following shape

```
#************************************************************************
#
#     File:  lab4-3.prb
#
#     Contents:  Input for computational part of the example as described
```

```
#                   in the SEPRAN Lab manual 4.4
#
#      Usage:     lab4-3 < lab4-3.prb
#
#      It has been supposed that the following actions have been carried out
#      with success:
#
#      sepmesh lab4-3.msh
#      compile elemsubr.f90
#      seplink sepcompexe
#*****************************************************************************
#
#
#
#  Problem definition,
#
problem
   types                                 # Define type numbers per element group
      elgrp1 = (type=1)                  # Element group 1: itype = 1
   essboundcond                          # Define where essential boundary
                                         # conditions are defined (not there
                                         # value)
      curves(c1 to c4)                   # The potential is given on c1 ... c4
end
#
#  Define the structure of the main program
#
structure
   matrix_structure: symmetric   #  a symmetric profile matrix is used
   create_vector potential = 0
   diff = 1
   iter = 0
   eps = 1e-3
   print '  iter  difference   speed'
   while ( diff > eps and iter<10 ) do
      iter = iter+1
      potential1 = potential
      diffprev = diff
      solve_linear_system potential
      diff = inf_norm(potential1-potential)
      speed = diff/diffprev
      print iter, diff, speed
   end_while  ! diff > eps do

   print potential
   plot_contour potential
   plot_coloured_levels potential
end
end_of_sepran_input
```

Explanation:
Also this part is almost identical to the file in Section 2.3.2. There are two major differences:

- In the block `structure` we have replaced `solve_linear_system potential` by a series of
  statements performing a non-linear iteration. In this particular case the start vector is created

by `Create_vector`, which has the same syntax as as prescribe_boundary_conditions.
The iteration loop itself is quite trivial. Some parameters are initialized and the iteration is performed in a while loop. The iteration is stopped when the difference is less than eps or if the number of iterations exceeds 10. Not only the difference between two succeeding iterations is printed but also the speed of convergence. During the iteration in each step a new matrix and right-hand side is built and a linear system of equations is solved.

4.4   How to compute derivatives and integrals using user elements

In this section it is shown how one can compute integrals and derivatives of the solution, once the solution has been computed.

Starting point is the example treated in Section 2.3.2.

This example is extended with the computation of the x-derivative $\frac{\partial \phi}{\partial x}$ of the potential ϕ. After that the integral $\int_{\Omega} \phi d\Omega$ is computed.

To avoid confusion with the potential ϕ we denote the basis functions by $\psi_i(\mathbf{x})$.

The x-derivative of the potential is defined by

$$\frac{\partial \phi}{\partial x} = \sum_{i=1}^{n} \phi(\mathbf{x}_i) \frac{\partial \psi_i}{\partial x}(\mathbf{x}). \tag{4.4.3}$$

This derivative is constant per triangle (why?) and is defined by the three values of the potential in the nodes of the triangle. In order to compute the derivatives in each point, SEPRAN expects you to write a subroutine `eldervsubr` as described in Section 4.7.2. Output of this element subroutine must be the computed derivative in each node. An example of the subroutine `eldervsubr` is given by:

```
      subroutine eldervsubr ( npelm, x, y, nunk_pel, elem_vec,  &
                              solution, itype, icheld, len_outvec )
!
!                         INPUT / OUTPUT PARAMETERS
!

      implicit none
      integer, intent(in) :: npelm, nunk_pel, itype, icheld, len_outvec
      double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                      solution(1:nunk_pel)
      double precision, intent(out) :: elem_vec(1:len_outvec)


! ***************************************************************************
!
!                         LOCAL PARAMETERS
!

      double precision beta(1:3), gamma(1:3), delta, dudx
      integer i, j


! =========================================================================
!
!      --- Compute the factors beta, gamma and delta as defined in NMSC

      delta = (x(2)-x(1))*(y(3)-y(1))-(y(2)-y(1))*(x(3)-x(1))
      beta(1) = (y(2)-y(3))/delta
      beta(2) = (y(3)-y(1))/delta
      beta(3) = (y(1)-y(2))/delta
      gamma(1) = (x(3)-x(2))/delta
      gamma(2) = (x(1)-x(3))/delta
      gamma(3) = (x(2)-x(1))/delta

!      --- Compute the derivative of the solution in the triangle

      dudx = 0d0
      do i = 1, 3
         dudx = dudx + solution(i)*beta(i)
      end do  ! i = 1, 3
```

```
!      --- Store the derivative into elem_vec

       elem_vec(1:3) = dudx

       end subroutine eldervsubr
```

The computation of the integral over the solution is a direct consequence of the Finite Element Method. The integral $\int_\Omega \phi d\Omega$ can be split into a sum of the same integrals per element. For that reason SEPRAN expects you to write a function subroutine `elintsubr` as described in Section 4.7.3. The output of this function subroutine, must be the integral defined over one element. This value must be assigned to the variable `elintsubr`, which is the name of the function. An example of the function subroutine is given by:

```
       function elintsubr ( npelm, x, y, nunk_pel, solution, itype, icheli )
!
!                      INPUT / OUTPUT PARAMETERS
!
       implicit none
       integer, intent(in) :: npelm, nunk_pel, itype, icheli
       double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                       solution(1:nunk_pel)
       double precision :: elintsubr
! ***********************************************************************
!
!                      LOCAL PARAMETERS
!
       double precision delta
       integer i, j

! =====================================================================
!
!      --- Compute delta as defined in the manual

       delta = (x(2)-x(1))*(y(3)-y(1))-(y(2)-y(1))*(x(3)-x(1))

!      --- Compute the integral and put into elintsubr

       elintsubr = 0d0
       do i = 1, 3
           elintsubr = elintsubr + solution(i)*abs(delta)/6d0
       end do  ! i = 1, 3

       end function elintsubr
```

