Project 3

NUMN12 Numerical Methods for Solving Differential Equations

Felicia Segui
Ebba Rickard

December 2019

1 Part 1. The diffusion equation

The diffusion equation is presented in equation 1 with the boundary values and initial values used in this project.

$$\begin{aligned} u_t &= u_{xx} \\ u(t,0) &= u(t,1) = 0 \\ u(0,x) &= g(x) \end{aligned} \tag{1}$$

The differential operator $\frac{\partial^2}{\partial x^2}$ is approximated with finite differences, see equation 2. N approximations are made inside the interval [0,1] spaced $\Delta x = \frac{1}{N+1}$ apart creating the approximated vector $\overline{y}$. The problem is now linearized and can be expressed as a matrix equation, resulting in equation 3. On the boundaries of the interval, the boundary conditions in equation 1 are used. These are also inserted in equation 2 for j = 1 and j = N.

$$\frac{\partial^2}{\partial x^2} y_j = \frac{y_{j-1} - 2y_j + y_{j+1}}{\Delta x^2} \tag{2}$$

$$\begin{bmatrix} \frac{\partial}{\partial t} y_1 \\ \frac{\partial}{\partial t} y_2 \\ \vdots \\ \frac{\partial}{\partial t} y_{N-1} \\ \frac{\partial}{\partial t} y_N \end{bmatrix} = \frac{1}{\Delta x^2} \begin{bmatrix} -2 & 1 & 0 & \dots & & \\ & 1 & -2 & 1 & & \\ & & 1 & -2 & 1 & \\ & & & \ddots & & \\ & \dots & 0 & 1 & -2 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{N-1} \\ y_N \end{bmatrix} \iff y_t = T_{\Delta x} y \tag{3}$$

1.1 Explicit Euler Method

The explicit Euler method can be used to solve equation 3. The time interval is divided into M time steps which inserted into equation 4 gives an approximation to the solution at time t = mΔt. Here the initial condition $u(0,x) = g(x)$ in equation 1 is set equal to the approximation at t = 0 $\rightarrow$ m = 0. A function using explicit Euler to solve the diffusion equation was created in Matlab, see scripts in figure 9 and 10 in appendix.

$$y^{m+1} = y^m + \Delta t \cdot T_{\Delta x} y \tag{4}$$

The method is Explicit, meaning that when using it to solve a stiff problem it has strict stability conditions. In figure 1 the time step size is set to $\Delta t = \frac{\Delta x^2}{1{,}8}$. It is unstable, but just outside of the stability region. When the time step is made smaller the solution becomes stable. In figure 2 the solution is stable with a step size of $\Delta t = \frac{\Delta x^2}{4}$. To obtain stability the CFL condition need to be fulfilled. For Explicit Euler this means fulfilling the condition in equation 5.

$$4\frac{\Delta t}{\Delta x^2} \leq 2 \quad \Leftrightarrow \quad \Delta t \leq \frac{\Delta x^2}{2} \tag{5}$$

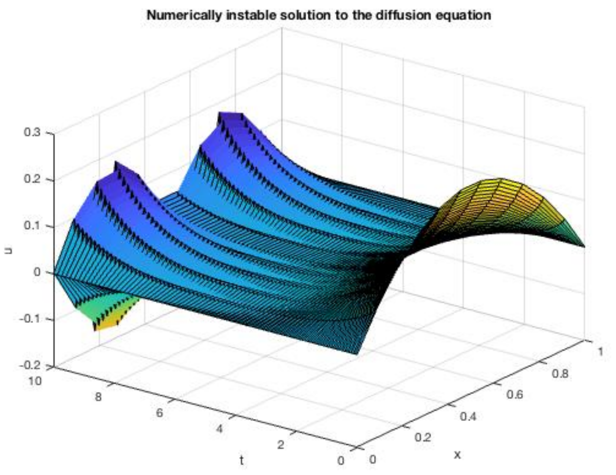


Figure 1: Diffusion equation solved with Explicit Euler in the time domain and finite discretization method in the space domain.

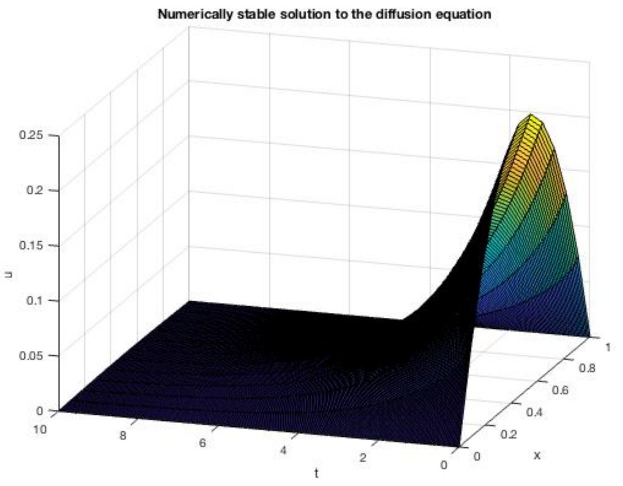


Figure 2: Diffusion equation solved with Explicit Euler in the time domain and finite discretization method in the space domain.

1.2 Crank-Nicolson Method

To avoid CFL-conditions on the time step size an A-stable method can be used, as these are unconditionally stable if the problem is mathematically stable. The Crank-Nicolson method is an A-stable method, as it is based on the trapezoidal rule which is A-stable. A function solving the diffusion equation using Crank-Nicolson was created in Matlab. See figure 11 and 12 in Appendix for scripts. The solution in figure 3 has time step size $\Delta t = 0,9091$ and space step size $\Delta x = 0,0909$ which gives the Courant number $\frac{\Delta t}{\Delta x^2} = 110$. This Courant number would have violated the CFL restriction of $\frac{\Delta t}{\Delta x^2} \leq \frac{1}{2}$ for Explicit Euler, but gives a stable solution when using Crank-Nicolson.

