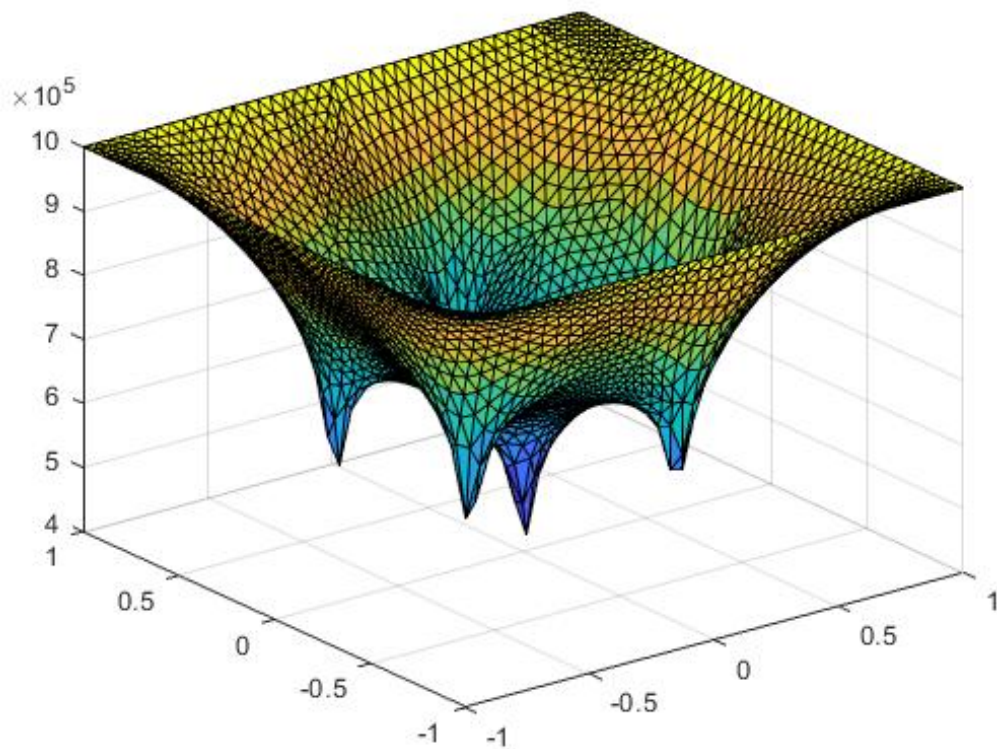# Finite Elements

Rick Koster
Ruben Termaat
February 28, 2018

Preface

This report was written in order to better understand and demonstrate what one can do with the theory of finite elements in combination with MATLAB. Solving boundary value problem through finite elements will help engineers understand difficult dynamics of systems.To show this first a general 1D problem with boundary conditions is presented, solved and computed. Secondly a real life problem is presented, where the flow velocity and pressure within a square reservoir for water filtration is calculated. This is done by solving the boundary value problem of a square reservoir of domain $\Omega = (-1; 1) \times (-1; 1)$ and its boundary conditions, then adapting the 1D MATLAB code to this 2D case and finally computing and plotting velocity flow charts and a 3D surface plot.

Contents

Chapter 1

1D-case

On the 1D interval of $x = [0, 1]$, we consider a steady-state convection-diffusion-reaction equation, with homogeneous Neumann boundary conditions. The following equations apply to this domain:

$$\begin{cases} -D\triangle u + \lambda u = f(x), \\ -D\frac{du}{dx}(0) = 0, \\ -D\frac{du}{dx}(1) = 0 \end{cases} \tag{1.1}$$

In this report $\triangle$ denotes the laplacian operator. The function f(x) is a given funtion, where D and λ are positive real constants. In order to solve this boundary value problem (BVP), first the interval is divided in n-1 elements(n = positive integer). This results in the domain being divided in elements: $e_i = [x_i, x_{i-1}]$ where $i = 1, 2, .., n$.

In order to solve this BVP, the solutions for the given equations will first be calculated and then computed using MATLAB codes.

1.1 Boundary value problem 1D

In order to find the Weakform of the given equations(1.1), both sides are multiplied by a test function $\phi(x)$ and then integrate both sides over the domain Ω. In the equations $\phi(x)$ is written as ϕ

$$\int_\Omega \phi(-D\triangle u + \lambda u)d\Omega = \int_\Omega \phi f(x)d\Omega \tag{1.2}$$

Now by rewriting and then using partial integration the following equation can be found:

$$\int_\Omega (\nabla \cdot (-D\phi \cdot \nabla u) + D\nabla\phi\nabla u + \phi\lambda u)d\Omega = \int_\Omega \phi f(x)d\Omega \tag{1.3}$$

Applying Gauss on the first term on the left side of equation(1.3):

$$\int_\Omega \vec{n} \cdot (-D\phi\nabla u)d\tau + \int_\Omega (D\nabla\phi \cdot \nabla u + \phi\lambda u)d\Omega = \int_\Omega \phi f(x)d\Omega \tag{1.4}$$

Using the boundary conditions from equations(1.1) the boundary integral equals to 0 and then the following weak formulation(WF) is found:

(WF):

$$\begin{cases} \text{find u } \epsilon \sum = \{u \; smooth\} \text{ Such that:} \\ \int_\Omega (D\nabla\phi \cdot \nabla u + \phi\lambda u)d\Omega = \int_\Omega \phi f(x)d\Omega \\ \forall \phi \in \sum \end{cases} \tag{1.5}$$

The next step is to substitute the Galerkin equations into the found diferential equation, where u is replaced by $\sum_{j=1}^{n} c_i \phi_j$ and $\phi(x) = \phi_i(x)$ with $i = [1,..,n]$. Filling this in equation (1.5) the following equation is found:

$$\sum_{j=1}^{n} c_i \int_0^1 (D\nabla\phi_i \cdot \nabla\phi_j + \lambda\phi_i\phi_j)d\Omega = \int_0^1 \phi_i f(x)d\Omega \tag{1.6}$$

Which is of the form of $S\vec{c} = \vec{f}$

1.2 Element matrix

Now the found Galerkin equations can be used to compute S_{ij} the element matrix, over a generic line element e_i.

$$S\vec{c} = \sum_{j=1}^{n} c_i \int_0^1 (D\nabla\phi_i \cdot \nabla\phi_j + \lambda\phi_i\phi_j)d\Omega \tag{1.7}$$

$$S_{ij} = \sum_{l=1}^{n-1} S_{ij}^{e_k} \tag{1.8}$$

Now to solve S we solve the following equation, over the internal line element.

$$S_{ij}^{e_k} = -D \int_{e_k} \nabla\phi_i \cdot \nabla\phi_j d\Omega + \lambda \int_{e_k} \phi_i\phi_j dx \tag{1.9}$$

1.3 Element vector

Again the found Galerkin Equations(1.6) are used in order to compute the element vector f_i over a generic line-element.

$$f_i^{e_k} = \int_{e_k} \phi_i f dx \tag{1.10}$$

$$f_i^{e_k} = \frac{|x_k - x_{k-1}|}{(1+1+0)!} f(\vec{x}) = \frac{|x_k - x_{k-1}|}{2} \begin{bmatrix} f_{k-1}^{e_n} \\ f_k^{e_n} \end{bmatrix} \tag{1.11}$$

1.4 Boundary value problem 1D MATLAB routine

1.4.1 mesh and elmat code

The first step in order to solve the BVP is to write a MATLAB routine that generates an equidistant distrubition of points over the given interval of $[0,1]$(generate a mesh with n-1 elements).

```
1    function [ x ] = GenerateMesh(int, N_elem)
2    %GenerateMesh Creates a mesh for 1D problems
3
4    x = linspace(int(1,1),int(1,2),N_elem);
5    end
```

Using the codes to generate a mesh and the elmat, it is easier to use this 1D problem and adapt to a higher dimensional problem. The next step is to write a code that generates a two dimensional array, called the elmat.

```
1        function [ elmat ] = GenerateTopology( N_elem )
2        %GenerateTopology Creates the topology for a 1D problem given mesh 'x'.
3
4        elmat = zeros(N_elem,2);
5        elmat(i,1) = i;
6        elmat(i,2) = i + 1;
7        end
```

1.4.2   Element matrix

Now that the base MATLAB codes are made the element matrix and element vector codes can be written. The first step in this process is, is to compute the element matrix S_{elem}.

```
1        function [ Selem ] = GenerateElementMatrix( k, elmat, D, lambda, mesh)
2        %GenerateElementMatrix Creates element matrix S_ek
3
4        Selem = zeros(2,2);
5
6        i = elmat(k,1);
7        j = elmat(k,2);
8
9        x1 = mesh(i);
10       x2 = mesh(j);
11
12       element_length = abs(x1-x2);
13
14       slope = 1/element_length;
15
16       for m = 1:2
17           for n = 1:2
18               if m == n
19                   Selem(m,n) = element_length*((-1)^(abs(m-n))*D*slope^2
20                   + (2)*lambda/6);
21               else
22                   Selem(m,n) = element_length*((-1)^(abs(m-n))*D*slope^2
23                   + (1)*lambda/6);
24               end
25           end
26       end
27       end
```

1.4.3   Assemble matrix S

To generate a n-by-n matrix S, the sum over the connections of the vertices in each element matrix, over all elements has to be calculated. The following code computes this matrix:

```
1        function [ S ] = AssembleMatrix( N_elem, int, lambda, D)
2        % global N_elem
3
4        elmat = GenerateTopology(N_elem);
5
6        S = zeros(N_elem,N_elem);
7
8        for i = 1:N_elem-1
9            Selem = GenerateElementMatrix(i, elmat, int, N_elem, D, lambda);
10           for j = 1:2
11               for k = 1:2
12                   S(elmat(i,j), elmat(i,k)) =
13                   S(elmat(i,j), elmat(i,k)) + Selem(j,k);
14               end
15           end
16       end
17       end
```

All the previous code will generate a large matrix S, from the element matrices S_{elem} over each element.

