Technical University of Denmark

02685 Scientific Computing for Differential Equations
2017

Assignment 3

Authors:
Miguel Suau de Castro (s161333)
Michal Baumgartner (s161636)

May 8, 2017

Contents

1 Pure parabolic model problem

In this section we shall study and analyze two different approaches for solving parabolic differential equations. Concretely we will focus on the heat diffusion problem, which can be modeled by the following partial differential equation and its initial and boundary conditions.

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

The first thing we should notice is that, a part from having the usual spatial dependence, the equation also describes how the heat distribution evolves in time. Hence, if we want to solve this equation numerically we need to discretize both time and space, which means that the solution will be represented by a two dimensional grid.

1.1 Matlab implementation of the θ-scheme

Applying euler approximation to the time derivative and the second order central finite difference to the right hand side in equation **??**, an explicit scheme is obtained.

$$\frac{U_i^{n+1} - U_i^n}{k} = \frac{\kappa}{h^2}(U_{i-1}^n - 2U_i^n + U_{i+1}^n) \tag{2}$$

where h and k represent the distance between two grid points in space and time respectively.

A more general scheme for solving this problem can be formulated as:

$$\frac{U_i^{n+1} - U_i^n}{k} = \frac{\kappa}{h^2}((1-\theta)(U_{i-1}^n - 2U_i^n + U_{i+1}^n) + \theta(U_{i-1}^{n+1} - 2U_i^{n+1} + U_{i+1}^{n+1}) \tag{3}$$

where $0 \leq \theta \leq 1$

It is easy to see that when $\theta = 0$ equation 3 is equivalent to 2. However for any other value of θ the scheme becomes implicit but also more accurate.

The scheme takes the form of system of equations which can be written in matrix form as:

$$(I - \theta\mu A_0)\mathbf{u}^{n+1} = (I + (1-\theta)\mu A_0)\mathbf{u}^n + (\mathbf{g}^{n+1} - \mathbf{g}^n) \tag{4}$$

$$A_L \mathbf{u}^{n+1} = A_E \mathbf{u}^n + (\mathbf{g}^{n+1} - \mathbf{g}^n) \tag{5}$$

where I is the identity matrix and u^n and g^n are vectors of the form

$$\mathbf{u^n} = \begin{bmatrix} U_0^n & U_1^n & \dots & U_{M-1}^n & U_M^n \end{bmatrix}^T \qquad \mathbf{g^n} = \begin{bmatrix} g_L^n & 0 & \dots & 0 & g_R^n \end{bmatrix}^T \tag{6}$$

while A_0 is a matrix of dimensions $(M+1) \times (N+1)$ of the form

$$A_0 = \begin{bmatrix} 0 & 0 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \end{bmatrix} \tag{7}$$

Equation 5 is nothing but a system of ordinary differential equations and thus, the solution at a given time can be obtained by inverting the tridiagonal matrix in the left hand side. Consequently, using a loop the global solution can be computed sequentially at every point in the time grid.

The code snippet below shows the implementation of this scheme in MATLAB where the function given by equation 2 in the assignment specifications is used to compute the initial and boundary conditions.

```
function U = parabolicSolver(boundaryFun,h,k,theta,mu)
% The following function solves the heat diffusion equation using the
% theta-scheme. It takes as input the two step sizes h and k and the
% parameters theta and mu and returns the solution at every grid
    point
% Define the grid size
M = ceil(2/h);
N = ceil(1/k);
% Build A0
A0 = diag(-2*ones(M+1,1)) + diag(ones(M,1),-1) + diag(ones(M,1),1);
A0(1,:) = zeros(1,M+1);
A0(end,:) = zeros(1,M+1);
I = eye(M+1);
x = linspace(-1,1,M+1);
t = linspace(0,1,N+1);
% Obtain initial and boundary conditions
U(:,1) = boundaryFun(x,0);
g = [boundaryFun(-1,t); zeros(M-1,N+1); boundaryFun(1,t)];
% Solve the ODEs system
for k=2:N+1
    U(:,k) = (I-theta*mu*A0)\((I+(1-theta)*mu*A0)*U(:,k-1)+g(:,k)-g
        (:,k-1));
end
end
```

Figure 1 represents the solution at different points in time. It is easy to see in the graphs how the higher frequencies in the heat wave are damped quickly while the lower ones remain longer.

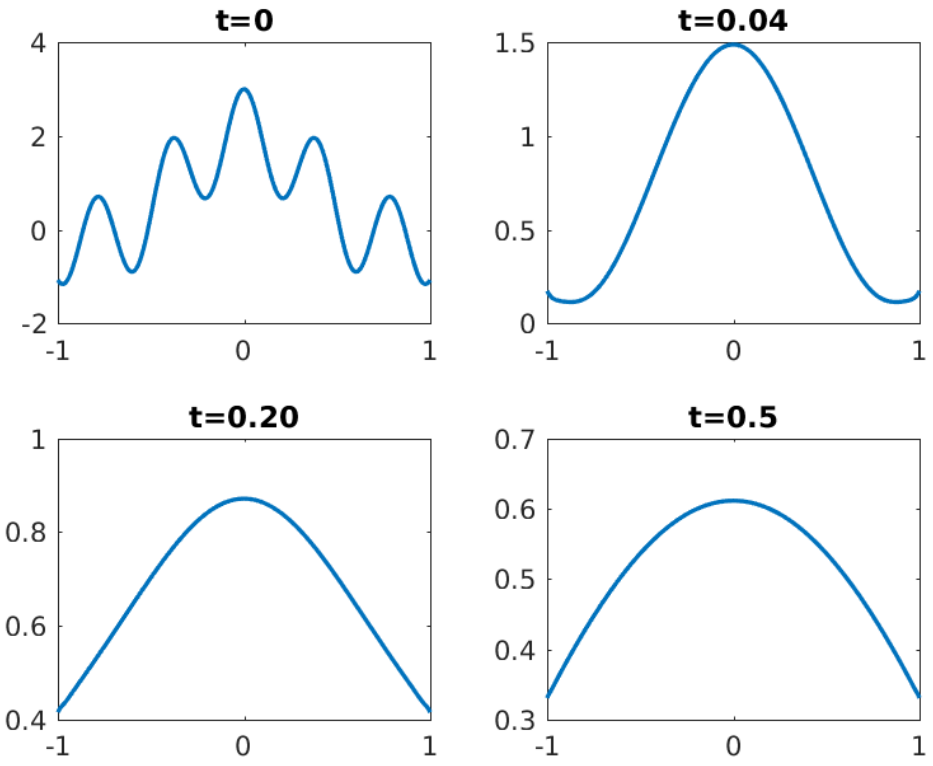

Figure 1: Solution of the heat diffusion equation at different points in time using the θ-scheme

1.2 Scheme analysis $\theta = 0$

Once the the scheme for solving the problem numerically has been built, it might be interesting to make an analysis on the system and draw some conclusions about how large the grid should be for the solver to be stable, what is its order of consistency or under what circumstances it satisfies the discrete maximum principle.

1.2.1 Consistency

As mentioned, for $\theta = 0$ equation 3 corresponds to the explicit forward time and central space scheme. We first check that the finite difference approximation is consistent, that is, the local truncation error approaches to 0 as the grid size increases.

$$\tau(x,t) = \frac{u(x,t+k) - u(x,t)}{k} - \frac{\kappa}{h^2}(u(x-h,t) - 2u(x,t) + u(x+h,t)) \tag{8}$$

by applying the Taylor series expansions about u(x,t)

$$\tau(x,t) = \left(u_t + \frac{1}{2}ku_{tt} + \frac{1}{6}k^2u_{ttt} + ...\right) - \kappa\left(u_xx + \frac{1}{12}h^2u_{xxxx} + ...\right) \tag{9}$$

and since we know that $u_t = \kappa u_x x$ and $u_t t = \kappa^2 u_x x$ then

$$\tau(x,t) = \left(\frac{\kappa^2}{2}k - \frac{\kappa}{12}h^2\right)u_{xxxx} + \mathcal{O}(k^2 + h^4) \tag{10}$$

where the dominant term depends on k and h^2 which means that the scheme is consistent of order $\mathcal{O}(k + h^2)$.

Besides, for $\mu = 1/6$

$$\tau(x,t) = h^2\left(\frac{\kappa}{12} - \frac{\kappa}{12}\right)u_{xxxx} + \mathcal{O}(k^2 + h^4) \tag{11}$$

the first term in the right hand side of equation 11 disappears and the order becomes $\mathcal{O}(k^2 + h^4)$.

1.2.2 Stability

Secondly, in order to find the stability of the method we need to check when the eigenvalues of the matrix A_E in the right hand side of equation 5 lie within the unit circle.

According to 2.23 in Randall Leveque the eigenvalues of A_0 are given by

$$\lambda_p = 2(\cos(p\pi h) - 1) \qquad \text{for} \qquad p = 1, 2, ..., m \tag{12}$$

Therefore, the system would be absolutely stable when $|1 + \mu\lambda_p| \leq 1$ and since the smallest λ_p is approximately -4 we have that the region is defined by:

$$-2 \leq -4\mu \leq 0 \tag{13}$$

which means that for the system to be stable the following relation must be satisfied

$$\frac{\kappa k}{h^2} \leq \frac{1}{2} \tag{14}$$

1.2.3 Convergence

According to theorem 9.2 and definition 9.1 in R.L. a system of the form (9.16) is convergent if there is a constant $C_t > 0$ such that

$$||B(k)^n|| \leq C_t \tag{15}$$

in our case

$$B(k) = A_E = I + \mu A_0 \tag{16}$$

We already showed in section 1.2.2 that for $\mu \leq 1/2$ the eigenvalues of A_E lied within the unit circle and hence, $||A_E|| \leq 1$, which means that for this choice of μ the system is both Lax-Richtmyer stable and convergent.

1.2.4 Discrete Maximum Principle

The discrete maximum principle states that if the solution is bounded from above and below by the most extreme values of the initial and boundary conditions then, as long as the conditions are consistent and smooth, it will converge uniformly.

According to Lecture 21 slide 17 the θ-scheme will satisfy the discrete maximum principle when:

$$\mu(1-\theta) \leq \frac{1}{2} \tag{17}$$

and in this case since $\theta = 0$

$$\frac{\kappa k}{h^2} \leq \frac{1}{2} \tag{18}$$

which is consistent with the stability criteria.

1.3 Scheme analysis $\theta = 1/2 + h^2/(12k\kappa)$

1.3.1 Consistency

According to what we saw when developing the θ-scheme we should expect a higher order for this particular choice of θ than in the previous case. The LTE is in this case

$$\tau(x,t) = \frac{u(x,t+k)-u(x,t)}{k} - \frac{\kappa}{h^2}\left(\left(\frac{1}{2}-\frac{h^2}{12k\kappa}\right)(u(x-h,t)-2u(x,t)+u(x+h,t))+ \left(\frac{1}{2}+\frac{h^2}{12k\kappa}\right)(u(x-h,t+k)-2u(x,t+k)+u(x+h,t+k))\right) \quad (19)$$

by taking Taylor expansions in the expression above

$$\tau(x,t) = \left(u_t + \frac{1}{2}ku_{tt} + \frac{1}{6}k^2u_{ttt} + ...\right) - \frac{\kappa}{h^2}\left(\left(\frac{1}{2}-\frac{h^2}{12k\kappa}\right)(h^2u_{xx}+\frac{1}{12}h^4u_{xxxx}+...)+ \left(\frac{1}{2}+\frac{h^2}{12k\kappa}\right)(h^2u_{xx}+\frac{k^3}{6}u_{ttt}+...)\right) \quad (20)$$

and since we know that $u_t = \kappa u_{xx}$, $u_{tt} = \kappa^2 u_{xxxx}$ and $u_{xxt} = \kappa u_{xxxx}$ then

$$\tau(x,t) = \left(\frac{\kappa h^2}{2} - \frac{h^2}{24}\right)u_{xxxx} + \left(\frac{k^2}{6}+\frac{k^3}{6}\right)u_{ttt} \quad ... \quad (21)$$

where the dominant term depends on k^2 and h^2 which means that the scheme is consistent of order $\mathcal{O}(k^2+h^2)$.

1.3.2 Stability

Following the same approach as before the eigenvalues of $B = A_L^{-1}A_E$ should lie inside the unit circle.

$$\lambda_B = \frac{1+\theta\mu\lambda_p}{1-\theta\mu\lambda_p} = \frac{1+(\frac{1}{2}\mu+\frac{1}{12})\lambda_p}{1-(\frac{1}{2}\mu+\frac{1}{12})\lambda_p} \quad (22)$$

where λ_B is given by equation 12 and $\mu = \kappa k/h^2$. We know that λ_p is strictly negative and since for the scheme to be stable $|\lambda_B| \leq 1$, we need μ to be positive. Which means that the system will be stable for any choice of $h > 0$ and $k > 0$.

1.3.3 Convergence

Again, a system of the form (9.16) is convergent if it is Lax-Richtmyer stable. We have proved in the previous section that the eigenvalues of B are within the unit circle for any positive choice of the step size. Hence, in this case

$$||B(k)|| \leq 1 \quad (23)$$

for any $k > 0$ and $h > 0$.

1.3.4 Discrete Maximum Principle

The θ-scheme will satisfy the discrete maximum principle when:

$$\mu(1-\theta) \leq \frac{1}{2} \tag{24}$$

and in this case since $\theta = \frac{1}{2} + \frac{1}{12\mu}$

$$\frac{\kappa k}{h^2} \leq \frac{7}{6} \tag{25}$$

1.4 Numerical evaluation

After deriving the analytical expressions, we shall check that the rates and the conditions obtained are satisfied in reality. To do so, we insert the solver implemented in MATLAB in a loop where we vary the time and space grids accordingly so that the system remains within the stability region and satisfies both the convergence criteria and the discrete maximum principle.

For the choice of $\theta = 0$, figure 2 represents the variation of the LTE when changing the size of h and k. In this case, we set the value of k to be equal to $\frac{h^2}{2}$ so that the eigenvalues of the system lie within the unit circle. The dashed lines, which are the theoretical order of the system ($\mathcal{O}(h^2)$ and $\mathcal{O}(k)$), can be thought as an upper bound for the empirical curves. It is easy to see that in both comparisons the slope of the LTE in logarithmic scale is greater than the slope of the help lines.

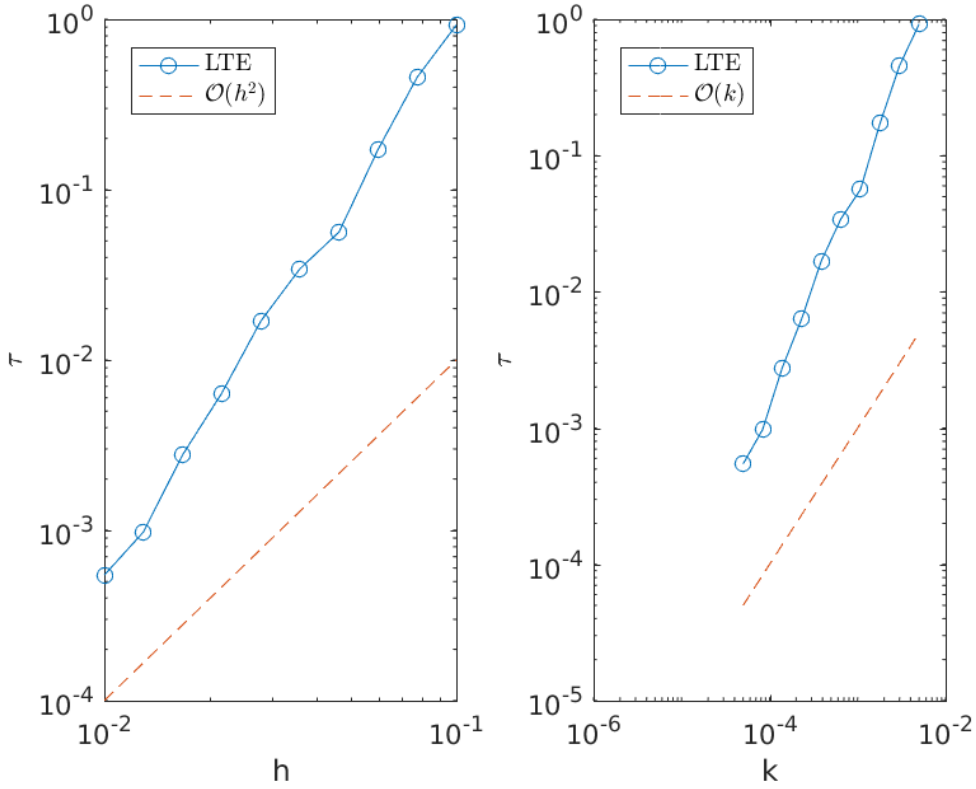

Figure 2: Local truncation error for different values of h and k, $\mu = 1/2$ and $\theta = 0$ in logarithmic scale.

