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/Assimilater/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):

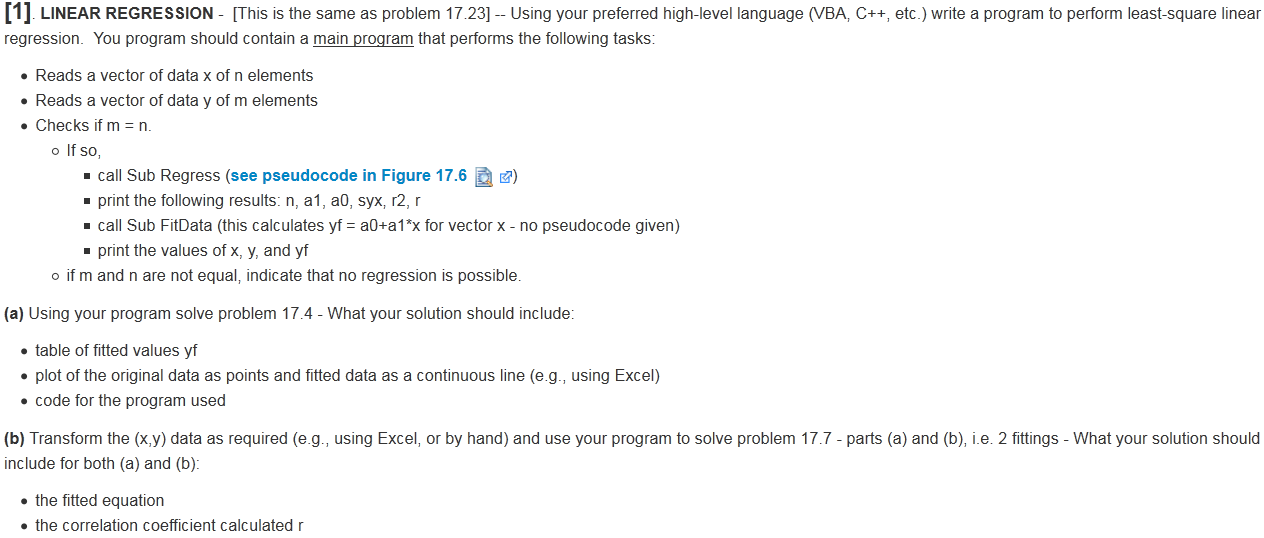
#pragma once

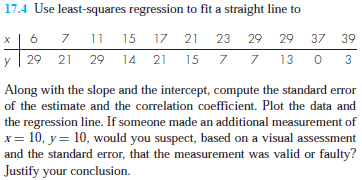
namespace assign4 {

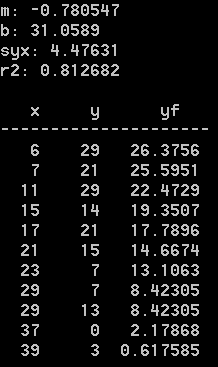
void ProblemX();

}

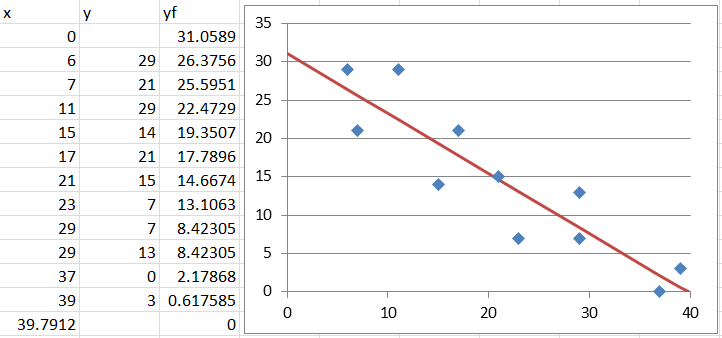
Problem 1



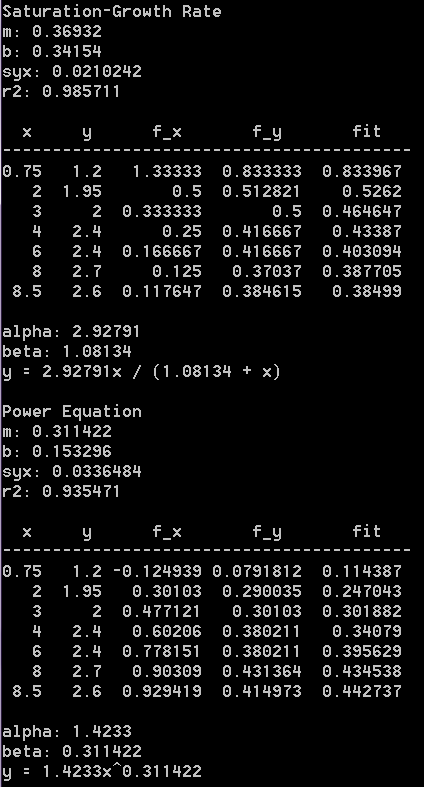
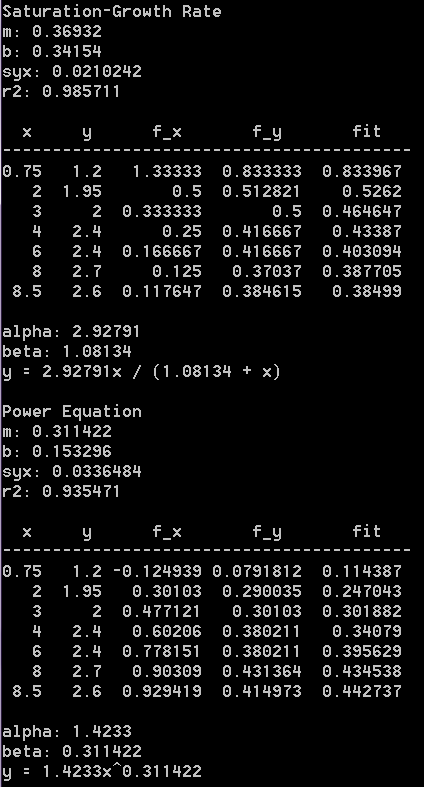




Part A:



Part B:

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

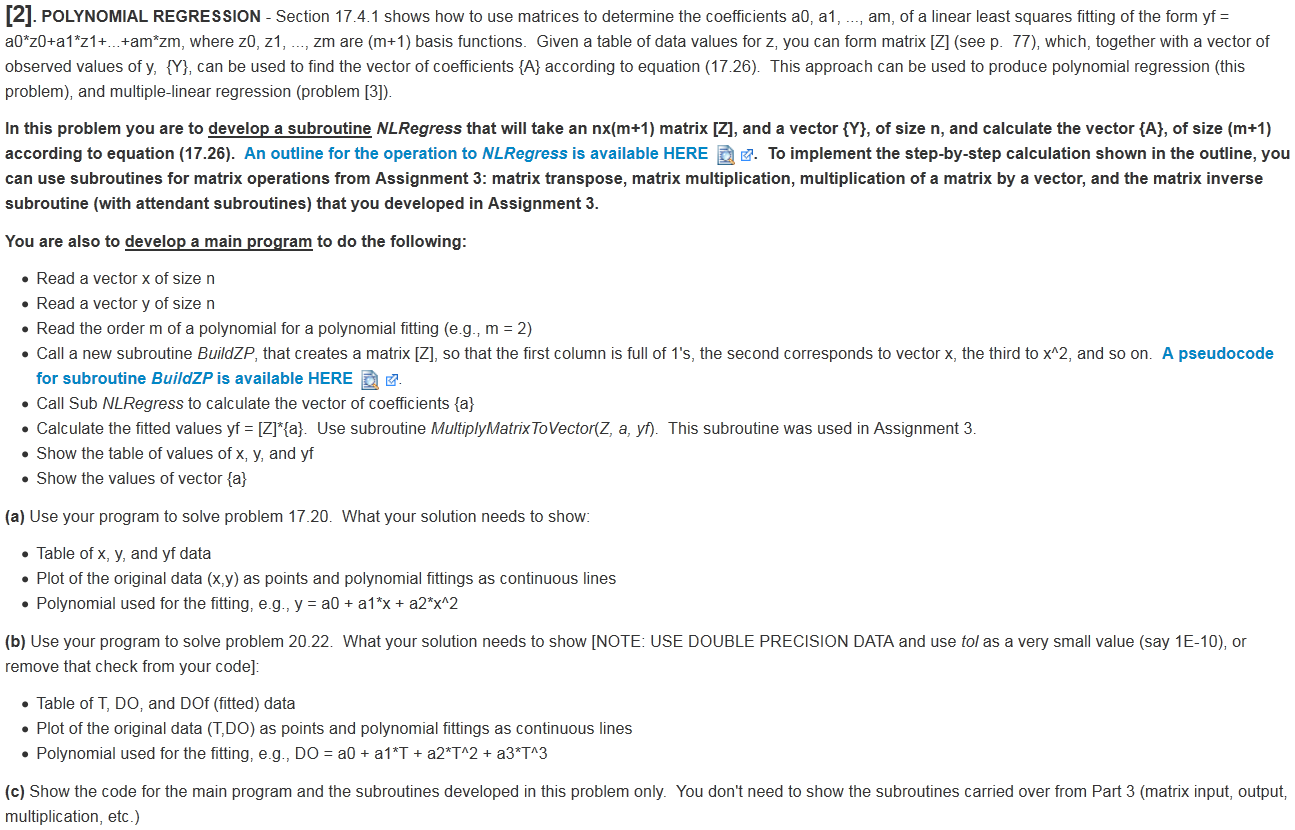
<< "beta: " << beta << std::endl

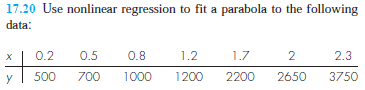
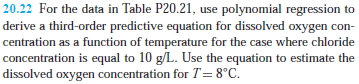
<< "y = " << alpha << "x^" << beta << std::endl

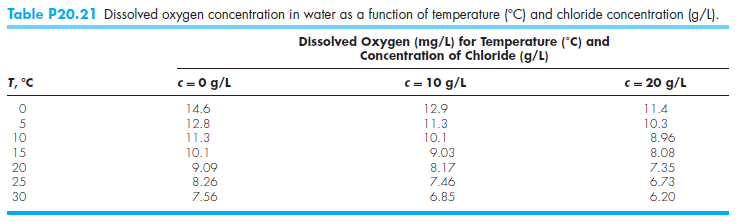
<< std::endl;

}

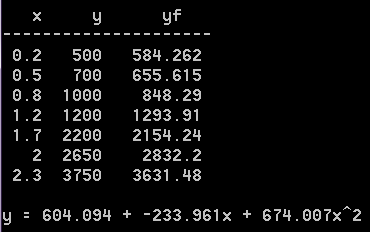
Problem 2

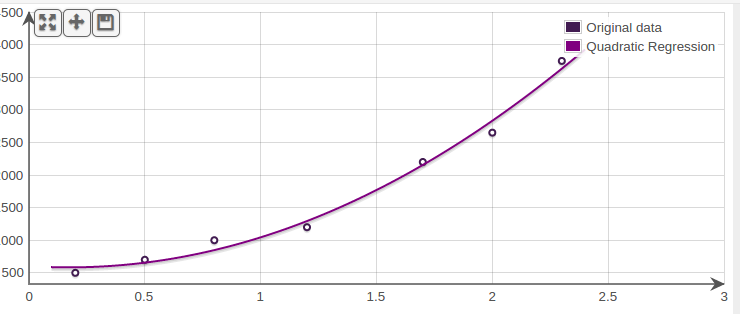


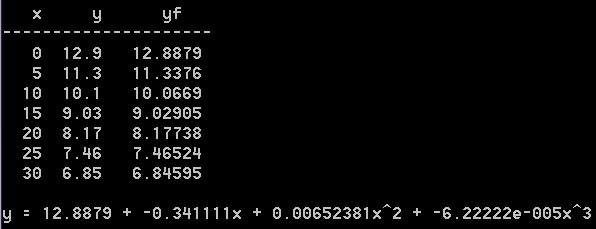


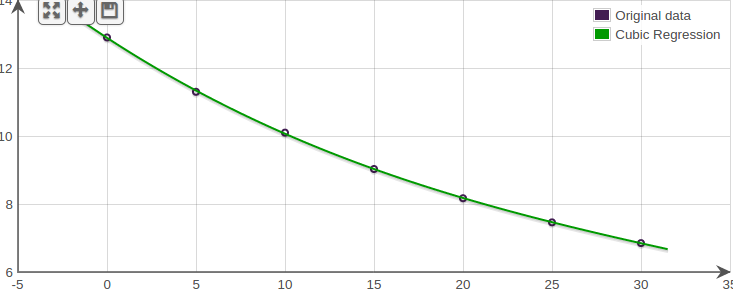
Part a:





Part b:





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};

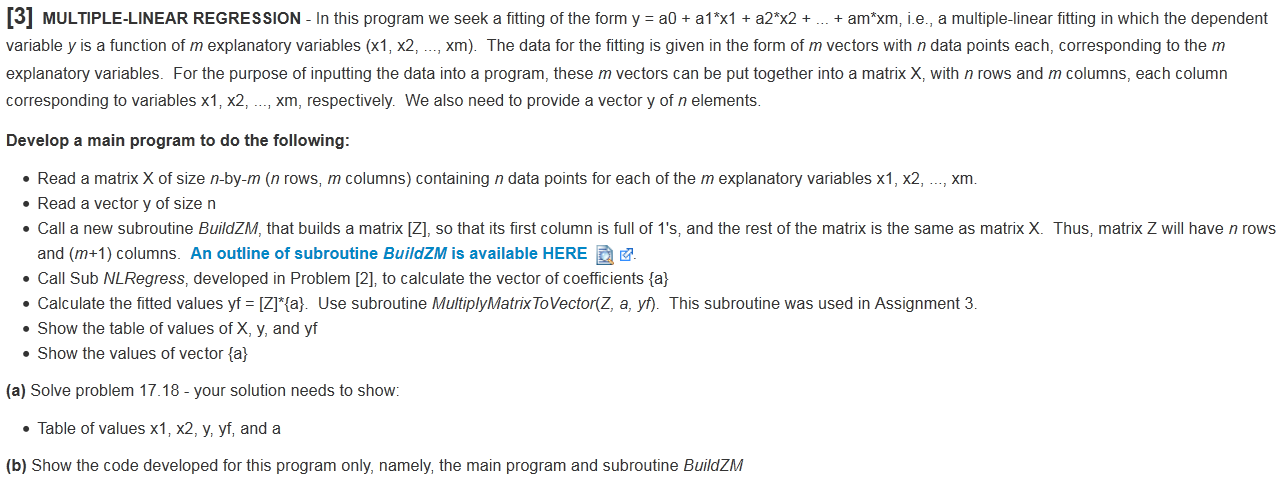
y = {12.9, 11.3, 10.1, 9.03, 8.17, 7.46, 6.85};

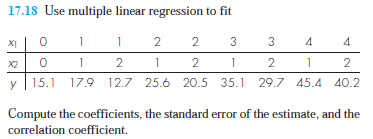
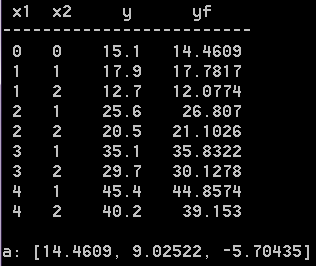
std::cout << "Part B:" << std::endl << std::endl;