1.4.4 Element vector MATLAB routine

The next step In order to solve the equation $S\vec{c} = f$ is to create a code to generate the element vector. This element vector provides information about node i and node $i+1$, which are the vertices of element e_i.

```matlab
function [ felem ] = GenerateElementVector( i, elmat, mesh )
%GenerateElementVector Creates element vector f_ek


felem = [0;0];

k1 = elmat(i,1);
k2 = elmat(i,2);

x1 = mesh(k1);
x2 = mesh(k2);

element_length = abs(x1-x2);

felem = (element_length/2*arrayfun(@functionBVP,[x1,x2]))';

end
```

Where the function $f(x)$ from the BVP is defined in the following function. The different definitions of $f(x)$ will be used in different assignments.

```matlab
function [f] = functionBVP(x)
f = 1;
%f = sin(20*x);
%f = x;
end
```

To generate the vector f, the sum over the connections of the vertices in each element matrix, over all elements $i \in \{1, ..., n-1\}$ has to be calculated.

```matlab
function [ f ] = AssembleVector( N_elem, int, lambda, D )

f = zeros(N_elem,1);
elmat = GenerateTopology(N_elem);

for i = 1:N_elem-1
    felem = GenerateElementVector(i, elmat, int, N_elem);
    for j = 1:2
        f(elmat(i,j)) = f(elmat(i,j)) + felem(j);
    end
end
```

1.4.5 Computing S and f

Now if the previous MATLAB codes are run the following happens. First, a mesh and 1D topology are build. These are needed for the S matrix and f vector. The second step is to calculate the S matrix and f vector themselves through the found equations of section 1.2 and 1.3. The final step is to use the found matrix and vector to solve the equation $Su = \vec{f}$.

1.5   Main program

The main program is simply written by assembling the previous created MATLAB code AssembleMatrix and AssembleVector and deviding the vector f by the matrix S.

```matlab
1       function [ u ] = SolveBVP(  N_elem, int, lambda, D )
2
3       S = AssembleMatrix( N_elem, int, lambda, D);
4       f = AssembleVector( N_elem, int, lambda, D);
5
6       %% Calculate u
7       x = linspace(int(1),int(2),N_elem);
8
9       u = S\f;
10      plot(x,u);
```

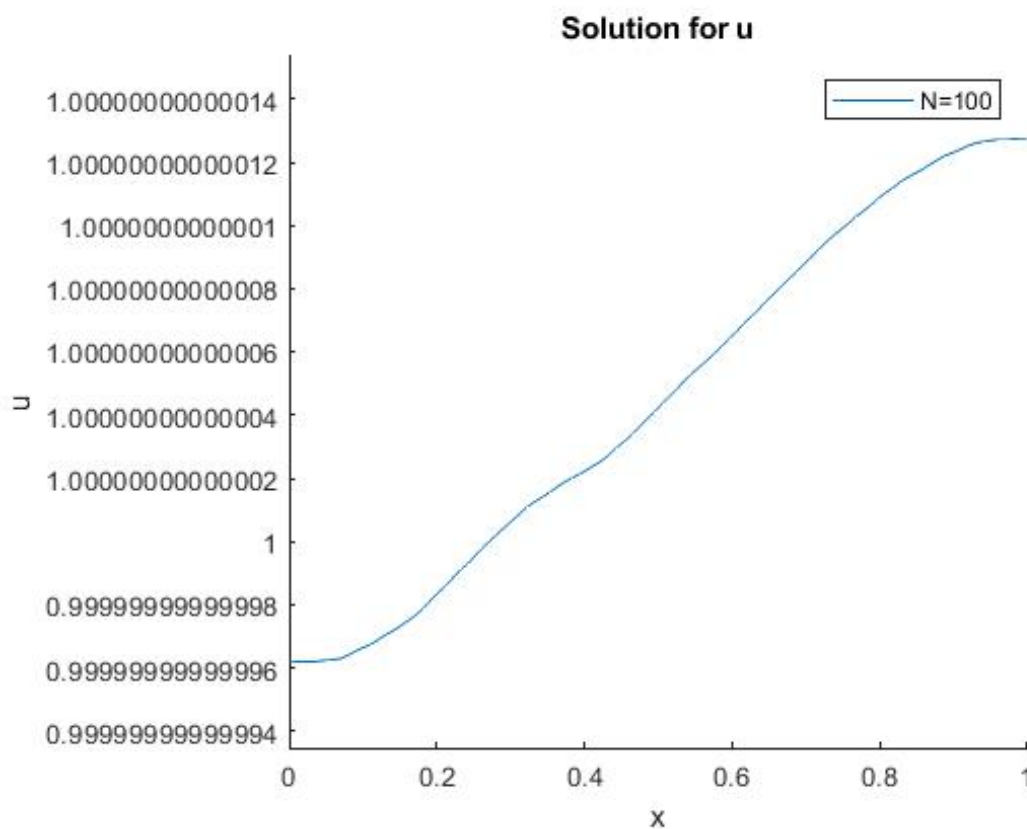The result of running this MATLAB code is shown in figure(1.1).

Figure 1.1: showing the calculated u versus x, with N = 100,f(x)=1

Figure 1.1 shows the solution for $u = \vec{f}/S$ for $N = 100$ and $f(x) = 1$. The domain is divided by equal spaced elements(follows from $f(x) = 1$). Even though it is not a very stable or smooth curve, the solution does meet the requirement for the boundary condition where $du/dx = 0$ at $x = 0$ and $x = 1$.

1.6   Solution for u

The final step is to combine all the codes in a main code to solve $Su = \vec{f}$. This code can be found in Appendix A. Previously the S matrix and f vector were computed for $n = 100$. Now u will be calculated for $f(x) = 1$, $D = 1$, $\lambda = 1$ and $N = 100$. The result of this is plotted in figure(1.2).

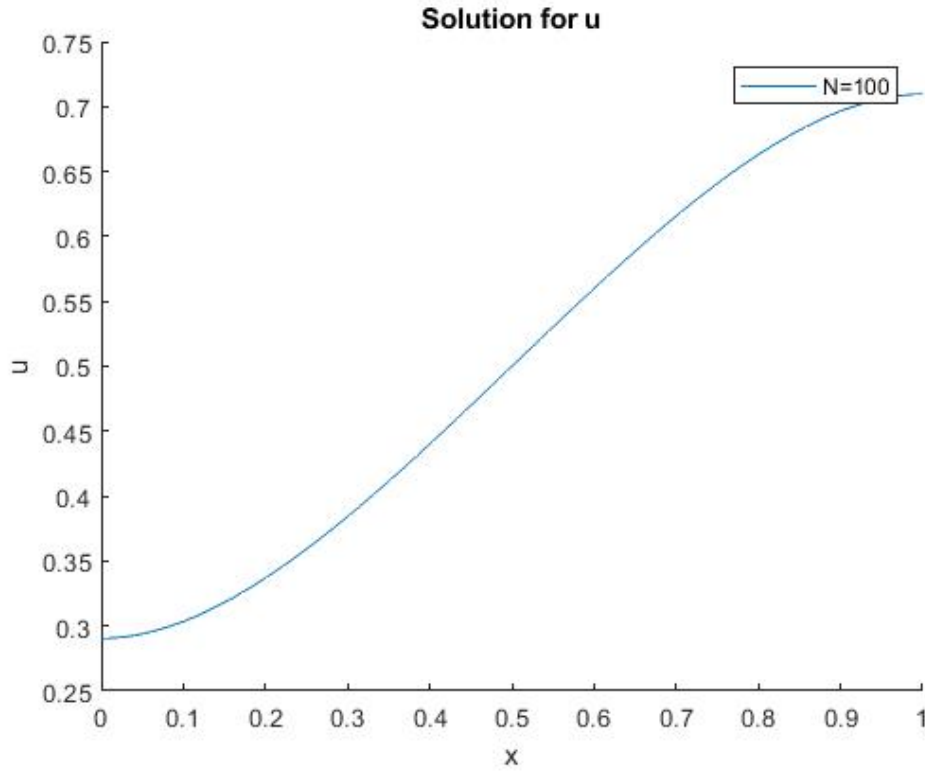Figure 1.2: showing the calculated u versus x, with $N = 100$, $f(x) = x$

Figure(1.2) shows the solution of $u = \frac{\vec{f}}{S}$ for $N = 100$ elements. In this case f(x)=x is used as function to divide the elements. Compared to the case where f(x)=1 it can be seen that now a smoother curve is found. The solution found meets the requirement for the boundary condition where $du/dx = 0$ at $x = 0$ and $x = 1$.

1.7 Experiment

The next step is to see what happens when changing f(x) to $f(x) = sin(20x)$ and to see the difference for several values for N $(n = 10, 20, 30, 40, 80, 160)$.

```matlab
function [f] = functionBVP(x)
    f = sin(20*x);
    %f = x;
    %f = 1;
end

figure
hold on

for N_elem = [10 20 40 80 100 160]
mesh = GenerateMesh(int,N_elem);
elmat = GenerateTopology(N_elem);
S = AssembleMatrix( N_elem, lambda, D, mesh, elmat);
f = AssembleVector( N_elem, mesh, elmat);

x = linspace(int(1),int(2),N_elem);

u = S\f;
plot(x,u);

legend('N=100')
title('Solution for u')
xlabel('x')
ylabel('u')
ax.box='on'
end
hold off
```
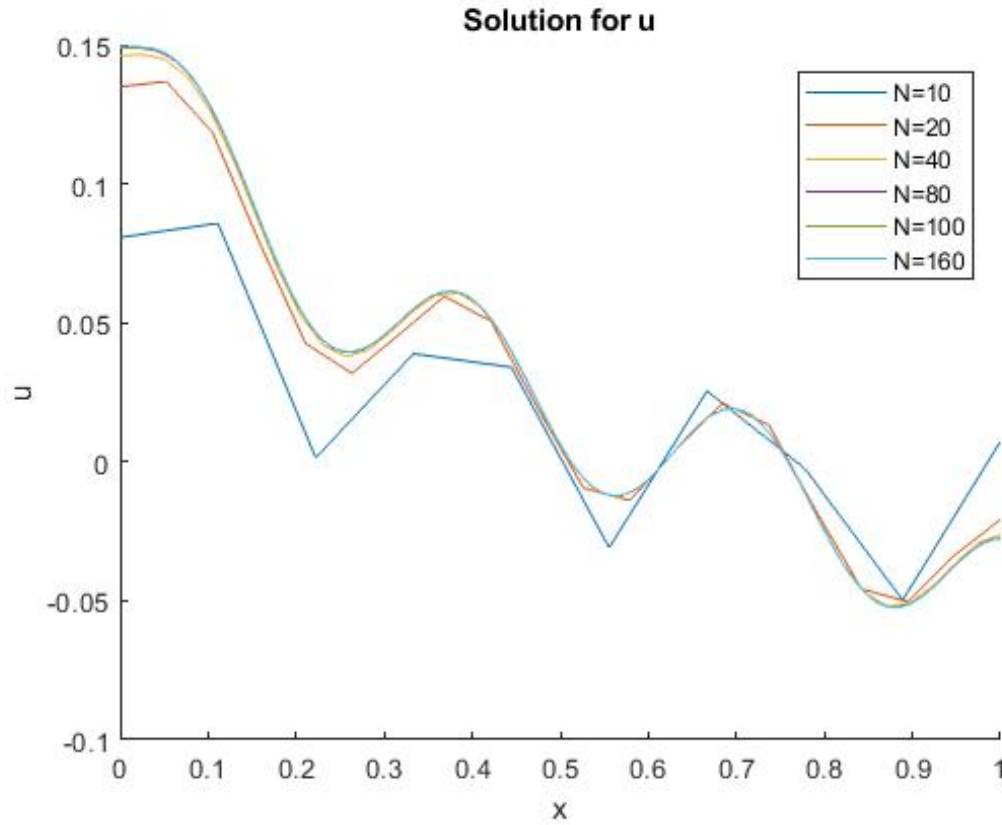
Figure 1.3: showing the calculated u versus x, with $N = [10, 20, 40, 80, 100, 160], f(x) = sin(x)$.

