PHY1112: Assignment 10

> Ode to ODEs

Assigned: March 19th, 2024

Due: March 26th, 2024

Learning Objectives

1. Solve a first order ODE in Python
2. Solve a second order ODE in Python

Grade Breakdown

|  |  |  |  |
| --- | --- | --- | --- |
| Part | 1 | 2 | Total |
| Points | 14 | 17 | 31 |
| Score |  |  |  |

**Question 1: A decaying debacle in the lab**

After many weeks in the lab of attempting to create rare, short-lived isotopes of Uranium, Uranium-214, it seems like today was finally the day. Rather than seeing the common, boring, and long-lived Uranium-238 isotope in the sample after the atomic bombardment, you finally saw the tell-tale sign: Astatine-202 and Polonium-198, with trace amounts of Radon-202, Radium-206, and Thorium-210! Based on these daughter isotopes, it’s clear that you must have created Uranium-214. Congratulations!

But now comes the hard part: how much Uranium-214 did you create? The half-life of Uranium-214 is . Since the half-life is so short, measuring the yield of the sample before any of it decays is not an easy task.

At the moment of the atomic collision that would create the Uranium-214, a mass-spectrometer is triggered, and we take this time as . Unfortunately, it is not until that your mass-spectrometer is able to measure the mass of the sample. By this time, an appreciable amount of the sample has decayed. Our task is to use the measurement to determine what the mass of the sample was at .

It is known that the rate at which a radioactive sample decays is dependent on how much of that sample is present. This is because radioactive decay is a random process, and for each atom in the sample, there is a probability for it to decay at any given time. For a single molecule, the probability of that one molecule decaying is relatively low in any given time frame. For a large collection of molecules though, there is a good chance that in any given time frame at least one molecule will decay.

Quantifying this can be done simply – if we call the mass of Uranium-214 at a specific time , then the change in the mass at a specific time is related to the current mass of Uranium-214 as

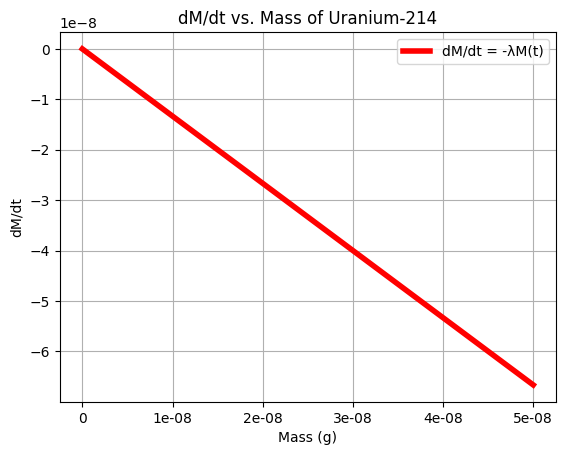
where is referred to as the decay constant. This decay constant is related to the half-life of a specific isotope by

To determine the initial weight of our sample, you will apply your Euler’s method function euler\_method() that you developed in the lab to the differential equation above for .

1. Find the decay constant based on the half-life of Uranium-214 of . The units of should be . **(1 mark)**



1. Using this decay constant, define a python function dMdt() based on the equation above. To be consistent with our Euler’s method function, it should take two inputs, ant , even though only is really needed. **(2 marks)**
2. Plot as a function of mass , for values of the mass ranging from 0 to 50 ng, where one nanogram (ng) is grams. **(2 marks)**



**Figure 1.** The relation between the change in mass of Uranium-214 with respect to time (y-axis) and the mass of Uranium-214 (x-axis) with a decay constant of ≈ 1.33/ms.

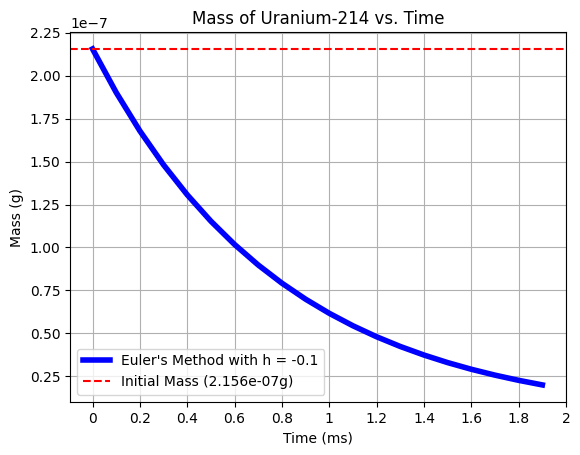
1. The mass of Uranium-214 measured at was