Creating and compiling of these subroutines is not sufficient. You must also adapt the input file for the program sepcompexe. Compared to the example in Section 2.3.2 we have to add statements in the structure block for the computation of the derivative as well as for the computation of the integrals. Besides that the constants block (3.3) must be extended with the names of the derivative vector as well as a scalar name to store the integral.
An example of the input file is given by:

```
#*****************************************************************************
#
#     File:  lab4-4.prb
```

```
#
#       Contents:   Input for computational part of the example as described
#                   in the SEPRAN Lab manual 4.4
#
#       Usage:      sepcompexe < lab4-4.prb
#
#       It has been supposed that the following actions have been carried out
#       with success:
#
#       sepmesh lab.msh
#       compile elemsubr.f90
#       compile eldervsubr.f90
#       compile elintsubr.f90
#       seplink sepcompexe
#*****************************************************************************
#
#
#
#  Problem definition
#
problem
   types                                # Define type numbers per element group
      elgrp1 = (type=1)                 # Element group 1: itype = 1
      elgrp2 = (type=2)                 # Element group 1: itype = 2
   essboundcond                         # Define where essential boundary
                                        # conditions are defined (not there
                                        # value)
      curves(c1)                        # The potential on c1 is given
      curves(c5)                        # The potential on c5 is given
end
#
#  Define the structure of the main program, see Section 4.7.9
#
structure
  # Define the structure of the large matrix
   matrix_structure: symmetric   #  a symmetric profile matrix is used
  # Put the essential boundary conditions in the solution vector
   prescribe_boundary_conditions potential = 1, curves(c5)
  # Build and solve the system of linear equations
   solve_linear_system potential
  # Compute the x-derivatives of the potential
   dphi_dx = derivatives(potential, icheld = 1)
  # Compute the integral over the potential
   solint = integral (potential, icheli = 1)
  # Print the integral over the potential
   print solint
  # Rest of output
   plot_contour potential
   plot_coloured_levels potential
   plot_coloured_levels dphi_dx
end
end_of_sepran_input
```

To run this example you have to make the mesh as in example (2.3.2). After that you have to
compile all element subroutines and link them with the main program sepcompexe in the following

way:

```
compile elemsubr.f90
compile eldervsubr.f90
compile elintsubr.f90
seplink sepcompexe
```

If no errors are found, running is done by:

```
sepcompexe < lab4-4.prb
```

Remark:

The number of unknowns per point in this example is 1. By default also the derivative contains only one unknown per point. If you need to compute for example the gradient, you should add the following two statements to teh structure input block:

```
dphi_dy = derivatives(potential, icheld = 2)
gradphi = (dphi_dx,dphi_dy)
```

The parameter icheld in subroutine eldervsubr may be used to distinguish between both cases. The statement `gradphi = (dphi_dx,dphi_dy)` combines both vectors to one vector with two unknowns per point.

4.5   A mathematical test example showing the use of two unknowns per point

Consider the pure artificial problem:
$\Delta u + v = 0$, $\Delta v + u = 0$ $\mathbf{x} \in (0,1) \times (0,1)$
with boundary conditions:
$u = 0$ on curves c1, c2 and c4, $u = 1$ on curve c3 and $v = 0$ on curves c2, c3 and c4, $v = 1$ on curve c1.
The region is shown in Figure 4.2.1.

The special thing in this example is the use of 2 unknowns u and v per point.
The element matrix can be written in the following form:

$$\mathbf{S}^{e_k} = \left[\begin{array}{cc} \mathbf{S}^{uu} & \mathbf{S}^{uv} \\ \mathbf{S}^{vu} & \mathbf{S}^{vv} \end{array} \right] \tag{4.5.4}$$

where the matrix $\mathbf{S}^{uu}$ relates to the u-unknowns in the u-equations, the matrix $\mathbf{S}^{uv}$ relates to the v-unknowns in the u-equations and so on.
Using a standard Galerkin approach in combination with a Newton Cotes integration rule and linear triangles, the elements of the sub-element matrices can be written as:

$$\mathbf{S}^{uu}_{ij} = \mathbf{S}^{vv}_{ij} = \frac{|\Delta|}{2}(\nabla\phi_i \cdot \nabla\phi_j) \tag{4.5.5}$$

$$\mathbf{S}^{uv}_{ij} = \mathbf{S}^{vu}_{ij} = \frac{|\Delta|}{6}\delta_{ij} \tag{4.5.6}$$

Unfortunately the internal storage per element in SEPRAN is not $u_1, u_2, u_3, v_1, v_2, v_3$ but $u_1, v_1, u_2, v_2, u_3, v_3$. In order to fill the element matrix, the easiest way is to create the submatrices first and then to fill them in the correct sequence into the large matrix.

The student must create 3 files in this particular case:

```
lab4-5.msh
elemmsubr.f90
lab4-5.prb
```

The file `lab4-5.msh` contains the mesh input. The user can create this file himself with the information of Chapter 3.
The file `elemsubr.f90` contains the element subroutine. elemsubr.f90 must be compiled and linked to the program sepcompexe, like in Section 2.3.2.
Use the command `sepgetlab elemsubr` to get the template into your local directory. The file `lab4-5.prb` contains the input for the computational program.

The commands to be carried out are:

```
sepmesh lab4-5.msh
compile elemsubr.f90
seplink sepcompexe
sepcompexe < lab4-5.prb > outputfile
```

Mark that the next command may only be carried out if you are sure that the previous one has been finished successfully.

