Overview

In this lesson you will learn about 2nd order Runge Kutta and the python coding concepts needed to implement it. You'll also see some examples in the context of oscillatory systems.

Coding concepts

Numpy Arrays

By now you should be feeling comfortable using lists and hopefully you know that lists are not to be used for mathematica calculations. To see this consider the code below and add comments after seeing what happens.

In [ ]:
```
from math import sqrt

a = [1,2,3,4.2,6]
b = [8,2,1,3.3]
#c = a**2 # not a valid operand on a python list
d = a + b # addition is like appending lists, but is not element-wise summation (like how vectors add)
#e = sqrt(a) # sqrt does not work either
f = 2 * a # doubles the length of the list by repeatining its current values and attching it to the end of the list
d, f
```

Out[ ]:
```
([1, 2, 3, 4.2, 6, 8, 2, 1, 3.3], [1, 2, 3, 4.2, 6, 1, 2, 3, 4.2, 6])
```

If you want to perform mathematical calculations on large sets of numbers (and you will) you need to use numpy arrays. (pronounced num-pie, short for numerical python.) Study the cell below and add comments after you figure out the take-home message.

In [ ]:
```
from numpy import array,sqrt,sin

"""
Overall, numpy arrays are operable with all algebraic operators unlike python lists offering better flexibility
"""

a = array([1,2,3])
b = array([4,5,6])
c = 5 * a**2 + sqrt(2 * b)
d = a * sin(b)

a, b, c, d
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (Intel(R) AVX) instructions.
```

Out[ ]:
```
(array([1, 2, 3]),
 array([4, 5, 6]),
 array([ 7.82842712, 23.16227766, 48.46410162]),
 array([-0.7568025 , -1.91784855, -0.83824649]))
```

In the cell above, we used the 'array' function to turn a list into an array. You can also use functions to generate arrays.(add comments after you figure out what each function does)

In [ ]:
```
import numpy as np

a = np.linspace(5,15,200) # makes evenly spaced points in an array from 5 to 15 with 200 points in the array
b = np.arange(5,15,.1) # makes evenly spaced points from 5 to 15 as well, but has a step size of 0.1
c = np.ones([5,5]) # makes a 5x5 array filled with ones
d = np.zeros([10]) # make a 10 dimensional vector filled with zeros
e = np.zeros_like(c) # makes a zeros array in the same shape as vector d

a, b, c, d, e
```

