Experiment 215: Basic Numerical Modelling with Python

Aim

Simulate various physical systems by numerically solving ordinary differential equations in SCIPY. You will have the opportunity to choose one out of two systems to model for the last part.

Introduction

Many physical systems are governed by quantitative relationships that manifest as differential equations. Analytical solutions of these equations are often difficult or impossible to obtain except in the simplest cases. Therefore, in order to predict a system's behaviour, we construct models which simulate the original system.

You might be more familiar with the "physical" models, namely scaled versions of the actual system. For example, model aeroplanes that are placed in wind tunnels to study the behaviour of real aeroplanes in flight. However, when possible, it is more convenient and flexible to build a mathematical model which can be studied on a computer.

This practice has become so important and prevalent in almost every branch of science today that the need for larger and more sophisticated simulations is a major driving force behind the evolution of supercomputers.

A simulation consists of the following steps:

- We extract the differential equations describing the physics of the system.
- We construct a model of the system which obeys the same equations as closely as possible.
- We observe the model running to see what it does.

In this experiment, we consider one class of simulation in which the physical system is described by one or more *ordinary* differential equations with specified initial conditions. We solve these equations by finding functions which satisfy the equations and the initial conditions.

It is convenient to refer to the independent variable as time t, although this can represent some other quantity in the physical system.

Integrator Block Diagrams

A mathematical model of a differential equation can be represented graphically by a block diagram. These diagrams give you a good idea of the model's structure, as they show you how its subsystems are connected, and where each intermediate variable should go in order to obtain results.

The "integrator" block (Figure 1) represents an operation which integrates some input function $x_{IN}(t)$ with respect to time, and gives an output. The initial condition x_{IC} is a constant which specifies the output value at the initial time.

Other useful "elementary blocks" include $\otimes$ and $\oplus$ which multiply and add together two or more inputs respectively.

The following is a systematic method of drawing block diagrams for differential equations.

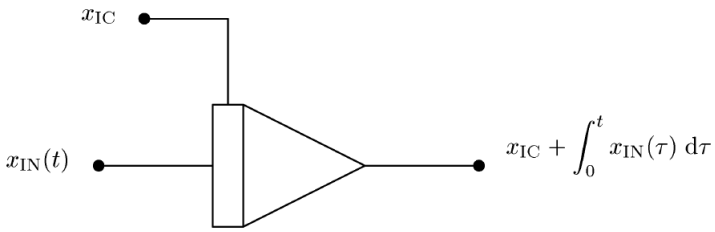
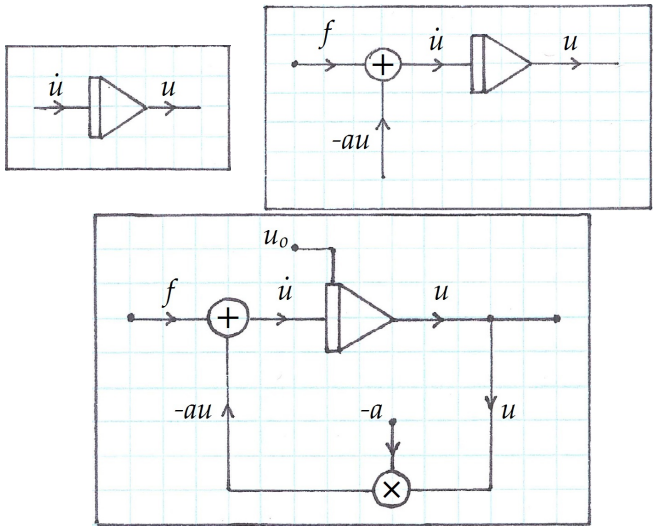


Figure 1: The integrator

1. Rearrange the differential equation so that the highest-order derivative of the variable (u, say) is expressed in terms of everything else.
2. If you have a
   (a) first-order time derivative, you need one integrator block for variable u which appears with its time derivative. The time derivative is the integrator input. The variable itself is the output.
   (b) second-order time derivative, connect two integrators in series from left to right. The input to the leftmost integrator is $\ddot{u}(t)$ so that the input for the next integrator is $\dot{u}(t)$. Then the final output is $u(t)$.
   (c) higher-order time derivative, extend the above procedure.