Looking at figure(1.3) one shows the solution for u when picking $f(x) = sin(x)$. As the number of elements N is increased, it can be seen that an increasingly smooth line is drawn. The created MATLAB code divides the domain $x = [0, 1]$ in N elements and then tries to solve the found equation while being consistent with the boundary condition. The boundary condition states that $du/dx = 0$ at $x = 0$ and $x = 1$. For low numbers of N this is not achieved, but the higher number of elements shows that above $N = 40$ the solution meets the boundary condition. Another thing one can observe is how the curve is similiar to a $sin(x)$, where a higher N value shows a better fit to a sinus.

Chapter 2

2D-case

The obvious next step after solving a 1 dimensional BVP is to adept the 1D solutions into code to solve a 2 dimensional BVP. To do this, a real life problem is going to be solved. In 3rd world countries one of the big issues is the supply of fresh water. One way is to do this is to take square reservoirs, which is a porous medium, with several wells where water is extracted from the subsurface. The water pressure is equal to the hydrostatic pressure. As this is not on an infinite domain, mixed boundary conditions are used. These boundary conditions represent a model for the transfer of the water over the boundary to locations far away. To this extent, a square domain is considered with length 2 in meter: $\Omega = (-1; 1) \times (-1; 1)$ with boundary $\partial\Omega$. Darcy's law for fluid determines the steady state equilibrium of this BVP, given by equation (2.1):

$$\vec{v} = -\frac{k}{\mu}\nabla p \tag{2.1}$$

Where p, k, μ and $\vec{v}$, respectively denote the fluid pressure, permeability of the porous medium, viscosity of water and the fluid flow velocity. In this BVP the effect of gravity will not play a part as the problem is looked at in 2D. An accompanying assumption is incompressibility, so the extraction wells are treated as point sinks. This assumption can be made as the well its diameter is much smaller than the dimensions of the square reservoir. The extraction wells extract at the same rate in each direction, leading to the following boundary conditions (2.2).

$$\nabla \cdot \vec{v} = -\sum_{p=1}^{n_{well}} Q_p \delta(\vec{x} - \vec{x}_p) = 0, \quad (x, y) \in \Omega \tag{2.2}$$

Where Q_p denotes the water extraction rate by well k, which is located at x_p. Here x equals $(x; y)$, the spatial coordinates. The convention $\vec{x} = (x; y)$ to represent the spatial coordinates is used. The dirac Delta Distribution is characterized by equation(2.3).

$$\begin{cases} \delta(\vec{x}) = 0, & \vec{x} \neq 0 \\ \int_\Omega \delta(\vec{x})d\Omega = 1, & \text{where } \Omega \text{ contains the origin.} \end{cases} \tag{2.3}$$

For this BVP the following boundary condition is considered:

$$\vec{v} \cdot \vec{n} = K(p - p^H), \quad (x, y) \in \partial\Omega \tag{2.4}$$

Where K denotes the transfer rate coefficient of the water between the boundary of the domain and its surroundings. The constant p^H represents the hydrostatic pressure.

The boundary $\partial\Omega$ is divided into four parts described by the sides of the square domain Ω. $\partial\Omega_1$ is the part with $x = -1$, $\partial\Omega_2$ is the part with $y = 1$, $\partial\Omega_3$ is the part with $x = 1$, $\partial\Omega_4$ is the part with $y = -1$.

In order to solve this BVP the values needed for all the constants are given in table (2.1).

Table 2.1: Values of input parameters

Symbol	Value	Unit
Q_p	50	m^2/s
k	10^{-7}	m^2
μ	$1.002 \cdot 10^{-3}$	$Pa \cdot s$
K	10	m/s
p^H	10^6	Pa

In this BVP six wells are considered, located at:

$$\begin{cases} x_p = 0.6\cos(\frac{2\pi(p-1)}{5}) \\ x_p = 0.6\sin(\frac{2\pi(p-1)}{5}) \end{cases} \tag{2.5}$$

For $p \in \{1, \ldots, 5\}$ and for $p = 6$ we have $x_6 = 0$ and $y_6 = 0$.

2.1   Boundary value problem 2D

The first step to solving these equations using finite elements is to find the boundary value problem to solve. This is done by filling in equation(2.1) in both equation 2.2 and the boundary condition(2.4) in order to find the BVP in terms of p:

$$\text{BVP} \begin{cases} -\frac{k}{\mu}\Delta\vec{p} = -\sum_{p=1}^{n_{\text{well}}} Q_p\delta(\vec{x} - \vec{x}_p) = 0, & (x,y) \in \Omega \\ -\frac{k}{\mu}\nabla\vec{p} \cdot \vec{n} = -\frac{k}{\mu}\frac{\partial p}{\partial n} = K(p - p^H), & (x,y) \in \partial\Omega \end{cases} \tag{2.6}$$

The next step is to compute the weak formulation using the previous found BVP(2.6). By multiplying both sides by test function $\phi(x) = \alpha_i + \beta_i x + \gamma_i y$ and integrating both sides over the domain Ω the weak formulation can be found.

$$\int_\Omega \phi(\vec{x})\nabla \cdot (-\frac{k}{\mu}\nabla\vec{p})d\Omega = \int_\Omega -\sum_{p=1}^{n_{well}} \phi(\vec{x})Q_p\delta(\vec{x} - \vec{x}_p)d\Omega \tag{2.7}$$

Using integrating by parts on the left side of equation(2.7) results in:

$$\int_\Omega \nabla \cdot \left[\phi(\vec{x})(-\frac{k}{\mu}\nabla\vec{p})\right] + \frac{k}{\mu}\nabla\phi(\vec{x}) \cdot \nabla p d\Omega = -\int_\Omega \sum_{p=1}^{n_{well}} \phi(\vec{x})Q_p\delta(\vec{x} - \vec{x}_p)d\Omega \tag{2.8}$$

Next is to apply Gauss's Theorem on the first term of the left side.

$$\int_{\partial\Omega} \vec{n} \cdot \left[\phi(\vec{x})(-\frac{k}{\mu}\nabla\vec{p})\right]d\tau + \int_\Omega \frac{k}{\mu}\nabla\phi(\vec{x}) \cdot \nabla p d\Omega = -\int_\Omega \sum_{p=1}^{n_{well}} \phi(\vec{x})Q_p\delta(\vec{x} - \vec{x}_p)d\Omega \tag{2.9}$$

Switching the integral and summation on the right side of equation(2.9) and simplifying terms:

$$\int_{\partial\Omega} (\phi(\vec{x})(-\frac{k}{\mu}\frac{\partial\vec{p}}{\partial n}))d\tau + \int_\Omega \frac{k}{\mu}\nabla\phi(\vec{x}) \cdot \nabla p d\Omega = -\sum_{p=1}^{n_{well}} \int_\Omega \phi(\vec{x})Q_p\delta(\vec{x} - \vec{x}_p)d\Omega \tag{2.10}$$

Equation(2.10) can be simplified, by using the boundary conditions (equation(2.6)) and the following property, into equation(2.12):

$$\int_\Omega \delta(\vec{x})f(\vec{x})d\Omega = f(0) \tag{2.11}$$

14

$$\int_{\partial\Omega} \phi(\vec{x})K(p - p^H)d\tau + \int_\Omega \frac{k}{\mu}\nabla\phi(\vec{x}) \cdot \nabla p d\Omega = -\sum_{p=1}^{n_{\text{well}}} \phi(\vec{x}_p)Q_p \tag{2.12}$$

Rearranging equation(2.12) so that the unknowns are on the left and the constant parts on the right leads to the following WF:

$$(\text{WF}) \begin{cases} \text{Find } p \in \sum = \{p \; smooth\} \text{ Such that:} \\ \int_{\partial\Omega} \phi(\vec{x})Kpd\tau + \int_\Omega \frac{k}{\mu}\nabla\phi(\vec{x}) \cdot \nabla p d\Omega = -\sum_{p=1}^{n_{well}} \phi(\vec{x}_p)Q_p + \int_{\partial\Omega} \phi(\vec{x})Kp^H d\tau \\ \forall \phi \in \sum \end{cases} \tag{2.13}$$

To solve the WF the Galerkin equations are applied, where p is replaced by $\sum_{j=1}^{n} c_j\phi_j$ and $\phi(x) = \phi_i(x)$.

$$\sum_{j=1}^{n} c_i \int_{\partial\Omega} \phi_i K\phi_j d\tau + \int_\Omega \frac{k}{\mu}\nabla\phi(x) \cdot \nabla\phi_j d\Omega = -\sum_{p=1}^{n_{well}} \phi(x_p)Q_p + \int_{\partial\Omega} \phi_i Kp^H d\tau \tag{2.14}$$

Equation(2.14) now is of the form $S\vec{c} = \vec{f}$ and, like with the 1D problem, $\vec{c}$ can be computed. First the element and boundary elements are determined from the Galerkin equations.

