AM 129, Fall 2025
Final Project Type A – Numerical ODE: The Fermi-Pasta-Ulam-Tsingou problem

Contents

1 Mathematical Background

1.1 Overview of the Problem

Here we are interested in the behavior of a system of point masses connected by springs, and how they move given an initial configuration. Additionally, we will include a *quadratic* nonlinearity into the system. This amounts to solving a coupled system of second order ordinary differential equations

$$\begin{cases} \ddot{x}_i = f_i = K\left(x_{i+1} - 2x_i + x_{i-1}\right)\left(1 + \alpha(x_{i+1} - x_{i-1})\right) \\ x_i(0) = x_i^0 \\ \dot{x}_i(0) = v_i^0 \end{cases}, \tag{1.1}$$

where K is the spring constant, α is the strength of the nonlinearity, and $x_i(t)$ is the deviation of the i$^{\text{th}}$ mass from its equilibrium position. Additionally, x_i^0 is the initial displacement of the i$^{\text{th}}$ mass, and v_i^0 is the intial velocity of the i$^{\text{th}}$ mass. We assume that all masses are normalized to one.

The above works for an infinite system of point masses, but of course we can not compute on an infinite system. Instead, we'll truncate that system to N total masses, so $i = 1, \ldots, N$. We'll assume that the first and last masses are connected by springs to rigid walls. We can encode this by setting

$$\begin{cases} x_0(t) = 0, & t \geq 0 \\ x_{N+1}(t) = 0, & t \geq 0 \end{cases}, \tag{1.2}$$

as so-called dummy masses.

1.2 The linear case

To start we'll consider $\alpha = 0$ so that the forcing term is simply $f_i = K\left(x_{i+1} - 2x_i + x_{i-1}\right)$. To solve equation (2.1) we need some way to advance solution from the initial condition x_i^0 forward in time.

We are going to seek the solution on a uniformly spaced sequence of points through time denoted by t^n satisfying

$$t^n = n\Delta t, \tag{1.3}$$

where Δt is the fixed time step size. Then we'll denote our approximate solution at each time point by x_i^n. Our task is to generate this approximate solution such that $x_i(t^n) \approx x_i^n$.

1.2.1 Stepping through time

We can discretize the second order time derivative by the finite difference formula

$$\ddot{x}_i(t^n) = \frac{x_i^{n+1} - 2x_i^n + x_i^{n-1}}{\Delta t^2} + \mathcal{O}(\Delta t^2), \tag{1.4}$$

which becomes an approximation after we chop off the unknown error term $\mathcal{O}(\Delta t^2)$. Inserting this into equation (2.1) gives us a way to advance the solution through time

$$\frac{x_i^{n+1} - 2x_i^n + x_i^{n-1}}{\Delta t^2} = K\left(x_{i+1}^n - 2x_i^n + x_{i-1}^n\right) \tag{1.5}$$

$$x_i^{n+1} - 2x_i^n + x_i^{n-1} = K\Delta t^2\left(x_{i+1}^n - 2x_i^n + x_{i-1}^n\right) \tag{1.6}$$

$$x_i^{n+1} = 2x_i^n - x_i^{n-1} + K\Delta t^2\left(x_{i+1}^n - 2x_i^n + x_{i-1}^n\right), \tag{1.7}$$

where this final line shows how to generate the future solution x_i^{n+1} from the past solutions x_i^n and x_i^{n-1}. Note also that the forcing term couples the future solution for the i$^{\text{th}}$ mass to the past solutions of its neighboring masses. Equation (1.7) is called the *leapfrog* method and works similiarly for other forcing functions (depending on their properties).

1.2.2 Initializing the problem

The evolution equation (1.7) generates the future solution x_i^{n+1} using *two* previous time steps. Doesn't this mean that to come up with x_i^1 we need the initial condition x_i^0 and the past solution x_i^{-1}? This seems like a problem, we know x_i^0 but have no idea what x_i^{-1} is. Furthermore, we aren't using v_i^0 at all yet.

To get around this we'll initialize the problem in the simplest way. We will set the solution x_i^1 by linear extrapolation from the initial condition

$$x_i^1 = x_i^0 + \Delta t v_i^0, \tag{1.8}$$

then use equation (1.7) to generate the solution starting with x_i^2.

1.2.3 Choosing the time step size

The remaining aspect of the discretization to pin down is the size of the time step Δt. For the solution generated by equation (1.7) to be *stable* one can show that the maximum allowed time step size is bounded by $K^{-1/2}$. This means that if we want to evolve the solution to a final time of T_f we need to use M total steps such that

$$\Delta t = \frac{T_f}{M}, \tag{1.9}$$

and

$$M \geq T_f\sqrt{K}, \tag{1.10}$$

is satisfied. You can therefore set M to be the ceiling of $T_f\sqrt{K}$.

1.3 The nonlinear case

We'll now consider the case where $\alpha \neq 0$. The set up for the problem is exactly the same as above. The solution at the first time step is still set from equation (1.8), and we still set $x_0^n = 0$ and $x_{N+1}^n = 0$ as the position of the left and right dummy masses for all time.

The main difference is that now the solution is advanced in time using

$$x_i^{n+1} = 2x_i^n - x_i^{n-1} + K\Delta t^2 \left(x_{i+1}^n - 2x_i^n + x_{i-1}^n\right)\left(1 + \alpha(x_{i+1}^n - x_{i-1}^n)\right). \tag{1.11}$$

Additionally, the time step restriction given in equation (1.10) is no longer sufficient. To get around this we'll introduce an extra safety factor C such that

$$M = \left\lceil \frac{T_f\sqrt{K}}{C} \right\rceil, \tag{1.12}$$

where $C < 1$ will give smaller time steps than the linear bound. More explicit bounds can be found, but we will just use this as something interesting to probe computationally.

2 Project Description

There are three parts to this project, all of which you will submit through your git repository.

- A Fortran implementation of the leapfrog method (in your `project/code` directory)
- A Python plotting and post-processing tool (in your `project/code` directory)
- A written report (in your `project/report` directory)

2.1 Fortran Implementation

In the Fortran part of the project you will implement a leapfrog scheme to solve a coupled system of second order ordinary differential equations describing a chain of sprung masses

$$\begin{cases} \ddot{x}_i = f_i = K\left(x_{i+1} - 2x_i + x_{i-1}\right)\left(1 + \alpha(x_{i+1} - x_{i-1})\right) \\ x_i(0) = x_i^0 \\ \dot{x}_i(0) = v_i^0 \end{cases}, \tag{2.1}$$

where K is the spring constant, α is the strength of the nonlinearity, and $x_i(t)$ is the deviation of the i$^{\text{th}}$ mass from its equilibrium position. Additionally, x_i^0 is the initial displacement of the i$^{\text{th}}$ mass, and v_i^0 is the initial velocity of the i$^{\text{th}}$ mass. We assume that all masses are normalized to one. See the background reading from section 1 for more information on this system of equations.

2.1.1 How to structure your Fortran code

You should practice **modular programming** by breaking up your code into separate files. You should structure your Fortran code to consist of at least the following files:

- `utility.f90` – A file that contains a few useful parameters that will be used by the other Fortran files.
- `read_initFile.f90` – This file is already written for you. It contains functions that allow you to read in data from an input file.
- `problemSetup.f90` – This reads an initialization file using functions defined inside of `read_initFile.f90`, and sets any needed parameters as well as the initial conditions. A starter version of this file that you may add to as needed is provided for you.

- `leapfrog.f90` – This advances the solution by a single time step based on the solution at the current and previous time steps.
- `output.f90` – This module will hold the subroutine(s) that write the solution to data files as needed.
- `fput.f90` – This is going to be your main driver routine which calls the appropriate routines from the other modules to fill in the solution.
- `simulation.init` – A file that contains any runtime parameters that you want to pass to the code. Each parameter should be on it's own line with the parameter name followed by the value of the parameter. See the sample `simulation.init` file for reference.
- `Makefile` – You should compile your code using a `Makefile`. You should be able to use the `Makefile` from the second assignment with a few modifications. Make sure you use useful *compiler flags*! See the sections on **Fortran Flags** and **Makefiles** in the lecture notes.

