Project 2 Numerical Methods

- **Nicholas Sindoro C14220142**
- **Christophorus Ivan C14220210**
- **Marvel Wilbert Odelio C14220223**

Provision:

1. **Work in groups: Max 3 people.**
2. **All files are made into 1 folder and compressed into one before uploading.**
3. **Collected by only 1 representative/student.**
4. **Descriptions, explanations and how to use the program are written in Word, if there are images, please insert them into Word too.**
5. **Collected Before June 15, 2024; at 23.59.**

Github: https://github.com/M4RV3L24/Project-2-Numerical-Method.git

Question:

1. **Write a program in Matlab/Octave and Python that calculates the roots of n equations with n unknown variables using the method:**
    a. **Gaus-Seidel**
    b. **Jacobi**
    c. **Newton Raphson**

    with input:
    a. **n as an order of equations/unknown variables and**
    b. **matrix A as the equation coefficient matrix,**
    c. **the number of decimal digits of the accuracy to be achieved**

    Provide explanations/comments on the program, how to use it and examples of calculations with a minimum of n = 3.

2. **Look for data from the internet, for example from https://www.linkedin.com/pulse/your-data-science-projects-here-30-free-datasets-paresh-patil-qin6f/**
    Then calculate the mathematical model that meets the "Best Fit" criteria from the data using Regression (Least Square) both Linear and Non-Linear and Interpolation. Write a program in Matlab/Octave and Python to help with calculations.

Provide an explanation of step by step calculations, plot data and regression and interpolation results, display table 2 of calculation results and provide analysis of the results obtained

No 1
these are the equations and stop criteria that we will use for program testing.

$$3\,x1 - 0.1\,x2 - 0.2\,x3 = 7.85$$
$$0.1\,x1 - 7\,x2 - 0.3\,x3 = -19.3$$
$$0.3\,x1 - 0.2\,x2 - 10\,x3 = 71.4$$

Ɛs = 0.000001

The equations above have the following solutions:
x1 = 3
x2 = -2.5
x3 = 7

Gauss Seidel

a. **Python**

To run the program below, several modules are needed such as tabular, numpy, and matplotlib. If when running it the error "Module Not Found" appears, then you need to install the module by typing the following command in the terminal:

1. **pip install module_name**
2. **pip show module_name // just to make sure that the installation was successful**
3. **Alternatively, you can also copy and paste the code below into Google Collab, and then run the code**

```
# Nicholas Sindoro      C14220142
# Christophorus Ivan    C14220210
# Marvel Wilbert O      C14220223

import numpy as np
import matplotlib.pyplot as plt
from tabulate import tabulate as tab


def gauss_seidel(A, b, n, es):

    global table_content, table_header

    # check if n < 3
    if n < 3:
```

```
        print("Number of unknowns must be at least 3")
        return

    # Initialize the solution vector with zeros
    x = np.zeros(n)

    # Maximum number of iterations to avoid infinite loops
    max_iter = 1000

    # Initialize the approximate relative error vector
    ea = np.zeros(n)

    # Iterative process
    for k in range(max_iter):
        # Store the current solution to compare later

        x_old = x.copy()

        # Iterate through each variable
        for i in range(n):
            # Calculate the sum of A[i, j] * x[j] for j < i
            sum1 = np.dot(A[i, :i], x[:i])

            # Calculate the sum of A[i, j] * x_old[j] for j > i
            sum2 = np.dot(A[i, i+1:], x_old[i+1:])

            # Update the value of x[i] using the Gauss-Seidel formula
            x[i] = (b[i] - sum1 - sum2) / A[i, i]

            # Calculate the approximate relative error
            ea[i] = abs((x[i] - x_old[i])/x[i]) * 100

        table_content.append([k+1, np.array2string(x), np.array2string(ea)])

        # Check for convergence using Eucledean norm to ensure all x is
considered.
        # if np.linalg.norm(x - x_old) < es:
        #     # If the solution has converged, exit the loop
        #     break
```

```
        # Check for convergence using the maximum relative error
        if np.max(ea) <= es:
            # If the solution has converged, exit the loop
            break

    # Return the final solution vector
    print(tab(table_content, headers=table_header, tablefmt="grid"))
    return x

# data for table
table_header = ['iteration', 'xi', 'ea(%)']
table_content = []

# Define the coefficient matrix A
A = np.array([
    [3, -0.1, -0.2],  # Coefficients of the first equation
    [0.1, 7, -0.3],  # Coefficients of the second equation
    [0.3, -0.2, 10]   # Coefficients of the third equation
])

# Define the constant vector b
b = np.array([7.85, -19.3, 71.4])  # Constants on the right-hand side of the
equations

# Define the number of unknowns n
n = len(b)  # Number of equations/unknowns

# Define the tolerance for convergence
es = 1e-6  # Accuracy to be achieved

# Solve the system using the Gauss-Seidel method
x = gauss_seidel(A, b, n, es) #same as using alpha = 1 for relaxation

# Print the solution
print("Solution:", x)
```

Output

```
+-------------+-----------------------------------------+-----------------------------------------------+
|   iteration | xi                                      | ea(%)                                         |
+=============+=========================================+===============================================+
|           1 | [ 2.61666667 -2.79452381  7.00560952] | [100. 100. 100.]                              |
+-------------+-----------------------------------------+-----------------------------------------------+
|           2 | [ 2.99055651 -2.49962468  7.00029081] | [12.50234999 11.79773614  0.07597845]         |
+-------------+-----------------------------------------+-----------------------------------------------+
|           3 | [ 3.0000319  -2.49998799  6.99999928] | [0.31584297 0.01453237 0.00416468]            |
+-------------+-----------------------------------------+-----------------------------------------------+
|           4 | [ 3.00000035 -2.50000004  6.99999999] | [1.05151459e-03 4.81736055e-04 1.00785031e-05] |
+-------------+-----------------------------------------+-----------------------------------------------+
|           5 | [ 3.  -2.5  7. ]                      | [1.18137902e-05 1.41194249e-06 1.61976883e-07] |
+-------------+-----------------------------------------+-----------------------------------------------+
|           6 | [ 3.  -2.5  7. ]                      | [6.44170347e-08 1.83329441e-08 6.97270813e-10] |
+-------------+-----------------------------------------+-----------------------------------------------+
Solution: [ 3.  -2.5  7. ]
```

b. **Octave**

The program below is a function for finding the results of linear algebra equations using the Gauss Seidel method. Run this program through the main.m file that is available in the folder MATLAB. make sure the main.m file is in the same folder as the file GaussSeidel.m

```
% Nicholas Sindoro      C14220142
% Christophorus Ivan    C14220210
% Marvel Wilbert O      C14220223

function x = GaussSeidel(A, b, n, es, maxit)
    % GaussSeidel: Solves system of linear equations using Gauss-Seidel method
    %
    % Inputs:
    % A - coefficient matrix
    % b - constant vector
    % n - number of equations
    % es - desired relative error (default: 0.00001)
    % maxit - maximum number of iterations (default: 50)
    %
    % Output:
    % x - solution vector

    % Check if at least 3 input arguments are provided
    if nargin < 3
        error('At least 3 input arguments required');
    end

    % Set default maximum iterations if not provided
    if nargin < 5 || isempty(maxit)
```

```
    maxit = 50;
end

% Set default tolerance if not provided
if nargin < 4 || isempty(es)
    es = 0.00001;
end

% Get the size of matrix A
[m, n_A] = size(A);

% Check if A is a square matrix and matches the number of equations n
if m ~= n_A || m ~= n
    error('Matrix A must be square and match the number of equations n');
end

% Initialize C matrix and x vector
C = A;
x = zeros(n, 1);

% Set the diagonal elements of C to zero
for i = 1:n
    C(i, i) = 0;
end

% Normalize the rows of C and calculate the d vector
for i = 1:n
    C(i, :) = C(i, :) / A(i, i);
    d(i) = b(i) / A(i, i);
end

% Initialize iteration counter and error array
iter = 0;
ea = zeros(n, 1);
errors = zeros(maxit, 1);


% Start iteration
while (1)
    x_old = x;  % Store the old x values for error calculation

    % Update each element of x
    for i = 1:n
        x(i) = d(i) - C(i, :) * x;
```

```
        % Calculate the approximate relative error
        if x(i) ~= 0
            ea(i) = abs((x(i) - x_old(i)) / x(i)) * 100;
        end
    end

    % Increment the iteration counter
    iter = iter + 1;

    max_ea = max(ea);
    errors(iter) = max_ea;
    disp(['Iteration ', num2str(iter), ': Approximation error = ', num2str(max_ea)]);

    % Check for convergence or maximum iterations
    if max(ea) <= es || iter >= maxit
        break;
    end
  end
end
```

Jacobi

a. **Python**

To run the program below, several modules are needed such as tabular, numpy, and matplotlib. If when running it the error "Module Not Found" appears, then you need to install the module by typing the following command in the terminal:

1. **pip install module_name**
2. **pip show module_name // just to make sure that the installation was successful**
3. **Alternatively, you can also copy and paste the code below into Google Collab, and then run the code**

```python
# Nicholas Sindoro      C14220142
# Christophorus Ivan    C14220210
# Marvel Wilbert O      C14220223

import numpy as np
import matplotlib.pyplot as plt
from tabulate import tabulate as tab

def jacobi(A, b, n, es):

    global table_content, table_header
```

```
# check if n < 3
if n < 3:
    print("Number of unknowns must be at least 3")
    return

# Initialize the solution vector with zeros
x = np.zeros(n)

# Maximum number of iterations to avoid infinite loops
max_iter = 1000

# Initialize the approximate relative error vector
ea = np.zeros(n)

# Iterative process
for k in range(max_iter):
    # Store the current solution to compare later
    x_old = x.copy()

    # Iterate through each variable
    for i in range(n):
        # Calculate the sum of A[i, j] * x_old[j] for all j
        sum1 = np.dot(A[i, :], x_old)

        # Update the value of x[i] using the Jacobi formula
        x[i] = (b[i] - sum1 + A[i, i] * x_old[i]) / A[i, i]

        # Calculate the approximate relative error
        ea[i] = abs((x[i] - x_old[i])/x[i]) * 100


    # Compute the Euclidean norm of the error
    # error = np.linalg.norm(x - x_old)

    # Append the current error to the list
    table_content.append([k+1, np.array2string(x), np.array2string(ea)])
```

```
        # Check for convergence
        if np.max(ea) < es:
            # If the solution has converged, exit the loop
            break


    # Return the final solution vector
    print(tab(table_content, headers=table_header, tablefmt="grid"))

    # Return the final solution vector and the list of errors
    return x


# data for table
table_header = ['iteration', 'xi', 'ea(%)']
table_content = []

# Define the coefficient matrix A
A = np.array([
    [3, -0.1, -0.2],  # Coefficients of the first equation
    [0.1, 7, -0.3],  # Coefficients of the second equation
    [0.3, -0.2, 10]   # Coefficients of the third equation
])

# Define the constant vector b
b = np.array([7.85, -19.3, 71.4])  # Constants on the right-hand side of the
equations

# Define the number of unknowns n
n = len(b)  # Number of equations/unknowns

# Define the tolerance for convergence
es = 1e-6  # Accuracy to be achieved

# Solve the system using the Jacobi method
x = jacobi(A, b, n, es)

# Print the solution
print("Solution:", x)
```

Output

```
+-------------+---------------------------------------+-----------------------------------------------+
|   iteration | xi                                    | ea(%)                                         |
+=============+=======================================+===============================================+
|           1 | [ 2.61666667 -2.75714286  7.14      ] | [100. 100. 100.]                              |
+-------------+---------------------------------------+-----------------------------------------------+
|           2 | [ 3.0007619  -2.48852381  7.00635714] | [12.79992383 10.79431294  1.9074514 ]         |
+-------------+---------------------------------------+-----------------------------------------------+
|           3 | [ 3.00080635 -2.49973844  7.00020667] | [0.00148108 0.44863197 0.08786135]            |
+-------------+---------------------------------------+-----------------------------------------------+
|           4 | [ 3.0000225  -2.50000266  6.99998104] | [0.02612822 0.01056906 0.00322324]            |
+-------------+---------------------------------------+-----------------------------------------------+
|           5 | [ 2.99999865 -2.50000113  6.99999927] | [7.94976423e-04 6.11285761e-05 2.60443500e-04] |
+-------------+---------------------------------------+-----------------------------------------------+
|           6 | [ 2.99999991 -2.50000001  7.00000002] | [4.22114470e-05 4.48813776e-05 1.06577537e-05] |
+-------------+---------------------------------------+-----------------------------------------------+
|           7 | [ 3.  -2.5  7. ]                      | [2.90457773e-06 5.55305669e-07 2.22137318e-07] |
+-------------+---------------------------------------+-----------------------------------------------+
|           8 | [ 3.  -2.5  7. ]                      | [1.91295276e-08 7.64492292e-08 3.33781130e-08] |
+-------------+---------------------------------------+-----------------------------------------------+
Solution: [ 3.  -2.5  7. ]
```