2.2 Element matrix and element vector

First the galerkin equation is seperated in its element and boundary components. The element matrix $S_{ij}^{e_k}$ and the element vector $f_i^{e_k}$ are given in equations (2.15) and 2.16 respectively. For the element matrix, boundary element matrix and boundary element vector Newton-Côtes theorem and Holand-Bell theorem are applied to simplify the equations into a set of equations that can be used for computing with MATLAB code.

$$S_{ij}^{e_k} = \int_{e_k} \frac{k}{\mu}\nabla\phi_i \cdot \nabla\phi_j d\Omega = (\beta_i\beta_j + \gamma_i\gamma_j)\frac{k}{\mu}\frac{|\triangle e_k|}{2} \tag{2.15}$$

$$f_i^{e_k} = -\sum_{p=1}^{n_{\text{well}}} \phi_i(\vec{x}_p)Q_p \tag{2.16}$$

2.3 Boundary matrix and boundary vector

The boundary matrix $S_{ij}^{be_l}$ and boundary vector $f_i^{be_l}$ can be found in the following equations:

$$S_{ij}^{be_l} = \int_{be_l} K\phi_i\phi_j dx = K\frac{|be_l|}{6}(1 + \delta_{ij}) \tag{2.17}$$

$$f_i^{be_l} = Kp^H \int_{be_l} \phi_i dx = Kp^H\frac{|be_l|}{2} \tag{2.18}$$

Where δ_{ij} is the Kronicker delta. It is assumed there are no wells on the boundary.

2.4 Wells within an internal element

To solve the BVP in 2D, one of the aspects that need to be determined is whether each internal element contains a well. So it must be determined whether the well with index p and position x_p is contained within element e_k with vertices x_{k1}, x_{k2} and x_{k3}.

This can be done according the following criterion:

$$|\Delta(\vec{x}_p, \vec{x}_{k2}, \vec{x}_{k3})| + |\Delta(\vec{x}_{k1}, \vec{x}_p, \vec{x}_{k3})| + |\Delta(\vec{x}_{k1}, \vec{x}_{x2}, \vec{x}_p)| : \begin{cases} = |e_k|, \; \vec{x}_p \in \vec{e}_k \\ > |e_k|, \; \vec{x}_p \notin \vec{e}_k \end{cases} \tag{2.19}$$

In the criterion $\Delta(\vec{x}_p, \vec{x}_q, \vec{x}_r)$ denotes the triangle with vertices $\vec{x}_p$, $\vec{x}_q$ and $\vec{x}_r$, where $|\Delta(\vec{x}_{k1}, \vec{x}_{k2}, \vec{x}_{k3})|$ denote its area. The triangular element k is given by $e_k = \Delta(\vec{x}_{k1}, \vec{x}_{k2}, \vec{x}_{k3})$ with vertices $\vec{x}_{k1}$, $\vec{x}_{k2}$ and $\vec{x}_{k3}$ and $\vec{e_k}$ includes the boundary of element e_k. To solve this BVP a certain tolerance has to be accounted for in the MATLAB code. However, while using this method it proved difficult to determine a correct tolerance to ensure every well was contained in a single element.

Therefore, an alternative method was used. A check was done whether the linear basis functions ϕ_{k1}, ϕ_{k2} and ϕ_{k3} all have values in the interval $[0,1]$ at position $\vec{x}_p$. If this is true, then the well at $\vec{x}_p$ is within the triangular element e_k.

If it is found that an element does contain a well, we see in equation (2.16) that $\phi_i(\vec{x}_p)$ for $i = \{k1, k2, k3\}$ must be found. The following code does this by determining α_i, β_i, and γ_i and subsequently filling $\vec{x}_p$ into the found $\phi_i(\vec{x})$.

```
1  for index1=1:topology
2      xc(index1) = x(elmat(i,index1));
3      yc(index1) = y(elmat(i,index1));
4  end;
5
6  Delta = det([1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)]);
7
8  B_mat = [1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)] \  eye(3);
9
10 alpha = B_mat(1,1:3);
11 beta  = B_mat(2,1:3);
12 gamma = B_mat(3,1:3);
13
14 felem = zeros(1,topology);
15
16 if ¬exist('u','var')  % Only if u is already know can the calculation of the velocity ...
        begin.
17     for N = 1:N_wells
18         for index3 = 1:topology
19             phi_p(index3) = alpha(index3) + beta(index3)*xp(N) + gamma(index3)*yp(N);
20         end
21         if (phi_p(1) ≤ 1) && (phi_p(1) ≥ 0) && (phi_p(2) ≤ 1) && (phi_p(2) ≥ 0) && ...
                (phi_p(3) ≤ 1) && (phi_p(3) ≥ 0);
22             for index1 = 1:topology
23                 felem(index1) = felem(index1) + -Qp*phi_p(index1);
24             end
25
26         end
27 end
```

2.5   Generating MATLAB code

Similar as with the 1D BVP, MATLAB code is written in order to generate a mesh, element matrix, element vector, boundary element matrix and boundary element vector. These codes can be found in appendix B.1 through B.6. The scripts have been written such that they can be used to find both the pressure field and the velocity field(the derivation fort the velocities is found in the next section).

2.6   Velocities

In order to find the velocities in x,y-direction, Darcy's law is used to compute the speed in both directions.

In order to find v_x and v_y the first step is to rewrite equation(2.20).

$$\vec{v} = -\frac{k}{\mu}\nabla p \tag{2.20}$$

$$v_x = -\frac{k}{\mu}\frac{\partial p}{\partial x} \tag{2.21}$$

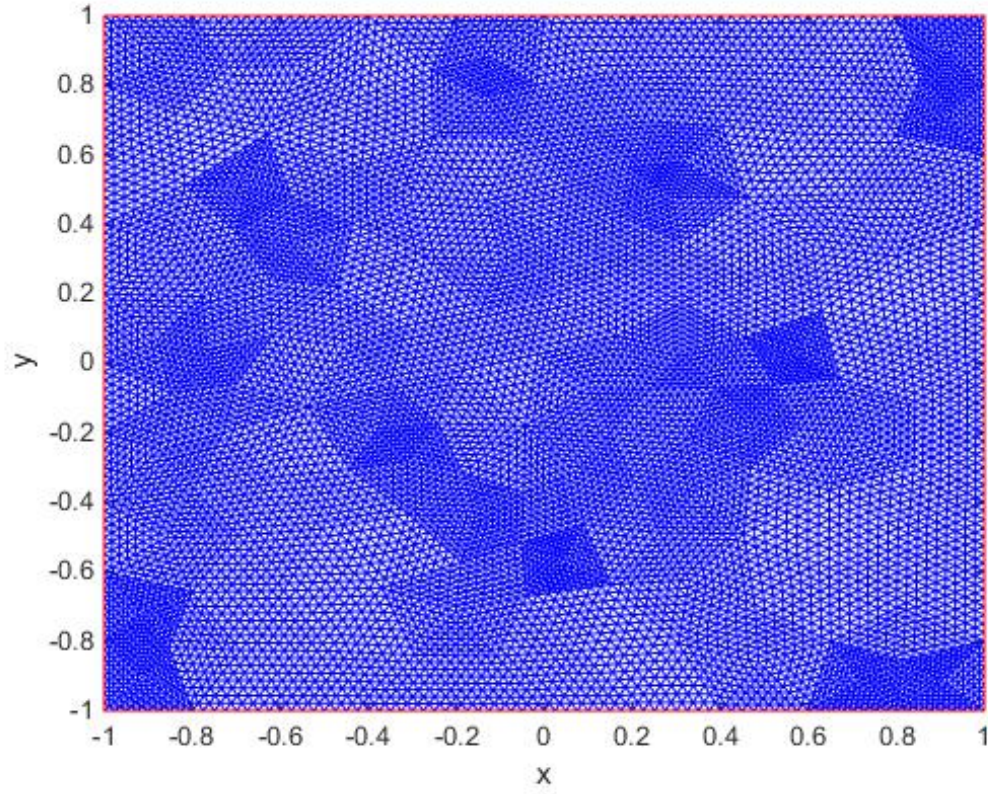$$v_y = -\frac{k}{\mu}\frac{\partial p}{\partial y} \tag{2.22}$$

Figure 2.1: Triangle element mesh over the square reservoir domain $\Omega = (-1; 1) \times (-1; 1)$.

Equation(2.24) shows the relation between $\vec{v}$ and pressure p.

$$\vec{v} \cdot \vec{n} = k(p - p^H) \text{ on } \partial\Omega \tag{2.23}$$

This relation gives the boundary conditions for v_x and v_y:

$$v_x(x = -1) = -k(p - p^H) \tag{2.24}$$

$$v_x(x = 1) = k(p - p^H) \tag{2.25}$$

$$v_y(y = -1) = -k(p - p^H) \tag{2.26}$$

$$v_y(y = 1) = k(p - p^H) \tag{2.27}$$

In order to find the weak form, again, the test function ϕ and integration over the domain Ω is used. Here follows the derivation for finding v_x, the steps for deriving v_y are similar.

$$\int_\Omega \phi v_x d\Omega = -\frac{k}{\mu} \int_\Omega \phi \frac{\partial p}{\partial x} d\Omega \tag{2.28}$$

Partial integration is applied on the right side term.

$$\int_\Omega \phi v_x d\Omega = -\frac{k}{\mu} \{ \int_\Omega \frac{\partial}{\partial x}(\phi p) - p\frac{\partial \phi}{\partial x} d\Omega \} \tag{2.29}$$

Rewriting the integral:

$$\int_\Omega -\frac{k}{\mu}\frac{\partial \phi p}{\partial x}dxdy = \int_{-1}^{1} -\frac{k}{\mu}[\phi p]_{-1}^{1}dy \tag{2.30}$$

The surface integral turns into a set of line integrals along parts of the boundary $\partial\Omega$.

$$\int_\Omega -\frac{k}{\mu}\frac{\partial \phi p}{\partial x}dxdy = \int_{-1}^{1} -\frac{k}{\mu}(\phi(x=1,y)p(x=1,y)) + \frac{k}{\mu}(\phi(x=-1)p(x=-1,y))dy \tag{2.31}$$

Simplifying the previous equations.

$$\int_\Omega -\frac{k}{\mu}\frac{\partial \phi p}{\partial x}dxdy = \int_{\partial\Omega_3} -\frac{k}{\mu}\phi p d\tau + \int_{\partial\Omega_1} \frac{k}{\mu}\phi p d\tau \tag{2.32}$$

Inserting this into equation(2.29) the following equation is found.

$$\int_\Omega \phi v_x = \frac{k}{\mu}\{\int_{\partial\Omega_3} -\phi p d\tau + \int_{\partial\Omega_1} \phi p d\tau + \int_\Omega p\frac{\partial \phi}{\partial x}d\Omega\} \tag{2.33}$$

The boundary conditions is used to rewrite p in the two boundary integrals.

$$\text{on}\begin{cases}\partial\Omega_3 : & -v_x = K(p-p^H) \rightarrow p = -\frac{v_x}{K} + p^H \\ \partial\Omega_1 : & v_x = K(p-p^H) \rightarrow p = \frac{v_x}{K} + p^H\end{cases} \tag{2.34}$$

The following weakform is derived:

$$\text{WF}\begin{cases}\text{Find } v_x \in \Sigma = \{v_x \text{ smooth}\}, \text{ such that} \\ \int_\Omega \phi v_x d\Omega + \int_{\partial\Omega_3} -\frac{k}{\mu}\frac{1}{K}\phi v_x d\tau + \int_{\partial\Omega_1} -\frac{k}{\mu}\frac{1}{K}\phi v_x d\tau = \int_{\partial\Omega_3} -\frac{k}{\mu}\phi p^H d\tau + \int_{\partial\Omega_1} \frac{k}{\mu}\phi p^H d\tau + \int_\Omega \frac{k}{\mu}p\frac{\partial \phi}{\partial x}d\Omega \\ \forall \phi \in \Sigma\end{cases} \tag{2.35}$$

To find the system of equations the following equations are filled in equation(2.35) $\phi(\vec{x}) = \phi_i(\vec{x}) = \alpha_i + \beta_i x + \gamma_i y$ and $v_x \approx \sum_{j=1}^{n} c_j\phi_j(\vec{x})$.

$$\sum_{j=1}^{n} c_j\{\int_\Omega \phi_i\phi_j d\Omega + \int_{\partial\Omega_3} -\frac{k}{\mu}\frac{1}{k}\phi_i\phi_j d\tau + \int_{\partial\Omega_1} -\frac{k}{\mu}\frac{1}{k}\phi_i\phi_j d\tau\} = \int_{\partial\Omega_3} -\frac{k}{\mu}\phi_i p^H d\tau + \int_{\partial\Omega_1} \frac{k}{\mu}\phi_i p^H d\tau + \int_\Omega \frac{k}{\mu}p\beta_i d\Omega \tag{2.36}$$

From this system of equations the element matrix and element vector are extracted.

$$S_{ij} = \int_\Omega \phi_i\phi_j d\Omega + \int_{\partial\Omega_3} -\frac{k}{\mu}\frac{1}{k}\phi_i\phi_j d\tau + \int_{\partial\Omega_1} -\frac{k}{\mu}\frac{1}{k}\phi_i\phi_j d\tau \tag{2.37}$$

$$f_i = \int_{\partial\Omega_3} -\frac{k}{\mu}\phi_i p^H d\tau + \int_{\partial\Omega_1} \frac{k}{\mu}\phi_i p^H d\tau + \int_\Omega \frac{k}{\mu}p\beta_i d\Omega \tag{2.38}$$

Now separating contributions to both S_{ij} and f_i from boundary and internal elements into $S_{ij}^{be_l}$, $S_{ij}^{e_k}$, $f_i^{be_l}$ and $f_i^{e_k}$ such that:

$$S_{ij} = \sum_{l=1}^{n_{be}} S_{ij}^{be_l} + \sum_{k=1}^{n_e} S_{ij}^{e_k} \tag{2.39}$$

$$f_i = \sum_{l=1}^{n_{be}} f_i^{be_l} + \sum_{k=1}^{n_e} f_i^{e_k} \tag{2.40}$$

Applying Newton-Côtes theorem and Holand-Bell theorem results in the following new expressions for the (boundary) element-matrix and -vector are found.

$$S_{ij}^{e_k} = \int\limits_{e_k} \phi_i \phi_j d\Omega = \frac{|\triangle e_k|}{24} \tag{2.41}$$

$$S_{ij}^{be_l} = \int\limits_{be_l} -\frac{k}{\mu} \phi_i \phi_j d\tau = \frac{k}{\mu}\frac{1}{k}\frac{|be_l|}{6}(1 + \delta_{ij}) \tag{2.42}$$

$$f_i^{e_n} = \int\limits_{e_k} \frac{k}{\mu} p \beta_i d\Omega = \frac{k}{\mu} \beta_i \sum_{m=\in\{k_1,k_2,k_3\}} p(\vec{x}_m)\frac{|\triangle e_k|}{6} \tag{2.43}$$

$$f_i^{be_l} = \int\limits_{be_l} \pm\frac{k}{\mu} \phi_i p^H d\tau = \pm\frac{k}{\mu} p^H \frac{|be_l|}{2} \tag{2.44}$$

With '+' if be_l is on $\partial\Omega_1$ and '$-$' if it is on $\partial\Omega_3$

Since the pressure field p was previously calculated, all the necessary information to compute v_x (and similarly v_y) is now derived. Therefore it is possible to now evaluate our square reservoir. In the following plots the velocities for $K = 10\,m/s$ that are computed are shown using a vector plot, contour plot and a 3D surface plot.
In figure 2.2, 2.3 and 2.4 it can be seen that there are 6 spots that stick out. Looking at the velocity field, it can be seen that the velocity is highest around these spots and lowest near the boundaries. The heat contour plot implies that where the velocity is highest, the pressure is lowest in the square reservoir and at the boundary is the highest pressure.
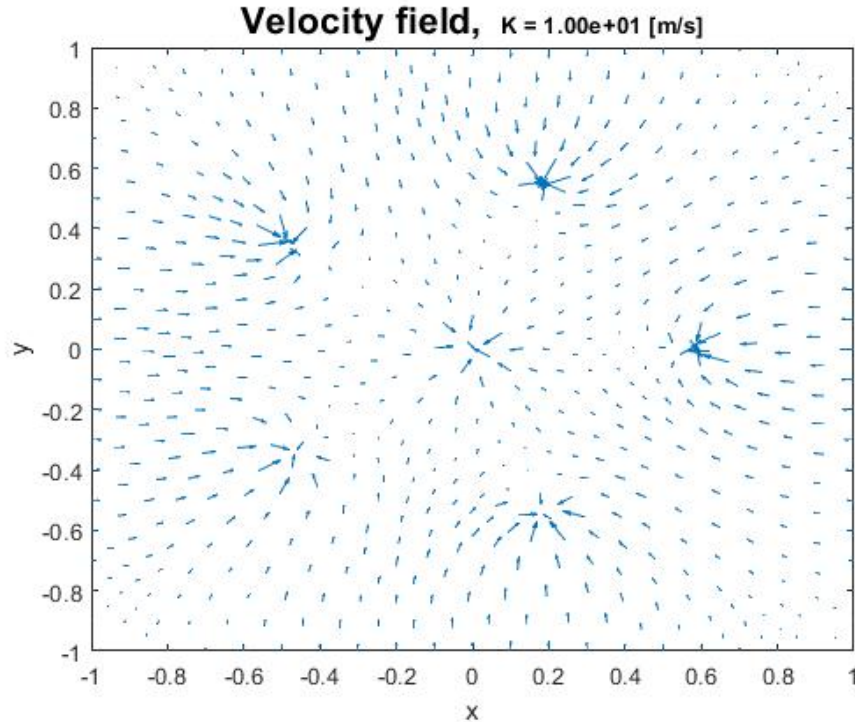
Figure 2.2: Arrow velocity plot, indicating the direction and velocity of the water(longer arrows indicate higher velocity).
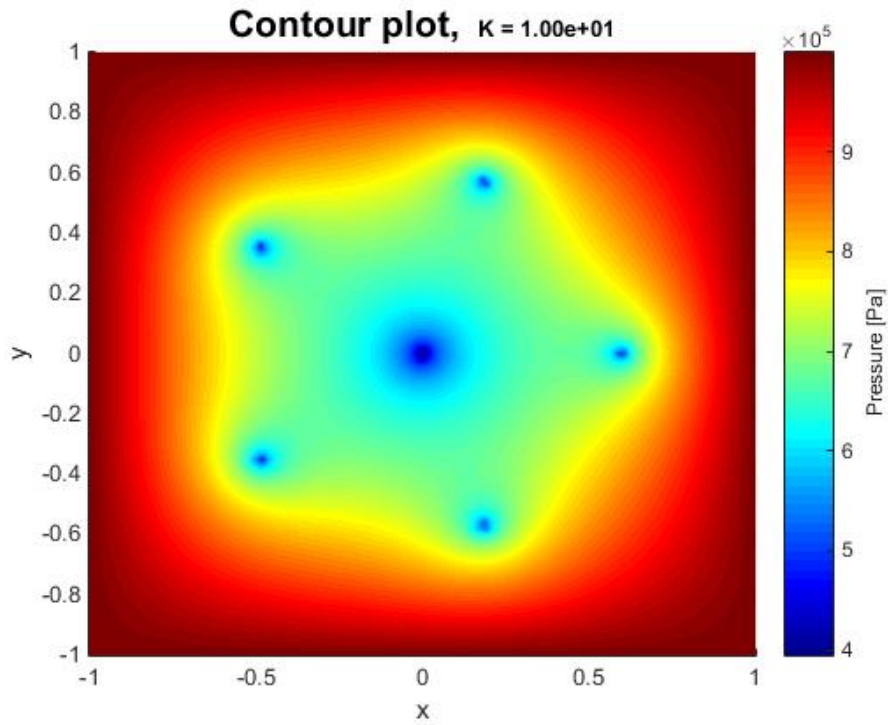
Figure 2.3: Contour plot of the velocity in the square reservoir on domain $\Omega = (-1, 1) \times (-1, 1)$, showing six areas where a drop of about two times the pressure can be observed, compared to the boundary pressure.
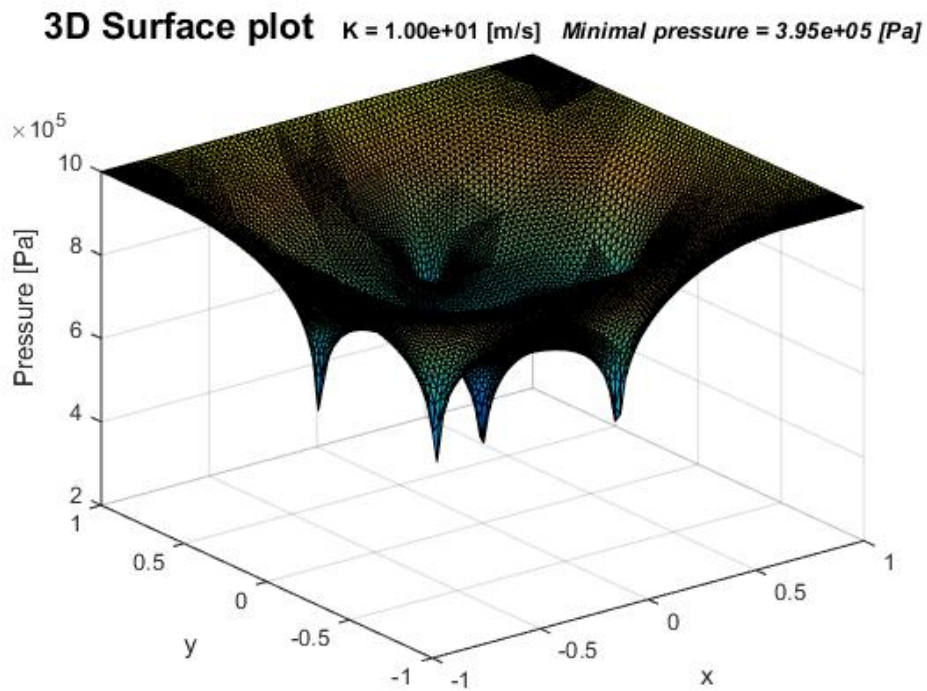
Figure 2.4: 3D surface plot for $K = 10$. The lowest peaks indicate where the pressure is at a minimum in the square water reservoir.