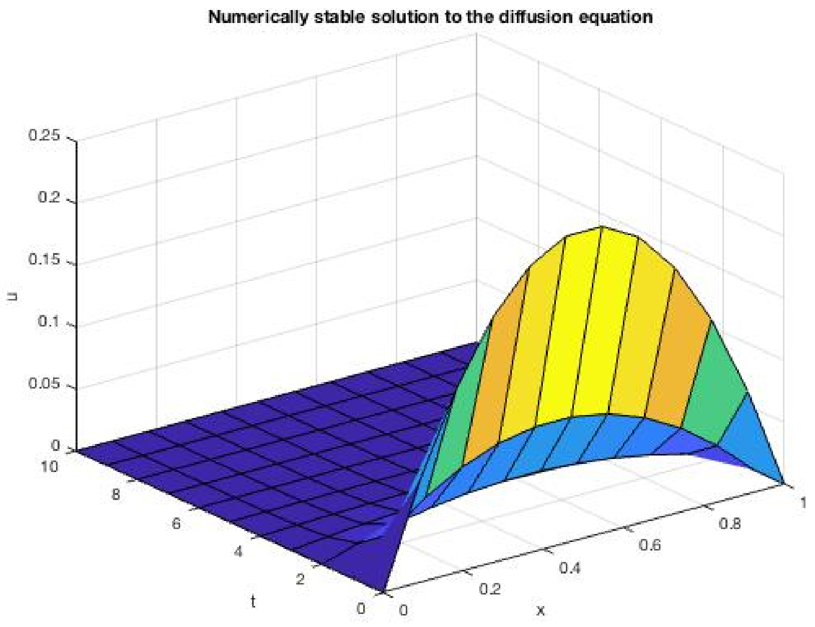


Figure 3: Diffusion equation solved with the Crank-Nicolson method in the time domain and finite discretization method in the space domain.

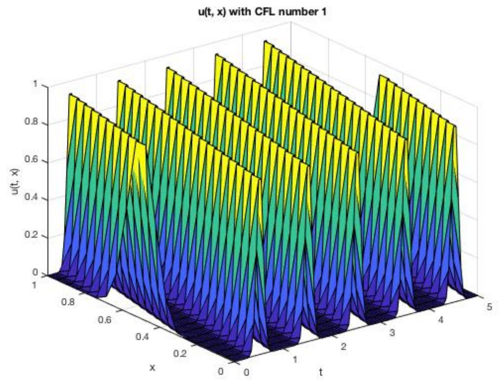
2 Part 2. The advection equation

The linear advection equation is presented in equation 6 [1].

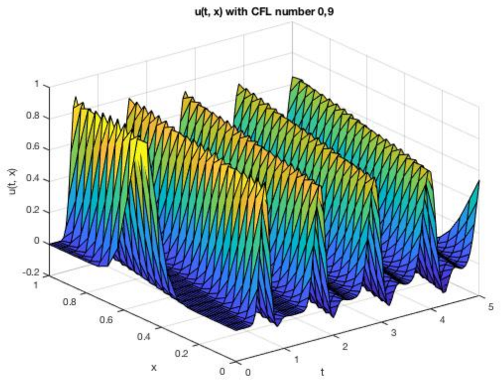
$$u_t + au_x = 0 \tag{6}$$

In figure 4a and figure 4b the solutions to the advection equation are plotted. $g(x) = e^{-100(x-0.5)^2}$ was used as the initial condition since it satisfies the conditions presented in equation 7 well. It was also given periodic boundary conditions.

$$\begin{aligned} u(t,0) &= u(t,1) \\ u(0,x) &= g(x) \\ g(0) &= g(1) \\ g'(0) &= g'(1) \end{aligned} \tag{7}$$



(a) Advection equation solved using the Lax-Wendroff scheme, initial condition $g(x) = e^{-100(x-0.5)^2}$ with CFL number aμ=1.



(b) Advection equation solved using the Lax-Wendroff scheme, initial condition $g(x) = e^{-100(x-0.5)^2}$ with CFL number aμ=0,9.

The solutions in figure 4a and 4b is plotted on an equidistant time- and space grid, with $\Delta x = \frac{1}{N}$ on the interval [0, 1] and $\Delta t = \frac{1}{M}$ on the interval [0, 5], with N = 20, M = 100 and $a\mu = 1$ for plot in figure 4a and $a\mu$ = 0,9 for plot in figure 4b. In figure 4a transportation of materia is shown, with conserved quantity. However, in figure 4b the quantity decreases, which only by looking at the graph, would indicate a diffusion process. There is no diffusion term in the advection equation, see equation 6 and therefore $a\mu = 0{,}9$ should not be chosen for solving the advection equation. The difference in amplitude is even better shown when the RMS norms are computed, see figure 5a and figure 5b. To compute the solution u(t, x) a method of order 2, called the *Lax Wendroff scheme*, was used. The method is derived from a Taylor expansion which gives equation 8 [1].

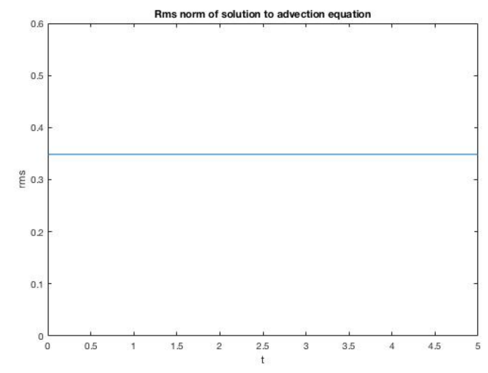
$$\begin{aligned} y_j^{n+1} &= \frac{a\mu}{2}(1+a\mu)y_{j-1}^n + (1-(a\mu)^2)y_j^n - \frac{a\mu}{2}(1-a\mu)y_{j+1}^n \\ \mu &= \frac{\Delta t}{\Delta x} \end{aligned} \tag{8}$$

This scheme and a test script where implemented in Matlab, see script in Appendix, figure 15 and 16. The matrix was made circulant through the first and last equation of the interval, described in equation 9.

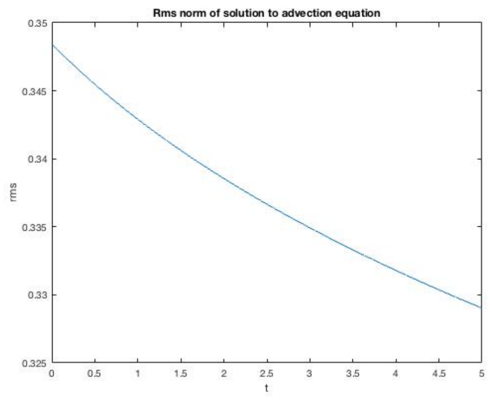
$$\begin{aligned} y_1^{n+1} &= \frac{a\mu}{2}(1+a\mu)y_N^n + (1-(a\mu)^2)y_1^n - \frac{a\mu}{2}(1-a\mu)y_2^n \\ y_N^{n+1} &= \frac{a\mu}{2}(1+a\mu)y_{N-1}^n + (1-(a\mu)^2)y_N^n - \frac{a\mu}{2}(1-a\mu)y_1^n \end{aligned} \tag{9}$$

The sign of the advection coefficient represent the direction of the flow [1], which easily could be confirmed as the plot for a = -1 only differed from figure 4a in the direction of the waves.

The RMS norm is plotted for CFL number = 1 in figure 5a and for CFL number = 0,9 in figure 5b. The RMS norm for CFL number = 1 is constant, while the RMS norm for CFL number = 0,9 is not. This can also be seen in their corresponding solutions in graphs 4a and 4b. With CFL number = 1 the exact solution is given, as the advection only represents a transport of materia and should therefore not give a solution with an RMS norm that increases or decreases over time.



(a) Rms norm of solution to advection equation with initial condition $g(x) = e^{-100(x-0.5)^2}$ and a·μ=1.



(b) Rms norm of solution to advection equation with initial condition $g(x) = e^{-100(x-0.5)^2}$ and a·μ=0,9.

3 Part 3. The convection-diffusion equation

The convection-diffusion equation is presented in equation 10 where a is the convection velocity and d the diffusivity. Explicit time stepping methods are not well suited to convection-diffusion problems, due to its stiffness. Therefore the trapezoidal method is used to solve the equation[1], as it is implicit.

$$u_t + a \cdot u_x = d \cdot u_{xx} \tag{10}$$

Periodic boundary conditions are used and an initial condition equation is picked such that it satisfies conditions presented below, see equation 11.

$$\begin{aligned} &u(0, x) = g(x) \\ &g(0) = g(1) \\ &g'(0) \approx g'(1) \end{aligned} \tag{11}$$

The convection-diffusion equation is solved with an equidistant space- and time grid with $\Delta x = \frac{1}{N}$ and $\Delta t = \frac{t_{end}}{M}$ on the intervall [0, 1]. In figure 6 the solution is presented with N = 20, M = 100, a = 1 and d = 0,1. A negative d gives an unstable solution, which can be explained with that it would represent a spontaneous concentrating process and therefore is impossible, as it contradicts the law about the always increasing entropy.

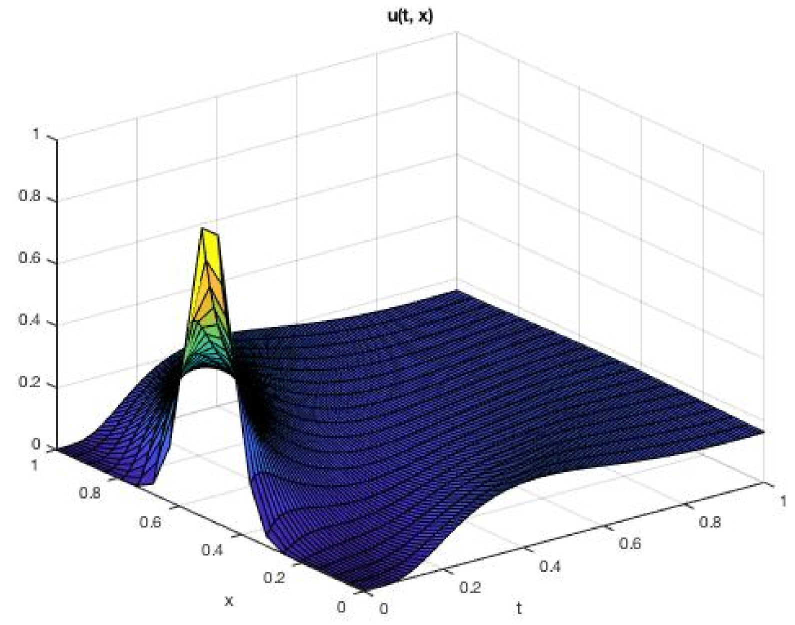


Figure 6: Convection-diffusion equation solved with the trapezoidal method, with Peclét number = 10 and CFL number = 0,5.

The problem was solved with a test script *test3.m* and a function called *convdiff.m*, see figure 18 and 17 in appendix. The method used in the function solves equation 12, where the matrix is circulant to meet the periodic boundary conditions.

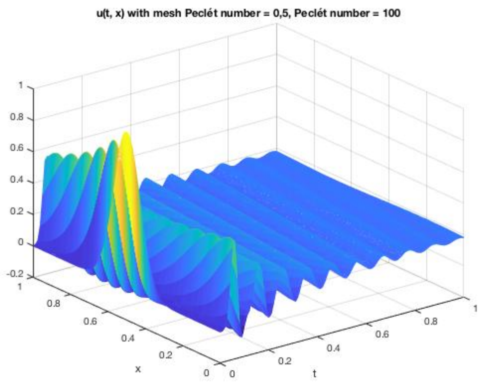
$$\dot{y} = (d \cdot T_{\Delta x} - a \cdot S_{\Delta x})y \tag{12}$$

The Péclet number is the absolute value of the ratio between the convection term and the diffusion term. This together with Δx needs to follow the condition in equation 13 [1].

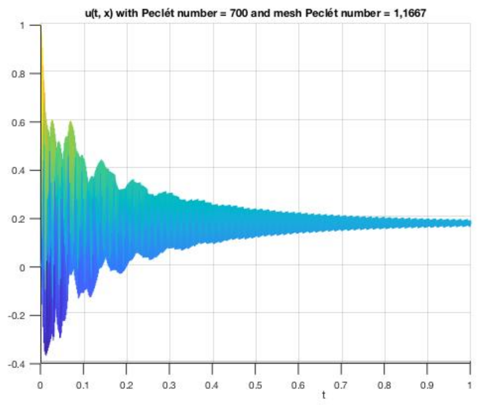
$$\begin{aligned} &Pe \cdot \Delta x < 2 \\ &Pe \approx [1, 1000] \end{aligned} \tag{13}$$

The solution presented in figure 6 has a corresponding mesh Péclet number of 0,5 and a Péclet number of 10, which fulfills the condition presented in equation 13 above.

In figure 7a a solution with Péclet number = 100 and CFL number = 0,5 is presented and in figure 7b a solution with Péclet number = 700 and CFL number = 1,1667 is presented. A big Péclet number indicates a big convection coefficient, in comparison with the diffusion coefficient. This is visualized in figure 7a and even more in figure 7b, unlike the solution presented in figure 6 in which the convection barely can be seen.



(a) Convection-diffusion equation solved with the trapezoidal method with Péclet number = 100 and CFL number = 0,5.



(b) Convection-diffusion equation solved with the trapezoidal method with Péclet number = 700 and CFL number = 1,1667.

4 Part 4. The viscous Burger equation

The viscous Burger equation is a nonlinear partial differential equation. As is seen in equation 14 it has both an advection and a diffusion term. Instead of a scalar deciding the rate and the direction of the advection it is now ruled by u, thus making it nonlinear. This means that the advection term cannot be discretized with a matrix like in the previous tasks.

$$u_t + uu_x = d \cdot u_{xx} \tag{14}$$

To discretize the advection term the Lax-Wendroff scheme is used. The derivative is approximated to the first three terms of the Taylor expansion of the exact solution, and given in equation 15.

$$u(t+\Delta t, x) \approx u(t,x) + \Delta t u_t + \frac{\Delta t^2}{2!} u_{tt} \tag{15}$$

Deriving the inviscid Burgers equation in equation 16 with respect to both t and x gives expressions for u_t and u_{tt} only containing space derivatives which we can discretize with finite difference matrices like before. This results in the final recursion equation 17.

$$\begin{aligned} u_t &= -uu_x \\ u_{tt} &= -u_t u_x - uu_{tx} \\ u_{tx} &= -u_x u_x - uu_{xx} \end{aligned} \tag{16}$$

$$y_j^{n+1} = \left(1 - \frac{\mu}{2}(y_{j+1}^n - y_{j-1}^n) + \frac{\mu^2}{4}(y_{j+1}^n - y_{j-1}^n)^2 + \frac{\mu^2}{2}(y_{j+1}^n - 2y_j^n + y_{j-1}^n) \cdot y_j^n\right) \cdot y_j^n \tag{17}$$

Equation 17 was used in the making of the Matlab function LW, which then takes a time step using the Lax-Wendroff scheme. See script in figure 19 in appendix. The approximation to the diffusion term was added and carried out using the trapezoidal rule, resulting in the time step recursion in equation 18.

$$y_j^{n+1} = LW(y_j^n) + d \cdot \frac{\Delta t}{2}(T_{\Delta x} y_j^{n+1} + T_{\Delta x} y_j^n) \tag{18}$$

Rearranging equation 18 gives the final recursion equation 19 used to approximate the solution over time. Using this the function *solveBurger* was created, see script in figure 20 and 21 in appendix.

$$y_j^{n+1} = (I - d \cdot \frac{\Delta t}{2} T_{\Delta x})^{-1}(LW(y_j^n) + d \cdot \frac{\Delta t}{2} T_{\Delta x}) \cdot y_j^n \tag{19}$$

The solution to the viscous Burgers equation is presented in figure 8. It shows a nonlinear behaviour as the characteristics are non-parallel. If the characteristics collide a shock is created. There is no shock in figure 8, but the propagating waves have steep gradients which can be seen from the sharp contrast between the slopes on the side of the wave.

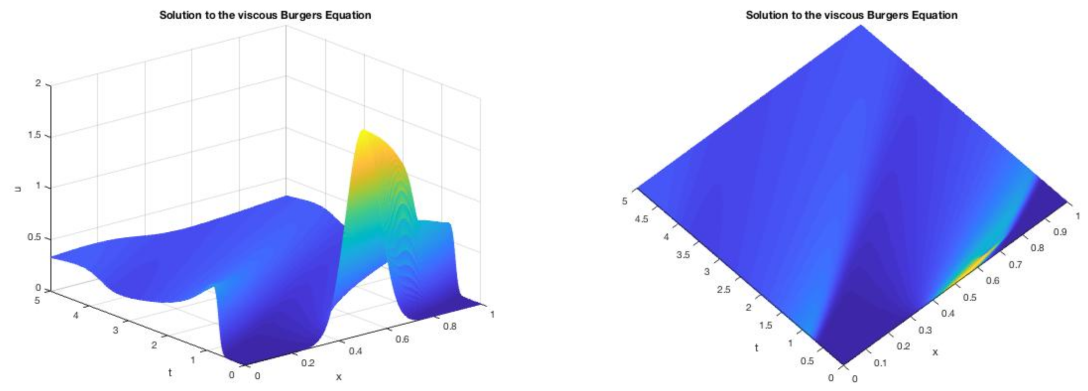


Figure 8: Solution to the viscous Burgers equation with initial condition $g(x) = 2e^{-100(x-0.5)^2}$ and diffusion constant d=0,008.

5 Acknowledgements

We want to thank our exercise mentor Peter Meisrimel, for explaining and helping with Matlab scripts. In addition we want to acknowledge our classmates Louise Karsten, Anna Sjerling, Isabelle Frodé and Sara Enander, for good discussions and sharing of knowledge.

6 Summary

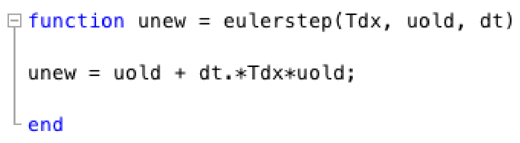
During these three projects we have gained knowledge about different numerical methods for solving differential equations. We have computed solutions for initial value problems, boundary value problems and initial value problems with one space dimension, which has given us a greater ability to understand and handle these kind of equations.

References

[1] Stillfjord, T, Söderlind, G. *Project 2 in FMNN10 and NUMN12.* Lunds Tekniska Högskola. 2019.

7 Appendix - Matlab scripts

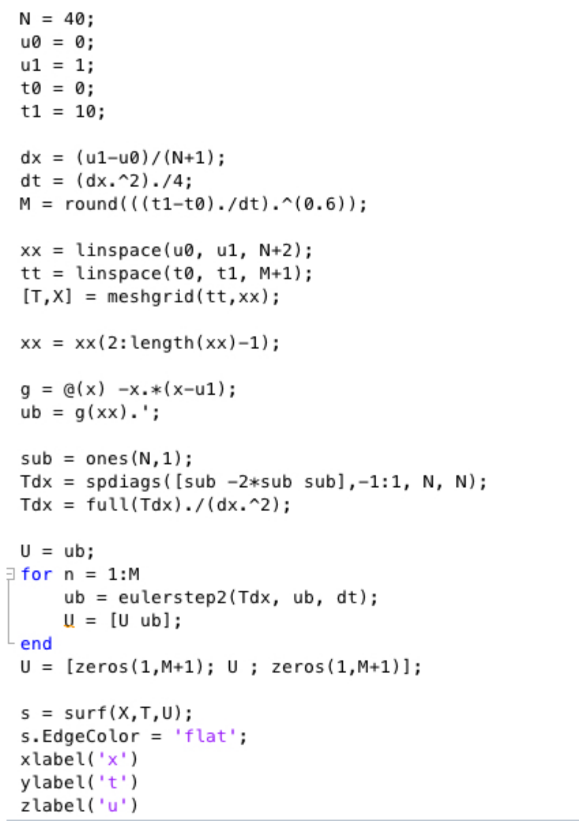
7.1 eulerstep.m



```
function unew = eulerstep(Tdx, uold, dt)

unew = uold + dt.*Tdx*uold;

end
```

Figure 9: Matlab script for function eulerstep.m.

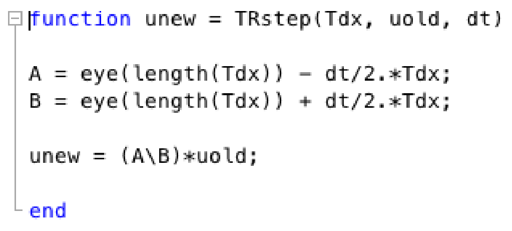
7.2 Script task 1.1



```
N = 40;
u0 = 0;
u1 = 1;
t0 = 0;
t1 = 10;

dx = (u1-u0)/(N+1);
dt = (dx.^2)./4;
M = round(((t1-t0)./dt).^(0.6));

xx = linspace(u0, u1, N+2);
tt = linspace(t0, t1, M+1);
[T,X] = meshgrid(tt,xx);

xx = xx(2:length(xx)-1);

g = @(x) -x.*(x-u1);
ub = g(xx).';

sub = ones(N,1);
Tdx = spdiags([sub -2*sub sub],-1:1, N, N);
Tdx = full(Tdx)./(dx.^2);

U = ub;
for n = 1:M
    ub = eulerstep2(Tdx, ub, dt);
    U = [U ub];
end
U = [zeros(1,M+1); U ; zeros(1,M+1)];

s = surf(X,T,U);
s.EdgeColor = 'flat';
xlabel('x')
ylabel('t')
zlabel('u')
```

Figure 10: Matlab script for solving the diffusion equation with explicit Euler.

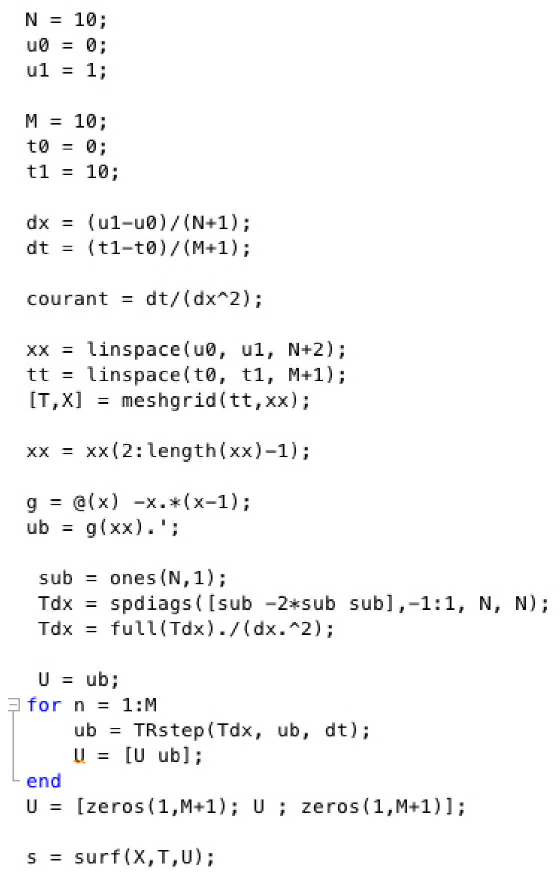
7.3 TRstep.m



```
function unew = TRstep(Tdx, uold, dt)

A = eye(length(Tdx)) - dt/2.*Tdx;
B = eye(length(Tdx)) + dt/2.*Tdx;

unew = (A\B)*uold;

end
```

Figure 11: Matlab script for function TRstep.

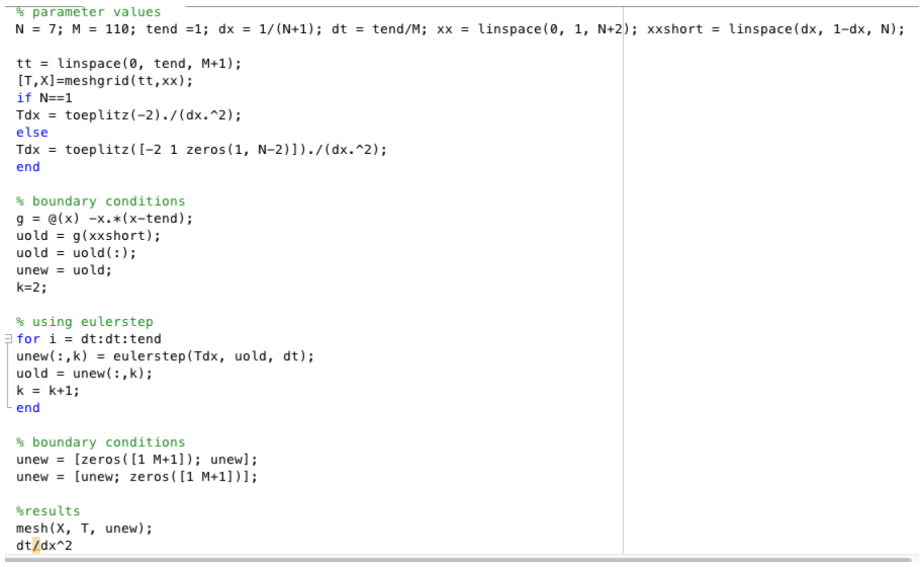
7.4 Script task 1.2



```
N = 10;
u0 = 0;
u1 = 1;

M = 10;
t0 = 0;
t1 = 10;

dx = (u1-u0)/(N+1);
dt = (t1-t0)/(M+1);

courant = dt/(dx^2);

xx = linspace(u0, u1, N+2);
tt = linspace(t0, t1, M+1);
[T,X] = meshgrid(tt,xx);

xx = xx(2:length(xx)-1);

g = @(x) -x.*(x-1);
ub = g(xx).';

 sub = ones(N,1);
 Tdx = spdiags([sub -2*sub sub],-1:1, N, N);
 Tdx = full(Tdx)./(dx.^2);

 U = ub;
for n = 1:M
    ub = TRstep(Tdx, ub, dt);
    U = [U ub];
end
U = [zeros(1,M+1); U ; zeros(1,M+1)];

s = surf(X,T,U);
```

Figure 12: Matlab script for solving the diffusion equation with Crank-Nicolson.

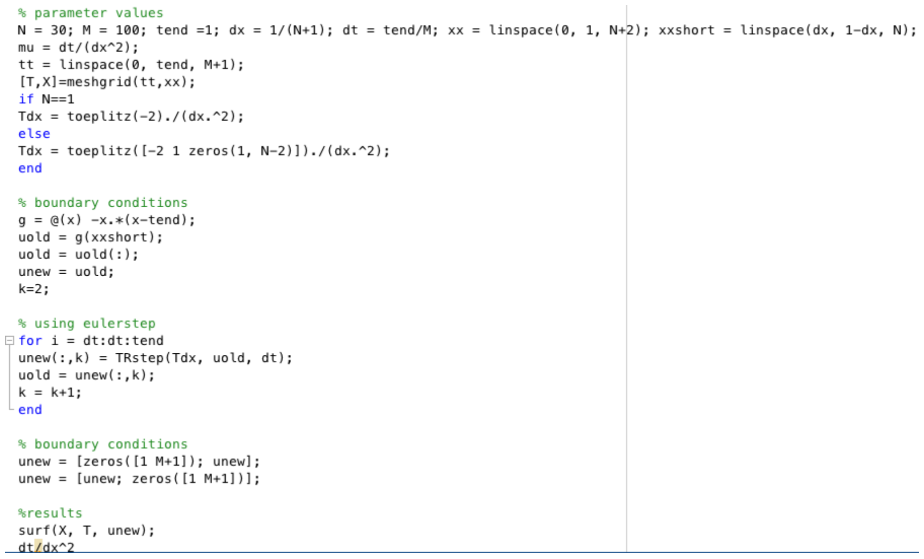
7.5 test11.m



```
% parameter values
N = 7; M = 110; tend =1; dx = 1/(N+1); dt = tend/M; xx = linspace(0, 1, N+2); xxshort = linspace(dx, 1-dx, N);

tt = linspace(0, tend, M+1);
[T,X]=meshgrid(tt,xx);
if N==1
Tdx = toeplitz(-2)./(dx.^2);
else
Tdx = toeplitz([-2 1 zeros(1, N-2)])./(dx.^2);
end

% boundary conditions
g = @(x) -x.*(x-tend);
uold = g(xxshort);
uold = uold(:);
unew = uold;
k=2;

% using eulerstep
for i = dt:dt:tend
unew(:,k) = eulerstep(Tdx, uold, dt);
uold = unew(:,k);
k = k+1;
end

% boundary conditions
unew = [zeros([1 M+1]); unew];
unew = [unew; zeros([1 M+1])];

%results
mesh(X, T, unew);
dt/dx^2
```

Figure 13: Matlab script for task 1.1

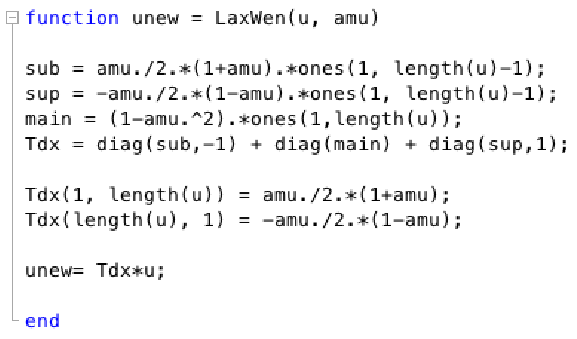
7.6 test12.m



```
% parameter values
N = 30; M = 100; tend =1; dx = 1/(N+1); dt = tend/M; xx = linspace(0, 1, N+2); xxshort = linspace(dx, 1-dx, N);
mu = dt/(dx^2);
tt = linspace(0, tend, M+1);
[T,X]=meshgrid(tt,xx);
if N==1
Tdx = toeplitz(-2)./(dx.^2);
else
Tdx = toeplitz([-2 1 zeros(1, N-2)])./(dx.^2);
end

% boundary conditions
g = @(x) -x.*(x-tend);
uold = g(xxshort);
uold = uold(:);
unew = uold;
k=2;

% using eulerstep
for i = dt:dt:tend
unew(:,k) = TRstep(Tdx, uold, dt);
uold = unew(:,k);
k = k+1;
end

% boundary conditions
unew = [zeros([1 M+1]); unew];
unew = [unew; zeros([1 M+1])];

%results
surf(X, T, unew);
dt/dx^2
```

Figure 14: Matlab script for task 1.2.