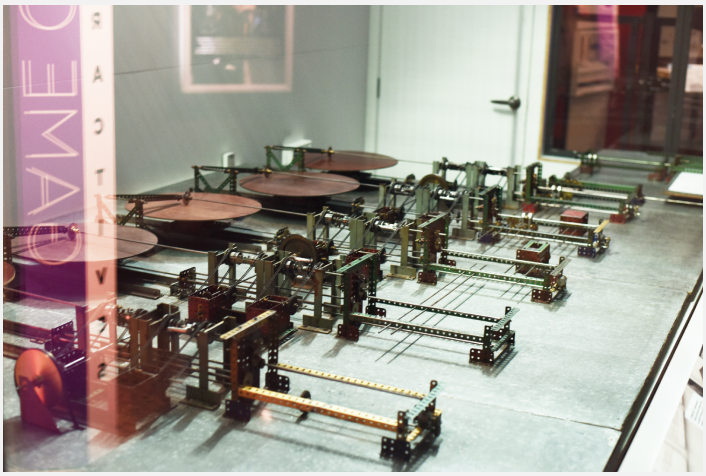
   Variables at the outputs are called "state variables" and are usually relabelled x_1, x_2, etc. Their values at any time represent the system's "state".
3. Include the initial conditions. Connect the integrators with other elementary blocks described above, according to the differential equation. This is usually the trickiest part.

Say, for example, that we want to solve the initial value problem $\dot{u}(t) = -au(t) + f(t)$ with $u(0) = u_0$.



As you might have guessed, block diagrams have deep roots in our technological history. Back in the days where analog electronics were in the main stream, specialised electronic circuits performed differentiation and definite integration of a diverse range of input functions. And even earlier than that, mechanical difference machines were constructed and utilised by scientists and engineers. Figure 2 below shows one of such machines.

FIGURE 2: A MECHANICAL DIFFERENTIAL ANALYSER



This device, on display at MOTAT (the Museum of Transportation and Technology), was built in Cambridge, England in 1935. It is capable of solving a range of problems related to harmonic oscillation. If the museum is open and you ask really nicely, you should technically be able to perform procedures (1) through (10) below with this marvelous construct[a].

[a] Message not endorsed by MOTAT.

Numerical Integration

Numerical analysis is a intricate and sophisticated subject. Thanks to it, there exist many ways to evaluate ordinary differential equations on a digital computer (or, if you are really dedicated, on paper).

Euler Integration

The Euler method is the simplest to describe. Here, we take the value of the function at the initial time, $f(t_0)$, and the expression for the derivative, then evaluate the function value a short time later, Δt. In otherwords, we perform the following computation:

$$f(t_0 + \Delta t) \approx f(t_0) + \Delta t * \dot{f}(t_0) \tag{1}$$

where $\dot{f}$ represents the time derivative.

Then this result is used as the starting value for a second iteration to find the function value after $2\Delta t$. This process can be repeated until we obtain a set of function values at times $\Delta t, 2\Delta t, 3\Delta t$, etc.

A single step of Euler's method is usually fast, and in the limit that $\Delta t \to 0$, our trajectory obviously converges to the actual solution of the original ODE, if it exists. However, it also has obvious limitations. If $\dot{f}$ changes dramatically across a scale comparable to Δt, for example, our numerical solution will quickly get led astray.

The Runge-Kutta (RK) Methods

The RK Methods are a family of iterative methods to numerically solve ordinary differential equations. They were developed around 1900 by the mathematicians Carl Runge and Wilhelm Kutta. Sophisticated implementations of them exist in all major computing languages. The explicit algorithms are beyond the scope of this lab manual, but feel free to read about and discuss them in your report.

In this experiment, we shall use the PYTHON program `scipy.integrate.solve_ivp`, which by default uses the RK45 integrator, to solve the differential equations.

First, import the functions in your PYTHON script by calling

```
from scipy.integrate import solve_ivp
```

You may then invoke the function as `solve_ivp`.

During the course of the experiment, we shall see how it can be used to solve higher-order differential equations as well as coupled systems of equations.

The next section gives a brief description of the input and output arguments of the function for reference, and you can either ask your demonstrator or consult the SciPy Manual[(1)] if you have further questions.

The solve_ivp Function

`solve_ivp` replaces the well-known `odeint` (taught in the Advanced Lab prior to 2020), and provides a greater level of control for our integration actions. It returns a structure in which both time and the evaluated function values are stored. For our purposes, a typical call to it should be in the following format:

```
Solution = solve_ivp(fun = name(t,y), [start time, end time],
 y0 = [IV], method='RK45',
 t_eval=[array of times of interest], max_step = max step size)
```

Here, `name(t,y)` is the expression of the $\dot{f}(t, f)$ that we define. `IV` is a vector that has the same number of elements as the output of `name(t,y)`, and `t_eval` is a list of all the points within the interval $[t_{\text{start}}, t_{\text{end}}]$, for which you wish to know the function value.

The algorithm automatically picks the step size of each integration iteration, so if `t_eval` is not supplied, the output time points may not be uniformly distributed in time. You can also pick the biggest allowed integration step by using the flag `max_step = ...`

We are ready to start. From this point on, assessed sections will be numbered.

Integrating a Known Function

As a first example, consider integrating a function of time whose explicit solution is known. Suppose that the force in newtons on a unit-mass object varies in time as

$$F(t) = 1 + 2\sin(3t)$$

We wish to find the velocity $v(t)$, given that at $t = 0$, the mass is travelling at -1 ms^{-1}.

This problem can be written as an initial value problem for v as

$$\frac{dv}{dt} = 1 + 2\sin(3t) \qquad v(0) = -1$$

The solution may be written as

$$v(t) = -1 + \int_0^t [1 + 2\sin(3t)]\, d\tau$$

This leads to the block diagram shown in Figure 3.

`solve_ivp` accepts anonymous function calls known in computing as (`lambda`), but it is almost always good practice to define a separate function, which could clearly tell us how to find the input to the integrator (i.e., the derivative of v) at time t.

(1) `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html`

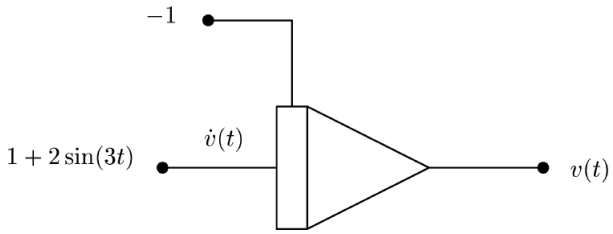


Figure 3: Integrating a known function

(1) Create the following function:

```
def deriv1(t,v): # Notice the ordering!
    return 1.0+2.0*np.sin(3.0*t)
```

Note that this function always has (at least) two input arguments. The first is the independent variable t and the second is the current value of v. In general the derivative $\dot{v}(t)$ depends on both of these variables, but in this case it does not depend on v.

(2) Having written this file, run the integrator in Python as follows:

```
#Always import these two ...
import numpy as np
import matplotlib.pyplot as plt

from scipy.integrate import solve_ivp

t0 = 0
tMax = 10
vout = solve_ivp(fun = deriv1, t_span = [t0,tMax],y0 = [-1.0],max_step = 0.1)

plt.figure()
plt.plot([vout.t],vout.y,'ko')
```

Again, the output of `solve_int` is a structure that contains everything we need. The last line in the snippet above plots the integration result by calling the `t` and `y` data stored in the structure.

It's easy to show that the analytic solution is

$$v = t - \frac{1 + 2\cos(3t)}{3}$$

(3) Plot the velocity as calculated by the numerical and analytic solutions by entering

```
tout = np.arange(t0,tMax,0.1)
vanalyt = tout-(1.0+2.0*np.cos(3.0*tout))/3.0
plot(tout,vanalyt)
```

Print out your graphs and check that they do coincide.

Another Scalar First-order Differential Equation

Now consider a slightly different initial value problem,

$$\frac{dv}{dt} = -v + 1 + 2\sin(3t) \qquad v(0) = -1$$

This could model a problem in which there is an external force $F(t)$, plus a frictional force proportional to the velocity (Figure 4).

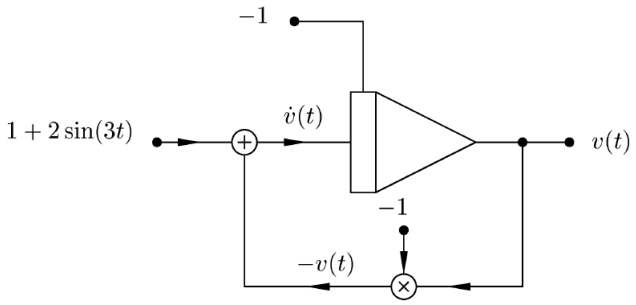


Figure 4: Solving a first-order scalar differential equation

(4) Write the function `deriv2` to calculate the derivative $\dot{v}(t)$ for this problem by modifying `deriv1`.

