Common Code

**main.cpp:**

//--------------------------------------------------------------------------+

// John Call - A01283897 |

// Driver for ENGR 2450 homework |

//--------------------------------------------------------------------------+

#include "assign3\assign3.h"

#include <iostream>

int main() {

assign3::main();

system("pause");

return 0;

}

**assign3.h:**

#pragma once

namespace assign3 {

enum MatrixError {

GOOD = 0,

INV\_DIM = 1,

SINGULAR = 2

};

void main();

}

**assign3.cpp:**

#include "assign3.h"

#include "a3p1.h"

#include "a3p3.h"

void assign3::main() {

assign3::Problem1();

assign3::Problem3();

assign3::Problem4();

}

**matrix.cpp:**

#include "matrix.h"

Matrix<double> Identity(int n) {

Matrix<double> a(n, n, 0);

for (int i = 0; i < n; ++i) {

a[i][i] = 1;

}

return a;

}

**matrix.h:**

#pragma once

#include <vector>

#include <complex>

#include <iostream>

#include <type\_traits>

#include <initializer\_list>

template <typename T>

class Matrix {

private:

T\*\* \_array;

static T \_default;

public:

// Important properties

int Rows, Cols;

bool isSingular() { return \_array == nullptr; };

static void setDefault(T val) { \_default = val; };

// public access to \_array

T\* operator [](int r) { return \_array[r]; };

// Common operations

Matrix<T> Transpose() { return Transpose(\*this); };

T Trace() { return Trace(\*this); };

// Constructors and destructor

Matrix(int m, int n) {

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] = \_default;

}

}

};

// Allow initialized values

Matrix(int m, int n, T val) {

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] = val;

}

}

};

// Allow for C++11 initializer\_list

Matrix(std::initializer\_list<std::initializer\_list<T>> s) {

// Take advantage of vector handling initalizer\_list

std::vector<std::initializer\_list<T>> init = s;

std::vector<std::vector<T>> data;

for (auto i = init.begin(); i != init.end(); ++i) {

data.push\_back(std::vector<T>(\*i));

}

Rows = data.size();

Cols = data[0].size();

\_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] = data[i][j];

}

}

};

// Override default operator=, constructors, and destructor

Matrix<T>& operator=(const Matrix<T> &a) {

clean();

copy(a);

return \*this;

};

Matrix() {

Rows = 0;

Cols = 0;

\_array = nullptr;

};

Matrix(const Matrix<T> &a) { copy(a); }

~Matrix() { clean(); };

private:

void clean() {

for (int i = 0; i < Cols; ++i) {

delete \_array[i];

}

delete \_array;

\_array = nullptr;

};

void copy(const Matrix<T> &a) {

this->Rows = a.Rows;

this->Cols = a.Cols;

this->\_array = new T\*[this->Rows];

for (int i = 0; i < this->Rows; ++i) {

this->\_array[i] = new T[this->Cols];

for (int j = 0; j < this->Cols; ++j) {

this->\_array[i][j] = a.\_array[i][j];

}

}

};

public:

// friend templates

template <typename f\_T>

friend Matrix<f\_T> Transpose(const Matrix<f\_T>&);

template <typename f\_T>

friend f\_T Trace(const Matrix<f\_T>&, bool&);

template <typename f\_T>

friend std::ostream& operator<<(std::ostream&, const Matrix<f\_T>&);

template <typename f\_T>

friend std::vector<f\_T> operator\*(const Matrix<f\_T>&, const std::vector<f\_T>&);

template <typename f\_T1, typename f\_T2>

friend Matrix<f\_T1> operator\*(const Matrix<f\_T1>&, const Matrix<f\_T2>&);

template <typename f\_T1, typename f\_T2>

friend Matrix<f\_T1>& operator+=(Matrix<f\_T1>&, const Matrix<f\_T2>&);

};

template<typename T>

T Matrix<T>::\_default = 0;

// common operations and helpers

template <typename f\_T>

Matrix<f\_T> Transpose(const Matrix<f\_T> &a) {

Matrix<f\_T> trans(a.Cols, a.Rows);

for (int i = 0; i < a.Rows; ++i) {

for (int j = 0; j < a.Cols; ++j) {

trans[i][j] = a.\_array[j][i];

}

}

return trans;

}

template <typename f\_T>

f\_T Trace(const Matrix<f\_T> &a, bool& error) {

f\_T sum = a.\_default;

if (a.Rows != a.Cols) {

error = true;

} else {

error = false;

for (int i = 0; i < a.Rows; ++i) {

sum += a.\_array[i][i];

}

}

return sum;

}

// iostream handlers

template <typename T>

std::istream& operator>>(std::istream &in, Matrix<T> &obj) {

int m, n;

in >> m;

in >> n;

obj = Matrix<T>(m, n);

for (int i = 0; i < m; ++i) {

for (int j = 0; j < n; ++j) {

in >> obj[i][j];

}

}

return in;

