Summer 2020 Research

Nadia Aiaseh

August 2020

Writing the Kawahara in moving frame of reference we get

$$u_t = Vu_x + \alpha u_{3x} + \beta u_{5x} + \sigma(u^2)_x. \tag{1}$$

1 Deriving Stability Matrix [3] [1]

$$\begin{aligned} u(x,t) &= u^{(0)}(x) + \delta e^{\lambda t} u^{(1)}(x) + O(\delta^2) \\ u^{(0)}(x) &= \sum_{k=-\infty}^{\infty} \hat{u}_k^{(0)} e^{ikx} \\ u^{(1)}(x) &= e^{i\mu x} \sum_{m=-\infty}^{\infty} \hat{u}_m^{(1)} e^{imx} + c.c. \end{aligned} \tag{2}$$

Substituting the soution and collecting $O(\delta)$ terms yield

$$\lambda u^{(1)}(x) = Vu_x^{(1)}(x) + \alpha u_{3x}^{(1)}(x) + \beta u_{5x}^{(1)}(x) + 2\sigma[(u^{(0)}(x))u^{(1)}(x)]_x. \tag{3}$$

We then substitute for $u^{(1)}(x)$ to get

$$\lambda \sum_{m=-\infty}^{\infty} \hat{u}_m^{(1)} e^{imx} = \sum_{m=-\infty}^{\infty} i[V(m+\mu) - \alpha(m+\mu)^3 + \beta(m+\mu)^5]\hat{u}_m^{(1)} e^{imx} + 2\sigma[u_x^{(0)}(x) \sum_{m=-\infty}^{\infty} \hat{u}_m^{(1)} e^{imx} + u^{(0)}(x) \sum_{m=-\infty}^{\infty} i(m+\mu)\hat{u}_m^{(1)} e^{imx}], \tag{4}$$

where all the $e^{i\mu x}$ terms have been cancelled out.
multiplying by e^{-ikx} and integrating over one period yields

$$\begin{aligned} \lambda \hat{u}_k^{(1)} &= i[V(k+\mu) - \alpha(k+\mu)^3 + \beta(k+\mu)^5] + \hat{u}_k^{(1)} \\ 2\sigma \frac{1}{\pi} \int_0^{2\pi} e^{-ikx}[& \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} (in)\hat{u}_n^{(0)} e^{inx} \hat{u}_m^{(1)} e^{imx} + \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} i(m+\mu)\hat{u}_n^{(0)} e^{inx} \hat{u}_m^{(1)} e^{imx}]dx. \end{aligned} \tag{5}$$

Where we have taken advantage of the fact that due to the symmetry of $e^{inx} = cos(nx) + isin(nx)$, the only way that the integral will not be zero, is for the exponent to be zero resulting in $e^0 = 1$. Applying the same logic to the integral regarding the nonlinear term we get

$$\begin{aligned} \lambda \hat{u}_k^{(1)} &= i[V(k+\mu) - \alpha(k+\mu)^3 + \beta(k+\mu)^5] + \hat{u}_k^{(1)} \\ 2\sigma & \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} (in)\hat{u}_n^{(0)} \hat{u}_m^{(1)} \delta_{0,-k+n+m} \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} i(m+\mu)\hat{u}_n^{(0)} \hat{u}_m^{(1)} \delta_{0,-k+n+m}, \end{aligned} \tag{6}$$

which means that $n = k - m$. Writing one index in terms of the other, allows us to get rid of one sum and write

$$\lambda \hat{u}_k^{(1)} = i[V(k+\mu) - \alpha(k+\mu)^3 + \beta(k+\mu)^5]\hat{u}_k^{(1)} + 2\sigma i(k+\mu)\sum_{m=-\infty}^{\infty} \hat{u}_{k-m}^{(0)}\hat{u}_m^{(1)}. \tag{7}$$

When putting in matrix form $SU = \lambda U$, we let our matrix be addition of two matrices $S = iD + iT$ with D being responsible for the linear and T being responsible for the nonlinear term. We let m be the row index (corresponds to k in the equation above) and n be the number of columns. Then n is also the mode of U and (corresponds to m in the equation above) so $n \in [-M, M]$.

$$d_{m,n} = \begin{cases} (n+\mu)V - (n+\mu)^3\alpha + (n+\mu)^5\beta \ for\ m = n \\ 0\ for\ m \neq n \end{cases} \tag{8}$$

$$t_{m,n} = 2\sigma \begin{cases} 0\ for\ m = n \\ (\mu + m)a_{|n-m|}\ for\ m \neq n. \end{cases} \tag{9}$$

2 Damped Stability Matrix

We can add a damping term of γu_{xx} to equation 7 so that our stability matrix takes into account damping,

$$\lambda \hat{u}_k^{(1)} = i[V(k+\mu) - \alpha(k+\mu)^3 + \beta(k+\mu)^5 + i\gamma(k+\mu)^2]\hat{u}_k^{(1)} + 2\sigma i(k+\mu)\sum_{m=-\infty}^{\infty} \hat{u}_{k-m}^{(0)}\hat{u}_m^{(1)}. \tag{10}$$

3 Results

$\alpha = 1, \beta = \frac{3}{160}, \sigma = 1,\ \mu \in [0, \frac{1}{2}],\ a_1 \in [1e-6, 0.5]$

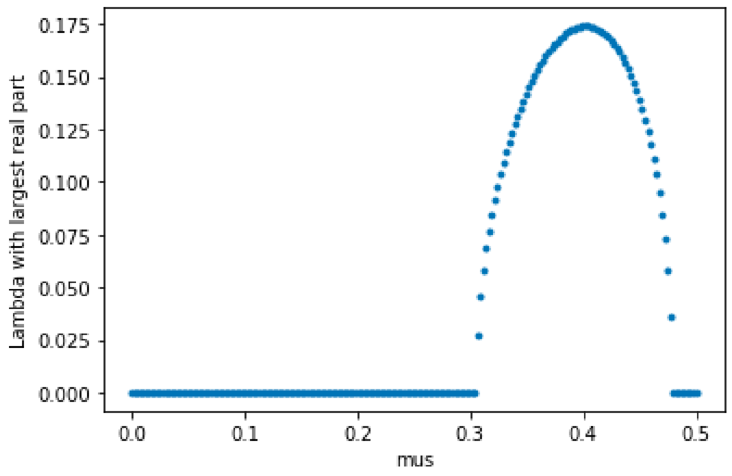
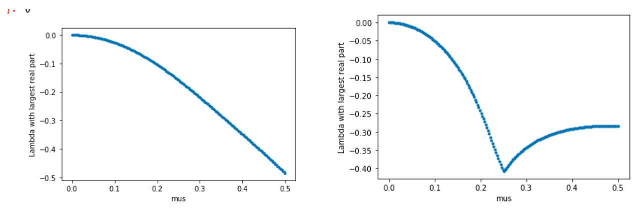


Figure 1: largest $Re(\lambda)$ vs. μ for the undamped case



(a) $\gamma = 1$ (b) $\gamma = 0.5$

Figure 2: We can see that as we decrease the damping coefficient, wer gain back our original undamped shape

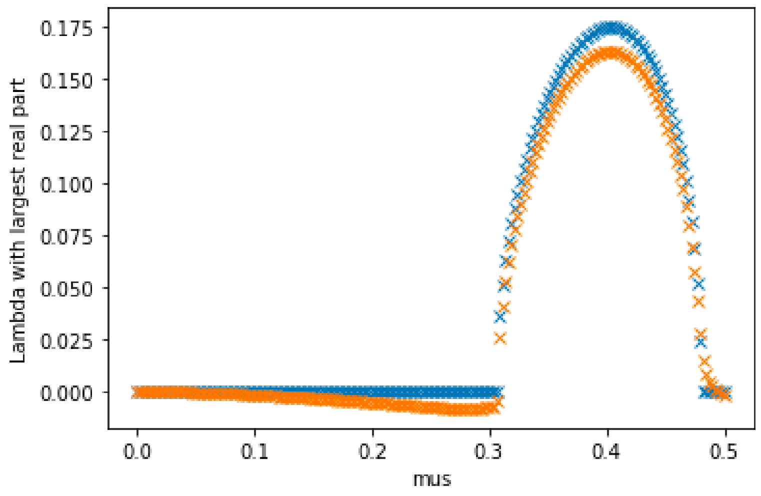


Figure 3: Orange plot is the instability bubble for the damped case with $\gamma = 0.01$ and the blue plot is the original undamped instability bubble

4 Testing Code Using Perturbation Methods

Recall the assumed form of u in 2 as $u(x,t) = u^{(0)}(x) + \delta e^{\lambda t} u^{(1)}(x) + O(\delta^2)$. Where δ is an arbitrarily small perturbation factor. The code for the stability matrix from previous sections, calculate eigenvalues for a given μ. As we saw, we used this to construct instability bubbles, by calculating eigenvalues for a range of μs and picking the ones with the largest real part and plotting them against that μ (i.e. $f(\mu) = Max(Re(\lambda))$. We can then use this λ and μ to construct u as

$$u(x,t) = \sum_{k=-\infty}^{\infty} \hat{u}_k^{(0)} e^{ikx} + \delta e^{\lambda t} e^{i\mu x} \sum_{m=-\infty}^{\infty} \hat{u}_m^{(1)} e^{imx} + c.c. \tag{11}$$

Where $\hat{u}_k^{(0)}$ are the Fourier coefficients output by the code that solves for the time-independent Kawahara. Other than the parameters in the Kawahara equation itself, these coefficients, and hence the time-independent solution, are dependent on number of continuation steps and the range of amplitudes covered. Recall used $\hat{u}_1^{(0)}$ as a free parameter representing the amplitude. The coefficients making up the time-independent solution $u^{(0)}(x)$ come into play once more in construction of the stability matrix according to equation 7.