In this particular case we have to supply both an element subroutine `elemsubr` as well as a function subroutine `funcbc`, which is used to define the boundary conditions that depend on space. For simplicity we have stored both in the file lab4-5.f90

The file elemsubr.f90 has the following shape

```fortran
      subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                            elem_vec, elem_mass, prevsolution, itype )
!
!                      INPUT / OUTPUT PARAMETERS
!
      implicit none
      integer, intent(in) :: npelm, nunk_pel, itype
      double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                      prevsolution(1:nunk_pel)
      double precision, intent(out) :: elem_mat(1:nunk_pel,1:nunk_pel),  &
                                       elem_vec(1:nunk_pel),  &
                                       elem_mass(1:nunk_pel)
! **********************************************************************
!
!                      LOCAL PARAMETERS
!
      double precision :: beta(1:3), gamma(1:3), delta, suu(3,3),  &
                          suv(3,3), svu(3,3), svv(3,3)
      integer :: i, j
! ======================================================================
!
!     --- Compute the factors beta, gamma and delta as defined in NMSC

      delta = (x(2)-x(1))*(y(3)-y(1))-(y(2)-y(1))*(x(3)-x(1))
      beta(1) = (y(2)-y(3))/delta
      beta(2) = (y(3)-y(1))/delta
      beta(3) = (y(1)-y(2))/delta
      gamma(1) = (x(3)-x(2))/delta
      gamma(2) = (x(1)-x(3))/delta
      gamma(3) = (x(2)-x(1))/delta

!     --- Fill the element matrix as defined in the Lecture Notes
!         We start with the four submatrices

      do j = 1, 3
         do i = 1, 3
            suu(i,j) = 0.5d0 * abs(delta) *     &
               ( beta(i)*beta(j) + gamma(i)*gamma(j) )
         end do
      end do
      svv = suu
      suv = 0d0
      svu = 0d0

!     --- Fill the diagonal of the matrices suv and svu

      do i = 1, 3
         suv(i,i) = abs(delta)/6d0
         svu(i,i) = abs(delta)/6d0
      end do

!     --- Put the four submatrices in the element matrix in the
!         correct sequence, i.e. u1, v1, u2, v2, u3, v3

      do j = 1, 3
```

```
        do i = 1, 3
           elem_mat(2*i-1,2*j-1) = suu(i,j)
           elem_mat(2*i-1,2*j) = suv(i,j)
           elem_mat(2*i,2*j-1) = svu(i,j)
           elem_mat(2*i,2*j) = svv(i,j)
        end do
     end do

!    --- The element vector is zero

     elem_vec(1:6) = 0d0

     end subroutine elemsubr
```

Explanation: The subroutine elemsubr is almost identical to the one in Section 2.3.2. Extra are the submatrices and the copying into the 6×6 matrix.

The input file for this example may have the following shape

```
#*******************************************************************************
#
#      File:  lab4-5.prb
#
#      Contents:  Input for computational part of the example as described
#                 in the SEPRAN Lab manual 4.5
#
#      Usage:    sepcompexe < lab4-5.prb
#
#      It has been supposed that the following actions have been carried out
#      with success:
#
#      sepmesh lab4-5.msh
#      compile elemsubr.f90
#      seplink sepcompexe
#*******************************************************************************
#
#  Problem definition, see Section 4.7.1
#
problem
   types                                # Define type numbers per element group
      elgrp1 = (type=1)                 # Element group 1: itype = 1
      numdegfd = 2                      # There are 2 degrees of freedom
                                        # (number of unknowns) in each point
   essboundcond                         # Define where essential boundary
                                        # conditions are defined (not there
                                        # value)
     degfd1, degfd2, curves(c1 to c4)   # The vector potential is given
                                        # on all boundaries
end
#
#  Define the structure of the main program, see Section 4.7.9
#
structure
   matrix_structure: symmetric   #  a symmetric profile matrix is used
   prescribe_boundary_conditions potential = 1, curves(c3), degfd1
   prescribe_boundary_conditions potential = 1, curves(c1), degfd2
```

```
   solve_linear_system, potential
   plot_contour potential, degfd1, text='potential first component'
   plot_contour potential, degfd2, text='potential second component'
   plot_vector potential
end
end_of_sepran_input
```

Explanation:
Also this part is almost identical to the file in Section 2.3.2. There are two major differences:

- In the problem block we have added a line `numdegfd = 2` following `elgrp1 = (type=1)`. This indicates that the number of unknowns per point (number of degrees of freedom) is equal to 2 instead of 1. This implies that nunk_pel in the routine elemmsubr.f90 is equal to 6 for linear triangles. Furthermore, the line in essential boundary conditions is extended with `degfd1, degfd2` to indicate that both degrees of freedom are prescribed on the corresponding curves.

- in the `prescribe_boundary_conditions` in the structure block we first prescribe the first degree of freedom at curve c3 with the value one. This automatically sets all other values to zero In the next statement we set the second degree of freedom at curve c1 to one. The other values remain unchanged.

4.6   Description of the input for program SEPCOMP

Examples of input files have been treated in the previous sections. In this section we shall give some general rules for the input, followed by a minimal description of the input blocks separately. A very extended description can be found in the SEPRAN Users Manual, however, for the Lab there is no need to use that manual.

General rules

The input for program SEPCOMP is subdivided into a number of blocks. In the lab only two blocks are important: input block PROBLEM, which describes the problem definition and input block STRUCTURE, which gives the sequence in which the program must be carried out. Each of these blocks must begin with the name of the block and end with END. The end of the input is indicated by the keyword END_OF_SEPRAN_INPUT. The sequence is always first the PROBLEM block and then the STRUCTURE block.
A typical input for a problem will look like:

```
problem
  .
  .
  .
end
structure
  .
  .
  .
end
end_of sepran_input
```

In the next subsections the input of each of the blocks is described.

4.6.1   The main keyword PROBLEM

The block defined by the main keyword PROBLEM defines which problem is to be solved by program SEPCOMP. For each element group defined in SEPMESH the user must indicate what type of problem has to be solved. Problems are indicated by so-called type numbers.
For the lab you have to use type numbers between 1 and 99.
For each differential equation it is necessary to give boundary conditions. SEPRAN distinguishes between so-called essential boundary conditions and natural boundary conditions. An essential boundary condition is a boundary condition that prescribes unknowns at the boundary explicitly, natural boundary conditions in general give some information about derivatives or combinations of unknowns and derivatives at the boundary. Before using SEPRAN, the student himself must decide which boundary conditions are natural.

Natural boundary conditions sometimes require extra elements, the so-called boundary elements. These elements may be defined in the part PROBLEM as boundary elements.

