Project - Nonlinear convection-diffusion equations

Oscar Jacobson *

December 29, 2022

Abstract

In this assignment, we are interested in solving a time-dependent nonlinear convectiondiffusion (NCD) equation in one space dimension. The project consists of three parts: Part A is on numerical linear algebra; Part B is on a finite difference method, and Part C is on a finite element method.[1] Implementation is done in matlab and results are linked back to theoretical explanations.

*Student, Masters Programme Engineering Physics, F3C, Uppsala University, E–mail: oscar.jacobson.9201@student.uu.se

[1]Project description

1 Introduction/Theory

The goal of this assignment is to solve the following initial value problem:

$$\partial_t u + \partial_x \beta(u) - \partial_x(\epsilon \partial_x u) = f, \quad (x,t) \in I \times (0,T],$$

$$\text{u(x,0)} = \text{u}_0, \qquad x \in I.$$

(1)

where $f := f(x,t), u_0 := u0(x)$ are given functions, T > 0 is the final time and the boundary conditions are to be defined.

The goal of the project is (1) to get familiar with partial differential equations (PDEs); (2) to learn some basic discretization techniques to numerically approximating these PDEs; to learn some basic iterative and direct methods to solve the resulting linear system of equations.[2]

The equation

$$Ax = b \tag{2}$$

where A is a matrix and b is a so called load vector can be solved in a lot of ways. The most straight forward method to solving this system is a direct method. A direct method rely on Gaussian elimination and matrix-matrix multiplication to get x. This can however, be very costly, in terms of computational power, especially if the matrix A is large or the equation is to be solved for a large number of different right hand side vectors.

For large systems of equations, methods that iteratively approximate solutions can be beneficial. Iterative methods have increased speed at the cost of precise accuracy. Direct methods are matrix-matrix multiplication and iterative methods are matrix-vector multiplications which use less computational power. As long as iterations are much smaller than the matrix size we save time and computational work.[3]

[2]Project description

[3]Lecture 2

Solving speed for iterative solvers like the Jacobi method and Gauss-Seidl method are highly dependent on the problem and the amount of iterations are hard to predict. The iterative conjugate gradient method can limit the amount of iterations to a known number. The drawback of the conjugate gradient method is that it is highly dependent on the problem as it only can be used on a symmetric positive definite matrix. The conjugate gradient matrix always converges in at most N (Matrix size) iterations and is therefore always beneficial to use when the problem allows to.[4]

All iterative methods have some requirements to ensure a converging solution. Classical iterative methods like the Jacobi or Gauss-Seidl method are only convergent if the spectral radius of the posed system is less than one. A simple way to check this condition is to realize that the spectral radius is always less than one when A is strictly diagonally dominant. This means that the absolute value of the diagonal element in the A matrix of equation 2 is larger than the sum of the absolute value for the rest of the values on the same row, for every row.[5]

When solving systems of differential equations, pure matrix solvers cannot be used. The problem has to be formulated in a way that makes the matrix solvers usable first.

A finite difference method is the computational approximation of derivatives and can be used to solve systems of equations when applied to a discretized space.

[4]Lecture 2

[5]Lecture 3

I havn't been able to finish the last part of the project and have put a lot of focus on trying to figure out how to formulate my matrices. Won't be able to provide a full rapport as of today.

Lecture 4 about FDE 1 -2 1 matrix for central difference

Epsilon is calculated to be h/2 on paper. Basic solution is that

(-1/(2h) + eps* $1/h^2$) = 0

=> eps = 1/(2h)

Accuracy approximation tau -> 0 when h -> 0

Lecture 6 von neuman and $h^2/2 < dt$

2 Part A

2.1

In part A a few select numerical solvers are implemented in matlab and evaluated for some set systems of equations. Table 1 shows the iterations as well as time taken for the different solvers to compute a four by four Ax = b problem.
The second and third part of A compare the solution time for the different methods when the A matrix of equation 2 is firstly a randomly generated matrix of sizes (100, 500, 1000) and a diagonal weight of (1, 5, 10, 100) and secondly the A matrix is tridiagonal with negative two plus an epsilon at the diagonal and ones at the upper and lower diagonal.

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	12	0.0075274
$Gauss - Seidel$	5	0.0087708
CG	4	0.0111283
$MyownLU$		0.0079417
$Matlab - LU$		0.0005688
$Matlab - backslash$		0.0014866

Table 1: Iteration count and solving time for the select solvers

2.2 Discussion

Matlab's built in functions are written in C language and are therefore highly optimized compared to the ones written by me in matlab. This can clearly be seen in the time taken by the solver myownLU for large matrix sizes.

In the second part of this part of the project the value of w affects the random matrix to be more likely to be diagonally dominant. For large values of w the

diagonal is more likely to be dominant and the larger the matrix is the less likely the matrix is to be diagonally dominant for all rows.

Diagonal domination is as stated earlier connected to the eigenvalues and spectral radius of the solvers which require the spectral radius to be less than one for convergence. A lot of the times the solutions does not converge and in appendix A for tables these non-converging cases are marked with either a star (*) for a NAN value or a dash (-) for a INF value.

Worth noting from this part is that the large values of w causes the matrix to inherit positive definite qualities and can be solved bu the conjugate gradient method if allowed for small errors. This can be seen in tables 11, 12, 13 in the appendix A.

The convergence rate of the iterative methods in the last part of the A part are, again, dependent on the matrix being diagonally dominant. When the epsilon on the diagonal is shrinking the spectral radius of the matrix is affected. When the diagonal is almost equal to two the matrix spectral radius is dangerously close to one and the method is having a hard time to converge. This can be seen in table 17 where the iterative methods converge slowly.

3 Part B

3.1 Discussion

The method strictly converges as h goes to zero for the simulation over a smooth initial value problem. The second part introduces sharp edges which the method has a hard time simulating in a reasonable mesh size over time and space. This causes the solution to oscillate and does not obviously converge when h goes to zero.

The epsilon introduced in part 3 causes a dampening/diffusion of the solution resulting from the diffusion matrix. The oscillations get reduced but so does the original signal. The error estimate seems to be of the same order and magnitude as the estimate for the second part of B.

Order of convergence can be calculated by:

$$p = log(e_1/e_2)/log(h_1/h_2) \tag{3}$$

Have been focusing on finishing the last part and will add this later.

