19.8/20

Homework 1

Deadline: Monday 24 March 2024 (by 19h00)

Credits: 20 points

Instructions:

- When you finish, please submit a single **.ipynb** file via email to wbanda@yachaytech.edu.ec (mailto:wbanda@yachaytech.edu.ec)
- The homework is **individual**. Please include your name in the notebook.
- Within a **single python notebook**, solve the following problems.

Name: Alan Palma Travez

Solution:

1. Population dynamics (8 points)

The system of ordinary differential equations (ODEs) describing the population dynamics of two prey species (x and y) and one predator species (z) is given by:

$$\begin{aligned}
\frac{dx}{dt} &= g_1\, x \left(1 - \frac{x}{c_1}\right) - p_1\, x\, z, \\
\frac{dy}{dt} &= g_2\, y \left(1 - \frac{y}{c_2}\right) - p_2\, y\, z, \\
\frac{dz}{dt} &= e_1\, p_1\, x\, z + e_2\, p_2\, y\, z - d\, z.
\end{aligned}$$

where:

- g_1, g_2 are the intrinsic growth rates (e.g. birth rates) of the prey populations,
- c_1, c_2 are the carrying capacities of the prey populations (the carrying capacity of an environment is the maximum population size of a biological species that can be sustained by that specific environment),
- p_1, p_2 are the predation rates of the predator on each prey (e.g. how successful a hunt is),
- e_1, e_2 are the conversion efficiencies of consumed prey into predator biomass (the conversion efficiency tells us how efficiently a predator can use the energy from its prey to reproduce),
- d is the natural death rate of the predator. Note that the death rates of the two prey species are intrinsically given by their individual carrying capacities.

This system models the interactions where both prey species grow logistically but are

State vector and slope:

(a) Write down this system of ODEs in terms of the system state vector $S(t)$, i.e. $\frac{dS}{dt} = F(S)$. Identify the slope function $F(S)$ and indicate whether or not it can be explicitly written as a function of S. Create a python function for the slope $F(S)$.

$$S(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix} \Rightarrow F = \frac{ds(t)}{dt} = \begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \end{bmatrix}$$

Then,

$$F(S) = \begin{bmatrix} x'(t) \\ y'(t) \\ z'(t) \end{bmatrix} = \begin{bmatrix} g_1 x - \frac{g_1}{c_1} x^2 - p_1 xz \\ g_2 y - \frac{g_2}{c_2} y^2 - p_2 yz \\ e_1 p_1 xz + e_2 p_2 yz - dz \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Form here we can deduce the matrix components:

$$= \begin{bmatrix} g_1 & -\frac{g_1}{c_1}\frac{x^2}{y} & -p_1 x \\ g_2 \frac{y}{x} & -\frac{g_2}{c_2} y & -p_2 y \\ e_1 p_1 z & +e_2 p_2 z & -d \end{bmatrix} \begin{bmatrix} x(t) \\ y(t) \\ z(t) \end{bmatrix}$$

Finally the system of ODEs can be expressed in function of F(s) as follow:

$$F(S) = \begin{bmatrix} g_1 & -\frac{g_1}{c_1}\frac{x^2}{y} & -p_1 x \\ g_2 \frac{y}{x} & -\frac{g_2}{c_2} y & -p_2 y \\ e_1 p_1 z & +e_2 p_2 z & -d \end{bmatrix} S(t)$$

Notice that the slope matrix has components of the vector S(t), i.e., $S[0] = x(t)$, $S[1] = y(t)$ and $S[2] = z(t)$

$$F(S) = \begin{bmatrix} g_1 & -\frac{g_1}{c_1}\frac{S[0]^2}{S[1]} & -p_1 S[0] \\ g_2 \frac{S[1]}{S[0]} & -\frac{g_2}{c_2} S[1] & -p_2 S[1] \\ e_1 p_1 S[2] & +e_2 p_2 S[2] & -d \end{bmatrix} S(t)$$

As it is seen the function slope is explicitly written as function of $S(t)$, that is: $F = F(S(t))$. Define M as the matriz slope that also depent on $S(t)$, then:

$$F(S(t)) = M(S(t)) * S(t)$$

In [ ]:
```
# Third party libraries

import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import scienceplots
import sympy as sp
from scipy.integrate import solve_ivp
from sympy.solvers.ode.systems import dsolve_system
import scipy.optimize as opt
from scipy import constants

# To see the outputs in latex format, we use:
from sympy.interactive import printing
printing.init_printing(use_latex = True)

# Define the style
plt.style.use(['science', 'notebook', 'no-latex']) # Use a specific si
```

In [ ]:
```
def F(d, t, s):
    """
    Function that contains the slope of a system of ODEs of first orde
    acoording to the slope matrix should be arranged this way s(t) =
    Inputs:
        t(float): time variable.
        s(array):  state vector of the system.
        d(dic): dictionary containing the parameters of the system.
    Outputs:
        slope(float): array of the computed slope of the system.
    """
    # Slope matrix based on the system of ODEs and paremeters
    m = np.array([
        [d["g_1"], -(d["g_1"]*s[0]**2)/(d["c_1"]*s[1]), -d["p_1"]*s[0]
        [d["g_2"]*(s[1]/s[0]) , -(d["g_2"]/d["c_2"])*s[1], -d["p_2"]*s
        [d["e_1"]*d["p_1"]*s[2] , d["e_2"]*d["p_2"]*s[2] , -d["d"]]
        ])

    # Compute the slope vector
    slope = np.dot(m, s)

    return slope
```

ODE integration methods:

(b) Create a python function that implements a trapezoidal Euler method for ODE integration.

For trapezoidal Euler method, the predictor sptep is implemented as:

$$S_{j+1} = S_j + hF(t_j, S_j)$$

$$S_{j+1} = S_j + \frac{h}{2}(F(t_j, S_j) + F(t_{j+1}, S_{j+1}))$$

In [ ]:
```python
def trapezoidal_E(d, t, sol):
    """
    Solves an ODE using the explicit trapezoidal Euler method.
    This function implements a predictor-corrector scheme based on the
    to approximate the solution of a first-order differential equation
    Inputs:
        d(dic): dictionary containing the parameters of the system.
        h (float): Step size.
        t (array): Time points where the solution is computed.
        sol (array): Array to store the solution.
    Outputs:
        sol (array): Updated array with the numerical solution of the
    """
    h = d["dt"] # Define the step size

    for j in range(0, len(t) - 1):
        sol[j + 1] = sol[j] + h*F(d, t[j], sol[j]) # Predictor step
        sol[j + 1] = sol[j] + h*(F(d, t[j], sol[j]) + F(d, t[j + 1], s

    return sol
```

(c) Design your own third-order RK method (RK3), and create a python function for this integrator.

Define the 3 slopes for third-order RK method as follows:

$$k_1 = F(t_j, S(t_j))$$

$$k_2 = F\left(t_j + \frac{h}{2}, S(t_j) + \frac{1}{2}k_1 h\right)$$

$$k_3 = F\left(t_j + h, S(t_j) + k_2 h\right)$$

Therefore, we will have:

$$S(t_{j+1}) = S(t_j) + \frac{h}{4}(k_1 + 2k_2 + k_3)\,.$$

In [ ]:
```python
def RK3(d, t, sol):
    """
    Solves an ODE using the third-order Runge-Kutta (RK3) method.
    Inputs:
        d(dic): dictionary containing the parameters of the system.
        h (float): Step size.
        t (array): Time points where the solution is computed.
        sol (array): Array to store the solution.
    Outputs:
        sol (array): Updated array with the numerical solution of the
    """
    h = d["dt"] # Define the step size

    for j in range(0, len(t) - 1):

        # Compute RK3 intermediate slopes
        k1 = F(d, t[j], sol[j])  # Slope 1 -> k1
        k2 = F(d, t[j] + h/2, sol[j] + h*k1/2) # Slope 2 -> k2
        k3 = F(d, t[j] + h, sol[j] + h*(k2)) # Slope 3 -> k3

        # Compute the weighted sum of slopes for the final approximati
        sol[j + 1] = sol[j] + h*(k1 + 2.*k2 + k3)/4.

    return sol
```

(d) Write a python function for the **Butcher's Runge-Kutta method**, which is a popular method for integrating ODEs with a high order of accuracy. It is obtained by following a similar approach to the one we discussed in class. This method uses six points k_1, k_2, k_3, k_4, k_5, and k_6. A weighted average of these points is used to produce the approximation of the solution. The algorithm relies on computing the following slopes:

$$k_1 = F(t_j, S(t_j))$$

$$k_2 = F\left(t_j + \frac{h}{4}, S(t_j) + \frac{1}{4}k_1 h\right)$$

$$k_3 = F\left(t_j + \frac{h}{4}, S(t_j) + \frac{1}{8}k_1 h + \frac{1}{8}k_2 h\right)$$

$$k_4 = F\left(t_j + \frac{h}{2}, S(t_j) - \frac{1}{2}k_2 h + k_3 h\right)$$

$$k_5 = F\left(t_j + \frac{3h}{4}, S(t_j) + \frac{3}{16}k_1 h + \frac{9}{16}k_4 h\right)$$

$$k_6 = F\left(t_j + h, S(t_j) - \frac{3}{7}k_1 h + \frac{2}{7}k_2 h + \frac{12}{7}k_3 h - \frac{12}{7}k_4 h + \frac{8}{7}k_5 h\right)$$

The solution is then constructed with the step size (h, same as `dt`)as follows:

$$S(t_{j+1}) = S(t_j) + \frac{h}{90}(7k_1 + 32k_3 + 12k_4 + 32k_5 + 7k_6)\,.$$

In [ ]:
```python
def B_RK(d, t, sol):
    """
    Solves an ODE using the Butcher's Runge-Kutta (B_RK) method.
    Inputs:
        d(dic): dictionary containing the parameters of the system.
        h (float): Step size.
        t (array): Time points where the solution is computed.
        sol (array): Array to store the solution.
    Outputs:
        sol (array): Updated array with the numerical solution of the
    """

    h = d["dt"] # Define the step size

    for j in range(0, len(t) - 1):

        # Compute BRK intermediate slopes
        k1 = F(d, t[j], sol[j]) # Slope 1 -> k1
        k2 = F(d, t[j] + h/4, sol[j] + h*k1/4) # Slope 2 -> k2
        k3 = F(d, t[j] + h/4, sol[j] + h*k1/8 + h*k2/8) # Slope 3 -> k
        k4 = F(d, t[j] + h/2, sol[j] - h*k2/2 + h*k3) # Slope 3 -> k3
        k5 = F(d, t[j] + 3*h/4, sol[j] + 3*h*k1/16 + 9*h*k4/16) # Slop
        k6 = F(d, t[j] + h, sol[j] - 3*h*k1/7 + 2*h*k2/7 + 12*h*k3/7 -

        # Compute the weighted sum of slopes for the final approximati
        sol[j + 1] = sol[j] + h*(7*k1 + 32*k3 + 12*k4 + 32*k5 + 7*k6)/

    return sol
```

Settings and initial conditions:

(e) Create a dictionary that allocates all the user-defined initial conditions needed for integration, including an option for the user to select the integration method. The default parameters in the dictionary should be:

Parameter	Value	Description
g1	1.0	Growth rate of prey 1
c1	200.0	Carrying capacity of prey 1
p1	0.01	Predation rate of predator on prey 1
g2	1.2	Growth rate of prey 2
c2	150.0	Carrying capacity of prey 2
p2	0.008	Predation rate of predator on prey 2
e1	0.08	Conversion efficiency of prey 1 to predator
e2	0.07	Conversion efficiency of prey 2 to predator
d	0.15	Death rate of predator
x0	100.0	Initial population of prey 1
y0	80.0	Initial population of prey 2
z0	20.0	Initial population of predator
t_span	(0, 200)	Time span for simulation
dt	0.01	Time step size (= h)

Parameter	Value	Description

In [ ]:
```python
# Define the parameters of the system in a dictionary
d = {
    "g_1": 1.0, "g_2": 1.2,
    "c_1": 200.0, "c_2": 150.0,
    "p_1": 0.01, "p_2": 0.008,
    "e_1": 0.08, "e_2": 0.07,
    "x0" : 100.0, "y0" : 80.0, "z0" : 20.0,
    "t_span": (0,200),
    "dt" : 0.01, "d": 0.15,
    "method": "RK3"}
```

✓

Time stepping:

(f) Create a python function that integrates the system of ODEs using the settings and initial conditions as arguments. The function should perform integration for all three methods (according to the option `method`) and return the resulting times (t) and system states $S(t)$ as arrays.

In [ ]:
```python
def solve_ODE(d):
    """
    Solves a system of first-order ODEs using a specified numerical me
    provided in the dictionary d.
    Inputs:
        d (dict): Dictionary containing the parameters for solving the
    Outputs:
        time (array): Array of time values.
        sol (array): Solution array containing the computed states at
    """
    # Define the evaluation time
    dt = d["dt"]
    time = np.arange(d["t_span"][0], d["t_span"][1]+dt, dt)fontsize =

    #Initialize the solution vector
    s_i1 = np.zeros((len(time), 3))

    # Add intial conditions
    s0 = np.array([[d["x0"]], [d["y0"]], [d["z0"]]])
    s_i1[0, :] = s0.T # Assign initial conditions to the first row

    # Compute the solution using the trapezoidal method
    sol = d["method"](d, time, s_i1)

    return time, sol
```

-0.25

This needs to be a new line.

✓

✓

Plotting function:

(g) Create a python function that takes the times (t) and system states $S(t)$ as arguments and returns a 3-panel figure showing the evolution of each species (x, y, and z) in the system as a function of time, t.

In [ ]:

```python
def plot_species(t, s):
    """
    This function generates three subplots comparing the evolution of
    populations using three different integration methods.
    Inputs:
        t (array): Time points corresponding to the computed solutions
        s (list of array): A list containing solution arrays from the
                           methods. Each array has shape (len(t), 3),
                           represent [Prey 1 (x), Prey 2 (y), Predato
    Output:
        Displays the plots.
    """

    fig, ax = plt.subplots(1, 3, figsize=(25, 4))
    ax1, ax2, ax3 = ax.flatten()

    # Trapezoidal method
    ax1.plot(t, s[0][:, 0], color = "skyblue", label="Prey 1")
    ax1.plot(t, s[0][:, 1], color = "yellowgreen", label="Prey 2")
    ax1.plot(t, s[0][:, 2], color = "red", label="Predator")

    ax1.grid(linestyle = "--", alpha = 0.3 )
    ax1.legend(frameon = True, loc = 1, prop={'size': 13})
    ax1.set_xlabel("Time")
    ax1.set_ylabel("Population")
    ax1.set_title("Trapezoidal Method")

    # RK3 method
    ax2.plot(t, s[1][:, 0], color = "skyblue", label="Prey 1")
    ax2.plot(t, s[1][:, 1], color = "yellowgreen", label="Prey 2")
    ax2.plot(t, s[1][:, 2], color = "red", label="Predator")

    ax2.grid(linestyle = "--", alpha = 0.3 )
    ax2.legend(frameon = True, loc = 1, prop={'size': 13})
    ax2.set_xlabel("Time")
    ax2.set_ylabel("Population")
    ax2.set_title("RK3 Method")

    # B_RK3 method
    ax3.plot(t, s[2][:, 0], color = "skyblue", label="Prey 1")
    ax3.plot(t, s[2][:, 1], color = "yellowgreen", label="Prey 2")
    ax3.plot(t, s[2][:, 2], color = "red", label="Predator")

    ax3.grid(linestyle = "--", alpha = 0.3 )
    ax3.legend(frameon = True, loc = 1, prop={'size': 13})
    ax3.set_xlabel("Time")
    ax3.set_ylabel("Population")
    ax3.set_title("Butcher's RK Method")

    plt.show()
```

The two plotting functions are a nice solution for comparing the three methods.

In [ ]:
```
def plot_species_CM(t, s):
    """
    This function generates three subplots comparing three different
    the evolution of prey and predator populations.
    Inputs:
        t (array): Time points corresponding to the computed solutions
        s (list of array): A list containing solution arrays from the
                           methods. Each array has shape (len(t), 3),
                           represent [Prey 1 (x), Prey 2 (y), Predato
    Output:
        Displays the plots.
    """

    fig, ax = plt.subplots(1, 3, figsize=(25, 4))
    ax1, ax2, ax3 = ax.flatten()

    # Trapezoidal method
    ax1.plot(t, s[0][:, 0], color = "skyblue", label="Trapeziodal")
    ax1.plot(t, s[1][:, 0], color = "blue", linestyle = "--", label="P
    ax1.plot(t, s[2][:, 0], color = "darkcyan", linestyle = "-.", labe

    ax1.grid(linestyle = "--", alpha = 0.3 )
    ax1.legend(frameon = True, loc = 1, prop={'size': 13})
    ax1.set_xlabel("Time")
    ax1.set_ylabel("Population")
    ax1.set_title("Prey 1")
    ax1.set_ylim(0, 160)

    # RK3 method
    ax2.plot(t, s[1][:, 1], color = "yellowgreen", label="Trapeziodal"
    ax2.plot(t, s[1][:, 1], color = "lime", linestyle = "--", label="P
    ax2.plot(t, s[2][:, 1], color = "darkgreen", linestyle = "-.", lab

    ax2.grid(linestyle = "--", alpha = 0.3 )
    ax2.legend(frameon = True, loc = 1, prop={'size': 13})
    ax2.set_xlabel("Time")
    ax2.set_ylabel("Population")
    ax2.set_title("Prey 2")
    ax2.set_ylim(0, 160)

    # B_RK3 method
    ax3.plot(t, s[0][:, 2], color = "red", label="Trapeziodal")
    ax3.plot(t, s[1][:, 2], color = "coral", linestyle = "--", label="
    ax3.plot(t, s[2][:, 2], color = "firebrick", linestyle = "-.", lab

    ax3.grid(linestyle = "--", alpha = 0.3 )
    ax3.legend(frameon = True, loc = 1, prop={'size': 13})
    ax3.set_xlabel("Time")
    ax3.set_ylabel("Population")
    ax3.set_title("Predator")
    ax3.set_ylim(0, 160)

    plt.show()
```

Simulation:

(h) Call your time-stepping and plotting functions to run and display the results of three simulations (one per integration method), using the default settings and initial conditions

In [ ]:
```
# Call the function for simulating the three methods

d["method"] = trapezoidal_E # Define the method
t, s1 = solve_ODE(d)

d["method"] = RK3 # Define the method
_, s2 = solve_ODE(d)

d["method"] = B_RK # Define the method
_, s3 = solve_ODE(d)
```

In [ ]:
```
# Put the results together in array

solution = np.array([s1, s2, s3])
```

In [ ]:
```
# Call the function for plotting the simulations (three different meth

plot_species(t, solution)
```

In [ ]:
```
# Call the function for plotting the comparison of teh three methods

plot_species_CM(t, solution)
```

According to the generated figures, the system reaches an equilibrium point in the population of prey 1, prey 2, and the predator. Approximately in 40 units of time, the number of individuals of each species remains constant until the end of the simulation.

2. Dynamical systems and equilibrium (5 points)

Python class:

(a) Reorganise all your code from problem 1 into a single python class that contains attributes and methods. The settings and initial conditions should be attributes and all the python functions should become methods. Add a method to compute L_2-norm errors.

The $L_2 - norm$ error is calculated with the method `L2_error()` within the class `PreyPredator` and is definen as follows:

$$e_m = \frac{1}{n}\sqrt{\left(\sum_{i=1}^{n}(x_{r,i}(t) - x_{a,i}(t))^2\right)},$$

where x_r is the reference solution computed with the explicit Runge-Kutta method of order 8 (DOP853) using scipy package, x_a stands for aproximate solution computed with the methods implemented in this notebook, and n is the length of the evaluation time array.

In [ ]:

```python
class PreyPredator:
    """
    A class for solving a system of first-order ordinary differential
    modeling a prey-predator system using various numerical methods.

    Attributes:
        d (dict): Dictionary containing the system parameters such as
                        coefficients, and time step.
        method_name (str): The numerical method chosen to solve the OD
                            (Trapezoidal Euler -> "trapeziondal_E",
                            Third-order Runge-Kutta method-> "RK3",
                            Butcher's Runge-Kutta method -> "B_RK").
    Author: Alan Palma
    """

    def __init__(self, d):
        """
        Initializes the PreyPredator system with given d (parameters)
        """
        self.ic = d

        # Compute the reference solution when initializing

        _ , sol = self.DOP853()
        self.sol_ref = sol

    def select_method(self, method_name):

        # Methods available
        methods = {"trapezoidal_E" : self.trapezoidal_E,
                   "RK3" : self.RK3,
                   "B_RK" : self.B_RK}

        # Assign the methood in base of the entry
        if method_name in methods:
            self.ic["method"] = methods[method_name]
        elif method_name == "":
            raise ValueError("Specify a method")
        else:
            raise ValueError(f" {method_name} is not available")

    def F(self, t, s):
        """
        Function that contains the slope of a system of ODEs of first
        acoording to the slope matrix should be arranged this way s(t)
        Inputs:
            t(float): time variable.
            s(array):  state vector of the system.
            d(dic): dictionary containing the parameters of the syster
        Outputs:
            slope(float): array of the computed slope of the system.
        """
        d = self.ic # Access stored dictionary

        # Slope matrix based on the system of ODEs and paremeters
        m = np.array([
            [d["g_1"], -(d["g_1"]*s[0]**2)/(d["c_1"]*s[1]), -d["p_1"]*
            [d["g_2"]*(s[1]/s[0]) , -(d["g_2"]/d["c_2"])*s[1], -d["p_2
            [d["e_1"]*d["p_1"]*s[2] , d["e_2"]*d["p_2"]*s[2] , -d["d"]
            ])
```

```
        # Compute the slope vector
        slope = np.dot(m, s)

        return slope


    def trapezoidal_E(self, h, t, sol):
        """
        Solves an ODE using the explicit trapezoidal Euler method.
        This function implements a predictor-corrector scheme based or
        to approximate the solution of a first-order differential equa
        Inputs:
            h (float): Step size.
            t (array): Time points where the solution is computed.
            sol (array): Array to store the solution.
        Outputs:
            sol (array): Updated array with the numerical solution of
        """
        F = self.F # Access to the method containing slope function

        for j in range(0, len(t) - 1):
            sol[j + 1] = sol[j] + h*F(t[j], sol[j]) # Predictor step
            sol[j + 1] = sol[j] + h*(F(t[j], sol[j]) + F(t[j + 1], so

        return sol


    def RK3(self, h, t, sol):
        """
        Solves an ODE using the third-order Runge-Kutta (RK3) method.
        Inputs:
            h (float): Step size.
            t (array): Time points where the solution is computed.
            sol (array): Array to store the solution.
        Outputs:
            sol (array): Updated array with the numerical solution of
        """
        F = self.F # Access to the method containing slope function

        for j in range(0, len(t) - 1):

            # Compute RK3 intermediate slopes
            k1 = F(t[j], sol[j])  # Slope 1 -> k1
            k2 = F(t[j] + h/2, sol[j] + h*k1/2) # Slope 2 -> k2
            k3 = F(t[j] + h, sol[j] + h*(k2)) # Slope 3 -> k3

            # Compute the weighted sum of slopes for the final approxi
            sol[j + 1] = sol[j] + h*(k1 + 2.*k2 + k3)/4.

        return sol


    def B_RK(self, h, t, sol):
        """
        Solves an ODE using the Butcher's Runge-Kutta (B_RK) method.
        Inputs:
            h (float): Step size.
            t (array): Time points where the solution is computed.
            sol (array): Array to store the solution.
        Outputs:
            sol (array): Updated array with the numerical solution of
```

```
        """
        F = self.F # Access to the method containing slope function
        for j in range(0, len(t) - 1):

            # Compute BRK intermediate slopes
            k1 = F(t[j], sol[j]) # Slope 1 -> k1
            k2 = F(t[j] + h/4, sol[j] + h*k1/4) # Slope 2 -> k2
            k3 = F(t[j] + h/4, sol[j] + h*k1/8 + h*k2/8) # Slope 3 ->
            k4 = F(t[j] + h/2, sol[j] - h*k2/2 + h*k3) # Slope 3 -> k3
            k5 = F(t[j] + 3*h/4, sol[j] + 3*h*k1/16 + 9*h*k4/16) # Slo
            k6 = F(t[j] + h, sol[j] - 3*h*k1/7 + 2*h*k2/7 + 12*h*k3/7

            # Compute the weighted sum of slopes for the final approxi
            sol[j + 1] = sol[j] + h*(7*k1 + 32*k3 + 12*k4 + 32*k5 + 7*

        return sol


    def solve_ODE(self, method_name = "RK3"):
        """
        Solves a system of first-order ODEs using a specified numerica
        provided in the dictionary d.
        Inputs:
            d (dict): Dictionary containing the Butcher's rungue kuta
        Outputs:
            time (array): Array of time values.
            sol (array): Solution array containing the computed states
        """

        d = self.ic # Access stored dictionary
        self.select_method(method_name)

        # Define the evaluation time
        dt = d["dt"]
        time = np.arange(d["t_span"][0], d["t_span"][1]+dt, dt)

        #Initializate the solution vector
        s_i1 = np.zeros((len(time), 3))

        # Add intial conditions
        s0 = np.array([[d["x0"]], [d["y0"]], [d["z0"]]])
        s_i1[0, :] = s0.T # Assign initial conditions

        # Compute the solution using the trapezoidal method
        sol = d["method"](dt, time, s_i1)

        return time, sol

    def DOP853(self):
        """
        Computes the reference solution using Runge-Kutta of order 5 t
        Output:
            time (array): time points where the reference solution is
            sln.y (array): Computed reference solution.
        """
        d = self.ic # Access stored dictionary

        # Define the evaluation time
        dt = d["dt"]
        time = np.arange(d["t_span"][0], d["t_span"][1]+dt, dt)
```

```
        sln = solve_ivp(self.F, [time[0], time[-1]] , [d["x0"], d["y0"
                method= "DOP853", t_eval = time, rtol = 1e-10, atol =

        return time, sln.y

    def L2_error(self, sol_s):
        """
        Computes the L2-norm error between the reference solution and
        Input:
            sol_s(array): Solution array obtained from a numerical met
        Output:
            error(array): L2-norm error for each equation solution [e_
        """

        # Compute the squared differences and sum along each time step
        s = np.sum((self.sol_ref - sol_s.T)**(2), axis=1)

        # Compute the square root of the mean squared error
        error = np.sqrt(s)/self.sol_ref.shape[1]

        return error
```

Error analysis:

(b) Call the methods from your python class above to run 15 simulations (5 simulations per integration method) for 5 decreasing values of the time step size, `dt` (i.e. h). Then, compute the L_2-norm errors for all these 15 runs and report the results in a single figure with the L_2-norm errors in the Y axis and h in the X axis. Which method produces the most accurate results?

In [ ]:
```python
def simulation_dt(dt_arr):
    """
    Runs a numerical simulation for different time step sizes using mu
    and computes the L2-norm error for each method.
    Input:
        dt_arr(array): Array of time step sizes to evaluate.
    Ouput:
        t_m1(array): Time points used in the simulation.
        e(array): L2-norm errors for each method and solution
                    The resulting array should be indexed as follows e
                    i1 -> points the step size
                    i2 -> points the method
                    i3 -> points the population (x(t), y(t), z(t)).
    """

    sol_me = []
    e = []

    for h in dt_arr:

        d["dt"] = h

        pP = PreyPredator(d)
        t_m1, sol_m1 = pP.solve_ODE("trapezoidal_E")
        _, sol_m2 = pP.solve_ODE("RK3")
        _, sol_m3 = pP.solve_ODE("B_RK")

        sol_me.append([sol_m1, sol_m2, sol_m3])

        e1 = pP.L2_error(sol_m1)
        e2 = pP.L2_error(sol_m2)
        e3 = pP.L2_error(sol_m3)

        e.append(np.array([e1, e2, e3]))

    return t_m1, np.array(e)
```

In [ ]:
```python
# Call the function to compute all simulations

#dt_list = [0.01, 0.008, 0.005, 0.003, 0.001, 0.0005]
dt_list = [0.01, 0.04, 0.07, 0.09, 0.1, 0.12]

_, error = simulation_dt(dt_list)
```

In [ ]:
```
# Plotting

fig, ax = plt.subplots(1, 3, figsize=(19,4))
ax1, ax2, ax3 = ax.flatten()

ax1.plot(dt_list, np.log10(error[:, 0, 0]), marker = "o", color = "red
ax1.plot(dt_list, np.log10(error[:, 1, 0]), marker = "d", color = "blu
ax1.plot(dt_list, np.log10(error[:, 2, 0]), marker = "s", color = "gre

ax1.grid(linestyle = "--", alpha = 0.3 )
ax1.legend(frameon = True, fontsize = 11)
ax1.set_xlabel(r"h")
ax1.set_ylabel(r"$log_{10}$Error")
ax1.set_title(r"x(t) $L_2$-norm error (Prey 1)")

ax2.plot(dt_list, np.log10(error[:, 0, 1]), marker = "o", color = "red
ax2.plot(dt_list, np.log10(error[:, 1, 1]), marker = "d", color = "blu
ax2.plot(dt_list, np.log10(error[:, 2, 1]), marker = "s", color = "gre

ax2.grid(linestyle = "--", alpha = 0.3 )
ax2.legend(frameon = True, fontsize = 11)
ax2.set_xlabel(r"h")
ax2.set_ylabel(r"$log_{10}$Error")
ax2.set_title(r"y(t) $L_2$-norm error (Prey 2)")

ax3.plot(dt_list, np.log10(error[:, 0, 2]), marker = "o", color = "red
ax3.plot(dt_list, np.log10(error[:, 1, 2]), marker = "d", color = "blu
ax3.plot(dt_list, np.log10(error[:, 2, 2]), marker = "s", color = "gre

ax3.grid(linestyle = "--", alpha = 0.3 )
ax3.legend(frameon = True, fontsize = 11)
ax3.set_xlabel(r"h")
ax3.set_ylabel(r"$log_{10}$Error")
ax3.set_title(r"z(t) $L_2$-norm error (Predator)")

plt.tight_layout()

plt.show()
```

As expected, the Butcher's Runge-Kutta (B. RK) method demonstrates greater accuracy compared to the reference solution, as it has the lowest $\log_{10}$ error. On the other hand, the method with the lowest accuracy is the trapezoidal method, a second-order method.

Equilibrium conditions:

(c) Write down the equilibrium condition for the system, $S(t)$. Then, create a python function that uses sympy to study the equilibrium populations of x, y, and z for a range of values of the natural death rate of the predator, d. The function should accept a range of d values and return the equilibrium populations for all the d values in the range. Ensure that the solutions are filtered to exclude extinction cases.

In the equilibrium:

$$F(S(t)) = 0 \Rightarrow \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{z}(t) \end{bmatrix} = 0,$$

then

$$\frac{dx}{dt} = g_1\, x \left(1 - \frac{x}{c_1}\right) - p_1\, x\, z = 0,$$
$$\frac{dy}{dt} = g_2\, y \left(1 - \frac{y}{c_2}\right) - p_2\, y\, z = 0,$$
$$\frac{dz}{dt} = e_1\, p_1\, x\, z + e_2\, p_2\, y\, z - d\, z = 0.$$

Using the slope matrix $M(S(t))$:

$$M * S(t) = 0$$

For excluding the extinction cases the trivial solution ($S(t) = 0$) should be avoided, and any population can be 0 ($x(t) \neq 0$, $y(t) \neq 0$, $z(t) \neq 0$).

In [ ]:
```python
def eq_system(d_val_list):
    """
    Function to compute the equilibrium populations (x, y, z) of a pre
    for a given range of predator death rate values (d).
    Inputs:
        d_val_list (array): Different values of the natural death rate
    Outputs:
        d_val_list_new (array): Array of d values that reach valid equ
        sol_list (array): Array of corresponding equilibrium populatio
    """

    sol_list = [] # Empty list to store all solutions
    d_val_list_new = [] # Empty list to store correct values for d

    for d_val in d_val_list:

        # Define the parameters of the system in a dictionary
        d = {
            "g_1": 1.0, "g_2": 1.2,
            "c_1": 200.0, "c_2": 150.0,
            "p_1": 0.01, "p_2": 0.008,
            "e_1": 0.08, "e_2": 0.07,
            "x0" : 100.0, "y0" : 80.0, "z0" : 20.0,
            "t_span": (0,200), "dt" : 0.01,
            "d": 0.15,
            "method": "RK3"}

        d["d"] = d_val

        # Define Symbols
        x, y, z = sp.symbols('x y z')

        # Create the slope vector with them
        s = [x, y, z]

        # Compute the solution using the slope function defined before
        sol = sp.solve(F(d,_, s), (x, y, z))

        # For loop to filtering the solutions
        for s in sol:
            if s[0] > 0 and s[1] > 0 and s[2] >0:

                # After checking that allways there is a single soluti

                sol_list.append(np.array(s, dtype = float))  # Append
                d_val_list_new.append(d_val)# Append the corresponding

    return np.array(d_val_list_new), np.array(sol_list)
```

(d) After computing equilibrium solutions for multiple values of the predator mortality d, you should create a high-quality figure of the equilibrium populations of x, y, and z (on the Y axis) versus d (on the x-axis), and label what happens in the parameter regions outside of equilibrium conditions.

In [ ]:
```
# Define the list values for d

d0 = 0.0 # Initial d
df = 0.50 #Final d
dn = 500 # Number of points

# Generate the list array
d_list = np.linspace(d0, df, dn)
```

In [ ]:
```
# Call the function to get solutions for equilibrium

d_list_new, sol_eq = eq_system(d_list)
```

In [ ]:
```
# Plotting the equilibrium population results

fig, axs = plt.subplots(1, 1, figsize=(8,4))

axs.plot(d_list_new, sol_eq.T[0], color = "skyblue", label=r"x(t) $\ri
axs.plot(d_list_new, sol_eq.T[1], color = "yellowgreen", label=r"y(t)
axs.plot(d_list_new, sol_eq.T[2], color = "red", label=r"z(t) $\righta
axs.vlines(d_list_new[0], -1, 200, color = "hotpink", linestyle = "--'
axs.vlines(d_list_new[-1], -1, 200, color = "hotpink", linestyle = "--

axs.grid(linestyle = "--", alpha = 0.3 )
axs.set_xlim(d_list_new[0] - 0.03, d_list_new[-1] + 0.03)
axs.set_ylim(0, 200)
axs.set_title("Equilibrium Population vs. \n Predator Mortality Rate",
axs.set_xlabel("Predator Mortality (d)", fontsize = 11)
axs.set_ylabel("Equilibrium Population", fontsize = 11)
axs.legend(frameon = True, fontsize = 10)

plt.show()
```

(e) Based on your analysis, for what range of d values the system reaches equilibrium? What happens outside of that range? Call your class again to run and compare equilibrium versus extinction scenarios using 3 different values of d.

In [ ]:
```
# Report range of values for d:

print("The values of d to reach the equilibrium are within %.5f, and %
```

- According to the generated results, outside the equilibrium limit, a specie becomes almost extinct. The range of values for d are within $\approx [0.028, 0.243]$
- On the left, when there is an overpopulation of predators (low mortality rate), the population of prey 1 and prey 2 is lower than in other cases, and prey 1 becomes almost extinct. They become extinct outside the equilibrium range.
- On the right, the predators become almost extinct, and prey 1 and prey 2 reach their highest population levels.
- When there are few predators, both prey populations reach their carrying capacities: 200 for prey 1 and 150 for prey 2.

In [ ]:
```python
# Define parameters

d = {
    "g_1": 1.0, "g_2": 1.2,
    "c_1": 200.0, "c_2": 150.0,
    "p_1": 0.01, "p_2": 0.008,
    "e_1": 0.08, "e_2": 0.07,
    "x0" : 100.0, "y0" : 80.0, "z0" : 20.0,
    "t_span": (0,200), "dt" : 0.01,
    "d": 0.15,
    "method": "RK3"}
```

In [ ]:
```python
# Equilibirum scenarios

d_eq_list = [0.1, 0.15, 0.2]
sol_eq_list = []

for d_val in d_eq_list:
    d["d"] = d_val
    pP1 = PreyPredator(d)
    t_eq, sol_eq = pP1.solve_ODE("B_RK")
    sol_eq_list.append(sol_eq)

sol_eq_list = np.array(sol_eq_list)
```

In [ ]:
```python
# No equilibirum scenarios

d_n_eq_list = [0.0001, 0.01, 0.5]
sol_n_eq_list = []

for d_n_val in d_n_eq_list:
    d["d"] = d_n_val
    pP2 = PreyPredator(d)
    t_n_eq, sol_n_eq = pP2.solve_ODE("B_RK")
    sol_n_eq_list.append(sol_n_eq)

sol_n_eq_list = np.array(sol_n_eq_list)
```

In [ ]:

```python
def plot_species_eq(t, s, d_list, name):
    """
    This function generates three subplots comparing the evolution of
    populations for three different values of d (predator mortality ra
    Inputs:
        t (array): Time points corresponding to the computed solutions
        s (array): Array containing solution arrays from the three dif
                   methods. Each array has shape (len(t), 3),
                   represent [Prey 1 (x), Prey 2 (y), Predato
        d_list (array): Array containing the values for predtor mortal
        name (str): name for plotting the title (equilibirum or no equ
    Output:
        Displays the plots.
    """

    fig, ax = plt.subplots(1, 3, figsize=(19, 5), sharey=True)
    ax1, ax2, ax3 = ax.flatten()

    # Trapezoidal method
    ax1.plot(t, s[0][:, 0], color = "skyblue", label="Prey 1")
    ax1.plot(t, s[0][:, 1], color = "yellowgreen", label="Prey 2")
    ax1.plot(t, s[0][:, 2], color = "red", label="Predator")

    ax1.grid(linestyle = "--", alpha = 0.3 )
    ax1.legend(frameon = True, loc = 1, prop={'size': 10})
    ax1.set_xlabel("Time", fontsize = 13)
    ax1.set_ylabel("Population", fontsize = 13)
    ax1.set_title(f"d = {d_list[0]}")

    # RK3 method
    ax2.plot(t, s[1][:, 0], color = "skyblue", label="Prey 1")
    ax2.plot(t, s[1][:, 1], color = "yellowgreen", label="Prey 2")
    ax2.plot(t, s[1][:, 2], color = "red", label="Predator")

    ax2.grid(linestyle = "--", alpha = 0.3 )
    ax2.legend(frameon = True, loc = 1, prop={'size': 10})
    ax2.set_xlabel("Time", fontsize = 13)
    #ax2.set_ylabel("Population", fontsize = 13)
    ax2.set_title(f"d = {d_list[1]}")

    # B_RK3 method
    ax3.plot(t, s[2][:, 0], color = "skyblue", label="Prey 1")
    ax3.plot(t, s[2][:, 1], color = "yellowgreen", label="Prey 2")
    ax3.plot(t, s[2][:, 2], color = "red", label="Predator")

    ax3.grid(linestyle = "--", alpha = 0.3 )
    ax3.legend(frameon = True, loc = 1, prop={'size': 10})
    ax3.set_xlabel("Time", fontsize = 13)
    #ax3.set_ylabel("Population", fontsize = 13)
    ax3.set_title(f"d = {d_list[2]}")

    fig.suptitle(f"Population Dynamics for Different Predator Mortalit
    plt.tight_layout()

    plt.show()
```

In [ ]:
```
# Plotting within equilibrium conditions

plot_species_eq(t_eq, sol_eq_list, d_eq_list, "Equilibrium Conditions"

# Plotting out equilibrium conditions

plot_species_eq(t_n_eq, sol_n_eq_list, d_n_eq_list, "Non-equilibrium C
```

- As expected, the population dynamics within the equilibrium range of death rate show that no species go extinct. However, it is observed that as the predator population declines toward the equilibrium limit, the prey population increases, approaching their carrying capacities and maintaining this level over time.
- In the non-equilibrium case, when the predator's death rate is very low, the population of prey 1 declines to zero, meaning prey 1 goes extinct. On the other hand, when the mortality rate exceeds the limit, the predator population reaches zero, while the prey populations grow to their carrying capacities and remain at this state over time.
- Notice that when the death rate of prey is extremely low (d = 0.0001), the populations of prey 1 and prey 2 are nearly zero, while the predator population naturally increases.

3. Quantum Harmonic Oscillator (7 points)

The Schrödinger equation for the quantum harmonic oscillator is:

$$-\frac{\hbar^2}{2m}\frac{d^2\psi}{dx^2} + \frac{1}{2}m\omega^2 x^2\psi = E\psi$$

It can be rewritten, in terms of a new variable, $\xi \equiv \sqrt{\frac{m\,\omega}{\hbar}}x$, as follows:

$$\frac{d^2\psi}{d\xi^2} = \left(\xi^2 - K\right)\psi$$

where $K \equiv \frac{2E}{\hbar\omega}$ is the energy in units of $\frac{1}{2}\hbar\omega$.

Order reduction and slope function:

(a) Reduce the above ODE to first order. Write down the resulting slope function.

Let's define the slope vector $S(x)$:

$$S(\xi) = \begin{bmatrix} \psi(\xi) \\ \psi'(\xi) \end{bmatrix}$$

Taking the derivative of $S(x)$ and substituting

$$\frac{dS(\xi)}{d\xi} = \begin{bmatrix} \psi'(\xi) \\ \psi''(\xi) \end{bmatrix} = \begin{bmatrix} \psi'(\xi) \\ (\xi^2 - K)\psi(\xi) \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} S(\xi) = \begin{bmatrix} 0 & 1 \\ (\xi^2 - K) & 0 \end{bmatrix} S(\xi)$$

Then the slope function is

$$F(S(\xi)) = \begin{bmatrix} 0 & 1 \\ (\xi^2 - K) & 0 \end{bmatrix} S(\xi)$$

In [ ]:

```python
def F_ho(xi, s, k_guess):
    """
    Computes the derivative (slope function) for the Schrödinger equat
    quantum harmonic oscillator.

    Inputs:
    xi(float): The independent variable.
    s (array-like):
        The state vector, where:
        s[0] = psi(xi)  -> Wave function value
        s[1] = psi'(xi) -> First derivative of the wave function
    k_guess(float): The trial energy value for the shooting method.

    Output:
    slope (array-like): The derivative of the state vector.
    """

    #Define the slope matrix for the ODE

    M = np.array(
        [[0, 1],
         [xi**2-k_guess, 0]], dtype = object
    )

    # Compute the slope
    slope = np.dot(M,s)

    return slope
```

Shooting method class:

(b) Carefully read the tasks (c-g) below and design a suitable python class with attributes and methods that solves the above ODE using **the shooting method** with the slope computed in (a). You may use scipy integrators; there is no need to design your own integrators in this problem.

In [ ]:

```python
class HarmoniOscillatorSolve():
    """
    A class to solve the Schrödinger equation for the quantum harmonic
    using the shooting method.
    """
    def __init__(self, params, n = 0):
        """
        Initializes the solver for the quantum harmonic oscillator.

        Inputs:
        params (dict): Dictionary containing the solver parameters:
            - "dxi" (float): Step size for xi.
            - "xi_span" (list): Range of xi values (xi_min, xi_max).
        n (int): Quantum number defining the energy level (default is
        """
        # Set the parameters
        self.dxi = params.get("dxi")
        self.xi_span_i = params.get("xi_span")[0]
        self.xi_span_f = params.get("xi_span")[1]
        self.n = n

        # Set initial values depending on the parity of n
        if n % 2 == 0:
            self.initial_values = [1, 0]
        else:
            self.initial_values = [0, 1]

    def F_ho(self, xi, s, k_guess):
        """
        Computes the derivative (slope function) for the Schrödinger e
        quantum harmonic oscillator.

        Inputs:
        xi(float): The independent variable.
        s (array-like):
            The state vector, where:
            s[0] = psi(xi)  -> Wave function value
            s[1] = psi'(xi) -> First derivative of the wave function
        k_guess(float): The trial energy value for the shooting metho

        Output:
        slope (array-like): The derivative of the state vector.
        """
        #Define the slope matrix for the ODE

        M = np.array(
            [[0, 1],
                [xi**2-k_guess, 0]], dtype = object)

        # Compute the slope
        slope = np.dot(M,s)

        return slope

    def solve_ODE_ho(self, k):
        """
        Solves the Schrödinger equation using an initial guess for the

        Inputs:
        k (float): Trial energy value.
        Outputs:
```

```
        xi_range (array): The range of xi values.
        psi (array): The computed wave function values.
        """

        # Define the evaluation range
        dxi = self.dxi
        xi_range = np.arange(self.xi_span_i, self.xi_span_f + dxi, dxi

        # Define the initial conditions
        ivs = self.initial_values

        # Compute solution:
        psi = solve_ivp(self.F_ho, [self.xi_span_i, self.xi_span_f], i
                        method= "DOP853", t_eval = xi_range,  args=(k,

        return xi_range, np.around(psi.y[0], 5) # Round the final solu

    def shooting_function(self, k_guess):
        """
        Computes the wave function at the boundary xi_max for a given

        Input:
        k_guess (float): Trial energy value.
        Output:
        psi (float): The wave function at the boundary xi_max.
        """
        # Compute the solution for a given k_guess
        _, psi = self.solve_ODE_ho(k_guess)

        return psi[-1]  # Returns psi(xi_max)

    def optimization(self):
        """
        Finds the energy eigenvalue using the shooting method.

        Output:
        k_new (float): Optimized energy value.
        """

        # Define the intial guess for k acoording to energy level n
        k_guess = 2*self.n+1

        # Compute the new k value by optimization
        k_new = opt.fsolve(self.shooting_function, k_guess)

        return k_new

    def plot_psi_solutions(self, n, k, xi_range, psi_sol, xlim = None,
        """
        Plots the computed wave function along with slight perturbatio
        to visualize shooting.

        Inputs:
        n (int): Energy Level.
        k (float): Computed energy.
        xi_range (array): The range of xi values.
        psi_sol (array): The computed wave function values.
        """

        # Define the values of k
```

```
        k_1 = k + 1.*10**(-8.5)
        k_2 = k - 1.*10**(-8.7)
        k_3 = k + 1.*10**(-9.0)


        # Compute the solutions for this values

        _, psi_1 = self.solve_ODE_ho(k_1)
        _, psi_2 = self.solve_ODE_ho(k_2)
        _, psi_3 = self.solve_ODE_ho(k_3)


        # Plot the ground state solution

        fig, ax = plt.subplots(1, 3, figsize = (20, 4))
        ax1, ax2, ax3 = ax.flatten()

        ax1.plot(xi_range, psi_sol / np.linalg.norm(psi_sol), color =
        ax1.plot(xi_range, psi_1 / np.linalg.norm(psi_1), linestyle =

        ax1.grid(linestyle = "--", alpha = 0.3)
        ax1.set_xlabel(r"$\xi$")
        ax1.set_ylabel(f"$\psi_{n}$")
        ax1.legend(frameon = True, loc = 3, fontsize = 11)
        ax1.set_ylim(-0.1, 0.1)


        ax2.plot(xi_range, psi_sol / np.linalg.norm(psi_sol), color =
        ax2.plot(xi_range, psi_2 / np.linalg.norm(psi_2), linestyle =

        ax2.grid(linestyle = "--", alpha = 0.3)
        ax2.set_xlabel(r"$\xi$")
        ax2.set_ylabel(f"$\psi_{n}$")
        ax2.legend(frameon = True, loc = 3, fontsize = 11)
        ax2.set_ylim(-0.1, 0.1)


        ax3.plot(xi_range, psi_sol / np.linalg.norm(psi_sol), color =
        ax3.plot(xi_range, psi_3 / np.linalg.norm(psi_3), linestyle =

        ax3.grid(linestyle = "--", alpha = 0.3)
        ax3.set_xlabel(r"$\xi$")
        ax3.set_ylabel(f"$\psi_{n}$")
        ax3.legend(frameon = True, loc = 3, fontsize = 11)
        ax3.set_ylim(-0.1, 0.1)

        plt.suptitle(f"Wave Function: Quantum Harmonic Oscillator (n=

        plt.tight_layout()

        if xlim and ylim:

            ax1.set_xlim(6.9, 7.0)
            ax2.set_xlim(6.9, 7.0)
            ax3.set_xlim(6.9, 7.0)

            ax1.set_ylim(-0.002, 0.002)
            ax2.set_ylim(-0.002, 0.002)
            ax3.set_ylim(-0.002, 0.002)

        plt.show()
```

Tasks to be performed by your python class:

(c) Find the **ground state energy** of the harmonic oscillator, to six significant digits, by using **the shooting method**. That is, solve the above equation numerically, varying K until you get a wave function that goes to zero at large ξ. The appropriate boundary conditions for the ground state (and any even state) are $\psi(0) = 1, \psi'(0) = 0$.

In [ ]:
```
# Define the parameters
param = {"dxi": 0.01,
       "xi_span": [-7, 7]}

# Instance the class for the groud state solution (n = 0)
lvl_0 = HarmoniOscillatorSolve(param, n = 0)
k_0 = lvl_0.optimization() # Compute the new k value
xi_range_0, psi_0 = lvl_0.solve_ODE_ho(k_0) # Compute the solution
```

In [ ]:
```
# Plot the ground state solution

plt.figure(figsize = (10, 4))

plt.plot(xi_range_0, psi_0 / np.linalg.norm(psi_0), color = "springgre
plt.grid(linestyle = "--", alpha = 0.3)

plt.xlabel(r"$\xi$")
plt.ylabel(r"$\psi_0$")
plt.title("Normalized Harmonic Oscillator \n Ground State Solution")
plt.legend(frameon = True, fontsize = 11)

plt.show()
```

In [ ]:
```
# Repoting the result

print("The ground state energy is K = %.5f" % k_0[0])
```

Considering that K is expressed in units of $\frac{1}{2}\hbar\omega$, it should be dimensionless and equal to 1 for the ground state. Recall the eigenenergy solution of the quantum harmonic oscillator:

$$E_n = \hbar\omega\left(n + \frac{1}{2}\right),$$

then, for $n = 0$:

$$E_0 = \frac{1}{2}\hbar\omega.$$

(d) Make a few illustrative panels showing plots of the wave function for different values of K as it converges to the solution. What does the tail of the wave function does when the values are slightly above or below the correct solution?

In [ ]:
```
# Plot the solution for the ground state

lvl_0.plot_psi_solutions(0, k_0, xi_range_0, psi_0)
```

It can be observed that the wave function does not converge at the tails when varying the value of K. Instead, it increases or decreases rapidly depending on the case. This behavior can be explained in the context of the quantum harmonic oscillator. The energy levels are quantized, meaning each energy level "corresponds" to a specific wave function (solution). When the value of K deviates from the correct energy value, the ODE solver fails to converge to the correct solution. Instead of a smooth decay of the wave function at large ξ, the tail shows either rapid growth or decay. This behavior is consistent with the fact that K represents the energy, and an incorrect value for K leads to a non-physical solution. As will be seen in the following question, this behavior occurs for all excited states.

(e) Find the **first four excited state energies** (to six significant digits) for the harmonic oscillator, using **the shooting method**. For the first (and third) excited state you will need to set $\psi(0) = 0, \psi'(0) = 1$).

In [ ]:
```
# Empty list to store the solutions
psi_solutions = []

# Empty list to store the resultant k values
k_n_list = []

# Define the parameters
param = {"dxi": 0.01,
         "xi_span": [-7, 7]}

# For loop to compute the solutions for the first four exited states
for n in range(5):

    # Instance the class for the exited states
    lvl_n = HarmoniOscillatorSolve(param, n)

    # Compute the new k value
    k_n = lvl_n.optimization()

    # Compute the solution
    xi_range, psi_n = lvl_n.solve_ODE_ho(k_n)

    # Append the  k and the solution to the list
    k_n_list.append(k_n)
    psi_solutions.append(np.around(psi_n, 6))
```

(f) Make a few illustrative panels showing plots of the wave functions for different values of K as they converge to their respective solutions.

In [ ]:
```
# For loop to plot with the class method

for n in range(5):
    lvl_n.plot_psi_solutions(n, k_n_list[n], xi_range, psi_solutions[n
```

In [ ]:
```
# Plot the last exited state at the tails
n = 4
lvl_n.plot_psi_solutions(n, k_n_list[n], xi_range, psi_solutions[n], x
```

(g) Make a single plot showing the **harmonic potential** joinly with the **energy ladder** of the quantum harmonic oscillator (include the ground state plus the first four excited states that you calculated above, each with their **respective wave functions**). Consider the particle to be an electron and choose appropriate units for any physical parameters you may need.

In [ ]:
```
# Define the constants
m_e = constants.m_e # Mass of an electron in kg
w_e = 7.81e20 # Frequency of the harmonic oscillator in Hz
hbar = constants.hbar # Planck's constant in J s

# Define the harmonic oscillator potential
v_ho = lambda xi: 0.5*w_e*hbar*xi**2
```

In [ ]:
```
# Compute the energy ladder

E_n_list = []

for n in range(5):
    E_n = (n + 0.5)*hbar*w_e
    E_n_list.append(E_n)
```

In [ ]:
```
# Convert the xi array into a lenght array in m to map the potential e
x_range = xi_range_0 * (hbar/(m_e*w_e))**(1/2)

# Compute the potential energy
v_ho_x = v_ho(xi_range)
```

In [ ]:

```
# Define colors for wave fucuntion solutions
colors = ["springgreen", "fuchsia", "red", "purple", "orange", "black"

# Scaling factor for the schematic view of the wave functions
sfac = 10**(-12.2)

# Plot the wave functions

plt.figure(figsize = (10, 9))

plt.plot(x_range, v_ho_x, color = "deepskyblue", label = "V(x)") # Plo
plt.vlines(0, 0, 5.5e-13, color = "gray", linestyle = "--", linewidth

# For loop to plot all the wave functions together
for n in range(5):
    # Plot the wave function
    plt.plot(x_range, (psi_solutions[n] / np.linalg.norm(psi_solutions
    # Plot the energy levels
    plt.axhline (E_n_list[n], color = "gray", linestyle = "--", linewi
    plt.text(1.6e-12, E_n_list[n] + 1e-14, f"$n={n}$", fontsize = 16)

plt.xlabel("Position (m)", fontsize = 13)
plt.ylabel("Potential Energy (J)", fontsize = 13)
plt.title("Quantum Harmonic Oscillator \n Potential and Solutions")
plt.legend(loc = 2, frameon = True, fontsize = 10)

plt.xlim(-2*1.e-12, 2*1.e-12)
plt.ylim(0, 4.8e-13)

plt.show()
```