

Practical class: Numerical Methods for solving ODEs

In this exercise class, we will explore methods for solving ordinary differential equations (ODEs) using Python's powerful scientific computing libraries, NumPy and SciPy. These libraries provide robust tools for numerical computation and include pre-built functions for integrating ODEs efficiently.

We will begin by implementing simple numerical methods such as the Forward Euler method to gain a deeper understanding of the underlying principles. Then, we will leverage the capabilities of SciPy's initial conditions solvers.

The focus will be on the same model problems that were explained during the lecture:

- Population Growth,
- Pendulum,
- Chemical Reaction Systems and
- Astrophysical N-body Problem.

By the end of this session, you will have hands-on experience applying numerical techniques to solve ODEs and you will be introduced to these Python's scientific computing tools.

Exercise 1. Implementation of explicit methods for solving ODEs

Remainder of Runge-Kutta Methods from the lecture

In the slide of p83 it is explained how the Butcher tableau works to summarize the definition of a RK integration method. Given the ODE:

$$\frac{dy}{dt} = f(t, y),$$

we approximate $(y(t))$ in steps $(t^n \rightarrow t^{n+1})$ using the following procedure:

Stage Calculations

For each stage (i) (where $(i = 1, 2, \dots, s)$), compute:

$$s_i^n = f \left(t^n + c_i h, y^n + \Delta t_n \sum_{j=1}^s A_{ij} s_j^n \right),$$

where:

- $(\Delta t_n = t^{n+1} - t^n)$ is the step size.
- (s_i^n) is the stage evaluation at time $(t^n + c_i \Delta t_n)$, incorporating information from other stages through $(\sum_{j=1}^s A_{ij} s_j^n)$.

Final Update

After computing all (s_i^n) , update the solution (y^{n+1}) :

$$y^{n+1} = y^n + \Delta t_n \sum_{i=1}^s b_i s_i^n.$$

First of all import all the scientific computing packages that are needed for the next exercises.

```
In [1]: # Import numpy package and name it 'np'
import numpy as np

# Import the package Matplotlib for the visualizations (Already done!)
import matplotlib.pyplot as plt
```

Exercise 1.1: Implement the Forward Euler method (RK1) in the blank space of the function `solver_euler` (p84).

The Butcher tableau:

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

```
In [2]: # Definition of the Forward Euler method
def solver_euler(t, y, dt, f, args=[]):
    """
    Forward Euler solver for ordinary differential equations (ODEs).

    This function implements the Forward Euler method, a first-order numeric
    method for solving initial value problems of the form:
        dy/dt = f(t, y)

    Parameters:
    -----
    t : float
        The current time step value.
    y : float or array-like (in case is a vector)
```

```

        The current value of the dependent variable(s).
    dt : float
        The time step size for the integration.
    f : callable
        The function representing the ODE, with the signature f(t, y, *args)
    args : list, optional
        Additional arguments to pass to the function `f`.

Returns:
-----
float or array-like
    The estimated value of `y` at the next time step, computed with Forward Euler.
"""
# Write here your code ...
return y + dt * f(t, y, args)

```

Exercise 1.2: Implement a general Θ -2nd order method (RK2), considering $\Theta > 0$, in the blank space of the function `solver_theta_rk2` (p85).

The Butcher tableau:

$$\begin{array}{c|cc}
 0 & & \\
 \hline
 \Theta & \Theta & \\
 \hline
 & 1 - \frac{1}{2\Theta} & \frac{1}{2\Theta}
 \end{array}
 \quad \Theta\text{-RK 2nd order}$$

The matrix A and the vectors c and b^T are as follows:

$$c = \begin{bmatrix} 0 \\ \theta \end{bmatrix}, \quad A = \begin{bmatrix} 0 & 0 \\ \theta & 0 \end{bmatrix}, \quad b^T = \begin{bmatrix} 1 - \frac{1}{2\theta} \\ \frac{1}{2\theta} \end{bmatrix}$$

```

In [3]: def solver_theta_rk2(t, y, dt, f, args=[]):
        """
        General second order theta-scheme solver for ordinary differential equations.

        This function implements a general theta-scheme solver method, a second-order
        method for solving initial value problems of the form:
            dy/dt = f(t, y)

        The parameter Theta is chosen as an arbitrary positive number

        Parameters:
        -----
        t : float
            The current time step value.
        y : float or array-like (in case is a vector)
            The current value of the dependent variable(s).
        dt : float
            The time step size for the integration.
        f : callable

```

```

    The function representing the ODE, with the signature f(t, y, *args)
    args : list, optional
        Additional arguments to pass to the function `f`.
    theta : float
        The theta-scheme parameter of the method. Is an arbitrary positive float number.
        It is always the last element in the list args of arguments.

Returns:
-----
float or array-like
    The estimated value of `y` at the next time step
"""

theta = args[-1]

# Check that theta is a strictly positive float number
assert theta > 0.0, "The parameter theta is not a positive number."

# Write here your code ...

s1 = f(t, y, args[:-1]) # The last argument doesn't belong to the function f
s2 = f(t + theta * dt, y + theta * dt * s1, args[:-1]) # s2. Note that t is updated

# Assembling the vector s of the values of the function for stage s1 and s2
s = np.array([s1, s2])

# Assembling the vector b^T
bt = np.array([1.0 - (1 / (2. * theta)), 1 / (2. * theta)])

# Computing the final update of the method
return y + dt * np.dot(bt, s)

```

Exercise 1.3: Implement the Heun's 3rd order method (RK3) in the blank space of the function `solver_heun_rk3` (p87).

The Butcher tableau:

$$\begin{array}{c|ccc}
 0 & & & \\
 \frac{1}{3} & \frac{1}{3} & & \\
 \frac{2}{3} & 0 & \frac{2}{3} & \\
 \hline
 & \frac{1}{4} & 0 & \frac{3}{4}
 \end{array}
 \quad \text{Heun's 3rd order method}$$

[*] Determine the matrix A and vectors c y b^T .

```

In [4]: def solver_heun_rk3(t, y, dt, f, args=[]):
        """
        Heun's 3rd order method solver for ordinary differential equations (ODEs)

        This function implements the Heun's 3rd order method solver method, a
        method for solving initial value problems of the form:

```

```

dy/dt = f(t, y)

Parameters:
-----
t : float
    The current time step value.
y : float or array-like (in case is a vector)
    The current value of the dependent variable(s).
dt : float
    The time step size for the integration.
f : callable
    The function representing the ODE, with the signature f(t, y, *args)
args : list, optional
    Additional arguments to pass to the function `f`.

Returns:
-----
float or array-like
    The estimated value of `y` at the next time step
"""

# Write here your code ...

# Assembling the vector s of the values of the function
s1 = f(t, y, args)
s2 = f(t + 1./3 * dt, y + 1./3 * dt * s1, args)
s3 = f(t + 2./3 * dt, y + 2./3 * dt * s2, args)

# for stage s1, s2 and s3 respectively.
s = np.array([s1, s2, s3])

# Assembling the vector b^T
bt = np.array([0.25, 0, 0.25])

# Computing the final update of the method
return y + dt * np.dot(bt, s)