}

template <typename f\_T>

std::ostream& operator<<(std::ostream &out, const Matrix<f\_T> &obj) {

out << std::endl;

for (int i = 0; i < obj.Rows; ++i) {

out << "[";

for (int j = 0; j < obj.Cols; ++j) {

out << obj.\_array[i][j];

if (j < obj.Cols - 1) {

out << ", ";

}

}

out << "]" << std::endl;

}

out << std::endl;

return out;

}

template <typename T>

std::ostream& operator<<(std::ostream &out, const std::vector<T> &obj) {

out << "[";

for (auto i = obj.begin(); i < obj.end(); ++i) {

out << \*i;

if (i < obj.end() - 1) {

out << ", ";

}

}

out << "]";

return out;

}

// scalar multiplication

template <typename T, typename N,

typename std::enable\_if<std::is\_arithmetic<N>::value>::type\* = nullptr>

Matrix<T> operator\*(const N &c, const Matrix<T> &a) {

Matrix<T> b(a);

for (int i = 0; i < b.Rows; ++i) {

for (int j = 0; j < b.Cols; ++j) {

b[i][j] \*= c;

}

}

return b;

}

template <typename T, typename N,

typename std::enable\_if<std::is\_arithmetic<N>::value>::type\* = nullptr>

Matrix<T> operator\*(const Matrix<T> &a, const N &c) { return c \* a; }

// complex scalar multiplication

template <typename T, typename N, typename C,

typename std::enable\_if<std::is\_arithmetic<N>::value>::type\* = nullptr,

typename std::enable\_if<std::is\_same<C, std::complex<N>>::value>::type\* = nullptr>

Matrix<T> operator\*(const C &c, const Matrix<T> &a) {

Matrix<T> b(a);

for (int i = 0; i < b.Rows; ++i) {

for (int j = 0; j < b.Cols; ++j) {

b[i][j] \*= c;

}

}

return b;

}

template <typename T, typename N, typename C,

typename std::enable\_if<std::is\_arithmetic<N>::value>::type\* = nullptr,

typename std::enable\_if<std::is\_same<C, std::complex<N>>::value>::type\* = nullptr>

Matrix<T> operator\*(const Matrix<T> &a, const C &c) { return c \* a; }

// vector multiplication

template <typename f\_T>

std::vector<f\_T> operator\*(const Matrix<f\_T> &a, const std::vector<f\_T> &x) {

std::vector<f\_T> b(a.Rows);

for (int i = 0; i < a.Rows; ++i) {

b[i] = a.\_default;

for (int j = 0; j < a.Cols; ++j) {

b[i] += a.\_array[i][j] \* x[j];

}

}

return b;

}

// matrix multiplication

template <typename T1, typename T2>

Matrix<T1>& operator\*=(Matrix<T1> &a, const Matrix<T2> &b) {

// Matrix multiplication is more complicated than numeric addition

// Because 'a' would need to be resized, it is more efficient

// To use operator\* here than to use '\*=' in 'operator\*'

return a = a \* b;

}

template <typename f\_T1, typename f\_T2>

Matrix<f\_T1> operator\*(const Matrix<f\_T1> &a, const Matrix<f\_T2> &b) {

Matrix<f\_T1> c(a.Rows, b.Cols, a.\_default);

for (int i = 0; i < a.Rows; ++i) {

for (int j = 0; j < a.Cols; ++j) { // a.Cols = b.Rows (or error)

for (int k = 0; k < b.Cols; ++k) {

c[i][k] += a.\_array[i][j] \* b.\_array[j][k];

}

}

}

return c;

}

// matrix addition

template <typename f\_T1, typename f\_T2>

Matrix<f\_T1>& operator+=(Matrix<f\_T1> &a, const Matrix<f\_T2> &b) {

for (int i = 0; i < a.Rows; ++i) {

for (int j = 0; j < a.Cols; ++j) {

a.\_array[i][j] += b.\_array[i][j];

}

}

return a;

}

template <typename T1, typename T2>

Matrix<T1> operator+(const Matrix<T1> &a, const Matrix<T2> &b) {

Matrix<T1> c(a);

c += b;

return c;

}

// matrix subtraction

template <typename f\_T1, typename f\_T2>

Matrix<f\_T1>& operator-=(Matrix<f\_T1> &a, const Matrix<f\_T2> &b) {

for (int i = 0; i < a.Rows; ++i) {

for (int j = 0; j < a.Cols; ++j) {

a.\_array[i][j] -= b.\_array[i][j];

}

}

return a;

}

template <typename T1, typename T2>

Matrix<T1> operator-(const Matrix<T1> &a, const Matrix<T2> &b) {

Matrix<T1> c(a);

c -= b;

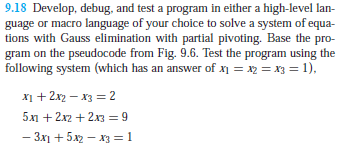
return c;

}

// prototypes defined in cpp file

Matrix<double> Identity(int);

Problem 1

**PROBLEM [1]**. Solve problem 9.18 (p. 273) in the Chapra & Canale textbook -- write the required program in your favorite computer language, and show your solution and your code.

