Getting started with EZTFEM

Martien A. Hulsen

October 4, 2025

Getting started with EZTFEM

Martien A. Hulsen
Eindhoven University of Technology
Department of Mechanical Engineering
Section Polymer Technology
PO Box 513
5600 MB Eindhoven
The Netherlands
email: m.a.hulsen@tue.nl

Contents

Chapter 1

Introduction

1.1 What is EZTFEM?

EZTFEM is a simple toolkit for the finite element method (FEM) intended for use in teaching finite elements for fluid flow. It can be seen as a significantly reduced version of TFEM [1]. Since EZTFEM consists of Matlab functions only, it is easily accessible for students.

1.2 Installation

First, make sure you have Matlab on your system. Unzip the supplied zip-file of EZTFEM somewhere on your system. Start Matlab and (in the Matlab file browser) go to the top EZTFEM folder of the unzipped tree. Then run from the Matlab command line

```
>> setpath
```

to set the path of Matlab. Now you are ready to use EZTFEM. Note, that you have to run `setpath` every time you restart Matlab.

1.3 Folder layout of EZTFEM

In the file browser you see the following folders in the EZTFEM tree:

```
src
examples
```

The folder `src` contains the folders `core` and `addons`. The folder `core` contains core functions of EZTFEM, whereas the folder `addons` contains the add-ons, with a separate folder for each add-on. The following add-ons are available:

```
plotlib
poisson
stokes
meshes
```

which contain plot functions, element routines for the Poisson and Stokes equations and functions for more complicated meshes, respectively. The folder `examples` contain the examples, divided into separate folders. Each folder might contain several examples, but all of the same kind, e.g. a problem involving the Stokes equation. The following example folders are available:

```
poisson
stokes
```

1.4 Running examples

Examples are just script M-files that can be edited and run from within Matlab. For running an example, make the example folder (e.g. `examples/poisson`) your current folder in Matlab. Now run an example from the Matlab command line:

```
>> poisson1
```

If you want to edit the script files, it is advised to make a copy of the example folder first. For more details on the examples, see Chapter 2.

1.5 Documentation

Documentation on the functions in EZTFEM can be obtained using the `help` or `doc` command in Matlab. For example

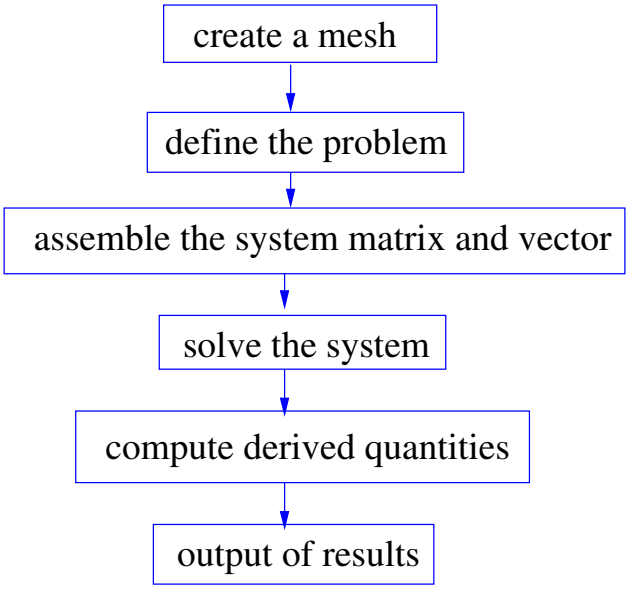
```
>> doc build_system
```

gives all the documentation for the `build_system` function. Also the `F1` function key to get help on the word 'under the cursor', works as expected.

Chapter 2

Examples

In this chapter several example problems are explained in more detail. A typical script for a problem is structured as a follows:



where the last two steps are often performed interactively outside the script.

2.1 A Poisson problem on a unit square with Dirichlet boundary conditions

Consider a Poisson problem:

$$-\nabla^2 u = f \tag{2.1}$$

on a unit square $(x, y) \in (0, 1) \times (0, 1)$. Substituting the following solution for u:

$$u = 1 + \cos(\pi x)\cos(\pi y) \tag{2.2}$$

gives the corresponding f:

$$f = 2\pi^2 \cos(\pi x)\cos(\pi y) \tag{2.3}$$

The M-file script that solves this problem is `poisson1.m` in the `poisson` folder of the examples folder. The script is listed below and is explained line by line after the listing.

```
1  % Poisson problem on a unit square with Dirichlet boundary conditions
2
3  % create mesh
4
5  mesh=quadrilateral2d([20,20],'quad9');
6
7  % define the problem
8
```

```
elementdof=[1,1,1,1,1,1,1,1,1;
            2,2,2,2,2,2,2,2,2]' ;
problem=problem_definition(mesh,elementdof,'nphysq',1) ;

% define Gauss integration and basis functions

shape='quad' ;
[xr,user.wg]=gauss_legendre(shape,'n', 3 ) ;
[user.phi,user.dphi]=basis_function(shape,'Q2', xr ) ;

% user struct for setting problem coefficients, ...

user.coorsys = 0 ;
user.alpha = 1 ;
user.funcnr = 4 ;
user.func = @func ;

% assemble the system matrix and vector

[A,f]=build_system ( mesh, problem, @poisson_elem, user ) ;

% define essential boundary conditions (Dirichlet)

iess = define_essential ( mesh, problem, 'curves', [1 2 3 4] ) ;

% fill values for the essential boundary conditions

uess = fill_system_vector ( mesh, problem, 'curves', [1 2 3 4], @func, ...
  'funcnr', 3 ) ;

% apply essential boundary conditions to the system

[A,f] = apply_essential ( A, f, uess, iess ) ;

% solve the system

u = A\f ;

% compare with exact solution

uex=fill_system_vector ( mesh, problem, 'nodes', 1:mesh.nnodes, @func, ...
   'funcnr',3 ) ;

max(abs(u-uex))

% gradient (dudx,dudy) of the solution

xr = refcoor_nodal_points ( mesh ) ;
[user.phi,user.dphi]=basis_function('quad','Q2', xr ) ;
user.u = u ;
gradu = deriv_vector ( mesh, problem, @poisson_deriv, user ) ;