2.7   Varying constant K

Now that the velocities have been calculated the last thing to vary is, is the factor K, the transfer rate coefficient of the water between the boundary of the domain and its surroundings. A few different plots will now follow in which the transfer coefficient K has different values between 0.00001 and 10000.

Table 2.2: Pressure minima and velocity maxima for different values of K

| K $[m/s]$	Pressure minimum $[Pa]$	Maximum speed $[m/s]$
0.00001	$-3.40 \cdot 10^6$	616.9
0.001	$3.49 \cdot 10^5$	616.0
0.1	$3.94 \cdot 10^5$	615.9
10	$3.95 \cdot 10^5$	615.9
1000	$3.95 \cdot 10^5$	615.9
10000	$3.95 \cdot 10^5$	615.9

The purpose of the wells is to extract water from the subsurface to near the surface. In order to lift the water against gravity there must be sufficient pressure difference between the bottom and top of the well. What has been calculated in this assignment is the pressure at the bottom of the well. With an atmospheric pressure of around 101 kPa in mind the pressure at the bottow of the well must be greater by an amount depending on the depth of the well.

From the 3D surface plots it is evident that the pressure at the well in the middle, which is closer to more wells, is lower than those around it. It is thus important to know how many and how close together wells can be dug while maintaining a sufficient pressure difference between the bottom and top of the wells.

The final step is to determine what happens when $K = 0$ and why? When looking at our previous plots, when varying K from 0.00001 to 10000 it can be seen that the pressure drops the lower the value for K is. This would mean that when $K = 0$, the transfer coefficient of the water between the boundary of the square reservoir and its surroundings is zero. There will be no flow from the square reservoir into the surrounding through the boundary.
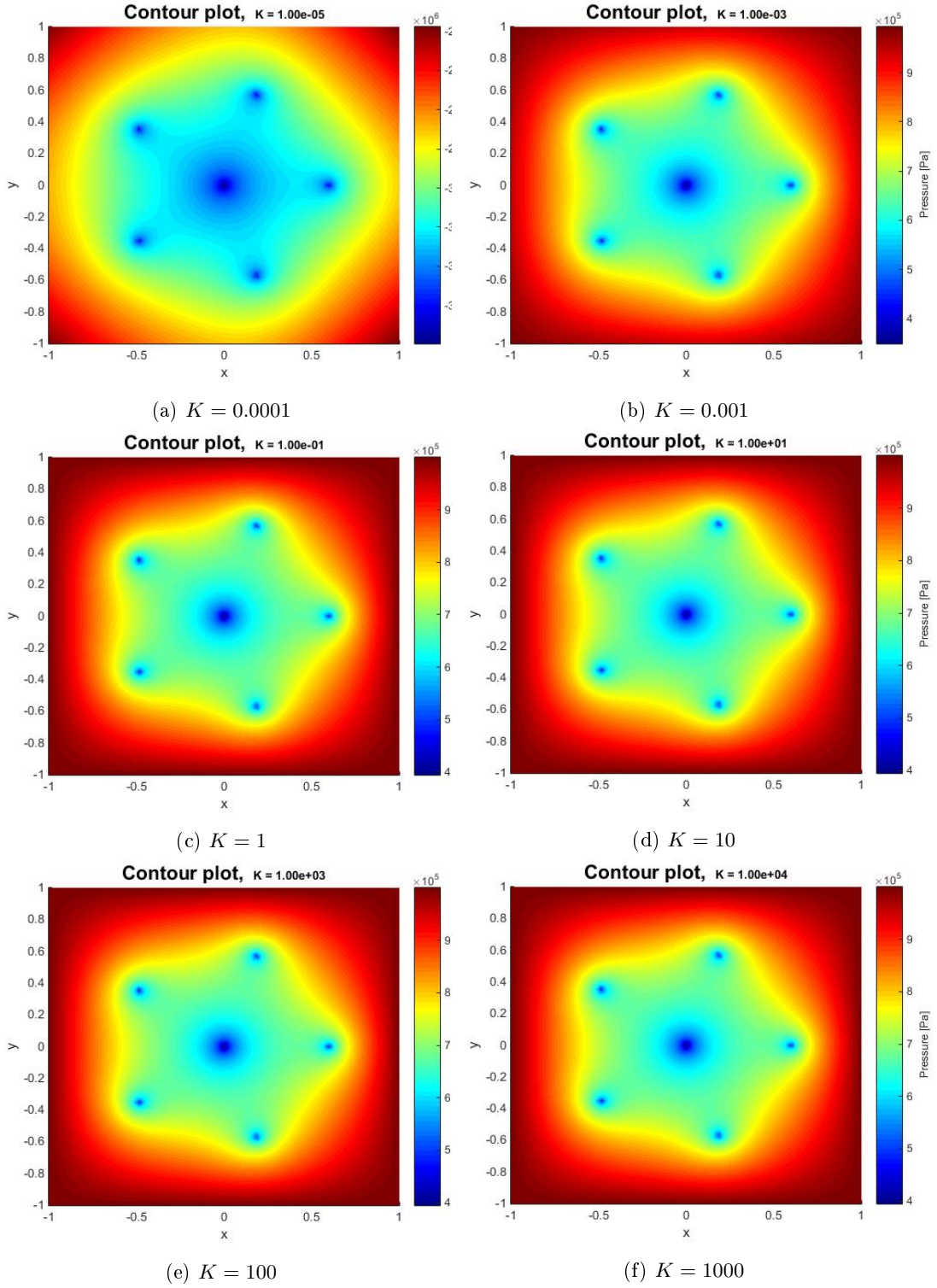
(a) $K = 0.0001$ (b) $K = 0.001$

(c) $K = 1$ (d) $K = 10$

(e) $K = 100$ (f) $K = 1000$

Figure 2.5: Contour plots for different values for K

Chapter 3

Conclusion

This report was written in order to better understand and demonstrate the application of finite elements in combination with MATLAB. This was shown by considering two boundary value problems: a 1Dcase and a 2D-case. In the 1D-case a general BVP with boundary conditions were used to show how to solve this BVP. From the 1D-case it was derived that depending on how many N and the chosen function $f(x) = function$ can greatly change the outcome of your solution. More elements(higher N) will result in a smoother curve. The function $f(x) = function$ was changed from $f(x) = 1$ to $f(x) = x$ to $f(x) = sin(x)$. The graphs showed the solution curves became increasingly smoother.

In the 2D-case a real life problem was presented. The 2D BVP solved was an underground square reservoir with six wells through which water is extracted from the subsurface. Applying finite elements to solve this BVP and then adapting the MATLAB code used in the 1D-case it was possible to create velocity field plots and heat contour and 3D surface contour plots of the pressure P. From these plots it was derived that at the wells a decrease of pressure and increase of velocity is observed. The height of the pressure and velocity depends on the K factor, the transfer rate coeficient by which the water flows from the reservoir to its surroundings. From the heat contour plots it was concluded that the pressure the overall pressure increases as the K factor is increased. When the K factor is too low, a heat contour plot is observed that could indicate that the pressure is too low at the wells for water to flow through the wells and instead will dissipate in the reservoir. At $K = 0$ it is expected that the pressure is equal everywhere in the reservoir and no transfer flow is possible. From an engineering point it is thus necessary that the K factor is above a certain treshold that allows water to flow through the well, with high enough pressure to counteract the gravity.