(5) Find the solution of the initial value problem from $t = 0$ s to $t = 10$ s using a time step no bigger than $h_{\text{Max}} = 0.1$ s. Plot the result of your simulation using crosses together with the analytic solution.

$$v = \frac{5 + \sin(3t) - 3\cos(3t) - 7\exp(-t)}{5}$$

Damped Harmonic Oscillator

The differential equation for damped simple harmonic motion is

$$\frac{d^2y}{dt^2} + B\frac{dy}{dt} + C\,y = 0$$

with initial velocity $\dot{y}(0)$ and initial position $y(0)$. Although this is a second-order differential equation, we can convert it into a form which can be more easily integrated.

In Figure 5, two integrators are connected in series to give the desired output $y(t)$. Note that the input to the second integrator is $\dot{y}(t)$ and the input to the first $\ddot{y}(t)$. By rearranging the differential equation, we have

$$\frac{d^2y}{dt^2} = -B\frac{dy}{dt} - C\,y$$

The input to the first integrator is found by combining $y(t)$ and $\dot{y}(t)$.

We now relabel the outputs of the integrators x_1 and x_2 and regard them as components of a column vector $\mathbf{x}$, the *state vector* of the system. From the block diagram, we see that the scalar second-order differential equation is equivalent to the *vector* first-order equations

$$\frac{dx_1}{dt} = -B\,x_1 - C\,x_2$$
$$\frac{dx_2}{dt} = x_1$$

with initial conditions $x_1(0) = \dot{y}(0)$ and $x_2(0) = y(0)$.

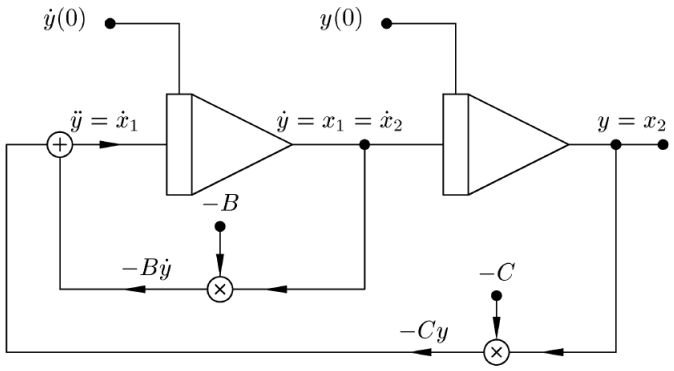


Figure 5: Solving the damped simple harmonic oscillator

(6) Create the function `shm` which calculates the derivatives for the damped simple harmonic oscillator problem from the vector differential equation above.

```
def shm(t,x,B,C):
    return np.array([-B*x[0]-C*x[1], x[0]])
```

Notice that the derivative `dx` is now a *column vector* as is the variable `x`. Another new feature of this example is the additional parameters B and C which are required by this function. Such parameters can either be passed to the derivative function by specifying them at the end of the parameter list of `solve_ivp`, or, more elegantly, by using `lambda`.

(7) Use the following to obtain solutions for the undamped simple harmonic oscillator starting from rest.

```
t0 = 0
tMax = 2
hMax = 0.02

x0 = [0,10]

B = 0
C = 4*np.pi**2


Solution = solve_ivp(fun = lambda t,y: shm(t,y,B,C),
t_span = [t0,tMax], y0 = x0,
max_step = hMax)

tout = Solution.t
xout = Solution.y
xout = np.transpose(xout) # Puts it in a format matplotlib Likes
```

i.e. a solution is required from $t = 0$ s to $t = 2$ s using a time step of $h_{\max} = 0.02$ s, initial conditions are $x_1(0) = \dot{y}(0) = 0$ and $x_2(0) = y(0) = 10$, parameters B and C required by the function `shm` have the values 0 and $4\pi^2$ respectively.

(8) Plot both the velocity $\dot{y}(t)$ and the displacement $y(t)$ using:

```
plt.figure()
plt.plot(tout,xout)
```

where the first column of `xout` contains the velocity $x_1(t)$ and the second column contains the displacement $x_2(t)$. Confirm that the oscillation frequency, the amplitudes and relative phase of the velocity and displacement functions are as expected.