```

Explanation

line 5 The mesh is created using a 20×20 mesh of nine-node quadrilaterals. The variable `mesh` is a structure having several components, including the coordinates of the nodes, the topology, the points and curves (see Figure 2.1). For more info on the `mesh`-structure see Sec. 4.1 and the documentation of the function `quadrilateral2d`). The function `quadrilateral2d` has several options[1] to create a mesh on a quadrilateral.

lines 9–11 The problem is defined. The variable `problem` is a structure having several components. The argument `elementdof` of the function `problem_definition` is a matrix where each column defines degrees of freedom in the nodes of an element. Note, that there is a transpose operator (`'`) on line 10 in the assignment of `elementdof`. The 'physical quantities' define the unknown degrees of freedom in the system vector. In this case the number of physical quantities is 1, which means that only the first column of `elementdof` defines the system vector of unknowns. The second column of `elementdof` is only used for postprocessing (see below).

lines 15–17 The Gauss-Legendre integration points (`xr`) and weights (`wg`) are computed, together with the basis functions ϕ_k, $k = 1, \ldots, 9$ and the derivatives $\partial\phi_k/\partial\xi_j$, $k = 1, \ldots, 9$, $j = 1, 2$ in these points for a bi-quadratic (Q_2) interpolation. The weights, basisfunctions and derivative are stored in the structure `user` for later use on element level.

lines 21–24 The `user` structure is further filled with data needed on elementlevel:

`coorsys` coordinate system: 0=Cartesian, 1=axisymmetric.
`alpha` diffusion coefficient α which is 1 for the Poisson equation.
`funcnr` function number in the function `func` for setting the right-hand side.
`@func` function handle to the function `func`.

line 28 Assemble the system matrix $\underline{A}$ and vector $\underset{\sim}{f}$ using the element function `poisson_elem`. Note, that the element function is supplied to the function using a function handle (there is a `@`-character in front of the function name).

lines 32–41 Define and apply Dirichlet boundary conditions. First, at line 32, an index array `iess` is computed to indicate that the degrees u(iess) need to prescribed. Then, at line 36, the prescribed values are filled in the (system) vector `uess`. Finally, at line 41, the system matrix $\underline{A}$ and vector $\underset{\sim}{f}$ are modified to take the Dirichlet conditions into account. See Sec. 4.4 how this is done.

line 45 Solve the system $\underline{A}\underset{\sim}{u} = \underset{\sim}{f}$.

lines 49–52 Print the maximum difference in the nodes, i.e. $\max |u_i - u_{i,\text{exact}}|$.

lines 56–59 Derive a column vector (array) with $\boldsymbol{\nabla} u$ in the nodes by averaging the values in elements connected to the nodes. This column vector (array), `gradu`, is defined by the second column of `elementdof` and is a structure with two components: `gradu.vec` the vector number (=column number in `elementdof`) and `gradu.u` the actual data in all nodes. In order to derive $\boldsymbol{\nabla} u$, the basis functions and the derivatives of the basis functions need to be replaced by the values in the nodes (using the function `refcoor_nodal_points`). Also, the system vector `u` needs to be available at the element level and is supplied via a component of `user` (line 58).

To plot the curves and points with numbers use

```
>> plot_points_curves(mesh)
```

[1] In the functions of EZTFEM optional arguments are specified by ..., string, value, ... couples, such as ..., 'length', [2,4], '... to specify the size of the domain in `quadrilateral2d`.

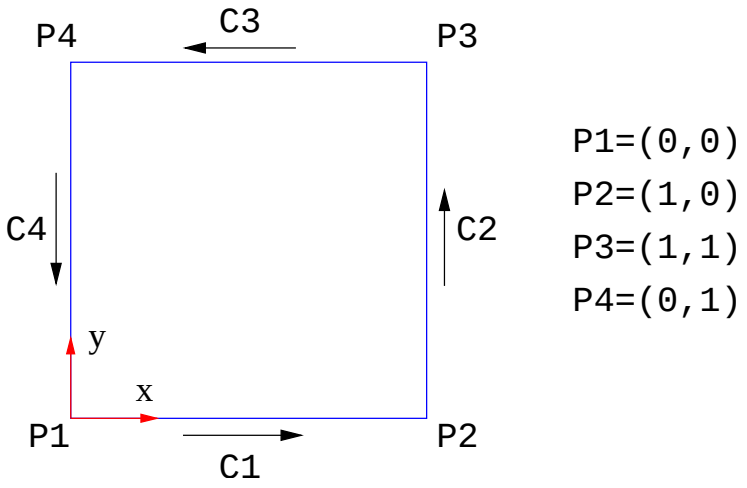


Figure 2.1: The points and curves defined by the mesh generator function `quadrilateral2d`. Points and curves are basically only placeholders for certain nodes and element edges respectively. Note the default directions of the curves.

To plot the mesh use

```
>> plot_mesh(mesh)
```

To plot the system vector (solution) use

```
>> plot_sol(mesh,problem,u)
```

or

```
>> plot_sol_contour(mesh,problem,u)
```

or

```
>> plot_sol_over_line(mesh,problem,u,[0,0;1,1])
```

In the last command, the solution is plotted over a line from the point $(0, 0)$ until $(1, 1)$. To extract the data of the plot into an array `d`, add it as an output argument:

```
>> d=plot_sol_over_line(mesh,problem,u,[0,0;1,1]);
```

or

```
>> [x,y,d]=plot_sol_over_line(mesh,problem,u,[0,0;1,1]);
```

to extract both the data and the corresponding coordinates of the data.
To plot $\partial u/\partial x$ use

```
>> plot_vector(mesh,problem,gradu,'degfd',1)
```

To plot $\partial u/\partial y$ use

```
>> plot_vector(mesh,problem,gradu,'degfd',2)
```

The functions `plot_vector_contour` and `plot_vector_over_line` are also available (see Ch. 3). See the documentation of the plot functions for the available options.

