21 Solitons

Lab Objective: *Use a pseudospectral method to study solitons, the traveling wave solutions of the Korteweg-de Vries equation.*

The Korteweg-de Vries (KdV) equation is a partial differential equation given by

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} = 0.$$

that describes shallow water waves.

The KdV equation possesses traveling wave solutions called *solitons*. These traveling waves have the form

$$u(x,t) = 3s\,\mathrm{sech}^2\left(\frac{\sqrt{s}}{2}(x - st - a)\right),$$

where s is the speed of the wave. Solitons were first studied by John Scott Russell in 1834, in the Union Canal in Scotland. When a canal boat suddenly stopped, the water piled up in front of the boat continued moving down the canal in the shape of a pulse.

Note that there is a soliton solution for each wave speed s, and that the amplitude and speed of the soliton determine each other. Solitons are nonlinearly stable (bumped waves return to their previous shape), and they maintain their energy as they travel. Two interacting solitons will also both maintain their shapes after crossing paths.

Numerical solution

Consider the KdV equation on $[-\pi, \pi]$, together with an appropriate initial condition:

$$\begin{aligned} u_t &= -\frac{1}{2}\left(u^2\right)_x - u_{xxx}, \\ u(x,0) &= u_0(x). \end{aligned}$$

This form of the equation is slightly more convenient for the approach we will take. We will suppose the initial condition is equal at the two endpoints; that is, $u_0(-\pi) = u_0(\pi)$. This will allow us to use the pseudospectral method to find a numerical approximation for the solution $u(x,t)$.

As a reminder, the pseudospectral method involves writing the solution at each point in time using a set of basis functions, complex exponentials being the most common, and using this representation to convert the PDE into an ODE. Specifically, we can write any solution $u(x,t)$ as

$$u(x,t) = \sum_{k=-\infty}^{\infty} y_k(t)e^{ikx}.$$

Recall that k is known as the wave number. Note that all time-dependence of the solution is contained in the coefficients. We can only compute this to some finite precision, so we will choose some n and truncate the series as

$$u(x,t) = \sum_{k=-n}^{n} y_k(t)e^{ikx}.$$

The objective is to obtain an ordinary differential equation for the coefficients $y_k(t)$. We now plug it into the PDE:

$$\begin{aligned}\frac{\partial}{\partial t}\sum_{k=-n}^{n} y_k(t)e^{ikx} &= -\frac{\partial}{\partial x}\left(\sum_{k=-n}^{n} y_k(t)e^{ikx}\right)^2 - \frac{\partial^3}{\partial^3 x}\sum_{k=-n}^{n} y_k(t)e^{ikx}\\ \sum_{k=-n}^{n} y_k'(t)e^{ikx} &= -\frac{\partial}{\partial x}\left(\sum_{k=-n}^{n} y_k(t)e^{ikx}\right)^2 + \sum_{k=-n}^{n} ik^3 y_k(t)e^{ikx}\end{aligned}$$

For this particular PDE, this leads to an apparent problem: the u^2 term will be difficult and computationally costly to differentiate. However, we can get around this difficulty using the fast Fourier transform.

Divide $[-\pi,\pi]$ into $2n+1$ intervals of equal width $\frac{2\pi}{2n+1}$, and let $-\pi = x_{-n}, x_{-n+1}, \ldots, x_n, x_{n+1} = \pi$ be the $2n+2$ evenly-spaced gridpoints. For any function f on that interval with Fourier series $f(x) = \sum_{k=-\infty}^{\infty} a_k e^{ikx}$, we can use the discrete Fourier transform on the values $f(x_{-n}), \ldots, f(x_{n+1})$ at the gridpoints to quickly get the Fourier coefficients $a_{-n}, \ldots, a_n$. The inverse Fourier transform can be used to get the function values at the grid points from the Fourier coefficients. Both of these operations are very efficient, having complexity $O(n\log n)$. This sets up our strategy.

At each time t, we can use the inverse Fourier transform to compute the values of $u(x_m,t)$ for $m = -n, \ldots, n+1$. Then, we apply the Fourier transform to u^2 to get its Fourier coefficients. We will denote these as w_k, so

$$u^2(x,t) = \sum_{k=-n}^{n} w_k(t)e^{ikx}.$$

Then,

$$\frac{\partial}{\partial x}u^2(x,t) = \sum_{k=-n}^{n} ikw_k(t)e^{ikx},$$

so the KdV equation can be written as

$$\sum_{k=-n}^{n} y_k'(t)e^{ikx} = \sum_{k=-n}^{n}\left(-\frac{1}{2}ikw_k(t) + ik^3 y_k(t)\right)e^{ikx}$$

Equating terms in the Fourier series yields the ordinary system of differential equations

$$y_k' = -\frac{1}{2}ikw_k + ik^3 y_k, \quad k = -n, \ldots, n.$$

We can also write this in a vectorized form as

$$\mathbf{y}' = -\frac{1}{2} i\mathbf{k}\mathcal{F}(\mathcal{F}^{-1}(\mathbf{y})^2) + ik^3\mathbf{y} \tag{21.1}$$

where $\mathcal{F}$ denotes the discrete Fourier transform and multiplication of vectors is componentwise. To obtain the initial condition for the y_k, we can simply use the discrete Fourier transform again:

$$\mathbf{y}(0) = \mathcal{F}(u_0(x_{-n}), \ldots, u_0(x_{n+1}))$$

To compute the fast Fourier and inverse fast Fourier transforms numerically, we will use the `scipy.fft` module, which has functions `fft` for the fast Fourier transform and `ifft` for the inverse fast Fourier transform. These functions use an order for the coefficients that is slightly nonintuitive: the coefficients for $k \geq 0$ are all listed first, followed by the coefficients for $k < 0$. The vector of wavenumbers can be created as follows:

```
k = np.concatenate([
        np.arange(0,n+1),
        np.arange(-n-1,0)
])
```

We are now prepared to numerically solve the KdV equation.

Problem 1. Write a function that accepts the time value t (which won't be used here, but will be useful later) the vector $\mathbf{y} = (y_0, y_1, \ldots, y_n, y_{-n-1}, \ldots, y_{-1})$ and the vector of k values and returns $\mathbf{y}'$.

To numerically solve this ODE, use the following implementation of the RK4 algorithm:

```
def RK4(f, y0, T, dt, k):
    """
    Solves the ODE y'=f(t, y) using the Runge-Kutta 4 method with initial
    condition y0 on the time interval [0,T] using a time step of dt.
    The value of k is passed directly into the function f.

    Returns:
        t ((T,) ndarray) - the time values
        Y ((T, 2n+2) ndarray) - the solution values. The solution at the
                i-th time step can be indexed as Y[i].
    """
    # Set up matrices for the solution
    ts = np.arange(0, T+dt, dt)
    Y = np.empty((len(t),len(k)), dtype=complex)
    y = y0
    Y[0] = y
    for i in range(1, len(t)):
        # Use RK4
        t = ts[i]
        K1 = f(t, y, k)
```

```python
        K2 = f(t + dt/2, y + 0.5*dt*K1, k)
        K3 = f(t + dt/2, y + 0.5*dt*K2, k)
        K4 = f(t + dt, y + dt*K3, k)
        y = y + (dt / 6.) * (K1 + 2*K2 + 2*K3 + K4)
        Y[i] = y
    return ts, Y
```

Once we have solved for the coefficients $\mathbf{y}(t)$, we need to convert them back into function values $u(x, t)$ in order to visualize the solution. This is accomplished by using the `ifft` function on the coefficient values at each time step. However, this function is designed to work with complex numbers, and returns a complex-valued array. Due to numerical error, even though our ODE solution is real-valued, there may be small imaginary components to the result; use `np.real` on the result to discard these.

Problem 2. Write a function that accepts an initial condition u_0, a final time `T`, the timestep `dt`, an integer n for the number of coefficients to use, and another integer `skip`. Numerically solve for the coefficients $\mathbf{y}(t)$ of a solution to the KdV equation.

Next, convert the Fourier coefficients back into function values at the gridpoints using the inverse Fourier transform. However, only do this for every `skip`-th timestep; we will be using far more timesteps than we want to plot. Return the gridpoints, the timesteps, and the solution $u(x, t)$.

Once we have the function values, we can plot them as a surface as follows:

```python
fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')

T, X = np.meshgrid(t, x, indexing='ij')
ax.plot_surface(T, X, u, cmap='coolwarm', rstride=1, cstride=1)
```

Problem 3. Numerically solve the KdV equation on $[-\pi, \pi]$ with initial conditions

$$u(x, t = 0) = 3s\,\text{sech}^2\left(\frac{\sqrt{s}}{2}(x + a)\right),$$

where $s = 25^2$, $a = 2$. Solve on the time domain $[0, 0.0075]$, and use $n = 127$. Compare with Figure 21.1; to get a similar viewpoint, use the following:

```python
ax.view_init(elev=45, azim=-45)}
ax.set_zlim(0, 4000)
ax.invert_xaxis()
```

How small of a timestep did you need to use for the numerical integration to be stable?

Hint: `numpy` does not have a `sech` function; use `1/cosh(x)` to compute it instead.

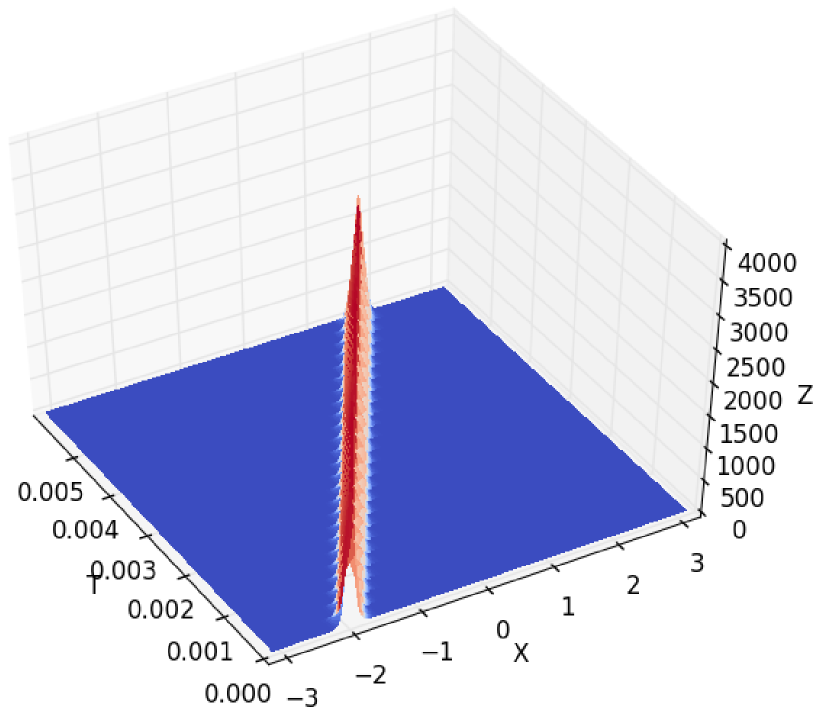

Figure 21.1: The solution to Problem 3.

Problem 4. Numerically solve the KdV equation on $[-\pi, \pi]$. This time we define the initial condition to be the superposition of two solitons:

$$u(x, t=0) = 3s_1 \operatorname{sech}^2\left(\frac{\sqrt{s_1}}{2}(x+a_1)\right) + 3s_2 \operatorname{sech}^2\left(\frac{\sqrt{s_2}}{2}(x+a_2)\right),$$

where $s_1 = 25^2$, $a_1 = 2$, and $s_2 = 16^2$, $a_2 = 1$.[a] Solve on the time domain $[0, 0.0075]$. The solution is shown in Figure 21.2.

[a]This problem is from *Spectral Methods in MATLAB*, by Trefethen.

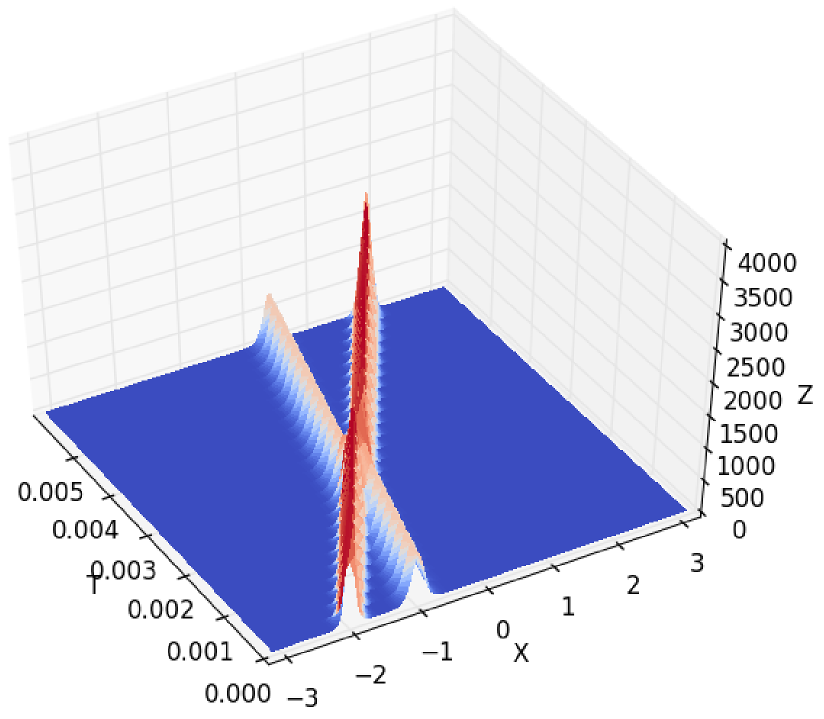

Figure 21.2: The solution to Problem 4.

Problem 5. Consider again equation (21.1). The linear term in this equation is $i\mathbf{k}^3\mathbf{y}$. This term contributes much of the exponential growth in the ODE, and contributes to how short the time step must be to ensure numerical stability. Make the substitution $z_k(t) = e^{-ik^3t}y_k(t)$ and find a similar ODE for $\mathbf{z}$. This essentially allows the exponential growth to be scaled out (it's solved for analytically, replacing it with rotation in the complex plane). Use the resulting equation to solve the previous problem. How much larger of a timestep can you use while this method remains stable?