The block defined by the main keyword PROBLEM has the following structure:

```
PROBLEM
   TYPES
     data corresponding to TYPES
   NATBOUNCOND
     data corresponding to NATBOUNCOND
   BOUNELEMENTS
     data corresponding to BOUNELEMENTS
   ESSBOUNDCOND
     data corresponding to ESSBOUNDCOND
   PERIODICAL_BOUNDARY_CONDITIONS
     data corresponding to PERIODICAL_BOUNDARY_CONDITIONS
END
```

The keywords PROBLEM, END and TYPES are mandatory. All subkeywords may be given in arbitrary order as long as they appear only once. The data corresponding to these subkeywords must be given immediately after the keywords themselves.
If the keyword NATBOUNCOND is given then also the keyword BOUNELEMENTS must be present.

Explanation of the subkeywords and description of the records:

PROBLEM (mandatory)
   opens the input for this block.

TYPES (mandatory)
   defines the problem definition numbers of the standard elements. Must be followed by records of the type:

```
     ELGRP 1 = (type = n1)
     ELGRP 2 = (type = n2)
     ELGRP i = (type = n3)
```

   with i the element group number. ni is the problem definition number of the ith element group.
   The element group number refers to the element group number defined in the mesh generation part. The number of element groups to be defined in this part TYPES must be exactly equal to the number of element groups defined in the mesh generation.
   The type number is used to define which type of problem must be solved. This type number is available in the element subroutine, where it can be used to distinguish between different

element types. For the lab only type numbers between 1 and 99 may be used.

If the number of degrees of freedom per point is not equal to 1 then each record with `ELGRP` must be followed by a record with

    NUMDEGFD = n

where n is the number of degrees of freedom per point in that element. For almost all exercises there is no need to give this statement.

NATBOUNCOND (optional)

indicates that standard boundary elements are used. Must be followed by data records of the type:

    BNGRP 1 = (type = n1)
    BNGRP i = (type = ni)

with i the boundary element group number and ni is the boundary problem number of the ith boundary element group.

The boundary element groups must be defined sequentially from 1. No boundary element group numbers may be skipped. The largest boundary element group number defines the number of boundary element groups (NUMNATBND).

Internally in the element subroutines the boundary groups get as element sequence number NELGRP + IBNGRP, where IBNGRP is the boundary element group sequence number and NELGRP is the number of element groups defined in the mesh generation.

BOUNELEMENTS (must only be used when NATBOUNCOND is used)

indicates that boundary elements are created. Must be followed by records of the following type:

    BELM1 = CURVES (C1 to C2)
    BELMi = CURVES (C5)

These records take care of the generation of boundary elements.

i is the boundary element group number; i may be used more than once. The boundary elements must be created with increasing boundary element group number.

When the boundary elements consist of curve elements, the function CURVES must be used, followed by the curve numbers.

C1 to C2: means that boundary elements are generated along the curves C1 to C2, when C2 is not given only curve C1 is used. When C2 is given, the curves C1 to C2 must be subsequent curves with coinciding initial and end point, i.e. the end point of C1 must be equal to the initial point of C1 + 1 etc.

For each boundary element group defined before it is necessary to create boundary elements.

ESSBOUNCOND (optional)

indicates that essential boundary conditions will be prescribed. In this part it is described in which positions we have essential boundary conditions and which unknowns are prescribed. However, the values of these boundary conditions are not yet given. They are described by either the separate command ESSENTIAL BOUNDARY CONDITIONS or by CREATE VECTOR. Both do not belong to the part PROBLEM. If, however, a degree of freedom is not identified as essential boundary condition in this part of the input, it will never become an essential boundary condition and values defined in other parts of the input given to these unknowns will never be recognized as essential boundary conditions.

This record must be followed by records of the type:

    CURVES (C1)
    DEGFD2 = CURVES (C2, C3)
    DEGFD1, DEGFD2, DEGFD3 = CURVES (C1 to C5)

DEGFDj indicates that the jth degree of freedom will be prescribed. The values are given in the block.

Hence DEGFD1, DEGFD3 indicates that the first and third degree of freedom in the corresponding nodal points are prescribed. When DEGFDj = is omitted all degrees of freedom are supposed to be prescribed in the corresponding nodal points.

When essential boundary conditions are given on curves the function CURVES must be used followed by the curve numbers for example C1 to C5 indicating that essential boundary conditions of this type are defined on the curves C1 to C5, or C1 only when C5 is omitted. When C5 is given, the curves C1 to C5 must be subsequent curves with coinciding initial and end point, i.e. the end point of C1 must be equal to the initial point of C1 + 1 etc.

PERIODICAL_BOUNDARY_CONDITIONS (optional)

is used if there are periodical boundary conditions. It must be followed by records of the shape

```
CURVES ( C2, C3 )
```

This means that the curves C2 and C3 are periodical boundaries and that each node on C2 is coupled (identified) to each node on C3 in the direction of the curves. If C2 and C3 are opposite curves with different direction one should use

```
CURVES ( C2, -C3 )
```

END (mandatory)

4.6.2 The main keyword STRUCTURE

The block defined by the main keyword STRUCTURE defines which actions should be performed by program SEPCOMP. In fact this block defines the complete structure of the main program.

In the block STRUCTURE it is precisely described which vectors and scalars are created, how they are created and in which sequence.
The block defined by the main keyword STRUCTURE starts with the command STRUCTURE at a separate record and ends with the keyword END on another separate record. In between commands may be given in any sequence and on separate records. However, the commands itself are carried out in exactly the sequence as given in this block. This means that the user himself is responsible for the correctness of the sequence of the commands. The only check that is performed is that vectors and scalars that are used as input have already been filled before.

The block STRUCTURE consists of a series of commands that may be repeated. The following types of commands may be used in the block STRUCTURE:
(options are indicated between the square brackets "[" and "]"):

```
STRUCTURE
  MATRIX_STRUCTURE, options
  PRESCRIBE_BOUNDARY_CONDITIONS vector = expression, options
  CREATE_VECTOR vector = expression, options
  SOLVE_LINEAR_SYSTEM vector
  vector = DERIVATIVES (name, options )
  vector = expression
  scalar = expression
  vector = (vector1, vector2)
  scalar = INTEGRAL (name, options)
  scalar = BOUN_INT (name, options)
  PRINT commands
  PLOT commands

  while ( ... ) do
  end_while

  if ( ... ) then
  end_if
END
```

Here `vector` stands for a vector name and `scalar` for a scalar name.
Mark that the input file is case insensitive except for texts between quotes. Hence the use of capitals in the previous part is only to emphasize the commands.