2.2 A Poisson problem on a rectangle with Dirichlet and Neumann boundary conditions

We consider the Poisson problem on a rectangle. The problem is similar to `poisson1.m` of the previous section, but now we change to domain to $(x, y) \in (0, 1.2) \times (0, 1)$ and change the boundary condition on curve `C2` to a natural boundary. The latter means that we have to specify $h_N = -\partial u/\partial n$ on curve `C2`. We substitute the following exact solution for u:

$$u = \cos(\pi x)\cos(\pi y) + x^3 y^3 \tag{2.4}$$

and find the corresponding f:

$$f = 2\pi^2 \cos(\pi x)\cos(\pi y) - 6(xy^3 + x^3 y) \tag{2.5}$$

For the natural boundary condition we need

$$h_{\mathrm{N}} = -\frac{\partial u}{\partial n} = -\frac{\partial u}{\partial x} = \pi \sin(\pi x)\cos(\pi y) - 3x^2 y^3 \tag{2.6}$$

to be evaluated on curve `C2`.

The M-file script that solves this problem is `poisson4.m` in the `poisson` folder of the examples folder. The main differences with respect to `poisson1.m` will be discussed.

First, the essential boundary conditions need to be applied on curves 1, 3 and 4. Second, the boundary integral for the natural boundary condition needs be added:

```
% define Gauss integration and basis functions (for boundary integral)

[xr,user.wg]=gauss_legendre('line','n', 3 ) ;

[user.phi,user.dphi]=basis_function('line','P2', xr ) ;

% add natural boundary condition

user.funcnr = 8 ;
f=add_boundary_elements ( mesh, problem, f, ...
       @poisson_natboun_curve, user, 'curve', 2 ) ;
```

Note, that the Gauss points and the basis functions and it's derivatives need to be redefined before using the boundary element routine! Note also, that the right-hand side vector `f` is part of the argument list for the function `add_boundary_elements`.

2.3 A Stokes problem on a unit square with Dirichlet boundary conditions: driven cavity problem

We consider the Stokes equation

$$-\nabla \cdot (2\mu \boldsymbol{D}) + \nabla p = \boldsymbol{f} \tag{2.7}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.8}$$

where $\boldsymbol{D} = (\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T)/2$, for a driven cavity problem on a unit square (see Fig. 2.2). On the upper boundary a tangential velocity of 1 is imposed and on all other walls the velocity components are zero. In the lower left corner the pressure value is set to zero (Dirichlet). The M-file script that solves this problem is `stokes1.m` in the `stokes` folder of the examples folder. We only discuss the most important difference with the `poisson1.m` problem of Sec. 2.1.

First the problem definition is different:

```
elementdof=[2,2,2,2,2,2,2,2,2;
            1,0,1,0,1,0,1,0,0;
            1,1,1,1,1,1,1,1,1]' ;
problem=problem_definition(mesh,elementdof,'nphysq',2);
```

We have two physical quantities (velocity and pressure) that make up the system vector. The first column of `elementdof` represents the velocity and the second represents the pressure (a Taylor Hood Q_2/Q_1 element). The third column is used for post processing the pressure (see below) and the velocity gradients.

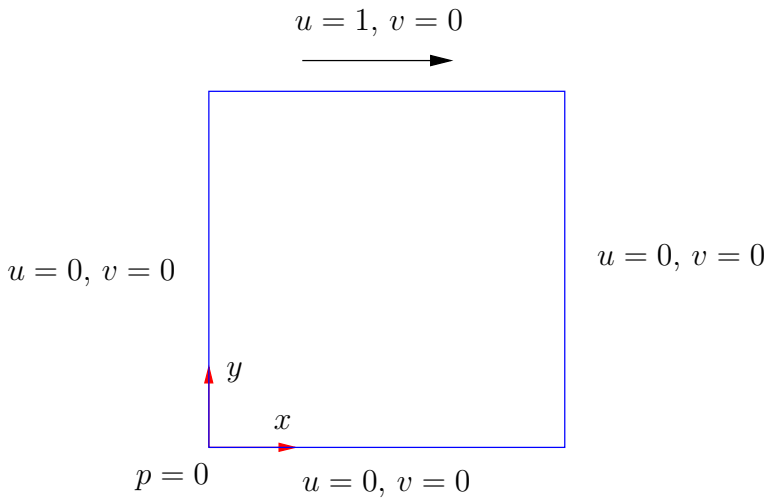


Figure 2.2: Driven cavity problem

In addition to the basis functions ϕ for the velocity, we also need basis functions ψ for the pressure:

```
[user.psi]=basis_function(shape,'Q1', xr ) ;
```

The definition of the essential boundary conditions needs a multiple call of the function `define_essential`:

```
iess = define_essential ( mesh, problem, 'curves', [1 2 3 4], 'degfd', 1 ) ;
iess = define_essential ( mesh, problem, 'curves', [1 2 3 4], 'degfd', 2, ...
  'iessp', iess ) ;
iess = define_essential ( mesh, problem, 'points', [1], 'physq', 2, ...
  'iessp', iess ) ;
```

Note, that the x and y components of the velocity requires to be treated separately. Also in subsequent calls of `define_essential`, the previously computed index array `iess` needs to be included in the arguments (`'iessp'`). Note also, that the Dirichlet condition needs the physical quantity to be specified explicitly (=2), since the default physical quantity=1.