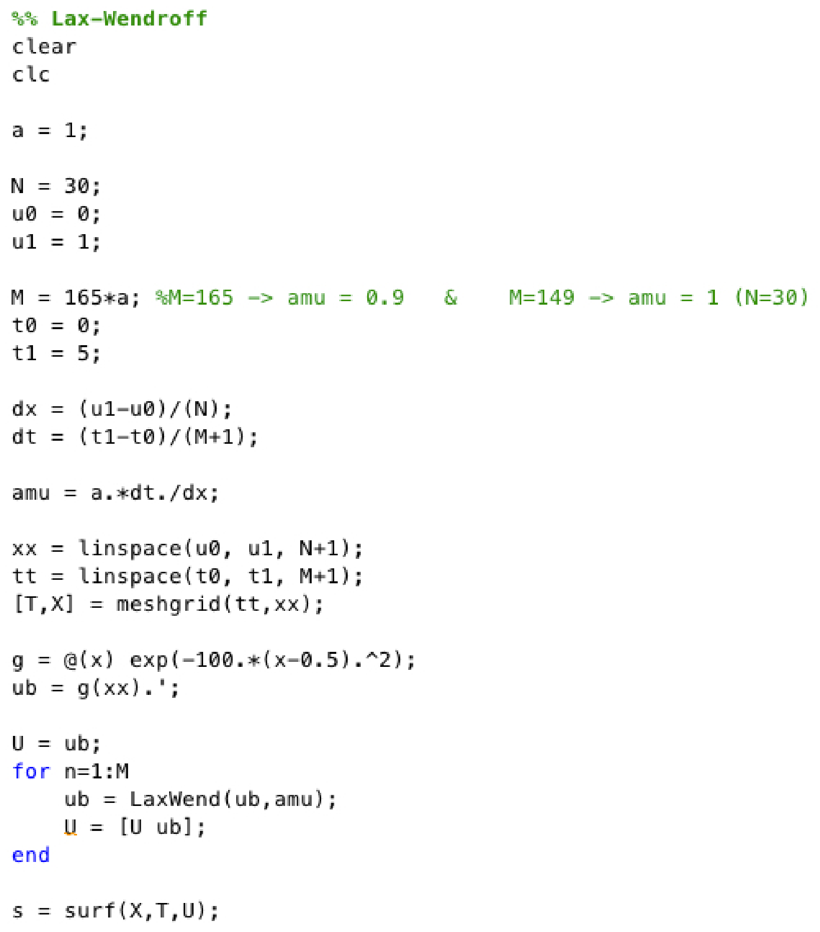
7.7 LaxWen.m



```
function unew = LaxWen(u, amu)

sub = amu./2.*(1+amu).*ones(1, length(u)-1);
sup = -amu./2.*(1-amu).*ones(1, length(u)-1);
main = (1-amu.^2).*ones(1,length(u));
Tdx = diag(sub,-1) + diag(main) + diag(sup,1);

Tdx(1, length(u)) = amu./2.*(1+amu);
Tdx(length(u), 1) = -amu./2.*(1-amu);

unew= Tdx*u;

end
```

Figure 15: Matlab script for function LaxWen.

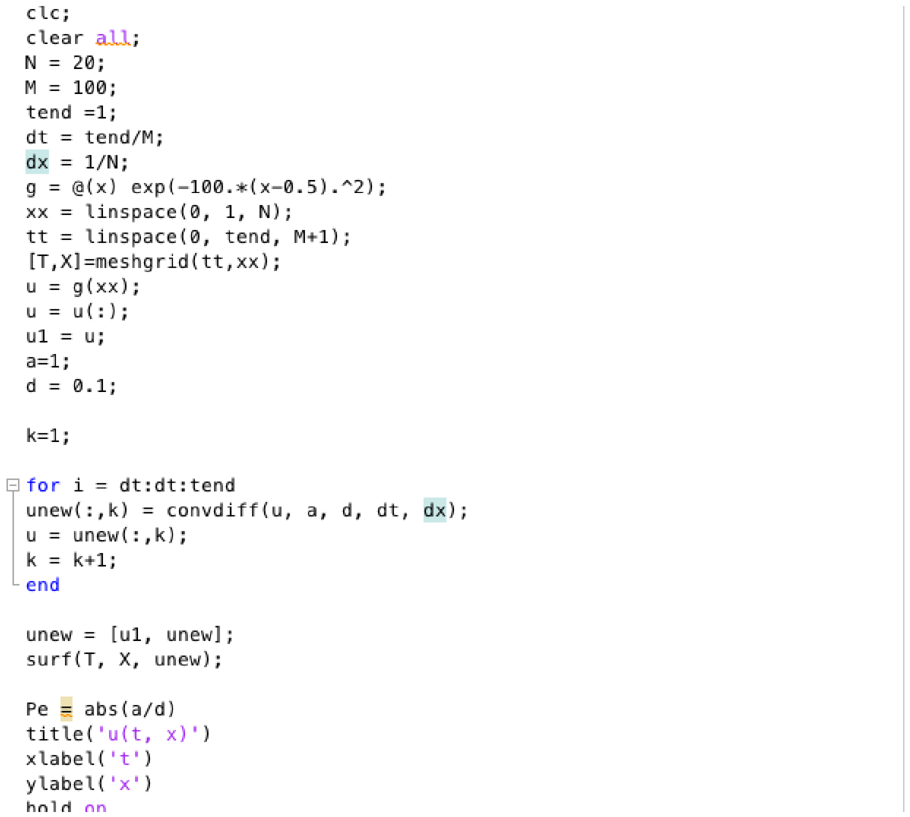
7.8 test21.m



```
%% Lax-Wendroff
clear
clc

a = 1;

N = 30;
u0 = 0;
u1 = 1;

M = 165*a; %M=165 -> amu = 0.9   &    M=149 -> amu = 1 (N=30)
t0 = 0;
t1 = 5;

dx = (u1-u0)/(N);
dt = (t1-t0)/(M+1);

amu = a.*dt./dx;

xx = linspace(u0, u1, N+1);
tt = linspace(t0, t1, M+1);
[T,X] = meshgrid(tt,xx);

g = @(x) exp(-100.*(x-0.5).^2);
ub = g(xx).';

U = ub;
for n=1:M
    ub = LaxWend(ub,amu);
    U = [U ub];
end

s = surf(X,T,U);
```

Figure 16: Matlab test script for task 2.1.

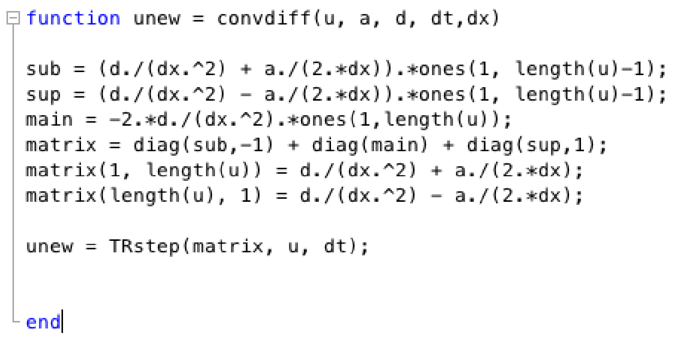
7.9 test3.m



```
clc;
clear all;
N = 20;
M = 100;
tend =1;
dt = tend/M;
dx = 1/N;
g = @(x) exp(-100.*(x-0.5).^2);
xx = linspace(0, 1, N);
tt = linspace(0, tend, M+1);
[T,X]=meshgrid(tt,xx);
u = g(xx);
u = u(:);
u1 = u;
a=1;
d = 0.1;

k=1;

for i = dt:dt:tend
unew(:,k) = convdiff(u, a, d, dt, dx);
u = unew(:,k);
k = k+1;
end

unew = [u1, unew];
surf(T, X, unew);

Pe = abs(a/d)
title('u(t, x)')
xlabel('t')
ylabel('x')
hold on
```

Figure 17: Matlab test script for task 3.