4.1 Tests

One of the ways we can test our solver for the full Kawahara is to give it the initial condition in form of equation 11. We choose λs both from inside and outside the instability bubble. We expect that over time our solution will be unstable if the λ given has a large real part and we expect a stable solution otherwise.

5 Solvers

5.1 Finite Difference Solver

5.1.1 Function: Kawaharafull

The first input to this function is the unknown vector U which will always be the future time step solution vector. This function is defined for one particular time t^n and u^n is the known solution vector. We used finite difference of 2^{nd} order accuracy in space and 1^{st} with respect to time to model our equations. The reason is because higher order accuracy does not always imply a more accurate modelling of the equation. In fact, the code first used 6^{th} order accuracy with respect to space, which introduced oscillatory noise into the solution, causing the wave profile to look thicker than expected.
The function constructs the Kawahara using finite difference methods for every single space index. First, a vector of zeros called soln is created that is nx (number of spatial points) long. The function iterates through all of the space indices labelled by i and assigns it the kawahara equation. Crank Nicolson scheme is also implemented, making the construction implicit by substituting every u^n in the discretized spatial derivatives by an average with the unknown U (i.e. $u^n \to \frac{u^n+U}{2}$). We will call this averaged value uav at the end, the i^{th} index of the vector returned will be

$$soln_i = \frac{-1(U_i - u_i^n)}{dt} + \frac{\alpha}{dx^3}(-\frac{1}{2}uav_{i-2} + 1uav_{i-1} - 1uav_{i+1} + \frac{1}{2}uav_{i+2}) + 2\frac{\sigma}{dx}(-\frac{1}{2}uav_{i-1} + \frac{1}{2}uav_{i+1})(uav_i) + \frac{\beta}{dx^5}(-\frac{1}{2}uav_{i-3} + 2uav_{i-2} - \frac{5}{2}uav_{i-1} + \frac{5}{2}uav_{i+1} - 2uav_{i+2} + \frac{1}{2}uav_{i+3}) + c(-\frac{1}{2}uav_{i-1} + \frac{1}{2}uav_{i+1}) \tag{12}$$

Periodic boundary conditions were applied. The boundary was handled using padding. This means that our vectors U and u^n were redefined to be a slightly larger vectors that contained U and u^n in the middle but were padded by 3 entries before and after, we show u^n as an example below

$$\vec{u}^n = \left[u^n_{nx-4}, u^n_{nx-3}, u^n_{nx-2}, \vec{u}^n, u^n_1, u^n_2, u^n_3\right]. \tag{13}$$

The reason that we padded with 3 entries before and after is because of how the highest order derivative in our equations requires indices $i-3$ and $i+3$. We then start our iteration at 3 and iterate all the way through to nx+3. The equation vector soln is then shifted 3 units back to account for this range. We do not shift the padded vectors because it is already shfited (i.e. index 3 corresponds to 0).

5.2 integration

The numerical integration was done by looping through nt temporal points. We start from time 0 where u^n is our initial condition u. we use fsolve to find the Us that make our system of nx equations equal to zero. Once we have U, we assign it to the corresponding time index in our solution matrix and overwrite initial condition by naming this solution u. We do this every iteration for all nt temporal points to get a full solution.

5.3 FFT

5.3.1 Function: FFTKawhara

This function is written to be then integrated by odeint. The FFT is from numpy's FFT package. Therefore, either the frequencies or the solution itself needs to be shifted to line up correctly with Fourier frequencies. We shift the N κ frequencies as it is the easier option. The first argument to the function is the initial condition in Fourier space. However, in order for everything to be compatible with odeint, we need to handle imaginary parts separately. Initial condition is thus a concatenation of the real and imaginary parts of the transformed initial condition, making it $2N$ long. Once handed over to the function, the initial condition is once again taken apart, being rewritten in the form of $c = a + ib$ where a is the real part corresponding to the first N κ frequencies and b is the imaginary component, corresponding to the second half of initial condition. The function then creates the time derivative vector *dudt* that is has a length N, one ODE per frequency. The system of equations has the form

$$dudt = \alpha(i\kappa)^3\hat{u} + \beta(i\kappa)^5\hat{u} + c(i\kappa)\hat{u} + 2\sigma Re(FFT(iFFT(\hat{u})iFFT((i\kappa)\hat{u}))). \tag{14}$$

Then $\vec{dudt}$ is once again concatenated by its real and imaginary parts and the function outputs this vector which is 2N long.

5.3.2 Integration

odeint is used to integrate the function based on a given initial conditions using built-in time steps. The output solution matrix is of shape $nt \times 2N$, which is then redefined as $solution = a + ib$ so that we can extract the real part easily and end up with a matrix of size $nt \times N$. We loop through all the nt time steps to inverse transform our solution.

Note that features such as number of frequencies and period can affect the performance of this computation. We used mostly 21 frequencies and the spatial period was 2π.

6 The Nudging Method [2]

Now that we have solvers for the Kawahara, we can explore other areas. One such explorations we did was trying out the nudging method. In this method, we add an extra term to our equation called the nudging term. The nudging term can then make our solution converge to whatever data we give it. Artifical data is computed by just solving the Kawahara alone. The convergence will happen regardless of the initial condition given to the new nudged equation. The nudged equation is

$$v_t = Vv_x + \alpha v_{3x} + \beta v_{5x} + \sigma(v^2)_x + cv_x - \eta(I_h(v) - I_h(u)). \tag{15}$$

Where η is called the nudging coefficient and it is a constant. Its value depends on many things but in general the lower bound is based on factors inherent to the equation while the upper bound is based on spacing h. I represents an interpolation function and h is the spacing prior to interpolation. $I_h(v)$ and $I_h(u)$ are the interpolated solution and data vectors respectively. The steps of implementing the nudging method are as follows:

- Generate the data u by first solving the Kawahara
- Delete parts of the data in order to make it sparse, the sparse data has a spacing of h
- Use linear or constant piecewise interpolation to interpolate the data and gain back the original size, this is $I_h(u)$
- Modify the solver function to include a sparse and then interpolated v, this is $I_h(v)$

- Solve the nudged equation
- Measure convergence

6.1 Finite Difference

Once the data was obtained, parts of the data were deleted and we used the built-in function of Scipy's girddata to interpolate. A second function was defined for the nudged equation, which is identical to the function described in previous section Kawaharafull but the equations now contain the nudging term

$$\begin{split} soln_i = \frac{-1(U_i - u_i^n)}{dt} + \frac{\alpha}{dx^3}(-\frac{1}{2}uav_{i-2} + 1uav_{i-1} - 1uav_{i+1} + \frac{1}{2}uav_{i+2}) + \\ 2\frac{\sigma}{dx}(-\frac{1}{2}uav_{i-1} + \frac{1}{2}uav_{i+1})(uav_i) + \frac{\beta}{dx^5}(-\frac{1}{2}uav_{i-3} + 2uav_{i-2} - \frac{5}{2}uav_{i-1} + \frac{5}{2}uav_{i+1} - 2uav_{i+2} + \frac{1}{2}uav_{i+3}) + \\ c(-\frac{1}{2}uav_{i-1} + \frac{1}{2}uavi + 1) - \eta(I_h(v)_i - I_h(u)_i). \end{split} \tag{16}$$

We then use the same method of integration. We measure convergence by plotting the maximum absolute difference in each time step against time. In event of a convergence, we expect to see an exponential decay to zero, even if there might be a transient phase at the beginning.

6.2 FFT

Implementing the nudging method into FFT is a bit more challenging. To demonstrate this, let's first think about what sparsifying means in the Fourier domain. In Fourier domain, sparsifying and interpolating translates to cutting off the high frequencies and setting them to zero. The challenge is introduced when this needs to be done for every time step. As mentioned before, odeint uses built-in steps with varying sizes which means time cannot be referenced or indexed as we have it defined. However, since odeint is used to solve a system of equations - in this case a system of ODEs, one for every frequency κ - we can modify our function such that it returns a system of ODEs that is twice the length of κs (i.e. 2N) and have the first half to be the data, the transformed Kawahara ODE, and the second half to be the nudged ODE. We have to also modify the initial condition accordingly, with the first half being initial condition to data and the second, initial condition to the nudged equation.

Another important remark is to pay attention when doing any sort of concatenation of real and imaginary parts so that the data and solution to the nudged equation do not get mixed up. For example if we concatenate our derivatives vector at the end just as before, the order will become misplaced and our vector will contain real part of data, the real part of the solution, the imaginary part of data, and the imaginary part of the solution. Thus, we need to specify the concatenation further and have it be in the order of real part of data, imaginary part of data, real part of the solution, imaginary part of the solution.

References

[1] Bernard Deconinck and J Nathan Kutz. Computing spectra of linear operators using the floquet–fourier–hill method. *Journal of Computational Physics*, 219(1):296–321, 2006.

[2] Cecilia Mondaini. a downscaling data assimilation algorithm - from finite to infinite dimensions and back.

[3] Olga Trichtchenko, Bernard Deconinck, and Richard Kollár. Stability of periodic traveling wave solutions to the kawahara equation. *SIAM Journal on Applied Dynamical Systems*, 17(4):2761–2783, 2018.