

A few notes about this assignment

This assignment re-uses the matrix class I developed for the previous assignment. However, there are two significant additions (code pasted below): the implementation of a constructor that lets you pass a lambda to define the values of a matrix and a member function that determines the inverse of a matrix.

The member function “Inverse” uses a process similar to Gauss-Jordan elimination, but I find it more straight-forward. However, if performance were a concern for large matrices, it is not an ideal way to determine the inverse. For these assignments it’s not a concern, but saving the factorization of a matrix would be more efficient than explicitly calculating and storing its inverse in most cases.

I also wish to note that I’ve shared my matrix class with a few people: Johnathan Tousley, Josh Lake, and Aaron Kunz. They may submit older versions of this class than what I end up submitting as I have since added features which they solved the problems without, but I found made the code even nicer.

I figured this wouldn’t be an issue for a few reasons. First, the process of matrix/vector multiplication and other things that this code simplifies are not the principal algorithms we are meant to code in this assignment. Also, the algorithms were provided by the professor (mine is just nicer) with the exception of the inverse function (which was developed in the last assignment). Finally, my code is on public domain: <https://github.com/Assimilator/ENGR2450/blob/master/src/shared/matrix.hpp>

```
class Matrix {
private:
    typedef std::function<double(int, int)> matrix_map;

public:
    Matrix(int m, int n, matrix_map f) { resize(m, n, f); };

    void resize(int m, int n, matrix_map f) {
        clean();
        Rows = m;
        Cols = n;
        _array = new T*[Rows];
        for (int i = 0; i < Rows; ++i) {
            _array[i] = new T[Cols];
            for (int j = 0; j < Cols; ++j) {
                _array[i][j] = f(i, j);
            }
        }
    }

    void each(std::function<void(T&, int, int)> f) {
        for (int i = 0; i < Rows; ++i) {
            for (int j = 0; j < Cols; ++j) {
                f(_array[i][j], i, j);
            }
        }
    };
};
```

```

template <typename T>
Matrix<T> Matrix<T>::Inverse(bool& error) const {
    // Only deal with square matrices
    if (Rows != Cols) {
        error = true;
        return *this;
    }

    T temp;
    int n = Rows;
    Matrix<T> left(*this);
    Matrix<T> right(n, n);

    // Make the Identity Matrix
    for (int i = 0; i < n; ++i) {
        right[i][i] = 1;
    }

    // Perform Gaussian Elimination column by column
    for (int col = 0; col < n; ++col) {
        // find first pivot row where left[pivot][col] != 0
        int pivot = col;
        while (left[pivot][col] == 0) {
            if (++pivot >= n) {
                error = true;
                return left;
            }
        }

        // basic swap of rows between two matrices
        if (pivot != col) {
            for (int i = 0; i < n; ++i) {
                temp = left[pivot][i];
                left[pivot][i] = left[col][i];
                left[col][i] = temp;

                temp = right[pivot][i];
                right[pivot][i] = right[col][i];
                right[col][i] = temp;
            }
        }

        // Normalize the row so left[col][col] = 1
        temp = left[col][col];
        for (int i = 0; i < n; ++i) {
            left[col][i] /= temp;
            right[col][i] /= temp;
        }

        // Substitute up and down
        for (int row = 0; row < n; ++row) {
            if (row != col) {
                temp = left[row][col];
                for (int i = 0; i < n; ++i) {
                    left[row][i] -= temp * left[col][i];
                    right[row][i] -= temp * right[col][i];
                }
            }
        }
    }
}

```

```

    }
}

// Verify the identity matrix is resulting
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        if (left[i][j] != (i == j ? 1 : 0)) {
            error = true;
            return left;
        }
    }
}

error = false;
return right;
}

```

Common Code

main.cpp:

```
//-----+
// John Call - A01283897
// Driver for ENGR 2450 homework
//-----+
#include "assign4\assign4.hpp"
#include <iostream>

int main() {
    assign4::main();

    system("pause");
    return 0;
}
```

assign4.hpp:

```
#pragma once
#include <vector>

namespace assign4 {
    typedef const std::vector<double> const_vector;
    struct LRegress { double m, b, syx, r2; };
    LRegress Regress(const_vector&, const_vector&);
    void main();
}
```

