



# A lab on Kepler's laws that introduces students to coding



Glenda Denicolo<sup>1</sup> and Michael Zingale<sup>2</sup>  
<sup>1</sup>Suffolk County Community College, NY  
<sup>2</sup>Stony Brook University, NY  
denicog@sunsuffolk.edu

Aligned with AAPT recommendations to incorporate computational physics into the curriculum, we prepared a coding exercise about the verification of Kepler's laws and offered it to our second semester calculus-based physics students as an in-class laboratory activity. The exercise is coded in Python using Jupyter notebooks. The notebooks are placed in a github repository, and are deployed to the students via mybinder.org. Binder opens the notebooks in a browser for the students, avoiding the need to install any software. During the two hours lab, students followed the instructions in the notebook, which gradually introduced them to Python commands, and explained the 4th order Runge-Kutta integration method applied to calculate orbits. Students are guided in gradual steps on how to add lines to the code, change initial conditions and run the integration for new orbits to collect data. The information collected is used to write a lab report on the verification of Kepler's three laws.

<https://github.com/zingale/orbits>

The screenshot shows the GitHub interface for the 'orbits' repository. It includes the README file and a Jupyter notebook named 'orbits.ipynb'. A red arrow points from the GitHub logo to a 'binder' button, which is currently loading. A blue arrow points from the 'orbits.ipynb' file towards the Jupyter notebook content.

## Kepler's Second Law

Kepler's second law says that a planet sweeps out equal areas in equal times. Mathematically, this means that

$$\frac{\Delta A}{\Delta t} = \text{constant}$$

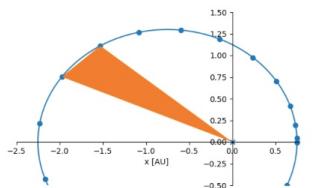
We can test this (approximately) by computing the area of an orbital segment,  $\Delta A$  swept out in a single step and dividing by the corresponding change in time,  $\Delta t$ . To make life easier, we will imagine the area swept out from one step to the next to be a triangle, with one point as the focus (Sun), and the other two are successive points on the ellipse.

Our primary focus is located at the origin, (0,0). In this case, the area of a triangle with vertices (0,0), ( $x_1, y_1$ ), and ( $x_2, y_2$ ) is:

$$\Delta A = \frac{1}{2}x_1y_2 - x_2y_1$$

Note: this approximation means that our  $\Delta A/\Delta t$  should only agree to 2-3 significant digits.

This approximation is illustrated as the orange area in the figure below:



## Question 2

Test out Kepler's second law by computing  $\Delta A/\Delta t$  for several different segments in the orbit. Always use two successive points (e.g.,  $n, n+1$ ) for a single interval.

Collect several values of  $\Delta A/\Delta t$  and organize on a table.

For the numerator, you will construct  $\Delta A$  using the  $x$  and  $y$  coordinates for points in the orbit, accessed as  $o.x[n]$  and  $o.y[n]$ , where  $n$  is an integer in the table of points we displayed above. For the time difference in the denominator, you will use  $o.t[n]$ .

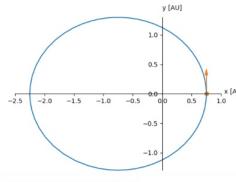
In [17]: # do you work here

## Orbits

We want to consider planetary orbits. To do this, we need to solve Newton's second law together with Newton's law of gravity. If we restrict ourselves to the  $x$ - $y$  plane, then there are 4 quantities we need to solve for:  $x$ ,  $y$ ,  $v_x$ , and  $v_y$ . These evolve according to:

$$\begin{aligned}\frac{dx}{dt} &= v_x \\ \frac{dy}{dt} &= v_y \\ \frac{dv_x}{dt} &= a_x = -\frac{GM_x}{r^3} \\ \frac{dv_y}{dt} &= a_y = -\frac{GM_y}{r^3}\end{aligned}$$

To integrate these forward in time, we need an initial condition for each quantity. We'll setup our system such that the Sun is at the origin (that will be one focus), and the planet begins at perihelion and orbits counterclockwise.



Recall that the distance of perihelion from the focus is:

$$r_p = a(1 - e)$$

where  $a$  is the semi-major axis and  $e$  is the eccentricity. The perihelion velocity is all in the  $y$  direction and is:

$$v_{p,y} = v_p = \sqrt{\frac{GM}{a(1-e)}}$$

We will integrate this system numerically. The basic idea is that we start with our initial conditions and then we advance the position of the planet through a very small amount of time,  $\Delta t$ . This gives an updated position. A very simple way is to just use the initial position to evaluate the right-hand side of our equations. This gives us update:

$$x_t = x_0 + \Delta t v_{x,0}$$

$$v_{x,t} = v_{x,0} - \Delta t \frac{GM_x}{r_t^3}$$

$$v_{y,t} = v_{y,0} - \Delta t \frac{GM_y}{r_t^3}$$

Recently mybinder.org has not been working reliably.