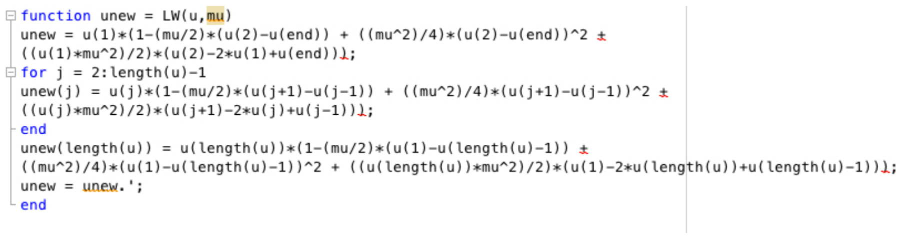
7.10 convdiff.m



```
function unew = convdiff(u, a, d, dt,dx)

sub = (d./(dx.^2) + a./(2.*dx)).*ones(1, length(u)-1);
sup = (d./(dx.^2) - a./(2.*dx)).*ones(1, length(u)-1);
main = -2.*d./(dx.^2).*ones(1,length(u));
matrix = diag(sub,-1) + diag(main) + diag(sup,1);
matrix(1, length(u)) = d./(dx.^2) + a./(2.*dx);
matrix(length(u), 1) = d./(dx.^2) - a./(2.*dx);

unew = TRstep(matrix, u, dt);


end
```

Figure 18: Script for solving convection-diffusion equations.

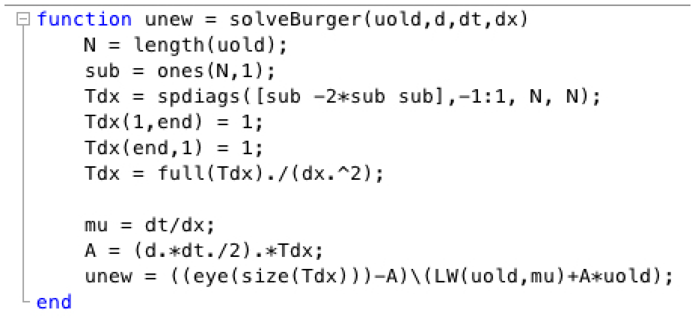
7.11 LW.m



```
function unew = LW(u,mu)
unew = u(1)*(1-(mu/2)*(u(2)-u(end)) + ((mu^2)/4)*(u(2)-u(end))^2 +
((u(1)*mu^2)/2)*(u(2)-2*u(1)+u(end)));
for j = 2:length(u)-1
unew(j) = u(j)*(1-(mu/2)*(u(j+1)-u(j-1)) + ((mu^2)/4)*(u(j+1)-u(j-1))^2 +
((u(j)*mu^2)/2)*(u(j+1)-2*u(j)+u(j-1)));
end
unew(length(u)) = u(length(u))*(1-(mu/2)*(u(1)-u(length(u)-1)) +
((mu^2)/4)*(u(1)-u(length(u)-1))^2 + ((u(length(u))*mu^2)/2)*(u(1)-2*u(length(u))+u(length(u)-1)));
unew = unew.';
end
```

Figure 19: Script discretizing the advection term in the viscous Burger equation.

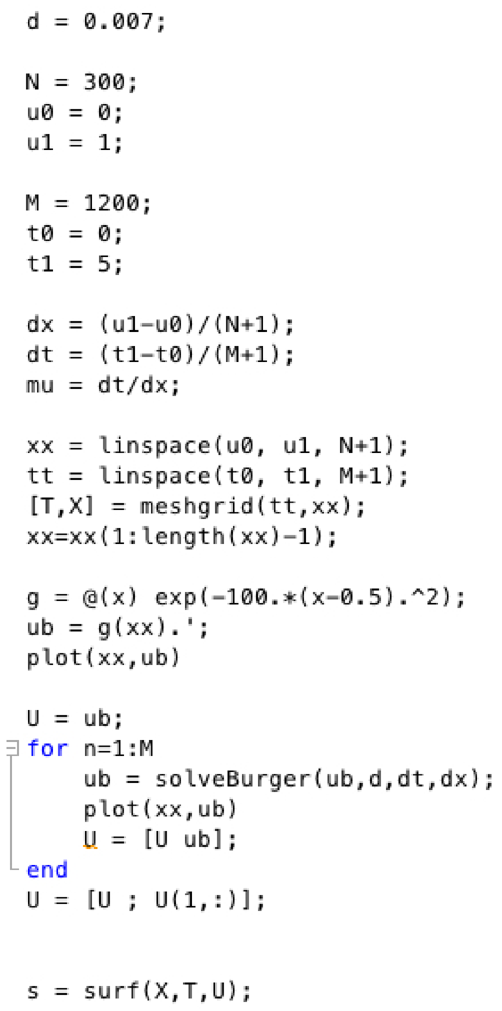
7.12 solveBurger.m



```
function unew = solveBurger(uold,d,dt,dx)
    N = length(uold);
    sub = ones(N,1);
    Tdx = spdiags([sub -2*sub sub],-1:1, N, N);
    Tdx(1,end) = 1;
    Tdx(end,1) = 1;
    Tdx = full(Tdx)./(dx.^2);

    mu = dt/dx;
    A = (d.*dt./2).*Tdx;
    unew = ((eye(size(Tdx)))-A)\(LW(uold,mu)+A*uold);
end
```

Figure 20: Script for taking a time step with viscous Burgers equation.

7.13 testBurger.m



```
d = 0.007;

N = 300;
u0 = 0;
u1 = 1;

M = 1200;
t0 = 0;
t1 = 5;

dx = (u1-u0)/(N+1);
dt = (t1-t0)/(M+1);
mu = dt/dx;

xx = linspace(u0, u1, N+1);
tt = linspace(t0, t1, M+1);
[T,X] = meshgrid(tt,xx);
xx=xx(1:length(xx)-1);

g = @(x) exp(-100.*(x-0.5).^2);
ub = g(xx).';
plot(xx,ub)

U = ub;
for n=1:M
    ub = solveBurger(ub,d,dt,dx);
    plot(xx,ub)
    U = [U ub];
end
U = [U ; U(1,:)];


s = surf(X,T,U);
```

Figure 21: Script solving the viscous Burger equation.