EG1002 Assignment 2—Solar system simulator

Introduction

In this assignment, you will develop a computer program to solve a physics problem. All fields of engineering sometimes use computer simulations to understand the behavior of a system. Here, you'll implement such a simulation program.

We will focus on the orbital motion of the planets. In the case of two objects (such as a single planet orbiting a star), the equations can be solved by hand. However, no-one has ever solved the case of three or more objects. Physicists call it the "three body problem" and it is famously difficult[1]. Fortunately, computers can be used to solve even very challenging problems.

[1] In fact, it has been proven that no solution exists in terms of standard mathematical functions.

Representing the state of the solar system

Consider initially a top-down view of the solar system with the Sun in the center and the planets moving in roughly circular orbits. We shall consider each celestial body as a point mass, and track its position and velocity. In the case of a 2D system, we might represent position by a two-element vector (x, y). In Matlab, you might write:

```
p1 = [0, 150e9]
```

for "position of object 1" to be at $x = 0$ and $y = 150 \times 10^9$ m.

Similarly, we will represent velocity by another two-element vector (v_x, v_y). It is then possible to manipulate the position and velocity by using Matlab's vectorised operations. The position and velocities of each object represent the current state of the solar system.

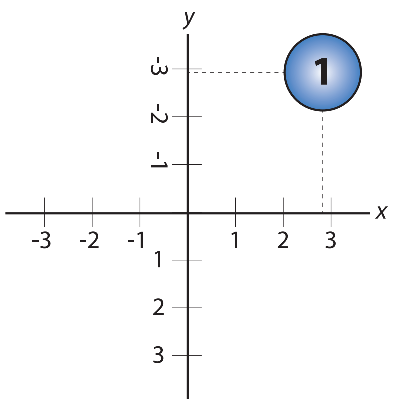

Figure 1: Position coordinates will represent the center of mass of the object.

The program will begin with knowledge of the planets' initial positions and initial velocities. Then, the physics of orbital motion will be applied in order to calculate the position and velocity at later times. You will then plot the planets' positions so that the user may observe the motion of the planets.

The first step is calculating the forces that act on the planets.

Calculating the forces

Let us define the center of mass of the first celestial body as position $\mathbf{p}_1$. (Bold letters are used to indicate vectors.) We shall number our bodies 1, 2, 3, ... so we will have positions $\mathbf{p}_1$, $\mathbf{p}_2$, ... for each celestial body. You might start with just two: the Sun and the Earth, which you might label as 1 and 2 respectively.

The positions vary with time, so we write them as $\mathbf{p}_1(t)$ and $\mathbf{p}_2(t)$ and so on. Any parameter that varies with time will be indicated as an explicit function of t.

Planets and moons are affected only by the force of gravity. Gravity causes massive objects to be attracted towards each other.

Figure 2 shows the case of two bodies. Planet 1 is attracted towards planet 2 due to gravity. The force is given by

$$\mathbf{F}_{12} = G\frac{m_1 m_2}{|\mathbf{r}_{12}|^3}\mathbf{r}_{12}, \tag{1}$$

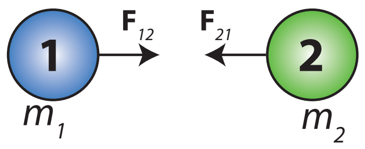

Figure 2: Forces in the two-body problem. Here, $\mathbf{F}_{12}$ is the force on object 1 due to object 2. Force $\mathbf{F}_{21}$ is the force on object 2 due to object 1. Note that $\mathbf{F}_{12}$ and $\mathbf{F}_{21}$ are equal in magnitude but opposite in direction.

where bold variables are vectors, $G = 6.673 \times 10^{-11}$ Nm^{2}kg^{-2} is a constant representing the strength of gravity, m_1 and m_2 are the masses of each object respectively, and

$$\mathbf{r}_{12} = \mathbf{p}_2 - \mathbf{p}_1$$

is the distance vector giving the position of object 2 relative to the position of object 1. You can subtract vectors in Matlab just as you might expect:

```
r12 = p2 - p1;
```

The term on the denominator in equation 1 involves the magnitude of $\mathbf{r}_{12}$. The magnitude is always a scalar (i.e. not a vector any more). It is simply the vector's length:

$$|\mathbf{r}| = \sqrt{x^2 + y^2},$$

where x and y are the components of the vector $\mathbf{r}$. In Matlab, the function `norm` computes the magnitude of a vector.

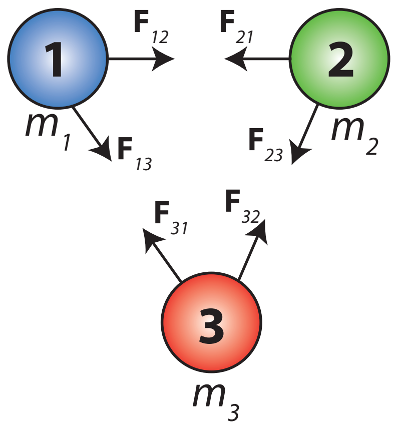

Figure 3: Forces in the three-body problem.

