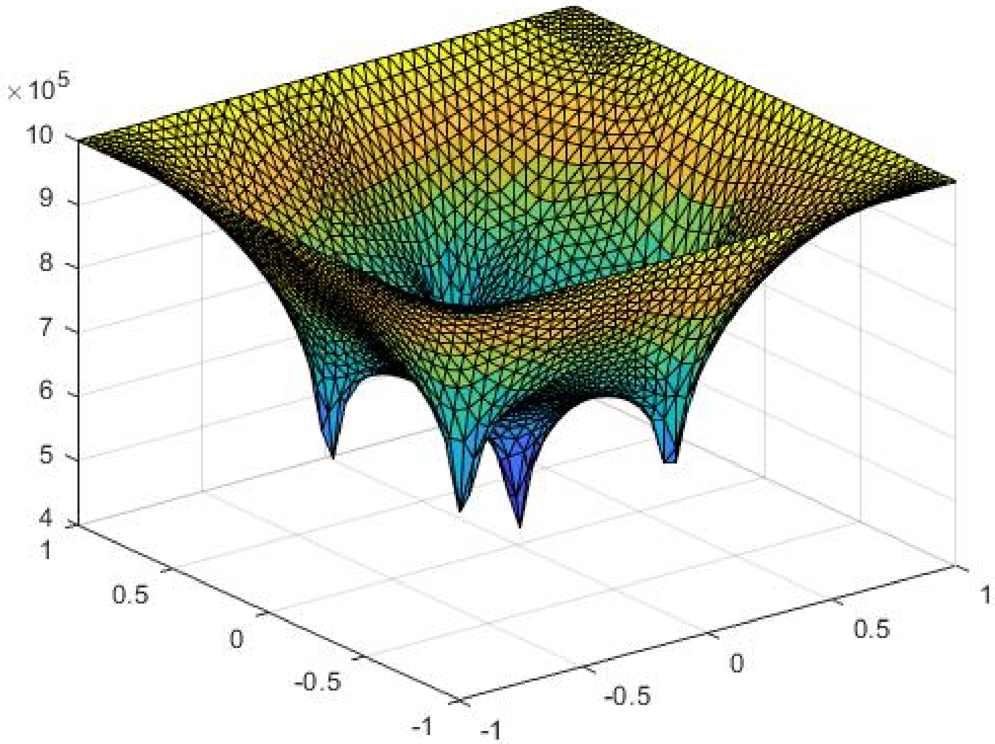
Finite Elements

Rick Koster
Ruben Termaat
February 28, 2018



Preface

This report was written in order to better understand and demonstrate how one can apply 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

Preface **2**

1 1D-case **5**
1.1 Boundary value problem 1D . . . 5
1.2 Element matrix . . . 6
1.3 Element vector . . . 6
1.4 Boundary value problem 1D MATLAB routine . . . 6
1.4.1 mesh and elmat code . . . 6
1.4.2 Element matrix . . . 7
1.4.3 Assemble matrix S . . . 7
1.4.4 Element vector MATLAB routine . . . 8
1.4.5 Computing S and f . . . 8
1.5 Main program . . . 9
1.6 Solution for u . . . 10
1.7 Experiment . . . 11

2 2D-case **13**
2.1 Boundary value problem 2D . . . 14
2.2 Element matrix and element vector . . . 15
2.3 Boundary matrix and boundary vector . . . 15
2.4 Wells within an internal element . . . 15
2.5 Generating MATLAB code . . . 16
2.6 Velocities . . . 16
2.7 Varying constant K . . . 21

3 Conclusion **23**

A 1D-case **24**
A.1 Script . . . 24
A.2 Functions . . . 25

B 2D-case **27**
B.1 Generate mesh . . . 27
B.2 Generate element matrix . . . 28
B.3 Generate element vector . . . 29
B.4 Generate Boundary element matrix . . . 30
B.5 Generate boundary element vector: . . . 30
B.6 Buildmatrices and vectors . . . 30
B.7 Compute u and v_x/v_y . . . 33
B.8 Full script . . . 34

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).

```
function [ x ] = GenerateMesh(int, N_elem)
%GenerateMesh Creates a mesh for 1D problems

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

```
function [ elmat ] = GenerateTopology( N_elem )
%GenerateTopology Creates the topology for a 1D problem given mesh 'x'.

elmat = zeros(N_elem,2);
elmat(i,1) = i;
elmat(i,2) = i + 1;
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}.

```
function [ Selem ] = GenerateElementMatrix( k, elmat, D, lambda, mesh)
%GenerateElementMatrix Creates element matrix S_ek

Selem = zeros(2,2);

i = elmat(k,1);
j = elmat(k,2);

x1 = mesh(i);
x2 = mesh(j);

element_length = abs(x1-x2);

slope = 1/element_length;

for m = 1:2
    for n = 1:2
        if m == n
            Selem(m,n) = element_length*((-1)^(abs(m-n))*D*slope^2
            + (2)*lambda/6);
        else
            Selem(m,n) = element_length*((-1)^(abs(m-n))*D*slope^2
            + (1)*lambda/6);
        end
    end
end
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:

```
function [ S ] = AssembleMatrix( N_elem, int, lambda, D)
% global N_elem

elmat = GenerateTopology(N_elem);

S = zeros(N_elem,N_elem);

for i = 1:N_elem-1
    Selem = GenerateElementMatrix(i, elmat, int, N_elem, D, lambda);
    for j = 1:2
        for k = 1:2
            S(elmat(i,j), elmat(i,k)) =
            S(elmat(i,j), elmat(i,k)) + Selem(j,k);
        end
    end
end
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.

```
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.

```
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 $i \epsilon \{1, ..., n-1\}$ has to be calculated.

```
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.

