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:

$$\frac{dC_A}{dt} = -k_3 \cdot C_A^3 \tag{1}$$

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

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

$$\frac{dC_A}{dt} = -k_3 \cdot C_A^3 \tag{2}$$

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

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

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

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

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

$$\tag{8}$$

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

```python
def third_order_analytical(t, c0, k3):
    '''Returns the concentration at a given time for a third order
       irreversible chemical reation.
    t = Time in seconds. If a vector it will retrun a vector of
       solutions
    c0 = initial concentration.
    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.

3. For this task we chose to set $C_{A,0} = 5\ \mathrm{mol\,L^{-1}}$. 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.

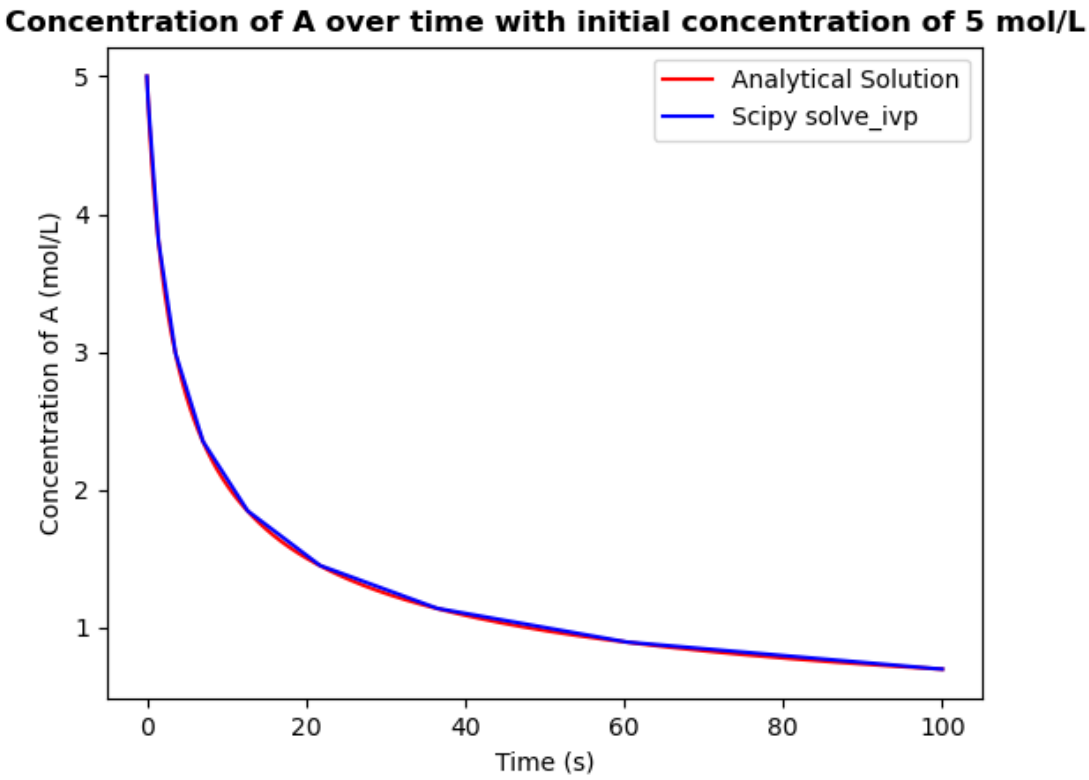

Figure 1: Analytical Solution vs Scipy "solve_ivp"

As can 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.

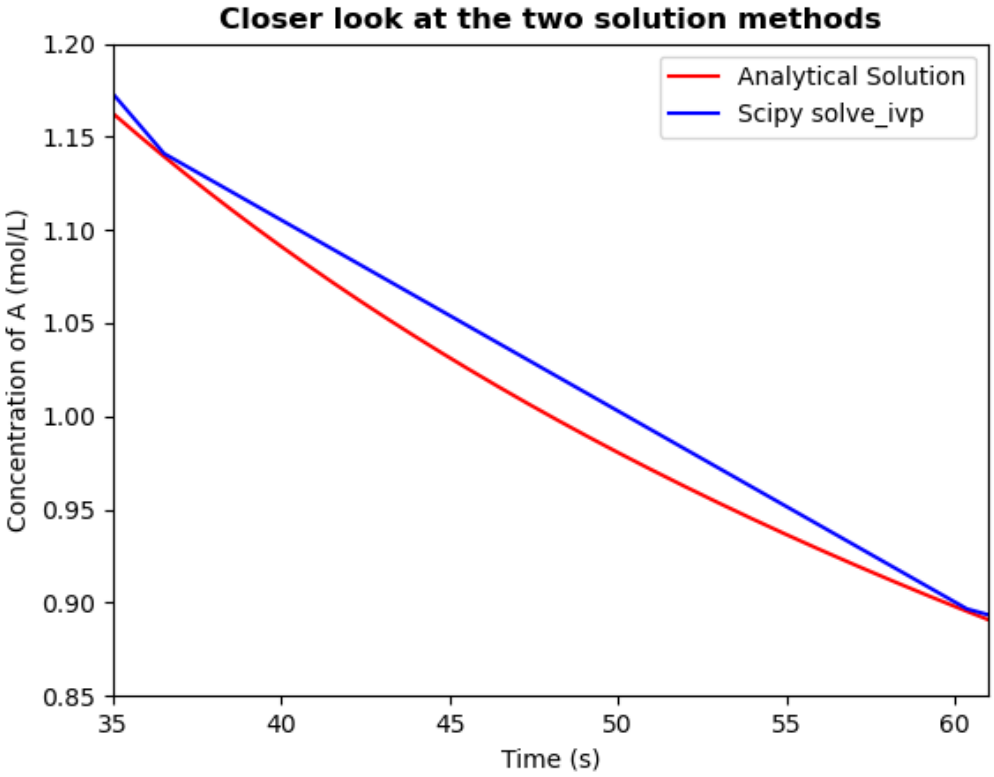

Figure 2: Closer look at the two curves.

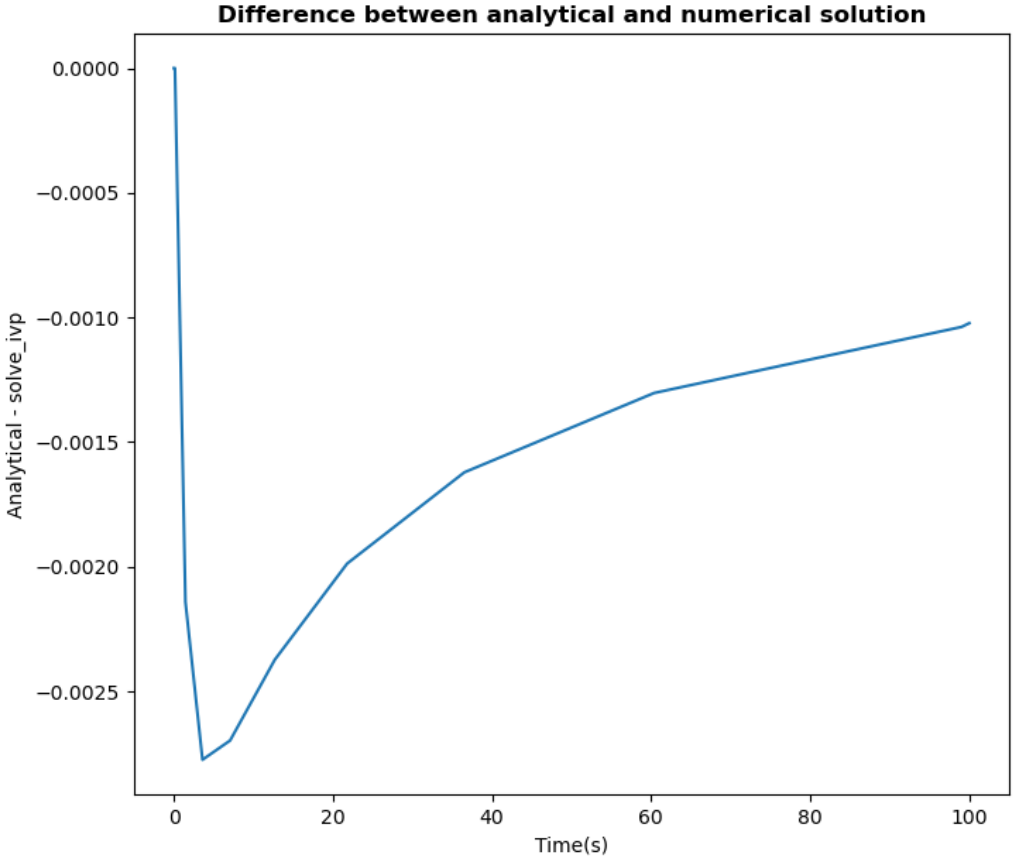

Figure 3: Difference between the analytical solution and "Scipy" solution

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

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.

```python
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
    '''Explicit first order euler solver for ODEs.\n
    fun = derivative function \n
    tspan = supply the time span as a vector ie [0,10] \n
    y0 = initial condition ie at t=0 y0=100 \n
    number_of_points = Number of points. Default set to 100000 in order to have dt = 0.001. Higher number means longer computation time'''
    dt = (tspan[1] - tspan[0])/number_of_points
    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
    y = np.zeros(number_of_points+1)
    y[0] = y0 ## Setting up the initial conditions
    for i in range(number_of_points):
        y[i+1] = y[i] + dt * fun(t[i], y[i])
    return t, y
```

Listing 3: Implementation of the first order explicit euler method

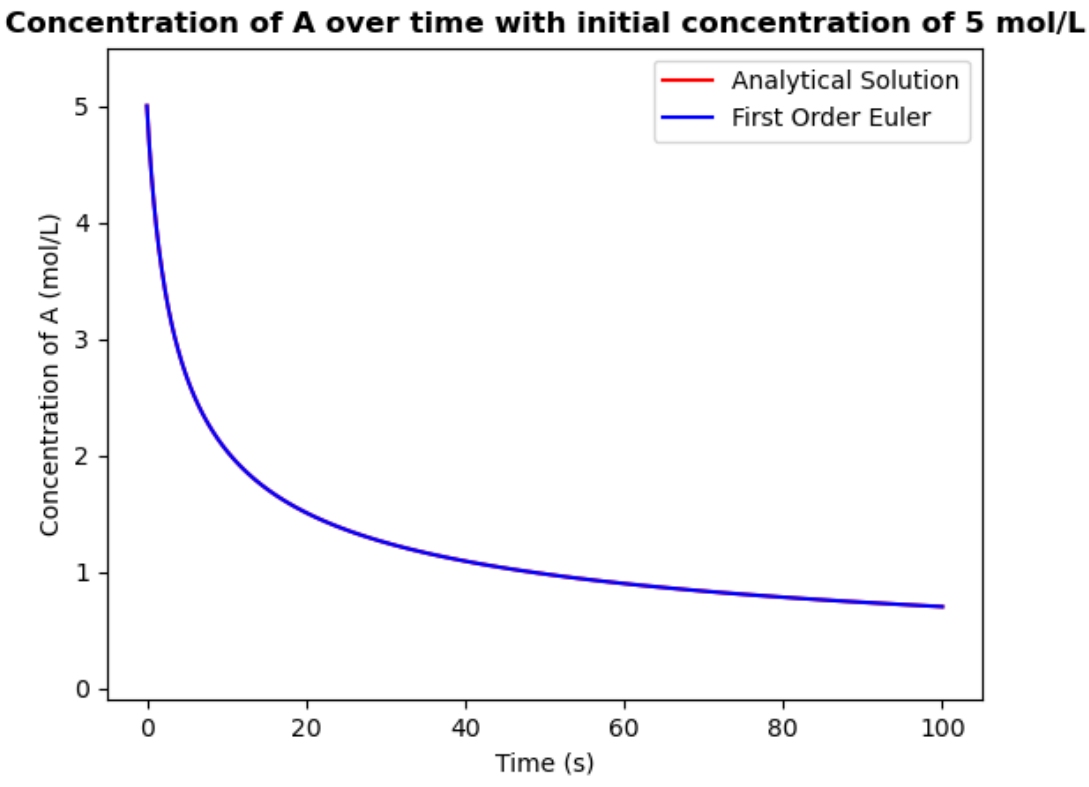

Figure 4: Analytical solution and first order Euler approximation visualization.

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.

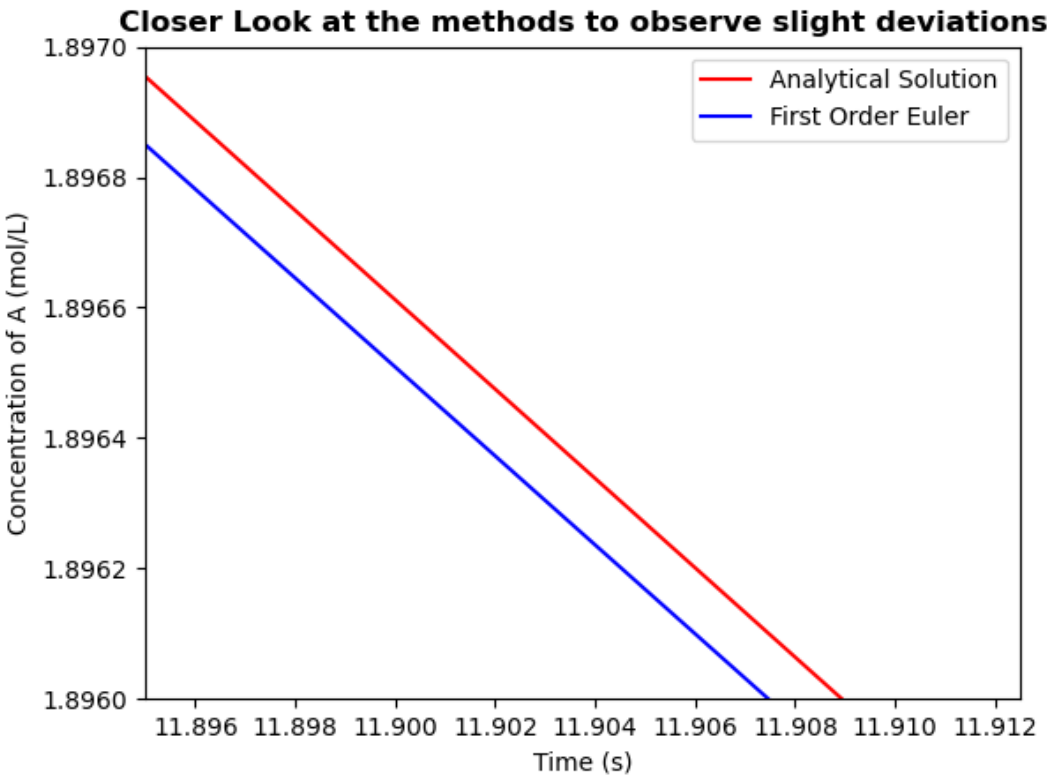

Figure 5: Closer look at the two curves.

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:

$$\epsilon_{rel} = \left| \frac{y_{\text{numerical}} - y_{\text{analytical}}}{y_{\text{analytical}}} \right| \tag{9}$$

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

$$r = \frac{\log \frac{\epsilon_2}{\epsilon_1}}{\log \frac{N_1}{N_2}} \tag{10}$$

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

```
N = np.linspace(100, 100000, 10, dtype=int) # We need dtype=int
    because we need an integer number of points for the function
solution_list = []
error_list = []
#all comparing at t=4
compare_time = 4
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
for i in range(len(N)):
    approx_sol_general = first_order_euler(third_order_derivative, [0,
         4], initial_conc, N[i])
    approx_sol = approx_sol_general[1][-1]  # Gets the last y value
       corresponding to t = 4
    exact_sol = third_order_analytical(4, initial_conc, 0.01)
    solution_list.append(approx_sol)
    rel_error = np.abs((approx_sol - exact_sol)/exact_sol) # Relative
       error calculation. No specific type of error was asked to
       calculate.
    error_list.append(rel_error)
    if i > 0:  # Doesn't make sense to calculate for the first point
        r = (np.log(error_list[i]/error_list[i-1]))/(np.log(N[i-1] / N
           [i]))
        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

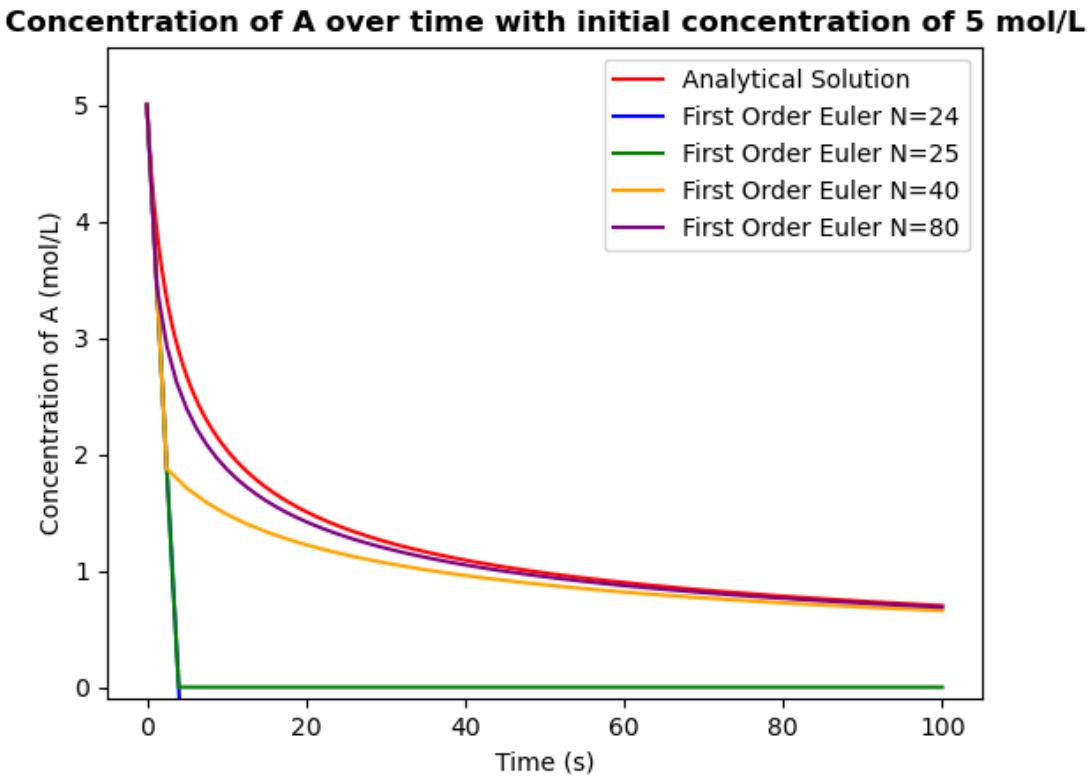

Figure 6: Closer look at the two curves.

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 :

$$\frac{dC_A}{dt} = -A \cdot \exp -\frac{E_A}{RT} \cdot C_A^3 \tag{11}$$

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

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

```
#Initialize the variables
A = 3.1
rho = 1.25 # kg * m^-3
Cp = 1200 # J * kg^-1 * K^-1
deltaH = -18400 # we need to keep units constant thus J * mol^-1
Ea = 25800 # Same logic as above because 8.3145 will be used for R
    thus J * mol^-1
c0 = 10
T0 = 313
R = 8.3145


def der_system_task6(t, x):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector where first position is the concentration and second is
        the temperature \n'''
    dxdt = np.zeros(2) # two equations
    dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents
        dcadt
    dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*
        Cp) #Represents dTdt
    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.

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

Listing 6: Solving the system

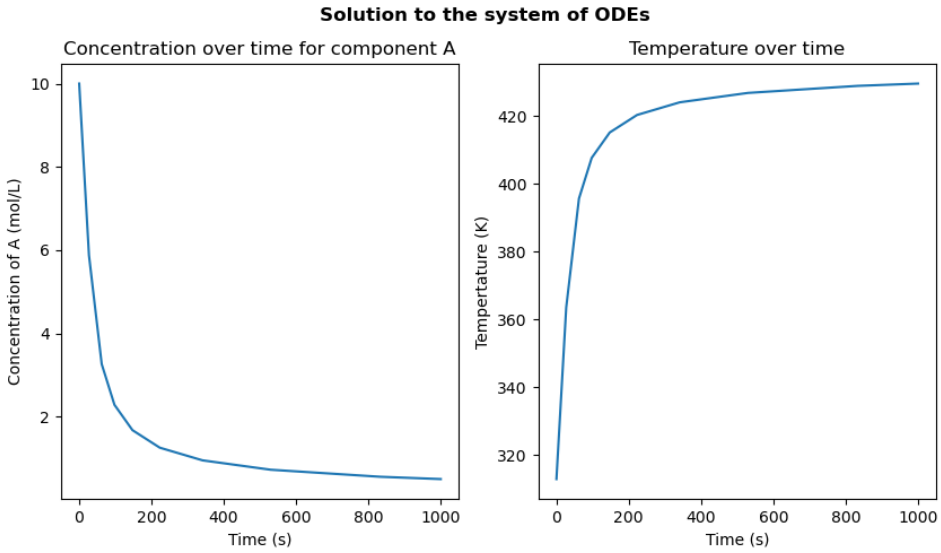

Figure 7: Visualization of the solution to the system of differential equations

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 an 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.

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

Listing 7: Updated Euler Function

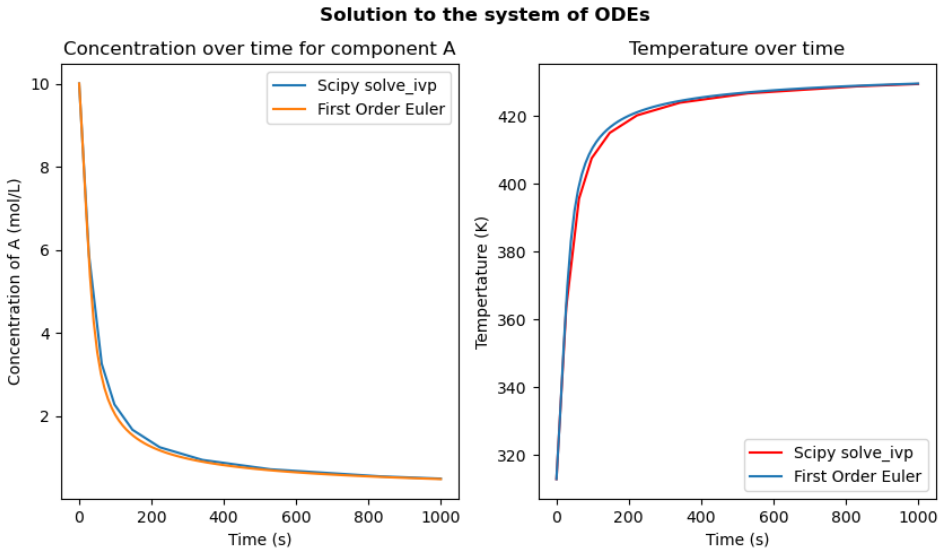

Figure 8: Two methods to solve the system of ODEs

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.

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

```
def master_function(fun,tspan, y0, method='rk2', number_of_points=100):
    '''General function to solve system of differential equations. Does not work on single differential equations. \n
    fun = function
    y0 = vector of initial conditions
    optional:\n
    method = You can select the method with which your system of differential equations will be evaluated. Default set to second order Runge-Kutta. \n
    Supported methods : midpoint method ('midpoint'), euler method ('euler'), Classical second order Runge-Kutta ('rk2'), classical fourth order Runge-Kutta ('rk4'). \n
    number_of_points = how many steps. Default set to 100. Increasing this reduces error but increases computation time. '''
    dt = (tspan[1] - tspan[0])/number_of_points
    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.
    for i in range(len(y0)): #initial conditions as a loop to ensure universability.
        y[0,i] = y0[i]
    if method == 'midpoint':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            y[i+1,:] = y[i,:] + dt * k2
    elif method == 'euler':
        for i in range(number_of_points):
            y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
    elif method == 'rk2':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt, y[i] + dt*k1)
            y[i+1,:] = y[i] + dt*0.5*(k1+k2)
    elif method == 'rk4':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
            k4 = fun(t[i] +dt, y[i,:] + dt*k3)
            y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
    else:
        return 'Unknown method specified. Check documentation for supported methods' # In case an unknown method is specified
    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.

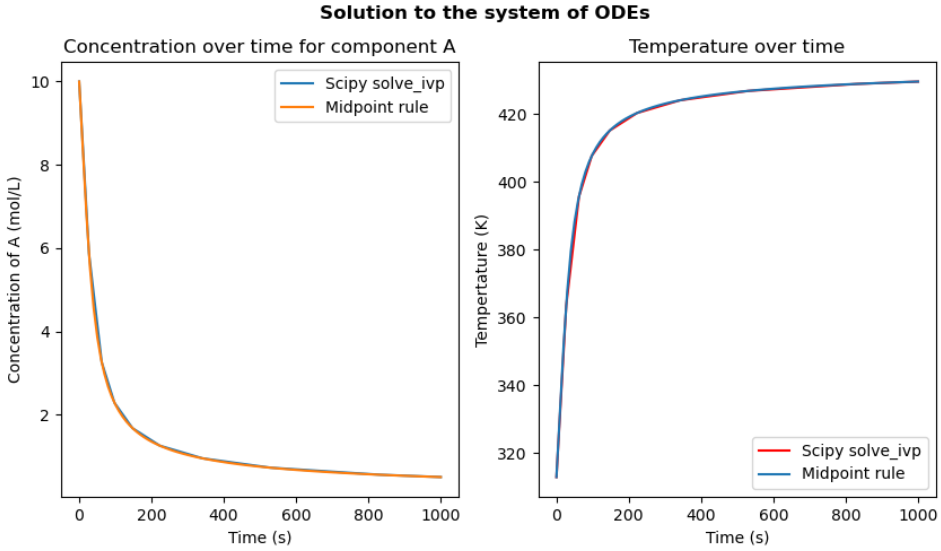

Figure 9: Solution to the system of ODEs with midpoint method

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.

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

Listing 10: Simplified midpoint method for characterization

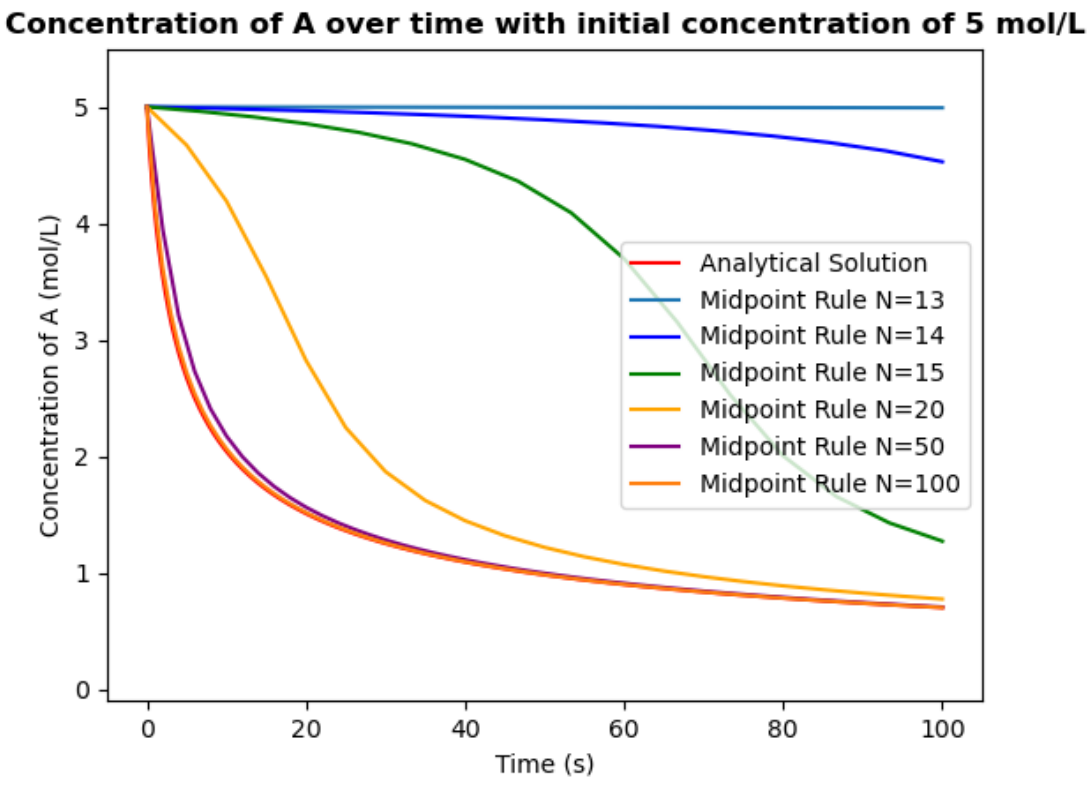

Figure 10: Minimum number of points required for the midpoint rule

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.

```
l1_value = 1
l2_value = 1
def der_system_task10(t, x, m1=1, m2=1, l1=l1_value, l2=l2_value):
    '''Solves the system of differential equations for a double
       pendulum \n
    t =  vector for time span \n
    x = vector (theta1, omega1, theta2, omega2) \n'''
    dxdt = np.zeros(4) # 4 equations
    delta = x[2] - x[0]
    dxdt[0] = x[1]
    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)
    dxdt[2] = x[3]
    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)
    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.

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

Listing 12: Solving the system

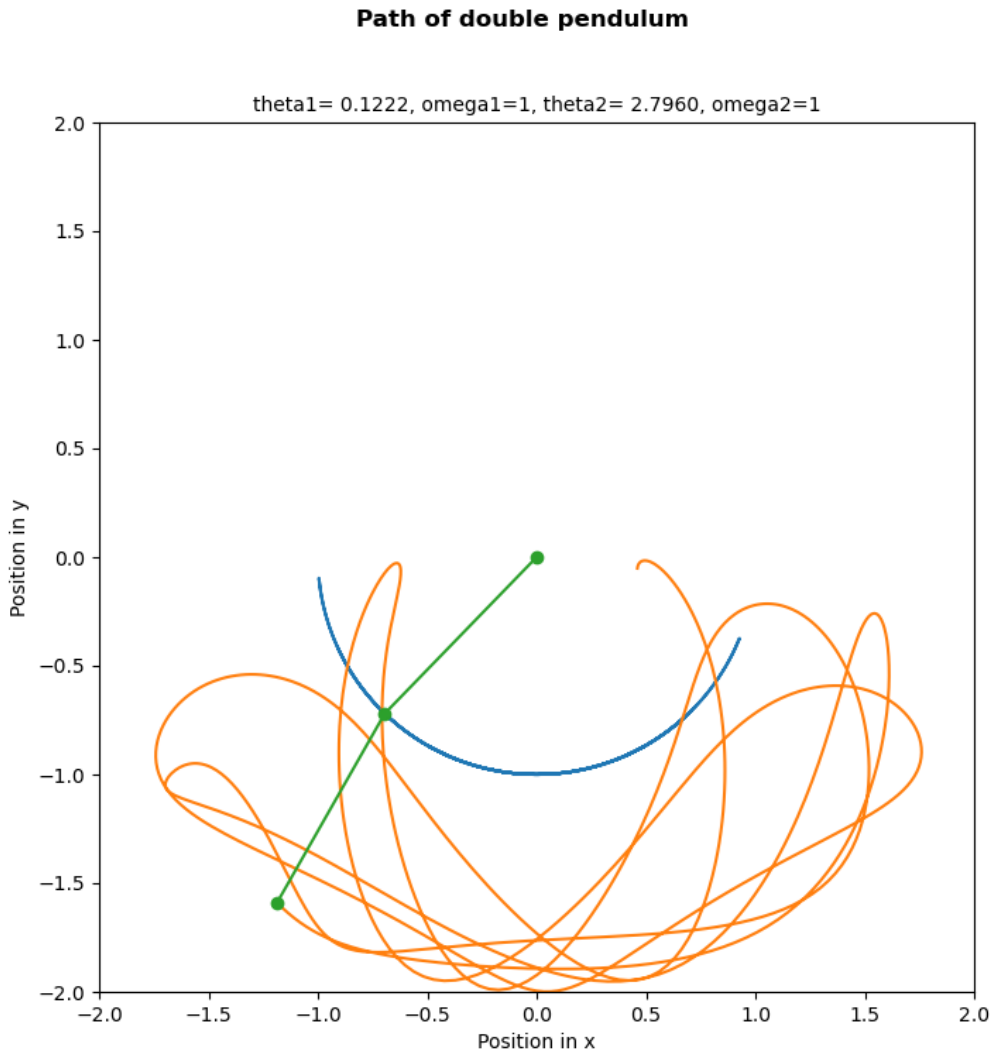

Figure 11: Final position of the double pendulum

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.

```
x_1st = np.sin(approx_sol[1][:,0])*l1_value #Like the first node of
    the double pendulum
y_1st = -np.cos(approx_sol[1][:,0])*l1_value

x_2 = np.sin(approx_sol[1][:,0])*l1_value + np.sin(approx_sol[1][:,2])
    *l2_value
y_2 = -np.cos(approx_sol[1][:,0])*l1_value - np.cos(approx_sol
    [1][:,2])*l2_value

x_connector = [0, x_1st[-1], x_2[-1]]
y_connector = [0, y_1st[-1], y_2[-1]]


fig, ax = plt.subplots(figsize=(8,8))
plt.plot(x_1st, y_1st, label='Path of second node')
plt.plot(x_2, y_2, label='Path of second node')
plt.xlabel('Position in x')
plt.ylabel('Position in y')
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)
plt.suptitle(f'Path of double pendulum', weight='bold')
plt.xlim(-2,2)
plt.ylim(-2,2)


#Animating the path taken
line, = ax.plot(x_1st, y_1st, marker='o')

plt.show(block=False)
x_connector = [0, x_1st[-1], x_2[-1]]
y_connector = [0, y_1st[-1], y_2[-1]]

for i in range(len(approx_sol[1][:,0])):
    line.set_data([0, x_1st[i], x_2[i]], [0, y_1st[i], y_2[i]])
    fig.canvas.draw()
    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]:

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

$$PE = -\left(m_1 + m_2\right) g L_1 \cos(\theta_1) - m_2 g L_2 \cos(\theta_2) \tag{14}$$

```
# from:
# http://www.physics.umd.edu/hep/drew/pendulum2.html
# We can get the equations for the kinetic and potential energy
# the time derivative in terms of theta1 is simply omega1

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])

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])


## Correcting for the zero of energy
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.

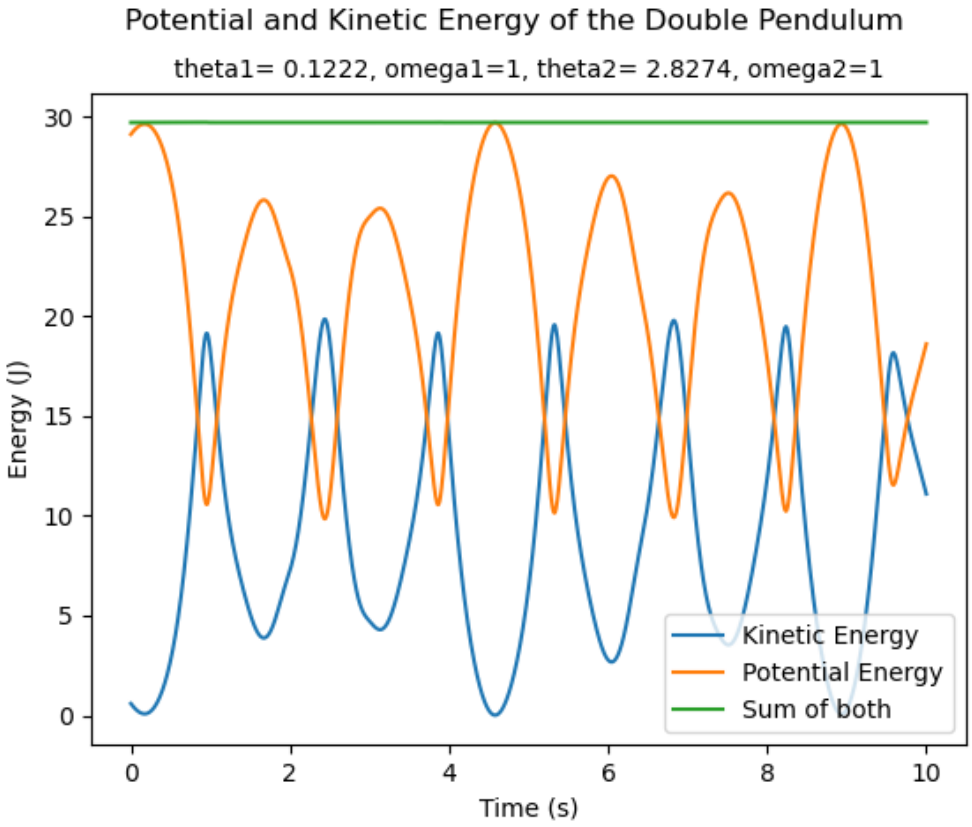

Figure 12: Energy of the double pendulum system

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.

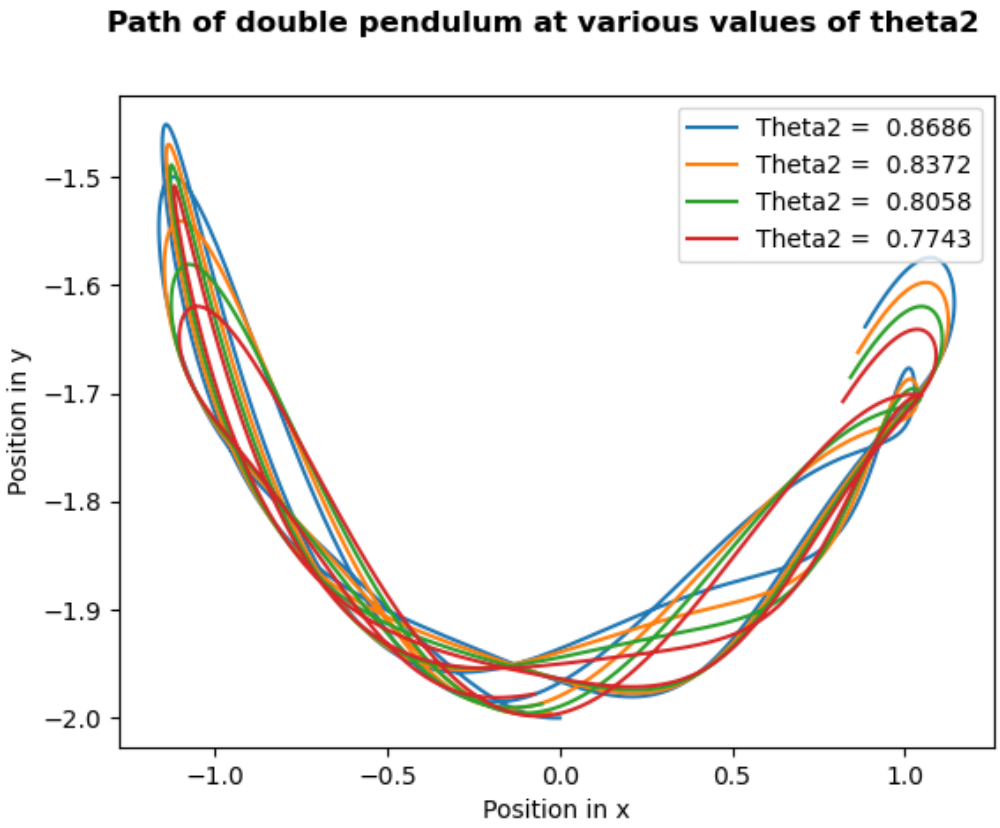

Figure 13: Path taken by second node at various θ_2 values

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

```python
# Third order so:
# rate = - k * (C_A)^3
# Negative sign because its being consumed
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
    '''Returns the concentration over time for a third order chemical reaction.
    t = time in seconds
    Ca = concentration of component a (a -> b)
    k = reaction rate constant (default set to 0.01 as this is the value for task 1.)'''
    dca_dt = - k * Ca**3
    return dca_dt
```

codes/Task 1.py

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

## You can feed the function t as a vector (like an np.array) and it
   will return a vector with the Ca at those times.
```

codes/Task 2.py

```python
import scipy.integrate
import matplotlib.pyplot as plt ## For plotting
import numpy as np

# Functions from previous sections
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
    '''Returns the concentration over time for a third order chemical reaction.
    t = time in secons
    Ca = concentration of component a (a -> b)
    k = reaction rate constant (default set to 0.01 as this is the value for task 1.)'''
    dca_dt = - k3 * Ca**3
    return dca_dt

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

# Initializing variables
initial_conc = 5
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
x_plotting = np.linspace(0, t_max, 1000) # For generating the output figure for the analytical solution
y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
y_scipy = scipy.integrate.solve_ivp(lambda t, Ca: third_order_derivative(t, Ca), [0, t_max], [initial_conc])
y_scipy_plotting = y_scipy.y[0]
x_scipy_plotting = y_scipy.t # the 't' variable corresponds to the x axis

#Figure output code
fig = plt.figure()
plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
plt.plot(x_scipy_plotting, y_scipy_plotting, 'b', label='Scipy solve_ivp')
plt.xlabel('Time (s)')
plt.ylabel('Concentration of A (mol/L)')
plt.title(f'Concentration of A over time with initial concentration of {initial_conc} mol/L', weight='bold')
plt.legend()
plt.show()
## The figure above shows that they are almost identical
```

```

## We can restrict the domain to see the deviations
fig2 = plt.figure()
plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
plt.plot(x_scipy_plotting, y_scipy_plotting, 'b', label='Scipy
    solve_ivp')
plt.xlabel('Time (s)')
plt.ylabel('Concentration of A (mol/L)')
plt.title(f'Closer look at the two solution methods', weight='bold')
plt.xlim(35, 61)
plt.ylim(0.85, 1.2)
plt.legend()
plt.show()
#Slight deviations but we need to zoom in quite close to see

# For difference over time
y_analytical_scipyxvalues = third_order_analytical(x_scipy_plotting,
    initial_conc, 0.01)
# New calculation because scipy does not use a linspace


diff = []

for i in range(len(x_scipy_plotting)):
    difference = y_analytical_scipyxvalues[i] -  y_scipy_plotting[i]
    diff.append(difference)

fig3 = plt.figure(figsize=(8,7))
plt.plot(x_scipy_plotting, diff)
plt.xlabel('Time(s)')
plt.ylabel('Analytical - solve_ivp')
plt.title('Difference between analytical and numerical solution',
    weight='bold')
plt.show()

print(f'Maximum deviation between analytical and Scipy solution: {np.
    abs(min(diff))}')
## Only prints the absolute value.
```

codes/Task 3.py

```python
import numpy as np
import matplotlib.pyplot as plt
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
    '''Explicit first order euler solver for ODEs.\n
    fun = derivative function \n
    tspan = supply the time span as a vector ie [0,10] \n
    y0 = initial condition ie at t=0 y0=100 \n
    number_of_points = Number of points. Default set to 100000 in order to have dt = 0.001. Higher number means longer computation time'''
    dt = (tspan[1] - tspan[0])/number_of_points
    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
    y = np.zeros(number_of_points+1)
    y[0] = y0 ## Setting up the initial conditions
    for i in range(number_of_points):
        y[i+1] = y[i] + dt * fun(t[i], y[i])
    return t, y

#From task 1
# 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.
# For some people importing from other files doesn't work.

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


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

# Same as task 3
initial_conc = 5
t_max = 100 # For tmax = 100, the number of points must be 100 000 to have a dt of 0.001
x_plotting = np.linspace(0, t_max, 1000)
y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
solution_euler = first_order_euler(third_order_derivative, [0, t_max],
```

```
    initial_conc)


fig = plt.figure()
plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
plt.plot(solution_euler[0], solution_euler[1], 'b', label='First Order
    Euler') # [0] holds the x position and [1] holds the 'y' position
plt.xlabel('Time (s)')
plt.ylabel('Concentration of A (mol/L)')
plt.title(f'Concentration of A over time with initial concentration of
    {initial_conc} mol/L', weight='bold')
plt.legend()
plt.ylim(-0.1, 5.5)
plt.show()


plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
plt.plot(solution_euler[0], solution_euler[1], 'b', label='First Order
    Euler') # [0] holds the x position and [1] holds the 'y' position
plt.xlabel('Time (s)')
plt.ylabel('Concentration of A (mol/L)')
plt.title(f'Closer Look at the methods to observe slight deviations',
   weight='bold')
plt.legend()
plt.ylim(1.896, 1.897)
plt.xlim(11.8950, 11.9125)
plt.show()
```

codes/Task 4.py

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

#This file computes the error and rate of convergence. The file 'Task 5,2' shows the maximum time step for a stable solution

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

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


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

# Based on the lecture slides we expect a first order rate of convergence

initial_conc = 5 # Same initial conc as before

N = np.linspace(100, 100000, 10, dtype=int) # We need dtype=int because we need an integer number of points for the function
solution_list = []
error_list = []
```

```python
#all comparing at t=4
compare_time = 4
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
for i in range(len(N)):
    approx_sol_general = first_order_euler(third_order_derivative, [0, 4], initial_conc, N[i])
    approx_sol = approx_sol_general[1][-1]  # Gets the last y value corresponding to t = 4
    exact_sol = third_order_analytical(4, initial_conc, 0.01)
    solution_list.append(approx_sol)
    rel_error = np.abs((approx_sol - exact_sol)/exact_sol) # Relative error calculation. No specific type of error was asked to calculate.
    error_list.append(rel_error)
    if i > 0:  # Doesn't make sense to calculate for the first point
        r = (np.log(error_list[i]/error_list[i-1]))/(np.log(N[i-1] / N[i]))
        rate_of_convergence.append(r)

#Generating output table
rows = []
for i in range(len(N)):
    rows.append([N[i], solution_list[i], error_list[i], rate_of_convergence[i]])

df = pd.DataFrame(rows, columns = ['N_t', 'Calculated solution', 'epsilon_rel', 'Rate of Convergence'])
print(df)

# We can se that as Nt increases, the rate of convergence approaches 1. This alligns with the expected results
```

codes/Task 5,1.py

```python
import numpy as np
import matplotlib.pyplot as plt

#Functions from previous tasks
def first_order_euler(fun, tspan, y0, number_of_points=100000):
    '''Explicit first order euler solver for ODEs.\n
    fun = derivative function \n
    tspan = supply the time span as a vector ie [0,10] \n
    y0 = initial condition ie at t=0 y0=100 \n
    number_of_points = Number of points. Default set to 100000 in order to have dt = 0.001. Higher number means longer computation time'''
    dt = (tspan[1] - tspan[0])/number_of_points
    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
    y = np.zeros(number_of_points+1)
    y[0] = y0 ## Setting up the initial conditions
    for i in range(number_of_points):
        y[i+1] = y[i] + dt * fun(t[i], y[i])
    return t, y

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
    '''Returns the concentration over time for a third order chemical reaction.
    t = time in secons
    Ca = concentration of component a (a -> b)
    k = reaction rate constant (default set to 0.01 as this is the value for task 1.)'''
    dca_dt = - k * Ca**3
    return dca_dt


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

# Same as task 3
initial_conc = 5
t_max = 100
x_plotting = np.linspace(0, t_max, 1000)
y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
solution_euler_24 = first_order_euler(third_order_derivative, [0, t_max], initial_conc, 24) # dt ~= 4.1667
solution_euler_25 = first_order_euler(third_order_derivative, [0, t_max], initial_conc, 25) # dt = 4
solution_euler_40 = first_order_euler(third_order_derivative, [0,
```

```
    t_max], initial_conc, 40)
solution_euler_80 = first_order_euler(third_order_derivative, [0,
    t_max], initial_conc, 80)


fig = plt.figure()
plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
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
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.
plt.plot(solution_euler_40[0], solution_euler_40[1], 'orange', label='
    First Order Euler N=40')
plt.plot(solution_euler_80[0], solution_euler_80[1], 'purple', label='
    First Order Euler N=80')

plt.xlabel('Time (s)')
plt.ylabel('Concentration of A (mol/L)')
plt.title(f'Concentration of A over time with initial concentration of
     {initial_conc} mol/L', weight='bold')
plt.legend()
plt.ylim(-0.1, 5.5)
plt.show()
```

codes/Task 5,2.py

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

#Initialize the variables
A = 3.1
rho = 1.25 # kg * m^-3
Cp = 1200 # J * kg^-1 * K^-1
deltaH = -18400 # we need to keep units constant thus J * mol^-1
Ea = 25800 # Same logic as above because 8.3145 will be used for R thus J * mol^-1
c0 = 10
T0 = 313
R = 8.3145


def der_system_task6(t, x):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector where first position is the concentration and second is the temperature \n'''
    dxdt = np.zeros(2) # two equations
    dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents dcadt
    dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*Cp) #Represents dTdt
    return dxdt

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

fig = plt.figure(figsize=(10,5))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)

ax1.plot(solution.t, solution.y[0])
ax2.plot(solution.t, solution.y[1])
ax1.set_ylabel('Concentration of A (mol/L)')
ax1.set_xlabel('Time (s)')
ax1.set_title('Concentration over time for component A')
ax2.set_title('Temperature over time')
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Tempertature (K)')
fig.suptitle('Solution to the system of ODEs', weight='bold')

plt.show()
```

codes/Task 6.py

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

# Taking the code from task 6
#Initialize the variables

A = 3.1
rho = 1.25 # kg * m^-3
Cp = 1200 # J * kg^-1 * K^-1
deltaH = -18400 # we need to keep units constant thus J * mol^-1
Ea = 25800 # Same logic as above because 8.3145 will be used for R thus J * mol^-1
c0 = 10
T0 = 313
R = 8.3145


def der_system_task6(t, x):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector where first position is the concentration and second is the temperature \n'''
    dxdt = np.zeros(2) # two equations
    dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents dcadt
    dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*Cp)
    return dxdt


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

ini_cond_vec = [c0, T0] # Initial conditions a vector
tspan = [0,1000]

```

```
approx_sol = first_order_euler_system(der_system_task6, tspan,
    ini_cond_vec)

solution = scipy.integrate.solve_ivp(der_system_task6, tspan,
    ini_cond_vec)

fig = plt.figure(figsize=(10,5))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)

ax1.plot(solution.t, solution.y[0], label='Scipy solve_ivp')
ax1.plot(approx_sol[0], approx_sol[1][:,0], label='First Order Euler')
ax1.set_ylabel('Concentration of A (mol/L)')
ax1.set_xlabel('Time (s)')
ax1.set_title('Concentration over time for component A')
ax1.legend()
ax2.plot(solution.t, solution.y[1],'r',  label='Scipy solve_ivp')
ax2.plot(approx_sol[0], approx_sol[1][:,1], label='First Order Euler')
ax2.set_title('Temperature over time')
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Tempertature (K)')
fig.suptitle('Solution to the system of ODEs', weight='bold')
ax2.legend()
plt.show()
```

codes/Task 7.py

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

# Taking the code from task 6
#Initialize the variables

A = 3.1
rho = 1.25 # kg * m^-3
Cp = 1200 # J * kg^-1 * K^-1
deltaH = -18400 # we need to keep units constant thus J * mol^-1
Ea = 25800 # Same logic as above because 8.3145 will be used for R thus J * mol^-1
c0 = 10
T0 = 313
R = 8.3145



def der_system_task6(t, x):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector where first position is the concentration and second is the temperature \n'''
    dxdt = np.zeros(2) # two equations
    dxdt[0] = -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3 ## Represents dcadt
    dxdt[1] = (deltaH * -1*A * np.exp((-Ea)/(R*x[1])) * x[0]**3)/(rho*Cp)
    return dxdt

# For the challenge the master function is provided below. The default method is set to the rk2 method.
# Check the documentation for the supported methods
def master_function(fun,tspan, y0, method='rk2', number_of_points=100):
    '''General function to solve system of differential equations. Does not work on single differential equations. \n
    fun = function
    y0 = vector of initial conditions
    optional:\n
    method = You can select the method with which your system of differential equations will be evaluated. Default set to second order Runge-Kutta. \n
    Supported methods : midpoint method ('midpoint'), euler method ('euler'), Classical second order Runge-Kutta ('rk2'), classical fourth order Runge-Kutta ('rk4'). \n
    number_of_points = how many steps. Default set to 100. Increasing this reduces error but increases computation time. '''
    dt = (tspan[1] - tspan[0])/number_of_points
    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.
    for i in range(len(y0)): #initial conditions as a loop to ensure universability.
        y[0,i] = y0[i]
    if method == 'midpoint':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            y[i+1,:] = y[i,:] + dt * k2
    elif method == 'euler':
        for i in range(number_of_points):
            y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
    elif method == 'rk2':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt, y[i] + dt*k1)
            y[i+1,:] = y[i] + dt*0.5*(k1+k2)
    elif method == 'rk4':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
            k4 = fun(t[i] +dt, y[i,:] + dt*k3)
            y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
    else:
        return 'Unknown method specified. Check documentation for supported methods' # In case an unknown method is specified
    return t, y

# Same plotting but now with the midpoint rule.
# Code copied from task 7



ini_cond_vec = [c0, T0] # Initial conditions a vector
tspan = [0,1000]


approx_sol = master_function(der_system_task6, tspan, ini_cond_vec, 'midpoint')

solution = scipy.integrate.solve_ivp(der_system_task6, tspan, ini_cond_vec)

fig = plt.figure(figsize=(10,5))
ax1 = plt.subplot(1,2,1)
ax2 = plt.subplot(1,2,2)

ax1.plot(solution.t, solution.y[0], label='Scipy solve_ivp')
ax1.plot(approx_sol[0], approx_sol[1][:,0], label='Midpoint rule')
ax1.set_ylabel('Concentration of A (mol/L)')
ax1.set_xlabel('Time (s)')
ax1.set_title('Concentration over time for component A')
```

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

codes/Task 8.py

```python
#Taking the more or less the same code from task 1,2,3,5
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import scipy.integrate

#Table print out (same as task 5.1)

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
    '''Returns the concentration over time for a third order chemical reaction.
    t = time in secons
    Ca = concentration of component a (a -> b)
    k = reaction rate constant (default set to 0.01 as this is the value for task 1.)'''
    dca_dt = - k * Ca**3
    return dca_dt

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

## We need to slightly modify it so that it only solves for one method

## We see a simplified version of the master function because we only solve one ODE at a time not a system
def midpoint_rule(fun,tspan, y0, number_of_points=100):
    '''Applies the midpoint rule on the system of two differential equations for task 7. \n
    fun = function
    y0 = vector of initial conditions
    optional:\n
    number_of_points = how many steps. Default set to 100. Increasing this reduces error but increases computation time. '''
    dt = (tspan[1] - tspan[0])/number_of_points
    t = np.linspace(tspan[0], tspan[1], number_of_points+1)
    y = np.zeros(number_of_points+1)
    y[0] = y0 ## Setting up the initial conditions
    for i in range(number_of_points):
        k1 = fun(t[i], y[i])
        k2 = fun(t[i] + dt*0.5, y[i] + 0.5*dt*k1)
        y[i+1] = y[i] + dt * k2
    return t, y


```

```
initial_conc = 5
t_max = 100 # For tmax = 100 number of points must be 100 000 to have a dt of 0.001
x_plotting = np.linspace(0, t_max, 1000)
y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)

N = np.linspace(100, 100000, 10, dtype=int)
solution_list = []
error_list = []
#all comparing at t=4
compare_time = 4
rate_of_convergence = ['N/A']
for i in range(len(N)):
    approx_sol_general = midpoint_rule(third_order_derivative, [0, 4], initial_conc, N[i])
    approx_sol = approx_sol_general[1][-1]  # Gets the last y value corresponding to t = 4
    exact_sol = third_order_analytical(4, initial_conc, 0.01)
    solution_list.append(approx_sol)
    rel_error = np.abs((approx_sol - exact_sol)/exact_sol)
    error_list.append(rel_error)
    if i > 0:  # Doesn't make sense to calculate for the first point
        r = (np.log(error_list[i]/error_list[i-1]))/(np.log(N[i-1] / N[i]))
        rate_of_convergence.append(r)

rows = []
for i in range(len(N)):
    rows.append([N[i], solution_list[i], error_list[i], rate_of_convergence[i]])

df = pd.DataFrame(rows, columns = ['N_t', 'Calculated solution', 'epsilon_rel', 'Rate of Convergence'])
print(df)

# Rate approaches 2. This behaviour is expected.
```

codes/Task 9,1.py

```python
import numpy as np
import matplotlib.pyplot as plt

# This file is similar to 'task5.2'
# Here we try to find the minimum number of iterations (max dt) to yield a stable result

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

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
    '''Returns the concentration over time for a third order chemical reaction.
    t = time in secons
    Ca = concentration of component a (a -> b)
    k = reaction rate constant (default set to 0.01 as this is the value for task 1.)'''
    dca_dt = - k * Ca**3
    return dca_dt


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

# Same as task 3
initial_conc = 5
t_max = 100
x_plotting = np.linspace(0, t_max, 1000)
y_analytical = third_order_analytical(x_plotting, initial_conc, 0.01)
```

```
solution_midpoint_13 = midpoint_rule(third_order_derivative, [0, t_max], initial_conc, 13) # dt ~= 7.69
solution_midpoint_14 = midpoint_rule(third_order_derivative, [0, t_max], initial_conc, 14) # dt ~= 7.14
solution_midpoint_15 = midpoint_rule(third_order_derivative, [0, t_max], initial_conc, 15) # dt ~= 6.67
solution_midpoint_20 = midpoint_rule(third_order_derivative, [0, t_max], initial_conc, 20) # dt ~= 5
solution_midpoint_50 = midpoint_rule(third_order_derivative, [0, t_max], initial_conc, 50)
solution_midpoint_100 = midpoint_rule(third_order_derivative, [0, t_max], initial_conc, 100)


fig = plt.figure()
plt.plot(x_plotting, y_analytical, 'r', label="Analytical Solution")
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
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
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.
plt.plot(solution_midpoint_20[0], solution_midpoint_20[1], 'orange', label='Midpoint Rule N=20')
plt.plot(solution_midpoint_50[0], solution_midpoint_50[1], 'purple', label='Midpoint Rule N=50')
plt.plot(solution_midpoint_100[0], solution_midpoint_100[1], label='Midpoint Rule N=100')
# N=13 is a straight line => Not expected and cannot be used.
# N=14 is getting there but still not perfect. May be used for large values but not for small
# N=15 is the first function that starts converging at a reasonable time.
# N=20 begins to approximate it better and then it can be seen that increasing the number of iterations will improve the approximation.
# N=100 nearly overlaps the solution.
plt.xlabel('Time (s)')
plt.ylabel('Concentration of A (mol/L)')
plt.title(f'Concentration of A over time with initial concentration of {initial_conc} mol/L', weight='bold')
plt.legend()
plt.ylim(-0.1, 5.5)
plt.show()
```

codes/Task 9.2.py

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate

# Taking code from task 8
l1_value = 1
l2_value = 1
def der_system_task10(t, x, m1=1, m2=1, l1=l1_value, l2=l2_value):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector (theta1, omega1, theta2, omega2) \n'''
    dxdt = np.zeros(4) # 4 equations
    delta = x[2] - x[0]
    dxdt[0] = x[1]
    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)
    dxdt[2] = x[3]
    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)
    return dxdt


def master_function(fun,tspan, y0, method='rk2', number_of_points
   =1000):
    '''General function to solve system of differential equations.
       Does not work on single differential equations. \n
    fun = function
    y0 = vector of initial conditions
    optional:\n
    method = You can select the method with which your system of
       differential equations will be evaluated. Default set to second
        order Runge-Kutta. \n
    Supported methods : midpoint method ('midpoint'), euler method ('
       euler'), Classical second order Runge-Kutta ('rk2'), classical
       fourth order Runge-Kutta ('rk4').
    number_of_points = how many steps. Default set to 100. Increasing
       this reduces error but increases computation time. '''
    dt = (tspan[1] - tspan[0])/number_of_points
    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.
    for i in range(len(y0)): #initial conditions as a loop to ensure
       universability.
        y[0,i] = y0[i]
    if method == 'midpoint':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
```

```
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            y[i+1,:] = y[i,:] + dt * k2
    elif method == 'euler':
        for i in range(number_of_points):
            y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
    elif method == 'rk2':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt, y[i] + dt*k1)
            y[i+1,:] = y[i] + dt*0.5*(k1+k2)
    elif method == 'rk4':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
            k4 = fun(t[i] +dt, y[i,:] + dt*k3)
            y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
    else:
        return 'Unknown method specified. Check documentation for
           supported methods' # In case an unknown method is specified
    return t, y

#(theta1, omega1, theta2, omega2)
ini_cond_vec = [7*np.pi/180, 1, 0.89*np.pi, 1] # Initial conditions a
    vector
tspan = [0,10]

approx_sol = master_function(der_system_task10, tspan, ini_cond_vec,
    method='rk4')
## For the approx solution don't use euler. It is unreliable and
    unsteady



x_1st = np.sin(approx_sol[1][:,0])*l1_value #Like the first node of
    the double pendulum
y_1st = -np.cos(approx_sol[1][:,0])*l1_value

x_2 = np.sin(approx_sol[1][:,0])*l1_value + np.sin(approx_sol[1][:,2])
    *l2_value
y_2 = -np.cos(approx_sol[1][:,0])*l1_value - np.cos(approx_sol
    [1][:,2])*l2_value

x_connector = [0, x_1st[-1], x_2[-1]]
y_connector = [0, y_1st[-1], y_2[-1]]


fig, ax = plt.subplots(figsize=(8,8))
plt.plot(x_1st, y_1st, label='Path of second node')
plt.plot(x_2, y_2, label='Path of second node')
plt.xlabel('Position in x')
plt.ylabel('Position in y')
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)
plt.suptitle(f'Path of double pendulum', weight='bold')
plt.xlim(-2,2)
plt.ylim(-2,2)


#Animating the path taken
line, = ax.plot(x_1st, y_1st, marker='o')

plt.show(block=False)
x_connector = [0, x_1st[-1], x_2[-1]]
y_connector = [0, y_1st[-1], y_2[-1]]

for i in range(len(approx_sol[1][:,0])):
    line.set_data([0, x_1st[i], x_2[i]], [0, y_1st[i], y_2[i]])
    fig.canvas.draw()
    fig.canvas.flush_events()
#    plt.pause(0.00000000001)
# Uncomment above like to slow down the animation

plt.show()
```

codes/Task 10.py

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
import pandas as pd
import time
# Taking code from task 10
# Needed later

l1_value = 1
l2_value = 1
m1_value = 1
m2_value = 1

def der_system_task10(t, x, m1=m1_value, m2=m2_value, l1=l1_value, l2=
    l2_value):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector (theta1, omega1, theta2, omega2) \n'''
    dxdt = np.zeros(4) # two equations
    delta = x[2] - x[0]
    dxdt[0] = x[1]
    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)
    dxdt[2] = x[3]
    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)
    return dxdt


def master_function(fun,tspan, y0, method='rk2', number_of_points=100)
    :
    '''General function to solve system of differential equations.
        Does not work on single differential equations. \n
    fun = function
    y0 = vector of initial conditions
    optional:\n
    method = You can select the method with which your system of
        differential equations will be evaluated. Default set to second
         order Runge-Kutta. \n
    Supported methods : midpoint method ('midpoint'), euler method ('
        euler'), Classical second order Runge-Kutta ('rk2'), classical
        fourth order Runge-Kutta ('rk4').
    number_of_points = how many steps. Default set to 100. Increasing
        this reduces error but increases computation time. '''
    dt = (tspan[1] - tspan[0])/number_of_points
    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.
    for i in range(len(y0)): #initial conditions as a loop to ensure
        universability.
        y[0,i] = y0[i]
    if method == 'midpoint':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            y[i+1,:] = y[i,:] + dt * k2
    elif method == 'euler':
        for i in range(number_of_points):
            y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
    elif method == 'rk2':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt, y[i] + dt*k1)
            y[i+1,:] = y[i] + dt*0.5*(k1+k2)
    elif method == 'rk4':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
            k4 = fun(t[i] +dt, y[i,:] + dt*k3)
            y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
    else:
        return 'Unknown method specified. Check documentation for
            supported methods' # In case an unknown method is specified
    return t, y

#(theta1, omega1, theta2, omega2)
ini_cond_vec = [7*np.pi/180, 1, 0.9*np.pi, 1] # Initial conditions a
    vector
tspan = [0,10]

approx_sol = master_function(der_system_task10, tspan, ini_cond_vec, '
    midpoint', 2000) #Midpoint used because its better than euler and
    easy to implement :)


# from:
# http://www.physics.umd.edu/hep/drew/pendulum2.html
# We can get the equations for the kinetic and potential energy
# the time derivative in terms of theta1 is simply omega1

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])

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])


## Correcting for the zero of energy
```

```
PE += (m1_value+m2_value)*9.81*(l1_value+l2_value)
## Without this correction we would see that the potential energy would be negative and that the sum of energies would be negative.
## Here we set the zero of energy to the lowest possible point the pendulum can reach.
## With l1=1 and l2=1 the lowest point would be y=-2. Thus, we must correct for this.



fig = plt.figure()
plt.plot(approx_sol[0], KE, label='Kinetic Energy')
plt.plot(approx_sol[0], PE, label='Potential Energy')
plt.plot(approx_sol[0], KE + PE, label='Sum of both')
plt.xlabel('Time (s)')
plt.ylabel('Energy (J)')
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)
plt.suptitle('Potential and Kinetic Energy of the Double Pendulum')
plt.legend()
plt.show()
## We can see that the graph matches expectations and the total energy is conserved.
## Moreover, the shape of the graphs alligns with the expectations that as kinetic energy increases potential decreases and vice versa .+


methods = ['midpoint', 'euler', 'rk2', 'rk4']
rows = []

for i in range(len(methods)):
    start_time = time.time()
    solution_loop = master_function(der_system_task10, tspan, ini_cond_vec, 'midpoint', 2000)
    end_time = time.time() - start_time
    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])
    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])
    PE_loop += (m1_value+m2_value)*9.81*(l1_value+l2_value)
    rows.append([methods[i], np.median(KE_loop+PE_loop), end_time])

df = pd.DataFrame(rows, columns = ['Method Used', 'Total Energy', 'Time taken (s)'])
print(df)
```

codes/Task 11.py

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
import pandas as pd
import time
# Taking code from task 10
# Needed later

l1_value = 1
l2_value = 1
m1_value = 1
m2_value = 1

def der_system_task10(t, x, m1=m1_value, m2=m2_value, l1=l1_value, l2=l2_value):
    '''Solves the following system of equations: \n
    dCadt = -A*exp(-Ea/RT) * Ca^3 \n
    dTdt = (deltaH * -A*exp(-Ea/RT) * Ca^3)/(rho*Cp) \n
    t =  vector for time span \n
    x = vector (theta1, omega1, theta2, omega2) \n'''
    dxdt = np.zeros(4) # two equations
    delta = x[2] - x[0]
    dxdt[0] = x[1]
    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)
    dxdt[2] = x[3]
    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)
    return dxdt


def master_function(fun,tspan, y0, method='rk2', number_of_points=100):
    '''General function to solve system of differential equations. Does not work on single differential equations. \n
    fun = function
    y0 = vector of initial conditions
    optional:\n
    method = You can select the method with which your system of differential equations will be evaluated. Default set to second order Runge-Kutta. \n
    Supported methods : midpoint method ('midpoint'), euler method ('euler'), Classical second order Runge-Kutta ('rk2'), classical fourth order Runge-Kutta ('rk4').
    number_of_points = how many steps. Default set to 100. Increasing this reduces error but increases computation time. '''
    dt = (tspan[1] - tspan[0])/number_of_points
    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.
    for i in range(len(y0)): #initial conditions as a loop to ensure
        universability.
        y[0,i] = y0[i]
    if method == 'midpoint':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            y[i+1,:] = y[i,:] + dt * k2
    elif method == 'euler':
        for i in range(number_of_points):
            y[i+1,:] = y[i,:] + dt * fun(t[i], y[i,:])
    elif method == 'rk2':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt, y[i] + dt*k1)
            y[i+1,:] = y[i] + dt*0.5*(k1+k2)
    elif method == 'rk4':
        for i in range(number_of_points):
            k1 = fun(t[i], y[i,:])
            k2 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k1)
            k3 = fun(t[i] + dt*0.5, y[i,:] + 0.5*dt*k2)
            k4 = fun(t[i] +dt, y[i,:] + dt*k3)
            y[i+1,:] = y[i] + dt*((1/6)*k1 + (1/3)*(k2+k3) + (1/6)*k4)
    else:
        return 'Unknown method specified. Check documentation for
            supported methods' # In case an unknown method is specified
    return t, y

#(theta1, omega1, theta2, omega2)

for i in range(1,5):
    ini_cond_vec = [7*np.pi/180, 1, 0.9-0.01*i*np.pi, 1] # Initial
        conditions a vector
    tspan = [0,5]
    approx_sol = master_function(der_system_task10, tspan,
        ini_cond_vec, 'midpoint', 2000) #Midpoint used because its
        better than euler and easy to implement :)
    x_2 = np.sin(approx_sol[1][:,0])*l1_value + np.sin(approx_sol
        [1][:,2])*l2_value
    y_2 = -np.cos(approx_sol[1][:,0])*l1_value - np.cos(approx_sol
        [1][:,2])*l2_value
    plt.plot(x_2, y_2, label=f'Theta2 = {0.9-0.01*i*np.pi: .4f}')


plt.xlabel('Position in x')
plt.ylabel('Position in y')
plt.suptitle(f'Path of double pendulum at various values of theta2',
    weight='bold')
plt.legend(loc='upper right')

plt.show()
```

codes/Task 11 investigation.py