

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:

$$\text{Accumulation} = \text{Inflow} - \text{Outflow} + \text{Production} \quad (1)$$

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

$$0 = \text{Inflow} - \text{Outflow} + \text{Production} \quad (2)$$

First applying the mass balance to Lake Superior:

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

$$\frac{S_{in,S}}{Q_{SH}} = C_S \quad (4)$$

$$C_S = \frac{S_{in}}{Q_{SH}} \quad (5)$$

$$(6)$$

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

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

$$C_M = \frac{S_{in,M}}{Q_{MH}} \quad (8)$$

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

- 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

```

1 #Define the values of the Q's (given in km-3 / yr)
2 Q_SH = 72
3 Q_MH = 38
4 Q_HE = 160
5 Q_EO = 185
6 Q_OO = 215
7
8 #Define values for the different PCB source S_in's (given in kg/yr)
9
10 S_in = 180
11 S_in_M = 810
12 S_in_H = 630
13 S_in_E = 2750
14 S_in_O = 3820
15
16 #Generating our coefficient matrix
17 M = np.array([[Q_SH,0,0,0,0], [0,Q_MH,0,0,0], [-Q_SH, -Q_MH, Q_HE, 0,
18 0], [0,0,-Q_HE, Q_EO, 0], [0,0,0,-Q_EO,Q_OO]])
19 #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

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

```

1 def gaussseidel(A, b, tol=1e-2):
2     # Set initial guess
3     x = b + 1e-16
4
5     # Initialize variables
6     x_diff = 1
7     N = A.shape[0]
8     it_gaussseidel = 1
9
10    # While not converged or max_it not reached
11    while (x_diff > tol and it_gaussseidel < 1000):
12        x_old = x.copy()
13        for i in range(N):
14            s = 0
15            s2 = 0
16            for j in range(N):
17                if j < i:
18                    # Sum off-diagonal*x_old
19                    s += A[i,j] * x[j]
20                if j > i:
21                    # Second summation
22                    s2 += A[i,j] * x_old[j]
23            # Compute new x value
24            x[i] = (b[i] - s - s2) / A[i,i]
25
26        # Increase number of iterations
27        it_gaussseidel += 1
28        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)
29
30    # Print number of iterations
31    #print(it_gaussseidel)
32
33    return x, it_gaussseidel]

```

Listing 2: Definition of Gauss-Seidel function

```

1 def gaussseidel_vec(A, b, tol=1e-2):
2     # Set initial guess
3     x = b + 1e-16
4
5     # Initialize variables
6     x_diff = 1
7     N = A.shape[0]
8     it_gaussseidel = 1
9
10    # While not converged or max_it not reached
11    while (x_diff > tol and it_gaussseidel < 1000):
12        x_old = x.copy()
13        for i in range(N):
14            s = 0
15            s2 = 0
16            j_indices_new = np.arange(0,i)
17            j_indices_old = np.arange(i+1, N)
18            s += A[i,j_indices_new] @ x[j_indices_new]
19            s2 += A[i,j_indices_old] @ x_old[j_indices_old]
20            # Compute new x value
21            x[i] = (b[i] - s - s2) / A[i,i]
22
23        # Increase number of iterations
24        it_gaussseidel += 1
25        x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)
26
27    # Print number of iterations
28    #print(it_gaussseidel)
29
30    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

- 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 [kg yr^{-1}] against bypass flow rate [$\text{km}^3 \text{yr}^{-1}$] shown in Figure 4. The code can be seen in Listing 4.

```

1 Q_MO_inputs = np.linspace(0,100, 1000)
2
3 #Empty lists to store the values
4 conc_lake_michigan_values = []
5 conc_lake_huron_values = []
6 conc_lake_erie_values = []
7 conc_lake_ontario_values = []
8
9 #Lake Superior is not included because its independent of the lake
  Michigan
10 for Q_MO in Q_MO_inputs: # Calculate the concentraions in the lakes at
    various bypass flowrates
11     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]])
12     sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
13     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
14     conc_lake_michigan_values.append(conc_lake_michigan)
15     conc_lake_huron_values.append(conc_lake_huron)
16     conc_lake_erie_values.append(conc_lake_erie)
17     conc_lake_ontario_values.append(conc_lake_ontario)

```

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

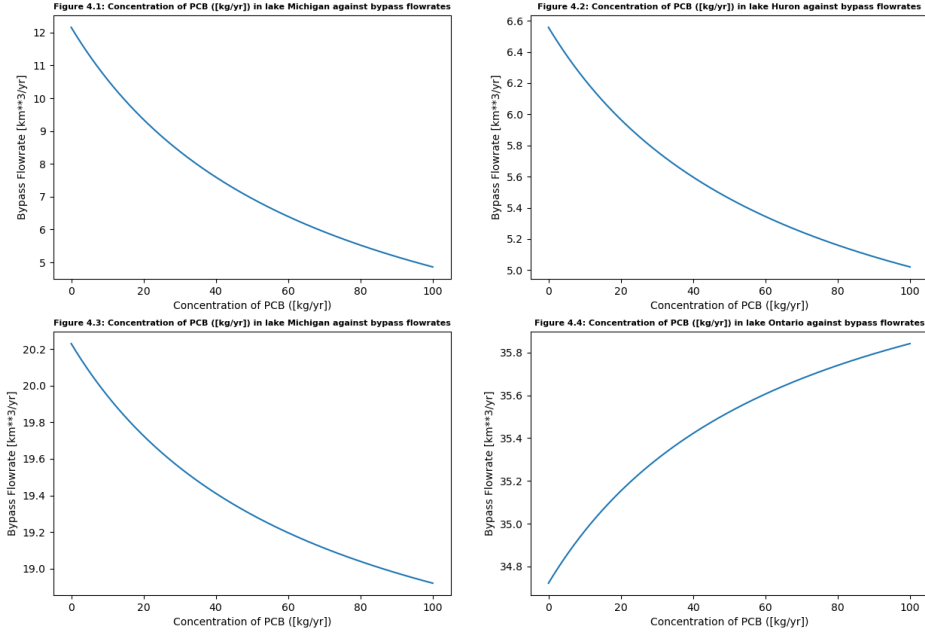


Figure 1: Effect of the bypass flow rate on the PCB concentration in various lakes

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.


```

1 PFR_matrix = np.zeros((1005,1005))
2 new_sol_vec = np.zeros(1005)
3 M = np.array([[Q_SH + k1*V_S,0,0,0,0], [0,Q_MH + k1*V_M + Q_M0,0,0,0],
4               [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_E0 + k1*V_E,
5               0], [0, 0,0,-Q_E0,Q_O0 + k1*V_E]])
6 PFR_matrix[:5, :5] = M # Update the new matrix
7 PFR_matrix[4, -1] = -Q_M0 # Inflow from the last PFR
8 sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
9 new_sol_vec[:5] = sol_vec
10 PFR_matrix[5, 1] = Q_M0 # First PFR is dependent on the concentration
11                             of lake Michigan
12 PFR_matrix[5, 5] = -(Q_M0 + V_tank*k2)
13
14 for i in range(999):
15     PFR_matrix[6+i, 5+i] = Q_M0
16     PFR_matrix[6+i, 6+i] = -(Q_M0 + V_tank*k2)
17
18 solution = np.linalg.solve(PFR_matrix, new_sol_vec)

```

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

```

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

```

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

```

1 def find_max(f,a,b):
2     '''Finds the maximum of a function on a given domain.
3     First argument is your function, second is the lowerbound, and
4     third is the upperbound.'''
5     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 number 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

```

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

```

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

```

1 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

- 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 store 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 an 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 interpolant with the initial points to visually see

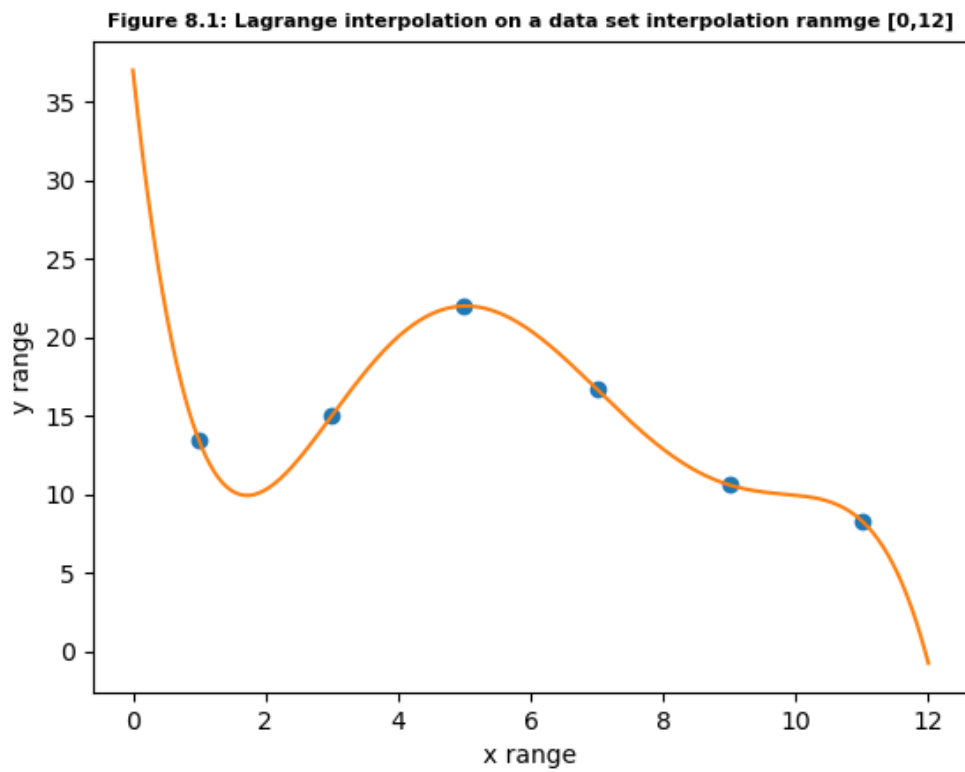


Figure 2: Task 8 Lagrange Interpolation polynomial on interval [0,12]

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.

```

1 #From assignment document
2 x = list ( range (1 ,12 ,2) )
3 y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]
4
5 def Lagrange_interpolation(g,h):
6     '''Returns the Lagrange polynomial using Lagrange interpolation.
7     Arguments g and h both of which must be lists'''
8     assert len(g) == len(h), "Every x coordinate must have a
9         corresponding y coordinate"
10    output_storage = 0
11    for k in range(len(g)):
12        polynomial_storage = 1 ## Defining polynomial storage and
13            resetting after every k
14        for i in range(len(g)):
15            if i!=k:
16                polynomial_storage = polynomial_storage * (a-g[i])/(g[
17                    k]-g[i])
18            output_storage = output_storage + h[k]*polynomial_storage
19    output_storage = sympy.simplify(output_storage) #This line is not
20        exactly necessary but the function become really ugly otherwise
21    return output_storage
22
23 function = sympy.lambdify(a, Lagrange_interpolation(x,y))

```

Listing 10: Definition of the lagrange interpolation

```

1 spline_eq = scipy.interpolate.splrep(x,y)
2 y_for_plotting_spline = scipy.interpolate.splev(x_for_plotting,
3     spline_eq)

```

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

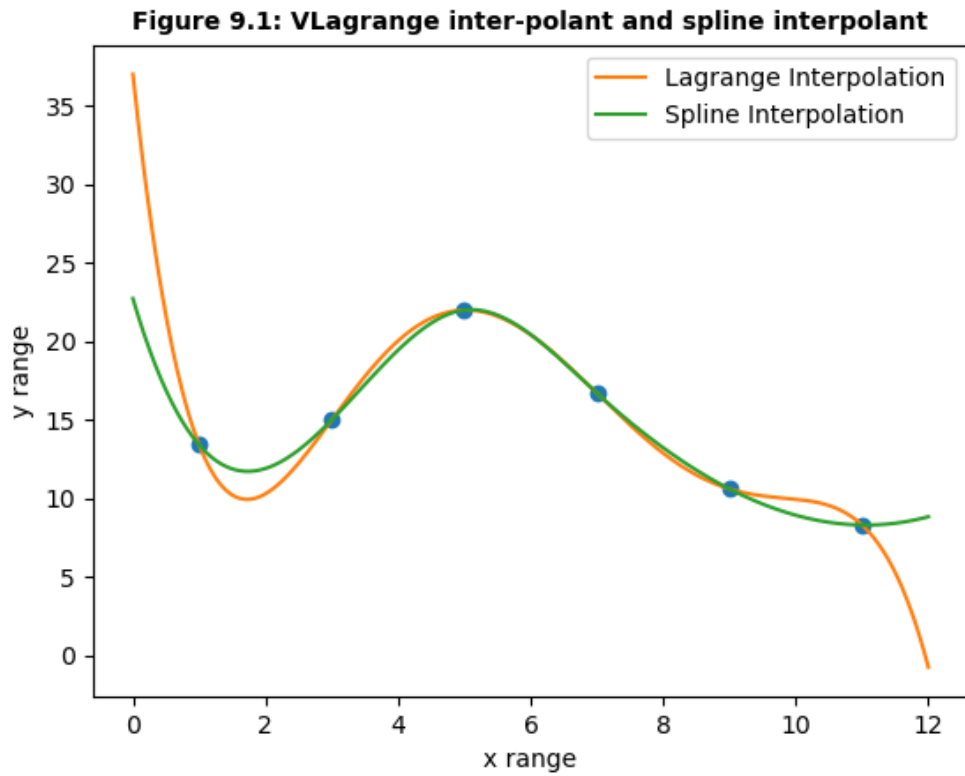


Figure 3: Task 9 graph Lagrange inter-polant and spline interpolant

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.

```

1 print(f'Integral of the spline interpolant: {scipy.integrate.quad(
    lambda x: scipy.interpolate.splev(x, spline_eq), 0, 12)[0]}')
2 print(f'Integral of the Lagrange interpolant: {scipy.integrate.quad(
    function, 0, 12)[0]}')
3 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

```

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

```

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

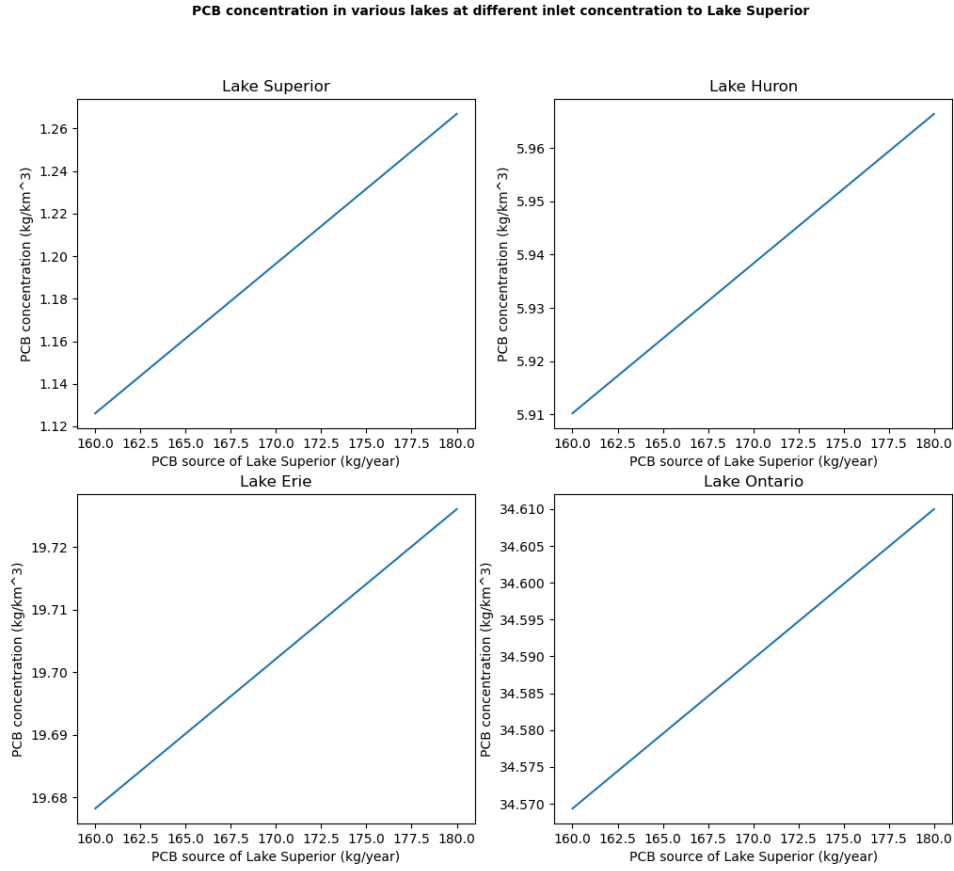


Figure 4: Concentration of PCBs in various lakes at different inlet concentrations of Lake Superior.

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

```

1 # Steady-state assumption for the mass balance (Accumulation is approx
   . 0)
2 # Derivations can be found in the report, here only the final answers
   are given
3
4 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")
5
6 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

```

1 #Libraries
2 import numpy as np
3 import timeit
4 from scipy.linalg import lu
5 import pandas as pd
6
7 # Functions taken from gaussjordan.py (Given in lecture 4)
8 def swap_rows(mat,i1,i2):
9     """Swap two rows in a matrix/vector"""
10    temp = mat[i1,...].copy()
11    mat[i1,...] = mat[i2,...]
12    mat[i2,...] = temp
13
14 def gaussian_eliminate_v2(A,b):
15     """Perform elimination to obtain an upper triangular matrix
16
17     Input:
18     A: Coefficient matrix
19     b: right hand side
20
21     Returns:
22     Aprime, bprime: row echelon form of matrix A and rhs vector b"""
23     A = np.array(A,dtype=np.float64)
24     b = np.array(b,dtype=np.float64)
25
26     assert A.shape[0] == A.shape[1], "Coefficient matrix should be
27         square"
28
29     N = len(b)
30     for col in range(N-1):
31         index = np.argmax(np.abs(A[col:, col])) + col
32         swap_rows(A,col,index)
33         swap_rows(b,col,index)
34         for row in range(col+1,N):
35             d = A[row,col] / A[col,col]
36             A[row,:] = A[row,:] - d * A[col,:]
37             b[row] = b[row] - d * b[col]
38
39     return A,b
40
41 def backsubstitution_v1(U,b):
42     """Back substitutes an upper triangular matrix to find x in Ax=b
43     """
44     x = np.empty_like(b)
45     N = len(b)
46
47     for row in range(N)[::-1]:
48         x[row] = (b[row] - np.sum(U[row,row+1:] * x[row+1:])) / U[row,
49             row]
50
51     return x
52
53 def forwardsubstitution(L,d):

```

```

51     N = len(L)
52     y = np.empty_like(d)
53
54     for row in range(N):
55         y[row] = (d[row] - np.sum(L[row,:row] * y[:row])) / L[row,row]
56
57     return y
58
59 #Taken from it_methods.py
60 def jacobi(A, b, tol=1e-2):
61
62     # Set initial guess
63     x = b + 1e-16
64
65     # Initialize variables
66     x_diff = 1
67     N = A.shape[0]
68     it_jac = 1
69
70     # While not converged or max_it not reached
71     while (x_diff > tol and it_jac < 1000):
72         x_old = x.copy()
73         for i in range(N):
74             s = 0
75             for j in range(N):
76                 if j != i:
77                     # Sum off-diagonal*x_old
78                     s += A[i,j] * x_old[j]
79             # Compute new x value
80             x[i] = (b[i] - s) / A[i,i]
81
82         # Increase number of iterations
83         it_jac += 1
84         x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)
85
86     # Print number of iterations
87     #print(it_jac)
88
89     return x, it_jac
90
91 def jacobi_vec(A, b, tol=1e-2, itmax=1000):
92     # Set initial guess
93     x = b + 1e-16
94
95     # Initialize variables
96     x_diff = 1
97     N = A.shape[0]
98     it_jac = 1
99
100    # While not converged or max_it not reached
101    while (x_diff > tol and it_jac < 1000):
102        x_old = x.copy()
103        for i in range(N):
104            s = 0

```

```

105         j_indices = np.concatenate((np.arange(0,i), np.arange(i+1,
106                                     N)))
107         s += A[i,j_indices] @ x_old[j_indices]
108         # Compute new x value
109         x[i] = (b[i] - s) / A[i,i]
110
111         # Increase number of iterations
112         it_jac += 1
113         x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)
114
115         # Print number of iterations
116         #print(it_jac)
117
118     return x, it_jac
119
120 def gaussseidel(A, b, tol=1e-2):
121     # Set initial guess
122     x = b + 1e-16
123
124     # Initialize variables
125     x_diff = 1
126     N = A.shape[0]
127     it_gaussseidel = 1
128
129     # While not converged or max_it not reached
130     while (x_diff > tol and it_gaussseidel < 1000):
131         x_old = x.copy()
132         for i in range(N):
133             s = 0
134             s2 = 0
135             for j in range(N):
136                 if j < i:
137                     # Sum off-diagonal*x_old
138                     s += A[i,j] * x[j]
139                 if j > i:
140                     # Second summation
141                     s2 += A[i,j] * x_old[j]
142             # Compute new x value
143             x[i] = (b[i] - s - s2) / A[i,i]
144
145             # Increase number of iterations
146             it_gaussseidel += 1
147             x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)
148
149             # Print number of iterations
150             #print(it_gaussseidel)
151
152     return x, it_gaussseidel
153
154 def gaussseidel_vec(A, b, tol=1e-2):
155     # Set initial guess
156     x = b + 1e-16
157
158     # Initialize variables

```

```

158     x_diff = 1
159     N = A.shape[0]
160     it_gaussseidel = 1
161
162     # While not converged or max_it not reached
163     while (x_diff > tol and it_gaussseidel < 1000):
164         x_old = x.copy()
165         for i in range(N):
166             s = 0
167             s2 = 0
168             j_indices_new = np.arange(0,i)
169             j_indices_old = np.arange(i+1, N)
170             s += A[i,j_indices_new] @ x[j_indices_new]
171             s2 += A[i,j_indices_old] @ x_old[j_indices_old]
172             # Compute new x value
173             x[i] = (b[i] - s - s2) / A[i,i]
174
175         # Increase number of iterations
176         it_gaussseidel += 1
177         x_diff = np.linalg.norm(A@x - b)/np.linalg.norm(b)
178
179     # Print number of iterations
180     #print(it_gaussseidel)
181
182     return x, it_gaussseidel
183
184
185 #task 2 matrix
186 #Define the values of the Q's (given in km-3 / yr)
187 Q_SH = 72
188 Q_MH = 38
189 Q_HE = 160
190 Q_EO = 185
191 Q_OO = 215
192
193 #Define values for the different PCB source S_in's (given in kg/yr)
194
195 S_in = 180
196 S_in_M = 810
197 S_in_H = 630
198 S_in_E = 2750
199 S_in_O = 3820
200
201 #Volumes (km3)
202
203 V_S = 12000
204 V_M = 4900
205 V_H = 3500
206 V_E = 480
207 V_O = 1640
208
209
210
211 #Generating our coefficient matrix

```

```

212 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]])
213 #print(M) #Uncomment to see the coefficient matrix
214
215
216 #Solution Vector
217 sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
218 #print(sol_vec) #Uncomment to see the Solution Vector
219
220
221
222 #Using a normal the solver from numpy
223
224 solver_code = lambda: np.linalg.solve(M,sol_vec)
225 Time_taken_solver = timeit.timeit(solver_code, number=1)
226
227
228 #Gaussian Elimination
229 #Using code provided in Lecture 4 (gaussjordan.py)
230
231 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
232
233 #LU decomposition
234
235
236
237 def LU_function_for_timing():
238     '''Function only written for task 2 for timing LU decomposition.
      This function only serves one purpose.'''
239     P, L, U = lu(M)
240     d = P @ sol_vec
241     y= forwardsubstitution(L, d)
242     lu_function = lambda: backsubstitution_v1( U,y) # time.time()
      prints out 0.0 otherwise
243 end_time_lu = timeit.timeit(LU_function_for_timing ,number=1)
244
245 #Jacobi (Function taken from lecture 5)
246 # First the non-vectorized
247
248
249 jacobi_code = lambda: jacobi(M, sol_vec)
250 end_time_Jacobi = timeit.timeit(jacobi_code, number=1)
251
252 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
253
254 #Then the vectorized
255
256 jacobi_sol_vec, jacobi_number_of_iterations_vec = jacobi_vec(M,
      sol_vec)
257

```

```

258 Jacobi_vec_code = lambda: jacobi_vec(M, sol_vec)
259 end_time_Jacobi_vec = timeit.timeit(Jacobi_vec_code, number=1)
260
261 #Gauss-Seidel
262 #First the non-vectorized
263
264
265 gaussseidel_sol, gaussseidel_number_of_iterations = gaussseidel(M,
    sol_vec)
266
267 gaussseidel_function = lambda:gaussseidel(M,sol_vec)
268 end_time_gaussseidel = timeit.timeit(gaussseidel_function, number=1)
269
270
271
272
273 #Then the vectorized
274
275 gaussseidel_sol_vec, gaussseidel_number_of_iterations_vec =
    gaussseidel_vec(M, sol_vec)
276
277 gaussseidel_function_vec = lambda:gaussseidel_vec(M, sol_vec)
278 end_time_gaussseidel_vec = timeit.timeit(gaussseidel_function_vec,
    number=1)
279
280
281
282 #Uncomment to see the output as as text
283 #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")
284
285 #To generate table
286 #Columns
287
288 def accuracy_checker(A,B):
289     equality = False
290     if np.array_equal(A,B):
291         equality = True
292         return equality
293     else:
294         return equality
295
296
297
298
299

```



```

300 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))]]
301
302
303 df = pd.DataFrame(rows, columns = ['Solution method', 'Time taken (s)'
    , 'Number of iterations', 'Accuracy'])
304 df['Time taken (s)'] = df['Time taken (s)'].apply(lambda x: f'{x:.11f}
    ')
305
306 print(df)

```

codes/Task 2.py

```

1 #Libraries
2 import numpy as np
3 import timeit
4 import time
5 from scipy.linalg import lu
6
7 #task 2 matrix
8 #Define the values of the Q's (given in km-3 / yr)
9 Q_SH = 72
10 Q_MH = 38
11 Q_HE = 160
12 Q_EO = 185
13 Q_OO = 215
14
15 #Define values for the different PCB source S_in's (given in kg/yr)
16
17 S_in = 180
18 S_in_M = 810
19 S_in_H = 630
20 S_in_E = 2750
21 S_in_O = 3820
22
23 #Volumes (km3)
24
25 V_S = 12000
26 V_M = 4900
27 V_H = 3500
28 V_E = 480
29 V_O = 1640
30
31 #Defining k1
32 k1 = 0.00584
33
34
35 #Generating our coefficient matrix
36 M = np.array([[Q_SH + k1*V_S, 0, 0, 0, 0], [0, Q_MH + k1*V_M, 0, 0, 0], [-Q_SH
37     , -Q_MH, Q_HE + k1*V_H, 0, 0], [0, 0, -Q_HE, Q_EO + k1*V_E, 0],
38     [0, 0, 0, -Q_EO, Q_OO + k1*V_E]])
39 #print(M) #Uncomment to see the coefficient matrix
40
41 #Solution Vector
42 sol_vec = np.array([S_in, S_in_M, S_in_H, S_in_E, S_in_O])
43
44 print(np.linalg.solve(M, sol_vec))

```

codes/Task 3.py

```

1 #Libraries
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 #task 2 matrix
6 #Define the values of the Q's (given in km-3 / yr)
7 Q_SH = 72
8 Q_MH = 38
9 Q_HE = 160
10 Q_EO = 185
11 Q_OO = 215
12
13 #Define values for the different PCB source S_in's (given in kg/yr)
14
15 S_in = 180
16 S_in_M = 810
17 S_in_H = 630
18 S_in_E = 2750
19 S_in_O = 3820
20
21 #Volumes (km3)
22
23 V_S = 12000
24 V_M = 4900
25 V_H = 3500
26 V_E = 480
27 V_O = 1640
28
29 #Defining k1
30 k1 = 0.00584
31
32
33 #New for task 4, we need to define a new QMO
34 Q_MO_inputs = np.linspace(0,100, 1000)
35
36 #Empty lists to store the values
37 conc_lake_michigan_values = []
38 conc_lake_huron_values = []
39 conc_lake_erie_values = []
40 conc_lake_ontario_values = []
41
42 #Lake Superior is not included because its independent of the lake
  Michigan
43 for Q_MO in Q_MO_inputs: # Calculate the concentraions in the lakes at
  various bypass flowrates
44     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]])
45     sol_vec = np.array([S_in, S_in_M, S_in_H,S_in_E, S_in_O])
46     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

```

```

47     but since it the matrix is so small it doesn't matter too much
48     conc_lake_michigan_values.append(conc_lake_michigan)
49     conc_lake_huron_values.append(conc_lake_huron)
50     conc_lake_erie_values.append(conc_lake_erie)
51     conc_lake_ontario_values.append(conc_lake_ontario)
52
53
54 #Making figures
55 fig = plt.figure(figsize=(15,10))
56 ax1 = plt.subplot(2, 2, 1)
57 ax1.plot(Q_MO_inputs, conc_lake_michigan_values)
58 ax1.set_title("Figure 4.1: Concentration of PCB ([kg/yr]) in lake
59     Michigan against bypass flowrates", size=8, weight='bold')
60 ax1.set_xlabel('Concentration of PCB ([kg/yr])')
61 ax1.set_ylabel('Bypass Flowrate [km**3/yr]')
62
63 ax2 = plt.subplot(2,2,2)
64 ax2.plot(Q_MO_inputs, conc_lake_huron_values)
65 ax2.set_title("Figure 4.2: Concentration of PCB ([kg/yr]) in lake
66     Huron against bypass flowrates", size=8, weight='bold')
67 ax2.set_xlabel('Concentration of PCB ([kg/yr])')
68 ax2.set_ylabel('Bypass Flowrate [km**3/yr]')
69
70 ax3 = plt.subplot(2,2,3)
71 ax3.plot(Q_MO_inputs, conc_lake_erie_values)
72 ax3.set_title("Figure 4.3: Concentration of PCB ([kg/yr]) in lake
73     Michigan against bypass flowrates", size=8, weight='bold')
74 ax3.set_xlabel('Concentration of PCB ([kg/yr])')
75 ax3.set_ylabel('Bypass Flowrate [km**3/yr]')
76
77 ax4 = plt.subplot(2,2,4)
78 ax4.plot(Q_MO_inputs, conc_lake_ontario_values)
79 ax4.set_title("Figure 4.4: Concentration of PCB ([kg/yr]) in lake
80     Ontario against bypass flowrates", size=8, weight='bold')
81 ax4.set_xlabel('Concentration of PCB ([kg/yr])')
82 ax4.set_ylabel('Bypass Flowrate [km**3/yr]')
83
84 plt.show()

```

codes/Task 4.py

```

1 #Libraries
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 #task 2 matrix
6 #Define the values of the Q's (given in km-3 / yr)
7 Q_SH = 72
8 Q_MH = 38
9 Q_HE = 160
10 Q_EO = 185
11 Q_OO = 215
12
13 #Define values for the different PCB source S_in's (given in kg/yr)
14
15 S_in = 180
16 S_in_M = 810
17 S_in_H = 630
18 S_in_E = 2750
19 S_in_O = 3820
20
21 #Volumes (km3)
22
23 V_S = 12000
24 V_M = 4900
25 V_H = 3500
26 V_E = 480
27 V_O = 1640
28
29 #Defining k1
30 k1 = 0.00584
31
32 #Defining k2
33 k2 = 2 * 10**7
34
35 #Volume of tank is given in m3 but everything else is in km3. Thus,
    the volume is given in km3
36 V_tank = 1*10**9
37
38
39 # Not given in the question so taken as 100
40 Q_MO = 20
41
42
43
44 ## For PFR Graph
45 # Making the matrix
46
47 PFR_matrix = np.zeros((1005,1005))
48 new_sol_vec = np.zeros(1005)
49 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]])
50 PFR_matrix[:5, :5] = M # Update the new matrix

```

```

51 PFR_matrix[4, -1] = -Q_M0 # Inflow from the last PFR
52 sol_vec = np.array([S_in, S_in_M, S_in_H, S_in_E, S_in_O])
53 new_sol_vec[:5] = sol_vec
54 PFR_matrix[5, 1] = Q_M0 # First PFR is dependent on the concentration
    of lake Michigan
55 PFR_matrix[5, 5] = -(Q_M0 + V_tank*k2)
56
57 for i in range(999):
58     PFR_matrix[6+i, 5+i] = Q_M0
59     PFR_matrix[6+i, 6+i] = -(Q_M0 + V_tank*k2)
60
61 solution = np.linalg.solve(PFR_matrix, new_sol_vec)
62 print(solution[0:5]) # Prints the concentraion of the PCBs in the
    lakes
63
64
65 x = np.linspace(1, 1000, 1000)
66 ax1 = plt.subplot(1,1,1)
67 ax1.plot(x, np.linalg.solve(PFR_matrix, new_sol_vec)[5:])
68 ax1.set_title("Figure 5.1: Concentration profile of PCB in the PFR",
    size=11, weight='bold')
69 ax1.set_xlabel('PFR Tank Number')
70 ax1.set_ylabel('Concentration of PCB ([kg/yr]')
71
72 plt.show()

```

codes/Task 5.py

```

1 import numpy as np
2 import timeit
3 import pandas as pd
4
5 def f(t):
6     return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t
7         /2.7) + 9.6
8
9 # t is given in months so we are looking for the integral from 0 to 12
10 # (One full year)
11 # Since we want 20 intervals, every step will be 12/20
12
13 ## The functions here were defined ourselves and not taken from
14 # lecture 8
15 def riemann_sum_lower(func, a, b, intervals=20):
16     '''Calculates the lower Riemann sum of a function.
17     Takes 3 arguments, first the function, then the lower integration
18     bound, and finally the upper integration bound.
19     Optional 4th argument to specify number of intervals (default set
20     to 20)'''
21     bin_size = (b-a)/intervals
22     sum = 0
23     for i in range(intervals):
24         sum += func(a+i*bin_size)*bin_size
25     return sum
26
27 def riemann_sum_upper(func, a, b, intervals=20):
28     '''Calculates the upper Riemann sum of a function.
29     Takes 3 arguments, first the function, then the lower integration
30     bound, and finally the upper integration bound.
31     Optional 4th argument to specify number of intervals (default set
32     to 20)'''
33     bin_size = (b-a)/intervals
34     sum = 0
35     for i in range(intervals):
36         sum += func(a+(i+1)*bin_size)*bin_size
37     return sum
38
39 def riemann_sum_middle(func, a, b, intervals=20):
40     '''Calculates the middle Riemann sum of a function.
41     Takes 3 arguments, first the function, then the lower integration
42     bound, and finally the upper integration bound.
43     Optional 4th argument to specify number of intervals (default set
44     to 20)'''
45     bin_size = (b-a)/intervals
46     sum = 0
47     for i in range(intervals):
48         sum += func((a+i*bin_size + a+(i+1)*bin_size)/2)*bin_size
49     return sum
50
51 def trapezoid_rule(func, a, b, intervals=20):

```

```

45     '''Calculates the area of a function using the trapezoid rule.
46     Takes 3 arguments, first the function, then the lower integration
47     bound, and finally the upper integration bound.
48     Optional 4th argument to specify number of intervals (default set
49     to 20)'''
50     bin_size = (b-a)/intervals
51     sum = 0
52     for i in range(intervals):
53         sum += (bin_size * (func(a+i*bin_size) + func(a+(i+1)*bin_size
54         )))/2
55     return sum
56
57 def simpson_rule(func, a, b, intervals=20):
58     '''Calculates the area of a function using the Simpson rule.
59     Takes 3 arguments, first the function, then the lower integration
60     bound, and finally the upper integration bound.
61     Optional 4th argument to specify number of intervals (default set
62     to 20)'''
63     bin_size = (b-a)/intervals
64     sum = 0
65     for i in range(intervals):
66         a_loop = a+i*bin_size
67         b_loop = a+(i+1)*bin_size
68         sum += (func(a_loop)+4*func((a_loop+b_loop)/(2))+func(b_loop))
69         * ((b_loop-a_loop)/(6))
70     return sum
71
72 #Calculating the times and outputs of functions
73 #Code remains constant and timeit is being used because time.time gave
74 0.0 sometimes
75
76 riemann_area_lower = riemann_sum_lower(f,0,12)
77 riemann_code_lower = lambda:riemann_sum_lower(f,0,12)
78 end_time_riemann_lower = timeit.timeit(riemann_code_lower, number=1)
79
80 riemann_area_upper = riemann_sum_upper(f,0,12)
81 riemann_code_upper = lambda:riemann_sum_upper(f,0,12)
82 end_time_riemann_upper = timeit.timeit(riemann_code_upper, number=1)
83
84 riemann_area_middle = riemann_sum_middle(f,0,12)
85 riemann_code_middle = lambda:riemann_sum_middle(f,0,12)
86 end_time_riemann_middle = timeit.timeit(riemann_code_middle, number=1)
87
88 trapezoid_area = trapezoid_rule(f,0,12)
89 trapezoid_code = lambda:trapezoid_rule(f,0,12)
90 end_time_trapezoid = timeit.timeit(trapezoid_code, number=1)
91
92 simpson_area = simpson_rule(f,0,12)
93 simpson_code = lambda:simpson_rule(f,0,12)
94 end_time_simpson = timeit.timeit(simpson_code, number=1)

```



```

92
93 # "Actual" value obtained from graphical display calculator
94
95 actual_result = 160.3582902978
96
97 # Needed to calculate the percentage error
98 def percentage_error(a, b):
99     '''Returns the percentage error of the function
100     First argument is the observed value and second argument is the
        expected value'''
101     return np.abs((a-b)/(b))*100
102
103
104 # Generating rows of the output table
105 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)]]
106
107
108 df = pd.DataFrame(rows, columns = ['Solution method', 'Total discharge
        into lake superior (kg)', 'Time taken (s)', 'Percentage error from
        expected value'])
109
110
111 print(df)

```

codes/Task 6.py

```

1 #Libraries
2 import numpy as np
3 import scipy
4 from scipy import optimize
5 import pandas as pd
6
7 #Same function as in task 6
8 def f(t):
9     return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t
10         /2.7) + 9.6
11
12 #We need to find its maximum
13 def find_max(f,a,b):
14     '''Finds the maximum of a function on a given domain.
15     First argument is your function, second is the lowerbound, and
16     third is the upperbound.'''
17     minimize_result = scipy.optimize.minimize_scalar(lambda x: -f((b-a
18         )*x), bounds=[0,1], method='bounded') #Finds maximum of a
19     function. According to information found online it has to be
20     bound from 0 to 1 thus the function is shrunk.
21     return -minimize_result.fun # Returns the maximum y value of a
22     function.
23
24 #From gdc, y max ~= 18.394515656
25 #print(find_max(f,0,12)) #Uncomment in case you want to verify that
26 #the function works
27
28 def monte_carlo(func, a, b, N=1000):
29     '''Returns the approximate area under a curve using the Monte
30     Carlo method.
31     First argument is the function you want to integrate.
32     Second argument is the lower bound and third is the upperbound.
33     Optional third argument to specify how many random points to
34     choose with default set to 1000.'''
35     approx_area_counter = 0
36     y_max = find_max(func, a, b)
37     for i in range(N):
38         x_rand = np.random.uniform(a,b) ## random number on the given
39         domain
40         y_rand = np.random.uniform(0,y_max) ## Area of the shape lies
41         between y = 0 and y = ymax
42         if y_rand < f(x_rand):
43             approx_area_counter += 1
44     approx_area = (y_max*(b-a)*approx_area_counter) / N
45     return approx_area
46
47 #Taken from task 6, needed for the table
48 def percentage_error(a, b):
49     '''Returns the percentage error of the function
50     First argument is the observed value and second argument is the
51     expected value'''

```

```

42     return np.abs((a-b)/(b))*100
43
44
45
46
47
48 #Same as task 6, taken from graphical display calculator.
49 actual_result = 160.3582902978
50
51
52 rows = []
53 big_numbers_for_table = [1000, 2000, 5000, 10000, 20000, 50000,
54     100000]
55 for i in big_numbers_for_table:
56     area_loop = monte_carlo(f,0,12,i)
57     error_loop = percentage_error(area_loop, actual_result)
58     rows.append([area_loop, i, error_loop])
59
60 df = pd.DataFrame(rows, columns = ['Total discharge into lake superior
61     (kg)', 'Number of Iterations', 'Percentage error from expected
62     value'])
63
64 print(df)

```

codes/Task 7.py

```

1 # The sole purpose of this file is to generate the table to compare
  methods from task 6 to the methods from task 7.
2 # As this is a continuation of task 7, the comments have been removed.
  They can be found in task 7.py
3 # Libraries Required
4 import numpy as np
5 import pandas as pd
6 import numpy as np
7 import scipy
8 from scipy import optimize
9 import pandas as pd
10
11
12 #Functions
13 # We know that we could use for example "import Task 6" to get the
  functions but we wanted to make sure that
14 # the code runs properly on all devices.
15 # Thus we simply copy-pasted the funtions into this file to make sure
  that the correctors can run the code perfectly
16 def f(t):
17     return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t
    /2.7) + 9.6
18
19 def riemann_sum_lower(func, a, b, intervals=20):
20     '''Calculates the lower Riemann sum of a function.
21     Takes 3 arguments, first the function, then the lower integration
22     bound, and finally the upper integration bound.
23     Optional 4th argument to specify number of intervals (default set
24     to 20)'''
25     bin_size = (b-a)/intervals
26     sum = 0
27     for i in range(intervals):
28         sum += func(a+i*bin_size)*bin_size
29     return sum
30
31 def riemann_sum_upper(func, a, b, intervals=20):
32     '''Calculates the upper Riemann sum of a function.
33     Takes 3 arguments, first the function, then the lower integration
34     bound, and finally the upper integration bound.
35     Optional 4th argument to specify number of intervals (default set
36     to 20)'''
37     bin_size = (b-a)/intervals
38     sum = 0
39     for i in range(intervals):
40         sum += func(a+(i+1)*bin_size)*bin_size
41     return sum
42
43 def riemann_sum_middle(func, a, b, intervals=20):
44     '''Calculates the middle Riemann sum of a function.
45     Takes 3 arguments, first the function, then the lower integration
46     bound, and finally the upper integration bound.
47     Optional 4th argument to specify number of intervals (default set
48     to 20)'''

```

```

43     bin_size = (b-a)/intervals
44     sum = 0
45     for i in range(intervals):
46         sum += func((a+i*bin_size + a+(i+1)*bin_size)/2)*bin_size
47     return sum
48
49 def trapezoid_rule(func, a, b, intervals=20):
50     '''Calculates the area of a function using the trapezoid rule.
51     Takes 3 arguments, first the function, then the lower integration
52     bound, and finally the upper integration bound.
53     Optional 4th argument to specify number of intervals (default set
54     to 20)'''
55     bin_size = (b-a)/intervals
56     sum = 0
57     for i in range(intervals):
58         sum += (bin_size * (func(a+i*bin_size) + func(a+(i+1)*bin_size
59         )))/2
60     return sum
61
62 def simpson_rule(func, a, b, intervals=20):
63     '''Calculates the area of a function using the Simpson rule.
64     Takes 3 arguments, first the function, then the lower integration
65     bound, and finally the upper integration bound.
66     Optional 4th argument to specify number of intervals (default set
67     to 20)'''
68     bin_size = (b-a)/intervals
69     sum = 0
70     for i in range(intervals):
71         a_loop = a+i*bin_size
72         b_loop = a+(i+1)*bin_size
73         sum += (func(a_loop)+4*func((a_loop+b_loop)/(2))+func(b_loop))
74         * ((b_loop-a_loop)/(6))
75     return sum
76
77 def find_max(f,a,b):
78     '''Finds the maximum of a function on a given domain.
79     First argument is your function, second is the lowerbound, and
80     third is the upperbound.'''
81     minimize_result = scipy.optimize.minimize_scalar(lambda x: -f((b-a
82     )*x), bounds=[0,1], method='bounded') #Finds maximum of a
83     function. According to information found online it has to be
84     bound from 0 to 1 thus the function is shrunk.
85     return -minimize_result.fun # Returns the maximum y value of a
86     function.
87
88 def monte_carlo(func, a, b, N=1000):
89     '''Returns the approximate area under a curve using the Monte
90     Carlo method.
91     First argument is the function you want to integrate.
92     Second argument is the lower bound and third is the upperbound.
93     Optional third argument to specify how many random points to
94     choose with default set to 1000.'''
95     approx_area_counter = 0
96     y_max = find_max(func, a, b)

```

```

84     for i in range(N):
85         x_rand = np.random.uniform(a,b) ## random number on the given
            domain
86         y_rand = np.random.uniform(0,y_max) ## Area of the shape lies
            between y = 0 and y = ymax
87         if y_rand < f(x_rand):
88             approx_area_counter += 1
89         approx_area = (y_max*(b-a)*approx_area_counter) / N
90     return approx_area
91
92 def percentage_error(a, b):
93     '''Returns the percentage error of the function
94     First argument is the observed value and second argument is the
        expected value'''
95     return np.abs((a-b)/(b))*100
96
97 riemann_area_lower = riemann_sum_lower(f,0,12)
98
99 riemann_area_upper = riemann_sum_upper(f,0,12)
100
101 riemann_area_middle = riemann_sum_middle(f,0,12)
102
103 trapezoid_area = trapezoid_rule(f,0,12)
104
105 simpson_area = simpson_rule(f,0,12)
106
107 actual_result = 160.3582902978
108
109
110 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)]]
111
112
113 big_numbers_for_table = [1000, 2000, 5000, 10000, 20000, 50000,
    100000]
114
115 for i in big_numbers_for_table:
116     area_loop = monte_carlo(f,0,12,i)
117     error_loop = percentage_error(area_loop, actual_result)
118     rows.append(['Monte-Carlo with N='+str(i),area_loop, error_loop])
119
120
121
122
123 df = pd.DataFrame(rows, columns = ['Method', 'Calculated Area', '
    Percentage error from expected value'])
124
125 print(df)

```

codes/Task 7 continued.py

```

1 #Libraries
2 import numpy as np
3 import sympy
4 import matplotlib.pyplot as plt
5 #Defining symbols
6 a = sympy.Symbol('x', real=True) #x is already used for the list
7
8
9 #From assignment document
10 x = list ( range (1 ,12 ,2) )
11 y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]
12
13 def Lagrange_interpolation(g,h):
14     '''Returns the Lagrange polynomial using Lagrange interpolation.
15     Arguments g and h both of which must be lists'''
16     assert len(g) == len(h), "Every x coordinate must have a
17         corresponding y coordinate"
18     output_storage = 0
19     for k in range(len(g)):
20         polynomial_storage = 1 ## Defining polynomial storage and
21             resetting after every k
22         for i in range(len(g)):
23             if i!=k:
24                 polynomial_storage = polynomial_storage * (a-g[i])/(g[
25                     k]-g[i])
26                 output_storage = output_storage + h[k]*polynomial_storage
27             output_storage = sympy.simplify(output_storage) #This line is not
28                 exactly necessary but the function become really ugly otherwise
29         return output_storage
30
31 function = sympy.lambdify(a, Lagrange_interpolation(x,y))
32
33 # Plotting
34 x_for_plotting = np.linspace(0, 12, 1000)
35 y_for_plotting = function(x_for_plotting)
36
37 fig = plt.subplot()
38 ax1 = fig.plot(x, y, marker='o', linestyle='None')
39 ax2 = fig.plot(x_for_plotting, y_for_plotting)
40 fig.set_title("Figure 8.1: Concentration of PCB ([kg/yr]) in lake
41     Michigan against bypass flowrates", size=8, weight='bold')
42 fig.set_xlabel('Concentration of PCB ([kg/yr])')
43 fig.set_ylabel('Bypass Flowrate [km**3/yr]')
44
45 plt.show()

```

codes/Task 8.py


```

1 #Libraries
2 import numpy as np
3 import sympy
4 from scipy import interpolate
5 import matplotlib.pyplot as plt
6 import scipy
7
8 #Prerequisites from the previous task, needed to plot the lagrange
  interpolant
9 a = sympy.Symbol('x', real=True) #x is already used for the list
10 p = sympy.Symbol('y', real=True) #y already used for list as well
11 #From assignment document
12 x = list ( range (1 ,12 ,2) )
13 y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]
14
15 def Lagrange_interpolation(g,h):
16     '''Returns the Lagrange polynomial using Lagrange interpolation.
17     Arguments g and h both of which must be lists'''
18     assert len(g) == len(h), "Every x coordinate must have a
19         corresponding y coordinate"
20     output_storage = 0
21     for k in range(len(g)):
22         polynomial_storage = 1 ## Defining polynomial storage and
23             resetting after every k
24         for i in range(len(g)):
25             if i!=k:
26                 polynomial_storage = polynomial_storage * (a-g[i])/(g[
27                     k]-g[i])
28                 output_storage = output_storage + h[k]*polynomial_storage
29             output_storage = sympy.simplify(output_storage) #This line is not
30                 exactly necessary but the function become really ugly otherwise
31         return output_storage
32
33 #Needed to generate the plots of the lagrange interpolant
34 function = sympy.lambdify(a, Lagrange_interpolation(x,y))
35 x_for_plotting = np.linspace(0, 12, 1000)
36 y_for_plotting_lagrange = function(x_for_plotting)
37
38 #The actual new code starts here
39 spline_eq = scipy.interpolate.splrep(x,y)
40 y_for_plotting_spline = scipy.interpolate.splev(x_for_plotting,
41     spline_eq)
42 #According to the discussions on canvas, we were not asked to make the
43     function ourselves
44
45 #Plotting
46 fig = plt.subplot()
47 ax1 = fig.plot(x, y, marker='o', linestyle='None')
48 ax2 = fig.plot(x_for_plotting, y_for_plotting_lagrange, label='
49     Lagrange Interpolation')

```

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

codes/Task 9.py

```

1 #Libraries
2 import numpy as np
3 import sympy
4 import matplotlib.pyplot as plt
5 import scipy
6 import scipy.interpolate ## I know it seems redundant to import scipy
   and scipy.interpolate separately but otherwise the code does not run
   on the TU/e laptop
7 import scipy.integrate ## Same as comment above
8 import pandas as pd
9
10 #Prerequisites from the previous task
11 a = sympy.Symbol('x', real=True) #x is already used for the list
12 ## No need for a y
13 #From assignment document
14 x = list ( range (1 ,12 ,2) )
15 y = [13.40 ,15.00 ,22.0 ,16.70 ,10.60 ,8.30]
16
17 def Lagrange_interpolation(g,h):
18     '''Returns the Lagrange polynomial using Lagrange interpolation.
19     Arguments g and h both of which must be lists'''
20     assert len(g) == len(h), "Every x coordinate must have a
       corresponding y coordinate"
21     output_storage = 0
22     for k in range(len(g)):
23         polynomial_storage = 1 ## Defining polynomial storage and
           resetting after every k
24         for i in range(len(g)):
25             if i!=k:
26                 polynomial_storage = polynomial_storage * (a-g[i])/(g[
                   k]-g[i])
27             output_storage = output_storage + h[k]*polynomial_storage
28         output_storage = sympy.simplify(output_storage) #This line is not
           exactly necessary but the function become really ugly otherwise
29     return output_storage
30 #Needed to generate the plots of the lagrange interpolant
31 function = sympy.lambdify(a, Lagrange_interpolation(x,y))
32 x_for_plotting = np.linspace(0, 12, 1000)
33 y_for_plotting_lagrange = function(x_for_plotting)
34 spline_eq = scipy.interpolate.splrep(x,y)
35 y_for_plotting_spline = scipy.interpolate.splev(x_for_plotting,
   spline_eq)
36 def f(t):
37     return -0.1 * t**3 + 0.58 * t**2 * np.cosh((-1)/(t+1)) + np.exp(t
   /2.7) + 9.6
38
39
40 ## Below, we can see the integral of the interpolated function and
   that of q
41 print(f'Integral of the spline interpolant: {scipy.integrate.quad(
   lambda x: scipy.interpolate.splev(x, spline_eq), 0, 12)[0]}')
42 print(f'Integral of the Lagrange interpolant: {scipy.integrate.quad(
   function, 0, 12)[0]}')

```

```

43 print(f'Integral of the function from task 6: {scipy.integrate.quad(
    lambda x: f(x), 0, 12)[0]}')
44
45 ## 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.
46 ## What we can do is assume that our real source for lake superior
    lies somewhere in between 160 and 180
47 ## Finally we can run a simulation at varrying inlet concentrations of
    lake superior and see the results,
48 ## on the lakes that are dependent on it (All except for Lake Michigan
    )
49
50 ## Below is code from task 5 required to perform the calculations
51
52 #task 2 matrix
53 #Define the values of the Q's (given in km-3 / yr)
54 Q_SH = 72
55 Q_MH = 38
56 Q_HE = 160
57 Q_EO = 185
58 Q_OO = 215
59
60 #Define values for the different PCB source S_in's (given in kg/yr)
61
62 # Now S_in can take different values
63 S_in_M = 810
64 S_in_H = 630
65 S_in_E = 2750
66 S_in_O = 3820
67
68 #Volumes (km3)
69
70 V_S = 12000
71 V_M = 4900
72 V_H = 3500
73 V_E = 480
74 V_O = 1640
75
76 #Defining k1
77 k1 = 0.00584
78
79 #Defining k2
80 k2 = 2 * 10**7
81
82 #Volume of tank is given in m3 but everything else is in km3. Thus,
    the volume is given in km3
83 V_tank = 1*10**9
84
85
86 # Not given in the question so taken as 100
87 Q_MO = 20
88
89

```

```

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_M0,0,0,0],
98             [-Q_SH, -Q_MH, Q_HE + k1*V_H, 0, 0], [0,0,-Q_HE, Q_E0 + k1*V_E,
99             0], [0, 0,0,-Q_E0,Q_O0 + k1*V_E]])
100 PFR_matrix[:5, :5] = M # Update the new matrix
101 PFR_matrix[4, -1] = -Q_M0 # Inflow from the last PFR
102 PFR_matrix[5, 1] = Q_M0 # First PFR is dependent on the concentration
103 of lake Michigan
104 PFR_matrix[5, 5] = -(Q_M0 + V_tank*k2)
105
106
107 for i in range(999):
108     PFR_matrix[6+i, 5+i] = Q_M0
109     PFR_matrix[6+i, 6+i] = -(Q_M0 + V_tank*k2)
110
111
112 x_task_10 = []
113 Superior_list = []
114 Huron_list = []
115 Erie_list = []
116 Ontario_list = []
117
118 for j in range(81): #
119     x_task_10.append(160 + j/4) # Will be needed later to generate our
120     plots
121     sol_vec = np.array([160 + j/4, S_in_M, S_in_H,S_in_E, S_in_O])
122     new_sol_vec[:5] = sol_vec
123     sup, mich, hur, er, ont = np.linalg.solve(PFR_matrix, new_sol_vec)
124     [0:5] # solve the matrix and store the solutions
125     Superior_list.append(sup) ## Appends the solutions to the lists
126     Huron_list.append(hur) ## Needed for plotting
127     Erie_list.append(er)
128     Ontario_list.append(ont)
129
130
131
132
133 fig = plt.figure(figsize=(12,10))
134 fig.suptitle('PCB concentration in various lakes at different inlet
135 concentration to Lake Superior', size=10, weight='bold')
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895

```

```

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
163         the x list and second arument is the y list'''
164     return (w[-1]-w[0])/(q[-1]-q[0])
165
166 rows = [['Lake Superior', derivative(x_task_10, Superior_list)], ['
167     Lake Huron', derivative(x_task_10, Huron_list)], ['Lake Erie',
168     derivative(x_task_10, Erie_list)], ['Lake Ontario', derivative(
169     x_task_10, Ontario_list)]]
170
171 df = pd.DataFrame(rows, columns = ['Lake', 'Rate of change'])
172
173 print(df)

```

codes/Task 10.py