Time-Marching Methods for ODEs

Devin Crossman
Student ID: 27490623

Concordia University, April 2018

Prepared for
Dr. Marius Paraschivoiu & Dr. Brian Vermeire
AERO 455, Winter 2018

Contents

List of Figures

1 Introduction

The objective of this assignment was to become more familiar with programming numerical solutions to partial differential equations (PDEs). The specific PDE that was used was the one-dimensional first order wave equation given by the following equation:

$$\frac{\delta u}{\delta t} + c\frac{\delta u}{\delta x} = 0 \tag{1}$$

This linear hyperbolic equation is also known as the one-dimensional advection equation and can be thought of as how some quantity or property u changes with time and by the flow of fluid in one direction. An easy analogy could be letting u be the concentration of a substance dissolved in a fluid. If the fluid is flowing in the x-direction and the substance is added in pulses from a source at one point, the concentration will vary as a function of time t and position x. The speed of the fluid flow is given by the constant c. The PDE given by equation 1 was required to be solved on the domain [0, 1] with periodic boundary conditions and the initial condition given in equation 2.

$$u(x, 0) = e^{-0.5[(x-0.5)/0.08]^2} \tag{2}$$

The task was to write a program to find the solution of this initial value problem (IVP) using a grid of 50 points and a variety of time-marching methods. In part 1, a 2nd order central finite difference in space was used with explicit Euler, implicit Euler, Trapezoidal (2nd order Adams-Moulton), 2nd order Adams-Bashforth, and 4th order Runge-Kutta time-marching methods. In part 2, a 4th order central finite difference in space was used with the 4th order Runge-Kutta time-marching method.

Satisfying the Courant-Friedrichs-Lewy (CFL) condition is necessary for stable convergence of a numerical solution for a PDE. It relates the time interval to the spatial interval and flow speed. For stable convergence the CFL condition specifies that the Courant number $\left|c\frac{\Delta t}{\Delta x}\right|$ is less than some calculated value that depends on the solution method. A Courant number of 0.1 was used for the solutions using explicit Euler and 2nd order Adams-Bashforth while a Courant number of 1 was used for the other methods.

The program was written in Python using the NumPy module to perform efficient linear algebra calculations using vectors and matrices, and the Matplotlib module to plot the results. A graphical user interface (GUI) was created using the TkInter package and safe interpretation of user input for the initial conditions was achieved with the NumExpr package.

2 Derivation of the equations

To apply one of the time stepping methods the PDE must first be turned into an ordinary differential equation (ODE) by discretizing the spatial domain while keeping the time step constant. One way to accomplish this is by using the Taylor series. The Taylor series is useful because it gives an approximation of the order of the truncation error. Using equal spacing between grid points on the x-axis, a second order central difference approximation of $\frac{du}{dx}$ was obtained by combining the Taylor series expansion about u_{j-1} and u_{j+1} where $u_j = u(x_j)$ in the following manner:

$$u_{j+1} = u_j + \frac{\Delta x}{1!}\frac{du}{dx} + \frac{(\Delta x)^2}{2!}\frac{d^2u}{dx^2} + \frac{(\Delta x)^3}{3!}\frac{d^3u}{dx^3} + \ldots + \frac{(\Delta x)^n}{n!}\frac{d^nu}{dx^n} \tag{3}$$

$$u_{j-1} = u_j - \frac{\Delta x}{1!}\frac{du}{dx} + \frac{(\Delta x)^2}{2!}\frac{d^2u}{dx^2} - \frac{(\Delta x)^3}{3!}\frac{d^3u}{dx^3} + \ldots + \frac{(\Delta x)^n}{n!}\frac{d^nu}{dx^n} \tag{4}$$

Subtracting 4 from 3

$$u_{j+1} - u_{j-1} = 2\Delta x\frac{du}{dx} + \frac{2(\Delta x)^3}{3}\frac{d^3u}{dx^3} + \ldots \tag{5}$$

Neglecting the higher order differential terms and solving for $\frac{du}{dx}$

$$\frac{du}{dx} = \frac{u_{j+1} - u_{j-1}}{2\Delta x} + O((\Delta x)^2) \tag{6}$$

Equation 1 has now been transformed into a first order ODE:

$$\frac{du}{dt} = -c\frac{u_{j+1} - u_{j-1}}{2\Delta x} = f(t, u) \tag{7}$$

2.1 Explicit Euler Method

The explicit Euler method can be derived using the Taylor series to obtain a forward difference approximation for the time derivative about point x_j using the values of the spatial derivative at time step n. Using the superscript notation to represent the time step and a constant time interval Δt, equation 7 becomes:

$$\left.\frac{du}{dt}\right|_{x_j} = \frac{u_j^{n+1} - u_j^n}{\Delta t} = -c\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} = f(t, u) \tag{8}$$

Solving equation 8 for u_j^{n+1} we get an explicit equation for u^{n+1} based on the values at the previous time step u^n.

$$u_j^{n+1} = u_j^n - \frac{c\Delta t}{2\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{9}$$

2.2 Implicit Euler Method

The implicit Euler method is similar except it evaluates the spatial derivative at time step $n+1$ giving:

$$u_j^{n+1} = u_j^n - \frac{c\Delta t}{2\Delta x}(u_{j+1}^{n+1} - u_{j-1}^{n+1}) \tag{10}$$

Moving all u^{n+1} terms to the left side of the equation:

$$u_j^{n+1} + \frac{c\Delta t}{2\Delta x}(u_{j+1}^{n+1} - u_{j-1}^{n+1}) = u_j^n \tag{11}$$

Substituting values of j for the 50 spatial grid points leads to a system of equations given by the following matrix equation:

$$\begin{bmatrix} 1 & \frac{c\Delta t}{2\Delta x} & 0 & \dots & 0 & -\frac{c\Delta t}{2\Delta x} \\ -\frac{c\Delta t}{2\Delta x} & 1 & \frac{c\Delta t}{2\Delta x} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & & & -\frac{c\Delta t}{2\Delta x} & 1 & \frac{c\Delta t}{2\Delta x} \\ \frac{c\Delta t}{2\Delta x} & 0 & \dots & 0 & -\frac{c\Delta t}{2\Delta x} & 1 \end{bmatrix} \begin{bmatrix} u_0^{n+1} \\ u_1^{n+1} \\ \vdots \\ \vdots \\ u_{49}^{n+1} \\ u_{50}^{n+1} \end{bmatrix} = \begin{bmatrix} u_0^n \\ u_1^n \\ \vdots \\ \vdots \\ u_{49}^n \\ u_{50}^n \end{bmatrix} \tag{12}$$

With the assumed periodic boundary condition, where $j = 0$, u_{j-1} is assumed to be u_{50} and where $j = 50$, u_{j+1} is assumed to be u_0.

2.3 Trapezoidal Method

The trapezoidal method is an implicit method also known as the 2nd order Adams-Moulton method. It can be derived by integrating equation 7 between time steps t^n and t^{n+1} at x_j:

$$\int_{t^n}^{t^{n+1}} \left.\frac{du}{dt}\right|_{x_j} dt = \int_{t^n}^{t^{n+1}} f(t,u)dt \tag{13}$$

The fundamental theorem of calculus gives us the left hand side of equation 13:

$$\int_{t^n}^{t^{n+1}} \left.\frac{du}{dt}\right|_{x_j} dt = u_j(t^n) - u_j(t^{n+1}) = u_j^{n+1} - u_j^n \tag{14}$$

The right side of equation 13 is evaluated by using a linear approximation of the integral between t^n and t^{n+1}. Using the approximation, the value of the integral is equal to the area of a triangle with base $\Delta t = t^{n+1} - t^n$ and height $u^{n+1} - u^n$:

$$\begin{aligned}\int_{t^n}^{t^{n+1}} f(t,u)dt &= \frac{1}{2}\Delta t(f(t^{n+1},u^{n+1}) - f(t^n,u^n) \\ &= \frac{-c\Delta t}{4\Delta x}[(u_{j+1}^{n+1} - u_{j-1}^{n+1}) - (u_{j+1}^n - u_{j-1}^n)]\end{aligned} \tag{15}$$

Substituting equations 14 and 15 back into equation 13 and rearranging to have all u^{n+1} terms on the left side and all u^n terms on the right:

$$u_j^{n+1} + \frac{c\Delta t}{4\Delta x}(u_{j+1}^{n+1} - u_{j-1}^{n+1}) = u_j^n - \frac{c\Delta t}{4\Delta x}(u_{j+1}^n - u_{j-1}^n) \tag{16}$$

After substituting values of j for the spatial grid nodes, equation 16 can be solved using the matrix equation:

$$\begin{aligned}&\begin{bmatrix} 1 & \frac{c\Delta t}{4\Delta x} & 0 & \dots & 0 & -\frac{c\Delta t}{4\Delta x} \\ -\frac{c\Delta t}{4\Delta x} & 1 & \frac{c\Delta t}{4\Delta x} & 0 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & & & -\frac{c\Delta t}{4\Delta x} & 1 & \frac{c\Delta t}{4\Delta x} \\ \frac{c\Delta t}{4\Delta x} & 0 & \dots & 0 & -\frac{c\Delta t}{4\Delta x} & 1 \end{bmatrix} \begin{bmatrix} u_0^{n+1} \\ u_1^{n+1} \\ \vdots \\ \vdots \\ u_{49}^{n+1} \\ u_{50}^{n+1} \end{bmatrix} \\ &= \begin{bmatrix} u_0^n - \frac{c\Delta t}{4\Delta x}(u_1^{n+1} - u_{50}^{n+1}) \\ u_1^n - \frac{c\Delta t}{4\Delta x}(u_2^{n+1} - u_0^{n+1}) \\ \vdots \\ \vdots \\ u_{49}^n - \frac{c\Delta t}{4\Delta x}(u_5 0^{n+1} - u_{48}^{n+1}) \\ u_{50}^n - \frac{c\Delta t}{4\Delta x}(u_0^{n+1} - u_{49}^{n+1}) \end{bmatrix}\end{aligned} \tag{17}$$

2.4 2nd Order Adams-Bashforth Method

The 2nd order Adams-Bashforth is an explicit multistep method that calculates the value of at the next time step using the values at the current and previous time step. The method for deriving the 2nd order Adams-Bashforth method was taken from reference [1]. The derivation begins, as in the trapezoidal method, by integrating the ODE given by equation 7 between time steps t^n and t^{n+1}:

$$\int_{t^n}^{t^{n+1}} \left.\frac{du}{dt}\right|_{x_j} dt = \int_{t^n}^{t^{n+1}} f(t,u)dt \tag{18}$$

$$u_j^{n+1} = u_j^n + \int_{t^n}^{t^{n+1}} f(t,u)dt \tag{19}$$

The remaining integral in equation 19 is evaluated by approximating the integrand with a first order interpolating polynomial between time steps t and $t-1$:

$$f(t,u) = f(t_i,u_i) + \frac{f(t_i,u_i) - f(t_{i-1},u_{i-1})}{t^n - t^{n-1}}(t-t^n) \tag{20}$$

Substituting equation 20 into the integral in equation 19 and performing the integration:

$$\begin{aligned}\int_{t^n}^{t^{n+1}} f(t,u)dt = \\ \int_{t^n}^{t^{n+1}} \left(f(t^n,u^n) + \frac{f(t^n,u^n) - f(t^{n-1},u^{n-1})}{t^n - t^{n-1}}(t-t^n) \right) dt\end{aligned} \tag{21}$$

$$= f(t^n,u^n)\int_{t^n}^{t^{n+1}} dt + \frac{f(t^n,u^n) - f(t^{n-1},t^{n-1})}{t^n - t^{n-1}} \int_{t^n}^{t^{n+1}} (t-t^n)dt \tag{22}$$

$$\begin{aligned}= f(t^n,u^n)(t^{n+1}-t^n) \\ + \frac{f(t^n,u^n) - f(t^{n-1},t^{n-1})}{t^n - t^{n-1}} \left(\frac{(t^{n+1}-t^n)^2}{2} - \frac{(t^n-t^n)^2}{2} \right)\end{aligned} \tag{23}$$

With equal time steps $\Delta t = t^{n+1} - t^n = t^n - t^{n-1}$, equation 23 becomes:

$$= f(t^n,u^n)\Delta t + \frac{f(t^n,u^n) - f(t^{n-1},t^{n-1})}{\Delta t} \left(\frac{(\Delta t)^2}{2} \right) \tag{24}$$

$$= \frac{\Delta t}{2}[3f(t^n, u^n) - f(t^{n-1}, u^{n-1})] \tag{25}$$

Finally, replacing the integral in equation 19 with the value arrived at in equation 25, the 2nd order Adams-Bashforth method is given by:

$$u^{n+1} = u^n + \frac{\Delta t}{2}[3f(t^n, u^n) - f(t^{n-1}, u^{n-1})] \tag{26}$$

The function $f(t, u)$ is given by equation 7 as $-c\frac{u_{j+1}-u_{j-1}}{2\Delta x}$. Substituting this into equation 26 gives:

$$u^{n+1} = u^n + \frac{\Delta t}{2}\left[3\left(-c\frac{u^n_{j+1} - u^n_{j-1}}{2\Delta x}\right) - \left(-c\frac{u^{n-1}_{j+1} - u^{n-1}_{j-1}}{2\Delta x}\right)\right] \tag{27}$$

The 2nd order Adams-Bashforth method is not self starting. If the values of u are given by the initial conditions at time step t^{n-1}, another method is required to obtain the values of u at time step t^n in order to compute u^{n+1}. In the program written for this assignment the 4th order Runge-Kutta method was used as a starter for the 2nd order Adams-Bashforth method.

2.5 4th Order Runge-Kutta Method

Runge-Kutta time stepping uses an equation of the form:

$$u^{n+1} = u^n + m \cdot \Delta t \tag{28}$$

where m is the slope between the points u^n and u^{n+1}. The value of m is obtained by using information about the slope at a combination of different points between t^n and t^{n+1}. The formula for the 4th order Runge-Kutta method used in this assignment was obtained by following the steps given in reference [2]. First, the wave equation is written as a "pseudo-ODE":

$$\frac{du}{dt} = R(u) \tag{29}$$

where $R(u) = -c\frac{du}{dx} = -cu_x$ and u_x is given by a central difference approximation.

Let

$$R^{()} = -cu_x^{()} \tag{30}$$

$$u^{(1)} = u^n + \frac{\Delta t}{2} R^n \tag{31}$$

$$u^{(2)} = u^n + \frac{\Delta t}{2} R^1 \tag{32}$$

$$u^{(3)} = u^n + \Delta t R^2 \tag{33}$$

so that

$$R^n = -cu_x^n \tag{34}$$

$$R^1 = -cu_x^1 = -c\left(u_x^n + \frac{\Delta t}{2} R_x^n\right) = -cu_x^n + c^2 \frac{\Delta t}{2} u_{xx}^n \tag{35}$$

$$R^2 = -cu_x^2 = -c\left(u_x^n + \frac{\Delta t}{2} R_x^1\right) = -cu_x^n + c^2 \frac{\Delta t}{2} u_{xx}^n - c^3 \frac{(\Delta t)^2}{4} u_{xxx}^n \tag{36}$$

$$\begin{aligned} R^3 &= -cu_x^{(3)} = -c(u_x^n + \Delta t R_x^2) \\ &= -cu_x^n + c^2 \Delta t u_{xx}^n - c^3 \frac{(\Delta t)^2}{2} u_{xxx}^n + c^4 \frac{(\Delta t)^3}{4} u_{xxxx}^n \end{aligned} \tag{37}$$

Where u_x, u_{xx}, u_{xxx}, and u_{xxxx} are 2nd order central difference approximations of the 1st, 2nd, 3rd, and 4th derivatives of u with respect to x, then the 4th order Runge-Kutta method is given by:

$$u_j^{n+1} = u_j^n + \frac{\Delta t}{6}(R^n + 2R^1 + 2R^2 + R^3) \tag{38}$$

The central difference approximations of u_{xx}, u_{xxx}, and u_{xxxx} can be derived using the Taylor series in a similar manner as was used to obtain equation 6. In part 2 of this assignment, the derivatives of u with respect to x were of 4th order accuracy. The formulas for these central differences were found in reference [1].

3 Results

For part 1, plots of the results of the time-marching schemes were required at t = 1s. The spatial grid was made up of $n = 50$ points. The size of the interval between points dx is equal to $\frac{1}{n-1} = \frac{1}{49} = 0.020408163265306\overline{12}$. For the explicit Euler and 2nd order Adams-Bashforth

methods a Courant number of 0.1 was used. Using a flow speed of $c = 1$, the equation for the Courant number was used to obtain the size of the time step dt.

$$\begin{aligned} dt &= \frac{Courant \cdot dx}{c} \\ &= \frac{0.1 \cdot 0.02040816326530612}{1} \\ &= 0.002040816326530612 \end{aligned} \tag{39}$$

The number of time steps needed to reach $t = 1s$ is determined by:

$$\frac{1}{dt} = \frac{1}{0.002040816326530612} = 490 \tag{40}$$

Running the program for 490 time steps of size dt using a 2^{nd} order central difference in space and the explicit Euler time-marching method results in the plot shown in figure 1. The plot using the 2^{nd} order Adams-Bashforth method is shown in figure 2.

For the remaining time-marching methods, a Courant number of 1 was used. In this case the value of dt is 10 times greater than in the explicit Euler and 2^{nd} order Adams-Bashforth methods. For $dt = 0.02040816326530612$ the number of time steps needed to reach $t = 1s$ is 49. The plots for the implicit Euler, trapezoidal, and 4^{th} order Runge-Kutta methods are shown in figures 3, 4, and 5 respectively.

In part 2, a 4^{th} order central difference was used to approximate the spatial derivatives with the Runge-Kutta time-marching method and a Courant number of 1. Plots using this scheme are shown in figure 6 for $t = 1s$ and figure 7 for $t = 10s$.

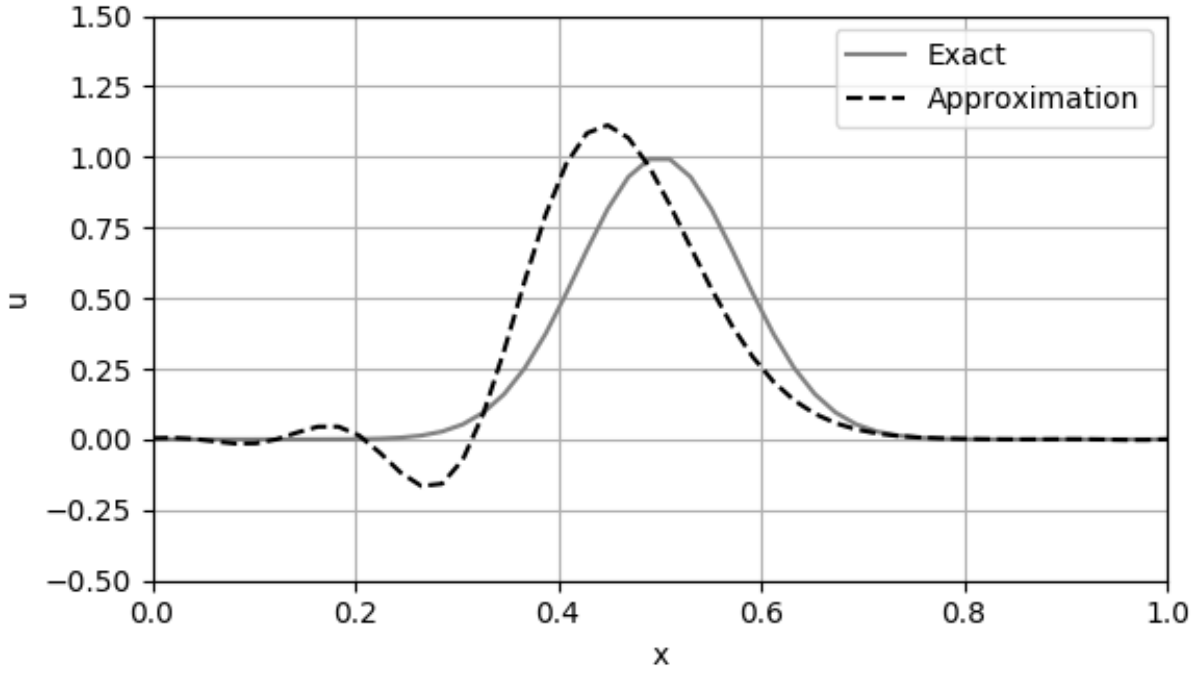

Figure 1: Plot for explicit Euler method

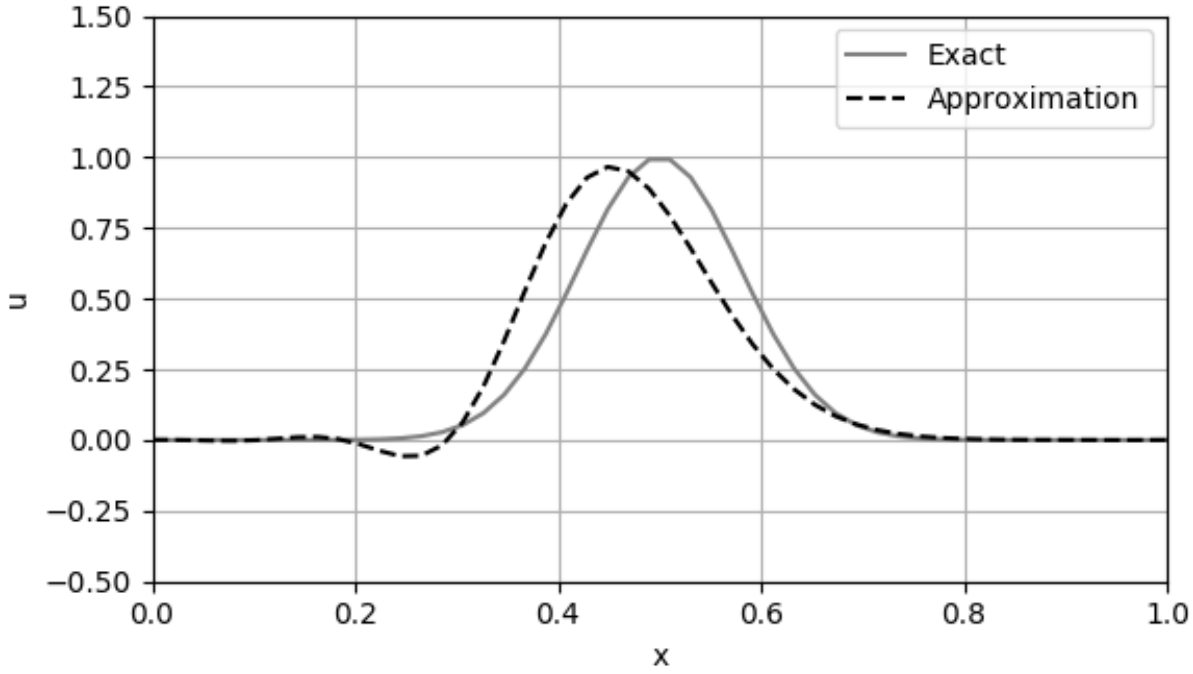

Figure 2: Plot for 2^{nd} order Adams-Bashforth method

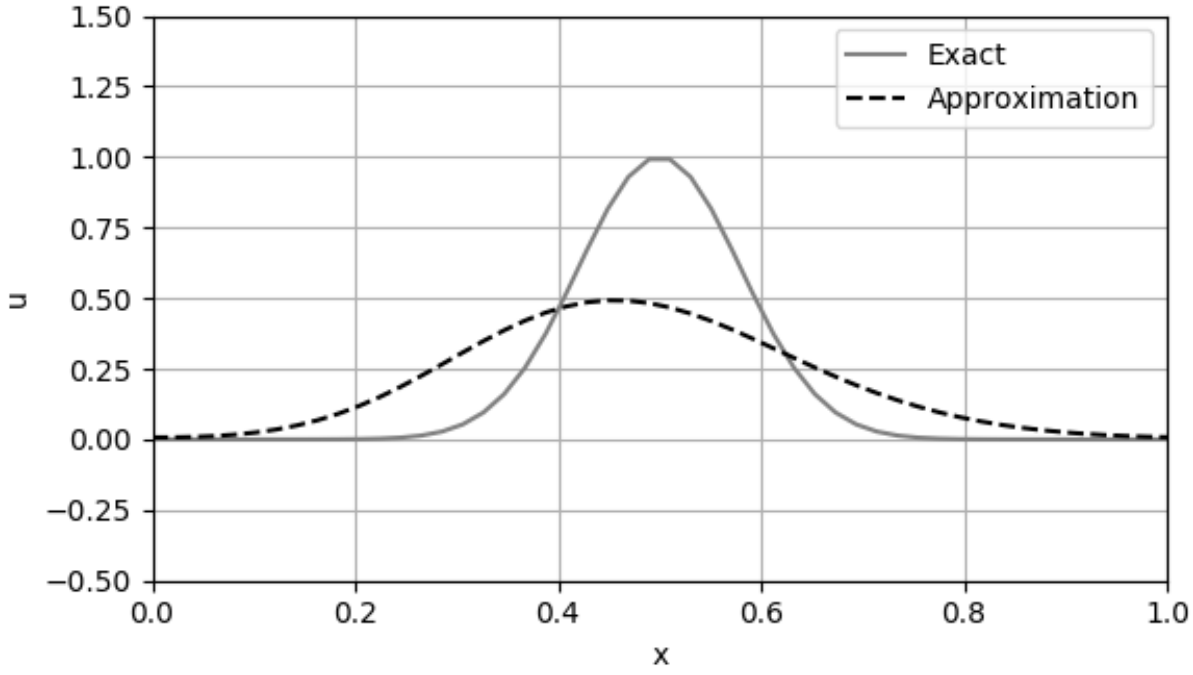

Figure 3: Plot for implicit Euler method

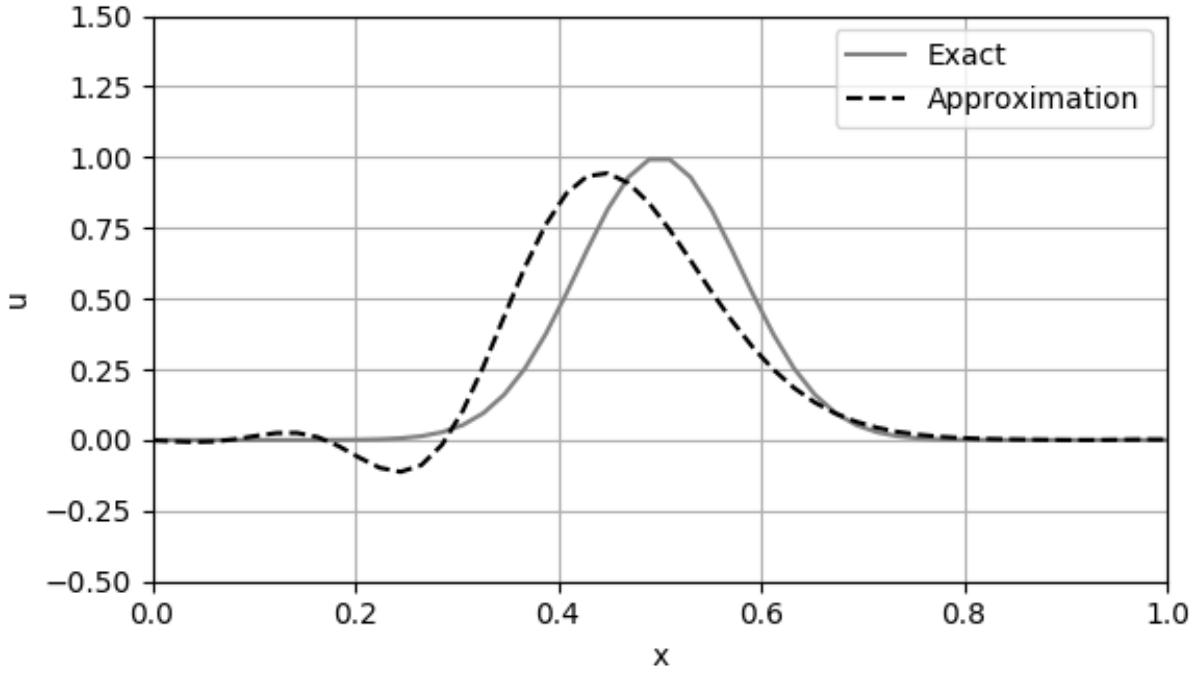

Figure 4: Plot for trapezoidal method

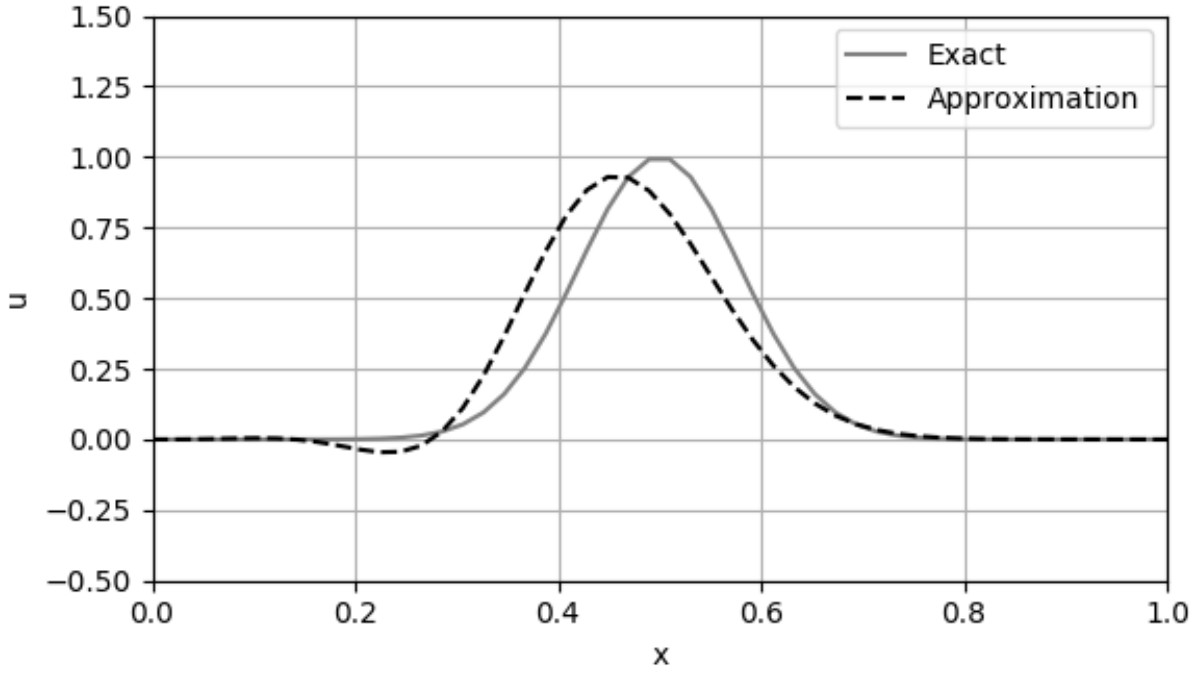

Figure 5: Plot for 4^{th} order Runge-Kutta method at t=1s with 2^{nd} order central difference in space

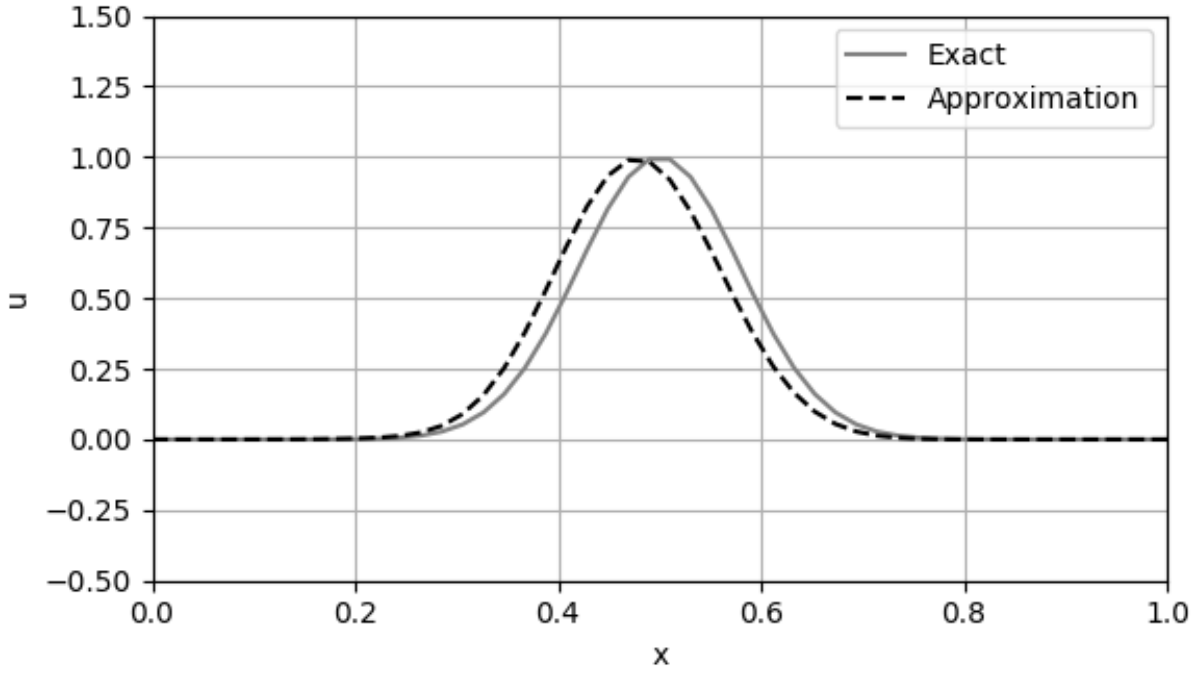

Figure 6: Plot for 4^{th} order Runge-Kutta method at t=1s with 4^{th} order central difference in space

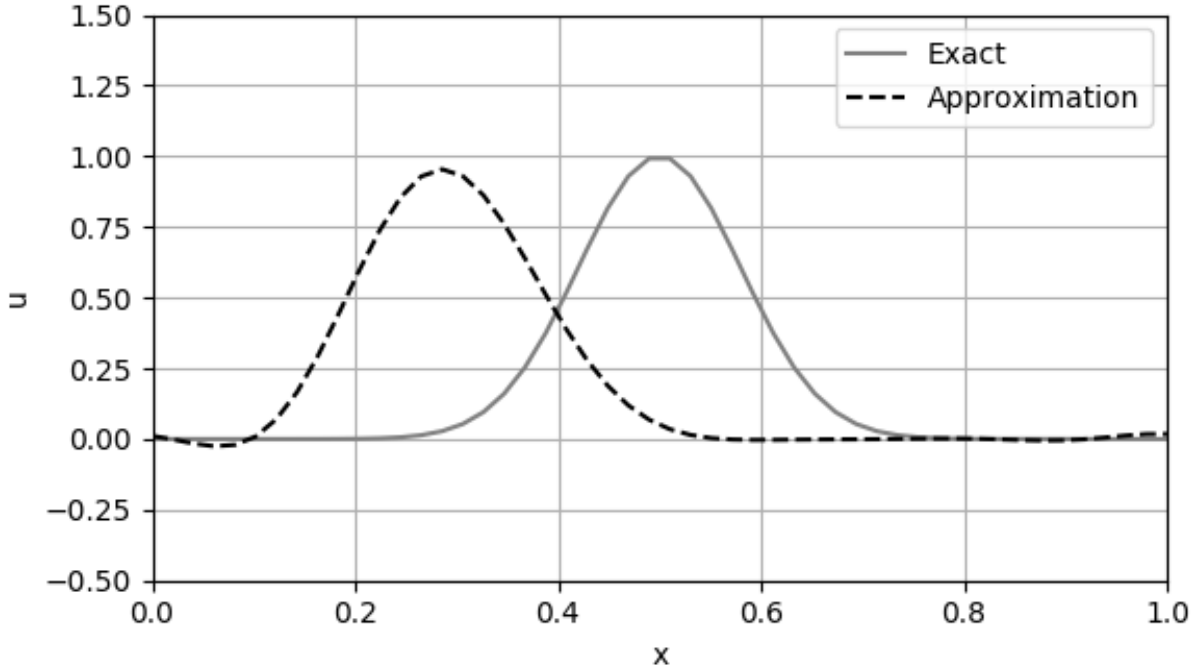

Figure 7: Plot for 4^{th} order Runge-Kutta method at t=10s with 4^{th} order central difference in space

4 Analysis

The results in figures 1 to 7 show that the numerical approximations differ from the exact solution by varying degree depending on the numerical scheme. All of the results show some amount of diffusion (dispersion and dissipation errors) and phase lag where the approximation appears to move at a different speed than the exact solution. Because the exact solution is known, the amount of error in the numerical approximation can be quantified by taking the Euclidean 2-norm of the error vector created by taking the square root of the sum of the squares of the difference between the exact and approximate values at each grid point.

The error in the approximation of the explicit Euler scheme is of order $O((\Delta t), (\Delta x)^2)$. The oscillations in the approximation that create new maximum and minimum values are a result of dispersion error. This type of error is dominant when the leading term of the truncation error contains an odd ordered derivative. The quantified error using the Euclidean 2-norm for this method at $t = 1s$ is approximately 1.109.

The 2^{nd} order Adams-Bashforth method also shows the effects of dispersion error however the amount of oscillation is less than in the explicit Euler method. The maximum amplitude of the Adams-Bashforth approximation is also slightly less than in the exact solution. The error in this scheme is of order $O((\Delta t)^2, (\Delta x)^2)$. The approximate error for this method at $t = 1s$ is 0.845.

The implicit Euler method shows a large difference in amplitude and a spreading out of the wave. This effect is known as dissipation error or implicit artificial viscosity and is caused by even ordered derivatives in the truncation error. In some cases, explicit artificial viscosity is added to a numerical approximation scheme to help resolve shock wave discontinuities, dampen dispersive oscillations, or stabilize the numerical calculations. The implicit Euler method has a truncation error with the same order as the explicit method however the amount of error at $t = 1s$ is greater, approximately 1.374.

The trapezoidal method produces a result that appears very similar to the 2^{nd} order Adams-Bashforth method but has a slightly lower minimum and maximum value and slightly more phase lag. The trapezoidal method is a 2^{nd} order scheme making the error of order $O((\Delta t)^2, (\Delta x)^2)$. The quantified error at $t = 1s$ is approximately 1.007.

The 4^{th} order Runge-Kutta method using a 2^{nd} order central difference for the spatial derivative gives a result similar to the 2^{nd} order Adams-Bashforth method but has a slightly lower maximum and slightly higher minimum value. The method uses 4^{th} order time marching and 2^{nd} order spatial difference so the order of the error is $O((\Delta t)^4, (\Delta x)^2)$. The calculated error at $t = 1$ is approximately 0.783.

Using a 4^{th} order central difference for $\frac{du}{dx}$ improves the Runge-Kutta approximation. The error becomes order $O((\Delta t)^4, (\Delta x)^4)$ and the maximum and minimum values stay very close to the maximum and minimum values of the exact solution. The total error at $t = 1s$ is about 0.485 which is mostly the result of the phase lag. When the approximation is run until $t = 10s$ the order of the error remains proportional to $(\Delta t)^4$ and $(\Delta x)^4$ which haven't changed, however, the effects of diffusion error start to appear more noticeably and the error due to the phase lag becomes very apparent as the total error in the approximation increases to approximately 3.312.

The periodic boundary conditions were applied by considering the value of u_{j-1} at the left boundary to be equal to the value at the right boundary, in this case u_{50} and using u_0 for u_{j+1} at the right boundary. Although this was confirmed to be the correct method by various sources, results were improved by choosing to use the values one step beyond the boundaries letting $u_{j-1} = u_{49}$ at the left boundary and $u_{j+1} = u_1$ at the right boundary. Using these modified boundary conditions for the 4^{th} order Runge-Kutta method with 4^{th} order central difference spatial approximation results in an error of only 0.2103 at $t = 10s$. The plot of this result is shown in figure 8. The reduction in error can be attributed to reduced amount of phase lag with this condition.

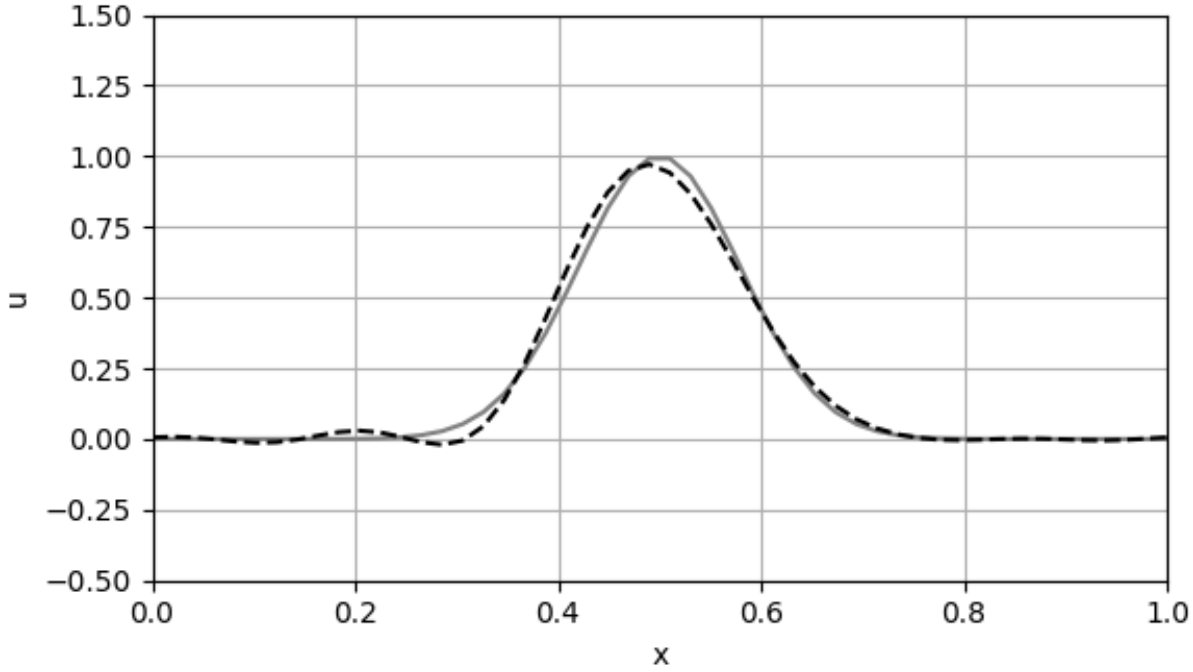

Figure 8: Plot for 4^{th} order Runge-Kutta method at t=10s with 4^{th} order central difference in space and modified boundary conditions

5 Conclusion

The preceding analysis shows the importance of selecting appropriate numerical methods to evaluate partial differential equations. The accuracy of the numerical solution depends on many factors. The amount of error inherent in discretizing the derivatives in space and time can vary significantly between different methods. The error in the approximation is also dependant on the equation being solved, some methods may work well for some equations and not as well for others. Rounding error is also a source of error coming from the precision with which the computer is able to represent numbers internally.

The truncation error for the time marching methods studied was found to be proportional to the size of the spacing between grid points in the spatial domain (Δx) and the size of the time step in the temporal domain (Δt). These factors are related to each other and the stability of the solution method through the Courant-Friedrichs-Lewy condition. Increasing the number of spatial grid points in the domain while keeping the Courant number constant reduces both Δx and Δt resulting in a better approximation.

For the particular problem posed in this assignment, the 4^{th} order Runge-Kutta method gave the best results in part 1. This was expected as this was the only 4^{th} order method applied. Using a 4^{th} order central difference for the spatial derivative in part 2 decreased the error by 38%. Generally, allowing the time marching to continue for a longer period

increased the amount of error as was seen by letting the Runge-Kutta method in part 2 run until $t = 10s$. This can be attributed to the growth of the phase lag and diffusive error over time.

References

[1] Amos Gilat and Vish Subramaniam. *Numerical Methods: An Introduction with Applications Using MATLAB*. 3rd. John Wiley & Sons, 2014. ISBN: 978-1-118-55493-7.

[2] Richard H. Pletcher, John C. Tannehill, and Dale A. Anderson. *Computational Fluid Mechanics and Heat Transfer*. 3rd. Taylor & Francis, 2013.

Appendices

Appendix A Running the program

Standalone versions of the application for Windows and Mac as well as the Python source code are available from https://www.devincrossman.com/aero/.

The Python script was written using Python 3.6.5 but may work with earlier versions. The script depends on a few different modules that must be installed prior to running the program. These modules can be installed system wide or within a virtual environment. Assuming Python 3.x and the package manager *pip* are already installed and located somewhere in your system path, the following procedure is used to run the program. First, create a directory for the Python virtual environment and copy the aero.py script to this directory. Then execute the following commands from a command line prompt from that directory.

Install virtualenv if not already installed.

```
$ pip install virtualenv
```

Create a new virtual environment in the current directory and activate it.

```
$ virtualenv .
$ source ./bin/activate # on Mac
$ .\Scripts\activate  # on Windows
```

This creates an isolated environment for the Python program to run in without affecting or being affected by other modules installed on the system. Next, install NumPy, Matplotlib, and NumExpr.

```
$ pip install numpy matplotlib numexpr
```

Run the program.

```
$ python aero.py
```

Use the command `deactivate` to exit the virtual environment.

Options to configure the numerical methods can be selected from drop downs and text inputs. A list of supported functions for the initial conditions can be found in the NumExpr documentation at the following url:

`http://numexpr.readthedocs.io/en/latest/user_guide.html#supported-functions`

Appendix B The Python code

```
"""This program solves the one-dimensional first order wave equation
with periodic boundary conditions on the domain 0 <= x <= 1"""
import tkinter as tk
from tkinter import messagebox
import numpy as np
import numexpr as ne
import matplotlib as mpl
#pylint: disable=wrong-import-position
mpl.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import matplotlib.pyplot as plt
import matplotlib.animation as animation
#pylint: enable=wrong-import-position

class App:
    """A tkinter app to demonstrate different methods of solving the
    first order wave equation (transport equation) with periodic
    boundary conditions.
    """
    def __init__(self):
        """Initialize the application."""

        # spatial domain
        self.xmin = 0
        self.xmax = 1

        # create the interface
        self.create_interface()

        # create the figure for the plots
        self.create_figure()

        # center the window on the screen
        self.center_window()

        # start the app
        self.root.mainloop()

    def create_interface(self):
        """Create the interface."""
        self.root = tk.Tk()
        self.root.title("AERO 455")

        time_stepping_options = [
            "Explicit Euler",
```

```
    "Implicit Euler",
    "Trapezoidal",
    "Adams-Bashforth 2",
    "Runge-Kutta 4"
]

time_steppping_label = tk.Label(self.root,
                                text="Time-marching method:")
time_steppping_label.grid(row=0, column=0, sticky=tk.W)
self.time_stepping = tk.StringVar(self.root)
self.time_stepping.set(time_stepping_options[0])

time_stepping_widget = tk.OptionMenu(
    self.root, self.time_stepping, *time_stepping_options)
time_stepping_widget.grid(row=0, column=1, sticky=tk.W)

spatial_derivative_options = [
    "2nd order",
    "4th order"
]
spatial_derivative_label = tk.Label(
    self.root, text="Spatial derivative accuracy:")
spatial_derivative_label.grid(row=1, column=0, sticky=tk.W)
self.spatial_derivative_order = tk.StringVar(self.root)
self.spatial_derivative_order.set(spatial_derivative_options[0])

self.spatial_derivative_order_widget = tk.OptionMenu(
    self.root, self.spatial_derivative_order,
    *spatial_derivative_options)
self.spatial_derivative_order_widget.grid(row=1, column=1,
                                          sticky=tk.W)

self.animate_btn = tk.Button(self.root, text="Animate",
                             fg="blue", command=self.animate)
self.animate_btn.grid(row=1, column=2, sticky=tk.W)

quit_btn = tk.Button(self.root, text="QUIT",
                     fg="red", command=self.root.quit)
quit_btn.grid(row=1, column=3, sticky=tk.W)

init_condition_label_widget = tk.Label(
    self.root,
    text="Initial Condition:")
init_condition_label_widget.grid(row=3, column=0, sticky=tk.W)

self.init_condition_input = tk.Entry(
    self.root, validate="focusout", width=20,
```

```
    vcmd=(
        self.root.register(self.validate_init_condition),
        '%P'))
self.init_condition_input.insert(0,
                                 "exp(-0.5*((x-0.5)/0.08)**2)")
self.init_condition_input.grid(row=3, column=1, sticky=tk.W)

wave_speed_label_widget = tk.Label(self.root,
                                   text="Wave speed:")
wave_speed_label_widget.grid(row=4, column=0, sticky=tk.W)

self.wave_speed_input = tk.Entry(
    self.root, validate="all", width=3,
    vcmd=(self.root.register(self.validate_wave_speed),
          '%V', '%P'))
self.wave_speed_input.insert(0, "1")
self.wave_speed_input.grid(row=4, column=1, sticky=tk.W)

cfl_label_widget = tk.Label(self.root, text="CFL number:")
cfl_label_widget.grid(row=5, column=0, sticky=tk.W)

self.cfl_input = tk.Entry(
    self.root, validate="all", width=3,
    vcmd=(self.root.register(self.validate_cfl), '%V', '%P'))
self.cfl_input.insert(0, "0.1")
self.cfl_input.grid(row=5, column=1, sticky=tk.W)

grid_size_label_widget = tk.Label(
    self.root,
    text="Number of spatial nodes:")
grid_size_label_widget.grid(row=6, column=0, sticky=tk.W)

self.grid_size_input = tk.Entry(
    self.root, validate="all", width=3,
    vcmd=(self.root.register(self.validate_grid_size),
          '%V', '%P'))
self.grid_size_input.insert(0, "50")
self.grid_size_input.grid(row=6, column=1, sticky=tk.W)

timesteps_label_widget = tk.Label(self.root,
                                  text="Number of time steps:")
timesteps_label_widget.grid(row=7, column=0, sticky=tk.W)

self.timesteps_input = tk.Entry(
    self.root, validate="all", width=4,
    vcmd=(self.root.register(self.validate_timesteps),
          '%V', '%P'))
```

```
        self.timesteps_input.insert(0, "490")
        self.timesteps_input.grid(row=7, column=1, sticky=tk.W)

        self.info_label = tk.StringVar(self.root)
        self.info_label.set("")

        info_label_widget = tk.Label(self.root,
                                     textvariable=self.info_label)
        info_label_widget.grid(row=8, columnspan=4)

        self.minmax_label = tk.StringVar(self.root)
        self.minmax_label.set("\n")

        minmax_label_widget = tk.Label(self.root,
                                       textvariable=self.minmax_label)
        minmax_label_widget.grid(row=9, columnspan=4)

        self.time_label = tk.StringVar(self.root)
        self.time_label.set("t = 0.00s")

        time_label_widget = tk.Label(self.root,
                                     textvariable=self.time_label)
        time_label_widget.grid(row=10, columnspan=4)

        self.error_label = tk.StringVar(self.root)
        self.error_label.set("error = 0")

        error_label_widget = tk.Label(self.root,
                                      textvariable=self.error_label)
        error_label_widget.grid(row=11, columnspan=4)

    def center_window(self):
        """Center the window on the screen."""
        self.root.withdraw()
        self.root.update_idletasks()
        screen_w = self.root.winfo_screenwidth()
        screen_h = self.root.winfo_screenheight()
        win_w = self.root.winfo_reqwidth()
        win_h = self.root.winfo_reqheight()
        x = screen_w/2 - win_w/2
        y = screen_h/2 - win_h/2
        self.root.geometry("%dx%d+%d+%d" % (win_w, win_h, x, y))
        self.root.deiconify()

    def create_figure(self):
        """Create the figure for plots."""
        canvas_w = 600
```

```
        canvas_h = 350
        #plt.style.use('dark_background')
        mpl.rcParams.update({'figure.autolayout': True})
        self.fig = plt.figure(figsize=(1, 1))
        fig_w = canvas_w/self.fig.get_dpi()
        fig_h = canvas_h/self.fig.get_dpi()
        self.fig.set_size_inches(fig_w, fig_h)
        self.ax = self.fig.add_subplot(111)
        self.ax.grid(True)
        self.ax.set_xlim([self.xmin, self.xmax])
        self.ax.set_ylim([-0.5, 1.5])
        self.ax.legend(loc='upper right')
        self.ax.set_xlabel('x')
        self.ax.set_ylabel('u')
        self.canvas = FigureCanvasTkAgg(self.fig, master=self.root)
        self.canvas.get_tk_widget().grid(columnspan=4)

    def validate_grid_size(self, operation, value):
        """Validate the grid size input
        must be an integer between 6 and 999.
        """
        if operation != 'focusout':
            if value == '':
                return True
        if len(value) > 3:
            return False
        if ' ' in value:
            return False
        try:
            int(value)
            if operation == 'focusout':
                if int(value) >= 6 and int(value) <= 999:
                    self.grid_size_input.configure(background='white')
                    return True
                raise ValueError
            return True
        except ValueError:
            messagebox.showerror(
                "Error",
                "The value for grid size must be an integer between "
                "6 and 999")
            self.grid_size_input.configure(background='#FFdddd')
            return False

    def validate_timesteps(self, operation, value):
        """Validate the grid size input
        must be an integer between 1 and 5000.
```

```
        """
        if operation != 'focusout':
            if value == '':
                return True
        if len(value) > 4:
            return False
        if ' ' in value:
            return False
        try:
            int(value)
            if operation == 'focusout':
                if int(value) >= 1 and int(value) <= 5000:
                    self.timesteps_input.configure(background='white')
                    return True
                raise ValueError
            return True
        except ValueError:
            messagebox.showerror(
                "Error",
                "The value for time steps must be an integer between "
                "1 and 5000")
            self.timesteps_input.configure(background='#FFdddd')
            return False

    def validate_wave_speed(self, operation, value):
        """Validate the wave speed input
        must be a floating point number between 0 and 10.
        """
        if operation != 'focusout':
            if value == '' or value == '-' or value == '.':
                return True
        if ' ' in value:
            return False
        try:
            float(value)
            if operation == 'focusout':
                if float(value) >= -10 and float(value) <= 10:
                    self.wave_speed_input.configure(background='white')
                    return True
                raise ValueError
            return True
        except ValueError:
            messagebox.showerror(
                "Error",
                "The value for wave speed must be a floating point "
                "number between -10 and 10")
            self.wave_speed_input.configure(background='#FFdddd')
```

```
        return False

    def validate_cfl(self, operation, value):
        """Validate the cfl input
        must be a floating point number between 0 and 10.
        """
        if operation != 'focusout':
            if value == '' or value == '.':
                return True
        if ' ' in value:
            return False
        try:
            float(value)
            if operation == 'focusout':
                if float(value) > 0 and float(value) <= 10:
                    self.cfl_input.configure(background='white')
                    return True
                raise ValueError
            return True
        except ValueError:
            messagebox.showerror(
                "Error",
                "The value for CFL must be a floating point number "
                "between 0 and 10")
            self.cfl_input.configure(background='#FFdddd')
            return False

    def validate_init_condition(self, value):
        """Validate the initial condition
        must be a function of x only.
        """
        try:
            x = np.linspace(self.xmin, self.xmax)
            ne.evaluate(value, {'x':x, 'pi':np.pi})
            self.init_condition_input.configure(background='white')
            return True
        except Exception:
            messagebox.showerror(
                "Error",
                "The value for the initial condition must be an "
                "evaluatable function of x")
            self.init_condition_input.configure(background='#FFdddd')
            return False

    def create_grid(self):
        """Create the solution grid based on user inputs and solve
        the exact equation.
```

```
        """
        self.n = int(self.grid_size_input.get())
        self.c = float(self.wave_speed_input.get())
        self.CFL = float(self.cfl_input.get())
        self.X, self.dx = np.linspace(self.xmin, self.xmax,
                                      self.n, retstep=True)
        # determine dt based on CFL
        self.dt = self.CFL * self.dx / np.abs(self.c)
        self.info_label.set(f"c = {self.c}, dx = {self.dx}, "
                            f"dt = {self.dt}, CFL = {self.CFL}")
        # solve U_exact for tmax time steps
        # using mod to make the function periodic
        self.U_exact = []
        for t in range(self.tmax + 1):
            self.U_exact.append(self.initial_u(
                np.mod(self.X - self.c * t * self.dt,
                       self.xmax - self.xmin)))

    def clear_plot(self):
        """Clear the previous solutions from the figure."""
        if hasattr(self, 'exact_line'):
            self.exact_line.remove()
        if hasattr(self, 'approximation_line'):
            self.approximation_line.remove()

    def current_timestep(self):
        """Return the current timestep for the animation."""
        self.timestep = 0
        while self.timestep < self.tmax:
            self.timestep += 1
            yield self.timestep

    def inputs_are_valid(self):
        """Check if user inputs are valid."""
        return (self.validate_init_condition(
            self.init_condition_input.get())
                and self.validate_cfl('focusout', self.cfl_input.get())
                and self.validate_wave_speed('focusout',
                                             self.wave_speed_input.get())
                and self.validate_timesteps('focusout',
                                            self.timesteps_input.get())
                and self.validate_grid_size('focusout',
                                            self.grid_size_input.get()))

    def animate(self):
        """Start the animation."""
        # ensure user input validation is performed
```

```
        if self.inputs_are_valid():
            self.solve_approximation()
            self.clear_plot()
            self.exact_line, = self.ax.plot(self.X, self.U_exact[0],
                                            color="grey",
                                            label='Exact')
            self.approximation_line, = self.ax.plot(
                self.X, self.U_exact[0], color="black",
                label='Approximation', linestyle='dashed')
            # stop previous animation by setting the timestep > tmax
            self.timestep = self.tmax + 1
            self.anim = animation.FuncAnimation(
                self.fig, self.do_animation,
                frames=self.current_timestep,
                repeat=False, interval=self.dt*1000)
            self.fig.canvas.draw()

    def do_animation(self, i):
        """Update the animation frame."""
        self.time_label.set(f"t = {i} * {self.dt}s = {i*self.dt:.2f}s")
        self.exact_line.set_ydata(self.U_exact[i])
        self.approximation_line.set_ydata(self.U[i])
        error = np.sqrt(np.sum(np.power(self.U_exact[i]-self.U[i], 2)))
        self.error_label.set(f'error = {error}')
        minmax = ('Exact min: ' + str(self.U_exact[i].min())
                  + ' Exact max: ' + str(self.U_exact[i].max())
                  + '\nNumerical min: ' + str(self.U[i].min())
                  + ' Numerical max: ' + str(self.U[i].max()))
        self.minmax_label.set(minmax)

    def solve_approximation(self):
        """Solve the differential equation
        using thecurrent settings."""
        selected_method = self.time_stepping.get()
        if selected_method == 'Explicit Euler':
            solution_method = self.euler_explicit
        elif selected_method == 'Implicit Euler':
            solution_method = self.euler_implicit
        elif selected_method == 'Trapezoidal':
            solution_method = self.trapezoidal
        elif selected_method == 'Adams-Bashforth 2':
            solution_method = self.ab2
        elif selected_method == 'Runge-Kutta 4':
            solution_method = self.rk4
        self.U = [] # reset the value of U
        self.tmax = int(self.timesteps_input.get())
        # create grid and solve exact solution
```

```
        self.create_grid()
        # solve approximation using solution method
        for t in range(self.tmax + 1):
            self.U.append(solution_method(t))

    def initial_u(self, _x):
        """Return the initial values for U."""
        return ne.evaluate(self.init_condition_input.get(),
                           {'x':_x, 'pi':np.pi})

    def euler_explicit(self, t):
        """Return the solution for the t-th time step
        using the euler explicit method.
        """
        U, c, dt = self.U, self.c, self.dt
        if t == 0: # initial condition
            return self.initial_u(self.X)
        return U[t-1] - c*dt*self.central_ux(t-1)

    def euler_implicit(self, t):
        """Return the solution for the t-th time step
        using the euler implicit method.
        """
        U, c, dt, dx, n = self.U, self.c, self.dt, self.dx, self.n
        if t == 0: # initial condition
            return self.initial_u(self.X)
        a = np.zeros((n, n))
        # set the corners of the matrix
        a[0][n-1] = -c*dt/(2*dx)
        a[n-1][0] = c*dt/(2*dx)
        for i in range(n):
            for j in range(n):
                if i == j:
                    a[i][j] = 1
                elif i == j-1:
                    a[i][j] = c*dt/(2*dx)
                elif i == j+1:
                    a[i][j] = -c*dt/(2*dx)
        return np.linalg.solve(a, U[t-1])

    def ab2(self, t):
        """Return the solution for the t-th time step using the
        Adams-Bashforth 2nd order method.
        """
        U, c, dt = self.U, self.c, self.dt
        if t == 0: # initial condition
            return self.initial_u(self.X)
```

```
    if t == 1: # first step with RK4 to start
        return self.rk4(t)
    return (U[t-1] - c*dt*((3/2)*self.central_ux(t-1)
                           - (1/2)*self.central_ux(t-2)))

def trapezoidal(self, t):
    """Return the solution for the t-th time step using the
    Adams-Moulton 2nd order (trapezoidal) method.
    """
    U, c, dt, dx, n = self.U, self.c, self.dt, self.dx, self.n
    if t == 0: # initial condition
        return self.initial_u(self.X)
    a = np.zeros((n, n))
    b = np.zeros(n)
    # set the corners of the matrix
    a[0][n-1] = -c*dt/(4*dx)
    a[n-1][0] = c*dt/(4*dx)
    for i in range(n):
        for j in range(n):
            # set diagonals
            if i == j:
                a[i][j] = 1
            elif i == j-1:
                a[i][j] = c*dt/(4*dx)
            elif i == j+1:
                a[i][j] = -c*dt/(4*dx)
    # set solution vector
    b = U[t-1]-c*dt/2*self.central_ux(t-1)
    return np.linalg.solve(a, b)

def rk4(self, t):
    """Return the solution for the t-th time step using the
    classical 4th order Runge-Kutta method.
    """
    U, c, dt = self.U, self.c, self.dt
    if t == 0: # initial condition
        return self.initial_u(self.X)
    ux = self.central_ux(t-1)
    uxx = self.central_uxx(t-1)
    uxxx = self.central_uxxx(t-1)
    uxxxx = self.central_uxxxx(t-1)
    Rn = -c*ux
    R1 = -c*ux + (c**2)*dt/2*uxx
    R2 = (-c*ux + (c**2)*dt/2*uxx
          - (c**3)*(dt**2)/4*uxxx)
    R3 = (-c*ux + (c**2)*dt*uxx
          - (c**3)*(dt**2)/2*uxxx
```

```
            + (c**4)*(dt**3)/4*uxxxx)
        return U[t-1] + dt/6*(Rn+2*R1+2*R2+R3)

    def central_ux(self, t):
        """Return the central difference approximation of the first
        derivative of u with respect to x at time step t.
        """
        if self.spatial_derivative_order.get() == '2nd order':
            return self.central_ux_2(t)
        return self.central_ux_4(t)

    def central_uxx(self, t):
        """Return the central difference approximation of the second
        derivative of u with respect to x at time step t.
        """
        if self.spatial_derivative_order.get() == '2nd order':
            return self.central_uxx_2(t)
        return self.central_uxx_4(t)

    def central_uxxx(self, t):
        """Return the central difference approximation of the third
        derivative of u with respect to x at time step t.
        """
        if self.spatial_derivative_order.get() == '2nd order':
            return self.central_uxxx_2(t)
        return self.central_uxxx_4(t)

    def central_uxxxx(self, t):
        """Return the central difference approximation of the fourth
        derivative of u with respect to x at time step t.
        """
        if self.spatial_derivative_order.get() == '2nd order':
            return self.central_uxxxx_2(t)
        return self.central_uxxxx_4(t)

    def central_ux_2(self, t):
        """Return the 2nd order central difference approximation
        of the first derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((U[t][j+1]-U[t][n-1])/(2*dx))
            elif j == n-1: # right boundary
                val.append((U[t][0]-U[t][j-1])/(2*dx))
            else:
```

```
            val.append((U[t][j+1]-U[t][j-1])/(2*dx))
        return np.array(val)

    def central_ux_4(self, t):
        """Return the 4th order central difference approximation
        of the first derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((U[t][n-2]-8*U[t][n-1]+8*U[t][j+1]
                            -U[t][j+2])/(12*dx))
            elif j == 1:
                val.append((U[t][n-1]-8*U[t][j-1]+8*U[t][j+1]
                            -U[t][j+2])/(12*dx))
            elif j == n-2:
                val.append((U[t][j-2]-8*U[t][j-1]+8*U[t][j+1]
                            -U[t][0])/(12*dx))
            elif j == n-1: # right boundary
                val.append((U[t][j-2]-8*U[t][j-1]+8*U[t][0]
                            -U[t][1])/(12*dx))
            else:
                val.append((U[t][j-2]-8*U[t][j-1]+8*U[t][j+1]
                            -U[t][j+2])/(12*dx))
        return np.array(val)

    def central_uxx_2(self, t):
        """Return the 2nd order central difference approximation
        of the second derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((U[t][j+1]-2*U[t][j]+U[t][n-1])/(dx**2))
            elif j == n-1: # right boundary
                val.append((U[t][0]-2*U[t][j]+U[t][j-1])/(dx**2))
            else:
                val.append((U[t][j+1]-2*U[t][j]+U[t][j-1])/(dx**2))
        return np.array(val)

    def central_uxx_4(self, t):
        """Return the 4th order central difference approximation
        of the second derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
```

```
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((-U[t][n-2]+16*U[t][n-1]-30*U[t][j]
                            +16*U[t][j+1]-U[t][j+2])/(12*dx**2))
            elif j == 1:
                val.append((-U[t][n-1]+16*U[t][j-1]-30*U[t][j]
                            +16*U[t][j+1]-U[t][j+2])/(12*dx**2))
            elif j == n-2:
                val.append((-U[t][j-2]+16*U[t][j-1]-30*U[t][j]
                            +16*U[t][j+1]-U[t][0])/(12*dx**2))
            elif j == n-1: # right boundary
                val.append((-U[t][j-2]+16*U[t][j-1]-30*U[t][j]
                            +16*U[t][0]-U[t][1])/(12*dx**2))
            else:
                val.append((-U[t][j-2]+16*U[t][j-1]-30*U[t][j]
                            +16*U[t][j+1]-U[t][j+2])/(12*dx**2))
        return np.array(val)

    def central_uxxx_2(self, t):
        """Return the 2nd order central difference approximation
        of the third derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((-U[t][n-2]+2*U[t][n-1]-2*U[t][j+1]
                            +U[t][j+2])/(2*dx**3))
            elif j == 1:
                val.append((-U[t][n-1]+2*U[t][j-1]-2*U[t][j+1]
                            +U[t][j+2])/(2*dx**3))
            elif j == n-2:
                val.append((-U[t][j-2]+2*U[t][j-1]-2*U[t][j+1]
                            +U[t][0])/(2*dx**3))
            elif j == n-1: # right boundary
                val.append((-U[t][j-2]+2*U[t][j-1]-2*U[t][0]
                            +U[t][1])/(2*dx**3))
            else:
                val.append((-U[t][j-2]+2*U[t][j-1]-2*U[t][j+1]
                            +U[t][j+2])/(2*dx**3))
        return np.array(val)

    def central_uxxx_4(self, t):
        """Return the 4th order central difference approximation
        of the third derivative of U with respect to x at time step t.
        """
```

```
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((U[t][n-3]-8*U[t][n-2]+13*U[t][n-1]
                            -13*U[t][j+1]+8*U[t][j+2]
                            -U[t][j+3])/(8*dx**3))
            elif j == 1:
                val.append((U[t][n-2]-8*U[t][n-1]+13*U[t][j-1]
                            -13*U[t][j+1]+8*U[t][j+2]
                            -U[t][j+3])/(8*dx**3))
            elif j == 2:
                val.append((U[t][n-1]-8*U[t][j-2]+13*U[t][j-1]
                            -13*U[t][j+1]+8*U[t][j+2]
                            -U[t][j+3])/(8*dx**3))
            elif j == n-3:
                val.append((U[t][j-3]-8*U[t][j-2]+13*U[t][j-1]
                            -13*U[t][j+1]+8*U[t][j+2]
                            -U[t][0])/(8*dx**3))
            elif j == n-2:
                val.append((U[t][j-3]-8*U[t][j-2]+13*U[t][j-1]
                            -13*U[t][j+1]+8*U[t][0]
                            -U[t][1])/(8*dx**3))
            elif j == n-1: # right boundary
                val.append((U[t][j-3]-8*U[t][j-2]+13*U[t][j-1]
                            -13*U[t][0]+8*U[t][1]-U[t][2])/(8*dx**3))
            else:
                val.append((U[t][j-3]-8*U[t][j-2]+13*U[t][j-1]
                            -13*U[t][j+1]+8*U[t][j+2]
                            -U[t][j+3])/(8*dx**3))
        return np.array(val)

    def central_uxxxx_2(self, t):
        """Return the 2nd order central difference approximation
        of the fourth derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((U[t][n-2]-4*U[t][n-1]+6*U[t][j]-4*U[t][j+1]
                            +U[t][j+2])/(dx**4))
            elif j == 1:
                val.append((U[t][n-1]-4*U[t][j-1]+6*U[t][j]-4*U[t][j+1]
                            +U[t][j+2])/(dx**4))
            elif j == n-2:
                val.append((U[t][j-2]-4*U[t][j-1]+6*U[t][j]-4*U[t][j+1]
```

```
                        +U[t][0])/(dx**4))
            elif j == n-1: # right boundary
                val.append((U[t][j-2]-4*U[t][j-1]+6*U[t][j]-4*U[t][0]
                        +U[t][1])/(dx**4))
            else:
                val.append((U[t][j-2]-4*U[t][j-1]+6*U[t][j]-4*U[t][j+1]
                        +U[t][j+2])/(dx**4))
        return np.array(val)

    def central_uxxxx_4(self, t):
        """Return the 4th order central difference approximation
        of the fourth derivative of U with respect to x at time step t.
        """
        U, X, dx, n = self.U, self.X, self.dx, self.n
        val = [] # values for the approximation at the t-th time step
        for j in range(len(X)):
            if j == 0: # left boundary
                val.append((-U[t][n-3]+12*U[t][n-2]-39*U[t][n-1]
                        +56*U[t][j]-39*U[t][j+1]+12*U[t][j+2]
                        -U[t][j+3])/(6*dx**4))
            elif j == 1:
                val.append((-U[t][n-2]+12*U[t][n-1]-39*U[t][j-1]
                        +56*U[t][j]-39*U[t][j+1]+12*U[t][j+2]
                        -U[t][j+3])/(6*dx**4))
            elif j == 2:
                val.append((-U[t][n-1]+12*U[t][j-2]-39*U[t][j-1]
                        +56*U[t][j]-39*U[t][j+1]+12*U[t][j+2]
                        -U[t][j+3])/(6*dx**4))
            elif j == n-3:
                val.append((-U[t][j-3]+12*U[t][j-2]-39*U[t][j-1]
                        +56*U[t][j]-39*U[t][j+1]+12*U[t][j+2]
                        -U[t][0])/(6*dx**4))
            elif j == n-2:
                val.append((-U[t][j-3]+12*U[t][j-2]-39*U[t][j-1]
                        +56*U[t][j]-39*U[t][j+1]+12*U[t][0]
                        -U[t][1])/(6*dx**4))
            elif j == n-1: # right boundary
                val.append((-U[t][j-3]+12*U[t][j-2]-39*U[t][j-1]
                        +56*U[t][j]-39*U[t][0]+12*U[t][1]
                        -U[t][2])/(6*dx**4))
            else:
                val.append((-U[t][j-3]+12*U[t][j-2]-39*U[t][j-1]
                        +56*U[t][j]-39*U[t][j+1]+12*U[t][j+2]
                        -U[t][j+3])/(6*dx**4))
        return np.array(val)
app = App()
```