```
function [ u ] = SolveBVP(  N_elem, int, lambda, D )

S = AssembleMatrix( N_elem, int, lambda, D);
f = AssembleVector( N_elem, int, lambda, D);

%% Calculate u
x = linspace(int(1),int(2),N_elem);

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

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

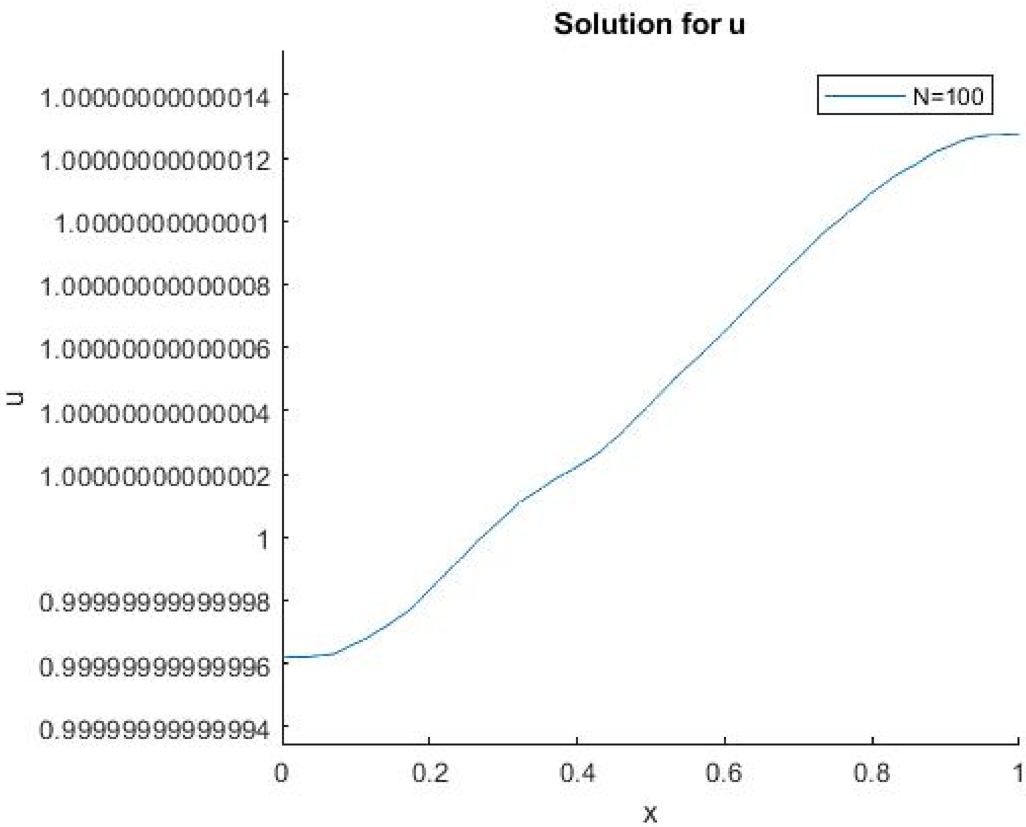


Figure 1.1: showing the calculated u versus x, with N = 100,f(x)=1

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).

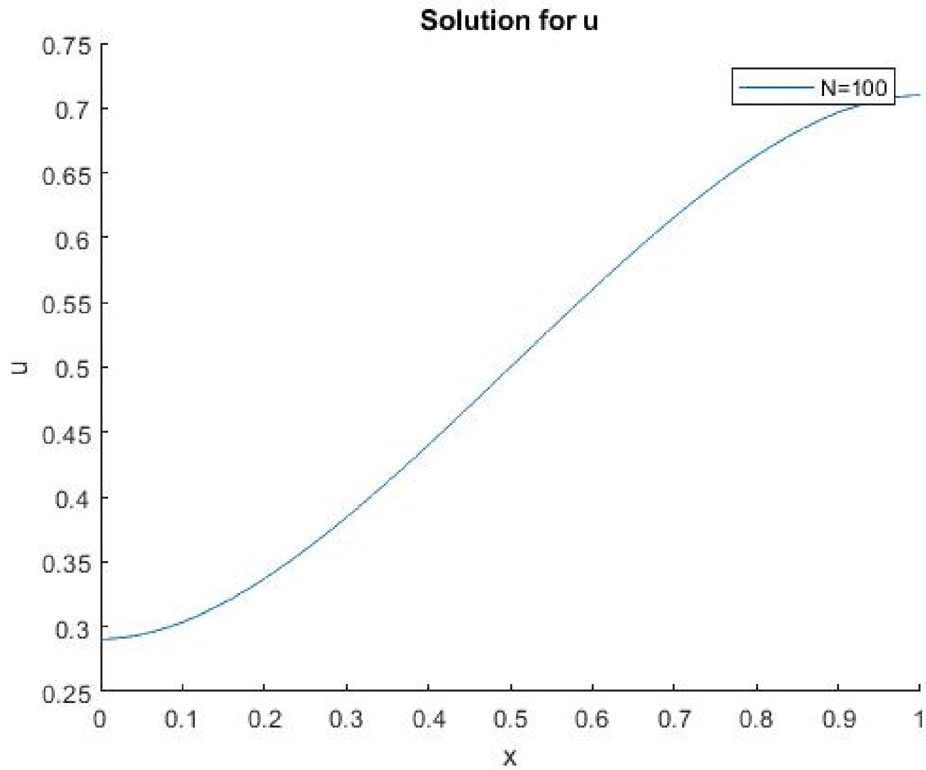


Figure 1.2: showing the calculated u versus x, with $N = 100$, $f(x) = x$

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 ($n = 10, 20, 30, 40, 80, 160$).

```
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
```