When $k = h^2/6$ the graph in figure 3 also reveals that the the local truncation error respects the theoretical limits of $\mathcal{O}(h^4)$ and $\mathcal{O}(k^2)$.

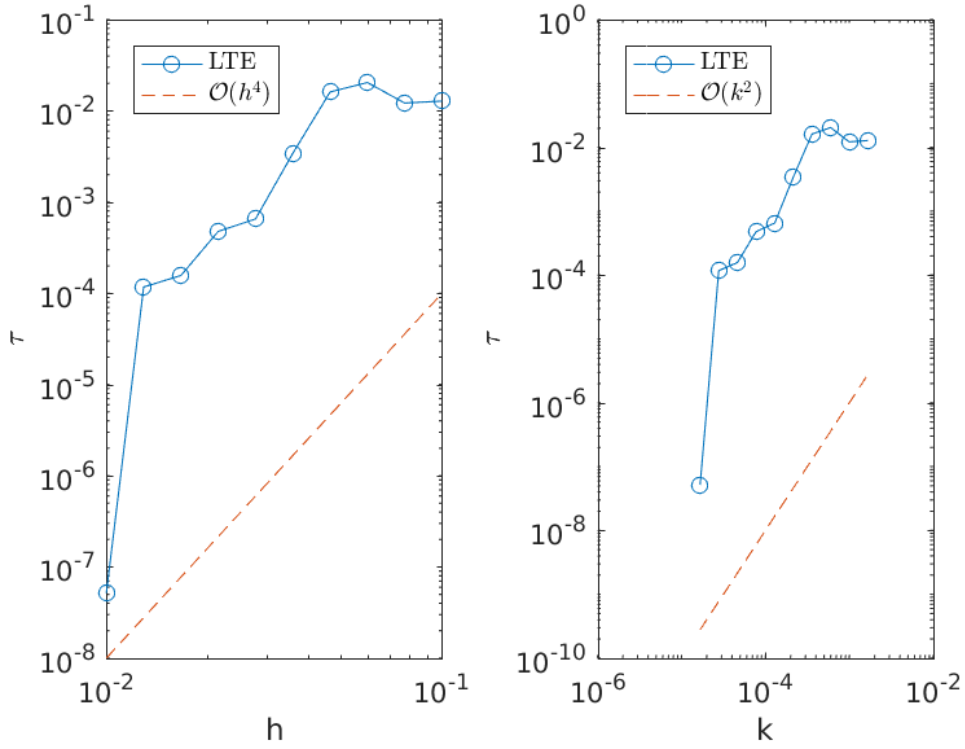

Figure 3: Local truncation error for different values of h and k, $\mu = 1/6$ and $\theta = 0$ in logarithmic scale.

On the other hand, for the special choice of $\theta = 1/2 + h^2/(12k\kappa)$ figure 4 also holds with what we found when we evaluated the scheme analytically in the previous section. Besides, in order to satisfy the discrete maximum principle k was selected so that $\mu = 7/6$.

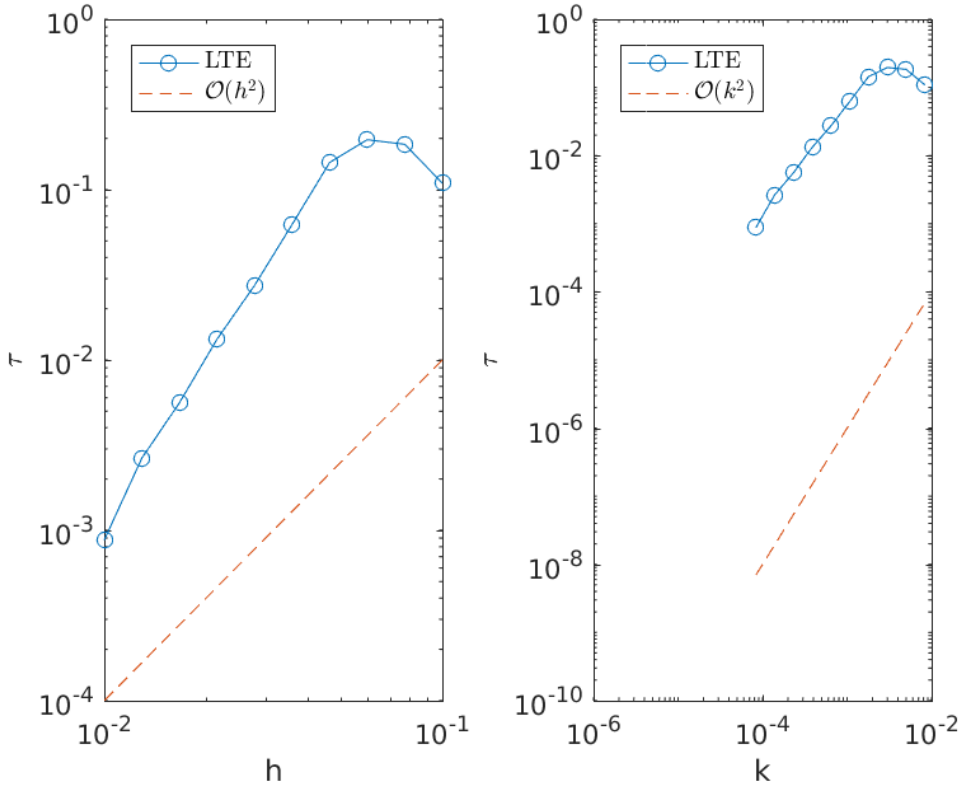

Figure 4: Local truncation error for different values of h and k, $\mu = 5/6$ and $\theta = 1/2 + h^2/(12k\kappa)$ in logarithmic scale.

2 Purely hyperbolic model problem

One dimensional unsteady advection problem is defined as follows:

$$\partial_t u(x,t) + a\partial_x u(x,t) = 0, \qquad -1 \leq x \leq 1, t > 0, \tag{26}$$
$$u(-1,t) = u(1,t), \qquad t > 0, \tag{27}$$
$$u(x,0) = \eta(x), \qquad -1 \leq x \leq 1. \tag{28}$$

the constant term a is called advection speed. Given the periodic boundary conditions and the constant advection speed solution arises in the form

$$u(x,t) = f(x - at) \tag{29}$$

(see the assignment handout). In this section f is assumed to be $f(x) = \sin(2\pi x)$ and $a = 0.5$. One could then write the solution as $u(x,t) = \sin(2\pi x - \pi t)$.

2.1 Stability of the FTBS scheme

The numerical scheme that will be used is called *Upwind*. It approximates the spatial derivative using only previously obtained information. Refering to LeVeque, the upwind method is used for positive values of a in the advection problem, i.e. the solution travels from left to right. If a was negative the downwind scheme would have been used.

Definition

The upwind scheme has non-symmetric approximations to derivatives (one-sided approximations, backwards space). Taking the forward time differencing the method is defined as:

$$U_j^{n+1} = U_j^n - \underbrace{a\frac{\Delta t}{\Delta x}}_{\nu}(U_j^n - U_{j-1}^n) \tag{30}$$

where Δt and Δx are the time and space steps respectively, and ν is the Courant number.

Stability – von Neumann analysis

In order to perform the analysis the term U_j^n is rewritten as $g(\xi)^n e^{i\xi jh}$ (see LV 10.5), where i is the imaginary number, h is the space step (sometimes referred to as Δx) and for convenience $\theta = \xi\Delta x$. The wave number ξ is defined as $\frac{2\pi}{L}$

and $g(\xi)$ is called the amplification factor. Substituting into the upwind method definition:

$$\begin{aligned} g(\xi)^{n+1}e^{i\xi jh} &= g(\xi)^n e^{i\xi jh} - \nu\left(g(\xi)^n e^{i\xi jh} - g(\xi)^n e^{i\xi(j-1)h}\right) \\ g(\xi)^{n+1} &= g(\xi)^n - \nu g(\xi)^n\left(1 - e^{-i\xi h}\right) \\ g(\xi) &= 1 - \nu(1 - e^{-i\theta}) \\ g(\xi) &= 1 - \nu + \nu e^{-i\theta} \\ g(\xi) &= 1 - \nu + \nu\left(\cos(\theta) - i\sin(\theta)\right) \end{aligned} \tag{31}$$

Requiring the amplification factor to be bound by 1 in magnitude produces the stability bounds for the method (LV 10.5). In this case it is clear that $g(\xi)$ is a circle centered at $1 - \nu$ with radius equal to the Courant number. To satisfy the aforementioned bounds $g(\xi)$ needs to be contained in a unit circle in the complex plane, which is only possible when the Courant number is between 0 and 1 (inclusive).

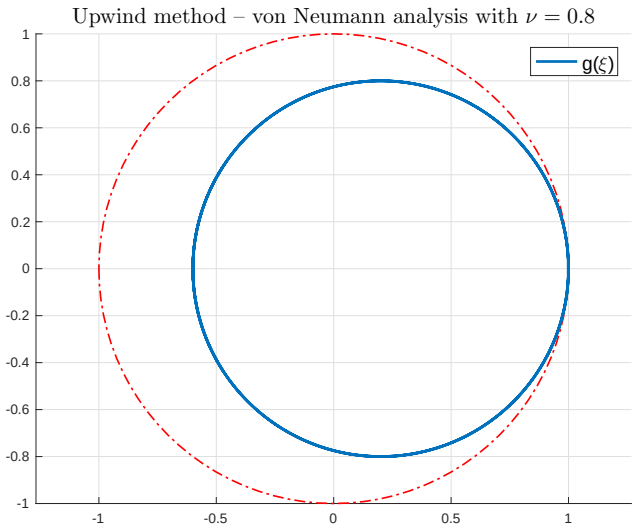

Figure 5: $g(\xi)$ evaluated on the complex plane for $\nu = 0.8$ and $\Delta x = \frac{1}{100}$. The red dashed circle indicates the unit circle and the blue $g(\xi)$ is clearly within bounds, which confirms results of the previous analysis.

2.2 Matlab implementation of FTBS and convergence

The Matlab implementation is rather straight forward and shown in the listing below with comments provided in the code.

The choice of the input arguments can vary as there are multiple ways of calculating the parameters. For example it would be trivial to change the method signature to accept the Courant number ν instead of Δx.

```
function [ U, Utrue ] = upwind( uFn, a, dx, dt, xmin, xmax, tmax )
% uFn - used to calculate u(x,0)
% a, dx, dt - user passed, used calculate Courant number (method
    signitature can be changed to pass in Cr directly if needed)
% xmin, xmax - spatial domain
% tmax - run until this time

t = 0;
steps = tmax/dt;
cr = (a*dt/dx);

x = xmin:dx:xmax;
N = length(x) - 1;
j = 2:N+1;

u0 = uFn(x,0);
u = u0;
unext = u0;

U = zeros(steps+1, N+1);
U(1,:) = u0;

Utrue = U; % this will be later overwritten, add only u0 for now

for n=2:steps+1
    % Keep track of time for calculating the true solution
    t = t+dt;
    % Next u
    unext(j) = u(j) - cr*(u(j) - u(j-1));
    unext(1) = u(1) - cr*(u(1) - u(N)); % account for periodic BCs
    % Update the iterate and save
    u = unext;
    U(n,:) = unext;
    % Calculate the true u
    Utrue(n,:) = uFn(x,t);
end

end
```

Convergence of the solver

Results from the theory show the convergence rate will be $\mathcal{O}(\Delta t + \Delta x)$ because of the spatial approximation combined with the Euler's method (forward in time) as shown in slide 20 from Lecture 22. Let's try to verify that.

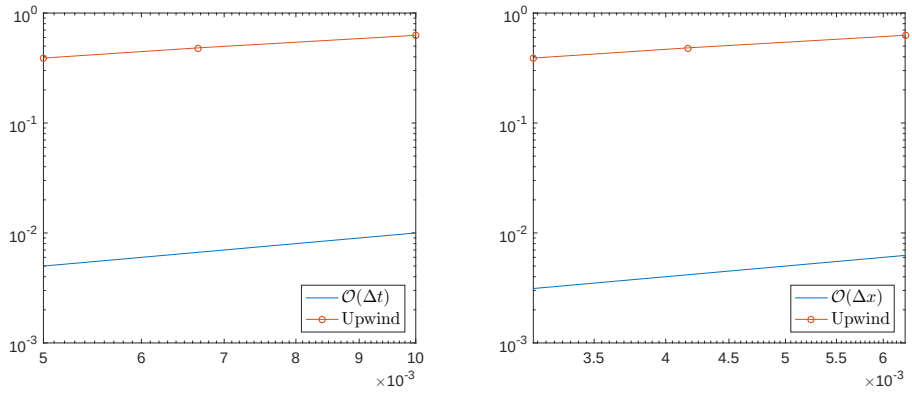

Figure 6: Convergence of the Upwind method shown on the advection problem from 2.3, left plot is showcasing $\mathcal{O}(\Delta t)$ while the right one depicts $\mathcal{O}(\Delta x)$. Courant number was fixed at $\nu = 0.8$. The theoretical convergence is shown as a line and the calculated $\mathcal{O}$ as circles indicating the evaluated Δt and Δx in the logarithmic scale.

2.3 Dispersion and diffusion

Diffusion refers to the amplitude and dispersion to the phase, according to the slides from Lecture 23 the amplification factor $g(\xi)$ is used in the following way:

$$\text{Diffusion} \quad |g(\xi)| = \sqrt{(1 - \nu + \nu\cos(\phi))^2 + (\nu\sin(\theta))^2} \tag{32}$$

$$\text{Dispersion} \quad y = \tan^{-1}\frac{\Im(g(\xi))}{\Re(g(\xi))} \tag{33}$$

Predicting dispersion and diffusion

Now the implementation of the upwind scheme will be tested and compared to the predicted dispersion and diffusion for the advection problem defined with the initial condition $u(x,t) = \sin(2\pi x)$ on a periodic domain with Courant number $\nu = 0.8$ after 40 wave periods. The handout also says to use 100 points per one wave length in order to calculate the wave.

Going back to the equation 31 it is clear that the next time step is resolved as $g(\xi)^{n+1} = g(\xi)^n\left(1 - \nu + \nu e^{-i\xi\Delta x}\right)$. For the given problem $\Delta x = \frac{1}{100}$, $\xi = 2\pi$ and $\nu = \frac{8}{10}$. Using Matlab's `abs` and `angle` the following results are obtained:

$$|g(\xi)|^{\frac{T}{\Delta t}} = |g(\xi)|^{\frac{80}{16/1000}} = |g(\xi)|^{5000} = 0.206157 \tag{34}$$

$$\frac{y}{y_{\text{true}}} = \frac{\tan^{-1}\frac{\Im(g(\xi))}{\Re(g(\xi))}}{\nu\theta} = 1.000078971174729 \tag{35}$$

Note that all the variables can be derived from the handout and using the definition of a sine wave moving to the right $y(x,t) = A\sin(kx - \omega t + \varphi) + D$, where $k = \frac{\omega}{v} = \frac{2\pi f}{v} = \frac{2\pi}{\lambda}$ (v – linear speed, λ – wave length). In this case u can be written as $u(x,t) = 1\sin(2\pi x - \pi t + 0) + 0$ and $v = a = 0.5$. Having all the necessary information the Matlab implementation of the upwind scheme can be ran to $t_{\max} = 80$, with $\nu = 0.8$, $a = 0.5$, $\Delta x = 1/100$ and $\Delta t = 16/1000$.

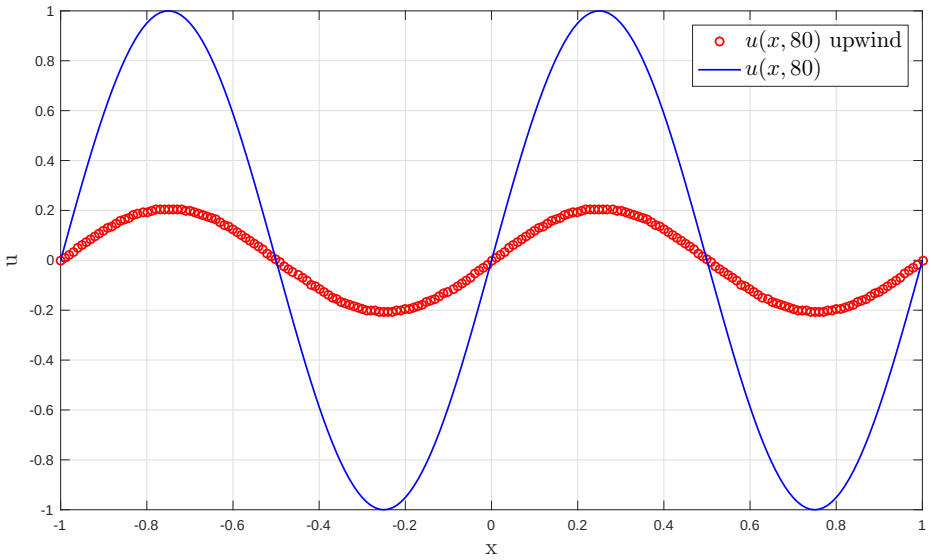

Figure 7: Upwind method – x vs u after 40 wave lengths for $\nu = 0.8$. The red points indicate the results from the implementation whereas the blue line is the true solution. Most notably the amplitude is damped to around 0.2 which corresponds to the theoretically obtained value. The phase difference is visibly 0, which can be also seen by zooming in at either multiples of $0.5x$ or at the peaks. It was also tried to find the phase difference by using Matlab's `xcorr` followed by `max` which also results in 0.

3 Non-linear advection-diffusion equation

Main focus of this section is going to be solving the viscous Burger's equation (VBE). As the title hints it contains parabolic and hyperbolic parts and is derived as a "toy" version from the family of non-linear hyperbolic equations with viscous terms (see 11.2 or E.15 in LV) in comparison to the well known Navier-Stokes equations.

$$\partial_t u(x,t) + \frac{1}{2}\partial_x \left(u(x,t)^2\right) = \varepsilon \partial_{xx} u(x,t), \qquad -1 < x < 1, t > 0, \tag{36}$$

$$u(-1,t) = g_L(t), \qquad t > 0, \tag{37}$$

$$u(1,t) = g_R(t), \qquad t > 0, \tag{38}$$

$$u(x,0) = \eta(x), \qquad -1 \leq x \leq 1. \tag{39}$$

The right hand side of equation 36 is called the viscous term (or heat conduction) and the term $\frac{1}{2}\partial_x \left(u(x,t)^2\right)$ refers to the flux function. The handout mentions the fact that the solution u is differentiable and the diffusion coefficient ε is constant and greater than zero. Another important piece of information from the handout is the solution to the IBV problem for appropriate ICs and BCs given as:

$$u(x,t) = -\tanh\left(\frac{x + 0.5 - t}{2\varepsilon}\right) + 1 \tag{40}$$

this function will be used to verify the numerical scheme that will proposed in the followin section.

3.1 Stable numerical scheme

Combining the numerical schemes from the first two sections gives rise to

$$U_j^{n+1} = U_j^n \left(1 - \frac{\Delta t}{\Delta x}\left(U_j^n - U_{j-1}^n\right)\right) + \frac{\varepsilon \Delta t}{\Delta x^2}\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right) \tag{41}$$

Von Neumann analysis was applied to this equation, however the term $g(\xi)$ could not be separated because of the multiplication in the $U_j^n(U_j^n - U_{j-1}^n)$ term, so it was decided to test out the stability by choosing Δt and Δx by hand which unfortunately did not lead anywhere.

Since the first scheme didn't yield much success, it was decided to find some similar scheme in the literature. C. Obertscheider's article on Burger's equations was chosen as a starting point and the scheme was modified to be:

$$U_j^{n+1} = U_j^n + \frac{\varepsilon \Delta t}{\Delta x^2}\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right) + \frac{1}{\Delta x}\left(f\left(U_{j+\frac{1}{2}}^n\right) - f\left(U_{j-\frac{1}{2}}^n\right)\right) \tag{42}$$

where f is the flux function $f(u) = \frac{1}{2}u^2$ and $f(U_{j\pm\frac{1}{2}}^n)$ denotes the average of $f(U_j^n)$ and $f(U_{j\pm1}^n)$.

3.2 Matlab implementation and convergence

The two schemes described were implemented in MATLAB although the second one seem to be more stable and produced better results.

```
function U = BurgerSolver4(boundaryFun,h,k,epsilon,tmax)
M = ceil(2/h);
N = ceil(tmax/k);
x = linspace(-1,1,M+1);
t = linspace(0,tmax,N+1);
U = zeros(M+1,N+1);
U(:,1) = boundaryFun(x,0,epsilon)';
U(1,:) = boundaryFun(-1,t,epsilon);
U(M,:) = boundaryFun(1,t,epsilon);
idx = 2:(M-1);
for n=2:N+1
    fm = 0.5*(U(2:M-1,n-1).^2 + U(1:M-2,n-1).^2);
    fp = 0.5*(U(2:M-1,n-1).^2 + U(3:M,n-1).^2);
    U(idx,n) = k*epsilon/h^2*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx+1,n-1))
        +...
        U(idx,n-1) - k/(2*h)*(fp-fm);
end
end
```

Convergence

Using the scheme given by equation 42 we compute the solution and calculated the local truncation error at the first point of the grid. Due to nonlinearities we were not able to obtain an expression for the stability of our methods. Hence, the grid was selected so that h and k satisfied the stability criteria of the FTCS solver found in the first exercise and the upwind method in the second exercise.

Figure 8 shows a convergence plot where the two help lines reveal that with the correction the model order seems to be $\mathcal{O}(h^3 + k^2)$.

3.3 Testing

To demonstrate the implementation proposed in the previous section a problem is formulated as follows: $g_L(t) = g_R(t) = 0$ (homogenous BCs) and $\eta(x) = -\sin(\pi x)$. Solution will be found using an uniform grid and with $\varepsilon = \frac{1}{100\pi}$. A figure will be presented demonstrating the solution at time $t_s = 1.6037/\pi$ along with value for $\partial_x u$ at $x = 0$, which should be -152.00516 according to the handout. This value will be calculated as $\frac{u_c(\Delta x,t_s)-u_c(-\Delta x,t_s)}{2\Delta x}$.

The derivative computed using the first scheme was not even close to the given value as can be seen in figure 9. This is due to the fact that the solver is unable to capture the stiff part.

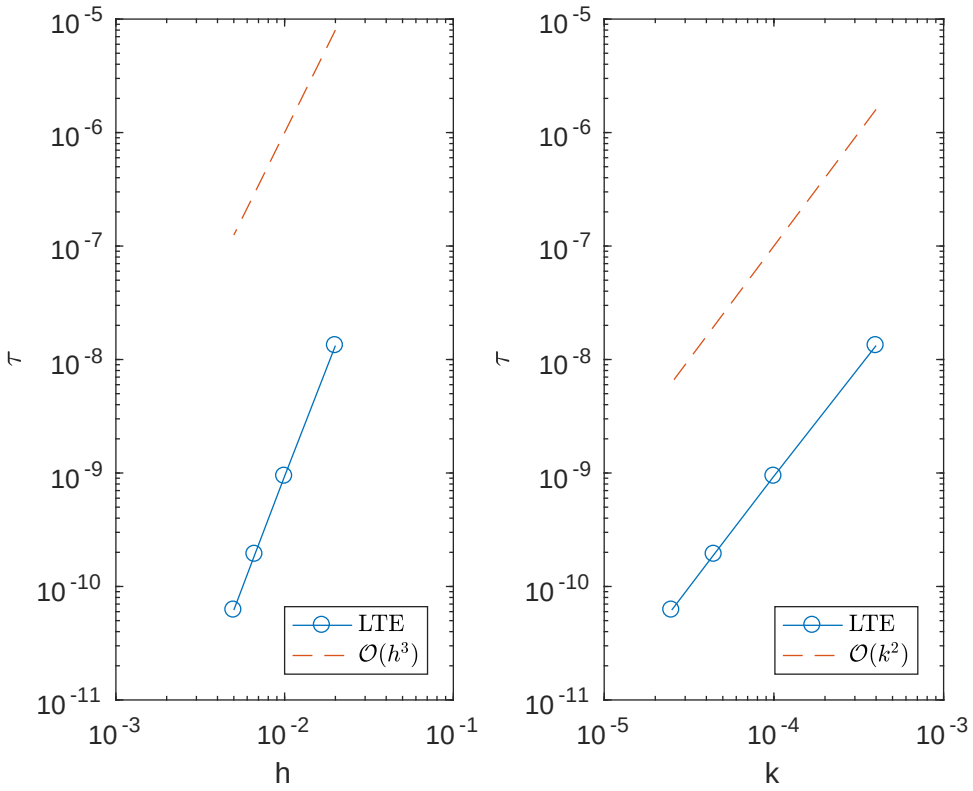

Figure 8: LTE for different values of h and k

The second approach yielded a value of around -132 when setting h to 0.005 and $k = h^2$.

The nonuniform grid solver that will be presented in the last section, just using 35 points for x produced a value of -150.5 for the derivative, which seems to be the best out of all solvers we have tried.

3.4 Approaches to maintain accuracy

As implementing a high-order accurate discretization method was too time consuming without leading to any results, it was decided to chose a non-uniform grid that captures the mid-points of the spatial domain in a finer manner.

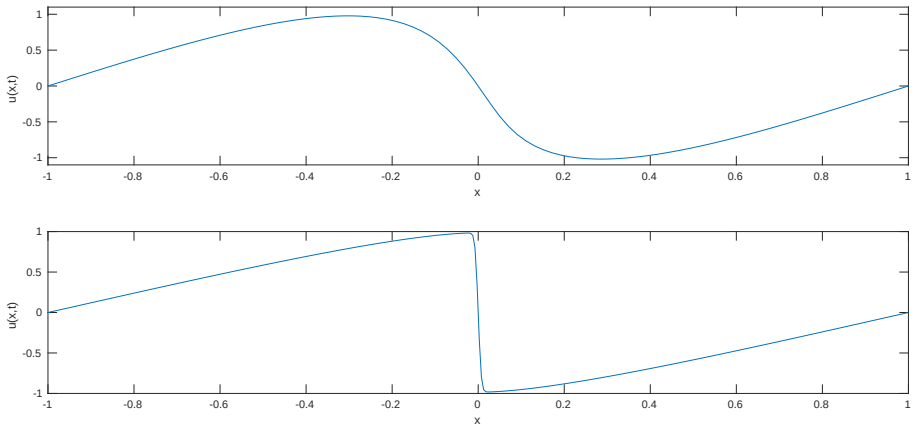

Figure 9: Solution at t=1.6037 using FTCS and upwind (top graph) FTCS and mid-point correction (bottom graph)

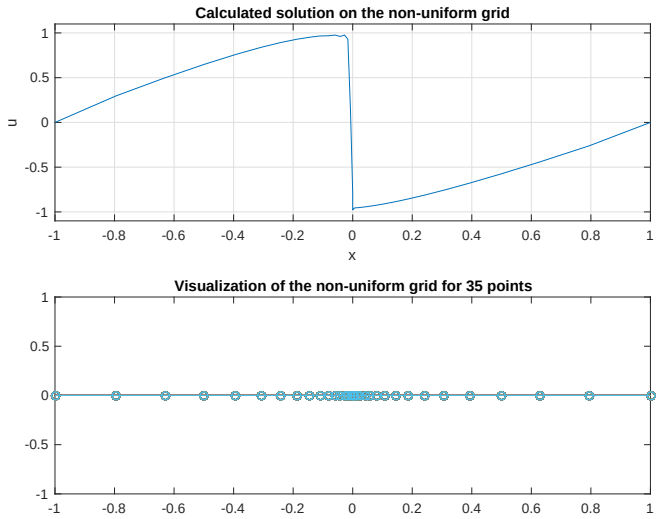

Figure 10: Non-uniform grid with 35 points concentrated mostly around 0 to try to capture the steep change of the slope

A Appendix

A.1 Exercise 1 code

A.1.1 Plots and local truncation errors

```
close all
clear
%% DIFFUSION PROBLEM
%% Implement the scheme
theta = 0.5;
h = 0.01;
k = 0.01;
mu = 1*k/h^2;
M = 2/h;
N = 1/k;
U = parabolicSolver(@boundaryFun,h,k,theta,mu);
x = linspace(-1,1,M+1);
t = linspace(0,1,N);
Utrue = zeros(M+1,N);
for j = 1:N
    Utrue(:,j) = boundaryFun(x,t(j));
end
subplot(2,2,1)
plot(x,U(:,1),'linewidth',1.6)
title('t=0')
subplot(2,2,2)
plot(x,U(:,5),'linewidth',1.6)
title('t=0.04')
subplot(2,2,3)
plot(x,U(:,20),'linewidth',1.6)
title('t=0.20')
subplot(2,2,4)
plot(x,U(:,50),'linewidth',1.6)
title('t=0.5')
print('c3','-dpng')
%% Demonstrate convergence

h = logspace(-2,-1,10);
k = 7/6*h.^2;
for i = 1:10
    theta(i) = 0.5 + h(i)^2/(12*k(i));
    mu = 1*k(i)/h(i)^2;
    M = ceil(2/h(i));
    N = ceil(1/k(i));
    U = parabolicSolver(@boundaryFun,h(i),k(i),theta(i),mu);
    x = linspace(-1,1,M+1);
    t = linspace(0,1,N+1);
```

```
    Utrue = zeros(M+1,N);
    for j = 1:2
        Utrue(:,j) = boundaryFun(x,t(j));
    end
    LTE(i) = max(norm(U(:,2)-Utrue(:,2),2));
end

%% LTE plots
figure
subplot(1,2,1)
loglog(h,LTE,'o-')
xlabel('h')
ylabel('\tau')
hold on
p1 = loglog(h,h.^2,'--');
legend({'LTE','$\mathcal{O}(h^2)$'},'Interpreter','latex','location',
    'northwest')
subplot(1,2,2)
loglog(k,LTE,'-o')
hold on
loglog(k,k.^2,'--')
xlabel('k')
ylabel('\tau')
legend({'LTE','$\mathcal{O}(k^2)$'},'Interpreter','latex','location',
    'southeast')
print('c3','-dpng')
```

A.1.2 Parabolic solver

```
function U = parabolicSolver(boundaryFun,h,k,theta,mu)
% The following function solves the heat diffusion equation using the
% theta-scheme. It takes as input the two step sizes h and k and the
% parameters theta and mu and returns the solution at every grid
    point
% Define the grid size
M = ceil(2/h);
N = ceil(1/k);
% Build A0
A0 = diag(-2*ones(M+1,1)) + diag(ones(M,1),-1) + diag(ones(M,1),1);
A0(1,:) = zeros(1,M+1);
A0(end,:) = zeros(1,M+1);
I = eye(M+1);
x = linspace(-1,1,M+1);
t = linspace(0,1,N+1);
% Obtain initial and boundary conditions
U(:,1) = boundaryFun(x,0);
g = [boundaryFun(-1,t); zeros(M-1,N+1); boundaryFun(1,t)];
% Solve the ODEs system
for k=2:N+1
```

```
20      U(:,k) = (I-theta*mu*A0)\((I+(1-theta)*mu*A0)*U(:,k-1)+g(:,k)-g
            (:,k-1));
21  end
22  end
```

A.1.3 Boundary function

```
1   function u = boundaryFun(x,t)
2
3   alpha = [1 4 16];
4   u = 0;
5   for i = 1:3
6       u = u + cos(alpha(i)*x)*exp(-alpha(i)^2*t);
7   end
8
9   end
```

A.2 Exercise 2 code

A.2.1 Upwind method implementation

```
1   function [ U, Utrue ] = upwind( uFn, a, dx, dt, xmin, xmax, tmax )
2   % uFn - used to calculate u(x,0)
3   % a, dx, dt - user passed, used calculate Courant number (method
        signitature can be changed to pass in Cr directly if needed)
4   % xmin, xmax - spatial domain
5   % tmax - run until this time
6
7   % For this problem
8   % a = 1/2;
9   % dx = 1/100;
10  % dt = 16/1000;
11  % xmin = -1;
12  % xmax = 1;
13
14  t = 0;
15  steps = tmax/dt;
16  cr = (a*dt/dx);
17
18  x = xmin:dx:xmax;
19  N = length(x) - 1;
20  j = 2:N+1;
21
22  u0 = uFn(x,0);
23  u = u0;
24  unext = u0;
25
26  U = zeros(steps+1, N+1);
```

```
U(1,:) = u0;

Utrue = U; % this will be later overwritten anyway

for n=2:steps+1
    % Keep track of time for calculating the true solution
    t = t+dt;
    % Next u
    unext(j) = u(j) - cr*(u(j) - u(j-1));
    unext(1) = u(1) - cr*(u(1) - u(N)); % account for periodic BCs
    % Update the iterate and save
    u = unext;
    U(n,:) = unext;
    % Calculate the true u
    Utrue(n,:) = uFn(x,t);
end


end
```

A.2.2 Convergence and the problems from the handout

```
%% von Neumann analysis of the Upwind method
v = 0.8; % Courant number
g = @(xi,h) 1 - v + v*exp(-1i*xi*h);
figure(1);
clf;
rectangle('Position',[-1 -1 2 2],'Curvature',[1 1], 'EdgeColor', 'r',
    'LineStyle', '-.', 'LineWidth', 1.2);
hold on;
plot(g(-1000:0.01:1000,0.01), 'LineWidth', 2); % verify that values
   are bound within a unit circle
hold off;
axis equal;
grid on;
title('Upwind method -- von Neumann analysis with $\nu = 0.8$', '
   Interpreter', 'latex', 'FontSize', 16);
legend({'g(\xi)'}, 'FontSize', 14);

%% Define the problem
f = @(x) sin(2*pi*x);
cr = 0.8;
u = @(x,t) f(x-0.5*t);

a=1/2;
wave_periods = 40;
n_wave_length = 100;
x = linspace(-1,1,100+2); % \Delta x = 1/100
t = linspace(0,a^(-1)*wave_periods,a^(-1)*n_wave_length*wave_periods)
```

```
    ;

[X,T] = meshgrid(x,t);
U = u(X,T);
U(:,1) = u(1,t); % at fixed x
U(:,end) = u(-1,t); % at fixed x
for k=1:(2*n_wave_length):length(t)
    U(k,:) = u(x,0); % periodic BCs
end

%% Animation of the solution at a const speed
figure
h = animatedline;
axis([-1 1 -1 1])
legend('');

for k = 1:length(t)
    h.DisplayName = num2str(round(T(k,1), 2));
    clearpoints(h)
    addpoints(h,X(k,:),U(k,:))
    drawnow
end

%% Upwind
ShowMovingSolution = false;
nWavePeriods = 40;

a = 1/2;          % speed
f = 1/2;          % frequency
T = f^(-1);       % period
lambda = 1;       % wave length
dx = 1/100;
dt = 16/1000;
Cr = a*(dt/dx);
xmin = -1;
xmax = 1;
t = 0;
tmax = nWavePeriods*T; % tmax = 80 s
steps = tmax/dt;

x = xmin-dx:dx:xmax+dx;


[Uup, Utrue] = upwind( u, a, dx, dt, xmin, xmax, tmax );
[nsteps, nx] = size(Uup);

% Indices of 1-40 wave periods
uIdxRange = 1:(steps/nWavePeriods):steps+1;
[um,un] = size(Uup);
```

```
xx = linspace(xmin,xmax,un);

figure(5);
plot(xx, Uup(uIdxRange(end), :), 'ro',...
     xx, Utrue(1,   :), 'b-',...
    'LineWidth', 1.2)
legend({'$u(x,80)$ upwind', '$u(x,80)$'}, 'Interpreter', 'latex', '
   FontSize', 16, 'Location', 'northeast')
xlabel('x', 'Interpreter', 'latex', 'FontSize', 16);
ylabel('u', 'Interpreter', 'latex', 'FontSize', 16);
grid on;


if ShowMovingSolution
    figure(6); clf;
    % Show step to t=3.6s
    for n=1:(3.6/dt)
        t = t+dt;
        figure(7);
        plot(x(2:end-1),Uup(n,2:end-1),'bx-'); hold on
        plot(x(2:end-1),Utrue(n,2:end-1),'g-','LineWidth',2); hold
            off
        axis([-1 1 -1 1]);
        title(sprintf('Time = %2.2f s', t))
        pause(0.0001);
    end
end

%% Analysis of the errors
nWavePeriods = 40;

a = 1/2;          % speed
f = 1/2;          % frequency
T = f^(-1);       % period
lambda = 1;       % wave length
dx = 1/100;
dt = 16/1000;
Cr = 0.8; % fixed
xmin = -1;
xmax = 1;
t = 0;
tmax = nWavePeriods*T; % tmax = 80 s
steps = tmax/dt;
err = [];
dxs = []; dts = [];
g = @(phi) 1 - Cr + Cr*exp(1i*phi); % phi = xi*dx
xi = 2*pi/1; % 2*pi/L, L = 1

fprintf('Running the convergence check loop...\n');
```

```
for vdt = [1/100 1/150 1/200 1/250 1/300]
    vdx = a*vdt/Cr; % Cr = a*(dt/dx) => dx = a*dt/Cr
    fprintf('Using Cr = %f, dt = %f, dx = %f\n', a*vdt/vdx, vdt, vdx)
        ;
    [Uup, Utrue] = upwind( u, a, vdx, vdt, xmin, xmax, tmax );
    err = [err max(abs(Uup(end,:)-Utrue(end,:)))];
    dxs = [dxs vdx]; dts = [dts vdt];
end
figure(1); clf;
subplot(1,2,1);
loglog(dts,dts,dts,err,'o-');
legend({'$\mathcal{O}(\Delta t)$', 'Upwind'},'Interpreter','latex','
    location','southeast','FontSize',16)
set(gca, 'FontSize', 14);
subplot(1,2,2);
loglog(dxs,dxs,dxs,err,'o-');
legend({'$\mathcal{O}(\Delta x)$', 'Upwind'},'Interpreter','latex','
    location','southeast','FontSize',16)
set(gca, 'FontSize', 14);

%% Finding the phase difference
[c,lag] = xcorr(Uup(end,:), Utrue(1,:));
[maxC,cIdx]=max(c);
fprintf('Phase difference is %e\n', lag(cIdx));
```

A.2.3 Von Neumann results

```
v = 0.8; % Courant number
% g(xi)
% a circle, centered at 1-cr with radius cr
g = @(phi) 1 - v + v*exp(1i*phi); % phi = xi*dx

xi = 2*pi/1; % 2*pi/L, L = 1
phi = xi*0.01; % xi*dx, dx = 1/100

G = g(phi);

time_steps = 80/(16/1000);

fprintf(' Von Neumann analysis\n%s\n------\n', 'g(xi) = 1-v-v*(cos(xi
    *dx) - 1i*sin(xi*dx)); phi = xi*dx');
fprintf('Courant number = %f\n', v);
fprintf('xi = %f\n', xi);
fprintf('phi = %f\n', phi);
fprintf('g(2*pi): Re = %f, Im = %f\n', real(G), imag(G));
fprintf('Abs(g(xi)) = %f\n', abs(G));
fprintf('Angle(g(xi)) = %f radians, %f degrees\n', angle(G), rad2deg(
    angle(G)));

```

```
fprintf('------\nTime steps = %d\n', time_steps)
fprintf('Diffusion = %f\n', abs(G^time_steps));
fprintf('y = %f radians, %f degrees\n', angle(G), rad2deg(angle(G)));
fprintf('y_exact = -Cr*phi = %f\n', -v*phi);
```

A.3 Exercise 3 code

A.3.1 Implementations of the solvers

There were multiple solvers that we've tried so we decided to include all of them, even the one that didn't work well (Solver 3 Crank-Nicolson attempt). **First solver**

```
function U = BurgerSolver(boundaryFun,h,k,epsilon,tmax)
M = ceil(2/h);
N = ceil(tmax/k);
x = linspace(-1,1,M);
t = linspace(0,tmax,N);
U = zeros(M,N);
U(:,1) = boundaryFun(x,0,epsilon)';
U(1,:) = boundaryFun(-1,t,epsilon);
U(M,:) = boundaryFun(1,t,epsilon);
idx = 2:(M-1);
for n=2:N
    U(idx,n) = k*epsilon/h^2*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx+1,n-1))
        +...
        U(idx,n-1) - k/h*U(idx,n-1).*(U(idx,n-1)-U(idx-1,n-1));
end
[T,X] = meshgrid(t,x);
mesh(X,T,U)
end
```

Second solver

```
function [U,t,x] = BurgerSolver2(h,k,epsilon,tmax)
M = ceil(2/h);
N = ceil(tmax/k);
x = linspace(-1,1,M);
t = linspace(0,tmax,N);
M = length(x);
U = zeros(M,N);
U(:,1) = -sin(pi*x);
U(1,:) = 0;
U(M,:) = 0;
idx = 2:(M-1);
for n=2:N
    U(idx,n) = k*epsilon/h^2*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx+1,n-1))
        +...
        U(idx,n-1) - k/h*U(idx,n-1).*(U(idx,n-1)-U(idx-1,n-1));
```

```
end
end
```

Third solver – CN not working

```
function U = BurgerSolver3(boundaryFun,h,k,epsilon,tmax)
M = ceil(2/h);
N = ceil(tmax/k);
x = linspace(-1,1,M+1);
t = linspace(0,tmax,N+1);
U = zeros(M+1,N+1);
% Build A0
A0 = diag(-2*ones(M+1,1)) + diag(ones(M,1),-1) + diag(ones(M,1),1);
A0(1,:) = zeros(1,M+1);
A0(end,:) = zeros(1,M+1);
% Build B0
B0 = diag(-1*ones(M+1,1)) + diag(ones(M,1),1);
B0(1,:) = zeros(1,M+1);
B0(end,:) = zeros(1,M+1);
I = eye(M+1);
% Obtain initial and boundary conditions
U(:,1) = boundaryFun(x,0,epsilon);
g = [boundaryFun(-1,t,epsilon); zeros(M-1,N+1); boundaryFun(1,t,
    epsilon)];
for k=2:N+1
    U(:,k) = (I-0.5*epsilon*k/h^2*A0)\((I+0.5*epsilon*k/h^2*A0)*U(:,k
        -1)-U(:,k-1).*(k/h*B0*U(:,k-1))+g(:,k)-g(:,k-1));
end
[T,X] = meshgrid(t,x);
mesh(X,T,U)
end
```

