Uppsala University


Scientific Computing for PDE

1TD354 12003 HT2022

# Project 1 (FEM and linear systems) - report

*Author:*
Csongor Horváth

November 9, 2022

# Part A

Let's start by creating the weak formulation of the problem. Let's multiply by a sufficiently smooth function $v(x)$ and got the weak form after integration

$$0 = \int_a^b \frac{\partial u_h}{\partial t} v dx + \int_a^b \frac{1}{2} \frac{\partial u_h^2}{\partial x} v dx - \epsilon \int_a^b \frac{\partial^2 u_h}{\partial x \partial x} v dx =$$

$$= \int_a^b \frac{\partial u_h}{\partial t} v dx + \int_a^b \frac{1}{2} \frac{\partial u_h^2}{\partial x} v dx + \epsilon \int_a^b \frac{\partial u_h}{\partial x} \frac{\partial v}{\partial x} dx - \left[ v \frac{\partial u_h}{\partial x} \right]_a^b$$

We want to find $u_h \in V_g = \{v : ||v||^2 < \infty,\ ||v'||^2 < \infty,\ v(a,t) = g_a(t), v(b,t) = g_b(t) \forall t \in (0,T]\}$. For which the above holds for all $v \in V_0 = \{v : ||v||^2 < \infty,\ ||v'||^2 < \infty,\ v(a,t) = 0, v(b,t) = 0 \forall t \in (0,T]\}$. So we got the week formulation with $V_g$ as trial space and $V_0$ as test space:

$$0 \int_a^b \frac{\partial u_h}{\partial t} v dx + \int_a^b \frac{1}{2} \frac{\partial u_h^2}{\partial x} v dx + \epsilon \int_a^b \frac{\partial u_h}{\partial x} \frac{\partial v}{\partial x} dx = 0 \quad \forall v \in V_0$$

Now we want to form the finite element formulation by discretizing the equation in space. So we can look at the time as fixed and we want to achive the space discretization by searching only in finite dimensional subspace of the above fields in a given time $t$. Let's be $I = \{I_1, I_2, \ldots, I_N\}$ and equidistance grid of the $I = [a,b]$ intervall with $I_j = [x_{j-1}, x_j]\ j = 1, 2, \ldots, N$. Here the mesh size is $h = \frac{1}{N}$. So we want to solv the above equations in the continouse picewise linear subspaces of the above spaces. So now we searching $u_h(x,t) \in V_{h,g} = \{v(x,t) \in \mathcal{C}([a,b] \times (0,T]) :\ v|_{I_i}$ linear in $I_i\ i = 1, 2, \ldots N,\quad ||v||^2 < \infty,\ ||v'||^2 < \infty,\ v(a,t) = g_a(t), v(b,t) = g_b(t)\ \forall t \in (0,T]\} \subset V_g|$ which satisfies the equation for all $v(x,t) \in V_{h,0}|_t = \{v(x,t) \in \mathcal{C}([a,b] \times (0,T]) :\ v(x,t)|_{I_i}$ linear in $I_i\ i = 1, 2, \ldots N,\ \ ||v||^2 < \infty,\ ||v'||^2 < \infty,\ v(a,t) = 0, v(b,t) = 0\ \forall t \in (0,T]\} \subset V_0$

Now to get the discretization in space we use the base functions $\varphi_j = \frac{x - x_{j-1}}{h}$ in $x \in [x_{j-1}, x_j]$, $\varphi_j = \frac{x - x_{j+1}}{-h}$ in $x \in [x_j, x_{j+1}]$, 0 otherwise. Note $span(\varphi_j)_0^N \supset V_{h,g}|_t, V_{h,0}|_t$.
So in general we can search $u_h$ in a form: $u_h(x,t) = \sum_{j=0}^n U_j(t) \varphi_j$. And since $v \in V_{0,h}$, so it is would be enough if we write the equations for all base

function $\varphi_j$, $j = 1, \dots N-1$, since we know that $v(a) = v(b) = 0$ and we also $u_h(a,t) = g_a(t)$ and $u_h(b,t) = g_b(t)$ is known.
*Note: we use $N+1 \times N+1$ matrixes in the implementation so we use the below formulation and to hold BC we set the boundary data to the given value after each time step.*
So we get from here the Galerkin formulation for the problem:
$\int_a^b \left( \sum_{i=0}^N \frac{\partial U_i}{\partial t} \varphi_i \varphi_j dx \right) + \int_a^b \left( \sum_{i=0}^N \frac{1}{2} \frac{\partial U_i^2 \varphi_i}{\partial x} \varphi_j dx \right) + \int_a^b \left( \epsilon \sum_{i=0}^N \frac{\partial U_i \varphi_i}{\partial x} \frac{\partial \varphi_j}{\partial x} dx \right) =$
$\sum_{i=0}^N \left( \frac{\partial U_i}{\partial t} \int_a^b \varphi_i \varphi_j dx \right) + \sum_{i=0}^N \left( \frac{1}{2} U_i^2 \int_a^b \frac{\partial \varphi_i}{\partial x} \varphi_j dx \right) + \sum_{i=0}^N \left( \epsilon U_i \int_a^b \frac{\partial \varphi_i}{\partial x} \frac{\partial \varphi_j}{\partial x} dx \right) =$
$0, \ \forall j = 1, \dots n$
We can write this to matrix form with the given matrices:

$$\mathbf{M} \frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = -\mathbf{A}\beta(\boldsymbol{U}) - \varepsilon \mathbf{S}\boldsymbol{U},$$

where $\beta(U)$ is the elementwise $\frac{1}{2}x^2$ function and the matrices are:

$$\mathbf{M}_{ij} := \int_I \varphi_j \varphi_i \, \mathrm{d}x, \quad \mathbf{A}_{ij} := \int_I \partial_x \varphi_i \varphi_j \, \mathrm{d}x, \quad \mathbf{S}_{ij} := \int_I \partial_x \varphi_j \partial_x \varphi_i \, \mathrm{d}x.$$