b. **Octave**

The program below is a function for finding the results of linear algebra equations using the Jacobi method. Run this program through the main.m file that is available in the folder MATLAB. make sure the main.m file is in the same folder as the file Jacobi.m

```
% Nicholas Sindoro      C14220142
% Christophorus Ivan    C14220210
% Marvel Wilbert O      C14220223

function [x, errors] = Jacobi(A, b, n, es, maxit)
   % Jacobi: Solves system of linear equations using Jacobi method
   %
   % Inputs:
   % A - coefficient matrix
   % b - constant vector
   % n - number of equations
   % es - desired relative error (default: 0.00001)
   % maxit - maximum number of iterations (default: 50)
   %
   % Output:
   % x - solution vector
   % errors - array of approximation errors at each iteration
```

```
    % Check if at least 3 input arguments are provided
    if nargin < 3
        error('At least 3 input arguments required');
    end

    % Set default maximum iterations if not provided
    if nargin < 5 || isempty(maxit)
        maxit = 50;
    end

    % Set default tolerance if not provided
    if nargin < 4 || isempty(es)
        es = 0.00001;
    end

    % Get the size of matrix A
    [m, n_A] = size(A);

    % Check if A is a square matrix and matches the number of equations n
    if m ~= n_A || m ~= n
        error('Matrix A must be square and match the number of equations n');
    end

    % Initialize x vector
    x = zeros(n, 1);

    % Initialize iteration counter and error array
    iter = 0;
    errors = zeros(maxit, 1);

    % Start iteration
    while true
        x_old = x;  % Store the old x values for error calculation

        % Update each element of x
        for i = 1:n
            sigma = A(i, :) * x_old - A(i, i) * x_old(i);
            x(i) = (b(i) - sigma) / A(i, i);

            % Calculate the approximate relative error
            if x(i) ~= 0
                ea(i) = abs((x(i) - x_old(i)) / x(i)) * 100;
            end
        end
```

```
        % Increment the iteration counter
        iter = iter + 1;

        % Calculate maximum error and store in errors array
        max_ea = max(ea);
        errors(iter) = max_ea;
        disp(['Iteration ', num2str(iter), ': Approximation error = ', num2str(max_ea)]);

        % Check for convergence or maximum iterations
        if max(ea) <= es || iter >= maxit
            break;
        end
    end

    % Trim errors array to remove excess zeros
    errors = errors(1:iter);
end
```

Newton Raphson

a. Python

To run the program below, several modules are needed such as tabular and numpy. If when running it the error "Module Not Found" appears, then you need to install the module by typing the following command in the terminal:

1. pip install module_name
2. pip show module_name // just to make sure that the installation was successful
3. Alternatively, you can also copy and paste the code below into Google Collab, and then run the code

```python
# Nicholas Sindoro      C14220142
# Christophorus Ivan    C14220210
# Marvel Wilbert O      C14220223

import numpy as np
from tabulate import tabulate as tab

def newton_raphson(n, A, b, precision):
    global table_content, table_header

    # Initialize solution vector x with zeros
    x = np.zeros(n)
```

```
    # Error tolerance
    es = (0.5 * 10**(2 - precision)) / 100

    # Maximum number of iterations to avoid infinite loops
    max_iter = 1000

    # Initialize the iteration counter
    iter_count = 0

    while iter_count < max_iter:
        # Make a copy of x to calculate the error (ea)
        x_temp = np.copy(x)

        # Initialize Jacobian matrix j
        j = np.zeros((n, n))
        for i in range(n):
            for k in range(n):
                j[i][k] = A[i][k]

        # Calculate f(x), which is the function values at the current
approximation
        fx = np.zeros(n)
        for i in range(n):
            equation = -b[i]
            for k in range(n):
                equation += (A[i][k] * x[k])
            fx[i] = equation

        # Update the solution vector using the Newton-Raphson formula: x =
x - J^-1 * f(x)
        x -= np.dot(np.linalg.inv(j), fx)

        # Calculate the error for each element of the solution vector
        ea = np.abs((x - x_temp) / x) * 100

        # Add iteration results to the table
```

```
        table_content.append([iter_count + 1, np.array2string(x,
precision=precision), np.array2string(ea, precision=precision)])

        # Check if the stopping criteria is met
        if np.all(ea <= es):
            break

        iter_count += 1

    # Print the table
    print(tab(table_content, headers=table_header, tablefmt="grid"))

    return x

# data for table
table_header = ['Iteration', 'xi', 'ea(%)']
table_content = []

# Define the coefficient matrix A
A = np.array([
    [3, -0.1, -0.2],  # Coefficients of the first equation
    [0.1, 7, -0.3],   # Coefficients of the second equation
    [0.3, -0.2, 10]   # Coefficients of the third equation
])

# Define the constant vector b
b = np.array([7.85, -19.3, 71.4])  # Constants on the right-hand side of
the equations

# Number of unknowns
n = len(b)

# Decimal precision
precision = 5

# Solve the system using the Newton-Raphson method
result = newton_raphson(n, A, b, precision)
```

```
# Print the solution
print("Solution vector:")
print(result)
```

Output

```
+-------------+------------------+-----------------------------------------+
|   Iteration | xi               | ea(%)                                   |
+=============+==================+=========================================+
|           1 | [ 3.  -2.5  7. ] | [100. 100. 100.]                        |
+-------------+------------------+-----------------------------------------+
|           2 | [ 3.  -2.5  7. ] | [0.00000e+00 3.55271e-14 2.53765e-14] |
+-------------+------------------+-----------------------------------------+
Solution vector:
[ 3.  -2.5  7. ]
```

b. **Octave**

The program below is a function for finding the results of linear algebra equations using the Newton Raphson method. Run this program through the main.m file that is available in the folder MATLAB. make sure the main.m file is in the same folder as the file Raphson.m

```
% Nicholas Sindoro      C14220142
% Christophorus Ivan    C14220210
% Marvel Wilbert O      C14220223

function x = Raphson(A, b, n, precision)
    % Newton-Raphson function to find the solution of a system of linear equations
    % Using the Newton-Raphson method.
    %
    % Inputs:
    % n - number of equations/unknown variables
    % A - matrix of equation coefficients
    % b - vector of constants on the right side of the equations
    % precision - number of decimal digits for the desired accuracy
    %
    % Outputs:
    % x - solution vector

    % Initialize the solution vector x with zeros
    x = zeros(n, 1);
```

```
    % Error tolerance
    es = (0.5 * 10^(2 - precision)) / 100;

    % Initialize iteration counter
    iter_count = 0;

    % Perform iterations until convergence
    while true
        % Copy x to calculate the error (ea)
        x_temp = x;

        % Initialize the Jacobian matrix j
        j = zeros(n, n);
        for i = 1:n
            for k = 1:n
                j(i, k) = A(i, k);
            end
        end

        % Calculate f(x), the function value at the current approximation
        fx = zeros(n, 1);
        for i = 1:n
            equation = -b(i);
            for k = 1:n
                equation += (A(i, k) * x(k));
            end
            fx(i) = equation;
        end

        % Update the solution vector using the Newton-Raphson formula: x = x - J^-1 * f(x)
        x -= inv(j) * fx;

        % Calculate the error for each element of the solution vector
        ea = abs((x - x_temp) ./ x) * 100;

        % Print the results of the iteration
        printf('Iteration %d:\n', iter_count + 1);
        printf('xi = %s\n', mat2str(x, precision));
        printf('ea(%%) = %s\n\n', mat2str(ea, precision));

        % Check if the stopping criterion is met
        if all(ea <= es)
            break;
        end
```

```
        iter_count += 1;
    end
End
```

Additional File - main.m
used to run all the octave code in problem number 1

```
n = 3; % Number of equations/unknown variables
A = [3, -0.1, -0.2;0.1, 7, -0.3; 0.3, -0.2, 10]; % Coefficient matrix A
b = [7.85, -19.3, 71.4]; % Right-hand side vector b
es = 1e-6; % Tolerance for accuracy

x_gs = GaussSeidel(A, b, n, es);
disp('Solusi Gauss Seidel: ');
disp(x_gs);

disp('');

x_jacobi = Jacobi(A, b, n, es);
disp("Solusi Jacobi: ");
disp(x_jacobi);

disp('');

x_raphson = Raphson(A, b, n, precision);
disp("Solusi Newton-Raphson: ");
disp(x_raphson);
```

Output

No 2

Dataset: https://www.kaggle.com/datasets/mikhail1681/walmart-sales

For number 2, we will evaluate the data of walmart sales. The data consist of several columns, such as store number, date, weekly_sales, holiday_flag, temperature, fuel_price, CPI, and unemployment rate. For this question, we will evaluate the correlation between unemployment rate and weekly_sales. The unemployment rate will be on the x axis as an independent variable whereas the weekly_sales will be on the y axis as the dependent variable.

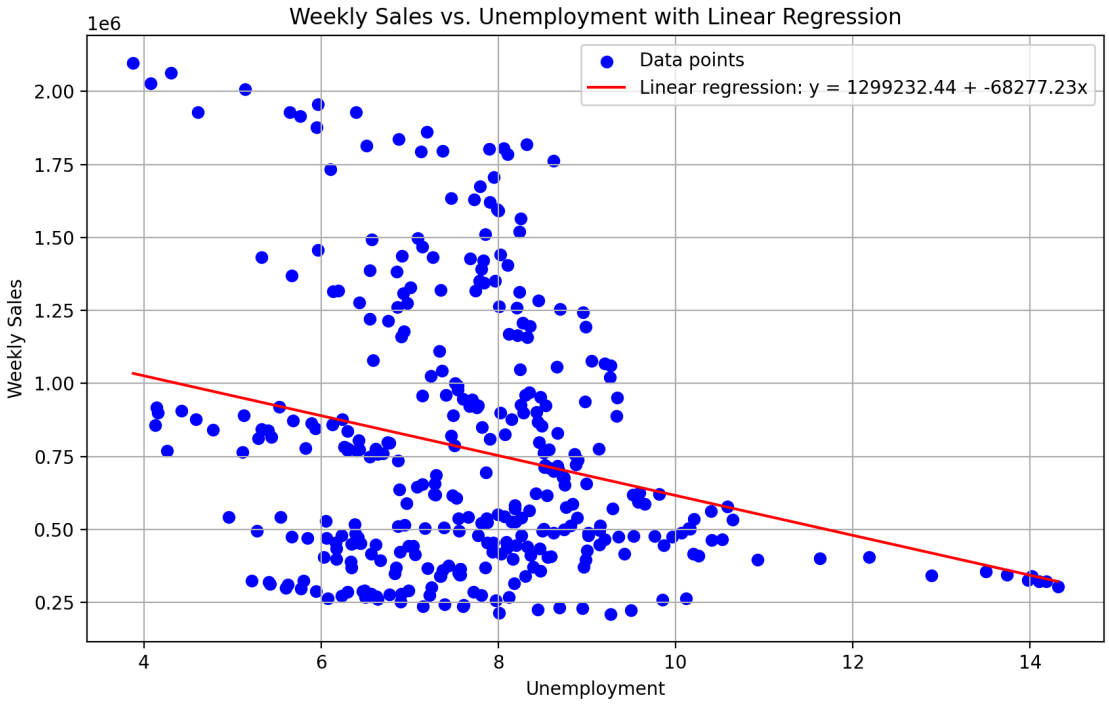
Linear Regression

Python

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

# Read csv file
df = pd.read_csv('Walmart_Sales.csv')

# Get unemployment and weekly_sales data
x = df['Unemployment']
y = df['Weekly_Sales']

# Number of data
n = len(x)

# Calculate data for linear regression
sum_x = np.sum(x)
sum_y = np.sum(y)
sum_xy = np.sum(x * y)
sum_x2 = np.sum(x ** 2)
mean_x = np.mean(x)
mean_y = np.mean(y)
```

```
# Calculate a1, a0
a1 = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x ** 2)
a0 = mean_y - a1 * mean_x

# Calculate the linear regression
regression_line = a0 + a1 * x

# Print the coefficients
print(f'Slope (a1): {a1}')
print(f'Intercept (a0): {a0}')

# Create the plot
plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='blue', label='Data points')
plt.plot(x, regression_line, color='red', label=f'Linear
regression: y = {a0:.2f} + {a1:.2f}x')

# Add title & label
plt.title('Weekly Sales vs. Unemployment with Linear
Regression')
plt.xlabel('Unemployment')
plt.ylabel('Weekly Sales')
plt.legend()

# Show the plot
plt.grid(True)
plt.show()
```

Output



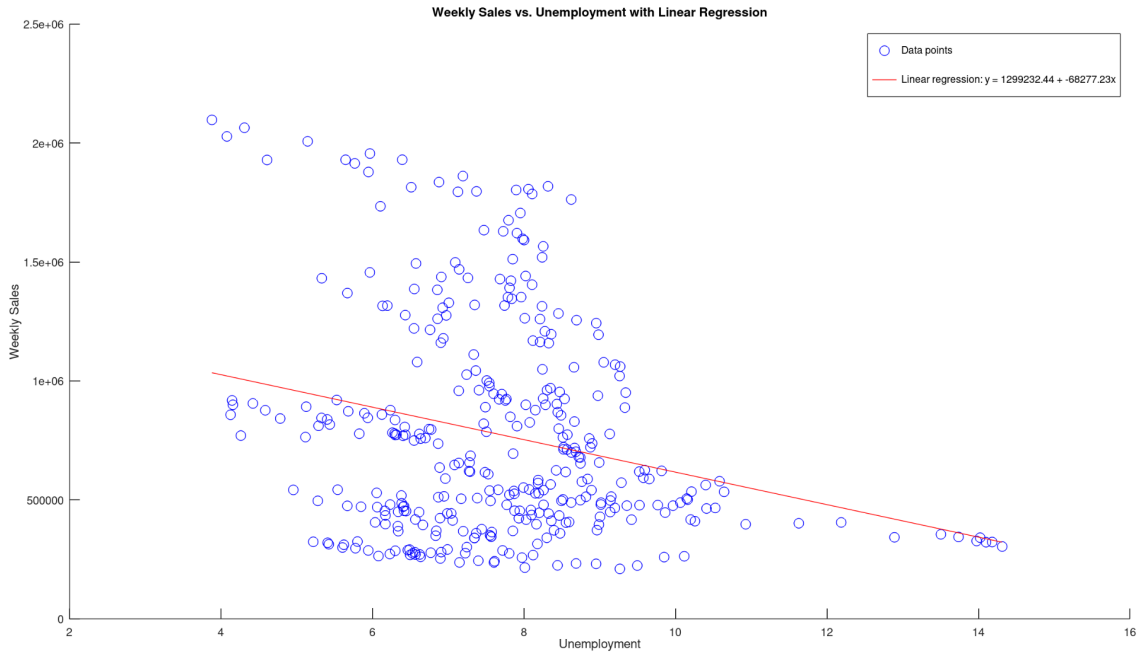

```
Slope (a1): -68277.22527865427
Intercept (a0): 1299232.4395259824
```

Matlab

```
% Read the CSV file
data = csvread('Walmart_Sales.csv', 1, 0);

% Extract columns
x = data(:, 8);  % Unemployment
y = data(:, 3);  % Weekly_sales

% Number of data points
n = length(x);

% Calculate data for linear regression
sum_x = sum(x);
sum_y = sum(y);
sum_xy = sum(x .* y);
```

```
sum_x2 = sum(x .^ 2);
mean_x = mean(x);
mean_y = mean(y);

% Calculate a1, a0
a1 = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x^2);
a0 = mean_y - a1 * mean_x;

% Calculate the linear regression line
regression_line = a0 + a1 * x;

% Print the coefficients
printf('Slope (a1): %.2f\n', a1);
printf('Intercept (a0): %.2f\n', a0);

% Create the plot
figure;
scatter(x, y, 'b', 'DisplayName', 'Data points');
hold on;
plot(x, regression_line, 'r', 'DisplayName', sprintf('Linear regression: y = %.2f + %.2fx', a0, a1));

% Add title and labels
title('Weekly Sales vs. Unemployment with Linear Regression');
xlabel('Unemployment');
ylabel('Weekly Sales');
legend('show');
set(gca, 'FontSize', 24);
```

Output



```
Slope (a1): -68277.23
Intercept (a0): 1299232.44
```

Explaination

1. Firstly, we will read the csv file using library.

```
# Read csv file
df = pd.read_csv('Walmart_Sales.csv')
```

2. Then, read the columns that we will examine, in this case ‘Unemployment’ and ‘Weekly_Sales’

```
# Get unemployment and weekly_sales data
x = df['Unemployment']
y = df['Weekly_Sales']
```

3. After obtaining the data, we will calculate the linear regression line with sum_x, sum_y, sum_xy, sum_x2 (x^2), mean_x, and mean_y. We already have the equation for linear regression, so we just substitute the data to the equation.