Solving for acceleration

Knowing the forces, we will calculate acceleration by using Newton's Second Law:

$$\mathbf{a}_1 = \frac{1}{m_1}\left(\mathbf{F}_{12} + \mathbf{F}_{13} + \cdots\right), \tag{2}$$

where m_1 is the mass of body 1, $\mathbf{F}_{12}$ is the force on object 1 due to object 2, $\mathbf{F}_{13}$ is the force on object 1 due to object 3, and the summation extends over all the other celestial bodies in the simulation.

Solving for velocity

Suppose that we know the velocity at a time t, and we would like to calculate a new, updated velocity at some later time which we will write as $t + \Delta t$. (The capital Greek letter Delta (Δ) is a commonly used symbol that indicates an interval or a change. Consider Δt as a single variable.)

Since acceleration (which we already calculated) is the rate of change of velocity, we have[2]

$$\mathbf{a}_1(t) \approx \frac{\mathbf{v}_1(t+\Delta t) - \mathbf{v}_1(t)}{\Delta t}.$$

[2] You might recognise this equation from your early studies of calculus. In the limit of $\Delta t \to 0$, the RHS is the derivative of $\mathbf{v}_1$ with respect to time. We will make Δt small but not zero, and therefore the equation is only approximate.

Rearranging:

$$\mathbf{v}_1(t+\Delta t) \approx \mathbf{v}_1(t) + \mathbf{a}_1(t)\Delta t.$$

Solving for position

Similarly to before, the update step for position is:

$$\mathbf{p}_1(t+\Delta t) \approx \mathbf{p}_1(t) + \mathbf{v}_1(t)\Delta t. \tag{3}$$

This equation is approximate because it neglects any change in velocity during the interval Δt. It assumes that $\mathbf{v}_1(t)$ at the exact instant t is representative of the velocity across the entire Δt interval. Therefore if Δt is small, $\mathbf{v}_1(t)$ will only change a little bit, and the approximation will be good enough. You will find that there is a computation speed versus accuracy tradeoff in choosing Δt.

The above set of equations provide a recipe for simulating motion. An initial position and initial velocity are supplied as input to the program. For example, the initial position might be the location of the Earth on a given day and its velocity can be calculated from the orbital length (1 year) and the circumference of its orbit. We choose a suitable time step Δt, and then repeatedly apply the above equations to advance forward in time.

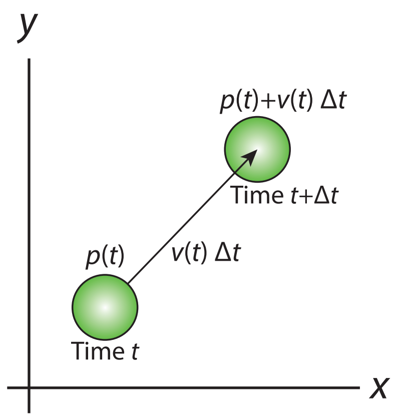

Figure 4: Visual representation of the position update. The new position is the vector sum of the old position plus the motion that occurred during the time step. The motion that occurred during the time step is $\mathbf{v}(t)\Delta t$, indicated by the arrow.

Summary of algorithm

1. Begin with knowledge of the position and velocity of every celestial body in the simulation.
2. For each celestial body, calculate:
   (a) The forces that act on that body.
   (b) The acceleration that results from those forces.

   (c) The new position. Note that the velocity term in Eq. (3) is $\mathbf{v}(t)$ not $\mathbf{v}(t + \Delta t)$, so in other words, use the old velocity when calculating the new position.
   (d) The new velocity.

3. Update your plot to show the latest position of each object. Use `drawnow limitrate;` to trigger Matlab to refresh the screen.

Your task

Your task is to implement a Matlab function that is compliant with the specifications below. Your function must have the signature

```
function [p, v] = solarsystem(p, v, mass, stop_time, hide_animation)
```

The input parameters are as follows:

- `p` is the initial position of each object, as an N-by-2 matrix. The x and y positions of the first object are are `p(1,1)`and `p(1,2)` respectively; similarly the position of the second object is `p(2,1)`and `p(2,2)`. In the case of a 3D simulation, `p` will be an N-by-3 matrix. Positions are measured in meters.
- `v` is the initial velocity of each object, as an N-by-2 matrix. The x and y velocities of the first object are are `v(1,1)`and `v(1,2)` respectively. In the case of a 3D simulation, `v` will be an N-by-3 matrix. Velocities are measured in meters per second.
- `mass` gives the masses of each object as an N-by-1 vector. Masses are measured in kilograms.
- `stop_time` gives the final time (in seconds) at which to stop the simulation. After this many seconds have elapsed, return the calculated position and velocity.
- `hide_animation` is an optional parameter that indicates whether your program's raw computation speed is being assessed. The higher level tests (see below) will sometimes call your program with `hide_animation` set to true, in which case you can skip the animation and try to be as fast as possible. Implementing this behaviour is entirely optional.

An example template is available for download from LearnJCU that implements this signature and further explains the data format for the inputs and outputs.

You must comply with this template because your code will be tested for speed and accuracy by an external test script. You are also supplied with test scripts that you can use to validate your program.

Levels of achievement

The level of achievement controls the maximum mark that your work will be assessed against, as detailed below.

Base level (up to 65% maximum mark)

In this level you will implement:

- Two dimensional simulations with two bodies.
- Accuracy that passes the tests in the `test_base_level.m` script provided on LearnJCU.
- A graphical display that animates the movement of the two objects.

Moderate level (up to 85% maximum mark)

At this level you will implement:

- A variable number of celestial bodies.
- Improved graphical display, such as visualisation of the entire path followed by the planets, varying size of the planets, or choosing nice colours, etc.
- Accuracy that passes the tests in the `test_moderate_level.m` script provided on LearnJCU.
- An efficient implementation that computes the answers quickly. Use the Matlab profiler to find out where your code is slow. Use the "Run and Time" button on the editor toolbar[3].

[3] It's also a good idea to compare your execution time (as reported by `test_moderate_level.m`) with others in your class.

Advanced level (up to 100% maximum mark)

At this level you will implement all of the features of the other levels and one or more of the following:

- 3D geometry so you may simulate objects with orbits out of the plane of the solar system. Your program must detect 3D mode by looking at the dimensions of the input matrices.
- Accuracy that passes the tests in the `test_advanced_level.m` script provided on LearnJCU.
- Your own test harness that assesses the accuracy of your simulation by verifying the conservation of energy. In this way you can independently validate your simulations without needing to trust the correct answers provided by your lecturer[4].

[4] What if they are wrong!

- Calculate the kinetic energy

  $$K = \frac{1}{2}m|\mathbf{v}|^2$$

  for each body in your simulation.
- Calculate the potential energy of the entire configuration. The potential energy of two bodies is

  $$U = -\frac{Gm_1m_2}{|\mathbf{p}_2 - \mathbf{p}_1|},$$

  where the term on the denominator is simply the distance between the two objects. If there are 3+ objects in your simulation, the total potential energy is the sum over every possible interaction.
- The total energy is then the sum of kinetic energy and potential energy. To summarise, where summations are taken over all bodies in the simulation:

  $$E = K + U = \sum_i \frac{1}{2}m_i v_i^2 - \sum_{i \neq j} \frac{Gm_im_j}{|\mathbf{p}_j - \mathbf{p}_i|}.$$

  An interesting property of E is that negative values correspond to bound orbits, whereas positive values indicate that the object has reached the "escape velocity" and will never return.
- The energy at the start of the simulation should be equal to the energy at the end of the simulation. It's recommended that you calculate a percentage energy difference as

  $$\varepsilon = \left|\frac{E_{final} - E_{initial}}{E_{initial}}\right| \times 100\%.$$

- *Optional extension task:* Collisions between objects where objects may "clump" together to form a larger planet. All collisions must conserve momentum, so use this conservation law to calculate the new velocity of the combined clump. If you implement this fully, you should be able to show a dense field of small rocks coalescing into planets.
- Any other advanced features that are beneficial to your program. You might want to create a "readme" text file in the same folder as your source code to highlight these features for the benefit of the assessor.

Submission

Due 5pm Friday 5 May.

Prepare a Zip file containing all the files needed to run your program. This zip file must be uploaded to LearnJCU under "Assessment submission." In the event of any technical problems with

LearnJCU, email your work to `bronson.philippa@jcu.edu.au` before the deadline.

Warning: we will use automatic plagiarism detection software to analyse the originality of your work. It will be compared against sources on the Internet as well as submissions from other students. If we believe that your code is not your own, we will refer you to the Dean of College for an academic misconduct hearing. This is an unpleasant situation and you can avoid it by giving due credit to any work that you include that is not yours.

Assessment criteria

Your submission will be assessed on the following criteria. Your overall assignment score will be scaled by the maximum mark associated with your level of achievement.

Criteria	High achievement	Minimum pass	Unsatisfactory
Functionality and correctness (50%)	The program is highly functional, with respect to the level of achievement. All analysis correctly achieves the set objectives. The program is highly functional and would easily adapt to new requirements. Correct units and axis labels are displayed where needed.	The program generally achieves most of the requirements of the level of achievement. The approach is generally correct, although there may be occasional small issues that appear to be mistakes rather than fundamental misunderstandings.	The program does not work correctly and there are significant or substantial technical flaws in its design. There is evidence of fundamental misunderstandings in the way that the problem was approached.
Elegance, efficiency, and software architecture (40%)	The program is well structured, easy to read and understand, well commented, and uses appropriate variable and function names. The software architecture is modular. The program is highly performant. It would be easy to adapt to new requirements in the future.	The program achieves the objectives of the task but it is not necessarily easy to understand or easy to modify if the requirements were to change in the future. The program could be improved to make it faster.	The program is difficult to understand or the process is unnecessarily complex.
Coding style and presentation (10%)	The program code is professional in appearance. An accepted coding style is used consistently throughout the program.	The coding style is generally acceptable, with occasional lapses.	The code is unacceptably difficult to read or understand. It would be rejected when considered against typical community standards of software engineering.