PrintFit(x, y, 3);

}

Problem 3



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;

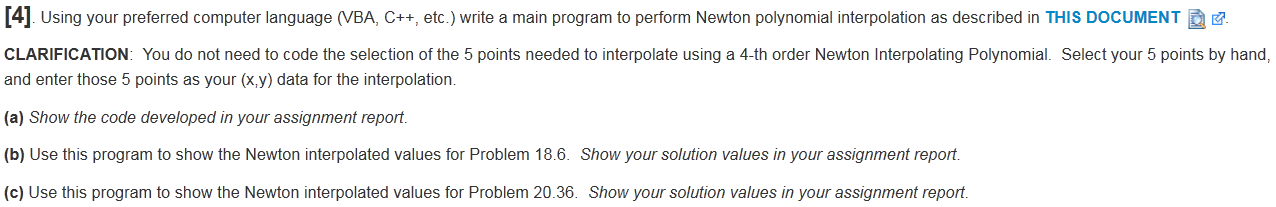
}

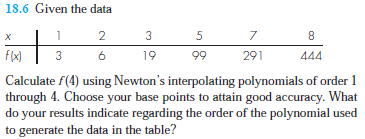
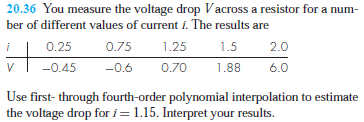
std::cout << std::endl

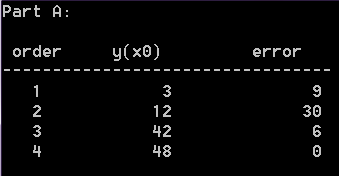
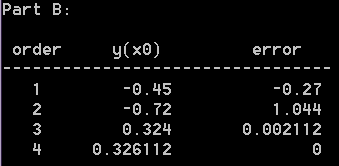
<< "a: " << a << std::endl << std::endl;

}

Problem 4



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 regession 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 deteremining 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};

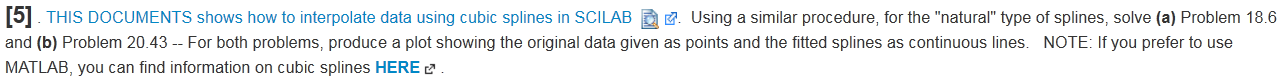
const\_vector v {-0.45, -0.6, 0.7, 1.88, 6};

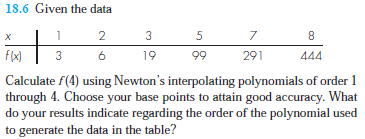
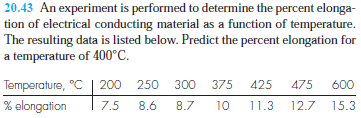
std::cout << "Part B: " << std::endl << std::endl;

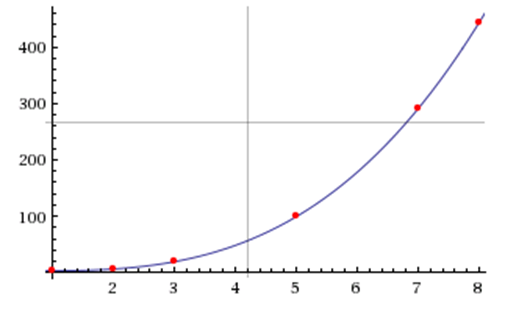
PrintInterp(NewtInterp(i, v, 4, 1.15));

}

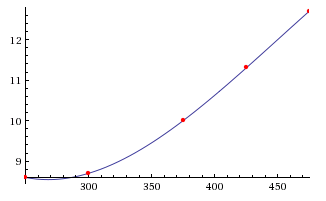
Problem 5



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