```
# Number of data
n = len(x)

# Calculate data for linear regression
sum_x = np.sum(x)
sum_y = np.sum(y)
sum_xy = np.sum(x * y)
sum_x2 = np.sum(x ** 2)
mean_x = np.mean(x)
mean_y = np.mean(y)

# Calculate a1, a0
a1 = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x ** 2)
a0 = mean_y - a1 * mean_x

# Calculate the linear regression
regression_line = a0 + a1 * x
```

4. Lastly, we just simply plot the graph.

Analysis

We can see that the linear regression line have a negative slope, which means that unemployment and weekly_sales have a negative correlation. The bigger the unemployment, the smaller the weekly_sales and vice versa. However, linear regression still cannot describe the trend perfectly since there are still a lot of data scattered far away from the regression line.

Nonlinear Regression

For the nonlinear regression, we use the model function of y = a * (1 - e^(b * x)) + c, where a, b, and c is the coefficients to be optimized. To optimize the coefficients, we use the SSE function and fminsearch function to search for the optimal number.

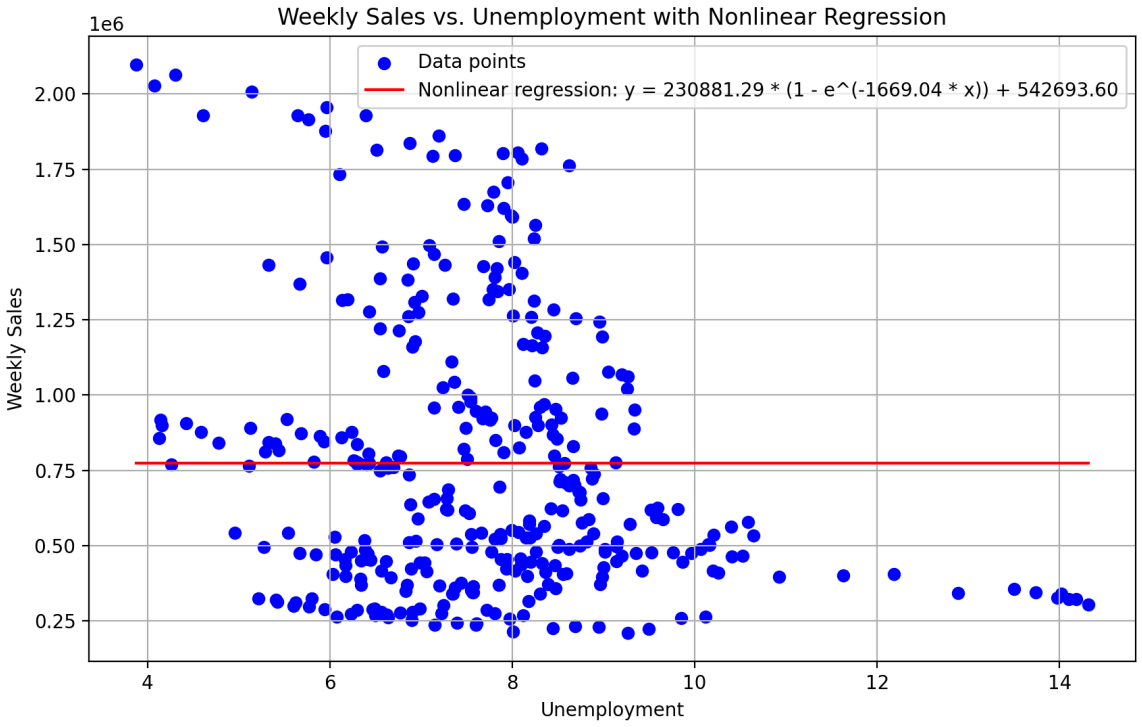
Python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize

# Read csv file
df = pd.read_csv('Walmart_Sales.csv')

# Get unemployment and weekly_sales data
x = df['Unemployment'].values
y = df['Weekly_Sales'].values

# Define the nonlinear model function as y = a * (1 - e^(b *
x)) + c
def model(params, x):
    a, b, c = params
    return a * (1 - np.exp(b * x)) + c

# Define function for SSE
```

```
def SSE(params, x, y):
    y_model = model(params, x)
    return np.sum((y - y_model) ** 2)

# Initial guess for a, b, c
initial_params = [1.0, -0.1, 1.0]

# Search the optimal a, b, and c using fmin function
result = scipy.optimize.fmin(func=SSE, x0=initial_params,
args=(x, y))

# Get the optimized parameters
a_opt, b_opt, c_opt = result

# Print final parameters
print(f'Optimized parameters: a = {a_opt:.4f}, b = {b_opt:.4f},
c = {c_opt:.4f}')

# Calculate fitted values
y_pred = model(result, x)

# Create the plot
plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='blue', label='Data points')
plt.plot(x, y_pred, color='red', label=f'Nonlinear regression:
y = {a_opt:.2f} * (1 - e^({b_opt:.2f} * x)) + {c_opt:.2f}')

# Add title & label
plt.title('Weekly Sales vs. Unemployment with Nonlinear
Regression')
```

```
plt.xlabel('Unemployment')
plt.ylabel('Weekly Sales')
plt.legend()

# Show the plot
plt.grid(True)
plt.show()
```

Output




```
Optimized parameters: a = 230881.2877, b = -1669.0366, c = 542693.5958
```

Matlab

```
% Read the CSV file
data = csvread('Walmart_Sales.csv', 1, 0);

% Extract columns
x = data(:, 8);  % Unemployment
y = data(:, 3);  % Weekly_sales

% Define the nonlinear model function as y = a * (1 - e^(b * x)) + c
function y_model = model(params, x)
    a = params(1);
    b = params(2);
    c = params(3);
    y_model = a * (1 - exp(b * x)) + c;
end

% Define function for SSE
function sse = SSE(params, x, y)
    y_model = model(params, x);
    sse = sum((y - y_model) .^ 2);
end

% Initial guess for a, b, c
initial_params = [1.0, -0.1, 1.0];

% Search the optimal a, b, and c using fminsearch function
result = fminsearch(@(params) SSE(params, x, y), initial_params);

% Get the optimized parameters
a_opt = result(1);
b_opt = result(2);
c_opt = result(3);

% Print final parameters
printf('Optimized parameters: a = %.4f, b = %.4f, c = %.4f\n', a_opt, b_opt, c_opt);

% Calculate fitted values
y_pred = model(result, x);

% Create the plot
figure;
scatter(x, y, 'b', 'DisplayName', 'Data points');
hold on;
```

```
plot(x, y_pred, 'r', 'DisplayName', sprintf('Nonlinear regression: y = %.2f * (1 - e^{%.2f * x}) + %.2f', a_opt, b_opt, c_opt));

% Add title & label
title('Weekly Sales vs. Unemployment with Nonlinear Regression');
xlabel('Unemployment');
ylabel('Weekly Sales');
legend('show');
set(gca, 'FontSize', 24);
```

Output


```
Optimized parameters: a =
                        368052.9988, b = -104796.9266, c = 405521.8924
```

Explaination

1. Firstly, we will read the csv file using library.

```
# Read csv file
df = pd.read_csv('Walmart_Sales.csv')
```

2. Then, read the columns that we will examine, in this case ‘Unemployment’ and ‘Weekly_Sales’

```
# Get unemployment and weekly_sales data
x = df['Unemployment']
y = df['Weekly_Sales']
```

3. Define the model function for nonlinear regression

```
# Define the nonlinear model function as y = a * (1 - e^(b * x)) + c
def model(params, x):
    a, b, c = params
    return a * (1 - np.exp(b * x)) + c
```

4. Define the function to calculate SSE

```
# Define function for SSE
def SSE(params, x, y):
    y_model = model(params, x)
    return np.sum((y - y_model) ** 2)
```

5. Set initial guess for coefficients

```
# Initial guess for a, b, c
initial_params = [1.0, -0.1, 1.0]
```

6. With the help of fmin, calculate the optimal a, b, c based on initial guess and SSE function

```
# Search the optimal a, b, and c using fmin function
result = scipy.optimize.fmin(func=SSE, x0=initial_params, args=(x, y), xtol=1e-8, ftol=1e-8)

# Get the optimized parameters
a_opt, b_opt, c_opt = result
```

7. Plot the graph with the optimal model

Analysis

The nonlinear model y = a * (1 - e^(b * x)) + c didn't really help to describe the data. As we can see, the the model resulting a straight line with almost no slope at all. The data plots are also scattered far away from the nonlinear regression line. We can conclude that with this nonlinear regression line model, we can't make a conclusion of the data behaviour.

Interpolation

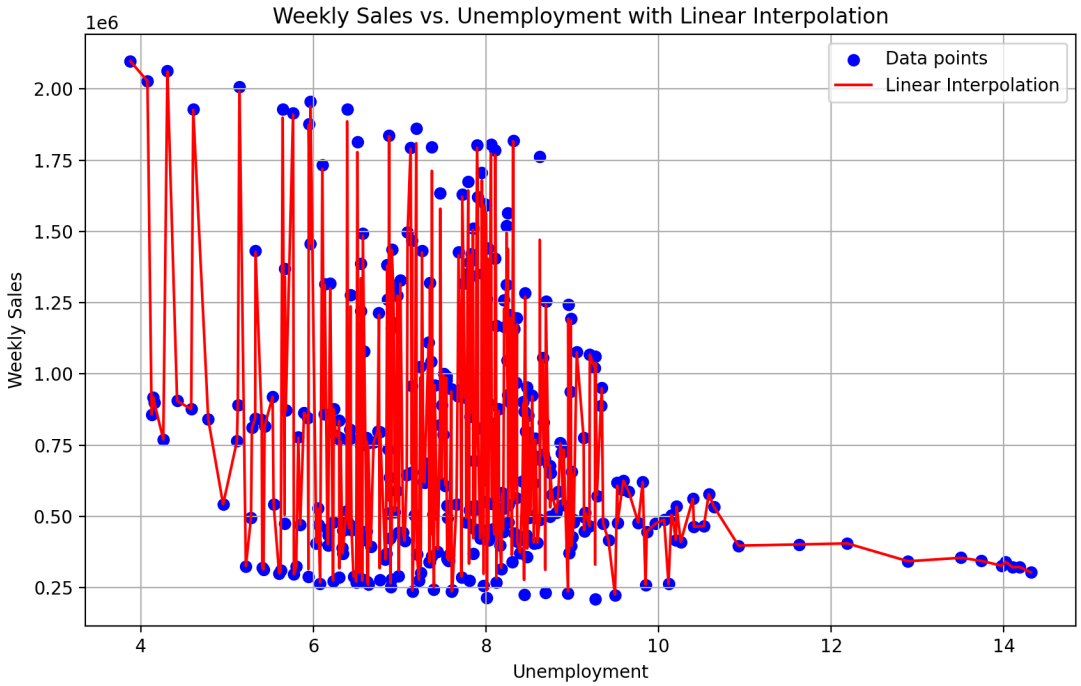
We will use linear interpolation to examine the data. Linear interpolation can be found with the equation y = y1 + (y2 - y1) / (x2 - x1) * (x - x1).

Python

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scipy.optimize

# Read csv file
df = pd.read_csv('Walmart_Sales.csv')

# Get unemployment and weekly_sales data
x = df['Unemployment'].values
y = df['Weekly_Sales'].values

# Function to do linear interpolation
def linearInterpolate(x_search):
    # Find the closest 2 point
    for i in range(0, len(x)-1):
        if x[i] <= x_search and x_search <= x[i+1]:
            return y[i] + (y[i+1] - y[i]) * (x_search - x[i]) /
(x[i+1] - x[i])
```

```
# Generate 10,000 new x point to show the interpolation result
x_interpolate = np.linspace(x.min(), x.max(), 10000)

# Interpolate to find corresponding weekly_sales values using
Linear Interpolation
y_interpolate = np.array([])
for x_search in x_interpolate:
                y_interpolate    =    np.append(y_interpolate,
linearInterpolate(x_search))

# Create the plot
plt.figure(figsize=(10, 6))
plt.scatter(x, y, color='blue', label='Data points')
plt.plot(x_interpolate,       y_interpolate,       color='red',
label=f'Linear Interpolation')

# Add title & label
plt.title('Weekly   Sales   vs.   Unemployment   with   Linear
Interpolation')
plt.xlabel('Unemployment')
plt.ylabel('Weekly Sales')
plt.legend()