Out[ ]:
```
(array([ 5.        ,  5.05025126,  5.10050251,  5.15075377,  5.20100503,
         5.25125628,  5.30150754,  5.35175879,  5.40201005,  5.45226131,
         5.50251256,  5.55276382,  5.60301508,  5.65326633,  5.70351759,
         5.75376884,  5.8040201 ,  5.85427136,  5.90452261,  5.95477387,
         6.00502513,  6.05527638,  6.10552764,  6.15577889,  6.20603015,
         6.25628141,  6.30653266,  6.35678392,  6.40703518,  6.45728643,
         6.50753769,  6.55778894,  6.6080402 ,  6.65829146,  6.70854271,
         6.75879397,  6.80904523,  6.85929648,  6.90954774,  6.95979899,
         7.01005025,  7.06030151,  7.11055276,  7.16080402,  7.21105528,
         7.26130653,  7.31155779,  7.36180905,  7.4120603 ,  7.46231156,
         7.51256281,  7.56281407,  7.61306533,  7.66331658,  7.71356784,
         7.7638191 ,  7.81407035,  7.86432161,  7.91457286,  7.96482412,
         8.01507538,  8.06532663,  8.11557789,  8.16582915,  8.2160804 ,
         8.26633166,  8.31658291,  8.36683417,  8.41708543,  8.46733668,
         8.51758794,  8.5678392 ,  8.61809045,  8.66834171,  8.71859296,
         8.76884422,  8.81909548,  8.86934673,  8.91959799,  8.96984925,
         9.0201005 ,  9.07035176,  9.12060302,  9.17085427,  9.22110553,
         9.27135678,  9.32160804,  9.3718593 ,  9.42211055,  9.47236181,
         9.52261307,  9.57286432,  9.62311558,  9.67336683,  9.72361809,
         9.77386935,  9.8241206 ,  9.87437186,  9.92462312,  9.97487437,
        10.02512563, 10.07537688, 10.12562814, 10.1758794 , 10.22613065,
        10.27638191, 10.32663317, 10.37688442, 10.42713568, 10.47738693,
        10.52763819, 10.57788945, 10.6281407 , 10.67839196, 10.72864322,
        10.77889447, 10.82914573, 10.87939698, 10.92964824, 10.9798995 ,
        11.03015075, 11.08040201, 11.13065327, 11.18090452, 11.23115578,
        11.28140704, 11.33165829, 11.38190955, 11.4321608 , 11.48241206,
        11.53266332, 11.58291457, 11.63316583, 11.68341709, 11.73366834,
        11.7839196 , 11.83417085, 11.88442211, 11.93467337, 11.98492462,
        12.03517588, 12.08542714, 12.13567839, 12.18592965, 12.2361809 ,
        12.28643216, 12.33668342, 12.38693467, 12.43718593, 12.48743719,
        12.53768844, 12.5879397 , 12.63819095, 12.68844221, 12.73869347,
        12.78894472, 12.83919598, 12.88944724, 12.93969849, 12.98994975,
        13.04020101, 13.09045226, 13.14070352, 13.19095477, 13.24120603,
        13.29145729, 13.34170854, 13.3919598 , 13.44221106, 13.49246231,
        13.54271357, 13.59296482, 13.64321608, 13.69346734, 13.74371859,
        13.79396985, 13.84422111, 13.89447236, 13.94472362, 13.99497487,
        14.04522613, 14.09547739, 14.14572864, 14.1959799 , 14.24623116,
        14.29648241, 14.34673367, 14.39698492, 14.44723618, 14.49748744,
        14.54773869, 14.59798995, 14.64824121, 14.69849246, 14.74874372,
        14.79899497, 14.84924623, 14.89949749, 14.94974874, 15.        ]),
 array([ 5. ,  5.1,  5.2,  5.3,  5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6. ,
         6.1,  6.2,  6.3,  6.4,  6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1,
         7.2,  7.3,  7.4,  7.5,  7.6,  7.7,  7.8,  7.9,  8. ,  8.1,  8.2,
         8.3,  8.4,  8.5,  8.6,  8.7,  8.8,  8.9,  9. ,  9.1,  9.2,  9.3,
         9.4,  9.5,  9.6,  9.7,  9.8,  9.9, 10. , 10.1, 10.2, 10.3, 10.4,
        10.5, 10.6, 10.7, 10.8, 10.9, 11. , 11.1, 11.2, 11.3, 11.4, 11.5,
        11.6, 11.7, 11.8, 11.9, 12. , 12.1, 12.2, 12.3, 12.4, 12.5, 12.6,
        12.7, 12.8, 12.9, 13. , 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, 13.7,
        13.8, 13.9, 14. , 14.1, 14.2, 14.3, 14.4, 14.5, 14.6, 14.7, 14.8,
        14.9]),
 array([[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]),
 array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]),
 array([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]]))
```

Numpy Functions

A wealth of useful functions involving array are available with numpy. I won't try to show all of them to you, but below you will find some useful ones. (add comments after you figure out what each function does) For a more comprehensive list of functions available in numpy, see here

In [ ]:
```
a = np.linspace(5,15,200) # again, the linspace and arange funcs ****
b = np.arange(5,15,.1)
c = np.copy(b) # copies the array components to a new place in memory - changes to the copy are not inhertited to the original
d = np.linalg.norm(b) # gets the magnitude of a vector - sqrt(a^2 + b^2 + c^2 ...) = norm(v)
e = np.sum(a) # adds the elements of the array together
f = np.prod(b) # takes the product of every element in the array
x = array([1,2,3])
y = array([3,1,9])
g = np.cross(x,y) # cross product
h = np.dot(x,y) # and dot product
a, b, c, d, e, f, g, h
```