Commands may be repeated and given in any order. However, they are executed in exactly the sequence given in the block which means that this sequence defines the complete program and hence must be logical. So it is for example necessary to prescribe the boundary conditions first and then to solve the system of linear equations, since otherwise the effect of the essential boundary conditions to the solution is not present and the solution may be undefined.

These commands have the following meaning:

STRUCTURE (mandatory) This keyword indicates the start of the input block STRUCTURE. All records following it until the record END is found define the complete structure of the program.

MATRIX_STRUCTURE options
    This command is only necessary if the matrix that you create has a structure that is either

symmetric or should be stored as a compact matrix, since an iterative linear solver is required. The following options are available:

```
storage_scheme = i, symmetric
```

STORAGE_SCHEME $= i$ gives information of the structure of the large matrix. Depending on the value of i the system of equations is solved by a direct solution method (Gaussian elimination) or by an iterative method.
Possible values for i are:

PROFILE The matrix is stored as a so-called profile matrix, which implies that a direct solution method is used. This is the default value, so in this case `STORAGE_SCHEME` may be skipped

COMPACT The matrix is stored as a so-called COMPACT matrix, which implies that an iterative solution method is used. This is only necessary in case of very large problems, or if the matrix is singular due to boundary conditions.

SYMMETRIC indicates that the matrix is symmetric. Only one half of the matrix is stored. If omitted the matrix is supposed to be unsymmetric.

PRESCRIBE_BOUNDARY_CONDITIONS name = expression , options
With this command the vector `name` is provided with essential boundary conditions.
The result of this operation is that the vector `name` has been filled or changed.
`expression` defines the value that is used to fill the boundary condition. This may be a constant, for example 0, but also a vector like `sin(x_coor)`, where `x_coor` and `y_coor` represent the x and y coordinates respectively.
The options consist of a part describing the place where the essential boundary conditions are prescribed, a part which degrees of freedom are prescribed and a part how the degrees of freedom are prescribed. A typical example is

```
CURVES ( C1 to C5 ), DEGFD1
```

If omitted the whole vector is filled.

CURVES i **(C1 to C5)** indicates that essential boundary conditions are prescribed on the curves C1 to C5. Also a set of curves is allowed.

DEGFDi = means that the ith degree of freedom is prescribed by this record. If omitted, all degrees of freedom are prescribed.

The statement `prescribe_boundary_conditions` may be used repeatedly, in order to give different values for the boundary conditions for different parts of the boundary.

SOLVE_LINEAR_SYSTEM [name]
The command solve_linear_system performs actually two independent steps.
Firstly the matrix and right-hand-side vector is built. Finally the system of linear equations is solved by the linear solver. Before applying the command solve_linear_system it is necessary that the essential boundary conditions have already been filled into the solution vector `name`. `name` must be a solution vector.
The result of the total operation is that `name` has been filled with the solution of a linear differential equation.

CREATE_VECTOR name, options
The command create_vector has the same meaning as PRESCRIBE_BOUNDARY_CONDITIONS,

name = DERIVATIVES (options)
The command derivatives may be used to create the vector `name` as derived quantity of previously constructed vectors.
The following options are available (in one line)

```
        name_vec
        icheld = k
```

name_vec defines the name of the vector from which the derivative is computed.

ICHELD $= k$ defines the type of derived quantity to be computed. This parameter is passed undisturbed to the element subroutine ELDERVSUBR see Section 4.7.2. This parameter may be used to distinguish between several possibilities.
The default value for ICHELD is 1.

The result of this operation is that a vector `name` has been created.

scalarname = expression gives the scalar (variable) with name scalarname the value of the expression. The expression may contain variables and constants as defined before.

scalarname = INTEGRAL (name, options)
The command integral may be used to compute scalar *scalarname* as integral over vector `name`. The result of this operation is that the scalar scalarname has got a value.
The following options are available (in one line)

```
        icheli = i
```

ICHELI $= k$ defines the type of integral to be computed. This parameter is passed undisturbed to the element subroutine ELINTSUBR see Section 4.7.3. This parameter may be used to distinguish between several possibilities.
The default value for ICHELI is 1.

scalarname = BOUN_INT (name, options)
The command boundary integral may be used to compute scalar *scalarname* as boundary integral over vector `name`. The result of this operation is that the scalar scalarname has got a value.
The following options are available (in one line)

```
        curves = (ci, cj, ck )
```

CURVES $= ci, cj, ck$ defines the curves over which this boundary integral must be computed.
The default value for ICHELI is 1.

vectorname = (vectorname1, vectorname2) creates a vector with components vectorname1 and vectorname2. For example if you want to compute the velocity defined by $\mathbf{u} = (\frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y})$, you may first define the vectors `dphidx` and `dphidy` using the command `DERIVATIVES` and compute the vector `velocity` by

```
        velocity = (dphidx,dphidy)
```

PRINT and PLOT commands See Section (4.6.3)

while (expr) do starts a while loop, where `expr` is a boolean expression. You can use the operators `< > <= >=` and `and` and `or`.

end_while ends the while loop.

END (mandatory) Indicates the end of the STRUCTURE block.

4.6.3   The print and plot commands in the STRUCTURE block

The following print and plot options are available in the structure block.

```
PRINT scalarname [text='some text']
PRINT name [options]
PRINT 'text between quotes'
PLOT_CONTOUR name [options]
PLOT_COLOURED_LEVELS name [options]
PLOT_VECTOR name [options]
PLOT_3D name [options]
PLOT_FUNCTION name   [options]
PLOT_INTERSECTION name  [options]
```

PRINT scalarname [text='some text']
> The command PRINT prints the value of scalarname to the output file. If text is given it should be followed by some text between quotes. This text is used to identify the scalar to be printed in the following way:
>
>> text = scalarname
>
> where scalarname denotes the value of scalarname.