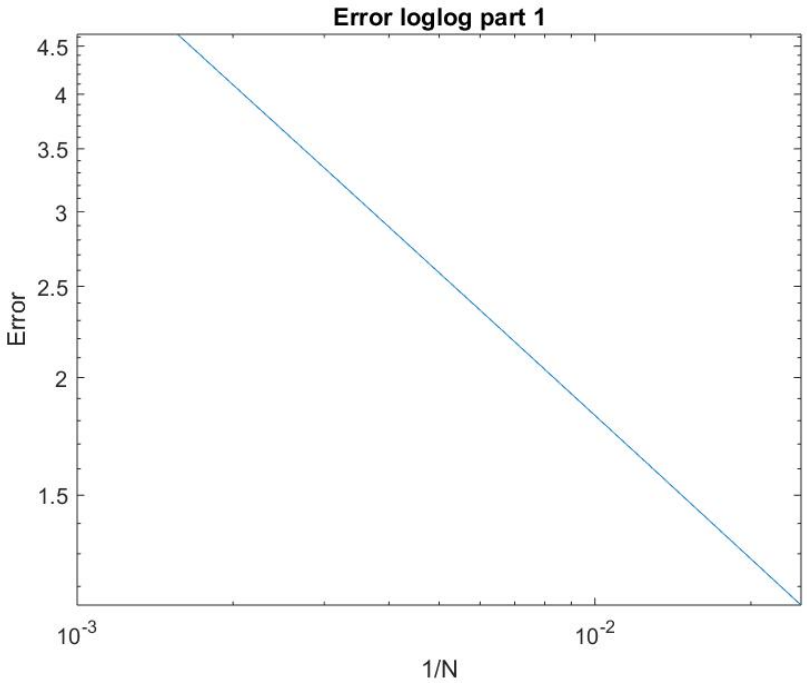

Figure 1: *Error plot for part B.1*

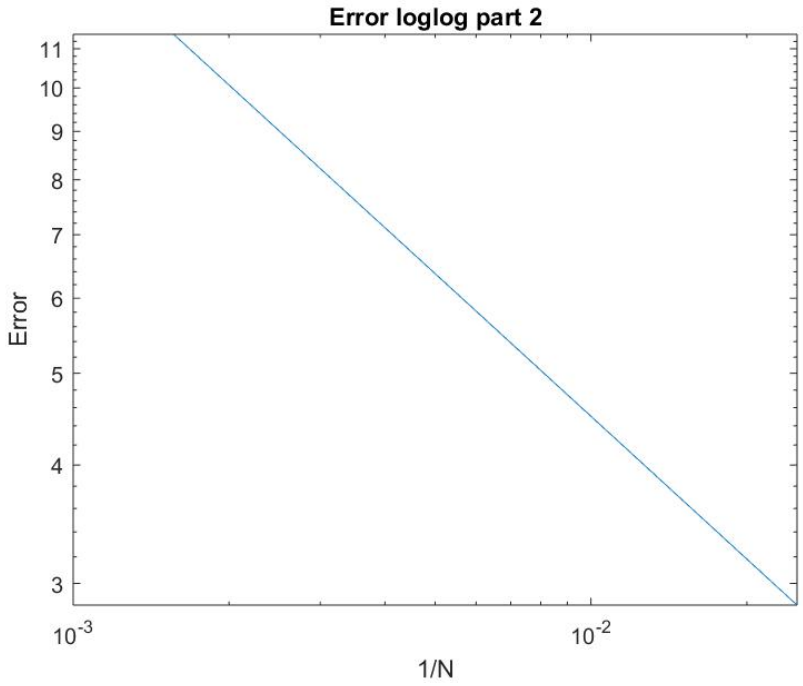

Figure 2: *Error plot for part B.2*

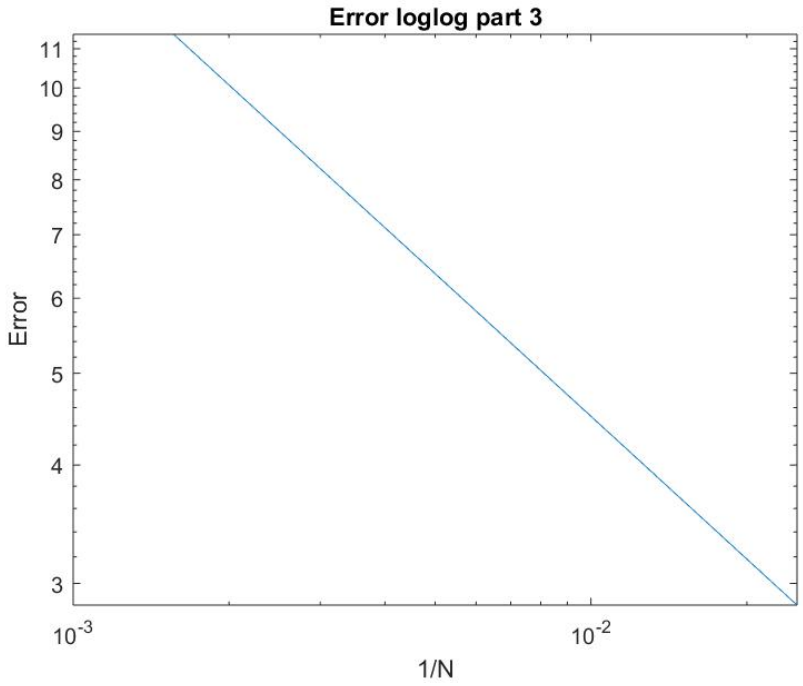

Figure 3: *Error plot for part B.3*

4 Part C

Im having a hard time formulating this part so that it does not oscillate, I'm convinced that my matrices are slightly faulty. Will not be able to provide the error estimates for this part as of now.

4.1 PDE toolbox

A convection diffusion problem was solved using the PDE toolbox in matlab. Following figures are the resulting graphs from using different epsilon. (Will formulate the problem better when i can provide the entirety of section C)

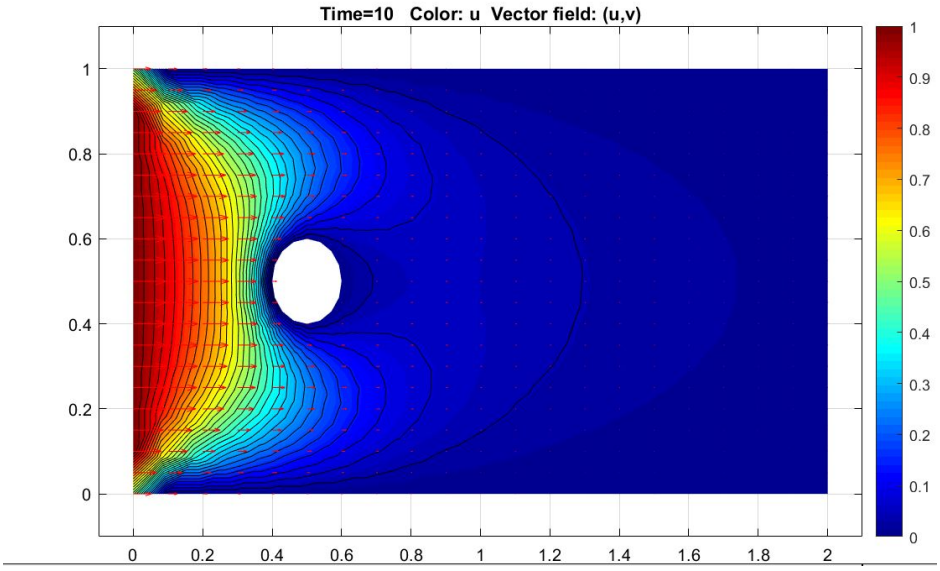

Figure 4: *Eps = 0.1*

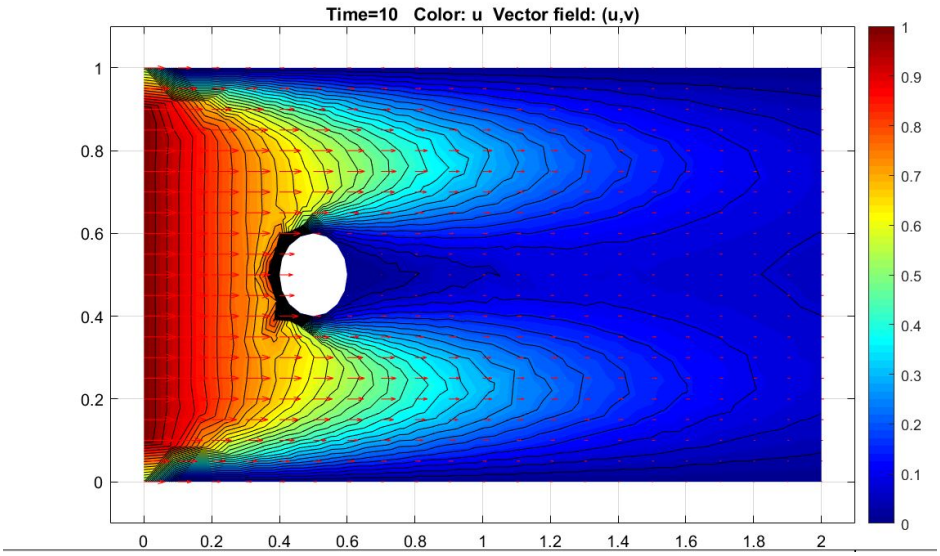

Figure 5: *Eps = 0.01*

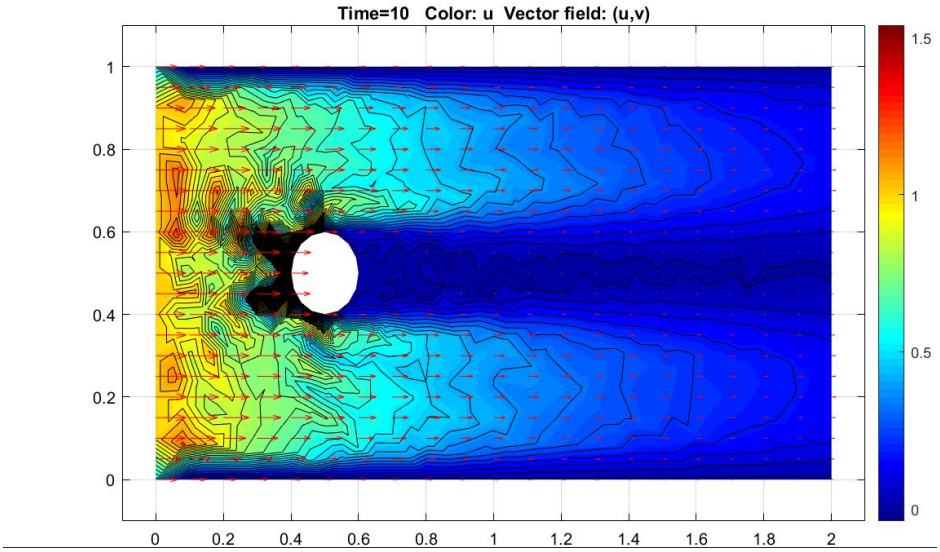

Figure 6: *Eps = 0.001*

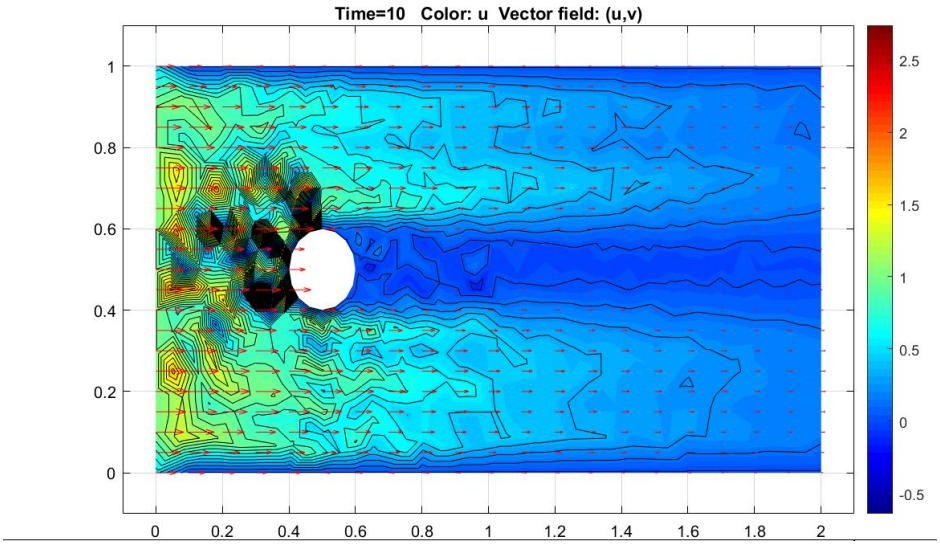

Figure 7: *Eps = 0.0001*

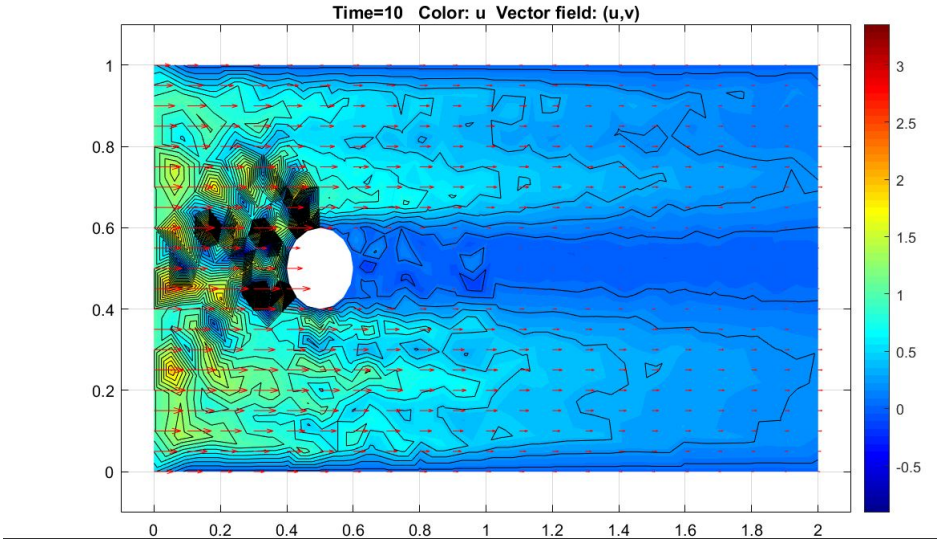

Figure 8: $Eps = 0$

A Appendix tables A

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	201*	0.0187561
$Gauss - Seidel$	240-	0.0131204
CG	*	
$MyownLU$		0.061903
$Matlab - LU$		0.002728
$Matlab - backslash$		0.0025158

Table 2: Part B.2 w=1 n = 100

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	140*	0.451327
$Gauss - Seidel$	65-	0.116249
CG	*	
$MyownLU$		108.664
$Matlab - LU$		0.0080861
$Matlab - backslash$		0.0099613

Table 3: Part B.2 w = 1 n = 500

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	123*	0.920009
$Gauss - Seidel$	31-	0.247871
CG	*	
$MyownLU$		1772.67
$Matlab - LU$		0.0512989
$Matlab - backslash$		0.0474231

Table 4: Part B.2 w=1 n = 1000

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	323*	0.0399747
$Gauss - Seidel$	54-	0.024575
CG	*	
$MyownLU$		0.0547081
$Matlab - LU$		0.0029901
$Matlab - backslash$		0.0011714

Table 5: Part B.2 w = 5 n = 100

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	188*	0.414586
$Gauss - Seidel$	1254*	2.02861
CG	*	
$MyownLU$		102.252
$Matlab - LU$		0.0152326
$Matlab - backslash$		0.0204017

Table 6: Part B.2 w = 5 n = 500

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	159*	1.15178
$Gauss - Seidel$	471*	2.849
CG	*	
$MyownLU$		1752.27
$Matlab - LU$		0.0377167
$Matlab - backslash$		0.0570014

Table 7: Part B.2 w = 5 n = 1000

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	460*	0.0287274
$Gauss - Seidel$	30	0.0036566
CG	*	
$MyownLU$		0.0526854
$Matlab - LU$		0.0006151
$Matlab - backslash$		0.0031379

Table 8: Part B.2 w = 10 n = 100

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	226*	0.431192
$Gauss-Seidel$	396	0.830012
CG	*	
$MyownLU$		102.534
$Matlab-LU$		0.0069429
$Matlab-backslash$		0.0099414

Table 9: Part B.2 w = 10 n = 500

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	185*	1.68696
$Gauss-Seidel$	1222	7.85184
CG	*	
$MyownLU$		1839.98
$Matlab-LU$		0.123911
$Matlab-backslash$		0.130965

Table 10: Part B.2 w = 10 n = 1000

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	16	0.0104627
$Gauss-Seidel$	6	0.0079048
CG	8(small error)	0.0091671
$MyownLU$		0.0474276
$Matlab-LU$		0.0015348
$Matlab-backslash$		0.0006297

Table 11: Part B.2 w = 100 n = 100

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	783*	1.57256
$Gauss-Seidel$	15	0.0430503
CG	15(Small error)	0.0098688
$MyownLU$		128.936
$Matlab-LU$		0.0062739
$Matlab-backslash$		0.0085494

Table 12: Part B.2 w = 100 n = 500

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	445*	3.87613
$Gauss-Seidel$	29	0.360467
CG	40 (Small error)	0.0390879
$MyownLU$		2093.68
$Matlab-LU$		0.138288
$Matlab-backslash$		0.0484388

Table 13: Part B.2 w = 100 n = 1000

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	28	0.0971871
$Gauss-Seidel$	17	2.30117
CG	16	1.50825
$MyownLU$		
$Matlab-LU$		0.0071926
$Matlab-backslash$		0.0004348

Table 14: Part B.3 alpha = 1

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	229	0.218631
$Gauss-Seidel$	116	6.24477
CG	49	1.56969
$MyownLU$		
$Matlab-LU$		0.0123578
$Matlab-backslash$		0.0016464

Table 15: Part B.3 alpha = 0.1

$Method$	$Iteration$	$Time(sec)$
$Jacobi$	21777	5.98403
$Gauss-Seidel$	10862	432.141
CG	503	1.56318
$MyownLU$		
$Matlab-LU$		0.0052872
$Matlab-backslash$		0.0011253

Table 16: Part B.3 alpha = 0.001

Method	*Iteration*	*Time(sec)*
Jacobi	2166480	421.476
Gauss – Seidel	Hardly converging	(8.5 hours if at the same rate as alpha = 0.001)
CG	4941	2.16111
MyownLU		
Matlab – LU		0.0114659
Matlab – backslash		0.0007518

Table 17: Part B.3 alpha = 0.00001

B Appendix code A

Project A.1

```
%  A = [10,-1,2,0;-1,11,-1,3;2,-1,10,-1;0,3,-1,8];
%  b = [6;25;-11;15];
%  TOL = 0.001;


tic ;
[x2,i2] = jacobi (A ,b , TOL );
fprintf ('Jacobi took %g sec  in %g iterations \n' , toc, i2);
tic ;
[x3,i3] = gs (A ,b , TOL );
fprintf (' GS took %g sec  in %g iterations \n' , toc, i3 );
 tic ;
%  [x4,i4,] = cg (A ,b , TOL );
%  fprintf (' CG took %g sec  in %g iterations \n'  , toc, ...
    i4 );
tic ;
% x5 = myownLU (A , b );
% fprintf (' myownLU took %g sec \n ' , toc );
tic ;
[L , U ] = lu( A );
y = L \ b ;
x6 = U \ y ;
fprintf ('Matlab LU took %g sec \n ' , toc );
tic ;
x1 = A \ b ;
fprintf ( ' Backslash took %g sec \n ' , toc );


norm(x2-x1,2)
norm(x3-x1,2)
```

```
%norm(x4-x1,2)
%norm(x5-x1,2)
```

Project A.2

```
w = [1,5,10,100]; % the diagonal weight
N = [100,500,1000]; % the size of the linear system
TOL = 0.00001; % torelance

n = 3;
W = 4;

A = rand(N(n)) + diag ( w(W) * ones(N(n) ,1));
b = rand (N(n) ,1);


% n = 1 w = 1
% Backslash took 0.0025158 sec
%  Jacobi took 0.0187561 sec  in 201 iterations
%  GS took 0.0131204 sec  in 240 iterations
%  myownLU took 0.061903 sec
%  Matlab LU took 0.002728 sec

% n = 2 w = 1
% Backslash took 0.0099613 sec
%  Jacobi took 0.451327 sec  in 140 iterations
%  GS took 0.116249 sec  in 65 iterations
%  myownLU took 108.664 sec
%  Matlab LU took 0.0080861 sec

% n = 3 w = 1
%  Backslash took 0.0474231 sec
%  Jacobi took 0.920009 sec  in 123 iterations
%  GS took 0.247871 sec  in 31 iterations
%  myownLU took 1772.67 sec
%  Matlab LU took 0.0512989 sec

% n = 1 w = 2
%  Backslash took 0.0011714 sec
%  Jacobi took 0.0399747 sec  in 323 iterations
%  GS took 0.024575 sec  in 54 iterations
%  myownLU took 0.0547081 sec
%  Matlab LU took 0.0029901 sec

%n = 2 w = 2
%  Backslash took 0.0204017 sec
%  Jacobi took 0.414586 sec  in 188 iterations
%  GS took 2.02861 sec  in 1254 iterations
```

```
%  myownLU took 102.252 sec
%  Matlab LU took 0.0152326 sec

% n = 3 w =2
%  Backslash took 0.0570014 sec
%  Jacobi took 1.15178 sec  in 159 iterations
%  GS took 2.849 sec  in 471 iterations
%  myownLU took 1752.27 sec
%  Matlab LU took 0.0377167 sec

% n = 1 w = 3
%  Backslash took 0.0031379 sec
%  Jacobi took 0.0287274 sec  in 460 iterations
%  GS took 0.0036566 sec  in 30 iterations
%  myownLU took 0.0526854 sec
%  Matlab LU took 0.0006151 sec

% n = 2 w = 3
%  Backslash took 0.0099414 sec
%  Jacobi took 0.431192 sec  in 226 iterations
%  GS took 0.830012 sec  in 393 iterations
%  myownLU took 102.534 sec
%  Matlab LU took 0.0069429 sec

% n = 3 w = 3
% ProjectA
%  Backslash took 0.130965 sec
%  Jacobi took 1.68696 sec  in 185 iterations
%  GS took 7.85184 sec  in 1222 iterations
%  myownLU took 1839.98 sec
%  Matlab LU took 0.123911 sec
```

Project A.3

```
a = [1, 0.1, 0.001, 0.00001]; % the diagonal weight
N = 10000; % the size of the linear system
TOL = 0.00001; % torelance

A = 2;

%A from lecture notes
A = diag(repmat(2+a(A),1,N), 0) + diag(repmat(-1,1,N-1), -1) ...
    + diag(repmat(-1,1,N-1), 1);
A = sparse(A);
b = rand (N ,1);
```

Jacobi solver

```
function [x,i] = jacobi(A,b,TOL)
%Jacobi's iterative method

%m = height,rows   n = lenght,columns
[m,n] = size(A);
x = zeros(n,1);

D = diag(diag(A));
Di = inv(D);
U = triu(A,1);
L = tril(A,-1);
i = 0;
%x = d^-1[-(L+U)x +b]

while norm(A*x - b,inf) > TOL
    xn = Di*(-(L+U)*x +b);
    x = xn;
    i = i+1;
end
end
```

Gauss Seidel solver

```
function [x,i] = gs(A,b,TOL)
%Gauss-Seidel's iterative method

%m = height,rows   n = lenght,columns
[m,n] = size(A);
x = zeros(n,1);

D = diag(diag(A));
U = triu(A,1);
L = tril(A,-1);
DLi = inv(D+L);
i = 0;
%x = (d+L)^-1[-Ux +b]

while norm(A*x - b,inf) > TOL
    xn = DLi*(-U*x +b);
    x = xn;
    i = i+1;
end
```

My own LU solver

```
function x = myownLU(A,b)
%Egen LU faktorisering
%Test matrix A = [10,-1,2,0;-1,11,-1,3;2,-1,10,-3;0,3,-1,8]
```

```
4  %Test matrix b = [6;25;-11;15]
5
6  %m = height,rows   n = lenght,columns
7  [m,n] = size(A);
8  L = diag(ones(1,m));
9  %G? igenom column f?r column, en under diag till n?st sista, ...
       subtract same row as operated times s/f, s/f = first ...
       value of
10 %row over first value of row operated upon.
11 for col = 1:n-1
12     f = A(col,col);
13     for row = col+1:m
14         s = A(row,col);
15         L(row,col) = s/f;
16         A(row,1:n) = A(row,1:n)-(s/f).*A(col,1:n);
17         U = A;
18     end
19 end
20
21 y = L\b;
22 x = U\y;
23
24
25
26
27
28 %L*d =b f?s d med forward
29
30 %homemade forward subs
31 % d = zeros(n,1);
32 % x = zeros(n,1);
33
34 %for row = 1:m
35     %Times 1 on the diag
36     %d(row) =  L(row,row)*b(row);
37     %If row larger than 1 subtract previous d times lower of ...
          row,col
38    % if row > 1
39         %for col = 1:(row-1)
40            % d(row) = d(row)- L(row,col)*d(col);
41         %end
42   %  end
43 %end
44
45 %Homemade backward subs (not working properly)
46 %Ux = d f?s x med backward
47 % for ro = m:-1:1
48 %     x(ro) = U(ro,ro)*d(ro);
49 %     if ro < m
```

```
%          for co =(ro+1):m
%              x(ro) = x(ro) -U(ro,co)*x(co);
%          end
%      end
% end
end
```

Conjugate Gradient solver

```
function [x,i] = cg(A,b,TOL)
%Conjugate Gradient method

% if all(eig(A)) > 0 && issymmetric(A) ~= logical(true)
%     return
% end

%m = height,rows   n = lenght,columns
[m,n] = size(A);
x = ones(n,1);

r = b-A*x;
rho = r'*r;
i = 0;

while sqrt(rho) > TOL
    i = i+1;
    if i == 1
        P = r;
    else
        beta = rho/rho2;
        P = r+beta*P;
    end
    w = A*P;
    alph = (P'*r)/(P'*w);
    x = x+alph*P;
    r = r-alph*w;
    rho2 = rho;
    rho = r'*r;
end
end
```

C Appendix figures B

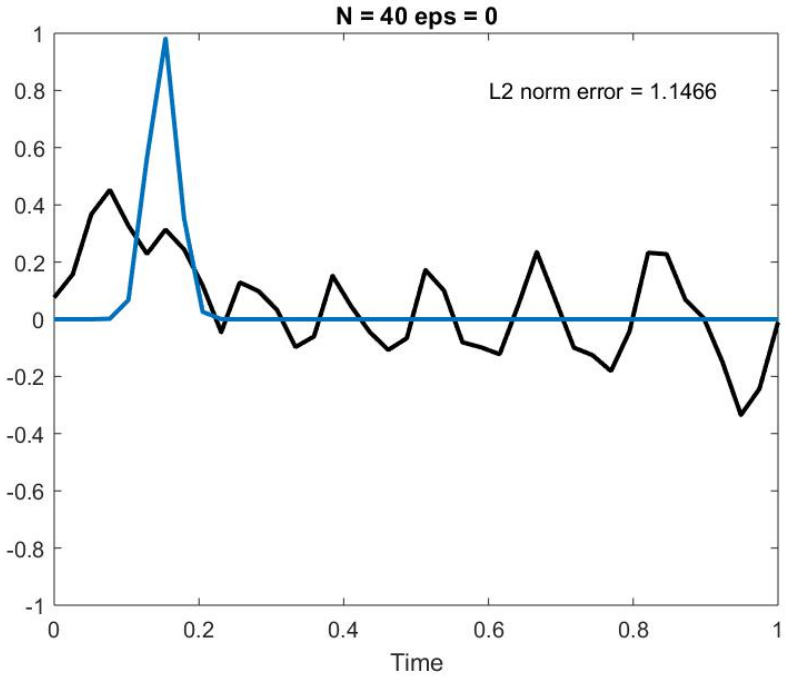

Figure 9: *Part B.1*

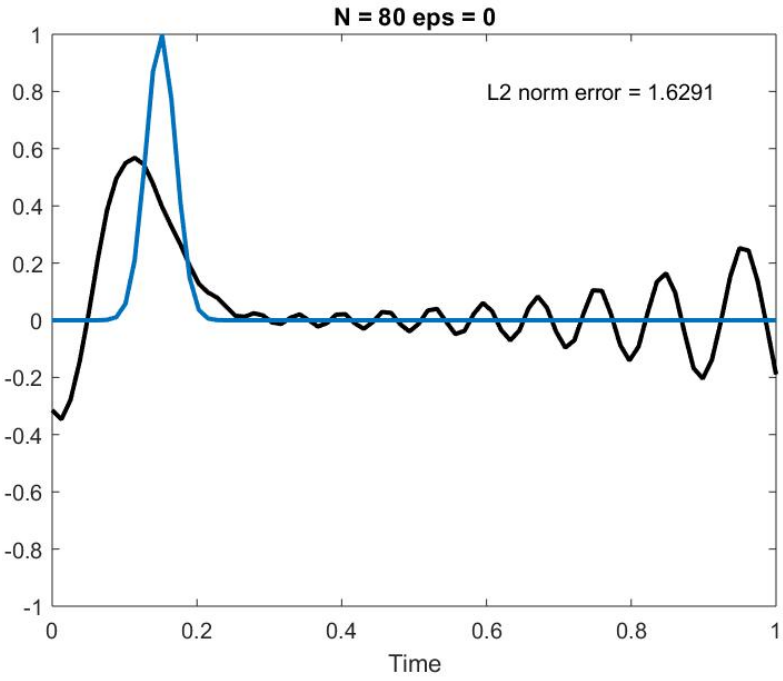

Figure 10: *Part B.1*

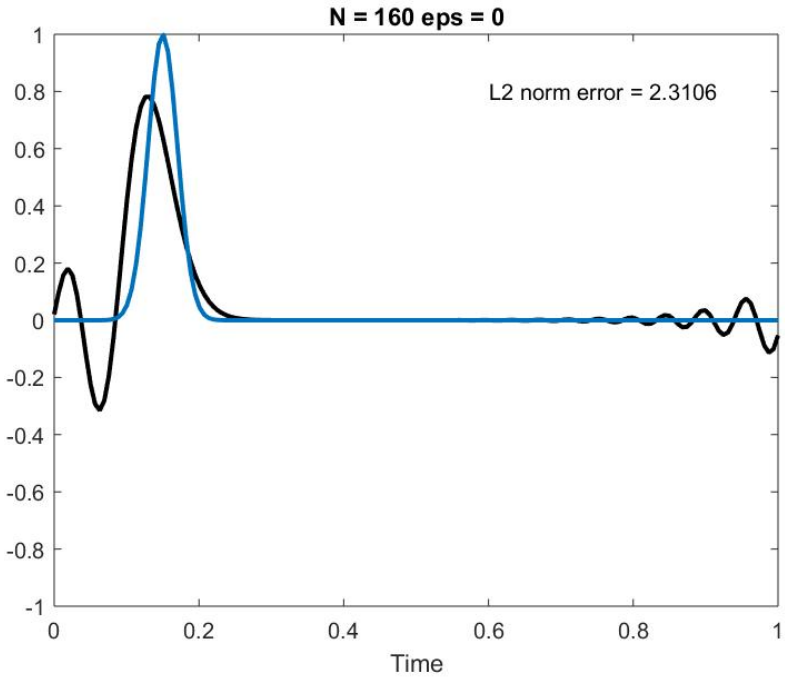

Figure 11: *Part B.1*

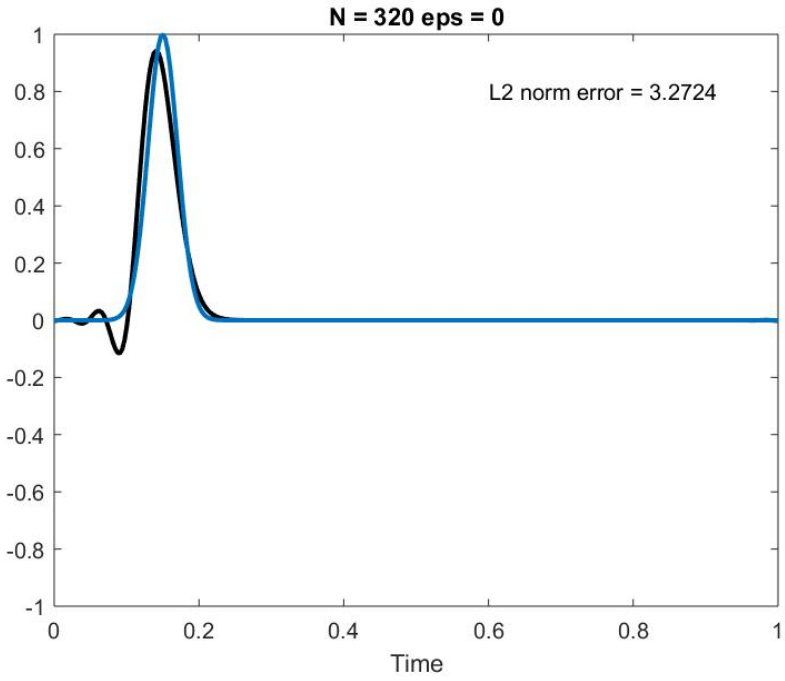

Figure 12: *Part B.1*

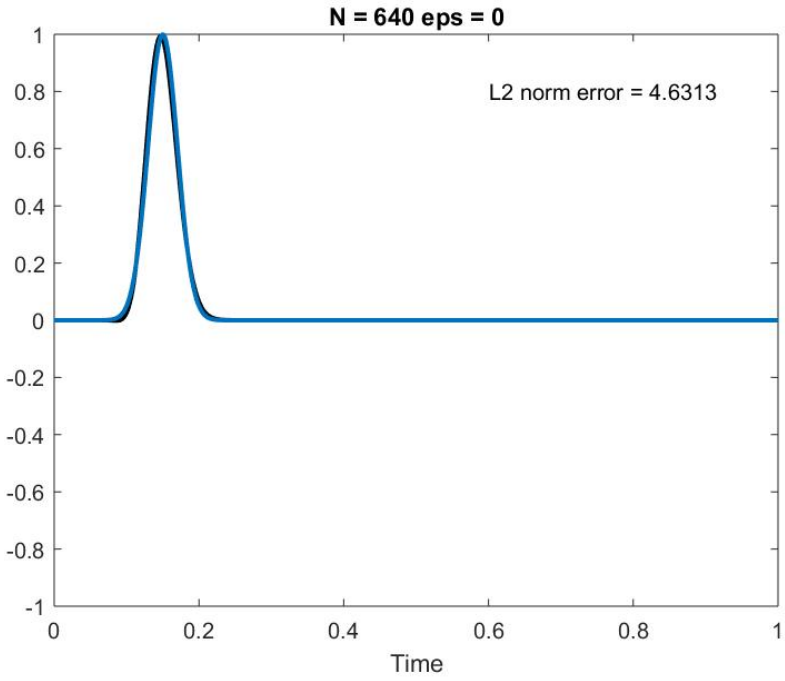

Figure 13: *Part B.1*

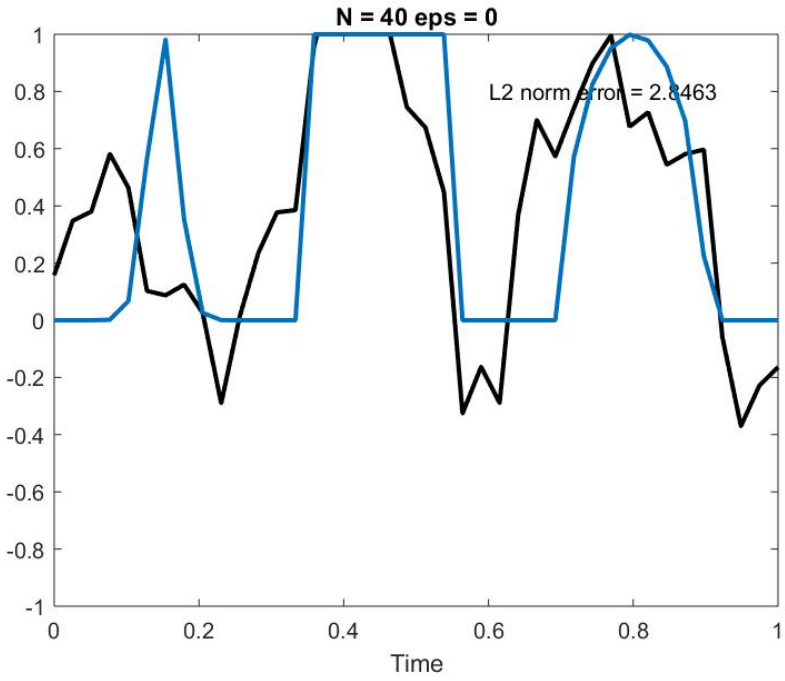

Figure 14: *Part B.2*

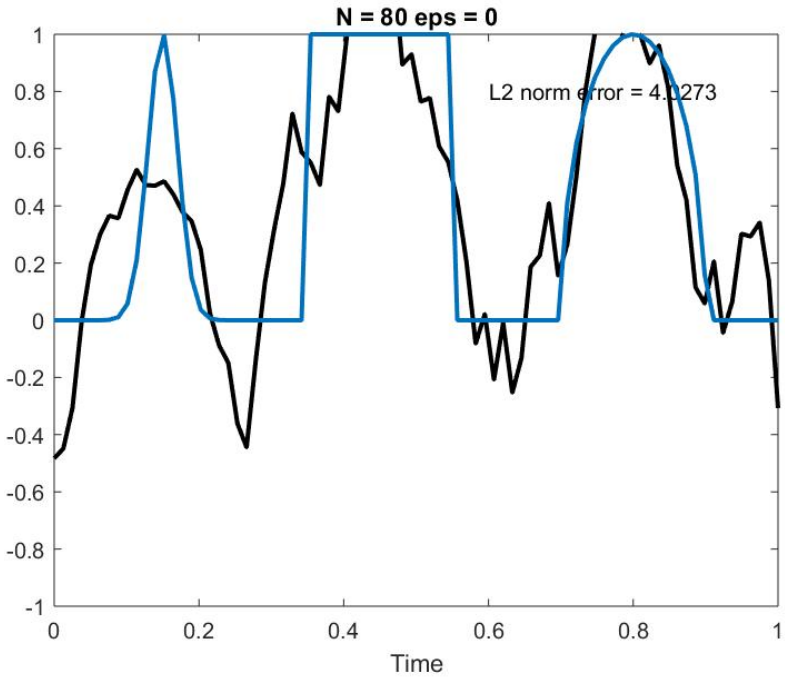

Figure 15: *Part B.2*

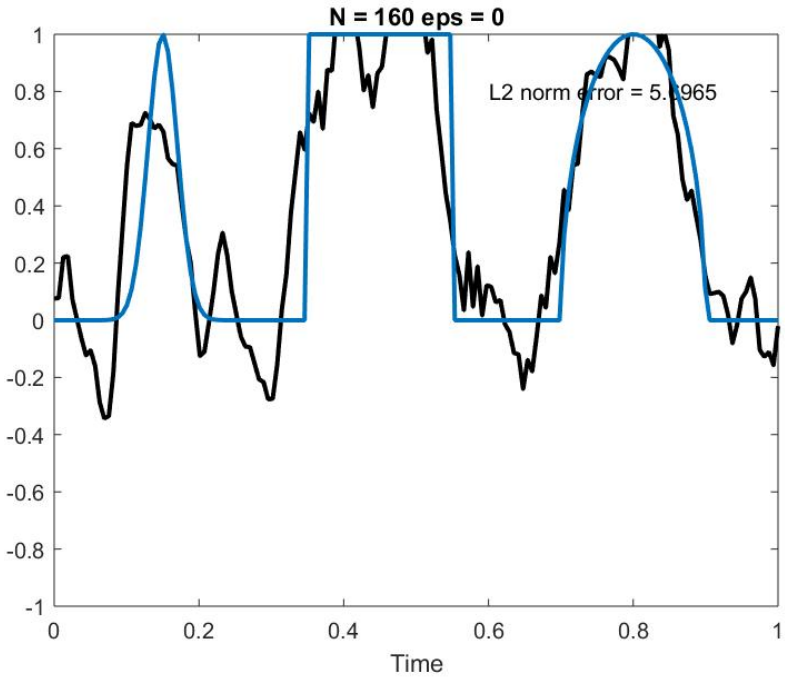

Figure 16: *Part B.2*

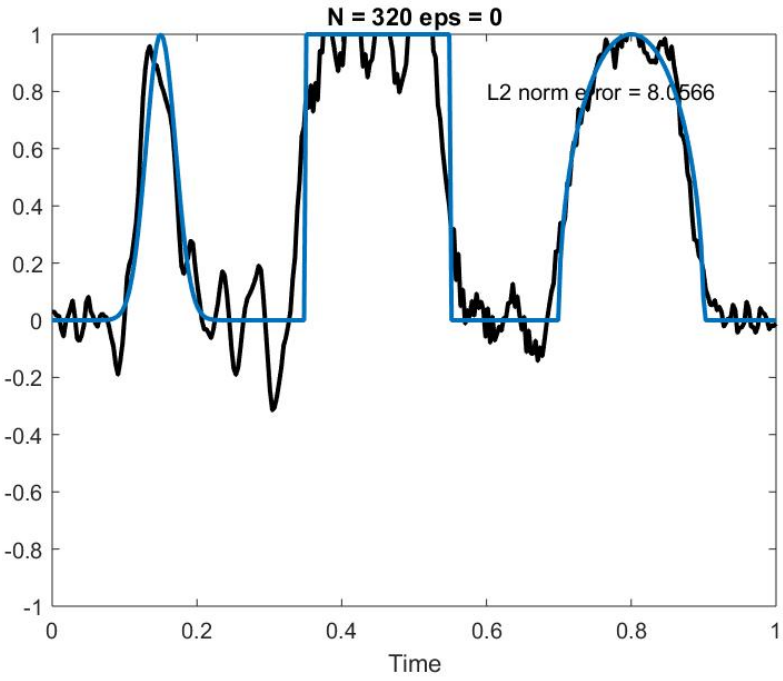

Figure 17: *Part B.2*

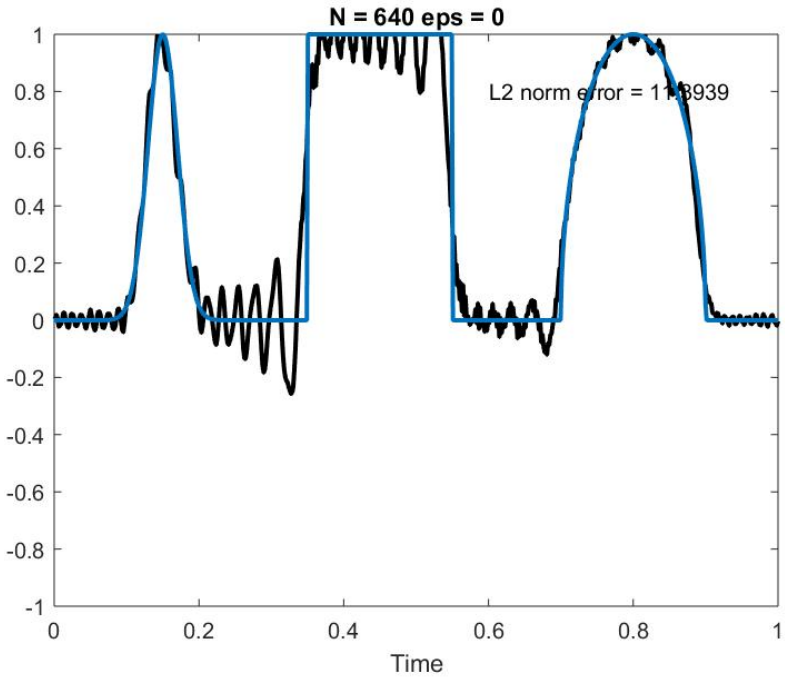

Figure 18: *Part B.2*

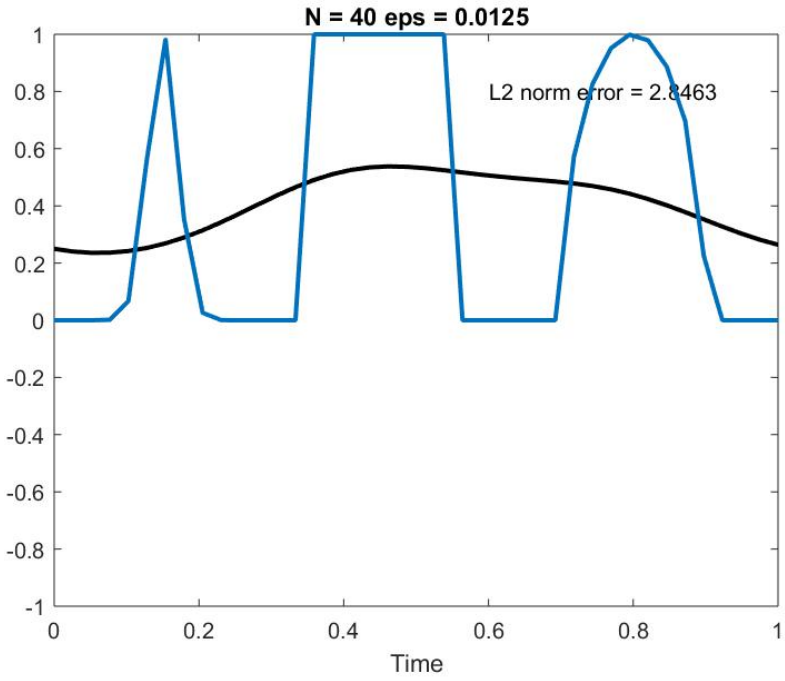

Figure 19: *Part B.3*

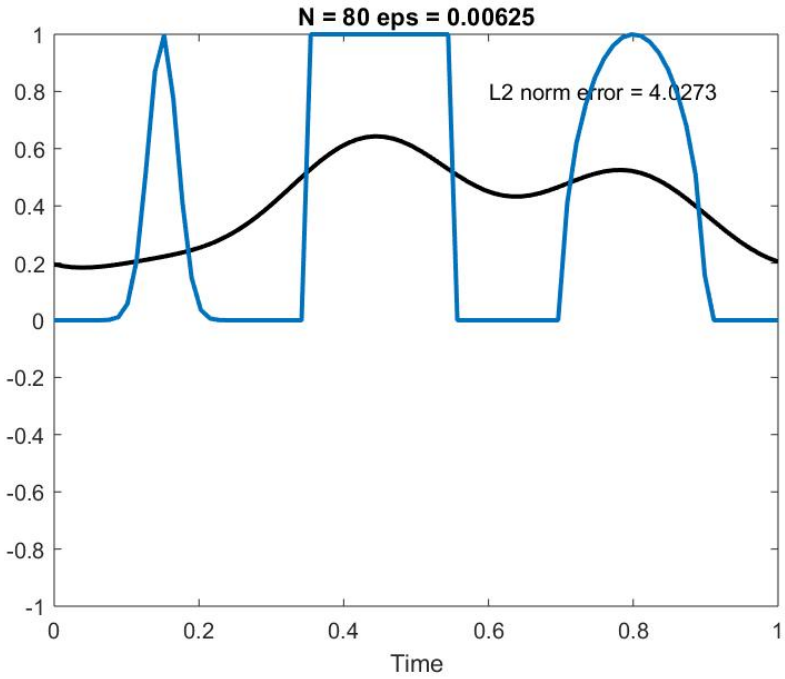

Figure 20: *Part B.3*

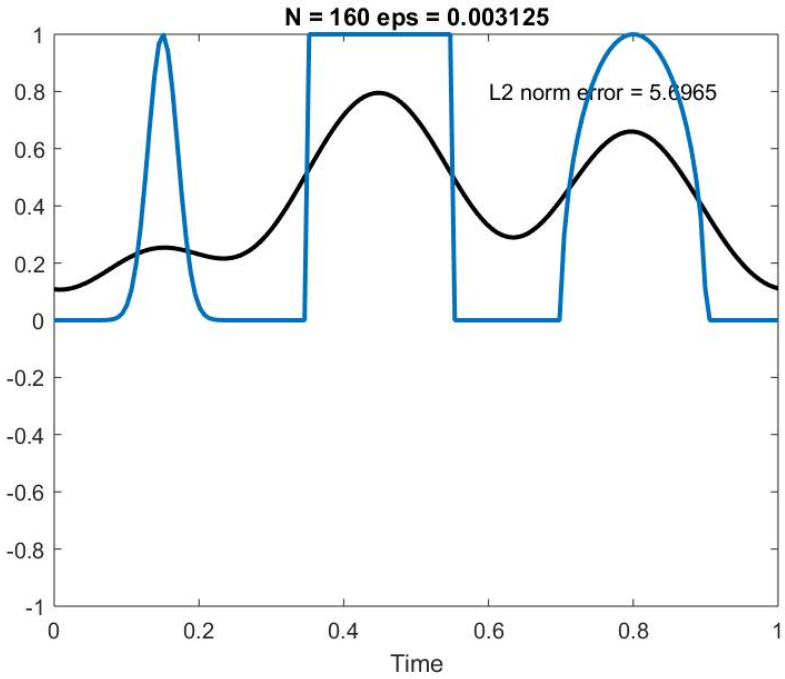

Figure 21: *Part B.3*

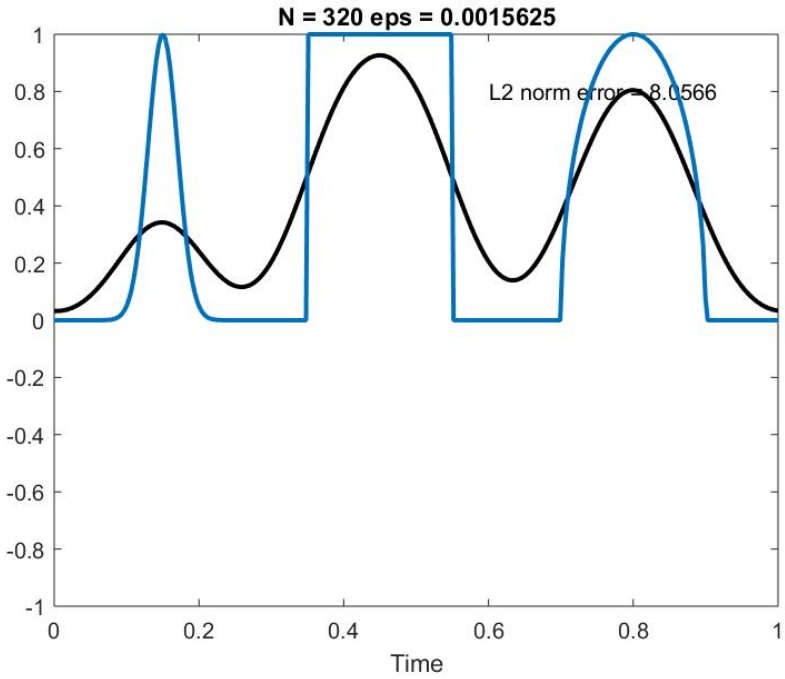

Figure 22: *Part B.3*

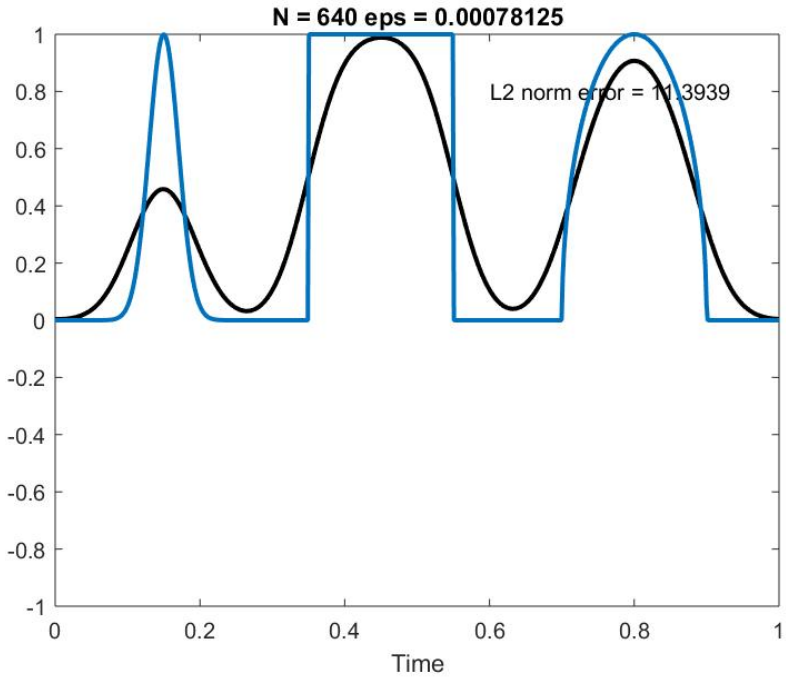

Figure 23: *part B.3*

D Appendix code B

RK4 solver for FDM

```
clear all

u_initial = @(x) exp(-300*(2*x-0.3).^2);

%Time domain
x0 = 0;
xend = 1;

%Timestep 40 80 160 320 640
%start end with 640 and the error will be plotted (if N = ...
    640 => plot)
N = 320;

%Epsilon for dxx
eps = 0;

%Discretise the spatial domain
X = linspace(x0,xend,N)';
```

```
18
19
20
21 %Spatial step
22 h = 1/N; % = X(2)-X(1)
23 %eps = h/2;
24
25 %Advection matrix dx
26 A =  1/h * spdiags(repmat([-1/2, 1/2], N, 1), [-1, 1], N, N);
27 A(1,end) = A(2,1);
28 A(end,1) = A(end-1,end);
29
30 %Diffusion matrix dxx
31 S = 1/h^2* spdiags(repmat([-2, 1, 1], N, 1), [0, -1, 1], N, N);
32 S(1,end) = S(2,1);
33 S(end,1) = S(end-1,end);
34
35
36
37 %Time domain and time step
38 tend = 1;
39 dt = 1/N;
40
41 %Discretized time domain
42 T = 0:dt:tend;
43
44 %Not sure if boundary conditions apply as xN = x1 and so on
45 %using the entire spatial domain for my initial value of u
46
47 %Create the initial values for u
48 for i = 1:length(X)
49     %Condition from proj
50     if abs(2*X(i)-0.3) <= 0.25
51         u0(i) = exp(-300*(2*X(i)-0.3).^2); %Creates the ...
               signal that we want to propagate
52
53     elseif abs(2*X(i)-0.9) <= 0.2
54         u0(i) = 1;
55
56     elseif abs(2*X(i)-1.6) <= 0.2
57         u0(i) = sqrt((1-((2*X(i)-1.6)/0.2)^2));
58
59     else
60         u0(i) = 0;
61     end
62 end
63
64
65
```

```
66
67
68  %Stepping in the time domain with classical RK4
69  u = zeros(length(X), length(T));
70  u(:,1) = u0;
71
72  %Define function for rk4
73  F = @(u,t) (-A + eps*S)*u;
74
75  for k = 2:length(T)
76      %Euler fram: u(:,k) = u(:,k-1) + dt*A*u(:,k-1);
77
78      %RK4;
79  %     k1 = h*f(t(i-1),y(i-1));
80  %     k2 = h*f(t(i-1)+h/2, y(i-1)+k1/2);
81  %     k3 = h*f(t(i-1)+h/2, y(i-1)+k2/2);
82  %     k4 = h*f(t(i-1)+h, y(i-1)+k3);
83  %     y(i) = y(i-1) + (k1+2*k2+2*k3+k4)/6;
84
85  %Right now the half-time steps (t + h/2 and t + h) are ...
        disregarded in the RK method, unsure
86  %if this is the correct implementation but it seems to work ...
        quite well.
87  %Update the initial RHS of the ODE system does not depend on ...
        time and i get
88  %away with leaving it out.
89
90
91  %Updated the code to work with an independen function F @(u,t)
92        k1 = dt*F(u(:,k-1), T(k));
93        k2 = dt*F((u(:,k-1)+k1/2), T(k)+dt/2);
94        k3 = dt*F((u(:,k-1)+k2/2), T(k)+dt/2);
95        k4 = dt*F((u(:,k-1)+k3), T(k)+dt);
96        u(:,k) = u(:,k-1) + (k1+2*k2+2*k3+k4)/6;
97  end
98
99  ue = u_initial(X);
100
101 e = u0-u(:,end);
102 e = norm(e,2)/sqrt(length(e));
103 %Paste this to get the error estimate
104 %part = 1 or 2
105 % q = [40 80 160 320 640];
106 % for i = 1:length(q)
107 %     N = q(i);
108 %     e(i) = errorprojB(N,1);
109 % end
110 % figure;
111 % loglog(1./q,e);
```

```
112
113 figure;
114 for k=1:length(T)
115     hold off;
116     plot(X, u(:,k), '-k', 'LineWidth', 2);
117
118     %' = 1/(2N)'
119     title(['N = ' num2str(N) ' eps = ' num2str(eps) ]);
120     axis([X(1), X(end), -1, 1]);
121     xlabel('Time');
122     text(0.6,0.8,['L2 norm error = ' num2str(e)]);
123     drawnow;
124 end
125 hold on
126 plot(X,u0, 'linewidth',2);
```