```

Population Growth Model

Implementation of the Bernoulli's population growth model:

- Formula: The model follows the equation:

$$dy/dt = \lambda y(1 - y/G)$$

where:

- λ is the growth rate.
- y is the current population size.
- G is the carrying capacity.
- Example: Example usage of the Bernoulli Population Growth Model

```

t = 0 # Initial time
y = 50 # Initial population
args = [0.1, 100] # Growth rate  $\lambda = 0.1$ , carrying capacity  $G = 100$ 

```

```

dydt = growth_bernoulli_model(t, y, args)
print("Rate of population growth:", dydt)

```

- Notes:
 - Ensure `args` contains exactly two parameters: `[λ , G]`.
 - If `y > G`, the rate of growth becomes negative, representing a decline in population due to overcapacity.

```

In [5]: def growth_bernoulli_model(t, y, args=[]):
        """
        Bernoulli Population Growth Model.

        This function represents the Bernoulli population growth model, a modification of the logistic model.
        It calculates the rate of change in a population (`y`) over time (`t`) given the growth rate parameter
        and carrying capacity.

        Parameters:
        -----
        t : float
            Time variable, representing the current point in time.
        y : float
            Current population size.
        args : list
            A list of parameters:
            - `args[0]` (float): Growth rate parameter ( $\lambda$ ), representing the intrinsic growth rate.
            - `args[1]` (float): Carrying capacity ( $G$ ), the maximum sustainable population.

        Returns:
        -----
        float
            The rate of change of the population (`dy/dt`) at time `t`.
        """
        l = args[0]
        G = args[1]
        return l * y * (1 - y / G)

```

Implement a problem solver using the ODE model and the integrator. As solution a list with the results for each time steps is given.

```

In [6]: def solve_growth_model(y0, tspan, model, integrator, dt, args=[]):
        """
        Numerically solves an ordinary differential equation (ODE) problem for a population growth model.

        Parameters:
        -----
        y0 (float or array):
            The initial value(s) of the dependent variable(s) at the start of the time span.
        tspan (list or array):
            The sequence of time points where the solution is computed. It must have the same length as y0.
        """

```

```

model (callable):
    The function defining the ODE model, typically of the form:
    def model(t, y, args):
        # Returns the derivative dy/dt

integrator (callable):
    The numerical method used to integrate the ODE. The function should
    def integrator(t, y, dt, model, args):
        # Returns the next value of y

dt (float):
    The time step used for integration.

args (list, optional):
    Additional parameters required by the model function.

Returns:
    list or array:
        A list containing the computed values of the dependent variable(s)
    """
    sol = [y0] # Store the solutions in sol list
    yk = y0    # Initialize solution
    for tk in tspan[:-1]: # iterate throw the time steps
        yk = integrator(tk, yk, dt, model, args) # Solve the ODE problem using
        sol.append(yk) # Storing the new solution
    return sol # Return at the end the list of solutions

```

Example:

```

import numpy as np

def growth_bernoulli_model(t, y, args=[]):
    l = args[0] # Growth rate
    G = args[1] # Carrying capacity
    return l * y * (1 - y / G)

def euler_integrator(t, y, dt, model, args):
    return y + dt * model(t, y, args)

y0 = 10 # Initial population
tspan = np.linspace(0, 10, 101) # Time from 0 to 10, with
0.1 intervals
dt = 0.1
args = [0.5, 100] # l = 0.5, G = 100

solution = solve_growth_model(y0, tspan,
growth_bernoulli_model, euler_integrator, dt, args)

for t, y in zip(tspan, solution):
    print(f"Time: {t:.2f}, Population: {y:.2f}")

```

Data on population growth in Colombia can be obtained from the source [Our World in Data](#). You can download this information from the prepared comma-separated text files: [population_1950-1999.csv](#)
[population_2000-2023.csv](#)

Note: This data is used solely within the context of this exercise class, and we are not responsible for the accuracy of the information presented therein. For more details, please refer to the cited site.

1. Your first task is to fit the bernoulli growth model with the information lying in the file [population_1950-1999.csv](#). In this way you will use the function `curve_fit` from the module `optimize` in the package `scipy` to fit the model parameters `l` and `G`.
 2. Use the fitted parameters to model how the population will grow up to 2040. Use the third-order method to fit the model.
 3. Compare the obtained solution with the data in file [population_2000-2023.csv](#). Try different time steps `dt` (0.01, 1.0, 3.0, 5.0) with the 3 different methods. Do you observe the change of accuracy? What would happen if you fit the model using a forwards Euler method keeping `dt=1.0`?
-

