Computational Physics Project 1 Part A: The Simple Pendulum

In order to develop our understanding of numerical integration, we will begin with the simple pendulum discussed in class. This has the advantage that a simple analytical solution is available for comparison, and also that we can go from a linear to nonlinear system as a function of some parameter. The equation to solve is,

$$\ddot{\theta} + k^2 \sin\theta = 0,$$

with initial conditions,

$$\theta(t = 0) = \theta_0,$$

the starting position and,

$$\dot{\theta}(t = 0) = 0,$$

i.e. starting from rest. The parameter $k = \sqrt{g/L}$ fixes the period of oscillation. A *Mathematica* notebook is provided which solves this problem both analytically and numerically using built in functions DSolve and NDSolve. For small oscillations, we can linearize the equation by using $\sin\theta \approx \theta$, obtaining

$$\ddot{\theta} + k^2 \sin\theta = 0.$$

With the boundary conditions above, the analytical solution is simply,

$$\theta(t) = \theta_0 \cos(kt).$$

In this section, each group will use a different method of integration to integrate the simple pendulum ODE numerically. For each, we want to see how varying the timestep δt affects the error, which we will define as the discrepency between the computed solution and the analytical solution. Each group should report this with a graph of error versus stepsize on log-log scales. Additionally, it will be interesting to report for each algorithm how many function evaluations are required to achieve no worse than 10^{-6} relative error at any point in the integration. Several of the more sophisticated here are documented in Chapter 17 of Numerical Recipes.

GROUP 1: EULER SCHEME

As you saw earlier, the Euler method is the scheme,

$$\begin{aligned}\theta_{i+1} &= \theta_i + \delta t\ u_i \\ u_{i+1} &= u_i - \delta t\ k^2 \sin\theta_i.\end{aligned}$$

Using the *Jupyter* notebook provided write a solver using this scheme. Test your solver against the analytical linearized solution for small θ_0 and also the analytical solution for large θ_0. What happens to the error with time? What happens if you change δt? What does varying k do? Plot the total energy as a function of time and discuss.

An interesting trick is to modify the system of equations above in the following way,

$$\begin{aligned}u_{i+1} &= u_i - \delta t\ k^2 \sin\theta_i \\ \theta_{i+1} &= \theta_i + \delta t\ u_{i+1},\end{aligned}$$

i.e. to first calculate an updated generalized velocity u_{i+1} and then to use that instead to calculate the updated angle. This yields the **Semi-Implicit Euler Method**. How does this change the performance of the algorithm?

GROUP 2: 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 couples ODEs in this form. Using the *Jupyter* notebook provided write a solver using this scheme. Test your solver against the analytical linearized solution for small θ_0 and also the analytical solution for large θ_0. What happens to the error with time? What happens if you change δt? What does varying k do? 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 the Euler method to start the integrator?]*

GROUP 3: WHAT DOES MATHEMATICA DO?

A legitimate question is to determine what *Mathematica* does in integrating the equations—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?

GROUP 4: ADAPTIVE STEPSIZE (*)

As you saw earlier, the Euler method is the scheme,

$$\theta_{i+1} = \theta_i + \delta t \ u_i$$
$$u_{i+1} = u_i - \delta t \ k^2 \sin\theta_i.$$

Using the *Mathematica* notebook provided (see the iterative method section for class 2) write a solver using this scheme. Test your solver against the analytical linearized solution for small θ_0 and also the analytical solution for large θ_0. What happens if you change δt?

One problem with the above 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.

As group 1 will find out above, the Euler method can be greatly improved by using the scheme,

$$u_{i+1} = u_i - \delta t \ k^2 \sin\theta_i$$
$$\theta_{i+1} = \theta_i + \delta t \ u_{i+1},$$

i.e. the Semi-Implict Euler method. Try using this instead and determine how the performance is modified.

Adaptive methods can be applied to more sophisticated integrators, such as Runge-Kutta, and a rather thorough explanation is provided in Numerical Recipes Section 17.2.

GROUP 5: 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`. You may also want to look in section 17.1 of Numerical Recipes that has a very helpful explanation. I highly recommend using the fourth order formulae provided.

Using the *Mathematica* notebook provided (see the iterative method section for class 2) write a solver using this scheme. Test your solver against the analytical linearized solution for small θ_0 and also the analytical solution for large θ_0. What happens to the error with time? What happens if you change δt? What does varying k do? Plot the total energy as a function of time and discuss.

GROUP 6: SYMPLECTIC INTEGRATORS (*)

Symplectic integrators are a special class of integrator designed to work well on Hamiltonian systems, i.e. those that conserve energy. A recipe is provided on the Wikipedia entry `https://en.wikipedia.org/wiki/Symplectic_integrator` (look at the bottom of the article). Its relatively easy to implement any of these in the same program, so try at least the third and fourth order examples. The second order Verlet method is the method of choice in Molecular Dynamics simulations in Chemistry. As for all other groups, test your solver against the analytical linearized solution for small θ_0 and also the analytical solution for large θ_0. What happens to the error with time? What happens if you change δt? What does varying k do? Plot the total energy as a function of time and discuss.

GROUP 7: STOERMER'S RULE

Typically, we wish to convert a system of high order ODEs into a set of first order ODEs. The pendulum equation, however, is a special class of ODE where it is actually slightly favorable (at least at second order) to difference the equation directly. It's thoroughly documented in Numerical Recipes section 17.4, although you'll have to discretize the ODE yourselves. Test your solver against the analytical linearized solution for small θ_0 and also the analytical solution for large θ_0. What happens to the error with time? What happens if you change δt? What does varying k do? Plot the total energy as a function of time and discuss.