Worksheet week 6 – Simulation of satellite orbits

1 Purpose

1. Learn how to numerically integrate a second order ordinary differential equation.
2. Develop a C++ code for solving a simple 2-body gravitational problem, using a few different numerical integration methods of varying accuracy.

2 Numerical integration of ODEs

2.1 ODEs as coupled first-order equations

In the lecture, we saw that a single nth order ODE can be converted into a system of n coupled first-order ODEs. For example, we can consider the second order ODE

$$A\ddot{y} + B\dot{y} + Cy = 0, \tag{1}$$

where $\dot{}$ denotes time differentiation. We first define a vector $\mathbf{X}$ containing the derivatives $d^{(i)}y/dt^{(i)}$ with $0 \leq i \leq n-1$ of the independent variable:

$$\begin{bmatrix} X_0 \\ X_1 \end{bmatrix} = \begin{bmatrix} y \\ \dot{y} \end{bmatrix}. \tag{2}$$

Differentiating this vector leads to a system of coupled first-order ODEs:

$$\begin{bmatrix} \dot{X}_0 \\ \dot{X}_1 \end{bmatrix} = \begin{bmatrix} \dot{y} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} X_1 \\ -(BX_1 + CX_0)/A \end{bmatrix}, \tag{3}$$

where the last equality comes from rearranging Eq. (1) for $\ddot{y}$. More compactly, the LHS and RHS can be written as a vectorised equation:

$$\dot{\mathbf{X}} = \mathbf{f}(\mathbf{X}, t), \tag{4}$$

where $\mathbf{X} = (X_0, X_1) = (y, \dot{y})$ and $\mathbf{f} = (f_0, f_1)$. In this case, the elements of our function vector $\mathbf{f}$ are $f_0(\mathbf{X}, t) = X_1$, $f_1(\mathbf{X}, t) = -(BX_1 + CX_0)/A$.

Some things to note:

- The vector $\mathbf{X}$ corresponds to a point in phase space, which uniquely describes the instantaneous state of the system (e.g. its position and velocity).
- The left hand side of Eq. (4) corresponds to the phase space 'velocity' $\dot{\mathbf{X}}$, which tells us where the system will go next in the phase space.
- To initiate the solution to the ODE, we need to specify an initial condition $\mathbf{X}(t = 0)$. Note that for an nth order ODE we therefore need to specify n initial conditions (position, velocity, etc).
- Given this initial condition, we want to solve for the subsequent trajectory $\mathbf{X}(t)$ through phase space.
- In the example above (and in the n-body problem you will be solving for this worksheet and the assignment), there is no explicit dependence on t, so $\mathbf{f}(\mathbf{X}, t) = \mathbf{f}(\mathbf{X})$. ODEs with this property are described as *autonomous*. (An explicit dependence on t usually arises from some external driving force on the system.)

Fig. 1 shows an example trajectory through phase space.

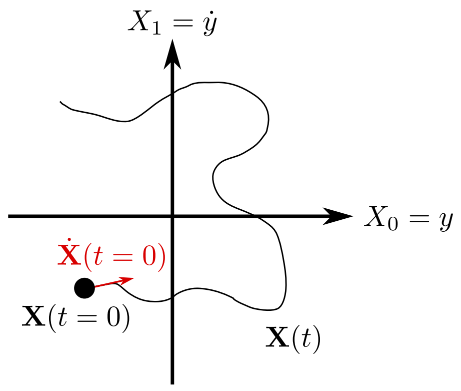

Figure 1: Evolution of the trajectory $\mathbf{X}(t)$ through phase space, given the initial condition $\mathbf{X}(t=0)$.

2.2 Methods for numerically solving ODEs

In this course, we will be solving ODEs numerically using *explicit* schemes of the form:

$$\mathbf{X}_{i+1} = \mathbf{X}_i + \mathbf{k}\Delta t \tag{5}$$

where Δt is our numerical timestep, $\mathbf{X}_i$ is a vector containing the numerical co-ordinates of our state at the ith timestep, and $\mathbf{k}$ is a vector containing a numerical approximation of our phase space 'velocity' $\dot{\mathbf{X}}_i$ over the timestep Δt. For example, if $\mathbf{X}$ contains (position, velocity), then $\mathbf{k} \approx \dot{\mathbf{X}}$ will contain (velocity, acceleration).