Appendix A

1D-case

A.1   Script

```matlab
1  clear all
2  close all
3
4  %%Finite Element 1D
5  %% Parameters
6
7  N_elem = 100; %Number of elements
8  int = [0,1]; %Interval
9  lambda = 1;
10 D = .1;
11
12 %% Mesh & Topology
13
14 mesh = GenerateMesh(int,N_elem);
15 elmat = GenerateTopology(N_elem); %1D topology!!
16
17 %% Assemble Matrix & Vector
18
19 S = AssembleMatrix( N_elem, lambda, D, mesh, elmat);
20 f = AssembleVector( N_elem, mesh, elmat);
21
22 %% Calculate u
23 x = linspace(int(1),int(2),N_elem);
24
25 u = S\f;
26
27 hold on
28 plot(x,u);
29 legend('N=100')
30 title('Solution for u')
31 xlabel('x')
32 ylabel('u')
33 ax.box='on'
34 hold off
35
36
37 % For this part change the function in functionBVP.m to 'f = sin(20*x)'
38
39 figure
40 hold on
41
42 for N_elem = [10 20 40 80 100 160]
43     mesh = GenerateMesh(int,N_elem);
44     elmat = GenerateTopology(N_elem);
45     S = AssembleMatrix( N_elem, lambda, D, mesh, elmat);
46     f = AssembleVector( N_elem, mesh, elmat);
47
48     x = linspace(int(1),int(2),N_elem);
49
50     u = S\f;
```

```
51     plot(x,u);
52
53
54 end
55
56 legend('N=10','N=20','N=40','N= 80','N=100','N=160')
57 title('Solution for u')
58 xlabel('x')
59 ylabel('u')
60 ax.box='on'
61 hold off
```

A.2   Functions

```
1 function [ x ] = GenerateMesh(int, N_elem)
2 %GenerateMesh Creates a mesh for 1D problems
3 %   Detailed explanation goes here
4
5 x = linspace(int(1,1),int(1,2),N_elem);
6
7 end
```

```
1 function [ elmat ] = GenerateTopology( N_elem )
2 %GenerateTopology Creates the topology for a 1D problem given mesh 'x'.
3 %   Detailed explanation goes here
4
5 elmat = zeros(N_elem,2);
6
7 for i = 1:N_elem-1
8     elmat(i,1) = i;
9     elmat(i,2) = i + 1;
10 end
11
12 end
```

```
1 function [ S ] = AssembleMatrix( N_elem, lambda, D, mesh, elmat)
2 %AssembleMatrix Assembles matrix S from element matrix S_ek
3 %   Detailed explanation goes here
4
5 S = zeros(N_elem,N_elem);
6
7 for i = 1:N_elem-1
8     Selem = GenerateElementMatrix(i, elmat, D, lambda, mesh);
9     for j = 1:2
10         for k = 1:2
11             S(elmat(i,j), elmat(i,k)) = S(elmat(i,j), elmat(i,k)) + Selem(j,k);
12         end
13     end
14 end
15
16 end
```

```matlab
1  function [ Selem ] = GenerateElementMatrix( k, elmat, D, lambda, mesh)
2  %GenerateElementMatrix Creates element matrix S_ek
3  %   Detailed explanation goes here
4
5  Selem = zeros(2,2);
6
7  i = elmat(k,1);
8  j = elmat(k,2);
9
10 x1 = mesh(i);
11 x2 = mesh(j);
12
13 element_length = abs(x1-x2);
14
15 slope = 1/element_length;
16
17 for m = 1:2
18     for n = 1:2
19         if m == n
20             Selem(m,n) = element_length*((-1)^(abs(m-n))*D*slope^2 + (2)*lambda/6);
21         else
22             Selem(m,n) = element_length*((-1)^(abs(m-n))*D*slope^2 + (1)*lambda/6);
23         end
24     end
25
26 end
27 end
```

```matlab
1  function [ f ] = AssembleVector( N_elem, mesh, elmat )
2  %AssembleVector Assembles vector f from element vector f_ek
3  %   Detailed explanation goes here
4
5  f = zeros(N_elem,1);
6
7  for i = 1:N_elem-1
8      felem = GenerateElementVector(i, elmat, mesh);
9      for j = 1:2
10         f(elmat(i,j)) = f(elmat(i,j)) + felem(j);
11     end
12 end
```

```matlab
1  function [ felem ] = GenerateElementVector( i, elmat, mesh )
2  %GenerateElementVector Creates element vector f_ek
3  %   Detailed explanation goes here
4
5  felem = [0;0];
6
7  k1 = elmat(i,1);
8  k2 = elmat(i,2);
9
10 x1 = mesh(k1);
11 x2 = mesh(k2);
12
13 element_length = abs(x1-x2);
14
15 felem = (element_length/2*arrayfun(@functionBVP,[x1,x2]))';
16
17 end
```

Appendix B

2D-case

B.1   Generate mesh

```matlab
1   clear all
2
3   Geometry = 'squareg';
4
5   DiffCoeff = 1;
6   h_transfer = 1;
7   u_inf = 1;
8
9
10  % Geometry = 'squareg'; % gives square [-1,1] x [-1,1]
11  % Geometry = 'circleg'; % gives unit circle centered at origin
12  % Geometry = 'lshapeg'; % gives L-shape
13
14  [p,e,t] = initmesh(Geometry);
15  [p,e,t] = refinemesh(Geometry,p,e,t); % gives gridrefinement
16  [p,e,t] = refinemesh(Geometry,p,e,t); % gives second gridrefinement
17  %[p,e,t] = refinemesh(Geometry,p,e,t); % gives third gridrefinement
18  pdemesh(p,e,t); % plots the geometry and mesh
19
20  x = p(1,:); y = p(2,:);
21  n = length(p(1,:));
22
23  elmat = t(1:3,:);
24  elmat = elmat';
25  elmatbnd = e(1:2,:);
26  elmatbnd = elmatbnd';
27  % h
28  topology = 3; topologybnd = 2;
```

B.2   Generate element matrix

```matlab
1  clear xc
2  clear yc
3  clear Selem
4
5  for index1 = 1:topology
6      xc(index1) = x(elmat(i,index1));
7      yc(index1) = y(elmat(i,index1));
8  end;
9
10 Delta = det([1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)]);
11 B_mat = [1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)] \  eye(3);
12
13 alpha = B_mat(1,1:3);
14 beta  = B_mat(2,1:3);
15 gamma = B_mat(3,1:3);
16
17 for index1 = 1:topology
18     for index2 = 1:topology
19         if ¬exist('u','var')
20             Selem(index1,index2) =
21             abs(Delta)/2*(k/mu)*(beta(index1)*beta(index2)+gamma(index1)*gamma(index2));
22         else
23             Selem(index1,index2) = abs(Delta)/24;
24         end
25     end;
26 end;
```

B.3 Generate element vector

```
1
2  % Module for element mass matrix for reactive term
3  %
4  % Output: felem  ====== vector of two components
5  %
6  % felem(1), felem(2) to be computed in this routine.
7
8  clear xc
9  clear yc
10 clear felem
11
12 for index1=1:topology
13     xc(index1) = x(elmat(i,index1));
14     yc(index1) = y(elmat(i,index1));
15 end;
16
17 Delta = det([1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)]);
18
19 B_mat = [1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)] \  eye(3);
20
21 alpha = B_mat(1,1:3);
22 beta  = B_mat(2,1:3);
23 gamma = B_mat(3,1:3);
24
25 felem = zeros(1,topology);
26
27 if ¬exist('u','var')  % Only if u is already know can the calculation of the velocity ...
       begin.
28     for N = 1:N_wells
29         for index3 = 1:topology
30             phi_p(index3) = alpha(index3) + beta(index3)*xp(N) + gamma(index3)*yp(N);
31         end
32
33         if (phi_p(1) ≤ 1) && (phi_p(1) ≥ 0) && (phi_p(2) ≤ 1) && (phi_p(2) ≥ 0) && ...
             (phi_p(3) ≤ 1) && (phi_p(3) ≥ 0);
34             for index1 = 1:topology
35                 felem(index1) = felem(index1) + -Qp*phi_p(index1);
36             end
37
38 %           N_Test = N_Test + 1;
39
40 % Components of f are zero except for those elements with a well! So no
41 % other contributions!
42 %       else
43 %           for index1 = 1:topology
44 %           global_index = elmat(N,index1);
45 %           end
46         end
47     end
48 else
49     switch direction
50         case 1 % x direction
51             for index1 = 1:topology
52                 felem(index1) = felem(index1) + ...
                        (k/mu)*(abs(Delta)/6)*beta(index1)*(u(elmat(i,1))+u(elmat(i,2))+u(elmat(i,3)));
53             end
54         case 2 % y direction
55             for index1 = 1:topology
56                 felem(index1) = felem(index1) + ...
                        (k/mu)*(abs(Delta)/6)*gamma(index1)*(u(elmat(i,1))+u(elmat(i,2))+u(elmat(i,3)));
57             end
58     end
59 end
```

B.4 Generate Boundary element matrix

```
1   clear xc
2   clear yc
3   clear BMelem
4
5   for index1=1:topologybnd
6       xc(index1) = x(elmatbnd(i,index1));
7       yc(index1) = y(elmatbnd(i,index1));
8   end;
9
10  lek = sqrt((xc(2)-xc(1))^2 + (yc(2)-yc(1))^2);
11
12  for index1=1:topologybnd
13      if ¬exist('u', 'var')
14          BMelem(index1,index1) = K*lek/2;   % NC used! not HB!!
15      else
16
17          BMelem(index1,index1) = -(k/(mu*K))*lek/6;
18      end
19  end;
```

B.5 Generate boundary element vector:

```
1   clear xc
2   clear yc
3   clear bfelem
4
5   for index1 = 1:topologybnd
6       xc(index1) = x(elmatbnd(i,index1));
7       yc(index1) = y(elmatbnd(i,index1));
8   end;
9
10  lek = sqrt((xc(2)-xc(1))^2+(yc(2)-yc(1))^2);
11
12  if ¬exist('u','var')
13      for index1 = 1:topologybnd
14          bfelem(index1) = K*pH*lek/2*u_inf;   %what is u_inf?
15      end;
16  else
17      for index1 = 1:topologybnd
18          bfelem(index1) = ((k*pH)/mu)*lek/2*u_inf;   %what is u_inf?
19  %          bfelem(index1) = -(k/mu)*lek/6*u(elmat(i,ind1));
20      end
21  end
```

