A FEM Algorithm in Octave

Version 0.1.4, August 22, 2002
Andreas Stahel

Contents

1 Basic description of the problem

This code can be used to solve elliptic, second order partial differential equations (PDE) on a domain $\Omega \subset \mathbb{R}^2$ with boundary $\Gamma = \Gamma_1 \dot\cup \Gamma_2$. For given functions a, b, f, gD and gN an approximate solution u of the boundary value problem (BVP)

$$\begin{aligned} \operatorname{div}(a \operatorname{grad} u) - b\,u &= f && \text{in} \quad \Omega \\ u &= gD && \text{on} \quad \Gamma_1 \\ a \tfrac{\partial u}{\partial n} &= gN && \text{on} \quad \Gamma_2 \end{aligned} \tag{1}$$

is computed. On a triangularization of the domain Ω a piecewise linear approximation of the exact solution is used. The mathematical background can be found in many books, e.g. [John87], [Loga92] and [Redd84]. The implementation is based on a set of lecture notes by this author (see [VarFem]).

The goal is to provide a set of *Octave* commands to solve the above problem. The commands are listed in table 1. For teaching purposes a full set of commands to compute element stiffness matrices, construct the system of linear equations and examine the solution is given. A set of sample problems is included and might help to illustrate the commands. More examples are given in [VarFem].

This code was developed and tested on a variety UNIX systems. Dynamically linked *Octave* functions are used to obtain shorter computation times. This author is not aware on any system dependent features.

Similar code was implemented in *Mathematica* and can be obtained through the authors home page[1] or at the Wolfram mathsource site[2]. The runtime performance under *Mathematica* is rather poor.

2 FEM commands

A list of all commands is given in table 1. The following section give a short description of each command.

[1] `http://www.hta-bi.bfh.ch/~sha`
[2] `http://www.mathsource.com`

Creating, reading and visualizing a mesh	
`ReadMesh()`	reading mesh information from *EasyMesh* files
`ReadMeshTriangle()`	reading mesh information from *triangle* files
`CreateRectMesh()`	create e rectangular mesh
`CreateEasyMesh()`	create a mesh by calling *EasyMesh*
`CreateMeshTriangle()`	create a mesh by calling *triangle*
`ShowMesh()`	visualize the mesh.
Creating and solving the system of linear equations, evaluate functions	
`FEMEquation()`	set up the system of linear equations
`FEMSolveSym()`	solve the system of linear equations, banded symmetric solver
`FEMEig()`	find eigenvalues and eigenfunctions
`FEMValue()`	evaluate the solution and its gradient at given points
`FEMGradient()`	evaluate the gradient at the nodes of the mesh
`FEMIntegrate()`	integrate a function over the domain
Visualization of the solution	
`ShowSolution()`	visualize the solution with the help of *Gnuplot*
`ShowLevelCurves()`	show the level curves of a function
`ShowVectorField()`	visualize the gradient of the solution as vector field
`ShowSolutionMTV()`	create an input file for *plotmtv*
Functions for education purposes	
`ReadMeshM()`	reading mesh information from *EasyMesh* files
`ReadMeshTriangleM()`	reading mesh information from *triangle* files
`ElementContribution()`	find element stiffness matrix and the element vector
`ElementContributionEdge()`	find contribution of one edge
`FEMEquationM()`	set up the system of linear equations
`FEMSolve()`	solve the system of linear equations, full matrix
`FEMValueM()`	evaluate the solution and its gradient at given points
`ShowSolutionM()`	visualize the solution
Operations for symmetric, banded matrices	
`SBSolve()`	solve a system of linear equations
`SBFactor()`	find the $R^T DR$ factorization
`SBBacksub()`	use back-substitution to solve system of equations
`SBEig()`	find a few of the smallest eigenvalues and eigenvectors
`SBProd()`	multiply symmetric banded matrix with full matrix
`FullToBand()`	convert a symmetric matrix to a banded matrix
`BandToFull()`	convert a banded matrix to a symmetric matrix
`BandToSparse()`	convert a banded matrix to a sparse matrix

Table 1: List of commands

2.1 Reading the mesh information and generating meshes

Currently the mesh may be be generated by *EasyMesh*[3]. Using one input file (e.g. `test.d`) three output files are generated by *EasyMesh*, with information on nodes (`test.n`), elements (`test.e`) and edges (`test.s`).

Another option is to use the code *triangle*, written by Jonathan Richard Shewchuk[4]. The source is included the distribution of *FEMoctave*. Since *triangle* does not number its nodes to minimize the bandwidth of the matrix the additional utility *CuthillMcKee* is provided. Sample applications are shown in demos 5, 8 and 10 and its `Makefile`. Some more documentation can be found on this authors home page at [www:sha].

2.1.1 `ReadMesh`

If the mesh is generated by *EasyMesh*, then the files are read by the command `ReadMesh()` to make the information available to *Octave*, i.e. read the variables `nodes`, `elem` und `edges` by calling `[nodes,elem,edges]=ReadMesh("test")`. The directory of the files can be be given as part of the filename. The type of boundary conditions have to be given in these mesh descriptions. Dirichlet conditions are of type 1 and Neumann conditions of type 2. The on-line help on `ReadMesh()` gives some more information.

```
[...] = ReadMesh (...)
  reads the information of EasyMesh output into Octave data structures

[nodes,elem,edges]=ReadMesh(filename);
   the files filename.n filename.e filename.s are read
   these files have to be generated first by 'EasyMesh filename'
   EasyMesh reads filename.d for the description of the domain
   The boundary markers in filename.d lead to the following boundary conditions
     1 leads to a Dirichlet condition
     2 leads to a Neumann condition

the matrices nodes elem edges describe the mesh

nodes contains the x and y coordinates of the nodes and the material
   type at each node, i.e. nodes =[x1,y1,m1;x2,y2,m2;...;xn,yn,mn]
elem contains the information about the elements
   One row shows the numbers of the three nodes forming the element and
   then the material type of the element
edges contains the information about the boundary segments
   One row shows the numbers of the two nodes forming the segment and
   then the boundary markers is shown (1=Dirichlet, 2=Neumann)

Currently material information ignored
```

2.1.2 `ReadMeshTriangle`

If the mesh is generated by *triangle*, then the files are read by the command `ReadMeshTriangle()` to make the information available to *Octave*, i.e. read the corresponding variables with the help of `[nodes,elem,edges]=ReadMeshTriangle("test.1")`. One should definitely use *CuthillMcKee* to renumber the nodes, see `./demos/demo5`.

[3] `http://www-dinma.univ.trieste.it/~nirftc/research/easymesh/`

[4] Information can be found at `http://www.cs.cmu.edu/~quake/triangle.html`

2.1.3 CreateRectMesh

With the command `CreateRectMesh()` a rectangular mesh can be generated, without the help of *EasyMesh* or *triangle*.

```
[...] = CreateRectMesh(...)
  generate a rectangular mesh

[nodes,elem,edges] = CreateRectMesh(x,y,blow,bup,bleft,bright)
nodes elem edges  contain the information on the mesh

x y  vectors containing the coordinates of the nodes
  a typical vertex is (x(j), y(i))

blow,bup,bleft,bright indicate the type of boundary condition
    at lower, upper, right and left edge of rectangle
  b*=1  Dirichlet condition
  b*=2  Neumann condition
```

An example is shown in demo 3.

2.1.4 CreateEasyMesh

If the domain to be meshed is enclosed by a simple curve, then the command `CreateEasyMesh` can be used. The nodes forming the curve and the type of boundary conditions, together with the typical length of the sides of the triangles and the filename, are given as input parameters. The command generates an input file for *EasyMesh* and then calls *EasyMesh*, thus *EasyMesh* needs to be installed for this to work. An example is shown in demos 8 and 11 .

```
[...] = CreateEasyMesh(...)
generate a mesh using EasyMesh

CreateMeshTriangle(name,xy,len)
 name the base filename: the file name.d will be generated
      then EasyMesh will generate files name.* with the mesh

 xy   vector containing the coordinates of the nodes forming the
      outer boundary. Currently no holes can be generated. The format is
      [x1,y1,t1;x2,y2,t2;...;xn,yn,tn] where
      xi x-coordinate of node i
      yi x-coordinate of node i
      ti boundary marker for segment from node i to node i+1
        bi=1  Dirichlet condition
        bi=2  Neumann condition
      the last given node will be connected to the first given node
      to create a closed curve

 len  vector with the typical length of triangle side at a point
      if len is a scalar the same length will be used for all points

 The information can then be read and used by
    [nodes,elem,edges]=ReadMesh("name");
```

2.1.5 CreateMeshTriangle

If the domain to be meshed is enclosed by a simple curve, then the command `CreateMeshTriangle` can be used. The nodes forming the curve and the type of boundary conditions, together with the maximal area of the triangles and the filename, are given as input parameters. The command generates an input file for *triangle* and then calls *triangle*, thus *triangle* needs to be installed for this to work. In addition *CuthillMcKee* is called to assure a band structure of the resulting matrix. Examples are shown in demos 6, 7, 8 9 and 11 .

```
[...] = CreateMeshTriangle(...)
generate a mesh using triangle

[...] = CreateMeshTriangle(...)
 generate a mesh using triangle
 CreateMeshTriangle(name,xy,area)
  name the base filename: the file name.poly will be generated
       then triangle will generate files name.1.* with the mesh

  xy   vector containing the coordinates of the nodes forming the
       outer boundary. Currently no holes can be generated. The format is
       [x1,y1,t1;x2,y2,t2;...;xn,yn,tn] where
       xi x-coordinate of node i
       yi x-coordinate of node i
       ti boundary marker for segment from node i to node i+1
         bi=1  Dirichlet condition
         bi=2  Neumann condition
       the last given node will be connected to the first given node
       to create a closed curve

  area  a scalar given the maximal area of the triangles to be generated

  The information can then be read and used by
     [nodes,elem,edges]=ReadMeshTriangle("name.1");
```

This simple script file only covers rather elementary situations. *triangle* allows for many more options to be specified and its web page show how to use them. To generate good meshes we clearly recommend to use *triangle* directly. An example of this is show in demo 8 where we compute the capacitance of a conductor.

2.1.6 ShowMesh

For a visual control of the mesh use `ShowMesh()`. With `ShowMesh(nodes,elem)` a temporary file will be written to the disk and then *Gnuplot* is called to show the mesh.

2.2 Setting up and solving the system of linear equations

In this section we try to give a brief explanation of the command used to convert the boundary value problem in equation (1) into a system of linear equations.

2.2.1 Writing the function files

The examples in `./demos/*` show different techniques to implement the functions.

To solve the boundary value problem (1) the functions a, b, f, gD and gN have to be given. They can be given by one constant or by a vector of values at the nodes of the mesh. One can also

implement the functions in script files, as function files ('*.m') or as dynamically linked functions ('*.oct'). The function accept a matrix with x and y coordinates of points as arguments and return a vector with the values of the function as result. As an example consider an implementation of the function $a(x, y) = 1 + x$. Calling `a([1,2;3,4;5,-6])` should return the answer `[2;4;6]` . The code below has to be in a file `a.m` .

```
function res = a(xy)
  [n,m]=size(xy);
  res=zeros(n,1);
  for k=1:n
    res(k)=1+xy(k,1);
  endfor
endfunction
```

A vectorized (faster) implementation of the same function is given by

```
function res = aVector(xy)
  res=1 + xy(:,1);
endfunction
```

If the applications has to run as fast as possible, then an implementation as a dynamically linked function should be considered. On good operating systems the command `mkoctfile -s a.cc` will create a file `a.oct` using the input below. The speed improvement can be considerable. Below find the file `a.cc`

```
#include <iostream.h>
#include <math.h>
#include <octave/oct.h>
#include <octave/parse.h>

DEFUN_DLD (a, args, , "[...] = a (...) bla ")
{
  octave_value_list retval;
  int nargin = args.length ();
  if (nargin !=1 ) {
    print_usage ("a");
    return retval;
  }
  octave_value X_arg = args(0);
  int nr= X_arg.rows();
  Matrix xy=X_arg.matrix_value();
  ColumnVector result (nr);

  for(int i= 0; i < nr; i++) { result(i) =1.0 + xy(i,0); }
  retval(0)=result;
  return retval;
}
```

Often is is convenient to create one source file for multiple functions and then use links to generate other `*.oct` files. This can save a considerable amount of disk space. An example is shown in demo 4.

2.2.2 FEMEquation

Once all functions and the mesh information are set up, then the system of linear equations can be setup up by `[A,b,n2d]=FEMEquation(nodes,elem,edges,'a','b','f','gD','gN');` to solve the boundary value problem in equation (1).

- The functions `a`, `b` and `f` can be given as string with the function–name, as array of values at the nodes or as one scalar value to be used on all nodes, i.e. constant coefficients.
- The boundary functions `gD` and `gN` can be given as string with the function–name or as a constant scalar value.

The command will create a representation of the symmetric matrix in `A` and the RHS in the vector `b`. The vector `n2d` shows the essential boundary conditions and numbers the actual degrees of freedom of the system. Almost FEM problems solved with this package will require a call of `FEMEquation()`.

The on-line help on `FEMEquation()` gives more information, as shown below.

```
[...] = FEMEquation (...)
  sets up the system of linear equations for a numerical solution of a PDE

  [A,b,n2d]=FEMEquation(nodes,elem,edges,'a','b','f','gD','gN')
  [A,b,n2d]=FEMEquation(nodes,elem,edges,aVec,bVec,fVec,'gD','gN')
     nodes elem edges  describe the mesh
         see ReadMesh() for the description of the format
     'a','b','f','gD','gN' are the names of the functions and coefficients
         in the boundary value problem given below
      the functions a, b and f may be given as constant scalar value
         or as vector with the values of the function at the nodes
      the functions gD and gN may be given as constant scalar value

  div(a*grad u) - b*u = f     in domain
                    u = gD    on Dirichlet section of the boundary
               a*du/dn = gN    on Neumann section of the boundary

 A   is the matrix of the system to be solved.
     It is stored in a symmetric, banded form (see SBSolve() )
 b   is the RHS of the system to be solved.
 n2d is the renumbering of the nodes to the DOF of the system
     n2d(k)=0  indicates that node k is a Dirichlet node
     n2d(k)=nn indicates that the value of the solution at node k
               is given by u(nn)
```

The contributions of each element and edge are computed internally by this function. The function files `ElementContribution` and `ElementContributionEdge` are not used.

2.2.3 FEMSolveSym

Once the equations are known they can be solve by `u=FEMSolveSym(nodes,A,b,n2d,'gD')`. The vector `u` will contain the values of the function at the nodes. The on-line help shows more information.

```
[...] = FEMESolveSym (...)
  solves the system of linear equations for a numerical solution of a PDE
```

```
u=FEMSolveSym(nodes,A,b,n2d,gDFunc)

nodes contains information about the mesh
        see ReadMesh() for the description of the format
A   is the matrix of the system to be solved.
    It is stored in a symmetric, banded form (see SBSolve() )
b   is the RHS of the system to be solved.
n2d is the renumbering of the nodes to the DOF of the system
    n2d(k)=0  indicates that node k is a Dirichlet node
    n2d(k)=nn indicates that the value of the solution at node k
              is given by u(nn)
'gD' is the function describing the Dirichlet boundary condition
        it may also be given as a scalar value

u   is the vector with the values of the solution
```

The source of `FEMSolveSym` is very simple. First the system of equations is solved by calling `SBSolve()`, then the solution is supplemented with the values on the Dirichlet boundary for the final solution vector `u`.

```
function u=FEMSolveSym(nodes,gMat,gVec,n2d,gDFunc)
if (nargin!=5)
  help("FEMSolveSym"); usage("FEMSolveSym(nodes,A,b,n2d,gDFunc)");
endif

ug=-SBSolve(gMat,gVec);
n=length(n2d);
u=zeros(n,1);

for k=1:n
  if n2d(k)>0
    u(k) = ug(n2d(k));
  else
    if is_scalar(gDFunc) u(k) = gDFunc;
       else              u(k) = feval(gDFunc,nodes(k,1:2));
    endif % scalar
  endif
endfor
endfunction
```

If the matrix `A` is not given in banded symmetric form, then the command `FEMSolve()` can be used instead. If possible `FEMSolveSym()` should be used, as it is considerably faster.

2.2.4 `FEMEig`

To determine eigenvalues λ and eigenfunctions u of the boundary value problem

$$\begin{aligned} \operatorname{div}(a\ \operatorname{grad} u) - b\,u &= \lambda\, f\, u && \text{in} \quad \Omega \\ u &= 0 && \text{on} \quad \Gamma_1 \\ a\,\frac{\partial u}{\partial n} &= 0 && \text{on} \quad \Gamma_2 \end{aligned} \tag{2}$$

the command `FEMEig()` can be used.

```
[...] = FEMEig (...)
  determine  eigenvalues and eigenfunctions for the given BVP

  div(a*grad u) - b*u = la*f*u   in domain
                    u = 0        on Dirichlet section of the boundary
               a*du/dn = 0       on Neumann section of the boundary

  la      = FEMEig(nodes,elem,edges,aFunc,bFunc,fFunc,eigVec,tol)
  [la,ev] = FEMEig(nodes,elem,edges,aFunc,bFunc,fFunc,eigVec,tol)

 nodes elem, edges contains information about the mesh
         see ReadMesh() for the description of the format
 aFunc bFunc fFunc  function files for the coefficient functions
         may also be given as vectors or scalar values
 eigVec  is the initial guess for the eigenvectors
         the number of columns determines the number of eigenvalues
         to be computed
         if a number n is given, then n eigenvalues will be computed
 tol is the tolerance for the relative error of the eigenvalues
         if not given tol = 1e-5 is used as default

 la  is the vector containing the eigenvalues
 ev  is the matrix with the eigenvectors as columns
```

The command will create the global stiffness matrix $\mathbf{A}$ and a mass matrix $\mathbf{B}$ and then call `SBEig()` to solve the generalized eigenvalue problem $\mathbf{A}\vec{v} = \lambda\mathbf{B}\vec{v}$.

An example is given in demos 4 and 7.

2.2.5 `FEMValue`

To compute the value of the solution at a specific point or at multiple use `FEMValue()`. Calling the function `values=FEMValue(xy,nodes,elem,u,defaultvalue)` returns the values of the linearly interpolated solution at the points given in `xy`. It is considerably more efficient to call the function once with multiple points in `xy` than to call it for each point separately. If a point is not in the domain, then `0` is returned, unless `defaultvalue` is specified. There are certainly faster algorithms than the one used here, but its it not extremely slow either.

If one also wishes to calculate the values of the gradient at the given points then one can call the function with 2 output arguments, e.g. `[values,grad]=FEMValue(xy,nodes,elem,u)`. Examples are shown in demos 1, 2, 4, 7, 8 and 11.

2.2.6 `FEMGradient`

To compute the value of the gradient of a function at all nodes of the mesh use `FEMGradient()`. Calling `grad=FEMGradient(nodes,elem,u)` will determine the gradient of the function u at the nodes. The function is constructed by linear interpolation, using the given values `u` at the `nodes`. Then the gradient is computed on each element. For each node a weighted average of the gradient on the neighboring elements is used. The weight is given by the angle of the element (triangle) at the node. For evaluation at the nodes `FEMGradient` returns better results than `FEMValue`. An example is shown in demo 7 .

2.2.7 FEMIntegrate

With the command `integral=FEMIntegrate(nodes,elem,u)` the linear interpolation of the function determined by u will be integrated over the domain, i.e. compute

$$\iint_{\Omega} u \, dA$$

```
[...] = FEMIntegrate (...)
  integrate a function over the mesh

  integral = FEMIntegrate(nodes,elem,'f')
  integral = FEMIntegrate(nodes,elem,fValues)
     nodes elem  describe the mesh
                 see ReadMesh() for the description of the format
     'f'         is the name of the function to be integrated
     fValues     a vector with the values of the function at the nodes

     integral    is the integral of the given function over the mesh
```

For linear function the result is exact. An example is shown in demo 7 .

2.2.8 FindDOF

The function file `FindDOF` determines the DOF (degrees of freedom) for the system at hand. It also determines the correct numbering of the DOF. It is an internal function, called in `FEMEquation` and `FEMEquationM`. The result is stored in a variable `n2d`, to be used in `FEMSolveSym` and `FEMSolve`. There should be no need to call this function explicitly.

2.3 Visualization

2.3.1 ShowSolution

The computed solution can be graphed using *Gnuplot* with the help of the command `ShowSolution()`. Its syntax is self-explanatory.

```
ShowSolution(...)
 shows a graph of a numerical solution of the PDE

ShowSolution(nodes,elem,u);
nodes, elem contain information about the mesh
        see ReadMesh() for the description of the format
u   contains the values of the solution at the nodes
```

To speed up the writing to the temporary file (`/tmp/meshdata.gnu`) a dynamically linked function `WriteSolution` is used. The source of `ShowSolution` is a function file. Examples are given in most demos.

2.3.2 ShowLevelCurves

The level curves of a computed function can be visualized by`ShowLevelCurves()`. Its syntax is self-explanatory.

```
ShowLevelCurves(...)
 shows level curves of a function on the mesh

ShowLevelCurves(nodes,elem,u,levels);
nodes, elem contain information about the mesh
        see ReadMesh() for the description of the format
u     contains the values of the solution at the nodes
level is a list of values for which the level curves are drawn

see also ReadMesh, ShowMesh, FEMEquation, FEMSolve, FEMValue
```

Examples are given in demos 3, 7 and 8 .

2.3.3 ShowVectorField

A vecor field (e.g. a gradient field) can be visualized the command `ShowVectorField()`. Its syntax is self-explanatory.

```
ShowVectorField(...)
  shows a vector field plot of a numerical solution of the PDE

ShowVectorField(nodes,vectors,factor)
 nodes    the coordinates of the points at which the vector field was computed
 vectors  components of the vectorfield at nodes
 factor   vector field is rescaled by this factor, if given

     if the argument factor is not specified an appropriate
     default value will be choosen
```

Examples are given in demos 4, 7, 10 and 11 .

2.3.4 ShowSolutionMTV

If a visualization with the help of *plotmtv* is desired, then use `ShowSolutionMTV()`. This command generates an input file to be displayed from a command line by `plotmtv filename`. The full power of *plotmtv* is at your disposition.

```
[...] = ShowSolutionMTV (...)
  generate the data to be plotted with plotmtv

ShowSolutionMTV(nodes,elem,u,'filename')
   nodes elem u  describe the mesh and the solution
  'filename' is the name of the file to be used
```

The header of the file `filename` allows for some modifications of the output. For more information consult the documentation of *plotmtv*.

2.4 Some function files for educational purposes

For educational purposes it can be useful to have pure *Octave* code, implementing all necessary steps of a FEM algorithm. This allows to compute the contributions of each element and edge to the system of equation to be solved and the assembling of the system of linear equations can be examined using *Octave* code. For larger problems this code should not be used, as the implementations in the previous sections are considerably faster.

- `ElementContribution`
  To compute the element stiffness matrix and the vector contribution of one given element call `[elMat,elVec]=ElementContribution(corners,aFunc,bFunc,fFunc)`
  See also the on-line help on `ElementContribution`

- `ElementContributionEdge`
  To compute the contribution of the RHS vector due to one line segment of the Neumann part of the boundary call `edgeVec=ElementContributionEdge(corners,gNFunc)`
  See also the on-line help on `ElementContributionEdge`

- `FEMEquationM`
  This function file implements the same procedures as `FEMEquation`. It calls the above two functions repeatedly. It returns a full matrix `A` and thus `FEMSolve` has to be used to solve the system of equations.

- `FEMSolve`
  This function serves the same purpose as `FEMSolveSym`, but for a full matrix. It is usually not as fast.

- `FEMValueM`
  This function serves the same purpose as `FEMValue`, but is considerably slower.

- `ShowSolutionM`, `ShowVectorField`
  Display the solution or its gradient using *Gnuplot*.

Demo 3 shows an elementary sample application.

3 Operations with banded, symmetric matrices

3.1 Basic description

Many matrices used to solve PDE (using FEM) are symmetric. It the nodes are numbered properly then the matrix will show a band structure, i.e. all nonzero elements are located close to the main diagonal. The algorithm of Cholesky or the LDL^T factorization can take advantage of this structure, see [GoluVanLoan96]. For a symmetric matrix A of size $n \times n$ with semi-bandwidth b the approximate computational cost to solve one system of equations is given by

$$\text{Gauss} \approx \frac{1}{3}\, n^3 \quad \text{and} \quad \text{Band Cholesky} \approx \frac{1}{2}\, n\, b^2$$

Obviously for $b \ll n$ is is advantageous to use a banded solver. A more detailed analysis and an implementation is given in [VarFem].

To take advantage of the symmetry and the band structure the matrices will be stored in a modified format, as illustrated below.

$$\begin{bmatrix} 10 & 2 & 3 & 0 & 0 \\ 2 & 20 & 4 & 5 & 0 \\ 3 & 4 & 30 & 6 & 7 \\ 0 & 5 & 6 & 40 & 8 \\ 0 & 0 & 7 & 8 & 50 \end{bmatrix} \longrightarrow \begin{bmatrix} 10 & 2 & 3 \\ 20 & 4 & 5 \\ 30 & 6 & 7 \\ 40 & 8 & 0 \\ 50 & 0 & 0 \end{bmatrix}$$

A banded version of the LDL^T factorization in [GoluVanLoan96] can be implemented. If the matrix A is strictly positive definite, then the algorithm is known to be stable. If A is not positive definite, then problems might occur, since no pivoting is done. The matrix A is positive definite if and only if the diagonal matrix D is positive.

For a given matrix some of its smallest eigenvalues can be computed with an algorithm based on inverse power iteration. Precise information on the numerical errors is provided. The code is capable of finding eigenvalues of medium size matrices, where the standard command `eig()` of *Octave* is either very slow or will fail.

3.2 Description of the commands

3.2.1 SBSolve

The basic factorization algorithm is implemented in `SBSolve`. The function can return the solution of the system of linear equations, or the solution and the factorization of the original matrix. Multiple sets of equations can be solved.

```
[...] = SBSolve (...)
  solve a system of linear equations with a symmetric banded matrix

  X=SBSolve(A,B)
  [X,R]=SBSolve(A,B)

   solves A X = B

   A is mxt where t-1 is number of non-zero super diagonals
   B is mxn
   X is mxn
   R is mxt

  if A would be ! 11000 ! then A= ! 11 !
                ! 14300 !         ! 43 !
                ! 03520 !         ! 52 !
                ! 00285 !         ! 85 !
                ! 00059 !         ! 90 !

  B is a full matrix

  The code is based on a LDL' decomposition (use L=R'), without pivoting.
  If A is positive definite, then it reduces to the Cholesky algorithm.

  R is an upper right band matrix
  The first column of R contains the entries of a diagonal matrix D.
  If the first column of R is filled by 1's, then we have R'*D*R = A
```

To determine the inverse matrix $\mathtt{A}^{-1}$ one can use the command `invA = SBSolve(A,eye(n));`. Be aware that calculating the inverse matrix is rarely a wise thing to do. Most often the inverse of a banded matrix will loose the band structure. If many system of linear equations have to be solved simultaneously, then use `SBSolve(A,B)` with a matrix `B`. If multiple systems need to be solved sequentially, use `SBFactor()` and then `SBBacksub` for each system to be solved.

If the matrix **A** is strictly positive definite, then the algorithm is stable and one can expect the solution to be as accurate as the condition number of **A** permits. If **A** is semidefinite, then large errors might occur, since **no pivoting** is implemented in the code. The matrix is positive definite iff all eigenvalues are positive, this can be verified by inspectiion of the signs of the numbers in the first column of **R**. The matrix is positive definite if the first column of the factorization matrix `R` (use `SBFactor()`) contains positive numbers only. A description of the algorithm can be found in [GoluVanLoan96] or [VarFem].

3.2.2 SBFactor and SBBacksub

Instead of calling `X=SBSolve(A,B)` one can first call `R=SBFactor(A)` to determine the factorization $A = R^T DR$ and then `B=SBBacksub(R,X)` to solve the system(s) $A \cdot X = B$. Since most of the computational effort is in the factorization, this can be useful if many system of linear equations have to be solved sequentially. If multiple system are to be solved simultaneously it is preferable to use `SBSolve(A,B)` with a matrix `B` .

```
[...] = SBFactor(...)
  find the R'DR factorization of a symmetric banded matrix

  R=SBFactor(A)

    A is mxt where t-1 is number of non-zero super diagonals
    R is mxt

  if A would be ! 11000 ! then A= ! 11 !
                ! 14300 !         ! 43 !
                ! 03520 !         ! 52 !
                ! 00285 !         ! 85 !
                ! 00059 !         ! 90 !


  The code is based on a LDL' decomposition (use L=R'), without pivoting.
  If A is positive definite, then it reduces to the Cholesky algorithm.

  R is an upper right band matrix
  The first column of R contains the entries of a diagonal matrix D.
  If the first column of R is filled by 1's, then we have R'*D*R = A

[...] = SBBacksub(...)
  using backsubstitution  to return the solution of a system of linear equations

  X=SBBacksub(R,B)

   B is mxn
   X is mxn
   R is mxt

   R is produced by a call of [X,R] = SBSolve(A,B) or R = SBFactor(A)
   It is an upper right band matrix
   The first column of R contains the entries of a diagonal matrix D.
   If the first column of R is filled by 1's, then we have R'*D*R = A
```

If there is interest in the classical Cholesky decomposition of the matrix `A` (i.e. $\mathtt{A} = \mathtt{R}' \cdot \mathtt{R}$) then `R` can be computed by

```
rBand=SBFactor(A);
d=sqrt(rBand(:,1));
rBand(:,1)=ones(n,1);
r=triu(diag(d)*rBand)
```

The number of positive/negative numbers in the first column of $\mathbf{R}$ equals the number of positive/negative eigenvalues of $\mathbf{A}$.

3.2.3 SBEig

For given symmetric matrices $\mathbf{A}$ and $\mathbf{B}$ the standard (resp. generalized) eigenvalue problem will be solved, i.e.

$$\mathbf{A}\,\vec{v} = \lambda\,\vec{v} \quad \text{resp.} \quad \mathbf{A}\,\vec{v} = \lambda\,\mathbf{B}\,\vec{v}$$

Using inverse power iteration a given number of the smallest (absolute value) eigenvalues if a symmetric matrix $\mathbf{A}$ are computed. If needed the eigenvectors are also generated. A set of initial vectors $\mathbf{V}$ have to be given. If those are already close to the eigenvectors, then the algorithm will converge rather quickly. For a precise description and analysis consult [GoluVanLoan96] or [VarFem].

```
[...] = SBEig(...)
  find a few eigenvalues of the symmetric, banded matrix
  inverse power iteration is used for the standard and generalized
  eigenvalue problem

  [Lambda,{Ev,err}] = SBEig(A,V,tol)     solve A*Ev = Ev*diag(Lambda)
                    standard eigenvalue problem

  [Lambda,{Ev,err}] = SBEig(A,B,V,tol)   solve A*Ev = B*Ev*diag(Lambda)
                    generalized eigenvalue problem

   A   is mxt, where t-1 is number of non-zero superdiagonals
   B   is mxs, where s-1 is number of non-zero superdiagonals
   V   is mxn, where n is the number of eigenvalues desired
       contains the initial eigenvectors for the iteration
   tol is the relative error, used as the stopping criterion

   X   is a column vector with the eigenvalues
   Ev  is a matrix whose columns represent normalized eigenvectors
   err is a vector with the a posteriori error estimates for the eigenvalues
```

The algorithm is based on inverse power iteration with n independent vectors. The iteration will proceed until the relative change of all eigenvalues is smaller than the given value of `tol`. This does not guarantee that the relative error is smaller than `tol`. The initial guesses $\mathbf{V}$ for the eigenvectors have to be linearly independent. The closer the initial guess is to the actual eigenvector, the faster the algorithm will converge. The algorithm returns the n eigenvalues closest to 0 .

For the standard eigenvalue problem $\mathbf{A}\,\vec{v}_i = \lambda_i\,\vec{v}_i$ the eigenvectors $\vec{v}_i$ will be orthonormal with respect to the standard scalar product, i.e, $\langle \vec{v}_i\,,\,\vec{v}_j \rangle = \delta_{i,j}$. For the generalized eigenvalue problem $\mathbf{A}\,\vec{v}_i = \lambda_i\,\mathbf{B}\,\vec{v}_i$ this translates to $\langle \vec{v}_i\,,\,\mathbf{B}\,\vec{v}_j \rangle = \delta_{i,j}$. The symmetric matrix $\mathbf{B}$ should be positive definite. The columns of `Ev` can be used to restart the algorithm if higher accuracy is required.

The algorithm will return reliable estimates for the errors in the eigenvalues. The à posteriori error estimate `err` is based on the residual $\vec{r} = \mathbf{A}\,\vec{v} - \lambda\,\vec{v}$ and

$$\min_{\lambda_i \in \sigma(\mathbf{A})} |\lambda - \lambda_i| \leq \langle \vec{r}\,,\,\vec{r} \rangle = \|\vec{r}\|$$

where we use the normalization $\langle \vec{v}\,,\,\vec{v} \rangle = 1$. If one of the eigenvalues has to be computed with high accuracy, the approximate value λ may be subtracted from the diagonal of the matrix. Then the eigenvalue closest to zero of the modified matrix $\mathbf{A} - \lambda\,\mathbb{I}$ can be computed, using the already computed eigenvector. If the eigenvalue is isolated the algorithm will converge very quickly. This algorithm is similar to the Rayleigh quotient iteration. A good description is given in [GoluVanLoan96].

If the eigenvalue closest to λ is denoted by λ_i we have the improved estimate

$$|\lambda - \lambda_i| \leq \frac{\|\vec{r}\|^2}{\text{gap}} \quad \text{where} \quad \text{gap} = \min\{|\lambda - \lambda_j| \ : \ \lambda_j \in \sigma(\mathbf{A}), j \neq i\}$$

It is very easy to implement this test in *Octave*. If the estimate is based on approximate values of the eigenvalues, then the result is not as reliable as the previous one. Since the value of `gap` will carry an approximation error. The situation is particularly bad if some eigenvalues are clustered.

For the generalized eigenvalue problem we use the residual $\vec{r} = \mathbf{A}\,\vec{v} - \lambda\,\mathbf{B}\,\vec{v}$ and the estimate

$$\min_{\lambda_i \in \sigma(\mathbf{A})} |\lambda - \lambda_i| \leq \sqrt{\langle \vec{r},\, \mathbf{B}^{-1}\vec{r}\rangle} \quad \text{and} \quad |\lambda - \lambda_i| \leq \frac{\langle \vec{r},\, \mathbf{B}^{-1}\vec{r}\rangle}{\text{gap}}$$

where we use the normalization $\langle \vec{v},\, \mathbf{B}\,\vec{v}\rangle = 1$. The variable `err` will return the first of the above estimates. The precise algorithm and proof of the above estimate is given in [VarFem].

3.2.4 `SBProd`

With this command a symmetric banded matrix can be multiplied with a full matrix.

```
[...] = SBProd(...)
  multiplies a symmetric banded matrix with a matrix

  X=SBProd(A,B)

   A is mxt where t-1 is number of non-zero super diagonals
   B is mxn
   X is mxn

  if A would be ! 11000 ! then A= ! 11 !
                ! 14300 !         ! 43 !
                ! 03520 !         ! 52 !
                ! 00285 !         ! 85 !
                ! 00059 !         ! 90 !

  B is full matrix Ax=B
```

3.2.5 `BandToFull`, `FullToBand` and `BandToSparse`

With these commands conversion between full, symmetric matrices and banded symmetric matrices is possible. A conversion to a sparse format is also included.

4 Auxiliary programs

An essential part of a FEM solution to a boundary value problem is the generation of a mesh. The package *FEMoctave* is building on external codes to generate the meshes.

- **EasyMesh** This code is available from a web site[5] or also from this author's home page. The original source is slightly modified.
  - On the first few lines replace `#define MAX_NODES 3000` by `#define MAX_NODES 100000` to allow for meshes with more than 3000 nodes.

[5] `http://www-dinma.univ.trieste.it/~nirftc/research/easymesh/`

– On the very last lines replace `return 1` by `return 0`, otherwise the `make` command will not do all of its job.

- `triangle` This is an excellent mesh generator by Jonathan Richard Shewchuk[6]. The source is included with this package.

- `CuthillKcKee` The numbering of the nodes in a mesh generated by *triangle* will not lead to a matrix with small bandwidth. The algorithm of Cuthill–McKee will improve this situation. The code is included with this package or also avaliable on the web site [www:sha].

5 Examples

There are a few examples distributed with this package. Find them in the subdirectories of `./demos` . To run a demo change into the appropriate directory, run `make`, start *Octave* and then use the script file `demorun.m`.

5.1 Demo 1: a first example

The source and a `Makefile` for this example can be found in in directory `./demos/demo1`. It is used to generate the mesh with the help of *EasyMesh*.

In this example the domain is the rectangle $\Omega = [0,5] \times [0,4]$ and the boundary value problem to be solved is

$$\begin{array}{rcll} \Delta u & = & -1 & \text{for } 0 < x < 5 \text{ and } 0 < y < 4 \\ u & = & 0 & \text{for } 0 < x < 5 \text{ and } y \in \{0,4\} \\ \frac{\partial u}{\partial n} & = & -1 & \text{for } x \in \{0,5\} \text{ and } 0 < y < 4 \end{array}$$

The domain and type of boundary condition are described in the file `test4.d` shown below.

```
5 # number of points #
# Nodes which define the boundary #
0:   0   0  0.2   1
1:   5   0  0.2   1
2:   5   4  0.2   1
3:   0   4  0.2   1
# material marker #
4: 2   2    0   1 # material 1 #
4 # Number Boundary of segments #
0:   0   1   1  #Dirichlet
1:   1   2   2  #Neumann
2:   2   3   1  #Dirichlet
3:   3   0   2  #Neumann
```

The command `EasyMesh test4` will then create the mesh. The coefficient functions are all given as constants. The *Octave* script below will generate a graph of the solution.

```
clear
tic
  [nodes,elem,edges]=ReadMesh("./test4");
readingtime=toc
tic
  [A,b,n2d]=FEMEquation(nodes,elem,edges,1,0,-1,0,-1);
setuptime=toc
```

[6] `http://www.cs.cmu.edu/~quake/triangle.html`

```
tic
  u=FEMSolveSym(nodes,A,b,n2d,0);
solvetime=toc
tic
  ShowSolution(nodes,elem,u)
graphtime=toc
```

This leads to figure 1 and the output below.

```
octave:1> demorun
readingtime = 0.11547
setuptime = 0.12725
solvetime = 0.13990
graphtime = 0.067842
```

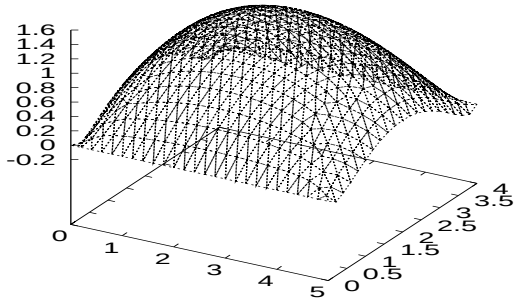

Figure 1: Solution of an elementary PDE

The additional lines in `demorun.m` evaluate the function along a diagonal in the domain.

```
np=25;
xy=[linspace(0,5,np);linspace(0,4,np)]';
[values,grad]=FEMValue(xy,nodes,elem,u)
```

5.2 Demo 2: parameter dependence and animation

This example is similar to demo 1, in fact the same mesh is used and only the function $f(x, y)$ changed. It depends on a parameter *par*, i.e. a global variable. It is given by

$$f(x,y) = \begin{cases} -par & \text{if} \quad x \leq 2 \\ -1 & \text{if} \quad x > 2 \end{cases}$$

and implemented in the script file **demorun.m**. The parameter *par* varies from -1 to 2 and the script file creates a poor man's animation with the help of *Gnuplot*.

```
page_screen_output=0;
global par;

clear f
```

```
function res = f(xy)
  global par;
  [n,m]=size(xy);
  res=-1*ones(n,1)*par;
  for k=1:max(size(xy))
    if(xy(k,1)>2) res(k)=-1;endif
  endfor
endfunction

[nodes,elem,edges]=ReadMesh("../demo1/test4");
gset zrange [-3:4]

for par=-1:0.1:2
  [A,b,n2d]=FEMEquation(nodes,elem,edges,1,0,'f',0,-1);
  u=FEMSolveSym(nodes,A,b,n2d,0);
  ShowSolution(nodes,elem,u)
  res=[par,FEMValue([2,2],nodes,elem,u)];
  printf("For lambda=%2.3f we find u(2,2)=%2.4f\n",res);
endfor
```

5.3 Demo 3: using function files only

The problem to be solved on the rectangular domain $\Omega = [0,1] \times [0,2]$ is

$$\begin{aligned} \Delta u &= -1 \quad \text{in} \quad \Omega \\ u &= 0 \quad \text{on} \quad \partial\Omega \end{aligned}$$

The mesh is generated by the function `CreateRectMesh()` and all definitions of the functions are given in the script file `demorun.m`

```
clear
x=linspace(0,1,5);
y=linspace(0,2,5);
[nodes,elem,edges]=CreateRectMesh(x,y,1,1,1,1);
[A,b,n2d]=FEMEquationM(nodes,elem,edges,1,0,-1,0,0);
u=FEMSolve(nodes,A,b,n2d,0);
gset zrange [*:*]
gset nokey
ShowSolutionM(nodes,elem,u)
A    % display the matrix
b=b' %display the vector

# evaluate along diagonal
npoints=10;
xv=linspace(0,1,npoints);
yv=linspace(0,2,npoints);
xy=[xv;yv]';

[values, grad] =FEMValueM(xy,nodes,elem,u,0)
```

Since no dynamically linked libraries are used this is a rather slow method to solve the problem. But for small meshes this is feasible nonetheless. The above computation leads to the global stiffness matrix `A` and the vector `b` below.

```
A =
   5.000  -2.000   0.000  -0.500   0.000   0.000   0.000   0.000   0.000
  -2.000   5.000  -2.000   0.000  -0.500   0.000   0.000   0.000   0.000
   0.000  -2.000   5.000   0.000   0.000  -0.500   0.000   0.000   0.000
  -0.500   0.000   0.000   5.000  -2.000   0.000  -0.500   0.000   0.000
   0.000  -0.500   0.000  -2.000   5.000  -2.000   0.000  -0.500   0.000
   0.000   0.000  -0.500   0.000  -2.000   5.000   0.000   0.000  -0.500
   0.000   0.000   0.000  -0.500   0.000   0.000   5.000  -2.000   0.000
   0.000   0.000   0.000   0.000  -0.500   0.000  -2.000   5.000  -2.000
   0.000   0.000   0.000   0.000   0.000  -0.500   0.000  -2.000   5.000
b'=
  -0.125  -0.125  -0.125  -0.125  -0.125  -0.125  -0.125  -0.125  -0.125
```

The script `demorun.m` also computes the values of the solution along a diagonal of the rectangular domain.

We may examine a single element stiffness matrices. To find the contributions from a triangular element with corners at $(0,0)$, $(1,0)$ and $(0,1)$ use

```
[mat,vec]=ElementContribution([0,0;1,0;0,1],1,0,-1)
```

to obtain

```
mat =   1.00000  -0.50000  -0.50000
       -0.50000   0.50000   0.00000
       -0.50000   0.00000   0.50000

vec =  -0.16667
       -0.16667
       -0.16667
```

A second problem

$$\begin{aligned} \Delta u &= 0 && \text{in } \Omega \\ u &= x + 3\,y && \text{on } \partial\Omega \end{aligned}$$

is solved in `demorun2.m`.

Results of this type can be useful to teach FEM algorithms.

5.4 Demo 4: a bigger problem, evaluation of the solution, its gradient, eigenvalues and eigenfunctions

On an L–shaped domain Ω we consider the boundary value problem

$$\begin{aligned} \Delta u &= 10\,(x-y) && \text{in } \Omega \\ u &= 0 && \text{on } \partial\Omega \end{aligned}$$

The mesh is generated, such that it is finer at the inside corner. It consists of 2496 nodes, forming 4802 elements.

The codes in this demo use compiled functions for the coefficient function. Thus before launching the FEM code one has to compile the function with the halp of the `make` programm. Then the script `demorun.m` shown below will compute the the solution and create a few plots.

```
tic
  [nodes,elem,edges]=ReadMesh("./Lshape");
NumberNodes=max(size(nodes))
readingtime=toc
tic
  [A,b,n2d]=FEMEquation(nodes,elem,edges,'a',0,'f',0,0);
setuptime=toc
[dof,semiband]=size(A)
tic
  u=FEMSolveSym(nodes,A,b,n2d,0);
solvetime=toc
figure(1);
tic
  ShowSolution(nodes,elem,u)
graphtime=toc
  ShowSolutionMTV(nodes,elem,u,'u.mtv')

gset zrange [*:*]
gset view 60, 300
gset title
gset nokey
replot

npoints=41;
x=linspace(-1,1,npoints);
y=linspace(1,-1,npoints);
xy=[x;y]';

pause(3)
tic
values=FEMValue(xy,nodes,elem,u,0);
evaluatetime=toc

figure(2);
gset nokey
gset title "A section"
plot (x,values)

tic
[la,vec]=FEMEig(nodes,elem,edges,'a' ,0,1,4,1e-4);

eigenvaluetime=toc
eigenvalues=la'
figure(3);
gset title "3rd eigenfunction"
ShowSolution(nodes,elem,vec(:,3))
gset title

# evaluate on a mesh
npoints=21;
xv=linspace(-1,1,npoints);  yv=linspace(-1,1,npoints);
```

```
[xx,yy]=meshgrid(xv,yv);
xy=[xx(:),yy(:)];

[values, grad] =FEMValue(xy,nodes,elem,u,0);

figure(4);
ShowVectorField(xy,-grad)
```

This leads to the output

```
octave:1> demorun
NumberNodes = 2496
readingtime = 0.26706
setuptime = 0.27529
dof = 2308
semiband = 133
solvetime = 0.63635
graphtime = 0.21737
evaluatetime = 0.018111
eigenvaluetime = 5.1880
eigenvalues =  9.6359  15.1744  19.6921  29.4253
```

This shows that we have a system with 2308 unknowns with a semi-bandwidth of 133. On a given computer[7] it takes 0.27 sec to read the mesh information, 0.28 sec to set up the equations and 0.6 sec to solve. The time to evaluate the solution along the diagonal from $(-1, 1)$ to $(1, -1)$ at 41 points takes 0.02 sec. To find the first four eigenvalues 5 sec are used. The shape of the third eigenfunction is plotted. The resulting graphs are shown in figure 2.

Using the command `plotmtv u.mtv &` we can generate 3–d plots and level curves, as shown in figure 3.

[7] A dual Pentium III 800MHz PC with 256KB cash per CPU and 256M RAM running Linux

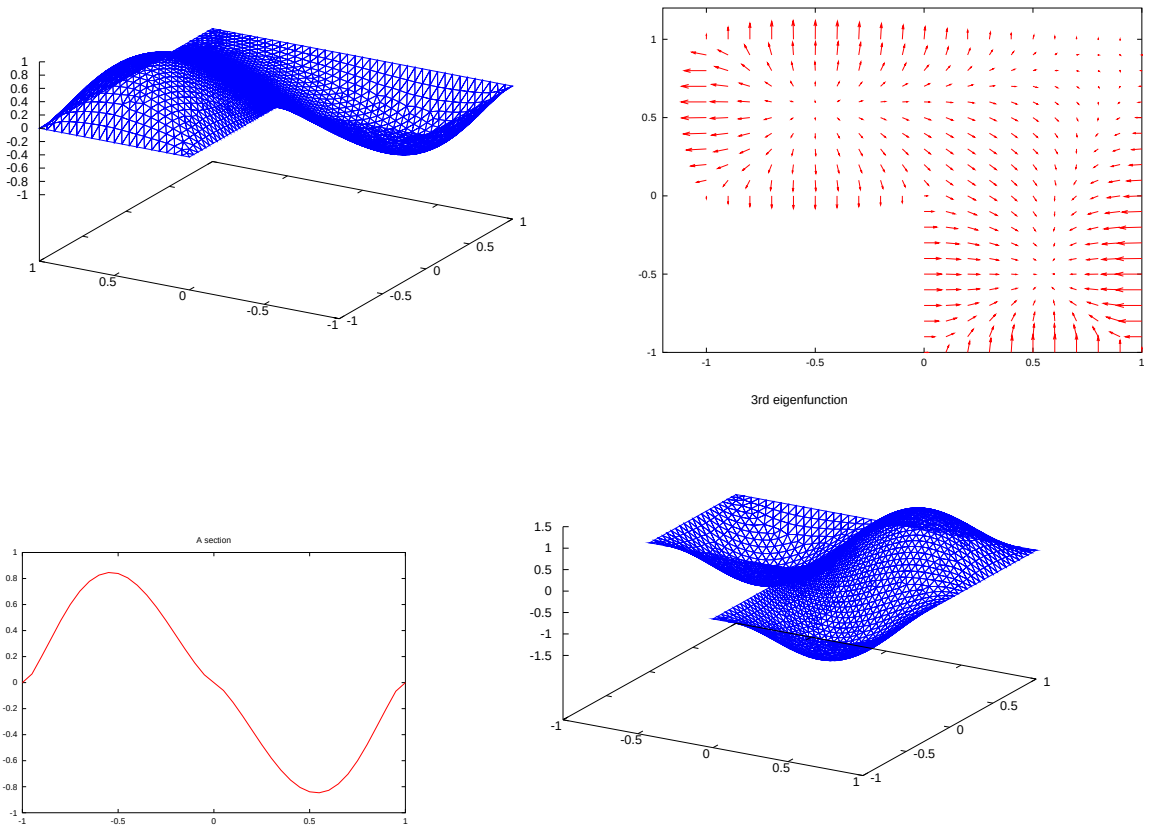

Figure 2: Solution of a PDE, the gradient vector field, a cross section and the third eigenfunction

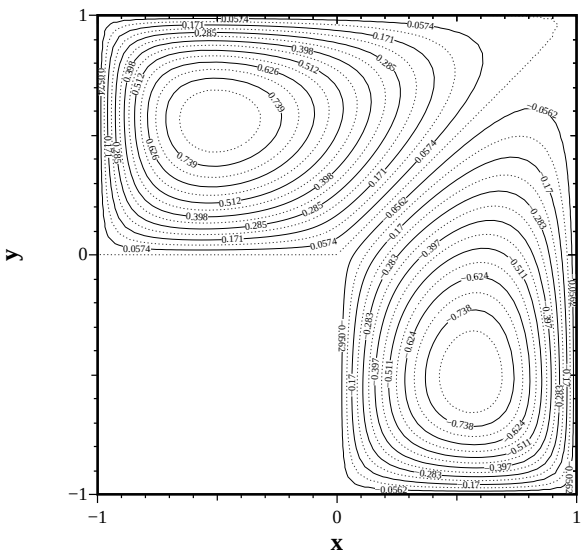

Figure 3: A plot of level curves, generated by *plotmtv*

5.5 Demo 5: using *triangle* and *CuthillMcKee*

The mesh can be generated by *triangle*. The input file for the mesh in figure 4 was generated using the input file `testA.poly` shown below. Further documentation can be found on the home page of *triangle*.

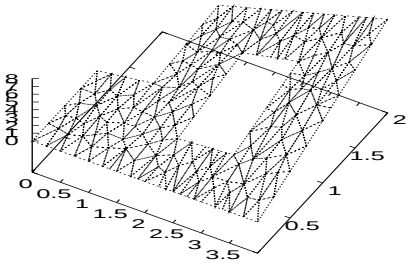

Figure 4: Solution on a structure with a hole

The problem to be solved

$$\begin{aligned} \Delta u &= 0 \qquad \text{for} \quad (x,y) \in \Omega \\ u &= x \cdot y \qquad \text{for} \quad (x,y) \in \Gamma \end{aligned}$$

The exact solution is $u(x,y) = x \cdot y$, thus we can compare the approximate solution and compute the error.

```
# nodes
10 2 0 0
1 0 0
2 4 0
3 4 2
4 1 2
5 1 1
6 0 1
7 2 0.5
8 3 0.5
9 3 1.5
10 2 1.5

# segments
10 1
1 1 2 1
2 2 3 1
3 3 4 1
4 4 5 1
5 5 6 1
6 6 1 1
# hole
7 7 8 1
8 8 9 1
9 9 10 1
```

```
10 10 7 1

1
1 2.5 1
```

Then the command `triangle -pqa0.03 testA.poly` will generate the mesh. To reduce the computation time it is important to use the utility `CuthillMcKee` to renumber the mesh. The structure of nonzero elements is shown in figure 5. This figure was generated by `sparseplot("testA.1",1)` and `sparseplot("testA.1")` . For a larger sample problem[8] with a matrix of size 1422 the computation time jumped from 0.38 sec to 84 sec as the bandwidth changed from 44 to 1398 .

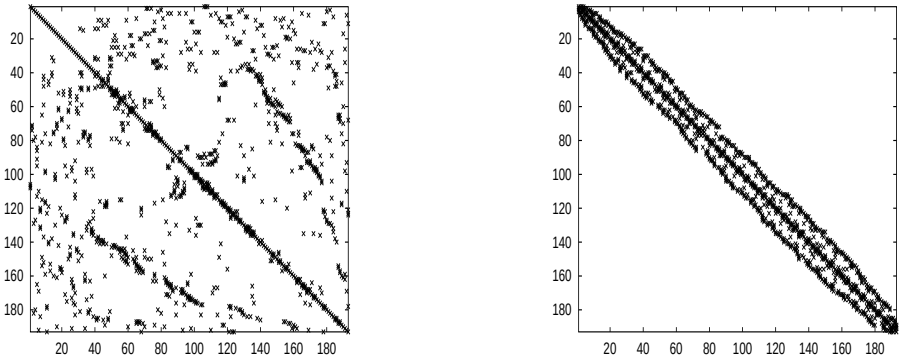

Figure 5: The structure of the matrix before and after renumbering

5.6 Demo 6: using *CreateMeshTriangle*

The script `demorun.m` shown below generates a mesh on the unit square, displays the mesh and then the solution of

$$\begin{array}{rcll} \Delta u & = & 1 & \text{for} \quad 0 < x < 1 \quad \text{and} \quad 0 < y < 1 \\ \frac{\partial u}{\partial n} & = & 1 & \text{for} \quad x = 1 \quad \text{and} \quad 0 < y < 1 \\ u & = & 0 & \text{on the remaining three sections of the boundary} \end{array}$$

```
CreateMeshTriangle("test",[0,0,1;1,0,2;1,1,1;0,1,1],0.01)
[nodes,elem,edges]=ReadMeshTriangle("./test.1");
ShowMesh(nodes,elem)

[A,b,n2d]=FEMEquation(nodes,elem,edges,1,0,1,0,1);
u=FEMSolveSym(nodes,A,b,n2d,0);
ShowSolution(nodes,elem,u)
```

5.7 Demo 7: eigenfunctions of circular membrane

Consider the eigenvalue problem

$$\begin{array}{rcll} \Delta u & = & \lambda\, u & \text{in} \quad \Omega \\ u & = & 0 & \text{on} \quad \Gamma = \partial\Omega \end{array}$$

where $\Omega \subset \mathbb{R}^2$ is the disk with radius 1. The code below will compute four eigenvalues and eigenfunctions, leading to figure 6.

[8] create the mesh by `triangle -pqa0.003 testA.poly`

```
R=1; % radius of circle
nR = 40 ; % number of divisions to create circle
area=0.004;
w=linspace(0,2*pi*(1-1/nR),nR);  % angles of points on circle
xy=[R*cos(w);R*sin(w);ones(1,nR)]';
CreateMeshTriangleQ("circle",xy,area)
[nodes,elem,edges]=ReadMeshTriangle("./circle.1");

[la,vec]=FEMEig(nodes,elem,edges,1,0,1,4,1e-6);
la=la'

figure(1);
ShowSolution(nodes,elem,vec(:,1))
figure(2);
vmin=min(vec(:,1)); vmax=max(vec(:,1));
ShowLevelCurves(nodes,elem,vec(:,1),linspace(vmin,vmax,11))
disp("Hit RETURN"); pause();

figure(1); ShowSolution(nodes,elem,vec(:,2))
figure(2); vmin=min(vec(:,2)); vmax=max(vec(:,2));
ShowLevelCurves(nodes,elem,vec(:,2),linspace(vmin,vmax,11))
disp("Hit RETURN"); pause();
figure(1); ShowSolution(nodes,elem,vec(:,4))
figure(2); vmin=min(vec(:,4)); vmax=max(vec(:,4));
ShowLevelCurves(nodes,elem,vec(:,4),linspace(vmin,vmax,11))
```

A separation of variable argument shows that the exact eigenvalues are given by

$$\sqrt{\lambda} = z_{n,m} = \text{ m}^{th} \text{ zero of } J_n(r)$$

where $J_n(r)$ are the Bessel functions of the first kind. The eigenfunctions are

$$u(r,\phi) = J_n(r/z_{n,m}) \cdot \cos(\frac{n}{2\,\pi}\,\phi) \quad \text{and} \quad u(r,\phi) = J_n(r/z_{n,m}) \cdot \sin(\frac{n}{2\,\pi}\,\phi)$$

Thus we can compare the results of this code with the exact solution. Observe that for $n > 1$ we have eigenvalues of multiplicity 2. The table below identifies the first 20 eigenvalues (resp. $\sqrt{\lambda}$) computed by *FEMoctave* with the exact values $z_{n,m}$.

	1	2	3	4	5	6	7	8	9	10
$\sqrt{\lambda}$	2.4089	3.8365	3.8365	5.1393	5.1390	5.5226	6.3799	6.3802	7.0120	7.0122
$z_{n,m}$	2.4048	3.8317		5.1356		5.5201	6.3802		7.0156	
n	0	1	1	2	2	0	3	3	1	1
m	1	1	1	1	1	2	1	1	2	2
	11	12	13	14	15	16	17	18	19	20
$\sqrt{\lambda}$	7.5826	7.5810	8.4047	8.4028	8.6384	8.7552	8.7555	9.7363	9.7307	9.9043
$z_{n,m}$	7.5883		8.4172		8.6537	8.7715		9.7610		9.9361
n	4	4	2	2	0	5	5	3	3	6
m	1	1	2	2	3	1	1	2	2	1

For the second eigenfunction in figure 6 we can compute and display the gradient field, shown in figure 7. First compute the first two eigenfunctions and define a regular grid on the circle.

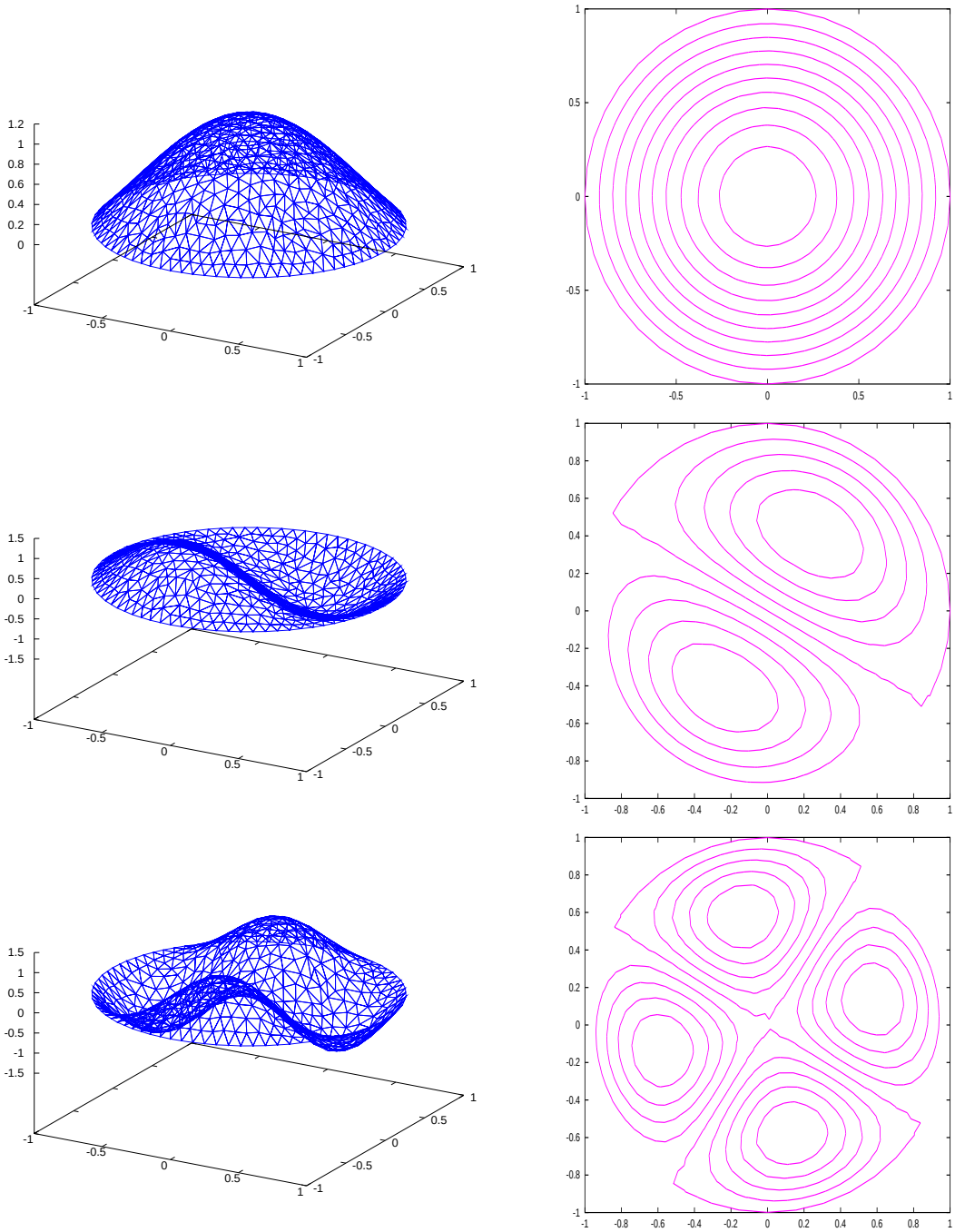

Figure 6: The first, second and fourth eigenfunction of a vibrating membrane

```
[nodes,elem,edges]=ReadMeshTriangle("./circle.1");
[la,vec]=FEMEig(nodes,elem,edges,1,0,1,2,1e-6);

R=1;
nn=21; v=linspace(-R,R,nn);
[xx,yy]=meshgrid(v,v); xy=[xx(:),yy(:)];
```

Then there are two different path to this result.

- Compute values of the function and gradient on the grid, then show the solution.

  ```
  [values, grad] =FEMValue(xy,nodes,elem,vec(:,2),0);
  ShowVectorField(xy,grad)
  ```

- Compute the gradient at the nodes of the mesh, then evaluate the components with the help of `FEMValue` and display the graph.

  ```
  grad=FEMGradient(nodes,elem,vec(:,2));
  g1=FEMValue(xy,nodes,elem,grad(:,1),0);
  g2=FEMValue(xy,nodes,elem,grad(:,2),0);
  ShowVectorField(xy,[g1';g2']')
  ```

The results should be similar but might not be identical.

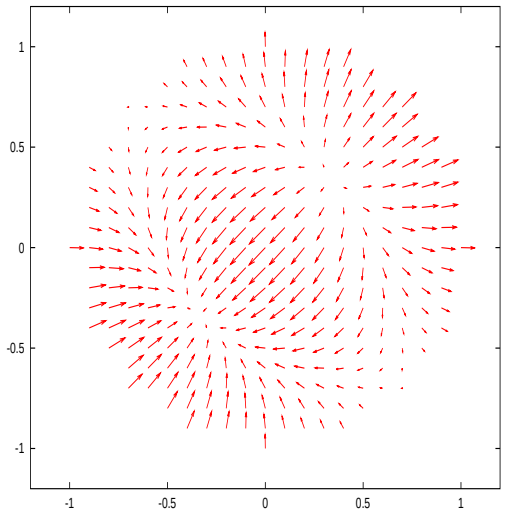

Figure 7: The gradient field of the second eigenfunction of a vibrating membrane

The command `FEMIntegrate()` allows to integrate functions over the meshed domain.

```
FEMIntegrate(nodes,elem,1)
FEMIntegrate(nodes,elem,vec(:,1))
FEMIntegrate(nodes,elem,vec(:,2))
FEMIntegrate(nodes,elem,vec(:,2).^2)
```

Leading to answers 3.1287 $\approx \pi$ (area of the circle), 1.4681 and 0.00043, the integrals of the first and second eigenfunction. The last answer of 1.000 shows that the eigenfunction is normalized in the $L_2(\Omega)$–norm.

5.8 Demo 8: computing a capacitance

Consider a radially symmetric capacitor consisting of two symmetric, conducting plates of radius r. The two conductors are $2\,h$ apart. The setup is enclosed in a cylinder of radius R and height $2\,H$ ($-H < y < H$). By modeling only the upper half of the capacitor we find the following differential equation for the voltage u (radius x, height y)

$$\begin{array}{rcll} \operatorname{div}(x\ \operatorname{grad} u) & = & 0 & \text{in domain} \\ u & = & 0 & \text{along edge } y = 0 \\ u & = & 1 & \text{along edge of upper conductor} \\ \frac{\partial\, u}{\partial n} & = & 0 & \text{on remaining boundary} \end{array}$$

Between the two plates the field is expected to be homogeneous and this is confirmed by figures 8. By computing the flux through the middle plain by

$$\text{flux} = \iint\limits_{\text{disk}} \frac{\partial\, u}{\partial y}\, dA = 2\,\pi \int_0^R x\, \frac{\partial\, u}{\partial y}\, dx \approx 2\,\pi \int_0^r x\, \frac{1}{h}\, dx = \frac{\pi\, r^2}{h}$$

we find the capacitance of this setup. If the setup would correspond to a perfect plane capacitor, then the normalized flux should be 1. The configuration in the script `demorunTriangle` (shown below) leads to the significantly different result of 1.5 . This illustrates that a capacitance with the dimensions above can not be treated as an idealized plate capacitor.

```
clear *
r=1;
R=2.5;
global h=0.2;
H=0.5;
N=201;
area=0.0001;

xy=[0,0,1; R,0,2; R,H,2; r,H,1; r,h,1; 0,h,2];

CreateMeshTriangle("cap",xy,area)
[nodes,elem,edges]=ReadMeshTriangle("./cap.1");

function res = aF(xy)
res=xy(:,1);
endfunction

function res = volt(xy)
global h;
[n,m]=size(xy);
res=zeros(n,1);
for k=1:n
  if (xy(k,2)<h/2) res(k)=0;
  else res(k)=1;endif
endfor
endfunction

function res = zero(xy)
[n,m]=size(xy);
res=zeros(n,1);
```

```
endfunction

[A,b,n2d]=FEMEquation(nodes,elem,edges,'aF','zero','zero','volt','zero');
u=FEMSolveSym(nodes,A,b,n2d,'volt');

x=linspace(0,R,N);
y=0.0*ones(1,N);
[uVal,grad]=FEMValue([x;y]',nodes,elem,u);
plot(x,grad(:,2))

# trapezoidal integration
flux=2*(x*grad(:,2)-x(1)*grad(1,2)/2-x(N)*grad(N,2)/2)*R/(N-1)*(h/r**2)
```

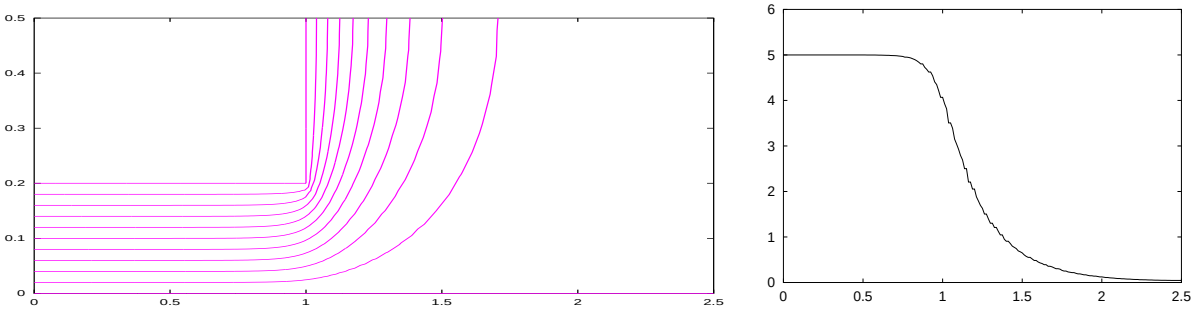

Figure 8: Voltage and vertical field in a capacitor

With the above code the triangles are of uniform size throughout the domain, but since the solution varies only very little in the right part of the domain we are using too many triangles there. By generating the mesh directly with *triangle* we can divide the domain in two sections and generate larger triangles in the right part. This is done in the file `capManual.poly`, which is the used as input for *triangle*. Consider the `Makefile` and `demorunManual.m` for details. The file `capManual2.poly` describes the situation of a very thin conducting plate, the result will be closer to an ideal capacitor.

The same problem is also solved by the script `demorunEasyMesh.m`, using a mesh generated by *EasyMesh*.

5.9 Demo 9: exact solution and convergence

Consider the unit square $\Omega = [0,1] \times [0,1]$. On can verify the $u_e(x,y) = \sin x \cdot \sin y$ is an exact solution of

$$\begin{array}{rcll} \operatorname{div} \operatorname{grad} u & = & -2\ \sin x \cdot \sin y & \text{in domain} \\ \frac{\partial u}{\partial n} & = & \sin x \cdot \cos y & \text{along edge } y = 1 \\ u & = & u_e & \text{on remaining boundary} \end{array}$$

Let $h > 0$ be the typical length of a side of a triangle. By choosing different values of h we should observe smaller errors for smaller values of h. We measure the error by computing the maximal difference of the exact and approximate solutions. A double logarithmic plot leads to figure 9. the slope of the curve is approximately 2 and thus we conclude $error \approx c \cdot h^2$, i.e. quadratic convergence of the approximate solutions to the exact solution.

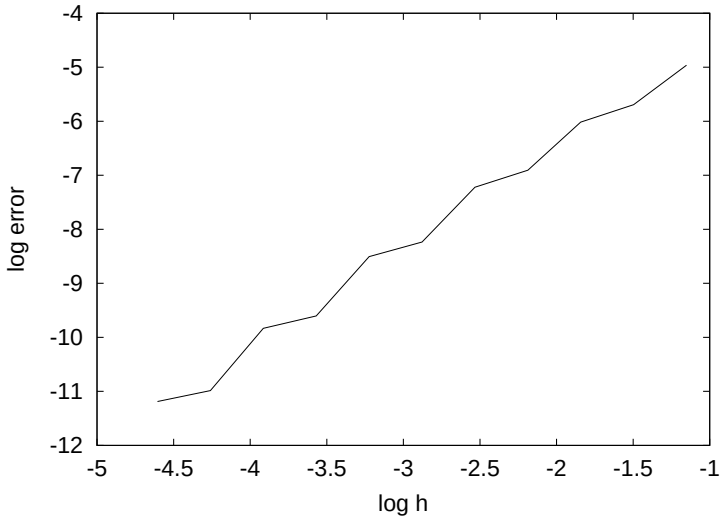

Figure 9: Logarithmic plot of error versus typical length of triangle

5.10 Demo 10: test of sparse solver

This example illustrates the use of a sparse solver. It was written by Andy Adler and is based on SuperLU. It may be found on sourceforge [www:octaveSF]. The FEM package might be modified to use this solver for all problems, as the performance is rather good.

5.11 Demo 11: potential flow problem

Consider a laminar flow between two plates with an obstacle between the two plates. We assume that the situation is independent on one of the spatial variables and consider a cross section only shown in the figure 10. The goal is to find the velocity field $\vec{v}$ of the fluid.

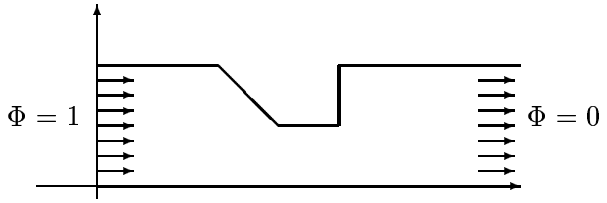

Figure 10: Fluid flow between two plates, the setup

We may introduce a velocity potential $\Phi(x, y)$. The velocity vector $\vec{v}$ is then given by

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = - \begin{pmatrix} \frac{\partial \Phi}{\partial x} \\ \frac{\partial \Phi}{\partial y} \end{pmatrix}$$

The flow is assumed to be uniform far away from the obstacle. Thus we set the potential to $\Phi = 0$ (resp. $\Phi = 1$) at the left (resp. right) end of the plates. Since the fluid can not flow through the plates we know that the normal component of the velocity has to vanish at the upper and lower boundary. The differential equation to be satisfied by Φ is

$$\Delta \Phi = \text{div} \, (\text{grad} \, \Phi) = 0$$

In figure 11 the vector field for the velocity $\vec{v}$ is shown. The vector filed was computed with the help of `FEMValue()`. Find the code in `demorun2.m` . With `demorun.m` and `demorun3.m` different meshes are used.

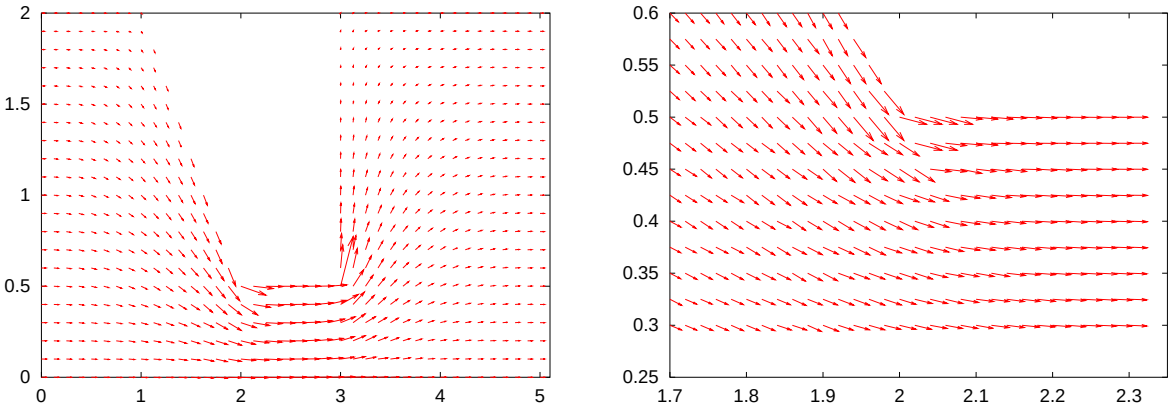

Figure 11: Velocity field of a ideal fluid, full view and details

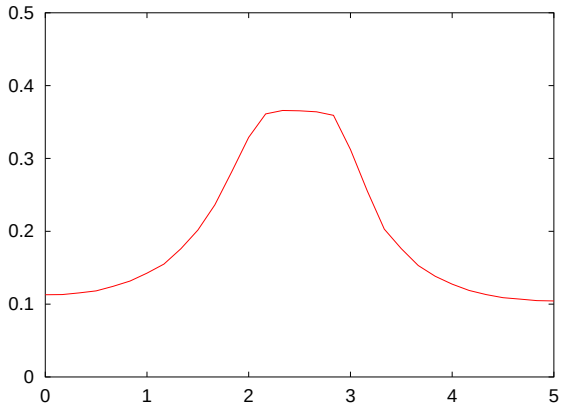

Figure 12: Horizontal velocity along a horizontal straight line

List of Figures

References

[GoluVanLoan96] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, third edition, 1996.

[John87] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method.* Cambridge University Press, 1987.

[Loga92] D. L. Logan. *A First Course in the Finite Element Method.* PWS–Kent, second edition, 1992.

[www:octaveSF] Octave at sourceforge. http://sourceforge.net/projects/octave/.

[Redd84] J. N. Reddy. *An Introduction to the Finite Element Analysis.* McGraw–Hill, 1984.

[www:sha] A. Stahel. Web page. www.hta-bi.bfh.ch/~sha.

[VarFem] A. Stahel. Calculus of Variations and Finite Elements. Lecture Notes used at HTA Biel, 2000.

Index