Some common methods are described below, listed in order of increasing complexity (and accuracy). The only difference between these methods is how the gradient $\mathbf{k}$ is obtained.

Euler method:
For the Euler method, we use Eq. (5) with the value of the gradient at the start of the step, t_i. In vectorised form, this method can be written:

$$\begin{aligned} t_{i+1} &= t_i + \Delta t && \\ \mathbf{k}_1 &= \mathbf{f}(\mathbf{X}_i, t_i) && \text{gradient at the start of step} \\ \mathbf{X}_{i+1} &= \mathbf{X}_i + \mathbf{k}_1\Delta t && \text{evolution step} \end{aligned}$$

This method is first-order accurate in Δt, which for most practical purposes is too inaccurate to be useful.

Midpoint method:
For this method, we use Eq. (5) with a gradient $\mathbf{k}_2$, which is an estimate of the gradient halfway between t_i and t_{i+1}. To obtain this gradient, we first evolve with $\mathbf{k}_1$ to $t_i + \Delta t/2$. The full algorithm is as follows:

$$\begin{aligned} t_{i+1} &= t_i + \Delta t && \\ \mathbf{k}_1 &= \mathbf{f}(\mathbf{X}_i, t_i) && \text{gradient at the start of step} \\ \mathbf{k}_2 &= \mathbf{f}\left(\mathbf{X}_i + \frac{\mathbf{k}_1\Delta t}{2},\ t + \frac{\Delta t}{2}\right) && \text{gradient at step midpoint} \\ \mathbf{X}_{i+1} &= \mathbf{X}_i + \mathbf{k}_2\Delta t && \text{evolution step} \end{aligned}$$

This method requires two calculations of the gradient per timestep, so it is slower than the Euler method. However, it also provides $O(\Delta t^2)$ accuracy.

Runge–Kutta methods:
For the Runge–Kutta methods, we again use Eq. (5), but this time with a gradient formed from a weighted average $\mathbf{k} = \alpha_1\mathbf{k}_1 + \alpha_2\mathbf{k}_2 + \ldots$. Here, $\{\alpha_1, \alpha_2, \ldots\}$ are weights (satisfying $\sum_i \alpha_i = 1$), and $\{\mathbf{k}_1, \mathbf{k}_2, \ldots\}$ are estimates of the gradient at various points between t_i and t_{i+1}.

The second-order scheme (RK2) is:

$$\begin{aligned}
t_{i+1} &= t_i + \Delta t \\
\mathbf{k}_1 &= \mathbf{f}(\mathbf{X}_i, t_i) && \text{gradient at the start of step} \\
\mathbf{k}_2 &= \mathbf{f}\left(\mathbf{X}_i + \mathbf{k}_1 \Delta t,\ t + \Delta t\right) && \text{gradient at end of step} \\
\mathbf{X}_{i+1} &= \mathbf{X}_i + \frac{1}{2}\left(\mathbf{k}_1 + \mathbf{k}_2\right)\Delta t && \text{evolution step}
\end{aligned}$$

Like the midpoint method, this scheme is $O(\Delta t^2)$ accurate.

The fourth-order method (RK4) can be written:

$$\begin{aligned}
t_{i+1} &= t_i + \Delta t \\
\mathbf{k}_1 &= \mathbf{f}(\mathbf{X}_i, t_i) && \text{gradient at the start of step} \\
\mathbf{k}_2 &= \mathbf{f}\left(\mathbf{X}_i + \frac{\mathbf{k}_1 \Delta t}{2},\ t + \frac{\Delta t}{2}\right) && \text{gradient at step midpoint (first estimate)} \\
\mathbf{k}_3 &= \mathbf{f}\left(\mathbf{X}_i + \frac{\mathbf{k}_2 \Delta t}{2},\ t + \frac{\Delta t}{2}\right) && \text{gradient at step midpoint (second estimate)} \\
\mathbf{k}_4 &= \mathbf{f}\left(\mathbf{X}_i + \mathbf{k}_3 \Delta t,\ t + \Delta t\right) && \text{gradient at end of step} \\
\mathbf{X}_{i+1} &= \mathbf{X}_i + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right)\Delta t && \text{evolution step}
\end{aligned}$$

As the name suggests, this method is $O(\Delta t^4)$ accurate.

For most problems, it turns out that fourth-order solvers provide a good balance between accuracy and computational difficulty. Lower order methods cost less in computational resources, but are not accurate enough for most purposes. Higher order methods, on the other hand, do not provide enough added benefit for most problems to justify their computational cost.

2.3 Converting these methods into code

Example code:
Some example code is provided to demonstrate how to solve an ODE with the Euler method, in vectorised form. You should be able to adapt this code to be used for the n-body problem in the exercise below, as well as in the assignment.

The code is set up to solve the damped harmonic oscillator,

$$m\ddot{x} + b\dot{x} + kx = 0, \tag{6}$$

where m is the object's mass, b is the friction coefficient, and k is the spring constant.

Vector class:
The example code uses the class template `std::vector` to construct the vectors like $\mathbf{X}_i$ and $\mathbf{k}$. Note that many useful mathematical operations with vectors are not built into C++, so the code provided defines new operators to do (e.g.) element-by-element addition, or constant-times-vector multiplication. To do this, it uses "operator overloading".

Operator overloading:
Operator overloading is a feature in C++ that allows you to redefine any function so that it acts on any object you like. The general form of operator overloading is:

```
data_type_out operator(data_type lhs, data_type rhs)
{
code
}
```

where `lhs` and `rhs` are the left-hand-side and right-hand-side of the operation, respectively. Whenever the compiler sees a combination of variables meeting this pattern it will run the code you define instead of whatever the default behaviour is.

3 Exercise – simulating satellite orbits

3.1 System setup: the n-body problem

For both this exercise and the assignment, we will be considering an n-body problem, where a number of masses m_i interact with each other via gravity. The system of equations, in general, has the form:

$$m_i \ddot{\mathbf{r}}_i = G \sum_{j \neq i} \frac{m_i m_j}{|\mathbf{r}_{ij}|^2} \hat{\mathbf{r}}_{ij}, \tag{7}$$

where G is the gravitational constant, $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$, and $\mathbf{r}_i = (x_i, y_i)$ (we will assume that all the dynamics are restricted to the xy-plane). The energy of this system is conserved during the dynamics, and is given by:

$$E = \frac{1}{2} \sum_i m_i |\dot{\mathbf{r}}_i|^2 - G \sum_i \sum_{j<i} \frac{m_i m_j}{|\mathbf{r}_{ij}|}. \tag{8}$$

Linear momenta $p_x = \sum_i m_i \dot{x}_i$ and $p_y = \sum_i m_i \dot{y}_i$ are also conserved, as well as the z-component of the angular momentum.

For this exercise, we will model the trajectory of an object moving around a central mass (e.g. a satellite around the earth). We will generate different types of trajectories by adjusting the initial parameters of the moving object.

To simplify the dynamics, we will assume that the larger object (with mass M) is fixed in place at the origin (and you can therefore set its velocity acceleration to zero for all time). We will use natural units, where $G = 1$, the larger mass $M = 1$, and the mass of the moving object $m = 0.01$.

Task 1: Convert the equations to a computer-friendly form
First, you should 'unpack' Eq. (7) for the case with two masses M and m, and write it in a form that you can enter into a computer. You will want to split up the equations into x- and y-components. Then, convert your equations into a system of first-order ODEs, as shown above. You can remove any equations involving the motion of mass M, since we are considering it to be fixed in place.

Task 2: Transfer equations to code
Have a look at the example code provided for the damped harmonic oscillator. You should be able to build off this code to write a solver for the two-body problem. For now, you can leave the method of integration as the Euler method.

You will also want to add a step that outputs the following to a data file '`orbits.dat`' in six columns, with tab separators:

$$t, x(t), y(t), v_x(t), v_y(t). \tag{9}$$

You can then load this data into external software for plotting.

Task 3: Verify your code
How do you know whether your solver is giving you the right physics? Think about how to test the solver's accuracy. Whenever you write a new program, it is important to cross-check the code in as many ways as possible. If possible, you should check the results against an analytic solution (you can do this here!). Remember that the timestep Δt plays an important role in the accuracy of your simulation.

It is also very useful to confirm that any predicted conservation laws are satisfied. You should write another function in your code to calculate the energy of the system as a function of time. Add another column to your output file for $E(t)$, so you can plot it as well.

Here are some simple examples of other consistency checks you can try, but you are encouraged to find more:

- Set the primary mass $M = 0$. In this case there will be no gravitational attraction between the two masses. The satellite's "orbit" should therefore be a straight line. Try changing the direction and magnitude of your initial velocity.
- Try to reproduce a circular orbit. The satellite should then have a constant speed $v_{\text{circle}} = \sqrt{GM/r}$. Try different time steps until you find that the energy stays constant in time. When you print out the orbit you should find it circular, repeating with a period $T = 2\pi\sqrt{\frac{r^3}{GM}}$.

Task 4: Try different integrators
Now that you are confident that your Euler method solver is working, you can try "upgrading" to one of the more accurate ODE solvers.

Make a new copy of your C++ code, and convert from the Euler integration method to the midpoint or RK2 method. Confirm that the accuracy is improved over the Euler method, for the same initial condition and timestep Δt.

Finally, make another copy of code, this time using the RK4 algorithm for integration. Again, you should compare against the lower order methods for the same initial condition and timestep.

You will need an RK4 solver for the assignment, so it would be a good idea to get your code working here.

Task 5: Create different types of satellite orbit by adjusting the initial velocity

- **Circular orbit (minimum bound orbit):** This type of orbit has a tangential velocity $v_{\text{circle}} = \sqrt{GM/r}$. It is the minimum velocity for which the satellite remains in orbit—any slower and the satellite will fall into the planet. To obtain a circular orbit, set the object's initial position to a radius of 1, and then set an initial velocity of v_{circle} in the tangential direction. Once you have achieved a circular orbit, try reducing the initial velocity slightly and see what happens. What if you set the tangential velocity to zero? What if you add some radial velocity?
- **Parabolic orbit (just escaping):** This type of orbit has a velocity equal to the escape velocity $v_{\text{escape}} = \sqrt{2GM/r}$ (the kinetic energy is equal to the gravitational energy, just enough energy to escape). Set the object's initial position to a radius of 1, and then set an initial velocity of v_{escape} in the tangential direction. Your orbit should become a parabolic orbit, indicating the planet has escape velocity. This means that it is moving directly away from the center mass $(m\mathbf{r}_m + M\mathbf{r}_M)/(mM)$.
- **Elliptical orbit (bound):** This type of orbit has a velocity $v_{\text{circle}} < v < v_{\text{escape}}$, so that the satellite remains bound to the planet in an elliptical orbit. Generate some elliptical orbits, making sure that the satellite remains a closed loop.
- **Hyperbolic orbit (unbound):** Such an orbit has a velocity $v > v_{\text{escape}}$ and as such, will escape the gravitational pull of the object and continue to travel infinitely until it is acted upon by another body with sufficient gravitational force. In the limit $v \gg v_{\text{escape}}$, the object will travel approximately in a straight line, essentially ignoring the planet's gravitational pull.

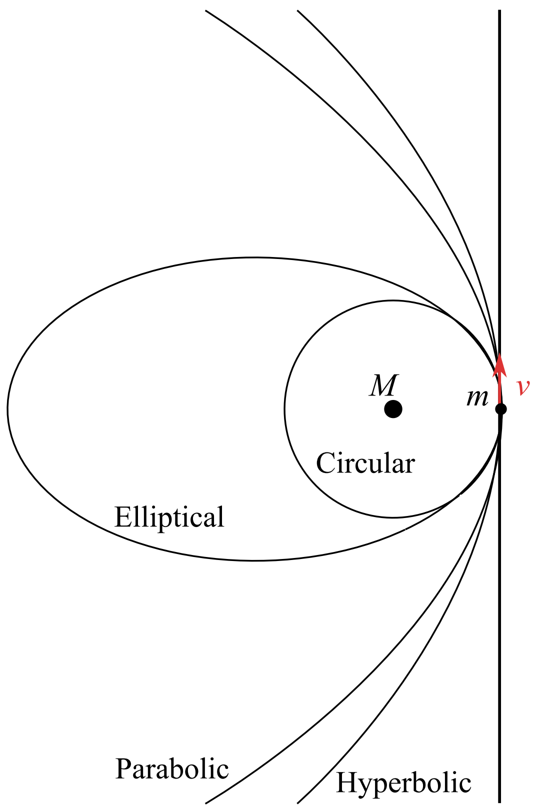

Figure 2: Schematic of the different types of orbits for your satellite.