Fourth solver

```
function U = BurgerSolver4(boundaryFun,h,k,epsilon,tmax)
M = ceil(2/h);
N = ceil(tmax/k);
x = linspace(-1,1,M+1);
t = linspace(0,tmax,N+1);
U = zeros(M+1,N+1);
U(:,1) = boundaryFun(x,0,epsilon)';
U(1,:) = boundaryFun(-1,t,epsilon);
U(M,:) = boundaryFun(1,t,epsilon);
idx = 2:(M-1);
for n=2:N+1
    fm = 0.5*(U(2:M-1,n-1).^2 + U(1:M-2,n-1).^2);
    fp = 0.5*(U(2:M-1,n-1).^2 + U(3:M,n-1).^2);
    U(idx,n) = k*epsilon/h^2*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx+1,n-1))
        +...
        U(idx,n-1) - k/(2*h)*(fp-fm);
end
```

```
17 end
```

Fifth solver

```
1  function [U,t,x] = BurgerSolver5(h,k,epsilon,tmax)
2  M = ceil(2/h);
3  N = ceil(tmax/k);
4  x = linspace(-1,1,M);
5  t = linspace(0,tmax,N);
6  M = length(x);
7  U = zeros(M,N);
8  U(:,1) = -sin(pi*x);
9  U(1,:) = 0;
10 U(M,:) = 0;
11 idx = 2:(M-1);
12 for n=2:N
13     fm = 0.5*(U(2:M-1,n-1).^2 + U(1:M-2,n-1).^2);
14     fp = 0.5*(U(2:M-1,n-1).^2 + U(3:M,n-1).^2);
15     U(idx,n) = k*epsilon/h^2*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx+1,n-1))
           +...
16         U(idx,n-1) - k/(2*h)*(fp-fm);
17 end
18 end
```

Boundary function used

```
1  function u = boundaryFun(x,t,epsilon)
2  % This function allows to evaluate the solution of the viscous Burger
       's
3  % equation at the boundaries
4      u = -tanh((x+0.5-t)/(2*epsilon)) + 1;
5  end
```

A.3.2 Plotting and driver code used throughout Ex3

```
1  %% Non-linear advection-diffusion equation
2  h = 0.005;
3  k = h^2;
4  epsilon = 0.5;
5  tmax = 1;
6  U1 = BurgerSolver(@boundaryFun,h,k,epsilon,tmax);
7  %% true solution
8  figure
9  x = linspace(-1,1,100);
10 t = linspace(0,1,100);
11 [X,T] = ndgrid(x,t);
12 U2 = boundaryFun(X,T,epsilon);
13 mesh(X,T,U2)
14
```

```
%% Demonstrate convergence

h = [1/50 1/100 1/150 1/200];
k = h.^2;
for i = 1:4
    theta(i) = 0.5 + h(i)^2/(12*k(i));
    mu = 1*k(i)/h(i)^2;
    M = ceil(2/h(i));
    N = ceil(1/k(i));
    U =BurgerSolver4(@boundaryFun,h(i),k(i),epsilon,tmax);
    x = linspace(-1,1,M+1);
    t = linspace(0,1,N+1);
    [X,T] = ndgrid(x,t);
    Utrue = boundaryFun(X,T,epsilon);
    LTE(i) = max(norm(U(2,2)-Utrue(2,2),2));
end

%% LTE plots
figure
subplot(1,2,1)
loglog(h,LTE,'o-')
xlabel('h')
ylabel('\tau')
hold on
p1 = loglog(h,h.^3,'--');
legend({'LTE','$\mathcal{O}(h^3)$'},'Interpreter','latex','location',
    'southeast')
subplot(1,2,2)
loglog(k,LTE,'-o')
hold on
loglog(k,k.^2*10,'--')
xlabel('k')
ylabel('\tau')
legend({'LTE','$\mathcal{O}(k^2)$'},'Interpreter','latex','location',
    'southeast')
print('c3','-dpng')
%% 3.3

h = 0.005;
k = 1/2*h^2;
tmax = 1.6037/pi;
xspan = [-1 1];
epsilon = 0.01/pi;
%% FTCS and upwind
[U1,t1,x1] = BurgerSolver2(h,k,epsilon,tmax);
plot(x1,U1(:,end))
% find closest point to 0
[~,idx] = min(abs(x2));
dx1 = (U(idx+1,end)-U(idx,end))/h;
```

```
%% Trapezoidal
[U2,t2,x2] = BurgerSolver5(h,k,epsilon,tmax);
[T,X] = meshgrid(t,x);
plot(x2,U2(:,end))
% find closest point to 0
[~,idx] = min(abs(x2));
dx2 = (U(idx+1,end)-U(idx,end))/h;
```

A.3.3 Non-uniform grid experiments

Visualizations and driver for non-uniform grid implementation

```
%% Yet another nonuniform grid
h = 0.003;
k = h^2;
tmax = 1.6037/pi;
xspan = [-1 1];
epsilon = 0.01/pi;

nonunifpoints = 35;

[Unu,tnu,xnu] = BurgerSolverNonUniform(nonunifpoints,k,epsilon,tmax);

idx = floor(length(xnu)/2); % mid point x = 0
dxnu = (Unu(idx+1,end)-Unu(idx-1,end))/(xnu(idx+1)-xnu(idx-1));
fprintf('Non uniform grid -> dx_nu = %f\n', dxnu);

figure(1);
% Show the solution
subplot(2,1,1);
plot(xnu,Unu(:,end));
title('Calculated solution on the non-uniform grid');
axis([-1 1 -1.1 1.1]); grid on;
xlabel('x'); ylabel('u');
% Show the grid
subplot(2,1,2);
plot(xnu,zeros(length(xnu)),'o-');
title(sprintf('Visualization of the non-uniform grid for %d points',
    nonunifpoints));
```

Non-uniform grid generator

```
function [ combined ] = nonuniformgrid( points )
% Generate non-uniform grid between -1 and 1, where most of the
    points are
% centered around 0.
% Number of points has to be even

if mod(points,2) == 1
    points = points - 1;
end
```

```
n = points/2;
r = 0.8;

lg = [0, (r - 1)/(r^(n-1) - 1)*cumsum(r.^(0:(n-2)))];
rg = 1+[0, (r - 1)/(r^(n-1) - 1)*cumsum(r.^((n-2):-1:0))];

if mod(points,2) == 1
    combined = [lg(:); 0; rg(:)]'; % combine left and right vectors
else
    combined = [lg(:); rg(:)]'; % combine left and right vectors
end
combined = combined - 1;    % center around 0

% Uncomment to visualize the spacing
%figure(1);
%plot(combined, zeros(length(combined)), 'o-');

end
```

Burger solver implementing the non-uniform grid

```
function [U,t,x] = BurgerSolverNonUniform(xNumPts, k,epsilon,tmax)
% Not sure how to deal with nonuniform grid so the implementation is
% adjusted to a bit so it doesn't blow up when deviding by zeros.
%
M = xNumPts; % 667 for example
N = ceil(tmax/k);
x = nonuniformgrid(M);
t = linspace(0,tmax,N);
M = length(x);
U = zeros(M,N);
U(:,1) = -sin(pi*x);
U(1,:) = 0;
U(M,:) = 0;
idx = 2:(M-1);
h = 0.01; % 0.01 for example

diffx = diff(x);
diffx(diffx < 1e-2) = 0.009; % replace small differences with a value
    for numerical stability

for n=2:N
    fm = 0.5*(U(2:M-1,n-1).^2 + U(1:M-2,n-1).^2);
    fp = 0.5*(U(2:M-1,n-1).^2 + U(3:M,n-1).^2);
    %U(idx,n) = k*epsilon/h^2*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx+1,n-1)
        ) +...
    %    U(idx,n-1) - k/(2*h)*(fp-fm);
    M1 = k*epsilon*diag(diffx.^-2); %k*epsilon/h^2;
    M2 = 0.5*k*diag(diffx.^-1);%k/(2*h);
    U(idx,n) = M1(1:end-1,1:end-1)*(U(idx-1,n-1)-2*U(idx,n-1)+U(idx
```

```
          +1,n-1)) +...
28          U(idx,n-1) - M2(1:end-1,1:end-1)*(fp-fm);
29  end
30  end
```