# Show the plot
plt.grid(True)
plt.show()
```

Output

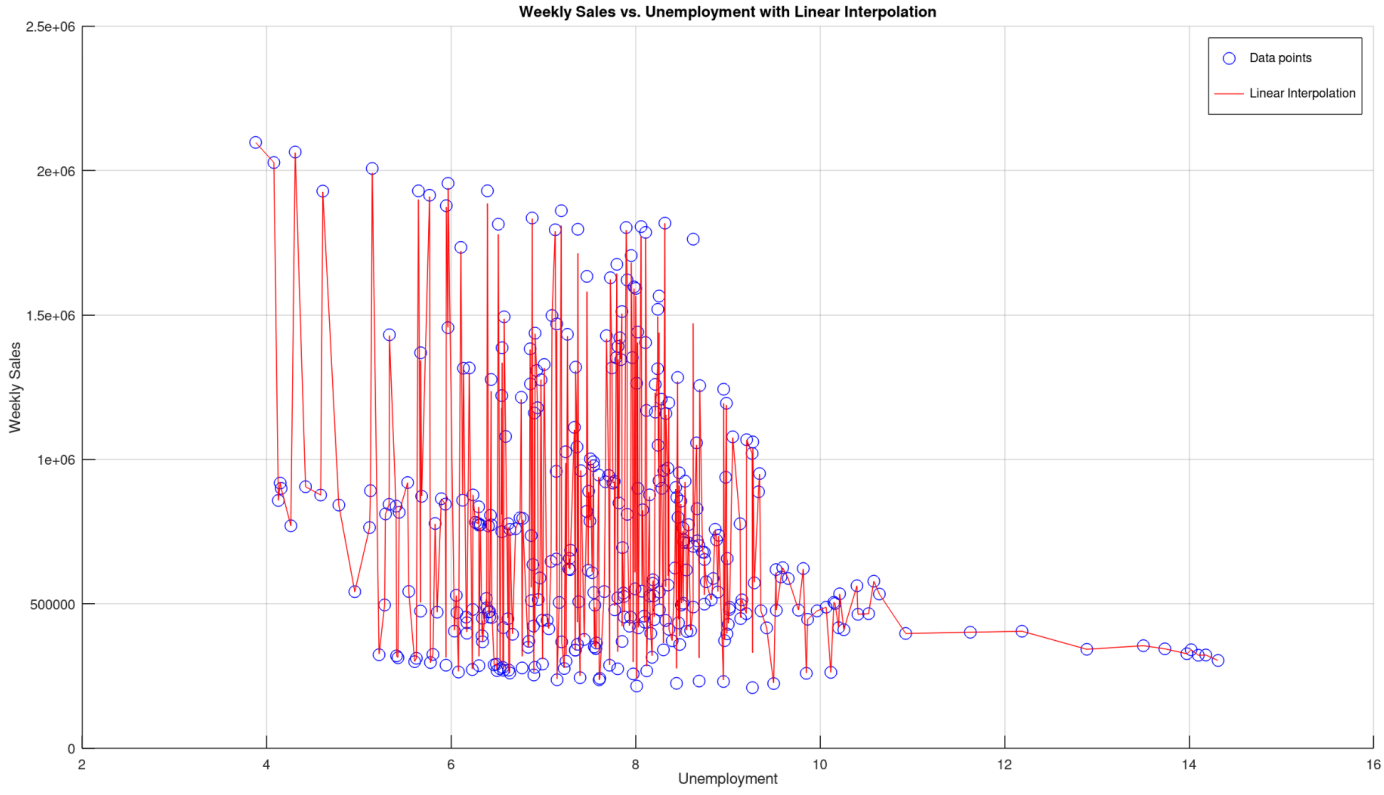


Matlab

```
% Read the CSV file
data = csvread('Walmart_Sales.csv', 1, 0);

% Extract unemployment and weekly_sales data
x = data(:, 8); % Assuming 'Unemployment' is the 8th column
y = data(:, 3); % Assuming 'Weekly_Sales' is the 3rd column

% Function to do linear interpolation
function y_search = linearInterpolate(x, y, x_search)
    % Find the closest 2 points
    for i = 1:length(x) - 1
        if x(i) <= x_search && x_search <= x(i+1)
            y_search = y(i) + (y(i+1) - y(i)) * (x_search - x(i)) / (x(i+1) - x(i));
            return;
        end
    end
end

% Generate 10,000 new x points to show the interpolation result
x_interpolate = linspace(min(x), max(x), 10000);
```

```
% Interpolate to find corresponding weekly_sales values using Linear Interpolation
y_interpolate = zeros(size(x_interpolate));
for i = 1:length(x_interpolate)
    y_interpolate(i) = linearInterpolate(x, y, x_interpolate(i));
end

% Create the plot
figure;
hold on;
scatter(x, y, 'b', 'DisplayName', 'Data points');
plot(x_interpolate, y_interpolate, 'r', 'DisplayName', 'Linear Interpolation');

% Add title & label
title('Weekly Sales vs. Unemployment with Linear Interpolation');
xlabel('Unemployment');
ylabel('Weekly Sales');
legend('show');
set(gca, 'FontSize', 24);
grid on;

hold off;
```

Output



Explaination

1. Firstly, we will read the csv file using library.

```
# Read csv file
df = pd.read_csv('Walmart_Sales.csv')
```

2. Then, read the columns that we will examine, in this case ‘Unemployment’ and ‘Weekly_Sales’

```
# Get unemployment and weekly_sales data
x = df['Unemployment']
y = df['Weekly_Sales']
```

3. We also define a function to count the interpolated y value from a x value in the parameter. Because we are using linear interpolation, the function will seek for the closest 2 point from the x_search and calculate the y value.

```
# Function to do linear interpolation
def linearInterpolate(x_search):
    # Find the closest 2 point
    for i in range(0, len(x)-1):
        if x[i] <= x_search and x_search <= x[i+1]:
            return y[i] + (y[i+1] - y[i]) * (x_search - x[i]) / (x[i+1] - x[i])
```

4. Then we will generate 10000 x points in the range minimal x and maximal x from our dataset. For each x, we will find the y using the interpolation function mentioned above.

```
# Generate 10,000 new x point to show the interpolation result
x_interpolate = np.linspace(x.min(), x.max(), 10000)

# Interpolate to find corresponding weekly_sales values using Linear Interpolation
y_interpolate = np.array([])
for x_search in x_interpolate:
    y_interpolate = np.append(y_interpolate, linearInterpolate(x_search))
```

5. After calculating all the interpolation points, we plot the graph.

Analysis

The interpolation technique is not really dependable on our dataset. This is due to the extreme change in y value between 2 neighboring x value. We can see the interpolation points is spiking up and down dramatically between 2 close x value.