2.1.2 Linear oscillators

Let $\alpha = 0$ in equation (2.1). The resulting equation is one for a system of linearly coupled sprung masses:

$$\begin{cases} \ddot{x}_i = K\left(x_{i+1} - 2x_i + x_{i-1}\right) \\ x_i(0) = x_i^0 \\ \dot{x}_i(0) = v_i^0. \end{cases} \tag{2.2}$$

These can be evolved in time by

$$x_i^{n+1} = 2x_i^n - x_i^{n-1} + K\Delta t^2\left(x_{i+1}^n - 2x_i^n + x_{i-1}^n\right), \tag{2.3}$$

which is a leapfrog time stepping method.

Initial condition: The initial conditions you should use are

$$\begin{cases} x_i^0 = 0, & 1 \le i \le N \\ v_i^0 = \sin\left(\frac{i\pi}{N+1}\right), & 1 \le i \le N \end{cases}, \tag{2.4}$$

note that you will need equation (1.8) from the background material to get your solution kick-started.

Physical properties: Use a spring constant of $K = 4\left(N+1\right)^2$. Note that since the total number of masses N is fixed for each run, this spring constant is also fixed.

The final time: Set the final time to be $T_f = 10\pi$. Note that this, and K given previously will determine the allowable time step size from equations (1.9) and (1.10) of the reading material.
Questions:
(a) Start with $N = 1$ in which case the governing equation reduces to a linear, second order, scalar equation. Find the exact solution for this equation and compare it to the solution produced by your code.
(b) Run your code for problems of size $N = \{8,\ 16,\ 32\}$. Save the solution for the the mass $i = N/2$ through all time steps, and save the solution for all masses at times of $t = \frac{T_f}{4}, \frac{T_f}{2}, \frac{3T_f}{4}, T_f$.

2.1.3 Nonlinear oscillators

Now set $\alpha = N/10$. You will now need to use equation (1.11) in the reading material to advance the solution through time, and equation (1.12) to fix the time step size.
Questions:
(c) Use $C = 0.5$ and run with the same resolutions as in question (b), and save the same data out. What happens to the solution as time advances? How is it different from before?
(d) How close to 1 can you make C without ruining the solution?
(e) Investigate what happens for $\alpha = -N/10$. Save data files to support your findings.

2.2 Python Implementation

Your Python implementation needs to perform the following actions:

- Data visualization:
  - Plots for (a): Generate a figure that shows the exact solution and the numerical solution up to the final time T_f
  - Plots for (b,c,e): For each question produce one figure with 6 subplots (3 rows, 2 columns). Dedicate each row of subplots to each resolution. In the left column plot the snapshots of the solution at times $t = \frac{T_f}{4}, \frac{T_f}{2}, \frac{3T_f}{4}, T_f$ as four overlapping lines. Ensure that each time has a unique line style and that the choice of line style is consistent across all figures. In the second column plot the time history for the mass $i = N/2$.

2.3 Report

The final report has a (soft) 7-page limit including figures, and should include the following sections:

- Abstract
- Body: methods, results, findings, comments, etc.
- Conclusion

Submit your report through your git repository in PDF format.

2.4 Extra Credit: Modal decomposition of the solution

The individual masses are all coupled to each other and evolve together. As this evolution proceeds they will take on different configurations. One helpful description of this configuration is to take a Fourier transform over the masses and watch how the amplitude of each mode evolves with time. Consult Homework 3 to see how this Fourier transform could be implemented.

From the rigid boundaries on the problem, we only need to consider a Fourier-Sine decomposition. Add the matrix $\mathbf{T}$ to your Python code defined as

$$T_{i,j} = \frac{2}{N+2} \sin\left(i\pi \frac{j}{N+1} \right) \tag{2.5}$$

where $j = 1, \ldots, N$ and $i = 1, \ldots, M$ where M is the number of modes to consider.

Then for each time step, t^n, the product $\mathbf{T}\mathbf{x}^n$ will give a vector of the first M amplitudes. Here, $\mathbf{x}^n$ is a vector formed from the individual solutions x_i^n. Be careful with the indices here, note that each column corresponds to a distinct mass and each row to a mode amplitude.

Equip your Python plotter to produce the first $M = 4$ amplitudes through time. Produce a figure with three subplots that show, for each resolution in question (c), how these amplitudes evolve through time.