(9) Rerun the simulation with $B = 0.1$ and $C = 1$ using $t = 0$ to 64 s with $h_{\max} = 0.2$ s. Use initial condition $\dot{y}(0) = 0$ and $y(0) = 1$. Plot $y(t)$ as a function of time from your simulation. Check the results of your simulation as follows:

The analytic solution to the differential equation is

$$y(t) = A \exp\left(-\frac{t}{\tau}\right) \cos(\omega' t + \delta)$$

Deduce the values of τ and ω' in terms of B and C.

Determine τ and ω from your simulation and compare them with the values you expect. Note that τ is equal to the time required for the oscillation amplitude to fall by a factor $1/e$ and that ω' can be found from the zero crossings.

(10) Rerun the simulation with $C = 1$ and adjust B to give critical damping. Calculate the analytic solution and display this together with your simulation results for initial conditions of your choosing.

Relaxation Oscillators

Naturally, the simulation technique also works for equations which do not possess analytic solutions. For example, consider the following non-linear differential equation

$$\frac{d^2y}{dt^2} + B\,y\frac{dy}{dt} + C\,y = 0.$$

For $B = 0$ this reduces to the equation for undamped simple harmonic motion. For other B values, the equation characterizes a class of systems called "relaxation oscillators". These have solutions that change slowly (relax) for half a cycle, then change rapidly in the second half cycle.

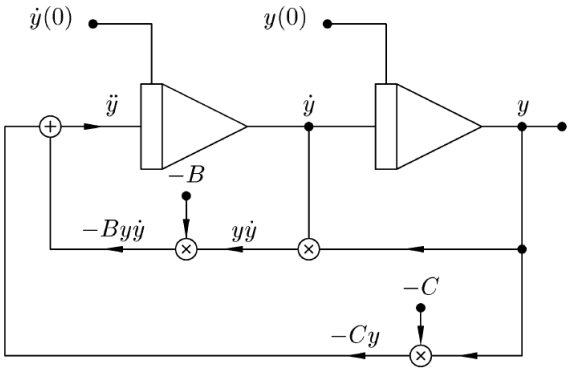


Figure 6: The relaxation oscillator

(11) Confirm that the block diagram for this differential equation is as shown in Figure 6. Write a function file corresponding to this differential equation and run the simulation with $B = 0.5$ and $C = 0.5$ starting from rest and initial displacement $y(0) = 1$. Use $h_{\max} = 0.2$ s and a final time of 20 s. Plot both the displacement and velocity in time synchronization (use the `subplot(211)` command).

(12) Repeat the simulation for $B = 5$. It is appropriate to time-steps $h_{\max} = 0.5$ s and to carry out the simulation until $t = 60$ s.

Compare the solution to that for simple harmonic motion. The displacement should decrease approximately linearly until it reaches some negative value, when it rapidly increases back to the initial value before repeating cyclically. Does the velocity curve agree with this?

The relaxation oscillator equation may be written in the form:

$$\frac{d^2y}{dt^2} + \left(B\,\frac{dy}{dt} + C \right) y = 0$$

If B is small, we may think of the coefficient of y as the square of an "instantaneous angular frequency". If $dy/dt > 0$, this coefficient is larger than C and y changes faster. If $dy/dt < 0$, the coefficient is smaller than C and y changes slower. These correspond to the "fast" and "slow" portions of the observed solutions.

As B becomes large, the solutions become increasingly non-sinusoidal. The (almost) linear negative-going portion of the $B = 5$ solution can be thought of as a section of a long period cosine wave.

The motion of bowed violin strings is of this form although it is not obvious how the differential equation arises from the physics of the violin.

Projectile Motion with Air Resistance

An object moving at high speed experiences air resistance approximately proportional to the square of the speed. Figure 7 shows a projectile travelling at speed v and the forces acting on it.

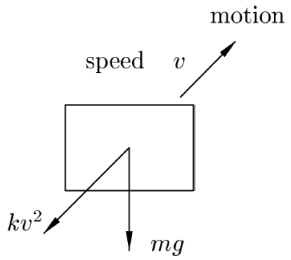


Figure 7: Forces on a projectile

If the horizontal and vertical coordinates of the projectile are x and y, Newton's second law gives:

$$m\frac{d^2x}{dt^2} = -k\,v\frac{dx}{dt}$$
$$m\frac{d^2y}{dt^2} = -m\,g - k\,v\frac{dy}{dt}$$

(13) Draw the block diagrams corresponding to this system. You may represent the evaluation of the speed, $v = \sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$ as a single block. You should find that four integrators are required and that it is necessary to have some connections between the two block diagrams. Use a state vector with four components to write the function file to simulate the projectile's motion. Choose the parameters so that a projectile with mass 0.01 kg has a terminal velocity of 100 ms^{-1} when falling vertically. Assume $g = 9.8$ ms^{-2}.

(14) Run a simulation with initial speed 600 ms^{-1} and initial angle $45°$. These correspond approximately to values for a 0.22 rifle bullet. A suitable value for $h_{\max}$ is 0.5 s. Simulate for 35 s. Test different initial angles to find the maximum range of the bullet and the angle of elevation. Plot the bullet trajectory for this case. Compare your result with the maximum range expected if air friction is neglected. You may find the following function useful for finding the range starting from vectors of x and y positions.

```
def findrange(x,y):
    i2=np.min(find(y<0.0))
    i1=i2-1
    r = (y[i1]*x[i2]-y[i2]*x[i1])/(y[i1]-y[i2])
    return r
```

(15) Change the time step to $h_{\max} = 0.1$ s and use trial-and-error to determine the initial angle required for ranges of 50, 100 and 200 m. An accuracy of ± 0.5 m in the range is acceptable. Calculate how far above the target the rifle should be aimed. For each range, prepare a table showing range, initial angle, maximum height of trajectory, distance above the target for initial aim and the final velocity.

For the final part of your report, please model one of the two systems offered below.

Feel free to do both, but only one may count towards your report mark.

A. Coupled Masses

Consider two masses connected by springs and free to slide on a horizontal frictionless surface (Figure 8). Suppose the masses are both equal to m and that the two outer springs have spring constants k. The central spring is assumed to have spring constant k_c.

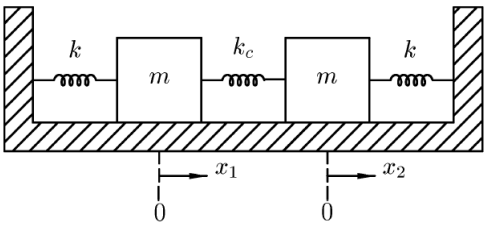


Figure 8: The coupled oscillator

By Hooke's law, if x_1 and x_2 are the displacements of the masses from equilibrium, the equations of motion are:

$$m\frac{d^2x_1}{dt^2} = -k\,x_1 - k_c\,(x_1 - x_2)$$
$$m\frac{d^2x_2}{dt^2} = -k\,x_2 - k_c\,(x_2 - x_1)$$

A1 Sketch the block diagram corresponding to the differential equations given above. Write a function file for simulating the system using the parameters $m = 1$ kg, $k = 1$ Nm^{-1}. Carry out a simulation with $k_c = 0.2$ Nm^{-1} starting with the masses at rest and $x_1\,(0) = 1, x_2\,(0) = 0$. A suitable value of $h_{\max}$ is 0.2 s and you should carry out the simulation for $0 < t < 60$ s.

A2 On the same set of axes, plot graphs of the total kinetic energy of each mass and the total potential energy of the springs assuming that they have their natural lengths when the masses are at equilibrium. Confirm that the total energy of the system remains constant during your simulation by plotting the sum on the same set of axes.

Notice how the energy of the system appears to move between the two masses (the left mass has maximum amplitude when the right mass has zero amplitude and vice versa). The motion of either mass appears to

be the result of "beats" between two solutions at different frequencies. These "normal mode" frequencies are

$$\omega_s = \sqrt{k} \qquad \text{for the "symmetric mode"}$$
$$\omega_a = \omega_s\sqrt{1+2k_c/k} \qquad \text{for the "antisymmetric mode"}$$

Normal modes of a system are those responses which persist unchanged with time. In the above simulation, the system is not in a normal mode since neither mass oscillates sinusoidally.

For a normal mode, both masses execute simple harmonic motion with the same frequency, and amplitudes of both masses are the same. In the symmetric mode, the two oscillations are in phase and the central spring length does not change, so the frequency must be independent of k_c. In the antisymmetric mode, oscillations are exactly in antiphase and the central spring length does change, so its frequency depends on k_c. Either mode can be excited independently by choosing the appropriate initial conditions.

A3 Run the simulation with the initial conditions:

(a) $x_1(0) = 1, x_2(0) = 1$, to excite the symmetric mode
(b) $x_1(0) = 1, x_2(0) = -1$, to excite the antisymmetric mode.

Plot your results and confirm the amplitudes, phases and oscillation frequencies of the masses are as expected.

B. The Three Body Problem

Suppose you have three point masses in the $x-y$ plane, each with mass m_i for $i = 0, 1, 2$ and described by spatial coordinates $\mathbf{x}_i(t) = (x_i(t), y_i(t))$ and velocities $\dot{\mathbf{x}}_i(t) = (\dot{x}_i(t), \dot{y}_i(t))$, bound together by Newtonian gravity. This is a problem that famously lacks general analytic solutions, and the types of motion that the three masses can exhibit are sometimes exquisite.

The equation of motion for mass i is:

$$\ddot{\mathbf{x}}_i = \sum_{j\neq i} \frac{Gm_j}{|\,\mathbf{x}_j - \mathbf{x}_i\,|^3}(\mathbf{x}_j - \mathbf{x}_i)$$

where $G = 6.674\times10^{-11}\text{m}^3\text{kg}^{-1}\text{s}^{-2}$ (we continue to work with SI units in this example).

B1 Translate the given equation of motion into a component-wise format. And, by first drawing the block diagram for one mass, adapt your answer into a function compatible with `solve_ivp`.

You might feel that the internal structure of this system yearns for iteration. This is indeed the case. To help get you started, a code outline is provided below.

```
def 3BodyPlane(t,x,masses):
        #Format of x: [x0,y0,xdot0,ydot0,x1,y1,...]
        v = x*0 # This makes sure that out put has same shape as input.

        for i in range(3):
                iLoc = int(i*4) # each point data starts at 0, 4 and 8

                v[iLoc] =     # [xDot]
                v[iLoc+1] =   # [yDot]

                for j in range(3):
                        iLoc = int(j*4)
                        # [Force Definition Happens Here]

        return v
```

In the snippet above, we have flattened our state vector (i.e. combined everything into one row) because `solve_ivp` prefers this format. `masses` is a numpy array containing the masses (in order) of our objects.

B2 As a sanity check on your code, simulate the earth-sun system over the time period of a year (by setting the third mass to zero and putting it aside). The initial conditions should be set such that the sun starts at rest at (0,0), and the earth is traveling in a circular orbit of radius 1A.U. (1.496×10^{11}m). Pick your $h_{\max}$ carefully, and present the trajectory that you obtain.

B3 Add the moon to your previous simulation, and plot its trajectory as it moves around the sun while orbiting earth. Again, you may assume the moon is in a circular orbit with respect to the earth, and work out the initial condition from there.

B4 In general, the three body problem is chaotic and, as such, is extremely sensitive to initial conditions. Set two stars with one solar mass each to initially orbit each other at a distance in the order of 10^7 km, and launch the third star of 0.5 solar masses towards them at various angles and speeds. Observe the system's evolution in time. In your report, show us a few of your favourite trajectories.

B5 There are special cases of three body motion that exhibit periodic trajectories that are fixed in space. As an example, if you start with three identical masses, and put them on the vertices of an equilateral triangle. Given suitable velocities, they will simply orbit in a circle around their common centre of mass, while staying 120 degrees apart from one another. Do some research, and simulate a few of such periodic orbits.

B6 By modifying your code, if necessary, make a plot of total energy (gravitational plus kinetic) over time. Is it conserved across all these celestial bodies during your integration period? If not, how big is the error?

Remarks

Through this lab experience, you should hopefully have become well-acquainted with `solve_ivp`. It is a powerful tool, and can be quite useful for all your future number-crunching efforts. The kind of numerical modelling you've just played with is just the tip of the iceberg of the myriad models that inform all aspects of our modern life. We have not looked into how people use clever computational tricks, and, increasingly, machine learning, to speed up simulations that can be millions of times more complicated.

This is a different experiment than most things offered at the Advanced Physics Laboratory, and as such, we recommend that you pay special attention to include the following when writing up your report.

- All reference materials you've looked up.
- All required discussions.
- All of your source code.

List of Equipment

1. PC with PYTHON or rather SPYDER installed.

S.M. Tan, May 1993, R. Au-Yeung, May 2013

This version: Y.F. Wang, August 2020