The above equation should hold for every time point in $(0, T]$. So we determinated an ODE which we should integrate in time by RK4.
So let's calculate the matrices **M,A,S** what we use. Note that the mesh-size is $h = (b-a)/N$.
For the **M** the values in the main diagonal will be equal to $2 \cdot \int_0^h \varphi_1 \varphi_1 dx = 2 \cdot \int_0^h x^2/h^2 dx = \frac{2h}{3}$ expect in the corner elements $m_{0,0}, m_{N,N}$. And on the not main diagonal the values will be equal to $\int_0^h \varphi_1 \varphi_0 dx = \int_0^h \frac{x}{h} \frac{h-x}{h} dx = \frac{h}{6}$.
For the matrix **A** we got values equal to $\int_0^h \frac{x}{h^2} dx + \int_0^h \frac{x-h}{h^2} dx = 0$ expect in the corner elements $m_{0,0} = -0.5, \ m_{N,N} = 0.5$. And in the subdiagonals we got values equal to $\int_0^h \frac{h-x}{h^2} dx = +0.5$ in the upper part and $\int_0^h \frac{x}{-h^2} dx = -0.5$ in the lower part.
And for the matrix **S** we got the derivatives and we know from Lecture 8 that here the main diagonal is $2/h$ and the subdiagonals are $-1/h$.

To make the implementation easier lets multiply by $\mathbf{M^{-1}}$. So we will run the RK4 for $\frac{\mathrm{d}\boldsymbol{U}}{\mathrm{d}t} = -\mathbf{M^{-1}A}\beta(\boldsymbol{U}) - \varepsilon \mathbf{M^{-1}S}\boldsymbol{U}$. Here $U$ is the timedependent unknown. So set $U$ to the initial value. Let the timestep be $k$. Than $U(t+k) = U(t) + \frac{h}{6}(k_1 + k_2 + k_3 + k_4)$, where $k_1 = -M^{-1}A\beta(U(t)) - \varepsilon M^{-1} S U(t)$,

$k_2 = -M^{-1}A\beta(U(t) + \frac{h}{2}k_1) - \varepsilon M^{-1}S\left(U(t) + \frac{h}{2}k_1\right)$, $k_3 = -M^{-1}A\beta(U(t) + \frac{h}{2}k_2) - \varepsilon M^{-1}S\left(U(t) + \frac{h}{2}k_2\right)$, $k_4 = -M^{-1}A\beta(U(t) + hk_3) - \varepsilon M^{-1}S\left(U(t) + hk_3\right)$
Now after ech time step we set $U_0(t) = g_a(t)$, and $U_N(t) = g_b(t)$ to make sure the BC holds.
So this includes the theory. Now we only need to implement it and run it with the given parameters while ploting the neccesery datas.
*Note for all implementation: To plot we dont need to do nothing extra, just plot* $\mathbf{U}$ *values and connect with line due to the* $\varphi_i$ *definitions.*

## 0.1 Problem A.1

For this implementation we followed the above theoretical part and use the given initial data and BC. We present the result here generated with the code in *pictures.m* using the *myproject.m* module.
*Note: We use $k = 2h^2$ as a time-step in the RK method. Where $k$ is the time-step and $h$ is the mesh size. It is small enough in theese case to gain a stable simulation.* As we can see as expected, as we increas $N$ the error is getting smaller.

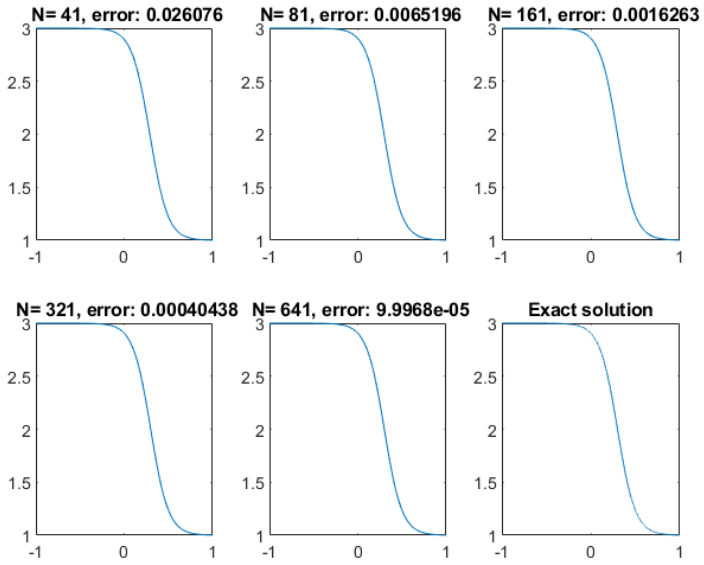


Figure 1: Results after $t = 0.4$ for $N = [41, 81, 161, 321, 641]$ and the corresponding errors

So instead of the previously used error estimation let's stick to the $L_2$ norm estimation. So we estimate the values $\sqrt{\int_a^b |u_h - ex|^2 dx}$, where $ex$ is the exact solution. (Note: we could use other error estimates as we did prior. e.g. another error estimate could be $\int_a^b |u_h - ex| dx$, but this would lead to similar result). Now since we can't compute these integrals, we will use an integral approximation always using intervall size equal to the mesh size in the given example and using the known $(x_i)$ points in the intervall. And

to make our code even more simply we will use $\sqrt{h\sum_{i=0}^{n}(u_h(x_i) - ex(x_i))^2}$, which is formally not the same as the above written integral approximation, for example because we use one more values, than we should from the used equidistance intervalls. But this estimation will be good for us, because if we think about this estimation as $h$ long intervalls around $x_i$ $i = 1, \ldots, N-1$ and two $h/2$ long intervalls aound $x_0, x_N$, then we would get the same value, since $u_h(x_0) = ex(x_0)$ and $u_h(x_N) = ex(x_N)$. So it really is a good integral approximation to use.
By calculating these values we will get the results shown in the figures (2, 3). In the figure 2 we can see the result plotted normally with a fitted line and a fitted second order curve. It suggest, since the 2nd order curve is approximating really nice, that it should be 2nd order convergence rate. To make sure we will use the suggested log-log figure. It is usefull, because in a log log figure a connection $y = ax^k$ seems like a line, because then $log(y) = klog(x) + log(a)$. And as we see in figure 3, we get almost a perfect line. So now to conclude the order of convergence we wanna find out $k$ in the above equation $y = ax^k$. From the log form it should mean the slope of the line in the log-log figure. To get this we fit a line to the logarithmic datapoints and look the sloop of that. As it is displayed on the image it is almost exactly 2. So in accordance with the theory ([1] 3.8.1 for case $p = 1$) our results shows that the convergence rate is second order, as we guesed it from the figure 2. This conclude our analysis of the error.

*Note: If we useing other kind of norm it may be lead to other rate of convergance as we can see it in [2] Theorem 5.2.2*

# References


[1] Ioannis Koutromanos, Fundamentals of Finite Element Analysis. (2018)

[2] Alan Demlow, Lecture notes of Numerical Analysis of Differential Equations in Cornell University. *Chapter 5* (2002) link: http://pi.math.cornell.edu/~demlow/425/chap5.pdf


(*Note: Due to an error in my implementation last time I used the error* $\sum_{i=0}^{N}(ex(x_i) - u(x_i))$, *where lack of absolute value wasn't a problem luckily, because now I checked and it is almost everywhere* $\geq 0$. *So it is eqvivalent to*

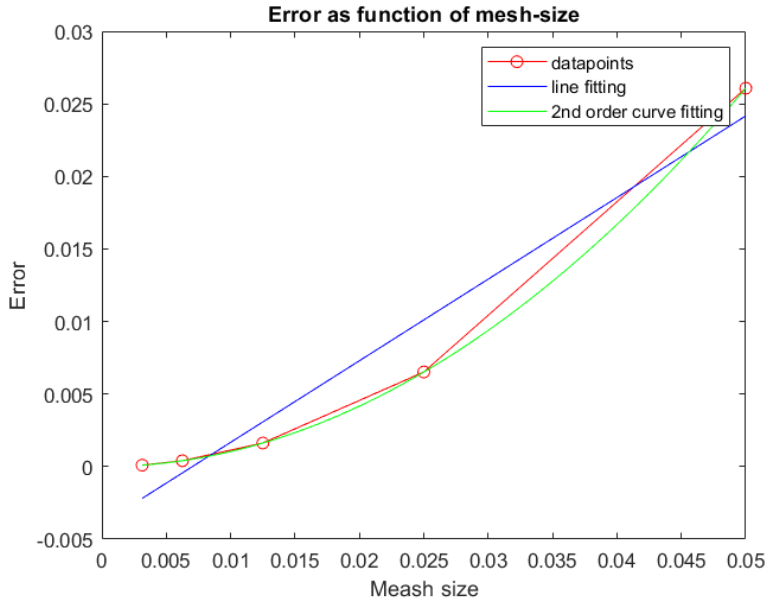


Figure 2: Error ploted normaly with fitted firs and second order polynomial

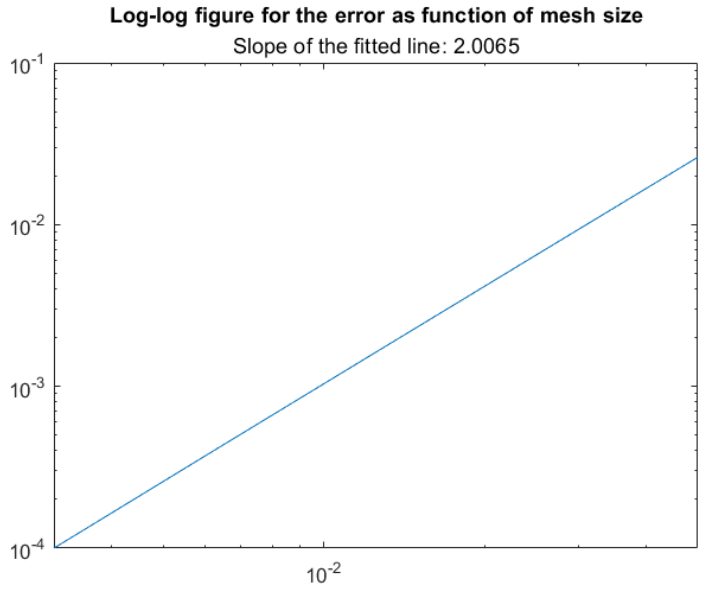


Figure 3: Error ploted in a log-log figure

$\sum_{i=0}^{N} |ex(x_i) - u(x_i)|$. *So indeed getting a line in the previous version also suggest a second order convergence rate.* )

## 0.2 Problem A.2

Here as in figure 4 we can see the figures generated with the code *myproject_2.m*. Here the model become unstable. It is logical if we consider, where the resonance starts it should be basically a vertical line and working with tridiagonal matrixes it results unsurprisingly anomaly in the system in the case, when epsilon is around or smaller than the steps ($h$) in the model.

*Note: We use $k = 0.2 \cdot h^2$ as a time-step in the RK method. Where $k$ is the time-step and $h$ is the mesh size. It was made smaller to make the simulations more stable.*

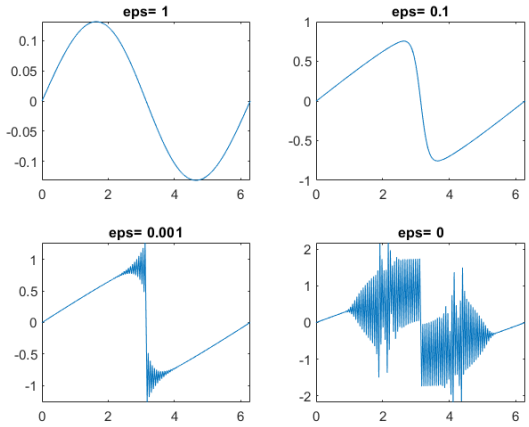


Figure 4: Numerical data

## 0.3 Problem A.3

Here as in figure 5 we can see the figure generated with the code *myproject_3.m*. Here the model become stable again, because the used $\epsilon$ values are small compared to the step size ($h$).
*Note: We use $k = h^2$ as a time-step in the RK method. Where $k$ is the time-step and $h$ is the mesh size. As we can se it is small enough in theese case to gain a stable simulation.*

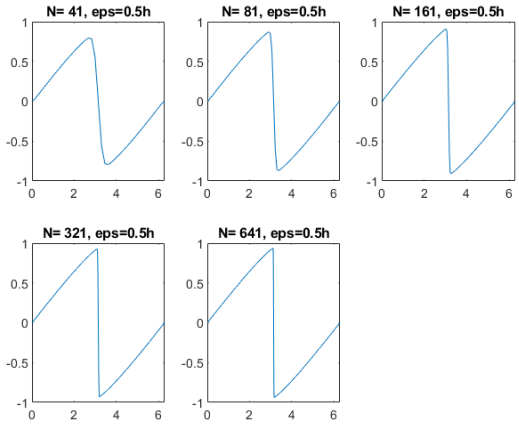


Figure 5: Numerical data

# Part B

In this following problems we are examining the performance (iteration and run-time) of 4 algorithms for solving system of linear equations implemented in MATLAB: Jacobi method (jacobi.m), Gauss-Seidel method (gs.m), LU decomposition (myownLU.m) and Conjugate gradient method (CG.m). The timing was done using the MATLAB script: main.m, where the scripts run the timing with all the required input except in part B3 were Gaus-Seidel is skipped for $\alpha = 0.00001$, because it would take somethings like 11 hour as explained later. Also, the runtime is around 20 min of the main function in this form.

*Note: myownLU is implemented without TOL, the project description wasn't consistent on this (somewhere says all function should have the same input), but it would be an unused input if given, so it doesn't matter.*

*Note: All implementation start the iteration with the vector* $\mathbf{1} = (1, \ldots, 1)^T$.

*Note to the implementations:* For the Jacobi and Gaus-Seidel in the submited form using the matrix form, because when it was written on a lower level with for loop iteration it was much slower, because the matrix multiplication is implemented in a fast way in MATLAB. So for this end these algorithm compute an iteration matrix and vector and work with the form $x_{n+1} = b - Mx_n$, where in the Jacobi iteration $M = (D + L)^{-1}U$, $b = (D + L)^{-1}b$ and in Gaus-Seidel $M = D^{-1}(L + U)$, $b = D^{-1}b$.

The myownLU implementation working with the $L, U, P$ decomposition of the matrix, because it is always exists, so in this way the myownLU works for all the case. Otherwise it wouldn't, because not all matrix have LU decomposition. In cases where a matrix has $LU$ decomposition it is the same, otherwise the input $A$ matrix is written $A = P'LU$, where $P'$ is not the identity if the LU decomposition don't exist. But with this information we can make the algorithm work for all cases whit multiplying $Ax = b$ by $P'^{-1}$ and using the basic $LU$ decomposition.

*Note:* The algorithms was tested to generate correct results in the belov presented cases where we presented results. This was used to make sure a method is implemented correctly and that the method is sufficient for the given matrix. (e.g: For the conjugate gradient method we can't say in general when it will reach the given convergence and when not if the matrix is not a symmetric positive definite)

# Problem B1

In this problem we are solving a small 4x4 system of lineare equations

$$\mathbf{A} = \begin{bmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 6 \\ 25 \\ -11 \\ 15 \end{bmatrix} \tag{1}$$

| **Method** | **Iteration** | **Time** |
|---|---|---|
| Jacobi | 11 | 0.0053973 |
| Gauss-Seidel | 5 | 0.0005 |
| CG | 5 | 0.000274 |
| myownLU | | 0.0071689 |
| $A\backslash b$ | | 0.000148 |
| Matlab LU | | 0.0003738 |

Table 1: Problem B1

We can observe that the MATLAB backslash function is the fastest of the selection of algorithms. This is because the backslash function is not only one algorithm but instead work by first examining the matrix and select method that is best suited for the problem. The MATLAB functions are also better designed than the implementation of algorithms we constructed using the hardware much more efficiently.
Otherwise as we can se the Gaus-Seidel took much less iteration as the Jacobi method as expected, since gs is in a way an improved version of the Jacobi. Also the conjugate-gradient method is also seemingly very efficient in this case. And it is garanteed to work, because $A$ is a symmetric positive definite matrix.
*Note: we can use all the methods safely here, since A is a small diagonally dominant matrix, so all the algorithm should work fine for it.*

# Problem B2

In this problem we are examining the run-time and number of iteration it takes to solve a randomly generated system of equations. A and b were generated with random numbers between 0 and 1. To the diagonal of A is extra "weight" added by changing the parameter w that adds w to all the diagonal entries of A. The test was then conducted for different sizes N of the systems 100,500 and 1000 and with different weights: 1,5,10 and 100.

The convergence condition for the Jacobi and Gaus-Seidel iterative methods is when the spectral radius of the iteration matrix is less than 1 to guarantee convergence. Otherwise it is almost certainly not convergent. The only case when it still convergent if there is only one eigenvalue which is greater than one and it is orthogonal to the initial solution vector.

The spectral radius of a square matrix is the maximum of the absolute values of the eigenvalues of the matrix. Spectral radius of the Jacobi iteration matrix nad Gauss-Seidel iteration matrix:

$$\rho(D^{-1}(L+U)) < 1, \qquad \rho((L+D)^{-1}U) < 1 \tag{2}$$

Another sufficient condition for convergence is that the matrix A is strictly diagonally dominant. Diagonal row dominant means:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \tag{3}$$

| **Method** | **Iterations** | **100** | **500** | **1000** |
|---|---|---|---|---|
| **Jacobi** | NaN/NaN/NaN | NaN | NaN | NaN |
| **Gauss-Seidel** | NaN/NaN/NaN | NaN | NaN | NaN |
| **CG** | NaN/NaN/NaN | NaN | NaN | NaN |
| **myownLU** | | 0.001808 | 0.0159877 | 0.146492 |
| $A\backslash b$ | | 0.0005598 | 0.016319 | 0.0371192 |
| **MATLAB LU** | | 0.0005978 | 0.0069524 | 0.0543279 |

Table 2: Problem B2, w = 1

| **Method** | **Iterations** | **100** | **500** | **1000** |
|---|---|---|---|---|
| **Jacobi** | NaN/NaN/NaN | NaN | NaN | NaN |
| **Gauss-Seidel** | 93/NaN/NaN | 0.0029686 | NaN | NaN |
| **CG** | 67/NaN/NaN | 0.0018423 | NaN | NaN |
| **myownLU** | | 0.0007919 | 0.0138657 | 0.104055 |
| $A\backslash b$ | | 0.000591 | 0.0049675 | 0.0394745 |
| **MATLAB LU** | | 0.0006555 | 0.0042364 | 0.044828 |