assign4.cpp:

```
#include "assign4.hpp"
#include "a4p1.hpp"
#include "a4p2.hpp"

#include "../shared/matrix.hpp"
using namespace assign4;

//-----+
// LRegress Regress
// Determine the coefficients for the linear regression of  $y = mx + b$ 
//-----+
LRegress assign4::Regress(const_vector& x, const_vector& y) {
    LRegress lReg;
    double
        sumx = 0, sumy = 0,
        sumxy = 0, sumx2 = 0,
        st = 0, sr = 0;

    int n = x.size();
    for (int i = 0; i < n; ++i) {
        sumx += x[i];
        sumy += y[i];
    }
```

```

        sumxy += x[i] * y[i];
        sumx2 += x[i] * x[i];
    }

    double xm = sumx / n;
    double ym = sumy / n;
    lReg.m = (n * sumxy - sumx * sumy) / (n * sumx2 - sumx * sumx);
    lReg.b = ym - lReg.m * xm;

    for (int i = 0; i < n; ++i) {
        st += (y[i] - ym) * (y[i] - ym);
        sr += (y[i] - lReg.m * x[i] - lReg.b) * (y[i] - lReg.m * x[i] -
lReg.b);
    }

    lReg.syx = sqrt(sr / (n - 2));
    lReg.r2 = (st - sr) / st;

    return lReg;
}

//-----+
// std::vector<double> NLRegress                                |
// Determine the coefficients for systems with many variables    |
//-----+
std::vector<double> assign4::NLRegress(Matrix<double> Z, const_vector& y,
bool& e) {
    if (y.size() != Z.Rows) { e = true; return y; }
    Matrix<double> ZT = Z.Transpose();
    Matrix<double> ZI = Inverse(ZT * Z, e);
    if (e) { return y; }
    return ZI * (ZT * y);
}

void assign4::main() {
    Problem1();
    Problem2();
    Problem3();
}

```

a4pX.hpp (replace X with any problem number, all headers looks like this):

```

#pragma once

namespace assign4 {
    void ProblemX();
}

```

Problem 1

[1] LINEAR REGRESSION - [This is the same as problem 17.23] -- Using your preferred high-level language (VBA, C++, etc.) write a program to perform least-square linear regression. Your program should contain a main program that performs the following tasks:

- Reads a vector of data x of n elements
- Reads a vector of data y of m elements
- Checks if $m = n$.
 - If so,
 - call Sub Regress (see pseudocode in Figure 17.6)
 - print the following results: n , $a1$, $a0$, syx , $r2$, r
 - call Sub FitData (this calculates $yf = a0 + a1 \cdot x$ for vector x - no pseudocode given)
 - print the values of x , y , and yf
 - if m and n are not equal, indicate that no regression is possible.

(a) Using your program solve problem 17.4 - What your solution should include:

- table of fitted values yf
- plot of the original data as points and fitted data as a continuous line (e.g., using Excel)
- code for the program used

(b) Transform the (x,y) data as required (e.g., using Excel, or by hand) and use your program to solve problem 17.7 - parts (a) and (b), i.e. 2 fittings - What your solution should include for both (a) and (b):

- the fitted equation
- the correlation coefficient calculated r

17.4 Use least-squares regression to fit a straight line to

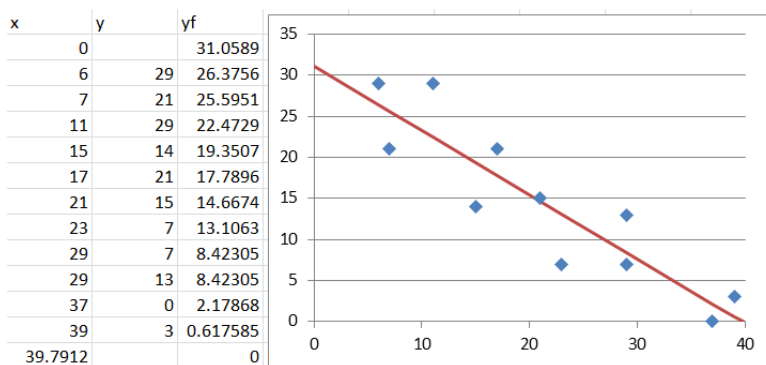
x	6	7	11	15	17	21	23	29	29	37	39
y	29	21	29	14	21	15	7	7	13	0	3

Along with the slope and the intercept, compute the standard error of the estimate and the correlation coefficient. Plot the data and the regression line. If someone made an additional measurement of $x = 10, y = 10$, would you suspect, based on a visual assessment and the standard error, that the measurement was valid or faulty? Justify your conclusion.