Out[ ]:
```
(array([ 5.        ,  5.05025126,  5.10050251,  5.15075377,  5.20100503,
         5.25125628,  5.30150754,  5.35175879,  5.40201005,  5.45226131,
         5.50251256,  5.55276382,  5.60301508,  5.65326633,  5.70351759,
         5.75376884,  5.8040201 ,  5.85427136,  5.90452261,  5.95477387,
         6.00502513,  6.05527638,  6.10552764,  6.15577889,  6.20603015,
         6.25628141,  6.30653266,  6.35678392,  6.40703518,  6.45728643,
         6.50753769,  6.55778894,  6.6080402 ,  6.65829146,  6.70854271,
         6.75879397,  6.80904523,  6.85929648,  6.90954774,  6.95979899,
         7.01005025,  7.06030151,  7.11055276,  7.16080402,  7.21105528,
         7.26130653,  7.31155779,  7.36180905,  7.4120603 ,  7.46231156,
         7.51256281,  7.56281407,  7.61306533,  7.66331658,  7.71356784,
         7.7638191 ,  7.81407035,  7.86432161,  7.91457286,  7.96482412,
         8.01507538,  8.06532663,  8.11557789,  8.16582915,  8.2160804 ,
         8.26633166,  8.31658291,  8.36683417,  8.41708543,  8.46733668,
         8.51758794,  8.5678392 ,  8.61809045,  8.66834171,  8.71859296,
         8.76884422,  8.81909548,  8.86934673,  8.91959799,  8.96984925,
         9.0201005 ,  9.07035176,  9.12060302,  9.17085427,  9.22110553,
         9.27135678,  9.32160804,  9.3718593 ,  9.42211055,  9.47236181,
         9.52261307,  9.57286432,  9.62311558,  9.67336683,  9.72361809,
         9.77386935,  9.8241206 ,  9.87437186,  9.92462312,  9.97487437,
        10.02512563, 10.07537688, 10.12562814, 10.1758794 , 10.22613065,
        10.27638191, 10.32663317, 10.37688442, 10.42713568, 10.47738693,
        10.52763819, 10.57788945, 10.6281407 , 10.67839196, 10.72864322,
        10.77889447, 10.82914573, 10.87939698, 10.92964824, 10.9798995 ,
        11.03015075, 11.08040201, 11.13065327, 11.18090452, 11.23115578,
        11.28140704, 11.33165829, 11.38190955, 11.4321608 , 11.48241206,
        11.53266332, 11.58291457, 11.63316583, 11.68341709, 11.73366834,
        11.7839196 , 11.83417085, 11.88442211, 11.93467337, 11.98492462,
        12.03517588, 12.08542714, 12.13567839, 12.18592965, 12.2361809 ,
        12.28643216, 12.33668342, 12.38693467, 12.43718593, 12.48743719,
        12.53768844, 12.5879397 , 12.63819095, 12.68844221, 12.73869347,
        12.78894472, 12.83919598, 12.88944724, 12.93969849, 12.98994975,
        13.04020101, 13.09045226, 13.14070352, 13.19095477, 13.24120603,
        13.29145729, 13.34170854, 13.3919598 , 13.44221106, 13.49246231,
        13.54271357, 13.59296482, 13.64321608, 13.69346734, 13.74371859,
        13.79396985, 13.84422111, 13.89447236, 13.94472362, 13.99497487,
        14.04522613, 14.09547739, 14.14572864, 14.1959799 , 14.24623116,
        14.29648241, 14.34673367, 14.39698492, 14.44723618, 14.49748744,
        14.54773869, 14.59798995, 14.64824121, 14.69849246, 14.74874372,
        14.79899497, 14.84924623, 14.89949749, 14.94974874, 15.        ]),
 array([ 5. ,  5.1,  5.2,  5.3,  5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6. ,
         6.1,  6.2,  6.3,  6.4,  6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1,
         7.2,  7.3,  7.4,  7.5,  7.6,  7.7,  7.8,  7.9,  8. ,  8.1,  8.2,
         8.3,  8.4,  8.5,  8.6,  8.7,  8.8,  8.9,  9. ,  9.1,  9.2,  9.3,
         9.4,  9.5,  9.6,  9.7,  9.8,  9.9, 10. , 10.1, 10.2, 10.3, 10.4,
        10.5, 10.6, 10.7, 10.8, 10.9, 11. , 11.1, 11.2, 11.3, 11.4, 11.5,
        11.6, 11.7, 11.8, 11.9, 12. , 12.1, 12.2, 12.3, 12.4, 12.5, 12.6,
        12.7, 12.8, 12.9, 13. , 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, 13.7,
        13.8, 13.9, 14. , 14.1, 14.2, 14.3, 14.4, 14.5, 14.6, 14.7, 14.8,
        14.9]),
 array([ 5. ,  5.1,  5.2,  5.3,  5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6. ,
         6.1,  6.2,  6.3,  6.4,  6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1,
         7.2,  7.3,  7.4,  7.5,  7.6,  7.7,  7.8,  7.9,  8. ,  8.1,  8.2,
         8.3,  8.4,  8.5,  8.6,  8.7,  8.8,  8.9,  9. ,  9.1,  9.2,  9.3,
         9.4,  9.5,  9.6,  9.7,  9.8,  9.9, 10. , 10.1, 10.2, 10.3, 10.4,
        10.5, 10.6, 10.7, 10.8, 10.9, 11. , 11.1, 11.2, 11.3, 11.4, 11.5,
        11.6, 11.7, 11.8, 11.9, 12. , 12.1, 12.2, 12.3, 12.4, 12.5, 12.6,
        12.7, 12.8, 12.9, 13. , 13.1, 13.2, 13.3, 13.4, 13.5, 13.6, 13.7,
        13.8, 13.9, 14. , 14.1, 14.2, 14.3, 14.4, 14.5, 14.6, 14.7, 14.8,
        14.9]),
 103.60260614482611,
 2000.0,
 6.261772482198133e+97,
 array([15,  0, -5]),
 32)
```