PRINT name [options]
> The command PRINT prints the value of name to the output file. The following options are available:
>
>> text = 't'
>> curves = c1, c2, cn, ...
>
> These options have the following meaning:
>
> **text** should be followed by some text between quotes. This text is used to identify the vector to be printed.
>
> **curves** followed by Ci, Cj, Ck, ... ensures that the printing of the solution is restricted to the curves given in the list.
>
> Remarks:

PRINT , 'text between quotes'
> The command PRINT prints the text between the quotes to the output file.

PLOT_CONTOUR name [options]
> indicates that contour lines (lines with constant function value) are plotted for the given function. The following options are available
>
>> text = 't'
>> degfd = k
>
> These options have the following meaning:
>
> **degfd** $= k$ the kth degree of freedom in each node is used as definition of the function, otherwise the first degree of freedom is used.
>
> **text** $=$ **'t'** the text t between the quotes is plotted at the bottom of the picture.

PLOT_COLOURED_LEVELS name [options]
> makes a colored contour plot of the vector name, where the region between two levels is colored.
> The options are the same as for `plot_contour`

PLOT_VECTOR name [options]

    makes a vector plot of two of the degrees of freedom in each point. These components may be defined by degfd1 $= k_1$, degfd2 $= k_2$ respectively. If omitted degfd1 $= 1$, and degfd2 $= 2$ is assumed.

PLOT_3D name [options]

    makes a three-dimensional plot with hidden lines of a scalar function defined on a two-dimensional mesh.

    The options are the same as for `plot_contour`

PLOT_FUNCTION name [options]

    makes a plot of a one dimensional function. This may be a function defined on a one-dimensional mesh or on a set of curves. The following options are available

```
textx = 'tx'
texty = 'ty'
degfd = k
curves ( ci, cj, ... )
```

These options have the following meaning:

degfd $= k$ the kth degree of freedom in each node is used as definition of the function, otherwise the first degree of freedom is used.

text $=$ **'tx'** the text tx between the quotes is plotted along the x-axis.

text $=$ **'ty'** the text ty between the quotes is plotted along the y-axis.

curves (ci, cj, ...) defines the set of curves.

PLOT_INTERSECTION name [options]

    makes a plot of a one dimensional function that is created by intersection of the mesh with a straight line. The following options are available

```
textx = 'tx'
texty = 'ty'
degfd = k
origin = (O_x,O_y)
angle = a
```

These 3 options are the same as for `plot_function` the other two define the intersection. Its origin is given by (O_x, O_y) and the angle with respect to the x-axis is given by a.

4.7   How to program your own element subroutines

In the Numerical Analysis Lab the student must program his own elements. Of course most of the exercises can be made with standard SEPRAN element subroutines using type numbers larger than 99. However, programming your own element is the main goal of the lab.

It is always necessary to program your own element subroutine ELEMSUBR, that is to be used both in the case of a linear and of a non-linear problem. See Section 4.7.1 for a description.

If derived quantities like a first derivative must be computed it is also necessary to program an element subroutine ELDERVSUBR. See Section 4.7.2 for a description.

If integrals must be computed it is necessary to program an element subroutine ELINTSUBR. See Section 4.7.3 for a description.

The Sections 4.7.4, 4.7.5 and 4.7.6 describe some help subroutine to simplify the printing of arrays. The general idea is the following:

If the large matrix is built a subroutine BUILD is called that makes a loop over all elements. For each element the element subroutine ELEMSUBR is called. This subroutine is supposed to compute the element matrix and element vector. BUILD then adds this element matrix and element vector to the large matrix and vector in the right positions.

In the same way a loop over the element subroutines is performed for the integration and derivative subroutines.

4.7.1   Subroutine ELEMSUBR

Description

Subroutine ELEMSUBR is called by a subroutine that builds the large matrix and vec-
tor.
This subroutine is only used for type numbers between 1 and 99, hence for the Numer-
ical Analysis Lab this subroutine is obliged.

Use the command `sepgetlab elemsubr` to get the correct interface in your local direc-
tory. See Section 2.3.2

The general structure of the matrix building is as follows:

```
clear large matrix and large vector
For all element groups and all boundary element groups do
   For all elements in the group do
      call ELEMSUBR
      add element matrix and element vector to large matrix and large vector
   end_For
end_For
```

Heading

```
subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                      elem_vec, elem_mass, prevsolution, itype )
```

Parameters

INTEGER npelm, nunk_pel, itype

DOUBLE PRECISION x(1:npelm), y(1:npelm), prevsolution(1:nunk_pel)

DOUBLE PRECISION elem_mat(1:nunk_pel,1:nunk_pel), elem_vec(1:nunk_pel), elem_mass(1:nunk_pel)

NPELM (input parameter)
Defines the number of points in the element. So for a linear triangle NPELM = 3,
and for a linear boundary element NPELM = 2.

X (input array)
Double precision array of size NPELM. Contains the x-coordinate of the ith node
in the element.
Mark that it concerns the local numbering of the element, not the global node
numbers.

Y (input array)
Same as X but now for the y-coordinates.

NUNK_PEL (input parameter)
Defines the number of degrees of freedom in the element.
Usually this number is equal to NPELM, but for example, if the number of degrees
of freedom per point is 2, it is $2 \times$ NPELM.

ELEM_MAT (output array)
In this double precision two-dimensional array the student must store the element
matrix, in the following way:
        ELEM_MAT(i,j) = s_{ij} ; i,j = 1(1)NUNK_PEL.
The degrees of freedom in an element are stored sequentially, first all degrees of
freedom corresponding to the first point, then to the second, etcetera.
The local sequence of the nodes is defined by Table 3.1.1.

ELEM_VEC (output array)

In this double precision array the student must store the element vector, in the following way:

ELEM_VEC(i) = f_i ; i = 1(1)NUNK_PEL.

ELEM_MASS (output array)

In this double precision two-dimensional array the student must store the element mass matrix, provided the mass matrix must be computed, in the following way:

ELEM_MASS(i,j) = s_{ij} ; i,j = 1(1)NUNK_PEL.

This matrix should only be filled if a mass matrix is required, for example for time-dependent problems.

PREVSOLUTION (input array)

In this array the previous solution is stored. This solution may contain the boundary conditions only, if the array has been created by prescribe_boundary_conditions, but also a starting vector if created by create_vector.

The sequence in which PREVSOLUTION is filled is the same as used in X and ELEM_MAT. Hence first all degrees of freedom for the first local point, then for the second one and so on.

This array is only used in case of non-linear problems.

ITYPE (input parameter)

This parameter defines the type number of the element. This type number has been defined in the input block PROBLEM as part of the statements:

```
ELGRP i = (type = n3)
BNGRP 1 = (type = n1)
```

The student may utilize ITYPE to distinguish between different types of element matrices, for example to distinguish between internal elements and boundary elements.

Input

Program SEPCOMP fills the arrays X, Y and array PREVSOLUTION before the call of ELEMSUBR.

Also the parameters NPELM, NUNK_PEL and ITYPE have got a value.

Output

The student must fill the arrays ELEM_MAT, ELEM_VEC and in case of time-dependent problems ELEM_MASS.

Interface

Subroutine ELEMSUBR must be programmed as follows:

```
      subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                            elem_vec, elem_mass, prevsolution, itype )
      implicit none
      integer, intent(in) :: npelm, nunk_pel, itype
      double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                      prevsolution(1:nunk_pel)
      double precision, intent(out) :: elem_mat(1:nunk_pel,1:nunk_pel),  &
                                       elem_vec(1:nunk_pel),  &
                                       elem_mass(1:nunk_pel)

!     --- declarations of local variables
```

```
!            for example:

         integer :: i, k

         if ( itype==1 ) then

!        --- statements to fill the arrays elem_mat and elem_vec

            do k = 1, nunk_pel
               do i = 1, nunk_pel
                  elem_mat(i,k) = "s(ik)"
               end do
               elem_vec(k) = "f(i)"
            end do

         else if ( itype==2 ) then

!        --- the same type of statements for itype = 2, etcetera

         end if
         end subroutine elemsubr
```

Remarks

- Almost all the errors that are made by students are in the subroutine ELEMSUBR.
  Since debugging of Fortran programs goes beyond the goal of the lab, it is advised
  to use print statements in the element subroutine to detect errors.
  For example to print the value of a variable `var` use

  ```
  print *, 'var = ', var
  ```

  To print the contents of a double precision array for example the element vector
  the SEPRAN subroutine PRINTREALARRAY may be used, see Section 4.7.4, to
  print the contents of an integer array: PRINTINTEGERARRAY, see Section 4.7.5.
  To print the contents of the element matrix use PRINTMATRIX, see Section 4.7.6.

  These subroutines may be for example used as follows:

  ```
          subroutine elemsubr ( npelm, x, y, nunk_pel, elem_mat,  &
                                elem_vec, elem_mass, prevsolution, itype )
          implicit none
               .
               .
               .
               .
          call printrealarray ( elem_vec, nunk_pel, 'element vector' )
          call printmatrix ( elem_mat, nunk_pel, 'element matrix' )
          end subroutine elemsubr
  ```

- **Constants from the input file.**
  Constants that are defined in the input file are not known in the element subroutine.
  However, there is a simple way to introduce them into the subroutine. This is
  done by a function subroutine GETCONST with one parameter: the name of the
  constant between quotes. The result is the value of the constant. Of course it is
  necessary to declare the constant as well as the function subroutine as a double
  precision:

  ```
          subroutine elemsubr ( ndim, npelm, x, nunk_pel, elem_mat,  &
  ```

```
                              elem_vec, elem_mass, uold, itype )
        implicit none
            .
            .
        double precision :: getconst, mu
            .
            .
            .
        end subroutine elemsubr
```

In this case mu is the constant. The call of getconst inside elemsubr is:

```
        mu = getconst ( 'mu' )
```

In case the constant is integer, use GETINT, which of course must be declared integer.

4.7.2   Subroutine ELDERVSUBR

Description

Subroutine ELDERVSUBR is called by a subroutine which builds a large vector by averaging over adjacent elements.
This subroutine is only used for type numbers between 1 and 99. It is called if and only if the option derivatives is used in the STRUCTURE block.

Use the command `sepgetlab eldervsubr` to get the correct interface in your local directory.
The general structure of the derivatives subroutine is as follows:

```
clear large vector
For all element groups do
   For all elements in the group do
      call ELDERVSUBR
      add element vector to large vector
   end_For
end_For
Use an averaging procedure to compute the derivates per node
```

Heading

```
subroutine eldervsubr ( npelm, x, y, nunk_pel, elem_vec,  &
                        solution, itype, icheld, len_outvec )
```

Parameters

INTEGER NPELM, NUNK_PEL, ITYPE, ICHELD, LEN_OUTVEC

DOUBLE PRECISION X(1:NPELM), Y(1:NPELM), ELEM_VEC(1:LEN_OUTVEC), SOLUTION(NUNK_PEL)

NPELM, NUNK_PEL, ITYPE, X, Y See subroutine elemsubr (4.7.1)

LEN_OUTVEC (input parameter)
Defines the number of degrees of freedom in the OUTPUT VECTOR ELEM_VEC. Usually this number is equal to NPELM, but for example, if the number of degrees of freedom per point is 2, it is $2 \times$ NPELM.

ELEM_VEC (output array)
In this double precision array the student must store the element vector, in the following way:
$$\text{ELEM\_VEC}(i) = f_i \ ; \ i = 1(1)\text{LEN\_OUTVEC}.$$
It concerns the derived quantity that must be computed.
The sequence that must be used in case of more unknowns per point, like for example when computing the gradient, is:
First all unknowns in the first point of the element, followed by all unknowns in the second point and so on. In case the output vector has more degrees of freedom per point than the input vector the length of the arrays `elem_vec`.
For example if the solution vector is a potential with one degree of freedom per point then NUNK_PEL = 3, for a linear triangle. If the output vector is the gradient with 2 degrees of freedom per point then elem_vec have length 6. elem_vec(1) refers to the x-derivative in the first local node of the element, elem_vec(2) to the y-derivative, elem_vec(3) to the x-derivative in the second local node and so on.

SOLUTION (input array)

In this array the solution from which the derived quantities must be computed, as indicated by Vi, is stored.

The sequence in which SOLUTION is filled is the same as used in X and ELEM_VEC. Hence first all degrees of freedom for the first local point, then for the second one and so on.

In SOLUTION the value in the vertices are stored.

ICHELD (input parameter)

This is the input parameter defined in the statement `DERIVATIVES name, icheld = ...` in the input block STRUCTURE. This parameter may be used to distinguish between possibilities.

Input

Program SEPCOMP fills the arrays X, Y and array SOLUTION before the call of EL-DERVSUBR.

Also the parameters NDIM, NPELM, NUNK_PEL, ITYPE and ICHELD have got a value.

Output

The student must fill array ELEM_VEC.

Interface

Subroutine ELDERVSUBR must be programmed as follows:

```fortran
      subroutine eldervsubr ( npelm, x, y, nunk_pel, elem_vec,  &
                              solution, itype, icheld, len_outvec )
      implicit none
      integer, intent(in) :: npelm, nunk_pel, itype, icheld, len_outvec
      double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                      solution(1:nunk_pel)
      double precision, intent(out) :: elem_vec(1:len_outvec)

!     ---  declarations of local variables
!          for example:

      integer :: i

      if ( itype==1 ) then

!     --- statements to fill the arrays elem_vec

         do i = 1, len_outvec
            elem_vec(i) = "f(i)"
         end do

      else if ( itype==2 ) then

!     --- the same type of statements for itype = 2, etcetera

      end if
      end subroutine eldervsubr
```

4.7.3   Function subroutine ELINTSUBR

Description

Function subroutine ELINTSUBR is called by a subroutine which computes the integral
over a region by adding integrals over elements.
This subroutine is only used for type numbers between 1 and 99. It is called if and only
if the option integrals is used in the STRUCTURE block.

Use the command `sepgetlab elintsubr` to get the correct interface in your local di-
rectory.

The general structure of the integral subroutine is as follows:

```
sum := 0
For all element groups do
   For all elements in the group do
      sum := sum + ELINTSUBR (...)
   end_For
end_For
```

Heading

```
function elintsubr ( npelm, x, y, nunk_pel, solution, itype, icheli )
```

Parameters

INTEGER NPELM, NUNK_PEL, ITYPE, ICHELI

DOUBLE PRECISION X(1:NPELM), y(1:NPELM), ELINTSUBR, SOLUTION(NUNK_PEL)

ELINTSUBR (output parameter)
> The student must give elintsubr the value of the integral over the element to be
> computed.

NPELM, NUNK_PEL, ITYPE, X, Y See subroutine elemsubr (4.7.1)

SOLUTION (input array)
> In this array the function to be integrated, as indicated by Vi, is stored.
> The sequence in which SOLUTION is filled is the same as used in X. Hence first
> all degrees of freedom for the first local point, then for the second one and so on.

ICHELI (input parameter)
> This is the input parameter defined in the statement `scalarname = INTEGRAL, icheli = ...`
> in the input block STRUCTURE. This parameter may be used to distinguish be-
> tween possibilities.

Input

Program SEPCOMP fills the arrays X, Y and array SOLUTION before the call of
ELINTSUBR.
Also the parameters NPELM, NUNK_PEL, ITYPE and ICHELI have got a value.

Output

The student must give ELINTSUBR a value.

Interface

Function subroutine ELINTSUBR must be programmed as follows:

```
function elintsubr ( npelm, x, y, nunk_pel, solution, itype, icheli )
implicit none
integer, intent(in) :: npelm, nunk_pel, itype, icheli
double precision, intent(in) :: x(1:npelm), y(1:npelm),  &
                                solution(1:nunk_pel)
double precision :: elintsubr

!     ---  declarations of local variables

      if ( itype==1 ) then

!     --- statements to compute the integral over the element

         elintsubr = ..

      else if ( itype==2 ) then

!     --- the same type of statements for itype = 2, etcetera

      end if
      end function elintsubr
```

4.7.4   Subroutine PRINTREALARRAY

Description

Subroutine PRINTREALARRAY is a special subroutine that is meant to print double precision arrays inside a user written element subroutine.

Heading

```
subroutine printrealarray ( array, n, text )
```

Parameters

INTEGER N

DOUBLE PRECISION ARRAY(N)

CHARACTER $* (*)$ TEXT

N (input parameter)
　　Defines the length of ARRAY.

ARRAY (input array)
　　Double precision array of size N to be printed.

TEXT (input parameter)
　　Text to be printed in the heading of the print.

Input

The parameters N and TEXT must have a value.
Array ARRAY must have been filled.

Output

The contents of array ARRAY are printed

4.7.5 Subroutine PRINTINTEGERARRAY

Description

Subroutine PRINTINTEGERARRAY is a special subroutine that is meant to print integer arrays inside a user written element subroutine.

Heading

    subroutine printintegerarray (iarray, n, text)

Parameters

INTEGER N, IARRAY(N)

CHARACTER $* (*)$ TEXT

N (input parameter)
Defines the length of IARRAY.

IARRAY (input array)
Integer array of size N to be printed.

TEXT (input parameter)
Text to be printed in the heading of the print.

Input

The parameters N and TEXT must have a value.
Array IARRAY must have been filled.

Output

The contents of array ARRAY are printed

4.7.6   Subroutine PRINTMATRIX

Description

Subroutine PRINTMATRIX is a special subroutine that is meant to print two-dimensional double precision arrays inside a user written element subroutine.

Heading

```
subroutine printmatrix ( array, n, text )
```

Parameters

INTEGER N

DOUBLE PRECISION ARRAY(N,N)

CHARACTER $*$ ($*$) TEXT

N (input parameter)
    Defines the length of ARRAY.

ARRAY (input array)
    Two dimensional double precision array of size N $\times$ N to be printed.

TEXT (input parameter)
    Text to be printed in the heading of the print.

Input

The parameters N and TEXT must have a value.
Array ARRAY must have been filled.

Output

The contents of array ARRAY are printed

6   Index