

Assignment 3 - The Butterfly Effect

Group number 9, 1796097, 1732722

1 Summary

This assignment provided a guided set of steps for us to be able to implement ODE solvers applied to the "Butterfly Effect" phenomenon for a double pendulum. The assignment aims to first set up ODE equations as Python functions. The ODEs are solved using the SciPy solver, implementing first and higher-order ODE solver methods. To continue, the implementations need to be tested and verified. Finally, a numerical analysis is carried out using the order of convergence of the solvers used. Firstly, the chemical reaction in a batch reactor is converted into an ODE resulting in a time-derivative output. A script was created to analyze analytically and numerically ODEs. To continue, Euler's method was coded and compared to the analytical result, computing the error margin between each solution as well as verifying its expected rate of convergence. Secondly, a similar procedure was repeated implementing the concentration and energy balances as a system of ODEs and a corresponding Euler method. Once we created our very own numerical solvers that are capable of solving systems of ODEs, we applied them to the Butterfly effect of a double pendulum; a tiny change in the initial conditions will ultimately result in completely different behaviour of the swinging motion of the pendula. The ODEs were analyzed using conservation of energy and path variations as can be observed in Tasks 10 and 11.

2 Results and discussion

1. Given that the reaction was of the third order, one can identify the equation corresponding to the derivative of the given chemical reaction is:

$$\frac{dC_A}{dt} = -k_3 \cdot C_A^3 \quad (1)$$

Based on the equation above, the function can be created as can be seen in Listing 1.

```
1 def third_order_derivative(t, Ca, k=0.01):
2     '''Returns the concentration over time for a third order chemical
3     reaction.
4     t = time in seconds
5     Ca = concentration of component a (a -> b)
6     k = reaction rate constant (default set to 0.01 as this is the
7     value for task 1.)'''
8     dca_dt = - k * Ca**3
9     return dca_dt
```

Listing 1: Implementation of the third order chemical reaction derivative

2. In order to obtain the analytical solutions, the differential equation given in [Equation 1](#) must be

solved algebraically as follows.

$$\frac{dC_A}{dt} = -k_3 \cdot C_A^3 \quad (2)$$

$$\int_{C_{A,0}}^{C_A} \frac{dC_A}{C_A^3} = -k \int_0^t dt \quad (3)$$

$$\left[\frac{-1}{C_A^2} \right]_{C_{A,0}}^{C_A} = -k \cdot t \quad (4)$$

$$\frac{1}{2 \cdot C_{A,0}^2} - \frac{1}{2 \cdot C_A^2} = -k \cdot t \quad (5)$$

$$\frac{1}{C_A^2} = \frac{1}{C_{A,0}^2} + 2 \cdot k \cdot t \quad (6)$$

$$C_A = \sqrt{\frac{1}{C_{A,0}^{-2} + 2 \cdot k \cdot t}} \quad (7)$$

(8)

By using equation (7), the function was created as can be seen in Listing 2.

```

1 def third_order_analytical(t, c0, k3):
2     '''Returns the concentration at a given time for a third order
3     irreversible chemical reaction.
4     t = Time in seconds. If a vector it will retrun a vector of
5     solutions
6     c0 = initial concentration.
7     k3 = reaction rate constant for the third order reaction.'''
    Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
    return Ca

```

Listing 2: Implementation of the analytical solution to the ordinary differential equation

It is important to note that the function can take numbers or numpy arrays as inputs.

- For this task we chose to set $C_{A,0} = 5 \text{ mol L}^{-1}$. This initial concentration was chosen arbitrarily. Moreover, the time span was chosen to go from 0 to 100 seconds as beyond 100 seconds the curves simply approach 0.

Concentration of A over time with initial concentration of 5 mol/L

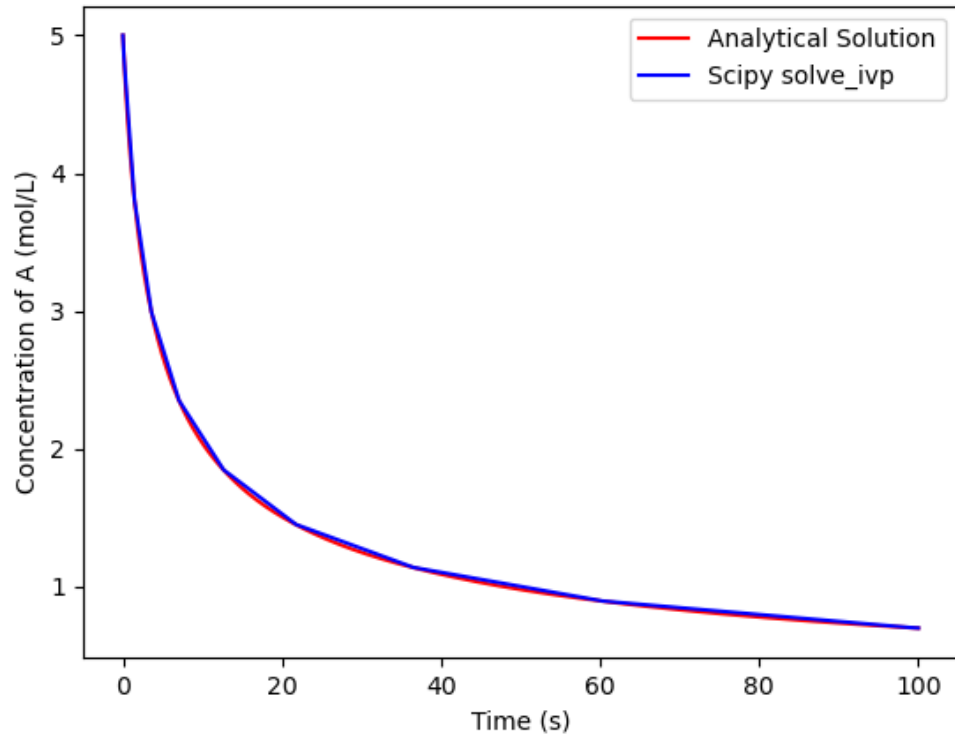


Figure 1: Analytical Solution vs Scipy "solve_ivp"

As can be seen in [Figure 1](#), the "solve_ivp" function from the "Scipy" library is able to approximate the analytical solution very well. However, as the "solve_ivp" function does not use constant intervals, the solution appears to be slightly "blocky". [Figure 2](#) shows a closer look of the two functions. Although the two functions deviate slightly, this can only be seen when one zooms in closer.

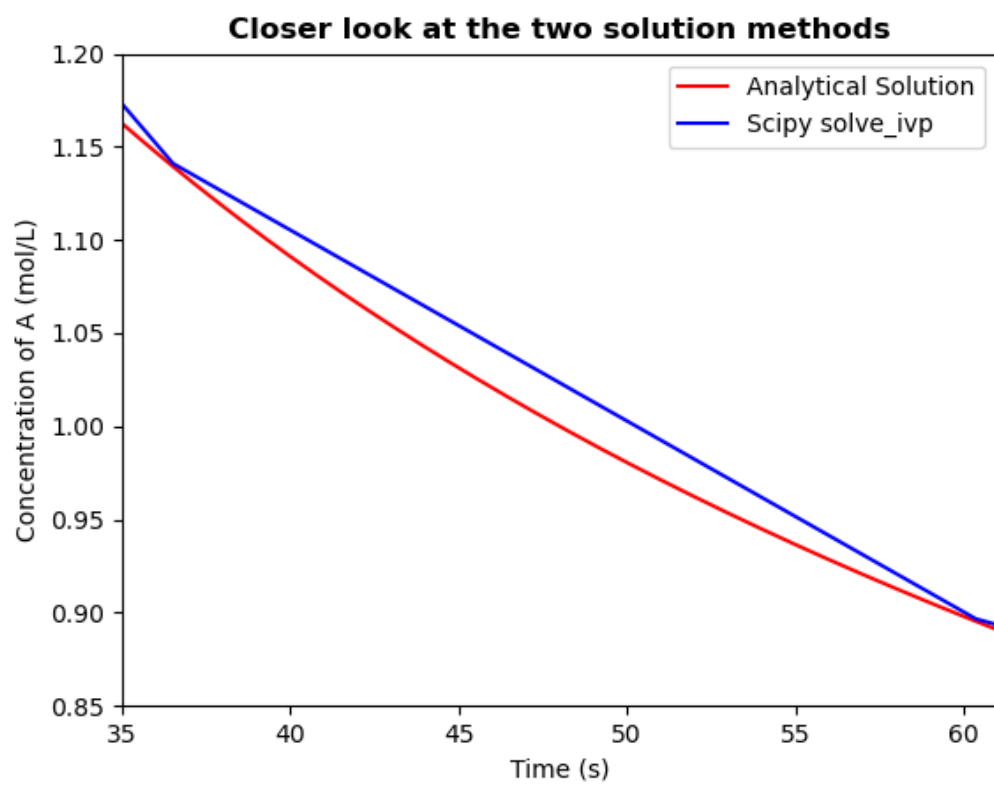


Figure 2: Closer look at the two curves.

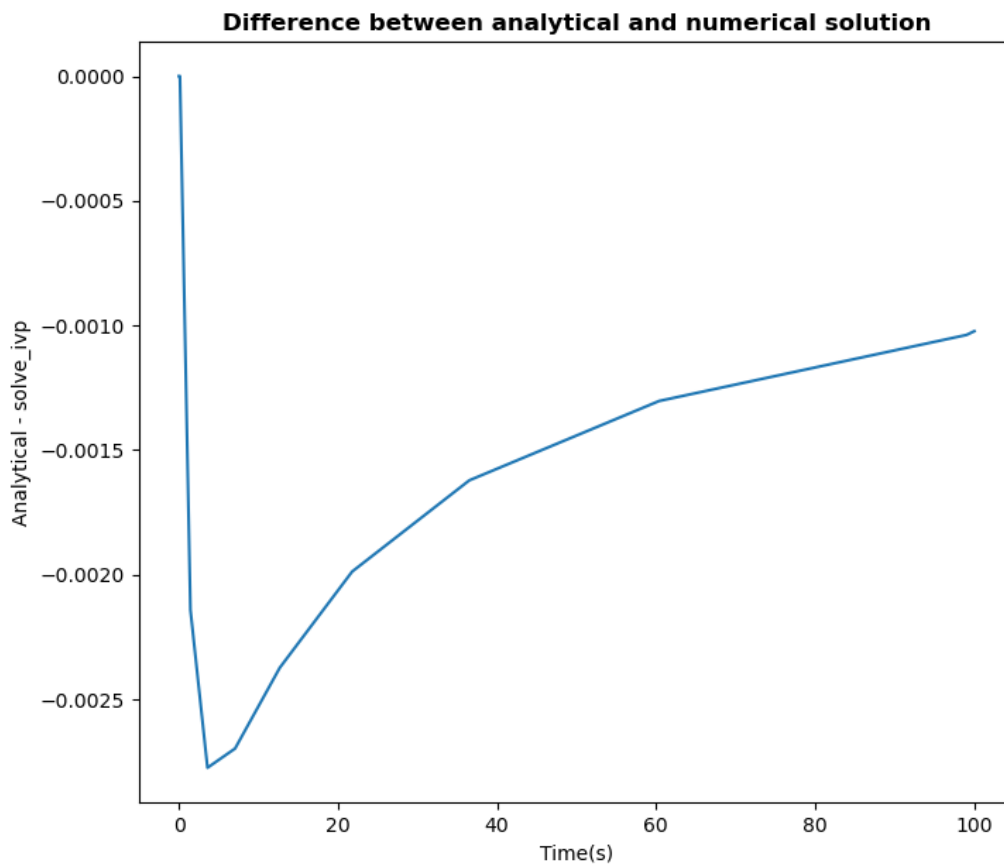


Figure 3: Difference between the analytical solution and "Scipy" solution

One can see in [Figure 3](#) that the deviation from the analytical solution and numerical solution is quite small having a maximum value of $0.00277 \text{ mol L}^{-1}$.

4. For this task a function was created which can solve ordinary differential equations using the explicit first-order Euler method as can be seen in [Listing 3](#). This function is very similar to the functions described in lecture 9. Although it was not specified in the assignment, a fourth optional argument was given in the function to ensure customizability.

```

1 def first_order_euler(fun, tspan, y0, number_of_points=100_000): # 100
2     _000 corresponds to to a dt of 0.001 as asked in the question
3     '''Explicit first order euler solver for ODEs.\n
4     fun = derivative function \n
5     tspan = supply the time span as a vector ie [0,10] \n
6     y0 = initial condition ie at t=0 y0=100 \n
7     number_of_points = Number of points. Default set to 100000 in
8         order to have dt = 0.001. Higher number means longer
9         computation time'''
10    dt = (tspan[1] - tspan[0])/number_of_points
11    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
12    y = np.zeros(number_of_points+1)
13    y[0] = y0 ## Setting up the initial conditions
14    for i in range(number_of_points):
15        y[i+1] = y[i] + dt * fun(t[i], y[i])
16    return t, y

```

Listing 3: Implementation of the first order explicit euler method

Concentration of A over time with initial concentration of 5 mol/L

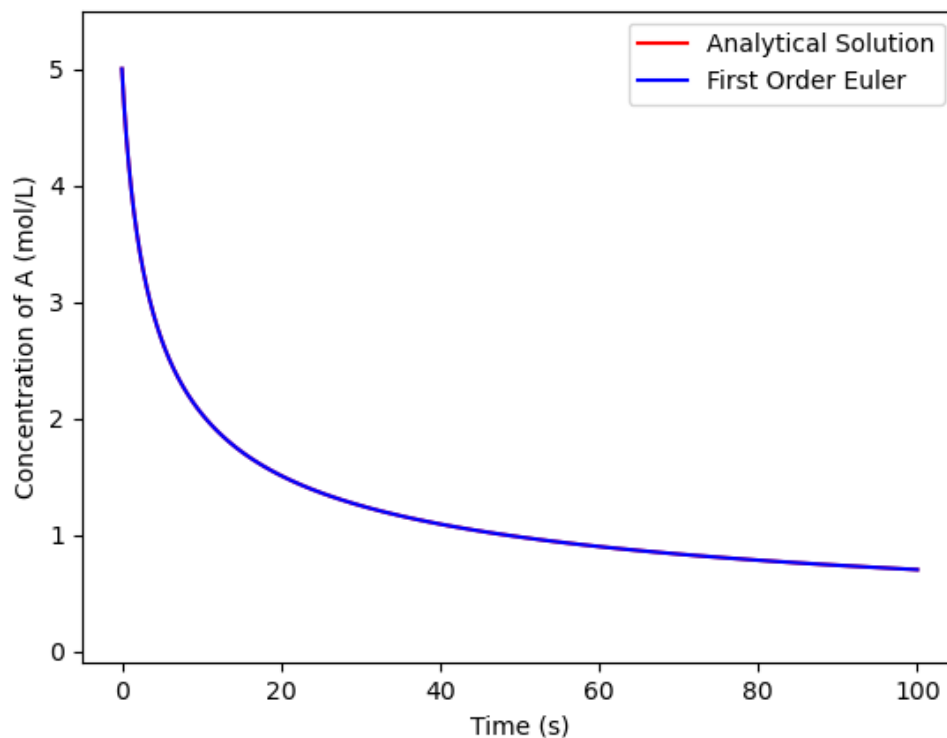


Figure 4: Analytical solution and first order Euler approximation visualization.

As can be seen in Figure 4, the two curves lie nearly directly over one another. This is most likely due to the very small value of Δt which yields an excellent approximation for the ODE on the specified domain.

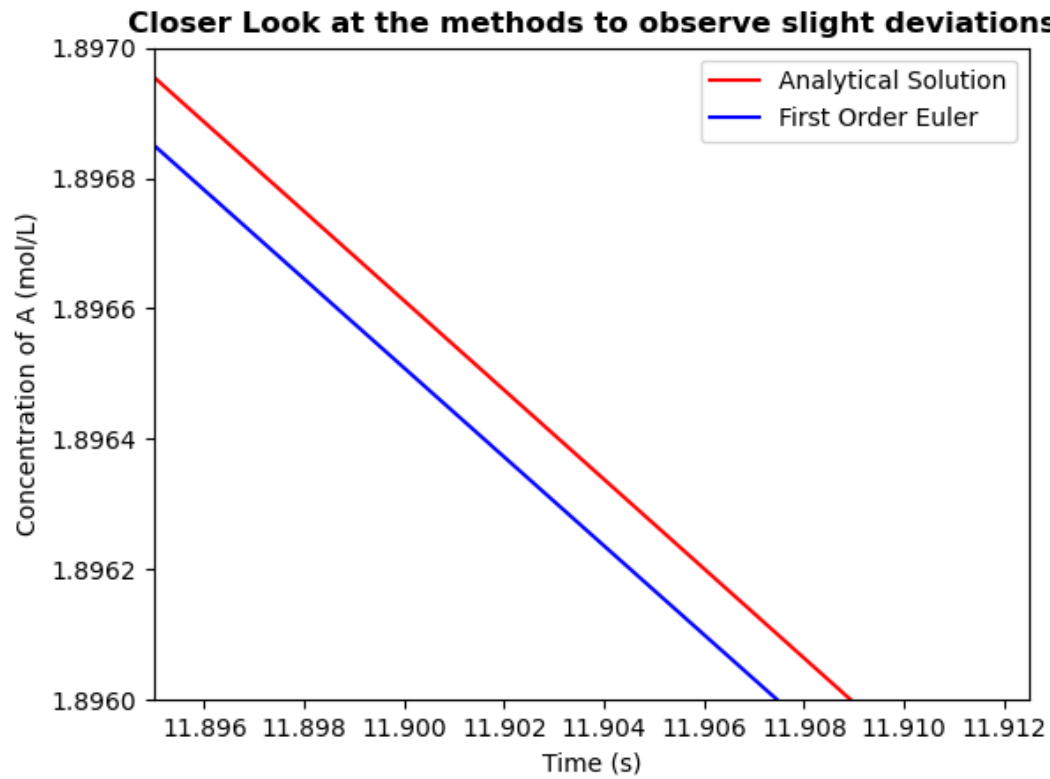


Figure 5: Closer look at the two curves.

Figure 5 shows a closer look at the two functions. One can see that although a deviation can be observed, the y -axis must be made smaller to be able to observe such a deviation.

5. In order to characterize the solution, the relative error and order of convergence were calculated. To calculate the relative error the following formula was used:

$$\epsilon_{rel} = \left| \frac{y_{numerical} - y_{analytical}}{y_{analytical}} \right| \quad (9)$$

Next, in order to calculate the rate of convergence the following formula was used:

$$r = \frac{\log \frac{\epsilon_2}{\epsilon_1}}{\log \frac{N_1}{N_2}} \quad (10)$$

Both formulae were obtained from lecture notes (Ode1_Ode2.pdf). Thus, by using these two equations the error and rate of convergence was calculated as can be seen in Listing 4

```

1 N = np.linspace(100, 100000, 10, dtype=int) # We need dtype=int
   because we need an integer number of points for the function
2 solution_list = []
3 error_list = []
4 #all comparing at t=4
5 compare_time = 4
6 rate_of_convergence = ['N/A'] # First element is N/A because it doesnt
   make sense to calculate the rate of convergence for the first
   point
7 for i in range(len(N)):
8     approx_sol_general = first_order_euler(third_order_derivative, [0,
   4], initial_conc, N[i])
9     approx_sol = approx_sol_general[1][-1] # Gets the last y value
   corresponding to t = 4
10    exact_sol = third_order_analytical(4, initial_conc, 0.01)
11    solution_list.append(approx_sol)
12    rel_error = np.abs((approx_sol - exact_sol)/exact_sol) # Relative
   error calculation. No specific type of error was asked to
   calculate.
13    error_list.append(rel_error)
14    if i > 0: # Doesn't make sense to calculate for the first point
15        r = (np.log(error_list[i]/error_list[i-1]))/(np.log(N[i-1] / N
   [i]))
16        rate_of_convergence.append(r)

```

Listing 4: Loop to calculate error and rate of convergence

An important thing to note is that when generating the `np.linspace` we had to specify that the elements of the linspace are integers as it does not make sense to have a non-interger number of points. The solutions were subsequently placed in a "pandas" dataframe. Table 1 demonstrates that the implemented Euler method has a first order rate of convergence alligns with expectation.

Table 1: Analysis of Euler Method

N_t	Calculated Solution	$\epsilon_{\text{relative}}$	Rate of Convergence
100	2.878775	2.76E-03	N/A
11200	2.886681	2.50E-05	1.001256
22300	2.886716	1.20E-05	1.000038
33400	2.886728	8.00E-06	1.000022
44500	2.886734	6.00E-06	1.000015
55600	2.886737	5.00E-06	1.000012
66700	2.886739	4.00E-06	1.00001
77800	2.886741	4.00E-06	1.000008
88900	2.886742	3.00E-06	1.000007
100000	2.886743	3.00E-06	1.000006

Concentration of A over time with initial concentration of 5 mol/L

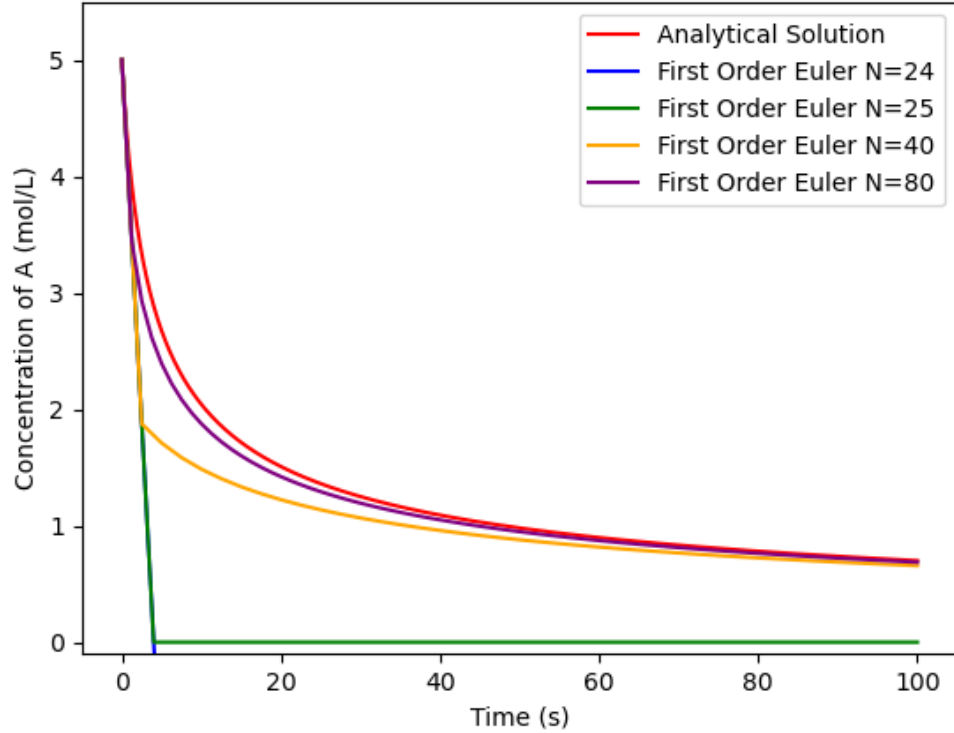


Figure 6: Closer look at the two curves.

In order to find the minimum number of iterations (thus the maximum Δt) the Euler method was plotted against the analytical solution with a varied number of points. It can be observed that using $N=24$ ($\Delta t = 4.1667$) the graph shoots off to negative infinity which does not align with expectations. The first "stable" solution can be observed at $N=25$ ($\Delta t = 4$) although the graph goes to zero quite rapidly. Moreover, one can observe that a small increase in the number will greatly improve the calculated solution.

6. For this task the system of differential equations had to be rewritten such that we had all the derivative terms on one side of the equation. This was done in the following way :

$$\frac{dC_A}{dt} = -A \cdot \exp\left(-\frac{E_A}{RT}\right) \cdot C_A^3 \quad (11)$$

$$\frac{dT}{dt} = \frac{1}{\rho \cdot C_p} \left(-A \cdot \exp\left(-\frac{E_A}{RT}\right) \cdot C_A^3 \cdot \Delta H_r \right) \quad (12)$$

Thus, by using the two equations above, we were able to define a function for the system as can be seen in Listing 5

```

1 #Initialize the variables
2 A = 3.1
3 rho = 1.25 # kg * m^-3
4 Cp = 1200 # J * kg^-1 * K^-1
5 deltaH = -18400 # we need to keep units constant thus J * mol^-1
6 Ea = 25800 # Same logic as above because 8.3145 will be used for R
   thus J * mol^-1
7 c0 = 10
8 T0 = 313
9 R = 8.3145
10
11
12 def der_system_task6(t, x):
13     '''Solves the following system of equations: \n
14     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
15     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
16     t = vector for time span \n
17     x = vector where first position is the concentration and second is
       the temperature \n'''
18     dxdt = np.zeros(2) # two equations
19     dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents
       dcadt
20     dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*
       Cp) #Represents dTdt
21     return dxdt

```

Listing 5: Implementation of system of differential equation solver

Once this function has been implemented, we can solve the system using the "solve_ivp" function in the following way.

```

1 ini_cond_vec = [c0, T0] # Initial conditions a vector
2 tspan = [0,1000]
3 solution = scipy.integrate.solve_ivp(der_system_task6, tspan,
   ini_cond_vec)

```

Listing 6: Solving the system

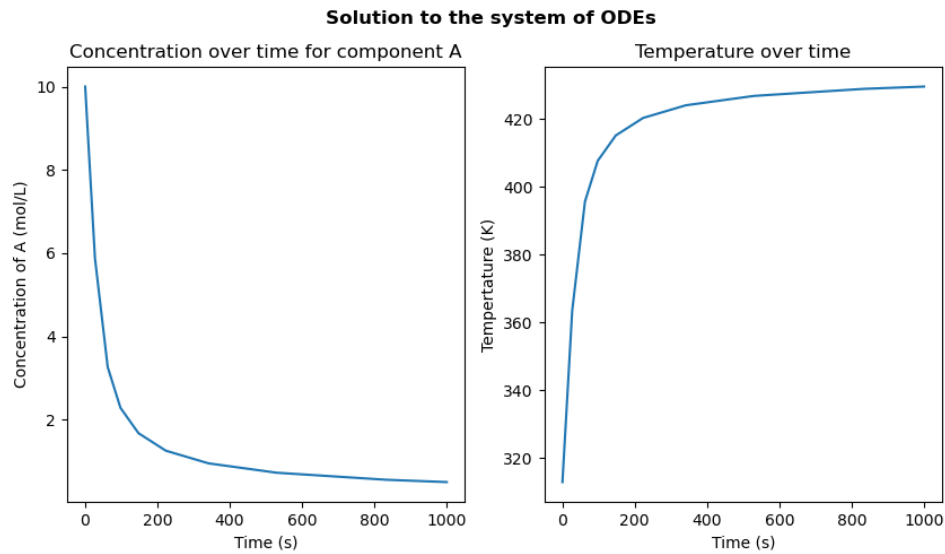


Figure 7: Visualization of the solution to the system of differential equations

Figure 7 provides a visual illustration to the system of differential equations. This solution aligns with expectations as the concentration of component A decreases over time and approaches zero. The temperature appears to keep increasing which at first may appear strange, however, as no heat losses are considered in the system it makes sense that the temperature continues rising.

7. For this task a slight modification was made to the existing first-order Euler solver to now allow for two derivatives. This modification can be seen in Listing 7.

```

1 def first_order_euler_system(fun, tspan, y0, number_of_points=100):
2     '''Applies the first order explicit euler method on the system of
3       two differential equations for task 7. \n
4       fun = function
5       y0 = vector of initial conditions
6       optional:\n
7       number_of_points = how many steps. Default set to 100. Increasing
8         this reduces error but increases computation time. '''
9     dt = (tspan[1] - tspan[0])/number_of_points
10    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
11    y = np.zeros((number_of_points+1, 2))
12    y[0,0] = y0[0] ## Setting up the initial conditions
13    y[0,1] = y0[1]
14    for i in range(number_of_points):
15        y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
16    return t, y

```

Listing 7: Updated Euler Function

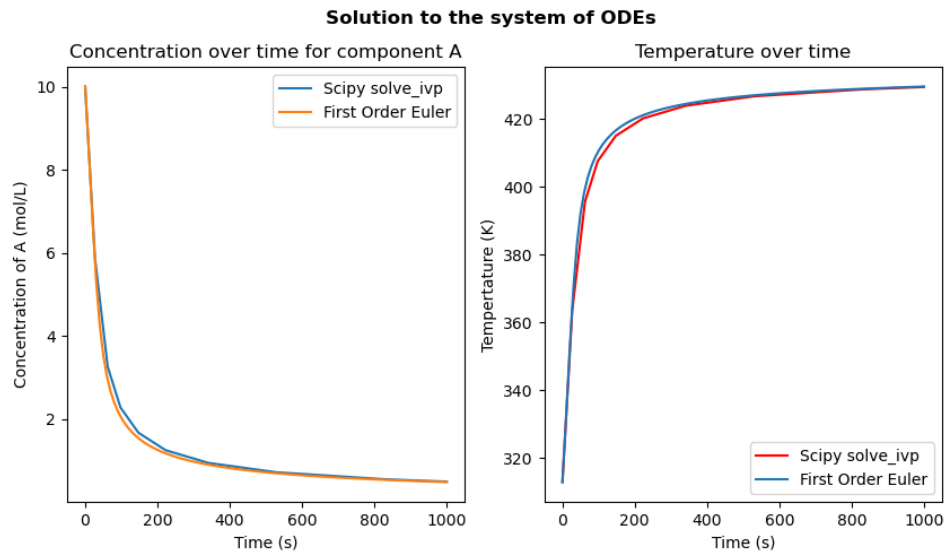


Figure 8: Two methods to solve the system of ODEs

Using the new function, the system could once again be solved. This new solution was plotted on the same axis as the previous "solve_ivp" function to compare its accuracy as can be seen in Figure 8. Although the curve appears to be "smooth" than the scipy solution, it is consistently closer to the left-hand side meaning it has some degree of error. In the following tasks, by using higher-order methods, this error was greatly reduced.

8. For this task, initially only the midpoint method was implemented. The midpoint method was chosen as the higher-order method of choice as it appeared to be a natural continuation of the Euler method. Listing 8 shows how the initial function was implemented.

```

1 def midpoint_rule(fun,tspan, y0, number_of_points=100):
2     '''Applies the midpoint rule on the system of two differential
3         equations for task 7. \n
4     fun = function
5     y0 = vector of initial conditions
6     optional:\n
7     number_of_points = how many steps. Default set to 100. Increasing
8         this reduces error but increases computation time. '''
9     dt = (tspan[1] - tspan[0])/number_of_points
10    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
11    y = np.zeros((number_of_points+1, 2))
12    y[0,0] = y0[0] ## Setting up the initial conditions
13    y[0,1] = y0[1]
14    for i in range(number_of_points):
15        k1 = fun(t[i], y[i,:])
16        k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
17        y[i+1,:] = y[i,:] + dt * k2
18    return t, y

```

Listing 8: Initial Midpoint Method

Although this function worked properly for this task, we wanted a bigger challenge. Hence, a mas-

ter function was created which can be seen in Listing 9. However, the midpoint method remained as the solution method of choice for this task.

```

1 def master_function(fun,tspan, y0, method='rk2', number_of_points=100)
2 :
3     '''General function to solve system of differential equations.
4     Does not work on single differential equations. \n
5     fun = function
6     y0 = vector of initial conditions
7     optional:\n
8     method = You can select the method with which your system of
9     differential equations will be evaluated. Default set to second
10    order Runge-Kutta. \n
11    Supported methods : midpoint method ('midpoint'), euler method ('
12    euler'), Classical second order Runge-Kutta ('rk2'), classical
13    fourth order Runge-Kutta ('rk4'). \n
14    number_of_points = how many steps. Default set to 100. Increasing
15    this reduces error but increases computation time. '''
16    dt = (tspan[1] - tspan[0])/number_of_points
17    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
18    y = np.zeros((number_of_points+1, len(y0))) # len(y0) because you
19    would need an initial condition for each derivative.
20    for i in range(len(y0)): #initial conditions as a loop to ensure
21    universability.
22        y[0,i] = y0[i]
23    if method == 'midpoint':
24        for i in range(number_of_points):
25            k1 = fun(t[i], y[i,:])
26            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
27            y[i+1,:] = y[i,:] + dt * k2
28    elif method == 'euler':
29        for i in range(number_of_points):
30            y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
31    elif method == 'rk2':
32        for i in range(number_of_points):
33            k1 = fun(t[i], y[i,:])
34            k2 = fun(t[i] + dt, y[i] + dt*k1)
35            y[i+1,:] = y[i] + dt*0.5*(k1+k2)
36    elif method == 'rk4':
37        for i in range(number_of_points):
38            k1 = fun(t[i], y[i,:])
39            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
40            k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
41            k4 = fun(t[i] +dt, y[i,:] + dt*k3)
42            y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
43    else:
44        return 'Unknown method specified. Check documentation for
45        supported methods' # In case an unknown method is specified
46    return t, y

```

Listing 9: Master function for systems of differential equations

The initial midpoint method function had some limitations such as being able to only solve systems

of two differential equations. In the master function, this limitation was fixed, however, a limitation remains that this master function is only able to solve systems of differential equations and not single differential equations. In the master function, the default method was chosen to be the "Runge-Kutta 2" method to illustrate in the code that the method can be chosen. Finally, the use of if, elif, and else statements ensured that should the chosen method be undefined, a clear error message is returned.

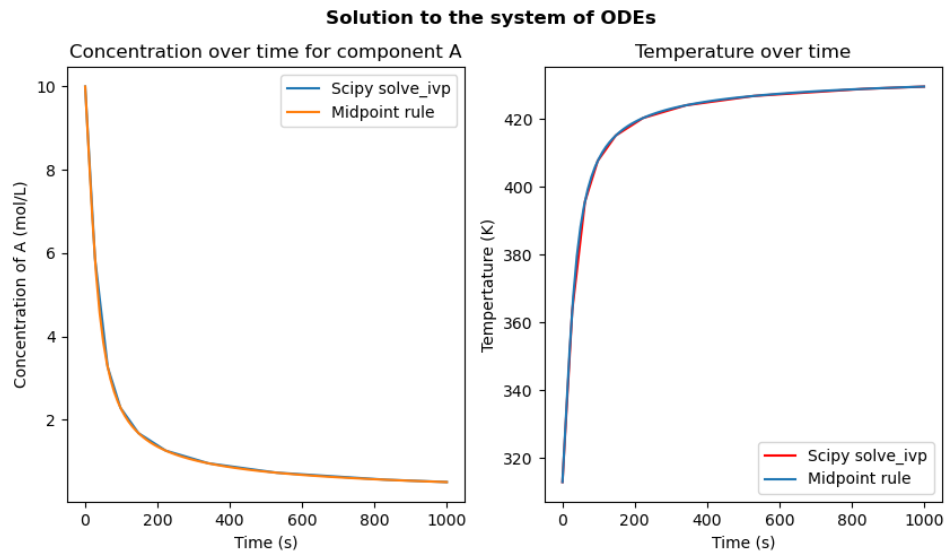


Figure 9: Solution to the system of ODEs with midpoint method

Figure 9 shows the solution of the system of ODEs compared to the solution provided by the scipy function. When comparing this to Figure 8, one can observe that the higher-order method more accurately approximates the function with the same number of steps.

9. Similar to the previous tasks the midpoint method was characterized. Due to the limitation mentioned in the previous task, a simplified function was used for the characterization.

```

1 def midpoint_rule(fun,tspan, y0, number_of_points=100):
2     '''Applies the midpoint rule on the system of two differential
3         equations for task 7. \n
4     fun = function
5     y0 = vector of initial conditions
6     optional:\n
7     number_of_points = how many steps. Default set to 100. Increasing
8         this reduces error but increases computation time. '''
9     dt = (tspan[1] - tspan[0])/number_of_points
10    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
11    y = np.zeros(number_of_points+1)
12    y[0] = y0 ## Setting up the initial conditions
13    for i in range(number_of_points):
14        k1 = fun(t[i], y[i])
15        k2 = fun(t[i] + dt*0.5, y[i] + 0.5*dt*k1)
16        y[i+1] = y[i] + dt * k2
17    return t, y

```

Listing 10: Simplified midpoint method for characterization

Concentration of A over time with initial concentration of 5 mol/L

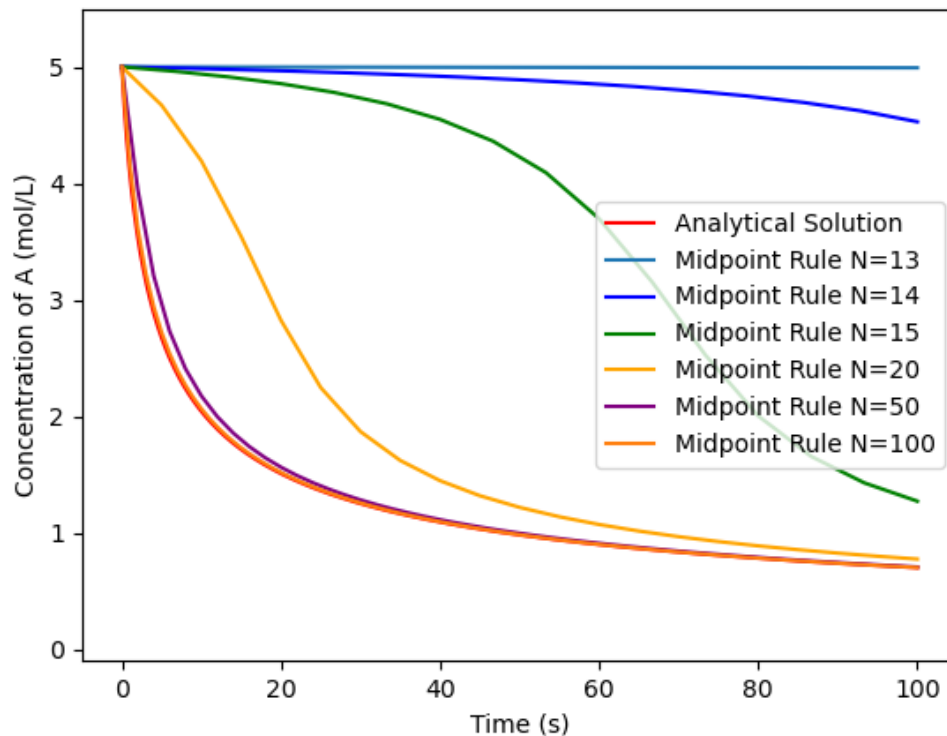


Figure 10: Minimum number of points required for the midpoint rule

Figure 10 shows the midpoint rule applied to the original ODE. It can be observed that using $N=13$ ($\Delta t \approx 7.69s$) returns a straight line which is completely nonsensical in regards to the analyti-

cal solution. Using $N=14$ ($\Delta t \approx 7.14s$) appears to begin getting closer to the expected solution, however, with the chosen time-span, using $N=14$ would not return proper results. Using $N=15$ ($\Delta t \approx 6.67s$) begins to have the correct shape, however, beyond this, the shape of the approximated solution more closely approximates the analytical solution. Finally using $N=100$ ($\Delta t \approx 1s$), the curve nearly directly overlaps the analytical solution.

Table 2: Analysis of the Midpoint method

N_t	Calculated Solution	$\epsilon_{relative}$	Rate of Convergence
100	2.886808	1.98E-05	N/A
11200	2.886751	1.55E-09	2.003615
22300	2.886751	3.91E-10	2.000111
33400	2.886751	1.74E-10	2.000167
44500	2.886751	9.82E-11	1.99979
55600	2.886751	6.29E-11	1.999723
66700	2.886751	4.37E-11	2.000559
77800	2.886751	3.21E-11	2.002302
88900	2.886751	2.46E-11	1.995259
100000	2.886751	1.94E-11	2.007853

Table 2 shows that the rate of convergence is 2 which aligns with expectations. Moreover, when compared to the Euler method in Table 1 we can observe that the absolute error is several orders of magnitude lowered compared to that of the midpoint method.

10. In order to solve this task, first, the system of differential equations had to be implemented. As can be seen in Listing 11, the variable delta was also defined to reduce calculations complexity and increase the readability of the code.


```

1 l1_value = 1
2 l2_value = 1
3 def der_system_task10(t, x, m1=1, m2=1, l1=l1_value, l2=l2_value):
4     '''Solves the system of differential equations for a double
5         pendulum \n
6         t = vector for time span \n
7         x = vector (theta1, omega1, theta2, omega2) \n'''
8     dxdt = np.zeros(4) # 4 equations
9     delta = x[2] - x[0]
10    dxdt[0] = x[1]
11    dxdt[1] = (m2 * l1 * (x[1]**2) * np.sin(delta)*np.cos(delta) + m2
12              * 9.81*np.sin(x[2])*np.cos(delta) + m2*l2*(x[3]**2) * np.sin(
13              delta) - (m1+m2)*9.81*np.sin(x[0]))/((m1+m2)*l2 - m2*l2*(np.cos(
14              delta)**2)
15    dxdt[2] = x[3]
16    dxdt[3] = (-m2*l2*(x[3]**2)*np.sin(delta)*np.cos(delta) + (m1+m2)
17              *(9.81*np.sin(x[0])*np.cos(delta) - l1 * (x[1]**2) * np.sin(
18              delta) - 9.81 * np.sin(x[2])))/((m1+m2)*l2 - m2*l2*(np.cos(
19              delta)**2)
20    return dxdt

```

Listing 11: System of ODEs for double pendulum

Following this, the system could be solved using the master function introduced in task 8. For this task, the "RK4" method was selected as we wanted to ensure that the solution was stable as it was observed that at times using the Euler method would give an unstable solution.

```

1 #(theta1, omega1, theta2, omega2)
2 ini_cond_vec = [7*np.pi/180, 1, 0.89*np.pi, 1] # Initial conditions a
3 vector
4 tspan = [0,10]
5 approx_sol = master_function(der_system_task10, tspan, ini_cond_vec,
6                             method='rk4')

```

Listing 12: Solving the system

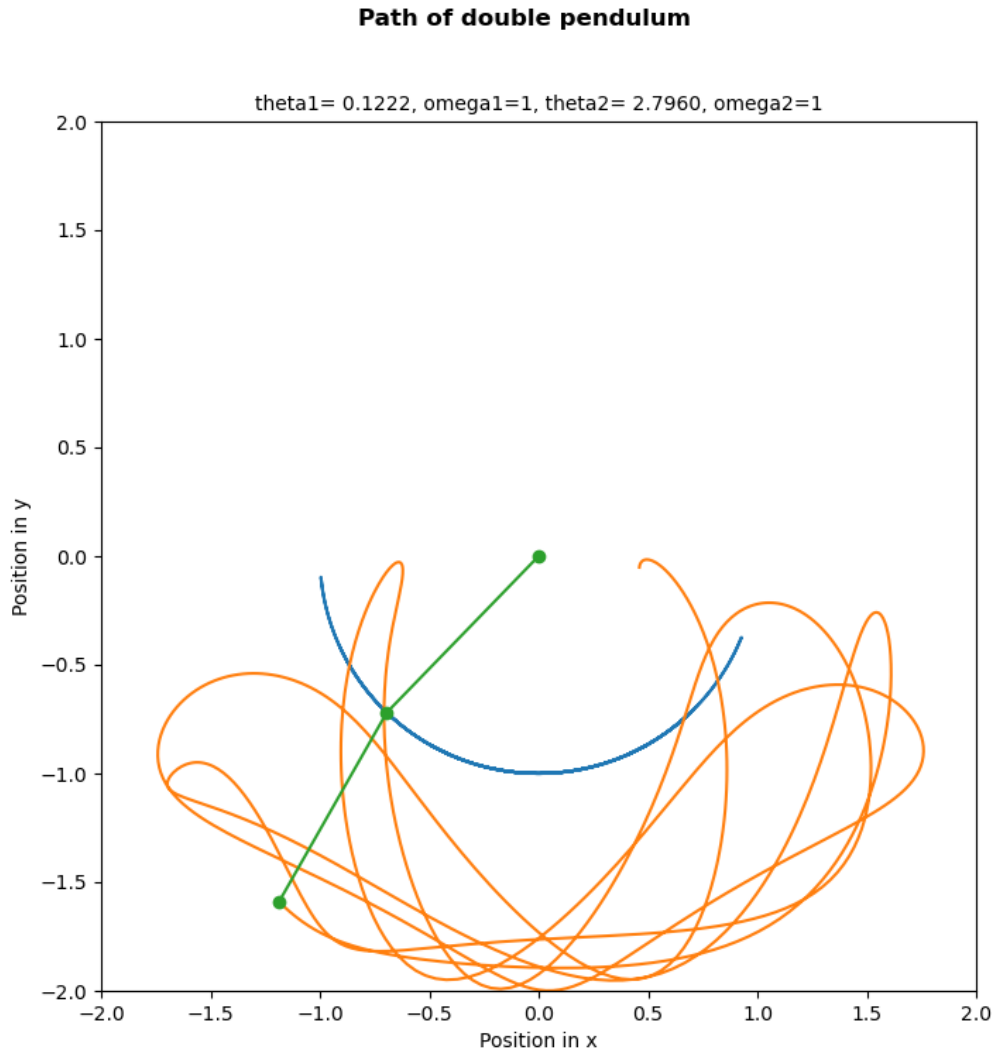


Figure 11: Final position of the double pendulum

After solving the system the solution was plotted. [Figure 11](#) shows the path taken by the first node of the pendulum in blue and the path of the second pendulum in orange. The green curve is an animated curve showing the position of both nodes. In order to see the animation, one must run the code from the file "task 10.py". Listing [13](#) shows how the plot was generated along with the implementation of the animation. In order to plot the first node, we multiplied the sine of the first angle (θ_1) to get the x -coordinate and the negative of the cosine to get the y -coordinate. Analogously, by adding the sine and negative cosine of the second angle (θ_2) we were able to obtain the x and y coordinate of the second node respectively.

```

1 x_1st = np.sin(approx_sol[1][:,0])*l1_value #Like the first node of
  the double pendulum
2 y_1st = -np.cos(approx_sol[1][:,0])*l1_value
3
4 x_2 = np.sin(approx_sol[1][:,0])*l1_value + np.sin(approx_sol[1][:,2])
  *l2_value
5 y_2 = -np.cos(approx_sol[1][:,0])*l1_value - np.cos(approx_sol
  [1][:,2])*l2_value
6
7 x_connector = [0, x_1st[-1], x_2[-1]]
8 y_connector = [0, y_1st[-1], y_2[-1]]
9
10
11 fig, ax = plt.subplots(figsize=(8,8))
12 plt.plot(x_1st, y_1st, label='Path of second node')
13 plt.plot(x_2, y_2, label='Path of second node')
14 plt.xlabel('Position in x')
15 plt.ylabel('Position in y')
16 plt.title(f'theta1={ini_cond_vec[0]: .4f}, omega1={ini_cond_vec[1]},
  theta2={ini_cond_vec[2]: .4f}, omega2={ini_cond_vec[3]}', fontsize
  =10)
17 plt.suptitle(f'Path of double pendulum', weight='bold')
18 plt.xlim(-2,2)
19 plt.ylim(-2,2)
20
21
22 #Animating the path taken
23 line, = ax.plot(x_1st, y_1st, marker='o')
24
25 plt.show(block=False)
26 x_connector = [0, x_1st[-1], x_2[-1]]
27 y_connector = [0, y_1st[-1], y_2[-1]]
28
29 for i in range(len(approx_sol[1][:,0])):
30     line.set_data([0, x_1st[i], x_2[i]], [0, y_1st[i], y_2[i]])
31     fig.canvas.draw()
32     fig.canvas.flush_events()

```

Listing 13: Plotting the path of the double pendulum

11. The equations below show how to calculate the kinetic and potential energy of a double pendulum. Both equations were obtained from [1]:

$$KE = \frac{1}{2} (m_1 + m_2) L_1^2 \left(\frac{d\theta_1}{dt} \right)^2 + \frac{1}{2} m_2 L_2^2 \left(\frac{d\theta_2}{dt} \right)^2 + m_2 L_1 L_2 \frac{d\theta_1}{dt} \frac{d\theta_2}{dt} \cos(\theta_1 - \theta_2) \quad (13)$$

$$PE = -(m_1 + m_2) g L_1 \cos(\theta_1) - m_2 g L_2 \cos(\theta_2) \quad (14)$$

```

1 # from:
2 # http://www.physics.umd.edu/hep/drew/pendulum2.html
3 # We can get the equations for the kinetic and potential energy
4 # the time derivative in terms of theta1 is simply omega1
5
6 KE = 0.5 * (m1_value+m2_value) * l1_value**2 * approx_sol[1][:,1]**2 +
7     0.5 * m2_value* l2_value**2 * approx_sol[1][:,3]**2 + m2_value*
8     l1_value*l2_value * approx_sol[1][:,1] * approx_sol[1][:,3] * np.
9     cos(approx_sol[1][:,0] - approx_sol[1][:,2])
10
11 PE = -(m1_value+m2_value) * 9.81 * l1_value * np.cos(approx_sol[1][:,0])
12     - m2_value*9.81*l2_value*np.cos(approx_sol[1][:,2])
13
14 ## Correcting for the zero of energy
15 PE += (m1_value+m2_value)*9.81*(l1_value+l2_value)

```

Listing 14: Calculation of the kinetic and potential energies a double pendulum

Listing 14 shows the implementation of the kinetic and potential energy calculation. As lowest y position of the pendulum is -2 (with both lengths equal to 1) the potential energy must be corrected to correct for the zero of energy.

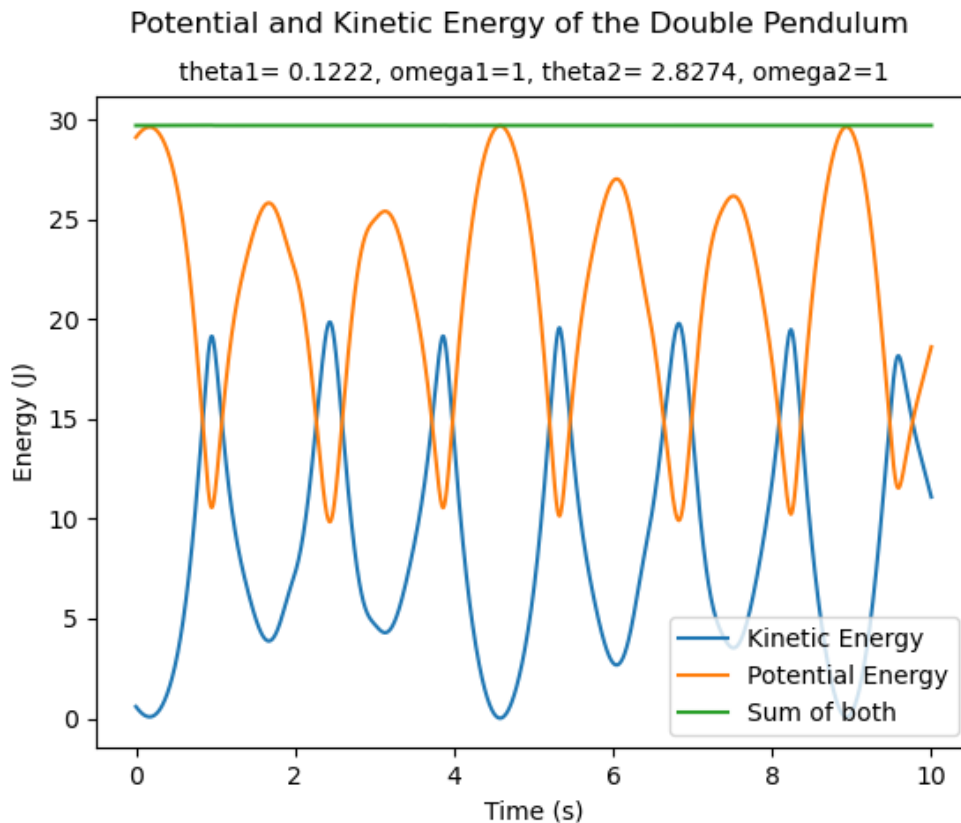


Figure 12: Energy of the double pendulum system

As can be seen in Figure 12, the energy of the double pendulum system remains constant over time. Moreover, due to the potential energy correction, the energy remains strictly positive aligning with expectations.

Path of double pendulum at various values of theta2

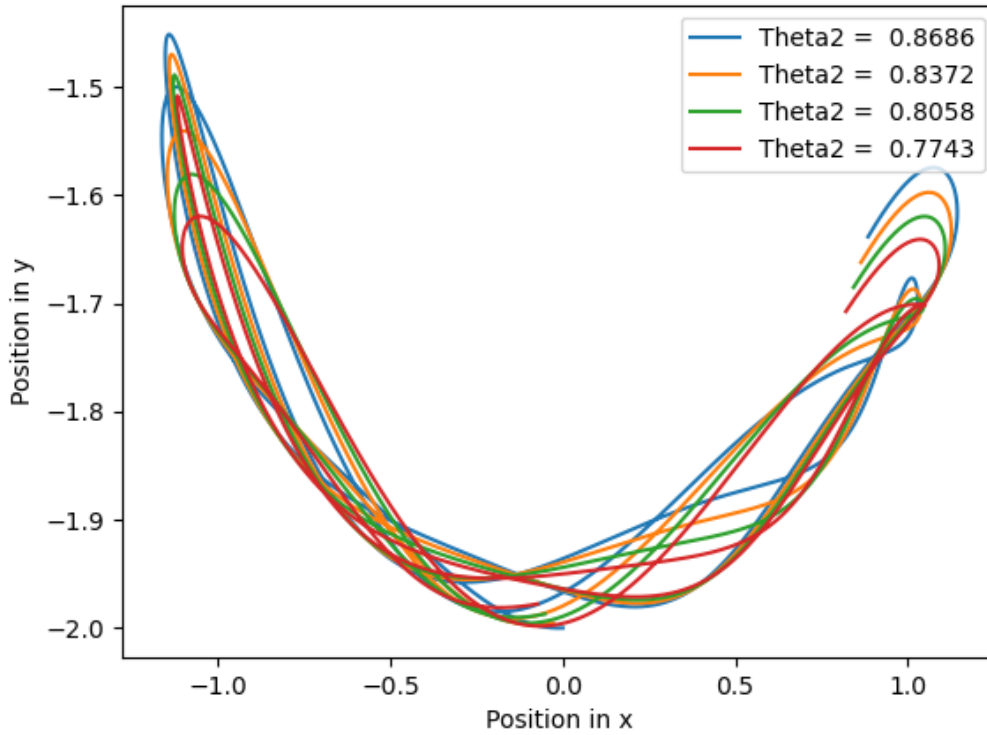


Figure 13: Path taken by second node at various θ_2 values

Figure 13 shows the effect of lowering the value of θ_2 has on the path taken by the second node of the double pendulum system. As this report is static, the time-span must be lowered to not overwhelm the plotted graph. However, even with a low time-span (from 0 to 5 seconds), the effect of a small variation can be observed.

3 Reflection

In conclusion, this assignment was a success. The black-box method of "scipy.integrate.solve_ivp" was elucidated. Investigating the rate of convergence aligned with expectations and it was observed that higher-order methods increase the accuracy of the solution to differential equations as well as increase stability. The "butterfly" effect was also investigated and it was observed that in the case of a double pendulum, a very slight modification in the initial conditions leads to a very different final path of the pendulum.

References

- [1] Drew Baden, <http://www.physics.umd.edu/hep/drew/pendulum2.html>, May 2016.

List of symbols

Symbol	Unit	Definition
C_i	mol L^{-1}	Concentration of component i
k	$\text{L}^2 \text{mol}^{-2} \text{s}$	Reaction rate constant of third order chemical reaction with a value of 0.01
t	s	Time
T	K	Temperature
E_a	J mol^{-1}	Activation energy
ρ	$\text{kg}^3 \text{m}^{-1}$	Density
C_p	$\text{J kg}^{-1} \text{K}$	Heat capacity at constant pressure
ΔH_r	J mol	Change in enthalpy
A	Unitless	Pre-exponential factor

```

1 # Third order so:
2 # rate = - k * (C_A)^3
3 # Negative sign because its being consumed
4 def third_order_derivative(t, Ca, k=0.01): # I know that the
    assignemnt does not require us to add k as an argument but I wanted
    the function to be as universal as possible
5     '''Returns the concentration over time for a third order chemical
        reaction.
6     t = time in seconds
7     Ca = concentration of component a (a -> b)
8     k = reaction rate constant (default set to 0.01 as this is the
        value for task 1.)'''
9     dca_dt = - k * Ca**3
10    return dca_dt

```

codes/Task 1.py

```

1 def third_order_analytical(t, c0, k3):
2     '''Returns the concentration at a given time for a third order
3     irreversible chemical reation.
4     t = Time in seconds. If a vector it will retrun a vector of
5     solutions
6     c0 = initial concentration.
7     k3 = reaction rate constant for the third order reaction.'''
8     Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
9     return Ca
10
11 ## You can feed the function t as a vector (like an np.array) and it
12    will return a vector with the Ca at those times.

```

codes/Task 2.py


```

1 import scipy.integrate
2 import matplotlib.pyplot as plt ## For plotting
3 import numpy as np
4
5 # Functions from previous sections
6 def third_order_derivative(t, Ca, k3=0.01): # I know that the
    assignemnt does not require us to add k as an argument but I wanted
    the function to be as universal as possible
7     '''Returns the concentration over time for a third order chemical
        reaction.
8     t = time in seconds
9     Ca = concentration of component a (a -> b)
10    k = reaction rate constant (default set to 0.01 as this is the
        value for task 1.)'''
11    dca_dt = - k3 * Ca**3
12    return dca_dt
13
14 def third_order_analytical(t, c0, k3):
15     '''Returns the concentration at a given time for a third order
        irreversible chemical reation.
16     t = Time in seconds. If a vector it will retrun a vector of
        solutions
17     c0 = initial concentration.
18     k3 = reaction rate constant for the third order reaction.'''
19     Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
20     return Ca
21
22 # Initializing variables
23 initial_conc = 5
24 t_max = 100 # not sure what a 'reasonable' time frame is but at t_max
    =100 we can see the overall shape of the curve and that it tends
    towards 0
25 x_plotting = np.linspace(0, t_max, 1000) # For generating the output
    figure for the analytical solution
26 y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
27 y_scipy = scipy.integrate.solve_ivp(lambda t, Ca:
    third_order_derivative(t, Ca), [0, t_max], [initial_conc])
28 y_scipy_plotting = y_scipy.y[0]
29 x_scipy_plotting = y_scipy.t # the 't' variable corresponds to the x
    axis
30
31 #Figure output code
32 fig = plt.figure()
33 plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
34 plt.plot(x_scipy_plotting, y_scipy_plotting, 'b', label='Scipy
    solve_ivp')
35 plt.xlabel('Time (s)')
36 plt.ylabel('Concentration of A (mol/L)')
37 plt.title(f'Concentration of A over time with initial concentration of
    {initial_conc} mol/L', weight='bold')
38 plt.legend()
39 plt.show()
40 ## The figure above shows that they are almost identical

```

```

41
42
43 ## We can restrict the domain to see the deviations
44 fig2 = plt.figure()
45 plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
46 plt.plot(x_scipy_plotting, y_scipy_plotting, 'b', label='Scipy
    solve_ivp')
47 plt.xlabel('Time (s)')
48 plt.ylabel('Concentration of A (mol/L)')
49 plt.title(f'Closer look at the two solution methods', weight='bold')
50 plt.xlim(35, 61)
51 plt.ylim(0.85, 1.2)
52 plt.legend()
53 plt.show()
54 #Slight deviations but we need to zoom in quite close to see
55
56 # For difference over time
57 y_analytical_scipyxvalues = third_order_analytical(x_scipy_plotting,
    initial_conc, 0.01)
58 # New calculation because scipy does not use a linspace
59
60
61 diff = []
62
63 for i in range(len(x_scipy_plotting)):
64     difference = y_analytical_scipyxvalues[i] - y_scipy_plotting[i]
65     diff.append(difference)
66
67 fig3 = plt.figure(figsize=(8,7))
68 plt.plot(x_scipy_plotting, diff)
69 plt.xlabel('Time(s)')
70 plt.ylabel('Analytical - solve_ivp')
71 plt.title('Difference between analytical and numerical solution',
    weight='bold')
72 plt.show()
73
74 print(f'Maximum deviation between analytical and Scipy solution: {np.
    abs(min(diff))}')
75 ## Only prints the absolute value.

```

codes/Task 3.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 def first_order_euler(fun, tspan, y0, number_of_points=100_000): # 100
    _000 corresponds to to a dt of 0.001 as asked in the question
4     '''Explicit first order euler solver for ODEs.\n
5     fun = derivative function \n
6     tspan = supply the time span as a vector ie [0,10] \n
7     y0 = initial condition ie at t=0 y0=100 \n
8     number_of_points = Number of points. Default set to 100000 in
        order to have dt = 0.001. Higher number means longer
        computation time'''
9     dt = (tspan[1] - tspan[0])/number_of_points
10    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
11    y = np.zeros(number_of_points+1)
12    y[0] = y0 ## Setting up the initial conditions
13    for i in range(number_of_points):
14        y[i+1] = y[i] + dt * fun(t[i], y[i])
15    return t, y
16
17 #From task 1
18 # I know I could use something like "from Task 1 import
    third_order_derivative" but I want to make sure that the code runs
    on all devices always.
19 # For some people importing from other files doesn't work.
20
21 def third_order_derivative(t, Ca, k=0.01):
22     '''Returns the concentration over time for a third order chemical
        reaction.
23     t = time in seconds
24     Ca = concentration of component a (a -> b)
25     k = reaction rate constant (default set to 0.01 as this is the
        value for task 1.)'''
26     dca_dt = - k * Ca**3
27     return dca_dt
28
29
30 def third_order_analytical(t, c0, k3):
31     '''Returns the concentration at a given time for a third order
        irreversible chemical reation.
32     t = Time in seconds. If a vector it will retrun a vector of
        solutions
33     c0 = initial concentration.
34     k3 = reaction rate constant for the third order reaction.'''
35     Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
36     return Ca
37
38 # Same as task 3
39 initial_conc = 5
40 t_max = 100 # For tmax = 100, the number of points must be 100 000 to
    have a dt of 0.001
41 x_plotting = np.linspace(0, t_max, 1000)
42 y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
43 solution_euler = first_order_euler(third_order_derivative, [0, t_max],

```

```

44     initial_conc)
45
46
47 fig = plt.figure()
48 plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
49 plt.plot(solution_euler[0], solution_euler[1], 'b', label='First Order
    Euler') # [0] holds the x position and [1] holds the 'y' position
50 plt.xlabel('Time (s)')
51 plt.ylabel('Concentration of A (mol/L)')
52 plt.title(f'Concentration of A over time with initial concentration of
    {initial_conc} mol/L', weight='bold')
53 plt.legend()
54 plt.ylim(-0.1, 5.5)
55 plt.show()
56
57
58 plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
59 plt.plot(solution_euler[0], solution_euler[1], 'b', label='First Order
    Euler') # [0] holds the x position and [1] holds the 'y' position
60 plt.xlabel('Time (s)')
61 plt.ylabel('Concentration of A (mol/L)')
62 plt.title(f'Closer Look at the methods to observe slight deviations',
    weight='bold')
63 plt.legend()
64 plt.ylim(1.896, 1.897)
65 plt.xlim(11.8950, 11.9125)
66 plt.show()

```

codes/Task 4.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4
5 #This file computes the error and rate of convergence. The file 'Task
  5,2' shows the maximum time step for a stable solution
6
7 def first_order_euler(fun, tspan, y0, number_of_points=100000):
8     '''Explicit first order euler solver for ODEs.\n
9     fun = derivative function \n
10    tspan = supply the time span as a vector ie [0,10] \n
11    y0 = initial condition ie at t=0 y0=100 \n
12    number_of_points = Number of points. Default set to 100000 in
        order to have dt = 0.001. Higher number means longer
        computation time'''
13    dt = (tspan[1] - tspan[0])/number_of_points
14    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
15    y = np.zeros(number_of_points+1)
16    y[0] = y0 ## Setting up the initial conditions
17    for i in range(number_of_points):
18        y[i+1] = y[i] + dt * fun(t[i], y[i])
19    return t, y
20
21 def third_order_derivative(t, Ca, k=0.01):
22     '''Returns the concentration over time for a third order chemical
        reaction.
23     t = time in seconds
24     Ca = concentration of component a (a -> b)
25     k = reaction rate constant (default set to 0.01 as this is the
        value for task 1.)'''
26     dca_dt = - k * Ca**3
27     return dca_dt
28
29
30 def third_order_analytical(t, c0, k3):
31     '''Returns the concentration at a given time for a third order
        irreversible chemical reaction.
32     t = Time in seconds. If a vector it will retrun a vector of
        solutions
33     c0 = initial concentration.
34     k3 = reaction rate constant for the third order reaction.'''
35     Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
36     return Ca
37
38 # Based on the lecture slides we expect a first order rate of
    convergence
39
40 initial_conc = 5 # Same initial conc as before
41
42 N = np.linspace(100, 100000, 10, dtype=int) # We need dtype=int
    because we need an integer number of points for the function
43 solution_list = []
44 error_list = []

```

```

45 #all comparing at t=4
46 compare_time = 4
47 rate_of_convergence = ['N/A'] # First element is N/A because it doesnt
    make sense to calculate the rate of convergence for the first
    point
48 for i in range(len(N)):
49     approx_sol_general = first_order_euler(third_order_derivative, [0,
        4], initial_conc, N[i])
50     approx_sol = approx_sol_general[1][-1] # Gets the last y value
        corresponding to t = 4
51     exact_sol = third_order_analytical(4, initial_conc, 0.01)
52     solution_list.append(approx_sol)
53     rel_error = np.abs((approx_sol - exact_sol)/exact_sol) # Relative
        error calculation. No specific type of error was asked to
        calculate.
54     error_list.append(rel_error)
55     if i > 0: # Doesn't make sense to calculate for the first point
56         r = (np.log(error_list[i]/error_list[i-1]))/(np.log(N[i-1] / N
            [i]))
57         rate_of_convergence.append(r)
58
59 #Generating output table
60 rows = []
61 for i in range(len(N)):
62     rows.append([N[i], solution_list[i], error_list[i],
        rate_of_convergence[i]])
63
64 df = pd.DataFrame(rows, columns = ['N_t', 'Calculated solution', '
    epsilon_rel', 'Rate of Convergence'])
65 print(df)
66
67 # We can se that as Nt increases, the rate of convergence approaches
    1. This alligns with the expected results

```

codes/Task 5,1.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 #Functions from previous tasks
5 def first_order_euler(fun, tspan, y0, number_of_points=100000):
6     '''Explicit first order euler solver for ODEs.\n
7     fun = derivative function \n
8     tspan = supply the time span as a vector ie [0,10] \n
9     y0 = initial condition ie at t=0 y0=100 \n
10    number_of_points = Number of points. Default set to 100000 in
        order to have dt = 0.001. Higher number means longer
        computation time'''
11    dt = (tspan[1] - tspan[0])/number_of_points
12    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
13    y = np.zeros(number_of_points+1)
14    y[0] = y0 ## Setting up the initial conditions
15    for i in range(number_of_points):
16        y[i+1] = y[i] + dt * fun(t[i], y[i])
17    return t, y
18
19 def third_order_derivative(t, Ca, k=0.01): # I know that the
        assignemnt does not require us to add k as an argument but I wanted
        the function to be as universal as possible
20    '''Returns the concentration over time for a third order chemical
        reaction.
21    t = time in seconds
22    Ca = concentration of component a (a -> b)
23    k = reaction rate constant (default set to 0.01 as this is the
        value for task 1.)'''
24    dca_dt = - k * Ca**3
25    return dca_dt
26
27
28 def third_order_analytical(t, c0, k3):
29    '''Returns the concentration at a given time for a third order
        irreversible chemical reation.
30    t = Time in seconds. If a vector it will retrun a vector of
        solutions
31    c0 = initial concentration.
32    k3 = reaction rate constant for the third order reaction.'''
33    Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
34    return Ca
35
36 # Same as task 3
37 initial_conc = 5
38 t_max = 100
39 x_plotting = np.linspace(0, t_max, 1000)
40 y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
41 solution_euler_24 = first_order_euler(third_order_derivative, [0,
        t_max], initial_conc, 24) # dt ~= 4.1667
42 solution_euler_25 = first_order_euler(third_order_derivative, [0,
        t_max], initial_conc, 25) # dt = 4
43 solution_euler_40 = first_order_euler(third_order_derivative, [0,

```

```

    t_max], initial_conc, 40)
44 solution_euler_80 = first_order_euler(third_order_derivative, [0,
    t_max], initial_conc, 80)
45
46
47 fig = plt.figure()
48 plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
49 plt.plot(solution_euler_24[0], solution_euler_24[1], 'b', label='First
    Order Euler N=24') # [0] holds the x position and [1] holds the 'y
    ' position
50 plt.plot(solution_euler_25[0], solution_euler_25[1], 'g', label='First
    Order Euler N=25') # Though it isnt perfect, at least now it
    doesnt go down to minus infinity.
51 plt.plot(solution_euler_40[0], solution_euler_40[1], 'orange', label='
    First Order Euler N=40')
52 plt.plot(solution_euler_80[0], solution_euler_80[1], 'purple', label='
    First Order Euler N=80')
53
54 plt.xlabel('Time (s)')
55 plt.ylabel('Concentration of A (mol/L)')
56 plt.title(f'Concentration of A over time with initial concentration of
    {initial_conc} mol/L', weight='bold')
57 plt.legend()
58 plt.ylim(-0.1, 5.5)
59 plt.show()

```

codes/Task 5,2.py


```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate
4
5 #Initialize the variables
6 A = 3.1
7 rho = 1.25 # kg * m^-3
8 Cp = 1200 # J * kg^-1 * K^-1
9 deltaH = -18400 # we need to keep units constant thus J * mol^-1
10 Ea = 25800 # Same logic as above because 8.3145 will be used for R
    thus J * mol^-1
11 c0 = 10
12 T0 = 313
13 R = 8.3145
14
15
16 def der_system_task6(t, x):
17     '''Solves the following system of equations: \n
18     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
19     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
20     t = vector for time span \n
21     x = vector where first position is the concentration and second is
        the temperature \n'''
22     dxdt = np.zeros(2) # two equations
23     dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents
        dcadt
24     dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*
        Cp) #Represents dTdt
25     return dxdt
26
27 ini_cond_vec = [c0, T0] # Initial conditions a vector
28 tspan = [0,1000]
29 solution = scipy.integrate.solve_ivp(der_system_task6, tspan,
    ini_cond_vec)
30
31 fig = plt.figure(figsize=(10,5))
32 ax1 = plt.subplot(1,2,1)
33 ax2 = plt.subplot(1,2,2)
34
35 ax1.plot(solution.t, solution.y[0])
36 ax2.plot(solution.t, solution.y[1])
37 ax1.set_ylabel('Concentration of A (mol/L)')
38 ax1.set_xlabel('Time (s)')
39 ax1.set_title('Concentration over time for component A')
40 ax2.set_title('Temperature over time')
41 ax2.set_xlabel('Time (s)')
42 ax2.set_ylabel('Tempertature (K)')
43 fig.suptitle('Solution to the system of ODEs', weight='bold')
44
45 plt.show()

```

codes/Task 6.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate
4
5 # Taking the code from task 6
6 #Initialize the variables
7
8 A = 3.1
9 rho = 1.25 # kg * m^-3
10 Cp = 1200 # J * kg^-1 * K^-1
11 deltaH = -18400 # we need to keep units constant thus J * mol^-1
12 Ea = 25800 # Same logic as above because 8.3145 will be used for R
    thus J * mol^-1
13 c0 = 10
14 T0 = 313
15 R = 8.3145
16
17
18 def der_system_task6(t, x):
19     '''Solves the following system of equations: \n
20     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
21     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
22     t = vector for time span \n
23     x = vector where first position is the concentration and second is
        the temperature \n'''
24     dxdt = np.zeros(2) # two equations
25     dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents
        dcadt
26     dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*
        Cp)
27     return dxdt
28
29
30 def first_order_euler_system(fun,tspan, y0, number_of_points=100):
31     '''Applies the first order explicit euler method on the system of
        two differential equations for task 7. \n
32     fun = function
33     y0 = vector of initial conditions
34     optional:\n
35     number_of_points = how many steps. Default set to 100. Increasing
        this reduces error but increases computation time. '''
36     dt = (tspan[1] - tspan[0])/number_of_points
37     t = np.linspace(tspan[0], tspan[1], number_of_points+1)
38     y = np.zeros((number_of_points+1, 2))
39     y[0,0] = y0[0] ## Setting up the initial conditions
40     y[0,1] = y0[1]
41     for i in range(number_of_points):
42         y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
43     return t, y
44
45 ini_cond_vec = [c0, T0] # Initial conditions a vector
46 tspan = [0,1000]
47

```

```

48 approx_sol = first_order_euler_system(der_system_task6, tspan,
    ini_cond_vec)
49
50 solution = scipy.integrate.solve_ivp(der_system_task6, tspan,
    ini_cond_vec)
51
52 fig = plt.figure(figsize=(10,5))
53 ax1 = plt.subplot(1,2,1)
54 ax2 = plt.subplot(1,2,2)
55
56 ax1.plot(solution.t, solution.y[0], label='Scipy solve_ivp')
57 ax1.plot(approx_sol[0], approx_sol[1][:,0], label='First Order Euler')
58 ax1.set_ylabel('Concentration of A (mol/L)')
59 ax1.set_xlabel('Time (s)')
60 ax1.set_title('Concentration over time for component A')
61 ax1.legend()
62 ax2.plot(solution.t, solution.y[1], 'r', label='Scipy solve_ivp')
63 ax2.plot(approx_sol[0], approx_sol[1][:,1], label='First Order Euler')
64 ax2.set_title('Temperature over time')
65 ax2.set_xlabel('Time (s)')
66 ax2.set_ylabel('Tempertature (K)')
67 fig.suptitle('Solution to the system of ODEs', weight='bold')
68 ax2.legend()
69 plt.show()

```

codes/Task 7.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate
4
5 # Taking the code from task 6
6 #Initialize the variables
7
8 A = 3.1
9 rho = 1.25 # kg * m^-3
10 Cp = 1200 # J * kg^-1 * K^-1
11 deltaH = -18400 # we need to keep units constant thus J * mol^-1
12 Ea = 25800 # Same logic as above because 8.3145 will be used for R
    thus J * mol^-1
13 c0 = 10
14 T0 = 313
15 R = 8.3145
16
17
18
19 def der_system_task6(t, x):
20     '''Solves the following system of equations: \n
21     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
22     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
23     t = vector for time span \n
24     x = vector where first position is the concentration and second is
        the temperature \n'''
25     dxdt = np.zeros(2) # two equations
26     dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents
        dcadt
27     dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*
        Cp)
28     return dxdt
29
30 # For the challenge the master function is provided below. The default
    method is set to the rk2 method.
31 # Check the documentation for the supported methods
32 def master_function(fun,tspan, y0, method='rk2', number_of_points=100)
    :
33     '''General function to solve system of differential equations.
        Does not work on single differential equations. \n
34     fun = function
35     y0 = vector of initial conditions
36     optional:\n
37     method = You can select the method with which your system of
        differential equations will be evaluated. Default set to second
        order Runge-Kutta. \n
38     Supported methods : midpoint method ('midpoint'), euler method ('
        euler'), Classical second order Runge-Kutta ('rk2'), classical
        fourth order Runge-Kutta ('rk4'). \n
39     number_of_points = how many steps. Default set to 100. Increasing
        this reduces error but increases computation time. '''
40     dt = (tspan[1] - tspan[0])/number_of_points
41     t = np.linspace(tspan[0], tspan[1], number_of_points+1)

```

```

42 y = np.zeros((number_of_points+1, len(y0))) # len(y0) because you
    would need an initial condition for each derivative.
43 for i in range(len(y0)): #initial conditions as a loop to ensure
    universability.
44     y[0,i] = y0[i]
45     if method == 'midpoint':
46         for i in range(number_of_points):
47             k1 = fun(t[i], y[i,:])
48             k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
49             y[i+1,:] = y[i,:] + dt * k2
50     elif method == 'euler':
51         for i in range(number_of_points):
52             y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
53     elif method == 'rk2':
54         for i in range(number_of_points):
55             k1 = fun(t[i], y[i,:])
56             k2 = fun(t[i] + dt, y[i] + dt*k1)
57             y[i+1,:] = y[i] + dt*0.5*(k1+k2)
58     elif method == 'rk4':
59         for i in range(number_of_points):
60             k1 = fun(t[i], y[i,:])
61             k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
62             k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
63             k4 = fun(t[i] +dt, y[i,:] + dt*k3)
64             y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
65     else:
66         return 'Unknown method specified. Check documentation for
            supported methods' # In case an unknown method is specified
67     return t, y
68
69 # Same plotting but now with the midpoint rule.
70 # Code copied from task 7
71
72
73
74 ini_cond_vec = [c0, T0] # Initial conditions a vector
75 tspan = [0,1000]
76
77
78 approx_sol = master_function(der_system_task6, tspan, ini_cond_vec, '
    midpoint')
79
80 solution = scipy.integrate.solve_ivp(der_system_task6, tspan,
    ini_cond_vec)
81
82 fig = plt.figure(figsize=(10,5))
83 ax1 = plt.subplot(1,2,1)
84 ax2 = plt.subplot(1,2,2)
85
86 ax1.plot(solution.t, solution.y[0], label='Scipy solve_ivp')
87 ax1.plot(approx_sol[0], approx_sol[1][:,0], label='Midpoint rule')
88 ax1.set_ylabel('Concentration of A (mol/L)')
89 ax1.set_xlabel('Time (s)')
90 ax1.set_title('Concentration over time for component A')

```

```

91 ax1.legend()
92 ax2.plot(solution.t, solution.y[1], 'r', label='Scipy solve_ivp')
93 ax2.plot(approx_sol[0], approx_sol[1][:,1], label='Midpoint rule')
94 ax2.set_title('Temperature over time')
95 ax2.set_xlabel('Time (s)')
96 ax2.set_ylabel('Tempertature (K)')
97 fig.suptitle('Solution to the system of ODEs', weight='bold')
98 ax2.legend()
99 plt.show()

```

codes/Task 8.py

```

1 #Taking the more or less the same code from task 1,2,3,5
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pandas as pd
5 import scipy.integrate
6
7 #Table print out (same as task 5.1)
8
9 def third_order_derivative(t, Ca, k=0.01): # I know that the
    assignemnt does not require us to add k as an argument but I wanted
    the function to be as universal as possible
10     '''Returns the concentration over time for a third order chemical
        reaction.
11     t = time in seconds
12     Ca = concentration of component a (a -> b)
13     k = reaction rate constant (default set to 0.01 as this is the
        value for task 1.)'''
14     dca_dt = - k * Ca**3
15     return dca_dt
16
17 def third_order_analytical(t, c0, k3):
18     '''Returns the concentration at a given time for a third order
        irreversible chemical reation.
19     t = Time in seconds. If a vector it will retrun a vector of
        solutions
20     c0 = initial concentration.
21     k3 = reaction rate constant for the third order reaction.'''
22     Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
23     return Ca
24
25 ## We need to slightly modify it so that it only solves for one method
26
27 ## We see a simplified version of the master function because we only
    solve one ODE at a time not a system
28 def midpoint_rule(fun,tspan, y0, number_of_points=100):
29     '''Applies the midpoint rule on the system of two differential
        equations for task 7. \n
30     fun = function
31     y0 = vector of initial conditions
32     optional:\n
33     number_of_points = how many steps. Default set to 100. Increasing
        this reduces error but increases computation time. '''
34     dt = (tspan[1] - tspan[0])/number_of_points
35     t = np.linspace(tspan[0], tspan[1], number_of_points+1)
36     y = np.zeros(number_of_points+1)
37     y[0] = y0 ## Setting up the initial conditions
38     for i in range(number_of_points):
39         k1 = fun(t[i], y[i])
40         k2 = fun(t[i] + dt*0.5, y[i] + 0.5*dt*k1)
41         y[i+1] = y[i] + dt * k2
42     return t, y
43
44

```

```

45 initial_conc = 5
46 t_max = 100 # For tmax = 100 number of points must be 100 000 to have
    a dt of 0.001
47 x_plotting = np.linspace(0, t_max, 1000)
48 y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
49
50 N = np.linspace(100, 100000, 10, dtype=int)
51 solution_list = []
52 error_list = []
53 #all comparing at t=4
54 compare_time = 4
55 rate_of_convergence = ['N/A']
56 for i in range(len(N)):
57     approx_sol_general = midpoint_rule(third_order_derivative, [0, 4],
        initial_conc, N[i])
58     approx_sol = approx_sol_general[1][-1] # Gets the last y value
        corresponding to t = 4
59     exact_sol = third_order_analytical(4, initial_conc, 0.01)
60     solution_list.append(approx_sol)
61     rel_error = np.abs((approx_sol - exact_sol)/exact_sol)
62     error_list.append(rel_error)
63     if i > 0: # Doesn't make sense to calculate for the first point
64         r = (np.log(error_list[i]/error_list[i-1]))/(np.log(N[i-1] / N
            [i]))
65         rate_of_convergence.append(r)
66
67 rows = []
68 for i in range(len(N)):
69     rows.append([N[i], solution_list[i], error_list[i],
        rate_of_convergence[i]])
70
71 df = pd.DataFrame(rows, columns = ['N_t', 'Calculated solution', '
    epsilon_rel', 'Rate of Convergence'])
72 print(df)
73
74 # Rate approaches 2. This behaviour is expected.

```

codes/Task 9,1.py


```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # This file is similar to 'task5.2'
5 # Here we try to find the minimum number of iterations (max dt) to
  yield a stable result
6
7 def midpoint_rule(fun,tspan, y0, number_of_points=100):
8     '''Applies the midpoint rule on the system of two differential
  equations for task 7. \n
9     fun = function
10    y0 = vector of initial conditions
11    optional:\n
12    number_of_points = how many steps. Default set to 100. Increasing
  this reduces error but increases computation time. '''
13    dt = (tspan[1] - tspan[0])/number_of_points
14    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
15    y = np.zeros(number_of_points+1)
16    y[0] = y0 ## Setting up the initial conditions
17    for i in range(number_of_points):
18        k1 = fun(t[i], y[i])
19        k2 = fun(t[i] + dt*0.5, y[i] + 0.5*dt*k1)
20        y[i+1] = y[i] + dt * k2
21    return t, y
22
23 def third_order_derivative(t, Ca, k=0.01): # I know that the
  assignemnt does not require us to add k as an argument but I wanted
  the function to be as universal as possible
24     '''Returns the concentration over time for a third order chemical
  reaction.
25     t = time in seconds
26     Ca = concentration of component a (a -> b)
27     k = reaction rate constant (default set to 0.01 as this is the
  value for task 1.)'''
28     dca_dt = - k * Ca**3
29     return dca_dt
30
31
32 def third_order_analytical(t, c0, k3):
33     '''Returns the concentration at a given time for a third order
  irreversible chemical reation.
34     t = Time in seconds. If a vector it will retrun a vector of
  solutions
35     c0 = initial concentration.
36     k3 = reaction rate constant for the third order reaction.'''
37     Ca = (1/(c0**(-2) + 2*k3*t))**0.5 ## Analytical solution
38     return Ca
39
40 # Same as task 3
41 initial_conc = 5
42 t_max = 100
43 x_plotting = np.linspace(0, t_max, 1000)
44 y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)

```

```

45 solution_midpoint_13 = midpoint_rule(third_order_derivative, [0, t_max
    ], initial_conc, 13) # dt ~= 7.69
46 solution_midpoint_14 = midpoint_rule(third_order_derivative, [0, t_max
    ], initial_conc, 14) # dt ~= 7.14
47 solution_midpoint_15 = midpoint_rule(third_order_derivative, [0, t_max
    ], initial_conc, 15) # dt ~= 6.67
48 solution_midpoint_20 = midpoint_rule(third_order_derivative, [0, t_max
    ], initial_conc, 20) # dt ~= 5
49 solution_midpoint_50 = midpoint_rule(third_order_derivative, [0, t_max
    ], initial_conc, 50)
50 solution_midpoint_100 = midpoint_rule(third_order_derivative, [0,
    t_max], initial_conc, 100)
51
52
53 fig = plt.figure()
54 plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
55 plt.plot(solution_midpoint_13[0], solution_midpoint_13[1], label='
    Midpoint Rule N=13') # [0] holds the x position and [1] holds the '
    y' position
56 plt.plot(solution_midpoint_14[0], solution_midpoint_14[1], 'b', label=
    'Midpoint Rule N=14') # [0] holds the x position and [1] holds the
    'y' position
57 plt.plot(solution_midpoint_15[0], solution_midpoint_15[1], 'g', label=
    'Midpoint Rule N=15') # Though it isnt perfect, at least now it
    doesnt go down to minus infinity.
58 plt.plot(solution_midpoint_20[0], solution_midpoint_20[1], 'orange',
    label='Midpoint Rule N=20')
59 plt.plot(solution_midpoint_50[0], solution_midpoint_50[1], 'purple',
    label='Midpoint Rule N=50')
60 plt.plot(solution_midpoint_100[0], solution_midpoint_100[1], label='
    Midpoint Rule N=100')
61 # N=13 is a straight line => Not expected and cannot be used.
62 # N=14 is getting there but still not perfect. May be used for large
    values but not for small
63 # N=15 is the first function that starts converging at a reasonable
    time.
64 # N=20 begins to approximate it better and then it can be seen that
    increasing the number of iterations will improve the approximation.
65 # N=100 nearly overlaps the solution.
66 plt.xlabel('Time (s)')
67 plt.ylabel('Concentration of A (mol/L)')
68 plt.title(f'Concentration of A over time with initial concentration of
    {initial_conc} mol/L', weight='bold')
69 plt.legend()
70 plt.ylim(-0.1, 5.5)
71 plt.show()

```

codes/Task 9.2.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate
4
5 # Taking code from task 8
6 l1_value = 1
7 l2_value = 1
8 def der_system_task10(t, x, m1=1, m2=1, l1=l1_value, l2=l2_value):
9     '''Solves the following system of equations: \n
10     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
11     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
12     t = vector for time span \n
13     x = vector (theta1, omega1, theta2, omega2) \n'''
14     dxdt = np.zeros(4) # 4 equations
15     delta = x[2] - x[0]
16     dxdt[0] = x[1]
17     dxdt[1] = (m2 * l1 * (x[1]**2) * np.sin(delta)*np.cos(delta) + m2
18               * 9.81*np.sin(x[2])*np.cos(delta) + m2*l2*(x[3]**2) * np.sin(
19               delta) - (m1+m2)*9.81*np.sin(x[0]))/((m1+m2)*l2 - m2*l2*(np.cos(
20               delta))**2)
21     dxdt[2] = x[3]
22     dxdt[3] = (-m2*l2*(x[3]**2)*np.sin(delta)*np.cos(delta) + (m1+m2)
23               *(9.81*np.sin(x[0])*np.cos(delta) - l1 * (x[1]**2) * np.sin(
24               delta) - 9.81 * np.sin(x[2])))/((m1+m2)*l2 - m2*l2*(np.cos(
25               delta))**2)
26     return dxdt
27
28 def master_function(fun,tspan, y0, method='rk2', number_of_points
29 =1000):
30     '''General function to solve system of differential equations.
31     Does not work on single differential equations. \n
32     fun = function
33     y0 = vector of initial conditions
34     optional:\n
35     method = You can select the method with which your system of
36               differential equations will be evaluated. Default set to second
37               order Runge-Kutta. \n
38     Supported methods : midpoint method ('midpoint'), euler method ('
39     euler'), Classical second order Runge-Kutta ('rk2'), classical
40     fourth order Runge-Kutta ('rk4').
41     number_of_points = how many steps. Default set to 100. Increasing
42     this reduces error but increases computation time. '''
43     dt = (tspan[1] - tspan[0])/number_of_points
44     t = np.linspace(tspan[0], tspan[1], number_of_points+1)
45     y = np.zeros((number_of_points+1, len(y0))) # len(y0) because you
46     would need an initial condition for each derivative.
47     for i in range(len(y0)): #initial conditions as a loop to ensure
48     universability.
49         y[0,i] = y0[i]
50     if method == 'midpoint':
51         for i in range(number_of_points):
52             k1 = fun(t[i], y[i,:])

```

```

39         k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
40         y[i+1,:] = y[i,:] + dt * k2
41     elif method == 'euler':
42         for i in range(number_of_points):
43             y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
44     elif method == 'rk2':
45         for i in range(number_of_points):
46             k1 = fun(t[i], y[i,:])
47             k2 = fun(t[i] + dt, y[i] + dt*k1)
48             y[i+1,:] = y[i] + dt*0.5*(k1+k2)
49     elif method == 'rk4':
50         for i in range(number_of_points):
51             k1 = fun(t[i], y[i,:])
52             k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
53             k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
54             k4 = fun(t[i] + dt, y[i,:] + dt*k3)
55             y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
56     else:
57         return 'Unknown method specified. Check documentation for
58             supported methods' # In case an unknown method is specified
59     return t, y
60
61 # (theta1, omega1, theta2, omega2)
62 ini_cond_vec = [7*np.pi/180, 1, 0.89*np.pi, 1] # Initial conditions a
63 vector
64 tspan = [0,10]
65
66 approx_sol = master_function(der_system_task10, tspan, ini_cond_vec,
67     method='rk4')
68
69 ## For the approx solution don't use euler. It is unreliable and
70 unsteady
71
72 x_1st = np.sin(approx_sol[1][:,0])*l1_value # Like the first node of
73 the double pendulum
74 y_1st = -np.cos(approx_sol[1][:,0])*l1_value
75
76 x_2 = np.sin(approx_sol[1][:,0])*l1_value + np.sin(approx_sol[1][:,2])
77 *l2_value
78 y_2 = -np.cos(approx_sol[1][:,0])*l1_value - np.cos(approx_sol
79 [1][:,2])*l2_value
80
81 x_connector = [0, x_1st[-1], x_2[-1]]
82 y_connector = [0, y_1st[-1], y_2[-1]]
83
84 fig, ax = plt.subplots(figsize=(8,8))
85 plt.plot(x_1st, y_1st, label='Path of second node')
86 plt.plot(x_2, y_2, label='Path of second node')
87 plt.xlabel('Position in x')
88 plt.ylabel('Position in y')
89 plt.title(f'theta1={ini_cond_vec[0]: .4f}, omega1={ini_cond_vec[1]},
90     theta2={ini_cond_vec[2]: .4f}, omega2={ini_cond_vec[3]}', fontsize

```

```

    =10)
85 plt.suptitle(f'Path of double pendulum', weight='bold')
86 plt.xlim(-2,2)
87 plt.ylim(-2,2)
88
89
90 #Animating the path taken
91 line, = ax.plot(x_1st, y_1st, marker='o')
92
93 plt.show(block=False)
94 x_connector = [0, x_1st[-1], x_2[-1]]
95 y_connector = [0, y_1st[-1], y_2[-1]]
96
97 for i in range(len(approx_sol[1][:,0])):
98     line.set_data([0, x_1st[i], x_2[i]], [0, y_1st[i], y_2[i]])
99     fig.canvas.draw()
100     fig.canvas.flush_events()
101 #     plt.pause(0.000000000001)
102 # Uncomment above line to slow down the animation
103
104 plt.show()

```

codes/Task 10.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate
4 import pandas as pd
5 import time
6 # Taking code from task 10
7 # Needed later
8
9 l1_value = 1
10 l2_value = 1
11 m1_value = 1
12 m2_value = 1
13
14 def der_system_task10(t, x, m1=m1_value, m2=m2_value, l1=l1_value, l2=
    l2_value):
15     '''Solves the following system of equations: \n
16     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
17     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
18     t = vector for time span \n
19     x = vector (theta1, omega1, theta2, omega2) \n'''
20     dxdt = np.zeros(4) # two equations
21     delta = x[2] - x[0]
22     dxdt[0] = x[1]
23     dxdt[1] = (m2 * l1 * (x[1]**2) * np.sin(delta)*np.cos(delta) + m2
        * 9.81*np.sin(x[2])*np.cos(delta) + m2*l2*(x[3]**2) * np.sin(
        delta) - (m1+m2)*9.81*np.sin(x[0]))/((m1+m2)*l2 - m2*l2*(np.cos(
        delta))**2)
24     dxdt[2] = x[3]
25     dxdt[3] = (-m2*l2*(x[3]**2)*np.sin(delta)*np.cos(delta) + (m1+m2)
        *(9.81*np.sin(x[0])*np.cos(delta) - l1 * (x[1]**2) * np.sin(
        delta) - 9.81 * np.sin(x[2])))/((m1+m2)*l2 - m2*l2*(np.cos(
        delta))**2)
26     return dxdt
27
28
29 def master_function(fun,tspan, y0, method='rk2', number_of_points=100)
    :
30     '''General function to solve system of differential equations.
31     Does not work on single differential equations. \n
32     fun = function
33     y0 = vector of initial conditions
34     optional:\n
35     method = You can select the method with which your system of
        differential equations will be evaluated. Default set to second
        order Runge-Kutta. \n
36     Supported methods : midpoint method ('midpoint'), euler method ('
        euler'), Classical second order Runge-Kutta ('rk2'), classical
        fourth order Runge-Kutta ('rk4').
37     number_of_points = how many steps. Default set to 100. Increasing
        this reduces error but increases computation time. '''
38     dt = (tspan[1] - tspan[0])/number_of_points
39     t = np.linspace(tspan[0], tspan[1], number_of_points+1)
    y = np.zeros((number_of_points+1, len(y0))) # len(y0) because you

```

```

would need an initial condition for each derivative.
40 for i in range(len(y0)): #initial conditions as a loop to ensure
    universability.
41     y[0,i] = y0[i]
42 if method == 'midpoint':
43     for i in range(number_of_points):
44         k1 = fun(t[i], y[i,:])
45         k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
46         y[i+1,:] = y[i,:] + dt * k2
47 elif method == 'euler':
48     for i in range(number_of_points):
49         y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
50 elif method == 'rk2':
51     for i in range(number_of_points):
52         k1 = fun(t[i], y[i,:])
53         k2 = fun(t[i] + dt, y[i] + dt*k1)
54         y[i+1,:] = y[i] + dt*0.5*(k1+k2)
55 elif method == 'rk4':
56     for i in range(number_of_points):
57         k1 = fun(t[i], y[i,:])
58         k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
59         k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
60         k4 = fun(t[i] +dt, y[i,:] + dt*k3)
61         y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
62 else:
63     return 'Unknown method specified. Check documentation for
        supported methods' # In case an unknown method is specified
64 return t, y
65
66 #(theta1, omega1, theta2, omega2)
67 ini_cond_vec = [7*np.pi/180, 1, 0.9*np.pi, 1] # Initial conditions a
    vector
68 tspan = [0,10]
69
70 approx_sol = master_function(der_system_task10, tspan, ini_cond_vec, '
    midpoint', 2000) #Midpoint used because its better than euler and
    easy to implement :)
71
72
73 # from:
74 # http://www.physics.umd.edu/hep/drew/pendulum2.html
75 # We can get the equations for the kinetic and potential energy
76 # the time derivative in terms of theta1 is simply omega1
77
78 KE = 0.5 * (m1_value+m2_value) * l1_value**2 * approx_sol[1][:,1]**2 +
    0.5 * m2_value* l2_value**2 * approx_sol[1][:,3]**2 + m2_value*
    l1_value*l2_value * approx_sol[1][:,1] * approx_sol[1][:,3] * np.
    cos(approx_sol[1][:,0] - approx_sol[1][:,2])
79
80 PE = -(m1_value+m2_value) * 9.81 *l1_value *np.cos(approx_sol[1][:,0])
    - m2_value*9.81*l2_value*np.cos(approx_sol[1][:,2])
81
82
83 ## Correcting for the zero of energy

```

```

84 PE += (m1_value+m2_value)*9.81*(l1_value+l2_value)
85 ## Without this correction we would see that the potential energy
    would be negative and that the sum of energies would be negative.
86 ## Here we set the zero of energy to the lowest possible point the
    pendulum can reach.
87 ## With l1=1 and l2=1 the lowest point would be y=-2. Thus, we must
    correct for this.
88
89
90
91 fig = plt.figure()
92 plt.plot(approx_sol[0], KE, label='Kinetic Energy')
93 plt.plot(approx_sol[0], PE, label='Potential Energy')
94 plt.plot(approx_sol[0], KE + PE, label='Sum of both')
95 plt.xlabel('Time (s)')
96 plt.ylabel('Energy (J)')
97 plt.title(f'theta1={ini_cond_vec[0]: .4f}, omega1={ini_cond_vec[1]},
    theta2={ini_cond_vec[2]: .4f}, omega2={ini_cond_vec[3]}', fontsize
    =10)
98 plt.suptitle('Potential and Kinetic Energy of the Double Pendulum')
99 plt.legend()
100 plt.show()
101 ## We can see that the graph matches expectations and the total energy
    is conserved.
102 ## Moreover, the shape of the graphs alligns with the expectations
    that as kinetic energy increases potential decreases and vice versa
    .+
103
104
105 methods = ['midpoint', 'euler', 'rk2', 'rk4']
106 rows = []
107
108 for i in range(len(methods)):
109     start_time = time.time()
110     solution_loop = master_function(der_system_task10, tspan,
        ini_cond_vec, 'midpoint', 2000)
111     end_time = time.time() - start_time
112     KE_loop = 0.5 * (m1_value+m2_value) * l1_value**2 * solution_loop
        [1][:,1]**2 + 0.5 * m2_value* l2_value**2 * solution_loop
        [1][:,3]**2 + m2_value*l1_value*l2_value * solution_loop
        [1][:,1] * solution_loop[1][:,3] * np.cos(solution_loop[1][:,0]
        - solution_loop[1][:,2])
113     PE_loop = -(m1_value+m2_value) * 9.81 *l1_value *np.cos(
        solution_loop[1][:,0]) - m2_value*9.81*l2_value*np.cos(
        solution_loop[1][:,2])
114     PE_loop += (m1_value+m2_value)*9.81*(l1_value+l2_value)
115     rows.append([methods[i], np.median(KE_loop+PE_loop), end_time])
116
117 df = pd.DataFrame(rows, columns = ['Method Used', 'Total Energy', '
    Time taken (s)'])
118 print(df)

```

codes/Task 11.py


```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.integrate
4 import pandas as pd
5 import time
6 # Taking code from task 10
7 # Needed later
8
9 l1_value = 1
10 l2_value = 1
11 m1_value = 1
12 m2_value = 1
13
14 def der_system_task10(t, x, m1=m1_value, m2=m2_value, l1=l1_value, l2=
    l2_value):
15     '''Solves the following system of equations: \n
16     dCadt = -A*exp(-Ea/RT) * Ca^3 \n
17     dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
18     t = vector for time span \n
19     x = vector (theta1, omega1, theta2, omega2) \n'''
20     dxdt = np.zeros(4) # two equations
21     delta = x[2] - x[0]
22     dxdt[0] = x[1]
23     dxdt[1] = (m2 * l1 * (x[1]**2) * np.sin(delta)*np.cos(delta) + m2
        * 9.81*np.sin(x[2])*np.cos(delta) + m2*l2*(x[3]**2) * np.sin(
        delta) - (m1+m2)*9.81*np.sin(x[0]))/((m1+m2)*l2 - m2*l2*(np.cos(
        delta))**2)
24     dxdt[2] = x[3]
25     dxdt[3] = (-m2*l2*(x[3]**2)*np.sin(delta)*np.cos(delta) + (m1+m2)
        *(9.81*np.sin(x[0])*np.cos(delta) - l1 * (x[1]**2) * np.sin(
        delta) - 9.81 * np.sin(x[2])))/((m1+m2)*l2 - m2*l2*(np.cos(
        delta))**2)
26     return dxdt
27
28
29 def master_function(fun,tspan, y0, method='rk2', number_of_points=100)
    :
30     '''General function to solve system of differential equations.
31     Does not work on single differential equations. \n
32     fun = function
33     y0 = vector of initial conditions
34     optional:\n
35     method = You can select the method with which your system of
        differential equations will be evaluated. Default set to second
        order Runge-Kutta. \n
36     Supported methods : midpoint method ('midpoint'), euler method ('
        euler'), Classical second order Runge-Kutta ('rk2'), classical
        fourth order Runge-Kutta ('rk4').
37     number_of_points = how many steps. Default set to 100. Increasing
        this reduces error but increases computation time. '''
38     dt = (tspan[1] - tspan[0])/number_of_points
39     t = np.linspace(tspan[0], tspan[1], number_of_points+1)
    y = np.zeros((number_of_points+1, len(y0))) # len(y0) because you

```

```

40     would need an initial condition for each derivative.
41     for i in range(len(y0)): #initial conditions as a loop to ensure
        universability.
42         y[0,i] = y0[i]
43     if method == 'midpoint':
44         for i in range(number_of_points):
45             k1 = fun(t[i], y[i,:])
46             k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
47             y[i+1,:] = y[i,:] + dt * k2
48     elif method == 'euler':
49         for i in range(number_of_points):
50             y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
51     elif method == 'rk2':
52         for i in range(number_of_points):
53             k1 = fun(t[i], y[i,:])
54             k2 = fun(t[i] + dt, y[i] + dt*k1)
55             y[i+1,:] = y[i] + dt*0.5*(k1+k2)
56     elif method == 'rk4':
57         for i in range(number_of_points):
58             k1 = fun(t[i], y[i,:])
59             k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
60             k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
61             k4 = fun(t[i] +dt, y[i,:] + dt*k3)
62             y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
63     else:
64         return 'Unknown method specified. Check documentation for
        supported methods' # In case an unknown method is specified
65     return t, y
66
67 # (theta1, omega1, theta2, omega2)
68
69 for i in range(1,5):
70     ini_cond_vec = [7*np.pi/180, 1, 0.9-0.01*i*np.pi, 1] # Initial
        conditions a vector
71     tspan = [0,5]
72     approx_sol = master_function(der_system_task10, tspan,
        ini_cond_vec, 'midpoint', 2000) #Midpoint used because its
        better than euler and easy to implement :)
73     x_2 = np.sin(approx_sol[1][:,0])*l1_value + np.sin(approx_sol
        [1][:,2])*l2_value
74     y_2 = -np.cos(approx_sol[1][:,0])*l1_value - np.cos(approx_sol
        [1][:,2])*l2_value
75     plt.plot(x_2, y_2, label=f'Theta2 = {0.9-0.01*i*np.pi: .4f}')
76
77 plt.xlabel('Position in x')
78 plt.ylabel('Position in y')
79 plt.suptitle(f'Path of double pendulum at various values of theta2',
        weight='bold')
80 plt.legend(loc='upper right')
81
82 plt.show()

```

codes/Task 11 investigation.py