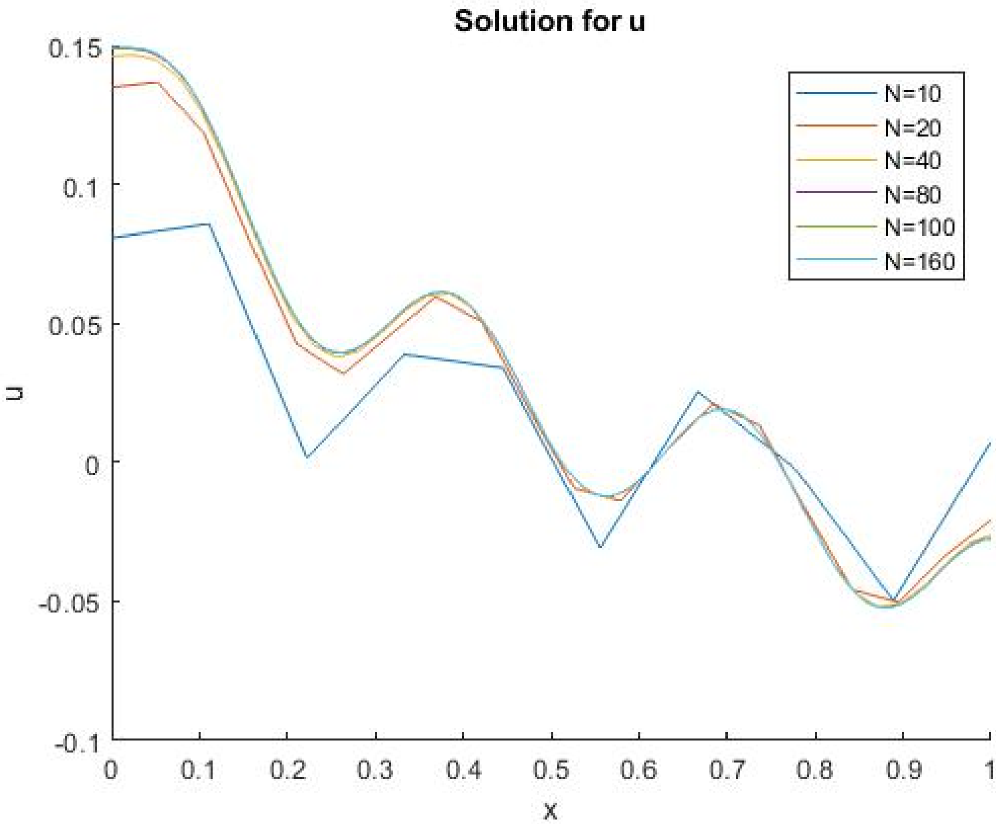


Figure 1.3: showing the calculated u versus x, with $N = [10, 20, 40, 80, 100, 160], f(x) = sin(x)$.

Figure(1.3) 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 and the created MATLAB code to solve a 2-dimensional BVP. To do this, a real life problem is now presented. 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).

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

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:

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

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:

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

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

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

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):

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

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

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

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

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

```
for index1=1:topology
    xc(index1) = x(elmat(i,index1));
    yc(index1) = y(elmat(i,index1));
end;

Delta = det([1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)]);

B_mat = [1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)] \  eye(3);

alpha = B_mat(1,1:3);
beta  = B_mat(2,1:3);
gamma = B_mat(3,1:3);

felem = zeros(1,topology);

if ¬exist('u','var')  % Only if u is already know can the calculation of the velocity ...
    begin.
    for N = 1:N_wells
        for index3 = 1:topology
            phi_p(index3) = alpha(index3) + beta(index3)*xp(N) + gamma(index3)*yp(N);
        end
        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);
            for index1 = 1:topology
                felem(index1) = felem(index1) + -Qp*phi_p(index1);
            end

        end
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).

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

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

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

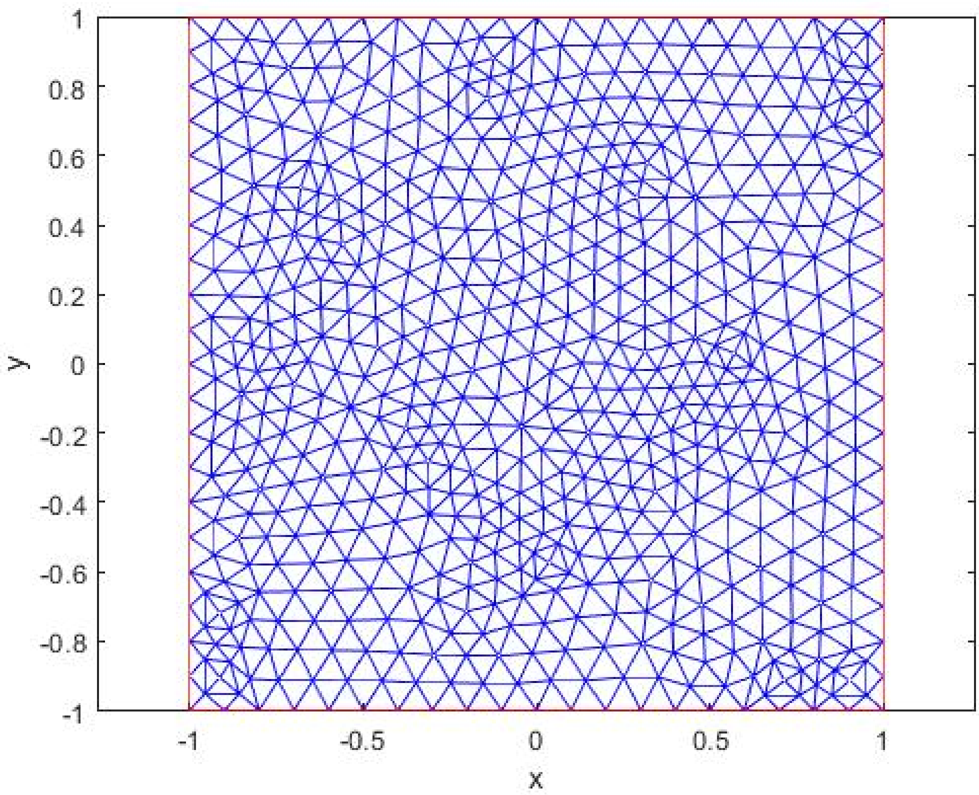


Figure 2.1: Triangle element mesh over the square reservoir domain $\Omega = (-1;1) \times (-1;1)$.

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.

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

Partial integration is applied on the right side term.

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

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.

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

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.

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

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

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

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

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: these are the wells. Looking at the velocity field, it can be seen that the velocity is highest around the wells 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 the highest pressure. The last figure shows a surface plot in accordance to the velocity field and pressure contour plot. Where the pressure difference is highest, one would expect the veločty to be highest. Velocity depends on pressure by the following relation: $\vec{v} = -\frac{k}{\mu}\nabla p$. At the peaks the gradient of p is largest and results in a higher velocityS.

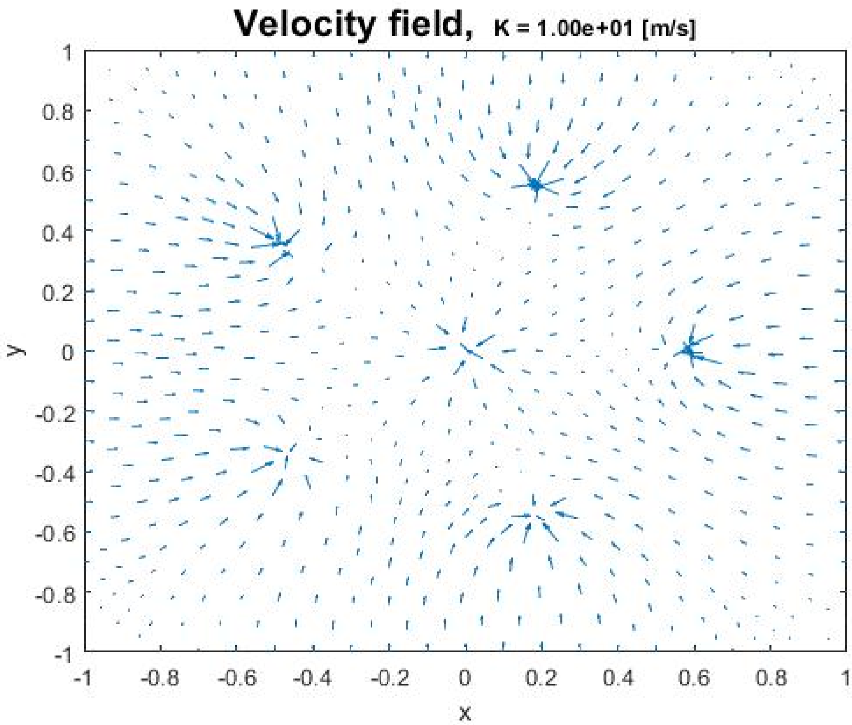


Figure 2.2: Arrow velocity plot, indicating the direction and velocity of the water(longer arrows indicate higher velocity).

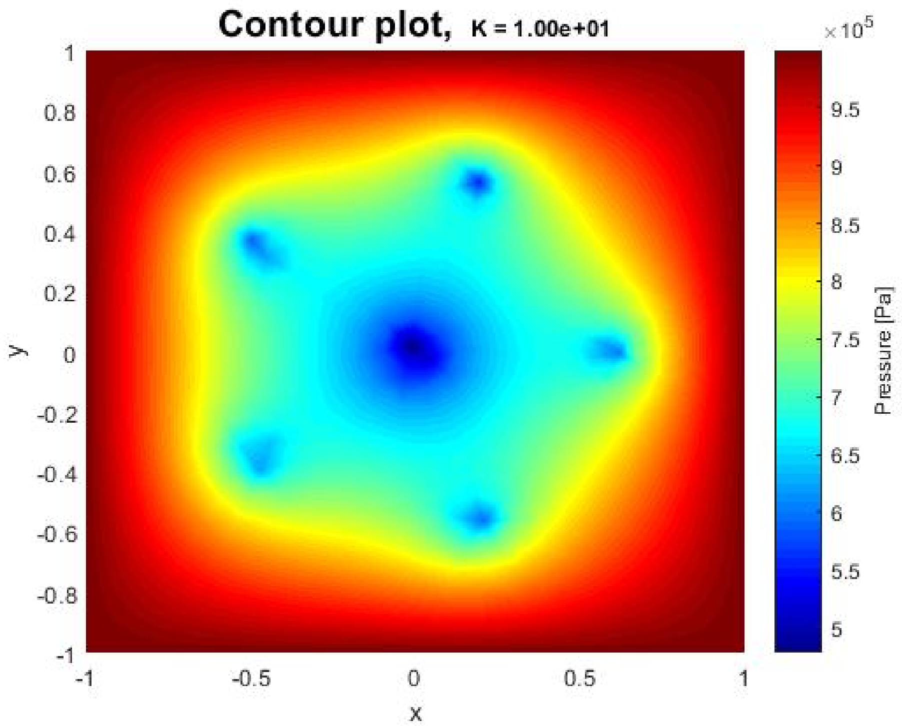


Figure 2.3: Contour plot of the velocity in the square reservoir on domain $\Omega = (-1, 1) \times (-1, 1)$, showing six areas(the wells) where a drop of about two times the pressure can be observed, compared to the boundary pressure.

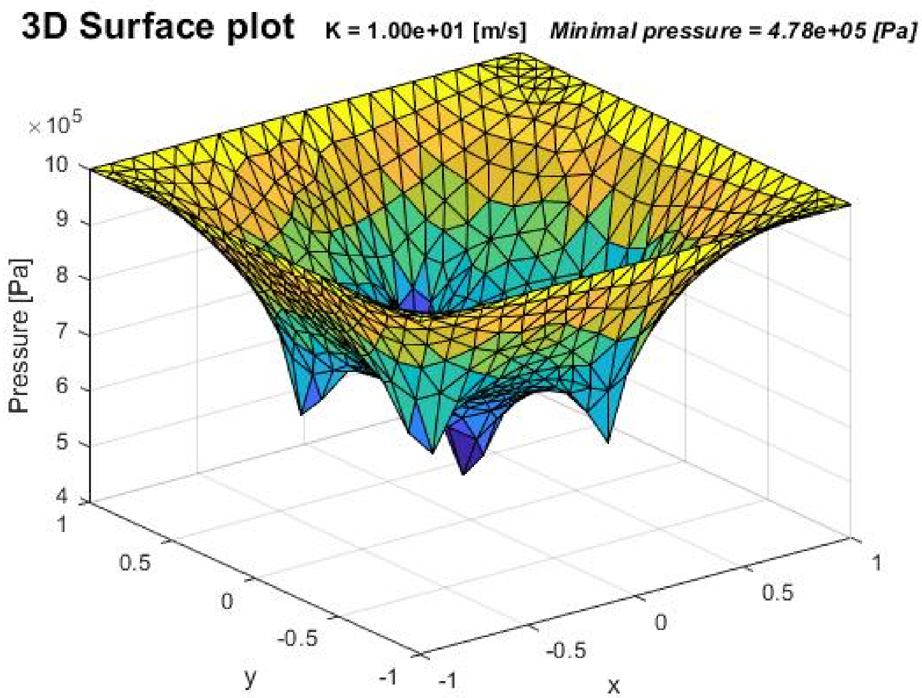
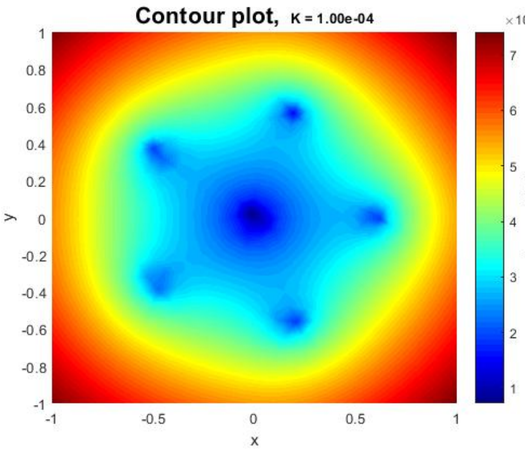


Figure 2.4: 3D surface plot for $K = 10$. The lowest peaks indicate where the pressure is at a minimum in the square water reservoir.

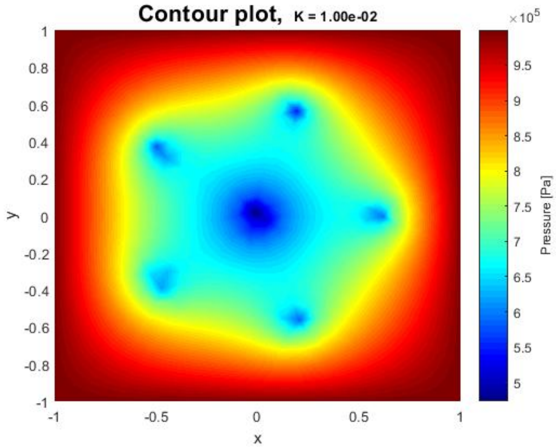
2.7 Varying constant K

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. These plots can be found on the next page(figure 2.5). Looking at the plots it can be seen that the plots become increasinly darker as the K factor increases. The pressure around the wells, where the pressure is at its minimum, also increases. When the K factor is above $K = 1$ the contour plots become similar. From an engineering point of view one could state that this means that a certain treshold ($K > x$) has to be achieved, with x being the minimum value at which the pressure profile shows enough pressure at the wells. If this treshold is not met the water will dissipate horizontaly in the reservoir instead of being pushed out through the well. The weight of the water due to the effect of gravity has to be overcome by the pressure of the reservoir.

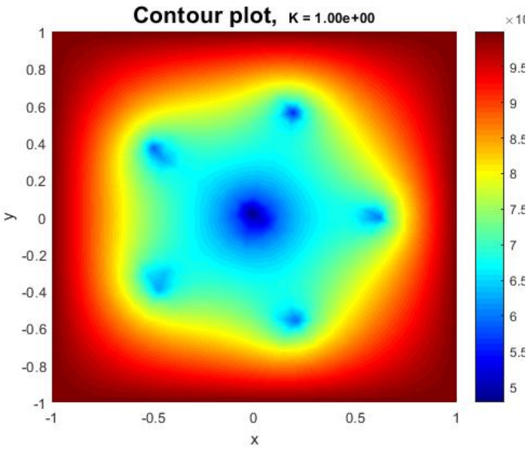
The final step is to determine what happens when $K = 0$ and why? When looking at our previous plots, when varying K from 0.00001 to 10000 it can be derived that the pressure drops the lower the value for K is. This would mean that when $K = 0$, the transfer coefficient of the water between the boundary of the square reservoir and its surroundings is zero. There will be no flow from the square reservoir into the surrounding through the wells. For engineering purpose designing a square reservoir such that it has a K value and therefore no flow, is an unwanted feature. The purpose of calculating the pressure is to make sure there is enough pressure present in the reservoir for water to flow up the wells.



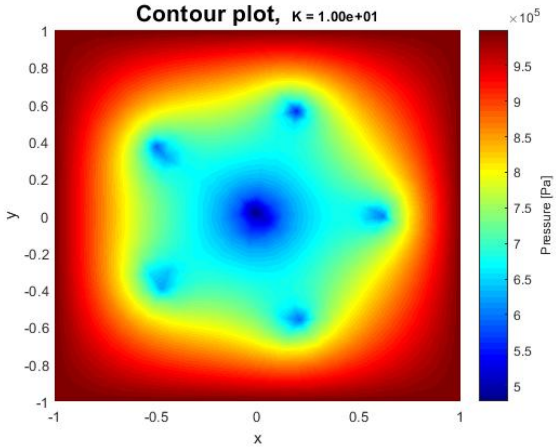
(a) $K = 0.0001$



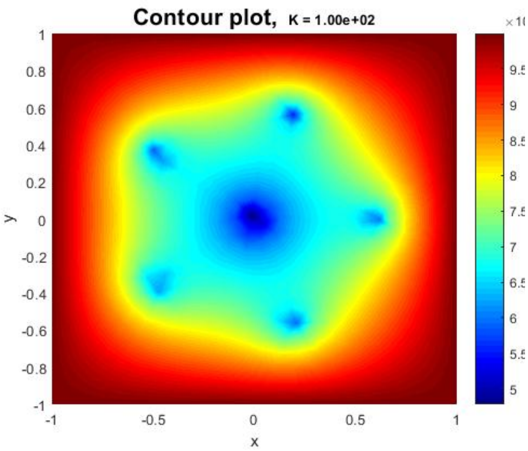
(b) $K = 0.001$



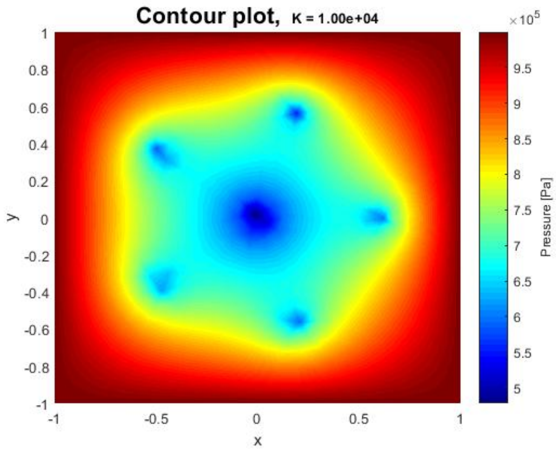
(c) A subfigure



(d) $K = 10$



(e) $K = 100$



(f) $K = 1000$

Figure 2.5: A figure with two subfigures

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 1D-case 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 graph. 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 coefficient 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

```
clear all
close all

%%Finite Element 1D
%% Parameters

N_elem = 100; %Number of elements
int = [0,1]; %Interval
lambda = 1;
D = .1;

%% Mesh & Topology

mesh = GenerateMesh(int,N_elem);
elmat = GenerateTopology(N_elem); %1D topology!!

%% Assemble Matrix & Vector

S = AssembleMatrix( N_elem, lambda, D, mesh, elmat);
f = AssembleVector( N_elem, mesh, elmat);

%% Calculate u
x = linspace(int(1),int(2),N_elem);

u = S\f;

hold on
plot(x,u);
legend('N=100')
title('Solution for u')
xlabel('x')
ylabel('u')
ax.box='on'
hold off


% For this part change the function in functionBVP.m to 'f = sin(20*x)'

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);


end

legend('N=10','N=20','N=40','N= 80','N=100','N=160')
title('Solution for u')
xlabel('x')
ylabel('u')
ax.box='on'
hold off
```

A.2 Functions

```
function [ x ] = GenerateMesh(int, N_elem)
%GenerateMesh Creates a mesh for 1D problems
%   Detailed explanation goes here

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

end
```

```
function [ elmat ] = GenerateTopology( N_elem )
%GenerateTopology Creates the topology for a 1D problem given mesh 'x'.
%   Detailed explanation goes here

elmat = zeros(N_elem,2);

for i = 1:N_elem-1
    elmat(i,1) = i;
    elmat(i,2) = i + 1;
end

end
```

```
function [ S ] = AssembleMatrix( N_elem, lambda, D, mesh, elmat)
%AssembleMatrix Assembles matrix S from element matrix S_ek
%   Detailed explanation goes here

S = zeros(N_elem,N_elem);

for i = 1:N_elem-1
    Selem = GenerateElementMatrix(i, elmat, D, lambda, mesh);
    for j = 1:2
        for k = 1:2
            S(elmat(i,j), elmat(i,k)) = S(elmat(i,j), elmat(i,k)) + Selem(j,k);
        end
    end
end

end
```

```matlab
function [ Selem ] = GenerateElementMatrix( k, elmat, D, lambda, mesh)
%GenerateElementMatrix Creates element matrix S_ek
%   Detailed explanation goes here

Selem = zeros(2,2);

i = elmat(k,1);
j = elmat(k,2);

x1 = mesh(i);
x2 = mesh(j);

element_length = abs(x1-x2);

slope = 1/element_length;

for m = 1:2
    for n = 1:2
        if m == n
            Selem(m,n)  = element_length*((-1)^(abs(m-n))*D*slope^2 + (2)*lambda/6);
            else
            Selem(m,n)  = element_length*((-1)^(abs(m-n))*D*slope^2 + (1)*lambda/6);
        end
    end

end
end
```

```matlab
function [ f ] = AssembleVector( N_elem, mesh, elmat )
%AssembleVector Assembles vector f from element vector f_ek
%   Detailed explanation goes here

f = zeros(N_elem,1);

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

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

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
```

Appendix B

2D-case

B.1 Generate mesh

```
clear all

Geometry = 'squareg';

DiffCoeff = 1;
h_transfer = 1;
u_inf = 1;


% Geometry = 'squareg'; % gives square [-1,1] x [-1,1]
% Geometry = 'circleg'; % gives unit circle centered at origin
% Geometry = 'lshapeg'; % gives L-shape

[p,e,t] = initmesh(Geometry);
[p,e,t] = refinemesh(Geometry,p,e,t); % gives gridrefinement
[p,e,t] = refinemesh(Geometry,p,e,t); % gives second gridrefinement
%[p,e,t] = refinemesh(Geometry,p,e,t); % gives third gridrefinement
pdemesh(p,e,t); % plots the geometry and mesh

x = p(1,:); y = p(2,:);
n = length(p(1,:));

elmat = t(1:3,:);
elmat = elmat';
elmatbnd = e(1:2,:);
elmatbnd = elmatbnd';
% h
topology = 3; topologybnd = 2;
```

B.2 Generate element matrix

```
clear xc
clear yc
clear Selem

for index1 = 1:topology
    xc(index1) = x(elmat(i,index1));
    yc(index1) = y(elmat(i,index1));
end;

Delta = det([1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)]);
B_mat = [1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)] \ eye(3);

alpha = B_mat(1,1:3);
beta  = B_mat(2,1:3);
gamma = B_mat(3,1:3);

for index1 = 1:topology
    for index2 = 1:topology
        if ¬exist('u','var')
            Selem(index1,index2) =
            abs(Delta)/2*(k/mu)*(beta(index1)*beta(index2)+gamma(index1)*gamma(index2));
        else
            Selem(index1,index2) = abs(Delta)/24;
        end
    end;
end;
```

B.3 Generate element vector

```

% Module for element mass matrix for reactive term
%
% Output: felem  ====== vector of two components
%
% felem(1), felem(2) to be computed in this routine.

clear xc
clear yc
clear felem

for index1=1:topology
    xc(index1) = x(elmat(i,index1));
    yc(index1) = y(elmat(i,index1));
end;

Delta = det([1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)]);

B_mat = [1 xc(1) yc(1);1 xc(2) yc(2);1 xc(3) yc(3)] \  eye(3);

alpha = B_mat(1,1:3);
beta  = B_mat(2,1:3);
gamma = B_mat(3,1:3);

felem = zeros(1,topology);

if ¬exist('u','var')  % Only if u is already know can the calculation of the velocity ...
    begin.
    for N = 1:N_wells
        for index3 = 1:topology
            phi_p(index3) = alpha(index3) + beta(index3)⋆xp(N) + gamma(index3)⋆yp(N);
        end

        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);
            for index1 = 1:topology
                felem(index1) = felem(index1) + -Qp⋆phi_p(index1);
            end

%          N_Test = N_Test + 1;

% Components of f are zero except for those elements with a well! So no
% other contributions!
%      else
%          for index1 = 1:topology
%          global_index = elmat(N,index1);
%          end
        end
    end
else
    switch direction
        case 1 % x direction
            for index1 = 1:topology
                felem(index1) = felem(index1) + ...
                    (k/mu)⋆(abs(Delta)/6)⋆beta(index1)⋆(u(elmat(i,1))+u(elmat(i,2))+u(elmat(i,3)));
            end
        case 2 % y direction
            for index1 = 1:topology
                felem(index1) = felem(index1) + ...
                    (k/mu)⋆(abs(Delta)/6)⋆gamma(index1)⋆(u(elmat(i,1))+u(elmat(i,2))+u(elmat(i,3)));
            end
    end
end
```

B.4 Generate Boundary element matrix

```
clear xc
clear yc
clear BMelem

for index1=1:topologybnd
    xc(index1) = x(elmatbnd(i,index1));
    yc(index1) = y(elmatbnd(i,index1));
end;

lek = sqrt((xc(2)-xc(1))^2 + (yc(2)-yc(1))^2);

for index1=1:topologybnd
    if ¬exist('u', 'var')
        BMelem(index1,index1) = K*lek/2;  % NC used! not HB!!
    else

        BMelem(index1,index1) = -(k/(mu*K))*lek/6;
    end
end;
```

B.5 Generate boundary element vector:

```
clear xc
clear yc
clear bfelem

for index1 = 1:topologybnd
    xc(index1) = x(elmatbnd(i,index1));
    yc(index1) = y(elmatbnd(i,index1));
end;

lek = sqrt((xc(2)-xc(1))^2+(yc(2)-yc(1))^2);

if ¬exist('u','var')
    for index1 = 1:topologybnd
        bfelem(index1) = K*pH*lek/2*u_inf;   %what is u_inf?
    end;
else
    for index1 = 1:topologybnd
        bfelem(index1) = ((k*pH)/mu)*lek/2*u_inf;   %what is u_inf?
%         bfelem(index1) = -(k/mu)*lek/6*u(elmat(i,ind1));
    end
end
```

B.6 Buildmatrices and vectors

```
% This routine constructs the large matrices and vector.
% The element matrices and vectors are also dealt with.
% First the internal element contributions
% First Initialisation of large discretisation matrix, right-hand side vector

% Treatment of the internal (triangular) elements

if ¬exist('u', 'var')

    S       = sparse(n,n); % stiffness matrix
    f       = zeros(n,1); % right-hand side vector

    for i = 1:length(elmat(:,1)) % for all internal elements
        GenerateElementMatrix; % Selem
```

```
        for ind1 = 1:topology
            for ind2 = 1:topology
                S(elmat(i,ind1),elmat(i,ind2))  = S(elmat(i,ind1),elmat(i,ind2)) + ...
                    Selem(ind1,ind2);
            end;
        end;

        GenerateElementVector; % felem
        for ind1 = 1:topology
            f(elmat(i,ind1)) = f(elmat(i,ind1)) + felem(ind1);
        end;
    end;

% Next the boundary contributions

    for i = 1:length(elmatbnd(:,1)); % for all boundary elements extension of mass ...
        matrix M and element vector f
    GenerateBoundaryElementMatrix; % BMelem
        for ind1 = 1:topologybnd
            for ind2 = 1:topologybnd
                S(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                    S(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
            end;
        end;
        GenerateBoundaryElementVector; % bfelem
        for ind1 = 1:topologybnd
            f(elmatbnd(i,ind1)) = f(elmatbnd(i,ind1)) + bfelem(ind1);
        end;
    end;

    else

    Sx      = sparse(n,n); % stiffness matrix

    fx      = zeros(n,1); % right-hand side vector

    left_nodes = find(p(1,:) == -1);
    top_nodes = find(p(2,:) == 1);
    right_nodes = find(p(1,:) == 1);
    bottom_nodes = find(p(2,:) == -1);

    bnd1_nodes = ismember(elmatbnd,left_nodes);
    bnd1 = find(bnd1_nodes(:,1) == 1 & bnd1_nodes(:,2) == 1);

    bnd2_nodes = ismember(elmatbnd,top_nodes);
    bnd2 = find(bnd2_nodes(:,1) == 1 & bnd2_nodes(:,2) == 1);

    bnd3_nodes = ismember(elmatbnd,right_nodes);
    bnd3 = find(bnd3_nodes(:,1) == 1 & bnd3_nodes(:,2) == 1);

    bnd4_nodes = ismember(elmatbnd,bottom_nodes);
    bnd4 = find(bnd4_nodes(:,1) == 1 & bnd4_nodes(:,2) == 1);


    direction = 1;

    for i = 1:length(elmat(:,1)) % for all internal elements
        GenerateElementMatrix; % Selem
        for ind1 = 1:topology
            for ind2 = 1:topology
                if elmat(i,ind1) == elmat(i,ind2)
                    Sx(elmat(i,ind1),elmat(i,ind2)) = Sx(elmat(i,ind1),elmat(i,ind2)) ...
                        + 2*Selem(ind1,ind2);
                else
                    Sx(elmat(i,ind1),elmat(i,ind2)) = Sx(elmat(i,ind1),elmat(i,ind2)) ...
                        + Selem(ind1,ind2);
                end
            end;
        end;
        GenerateElementVector; % felem
        for ind1 = 1:topology
            fx(elmat(i,ind1)) = fx(elmat(i,ind1)) + felem(ind1);
        end;
```

```
    end;

% Next the boundary contributions



    for j = 1:length(bnd1); % left boundary
        i = bnd1(j);
        GenerateBoundaryElementMatrix; % BMelem
        for ind1 = 1:topologybnd
            for ind2 = 1:topologybnd
                if elmatbnd(i,ind1) == elmatbnd(i,ind2)
                    Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
                else
                    Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
                end;
            end
        end;
        GenerateBoundaryElementVector; % bfelem
        for ind1 = 1:topologybnd
            fx(elmatbnd(i,ind1)) = fx(elmatbnd(i,ind1)) + bfelem(ind1);
        end;
    end;

    for j = 1:length(bnd3); % right boundary
        i = bnd3(j);
        GenerateBoundaryElementMatrix; % BMelem
        for ind1 = 1:topologybnd
            for ind2 = 1:topologybnd
                if elmatbnd(i,ind1) == elmatbnd(i,ind2)
                    Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
                else
                    Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sx(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
                end;
            end
        end;
        GenerateBoundaryElementVector; % bfelem
        for ind1 = 1:topologybnd
            fx(elmatbnd(i,ind1)) = fx(elmatbnd(i,ind1)) - bfelem(ind1);
        end;
    end;

    direction = 2;

    Sy      = sparse(n,n); % stiffness matrix

    fy      = zeros(n,1); % right-hand side vector

    for i = 1:length(elmat(:,1)) % for all internal elements
        GenerateElementMatrix; % Selem
        for ind1 = 1:topology
            for ind2 = 1:topology
                if elmat(i,ind1) == elmat(i,ind2)
                    Sy(elmat(i,ind1),elmat(i,ind2)) = Sy(elmat(i,ind1),elmat(i,ind2)) ...
                        + 2*Selem(ind1,ind2);
                else
                    Sy(elmat(i,ind1),elmat(i,ind2)) = Sy(elmat(i,ind1),elmat(i,ind2)) ...
                        + Selem(ind1,ind2);
                end
            end;
        end;
        GenerateElementVector; % felem
        for ind1 = 1:topology
            fy(elmat(i,ind1)) = fy(elmat(i,ind1)) + felem(ind1);
        end;
    end;

% Next the boundary contributions

```

```


    for j = 1:length(bnd2); % left boundary
        i = bnd2(j);
        GenerateBoundaryElementMatrix; % BMelem
        for ind1 = 1:topologybnd
            for ind2 = 1:topologybnd
                if elmatbnd(i,ind1) == elmatbnd(i,ind2)
                    Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
                else
                    Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
                end;
            end
        end;
        GenerateBoundaryElementVector; % bfelem
        for ind1 = 1:topologybnd
            fy(elmatbnd(i,ind1)) = fy(elmatbnd(i,ind1)) - bfelem(ind1);
        end;
    end;

    for j = 1:length(bnd4); % right boundary
        i = bnd4(j);
        GenerateBoundaryElementMatrix; % BMelem
        for ind1 = 1:topologybnd
            for ind2 = 1:topologybnd
                if elmatbnd(i,ind1) == elmatbnd(i,ind2)
                    Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + 2*BMelem(ind1,ind2);
                else
                    Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) = ...
                        Sy(elmatbnd(i,ind1),elmatbnd(i,ind2)) + BMelem(ind1,ind2);
                end;
            end
        end;
        GenerateBoundaryElementVector; % bfelem
        for ind1 = 1:topologybnd
            fy(elmatbnd(i,ind1)) = fy(elmatbnd(i,ind1)) + bfelem(ind1);
        end;
    end;
end
```

B.7 Compute u and v_x/v_y

```
% Construction of linear problem

BuildMatricesandVectors;

% Solution of linear problem

u = S \ f;

BuildMatricesandVectors;

vx = Sx \ fx;
vy = Sy \ fy;
```

B.8 Full script

```
close all
clear all

%% 2D Assignment
% Lab Assignment 7

%% Create Mesh
WI4243Mesh

%% Parameters

Qp = 50;              % [m^2/s]
k = 10^-7;            % [m^2]
mu = 1.002*10^-3;     % [Pa*s]
K = 10000;               % [m/s]
pH = 10^6;            % [Pa]
N_wells = 6;          % number of wells

epsilon1 = 0.03;
N_Test = 0;
%% Coordinates of wells

for i = 1:N_wells-1;
xp(i) = 0.6*cos((2*pi)*(i-1)/(N_wells-1));
yp(i) = 0.6*sin((2*pi)*(i-1)/(N_wells-1));
end

xp(N_wells) = 0;
yp(N_wells) = 0;
clear i;


%% Compute Problem
WI4243Comp

%% Post
hold on

figure(2);
ax.BoxStyle = 'full';
hold off
trisurf(elmat,x,y,u)
xlabel('x'); ylabel('y'); zlabel('Pressure [Pa]');
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]']);

lgd = legend();
% title(lgd,['3D Surface plot, K = ' num2str(K) '\it Minimal pressure = ' ...
    num2str(Pressure_minimum)]);

% title( {'Title';'subtitle'} )

figure(3);
trisurf(elmat,x,y,u);
xlabel('x'); ylabel('y');
title(['\bf\fontsize{16}Contour plot, \fontsize{10} K = ',num2str(K,'%10.2e\n')]);
view(2); shading interp; colormap jet; colorbar; set(gcf,'renderer','zbuffer')
h = colorbar; ylabel(h, 'Pressure [Pa]');


figure(4); quiver(x,y,vx',vy'); axis([-1 1 -1 1]);
xlabel('x'); ylabel('y');
title(['\bf\fontsize{16}Velocity field, \fontsize{10} K = ' num2str(K,'%10.2e\n') ' ...
    [m/s]']);
%% Velocity part
hold off
```