The pressure unknowns are not defined in all nodes of the element. Therefore a new vector is derived (using `derive_vector`) that is defined in all nodes[2] It automatically becomes the third vector (`vec=3`, i.e. defined by the third column of `elementdof`):

```
xr = refcoor_nodal_points ( mesh ) ;
[user.psi]=basis_function('quad','Q1', xr ) ;
user.u = u ;
pressure = deriv_vector ( mesh, problem, @stokes_pressure, user ) ;
```

The pressure can now be plotted using

```
>> plot_vector(mesh,problem,pressure)
```

Since the velocity has two components, a vector field can be plotted using

[2]It is also possible to plot the pressure directly using `plot_sol`, but then the element function `stokes_pressure` needs to be supplied as an argument to `plot_sol`. In this way it also possible to plot fields that are discontinuous across element edges.

```
>> plot_sol_quiver(mesh,problem,u)
```

2.4 A Stokes problem on a unit square: Poiseuille flow generated by a traction (pressure) difference

This problem is similar to the driven-cavity problem of the previous section, but now traction boundary conditions are introduced (see Figure 2.3). The M-file script is `stokes2.m`. The

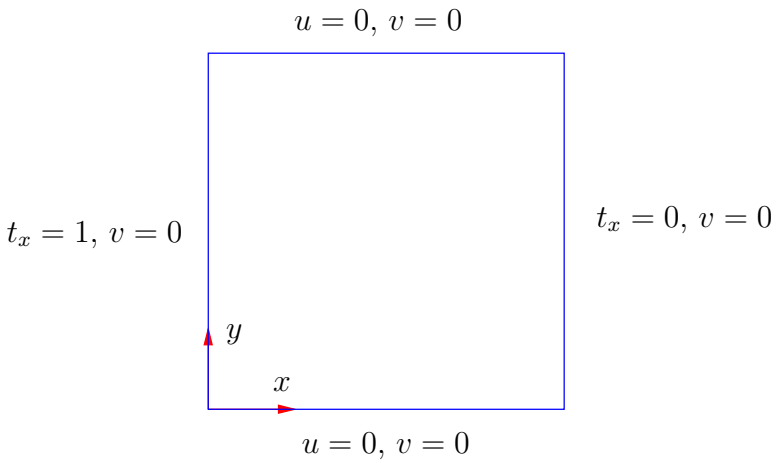


Figure 2.3: Poiseuille flow

main differences with the `stokes1.m` script are as follows. First, the traction boundary condition on curve 4 is added:

```
% define Gauss integration and basis functions (for boundary integral)

[xr,user.wg]=gauss_legendre('line','n', 3 ) ;

[user.phi,user.dphi]=basis_function('line','P2', xr ) ;

% add natural boundary condition

user.funcnr = 1 ;
user.func = @traction_func ;
f=add_boundary_elements ( mesh, problem, f, ...
       @stokes_natboun_curve, user, 'physqrow', 1, 'curve', 4 ) ;
```

Secondly, the Dirichlet conditions for the u and v components of the velocity is different:

```
% define essential boundary conditions (Dirichlet)

iess = define_essential ( mesh, problem, 'curves', [1 3], 'degfd', 1 ) ;
iess = define_essential ( mesh, problem, 'curves', [1 2 3 4], 'degfd', 2, ...
  'iessp', iess ) ;

% zero bc values

uess = zeros(problem.numdegfd,1) ;
```

where on curves 2 and 4 only v is imposed. Note, that all essential values are just zero. Hence, `uess` is just a zero vector.

Thirdly, the flow rate through curve 2 is computed using the function `integrate_boundary_elements` with the element function `stokes_flowrate_curve`:

```
% flow rate

user.u = u ;
flowrate = integrate_boundary_elements ( mesh, problem, ...
   @stokes_flowrate_curve, user, 'curve', 2 )
```

Note, that the basis functions on the curve have already been defined when adding the traction to the system!

2.5 Computing the streamfunction field from a known velocity field

For plotting streamlines in 2D flows we compute the streamfunction ψ from the velocity vector $\boldsymbol{u} = u\boldsymbol{e}_1 + v\boldsymbol{e}_2$. The definition of ψ is

$$\frac{\partial \psi}{\partial x} = -v, \qquad \frac{\partial \psi}{\partial y} = u$$

which shows that

$$\psi(\boldsymbol{x}_2) - \psi(\boldsymbol{x}_1) = \int_{\Gamma_{12}} \boldsymbol{n} \cdot \boldsymbol{u}\, ds \tag{2.9}$$

or in words: the streamfunction difference is the flow rate through a curve connecting two points. Eq. (2.9) can be used to compute ψ in all nodal points, but the procedure is cumbersome and not straightforward. Therefore, we prefer a procedure that solves the Poisson equation

$$-\nabla^2 \psi = \omega \tag{2.10}$$

with the vorticity $\omega = \partial v/\partial x - \partial u/\partial y$. We use a Neumann boundary condition:

$$\frac{\partial \psi}{\partial n} = \boldsymbol{n} \cdot \nabla \psi = \boldsymbol{n} \cdot (-v\boldsymbol{e}_1 + u\boldsymbol{e}_2) = -vn_1 + un_2 = -\boldsymbol{u} \cdot \boldsymbol{t} \tag{2.11}$$

where $\boldsymbol{t}$ is the tangential vector (with $\boldsymbol{n}$ to the 'right').