2nd order Runge-Kutta

Recall where Euler's method came from: We started with a Taylor's series approximation and truncated that expression to include only terms up to first order (second order and higher terms were neglected). We can get away with this if we require Δt to be very small. Euler's method is a first-order method precisely because it includes terms up to first order, but not beyond. To improve upon Euler's method let's see if we can do some nifty algebra to get the higher order terms to cancel out; hence we won't have to truncate them. We'll start with the familiar Taylor series of a general function, but this time we'll write down two expansion: i) one that steps forward $\frac{1}{2}\Delta x$ and ii) one that steps backward $-\frac{1}{2}\Delta x$:

$$f(x+\frac{1}{2}\Delta x) = f(x) + \frac{1}{2}\Delta x f'(x) + \frac{1}{8}\Delta x^2 f''(x) + \frac{1}{48}\Delta x^3 f'''(x) + \ldots \frac{(\frac{1}{2}\Delta x)^n}{n!} f^{(n)}(x)$$

$$f(x-\frac{1}{2}\Delta x) = f(x) - \frac{1}{2}\Delta x f'(x) + \frac{1}{8}\Delta x^2 f''(x) - \frac{1}{48}\Delta x^3 f'''(x) + \ldots \frac{(-\frac{1}{2}\Delta x)^n}{n!} f^{(n)}(x)$$

(notice the alternating signs on the terms of the second equation... Why?)

Let's subtract the second equation from the first and rearrange a little.

$$f(x+\frac{1}{2}\Delta x) - f(x-\frac{1}{2}\Delta x) = \Delta x f'(x) + \frac{1}{24}\Delta x^3 f'''(x) + \ldots$$

$$f(x+\frac{1}{2}\Delta x) = f(x-\frac{1}{2}\Delta x) + \Delta x f'(x) + \frac{1}{24}\Delta x^3 f'''(x) + \ldots$$

and now let's shift $x \rightarrow x + \frac{1}{2}\Delta x$:

$$f(x+\Delta x) = f(x) + \Delta x f'(x+\frac{1}{2}\Delta x) + \frac{1}{24}\Delta x^3 f'''(x+\frac{1}{2}\Delta x) + \ldots$$

Notice what has happened: By use of clever algebra, we've arrived at an expression that doesn't have a second order term. Hence, when we truncate the expansion, we will be neglecting third order terms and higher instead of second order terms and higher. The result (after truncation) is:

$$f(x+\Delta x) = f(x) + \Delta x f'(x+\frac{1}{2}\Delta x)$$

Before proceeding, let's re-write this equation using more familiar variables instead of the general x and $f(x)$ that we've been using:

$$v(t+\Delta t) = v(t) + a(t+\frac{1}{2}\Delta t)\Delta t$$

$$v_{n+1} = v_n + a_{n+\frac{1}{2}}\Delta t \text{ (index notation)}$$

This looks exactly like Euler's method except that instead of evaluating the acceleration at t_n, we'd like to evaluate it at $t_{n+\frac{1}{2}}$. But the acceleration function may depend on position (x) and velocity (v), which means that in order to evaluate the acceleration at the half time step, we'll have to know the position and velocity at the half time step. We'll do this using Euler's method.

$$x_{n+\frac{1}{2}} = x_n + v(t_n)\frac{\Delta t}{2} \text{ (Euler's Step)}$$

$$v_{n+\frac{1}{2}} = v_n + a(v_n, x_n, t_n)\frac{\Delta t}{2} \text{ (Euler's Step)}$$

$$x_{n+1} = x_n + v(t_{n+\frac{1}{2}})\Delta t \text{ (RK step)}$$

$$v_{n+1} = v_n + a(x_{n+\frac{1}{2}}, v_{n+\frac{1}{2}}, t_{n+\frac{1}{2}})\Delta t \text{ (RK step)}$$

These equations will work for finding the position and velocity function for one-dimensional motion, but often you are solving more than two equations. For example, in projectile motion, you may need to solve 6 equations for x, y,z, v_x,v_y, and v_z. Wouldn't it be nice if we could put all of the relevant variables into a **single vector** and update all of their values using array math. To that end, you'll often see Runge-Kutta written like this:

$$\vec{k}_1 = \vec{f}(\vec{r}_n, t_n)\Delta t$$

$$\vec{k}_2 = \vec{f}(\vec{r}_n + \frac{1}{2}\vec{k}_1, t_n + \frac{1}{2}\Delta t)\Delta t$$

$$\vec{r}_{n+1} = \vec{r}_n + \vec{k}_2$$

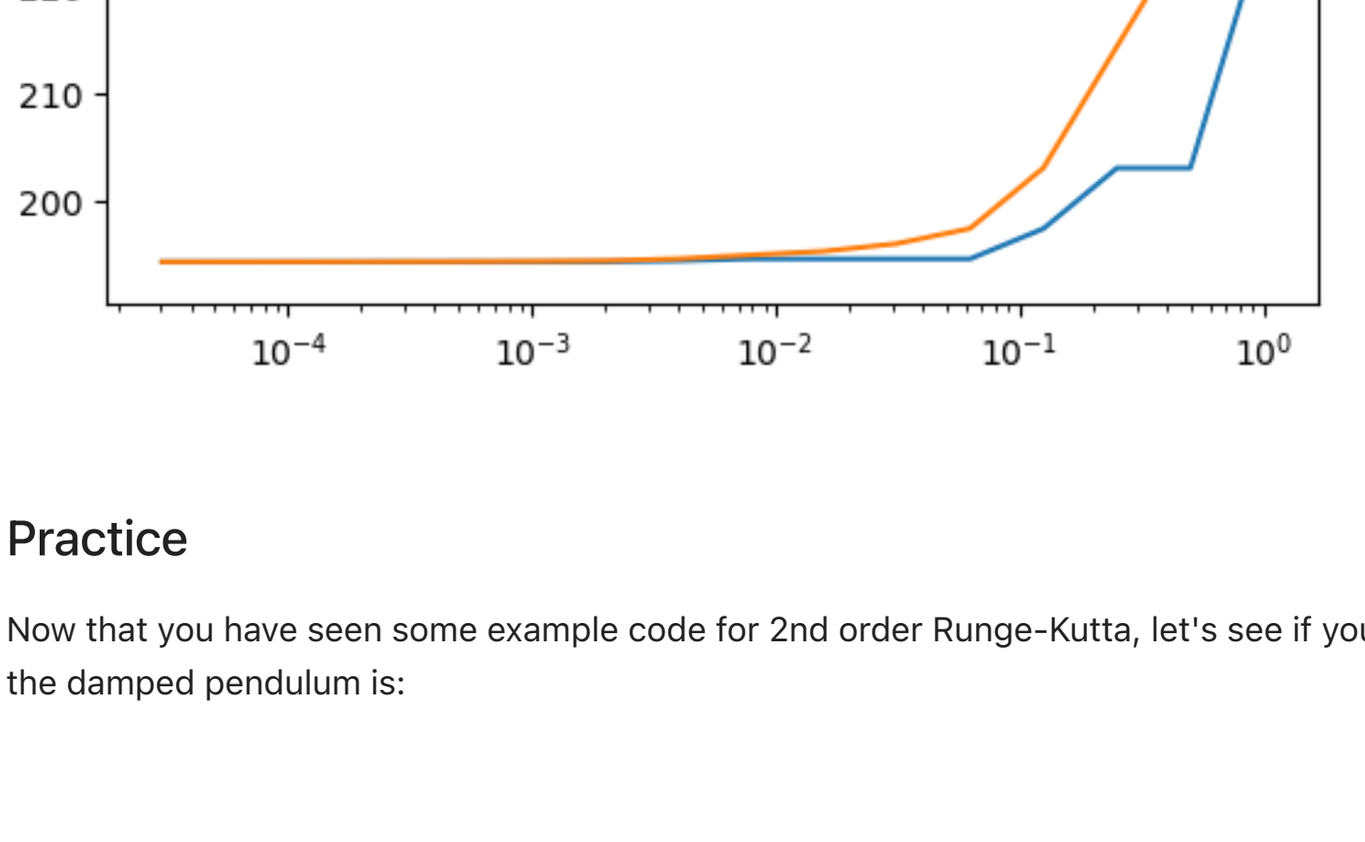
where $\vec{r}_n$ refers to a **vector** containing all of the relevant variables for the problem. Also note that $\vec{f}(r_n, t_n)$ will **return a vector of derivatives**, one corresponding to each relevant variable in the problem. Please take the time to understand the above discussion. Stop and ask questions if you must, but don't move on until it's clear in your mind.

Below you will find a somewhat lengthy code designed to help you see how to implement 2nd order Runge Kutta and to see the improvement of 2nd order Runge Kutta over Euler's method.

1. Study the discussion above thoroughly and ask all of the questions you need to understand second order Runge Kutta.
2. Below you will find a code that implements 2nd order R-K for a projectile with drag. The code is designed to i) help you see how RK is implemented in practice and ii) to help you see how 2nd order RK is better than Euler. Study the code and add comments explaining what each line does. Ask any questions that you have.
3. Execute the code and wait for the output (It might take 10-15 seconds)
4. After looking at the plot, draw a conclusion and write it down in the cell below the plot.

In [ ]:
```
from numpy.linalg import norm
from numpy import arange,array,pi,cos,sin,copy
from matplotlib import pyplot as plt

#  gets the incremented steps dx, dy and dvx, dvy to pass to the trjectory solvers
def derivs(vars):
    r = vars[:2]
    v = vars[2:]
    g = 9.8
    B2 = 4e-5

    xDeriv = v[0]
    yDeriv = v[1]
    speed = norm(v)
    vxDeriv = -B2 * speed * v[0]
    vyDeriv = -B2 * speed * v[1] - g
    return array([xDeriv,yDeriv,vxDeriv,vyDeriv])

# returns the euler algorithm for the next step in the trajectory
def EulerTrajectory(rInitial, vInitial,dt):
    B2 = 4e-5
    g=9.8
    r = copy(rInitial)
    v = copy(vInitial)
    while r[1] >= 0:
        r += v * dt
        speed = norm(v)
        a = array([-B2 * speed * v[0],-B2 * speed * v[1] - g])
        v += a * dt
    return r[0]

# implements the RK2 algorithm and returns a new array component as output
def RK2Trajectory(r,v,dt):
    B2 = 4e-5
    g=9.8
    vars = array([r[0],r[1],v[0],v[1]])
    while vars[1] >= 0:
        k1 = dt * derivs(vars)
        k2 = dt * derivs(vars + 1/2 * k1)
        vars += k2


    return vars[0]

# set the initial conditions
rInitial = array([0.,0.]) # pos
speed = 50
theta = 25 * pi/180
vInitial = array([speed * cos(theta), speed * sin(theta)]) # set velocity by giving direction to the speed
# make empty lists to append to for plotting later on
EulerRanges = []
RK2Ranges = []
# set dt
dt = 1
dts = []

# run a loop to test different dt values and compare euler to RK2
while dt > 2e-5:
    dts.append(dt)
    print("setting dt to {:10.6f}".format(dt))
    r = copy(rInitial)
    v = copy(vInitial)
    rangeEuler = EulerTrajectory(r,v,dt)
    rangeRK = RK2Trajectory(r,v,dt)
    EulerRanges.append(rangeEuler)
    RK2Ranges.append(rangeRK)
    dt /= 2

# make the subplots
fig,ax = plt.subplots(1,1)

# and finally make the plots
ax.plot(dts,RK2Ranges,label='RK')
ax.plot(dts,EulerRanges,label='Euler')
ax.set_xscale('log')
ax.legend()
plt.show()
```

```
setting dt to   1.000000
setting dt to   0.500000
setting dt to   0.250000
setting dt to   0.125000
setting dt to   0.062500
setting dt to   0.031250
setting dt to   0.015625
setting dt to   0.007812
setting dt to   0.003906
setting dt to   0.001953
setting dt to   0.000977
setting dt to   0.000488
setting dt to   0.000244
setting dt to   0.000122
setting dt to   0.000061
setting dt to   0.000031
```



Practice

Now that you have seen some example code for 2nd order Runge-Kutta, let's see if you can write your own. This week we will be studying the pendulum. The differential equation the emerges when considering the damped pendulum is:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta - q\frac{d\theta}{dt}$$

This is a second-order differential equation in θ. Can you write it as two coupled, first-order differential equations? Try it on your own and then look below to see if you got it right...

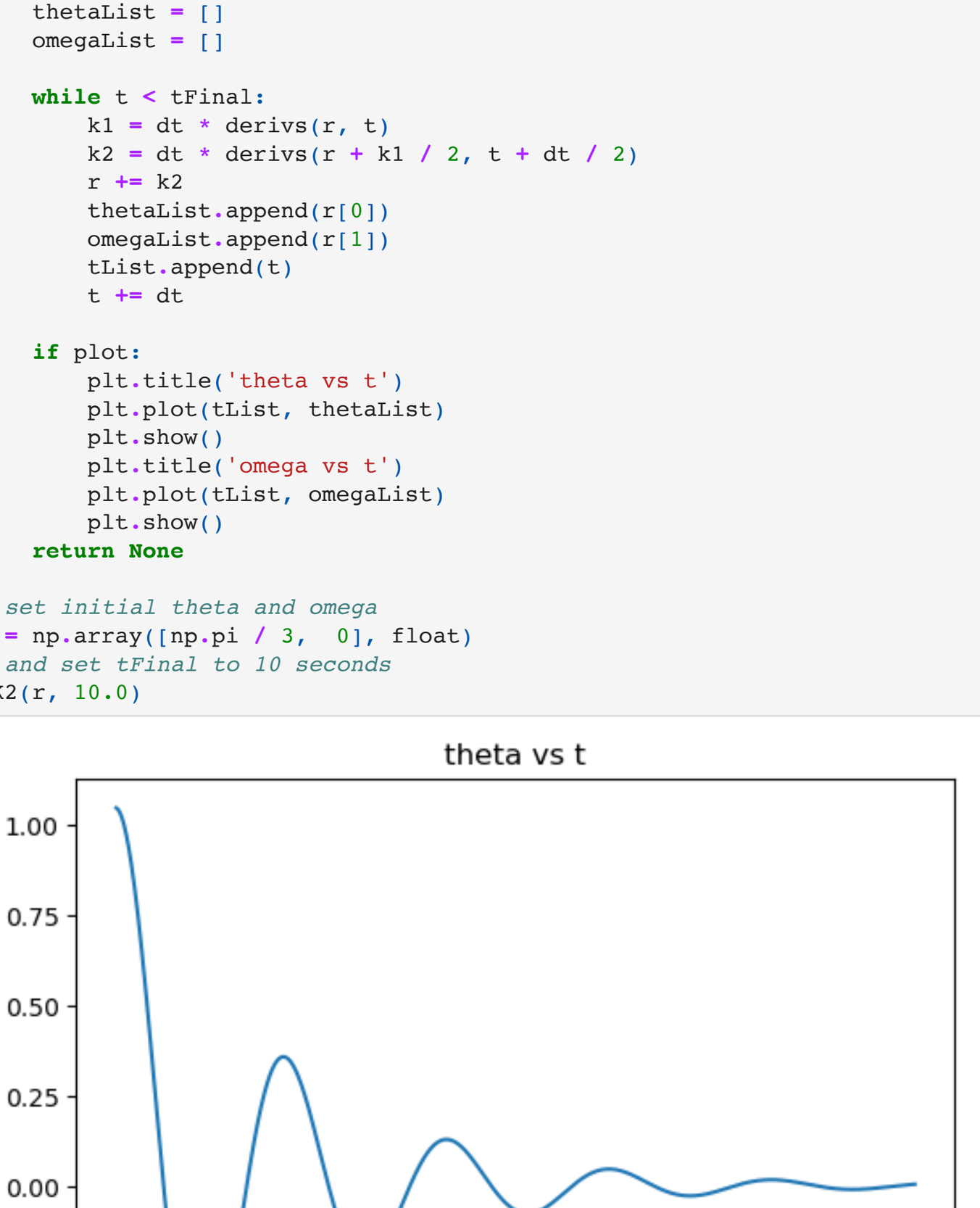
$$\frac{d\omega}{dt} = -\frac{g}{l}\sin\theta - q\omega$$

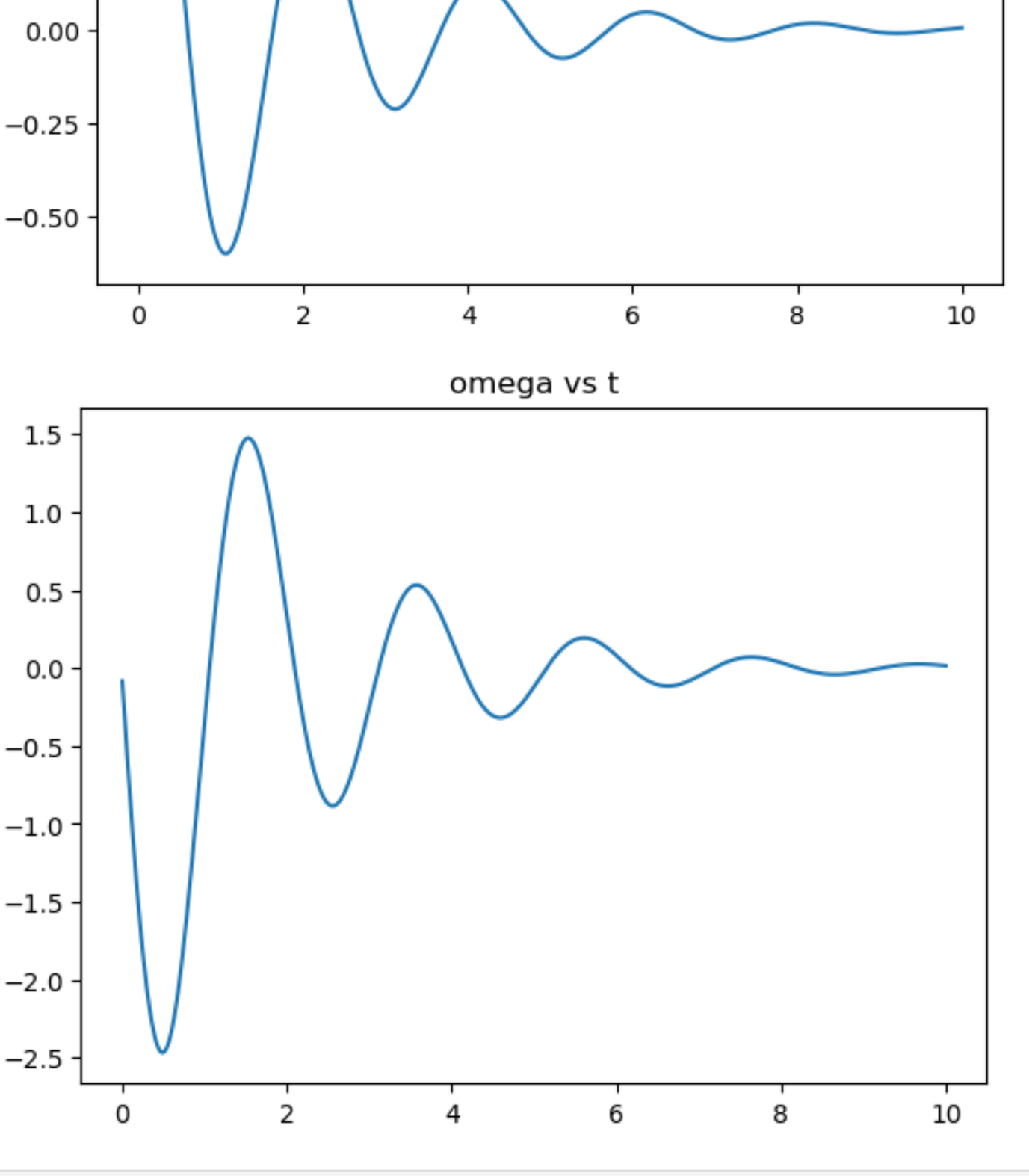
$$\frac{d\theta}{dt} = \omega$$

From this starting point, we can begin to build the code to model the problem:

1. Following the model code above, write a code the uses 2nd order Runge-Kutta to solve the damped pendulum. Plot θ vs time for $0 < t < 10$ s.
2. By playing with the physical parameters in the problem, verify that your results match your intiution for this situation. (You should be able to reproduce figure 3.4 in the book.)

In [ ]:
```
# solving the damped pendulum problem
%matplotlib inline

# define some constants
q = 1
l = 1.0
g = 9.8
dt = 1e-2

# get the differentials
def derivs(r, t):
    return array([r[1], -g / l * sin(r[0]) - q * r[1]])

def RK2(r, tFinal : float, plot : bool = True):
    t = 0.0
    tList = []
    thetaList = []
    omegaList = []

    while t < tFinal:
        k1 = dt * derivs(r, t)
        k2 = dt * derivs(r + k1 / 2, t + dt / 2)
        r += k2
        thetaList.append(r[0])
        omegaList.append(r[1])
        tList.append(t)
        t += dt

    if plot:
        plt.title('theta vs t')
        plt.plot(tList, thetaList)
        plt.show()
        plt.title('omega vs t')
        plt.plot(tList, omegaList)
        plt.show()
    return None

# set initial theta and omega
r = np.array([np.pi / 3,  0], float)
# and set tFinal to 10 seconds
RK2(r, 10.0)
```





In [ ]: