A Simple Numerical Solition Solution of the Korteweg-de-Vries Equation

Brianna Isola, Dominic Payne

The Korteweg-de-Vries (KdV) equation

The Korteweg-de-Vries (KdV) equation is a mathematical representation of shallow water waves which can be expressed in the partial differential form of,

$$\partial_t u + \partial_{xxx} u + 6u\partial_x u = 0$$

Here we define u is the height if the water on top of the seafloor, where the second and third terms of the equation describe the dispersive and advection behaviors of the wave, respectively. To solve for this equation numerically, one can convert this equation to an ordinary differential equation (ODE) through a spatial-differentiation technique. In ODE form, the KdV equation can be solved using the 4th order Runge-Kutta time stepping method to integrate the function over time to find values of $u(x,t)$. To compare against numerical results, there exists an analytical solution of $u(x,t)$ defined as the analytical traveling wave solution to the KdV equation,

$$u(x,t) = \frac{c}{2}\operatorname{sech}^2\left(\frac{1}{2}\sqrt{c}\,(x - ct - a)\right)$$

where c is the wave speed proportional to the waves amplitude and a is some arbitrary constant [1]. The position of the wave peak at time t is $x = ct$, where the inner term of the hyperbolic acosine $x - ct$ describes the movement of the wave [2].

In this report, we detail a simulation code to numerically integrate the 1D Korteweg-de-Vries equation through the method described above. We will compare the accuracy of our numerical through the analytical solution, and analyze our results through various plots and comparisons. Additionally, we aim to optimize our code using OpenMP parallelization.

Spatial Discretization

In order to solve the KdV equation, we must convert it from a partial differential equation to an ordinary differential equation. We can effectively turn this into an ODE if we take discrete derivatives between adjacent values of u at a given time. To do this, we must first define a spatial grid of size L. We will define a uniform grid $x_i = i\Delta x$ with integer index i and discretized field $u_i \equiv u(i\Delta x)$. Where derivatives are approximated using finite differences, we get the general equation,

$$(\partial_x u)_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x}$$

Where we can derive our second and third derivatives as,

$$(\partial_{xx} u)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$$

$$(\partial_{xxx}u)_i \approx \frac{(\partial_{xx}u)_{i+1} - (\partial_{xx}u)_{i-1}}{2\Delta x}$$

Using the finite difference approximations above, we can compute our equation in the form of a time-derivative ODE. It is important to note that the boundaries of our field u_0 and u_L are periodic and behave similarly to a cosine function such that the values at each boundary are the same,

$$u(x + Ln, t) = u(x, t)$$

This will be particularly useful for handling potential out-of-bounds errors from taking the derivative beyond the boundaries of our field. Our method for handling such cases will be documented in the next section with our code implementation and analysis.

The 4th Order Runge-Kutta Method

Once we are able to convert our equation into ODE form, we are able to integrate it through the standard RK4 time-stepping scheme, otherwise known as the Runge-Kutta method. The 4th Order Runge-Kutta (RK4) method is an iterative method used to approximate the solution of an ordinary differential equation (ODE). By taking the ODE in the form of,

$$\frac{dy(x)}{dx} = f(x, y), \qquad y(t_0) = y_0$$

we can use the rate $\frac{dy}{dx}$ at which y changes to approximate y at some point x. In many real-life applications, we can measure the rate of change of y over some period of time t where our equations take the form,

$$\frac{dy(t)}{dt} = f(t, y), \qquad y(t_0) = y_0$$

Where we define an interval with intial 'start' time t_0 and a final 'stop' time t_N. We can use our exact solution at $t = 0$ to determine the initial conditions for our code. A four-step slope approximation method is used determining the slope $\mathbf{k}$ at various points of time,

$$\begin{aligned}
\text{Step 1:} \quad & \mathbf{k}_1 = \Delta t \frac{dy}{dt}(y_n, t_n) \\
\text{Step 2:} \quad & \mathbf{k}_2 = \Delta t \frac{dy}{dt}\left(y_n + \frac{1}{2}\mathbf{k}_1, t_n + \frac{1}{2}\Delta t\right) \\
\text{Step 3:} \quad & \mathbf{k}_3 = \Delta t \frac{dy}{dt}\left(y_n + \frac{1}{2}\mathbf{k}_2, t_n + \frac{1}{2}\Delta t\right) \\
\text{Step 4:} \quad & \mathbf{k}_4 = \Delta t \frac{dy}{dt}(y_n + \mathbf{k}_3, t_n + \Delta t)
\end{aligned}$$

$$n = 0, 1, 2, 3...N$$

Where Δt is the step size on the interval of $[t_0, t_N]$ and y_n and t_n are the nth iterations at each step. We can define define each of the slope approximations as follows [3],

- $\mathbf{k}_1$: the slope at the beginning of the interval (Euler's); $y = y_n$
- $\mathbf{k}_2$: the slope at the midpoint of the interval; $y = y_n + \Delta t \frac{1}{2} k_1$
- $\mathbf{k}_3$: the slope at the midpoint of the interval; $y = y_n + \Delta t \frac{1}{2} k_2$
- $\mathbf{k}_4$: the slope at the end of the interval; $y = y_n + \Delta t k_3$

As a fifth and final step, we can use the weighted average slop approximation to determine a final estimate of y at $t = t_N$,

$$\text{Step 5: } y_N = y_n + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$$

Example: A simple ODE

Before solving for the 1-D KdV equation, we decided to create our RK4 code around a simpler ODE with a known set of solutions. This allowed us to thoroughly test the RK4 function with ease before applying it to the KdV equation.

For our test ODE, we used the example detailed in Interactive Mathematics which included complete numerical solutions at all of the steps. Here, they define an ODE in the form,

$$\frac{dy(t)}{dt} = \frac{5t^2 - y}{e^{t+y}}, \qquad y(0) = 1$$

With parameters $t = [0, 1]$, $y_0 = 1$, and $\Delta t = 0.1$. This code is contained within the `rk4_dydt` sub-directory, and we were happy to get results that aligned with theirs.

Implementation

Our numerical solution code is organized as follows,

File	Desc.
`kdv.cxx`	main compilation code where all variables/parameters are defined
`kdv_func.cxx`	general functions ued throughout, such as the function to write our data to a file
`kdv_param.h`	general header file
`rhs.cxx`	right-hand-side solver for the RK4 k-step values and the function for the derivative approximation
`rk4.cxx`	4th order runge-kutta method code
`KdV_Plotter.ipynb`	python notebook to plot results
`data/`	empty folder where output data files will go
`data/plots/`	empty folder where plot images will save

To run the code through the command line you must create a `build` directory. Within this directory, the code can be compiled and run as,

```
~/rk4_brianna/build$ cmake ..
~/rk4_brianna/build$ make
~/rk4_brianna/build$ ./kdv
```

Output

This code produces three output files, a "DAT," "SOL," and "VARS" text file that are written to the `data/` folder. The DAT and SOL files contain the our numerical and analytical solution data for all of our $u(x,t)$ values over time, where the first column are the x value and the preceding columns are all our $u(x,t)$ values at different times. If N is the dimensions of our x grid-space and M is the dimension of our time values, we find the data files in the form of,

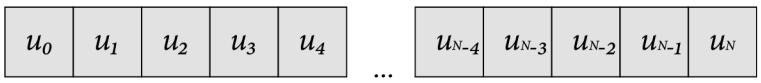
$$\begin{array}{cccccc} x_0 & u(x_0,t_0) & u(x_0,t_1) & u(x_0,t_2) & ... & u(x_0,t_M) \\ x_1 & u(x_1,t_0) & u(x_1,t_1) & u(x_1,t_2) & ... & u(x_1,t_M) \\ x_2 & u(x_2,t_0) & u(x_2,t_1) & u(x_2,t_2) & ... & u(x_2,t_M) \\ ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... & ... \\ x_N & u(x_N,t_0) & u(x_N,t_1) & u(x_N,t_2) & ... & u(x_N,t_M) \end{array}$$

The VARS file contains all of the parameters. Both data files follow a string of numbers that describe their dimensions and a and c constant parameters; however, the VARS file is updated with the parameters of the latest run of the code and does not have any identifiable parameters in its filename.

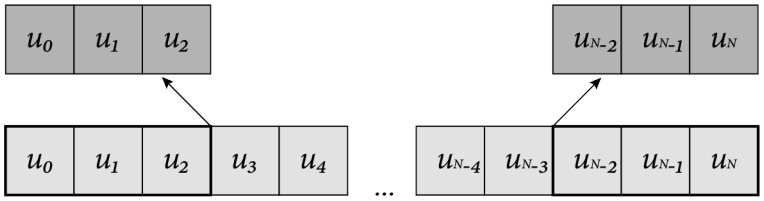
To plot the data, the VARS file is read by `KdV_Plotter` notebook which uses the parameters listed in the file to find the DAT and SOL files with the same parameters and extract their information. `KdV_Plotter` is already set up to plot the latest dataset created on the command line. Additionally, there exists sample data in `/data/plots/plotdata/` that is also read and plotted within `KdV_Plotter`. Plots generated are automatically saved in `data/plots/`.

Boundary Conditions

The finite difference method used solving for the time-derivative ODE form of the KdV equation finds itself reaching out-of-bounds at the first and last index of our gridspace array. Since we know the periodic behavior of our function, we were able to resolve these boundary conditions by implementing a 'padded array' as seen in `rhs.cxx`. In this function, we duplicate our original discretized field and 'pad' it by replicating the last 3 at each end of the array and amending it to the opposite side. As a visual example, if we have our field:

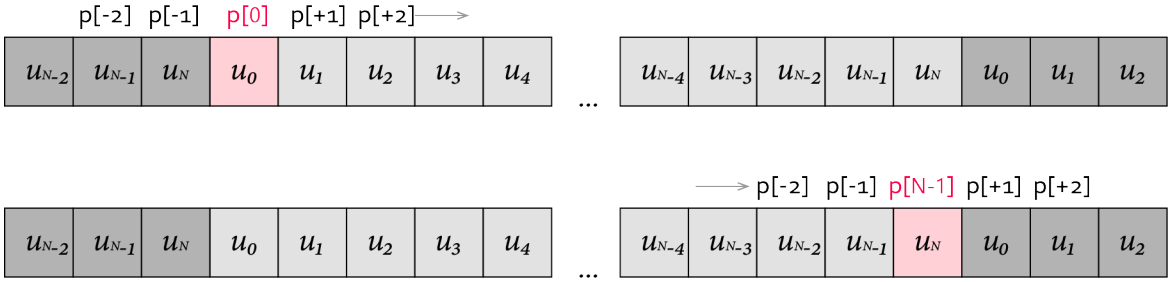


We then copy each end of the array by a few cells,



and amend it to each opposite end of our new temporary array,

The periodic behavior of our function allows us to do this where $u[-1] = u[N]$. By assigning a pointer value `*p = u[3]`, we are able to find our derivatives without crossing over our boundary



Testing

Tests were added throughout the code during its creation and in its final document to make sure our code was working as intended. For example, `assert(uval[0] == un[i])` in `rhs.cxx` was used to check that our end values in our gridspace were equal, as well as a `nan` check in `rk4.cxx` to make sure our parameter values did not cause our function to fall out of bounds. It was important to find the right parameters such that our values plotted smoothly but were also plausible mathematically, where, when implausible, produced `nan` values instead of numbers.

Paralellization

To add an element of parallelization to the project, we tried using openmp in various pieces of the code and look for improvements in the run time.

Ideally, the time iteration in the while loop would be the step most in need of improvement, but it is difficult to parallelize within the while loop, and any attempt to parallelize the multiple for loops in each steps resulted in a longer runtime.

We did include a pragma omp parallel for collapse statement to parallelize the nested for loops used to determine the theoretical soliton solutions. There is not a significant improvement in the timing, but it does not seem to slow down the run significantly either.

Plotting

As aforementioned, our plotting was done in a Jupyter Notebook found within `KdV_Plotter`. The output files produced by the kdv executable are automatically read and plotted when run in the notebook. For our data, a small enough timestep Δt had to be chosen to smooth out our numerical data. The next section provides some sample results and measure of accuracy of our numerical solution to the analytical one.

Results

To keep our initial conditions simple, we kept our constants of c and a to be $c = 4$ and $a = 0$ throughout. At $t_0 = 0$, our initial condition equation takes the form,

$$u(x,t) = 2\,\mathrm{sech}^2(x)$$

In Fig. 1, we see the general output for our solved KdV equation with these conditions at some time t.

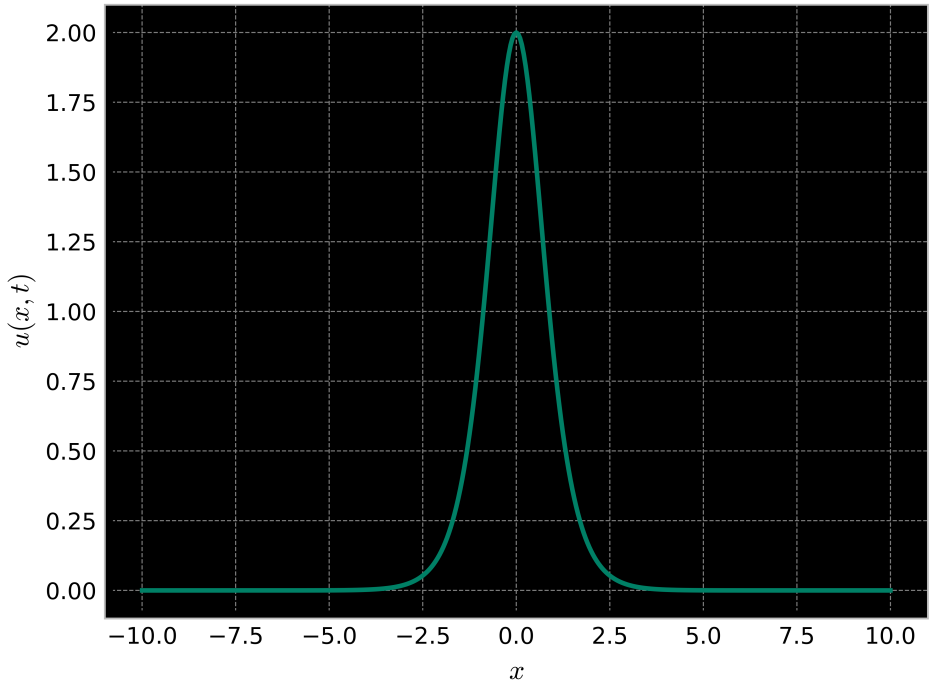


Figure 1: Numerical solution of $\partial_t u + \partial_{xxx} u + 6u\partial_x u = 0$ for $c = 4$ and $a = 0$. Grid cell step chosen as $\Delta x = 0.05$ with timestep $\Delta t = 0.0001$ integrated over the range of time $t = [0.0, 1.0]$.

Plotting both our numerical and analytical solutions together for $t = 0.1$, we find they agree (Fig. 2).

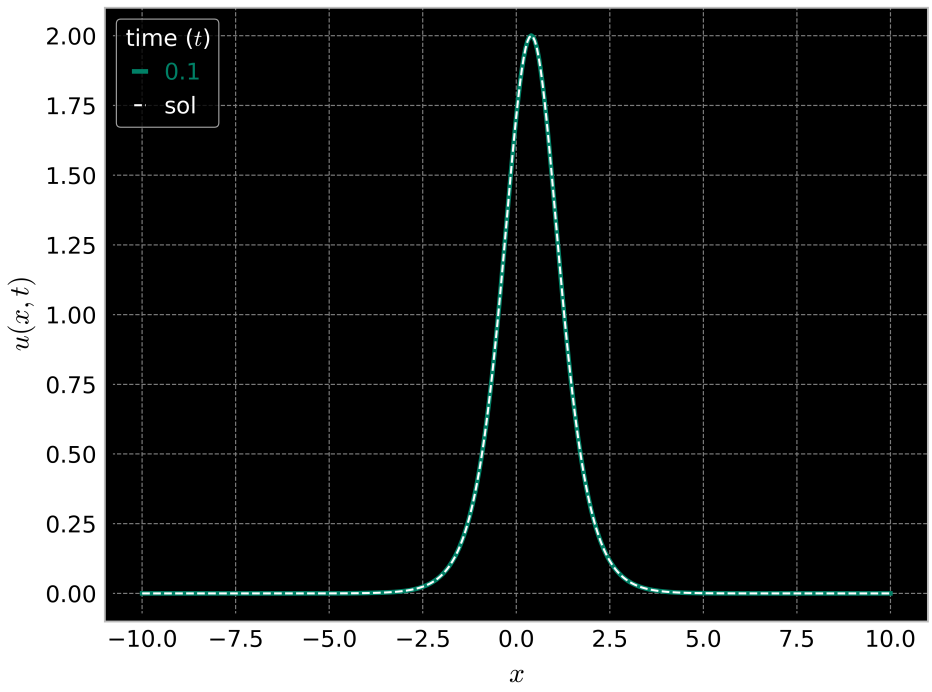


Figure 2: Numerical solution of $\partial_t u + \partial_{xxx} u + 6u\partial_x u = 0$ plotted against our theoretical value at t= 0.1 for $c = 4$ and $a = 0$. Grid cell step chosen as $\Delta x = 0.05$ with timestep $\Delta t = 0.0001$ integrated over the range of time $t = [0.0, 1.0]$.

To further exemplify this, we animated our traveling standing wave on two plots for both our numerical (experimental) and analytical (theoretical) solutions over time. The link to the gif animation is linked in Fig. 3. The code for producing this can also be found in `Kdv_Plotter.ipynb`.

Finally, we produce another plot of our results by superimposing multiple instances of our numerical solution over different times (Fig. 4). Our numerical solution behaves as expected.

Conclusion

The KdV equation can be used to model a traveling standing wave. We were able to numerically solve for this equation by using spatial discretization and finite difference approximation to convert our initial equation into an ODE, and solve $u(x, t)$ by integrating our time-derivative values through the RK4 time-stepping scheme. We provide a few results of

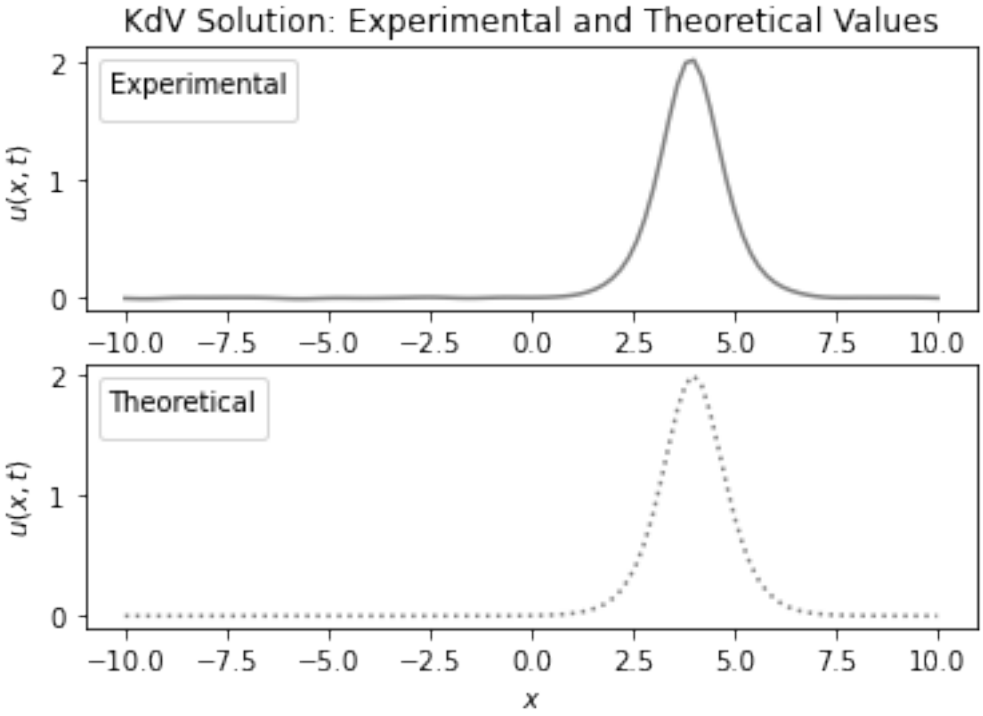


Figure 3: Animation gif file located HERE. Numerical solution of $\partial_t u + \partial_{xxx} u + 6u\partial_x u = 0$ (top) and our analytical values (bottom) for $c = 4$ and $a = 0$. Grid cell step chosen as $\Delta x = 0.05$ with timestep $\Delta t = 0.0001$ integrated over the range of time $t = [0.0, 1.0]$.

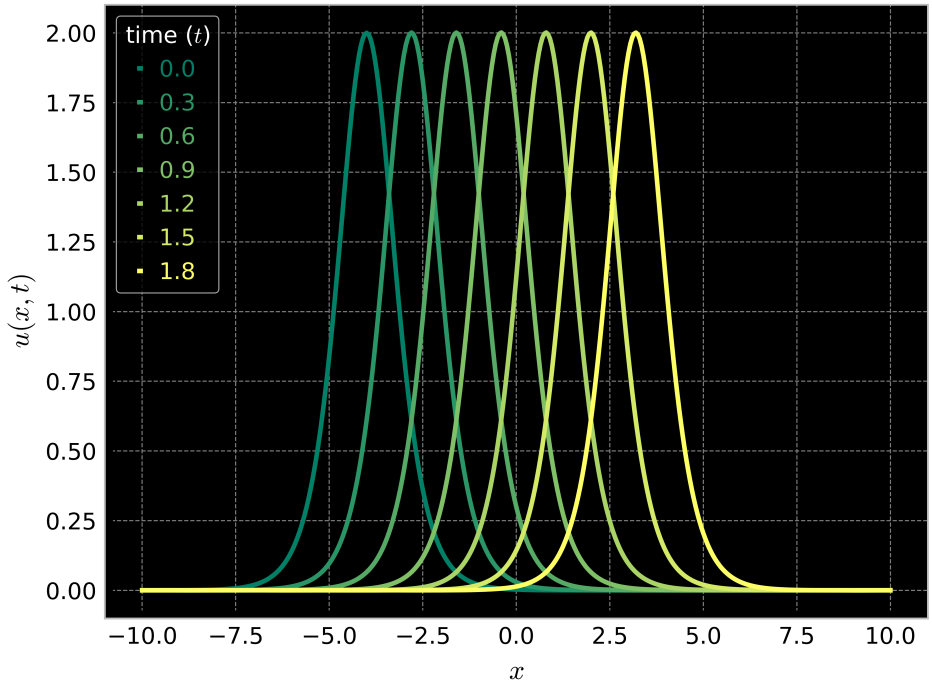


Figure 4: Numerical solution of $\partial_t u + \partial_{xxx} u + 6u\partial_x u = 0$ for $c = 4$ and $a = 0$. Grid cell step chosen as $\Delta x = 0.05$ with timestep $\Delta t = 0.0001$ integrated over the range of time t = [-1.0, 1.0].

this equation with initial conditions of $c = 4$ and $a = 0$ through both plots and an animated figure of our solution over time. Our results are as expected, and we can verify the validity of our numerical solution compared to the analytical form of $u(x, t)$.

Note on Work Distribution

This project was split between Brianna and Dominic with collaboration on all parts. Brianna mainly focused on development of the RK4 code and Dominic on OpenMP and parallelization; however, both partners often worked together simultaneously.