17.7 Fit the following data with (a) a saturation-growth-rate model, (b) a power equation, and (c) a parabola. In each case, plot the data and the equation.

x	0.75	2	3	4	6	8	8.5
y	1.2	1.95	2	2.4	2.4	2.7	2.6

Part A:



```
m: -0.780547
b: 31.0589
syx: 4.47631
r2: 0.812682
```

x	y	yf
6	29	26.3756
7	21	25.5951
11	29	22.4729
15	14	19.3507
17	21	17.7896
21	15	14.6674
23	7	13.1063
29	7	8.42305
29	13	8.42305
37	0	2.17868
39	3	0.617585

Part B:

```
Saturation-Growth Rate
m: 0.36932
b: 0.34154
syx: 0.0210242
r2: 0.985711
```

x	y	f_x	f_y	fit
0.75	1.2	1.33333	0.833333	0.833967
2	1.95	0.5	0.512821	0.5262
3	2	0.333333	0.5	0.464647
4	2.4	0.25	0.416667	0.43387
6	2.4	0.166667	0.416667	0.403094
8	2.7	0.125	0.37037	0.387705
8.5	2.6	0.117647	0.384615	0.38499

```
alpha: 2.92791
beta: 1.08134
y = 2.92791x / (1.08134 + x)
```

```
Power Equation
m: 0.311422
b: 0.153296
syx: 0.0336484
r2: 0.935471
```

x	y	f_x	f_y	fit
0.75	1.2	-0.124939	0.0791812	0.114387
2	1.95	0.30103	0.290035	0.247043
3	2	0.477121	0.30103	0.301882
4	2.4	0.60206	0.380211	0.34079
6	2.4	0.778151	0.380211	0.395629
8	2.7	0.90309	0.431364	0.434538
8.5	2.6	0.929419	0.414973	0.442737

```
alpha: 1.4233
beta: 0.311422
y = 1.4233x^0.311422
```

a4p1.cpp:

```
#include "a4p1.hpp"
#include "assign4.hpp"
using namespace assign4;

#include <math.h>
#include <vector>
#include <iomanip>
#include <functional>

std::vector<double> FitData(const_vector& x, double m, double b) {
    int n = x.size();
    std::vector<double> y(n);
    for (int i = 0; i < n; ++i) {
        y[i] = m * x[i] + b;
    }
    return y;
}

void PartA(const_vector& x, const_vector& y) {
    LRegress lReg = Regress(x, y);
    std::vector<double> yf = FitData(x, lReg.m, lReg.b);
    std::cout
        << "Part A:" << std::endl << std::endl
        << "m: " << lReg.m << std::endl
        << "b: " << lReg.b << std::endl
        << "syx: " << lReg.syx << std::endl
        << "r2: " << lReg.r2 << std::endl << std::endl
        << std::setw(4) << "x"
        << std::setw(6) << "y"
        << std::setw(8) << "yf" << std::endl
        << "-----" << std::endl;

    int n = x.size();
    for (int i = 0; i < n; ++i) {
        std::cout
            << std::setw(4) << x[i]
```

```

        << std::setw(6) << y[i]
        << std::setw(10) << yf[i] << std::endl;
    }

    std::cout << std::endl;
}

typedef std::function<double(double)> trans;
std::vector<double> PartB(const_vector& x, const_vector& y, LRegress& lReg,
trans x_map, trans y_map) {
    int n = x.size();
    std::vector<double> f_x(n), f_y(n);
    for (int i = 0; i < n; ++i) {
        f_x[i] = x_map(x[i]);
        f_y[i] = y_map(y[i]);
    }

    lReg = Regress(f_x, f_y);
    std::vector<double> fit = FitData(f_x, lReg.m, lReg.b);

    std::cout
        << "m: " << lReg.m << std::endl
        << "b: " << lReg.b << std::endl
        << "syx: " << lReg.syx << std::endl
        << "r2: " << lReg.r2 << std::endl << std::endl
        << std::setw(3) << "x"
        << std::setw(6) << "y"
        << std::setw(9) << "f_x"
        << std::setw(10) << "f_y"
        << std::setw(10) << "fit"
        << std::endl
        << "-----" << std::endl;

    for (int i = 0; i < n; ++i) {
        std::cout
            << std::setw(4) << x[i]
            << std::setw(6) << y[i]
            << std::setw(10) << f_x[i]
            << std::setw(10) << f_y[i]
            << std::setw(10) << fit[i]
            << std::endl;
    }

    std::cout << std::endl;
    return fit;
}

void assign4::Problem1() {
    std::cout << "Problem 1:" << std::endl;

    // Common variables
    std::vector<double> x {6, 7, 11, 15, 17, 21, 23, 29, 29, 37, 39};
    std::vector<double> y {29, 21, 29, 14, 21, 15, 7, 7, 13, 0, 3};

    // Do Part A
    PartA(x, y);
}

```



```

// Do Part B
x = {.75, 2, 3, 4, 6, 8, 8.5};
y = {1.2, 1.95, 2, 2.4, 2.4, 2.7, 2.6};

std::cout
    << "Part B:" << std::endl << std::endl
    << "Saturation-Growth Rate" << std::endl;

// Saturation growth rate
LRegress lReg;
double alpha, beta;

PartB(x, y, lReg,
    [](double x) -> double { return 1 / x; },
    [](double y) -> double { return 1 / y; });

alpha = 1 / lReg.b;
beta = lReg.m * alpha;

std::cout
    << "alpha: " << alpha << std::endl
    << "beta: " << beta << std::endl
    << "y = " << alpha << "x / (" << beta << " + x)" << std::endl
    << std::endl
    << "Power Equation" << std::endl;

// Power Equation
PartB(x, y, lReg,
    [](double x) -> double { return log10(x); },
    [](double y) -> double { return log10(y); });



alpha = pow(10, lReg.b);
beta = lReg.m;

std::cout
    << "alpha: " << alpha << std::endl
    << "beta: " << beta << std::endl
    << "y = " << alpha << "x^" << beta << std::endl
    << std::endl;
}



```

Problem 2

[2]. POLYNOMIAL REGRESSION - Section 17.4.1 shows how to use matrices to determine the coefficients a_0, a_1, \dots, a_m , of a linear least squares fitting of the form $y_f = a_0z_0 + a_1z_1 + \dots + a_mz_m$, where z_0, z_1, \dots, z_m are $(m+1)$ basis functions. Given a table of data values for z , you can form matrix $[Z]$ (see p. 77), which, together with a vector of observed values of y , $\{Y\}$, can be used to find the vector of coefficients $\{A\}$ according to equation (17.26). This approach can be used to produce polynomial regression (this problem), and multiple-linear regression (problem [3]).

In this problem you are to **develop a subroutine *NLRegress*** that will take an $n \times (m+1)$ matrix $[Z]$, and a vector $\{Y\}$, of size n , and calculate the vector $\{A\}$, of size $(m+1)$ according to equation (17.26). [An outline for the operation to *NLRegress* is available HERE](#)  . To implement the step-by-step calculation shown in the outline, you can use subroutines for matrix operations from Assignment 3: matrix transpose, matrix multiplication, multiplication of a matrix by a vector, and the matrix inverse subroutine (with attendant subroutines) that you developed in Assignment 3.

You are also to **develop a main program** to do the following:

- Read a vector x of size n
- Read a vector y of size n
- Read the order m of a polynomial for a polynomial fitting (e.g., $m = 2$)
- Call a new subroutine *BuildZP*, that creates a matrix $[Z]$, so that the first column is full of 1's, the second corresponds to vector x , the third to x^2 , and so on. [A pseudocode for subroutine *BuildZP* is available HERE](#)  .
- Call Sub *NLRegress* to calculate the vector of coefficients $\{a\}$
- Calculate the fitted values $y_f = [Z]^T \{a\}$. Use subroutine *MultiplyMatrixToVector*(Z, a, y_f). This subroutine was used in Assignment 3.
- Show the table of values of x, y , and y_f
- Show the values of vector $\{a\}$

(a) Use your program to solve problem 17.20. What your solution needs to show:

- Table of x, y , and y_f data
- Plot of the original data (x, y) as points and polynomial fittings as continuous lines
- Polynomial used for the fitting, e.g., $y = a_0 + a_1x + a_2x^2$

(b) Use your program to solve problem 20.22. What your solution needs to show [NOTE: USE DOUBLE PRECISION DATA and use *tol* as a very small value (say $1E-10$), or remove that check from your code]:

- Table of T, DO , and DO_f (fitted) data
- Plot of the original data (T, DO) as points and polynomial fittings as continuous lines
- Polynomial used for the fitting, e.g., $DO = a_0 + a_1T + a_2T^2 + a_3T^3$

(c) Show the code for the main program and the subroutines developed in this problem only. You don't need to show the subroutines carried over from Part 3 (matrix input, output, multiplication, etc.)

17.20 Use nonlinear regression to fit a parabola to the following data:

x	0.2	0.5	0.8	1.2	1.7	2	2.3
y	500	700	1000	1200	2200	2650	3750

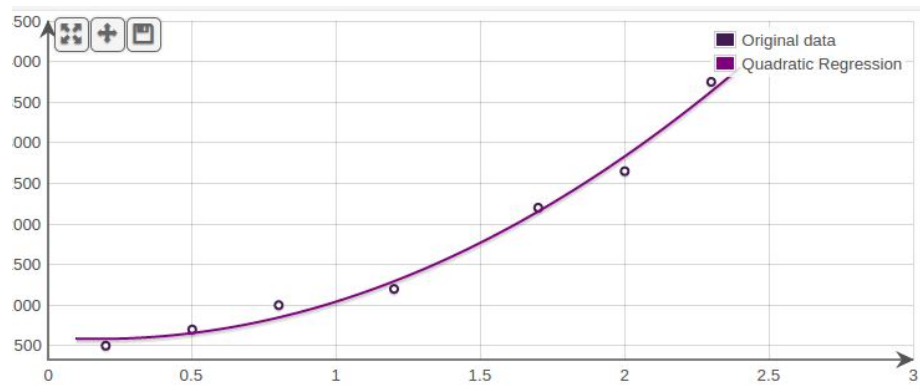
20.22 For the data in Table P20.21, use polynomial regression to derive a third-order predictive equation for dissolved oxygen concentration as a function of temperature for the case where chloride concentration is equal to 10 g/L. Use the equation to estimate the dissolved oxygen concentration for $T = 8^\circ\text{C}$.

Table P20.21 Dissolved oxygen concentration in water as a function of temperature ($^\circ\text{C}$) and chloride concentration (g/L).

$T, ^\circ\text{C}$	Dissolved Oxygen (mg/L) for Temperature ($^\circ\text{C}$) and Concentration of Chloride (g/L)		
	$c = 0 \text{ g/L}$	$c = 10 \text{ g/L}$	$c = 20 \text{ g/L}$
0	14.6	12.9	11.4
5	12.8	11.3	10.3
10	11.3	10.1	8.96
15	10.1	9.03	8.08
20	9.09	8.17	7.35
25	8.26	7.46	6.73
30	7.56	6.85	6.20

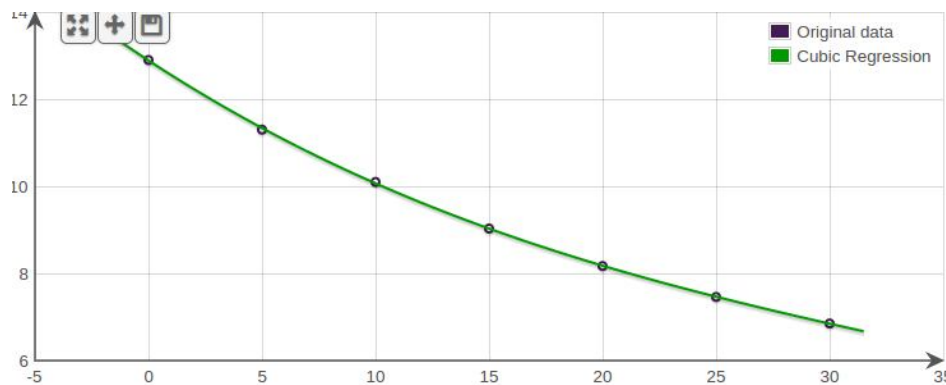
Part a:

x	y	yf
0.2	500	584.262
0.5	700	655.615
0.8	1000	848.29
1.2	1200	1293.91
1.7	2200	2154.24
2	2650	2832.2
2.3	3750	3631.48

$$y = 604.094 + -233.961x + 674.007x^2$$


Part b:

x	y	yf
0	12.9	12.8879
5	11.3	11.3376
10	10.1	10.0669
15	9.03	9.02905
20	8.17	8.17738
25	7.46	7.46524
30	6.85	6.84595

$$y = 12.8879 + -0.341111x + 0.00652381x^2 + -6.22222e-005x^3$$


a4p2.cpp:

```
#include "a4p2.hpp"
#include "assign4.hpp"
using namespace assign4;

#include <math.h>
#include <vector>
#include <iomanip>
#include <functional>

void PrintFit(const_vector& x, const_vector& y, int power) {
    bool error = false;
    Matrix<double> Z(x.size(), power + 1, [&x](int r, int c) {
        return pow(x[r], c);
    });
    std::vector<double>
        a = NLRegress(Z, y, error),
        yf = Z * a;

    std::cout
        << std::setw(4) << "x"
        << std::setw(6) << "y"
        << std::setw(8) << "yf" << std::endl
        << "-----" << std::endl;

    int n = x.size();
    for (int i = 0; i < n; ++i) {
        std::cout
            << std::setw(4) << x[i]
            << std::setw(6) << y[i]
            << std::setw(10) << yf[i] << std::endl;
    }
    std::cout << std::endl
        << "y = " << a[0] << " + " << a[1] << "x";
    for (int i = 2; i < power + 1; ++i) {
        std::cout << " + " << a[i] << "x^" << i;
    }
    std::cout << std::endl << std::endl;
}

void assign4::Problem2() {
    std::vector<double> x, y;
    std::cout << "Problem 2:" << std::endl;

    // Part A
    x = {0.2, 0.5, 0.8, 1.2, 1.7, 2, 2.3};
    y = {500, 700, 1000, 1200, 2200, 2650, 3750};
    std::cout << "Part A:" << std::endl << std::endl;
    PrintFit(x, y, 2);

    // Part B
    x = {0, 5, 10, 15, 20, 25, 30};
    y = {12.9, 11.3, 10.1, 9.03, 8.17, 7.46, 6.85};
    std::cout << "Part B:" << std::endl << std::endl;
    PrintFit(x, y, 3);
}
```

Problem 3

[3] MULTIPLE-LINEAR REGRESSION - In this program we seek a fitting of the form $y = a_0 + a_1x_1 + a_2x_2 + \dots + a_mx_m$, i.e., a multiple-linear fitting in which the dependent variable y is a function of m explanatory variables (x_1, x_2, \dots, x_m). The data for the fitting is given in the form of m vectors with n data points each, corresponding to the m explanatory variables. For the purpose of inputting the data into a program, these m vectors can be put together into a matrix X , with n rows and m columns, each column corresponding to variables x_1, x_2, \dots, x_m , respectively. We also need to provide a vector y of n elements.

Develop a main program to do the following:

- Read a matrix X of size n -by- m (n rows, m columns) containing n data points for each of the m explanatory variables x_1, x_2, \dots, x_m .
- Read a vector y of size n
- Call a new subroutine *BuildZM*, that builds a matrix $[Z]$, so that its first column is full of 1's, and the rest of the matrix is the same as matrix X . Thus, matrix Z will have n rows and $(m+1)$ columns. [An outline of subroutine *BuildZM* is available HERE](#)
- Call Sub *NLRegress*, developed in Problem [2], to calculate the vector of coefficients $\{a\}$
- Calculate the fitted values $y_f = [Z] \cdot \{a\}$. Use subroutine *MultiplyMatrixToVector*(Z, a, y_f). This subroutine was used in Assignment 3.
- Show the table of values of X, y , and y_f
- Show the values of vector $\{a\}$

(a) Solve problem 17.18 - your solution needs to show:

- Table of values x_1, x_2, y, y_f , and a

(b) Show the code developed for this program only, namely, the main program and subroutine *BuildZM*

17.18 Use multiple linear regression to fit

x_1	0	1	1	2	2	3	3	4	4
x_2	0	1	2	1	2	1	2	1	2
y	15.1	17.9	12.7	25.6	20.5	35.1	29.7	45.4	40.2

Compute the coefficients, the standard error of the estimate, and the correlation coefficient.

x_1	x_2	y	y_f
0	0	15.1	14.4609
1	1	17.9	17.7817
1	2	12.7	12.0774
2	1	25.6	26.807
2	2	20.5	21.1026
3	1	35.1	35.8322
3	2	29.7	30.1278
4	1	45.4	44.8574
4	2	40.2	39.153

$a: [14.4609, 9.02522, -5.70435]$

a4p3.cpp:

```
#include "a4p3.hpp"
#include "assign4.hpp"
using namespace assign4;

#include <vector>
#include <iomanip>
#include <functional>

void assign4::Problem3() {
    const_vector y {15.1, 17.9, 12.7, 25.6, 20.5, 35.1, 29.7, 45.4, 40.2};
    const Matrix <double> X {
        {0, 0},
        {1, 1},
        {1, 2},
        {2, 1},
        {2, 2},
        {3, 1},
        {3, 2},
        {4, 1},
        {4, 2},
    };

    bool error = false;
```

```

Matrix<double> Z(X.Rows, X.Cols + 1, [&X](int r, int c) {
    return c == 0 ? 1 : X[r][c - 1];
});

std::vector<double>
    a = NLRegress(Z, y, error),
    yf = Z * a;



std::cout
    << "Problem 2:" << std::endl << std::endl
    << std::setw(3) << "x1"
    << std::setw(4) << "x2"
    << std::setw(6) << "y"
    << std::setw(8) << "yf" << std::endl
    << "-----" << std::endl;

int n = X.Rows;
for (int i = 0; i < n; ++i) {
    std::cout
        << std::setw(2) << X[i][0]
        << std::setw(4) << X[i][1]
        << std::setw(8) << y[i]
        << std::setw(10) << yf[i] << std::endl;
}

std::cout << std::endl
    << "a: " << a << std::endl << std::endl;
}

```

Problem 4

[4] Using your preferred computer language (VBA, C++, etc.) write a main program to perform Newton polynomial interpolation as described in [THIS DOCUMENT](#)  .

CLARIFICATION: You do not need to code the selection of the 5 points needed to interpolate using a 4-th order Newton Interpolating Polynomial. Select your 5 points by hand, and enter those 5 points as your (x,y) data for the interpolation.

(a) Show the code developed in your assignment report.

(b) Use this program to show the Newton interpolated values for Problem 18.6. Show your solution values in your assignment report.

(c) Use this program to show the Newton interpolated values for Problem 20.36. Show your solution values in your assignment report.

18.6 Given the data

x	1	2	3	5	7	8
$f(x)$	3	6	19	99	291	444

Calculate $f(4)$ using Newton's interpolating polynomials of order 1 through 4. Choose your base points to attain good accuracy. What do your results indicate regarding the order of the polynomial used to generate the data in the table?

Part A:

order	$y(x_0)$	error
1	3	9
2	12	30
3	42	6
4	48	0

20.36 You measure the voltage drop V across a resistor for a number of different values of current i . The results are

i	0.25	0.75	1.25	1.5	2.0
V	-0.45	-0.6	0.70	1.88	6.0

Use first- through fourth-order polynomial interpolation to estimate the voltage drop for $i = 1.15$. Interpret your results.

Part B:

order	$y(x_0)$	error
1	-0.45	-0.27
2	-0.72	1.044
3	0.324	0.002112
4	0.326112	0

a4p4.cpp:

```
#include "a4p4.hpp"
#include "assign4.hpp"
using namespace assign4;

#include <math.h>
#include <vector>
#include <iomanip>
#include <iostream>
#include <algorithm>
#include <functional>

//-----+
// std::vector<newt_value> NewtInt
// Return an interpolation from a regression of orders 1 through n
//-----+
struct newt_val { int order; double yint, error; };
std::vector<newt_val> NewtInt(const_vector& x, const_vector& y, int n, double
x0) {
    std::vector<newt_val> set(n);
    Matrix<double> fdd(n, n);

    for (int i = 0; i < n; ++i) {
        set[i].order = i + 1;
        fdd[i][0] = y[i];
    }
}
```

```

    }
    for (int j = 1; j < n; ++j) {
        for (int i = 0; i < n - j; ++i) {
            fdd[i][j] = fdd[i + 1][j - 1] - fdd[i][j - 1];
            fdd[i][j] /= (x[i + j] - x[i]);
        }
    }

    double xterm = 1, yterm;
    set[0].yint = fdd[0][0];

    for (int order = 1; order < n; ++order) {
        xterm *= (x0 - x[order - 1]);
        yterm = set[order - 1].yint + fdd[0][order] * xterm;
        set[order - 1].error = yterm - set[order - 1].yint;
        set[order].yint = yterm;
    }

    return set;
}

//-----+
// std::vector<newt_value> NewtInterp
// Given x-y pairs, select n + 1 optimal points and execute NewtInt
// The assignment description said programming this wasn't required,
// but I prefer this over manually determining the points of interest
//-----+
struct sort_item { int row; double value; };
std::vector<newt_val> NewtInterp(const_vector& x, const_vector& y, int n,
double x0) {
    int count = x.size(), upper = count - 1, lower = 0;
    while (lower + 1 < upper && x[lower + 1] < x0) { ++lower; }
    while (upper - 1 > lower && x[upper - 1] > x0) { --upper; }
    double mid = (upper + lower) / 2.0;

    // Sort [l, r] in ascending order of importance
    // First determined by difference in index from median i
    // Then by difference in sampled x-value from point x0
    auto rank = [&x, x0, mid](const sort_item& l, const sort_item& r) {
        double
            left_dx = abs(l.value - x0),
            right_dx = abs(r.value - x0),
            left_di = abs(mid - l.row),
            right_di = abs(mid - r.row);

        return left_di != right_di ? left_di < right_di : left_dx < right_dx;
    };

    // Sort [l, r] by order of ascending sampled x-values
    // It's expected x and y are initially passed sorted like this
    auto sequential = [](const sort_item& l, const sort_item& r) {
        return l.value < r.value;
    };

    // Sort data by rank
    std::vector<sort_item> diffs;
    for (int i = 0; i < count; ++i) {

```



```

        diffs.push_back({i, x[i]});
    }
    sort(diffs.begin(), diffs.end(), rank);

    /* debugging code
    for (auto& elem : diffs) {
        std::cout << "[" << elem.row << "]" -> " << elem.value << std::endl;
    }
    std::cout << std::endl;/**/

    // Select n + 1 points and order sequentially
    diffs.resize(n + 1);
    sort(diffs.begin(), diffs.end(), sequential);

    /* debugging code
    for (auto& elem : diffs) {
        std::cout << "[" << elem.row << "]" -> " << elem.value << std::endl;
    }
    std::cout << std::endl;/**/

    // Generate reduced vectors to pass to NewtInt
    std::vector<double> x_red(n + 1), y_red(n + 1);
    for (int i = 0; i < n + 1; ++i) {
        x_red[i] = x[diffs[i].row];
        y_red[i] = y[diffs[i].row];

        /* debugging code
        std::cout << x_red[i] << ", " << y_red[i] << std::endl;/**/
    }

    /* debugging code
    std::cout
        << "x_red: " << x_red << std::endl
        << "y_red: " << y_red << std::endl
        << std::endl;/**/

    return NewtInt(x_red, y_red, n, x0);
}

void PrintInterp(const std::vector<newt_val>& values) {
    std::cout
        << std::setw(6) << "order"
        << std::setw(10) << "y(x0)"
        << std::setw(14) << "error" << std::endl
        << "-----" << std::endl;

    for (auto& elem : values) {
        std::cout
            << std::setw(4) << elem.order
            << std::setw(13) << elem.yint
            << std::setw(15) << elem.error << std::endl;
    }
    std::cout << std::endl << std::endl;
}

void assign4::Problem4() {
    /* debugging code

```

```

    const_vector x {1, 4, 6, 5, 3, 1.5, 2.5, 3.5};
    const_vector y {0, 1.3862944, 1.7917595, 1.6094379, 1.0986123,
0.40546411, 0.91629073, 1.2527630};
    PrintInterp(NewtInt(x, y, 8, 2));/**/

// Part A
const_vector x {1, 2, 3, 5, 7, 8};
const_vector y {3, 6, 19, 99, 291, 444};
std::cout << "Part A: " << std::endl << std::endl;
PrintInterp(NewtInterp(x, y, 4, 4));

// Part B
const_vector i {0.25, 0.75, 1.25, 1.5, 2};
const_vector v {-0.45, -0.6, 0.7, 1.88, 6};
std::cout << "Part B: " << std::endl << std::endl;
PrintInterp(NewtInterp(i, v, 4, 1.15));
}

```

Problem 5

[5] : THIS DOCUMENTS shows how to interpolate data using cubic splines in SCILAB . Using a similar procedure, for the "natural" type of splines, solve (a) Problem 18.6 and (b) Problem 20.43 -- For both problems, produce a plot showing the original data given as points and the fitted splines as continuous lines. NOTE: If you prefer to use MATLAB, you can find information on cubic splines [HERE](#) .

18.6 Given the data

x	1	2	3	5	7	8
$f(x)$	3	6	19	99	291	444

Calculate $f(4)$ using Newton's interpolating polynomials of order 1 through 4. Choose your base points to attain good accuracy. What do your results indicate regarding the order of the polynomial used to generate the data in the table?

Part A:

```
>> x = [1,2,3,5,7,8]
```

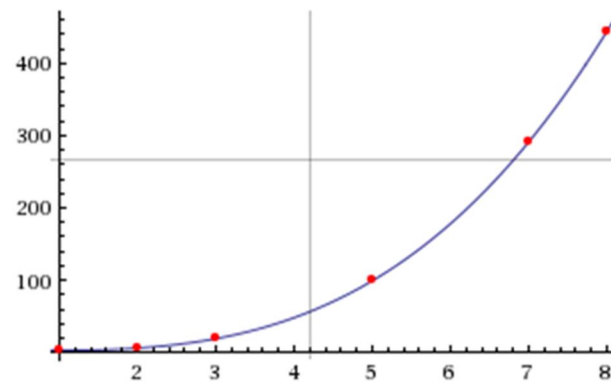
```
x = 1 2 3 5 7 8
```

```
>> y = [3,6,19,99,291,444]
```

```
y = 3 6 19 99 291 444
```

```
>> z = spline(x,y,4)
```

```
z = 48
```



Part B:

```
>> x = [200,250,300,375,425,475,600]
```

```
x = 200 250 300 375 425 475 600
```

```
>> y = [7.5,8.6,8.7,10,11.3,12.7,15.3]
```

```
y = 7.5000 8.6000 8.7000 10.0000 11.3000 12.7000 15.3000
```

```
>> z = spline(x,y,400)
```

```
z = 10.6310
```

