Assignment 2 - Great Lakes

Group number 9, 1796097, 1732722

1 Summary

This report follows along the guided set of steps necessary to solve the investigation "Great Lakes", with the scope to estimate the concentration of polychlorinated biphenyl (PCB) in the Great Lakes. The first step creates a system of linear equations from the mass balances of each lake. The matrix was later solved with a variety of methods (np.linalg.solve, Gaussian elimination, LU Decomposition, Jacobi and Gauss-Seidel vectorized and non-vectorized), and the function times were compared. Using the production rate equation given, PCB concentration in each lake was calculated using 'np.linalg.solve' at different bypass concentrations. To continue, a new flow rate Q_{MO} was defined to investigate the effect a bypass from one lake would have on other lakes in the system. The results were plotted to visualize the outcome of such a bypass. np.linalg.solve was again used to solve the matrix with the addition of a system of plug flow reactors (PFR), done by adding 1000 rows to the original matrix. The dependency of PFR on PCB concentration was described in Task 5. Given a function to model the deposition of PCB in Lake Superior, the lower, middle and upper Riemann sums were identified as well as the Simpson and Trapezoid rules to approximate the area under the curve with the lowest possible error. Observations were made based on each total discharge, time taken and percentage error (see task 6). In addition to the previous methods, the Monte-Carlo integration was also used with an increasing number of points. Generally, an increased iteration number decreased the area's percentage error. A table was created containing the Riemann sums, Trapezoid, and Simpson rules as well as the Monte-Carlo method at various iterations to compare each method's calculated area and its corresponding percentage error. To continue, a function was created to calculate the Lagrange interpolation polynomial of the data points. Secondly, the data were interpolated using spline interpolation from the 'scipy' library and the two techniques were compared on a graph to assess each method to determine the smoothest function. Finally, the sensitivity and bypass system of each lake to inlet concentration by identifying a range of values where average yearly concentrations lie.

2 Results and discussion

1. Based on the assignment it is clear for this task that we must apply the mass balance and solve for the unknown variables. First, we apply the general balance equation:

$$\text{Accumulation} = \text{Inflow} - \text{Outflow} + \text{Production} \tag{1}$$

In addition to this, we can also apply the steady state approximation which assumes that there is no accumulation which simplifies the equation to:

$$0 = \text{Inflow} - \text{Outflow} + \text{Production} \tag{2}$$

First applying the mass balance to Lake Superior:

$$0 = S_{in,S} - C_S \cdot Q_{SH} \tag{3}$$

$$\frac{S_{in,S}}{Q_{SH}} = C_S \tag{4}$$

$$C_S = \frac{S_{in}}{Q_{SH}} \tag{5}$$

$$\tag{6}$$

Note that as the value of $S_{in,S}$ is unknown we simply take $S_{in} = 180\text{kg}\,\text{yr}^{-1}$. Next taking a look at Lake Michigan:

$$0 = S_{in,M} - C_M \cdot Q_{MH} \tag{7}$$

$$C_M = \frac{S_{in,M}}{Q_{MH}} \tag{8}$$

Now, using equations (5) and (8) we can solve for the mass balance of Lake Huron:

$$0 = S_{in,H} + Q_{SH} \cdot C_S + Q_{MH} \cdot C_M - C_H \cdot Q_{HE} \tag{9}$$

$$C_H = \frac{1}{Q_{HE}} \cdot \left(S_{in,H} + Q_{SH} \cdot \frac{S_{in}}{Q_{SH}} + Q_{MH} \frac{S_{in,M}}{Q_{MH}} \right) \tag{10}$$

$$C_H = \frac{S_{in,H}}{Q_{HE}} + \frac{S_{in}}{Q_{HE}} + \frac{S_{in,M}}{Q_{HE}} \tag{11}$$

Following this we can now get the equation for Lake Erie using equation (11):

$$0 = S_{in,E} + Q_{HE} \cdot C_H - Q_{EO} \cdot C_E \tag{12}$$

$$C_E = \frac{1}{Q_{EO}} \left(S_{in,E} + S_{in,H} + S_{in} + S_{in,M} \right) \tag{13}$$

$$C_E = \frac{S_{in,E}}{Q_{EO}} + \frac{S_{in,H}}{Q_{EO}} + \frac{S_{in}}{Q_{EO}} + \frac{S_{in,M}}{Q_{EO}} \tag{14}$$

Finally, we can now solve for the mass balance on Lake Ontario using equation (14):

$$0 = S_{in,O} + Q_{EO} \cdot C_E - Q_{OO} \cdot C_O \tag{15}$$

$$C_O = \frac{1}{Q_{OO}} \left(S_{in,O} + S_{in,E} + S_{in,H} + S_{in} + S_{in,M} \right) \tag{16}$$

$$C_O = \frac{S_{in,O}}{Q_{OO}} + \frac{S_{in,E}}{Q_{OO}} + \frac{S_{in,H}}{Q_{OO}} + \frac{S_{in}}{Q_{OO}} + \frac{S_{in,M}}{Q_{OO}} \tag{17}$$

This sort of method is analogous to solving a system of linear equations. Doing so may take a lot of time thus for the rest of this assignment, matrices were employed. As the derivations above simply solve for the unknown variable, C_i, a system of linear equations can be generated in the following way:

$$\begin{cases} S_{in,S} &= Q_{SH} \cdot C_S \\ S_{in,M} &= Q_{MH} \cdot C_M \\ S_{in,H} &= -Q_{SH} \cdot C_S - Q_{MH} \cdot C_M + C_H \cdot Q_{HE} \\ S_{in,E} &= -Q_{HE} \cdot C_H + Q_{EO} \cdot C_E \\ S_{in,O} &= -Q_{EO} \cdot C_E + Q_{OO} \cdot C_O \end{cases} \tag{18}$$

2. For this task the matrix was given by defining each lake's mass balance at steady-state, then the Q and S values were listed for each lake and the coefficient matrix was computed (see Listing 1). This was done as for all lakes, we were tasked with solving for the unknown variable, the concentration of PCBs. To continue, the matrix-vector equation was solved with methods such as Gaussian elimination, LU decomposition, Jacobi, and Gauss-Seidel. Many of the functions that were necessary to perform these methods were imported from the file 'gaussianjordan.py', given in Lecture 4, and 'it_methods.py', given in Lecture 5. Lastly the function 'np.linalg.solve' was used as a fast "Black-Box" method. Note that the function Gauss-Seidel and the corresponding vector function are modified Jacobi methods but allow updated variables to be used as well as previous iteration values to solve linear systems such as the matrix defined in Listing 1. The code for the Gauss-Seidel and vector Gauss-Seidel functions are shown in Listing 2 and 3 respectively. Gaussian

```
#Define the values of the Q's (given in km^-3 / yr)
Q_SH = 72
Q_MH = 38
Q_HE = 160
Q_EO = 185
Q_OO = 215

#Define values for the different PCB source S_in's (given in kg/yr)

S_in   = 180
S_in_M = 810
S_in_H = 630
S_in_E = 2750
S_in_O = 3820

#Generating our coefficient matrix
M = np.array([[Q_SH,0,0,0,0], [0,Q_MH,0,0,0], [-Q_SH, -Q_MH, Q_HE, 0,
    0], [0,0,-Q_HE, Q_EO, 0], [0,0,0,-Q_EO,Q_OO]])
#print(M) #Uncomment to see the coefficient matrix
```

Listing 1: Task 2 Matrix

elimination and LU-decomposition (classified as direct methods) are preferred for short codes but have an increased risk of computation errors. Similarly, Jacobi and Gauss-Seidel (iterative) methods are preferred for complex and sparse codes, although convergence is not guaranteed for all linear systems.

The time calculation as was mentioned in class was not able to properly calculate the time. Oftentimes, while timing certain functions it would simply return 0.0. After some research online it was discovered that using the time package is not always the best for accurate time measurements. Instead, the package 'timeit' was employed and it was able to accurately measure the time taken to run the functions. Moreover, as can be seen in **??**, for the 'Jacobi', and 'Gauss-Seidel' methods, the use of vectorization was also investigated. It can be seen that for this small set of equations using non-vectorized operation ends up being faster. However, it is presumed that using a larger set of variables would instead cause the vectorized calculations to become faster. The column for accuracy is there to check whether the output of the iterative methods directly matches the solution provided by 'np.linalg.solve'. As can be seen, all iterative methods were able to find the exact solution in a small number of iterations and a very short time. Finally, it is important to note that the times taken to run this code may change on the user's computer. Thus the numbers simply serve as a means to compare methods relative to each other, not the exact values of the times.

Table 1: Various solution methods and the time taken to execute them.