Table 3: Problem B2, w = 5

| **Method** | **Iterations** | **100** | **500** | **1000** |
|---|---|---|---|---|
| **Jacobi** | NaN/NaN/NaN | NaN | NaN | NaN |
| **Gauss-Seidel** | 41/521/1336 | 0.0030233 | 0.190644 | 4.37544 |
| **CG** | 15/186 /NaN | 0.0007266 | 0.151463 | NaN |
| **myownLU** | | 0.0007708 | 0.0135646 | 0.05998 |
| $A\backslash b$ | | 0.0003634 | 0.0047279 | 0.0441139 |
| **MATLAB LU** | | 0.0001968 | 0.0064575 | 0.03725 |

Table 4: Problem B2, w = 10

| **Method** | **Iterations** | **100** | **500** | **1000** |
|---|---|---|---|---|
| **Jacobi** | 21/NaN/NaN | 0.0009957 | NaN | NaN |
| **Gauss-Seidel** | 8/21/44 | 0.001606 | 0.101273 | 1.42115 |
| **CG** | 5/7/8 | 0.0009666 | 0.0120854 | 0.0417331 |
| **myownLU** | | 0.001924 | 0.014731 | 0.0737177 |
| $A\backslash b$ | | 0.0003042 | 0.0040942 | 0.0432728 |
| **MATLAB LU** | | 0.0006758 | 0.0046803 | 0.0376683 |

Table 5: Problem B2, w = 100

| **N** | $\rho$ (w = 1) | $\rho$ (w = 5) | $\rho$ (w = 10) | $\rho$ (w = 100) |
|---|---|---|---|---|
| 100 | 33.42 | 9.15 | 4.76 | 0.49 |
| 500 | 171.81 | 45.53 | 23.26 | 2.48 |
| 1000 | 346.61 | 91.06 | 47.63 | 47.63 |

Table 6: Spectral radius of iterative matrix for Jacobi iteration

| **N** | $\rho$ (w = 1) | $\rho$ (w = 5) | $\rho$ (w = 10) | $\rho$ (w = 100) |
|---|---|---|---|---|
| 100 | 16.7839 | 0.8407 | 0.7059 | 0.1062 |
| 500 | 32174 | 1.6845 | 0.9781 | 0.4657 |
| 1000 | $1.7785 \cdot 10^{10}$ | 4.0094 | 0.9899 | 0.6993 |

Table 7: Spectral radius of iterative matrix for Gaus-Seidel iteration

When the spectral radius of the iterative matrix becomes less than 1. Or when the matrix becomes Diagonal dominant row wise as at w=100 and N=100 we can observe that the Jacobi methods works, this is because it guaranteed to solve systems with spectral radius $< 1$. On other cases we examined the spectral radius of the iterative matrix and skipped the method if it was greater than 1 both for the Jacobi and Gauss-Seidel iteration (see spectral radius on Table 6 and Taable 7). And as we can see the Jacobi method is not very useful in this case when using random matrixes.

Gauss-Seidel method is guaranteed to converge when the original matrix is Diagonally dominant or symmetric positive definite or the spectral radius of the iterative matrix is less than one. We can therefore not be certain that this method works for any other cases than N=100 and w = 100. Even though it converges in other cases we can see that the number of iterations could be high. So the runtime may be high and the convergent is also uncertain in general, so we looked at the spectral radius of the iteration matrix to decide it. And the conclusion is that this is also not a too useful method in this case with the random matrixes even if it is more applicable than Jacobi iteration.

For the Conjugate gradient method to guaranteed to work the matrix should be symmetric pozitive definite matrix. It is not the case in this part. But in some cases the method still converge. Mostly when it is closer to diag dominant matrixes e.g. $N = 100, w = 100$ the matrix is close to $100 \times id$ with some very small noise, where $id$ is the given size identity matrix. So in this case it is logical that the method still produces good solution. But in general we can't garantee convergence in this case so it is also not a usefull method here. But when it is applicable it is producing convergence quicly both in time and in number of iteration.
The myownLU works fine due to the general implementation with LUP factorization. And the build in methods are also works just fine with being the

most efficient in time.

So the data that we have matches our expectations in nr of iterations the conjugate gradient is the fastest and jacobi is slowest when exists.
Also the backslash operator is the fastest and MATLAB Lu is faster than myownLU.

# Problem B3

In this section is a large sparse matrix solved using the methods from earlier. In this case the matrix is symmetrix positive definite and diagonally dominant, so all iterative method should work just fine in theory. In application the gauss-seidel becomes very slow due to the nonespare iteration matrix.

For the other two methods (Jacobi and Gauss-Seidel) it was also clear that the condition number is correlates with the speed of convergence, making it unpractical to use them for the ill conditioned systems e.g. $\alpha = 0.001$ and 0.00001.

- for $\alpha = 1$ cond(A) $\approx 5$
- for $\alpha = 0.1$ cond(A) $\approx 41$
- for $\alpha = 0.001$ cond(A) $\approx 4000$
- for $\alpha = 0.00001$ cond(A) $\approx 40000$

| **Method** | **Iteration** | a = 1 | a = 0.1 | a = 0.001 | a = 0.0001 |
|---|---|---|---|---|---|
| Jacobi | 37/298/29425/2.89e+06 | 0.511056 | 0.496114 | 2.37093 | 340.283 |
| Gauss-Seidel | 23/160/15409/NaN | 2.2089 | 1.89242 | 598.832 | |
| CG | 16/50/547/5669 | 0.0080169 | 0.026066 | 0.286805 | 3.24823 |
| myownLU | | 1.06125 | 1.0725 | 1.22775 | 1.22256 |
| Ab | | 0.0003147 | 0.0003702 | 0.0064186 | 0.0002614 |
| Matlab LU | | 0.0017044 | 0.001678 | 0.0043749 | 0.0015742 |

Table 8: Result Problem B3

Since we working with a large spare matrix the Jacobi method become much faster than the Gaus-Seidel in runtime, because the iteration matrix

we use in the implementation is also a spare matrix in the Jacobi case. It is basically just to sub diagonal. But in the Gaus-Seaidel case the iteration matrix consist of a lot of not null element, which makes the runtime slow even if it needs less iteration than the jacobi method. So we didn't even run the case for $\alpha = 0.00001$ for the gauss-seidel, because in our estimation it would take around 11 hour. (the iteration takes 100steps around 4sec)
As we can see the bad conditioned case only caused problem for the iterative methods, because the number of iteration increased drastically as we decreased $\alpha$. The non iterative methods runtime mostly stayed the same independently from $\alpha$.

So here we can make conclusions for the iterative methods required number of iteration, since here we have a lot of data about htem. The runtimes were explained above, and in the number of iteration we got the expected results. For a given problem Jacobi is the slowest and the conjugate gradient is the fastest. And the number of iteration grow as we decreased the extra weight in the diagonal as expected.

The result once again also highlight that even though Conjugate gradient method is a good method for this kind of problem and our LU implementation is also effective the MATLAB functions are adaptive and much more efficiently constructed.

## Overall conclusions for the methods

As we could see even if in many cases the Jacobi, Gaus-Seidel and Conjugate gradient methods are not producing data due to lack of convergence. It is a clear observation that in the number of iteration the data is consistent with the theory. So the Jacobi is the slowest takes the most iteration and the conjugate gradient is the fastest, takes the least iteratin in all the observed case. So the iterative methods are only efficient in some cases and they are very unuseful if working we wnat to work with random matrixes without enough additional weight on the diagonals.
Our implementation of LU works just fine in every cases due to the usage of LUP decomposition. But in general the LU would fail as well as the other methods in many cases due to lack of $LU$ decomposition.
So we would recommend using the build in backslash operator, since it is clearly the most stable and fastest way of solving linear equations.

# Appendix -our matlab codes

## Code: *jacobi.m*

```
function x = jacobi(A,b, TOL)
D= sparse(diag(1./diag(A)));
LU= sparse(A- diag(diag(A)));
M=sparse(D*(LU));
b1=D * b;
iter = 0;
err = 0.1+TOL;
n = length(b);
x = ones(n,1);
while err>TOL
    iter = iter + 1;
    x0=x;
    x = b1-M*x0;
    err = norm(x-x0);
end
fprintf ( 'jacobi took %g iteration \n ' , iter );
end
```

### Code: *gs.m*

```
function x = gs(A,b, TOL)
L= inv(sparse(tril(A)));
U= sparse(triu(A,1));
M = sparse(L*U);
b1=L * b;
iter = 0;
err = 0.1+TOL;
n = length(b);
x = ones(n,1);
while err>TOL
    iter = iter + 1;
    x0=x;
    x = b1- M*x0;
    err = norm(x-x0);
end
fprintf('gs took %g iteration \n', iter);
end
```

## Code: *cg.m*

```
function x = cg(A,b,TOL)

    x = ones(length(b),1);
    r = b - A*x;
    p_old = r;
    x_old = zeros(1,length(b))';
    iter_CG = 0;


    iter_CG = iter_CG +1;
        alpha = (r'*r)/(r'*A*r);
        x = x + alpha*r;
        r = b-A*x;


    while norm(x-x_old) > TOL
        iter_CG = iter_CG +1;
        x_old = x;
        gamma = -(p_old'*A*r)/(p_old'*A*p_old);
        p_new = r + gamma*p_old;
        alpha = (p_new'*r)/(p_new'*A*p_new);
        x = x + alpha*p_new;
        r = r - alpha*A*p_new;
        p_old = p_new;
    end
    fprintf('cg took %g iteration \n', iter_CG);
%}
end
```

## Code: *myownLU.m*

```
function x = myownLU(A,b)
[L,U,P]=lu(A);
n=length(b);
b = inv(P')*b;
%Ly=b
y=zeros(n,1);
y(1)=b(1);
for i =2:n
    y(i) = b(i)-sum(L(i,1:i-1)*y(1:i-1));
end

%Ux=y
x=zeros(n,1);
x(n)=y(n)/U(n,n);
for i =(n-1):-1:1
    x(i) = (y(i)-sum(U(i,i+1:n)*x(i+1:n)))/U(i,i);
end
```

## Code: *main.m*

```
clc; clear;

%%%%%%%%% Problem B.1 %%%%%%%%%%
     fprintf('Runtime data for Problem B.1: \n\n');
      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 ;
     x1 = A \ b ;
     fprintf ( 'Backslash took %g sec \n \n ' , toc );
     tic ;
     x2 = jacobi (A ,b , TOL );
     fprintf ( 'Jacobi took %g sec \n \n' , toc );
     tic ;
     x3 = gs (A ,b , TOL );
     fprintf ( 'GS took %g sec \n \n' , toc );
     tic ;
     x4 = cg (A ,b , TOL );
     fprintf ( 'CG took %g sec \n \n' , toc );
     tic ;
     x5 = myownLU (A , b );
     fprintf ( 'myownLU took %g sec \n \n' , toc );
     tic ;
     [L , U ] = lu( A );
     y = L \ b ;
     x6 = U \ y ;
     fprintf ( 'Matlab LU took %g sec \n \n' , toc );
%}

%%%%%%%%% Problem B.2 %%%%%%%%%%
 TOL = 1e-5;
```

```
37
38 fprintf('\n#######################\n B.2\n
     #######################\n ')
39  for N = [100, 500, 1000]  % the size of the linear
       system
40      for w = [ 1, 5, 10, 100] % the diagonal weight
41          A = rand (N) + diag (w* ones (N ,1));
42          b = rand (N ,1);
43          fprintf ( '\n###################\n For N = %f, w
               = %f \n###################\n \n ' , N, w );
44
45          tic ;
46          x1 = A \ b ;
47          fprintf ( 'Backslash took % g sec \n \n ' , toc
                );
48
49
50          %####
51          D= diag(1./diag(A));
52          LU= A- diag(diag(A));
53          M=D*LU;
54          if max(abs(eig(M)))>=1
55              disp('Jacobi most likely not being
                    convergent');
56              disp('Skipping Jacobi');
57              disp('');
58              disp('');
59          else
60              tic ;
61              x2 = jacobi (A ,b , TOL );
62              fprintf ( 'Jacobi took % g sec \n \n' , toc
                    );
63
64          end
65          % ###
66
67
68          %####
```

```
69          L= inv(tril(A)) ;
70          U= triu(A,1);
71          M = L*U;
72          if max(abs(eig(M)))>=1
73              disp('GS likely not being convergent');
74              disp('Skipping GS');
75              disp('');
76              disp('');
77          else
78              tic;
79              x3 = gs(A ,b , TOL );
80              fprintf ( 'GS took % g sec \n \n' , toc );
81          end
82          % ###
83
84          tic ;
85          x4 = cg (A ,b , TOL );
86          res = toc;
87          if norm(x1-x4)>1 || isnan(norm(x1-x4))
88              fprintf('Skipping CG, because it wasnot
                    stable numerically and due to singular
                    values the method collapsed\n\n');
89          else
90              fprintf ( 'CG took % g sec \n \n' ,  res);
91          end
92
93          tic ;
94          x5 = myownLU (A , b );
95          fprintf ( 'myownLU took % g sec \n \n' , toc );
96
97          tic ;
98          [L , U ] = lu( A );
99          y = L \ b ;
100         x6 = U \ y ;
101         fprintf ( 'Matlab LU took % g sec \n \n' , toc
                );
102
103
```

```
104     end
105 end
106
107
108 %%%%%%%%% Problem B.3 %%%%%%%%%%
109 TOL = 1e-5;
110
111 fprintf('\n######################\n B.3\n
    ######################\n ')
112
113     for alpha = [1 0.1 0.001 0.00001]
114         [A,b] = Matrix(alpha);
115
116         fprintf ( '\n###################\n For alpha = %f
                \n###################\n \n ' , alpha );
117
118         disp(condest(A))
119         disp("");
120         tic ;
121         x1 = A \ b ;
122         fprintf ( 'Backslash took % g sec \n \n ' , toc
                );
123
124
125         tic ;
126         x2 = jacobi (A ,b , TOL );
127         fprintf ( 'Jacobi took % g sec \n \n' , toc );
128
129
130
131         tic;
132         if alpha >0.0001
133             x3 = gs(A ,b , TOL );
134             fprintf ( 'GS took % g sec \n \n' , toc );
135         end
136
137         tic ;
138         x4 = cg (A ,b , TOL );
```

```
139             res = toc;
140             if norm(x1-x4)>1
141                 fprintf('Skipping CG, because it wasnot
                        stable numerically and due to singular
                        values the method collapsed');
142             else
143                 fprintf ( 'CG took %g sec \n \n' ,  res);
144                 disp(norm(x1-x4));
145             end
146
147             tic ;
148             x5 = myownLU (A , b );
149             fprintf ( 'myownLU took %g sec \n \n' , toc );
150
151             tic ;
152             [L , U ] = lu( A );
153             y = L \ b ;
154             x6 = U \ y ;
155             fprintf ( 'Matlab LU took %g sec \n \n' , toc
                    );
156
157
158         end
159 %}
160
161 function [Q,b] = Matrix(alpha)
162     m = 10000;
163     Q = eye(m);
164     N = m;
165     for i =1:m-1
166         Q(i,i) = 2 + alpha;
167         Q(i+1,i) = -1;
168         Q(i,i+1) = -1;
169     end
170     Q(m,m) = 2 + alpha;
171     b = rand(N,1);
172     Q = sparse(Q);
173 end
```

```
174 %}
```

## Code: *pictures.m*

```
clear();

i=1;
h=[];
e=[];
figure(2)


for N = [ 41, 81, 161, 321, 641]
    u=myproject(N);
    x=-1:(2/(N-1)):1;
    ex=2-tanh((x+0.5-2*0.4)/(2*0.1));
    %err=sum(abs(ex'-u),"all")*(2/(N-1));
    err= sqrt((ex'-u)'*(ex'-u)*(2/(N-1)));
    h(i)=(2/(N-1));
    disp(err)
    e(i)=err;
    figure(2);
    subplot(2,3,i);
    plot(x,u )
    disp(err)
    title(" N= "+num2str(N)+", error: "+num2str(err))
    i = i+1;
end

x=-1:0.0001:1;
u=2-tanh((x+0.5-2*0.4)/(2*0.1));
subplot(2,3,i);
plot(x,u )
title("Exact solution")


figure(3);
Bp = polyfit(log10(h), log10(e), 1);
loglog(h,e);
slope=num2str(Bp(1));
```

```
37 title("Log-log figure for the error as function of mesh
       size")
38 subtitle("Slope of the fitted line: "+slope);
39
40
41 p1 = polyfit(h,e,1);
42 y1 = polyval(p1,h);
43 p2 = polyfit(h,e,2);
44 z=h(length(h)):0.00001:h(1);
45 y2 = polyval(p2,z);
46 figure(4);
47 plot(h,e, "-ro", h, y1, "b",z,y2,"g")
48 xlabel('Meash size')
49 ylabel('Error')
50 legend('datapoints','line fitting','2nd order curve
       fitting')
51 title("Error as function of mesh-size")
```

## Code: *myproject.m*

```
function u=myproject(n)

a=-1;
b=1;
N=n;
%N=641;
h=(b-a)/(N-1);
c=2;
e=0.1;
T=0.4;
M=Ma(N,h);
A=Aa(N);
S=e*Sa(N,h);
M_=inv(M);
A_=M_*A;
S_=M_*S;

    %%%%%%%%%%%% RK4 time integration
      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x= a:h:b;
x=x';
u= c-tanh((x+0.5)/(2*e));
t=0;
k=2*h^2;
%disp(A);

    for nr_itter=1:(T/k)
        u1=func(u,A_,S_);
        u2=func(u+k/2*u1,A_,S_);
        u3=func(u+k/2*u2,A_,S_);
        u4=func(u+k*u3,A_,S_);
        u=(u+k/6*(u1+2*u2+2*u3+u4));
        u(1)=u_ex(a,t,c,e);

        u(N)=u_ex(b,t,c,e);
        t=t+k;
```

```
36
37                  if mod(nr_itter,100)==0
38                  figure(1);
39                  clf('reset');
40
41                  plot(x,u)
42                  %plot(x,u,x,u_ex(x,t,c,e));
43
44                  title(['Numerical solution at t = ',num2str
                        (t)]);
45                  xlabel('x');ylabel('u');
46                  legend('v');
47                  end
48      end
49      disp(u);
50  end
51
52  function M=Ma(N,h)
53  a=2*h/3;
54  b=h/6;
55  c=h/6;
56  M = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
        ones(1,N-1),-1);
57  M(1,1)=h/3;
58  M(N,N)=h/3;
59  end
60
61  function A=Aa(N)
62  a=0;
63  b=0.5  ;
64  c=-0.5;
65  A = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
        ones(1,N-1),-1);
66  A(1,1)=-0.5;
67  A(N,N)=0.5;
68  end
69
70  function S=Sa(N,h)
```

```
71 a=2/h  ;
72 b=-1/h  ;
73 c=-1/h  ;
74 S = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
       ones(1,N-1),-1);
75 S(1,1)=1/h;
76 S(N,N)=1/h;
77 end
78
79 function u=func(u,A_,S_)
80 u = -A_*(u.^2*0.5)-S_*u;
81 end
82
83 function u=u_ex(x,t,c,e)
84 u=c-tanh((x+0.5-c*t)/(2*e));
85 end
```