Utilize your euler\_method() function from lab to find the initial mass, of the sample at time . Do this for spacing constants of and (you will need to figure out what values these correspond to). Report your two values. **(4 marks)**

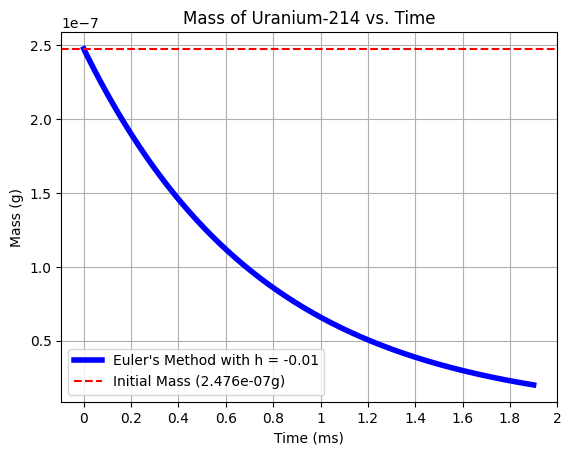
Hint: Euler’s method works just as well backwards as it does forwards. That is, if , Euler’s method still works correctly, as long as is negative.



1. Plot for to for both values. Include a horizontal line at the measured mass . **(2 marks)**



**Figure 2.** The mass of Uranium-214 with respect to time as found using Euler’s method with a step size h = -0.1, from t = 1.9 *ms* to t = 0 *ms* and a mass value of 20 ng at t = 1.9 *ms*. Note the initial mass of roughly g (215 ng).



**Figure 3.** The mass of Uranium-214 with respect to time as found using Euler’s method with a step size h = -0.01, from t = 1.9 *ms* to t = 0 *ms* and a mass value of 20 ng at t = 1.9 *ms*. Note the initial mass of roughly (247 ng).

While your data analysis from above is a good exercise, it would not actually be used in this situation in a real experiment, as the solution is well-known. Namely, we have the following equation:

where e is Euler’s number[[1]](#footnote-2). As such, we can calculate the initial mass of the sample analytically.

1. Using the values from part d), calculate the exact value of . Compare this to the two values you calculated using the Euler method. What can be said about how the magnitude of relates to the accuracy of Euler’s method? **(2 marks)**



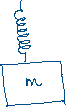
Comparing the different solutions (analytical, Euler h = -0.1, Euler h = -0.01), we can conclude that a smaller step size h (and as a result a greater number of subintervals n) results in a more accurate result.

**(14 marks total, 1 for docstrings/file header/variable naming/comments)**

**Question 2: The Mass-on-Spring Boogie.**

In physics you will encounter harmonic oscillators again and again. It describes any back-and-forth-type motion, such as the dynamics of a mass on a string, a pendulum, RLC electronic circuits, vibrations, and even (as you will see much later) certain quantum states of electrons.

Let’s consider a mass, , hanging from a spring with spring constant . As drawn, the mass is at rest – we define the top of the mass at rest to be at position



Let’s now imagine we pull the mass down, and let go. The mass will start bobbing up and down. Eventually it will return to its rest position because there is also damping in this system that slows the motion down.

We call the vertical displacement from the rest position, and its motion is described by the following second order ODE:

where is a damping constant.

For this problem, we will consider the following parameters, though keep your equations and functions general enough that you can change the parameters easily:

|  |  |
| --- | --- |
| **Parameter** | **Value** |
|  | 1 |
|  | 0.15 |
|  | 1 |
|  | 1 |
|  | 0 |
| Initial time, | 0 |
| Final time, |  |
| Step size, | 0.001 |

1. Re-write the second order ODE of Equation 1 as a system of two first order ODEs.  
   **(2 marks)**

A black background with white text

Description automatically generated

1. Write a Python function `euler\_solver\_2o` that solves the following system of ODEs

via Euler’s method, that is, with the update equations:

The inputs are:

* a function handle for (that itself takes in three inputs)
* a function handle for (that itself takes in three inputs)
* the starting point ,
* the end point,
* the initial condition
* the initial condition
* the number of points to calculate,

The outputs are:

* a 1D NumPy array containing
* a 1D NumPy array containing
* a 1D NumPy array containing

**(6 marks)**

*def* euler\_solver\_2o(*f*, *g*, *a*, *b*, *ua*, *va*, *n*):                                 # function to solve a system of 2 ODEs using Euler's Method

    '''

    (function, function, float, float, float, float, int) -> np.array, np.array, np.array

    Returns the x values, u(x) values, and v(x) values for a second order ODE using Euler's Method.

    Preconditions: f and g must be ODEs of the form f(x, u, v) and g(x, u, v)

    '''

    h = (b - a) / n                                                         # calculate the step size

    x = np.linspace(a, b, n + 1)                                            # create an array of x values from a to b

    u = np.zeros(n + 1)                                                     # create an array of zeros with length n + 1

    v = np.zeros(n + 1)                                                     # create an array of zeros with length n + 1

    u[0] = ua                                                               # set the initial value of u

    v[0] = va                                                               # set the initial value of v

    for i in range(n):                                                      # iterate through the x values

        u[i + 1] = u[i] + h \* f(x[i], u[i], v[i])                           # calculate the next value of u

        v[i + 1] = v[i] + h \* g(x[i], u[i], v[i])                           # calculate the next value of v

    return x, u, v                                                          # return the x values, u values, and v values

1. Solve the second order ODE in Equation 1 via the two single first order ODEs you derived in part a), using the Python function you wrote in part b). Use the parameters in the table given above (you will need to convert the step size to a number of points, ).   
   **(4 marks)**

**A black screen with white numbers

Description automatically generated**

**A graph of a graph showing a number of numbers and a line

Description automatically generated with medium confidence**

**Figure 4.** Graph of the damped harmonic motion of a spring with respect to time showing the displacement of the original motion u(x), the opposing motion v(x), and the combined damped motion u(x) + v(x) over a period of seconds, *m* = 1, = 0.15, and *k* = 1.

1. Rerun your code multiple times to experiment with the following two parameters:
2. What happens as you increase/decrease your step size ?

A smaller h value results in an apparent increase in magnitude of each oscillation. Consequently, a greater h value results in an apparent decrease in magnitude of each oscillation.

1. What happens as you increase/decrease ? As it goes to 0?

As gamma increases, the dampening increases, resulting in fewer oscillations. Conversely, as gamma decreases, the dampening decreases, resulting in more oscillations. If gamma were zero, the graph suggests that the oscillations would grow in magnitude, though they should instead remain constant due to the nature of harmonic motion.

Examine your plots and describe and discuss what you observe as you change the values (though your plots don’t need to be handed in for this part.) Be sure to list the values you used alongside your observations.

**(4 marks)**

1. ***(Optional Bonus 2 Marks – Very Hard!)***You can solve the ODE using SciPy’s `scipy.integrate.solve\_ivp` function, a simple ODE solver for initial value problems. Compare the answer you obtain with `scipy.integrate.solve\_ivp` to the results you obtained in part c).

**No thanks :D**

**(17 marks total, 1 for docstrings/file header/variable naming/comments)**

'''

Filename: assignment\_10.py

Author: Patrick Geraghty

Date Created: 2024-03-25

Date Modified: 2024-03-25

Description: Assignment 9

'''

import numpy as np

import matplotlib.pyplot as plt

# Part 1

*def* decay\_constant(*half\_life*):                                                  # function to calculate decay constant with half-life as input

    '''

    (float) -> float

    Returns the decay constant for a given half-life.

    Preconditions: half\_life > 0

    '''

    # formula to calculate decay constant

    return np.log(2) / half\_life

*def* dMdt(*t*, *M*):                                                                 # function to calculate the derivative of M with respect to t. Note that in this case t is ambiguous.

    '''

    (float, float) -> float

    Returns the derivative of M with respect to t.

    Preconditions: M > 0

    '''

    # formula to calculate the derivative of M with respect to t

    return -decay\_constant(0.52) \* M

*def* dMdt\_plot():                                                                # function to plot the derivative of M with respect to t

    '''

    () -> None

    Plots the derivative of M with respect to t.

    Preconditions: None

    '''

    plt.figure(1)                                                               # create a new figure

    M = np.linspace(0, 0.00000005)                                              # create an array of mass values from 0 to 5e-8

    y = dMdt(0, M)                                                              # calculate the derivative of M with respect to t for each mass value

    plt.plot(M, y, 'r', *label*='dM/dt = -λM(t)', *linewidth*=4, *alpha*=1)           # plot the derivative of M with respect to t

    plt.xlabel('Mass (g)')                                                      # set the x-axis label

    plt.ylabel('dM/dt')                                                         # set the y-axis label

    plt.title('dM/dt vs. Mass of Uranium-214')                                  # set the title of the plot

    xtick\_values = np.linspace(0, 5e-8, 6)                                      # set the x-tick values

    x\_tick\_labels = [0, 10e-9, 20e-9, 30e-9, 40e-9, 50e-9]                      # set the x-tick labels

    plt.xticks(xtick\_values, x\_tick\_labels)                                     # set the x-ticks

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* eulers\_method(*f*, *n*, *a*, *b*, *y0*):                                              # general function for Euler's Method

    h = (a - b) / n                                                             # calculate the step size, reversed a and b to account for negative n

    x = np.linspace(a, b, *num*=abs(n)+1)                                         # create an array of time values from a to b, using abs(n) to account for negative n

    y = np.zeros(abs(n) + 1)                                                    # create an array of zeros with length abs(n) + 1

    y[0] = y0                                                                   # set the initial value

    for i in range(abs(n)):                                                     # iterate through the time values

        y[i + 1] = y[i] + h \* f(x[i], y[i])                                     # calculate the next value of y

    return x, y                                                                 # return the x values and y values

*def* tenth\_step\_euler\_plot():                                                    # function to plot the mass of Uranium-214 using Euler's Method with h = -0.1

    plt.figure(2)                                                               # create a new figure

    tenth\_step\_euler = eulers\_method(dMdt, -19, 1.9, 0, 2e-8)                   # calculate the mass of Uranium-214 using Euler's Method with h = -0.1

    x, y = tenth\_step\_euler                                                     # unpack the x values and y values

    # plot the mass of Uranium-214 using Euler's Method with h = -0.1, followed by a dashed line representing the initial mass

    plt.plot(x, y, 'b', *label*='Euler\'s Method with h = -0.1', *linewidth*=4, *alpha*=1)

    plt.axhline(tenth\_step\_euler[1][-1], *color*='r', *linestyle*='dashed', *label*=*f*'Initial Mass ({tenth\_step\_euler[1][-1]*:.3e*}g)')

    xtick\_values = np.linspace(0, 2, 11)                                        # set the x-tick values

    xtick\_labels = [0, 0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8, 2]            # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (ms)')                                                     # set the x-axis label

    plt.ylabel('Mass (g)')                                                      # set the y-axis label

    plt.title('Mass of Uranium-214 vs. Time')                                   # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* hundredth\_step\_euler\_plot():                                                # function to plot the mass of Uranium-214 using Euler's Method with h = -0.01

    plt.figure(3)                                                               # create a new figure

    hundredth\_step\_euler = eulers\_method(dMdt, -190, 1.9, 0, 2e-8)              # calculate the mass of Uranium-214 using Euler's Method with h = -0.01

    x, y = hundredth\_step\_euler                                                 # unpack the x values and y values

    # plot the mass of Uranium-214 using Euler's Method with h = -0.01, followed by a dashed line representing the initial mass

    plt.plot(x, y, 'b', *label*='Euler\'s Method with h = -0.01', *linewidth*=4, *alpha*=1)

    plt.axhline(hundredth\_step\_euler[1][-1], *color*='r', *linestyle*='dashed', *label*=*f*'Initial Mass ({hundredth\_step\_euler[1][-1]*:.3e*}g)')

    xtick\_values = np.linspace(0, 2, 11)                                        # set the x-tick values

    xtick\_labels = [0, 0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8, 2]            # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (ms)')                                                     # set the x-axis label

    plt.ylabel('Mass (g)')                                                      # set the y-axis label

    plt.title('Mass of Uranium-214 vs. Time')                                   # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* analytical\_solution(*t*, *Ma*, *a*):                                              # function to calculate the analytical solution for the mass of Uranium-214

    return Ma \* np.e \*\* (-decay\_constant(0.52) \* (t - a))

# Part 2

*def* euler\_solver\_2o(*f*, *g*, *a*, *b*, *ua*, *va*, *n*, *gamma*=0.15):                         # function to solve a system of 2 ODEs using Euler's Method

    '''

    (function, function, float, float, float, float, int) -> np.array, np.array, np.array

    Returns the x values, u(x) values, and v(x) values for a second order ODE using Euler's Method.

    Preconditions: f and g must be ODEs of the form f(x, u, v) and g(x, u, v)

    '''

    h = (b - a) / n                                                             # calculate the step size

    x = np.linspace(a, b, n + 1)                                                # create an array of x values from a to b

    u = np.zeros(n + 1)                                                         # create an array of zeros with length n + 1

    v = np.zeros(n + 1)                                                         # create an array of zeros with length n + 1

    u[0] = ua                                                                   # set the initial value of u

    v[0] = va                                                                   # set the initial value of v

    for i in range(n):                                                          # iterate through the x values

        u[i + 1] = u[i] + h \* f(x[i], u[i], v[i])                               # calculate the next value of u

        v[i + 1] = v[i] + h \* g(x[i], u[i], v[i], gamma)                        # calculate the next value of v

    return x, u, v                                                              # return the x values, u values, and v values

*def* f(*x*, *u*, *v*):                                                                 # function to calculate the derivative of u with respect to x

    '''

    (float, float, float) -> float

    Returns the derivative of u with respect to x.

    Preconditions: None

    '''

    return v

*def* g(*x*, *u*, *v*, *gamma*):                                                          # function to calculate the derivative of v with respect to x

    '''

    (float, float, float) -> float

    Returns the derivative of v with respect to x.

    Preconditions: None

    '''

    return -gamma \* v - u

*def* damped\_harmonic\_motion\_plot():                                              # function to plot the solution to the second order ODE

    plt.figure(4)                                                               # create a new figure

    euler\_2o = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 20000)                # calculate the solution to the second order ODE

    x, u, v = euler\_2o                                                          # unpack the x values, u values, and v values

    # plot the solution to the second order ODE

    plt.plot(x, u, 'b', *label*='u(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, v, 'r', *label*='v(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, np.sqrt(u \*\* 2 + v \*\* 2), 'g', *label*='u(x) + v(x)', *linewidth*=4, *alpha*=1)

    xtick\_values = np.linspace(0, 20 \* np.pi, 11)                               # set the x-tick values

    xtick\_labels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]                      # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (t)')                                                      # set the x-axis label

    plt.ylabel('Displacement (m)')                                              # set the y-axis label

    plt.title('Damped Harmonic Motion of a Spring')                             # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* damped\_harmonic\_motion\_plot\_increase\_h():                                   # function to plot the solution to the second order ODE

    plt.figure(5)                                                               # create a new figure

    euler\_2o = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 200000)               # calculate the solution to the second order ODE with increased h

    x, u, v = euler\_2o                                                          # unpack the x values, u values, and v values

    # plot the solution to the second order ODE

    plt.plot(x, u, 'b', *label*='u(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, v, 'r', *label*='v(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, np.sqrt(u \*\* 2 + v \*\* 2), 'g', *label*='u(x) + v(x)', *linewidth*=4, *alpha*=1)

    xtick\_values = np.linspace(0, 20 \* np.pi, 11)                               # set the x-tick values

    xtick\_labels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]                      # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (t)')                                                      # set the x-axis label

    plt.ylabel('Displacement (m)')                                              # set the y-axis label

    plt.title('Damped Harmonic Motion of a Spring with larger h')               # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* damped\_harmonic\_motion\_plot\_decrease\_h():                                   # function to plot the solution to the second order ODE

    plt.figure(6)                                                               # create a new figure

    euler\_2o = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 2000)                 # calculate the solution to the second order ODE with decreased h

    x, u, v = euler\_2o                                                          # unpack the x values, u values, and v values

    # plot the solution to the second order ODE

    plt.plot(x, u, 'b', *label*='u(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, v, 'r', *label*='v(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, np.sqrt(u \*\* 2 + v \*\* 2), 'g', *label*='u(x) + v(x)', *linewidth*=4, *alpha*=1)

    xtick\_values = np.linspace(0, 20 \* np.pi, 11)                               # set the x-tick values

    xtick\_labels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]                      # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (t)')                                                      # set the x-axis label

    plt.ylabel('Displacement (m)')                                              # set the y-axis label

    plt.title('Damped Harmonic Motion of a Spring with smaller h')              # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* damped\_harmonic\_motion\_plot\_increase\_gamma():                               # function to plot the solution to the second order ODE

    plt.figure(7)                                                               # create a new figure

    euler\_2o = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 20000, 1.5)           # calculate the solution to the second order ODE with increased gamma

    x, u, v = euler\_2o                                                          # unpack the x values, u values, and v values

    # plot the solution to the second order ODE

    plt.plot(x, u, 'b', *label*='u(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, v, 'r', *label*='v(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, np.sqrt(u \*\* 2 + v \*\* 2), 'g', *label*='u(x) + v(x)', *linewidth*=4, *alpha*=1)

    xtick\_values = np.linspace(0, 20 \* np.pi, 11)                               # set the x-tick values

    xtick\_labels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]                      # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (t)')                                                      # set the x-axis label

    plt.ylabel('Displacement (m)')                                              # set the y-axis label

    plt.title('Damped Harmonic Motion of a Spring with increased gamma')                             # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* damped\_harmonic\_motion\_plot\_decrease\_gamma():                               # function to plot the solution to the second order ODE

    plt.figure(8)                                                               # create a new figure

    euler\_2o = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 20000, 0.015)         # calculate the solution to the second order ODE with decreased gamma

    x, u, v = euler\_2o                                                          # unpack the x values, u values, and v values

    # plot the solution to the second order ODE

    plt.plot(x, u, 'b', *label*='u(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, v, 'r', *label*='v(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, np.sqrt(u \*\* 2 + v \*\* 2), 'g', *label*='u(x) + v(x)', *linewidth*=4, *alpha*=1)

    xtick\_values = np.linspace(0, 20 \* np.pi, 11)                               # set the x-tick values

    xtick\_labels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]                      # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (t)')                                                      # set the x-axis label

    plt.ylabel('Displacement (m)')                                              # set the y-axis label

    plt.title('Damped Harmonic Motion of a Spring with decreased gamma')        # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* damped\_harmonic\_motion\_plot\_zero\_gamma():                                   # function to plot the solution to the second order ODE

    plt.figure(9)                                                               # create a new figure

    euler\_2o = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 20000, 0)             # calculate the solution to the second order ODE with gamma = 0

    x, u, v = euler\_2o                                                          # unpack the x values, u values, and v values

    # plot the solution to the second order ODE

    plt.plot(x, u, 'b', *label*='u(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, v, 'r', *label*='v(x)', *linewidth*=4, *alpha*=1)

    plt.plot(x, np.sqrt(u \*\* 2 + v \*\* 2), 'g', *label*='u(x) + v(x)', *linewidth*=4, *alpha*=1)

    xtick\_values = np.linspace(0, 20 \* np.pi, 11)                               # set the x-tick values

    xtick\_labels = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]                      # set the x-tick labels

    plt.xticks(xtick\_values, xtick\_labels)                                      # set the x-ticks

    plt.xlabel('Time (t)')                                                      # set the x-axis label

    plt.ylabel('Displacement (m)')                                              # set the y-axis label

    plt.title('Damped Harmonic Motion of a Spring, gamma = 0')                  # set the title of the plot

    plt.grid()                                                                  # add a grid to the plot

    plt.legend()                                                                # add a legend to the plot

    plt.show()

*def* main():

    # Part 1

    print('Part 1')

    print(*f*'Decay constant for Uranium-214: {decay\_constant(0.52)}/ms.')

    dMdt\_plot()

    tenth\_step\_euler = eulers\_method(dMdt, -19, 1.9, 0, 2e-8)

    hundredth\_step\_euler = eulers\_method(dMdt, -190, 1.9, 0, 2e-8)

    print(*f*'Initial Mass of Uranium-214 using Euler\'s Method with h = -0.1: {tenth\_step\_euler[1][-1]}g.')

    print(*f*'Initial Mass of Uranium-214 using Euler\'s Method with h = -0.01: {hundredth\_step\_euler[1][-1]}g.')

    tenth\_step\_euler\_plot()

    hundredth\_step\_euler\_plot()

    print(*f*'Analytical solution for Uranium-214 at t = 0: {analytical\_solution(0, 2e-8, 1.9)}g.')

    print()

    print()

    print()

    # Part 2

    print('Part 2')

    euler\_2o\_v1 = euler\_solver\_2o(f, g, 0, 20 \* np.pi, 1, 0, 20000)

    print('x: ',euler\_2o\_v1[0])

    print('u: ',euler\_2o\_v1[1])

    print('v: ',euler\_2o\_v1[2])

    damped\_harmonic\_motion\_plot()

    damped\_harmonic\_motion\_plot\_increase\_h()

    damped\_harmonic\_motion\_plot\_decrease\_h()

    damped\_harmonic\_motion\_plot\_increase\_gamma()

    damped\_harmonic\_motion\_plot\_decrease\_gamma()

    damped\_harmonic\_motion\_plot\_zero\_gamma()

main()

1. The name of this number is actually a coincidence in this case – while Euler was the first to publish Euler’s method, he was not the discoverer of Euler’s number. In fact, Euler’s number was first derived before Euler was even born! He was, however, the first person to use the letter to represent the number in a publication, and as such its discovery is often (mis-)attributed to him. [↑](#footnote-ref-2)