Solution method	**Time taken (s)**	**Number of iterations**	**Accuracy**
np.linalg.solve	0.000628500	N/A	N/A
Gaussian Elimination	0.000288800	N/A	N/A
LU Decomposition	0.000220900	N/A	N/A
Jacobi not Vectorized	0.000163000	5.00	TRUE
Jacobi Vectorized	0.000268700	5.00	TRUE
Gauss-Seidel not Vectorized	0.000042500	2.00	TRUE
Gauss-Seidel Vectorized	0.000081900	2.00	TRUE

3. For this task the production rate was added to each mass balance using the first order reaction

```python
def gaussseidel(A, b, tol=1e-2):
    # Set initial guess
    x = b + 1e-16

    # Initialize variables
    x_diff = 1
    N = A.shape[0]
    it_gaussseidel = 1

    # While not converged or max_it not reached
    while (x_diff > tol and it_gaussseidel < 1000):
        x_old = x.copy()
        for i in range(N):
            s = 0
            s2 = 0
            for j in range(N):
                if j < i:
                    # Sum off-diagonal*x_old
                    s += A[i,j] * x[j]
                if j > i:
                    # Second summation
                    s2 += A[i,j] * x_old[j]
            # Compute new x value
            x[i] = (b[i] - s - s2) / A[i,i]

        # Increase number of iterations
        it_gaussseidel += 1
        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)

    # Print number of iterations
    #print(it_gaussseidel)

    return x, it_gaussseidel]
```

Listing 2: Definition of Gauss-Seidel function

```python
def gaussseidel_vec(A, b, tol=1e-2):
    # Set initial guess
    x = b + 1e-16

    # Initialize variables
    x_diff = 1
    N = A.shape[0]
    it_gaussseidel = 1

    # While not converged or max_it not reached
    while (x_diff > tol and it_gaussseidel < 1000):
        x_old = x.copy()
        for i in range(N):
            s = 0
            s2 = 0
            j_indices_new = np.arange(0,i)
            j_indices_old = np.arange(i+1, N)
            s += A[i,j_indices_new] @  x[j_indices_new]
            s2 += A[i,j_indices_old] @  x_old[j_indices_old]
            # Compute new x value
            x[i] = (b[i] - s - s2) / A[i,i]

        # Increase number of iterations
        it_gaussseidel += 1
        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)

    # Print number of iterations
    #print(it_gaussseidel)

    return x, it_gaussseidel
```

Listing 3: definition of Gauss-Seidel vector function

equation r=K1Cx. For each matrix column the Q and S values were altered accordingly as shown in the matrix below.

$$\begin{bmatrix} Q_{SH} + k1 \cdot V_S & 0 & 0 & 0 & 0 \\ 0 & Q_{MH} + k1 \cdot V_M & 0 & 0 & 0 \\ -Q_{SH} & -Q_{MH} & Q_{HE} + k1 \cdot V_H & 0 & 0 \\ 0 & 0 & -Q_{HE} & Q_{EO} + k1 \cdot V_E & 0 \\ 0 & 0 & 0 & -Q_{EO} & Q_{OO} + k1 \cdot V_E \end{bmatrix}$$

The matrix was then solved using np.linalg.solve. Although in the previous step, it was shown that this method was the slowest for a matrix of this size, the speed differences appear to be imperceptible. Moreover, it was the most simple implementation available. Below one can find a table with the results of this system of linear equations.

Table 2: Concentration of PCBs in Different Lakes

Lake	**Concentration of PCBs**
Superior	1.26689189
Michigan	12.15924102
Huron	6.55767776
Erie	20.22983869
Ontario	34.72180463

4. For this task a new flow rate, Q_{MO} was defined. Lake Superior was not included in the output graphs as its PCB concentration is independent of Lake Michigan's PCB concentration. The new concentrations were calculated and stored in list and then a line plot was generated for each lake displaying PCB concentration $[\mathrm{kg\,yr^{-1}}]$ against bypass flow rate $[\mathrm{km^3\,yr^{-1}}]$ shown in Figure 4. The code can be seen in Listing 4.

```
Q_MO_inputs = np.linspace(0,100, 1000)

#Empty lists to store the values
conc_lake_michigan_values = []
conc_lake_huron_values = []
conc_lake_erie_values = []
conc_lake_ontario_values = []

#Lake Superior is not included because its independent of the lake
    Michigan
for Q_MO in Q_MO_inputs: # Calculate the concentraions in the lakes at
     various bypass flowrates
    M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M + Q_MO
        ,0,0,0], [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_EO
        + k1*V_E, 0], [0,-Q_MO,0,-Q_EO,Q_OO + k1*V_E]])
    sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
    conc_lake_michigan, conc_lake_huron, conc_lake_erie,
        conc_lake_ontario = np.linalg.solve(M,sol_vec)[1], np.linalg.
        solve(M,sol_vec)[2], np.linalg.solve(M,sol_vec)[3],np.linalg.
        solve(M, sol_vec)[4] #Yes its slower to calculate it 4 times
        but since it the matrix is so small it doesn't matter too much
    conc_lake_michigan_values.append(conc_lake_michigan)
    conc_lake_huron_values.append(conc_lake_huron)
    conc_lake_erie_values.append(conc_lake_erie)
    conc_lake_ontario_values.append(conc_lake_ontario)
```

Listing 4: Loop to calculate PCB concentration in various lakes at different bypass concentrations

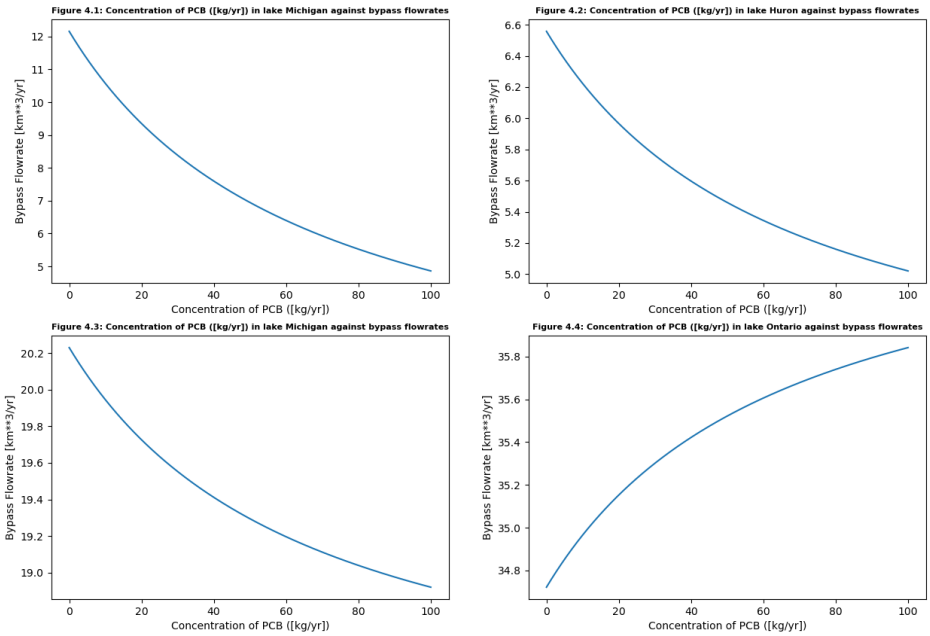

Figure 1: Effect of the bypass flow rate on the PCB concentration in various lakes

5. Adding the PFRs was initially quite difficult, however, it was quickly discovered that all that needed to be done was to add 1000 more rows and columns to the original matrix and then use any method to solve the system. It was observed that the concentration the first PFR is dependent on the concentration of Michigan and the concentration of lake Ontario is dependent on the last PFR. All the other PFRs were dependent on the PFR before them. Listing 5 shows how this system of PFRs was added to the existing set of linear equations. Once again the matrix was solved using numpy.linalg.solve. This was done as it was the easiest and simplest method. Moreover, the functionality of the other methods was not investigated on bigger systems.

6. For this task the group investigated various numerical methods for approximating the function:

$$f(t) = -0.1 \cdot t^3 + 0.58 \cdot t^2 \cosh\left(\frac{-1}{t+1}\right) + e^{\frac{t}{2.7}} + 9.6$$

In order to make the operations as universal as possible several functions were created to execute these methods. When calculating the Riemann sums, as which exact method to use was not specified, the group chose to investigate the lower, upper, and middle sums. Listing 6 shows one such function that was used. The other function can be found in the python file in the appendix. After these functions were defined, the areas were calculated and stored and the time take was also calculated. This was once again done using the 'timeit' package as the 'time' package would occasionally return a time of 0.0. The "exact" solution was found by typing the integral into a "TI-nspire CX CAS ii" graphical calculator and then the percentage error of the various methods was found by comparing the outputs to the "exact" solution. Finally, the results were printed in a table using pandas and the table can be seen in Table 3. As can be seen from the table the Simpson rule is able to approximate the function with the lowest error, however, at a higher computation time. What appears to be strange is that the middle Riemann sum seems to outperform the trapezoidal rule both in terms of calculation time and percentage error.

```
PFR_matrix = np.zeros((1005,1005))
new_sol_vec = np.zeros(1005)
M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M + Q_MO,0,0,0],
    [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_EO + k1*V_E,
   0], [0, 0,0,-Q_EO,Q_OO + k1*V_E]])
PFR_matrix[:5, :5] = M # Update the new matrix
PFR_matrix[4, -1] = -Q_MO # Inflow from the last PFR
sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
new_sol_vec[:5] = sol_vec
PFR_matrix[5, 1] = Q_MO # First PFR is dependent on the concentration
   of lake Michigan
PFR_matrix[5, 5] = -(Q_MO + V_tank*k2)

for i in range(999):
    PFR_matrix[6+i, 5+i] = Q_MO
    PFR_matrix[6+i, 6+i] = -(Q_MO + V_tank*k2)

solution = np.linalg.solve(PFR_matrix, new_sol_vec)
```

Listing 5: Code snippet showing how the new matrix was defined and solved

```
def riemann_sum_lower(func, a, b, intervals=20):
    '''Calculates the lower Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration
       bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set
       to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func(a+i*bin_size)*bin_size
    return sum
```

Listing 6: Code showing how the lower Riemann sum was defined and calculated

```python
def find_max(f,a,b):
    '''Finds the maximum of a function on a given domain.
    First argument is your function, second is the lowerbound, and
       third is the upperbound.'''
    minimize_result = scipy.optimize.minimize_scalar(lambda x: -f((b-a
       )*x), bounds=[0,1], method='bounded') #Finds maximum of a
       function. According to information found online it has to be
       bound from 0 to 1 thus the function is shrunk.
    return -minimize_result.fun # Returns the maximum y value of a
       function.
```

Listing 7: Code snippet showing how the group was able to find the max of a function on a domain

Table 3: Comparison of Solution Methods for Total Discharge into Lake Superior

Solution Method	Total Discharge (kg)	Time Taken (s)	Percentage Error
Lower Riemann Sum	161.877769	0.000048	0.947552
Upper Riemann Sum	158.949637	0.000047	0.878441
Middle Riemann Sum	160.331059	0.000048	0.016981
Trapezoid Rule	160.413703	0.000087	0.034556
Simpson Rule	160.358607	0.000143	0.000198

7. In order to apply the Monte-Carlo integration, first a function had to be made which would calculate the maximum of a function on a given domain. Some functions available in the 'scipy' library are able to find the maximum of a function but over all real numbers. Instead it was discovered that 'scipy.optimize.minimize_scalar' was able to find the minimum of a function on the domain from 0 to 1. Thus the function was shrunk to fit the domain and was inverted such the maximum of the original function would now correspond to the minimum of the inverted function around the x-axis. Listing 7 shows the function that was used. The functionality of this function was verified by finding the maximum of the function using the aforementioned graphical calculator. Now that the maximum value of the function can be found, the Monte-Carlo integration could be performed. Listing 8 shows the function that was used to calculate the area below the curve. The variable 'approx_area_counter' stores the number of points that lie below the curve which corresponds to the N_0 as given in the equation found in the assignment document. Table 4 shows how changing the umber of iterations increases the accuracy of the Monte-Carlo integration. A general trend can be observed, namely that increasing the number of iterations (so increasing the value of N) reduces the percentage error of the approximate area. It is important to recognize that this method does use some randomness thus, running the code in two instances will yield different percentage errors.

Table 4: Total Discharge into Lake Superior using Monte-Carlo Integration with Different Iterations

Total Discharge (kg)	Number of Iterations	Percentage Error
156.721273	1000	2.268057
160.694489	2000	0.209655
161.577426	5000	0.760257
160.782782	10000	0.264715
159.745332	20000	0.382243
159.966066	50000	0.244592
160.879905	100000	0.325281

```python
def monte_carlo(func, a, b, N=1000):
    '''Returns the approximate area under a curve using the Monte
       Carlo method.
    First argument is the function you want to integrate.
    Second argument is the lower bound and third is the upperbound.
    Optional third argument to specify how many random points to
       choose with default set to 1000.'''
    approx_area_counter = 0
    y_max = find_max(func, a, b)
    for i in range(N):
        x_rand = np.random.uniform(a,b) ## random number on the given
           domain
        y_rand = np.random.uniform(0,y_max) ## Area of the shape lies
           between y = 0 and y = ymax
        if y_rand < f(x_rand):
            approx_area_counter += 1
    approx_area = (y_max*(b-a)*approx_area_counter) / N
    return approx_area
```

Listing 8: Code showing how the Monte-Carlo integration method was defined

```python
a = sympy.Symbol('x', real=True) #x is already used for the list
```

Listing 9: Code snippet showing how sympy was used to store function variables.

Finally, Table 5 shows a comparison of all the methods that have been used.

Table 5: Comparison of all the methods used to find the area under a curve

Method	Calculated Area	Percentage Error
Lower Riemann Sum	161.877769	0.947552
Upper Riemann Sum	158.949637	0.878441
Middle Riemann Sum	160.331059	0.016981
Trapezoid Rule	160.413703	0.034556
Simpson Rule	160.358607	0.000198
Monte-Carlo with N=1000	167.316514	4.339173
Monte-Carlo with N=2000	161.025590	0.416131
Monte-Carlo with N=5000	161.577426	0.760257
Monte-Carlo with N=10000	160.937296	0.361070
Monte-Carlo with N=20000	160.970407	0.381718
Monte-Carlo with N=50000	160.217703	0.087671
Monte-Carlo with N=100000	160.517901	0.099534

8. For this task, the group made use of the 'sympy' library to calculate the Lagrange interpolation of the data points. First, the letter 'a' was used to sore the sympy symbol of x. This was done as from the assignment document, the variable 'x' was already taken. This can be seen in Listing 9 Once this completed, the Lagrange interpolant could be created using several for loops. The code can be seen in listing 10. The code runs as described by the frame found in the assignment document. In order to verify the correct functionality of the code, the group also used on online Lagrange interpolation calculator and found that it yielded the same function. While it was not required by the assignment, the group also plotted the Lagrange inerpolant with the initial points to visually see

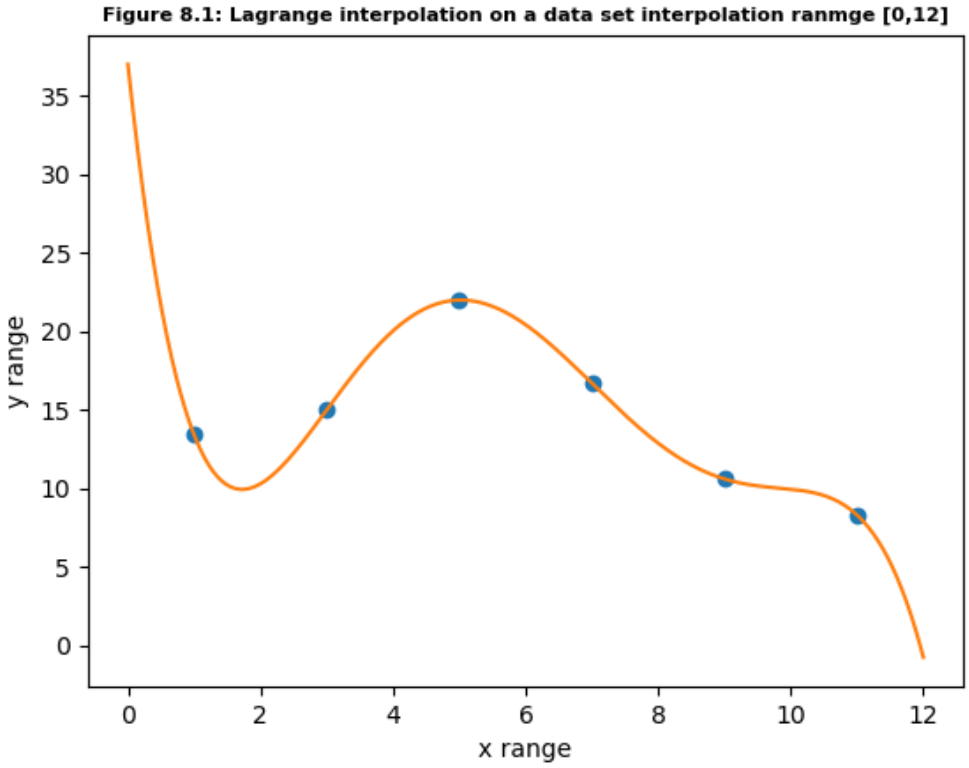

Figure 2: Task 8 Lagrange Interpolation polynomial on interval [0,12]

what the interpolated function looks like. This was done by using the 'sympy.lambdify' function which converted the sympy function into one that could be used for plotting.

9. As a second interpolation technique the group used spline interpolation. In short, this method generates a piecewise function of several cubics which generates an interpolation of the data. This was done by using the 'scipy' library as can be seen in Listing 11.

   Next, using the interpolation found in the previous task, the two interpolation techniques were plotted on the same graph which allowed for comparison between the interpolation techniques.

```python
#From assignment document
x = list ( range (1 ,12 ,2) )
y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]

def Lagrange_interpolation(g,h):
    '''Returns the Lagrange polynomial using Lagrange interpolation.
    Arguments g and h both of which must be lists'''
    assert len(g) == len(h), "Every x coordinate must have a
       corresponding y coordinate"
    output_storage = 0
    for k in range(len(g)):
        polynomial_storage = 1  ## Defining polynomial storage and
           resetting after every k
        for i in range(len(g)):
            if i!=k:
                polynomial_storage = polynomial_storage * (a-g[i])/(g[
                   k]-g[i])
        output_storage = output_storage + h[k]*polynomial_storage
    output_storage = sympy.simplify(output_storage) #This line is not
       exactly necessary but the function become really ugly otherwise
    return output_storage

function = sympy.lambdify(a, Lagrange_interpolation(x,y))
```

Listing 10: Definition of the lagrange interpolation

```python
spline_eq = scipy.interpolate.splrep(x,y)
y_for_plotting_spline = scipy.interpolate.splev(x_for_plotting,
   spline_eq)
```

Listing 11: Code snippet showing how the spline interpolation was implemented

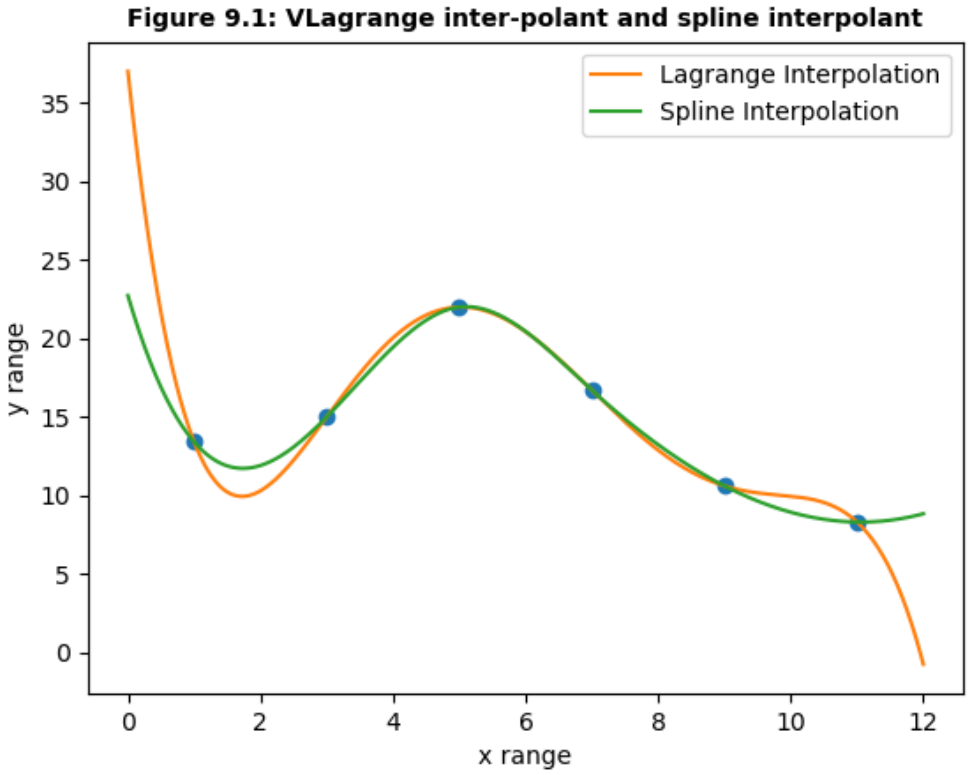

Figure 3: Task 9 graph Lagrange inter-polant and spline interpolant

Figure 9 shows the two functions plotted on the same axis. Initially, it can be observed that both interpolation techniques pass through all 6 data points. However, the Lagrange interpolant seems to be a worse method compared to spline interpolation, especially when talking about the extrapolation capabilities which can be seen at the edges of the graph. As the Lagrange interpolant is a continuous polynomial, it must approach positive and negative infinities at very low and very high values of x respectively. Because of this, it appears as though in the last month, the expected discharge rate would be 0. Spline interpolation appears to be "smoother' and especially around the second month it is able to interpolate the data more gently.

10. For the final task the group had to access the sensitivity of the lakes at various inlet concentrations of Lake Superior. However, the group still was not certain as to which inlet concentration to choose. The group wanted to find a range of possible values where the true average yearly inlet concentration could lie. It was observed that the inlet concentration provided in task 1 was the highest and the inlet concentration provided by the function in task 6 was the lowest. Moreover, the integral of the 2 interpolation techniques was also calculated which can be seen in Listing 12. It was found that the integral of both interpolations was ≈ 175 which lies in the range of $160 < S_{in,S} < 180$. The code from task 5 was slightly modified to have a for loop which allowed the code to calculate the PCB concentrations in the various lakes as a function of the inlet concentration of Lake superior. As before, Lake Michigan was not included in the final graphs as its concentration is independent of the inlet concentration of Lake Superior. Listing 13 shows how this code was written. The group used 0.25 steps and appended each data point to a list of each lake.

```
print(f'Integral of the spline interpolant: {scipy.integrate.quad(
    lambda x: scipy.interpolate.splev(x, spline_eq), 0, 12)[0]}')
print(f'Integral of the Lagrange interpolant: {scipy.integrate.quad(
    function, 0, 12)[0]}')
print(f'Integral of the function from task 6: {scipy.integrate.quad(
    lambda x: f(x), 0, 12)[0]}')
```

Listing 12: Code showing how the integral was calculated for various interpolation methods

```
x_task_10 = []
Superior_list = []
Huron_list = []
Erie_list = []
Ontario_list = []
for j in range(81): #
    x_task_10.append(160 + j/4) # Will be needed later to generate our
        plots
    sol_vec = np.array([160 + j/4, S_in_M, S_in_H,S_in_E, S_in_O])
    new_sol_vec[:5] = sol_vec
    sup, mich, hur, er, ont = np.linalg.solve(PFR_matrix, new_sol_vec)
       [0:5] # solve the matrix and store the solutions
    Superior_list.append(sup) ## Appends the solutions to the lists
    Huron_list.append(hur) ## Needed for plotting
    Erie_list.append(er)
    Ontario_list.append(ont)
```

Listing 13: Calculating concentrations in various lakes at different inlet concentrations

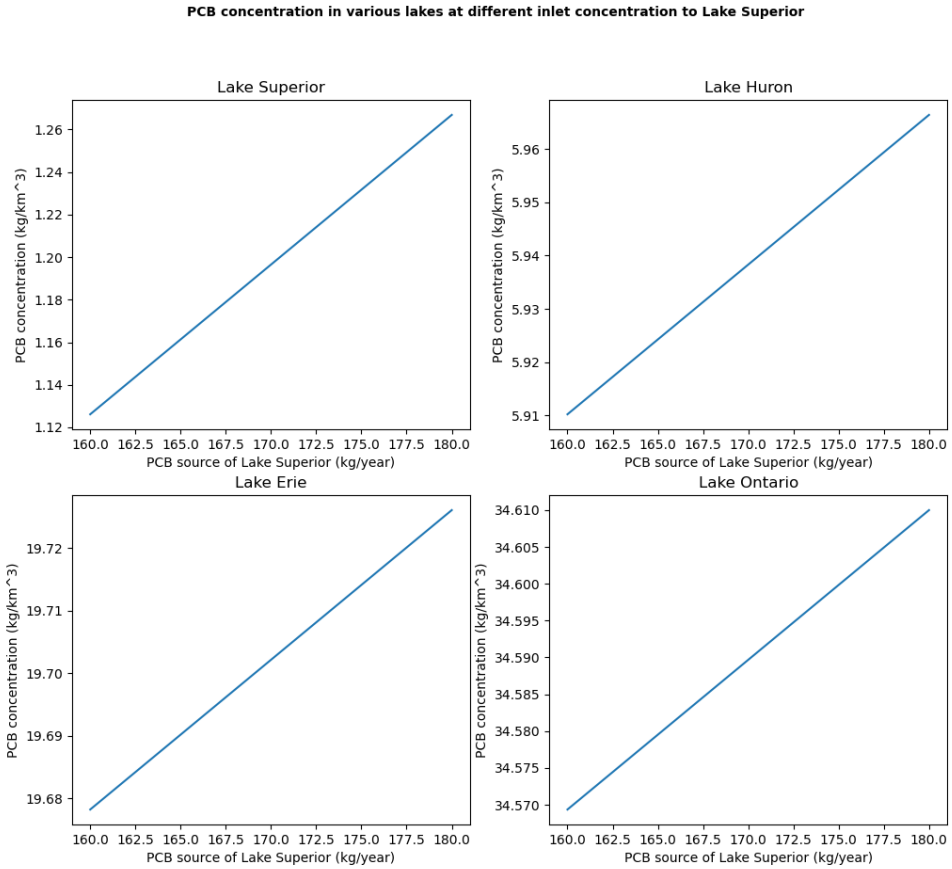

Figure 4: Concentration of PCBs in various lakes at different inlet concentrations of Lake Superior.

Figure 4 provides a visualization to how the concentration of PCBs change at various inlet concentration of Lake Superior. However, The graph is slightly limited as all graphs appear to have the same slope and thus it is difficult to evaluate their sensitivity to the concentration of Lake Superior. Since, these functions were linear, all that was needed to calculate the slope was simply:

$$\frac{\partial C_i}{\partial S_{in,S}} = \frac{C_{i,2} - C_{i,1}}{S_{in,S,1} - S_{in,S,1}}$$

Table 6: Rate of Change of Concentration in Various Lakes

Lake	Rate of Change
Lake Superior	0.007038
Lake Huron	0.002808
Lake Erie	0.002393
Lake Ontario	0.002032

Table 6 shows that the rate of change of concentration decreases the further away the lake is from

Lake superior. This is expected as the further one goes from the source, the effect of the inlet concentration gets dampened.

3 Reflection

Overall this assignment was a success. We were able to follow how matrices can be used to solve real world issues. We also investigated several methods of how to solve an equation of the type $A \cdot x = b$, many of which were not previously known. The techniques used to approximate the area of a function were familiar to us, however, neither group member had ever used them before this assignment. This provided an interesting insight into how one can numerically integrate a function that may not have an analytical solution. The interpolation techniques were also new and gave us insight on different methods of interpolation and doing the Lagrange interpolation method ourselves allowed us to look into the 'black-box' method.

List of symbols

Symbol	Unit	Definition
$S_{in,i}$	$\mathrm{kg\,yr^{-1}}$	PCB source in lake i
Q_{ij}	$\mathrm{km^3\,yr^{-1}}$	Flow rate from lake i to lake j
C_i	$\mathrm{kg\,km^{-3}}$	PCB concentration in lake i
V_i	$\mathrm{km^3}$	Volume of lake i

```
# Steady-state assumption for the mass balance (Accumulation is approx. 0)
# Derivations can be found in the report, here only the final answers are given

print("Rearranged equations for the concentrations of PCBs in various lakes: \n\n Lake Superior: C_S = S_in / Q_SH \n\n Lake Michigan : C_M=S_(in,M)/Q_MH \n\nLake Huron: C_H=S_(in,H)/Q_HE +S_in/Q_HE +S_(in,M)/Q_HE \n\n Lake Erie: C_E=S_(in,E)/Q_EO +S_(in,H)/Q_EO +S_in/Q_EO +S_(in,M)/Q_EO \n\n Lake Ontario: C_O=S_(in,O)/Q_OO +S_(in,E)/Q_OO +S_(in,H)/Q_OO +S_in/Q_OO +S_(in,M)/Q_OO \n")

print("The mass balances for the lakes read as follows, \n\n Lake Superior: S_(in,S)= Q _ S H C_S  \n\n Lake Michigan : S_(in,M)= Q _ M H C_M  \n\n Lake Huron: S_(in,H)=- Q _ S H C_S - Q _ M H C_M + C _ H Q_HE  \n\n Lake Erie:S_(in,E)=- Q _ H E C_H + Q _ E O C_E  \n\n Lake Ontario: S_(in,O)=- Q _ E O C_E + Q _ O O C_O  \n")
```

codes/Task 1.py

```python
#Libraries
import numpy as np
import timeit
from scipy.linalg import lu
import pandas as pd

# Functions taken from gaussjordan.py (Given in lecture 4)
def swap_rows(mat,i1,i2):
    """Swap two rows in a matrix/vector"""
    temp = mat[i1,...].copy()
    mat[i1,...] = mat[i2,...]
    mat[i2,...] = temp

def gaussian_eliminate_v2(A,b):
    """Perform elimination to obtain an upper triangular matrix

    Input:
    A: Coefficient matrix
    b: right hand side

    Returns:
    Aprime, bprime: row echelon form of matrix A and rhs vector b"""
    A = np.array(A,dtype=np.float64)
    b = np.array(b,dtype=np.float64)

    assert A.shape[0] == A.shape[1], "Coefficient matrix should be
        square"

    N = len(b)
    for col in range(N-1):
        index = np.argmax(np.abs(A[col:, col])) + col
        swap_rows(A,col,index)
        swap_rows(b,col,index)
        for row in range(col+1,N):
            d = A[row,col] / A[col,col]
            A[row,:] = A[row,:] - d * A[col,:]
            b[row] = b[row] - d * b[col]

    return A,b

def backsubstitution_v1(U,b):
    """Back substitutes an upper triangular matrix to find x in Ax=b
        """
    x = np.empty_like(b)
    N = len(b)

    for row in range(N)[::-1]:
        x[row] = (b[row] - np.sum(U[row,row+1:] * x[row+1:])) / U[row,
            row]

    return x

def forwardsubstitution(L,d):
```

```
    N = len(L)
    y = np.empty_like(d)

    for row in range(N):
        y[row] = (d[row] - np.sum(L[row,:row] * y[:row])) / L[row,row]

    return y

#Taken from it_methods.py
def jacobi(A, b, tol=1e-2):

    # Set initial guess
    x = b + 1e-16

    # Initialize variables
    x_diff = 1
    N = A.shape[0]
    it_jac = 1

    # While not converged or max_it not reached
    while (x_diff > tol and it_jac < 1000):
        x_old = x.copy()
        for i in range(N):
            s = 0
            for j in range(N):
                if j != i:
                    # Sum off-diagonal*x_old
                    s += A[i,j] * x_old[j]
            # Compute new x value
            x[i] = (b[i] - s) / A[i,i]

        # Increase number of iterations
        it_jac += 1
        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)

    # Print number of iterations
    #print(it_jac)

    return x, it_jac

def jacobi_vec(A, b, tol=1e-2, itmax=1000):
    # Set initial guess
    x = b + 1e-16

    # Initialize variables
    x_diff = 1
    N = A.shape[0]
    it_jac = 1

    # While not converged or max_it not reached
    while (x_diff > tol and it_jac < 1000):
        x_old = x.copy()
        for i in range(N):
            s = 0
```

```
            j_indices = np.concatenate((np.arange(0,i), np.arange(i+1,
                N)))
            s += A[i,j_indices] @  x_old[j_indices]
            # Compute new x value
            x[i] = (b[i] - s) / A[i,i]

        # Increase number of iterations
        it_jac += 1
        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)

    # Print number of iterations
    #print(it_jac)

    return x, it_jac

def gaussseidel(A, b, tol=1e-2):
    # Set initial guess
    x = b + 1e-16

    # Initialize variables
    x_diff = 1
    N = A.shape[0]
    it_gaussseidel = 1

    # While not converged or max_it not reached
    while (x_diff > tol and it_gaussseidel < 1000):
        x_old = x.copy()
        for i in range(N):
            s = 0
            s2 = 0
            for j in range(N):
                if j < i:
                    # Sum off-diagonal*x_old
                    s += A[i,j] * x[j]
                if j > i:
                    # Second summation
                    s2 += A[i,j] * x_old[j]
            # Compute new x value
            x[i] = (b[i] - s - s2) / A[i,i]

        # Increase number of iterations
        it_gaussseidel += 1
        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)

    # Print number of iterations
    #print(it_gaussseidel)

    return x, it_gaussseidel

def gaussseidel_vec(A, b, tol=1e-2):
    # Set initial guess
    x = b + 1e-16

    # Initialize variables
```

```
    x_diff = 1
    N = A.shape[0]
    it_gaussseidel = 1

    # While not converged or max_it not reached
    while (x_diff > tol and it_gaussseidel < 1000):
        x_old = x.copy()
        for i in range(N):
            s = 0
            s2 = 0
            j_indices_new = np.arange(0,i)
            j_indices_old = np.arange(i+1, N)
            s += A[i,j_indices_new] @  x[j_indices_new]
            s2 += A[i,j_indices_old] @  x_old[j_indices_old]
            # Compute new x value
            x[i] = (b[i] - s - s2) / A[i,i]

        # Increase number of iterations
        it_gaussseidel += 1
        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)

    # Print number of iterations
    #print(it_gaussseidel)

    return x, it_gaussseidel


#task 2 matrix
#Define the values of the Q's (given in km^-3 / yr)
Q_SH = 72
Q_MH = 38
Q_HE = 160
Q_EO = 185
Q_OO = 215

#Define values for the different PCB source S_in's (given in kg/yr)

S_in   = 180
S_in_M = 810
S_in_H = 630
S_in_E = 2750
S_in_O = 3820

#Volumes (km^3)

V_S = 12000
V_M = 4900
V_H = 3500
V_E = 480
V_O = 1640



#Generating our coefficient matrix
```

```
M = np.array([[Q_SH,0,0,0,0], [0,Q_MH,0,0,0], [-Q_SH, -Q_MH, Q_HE, 0, 0], [0,0,-Q_HE, Q_EO , 0], [0,0,0,-Q_EO,Q_OO]])
#print(M) #Uncomment to see the coefficient matrix


#Solution Vector
sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
#print(sol_vec) #Uncomment to see the Solution Vector



#Using a normal the solver from numpy

solver_code = lambda: np.linalg.solve(M,sol_vec)
Time_taken_solver = timeit.timeit(solver_code, number=1)


#Gaussian Elimination
#Using code provided in Lecture 4 (gaussjordan.py)

Time_taken_gauss = timeit.timeit(lambda: backsubstitution_v1(*gaussian_eliminate_v2(M, sol_vec)), number=1) # * asterisk used to get the outputs of the gauss_eliminate_v2 function

#LU decomposition



def LU_function_for_timing():
    '''Function only written for task 2 for timing LU decomposition. This function only serves one purpose.'''
    P, L, U = lu(M)
    d = P @ sol_vec
    y= forwardsubstitution(L, d)
    lu_function = lambda: backsubstitution_v1( U,y) # time.time() prints out 0.0 otherwise
end_time_lu = timeit.timeit(LU_function_for_timing ,number=1)

#Jacobi (Function taken from lecture 5)
# First the non-vectorized


jacobi_code = lambda:  jacobi(M, sol_vec)
end_time_Jacobi = timeit.timeit(jacobi_code, number=1)

jacobi_sol, jacobi_number_of_iterations = jacobi(M, sol_vec) #Yes code is ran twice but it runs so quickly that it doesnt matter. Plus again here using time.time() would print 0.0

#Then the vectorized

jacobi_sol_vec, jacobi_number_of_iterations_vec = jacobi_vec(M, sol_vec)

```

```
Jacobi_vec_code = lambda: jacobi_vec(M, sol_vec)
end_time_Jacobi_vec = timeit.timeit(Jacobi_vec_code, number=1)

#Gauss-Seidel
#First the non-vectorized


gaussseidel_sol, gaussseidel_number_of_iterations = gaussseidel(M,
    sol_vec)

gaussseidel_function = lambda:gaussseidel(M,sol_vec)
end_time_gaussseidel = timeit.timeit(gaussseidel_function, number=1)




#Then the vectorized

gaussseidel_sol_vec, gaussseidel_number_of_iterations_vec =
    gaussseidel_vec(M, sol_vec)

gaussseidel_function_vec = lambda:gaussseidel_vec(M, sol_vec)
end_time_gaussseidel_vec = timeit.timeit(gaussseidel_function_vec,
    number=1)



#Uncomment to see the output as as text
#print(f"Time for various solvers given in seconds, np.solve:{
    Time_taken_solver}, Gaussian elimination:{Time_taken_gauss}, LU
    decomposition:{end_time_lu}, Jacobi (not vectorized):{
    end_time_Jacobi} with {jacobi_number_of_iterations} iterations,
    Jacobi (Vectorized):{end_time_gaussseidel_vec} with {
    jacobi_number_of_iterations_vec} iterations, Gauss-Seidel (not
    vectorized):{end_time_gaussseidel} with {
    gaussseidel_number_of_iterations} iterations, Gauss-seidel (
    vectorized):{end_time_gaussseidel_vec} with {
    gaussseidel_number_of_iterations_vec} iterations")

#To generate table
#Columns

def accuracy_checker(A,B):
    equality = False
    if np.array_equal(A,B):
        equality = True
        return equality
    else:
        return equality
```

```
rows = [['np.linalg.solve', Time_taken_solver, 'N/A', 'N/A'], ['
    Gaussian Elimination', Time_taken_gauss, 'N/A', 'N/A'], ['LU
    Decomposition', end_time_lu, 'N/A', 'N/A'], ['Jacobi not Vectorized
    ', end_time_Jacobi, jacobi_number_of_iterations, accuracy_checker(
    jacobi(M, sol_vec)[0], np.linalg.solve(M,sol_vec))], ['Jacobi
    Vectorized', end_time_Jacobi_vec, jacobi_number_of_iterations_vec,
    accuracy_checker(jacobi_vec(M, sol_vec)[0], np.linalg.solve(M,
    sol_vec))], ['Gauss-Seidel not Vectorized', end_time_gaussseidel,
    gaussseidel_number_of_iterations, accuracy_checker(gaussseidel(M,
    sol_vec)[0], np.linalg.solve(M,sol_vec))], ['Gauss-Seidel
    Vectorized', end_time_gaussseidel_vec,
    gaussseidel_number_of_iterations_vec, accuracy_checker(
    gaussseidel_vec(M, sol_vec)[0], np.linalg.solve(M,sol_vec))]]


df = pd.DataFrame(rows, columns = ['Solution method', 'Time taken (s)'
    , 'Number of iterations', 'Accuracy'])
df['Time taken (s)'] = df['Time taken (s)'].apply(lambda x: f'{x:.11f}
    ')

print(df)
```

codes/Task 2.py

```python
#Libraries
import numpy as np
import timeit
import time
from scipy.linalg import lu

#task 2 matrix
#Define the values of the Q's (given in km^-3 / yr)
Q_SH = 72
Q_MH = 38
Q_HE = 160
Q_EO = 185
Q_OO = 215

#Define values for the different PCB source S_in's (given in kg/yr)

S_in   = 180
S_in_M = 810
S_in_H = 630
S_in_E = 2750
S_in_O = 3820

#Volumes (km^3)

V_S = 12000
V_M = 4900
V_H = 3500
V_E = 480
V_O = 1640

#Defining k1
k1 = 0.00584


#Generating our coefficient matrix
M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M,0,0,0], [-Q_SH
    , -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_EO + k1*V_E, 0],
   [0,0,0,-Q_EO,Q_OO + k1*V_E]])
#print(M) #Uncomment to see the coefficient matrix


#Solution Vector
sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])


print(np.linalg.solve(M, sol_vec))
```

codes/Task 3.py

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

#task 2 matrix
#Define the values of the Q's (given in km^-3 / yr)
Q_SH = 72
Q_MH = 38
Q_HE = 160
Q_EO = 185
Q_OO = 215

#Define values for the different PCB source S_in's (given in kg/yr)

S_in   = 180
S_in_M = 810
S_in_H = 630
S_in_E = 2750
S_in_O = 3820

#Volumes (km^3)

V_S = 12000
V_M = 4900
V_H = 3500
V_E = 480
V_O = 1640

#Defining k1
k1 = 0.00584


#New for task 4, we need to define a new QMO
Q_MO_inputs = np.linspace(0,100, 1000)

#Empty lists to store the values
conc_lake_michigan_values = []
conc_lake_huron_values = []
conc_lake_erie_values = []
conc_lake_ontario_values = []

#Lake Superior is not included because its independent of the lake
    Michigan
for Q_MO in Q_MO_inputs: # Calculate the concentraions in the lakes at
     various bypass flowrates
    M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M + Q_MO
        ,0,0,0], [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_EO
        + k1*V_E, 0], [0,-Q_MO,0,-Q_EO,Q_OO + k1*V_E]])
    sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
    conc_lake_michigan, conc_lake_huron, conc_lake_erie,
        conc_lake_ontario = np.linalg.solve(M,sol_vec)[1], np.linalg.
        solve(M,sol_vec)[2], np.linalg.solve(M,sol_vec)[3],np.linalg.
        solve(M, sol_vec)[4] #Yes its slower to calculate it 4 times
```

```
        but since it the matrix is so small it doesn't matter too much
    conc_lake_michigan_values.append(conc_lake_michigan)
    conc_lake_huron_values.append(conc_lake_huron)
    conc_lake_erie_values.append(conc_lake_erie)
    conc_lake_ontario_values.append(conc_lake_ontario)



#Making figures
fig = plt.figure(figsize=(15,10))
ax1 = plt.subplot(2, 2, 1)
ax1.plot(Q_MO_inputs, conc_lake_michigan_values)
ax1.set_title("Figure 4.1: Concentration of PCB ([kg/yr]) in lake
   Michigan against bypass flowrates", size=8, weight='bold')
ax1.set_xlabel('Concentration of PCB ([kg/yr])')
ax1.set_ylabel('Bypass Flowrate [km**3/yr]')

ax2 = plt.subplot(2,2,2)
ax2.plot(Q_MO_inputs, conc_lake_huron_values)
ax2.set_title("Figure 4.2: Concentration of PCB ([kg/yr]) in lake
   Huron against bypass flowrates", size=8, weight='bold')
ax2.set_xlabel('Concentration of PCB ([kg/yr])')
ax2.set_ylabel('Bypass Flowrate [km**3/yr]')

ax3 = plt.subplot(2,2,3)
ax3.plot(Q_MO_inputs, conc_lake_erie_values)
ax3.set_title("Figure 4.3: Concentration of PCB ([kg/yr]) in lake
   Michigan against bypass flowrates", size=8, weight='bold')
ax3.set_xlabel('Concentration of PCB ([kg/yr])')
ax3.set_ylabel('Bypass Flowrate [km**3/yr]')

ax4 = plt.subplot(2,2,4)
ax4.plot(Q_MO_inputs, conc_lake_ontario_values)
ax4.set_title("Figure 4.4: Concentration of PCB ([kg/yr]) in lake
   Ontario against bypass flowrates", size=8, weight='bold')
ax4.set_xlabel('Concentration of PCB ([kg/yr])')
ax4.set_ylabel('Bypass Flowrate [km**3/yr]')


plt.show()
```

codes/Task 4.py

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

#task 2 matrix
#Define the values of the Q's (given in km^-3 / yr)
Q_SH = 72
Q_MH = 38
Q_HE = 160
Q_EO = 185
Q_OO = 215

#Define values for the different PCB source S_in's (given in kg/yr)

S_in   = 180
S_in_M = 810
S_in_H = 630
S_in_E = 2750
S_in_O = 3820

#Volumes (km^3)

V_S = 12000
V_M = 4900
V_H = 3500
V_E = 480
V_O = 1640

#Defining k1
k1 = 0.00584

#Defining k2
k2 = 2 * 10**7

#Volume of tank is given in m^3 but everything else is in km^3. Thus, the volume is given in km^3
V_tank = 1*10**-9


# Not given in the question so taken as 100
Q_MO = 20



## For PFR Graph
# Making the matrix

PFR_matrix = np.zeros((1005,1005))
new_sol_vec = np.zeros(1005)
M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M + Q_MO,0,0,0], [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_EO + k1*V_E, 0], [0, 0,0,-Q_EO,Q_OO + k1*V_E]])
PFR_matrix[:5, :5] = M # Update the new matrix
```

```
PFR_matrix[4, -1] = -Q_MO # Inflow from the last PFR
sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
new_sol_vec[:5] = sol_vec
PFR_matrix[5, 1] = Q_MO # First PFR is dependent on the concentration
    of lake Michigan
PFR_matrix[5, 5] = -(Q_MO + V_tank*k2)

for i in range(999):
    PFR_matrix[6+i, 5+i] = Q_MO
    PFR_matrix[6+i, 6+i] = -(Q_MO + V_tank*k2)

solution = np.linalg.solve(PFR_matrix, new_sol_vec)
print(solution[0:5]) # Prints the concentraion of the PCBs in the
    lakes


x = np.linspace(1, 1000, 1000)
ax1 = plt.subplot(1,1,1)
ax1.plot(x, np.linalg.solve(PFR_matrix,new_sol_vec)[5:])
ax1.set_title("Figure 5.1: Concentration profile of PCB in the PFR",
    size=11, weight='bold')
ax1.set_xlabel('PFR Tank Number')
ax1.set_ylabel('Concentration of PCB ([kg/yr]')

plt.show()
```

codes/Task 5.py

```python
import numpy as np
import timeit
import pandas as pd

def f(t):
    return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t/2.7) + 9.6

# t is given in months so we are looking for the integral from 0 to 12 (One full year)
# Since we want 20 intervals, every step will be 12/20


## The functions here were defined ourselves and not taken from lecture 8
def riemann_sum_lower(func, a, b, intervals=20):
    '''Calculates the lower Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func(a+i*bin_size)*bin_size
    return sum

def riemann_sum_upper(func, a, b, intervals=20):
    '''Calculates the upper Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func(a+(i+1)*bin_size)*bin_size
    return sum

def riemann_sum_middle(func, a, b, intervals=20):
    '''Calculates the middle Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func((a+i*bin_size + a+(i+1)*bin_size)/2)*bin_size
    return sum


def trapezoid_rule(func, a, b, intervals=20):
```

```
    '''Calculates the area of a function using the trapezoid rule.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += (bin_size * (func(a+i*bin_size) + func(a+(i+1)*bin_size)))/2
    return sum

def simpson_rule(func, a, b, intervals=20):
    '''Calculates the area of a function using the Simpson rule.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        a_loop = a+i*bin_size
        b_loop = a+(i+1)*bin_size
        sum += (func(a_loop)+4*func((a_loop+b_loop)/(2))+func(b_loop)) * ((b_loop-a_loop)/(6))
    return sum



#Calculating the times and outputs of functions
#Code remains constant and timeit is being used because time.time gave 0.0 sometimes
riemann_area_lower = riemann_sum_lower(f,0,12)
riemann_code_lower = lambda:riemann_sum_lower(f,0,12)
end_time_riemann_lower = timeit.timeit(riemann_code_lower, number=1)


riemann_area_upper = riemann_sum_upper(f,0,12)
riemann_code_upper = lambda:riemann_sum_upper(f,0,12)
end_time_riemann_upper = timeit.timeit(riemann_code_upper, number=1)


riemann_area_middle = riemann_sum_middle(f,0,12)
riemann_code_middle = lambda:riemann_sum_middle(f,0,12)
end_time_riemann_middle = timeit.timeit(riemann_code_middle, number=1)


trapezoid_area = trapezoid_rule(f,0,12)
trapezoid_code = lambda:trapezoid_rule(f,0,12)
end_time_trapezoid = timeit.timeit(trapezoid_code, number=1)

simpson_area = simpson_rule(f,0,12)
simpson_code = lambda:simpson_rule(f,0,12)
end_time_simpson = timeit.timeit(simpson_code, number=1)
```

```python

#"Actual" value obtained from graphical display calculator

actual_result = 160.3582902978

#Needed to calculate the percentage error
def percentage_error(a, b):
    '''Returns the percentage error of the function
    First argument is the observed value and second argument is the
       expected value'''
    return np.abs((a-b)/(b))*100


#Generating rows of the output table
rows = [['Lower Riemann Sum', riemann_area_lower,
   end_time_riemann_lower, percentage_error(riemann_area_lower,
   actual_result)], ['Upper Riemann Sum', riemann_area_upper,
   end_time_riemann_upper, percentage_error(riemann_area_upper,
   actual_result)], ['Middle Riemann Sum', riemann_area_middle,
   end_time_riemann_middle, percentage_error(riemann_area_middle,
   actual_result)], ['Trapezoid Rule', trapezoid_area,
   end_time_trapezoid, percentage_error(trapezoid_area, actual_result)
   ], ['Simpson Rule', simpson_area, end_time_simpson,
   percentage_error(simpson_area, actual_result)]]


df = pd.DataFrame(rows, columns = ['Solution method', 'Total discharge
    into lake superior (kg)', 'Time taken (s)', 'Percentage error from
    expected value'])


print(df)
```

codes/Task 6.py

```python
#Libraries
import numpy as np
import scipy
from scipy import optimize
import pandas as pd

#Same function as in task 6
def f(t):
    return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t/2.7) + 9.6

#We need to find its maximum
def find_max(f,a,b):
    '''Finds the maximum of a function on a given domain.
    First argument is your function, second is the lowerbound, and third is the upperbound.'''
    minimize_result = scipy.optimize.minimize_scalar(lambda x: -f((b-a)*x), bounds=[0,1], method='bounded') #Finds maximum of a function. According to information found online it has to be bound from 0 to 1 thus the function is shrunk.
    return -minimize_result.fun # Returns the maximum y value of a function.


#From gdc, y max ~= 18.394515656
#print(find_max(f,0,12)) #Uncomment in case you want to verify that the function works


def monte_carlo(func, a, b, N=1000):
    '''Returns the approximate area under a curve using the Monte Carlo method.
    First argument is the function you want to integrate.
    Second argument is the lower bound and third is the upperbound.
    Optional third argument to specify how many random points to choose with default set to 1000.'''
    approx_area_counter = 0
    y_max = find_max(func, a, b)
    for i in range(N):
        x_rand = np.random.uniform(a,b) ## random number on the given domain
        y_rand = np.random.uniform(0,y_max) ## Area of the shape lies between y = 0 and y = ymax
        if y_rand < f(x_rand):
            approx_area_counter += 1
    approx_area = (y_max*(b-a)*approx_area_counter) / N
    return approx_area

#Taken from task 6, needed for the table
def percentage_error(a, b):
    '''Returns the percentage error of the function
    First argument is the observed value and second argument is the expected value'''
```

```
    return np.abs((a-b)/(b))*100




#Same as task 6, taken from graphical display calculator.
actual_result = 160.3582902978


rows = []
big_numbers_for_table = [1000, 2000, 5000, 10000, 20000, 50000,
    100000]

for i in big_numbers_for_table:
    area_loop = monte_carlo(f,0,12,i)
    error_loop = percentage_error(area_loop, actual_result)
    rows.append([area_loop, i, error_loop])

df = pd.DataFrame(rows, columns = ['Total discharge into lake superior
    (kg)', 'Number of Iterations', 'Percentage error from expected
    value'])

print(df)
```

codes/Task 7.py

```python
# The sole purpose of this file is to generate the table to compare methods from task 6 to the methods from task 7.
# As this is a continuation of task 7, the comments have been removed. They can be found in task 7.py
# Libraries Required
import numpy as np
import pandas as pd
import numpy as np
import scipy
from scipy import optimize
import pandas as pd


#Functions
# We know that we could use for example "import Task 6" to get the functions but we wanted to make sure that
# the code runs properly on all devices.
# Thus we simply copy-pasted the funtions into this file to make sure that the correctors can run the code perfectly
def f(t):
    return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t/2.7) + 9.6

def riemann_sum_lower(func, a, b, intervals=20):
    '''Calculates the lower Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func(a+i*bin_size)*bin_size
    return sum

def riemann_sum_upper(func, a, b, intervals=20):
    '''Calculates the upper Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func(a+(i+1)*bin_size)*bin_size
    return sum

def riemann_sum_middle(func, a, b, intervals=20):
    '''Calculates the middle Riemann sum of a function.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
```

```
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += func((a+i*bin_size + a+(i+1)*bin_size)/2)*bin_size
    return sum

def trapezoid_rule(func, a, b, intervals=20):
    '''Calculates the area of a function using the trapezoid rule.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        sum += (bin_size * (func(a+i*bin_size) + func(a+(i+1)*bin_size)))/2
    return sum

def simpson_rule(func, a, b, intervals=20):
    '''Calculates the area of a function using the Simpson rule.
    Takes 3 arguments, first the function, then the lower integration bound, and finally the upper integration bound.
    Optional 4th argument to specify number of intervals (default set to 20)'''
    bin_size = (b-a)/intervals
    sum = 0
    for i in range(intervals):
        a_loop = a+i*bin_size
        b_loop = a+(i+1)*bin_size
        sum += (func(a_loop)+4*func((a_loop+b_loop)/(2))+func(b_loop)) * ((b_loop-a_loop)/(6))
    return sum

def find_max(f,a,b):
    '''Finds the maximum of a function on a given domain.
    First argument is your function, second is the lowerbound, and third is the upperbound.'''
    minimize_result = scipy.optimize.minimize_scalar(lambda x: -f((b-a)*x), bounds=[0,1], method='bounded') #Finds maximum of a function. According to information found online it has to be bound from 0 to 1 thus the function is shrunk.
    return -minimize_result.fun # Returns the maximum y value of a function.

def monte_carlo(func, a, b, N=1000):
    '''Returns the approximate area under a curve using the Monte Carlo method.
    First argument is the function you want to integrate.
    Second argument is the lower bound and third is the upperbound.
    Optional third argument to specify how many random points to choose with default set to 1000.'''
    approx_area_counter = 0
    y_max = find_max(func, a, b)
```

```
    for i in range(N):
        x_rand = np.random.uniform(a,b) ## random number on the given
            domain
        y_rand = np.random.uniform(0,y_max) ## Area of the shape lies
            between y = 0 and y = ymax
        if y_rand < f(x_rand):
            approx_area_counter += 1
    approx_area = (y_max*(b-a)*approx_area_counter) / N
    return approx_area

def percentage_error(a, b):
    '''Returns the percentage error of the function
    First argument is the observed value and second argument is the
        expected value'''
    return np.abs((a-b)/(b))*100

riemann_area_lower = riemann_sum_lower(f,0,12)

riemann_area_upper = riemann_sum_upper(f,0,12)

riemann_area_middle = riemann_sum_middle(f,0,12)

trapezoid_area = trapezoid_rule(f,0,12)

simpson_area = simpson_rule(f,0,12)

actual_result = 160.3582902978


rows = [['Lower Riemann Sum', riemann_area_lower, percentage_error(
    riemann_area_lower, actual_result)], ['Upper Riemann Sum',
    riemann_area_upper, percentage_error(riemann_area_upper,
    actual_result)], ['Middle Riemann Sum', riemann_area_middle,
    percentage_error(riemann_area_middle, actual_result)], ['Trapezoid
    Rule', trapezoid_area,  percentage_error(trapezoid_area,
    actual_result)], ['Simpson Rule', simpson_area, percentage_error(
    simpson_area, actual_result)]]


big_numbers_for_table = [1000, 2000, 5000, 10000, 20000, 50000,
    100000]

for i in big_numbers_for_table:
    area_loop = monte_carlo(f,0,12,i)
    error_loop = percentage_error(area_loop, actual_result)
    rows.append(['Monte-Carlo with N='+str(i),area_loop, error_loop])




df = pd.DataFrame(rows, columns = ['Method', 'Calculated Area', '
    Percentage error from expected value'])

print(df)
```

codes/Task 7 continued.py

```python
#Libraries
import numpy as np
import sympy
import matplotlib.pyplot as plt
#Defining symbols
a = sympy.Symbol('x', real=True) #x is already used for the list


#From assignment document
x = list ( range (1 ,12 ,2) )
y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]

def Lagrange_interpolation(g,h):
    '''Returns the Lagrange polynomial using Lagrange interpolation.
    Arguments g and h both of which must be lists'''
    assert len(g) == len(h), "Every x coordinate must have a
        corresponding y coordinate"
    output_storage = 0
    for k in range(len(g)):
        polynomial_storage = 1  ## Defining polynomial storage and
            resetting after every k
        for i in range(len(g)):
            if i!=k:
                polynomial_storage = polynomial_storage * (a-g[i])/(g[
                    k]-g[i])
        output_storage = output_storage + h[k]*polynomial_storage
    output_storage = sympy.simplify(output_storage) #This line is not
        exactly necessary but the function become really ugly otherwise
    return output_storage

function = sympy.lambdify(a, Lagrange_interpolation(x,y))


# Plotting
x_for_plotting = np.linspace(0, 12, 1000)
y_for_plotting = function(x_for_plotting)

fig = plt.subplot()
ax1 = fig.plot(x, y, marker='o', linestyle='None')
ax2 = fig.plot(x_for_plotting, y_for_plotting)
fig.set_title("Figure 8.1: Concentration of PCB ([kg/yr]) in lake
    Michigan against bypass flowrates", size=8, weight='bold')
fig.set_xlabel('Concentration of PCB ([kg/yr])')
fig.set_ylabel('Bypass Flowrate [km**3/yr]')

plt.show()
```

codes/Task 8.py

```
#Libraries
import numpy as np
import sympy
from scipy import interpolate
import matplotlib.pyplot as plt
import scipy

#Prerequisites from the previous task, needed to plot the lagrange interpolant
a = sympy.Symbol('x', real=True) #x is already used for the list
p = sympy.Symbol('y', real=True) #y already used for list as well
#From assignment document
x = list ( range (1 ,12 ,2) )
y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]

def Lagrange_interpolation(g,h):
    '''Returns the Lagrange polynomial using Lagrange interpolation.
    Arguments g and h both of which must be lists'''
    assert len(g) == len(h), "Every x coordinate must have a corresponding y coordinate"
    output_storage = 0
    for k in range(len(g)):
        polynomial_storage = 1  ## Defining polynomial storage and resetting after every k
        for i in range(len(g)):
            if i!=k:
                polynomial_storage = polynomial_storage * (a-g[i])/(g[k]-g[i])
        output_storage = output_storage + h[k]*polynomial_storage
    output_storage = sympy.simplify(output_storage) #This line is not exactly necessary but the function become really ugly otherwise
    return output_storage
#Needed to generate the plots of the lagrange interpolant
function = sympy.lambdify(a, Lagrange_interpolation(x,y))
x_for_plotting = np.linspace(0, 12, 1000)
y_for_plotting_lagrange = function(x_for_plotting)



#The actual new code starts here
spline_eq = scipy.interpolate.splrep(x,y)
y_for_plotting_spline = scipy.interpolate.splev(x_for_plotting, spline_eq)
#According to the discussions on canvas, we were not asked to make the function ourselves



#Plotting
fig = plt.subplot()
ax1 = fig.plot(x, y, marker='o', linestyle='None')
ax2 = fig.plot(x_for_plotting, y_for_plotting_lagrange, label='Lagrange Interpolation')
```

```
ax3 = fig.plot(x_for_plotting, y_for_plotting_spline, label='Spline
    Interpolation')
plt.title('Various interpolation techniques on the Measured PCB
    discharge rate', size=10, weight='bold')
plt.xlabel("Month")
plt.ylabel("Discharge rate of PCB (kg/month)")
fig.legend()
plt.show()
```

codes/Task 9.py

```python
#Libraries
import numpy as np
import sympy
import matplotlib.pyplot as plt
import scipy
import scipy.interpolate ## I know it seems redundant to import scipy and scipy.interpolate separetly but otherwise the code does not run on the TU/e laptop
import scipy.integrate ## Same as comment above
import pandas as pd

#Prerequisites from the previous task
a = sympy.Symbol('x', real=True) #x is already used for the list
## No need for a y
#From assignment document
x = list ( range (1 ,12 ,2) )
y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]

def Lagrange_interpolation(g,h):
    '''Returns the Lagrange polynomial using Lagrange interpolation.
    Arguments g and h both of which must be lists'''
    assert len(g) == len(h), "Every x coordinate must have a corresponding y coordinate"
    output_storage = 0
    for k in range(len(g)):
        polynomial_storage = 1  ## Defining polynomial storage and resetting after every k
        for i in range(len(g)):
            if i!=k:
                polynomial_storage = polynomial_storage * (a-g[i])/(g[k]-g[i])
        output_storage = output_storage + h[k]*polynomial_storage
    output_storage = sympy.simplify(output_storage) #This line is not exactly necessary but the function become really ugly otherwise
    return output_storage
#Needed to generate the plots of the lagrange interpolant
function = sympy.lambdify(a, Lagrange_interpolation(x,y))
x_for_plotting = np.linspace(0, 12, 1000)
y_for_plotting_lagrange = function(x_for_plotting)
spline_eq = scipy.interpolate.splrep(x,y)
y_for_plotting_spline = scipy.interpolate.splev(x_for_plotting, spline_eq)
def f(t):
    return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t/2.7) + 9.6


## Below, we can see the integral of the interpolated function and that of q
print(f'Integral of the spline interpolant: {scipy.integrate.quad(lambda x: scipy.interpolate.splev(x, spline_eq), 0, 12)[0]}')
print(f'Integral of the Lagrange interpolant: {scipy.integrate.quad(function, 0, 12)[0]}')
```

```
print(f'Integral of the function from task 6: {scipy.integrate.quad(lambda x: f(x), 0, 12)[0]}')

## We can see that the integral from task 6 seems to underestimate while the initial assumption seems to overestimate. The two interpolants lie in between.
## What we can do is assume that our real source for lake superior lies somewhere in between 160 and 180
## Finally we can run a simulation at varrying inlet concentrations of lake superior and see the results,
## on the lakes that are dependent on it (All except for Lake Michigan )

## Below is code from task 5 required to perform the calculations

#task 2 matrix
#Define the values of the Q's (given in km^-3 / yr)
Q_SH = 72
Q_MH = 38
Q_HE = 160
Q_EO = 185
Q_OO = 215

#Define values for the different PCB source S_in's (given in kg/yr)

# Now S_in can take different values
S_in_M = 810
S_in_H = 630
S_in_E = 2750
S_in_O = 3820

#Volumes (km^3)

V_S = 12000
V_M = 4900
V_H = 3500
V_E = 480
V_O = 1640

#Defining k1
k1 = 0.00584

#Defining k2
k2 = 2 * 10**7

#Volume of tank is given in m^3 but everything else is in km^3. Thus, the volume is given in km^3
V_tank = 1*10**-9


# Not given in the question so taken as 100
Q_MO = 20


```

```
90
91  ## For PFR Graph
92  # Making the matrix
93
94  PFR_matrix = np.zeros((1005,1005))
95  new_sol_vec = np.zeros(1005)
96
97  M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M + Q_MO,0,0,0],
        [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_EO + k1*V_E,
    0], [0, 0,0,-Q_EO,Q_OO + k1*V_E]])
98  PFR_matrix[:5, :5] = M # Update the new matrix
99  PFR_matrix[4, -1] = -Q_MO # Inflow from the last PFR
100 PFR_matrix[5, 1] = Q_MO # First PFR is dependent on the concentration
        of lake Michigan
101 PFR_matrix[5, 5] = -(Q_MO + V_tank*k2)
102
103 for i in range(999):
104     PFR_matrix[6+i, 5+i] = Q_MO
105     PFR_matrix[6+i, 6+i] = -(Q_MO + V_tank*k2)
106
107
108 x_task_10 = []
109 Superior_list = []
110 Huron_list = []
111 Erie_list = []
112 Ontario_list = []
113 for j in range(81): #
114     x_task_10.append(160 + j/4) # Will be needed later to generate our
            plots
115     sol_vec = np.array([160 + j/4, S_in_M, S_in_H,S_in_E, S_in_O])
116     new_sol_vec[:5] = sol_vec
117     sup, mich, hur, er, ont = np.linalg.solve(PFR_matrix, new_sol_vec)
          [0:5] # solve the matrix and store the solutions
118     Superior_list.append(sup) ## Appends the solutions to the lists
119     Huron_list.append(hur) ## Needed for plotting
120     Erie_list.append(er)
121     Ontario_list.append(ont)
122
123
124 fig = plt.figure(figsize=(12,10))
125 fig.suptitle('PCB concentration in various lakes at different inlet
        concentration to Lake Superior', size=10, weight='bold')
126
127 axsup = plt.subplot(2,2,1)
128 axsup.plot(x_task_10, Superior_list)
129
130 axhur = plt.subplot(2,2,2)
131 axhur.plot(x_task_10, Huron_list)
132
133 axer = plt.subplot(2,2,3)
134 axer.plot(x_task_10, Erie_list)
135
136 axont = plt.subplot(2,2,4)
137 axont.plot(x_task_10, Ontario_list)
```

```
138
139 #y labels
140 axont.set_ylabel('PCB concentration (kg/km^3)')
141 axer.set_ylabel('PCB concentration (kg/km^3)')
142 axhur.set_ylabel('PCB concentration (kg/km^3)')
143 axsup.set_ylabel('PCB concentration (kg/km^3)')
144
145 #x labels
146 axont.set_xlabel('PCB source of Lake Superior (kg/year)')
147 axer.set_xlabel('PCB source of Lake Superior (kg/year)')
148 axhur.set_xlabel('PCB source of Lake Superior (kg/year)')
149 axsup.set_xlabel('PCB source of Lake Superior (kg/year)')
150
151 #axis titles
152 axont.set_title('Lake Ontario')
153 axer.set_title('Lake Erie')
154 axhur.set_title('Lake Huron')
155 axsup.set_title('Lake Superior')
156
157
158 plt.show()
159
160
161 def derivative(q,w):
162     '''Returns the derivative of a linear function. First argument is
        the x list and second arument is the y list'''
163     return (w[-1]-w[0])/(q[-1]-q[0])
164
165 rows = [['Lake Superior', derivative(x_task_10, Superior_list)], ['
    Lake Huron', derivative(x_task_10, Huron_list)], ['Lake Erie',
    derivative(x_task_10, Erie_list)], ['Lake Ontario', derivative(
    x_task_10, Ontario_list)]]
166
167 df = pd.DataFrame(rows, columns = ['Lake', 'Rate of change'])
168
169 print(df)
```

codes/Task 10.py