University of Colorado Boulder

APPM 4350: Fourier Series and Boundary Value Problems

Final Project Report

# Soliton Analysis: An Adventure in Higher Order, Nonlinear PDEs

*Authors*
Kevin Stull
Santiago Velasco
Gabe Wallon

January 3, 2023

**Abstract**

We test the theory that a soliton's speed is a function of its maximum amplitude by analyzing it's behavior in a viscous fluid conduit system. Given a mathematical model of the soliton's cross sectional area as a function of its location and time, we numerically solve for the equation describing the shape of the soliton and analytically solve for the theoretical relationship between the speed and amplitude of a traveling wave. We compare our theoretical solutions to the provided experimental data to confirm the accuracy of the derived speed-amplitude relation and solution to the equation of a soliton's shape.

## Attributions

Santiago, Kevin, and Gabe all contributed to the derivation of the speed-amplitude relation. Santiago formatted the final LaTeX document. Gabe wrote the code to verify the derived speed-amplitude relation according to the experiments. Kevin wrote the code to numerically solve for the shape of the wave.

# Introduction

A "solitary wave", or "soliton", is a physical phenomenon which arises in a variety of scientific contexts. A soliton is a localized wave of permanent form that travels with a fixed speed determined by its maximum wave amplitude. It has a single maximum and the amplitude tends to zero away from its point of origin (Figure 1).

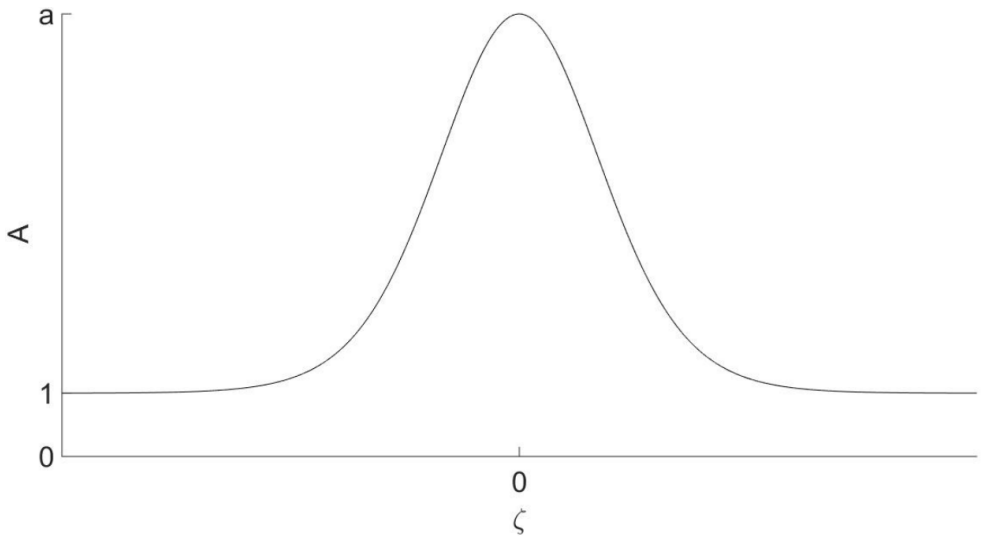


Figure 1: Soliton Wave with Maximum Amplitude a

The equation that models a soliton is a third order non-linear partial differential equation. The theoretical relation between the speed and the amplitude of a soliton will be obtained and compared to experimental results. The data we use in our analysis comes from a previously

performed experiment illustrated in the figure found in the next page. In this experiment [1],

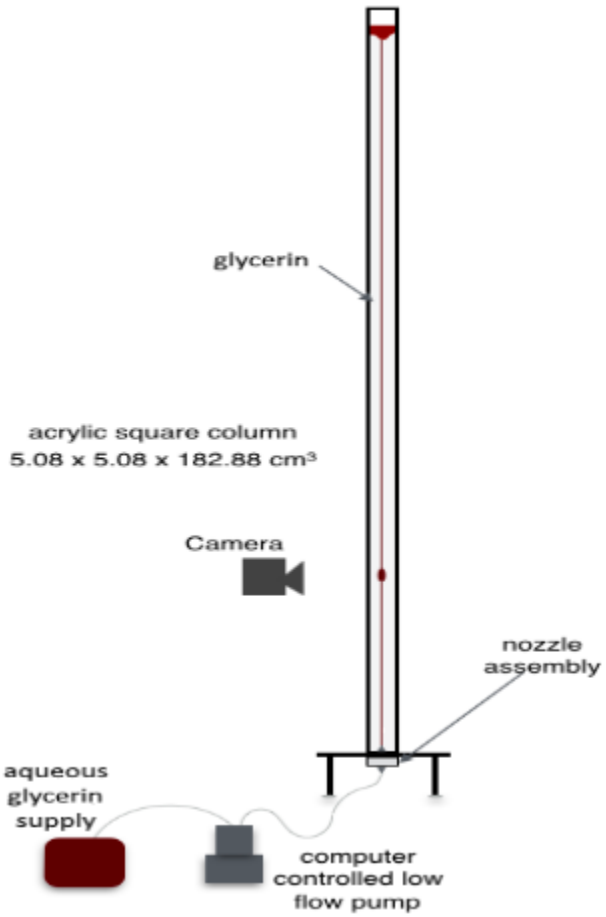


Figure 2: Experimental Setup

a pump pulse-modulates the flow rate of a lower density liquid into a tube filled with a higher density liquid, diluted Glycerin and Glycerin, respectively. The greater flow rate creates a bulge in the lower density conduit (red line in image). This bulge behaves like a soliton with a maximum amplitude equal to its maximum cross-sectional area. As this experiment exhibits the general behavior of a soliton, in this report we will use *bulge* and *wave* interchangeably, as well as *area* and *amplitude*.

# Model Development and Mathematical Formulation

The mathematical model for a wave begins with the Korteweg-de Vries Equation, which is a model for shallow water waves which act as solitonic waves. The equation is a third order, nonlinear partial differential equation, which must be simplified into a first order ODE prior to applying boundary conditions. Using this background, we determine that the equation [2] below describes the cross sectional area of a soliton in a conduit system, as a function of position and time.

$$A_t + (A^2)_z - \left( A^2 \left( \frac{A_t}{A} \right)_z \right)_z = 0 \tag{1}$$

where we impose the boundary conditions:

$$\begin{aligned} &\lim_{z \to \pm\infty} A(z,t) = 1 \\ &\lim_{z \to \pm\infty} A_z(z,t) = 0 \\ &\lim_{z \to \pm\infty} A_{zz}(z,t) = 0 \\ &\text{where } A(0,t) = a,\ A'(0,t) = 0 \end{aligned} \tag{2}$$

We perform a change of variables and let $A(z,t) = f(\zeta)$, where $\zeta = z - ct$, since we assume that the solution to the PDE will resemble a rightward traveling wave. Thus, we get the following equation with an unknown constant $c$, which we must eventually solve.

$$-cf' + (f^2)' - \left( f^2(-cf^{-1}f')' \right)' = 0 \tag{3}$$

The boundary conditions become:

$$\begin{aligned} &\lim_{\zeta \to \pm\infty} f(\zeta) = 1 \\ &\lim_{\zeta \to \pm\infty} f'(\zeta) = 0 \\ &\lim_{\zeta \to \pm\infty} f''(\zeta) = 0 \\ &\text{where } f(0) = a,\ f'(0) = 0 \end{aligned} \tag{4}$$

We can simply integrate Equation (3) once and obtain:

$$-cf + f^2 - f^2 \left( -cf^{-1}f' \right)' = D \tag{5}$$

Then, we take the limit as $\zeta \to \pm\infty$, applying the boundary conditions in (4) and solving for $D$,

$$\begin{aligned} -c(1) + (1)^2 - (1)^2 \left( -c\frac{1}{(1)}(0) \right)' &= D \\ \underline{-c + 1 = D} & \end{aligned}$$

We substitute $D$ into Equation (5) and multiplying by $-1$, obtaining:

$$cf - f^2 - f^2 \left( cf^{-1}f \right)' = c - 1 \tag{6}$$

We then use the integrating factor of the form $f'f^{-3}$ and apply it to both sides.

$$cf^{-2}f' - f^{-1}f' - f^{-1}f'\left(cf^{-1}f'\right)' = (c-1)f^{-3}f' \tag{7}$$

From here, we can integrate to find:

$$-cf^{-1} - \ln|f| - \frac{c}{2}\left(f^{-1}f'\right)^2 = \frac{-(c-1)}{2}f^{-2} + D'$$

Multiplying by -2 will then yield,

$$2cf^{-1} + 2\ln|f| + c(f^{-1}f')^2 = (c-1)f^{-2} + D'$$

which we can convert to the below expression using properties of logarithms:

$$2cf^{-1} + \ln|f^2| + c(f^{-1}f')^2 = (c-1)f^{-2} + D'$$
$$2cf^{-1} + \ln(f^2) + c(f^{-1}f')^2 = (c-1)f^{-2} + D'$$

We are now able to apply the boundary conditions in order to obtain the result of $D'$:

$$2c\frac{1}{(1)} + \ln((1)^2) + c\left(\frac{1}{(1)}(0)\right)^2 = (c-1)\frac{1}{(1)^2} + D'$$
$$2c + \ln(1) + c(0)^2 = (c-1) + D'$$
$$2c = (c-1) + D'$$
$$\underline{c+1 = D'}$$

Thus, we can now get a final equation, which is simply a first order ODE, the desired result:

$$\boxed{2cf^{-1} + \ln(f^2) + c(f^{-1}f')^2 = (c-1)f^{-2} + (c+1)} \tag{8}$$

We will then apply the boundary conditions where $\zeta = 0$ from (4) to solve for speed $c$, in terms of max soliton amplitude $a$:

$$2c\frac{1}{(a)} + \ln((a)^2) + c\left(\frac{1}{(a)}(0)\right)^2 = (c-1)\frac{1}{(a)^2} + (c+1)$$
$$\frac{2c}{a} + \ln(a^2) = \frac{c-1}{a^2} + (c+1)$$
$$2ac + a^2\ln(a^2) = c - 1 + a^2(c+1)$$
$$a^2\ln(a^2) + 1 = a^2c + a^2 + c - 2ac$$
$$a^2\ln(a^2) - a^2 + 1 = a^2c + c - 2ac$$
$$a^2(\ln(a^2) - 1) + 1 = c(a^2 - 2a + 1)$$
$$a^2(\ln(a^2) - 1) + 1 = c(a-1)^2$$

$$\therefore \quad \boxed{\frac{a^2(\ln(a^2) - 1) + 1}{(a-1)^2} = c} \tag{9}$$

# Speed-Amplitude Experimental Analysis

## Building a function

If we plot the derived speed-amplitude relation (9), with values of $a > 1$, we can visualize their relationship:

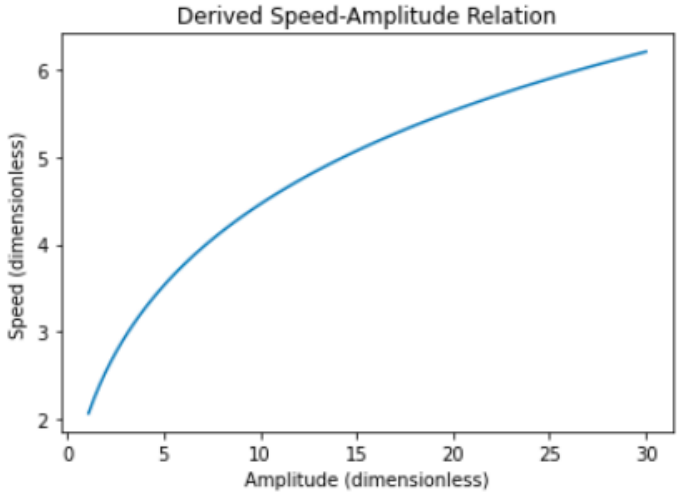


Figure 3: Derived Speed-Amplitude relation

To verify that this relationship is consistent with reality, we must analyze the given experimental data. Below is a table of the given information in each data set and a contour plot of the given amplitude matrix (Figure 4 and 5, respectively).

| **Variable Name** | **Variable Type** | **Variable Description** |
|---|---|---|
| t | vector | time vector (seconds) |
| z | vector | spacial vector (cm) |
| A | matrix | amplitude (dimensionless) |
| U0 | scalar | velocity scale (cm/s) |
| A0 | scalar | soliton amplitude (dimensionless) |
| A0 error | scalar | soliton amplitude error (dimensionless) |

Figure 4: Variable Descriptions of Data Set

Time and vertical distance are not properly scaled in this plot, and the values of the matrix elements are in dimensionless units of area. This plot shows the shape of the dispersive shock wave, one large bulge in the conduit, followed by bulges of decreasing cross sectional area. The unperturbed conduit, represented by the purple area in the plot, has dimensionless cross sectional area equal to 1.0. The leading edge, or light green line furthest to the right in the plot, behaves like a soliton and should exhibit our speed-amplitude relation. To find the speed of this soliton, we must find a way to extract the inverse slope (distance / time) of this leading

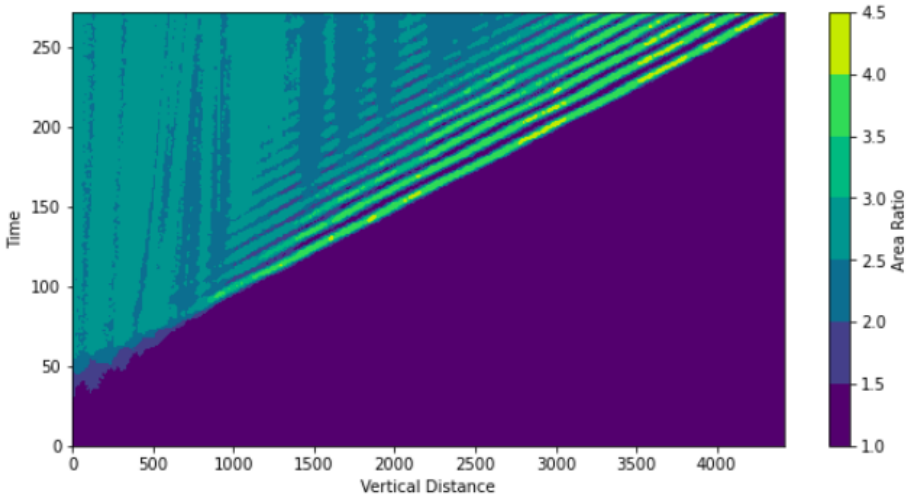


Figure 5: Contour Plot of Amplitude Matrix

edge. We create a function to do this. This function looks at the value of every index in the given amplitude matrix and adds the index to a list (`fullWaveIdxs` in the function) if its value is within the range [A0 - A0_error , A0 + A0_error]. Next, the function makes a list of all the column indices of the elements in `fullWaveIdxs`, `columns`, and iterates through each unique column. Within this loop, a temporary list of all the `fullWaveIdxs` in the column is added to a new list, `column_idxs`. This is followed by the `column_idxs` element with the smallest time index value being added to the `solitonIdxs` list. This code can be seen in Figure 6.

```
def compute_soliton_speed(dictionary):
    mat = dictionary['Amat']
    times = mat.shape[0]
    distances = mat.shape[1]
    a0 = dictionary['A0'][0,0]
    a0_error = dictionary['A0_error'][0,0]
    solitonAmpRange = ((a0 - a0_error), (a0 + a0_error))

    # first get possible soliton indexes
    fullWaveIdxs = []
    for i in range(times):
        for j in range(distances):
            if (mat[i,j]>solitonAmpRange[0]) & (mat[i,j]<solitonAmpRange[1]):
                fullWaveIdxs.append((i,j))

    # next, get the index with the smallest time coordinate from every column where there is a soliton
    solitonIdxs = []
    columns = [idx[1] for idx in fullWaveIdxs]
    for d in set(columns):
        column_idxs = [idx for idx in fullWaveIdxs if idx[1] == d]
        solitonIdxs.append(min(column_idxs))
```

Figure 6: compute_soliton_speed function 1

Next, the function changes the order of the index coordinates in each of the `solitonIdxs` elements so that they can be interpreted in Cartesian coordinates. We do this because we

next use the built-in function *numpy::polyfit* to find the least squares degree one polynomial fit, which requires Cartesian coordinate vector arguments, `xvals` and `yvals` in the function. We choose a degree one polynomial as the soliton appears to be moving at a constant velocity, and, by definition of a soliton, we expect speed to be fixed. *numpy::polyfit* returns a slope, $m$, and intercept, $b$, value, where $1/m$ is the speed of the soliton on the matrix index scale (ie., the change in z, or distance indices, per change in t, or time indices). The associated $R^2$ value for the linear model was 0.99, which supports the validity of our model. The code used for this function is displayed in Figure 7.

```
# change order of the coordinates for each soliton index to switch from matrix indexing to cartesian coordinates
solitonIdxs = [(idx[1],idx[0]) for idx in solitonIdxs]

# separate the time values and the distance values for use of polyfit() function
xvals = [idx[0] for idx in solitonIdxs]
yvals = [idx[1] for idx in solitonIdxs]
m, b = np.polyfit(xvals, yvals, deg = 1)
```

Figure 7: compute_soliton_speed function 2

This speed we call the `index_speed` of the soliton. To get the real experimental speed of the soliton, we must adjust the `index_speed` according to the real time and distance vectors given in the data. We make a simple conversion, `cm_per_zidx`, dividing the total length of the tube by the length of the distance vector, `z_vec`. Then we multiply the `index_speed` by this conversion to get `real_distance`, in cm / time index (tidx). We use the same conversion method to get `sec_per_tidx`, then divide `real_distance` by `sec_per_tidx` to get `real_speed`, the speed of the soliton in cm/sec.

```
# calculating real_speed by adjusting scales of the matrix indices
index_speed = 1/m    # zidx / tidx
cm_per_zidx = dictionary['z_vec'][-1][-1] / dictionary['z_vec'].size
real_distance = index_speed * cm_per_zidx  # cm / tidx
sec_per_tidx = dictionary['t_vec'][-1][-1] / dictionary['t_vec'].size

if dictionary['t_vec'][dictionary['t_vec'] < 0].size > 0:  # in case the time vector has negative time values
    average = 1.2371625245444737
    sec_per_tidx = average

real_speed = real_distance / sec_per_tidx  # cm / sec

# return all useful information for tracking speed and for plotting
return {'real_speed':real_speed, 'index_speed':index_speed,
        'solitonIdxs':solitonIdxs, 'xvals':xvals, 'yvals':yvals, 'b':b}
```

Figure 8: compute_soliton_speed function 3

We noticed that in two of our data sets, the given time vector, `z_vec` had values in around the range of negative 3,000. We realized that this must have been some kind of error thus we found the average `sec_per_tidx` conversion from all of the data sets, to use in the case that a `z_vec` had negative values (seen above in Figure 8). The code for how we computed this average conversion can be found in the appendix. The function returns a dictionary with the `real_speed` as well as other values which make plotting more convenient.

## Visualizing the Relationship

We proceed by applying the `compute_soliton_speed` function to every data set. By plotting the contour plot of an experiment's amplitude matrix next to a plot of the corresponding `solitonIdxs` and their line of best fit, we can get an idea of how well the function performed in calculating an accurate soliton speed. Below (Figure 9) are some examples of experiments where the function appears to be accurately measuring the slope of the soliton. The following page contains some examples of data sets that did not match our function's predictions(Figure 10).

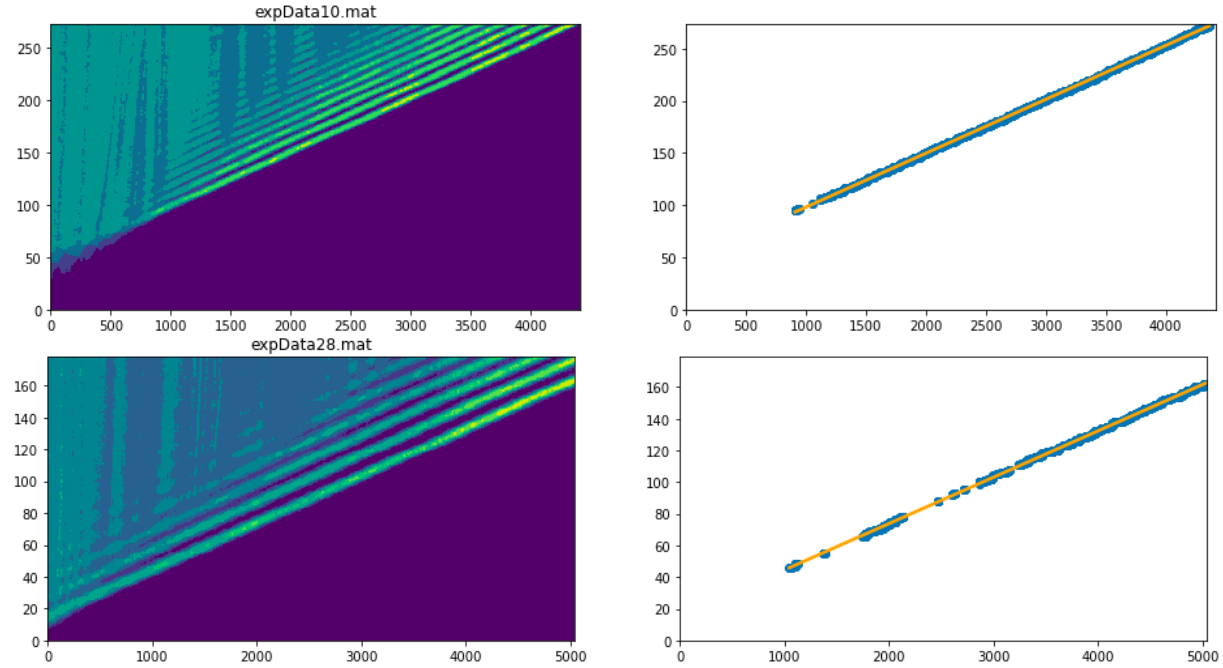


Figure 9: Matrices Compatible With Our Function

While computing all of these speeds, we append the A0 values for each experiment in sequential order to a list called `a0_list`, and the corresponding U0 values to `u0_list`. Now that we have our experimental speeds, we compute the theoretical speed for each A0, or soliton amplitude, in `a0_list`, and then scale this speed by the corresponding speed scale in `u0_list`. More explanation on this scaling procedure can be found in discussion question 2. The code for this process can be seen below in Figure 11.

Finally, we can compare our experimental findings to our theoretical predictions. Figure 12, on the next page, shows the speed-amplitude relation for every experiment except one. The outlier being the single data set that the `compute_soliton_speed` function was not compatible with. The data set and experiment number are included (Figure 10). The plot on the right, (Figure 12) displays the calculated theoretical speeds of the solitons. The plot on the left displays their experimentally observed behavior.

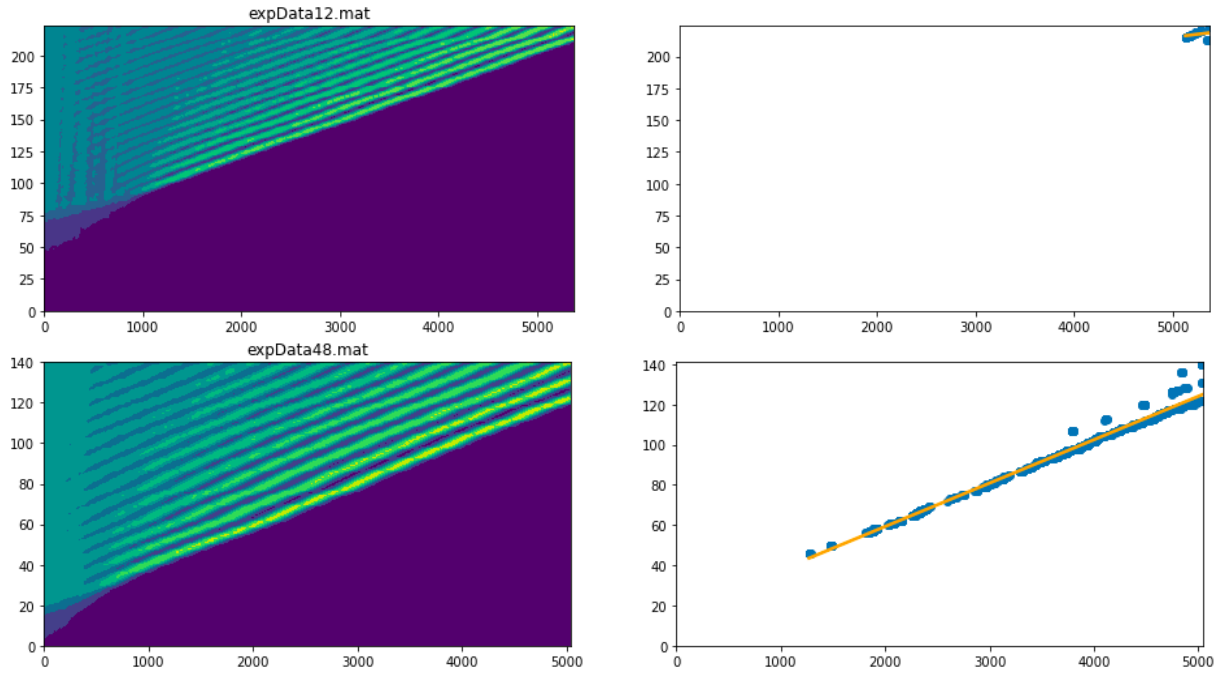


Figure 10: Matrices Not Compatible With the Function

```
# finding the dimensionalized theoretical speeds for comparison to the experimentally found speeds
theo_speeds_list = []
for a0 in a0_list:
    theo_speed = (a0[1]**2 * np.log(a0[1]**2) - a0[1]**2 + 1) / ((a0[1]-1)**2)
    theo_speeds_list.append(theo_speed)

theo_speeds_arr = np.array(theo_speeds_list)
corresponding_u0_arr = np.array(u0_list)
scaled_theos = theo_speeds_arr * corresponding_u0_arr
```

Figure 11: Calculating Theoretical Speeds

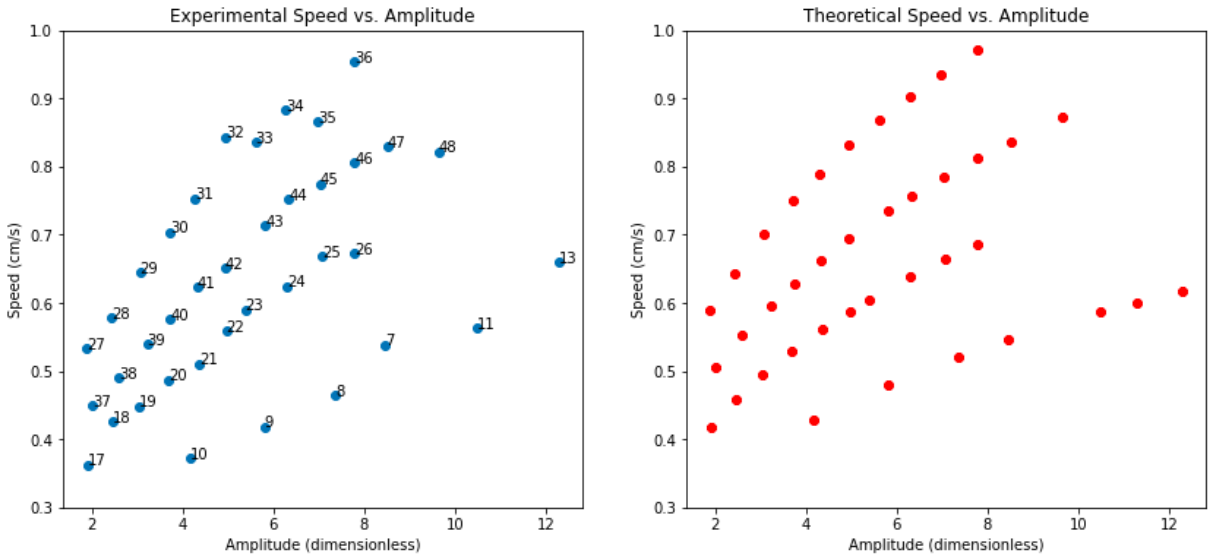


Figure 12: Initial Experimental Observations vs. Theoretical Predictions

Using the aforementioned plots, we infer that three trials per amplitudes A0, each with different speed scales U0, were performed. Experiments 27-36 used the first U0 scaling factor, 37-48 used the next highest, and so on. These scaling factors (U0) correspond to the viscosity of the conduit's (lower density) liquid. We believe the reason for these three trials was to confirm that the speed-amplitude relation holds true even with varying fluid properties. If we plot each of these groups of experiments independently, it becomes clear that a larger amplitude does indeed correspond to a faster soliton, as seen below (Figure 13).

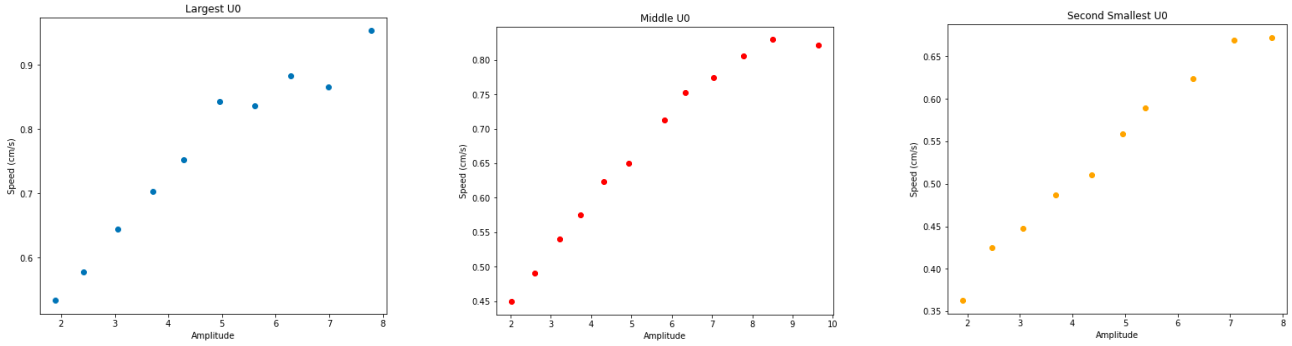


Figure 13: Experiments Grouped by Speed Scale U0

## Verifying Accuracy

To check the accuracy of our theoretical speed predictions, we must propagate the error from the measurement of A0 into our speed calculations. To do this, we use the general formula for error propagation, applied to our function $c(a)$ (10).

$$\delta c = \left| \frac{dc}{da} \right| \delta a \tag{10}$$

We then hand-picked a list of experiments which appeared to be most compatible with the `compute_soliton_speed` function, and calculated the errors in the theoretical speeds for those experiments. The code Below is a plot comparing the experimental speeds to the theoretical speeds for each of these experiments. Also below is a similar plot for all the experiments with middle U0 speed scales (Figure 14).

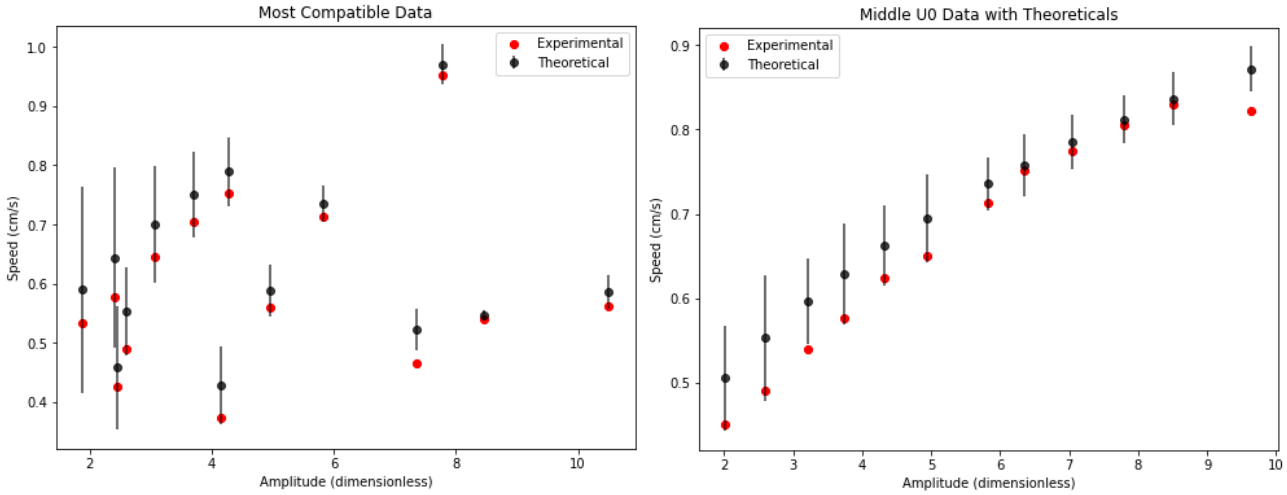


Figure 14: Experimental vs. Theoretical Speeds With Error Bars

An immediate observation is that our experimentally found speeds are all lower than their corresponding theoretical speeds. This could be explained by some kind of energy loss taking place in the experimental setting, that does not exist in a theoretical context. The conduit equation assumes a *sharp interface* between the two liquids, meaning that they do not mix together as the lower density passes through the higher. The liquid used in our particular experiment, Glycerin, has miscible properties, meaning it easily forms homogeneous mixtures when in contact with other liquids. We believe this to be the cause of the disparity in our experimental and theoretical speeds. A less *sharp* interface between the two liquids we assume would cause some kind of frictional effect to occur, slowing down our solitons. This would also explain the larger gaps between experimental and theoretical observed with our solitons of smaller amplitude A0, which with less mass, we suspect would be more susceptible to this friction effect.

# Numerical Results

There is no analytical solution to the first-order ODE. Thus, numerical methods must be used. We execute this approach using Euler's method. In order to find $f$, we must use its derivative, $\frac{df}{d\zeta}$, which will be labeled as $g(f, c)$. This can be obtained directly from (8), and the result will be:

$$\frac{df}{d\zeta} = g(f, c) = \sqrt{\frac{(c-1) + (c+1)f^2 - 2cf - f^2 \ln(f^2)}{c}}$$

However, a better way to solve for $f$ is to solve for its inverse, $\zeta(f)$. To do so using Euler's method, one must use $\frac{d\zeta}{df}$, which can be easily obtained by inverting $g(f, c)$. We must also be aware that since $\frac{df}{d\zeta}$ is 0 at $\zeta = \pm\infty$ and $\zeta = 0$, we must restrict the domain to $\zeta = (-\infty, 0)$ and solve for only one half of the soliton and reflecting the function to obtain the other half. Thus, we can obtain $\zeta(f)$ as such:

$$\zeta_{i+1} = \zeta_i + h\frac{d\zeta}{df} = \zeta_i + \frac{h}{g(f_i, c)} = \zeta_i + \frac{h}{\sqrt{\frac{(c-1)+(c+1)f_i^2 - 2cf_i - f_i^2 \ln(f_i^2)}{c}}}$$

where $\zeta_i = \zeta(f_i)$ and the initial position will be at $f_0 = 1 + \delta$ (where $\delta$ is an arbitrarily chosen tolerance). The initial position is chosen as such because it is near to where $f = 1$ and $f' = 0$. Since the $\zeta$ vector will later be shifted by a constant, the initial position is inconsequential, so long as the entire curve can be generated.

Using an arbitrarily chosen step size $h$, a vector of the form

$$f = [f_0, f_0 + h, f_0 + 2h, ..., f_n]$$

and $f_n$ is the value corresponding to the maximum magnitude $a$. Finally, we shifted the $\zeta$ vector such that $\zeta(f_n) = 0$ (since $f(0) = a$) and reflected the solution in order to obtain a solution:

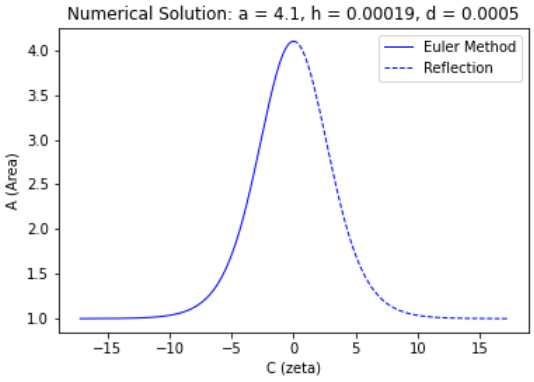


Figure 15: Solution of Soliton Using $a = 4.1$, $h = 0.00019$, $\delta = 0.0003$

With the obtained solution (Figure 15), we can change the parameters and visualize the results. We first start by changing the choice of tolerance:

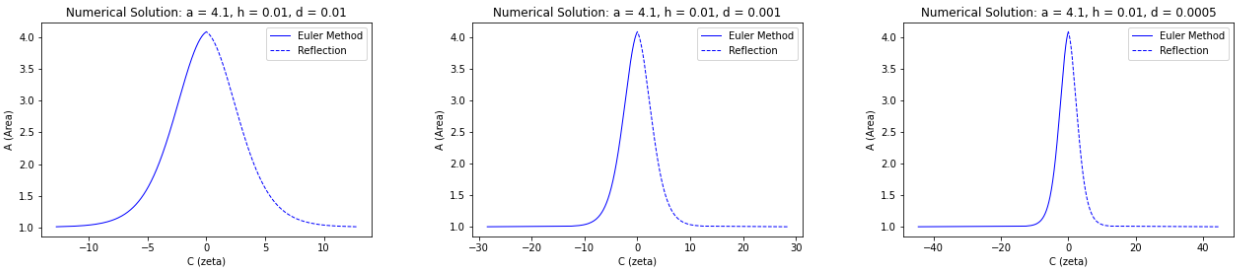


Figure 16: Effects of Changing Tolerance

As can be visualized, **decreasing tolerance will compress the shape of the wave** (Figure 16). In code, it simply results in the starting value of the curve being slightly higher. Theoretically, however, an increase in the tolerance accounts for the experimental and numerical uncertainty; as the model becomes more uncertain regarding the initial conditions of the wave, its distribution is widened to account for this uncertainty. This is analogous to decreasing the sample size in the Student's $t$-distribution – as more degrees of freedom are added (i.e., information becomes available to the model), it becomes more certain of the correct curve for a set of given initial conditions.

This convergence on a theoretical solution can also be seen through the modification of the step size (Figure 17). The numerical solution attempts to approximate that solution and the step size determines how close that approximation will be. As the step size approaches 0, the numerical approaches the theoretical. In the code, step size determines how much space there is between each entry in the $f$ vector. Which are used as inputs to the Euler approximation $\zeta(f)$. The theoretical solution will never be determined by this method, because a step size of 0 would require infinite amount of entries in the $f$ vector, which would take infinitely long to compute. Thus, we change the step size $h$ and visualize the shape of the wave as $h \to 0$:

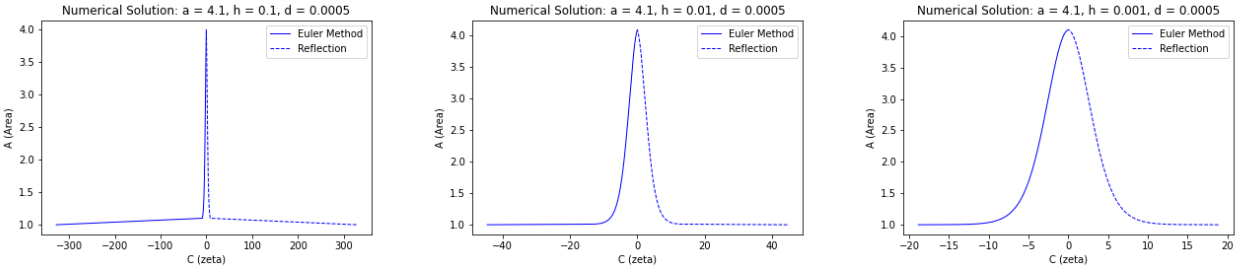


Figure 17: Effects of Changing Step Size

Clearly, **as step size decreases, the solution approaches a limit and thus becomes a smoother graph**.

Finally, we can visualize solitons at different amplitudes. The behavior of a wave that has a small amplitude which approaches 1 will look like such:

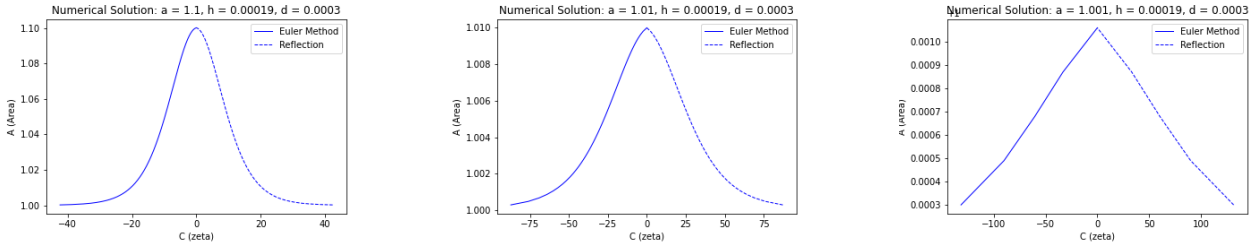


Figure 18: Solitons as $a \to 1$

This is much different than the shape of solitons with large amplitudes, which are picured below

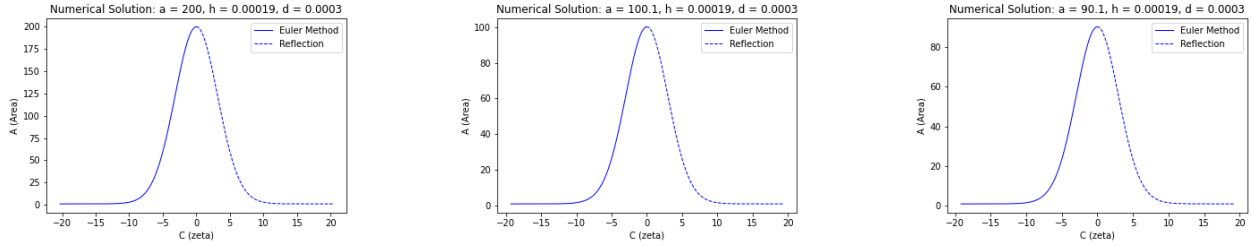


Figure 19: Large Amplitude Solitons

From the two sets of plots, we can infer that in Figure 17, **when $a$ is large, a change in the amplitude won't affect the shape of the soliton as drastically as a change in amplitude when $a$ is close to 1**. Interestingly enough, as $a \to 1$, the graph appears to lose its curvature and the amplitude of the graph very slowly approaches its minimum of 1. This fact is elucidated by the large values of $\zeta$ on the x axis of the rightmost plot (Figure 18). This is in accord with the speed-amplitude relationship. As discussed previously, we expect a soliton with a smaller amplitude to travel at a lower, fixed speed. It is also important to note that since the derivative of $f$ evaluates to 0 when $a = 1$, this leads to a division by zero in the numerical approximation, which is undefined. For values very near zero, the solution is very wide; it is expected behavior of any curve near a singularity. As $a$ increases, the solution quickly starts to take the familiar soliton shape. As one would expect, this trend holds true for larger and larger values of $a$. For very large values of $a$, the curve takes the form of a tall, narrow solitary wave. This leads to the conclusion that solitary waves are at their widest when they are short and at their narrowest when they are tall.

# Discussion and Conclusions

1. As seen above in the *Visualizing the relationship* section, our analysis of the experimental data does indeed show a positive correlation between soliton amplitude and speed.

2. Each experimental data set provides a value U0, a speed scaling factor, which is used to dimensionalize our theoretical speed calculation from dimensionless units to cm/sec. This speed scaling factor is dependent on the viscosity of the conduit liquid, the liquid of lower density.

3. Using the data, we can plot it against the theoretical waves:

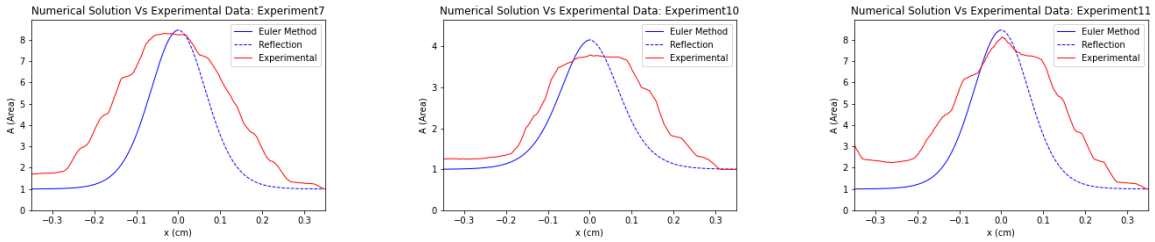


Figure 20: Theoretical vs. Experimental Waves for Experiments 7, 10, 11

Upon inspecting the three plots shown above (Figure 20), we see that the amplitude of all three roughly matches the amplitude of the experimental waves. It is worth nothing that there is rarely an exact match due to a margin of error between the expected amplitude and the measured amplitude. It seems to be the case that the numerical model underestimates the amount of area under the wave. This is analogous to under-estimating how much water (technically lower density glycerin) is in each wave.

It is possible that this is a symptom of the implementation of the Euler approximation. Due to the arrangement of the code, the step size can never be smaller than the tolerance. This leads to a situation where tolerance values must be negligible or numerical accuracy must be sacrificed during computation. When inspecting the images of the experimental waves, notice that the wave never returns completely to a fully lowered position. This implies that the wave is slightly shorter than it actually is, making is wider. When generating a quality approximation, the tolerance may not be able to adequately account for this disparity. It follows that a higher tolerance might widen the wave, giving behavior similar to what is being observed.

This error might be due to the method of analysis. The input is being manually indexed from a data array and fit by hand to align with the numerically predicted solution. It is possible that differences in scaling are leading to an artificial widening of the wave. This problem is exacerbated by the fact that the experiments have differing array sizes, making qualitative analysis more difficult.

While the amplitude of the numeric approximation is scaled to match the experiments, the window size is varied from case to case. Which could easily lead to an artificial widening of a curve. Careful calculation of the individual volumes of each wave would be required to adequately address the quality of the approximation.

Overall, our two main goals were to compare speed-amplitude relationship between theoretical derivation and experimental data as well as comparing soliton shape between a numerical solution and the observed soliton shape of the glycerin in the experiment, which were both achieved successfully. Overall, **we successfully found a connection between a mathematical model for soliton waves and real-world results**.

# References


[1] N. K. Lowman and M.A. Hoefer. Dispersive Hydrodynamics in Viscous Fluid Conduits. *American Physical Society*, 88, August 2013.

[2] Peter Olson and Ulrich Christensen. Solitary Wave Propagation in a Fluid Conduit within a Viscous Matrix. *Journal of Geophysical Research*, 91, May 1986.

# Appendix

## Links

More code can be found at links below:
https://github.com/gtwallon.
https://colab.research.google.com/drive/1HER0yW-67-3cEl3zy81YGGxHMGX3VMCm?usp=sharing

## Code

```
# Dependencies
import math
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as sp

# c(a)
def c_(a):  # default value of 'a' set to 0.5
  top = a**2 * (math.log(a**2) - 1) + 1
  bot = (a - 1)**2
  return top / bot

# g(f,c)
def g(f, c):
  top = (c - 1) + (f**2) * (c+1) - 2 * c * f - (f**2) * math.log(f**2)
  inner = top / c
  return inner ** 0.5

# Expressing our code as a function

# a : max amplitude of soliton
# h : step size of Euler approximation
# d : tolerance of initial height of soliton
def solNumerics(a=4.2495, h=0.05, d=0.003, title=fig):

  # make an array filled with 0's of correct size
  f0 = np.zeros(int((1/h)*a - (1/h)))

  # fill the f0 array with correct values
  for i in range (1, len(f0) + 1):
    f0[i-1] = (1.0 + d) + (i-1) * h

  # we make an array of the same size for C (zeta)
  C = np.zeros(len(f0))

```

```
  # the speed is a a function of the current height of the wave
  c = c_(a)

  # we perform Euler's method to fill C with correct values
  C[0] = -15.0 # choose a large negative value to generate full curve
  for j in range (1, len(C)): # start at 1 because we know 0
    C[j] = C[j-1] + (h / ( g(f0[j-1], c)))

  # Shifting our plots
  C1 = C - C[len(C) - 1]
  C2 = -1 * C1

  # Plotting
  fig, ax = plt.subplots()
  ax.plot(C1, f0, '-b', linewidth=1, label='Euler Method')
  ax.plot(C2, f0, '--b', linewidth=1, label='Reflection')
  plt.title(Numerical Solution:  a =  +
            str(a) + , h =  + str(h) + , d =  +str(d))
  plt.xlabel(C (zeta))
  plt.ylabel(A (Area))
  plt.legend()
  plt.savefig(str(title) + '.png')
  plt.show()

# amat is a dictionary containing experimental data
def numAndExp(amat, a=4.2495, h=0.01, d=0.003, row = 227,
              start = 3360, end = 3600, u0 = 0.1, num=7, title='fig'):

  # make an array filled with 0's of correct size
  f0 = np.zeros(int((1/h)*a - (1/h)))

  # fill the f0 array with correct values
  for i in range (1, len(f0) + 1):
    f0[i-1] = (1.0 + d) + (i-1) * h

  # we make an array of the same size for C (zeta)
  C = np.zeros(len(f0))

  # the speed is a a function of the current height of the wave
  c = c_(a)

  # we perform Euler's method to fill C with correct values
  C[0] = -15.0 # choose a large negative value to generate full curve
  for j in range (1, len(C)): # start at 1 because we know 0
    C[j] = C[j-1] + (h / ( g(f0[j-1], c)))

```

```
  # Shifting and scaling our plots
  C1 = (C - C[len(C) - 1]) * u0
  C2 = -1 * C1

  # generating the vector for our second plot
  y = amat[row, start:end]
  x = np.linspace(-15, 15, len(y)) * u0

  # Plotting
  fig, ax = plt.subplots()
  ax.plot(C1, f0, '-b', linewidth=1, label='Euler Method')
  ax.plot(C2, f0, '--b', linewidth=1, label='Reflection')
  ax. plot(x, y, '-r', linewidth=1, label='Experimental')
  ax.set(xlim=(-0.35, 0.35), ylim=(0, a + 0.5))
  plt.title(Numerical Solution Vs Experimental Data:  Experiment + str(num))
  plt.xlabel(x (cm))
  plt.ylabel(A (Area))
  plt.legend()
  plt.savefig(str(title) + '.png')
  plt.show()
```

### Speed-amplitude analysis code

**Function:**

```
def compute_soliton_speed(dictionary):
mat = dictionary['Amat']
times = mat.shape[0]
distances = mat.shape[1]
a0 = dictionary['A0'][0,0]
a0_error = dictionary['A0_error'][0,0]
solitonAmpRange = ((a0 - a0_error), (a0 + a0_error))

# first get possible soliton indexes
fullWaveIdxs = []
for i in range(times):
    for j in range(distances):
        if (mat[i,j]>solitonAmpRange[0]) & (mat[i,j]<solitonAmpRange[1]):
            fullWaveIdxs.append((i,j))

# next, get the index with the smallest time coordinate from every column where
    there is a soliton
solitonIdxs = []
columns = [idx[1] for idx in fullWaveIdxs]
for d in set(columns):
    column_idxs = [idx for idx in fullWaveIdxs if idx[1] == d]
```

```
    solitonIdxs.append(min(column_idxs))

# change order of the coordinates for each soliton index to switch from matrix
    indexing to cartesian coordinates
solitonIdxs = [(idx[1],idx[0]) for idx in solitonIdxs]

# separate the time values and the distance values for use of polyfit() function
xvals = [idx[0] for idx in solitonIdxs]
yvals = [idx[1] for idx in solitonIdxs]
m, b = np.polyfit(xvals, yvals, deg = 1)

# calculating real_speed by adjusting scales of the matrix indices
index_speed = 1/m    # zidx / tidx
cm_per_zidx = dictionary['z_vec'][-1][-1] / dictionary['z_vec'].size
real_distance = index_speed * cm_per_zidx  # cm / tidx
sec_per_tidx = dictionary['t_vec'][-1][-1] / dictionary['t_vec'].size

if dictionary['t_vec'][dictionary['t_vec'] < 0].size > 0:  # in case the time vector
     has negative time values
    average = 1.2371625245444737
    sec_per_tidx = average

real_speed = real_distance / sec_per_tidx  # cm / sec

# return all useful information for tracking speed and for plotting
return {'real_speed':real_speed, 'index_speed':index_speed,
        'solitonIdxs':solitonIdxs, 'xvals':xvals, 'yvals':yvals, 'b':b}
```

**Finding all the soliton speeds:**

```
speeds_list = []
a0_list = []
u0_list = []

# data sets [7-9]
for num in range(7,10):
    data = 'expData0' + str(num) + '.mat'
    dictionary = sp.loadmat(data)
    a0 = dictionary['A0'][0,0]
    a0_list.append((num, a0))
    u0_list.append(dictionary['U0'][0,0])

    new_dict = compute_soliton_speed(dictionary)

    # append speed to list
    speeds_list.append( (a0, new_dict['real_speed'], num) )

```

```
    #make two plots side by side
    m = (1 / new_dict['index_speed'])
    y_hat_vals = [m*x + new_dict['b'] for x in new_dict['xvals']]

    plt.figure(figsize = (16, 4))

    plt.subplot(1, 2, 1)
    plt.contourf(dictionary['Amat'])
    plt.title(data)

    plt.subplot(1, 2, 2)
    plt.scatter(*zip(*new_dict['solitonIdxs']))
    plt.plot(new_dict['xvals'], y_hat_vals, c = 'orange', linewidth = 2.5)
    plt.xlim(0,dictionary['z_vec'].size)
    plt.ylim(0,dictionary['t_vec'].size)

    plt.show()

# data sets [10—13]
for num in range(10,14):
    data = 'expData' + str(num) + '.mat'
    dictionary = sp.loadmat(data)
    a0 = dictionary['A0'][0,0]
    a0_list.append((num, a0))
    u0_list.append(dictionary['U0'][0,0])

    new_dict = compute_soliton_speed(dictionary)

    # append speed to list
    speeds_list.append( (a0, new_dict['real_speed'], num) )

    #make two plots side by side
    m = (1 / new_dict['index_speed'])
    y_hat_vals = [m*x + new_dict['b'] for x in new_dict['xvals']]

    plt.figure(figsize = (16, 4))

    plt.subplot(1, 2, 1)
    plt.contourf(dictionary['Amat'])

    plt.title(data)

    plt.subplot(1, 2, 2)
    plt.scatter(*zip(*new_dict['solitonIdxs']))
    plt.plot(new_dict['xvals'], y_hat_vals, c = 'orange', linewidth = 2.5)
    plt.xlim(0,dictionary['z_vec'].size)
```

```
    plt.ylim(0,dictionary['t_vec'].size)

    plt.show()

# data sets [17-48]
for num in range(17,49):
    data = 'expData' + str(num) + '.mat'
    dictionary = sp.loadmat(data)
    a0 = dictionary['A0'][0,0]
    a0_list.append((num, a0))
    u0_list.append(dictionary['U0'][0,0])

    new_dict = compute_soliton_speed(dictionary)

    # append speed to list
    speeds_list.append( (a0, new_dict['real_speed'], num) )

    #make two plots side by side
    m = (1 / new_dict['index_speed'])
    y_hat_vals = [m*x + new_dict['b'] for x in new_dict['xvals']]

    plt.figure(figsize = (16, 4))

    plt.subplot(1, 2, 1)
    plt.contourf(dictionary['Amat'])

    plt.title(data)

    plt.subplot(1, 2, 2)
    plt.scatter(*zip(*new_dict['solitonIdxs']))
    plt.plot(new_dict['xvals'], y_hat_vals, c = 'orange', linewidth = 2.5)
    plt.xlim(0,dictionary['z_vec'].size)
    plt.ylim(0,dictionary['t_vec'].size)

    plt.show()
```

### Calculating average time conversion

```
# sec_per_tidx = dictionary['t_vec'][-1][-1] / dictionary['t_vec'].size
sec_per_tidx_list = []
for num in range(7,10):
    data = 'expData0' + str(num) + '.mat'
    dictionary = sp.loadmat(data)
    if dictionary['t_vec'][dictionary['t_vec'] < 0].size == 0:
        sec_per_tidx_list.append(dictionary['t_vec'][-1][-1] / dictionary['t_vec'].
            size)

```

```
for num in range(10,14):
    data = 'expData' + str(num) + '.mat'
    dictionary = sp.loadmat(data)
    if dictionary['t_vec'][dictionary['t_vec'] < 0].size == 0:
        sec_per_tidx_list.append(dictionary['t_vec'][-1][-1] / dictionary['t_vec'].
            size)

for num in range(17,49):
    data = 'expData' + str(num) + '.mat'
    dictionary = sp.loadmat(data)
    if dictionary['t_vec'][dictionary['t_vec'] < 0].size == 0:
        sec_per_tidx_list.append(dictionary['t_vec'][-1][-1] / dictionary['t_vec'].
            size)

average = sum(sec_per_tidx_list) / len(sec_per_tidx_list)
average
```

### Calculating theoreticals

```
# finding the dimensionalized theoretical speeds for comparison to the
    experimentally found speeds
theo_speeds_list = []
for a0 in a0_list:
    theo_speed = (a0[1]**2 * np.log(a0[1]**2) - a0[1]**2 + 1) / ((a0[1]-1)**2)
    theo_speeds_list.append(theo_speed)

theo_speeds_arr = np.array(theo_speeds_list)
corresponding_u0_arr = np.array(u0_list)
scaled_theos = theo_speeds_arr * corresponding_u0_arr

# sorting our lists by increaseing values of A0
speeds_list.sort()
scaled_theos_list.sort()

# Removing outlier data set
speeds_list2 = [(speeds_list[i]) for i in range(37)]
speeds_list2.append(speeds_list[38])

#PLOTTING
plt.figure(figsize = (14,6))

# experimental speeds
plt.subplot(1,2,1)
x,y,z = zip(*speeds_list2)
plt.scatter(x, y)

z = list(z)
```

```
z = [str(num) for num in z]
for i, dataNum in enumerate(z):
    plt.text(x[i], y[i], dataNum)

plt.ylabel('Speed (cm/s)')
plt.ylim(.3,1)
plt.xlabel('Amplitude (dimensionless)')
plt.title('Experimental Speed vs. Amplitude')

# theoretical speeds
plt.subplot(1,2,2)
theo_a0, theo_speed, dataset = zip(*scaled_theos_list)
plt.plot(theo_a0, theo_speed, 'o', c = 'red')
plt.ylim(.3,1)
plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude (dimensionless)')
plt.title('Theoretical Speed vs. Amplitude')

plt.show()
```

### Plots of experiments grouped by U0 scales

```
# Make a speeds_list for the highest, middle, and second lowest values of U0

# Creating new lists
speeds_list_top = [speeds_list2[i] for i in range(len(speeds_list2)) if speeds_list2
    [i][2] in range(27,37)]
speeds_list_middle = [speeds_list2[i] for i in range(len(speeds_list2)) if
    speeds_list2[i][2] in range(37,49)]
speeds_list_bottom = [speeds_list2[i] for i in range(len(speeds_list2)) if
    speeds_list2[i][2] in range(17,27)]

#PLOTTING
plt.figure(figsize = (16,14))

# experimental speeds, top
plt.subplot(2,2,1)
x,y,z = zip(*speeds_list_top)
plt.scatter(x, y)
plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude')
plt.title('Largest U0')

# experimental speeds, middle
plt.subplot(2,2,2)
x,y,z = zip(*speeds_list_middle)
plt.scatter(x, y, c = 'red')
```

```
plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude')
plt.title('Middle U0')

# experimental speeds, bottom
plt.subplot(2,2,3)
x,y,z = zip(*speeds_list_bottom)
plt.scatter(x, y, c = 'orange')
plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude')
plt.title('Second Smallest U0')

# experimental speeds, all together
plt.subplot(2,2,4)
x,y,z = zip(*speeds_list_top)
plt.scatter(x, y)
x,y,z = zip(*speeds_list_middle)
plt.scatter(x, y, c = 'red')
x,y,z = zip(*speeds_list_bottom)
plt.scatter(x, y, c = 'orange')
plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude')
plt.title('All Together')

plt.show()
```

### Error Propagation

```
# Take the derivative of c(a)
from sympy import *
a = Symbol('a')
c = (a**2 * sym.log(a**2) - a**2 + 1) / ((a-1)**2)
derivative_c = c.diff(a)
derivative_c

# define the derivative c(a)
def c_prime(a0):
    top = 2*(-2*np.log(a0) + a0**2 -1)
    bottom = (a0-1)**3
    return top/bottom
```

### Plotting compatible data with error bars

```
# Create lists with values corresponding to only the most compatible experimental
    data sets
speeds_list_best = [speeds_list2[i] for i in range(len(speeds_list2)) if
    speeds_list2[i][2] in (43, 36, 28, 18, 10,  7, 38, 30, 29, 27, 22, 31, 11,  8)]
scaled_theos_list_best = [scaled_theos_list[i] for i in range(len(scaled_theos_list)
```

```
    ) if scaled_theos_list[i][2] in (43, 36, 28, 18, 10,  7, 38, 30, 29, 27, 22, 31,
    11,  8)]

# Corresponding errors
best_theo_error_list = []
for elm in [43, 36, 28, 18, 10,  7, 38, 30, 29, 27, 22, 31, 11,  8]:
    if elm < 10:
        num = str(0) + str(elm)
    else:
        num = str(elm)

    data = 'expData' + num + '.mat'
    dictionary = sp.loadmat(data)
    a0 = dictionary['A0'][0,0]
    a0_err = dictionary['A0_error'][0,0]
    u0 = dictionary['U0'][0,0]

    error = np.absolute(c_prime(a0)) * a0_err * u0

    best_theo_error_list.append( (a0, error, elm) )

#sorting new list
best_theo_error_list.sort()


# PLOTTING
plt.figure(figsize = (8,6))

# experimental speeds
x,y,data = zip(*speeds_list_best)
plt.scatter(x, y, c = 'red', label = 'Experimental')

# theoretical speeds
theo_a0, theo_speed, dataset = zip(*scaled_theos_list_best)
errors = list(zip(*best_theo_error_list))[1]

plt.errorbar(theo_a0, theo_speed, yerr = errors, fmt='o', c = 'k', ecolor = 'k',
    alpha = .75, label = 'Theoretical')

plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude (dimensionless)')
plt.title('Most Compatible Data')
plt.legend()

plt.show()
```

### Plotting middle U0 data with error bars

```
# Define new lists
speeds_list_middle = [speeds_list2[i] for i in range(len(speeds_list2)) if
    speeds_list2[i][2] in range(37,49)]
scaled_theos_list_middle = [scaled_theos_list[i] for i in range(len(
    scaled_theos_list)) if scaled_theos_list[i][2] in range(37,49)]

# make a theoretical error bar list for middle U0 values
middle_theo_error_list = []
for elm in range(37,49):    # experiment numbers of trials with middle U0 values
    if elm < 10:
        num = str(0) + str(elm)
    else:
        num = str(elm)

    data = 'expData' + num + '.mat'
    dictionary = sp.loadmat(data)
    a0 = dictionary['A0'][0,0]
    a0_err = dictionary['A0_error'][0,0]
    u0 = dictionary['U0'][0,0]

    error = np.absolute(c_prime(a0)) * a0_err * u0

    middle_theo_error_list.append( (a0, error, elm) )

#PLOTTING
# experimental
plt.figure(figsize = (8,6))
x,y,z = zip(*speeds_list_middle)
plt.scatter(x, y, c = 'red', label = 'Experimental')
plt.title('Middle U0 Data with Theoreticals')

# theoretical
theo_a0, theo_speed, dataset = zip(*scaled_theos_list_middle)
errors = list(zip(*middle_theo_error_list))[1]
plt.errorbar(theo_a0, theo_speed, yerr = errors, fmt='o', c = 'k', ecolor = 'k',
    alpha = .75, label = 'Theoretical')


plt.ylabel('Speed (cm/s)')
plt.xlabel('Amplitude (dimensionless)')
plt.legend()
plt.show()
```