## Code: *myproject_2.m*

```
function myproject()

a=0;
b=2*pi;
N=201;
h=(b-a)/(N-1);
c=2;
e=0.1;
T=2;
M=Ma(N,h);
A=Aa(N);
S=e*Sa(N,h);
M_=inv(M);
A_=M_*A;
S_=M_*S;

      %%%%%%%%%%%%% RK4 time integration
         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x= a:h:b;
x=x';
t=0;
k=0.2*h^2;

i=1;
for e = [ 1, 0.1, 0.001, 0]
t=0;
M=Ma(N,h);
A=Aa(N);
S=e*Sa(N,h);
M_=inv(M);
A_=M_*A;
S_=M_*S;
u=sin(x);
      for nr_itter=1:(T/k)
          u1=func(u,A_,S_);
          u2=func(u+k/2*u1,A_,S_);
```

```
36              u3=func(u+k/2*u2,A_,S_);
37              u4=func(u+k*u3,A_,S_);
38              u=(u+k/6*(u1+2*u2+2*u3+u4));
39              u(1)=0;
40              u(N)=0;
41              t=t+k;
42
43                  if mod(nr_itter,100)==0
44                  figure(1);
45                  clf('reset');
46
47                  plot(x,u)
48                  %plot(x,u,x,u_ex(x,t,c,e));
49
50                  title(['Numerical solution at t = ',num2str
                        (t)]);
51                  xlabel('x');ylabel('u');
52                  legend('v');
53                  end
54          end
55          figure(2)
56          subplot(2,2,i)
57          plot(x,u)
58          title("eps= " +num2str(e) )
59          i=i+1;
60      end
61      end
62
63      function M=Ma(N,h)
64      a=2*h/3;
65      b=h/6;
66      c=h/6;
67      M = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
            ones(1,N-1),-1);
68      M(1,1)=h/3;
69      M(N,N)=h/3;
70      end
71
```

```
72 function A=Aa(N)
73 a=0;
74 b=0.5 ;
75 c=-0.5;
76 A = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
       ones(1,N-1),-1);
77 A(1,1)=-0.5;
78 A(N,N)=0.5;
79 end
80
81 function S=Sa(N,h)
82 a=2/h ;
83 b=-1/h ;
84 c=-1/h ;
85 S = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
       ones(1,N-1),-1);
86 S(1,1)=1/h;
87 S(N,N)=1/h;
88 end
89
90 function u=func(u,A_,S_)
91 u = -A_*(u.^2*0.5)-S_*u;
92 end
```

## Code: *myproject_3.m*

```
function myproject()

a=0;
b=2*pi;
N=201;
h=(b-a)/(N-1);
c=2;
e=0.1;
T=2;
M=Ma(N,h);
A=Aa(N);
S=e*Sa(N,h);
M_=inv(M);
A_=M_*A;
S_=M_*S;

      %%%%%%%%%%%% RK4 time integration
         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
x= a:h:b;
x=x';
u= c-tanh((x+0.5)/(2*e));
t=0;
k=h^2;

i=1;
for N = [ 41, 81, 161, 321, 641]
h=(b-a)/(N-1);
e=0.5*h;
x= a:h:b;
x=x';
M=Ma(N,h);
A=Aa(N);
S=e*Sa(N,h);
M_=inv(M);
A_=M_*A;
S_=M_*S;
```

```
u=sin(x);
    for nr_itter=1:(T/k)
        u1=func(u,A_,S_);
        u2=func(u+k/2*u1,A_,S_);
        u3=func(u+k/2*u2,A_,S_);
        u4=func(u+k*u3,A_,S_);
        u=(u+k/6*(u1+2*u2+2*u3+u4));
        u(1)=0;
        u(N)=0;
        t=t+k;

            if mod(nr_itter,100)==0
            figure(1);
            clf('reset');

            plot(x,u)
            %plot(x,u,x,u_ex(x,t,c,e));

            title(['Numerical solution at t = ',num2str
                (t)]);
            xlabel('x');ylabel('u');
            legend('v');
            end
    end
    figure(2)
    subplot(2,3,i)
    plot(x,u)
    title("N= " +num2str(N) +", eps=0.5h")
    i=i+1;
end
end

function M=Ma(N,h)
a=2*h/3;
b=h/6;
c=h/6;
M = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
    ones(1,N-1),-1);
```

```
72 M(1,1)=h/3;
73 M(N,N)=h/3;
74 end
75
76 function A=Aa(N)
77 a=0;
78 b=0.5 ;
79 c=-0.5;
80 A = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
       ones(1,N-1),-1);
81 A(1,1)=-0.5;
82 A(N,N)=0.5;
83 end
84
85 function S=Sa(N,h)
86 a=2/h ;
87 b=-1/h ;
88 c=-1/h ;
89 S = diag(a*ones(1,N)) + diag(b*ones(1,N-1),1) + diag(c*
       ones(1,N-1),-1);
90 S(1,1)=1/h;
91 S(N,N)=1/h;
92 end
93
94 function u=func(u,A_,S_)
95 u = -A_*(u.^2*0.5)-S_*u;
96 end
97
98 function u=u_ex(x,t,c,e)
99 u=c-tanh((x+0.5-c*t)/(2*e));
100 end
```