Following are modified versions of the FDM solver to get the error estimate graph

```
1  clear all
2  %Paste this to get the error estimate
3  %part = 1 or 2
4  q = [40 80 160 320 640];
5  part = 3; %Part can be 1,2 or 3
6  for i = 1:length(q)
7      N = q(i);
8      e(i) = errorprojB(N,part);
9  end
10 figure;
11 loglog(1./q,e);
12 title([ 'Error loglog part ' num2str(part)]);
13 ylabel('Error');
14 xlabel('1/N');
```

```
1  function e = errorprojB(N,part)
2  %function handle and a modification to only get the error ...
       estimate without
3  %plots
4
5  u_initial = @(x) exp(-300*(2*x-0.3).^2);
6
7  %Time domain
8  x0 = 0;
9  xend = 1;
10 %Timestep 40 80 160 320 640
11 %start end with 640 and the error will be plotted (if N = ...
       640 => plot)
12
```

```
13
14 %Epsilon for dxx
15 %Eps around and lower than 0.003 is stable for dt = 0.0005
16
17
18 %Discretise the spatial domain
19 X = linspace(x0,xend,N)';
20
21
22
23 %Spatial step
24 h = 1/N; % = X(2)-X(1)
25 %eps = h/2;
26 if part ~= 3
27     eps = 0;
28 else
29     eps = h/2;
30 end
31
32
33 %Advection matrix dx
34 A =  1/h * spdiags(repmat([-1/2, 1/2], N, 1), [-1, 1], N, N);
35 A(1,end) = A(2,1);
36 A(end,1) = A(end-1,end);
37
38 %Diffusion matrix dxx
39 S = 1/h^2* spdiags(repmat([-2, 1, 1], N, 1), [0, -1, 1], N, N);
40 S(1,end) = S(2,1);
41 S(end,1) = S(end-1,end);
42
43
44 %Time domain and time step
45 tend = 1;
46 dt = 1/N;
47
48 %Discretized time domain
49 T = 0:dt:tend;
50
51 %Not sure if boundary conditions apply as xN = x1 and so on
52 %using the entire spatial domain for my initial value of u
53
54 if part == 1
55     for i = 1:length(X)
56         %Condition from proj
57         if abs(2*X(i)-0.3) <= 0.25
58             u0(i) = exp(-300*(2*X(i)-0.3).^2); %Creates the ...
                    signal that we want to propagate
59         else
60             u0(i) = 0;
```

```
61          end
62      end
63  else
64      for i = 1:length(X)
65      %Condition from proj
66          if abs(2*X(i)-0.3) <= 0.25
67              u0(i) = exp(-300*(2*X(i)-0.3).^2); %Creates the ...
                    signal that we want to propagate
68
69          elseif abs(2*X(i)-0.9) <= 0.2
70              u0(i) = 1;
71
72          elseif abs(2*X(i)-1.6) <= 0.2
73              u0(i) = sqrt((1-((2*X(i)-1.6)/0.2)^2));
74
75          else
76              u0(i) = 0;
77          end
78      end
79  end
80
81
82  %Stepping in the time domain with classical RK4
83  u = zeros(length(X), length(T));
84  u(:,1) = u0;
85
86  F = @(u,t) (-A + eps*S)*u;
87
88  for k = 2:length(T)
89      %Euler fram: u(:,k) = u(:,k-1) + dt*A*u(:,k-1);
90
91      %RK4;
92  %     k1 = h*f(t(i-1),y(i-1));
93  %     k2 = h*f(t(i-1)+h/2, y(i-1)+k1/2);
94  %     k3 = h*f(t(i-1)+h/2, y(i-1)+k2/2);
95  %     k4 = h*f(t(i-1)+h, y(i-1)+k3);
96  %     y(i) = y(i-1) + (k1+2*k2+2*k3+k4)/6;
97
98  %Right now the half-time steps (t + h/2 and t + h) are ...
        disregarded in the RK method, unsure
99  %if this is the correct implementation but it seems to work ...
        quite well.
100 %Update the initial RHS of the ODE system does not depend on ...
        time and i get
101 %away with leaving it out.
102 %Could write it as F =@(u,t) A*u which doesnt depend on time
103       k1 = dt*F(u(:,k-1), T(k));
104       k2 = dt*F((u(:,k-1)+k1/2), T(k)+dt/2);
105       k3 = dt*F((u(:,k-1)+k2/2), T(k)+dt/2);
```

```
        k4 = dt*F((u(:,k-1)+k3), T(k)+dt);
        u(:,k) = u(:,k-1) + (k1+2*k2+2*k3+k4)/6;
end

e = u0-u(:,end);
e = norm(e,2)/sqrt(length(e));


% figure;
% for k=1:length(T)
%     hold off;
%     plot(X, u(:,k), '-k', 'LineWidth', 2);
%     axis([X(1), X(end), -1, 1]);
%     drawnow;
% end
% hold on
% plot(X,u0, 'linewidth',2);
```

E Appendix C

FEM solver for part C

```
a = -1; % left end point of interval
%a = 0;
b = 1; % right
%b = 2*pi;
N = 641; % number of intervals
h = (b-a)/N; % mesh size
x = a:h:b; % node coords
c = 2;
eps = 0.1;
T = 0.4; %Final time
%T = 2;
%t = h; %Timestep size = step size unsure if this is correct
TT = linspace(0,0.4,N+1); %Discretized time
dt = TT(2)-TT(1);


%Mass matrix from course book
M = MassAssembler1D(x);
M = sparse(M);
%LU used in RK4
[L,U] = lu(M);

%A matrix
A = Aassembler(x);
```

```
25 A = sparse(A);
26
27 %Stiffnessmatrix
28 %Almost same as diffusion matrix from project B
29 S = 1/h* spdiags(repmat([-2, 1, 1], N+1, 1), [0, -1, 1], ...
       N+1, N+1);
30 S(end,end) = 1/2*S(end,end);
31
32
33 %Initial condition
34 u0 = @(x) 2 -tanh((x+1/2)/(2*eps));
35 %Exact solution
36 uexact = @(x,t) 2 -tanh((x+1/2-2*t)/(2*eps));
37
38 %Beta is the U vector squared over 2
39 beta = @(u) u.*u./2;
40
41
42 %Into RK4
43 %du/dt (= xi) = M^-1 (-A*beta(U)-eps*S*U)
44 %Can be LU factorized
45 %Q = -A*beta(u)-eps*S*u;
46 %M*x = Q
47 %This can be LU factirized
48 %F = @(u,t) A*u;
49
50 %Stepping in the time domain with classical RK4
51 u = zeros(length(x), length(TT));
52 u(2:end-1,1) = u0(x(2:end-1));
53 u(1,1:end) = uexact(a,TT);
54 %u(1,1:end) = 0;
55 u(end,1:end) = uexact(b,TT);
56 %u(end,1:end) = 0;
57
58 B = @(u,t) -A*beta(u) - eps*S*u;
59 F = @(b,t) U\(L\b);
60 for k = 2:length(TT)
61     %Euler fram: u(:,k) = u(:,k-1) + dt*A*u(:,k-1);
62     %RK4;
63 %     k1 = h*f(t(i-1),y(i-1));
64 %     k2 = h*f(t(i-1)+h/2, y(i-1)+k1/2);
65 %     k3 = h*f(t(i-1)+h/2, y(i-1)+k2/2);
66 %     k4 = h*f(t(i-1)+h, y(i-1)+k3);
67 %     y(i) = y(i-1) + (k1+2*k2+2*k3+k4)/6;
68
69
70 %Compute the vector b in Mx = b
71     b = B(u(:,k-1),TT(k));
72 %     y = L\b;
```

```
%      x = U\y;B


% Compute M inv* b gives ddt and is equivalent to solving Mx ...
    = b for x
% Have to define a F so that it is the LU factorization of ...
    Mx = b and then
% solve for x


%RK4 from project B, still doesn't depend on t?
%F is the solving step in the LU factorization
% F =>
% y =L\b
% x = U\y
       k1 = dt*F(b, TT(k));
       k2 = dt*F(b+k1/2, TT(k)+dt/2);
       k3 = dt*F(b+k2/2, TT(k)+dt/2);
       k4 = dt*F(b+k3, TT(k)+dt);
       k5 = (k1+2*k2+2*k3+k4)/6;
        u(2:end-1,k) = u(2:end-1,k-1) + k5(2:end-1);
end


figure;
for k=1:length(TT)
    hold off;
    plot(x, u(:,k), '-k', 'LineWidth', 2);


    title('Proj C');
    axis([x(1), x(end), 0, 3]);
    xlabel('Spatial domain');
    drawnow;
end

%Have to reevaluate the formation of my matrices, the ...
    solution is not stable
```

Mass matrix assembler from the course book[6]

```
%For loop taken from the course book
%M.G. Larson and F. Bengzon. The Finite Element Method: ...
    Theory, Implementation, and Practice
%s. 17

```

[6]M.G. Larson and F. Bengzon. The Finite Element Method: Theory, Implementation, and Practice s. 17

```
function M = MassAssembler1D(x)
n = length(x)-1; % number of subintervals
M = zeros(n+1,n+1); % allocate mass matrix
for i = 1:n % loop over subintervals
h = x(i+1) - x(i); % interval length
M(i,i) = M(i,i) + h/3; % add h/3 to M(i,i)
M(i,i+1) = M(i,i+1) + h/6;
M(i+1,i) = M(i+1,i) + h/6;
M(i+1,i+1) = M(i+1,i+1) + h/3;
end


%Add the first value
M(1,1) = 2*M(1,1);
```

Stiffness matrix assembler

```
function A = Aassembler(x)
%Similar to advection matrix in Project B

N = length(x)-1;
A = zeros(N+1,N+1);
for i = 1:N
    h = x(i+1)-x(i); %Length of element, not neccessary to ...
        use in proj C
    n = [i i+1]; %node
    A(n,n) = A(n,n) + [-1,1;-1,1]; %Assemble element
end

A(1,1) = 0; %Not sure about boundaries
A(end,end) = 0;
```

F Example: large matrix

$$
Q = \begin{bmatrix}
q_{1,1} & q_{1,2} & \cdots & q_{1,s} & & & & & & & & \\
q_{2,1} & \ddots & & q_{2,s} & & & & & & & & \\
\vdots & & & \vdots & q_{\frac{p}{2}+1} & & & & & & & \\
\vdots & & & \vdots & \vdots & & & & & & & \\
q_{s,1} & q_{s,2} & \cdots & q_{s,s} & q_1 & \ddots & \ddots & & & & & \\
 & & q_{1-\frac{p}{2}} & \cdots & q_0 & q_1 & \cdots & q_{\frac{p}{2}+1} & & & & \\
 & & & \ddots & & \ddots & & & \ddots & & & \\
 & & & & & q_{1-\frac{p}{2}} & \cdots & q_0 & q_1 \cdots & q_{\frac{p}{2}+1} & & \\
 & & & & & \ddots & & \ddots & q_{s,s} & q_{s-1,s} & \cdots & q_{1,s} \\
 & & & & & & \ddots & & q_{s,s-1} & \ddots & & q_{1,s-1} \\
 & & & & & & & q_{1-\frac{p}{2}} & \vdots & & & \vdots \\
 & & & & & & & & \vdots & & & \vdots \\
 & & & & & & & & q_{s,1} & \cdots & & q_{1,1}
\end{bmatrix}
$$

References