B.6 Buildmatrices and vectors

```
1   % This routine constructs the large matrices and vector.
2   % The element matrices and vectors are also dealt with.
3   % First the internal element contributions
4   % First Initialisation of large discretisation matrix, right-hand side vector
5
6   % Treatment of the internal (triangular) elements
7
8   if ¬exist('u', 'var')
9
10      S       = sparse(n,n); % stiffness matrix
11      f       = zeros(n,1); % right-hand side vector
12
13      for i = 1:length(elmat(:,1)) % for all internal elements
14          GenerateElementMatrix; % Selem
```

```matlab
15              for ind1 = 1:topology
16                  for ind2 = 1:topology
17                      S(elmat(i,ind1),elmat(i,ind2))  = S(elmat(i,ind1),elmat(i,ind2)) + ...
                            Selem(ind1,ind2);
18                  end;
19              end;
20
21              GenerateElementVector; % felem
22              for ind1 = 1:topology
23                  f(elmat(i,ind1)) = f(elmat(i,ind1)) + felem(ind1);
24              end;
25          end;
26
27  % Next the boundary contributions
28
29          for i = 1:length(elmatbnd(:,1)); % for all boundary elements extension of mass ...
                matrix M and element vector f
30          GenerateBoundaryElementMatrix; % BMelem
31              for ind1 = 1:topologybnd
32                  for ind2 = 1:topologybnd
33                      S(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                            S(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
34                  end;
35              end;
36              GenerateBoundaryElementVector; % bfelem
37              for ind1 = 1:topologybnd
38                  f(elmatbnd(i,ind1)) = f(elmatbnd(i,ind1)) + bfelem(ind1);
39              end;
40          end;
41
42          else
43
44          Sx      = sparse(n,n); % stiffness matrix
45
46          fx      = zeros(n,1); % right-hand side vector
47
48          left_nodes = find(p(1,:) == -1);
49          top_nodes = find(p(2,:) == 1);
50          right_nodes = find(p(1,:) == 1);
51          bottom_nodes = find(p(2,:) == -1);
52
53          bnd1_nodes = ismember(elmatbnd,left_nodes);
54          bnd1 = find(bnd1_nodes(:,1) == 1 & bnd1_nodes(:,2) == 1);
55
56          bnd2_nodes = ismember(elmatbnd,top_nodes);
57          bnd2 = find(bnd2_nodes(:,1) == 1 & bnd2_nodes(:,2) == 1);
58
59          bnd3_nodes = ismember(elmatbnd,right_nodes);
60          bnd3 = find(bnd3_nodes(:,1) == 1 & bnd3_nodes(:,2) == 1);
61
62          bnd4_nodes = ismember(elmatbnd,bottom_nodes);
63          bnd4 = find(bnd4_nodes(:,1) == 1 & bnd4_nodes(:,2) == 1);
64
65
66          direction = 1;
67
68          for i = 1:length(elmat(:,1)) % for all internal elements
69              GenerateElementMatrix; % Selem
70              for ind1 = 1:topology
71                  for ind2 = 1:topology
72                      if elmat(i,ind1) == elmat(i,ind2)
73                          Sx(elmat(i,ind1),elmat(i,ind2)) = Sx(elmat(i,ind1),elmat(i,ind2)) ...
                                + 2*Selem(ind1,ind2);
74                      else
75                          Sx(elmat(i,ind1),elmat(i,ind2)) = Sx(elmat(i,ind1),elmat(i,ind2)) ...
                                + Selem(ind1,ind2);
76                      end
77                  end;
78              end;
79              GenerateElementVector; % felem
80              for ind1 = 1:topology
81                  fx(elmat(i,ind1)) = fx(elmat(i,ind1)) + felem(ind1);
82              end;
```

31

```
 83        end;
 84
 85    % Next the boundary contributions
 86
 87
 88
 89        for j = 1:length(bnd1); % left boundary
 90            i = bnd1(j);
 91            GenerateBoundaryElementMatrix; % BMelem
 92            for ind1 = 1:topologybnd
 93                for ind2 = 1:topologybnd
 94                    if elmatbnd(i,ind1) == elmatbnd(i,ind2)
 95                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
 96                            Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
 96                    else
 97                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
 98                            Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
 98                    end;
 99                end
100            end;
101            GenerateBoundaryElementVector; % bfelem
102            for ind1 = 1:topologybnd
103                fx(elmatbnd(i,ind1)) = fx(elmatbnd(i,ind1)) + bfelem(ind1);
104            end;
105        end;
106
107        for j = 1:length(bnd3); % right boundary
108            i = bnd3(j);
109            GenerateBoundaryElementMatrix; % BMelem
110            for ind1 = 1:topologybnd
111                for ind2 = 1:topologybnd
112                    if elmatbnd(i,ind1) == elmatbnd(i,ind2)
113                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
114                            Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
114                    else
115                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
116                            Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
116                    end;
117                end
118            end;
119            GenerateBoundaryElementVector; % bfelem
120            for ind1 = 1:topologybnd
121                fx(elmatbnd(i,ind1)) = fx(elmatbnd(i,ind1)) - bfelem(ind1);
122            end;
123        end;
124
125        direction = 2;
126
127        Sy      = sparse(n,n); % stiffness matrix
128
129        fy      = zeros(n,1); % right-hand side vector
130
131        for i = 1:length(elmat(:,1)) % for all internal elements
132            GenerateElementMatrix; % Selem
133            for ind1 = 1:topology
134                for ind2 = 1:topology
135                    if elmat(i,ind1) == elmat(i,ind2)
136                        Sy(elmat(i,ind1),elmat(i,ind2)) = Sy(elmat(i,ind1),elmat(i,ind2)) ...
137                            + 2*Selem(ind1,ind2);
137                    else
138                        Sy(elmat(i,ind1),elmat(i,ind2)) = Sy(elmat(i,ind1),elmat(i,ind2)) ...
139                            + Selem(ind1,ind2);
139                    end
140                end;
141            end;
142            GenerateElementVector; % felem
143            for ind1 = 1:topology
144                fy(elmat(i,ind1)) = fy(elmat(i,ind1)) + felem(ind1);
145            end;
146        end;
147
148    % Next the boundary contributions
149
```

```
150
151
152      for j = 1:length(bnd2); % left boundary
153          i = bnd2(j);
154          GenerateBoundaryElementMatrix; % BMelem
155          for ind1 = 1:topologybnd
156              for ind2 = 1:topologybnd
157                  if elmatbnd(i,ind1) == elmatbnd(i,ind2)
158                      Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                              Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
159                  else
160                      Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                              Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
161                  end;
162              end
163          end;
164          GenerateBoundaryElementVector; % bfelem
165          for ind1 = 1:topologybnd
166              fy(elmatbnd(i,ind1)) = fy(elmatbnd(i,ind1)) - bfelem(ind1);
167          end;
168      end;
169
170      for j = 1:length(bnd4); % right boundary
171          i = bnd4(j);
172          GenerateBoundaryElementMatrix; % BMelem
173          for ind1 = 1:topologybnd
174              for ind2 = 1:topologybnd
175                  if elmatbnd(i,ind1) == elmatbnd(i,ind2)
176                      Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                              Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
177                  else
178                      Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                              Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
179                  end;
180              end
181          end;
182          GenerateBoundaryElementVector; % bfelem
183          for ind1 = 1:topologybnd
184              fy(elmatbnd(i,ind1)) = fy(elmatbnd(i,ind1)) + bfelem(ind1);
185          end;
186      end;
187  end
```

B.7   Compute u and v_x/v_y

```
1  % Construction of linear problem
2
3  BuildMatricesandVectors;
4
5  % Solution of linear problem
6
7  u = S \ f;
8
9  BuildMatricesandVectors;
10
11 vx = Sx \ fx;
12 vy = Sy \ fy;
```

B.8 Full script

```matlab
1  close all
2  clear all
3
4  %% 2D Assignment
5  % Lab Assignment 7
6
7  %% Create Mesh
8  WI4243Mesh
9
10  %% Parameters
11
12  Qp = 50;              % [m^2/s]
13  k = 10^-7;            % [m^2]
14  mu = 1.002*10^-3;     % [Pa*s]
15  K = 10000;                % [m/s]
16  pH = 10^6;            % [Pa]
17  N_wells = 6;          % number of wells
18
19  epsilon1 = 0.03;
20  N_Test = 0;
21  %% Coordinates of wells
22
23  for i = 1:N_wells-1;
24  xp(i) = 0.6*cos((2*pi)*(i-1)/(N_wells-1));
25  yp(i) = 0.6*sin((2*pi)*(i-1)/(N_wells-1));
26  end
27
28  xp(N_wells) = 0;
29  yp(N_wells) = 0;
30  clear i;
31
32
33  %% Compute Problem
34  WI4243Comp
35
36  %% Post
37  hold on
38
39  figure(2);
40  ax.BoxStyle = 'full';
41  hold off
42  trisurf(elmat,x,y,u)
43  xlabel('x'); ylabel('y'); zlabel('Pressure [Pa]');
44  title(['\bf\fontsize{16}3D Surface plot  \fontsize{10} K = ' num2str(K,'%10.2e\n') ' ...
          [m/s]  \it Minimal pressure = ' num2str(min(u), '%10.2e\n') ' [Pa]']);
45
46  lgd = legend();
47  % title(lgd,['3D Surface plot, K = ' num2str(K) '\it Minimal pressure = ' ...
          num2str(Pressure_minimum)]);
48
49  % title( {'Title';'subtitle'} )
50
51  figure(3);
52  trisurf(elmat,x,y,u);
53  xlabel('x'); ylabel('y');
54  title(['\bf\fontsize{16}Contour plot, \fontsize{10} K = ',num2str(K,'%10.2e\n')]);
55  view(2); shading interp; colormap jet; colorbar; set(gcf,'renderer','zbuffer')
56  h = colorbar; ylabel(h, 'Pressure [Pa]');
57
58
59  figure(4); quiver(x,y,vx',vy'); axis([-1 1 -1 1]);
60  xlabel('x'); ylabel('y');
61  title(['\bf\fontsize{16}Velocity field, \fontsize{10} K = ' num2str(K,'%10.2e\n') ' ...
          [m/s]']);
62  %% Velocity part
63  hold off
```