Solving task 1:

```
In [7]: # importing scipy optimize package...
        from scipy.optimize import curve_fit
```

```
In [8]: # Solving the population growth fitting problem for Colombia ( 1950 - 1999 )
        print("Solving the population growth fitting problem for Colombia ( 1950 - 1999 )")

        # Loading population data from 1950 to 1999
        data = np.loadtxt("population_1950-1999.csv", skiprows=1, delimiter=',')

        time = data[:, 0]
        population = data[:, 1]

        N = time.shape[0]

        y0 = 11766989 # Initial conditions
        tspan = np.linspace(time[0], time[-1], N) # Time from 0 to 10, with 0.1 in
        dt = 1.0
        l = 0.1
        G = 52321152
        theta = 0.5

        model = growth_bernoulli_model # Assign the growth model

        # integrator = solver_euler # Assign Forwards Euler integration method
        # integrator = solver_theta_rk2
        integrator = solver_heun_rk3
```



```

# Define the logistic growth model
def fitting_function(t, l, G):
    return solve_growth_model(y0, tspan, model, integrator, dt, args=[l, G])

# Fit the model to the data
params, _ = curve_fit(fitting_function, time, population, p0=[0.1, 52321152])
l, G = params

print(f"\n\n> Parameter l= {l}, Parameter G = {G}\n\n")

# Calculate the growth rate at each time step using the derivative of the logistic growth model
growth_rate = l * population * (1 - population / G)

population_result = solve_growth_model(y0, tspan, model, integrator, dt, args=[l, G])

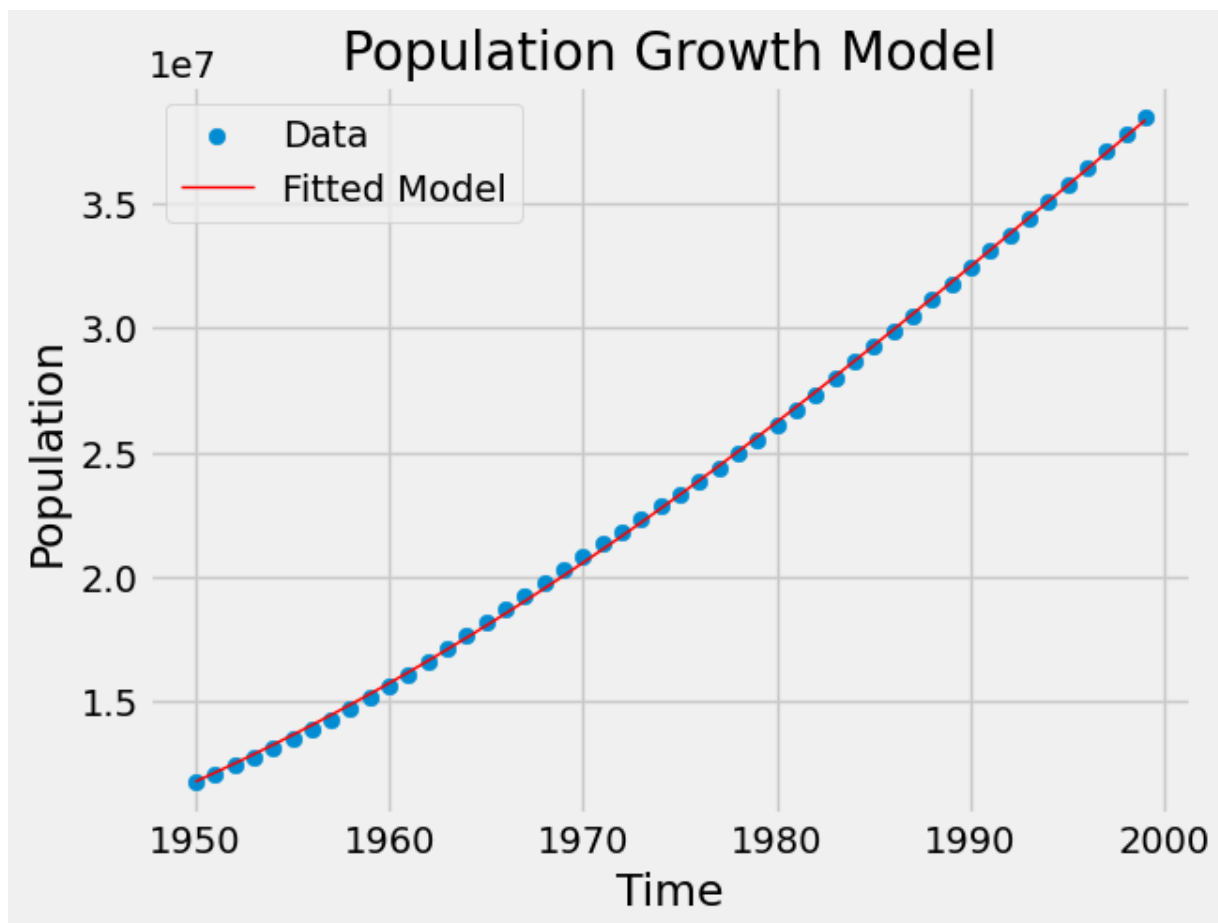
# Plot the results
with plt.style.context('fivethirtyeight'):
    plt.figure()
    plt.scatter(time, population, label="Data", )
    plt.plot(time, population_result, label="Fitted Model", color="red", linestyle='solid')
    plt.xlabel("Time")
    plt.ylabel("Population")
    plt.legend()
    plt.title("Population Growth Model")
    plt.show()

print("\n\nDisplay growth rates:\n")
# Display growth rates
for t, y, dy in zip(time, population, growth_rate):
    print(f"Time: {t}, Population: {y}, Growth Rate: {dy}")

```

Solving the population growth fitting problem for Colombia (1950 - 1999)

> Parameter l= 0.07092288849783045, Parameter G = 73887604.82974662



Display growth rates:

Time: 1950.0,	Population: 11766989.0,	Growth Rate: 701642.5630125055
Time: 1951.0,	Population: 12078812.0,	Growth Rate: 716620.635497896
Time: 1952.0,	Population: 12405851.0,	Growth Rate: 732129.0422639876
Time: 1953.0,	Population: 12746037.0,	Growth Rate: 748043.0154287516
Time: 1954.0,	Population: 13100844.0,	Growth Rate: 764404.2679696599
Time: 1955.0,	Population: 13473059.0,	Growth Rate: 781308.5070319491
Time: 1956.0,	Population: 13861804.0,	Growth Rate: 798679.5105186432
Time: 1957.0,	Population: 14270277.0,	Growth Rate: 816619.4805467512
Time: 1958.0,	Population: 14693293.0,	Growth Rate: 834860.553319716
Time: 1959.0,	Population: 15136220.0,	Growth Rate: 853592.0579961657
Time: 1960.0,	Population: 15606208.0,	Growth Rate: 873056.1365698212
Time: 1961.0,	Population: 16095207.0,	Growth Rate: 892857.4090473177
Time: 1962.0,	Population: 16599029.0,	Growth Rate: 912778.7819218184
Time: 1963.0,	Population: 17113825.0,	Growth Rate: 932630.7323488715
Time: 1964.0,	Population: 17637416.0,	Growth Rate: 952299.9674807206
Time: 1965.0,	Population: 18169052.0,	Growth Rate: 971732.934471534
Time: 1966.0,	Population: 18702692.0,	Growth Rate: 990693.4899086736
Time: 1967.0,	Population: 19235624.0,	Growth Rate: 1009083.2866059534
Time: 1968.0,	Population: 19766801.0,	Growth Rate: 1026869.9732618349
Time: 1969.0,	Population: 20291938.0,	Growth Rate: 1043921.955938909
Time: 1970.0,	Population: 20811790.0,	Growth Rate: 1060280.8844660053
Time: 1971.0,	Population: 21319101.0,	Growth Rate: 1075744.984881386
Time: 1972.0,	Population: 21819571.0,	Growth Rate: 1090516.4279027667
Time: 1973.0,	Population: 22325375.0,	Growth Rate: 1104956.749830983
Time: 1974.0,	Population: 22833475.0,	Growth Rate: 1118968.1270160011
Time: 1975.0,	Population: 23344650.0,	Growth Rate: 1132564.178649506
Time: 1976.0,	Population: 23858814.0,	Growth Rate: 1145733.6916701696
Time: 1977.0,	Population: 24393499.0,	Growth Rate: 1158890.516508241
Time: 1978.0,	Population: 24950341.0,	Growth Rate: 1172009.1316693525
Time: 1979.0,	Population: 25515626.0,	Growth Rate: 1184717.7854675937
Time: 1980.0,	Population: 26104502.0,	Growth Rate: 1197304.4219738059
Time: 1981.0,	Population: 26713266.0,	Growth Rate: 1209616.3186095841
Time: 1982.0,	Population: 27338334.0,	Growth Rate: 1221517.6705411419
Time: 1983.0,	Population: 27988176.0,	Growth Rate: 1233095.4745328357
Time: 1984.0,	Population: 28642689.0,	Growth Rate: 1243937.0364616862
Time: 1985.0,	Population: 29268471.0,	Growth Rate: 1253533.650696745
Time: 1986.0,	Population: 29879114.0,	Growth Rate: 1262173.382817975
Time: 1987.0,	Population: 30499472.0,	Growth Rate: 1270217.5487523177
Time: 1988.0,	Population: 31130528.0,	Growth Rate: 1277642.4100112834
Time: 1989.0,	Population: 31776281.0,	Growth Rate: 1284448.7725117106
Time: 1990.0,	Population: 32440064.0,	Growth Rate: 1290610.8052773643
Time: 1991.0,	Population: 33098371.0,	Growth Rate: 1295886.5842744396
Time: 1992.0,	Population: 33760570.0,	Growth Rate: 1300354.2031706613
Time: 1993.0,	Population: 34441479.0,	Growth Rate: 1304070.2127891171
Time: 1994.0,	Population: 35127408.0,	Growth Rate: 1306913.6839022185
Time: 1995.0,	Population: 35804671.0,	Growth Rate: 1308835.0358971937
Time: 1996.0,	Population: 36462740.0,	Growth Rate: 1309858.4543258962
Time: 1997.0,	Population: 37121156.0,	Growth Rate: 1310050.3974772284
Time: 1998.0,	Population: 37792170.0,	Growth Rate: 1309389.7409548527
Time: 1999.0,	Population: 38454863.0,	Growth Rate: 1307888.9024268903

Solving task 2:

```
In [9]: # Loading control data
data_test = np.loadtxt("population_2000-2023.csv", skiprows=1, delimiter=',')

time_test = data_test[:, 0]
population_test = data_test[:, 1]

# Simulating the population growth from 2000 - 2023

start = time_test[0]
stop = 2040

N = int(stop - start)
dt_test = 0.001

y0_test = 39089939 # Initial conditions from 1999
tspan_test = np.linspace(start, stop, int(N / dt_test)) # Time from 0 to 16

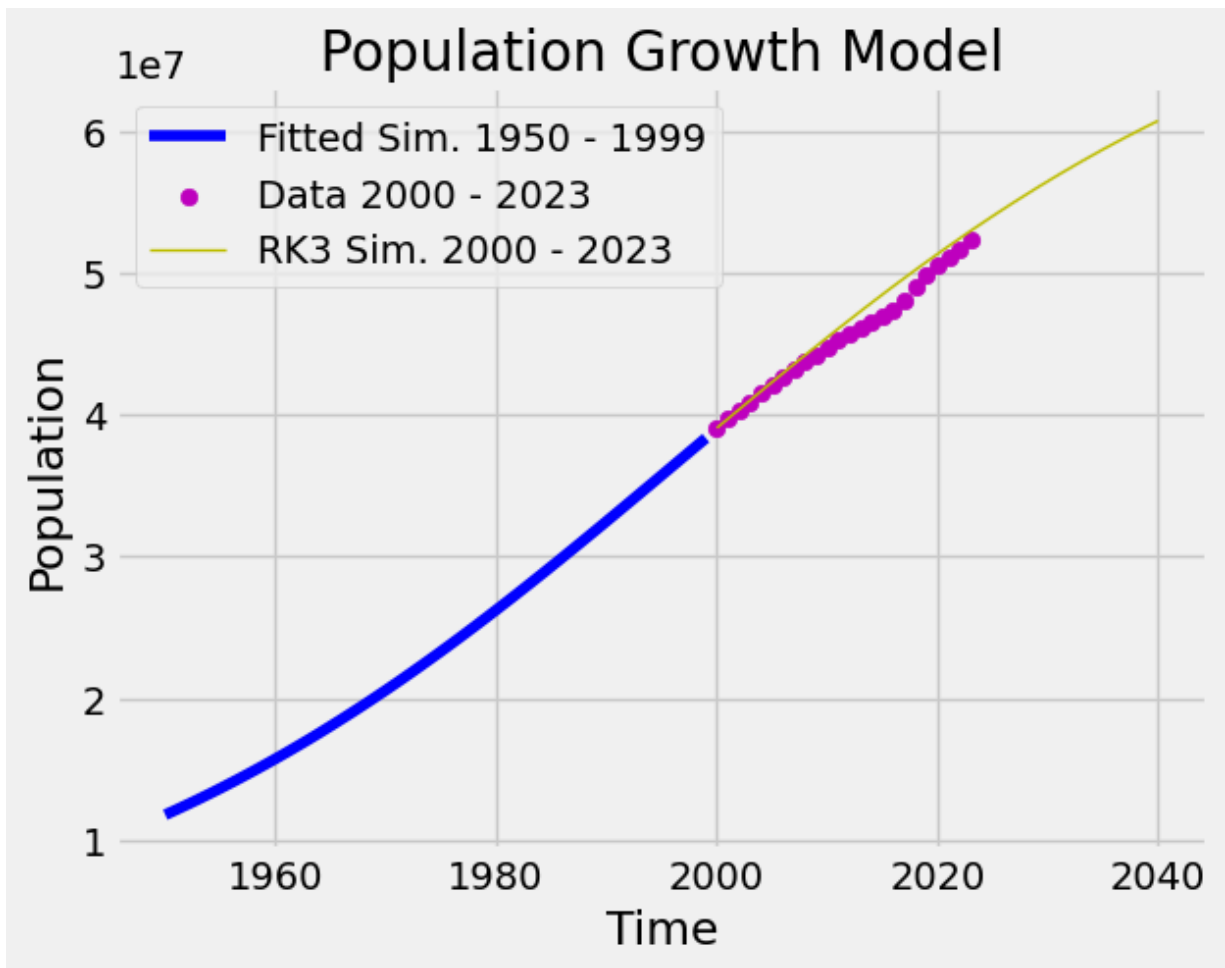
# parameters l and G are computed in the previous cell
print(f"\n\n> Parameter l= {l}, Parameter G = {G}\n\n")

model = growth_bernoulli_model # Assign the growth model

integrator = solver_heun_rk3 # Assign Forwards Euler integration method
population_result_test_heun_rk3 = solve_growth_model(y0_test, tspan_test, model)

# Plot the results
with plt.style.context('fivethirtyeight'):
    plt.figure()
    plt.plot(time, population_result, label="Fitted Sim. 1950 - 1999", color='r')
    plt.scatter(time_test, population_test, label="Data 2000 - 2023", color='b')
    plt.plot(tspan_test, population_result_test_heun_rk3, '-y', label="RK3 Solution")
    plt.xlabel("Time")
    plt.ylabel("Population")
    plt.legend()
    plt.title("Population Growth Model")
    plt.show()
```

> Parameter l= 0.07092288849783045, Parameter G = 73887604.82974662



Solving task 3:

```
In [10]: theta = 0.5 # for the theta-rk2 method

integrator = solver_euler
population_result_test_euler = solve_growth_model(y0_test, tspan_test, model)

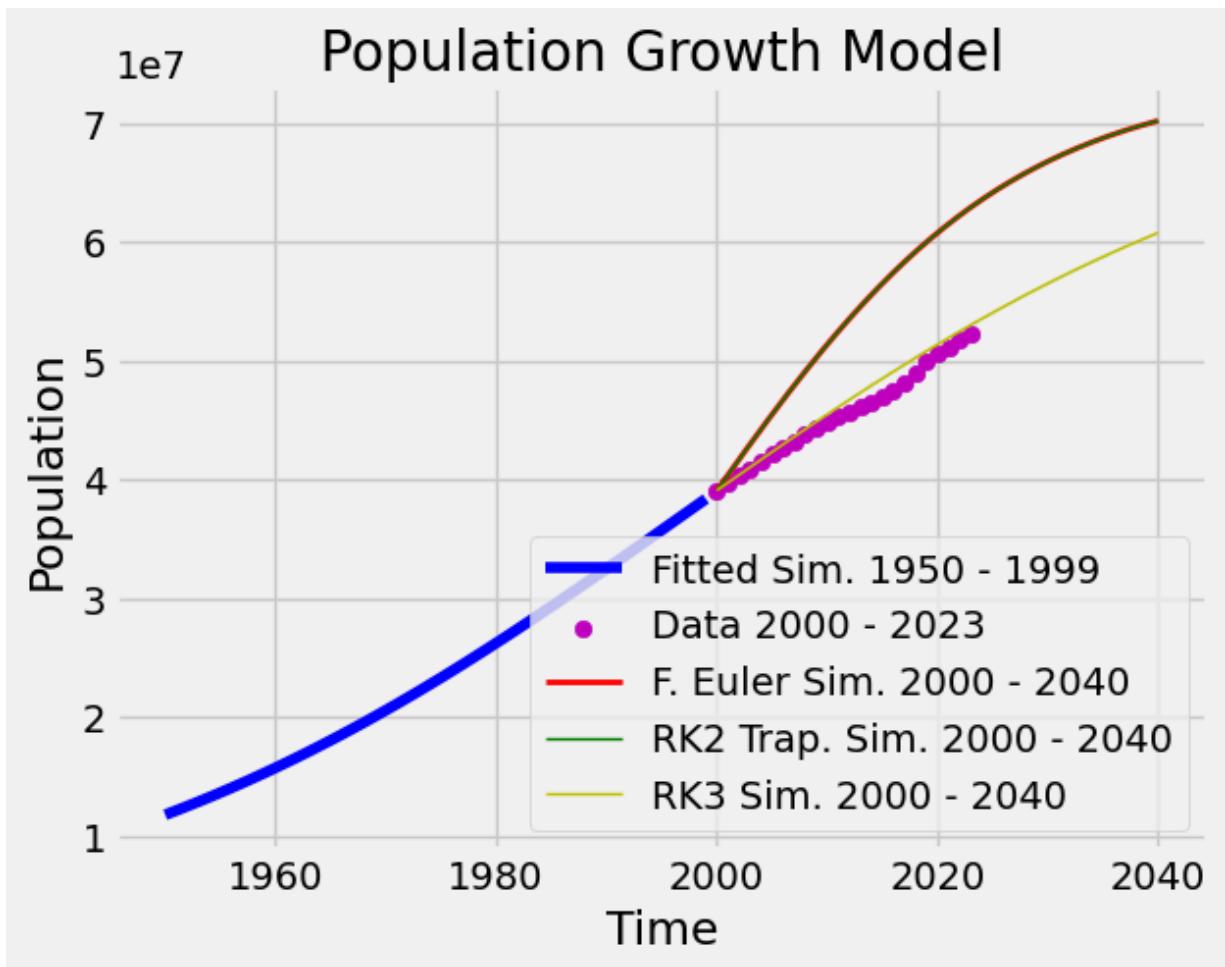
integrator = solver_theta_rk2
population_result_test_theta_rk2 = solve_growth_model(y0_test, tspan_test, model)

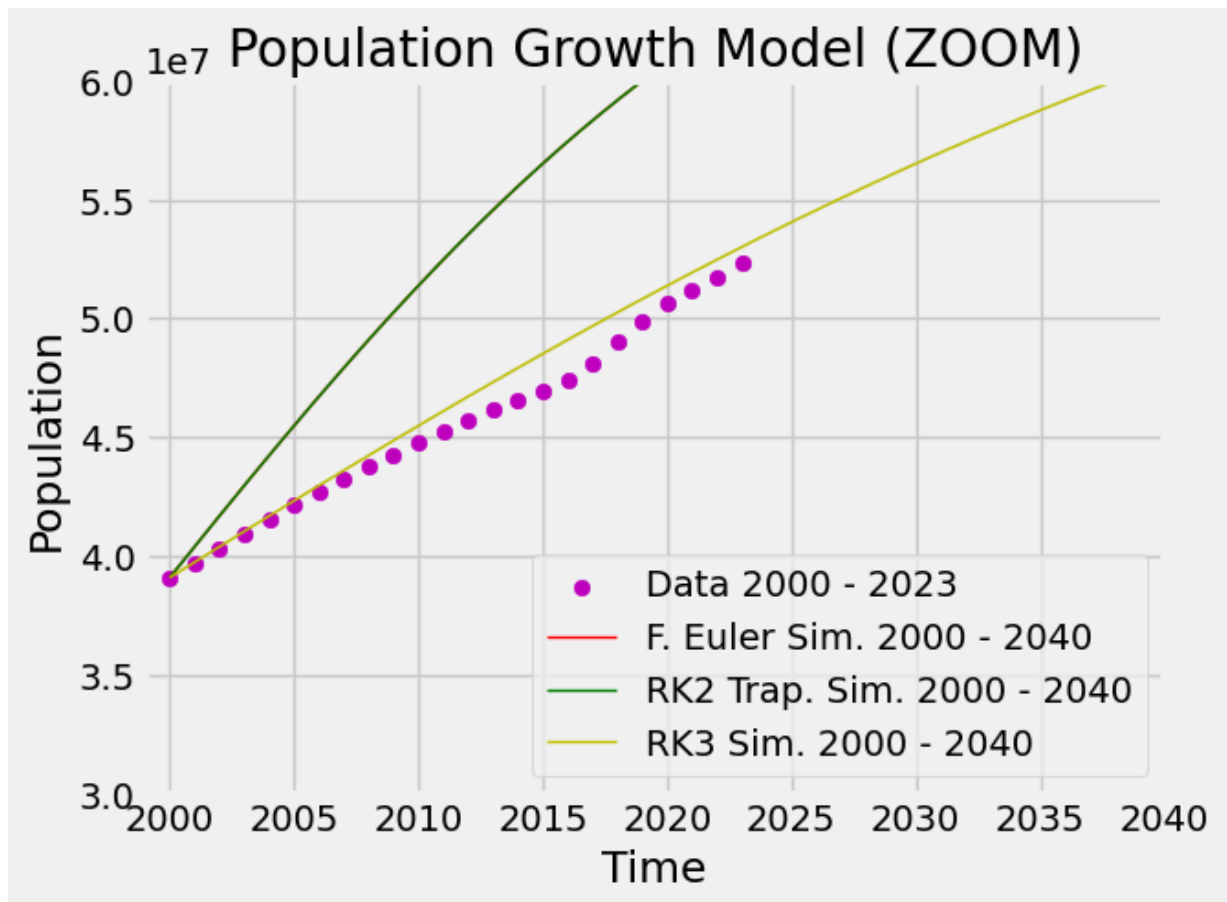
# Plot the results
with plt.style.context('fivethirtyeight'):
    plt.figure()
    plt.plot(time, population_result, label="Fitted Sim. 1950 - 1999", color='b')
    plt.scatter(time_test, population_test, label="Data 2000 - 2023", color='m')
    plt.plot(tspan_test, population_result_test_euler, '-r', label="F. Euler")
    plt.plot(tspan_test, population_result_test_theta_rk2, '-g', label="RK2")
    plt.plot(tspan_test, population_result_test_heun_rk3, '-y', label="RK3 S")
    plt.xlabel("Time")
    plt.ylabel("Population")
    plt.legend()
    plt.title("Population Growth Model")
    plt.show()
```

```

with plt.style.context('fivethirtyeight'):
    plt.figure()
    plt.scatter(time_test, population_test, label="Data 2000 - 2023", color=
    plt.plot(tspan_test, population_result_test_euler, '-r', label="F. Euler
    plt.plot(tspan_test, population_result_test_theta_rk2, '-g', label="RK2
    plt.plot(tspan_test, population_result_test_heun_rk3, '-y', label="RK3 S
    plt.xlim([1999, 2040])
    plt.ylim([3e+7, 6e+7])
    plt.xlabel("Time")
    plt.ylabel("Population")
    plt.legend()
    plt.title("Population Growth Model (ZOOM)")
    plt.show()


```





Pendulum Problem

Pendulum ODE System Exercise

 No description has been provided for this image

In this exercise, we will solve the equations of motion for a simple pendulum using two approaches:

1. **Using** `scipy.integrate.odeint`
2. **Using** `scipy.integrate.ode`

The ODE System

The simple pendulum is modeled, according to the Newton's second law, by the following second-order differential equation:

$$\frac{d^2\phi}{dt^2} + \frac{g}{\ell}\sin(\phi) = 0, \quad \phi(t_0) = \phi_0, \quad \frac{d\phi(t_0)}{dt} = \phi'_0$$

Where:

- $\phi(t)$ is the angular displacement at time $t > t_0$.
- g is the acceleration due to gravity.
- ℓ is the length of the pendulum.

To solve numerically, we convert it into a system of first-order ODEs:

$$\frac{d\phi}{dt} = \omega, \quad \frac{d\omega}{dt} = -\frac{g}{\ell}\sin(\phi)$$

Where $\omega(t)$ is the angular velocity.

Exercise Steps

Part 1: Solving Using `odeint`

1. Define the system of equations as a Python function.
2. Solve the ODE system using `scipy.integrate.odeint`.
3. Plot the angular displacement $\phi(t)$ and angular velocity $\omega(t)$ over time.

Part 2: Solving Using `ode`

1. Use `scipy.integrate.ode` to define and solve the same ODE system.
2. Compare the solutions with those obtained using `odeint`.

```
In [11]: import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint, ode

# Constants
g = 9.81 # Gravity (m/s^2)
L = 1.0 # Length of pendulum (m)
phi0 = np.pi / 4 # Initial angle (radians)
omega0 = 0.0 # Initial angular velocity (rad/s)
t = np.linspace(0, 10, 1000) # Time array

# Part 1: Using odeint
def pendulum_odeint(y, t, g, L):
    phi, omega = y
    dydt = [omega, -(g/L) * np.sin(phi)]
    return dydt

# Initial conditions
y0 = [phi0, omega0]

# Solve with odeint
solution_odeint = odeint(pendulum_odeint, y0, t, args=(g, L))
phi_odeint, omega_odeint = solution_odeint[:, 0], solution_odeint[:, 1]

# Part 2: Using ode
```



```

def pendulum_ode(t, y, g, L):
    phi, omega = y
    return [omega, -(g/L) * np.sin(phi)]

# Initializing ode solver
solver = ode(pendulum_ode)
solver.set_integrator('dopri5') # Dormand-Prince (Runge-Kutta method)
solver.set_f_params(g, L)
solver.set_initial_value(y0, t[0])

# Solve with ode
solution_ode = []
time_ode = []
while solver.successful() and solver.t < t[-1]:
    solver.integrate(solver.t + t[1] - t[0])
    solution_ode.append(solver.y)
    time_ode.append(solver.t)

solution_ode = np.array(solution_ode)
phi_ode, omega_ode = solution_ode[:, 0], solution_ode[:, 1]

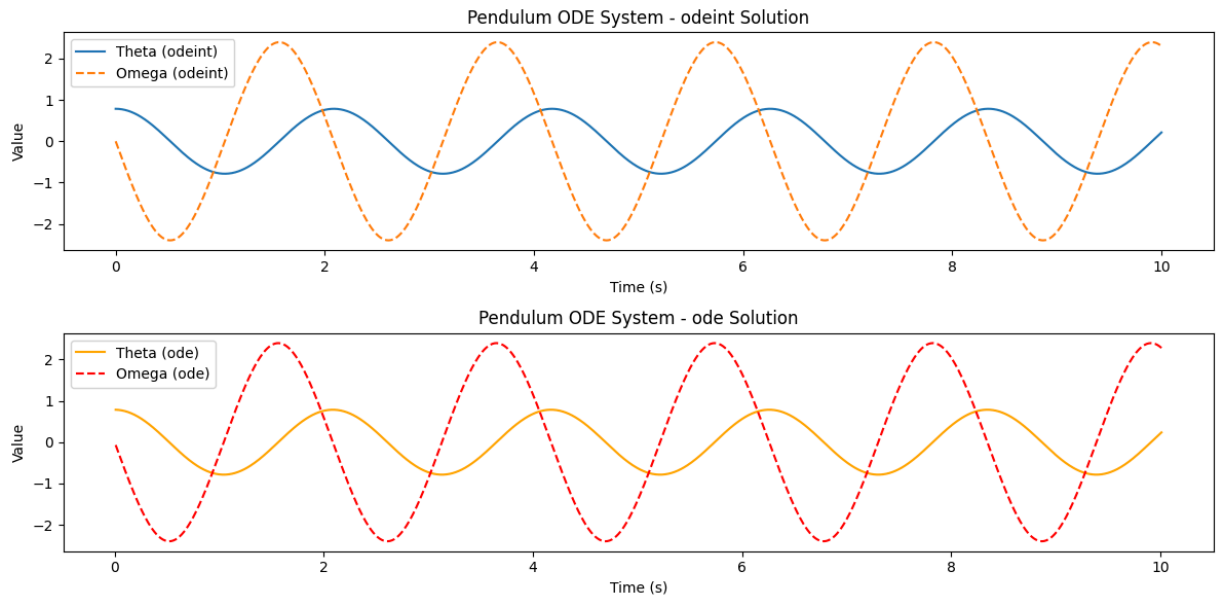
# Plotting results
plt.figure(figsize=(12, 6))

# Plot odeint solution
plt.subplot(2, 1, 1)
plt.plot(t, phi_odeint, label="Theta (odeint)")
plt.plot(t, omega_odeint, label="Omega (odeint)", linestyle="--")
plt.title("Pendulum ODE System - odeint Solution")
plt.xlabel("Time (s)")
plt.ylabel("Value")
plt.legend()

# Plot ode solution
plt.subplot(2, 1, 2)
plt.plot(time_ode, phi_ode, label="Theta (ode)", color="orange")
plt.plot(time_ode, omega_ode, label="Omega (ode)", linestyle="--", color="red")
plt.title("Pendulum ODE System - ode Solution")
plt.xlabel("Time (s)")
plt.ylabel("Value")
plt.legend()

plt.tight_layout()
plt.show()

```



Convergence Analysis Exercise for Numerical Methods

In this follow-up exercise, you will analyze the convergence of the numerical methods (`odeint` and `ode`) used to solve the pendulum ODE system. To do so, we will calculate the **convergence order** using a computational heuristic formula.

Computational convergence analysis

The **convergence order** (α) of a numerical method quantifies how the error decreases as the time step size (h) decreases. Given three numerical solutions obtained with step sizes h_1 , $h_2 = h_1/2$, and $h_3 = h_2/2$, we can estimate α using the following formula:

$$\alpha = \frac{1}{\log(2)} \log\left(\frac{|P_1 - P_2|}{|P_2 - P_3|}\right)$$

Where:

- P_1, P_2, P_3 are the numerical solutions obtained at the same time point using different step sizes h_1, h_2, h_3 .
- $\log(2)$ normalizes the result for a halving of step size.

This formula assumes that the method converges as $h \rightarrow 0$ and the error decreases consistently.

Exercise Steps

1. Set up smaller time steps:

Solve the pendulum system using three decreasing time step sizes:

$$h_1, h_2 = h_1/2, h_3 = h_2/2.$$

2. Compare solutions at the same time point:

Extract the numerical solutions (P_1, P_2, P_3) for angular displacement (ϕ) at a fixed time (e.g., $t = 5s$).

3. Calculate convergence order:

Use the formula to estimate α for both `odeint` and `ode` methods.

4. Interpret results:

Compare the convergence orders obtained and discuss the reliability of each method.

```
In [14]: import numpy as np
from scipy.integrate import odeint, ode

# Constants
g = 9.81 # Gravity (m/s^2)
L = 1.0 # Length of pendulum (m)
phi0 = np.pi / 4 # Initial angle (radians)
omega0 = 0.0 # Initial angular velocity (rad/s)

# Define the ODE system
def pendulum(y, t, g, L):
    phi, omega = y
    dydt = [omega, -(g/L) * np.sin(phi)]
    return dydt

# Time steps
t1 = np.linspace(0, 10, 100) # Coarse step size
t2 = np.linspace(0, 10, 200) # Medium step size
t3 = np.linspace(0, 10, 400) # Fine step size

# Solve using odeint
y0 = [phi0, omega0]
sol1_odeint = odeint(pendulum, y0, t1, args=(g, L))
sol2_odeint = odeint(pendulum, y0, t2, args=(g, L))
sol3_odeint = odeint(pendulum, y0, t3, args=(g, L))

# Extract solutions at t = 5s
phi1_odeint = sol1_odeint[np.isclose(t1, 5, atol=0.08)][0][0]
phi2_odeint = sol2_odeint[np.isclose(t2, 5, atol=0.08)][0][0]
phi3_odeint = sol3_odeint[np.isclose(t3, 5, atol=0.08)][0][0]

# Compute convergence order for odeint
alpha_odeint = (1 / np.log(2)) * np.log(abs((phi1_odeint - phi2_odeint) / (phi2_odeint - phi3_odeint)))

print(f"Convergence order (odeint): {alpha_odeint:.4f}, so O(dt^{int(np.round(alpha_odeint, 0))})")

# ===== Solve using ode =====
def pendulum_ode(t, y, g, L):
    phi, omega = y
    return [omega, -(g/L) * np.sin(phi)]

solver = ode(pendulum_ode)
```

```

solver.set_integrator('dopri5')
solver.set_f_params(g, L)
solver.set_initial_value(y0, 0)

# Function to solve with decreasing step sizes
def solve_with_ode(solver, t_values):
    solutions = []
    for t in t_values:
        #if solver.successful() and solver.t <= t:
        solver.integrate(t + t_values[1] - t_values[0])
        solutions.append(solver.y[0]) # Only store phi
    return np.array(solutions)

solver.set_initial_value(y0, t1[0])
phi1_ode = solve_with_ode(solver, t1)

solver.set_initial_value(y0, t2[0])
phi2_ode = solve_with_ode(solver, t2)

solver.set_initial_value(y0, t3[0])
phi3_ode = solve_with_ode(solver, t3)

# Extract solutions at t = 5s
phi1_ode_at_5 = phi1_ode[np.isclose(t1, 5, atol=0.08)][0]
phi2_ode_at_5 = phi2_ode[np.isclose(t2, 5, atol=0.08)][0]
phi3_ode_at_5 = phi3_ode[np.isclose(t3, 5, atol=0.08)][0]

# Compute convergence order for ode
alpha_ode = (1 / np.log(2)) * np.log(abs((phi1_ode_at_5 - phi2_ode_at_5) / (

print(f"Convergence order (ode): {alpha_ode:.4f}, so O(dt^{int(np.round(alpha_ode))})")

```

Convergence order (odeint): 0.9428, so $O(dt^1)$

Convergence order (ode): 2.3519, so $O(dt^2)$

```

In [13]: # Saving the data to file pendulum_data.dat
print(phi3_ode)
np.savetxt("pendulum_data.dat", phi3_ode, delimiter=" ")

```

[0.78322035 0.77669642 0.76585497 0.75074394 0.73143096 0.70800388
0.6805713 0.64926318 0.61423136 0.57565008 0.53371621 0.48864937
0.44069171 0.3901073 0.33718118 0.28221798 0.22554002 0.1674851
0.10840377 0.04865629 -0.01139064 -0.07136745 -0.13090521 -0.18963928
-0.2472127 -0.30327941 -0.35750718 -0.40958017 -0.45920097 -0.50609236
-0.54999848 -0.59068568 -0.62794297 -0.66158207 -0.69143728 -0.71736512
-0.73924379 -0.75697265 -0.77047156 -0.77968042 -0.78455865 -0.78508492
-0.78125693 -0.77309141 -0.76062426 -0.74391082 -0.72302635 -0.69806647
-0.66914784 -0.63640863 -0.60000915 -0.56013217 -0.51698322 -0.47079049
-0.42180454 -0.37029752 -0.31656209 -0.26090982 -0.20366922 -0.14518323
-0.08580647 -0.02590205 0.03416184 0.09401543 0.15329102 0.21162646
0.26866858 0.32407625 0.37752316 0.42870023 0.47731755 0.52310586
0.56581767 0.60522787 0.64113409 0.67335664 0.70173832 0.72614391
0.74645972 0.76259294 0.77447109 0.78204152 0.78527099 0.78414538
0.77866962 0.76886767 0.75478277 0.73647775 0.71403552 0.68755962
0.65717479 0.62302755 0.58528667 0.54414356 0.49981234 0.45252975
0.40255465 0.35016712 0.29566719 0.23937313 0.18161922 0.12275319
0.06313325 0.00312486 -0.05690277 -0.11658024 -0.17554155 -0.23342755
-0.2898893 -0.34459098 -0.39721256 -0.44745202 -0.49502713 -0.53967677
-0.58116189 -0.61926599 -0.6537953 -0.68457872 -0.71146742 -0.73433442
-0.75307402 -0.76760119 -0.77785101 -0.78377824 -0.78535692 -0.78258015
-0.77546006 -0.76402789 -0.74833427 -0.72844957 -0.70446448 -0.67649048
-0.64466052 -0.6091295 -0.57007476 -0.5276963 -0.48221686 -0.43388164
-0.38295767 -0.32973279 -0.27451416 -0.2176264 -0.15940921 -0.10021467
-0.04040414 0.0196551 0.07959333 0.13904201 0.19763735 0.25502371
0.31085678 0.36480643 0.41655924 0.46582049 0.51231583 0.55579242
0.59601973 0.63278986 0.66591767 0.69524055 0.72061803 0.74193125
0.75908242 0.7719942 0.7806092 0.7848895 0.78481639 0.78039019
0.77163027 0.75857516 0.74128293 0.71983159 0.69431965 0.66486667
0.63161385 0.59472455 0.55438469 0.51080292 0.46421052 0.41486108
0.36302965 0.3090116 0.25312097 0.19568837 0.13705857 0.07758752
0.01763923 -0.04241765 -0.10221344 -0.16138099 -0.21955921 -0.27639638
-0.33155327 -0.38470581 -0.43554744 -0.48379102 -0.52917024 -0.57144063
-0.61038023 -0.64578977 -0.6774927 -0.70533486 -0.72918402 -0.74892941
-0.76448104 -0.77576919 -0.7827439 -0.78537458 -0.78364974 -0.7775769
-0.76718267 -0.75251296 -0.73363334 -0.71062953 -0.68360793 -0.65269625
-0.61804406 -0.57982322 -0.53822825 -0.49347641 -0.44580753 -0.39548343
-0.34278703 -0.28802099 -0.23150591 -0.17357809 -0.11458688 -0.05489169
0.00514129 0.06514263 0.12474316 0.18357762 0.24128811 0.29752729
0.35196138 0.4042727 0.45416187 0.50134947 0.5455774 0.58660967
0.62423291 0.65825649 0.68851237 0.71485479 0.73715973 0.75532434
0.76926641 0.77892375 0.78425379 0.78523319 0.7818577 0.77414204
0.76212011 0.74584523 0.72539053 0.70084953 0.67233663 0.63998776
0.6039609 0.56443648 0.52161764 0.47573024 0.42702253 0.37576446
0.32224662 0.26677872 0.20968758 0.1513148 0.0920139 0.03214726
-0.02791727 -0.08780993 -0.14716265 -0.20561253 -0.26280531 -0.31839848
-0.37206407 -0.42349111 -0.47238761 -0.51848212 -0.56152483 -0.60128829
-0.63756777 -0.67018122 -0.69896911 -0.72379397 -0.74453989 -0.76111191
-0.77343548 -0.78145589 -0.7851379 -0.78446543 -0.77944139 -0.77008778
-0.75644582 -0.73857627 -0.71655994 -0.69049819 -0.66051353 -0.62675019
-0.5893746 -0.54857581 -0.50456558 -0.45757832 -0.4078706 -0.35572037
-0.30142565 -0.2453029 -0.18768485 -0.12891796 -0.06935951 -0.00937436
0.05066851 0.11039955 0.16945223 0.22746655 0.28409236 0.33899233
0.39184468 0.44234544 0.49021024 0.53517573 0.57700051 0.61546573
0.65037526 0.68155565 0.70885578 0.73214642 0.75131968 0.7662884
0.77698561 0.78336402 0.7853957 0.78307175 0.77640234 0.76541671

0.75016344	0.73071084	0.70714744	0.67958256	0.64814687	0.61299298
0.57429585	0.53225314	0.48708523	0.43903501	0.38836726	0.33536773
0.28034165	0.22361192	0.16551681	0.10640723	0.04664373	-0.01340685
-0.07337489	-0.13289156	-0.19159241	-0.24912079	-0.30513109	-0.35929157
-0.41128696	-0.46082054	-0.50761575	-0.5514175	-0.59199288	-0.62913166
-0.66264632	-0.69237192	-0.71816571	-0.73990662	-0.75749469	-0.77085048
-0.77991454	-0.78464695	-0.785027	-0.78105306	-0.77274247	-0.76013179
-0.74327703	-0.72225411	-0.69715936	-0.66811013	-0.63524535	-0.59872606
-0.5587358	-0.51548087	-0.46919021	-0.4201151	-0.36852843	-0.31472352
-0.25901253	-0.20172449	-0.1432028	-0.08380241	-0.02388664	0.03617618
0.09601632	0.15526618	0.21356389	0.27055661]		