Computational Physics Project 1

OBJECTIVES

1. Obtain data from a physical system that "falls in an interesting way".
2. Design and implement a computer model for the physical system above.
3. Implement and compare algorithms for Newton's equations.
4. Compare model with data; iterate to incorporate refinements.

EXPLORE

1. Compare the output of your algorithm with data generated from your experiment. Does your simulation predict the experiment correctly? What differences or discrepencies remain? Identify a good measure of error for the comparison and justify your choice.
2. Modify your model to incorporate additional physics, or your experiment to gain better data. Does this improve the agreement.
3. For a case where an analytical solution is available, plot the error as a function of time. Does your plot make sense?
4. Check if your model conserves energy and momentum. Should it?
5. Implement at least two algorithms from the list below.

DELIVERABLES

1. A short report (no more than 2 pages) summarizing the findings of your investigation. Centralize data, including graphs, and focus the report on interpreting the data rather than recapitulating physics.
2. Your code, commented and structured to professional standards.

ALGORITHMS

A. Bashforth-Adams

One possible improvement on Euler is to use more *previous* values of the function. The simplest scheme is to perform the discretization using:

$$y_{n+1} = y_n + \frac{3}{2}\delta t\ f(t_n, y_n) - \frac{1}{2}\delta t\ f(t_{n-1}, y_{n-1}).$$

First, put the required pair of coupled ODEs in this form. Write a solver using this scheme, copying the framework provided above. Test your solver against the analytical solution. How does the error solve for this t? Plot the total energy as a function of time and discuss. *[Note that you will have to generate two pairs of initial values to start the scheme; you may use the analytical solution to generate these. What happens if instead you use Euler to start the integrator?]*

B. Runge-Kutta

An improvement to the Euler scheme is to directly estimate *future* values of the variables in calculating the right hand side of the ODE. A strategy to do so is Runge Kutta and is helpfully documented on Wikipedia `http://en.wikipedia.org/wiki/Runge-Kutta_methods`.

Write a solver using this scheme. Test your solver against the analytical solution. What happens to the error with time? What happens if you change δt? Plot the total energy as a function of time and discuss.

C. Adaptive stepsize

One problem with what we've done so far is that we may have chosen a value of the stepsize either too small or too large for the precision required. A useful improvement is to determine the stepsize automatically. Take a look at the Wikipedia article `http://en.wikipedia.org/wiki/Adaptive_stepsize` which tells you a relatively simple method of implementing this. You can implement an adaptive algorithm for any choice of integrator; a good implementation keeps these things separate.

D. Physics-informed Neural Networks**

A very different way of solving the ODE is to represent the function using a Neural Network and train it. A guide and example is available here: `https://i-systems.github.io/tutorial/KSNVE/220525/01_PINN.html`

IDEAS FOR ADVANCED WORK

Packaged integrators—Practical numerical integrators use more sophisticated algorithms and often a combination of algorithms. One example is the built in integrator of *Mathematica*: and your mission is to find out! Set up an *NDSolve* command to integrate the equations and test it against the analytical solutions. What sort of discrepancy is there. You can artificially ask Mathematica to be less precise by specifying PrecisionGoal in NDSolve like so

```
NDSolve[ ... , PrecisionGoal->n]
```

where n is the number of digits of precision required. You can also fix the internal precision using

```
NDSolve[ ... , WorkingPrecision->n]
```

The special value $MachinePrecision uses the same level of precision as your computer. We'll talk about this in a future class.

Another helpful insight can be gained using the StepMonitor and EvaluationMonitor features of NDSolve to see exactly where it chooses to estimate the function. Take a look at the online help to see how to use this and plot where the function is being evaluated.

Mathematica's integrator actually has several methods to choose from (it performs some analysis on the problem to determine this choice) but you can force it to adopt a particular one by setting *Method*. Compare several methods using the online help as an example. Where does each evaluate the function? What's the error like? What happens to the energy in each case?

SciPi **also provides a similar numerical integrator—take a look at** ***odeint*** **in the documentation.** `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html` **You can similarly store the results of your function every time it is called.**