We will explore other notebook server alternatives such as

<https://notebooks.azure.com/>

## Kepler's Third Law

Kepler's third law is the harmonic relation that relates the period and semi-major axis of the planet and the mass of the star:

$$\frac{4\pi^2}{G} a^3 = M_* T^2$$

## Question 3

In this exercise we want you to verify that the ratio  $a^3/T^2$  is constant for all planets/satellites orbiting the same star. Choose different semi-major axis values and eccentricities, run

$$o = o.Orbit(x=xxx, e=yy)$$

where  $xxx$  is your choice of semi-major axis in AU and  $yy$  is your choice of eccentricity.

You will then print the value of the period using:

$$o.period()$$

This will be in seconds.

Build a table of values of  $a$ ,  $T$  and  $a^3/T^2$ .

From this information, what is the mass of the star that the planets are orbiting?

In [3]: o.period()

2s | 3s |

You then keep repeating this until you've evolved for the amount of time you want.

This method is called Euler's Method. It is simple to implement, but it is not very accurate. The formal reason for it not being accurate is that the error is proportional to  $\Delta t$ , so you need to make  $\Delta t$  really small to get a good solution.

We'll instead use a more complex integration method, called 4th order Runge-Kutta. Runge-Kutta takes several Euler-like steps (of  $\Delta t/4$  and  $\Delta t$ ) and combines them together to get a more accurate solution. The "4th order" means that the error is proportional to  $(\Delta t)^4$ , so if you make  $\Delta t$  a little smaller, the error drops a lot.

We have a module, `orbit.integrate` (shortened to `oi` here) that implements this for us. Let's look at how to use it.

In [10]: import orbit.orbit\_integrate as oi

We will use a library called `orbit.orbit_integrate` (abbreviated below as `oi`) to do the orbital integration for us. Using this is pretty simple – we first create an orbit object (we'll call it `o` in all the examples below), and then we interact with that object.

For example, to create an orbit with a semi-major axis,  $a$ , of 1.5 AU, and an eccentricity,  $e$ , of 0.5, we do:

o = oi.Orbit(a=1.5, e=0.5)

Now we set the orbit object that we want to integrate. By default, it will integrate for one period, but you can specify the `timescale` to integrate using the `run_periode` argument, giving a fraction of a period. For example, to integrate for 60% of an orbital period, we do:

o.integrate(0.6, run\_periode=1.0)

After integrating, the `o` object contains all the position and velocity information for the orbit at several different points in time (the time interval between points varies, since our integrator adjusts it as needed to get good accuracy). For example, the following information is available:

- o.x: the x position (in m)
- o.y: the y position (in m)
- o.z: the z position (in m)
- o.vx: the x velocity (in m/s)
- o.vy: the y velocity (in m/s)
- o.vz: the z velocity (in m/s)

The `orbit` object can also make a plot and print out this data in a nice format for you, using the `.plot()` and `.data()` methods. To get a more detailed plot, you can add the argument `show_periode=True` or `.plot()` command. Inside the `o`:

Here's an example:

In [11]: o = orbit.Orbit(a=1.5, e=0.5)

2)s | o.plot()



In [12]: x = o.x[0] y = o.y[0] print(x, y)

1.0000000000000002 0.0

The locations of the two foci of the ellipse are given as

- (o.focal1\_x, o.focal1\_y)
- (o.focal2\_x, o.focal2\_y)

We can compute the distance of our point from a focus as:

In [13]: x1 = math.sqrt((o.x[0]-o.focal1\_x)\*\*2 + (o.y[0]-o.focal1\_y)\*\*2)

1.137347173375819

## Kepler's First Law

Kepler's first law says that orbits are ellipses. An ellipse has the property that for any point on the curve, the sum of distances to each focus is constant.

### Question 1

Test out Kepler's first law by creating an orbit with an eccentricity of your choice (between 0 and 1), and compute the sum of the distances to each focus for several points.

Make a table of several values for  $x$ ,  $y$  and  $r+s$ . Express all distances in meters

In [14]: # do your calculations here. If you want to add additional cells to work in,

# you can use "control-n b" to insert a cell below this one

Data collection handout: