MACHINE PROBLEM NO. 1

ES 204 1st Sem 2023-2024

As a student of the University of the Philippines, I pledge to act ethically and uphold the values of honor and excellence.

I understand that suspected misconduct on this Assignment will be reported to the appropriate office and if established, will result in disciplinary action in accordance with University rules, policies and procedures. I may work with others only to the extent allowed by the Instructor.

Name: Jeryl Salas
Student Number: 2023111128

GENERAL INSTRUCTIONS: Solve all the problems using appropriate numerical and programming techniques, independently and completely. Cite all references or any assistance that you received during the development of your solution. Submit a brief but comprehensive EXECUTIVE SUMMARY of your solution to the problems. In this particular machine problem, for size $n \geq 100$, include only the middle 20 values of your solution. Add the complete solution(s) and source codes in the APPENDIX.

For PROBLEMS 1-3, use the given system **A·x=b**.

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} h^2 \\ h^2 \\ h^2 \\ \vdots \\ h^2 \\ h^2 \end{bmatrix}$$

where n is a positive integer and $h = \frac{1}{n+1}$.

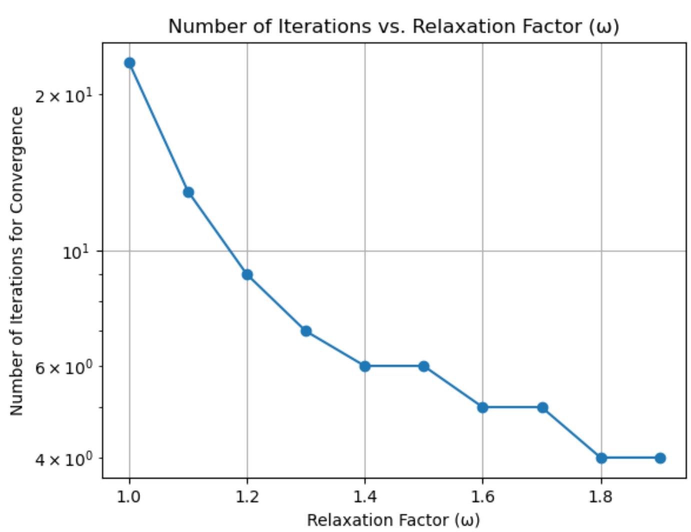
PROBLEM 1 [20 points] (a) Solve for x_i $(i = 1,2,\ldots,n)$ using SOR with $\omega = 1.0, 1.1, 1.2, 1.3, \ldots, 1.9$. (b) Plot the number of iterations for convergence vs. ω. Use the L_∞ norm for convergence and $n = 30$ with tolerance = 10^{-3}.

1.1 Method(s) of solution

$$\overline{x}_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$
$$x_i^{(k+1)} = \omega\overline{x}_i^{(k+1)} + (1-\omega)x_i^{(k)}$$

- Define matrix A and vector b
- Define the relaxation factors that will be used which is from 1.0 to 1.9
- For each ω in the list omegas, find vector x and get the number of iterations using the SOR method
- For each iteration, k, where k is less than or equal to the maximum number of iterations, compute $\bar{x}_i^{(k+1)}$ using the Gauss-Seidel Method
- From the $\bar{x}_i^{(k+1)}$, compute $x_i^{(k+1)}$ using the current relaxation factor, ω.
- Using L∞ norm, we check for convergence. We have set tolerance to 10^{-3}.
- We store the number of iterations it took to converge as well as the ω used in a list.
- From that list we plot a graph to show a line graph using matplotlib

1.2 Results



ω	Iterations
1.0	23
1.1	13
1.2	9
1.3	7
1.4	6
1.5	6
1.6	5
1.7	5
1.8	4
1.9	4

```
ω: 1.7
iterations: 5
Solution:
[-0.00959338 -0.0182667  -0.02607792 -0.03308279 -0.03933428 -0.04488212
 -0.04977233 -0.05404679 -0.05774284 -0.06089293 -0.06352435 -0.06565891
 -0.06731279 -0.06849636 -0.0692141  -0.06946452 -0.06924029 -0.06852823
 -0.0673095  -0.06555986 -0.06324988 -0.06034533 -0.05680756 -0.05259395
 -0.04765836 -0.04195171 -0.03542254 -0.02801754 -0.01968219 -0.01036139]
Omega 8:

ω: 1.7999999999999998
iterations: 4
Solution:
[-0.00991908 -0.01891191 -0.02703348 -0.03433664 -0.04087159 -0.0466854
 -0.05182163 -0.05631985 -0.0602153  -0.06353858 -0.06631534 -0.06856604
 -0.07030576 -0.0715441  -0.07228505 -0.07252701 -0.07226279 -0.07147975
 -0.07015992 -0.06828023 -0.06581278 -0.06272515 -0.05898082 -0.05453951
 -0.04935772 -0.04338918 -0.03658537 -0.02889611 -0.02027006 -0.01065532]
Omega 9:

ω: 1.9
iterations: 4
Solution:
[-0.01024457 -0.01955667 -0.02798826 -0.03558927 -0.0424071  -0.04848618
 -0.05386757 -0.05858855 -0.06268228 -0.06617751 -0.06909827 -0.0714637
 -0.07328786 -0.07457959 -0.07534247 -0.0755748  -0.07526962 -0.07441488
 -0.07299348 -0.07098359 -0.06835885 -0.06508867 -0.06113862 -0.05647081
 -0.05104431 -0.04481562 -0.03773916 -0.02976778 -0.02085328 -0.01094693]

omegas: [1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9]
iterations: [23, 13, 9, 7, 6, 6, 5, 5, 4, 4]
```

1.3 Problems encountered

- There aren't much issues in terms of running the algorithm
- It was challenging to create the algorithm itself since it has to iterate over the relaxation factors and the number of iterations had to be stored so I had to use a list.

1.4 References

- ES_204_L2_Linear_Equations.pdf

PROBLEM 2 [20 points]: Solve **A·x=b** using the Thomas algorithm for tridiagonal matrices with $n = 101$.

2.1 Method(s) of solution

- Define matrix A and vectors x and b

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 & 0 \\ 0 & 0 & 0 & a_5 & b_5 & c_5 \\ 0 & 0 & 0 & 0 & a_6 & b_6 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{pmatrix}$$

- Decompose matrix A into three vectors,
  - Vector a consists of lower diagonal elements
  - Vector b consists of principal diagonal elements
  - Vector c consists of upper diagonal elements
- We first do the forward elimination. For each row, i, in the system, we get the factor which is Vector a[i, i-1] / Vector b[I, i-1]
- Update principal diagonal element, b[i] by subtracting it to Vector c[i-1, i-1]
- Update the elements in vector b by subtracting it to factor times the element above it, b[i-1].

- We do a backward substitution from the last row of vector x using the right-hand side vector b values and the result of Matrix A after forward elimination.

2.2 Results (or partial results)

x vector:

```
[-4.80584391e-05 -7.20876586e-05 -8.41022684e-05 -9.01095732e-05
 -9.31132257e-05 -9.46150519e-05 -9.53659650e-05 -9.57414216e-05
 -9.59291498e-05 -9.60230140e-05 -9.60699461e-05 -9.60934121e-05
 -9.61051451e-05 -9.61110116e-05 -9.61139449e-05 -9.61154115e-05
 -9.61161448e-05 -9.61165115e-05 -9.61166948e-05 -9.61167865e-05
 -9.61168323e-05 -9.61168552e-05 -9.61168667e-05 -9.61168724e-05
 -9.61168753e-05 -9.61168767e-05 -9.61168774e-05 -9.61168778e-05
 -9.61168779e-05 -9.61168780e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05 -9.61168781e-05
 -9.61168781e-05]
```

2.3 Problems encountered

- The basis for my Thomas Algorithm comes from Lee, W. T. (n.d.) which does the same process but uses different symbols. It took me a while to figure out how to do it.

2.4 References

- Lee, W. T. (n.d.). Tridiagonal Matrices: Thomas Algorithm. MS6021, Scientific Computation, University of Limerick. Retrieved from http://www.industrial-maths.com/ms6021_thomas.pdf?fbclid=IwAR1kwmECuCPQGnGk5W378KKCnu_XWEX5LGTcC70hdvX1cELcfJiFJqkgiqE.

PROBLEM 3 [30 points]: Solve **A·x=b** using the biconjugate gradient method with $n = 300$.

3.1 Method(s) of solution

1. Choose initial guess x_0 , two other vectors x_0^* and b^* and a preconditioner M
2. $r_0 \leftarrow b - A\,x_0$
3. $r_0^* \leftarrow b^* - x_0^*\,A^*$
4. $p_0 \leftarrow M^{-1} r_0$
5. $p_0^* \leftarrow r_0^* M^{-1}$
6. for $k = 0, 1, \ldots$ do
   1. $\alpha_k \leftarrow \dfrac{r_k^* M^{-1} r_k}{p_k^* A p_k}$
   2. $x_{k+1} \leftarrow x_k + \alpha_k \cdot p_k$
   3. $x_{k+1}^* \leftarrow x_k^* + \overline{\alpha_k} \cdot p_k^*$
   4. $r_{k+1} \leftarrow r_k - \alpha_k \cdot A p_k$
   5. $r_{k+1}^* \leftarrow r_k^* - \overline{\alpha_k} \cdot p_k^*\,A^*$
   6. $\beta_k \leftarrow \dfrac{r_{k+1}^* M^{-1} r_{k+1}}{r_k^* M^{-1} r_k}$
   7. $p_{k+1} \leftarrow M^{-1} r_{k+1} + \beta_k \cdot p_k$
   8. $p_{k+1}^* \leftarrow r_{k+1}^* M^{-1} + \overline{\beta_k} \cdot p_k^*$

- Define matrix A and vectors x and b
- Initialize vector x0 as initial guess. I used numpy.random to generate random values
- Initialized p0, r0 as well as their complex conjugates r*0, p*0, b*
- For every iteration k, we compute for α_k and β_k as well as x_{k+1}, r_{k+1}, and p_{k+1} along with their complex conjugates, x_{k+1}^*, r_{k+1}^*, and p_{k+1}^*.
- The iteration ends once the L2 norm of r_{k+1}, r_{k+1}^* are lower than the tolerance limit (tolerance=$1e^{-6}$) or if the iteration reaches the max iteration (max_iter = 1000)

3.2 Results (or partial results)

```
MemoryError: Unable to allocate 703. KiB for an array with shape (300, 300
) and data type float64
```

3.3 Problems encountered

- Unlike the other problems in this problem set, this problem gave me computation problems
- The algorithm cannot converge within the max iteration
- Preconditioning might not be implemented correctly.

3.4 References

- Biconjugate gradient method. (n.d.). In Wikipedia. Retrieved April, 2023, from https://en.wikipedia.org/wiki/Biconjugate_gradient_method
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical Recipes: The Art of Scientific Computing (3rd ed.). Cambridge University Press. ISBN: 978-0-521-88068-8.

PROBLEM 4 [30 points]: Solve $10^x + x - 4 = 0$ using the fixed-point iteration method. Show a convergent, a divergent, and a nonexistent solution. Explain your answers thoroughly.

4.1 Method(s) of solution

- Rewrite the equation f(x) as g(x)
- Initialize x0 as initial guess
- Set tolerance limit (tolerance = $1e^{-6}$) and max iterations (max_iter = 1000)
- For iterations, i, Compute for new x = g(x_i)
- If abs(x_{i+1} – x) < tolerance limit, stop the iteration and return x

4.2 Results

- for g(x) = $e^{ln4-xln10}$
  - `root: 3.9962859476437558`
  - `Number of iterations: 100`
- for g(x) = 10^{4-x}
  - `root: 0`
  - `Number of iterations: 100`
- For g(x) = $x + 5$
  - `root: 500`
  - `Number of iterations: 100`

4.3 Problems encountered

- The challenge comes with finding the right g(x). The first g(x) that I used, $-10^x + 4$, didn't work well since -10^x results into a very large number so I had to find another g(x) that would help find a convergent solution.

4.4 References

- ES_204_L2_Linear_Equations.pdf

PROBLEM 5 [20 points]: Given the system of equations below, solve x_1, x_2, x_3.

$$x_1 + x_2 + x_3 = 4$$
$$x_1^2 + x_2^2 + x_3^2 = 6$$
$$x_1 x_2 x_3 = 2$$

5.1 Method(s) of solution

- Define the Fx vector of equations
- Create a Jacobian Matrix, Jx, that computes the partial derivative of each equation with respect to each variable and place them on a matrix.
- Create an initial guess for vector x
- Set values for tolerance limit (tolerance = $1e^{-6}$) and max iterations (max_iter = 100).
- For k iterations, compute for Fx and Jx for current value of x
- Solve for Δx = $J^{-1} \cdot -F$
- Update x = x + Δx = $J^{-1} \cdot -F$ for each iteration
- If L2 norm of Fx is less than the tolerance limit, return x

5.2 Results (or partial results)

- `Solution: [0.99998074 1.00001926 2.0]`

5.3 Problems encountered

- When I use [0,0,0] or [1,1,1] as my initial guess, the Newton-Rhapson method would cause an error, “Singular Jacobian matrix encountered. Non-convergence.” I had to change x0 into [5,7,8] for it to avoid non-convergence.

5.4 References

- ES_204_L2_Linear_Equations.pdf

APPENDIX

PROBLEM 1

SOURCE CODE:

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

# Machine Problem # 1

# PROBLEM 1

class SOR_Method:
    def __init__(self, A, b, omegas, tolerance, max_iter):
        self.A = A
        self.b = b
        self.omegas = omegas  # A list of relaxation factors to test
        self.tolerance = tolerance
        self.max_iter = max_iter
        self.n = len(self.b)  # Determine the length of vector b
        self.x = np.zeros(self.n)  # Initialize vector x with zeros
        self.iterations = []
        self.x_solutions = []

    def gauss_seidel_model(self, x_bar_k_plus_1, i):
        x_bar_k_plus_1[i] = (self.b[i] - np.dot(self.A[i, :i], x_bar_k_plus_1[:i]) - np.dot(self.A[i, i+1:], self.x[i+1:])) / self.A[i, i]
        return x_bar_k_plus_1

    def successive_overrelaxation_method(self, omega):
        for k in range(self.max_iter):
            x_k_plus_1 = self.x.copy()
            x_bar_k_plus_1 = np.zeros(self.n)

            for i in range(self.n):
                x_bar_k_plus_1 = self.gauss_seidel_model(x_bar_k_plus_1, i)
                x_k_plus_1[i] = omega* x_bar_k_plus_1[i] + (1 - omega) * self.x[i]

            # Check for convergence using L∞ norm
            if np.max(np.abs(x_k_plus_1 - self.x)) < self.tolerance:
                return x_k_plus_1, k + 1

            self.x = x_k_plus_1  # Update x for the next iteration
```

```
        raise Exception("SOR did not converge within the specified number of iterations.")

    def omegas_solutions(self):

        omegas = self.omegas

        for omega in omegas:
            x_solution, num_iterations = self.successive_overrelaxation_method(omega)
            self.iterations.append(num_iterations)
            self.x_solutions.append(x_solution)

    def plot_omegas_solutions(self):
        plt.yscale('log')
        plt.plot(self.omegas, self.iterations, marker='o')
        plt.xlabel('Relaxation Factor (ω)')
        plt.ylabel('Number of Iterations for Convergence')
        plt.title('Number of Iterations vs. Relaxation Factor (ω)')
        plt.grid(True)
        plt.show()

        for i in range(len(self.omegas)):

            print("\033[1mOmega \033[0m" + str(i) + ":")
            print("                                    ")
            print("\033[1mω: \033[0m" + str(self.omegas[i]))
            print("\033[1miterations: \033[0m" + str(self.iterations[i]))
            print("\033[1mSolution: \033[0m")
            print(str(self.x_solutions[i]))
# Problem 1

n = 30
h = 1 / (n + 1)
tolerance = 10**-3

# Define matrix A
A = np.diag(-2 * np.ones(n)) + np.diag(np.ones(n-1), 1) + np.diag(np.ones(n-1), -1)

# Define vector b
b = h**2 * np.ones(n)

# Define the omegas that will be used for the test
omegas = np.linspace(1.0, 1.9, 10)

#Iterations
max_iter = 1000
```

```
print("A matrix: ")
print(A)
print("          ")

print("b matrix: ")
print(b)
print("          ")

Solution  = SOR_Method(A, b, omegas, tolerance, max_iter)
Solution.omegas_solutions()
Solution.plot_omegas_solutions()
```

SOLUTION:

$$\overline{x}_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

$$x_i^{(k+1)} = \omega \overline{x}_i^{(k+1)} + (1 - \omega) x_i^{(k)}$$

PROBLEM 2

SOURCE CODE:

```
import numpy as np
# PROBLEM 2

class Thomas_Algorithm:
    def __init__(self, A, b):
        self.A = A
        self.A_a = np.tril(A, k=-1).copy()
        self.A_b = np.diagonal(A).copy()
        self.A_c = np.triu(A, k=1).copy()
        self.b = b
        self.b_values = b.flatten()
        self.n = len(b)
        self.x = np.zeros(self.n)

    def forward_elimination(self):
        for i in range(1, self.n):
            factor = self.A_a[i, i - 1] / self.A_b[i - 1]
            self.A_b[i] -= factor * self.A_c[i - 1, i - 1]
```

```
            self.b_values[i] -= factor * self.b_values[i - 1]  # Update b_values

    def backward_substitution(self):
        self.x[self.n - 1] = self.b_values[self.n - 1] / self.A_b[self.n - 1]
        for i in range(self.n - 2, -1, -1):
            self.x[i] = (self.b_values[i] - self.A_a[i, i] * self.x[i + 1]) / self.A_b[i]



    def print_x_vector(self):
        print("\033[1mA matrix: \033[0m")
        print(self.A)
        print("          ")
        print("\033[1mb vector: \033[0m")
        print(self.b)
        print("          ")
        print("\033[1mx vector: \033[0m")
        print("          ")
        print(self.x)
# Problem 2

n = 101
h = 1 / (n + 1)

# Define matrix A
A = np.diag(-2 * np.ones(n)) + np.diag(np.ones(n-1), -1) + np.diag(np.ones(n-1), 1)

# Define vector b
b = h**2 * np.ones(n)

x = Thomas_Algorithm(A, b)
x.forward_elimination()
x.backward_substitution()
x.print_x_vector()
```

SOLUTION:

PROBLEM 3

SOURCE CODE:

PROBLEM 3

```
import numpy as np
class Biconjugate_Gradient_Method:
```

```
    def __init__(self, A, b, max_iter, tolerance=1e-6):
        self.A = A
        self.b = b
        self.n = len(self.b)
        self.x_o = np.random.randn(self.n)
        self.r_o = self.b - np.dot(self.A, self.x_o)
        self.p_o = self.Preconditioned(self.r_o)
        self.ast_b = self.Complex_Conjugate(self.b)
        self.ast_x_o = self.Complex_Conjugate(self.x_o)
        self.ast_r_o = self.Complex_Conjugate(b) - self.DOT_PRODUCT(self.Complex_Conjugate(A),
self.Complex_Conjugate(self.x_o))
        self.ast_p_o = self.Preconditioned(self.Complex_Conjugate(self.r_o))
        self.x_k = []
        self.r_k = []
        self.p_k = []
        self.ast_x_k = []
        self.ast_r_k = []
        self.ast_p_k = []
        self.max_iter = max_iter
        self.tolerance = tolerance


    def DOT_PRODUCT(self, matrix_1, matrix_2):
        # Function for getting the dot product of two matrices
        result = np.dot(matrix_1, matrix_2)
        return result

    @staticmethod
    def Complex_Conjugate(vector):
        # Function to compute the complex conjugate of a vector
        return np.conjugate(vector)

    def Preconditioned(self, matrix):
        D = np.diag(matrix)
        modified_D = D + 1e-10  # Add a small constant to avoid division by zero
        preconditioned_matrix = np.diag(1.0 / modified_D).dot(matrix)

        return preconditioned_matrix

    def Biconjugate_gradient(self):
        self.r_k.append(self.r_o)
        self.p_k.append(self.p_o)
        self.x_k.append(self.x_o)
        self.ast_r_k.append(self.ast_r_o)
        self.ast_p_k.append(self.ast_p_o)
        self.ast_x_k.append(self.ast_x_o)
```

```
        for k in range(self.max_iter):

            # Update a_k
            epsilon = 1e-10
            if np.any(self.Complex_Conjugate(self.p_k[k])*self.A*self.p_k[k]+1e-10)!= 0:
                a_k = self.Preconditioned(self.Complex_Conjugate(self.r_k[k]))*self.Preconditioned(self.r_k[k]) /
self.Complex_Conjugate(self.p_k[k])*self.A*self.p_k[k]+1e-10
            else:
                a_k = 0

            # Update x_k
            x_k_plus_1 = self.x_k[k] + self.DOT_PRODUCT(a_k, self.p_k[k])
            self.x_k.append(x_k_plus_1)

            # Update x*_k
            ast_x_k_plus_1 = self.ast_x_k[k] + self.DOT_PRODUCT(self.Complex_Conjugate(a_k),
self.Complex_Conjugate(self.p_k[k]))
            self.ast_x_k.append(ast_x_k_plus_1)

            # Update r_k
            r_k_plus_1 = self.r_k[k] - self.DOT_PRODUCT(a_k, self.A*self.p_k[k])
            self.r_k.append(r_k_plus_1)

            # Update r*_k
            ast_r_k_plus_1 = self.Complex_Conjugate(self.r_k[k]) - self.DOT_PRODUCT(self.Complex_Conjugate(a_k),
self.Complex_Conjugate(self.A)*self.Complex_Conjugate(self.p_k[k]))
            self.ast_r_k.append(ast_r_k_plus_1)

            # Compute B_k
            if np.any(self.Preconditioned(self.Complex_Conjugate(self.r_k[k]))*self.Preconditioned(self.r_k[k])+1e-10) != 0:
                B_k = self.Preconditioned(self.Complex_Conjugate(self.r_k[k+1]))*self.Preconditioned(self.r_k[k+1]) /
self.Preconditioned(self.Complex_Conjugate(self.r_k[k]))*self.Preconditioned(self.r_k[k])+1e-10
            else:
                B_k = 0

            # Update p_k
            p_k_plus_1 = self.Preconditioned(self.r_k[k+1]) + self.DOT_PRODUCT(B_k, self.p_k[k])
            self.p_k.append(p_k_plus_1)

            # Update p*_k
            ast_p_k_plus_1 = self.Preconditioned(self.Complex_Conjugate(self.r_k[k+1])) +
self.DOT_PRODUCT(self.Complex_Conjugate(B_k), self.Complex_Conjugate(self.p_k[k]))
            self.ast_p_k.append(ast_p_k_plus_1)

            # Check for convergence
            print(np.linalg.norm(r_k_plus_1))
            if np.linalg.norm(r_k_plus_1) < self.tolerance:
```

```python
            if np.linalg.norm(ast_r_k_plus_1) < self.tolerance:
                return self.x_k[-1]  # Convergence achieved

          print(self.r_k[k])

        # Convergence not achieved
        raise ConvergenceError("Biconjugate Gradient did not converge within max_iter.")

    def print_x_vector(self):
        print("\033[1mA matrix: \033[0m")
        print(self.A)
        print("          ")
        print("\033[1mb vector: \033[0m")
        print(self.b)
        print("          ")
        print("\033[1mx vector: \033[0m")
        print("          ")
        print(self.x_k[-1])

class ConvergenceError(Exception):
    pass


# Problem 3

n = 300
max_iter = 1000
h = 1 / (n + 1)

# Define matrix A
A = np.diag(-2 * np.ones(n)) + np.diag(np.ones(n-1), -1) + np.diag(np.ones(n-1), 1)

# Define vector b
b = h**2 * np.ones(n)

Solution = Biconjugate_Gradient_Method(A, b, max_iter)
Solution.Biconjugate_gradient()
Solution.print_x_vector()
```

SOLUTION:

1. Choose initial guess x_0 , two other vectors x_0^* and b^* and a preconditioner M
2. $r_0 \leftarrow b - A\,x_0$
3. $r_0^* \leftarrow b^* - x_0^*\,A^*$
4. $p_0 \leftarrow M^{-1} r_0$
5. $p_0^* \leftarrow r_0^* M^{-1}$
6. for $k = 0, 1, \ldots$ do
    1. $\alpha_k \leftarrow \dfrac{r_k^* M^{-1} r_k}{p_k^* A p_k}$
    2. $x_{k+1} \leftarrow x_k + \alpha_k \cdot p_k$
    3. $x_{k+1}^* \leftarrow x_k^* + \overline{\alpha_k} \cdot p_k^*$
    4. $r_{k+1} \leftarrow r_k - \alpha_k \cdot A p_k$
    5. $r_{k+1}^* \leftarrow r_k^* - \overline{\alpha_k} \cdot p_k^*\,A^*$
    6. $\beta_k \leftarrow \dfrac{r_{k+1}^* M^{-1} r_{k+1}}{r_k^* M^{-1} r_k}$
    7. $p_{k+1} \leftarrow M^{-1} r_{k+1} + \beta_k \cdot p_k$
    8. $p_{k+1}^* \leftarrow r_{k+1}^* M^{-1} + \overline{\beta_k} \cdot p_k^*$

PROBLEM 4

SOURCE CODE:

```
# PROBLEM 4
import numpy as np
import math

def fixed_point_iteration(g, x_o, max_iter, tolerance):

    x = x_o
    for i in range(max_iter):
        x_i_plus_1 = g(x)
        if abs(x_i_plus_1 - x) < tolerance:
            return x_i_plus_1, i + 1 # Able to reach Convergence
        x = x_i_plus_1
    return x_i_plus_1, i + 1 # Ran out of iterations. Failed to reach Convergence

def g(x):
```

```
    return math.exp(math.log(4) - x * math.log(10))

# Initial guess and maximum number of iterations
x_o = 0
max_iter = 100
tolerance = 1e-6

# Solve the equation using fixed-point iteration
root, iterations = fixed_point_iteration(g, x_o, max_iter, tolerance)

print(f"root: {root}")
print(f"Number of iterations: {iterations}")
```

SOLUTION:

1. Rewrite equation as $x = g(x)$
2. Initial approximation, x_0
3. Compute for new value $x_{i+1} = g(x_i)$
4. Test for convergence, $|x_{i+1} - x_i| < \Delta$
5. If $|x_{i+1} - x_i| > \Delta$, iterate

Upon convergence, the root is x_{i+1}

PROBLEM 5

SOURCE CODE:

```
# PROBLEM 5
Import numpy as np

class Newton_Raphson_Method:
    def __init__(self, Fx, Jx, x_o, omega=0.4, tolerance=1e-8, max_iter=100):
        self.Fx = Fx  # Function representing the system of equations
        self.Jx = Jx  # Jacobian matrix of partial derivatives
        self.x_o = x_o  # Initial guess
        self.tolerance = tolerance  # Convergence tolerance
        self.max_iter = max_iter  # Maximum number of iterations

    def Solve_L2_Norm(self, Vector):
        L2_Norm = np.linalg.norm(Vector)
        return L2_Norm
```

```
    def newton_rhapson_method(self):

        x = self.x_o

        for k in range(self.max_iter):
            # Compute F(x) and J(x) for the current x
            F_x = self.Fx(x)
            J_x = self.Jx(x)

            try:
                delta_x = np.linalg.solve(J_x, -F_x) # Δx = J⁻¹·-F
            except np.linalg.LinAlgError:
                raise Exception("Singular Jacobian matrix encountered. Non-convergence.")

            # x = x + Δx
            x = x + delta_x

            # Check for convergence based on the norm of F(x)
            L2_Norm = self.Solve_L2_Norm(F_x)
            if L2_Norm < self.tolerance:
                return x  # Convergence achieved

        raise Exception("Maximum number of iterations reached. Non-convergence.")

       # problem 5

Fx = lambda x: np.array([x[0] + x[1] + x[2] - 4, x[0]**2 + x[1]**2 + x[2]**2 - 6, x[0]*x[1]*x[2] - 2])  # Fx Function

def Jacobian_Matrix(x):
    J_x = np.zeros((3, 3))  # Initialize a 3x3 Jacobian matrix

    # Compute the partial derivatives of each equation with respect to each variable
    J_x[0, 0] = 1
    J_x[0, 1] = 1
    J_x[0, 2] = 1

    J_x[1, 0] = 2 * x[0]
    J_x[1, 1] = 2 * x[1]
    J_x[1, 2] = 2 * x[2]

    J_x[2, 0] = x[1] * x[2]
    J_x[2, 1] = x[0] * x[2]
    J_x[2, 2] = x[0] * x[1]

    print(J_x)
```

```
    return J_x

x_o = np.array([5, 7, 8])  # Initial guess

Solution = Newton_Raphson_Method(Fx, Jacobian_Matrix, x_o)

Solution = Solution.newton_rhapson_method()

print("Solution:", Solution)
```

SOLUTION:

➤ Newton-Raphson procedure

1. Initialize $\mathbf{f}$
2. Solve for $\Delta\mathbf{x}$ using $\Delta\mathbf{x} = \mathbf{J}^{-1} \cdot -\mathbf{f}$
3. Update x

$$\begin{aligned} x_{1_{i+1}} &= x_{1_i} + \Delta x_{1_i} \\ x_{2_{i+1}} &= x_{2_i} + \Delta x_{2_i} \\ &\vdots \\ x_{n_{i+1}} &= x_{n_i} + \Delta x_{n_i} \end{aligned}$$

4. Check for convergence, if not yet converged, update $\mathbf{f}$ with new values of $\mathbf{x}$ then iterate.

REFERENCES

- ES_204_L2_Linear_Equations.pdf
- Lee, W. T. (n.d.). Tridiagonal Matrices: Thomas Algorithm. MS6021, Scientific Computation, University of Limerick. Retrieved from http://www.industrial-maths.com/ms6021_thomas.pdf?fbclid=IwAR1kwmECuCPQGnGk5W378KKCnu_XWEX5LGTcC70hdvX1cELcfJiFJqkgiqE.
- Biconjugate gradient method. (n.d.). In Wikipedia. Retrieved April, 2023, from https://en.wikipedia.org/wiki/Biconjugate_gradient_method
- Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical Recipes: The Art of Scientific Computing (3rd ed.). Cambridge University Press. ISBN: 978-0-521-88068-8.

OTHERS