For plotting streamlines in an axisymmetrical flow we have

$$\frac{\partial \psi}{\partial z} = -2\pi r v, \qquad \frac{\partial \psi}{\partial r} = 2\pi r u \tag{2.12}$$

where (z, r) are cylindrical coordinates[3] and u is the velocity component in axial (z) direction and v the velocity component in radial (r) direction ($\boldsymbol{u} = u\boldsymbol{e}_z + v\boldsymbol{e}_r$). We easily find that

$$\psi(\boldsymbol{x}_2) - \psi(\boldsymbol{x}_1) = \int_{\Gamma_{12}} \boldsymbol{n} \cdot \boldsymbol{v}\, 2\pi r\, ds$$

or in words: the streamfunction difference is the flow rate through a surface consisting of a curve connecting two points that is expanded over the full circumferential (θ) direction. We find from (2.12) that

$$-\nabla^2_{zr} \psi = -\left(\frac{\partial^2 \psi}{\partial z^2} + \frac{\partial^2 \psi}{\partial r^2}\right) == 2\pi(r\omega + u)$$

[3] We use the convention (z, r), i.e. z is the first coordinate, to preserve the ordering of the coordinates to represent a right-handed system.

with the vorticity $\omega = \partial v/\partial z - \partial u/\partial r$. Note that ∇^2_{zr} is the 'planar' (z, r) Laplace operator and not the real ∇^2 expressed in a cylindrical coordinate system. We use a Neumann boundary condition:

$$\frac{\partial \psi}{\partial n} = \boldsymbol{n} \cdot \nabla \psi = 2\pi r \boldsymbol{n} \cdot (-v\boldsymbol{e}_z + u\boldsymbol{e}_r) = 2\pi r(-vn_z + un_r) = -2\pi r \boldsymbol{u} \cdot \boldsymbol{t} \tag{2.13}$$

where $\boldsymbol{t}$ is the tangential vector (with $\boldsymbol{n}$ to the 'right'). Note, that the Neumann condition is zero on the center line ($r = 0$).

The M-file script that solves the streamfunction problem for `stokes1.m` is a separate script `streamfunction1.m` in the `stokes` folder of the examples folder. We only discuss the most important additional EZTFEM features as compared with the `poisson1.m` and `stokes1.m` problems. Note, that `stokes1.m` must be run first.

```
elementdof=[1,1,1,1,1,1,1,1,1,;
            2,2,2,2,2,2,2,2,2]' ;
problem_s=problem_definition(mesh,elementdof,'nphysq',1);
```

The stream function is just a scalar in all nodes. The second column of `elementdof` defines a vector to 'import' the velocity field from the stokes problem to the stream function problem. A new structure `problem_s` is created different from the Stokes equation (=`problem`).

```
% Get velocities node for node
pos = pos_array(problem, [1:mesh.nnodes], 'physq', 1, 'order', 'ND' ) ;
user.v = u(pos{1}) ;
```

Using the function `pos_array` an index array `pos{1}` is computed that holds the positions of the velocity solution in the order $(u_1, v_1, u_2, v_2, \ldots, u_N, v_N)$, where N is the number of nodes. Then the actual velocity vector is stored in the component `v` of the structure `user`. See Sec. 4.3 for some more info on the function `pos_array`.

```
[A,f]=build_system ( mesh, problem_s, @streamfunction_elem, user, ...
      'posvectors', 1 );
```

The function `build_system` requires an additional argument couple (`'posvectors',1`) to supply the index of the vectors on element level.

```
for i=1:4
  f=add_boundary_elements ( mesh, problem_s, f, ...
       @streamfunction_natboun_curve, user, 'posvectors', 1, 'curve', i ) ;
end
```

The function `add_boundary_elements` is similar to `build_system`, but now assembles elements on the boundary for the Neumann boundary condition.

Plot the stream function contours using

```
>> plot_sol_contour(mesh,problem_s,streamf)
```

Don't forget to use `problem_s` for plotting the stream function!

Chapter 3

Plotting

As already seen in the examples of Ch. 2 some basic plotting functions are available within EZTFEM. Here, we give an overview of all functions, but consulting the documentation from within Matlab is needed to find out about all the options available. Note, that standard figure windows within Matlab are produced and all the options for these are available, including `hold on/off` and `axis on/off`. Also changing the figure/object properties is possible to change object or background colors, scaling factor of arrows, number of contours, line thickness etc.

Plot functions available:

`plot_gauss_legendre`: Plot the Gauss-Legendre points (as available in EZTFEM) on a reference triangle and quadrilateral.

`plot_basis_function`: Plot the basis functions (as available in EZTFEM) on a reference triangle and quadrilateral.

`plot_points_curves`: Plot all the points and curves of the mesh with labelling included. This function uses a special case of the more general function `plot_curves`.

`plot_curves`: Plot curves, without any numbers if no further options are given. If a Matlab figure window is already open, it is just added to this figure. This can be used to plot the boundaries of the domain, if needed (see below for an example). To plot just a subset of the curves, curve numbers, nodal points etc., we refer to the documentation of the functions, as available within Matlab.

`plot_mesh`: Plot the mesh. Optional arguments are available to plot node marks, node numbers and element numbers.

`plot_sol`: A 2D or 3D (default) color plot of the solution (system) vector is made. Without any optional arguments, the first degree of the first physical quantity is plotted.

`plot_sol_contour`: A 2D (default) or 3D contour plot of the solution (system) vector is made. It uses the Matlab `griddata` function to interpolate nodal data to a regular grid[1]. To obtain a smoother plot, the number of grid points can optionally be increased. Without any optional arguments, the first degree of the first physical quantity is plotted.

`plot_sol_quiver`: A vectorial plot using quiver (arrows) of the solution (system) vector is made. Without any optional arguments, the first physical quantity is plotted. Note, that this function plots the arrows only. It is up to the user to add a domain boundary using `plot_curves`. Use `hold on` to plot the arrows on top of a mesh or a data plot.

`plot_sol_over_line`: Plot a solution over a line. The sampled data can optionally be output to regular Matlab arrays.

`plot_vector`: A 2D or 3D (default) color plot of a vector (as derived using `deriv_vector`) is made.

[1] Default grid: $N \times N$ points, with $N = 2\sqrt{n_{\text{elem}}} + 1$ on a rectangle that is fitted to the full domain.

`plot_vector_contour`: A 2D (default) or 3D contour plot of a vector (as derived using `deriv_vector`) is made. It uses the Matlab `griddata` function to interpolate nodal data to a regular grid. To obtain a smoother plot, the number of grid points can optionally be increased.

`plot_vector_over_line`: Plot a vector over a line. The sampled data can optionally be output to regular Matlab arrays.

Example:

```
>> plot_sol_contour(mesh,problem_s,streamf,'n',20)
```

will produce a figure like in Figure 3.1 with rectangular axes. The domain boundary is not plotted. Also plotting the curves, using

```
>> plot_curves(mesh)
```

will give Figure 3.2. Note, that the axes are gone. Use `axis on` to get them back.

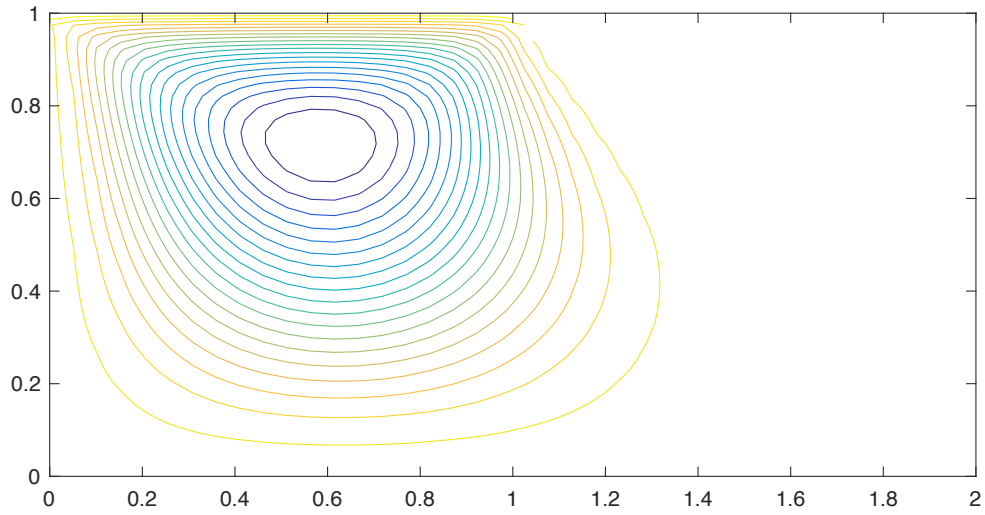


Figure 3.1: Stream function contours with rectangular axes.

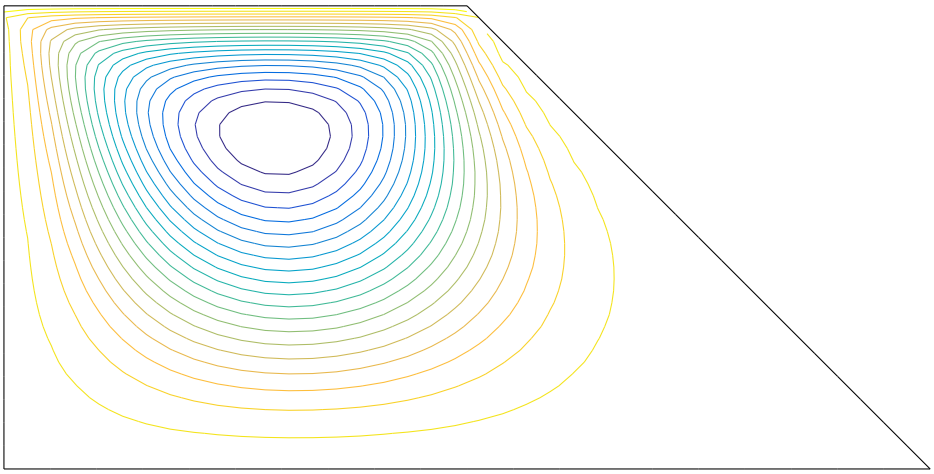


Figure 3.2: Stream function contours with boundary curves included.

Chapter 4

Data structures

4.1 mesh

The structure `mesh` contains all info related to the mesh, such as nodal coordinates and topology (connectivity) of the elements. The complete documentation of all components of the structure `mesh` is part of the documentation of the function `quadrilateral2d`. Some components are discussed here as well:

`ndim` The dimension of space (`ndim`=1 or 2).

`nnodes` The number of nodes.

`coor` An array of size `[nnodes,ndim]`, where `coor(i,:)` are the coordinates of node `i`, with $1 \leq$ `i` $\leq$ `nnodes`.

`nelem` The number of elements.

`elshape` The type of the elements. All elements have the same type. The available element types are given in Figure 4.1.

`elnumnod`: The number of nodes in a single element.

`topology` An array of size `[elnumnod,nelem]`, where the column `topology(:,elem)` contains the global node numbers element `elem` is connected to.

`npoints` The number of points.

`points` An array containing the node numbers of the points, hence `points(i)` is the node of point `i`, with $1 \leq$ `i` $\leq$ `npoints`. The main purpose of points is to easily impose boundary conditions and extract data.

`ncurves` The number of curves.

`curves` An array of structures of size `ncurves`, where each structure (curve) has components similar to a mesh. However, element types are restricted to 1 or 2 (line elements). Furthermore, a curve does not have it's own nodes. The elements of a curve are connected to the nodes of the mesh. The main purpose of curves is to easily impose boundary conditions and extract data.

It should be mentioned here, that there are various ways of defining the mesh. First, with the function `mesh_skeleton` it is possible to define a `mesh` structure, where the actual coordinates of the nodes and the topology of the elements are still missing and need to be filled by the user. Also, there are the simple internal mesh generators `line1d` and `quadrilateral2d` for 1D and 2D meshes, respectively. Furthermore, separate meshes generated by `line1d` and/or `quadrilateral2d` can be merged into one single mesh using the function `mesh_merge` to create more complicated geometries[1], such as shown in Figure. 4.2. Finally, it is of course possible to write a reading routine to import meshes from other programs, but none is available yet.

[1] The add-on `meshes` provides some functions for creating meshes for more complicated geometries in this way.

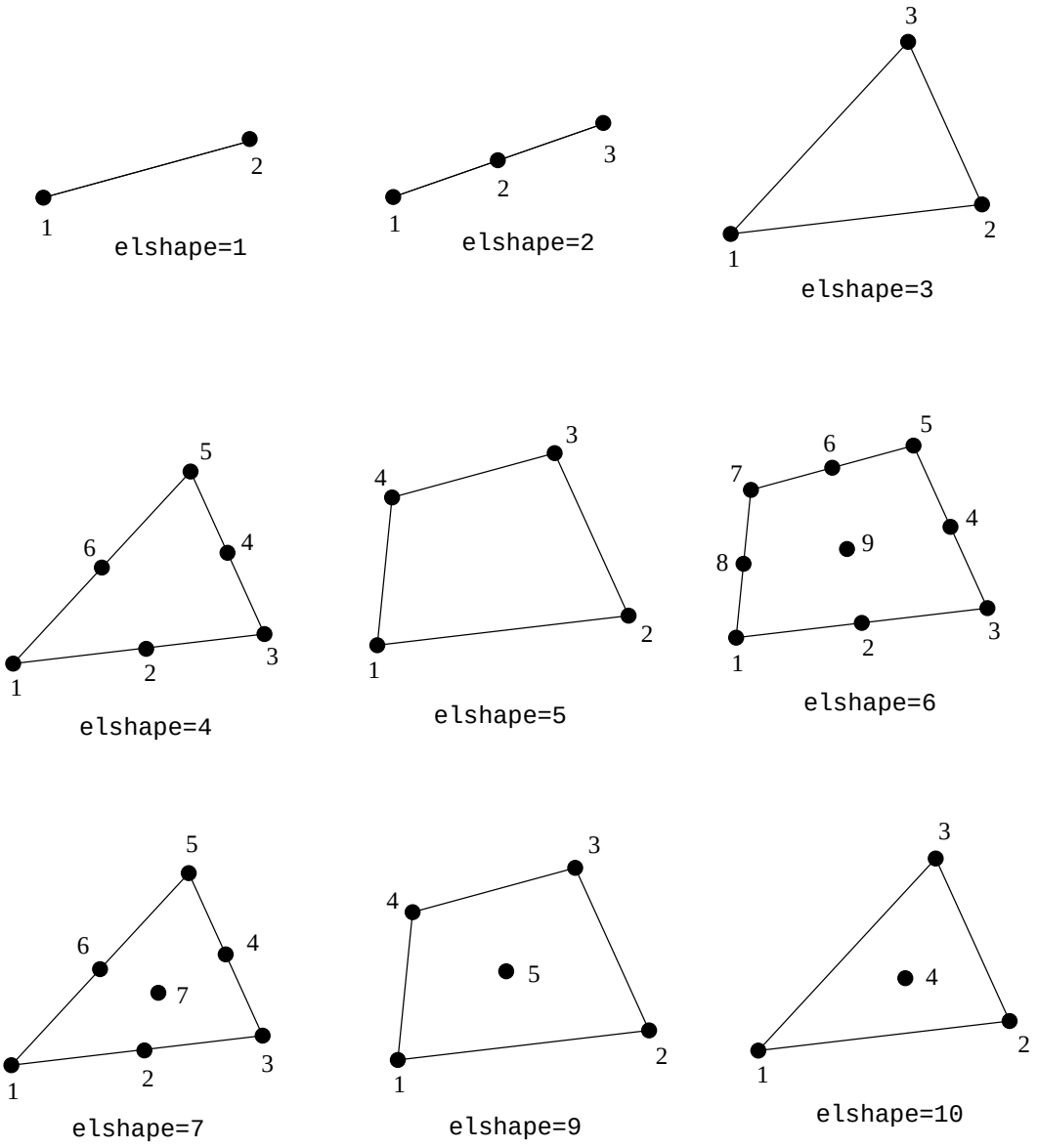


Figure 4.1: Available element types in EZTFEM.

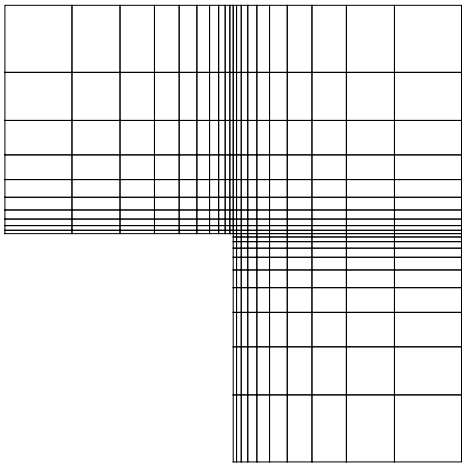


Figure 4.2: Mesh of an L-shaped domain.

4.2 `problem`

The structure `problem` contains info related to the problem, in particular the degrees of freedom defined on the mesh. The complete documentation of all components of the structure `problem` is part of the documentation of the function `problem_definition`. Some components are discussed here as well:

`nvec` The number of vectors defined on the mesh. A vector is just a storage container for degrees defined in the nodes, stored node-for-node, i.e. first the degrees in node 1, then 2 etc.

`vec_elnumdegfd` An array of size `[mesh.elnumnod,nvec]`, where a column of the matrix `vec_elnumdegfd(:,vec)` gives the number of degrees of freedom in each node of an element for vector `vec`. Note, that `vec_elnumdegfd` is identical to the matrix `elementdof` in the call of the function `problem_definition`.

`vec_numdegfd` An array of length `nvec` storing the number of degrees of freedom for each vector, i.e. `vec_numdegfd(vec)` gives the the number of degrees of freedom of vector `vec`.

`nphysq` The number of physical degrees of freedom. The physical degrees of freedom define the degrees of freedom for the system (solution) vector, i.e. the degrees of freedom that need to be solved in the assembled system of equations. These system degrees of freedom are defined by the first `nphysq` columns of the matrix `vec_elnumdegfd`, i.e. the first column defines the first physical degree, the second column defines the second physical degree etc.

`elnumdegfd` An array of size `mesh.elnumnod`, giving the number of (system) degrees of freedom in each nodal point of an element. In fact:

```
elnumdegfd=sum(vec_elnumdegfd(:,1:nphysq,2))
```

`numdegfd` The number of system degrees of freedom.

4.3 Vectors

EZTFEM makes a distinction between "vectors" and "system vectors". A vector is defined by the column `problem.vec_elnumdegfd(:,vec)` for vector number `vec` (see Sec. 4.2). A vector is used for data storage and are usually a result of the call to the function `deriv_vector`.

For the storage of a vector a structure with two components (`vec`,`u`) is used, where `vec` is the vector number and `u` the actual data.

System vectors are defined by `problem.elnumdegfd` and make up the degrees of freedom that end up in the system of equations. The storage container is a standard Matlab array/matrix with a single column.

Although both vectors and system vectors are stored 'node for node', it is advised to use pos_array functions to extract data from these arrays. For example

```
idx = pos_array_vec ( problem, [1,2,10], 'vec', 2 ) ;
```

where `idx` is a cell array with `idx{1}` the index array (positions) into the data of the vector (with `vec=2`) in the nodes 1, 2 and 10. Note, that the default 'ordering' is first degree 1 in all nodes, then degree 2 in all nodes, etc. See the documentation of `pos_array_vec` on how to change the ordering. Similarly,

```
idx = pos_array ( problem, [1,2,10] ) ;
```

gives a cell array `idx`, where `idx{i})` gives an index array into the system vector of the positions of the physical quantity `i` in the nodes 1, 2 and 10. Note, that the default 'ordering' is first degree 1 in all nodes, then degree 2 in all nodes, etc. See the documentation of `pos_array` on how to change the ordering.

4.4 System matrix and right-hand side

The system matrix $\underline{A}$ is stored in Matlab as a (square) sparse matrix. The right-hand side $\underset{\sim}{f}$ is stored just like the system (solution) vector $\underset{\sim}{u}$, i.e. the degrees of freedom are stored node-for-node. The column and row dimensions are given by all degrees, including the ones imposed by a Dirichlet condition. Therefore, we need to find a way to impose the known values and only solve for the unknown degrees. In TFEM [1] this is done by partitioning of the system of equations $\underline{A}\underset{\sim}{u} = \underset{\sim}{f}$, as follows:

$$\begin{pmatrix} \underline{A}_{\mathrm{uu}} & \underline{A}_{\mathrm{up}} \\ \underline{A}_{\mathrm{pu}} & \underline{A}_{\mathrm{pp}} \end{pmatrix} \begin{pmatrix} \underset{\sim}{u}_{\mathrm{u}} \\ \underset{\sim}{u}_{\mathrm{p}} \end{pmatrix} = \begin{pmatrix} \underset{\sim}{f}_{\mathrm{u}} \\ \underset{\sim}{f}_{\mathrm{p}} \end{pmatrix}$$

where subindices u and p mean unknown and prescribed (Dirichlet) degrees, respectively. From the partitioned system we find that, after substituting the Dirichlet condition $\underset{\sim}{u}_{\mathrm{p}} = \underset{\sim}{u}_{\mathrm{D}}$:

$$\underline{A}_{\mathrm{uu}}\underset{\sim}{u}_{\mathrm{u}} = \underset{\sim}{f}_{\mathrm{u}} - \underline{A}_{\mathrm{up}}\underset{\sim}{u}_{\mathrm{D}}$$

which is a system having a size equal to the unknown degrees of freedom $\underset{\sim}{u}_{\mathrm{u}}$. Although the system obtained is of optimal size, it is more complicated to implement and for EZTFEM we opt for the more simple approach put forward in [2]. We solve for the following system, which is of the size of the original full system:

$$\begin{pmatrix} \underline{A}_{\mathrm{uu}} & \underline{0} \\ \underline{0} & \underline{I} \end{pmatrix} \begin{pmatrix} \underset{\sim}{u}_{\mathrm{u}} \\ \underset{\sim}{u}_{\mathrm{p}} \end{pmatrix} = \begin{pmatrix} \underset{\sim}{f}_{\mathrm{u}} - \underline{A}_{\mathrm{up}}\underset{\sim}{u}_{\mathrm{D}} \\ \underset{\sim}{u}_{\mathrm{D}} \end{pmatrix}$$

where $\underline{I}$ is an identity matrix of the size of the prescribed degrees $\underset{\sim}{u}_{\mathrm{p}}$. This procedure is implemented in the function `apply_essential`. Note, that the partitioning is not really needed and the original node-for-node sequence of degrees of freedom is kept, where all degrees (unknown and prescribed) keep their original position. A disadvantage of this approach, apart from the slightly bigger system to solve, is the addition of eigenvalue 1 with a multiplicity equal to the number of prescribed degrees of freedom. The user should be aware of this of when analyzing the spectrum of the system matrix in Matlab.

References

[1] Martien A. Hulsen. TFEM *Userguide*.

[2] Howard C. Elman, David J. Silvester, and Andrew J. Wathen. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics*. Oxford University Press, Oxford, Second edition, 2014.