Output:



**a3p1.h:**

#pragma once

#include <vector>

#include "assign3.h"

#include "../shared/matrix.h"

namespace assign3 {

namespace p1 {

void Substitute(Matrix<double>&, std::vector<double>&, std::vector<double>&);

void Pivot(Matrix<double>&, std::vector<double>&, std::vector<double>&, int);

void Eliminate(Matrix<double>&, std::vector<double>&, std::vector<double>&, double, MatrixError&);

std::vector<double> Gauss(Matrix<double>, std::vector<double>&, double, MatrixError&);

}

void Problem1();

}

**a3p1.cpp:**

#include "a3p1.h"

#include <iostream>

#include <iomanip>

#include <cmath>

using namespace assign3::p1;

void assign3::p1::Substitute(

Matrix<double> &a,

std::vector<double> &b,

std::vector<double> &x)

{

x[a.Rows - 1] = b[a.Rows - 1] / a[a.Rows - 1][a.Cols - 1];

for (int i = a.Rows - 2; i > -1; --i) {

double sum = 0;

for (int j = i + 1; j < a.Rows; ++j) {

sum += a[i][j] \* x[j];

}

x[i] = (b[i] - sum) / a[i][i];

}

}

void assign3::p1::Pivot(

Matrix<double> &a,

std::vector<double> &b,

std::vector<double> &s,

int k)

{

int p = k;

double temp;

double max = abs(a[k][k] / s[k]);

for (int i = k + 1; i < a.Rows; ++i) {

temp = abs(a[i][k] / s[i]);

if (temp > max) {

max = temp;

p = i;

}

}

if (p != k) {

for (int j = k; j < a.Rows; ++j) {

temp = a[p][j];

a[p][j] = a[k][j];

a[k][j] = temp;

}

temp = b[p];

b[p] = b[k];

b[k] = temp;

temp = s[p];

s[p] = s[k];

s[k] = temp;

}

}

void assign3::p1::Eliminate(

Matrix<double> &a,

std::vector<double> &b,

std::vector<double> &s,

double tolerance,

MatrixError &error)

{

int k = 0;

for (; k < a.Rows - 1; ++k) {

Pivot(a, b, s, k);

if (abs(a[k][k] / s[k]) < tolerance) {

// Make sure the diagonals are all non-zero

error = MatrixError::SINGULAR;

return;

}

for (int i = k + 1; i < a.Rows; ++i) {

double factor = a[i][k] / a[k][k];

for (int j = k + 1; j < a.Rows; ++j) {

a[i][j] -= factor \* a[k][j];

}

b[i] -= factor \* b[k];

}

}

// Not sure what exactly this is checking

if (abs(a[a.Rows - 1][a.Rows - 1]) < tolerance) {

error = MatrixError::SINGULAR;

}

}

std::vector<double> assign3::p1::Gauss(

Matrix<double> a,

std::vector<double> &b,

double tolerance,

MatrixError& error)

{

if (a.Rows != a.Cols || a.Rows != b.size()) {

error = MatrixError::INV\_DIM;

return b;

}

double temp;

std::vector<double> x(a.Rows), s(a.Rows);

// Determine the largest coefficient of a in a given column

for (int i = 0; i < a.Rows; ++i) {

s[i] = 0;

for (int j = 0; j < a.Cols; ++j) {

temp = abs(a[i][j]);

if (temp > s[j]) {

s[i] = temp;

}

}

}

// I have no idea what this does anymore; it's someone else's code

Eliminate(a, b, s, tolerance, error);

if (error == MatrixError::GOOD) {

Substitute(a, b, x);

}

return x;

}

void assign3::Problem1() {

Matrix<double> a{

{1, 2, -1},

{5, 2, 2},

{-3, 5, -1},

};

std::vector<double> b{2, 9, 1};

std::vector<double> x;

MatrixError error = MatrixError::GOOD;

x = Gauss(a, b, .0001, error);

// Output the result of Gaussian elimination with partial pivoting

if (error == MatrixError::GOOD) {

std::cout << "The result is: " << x << std::endl;

} else if (error == MatrixError::INV\_DIM) {

std::cout << "The matrix a and the solution vector b have invalid dimensions..." << std::endl;

} else if (error == MatrixError::SINGULAR) {

std::cout << "The matrix a is singular, and ax = b has no solution..." << std::endl;

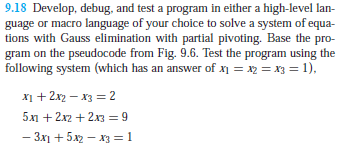
} else {

std::cout << "An unkown error has occured..." << std::endl;

}

}

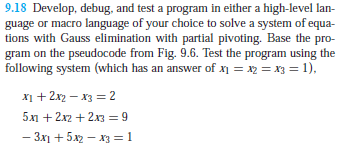
Problem 2

**PROBLEM [2]**. Solve the system of problem 9.18 (p. 273) in the Chapra & Canale textbook using matrices, the inverse matrix, and matrix multiplication in SCILAB, MATLAB, Excel, SMath Studio, or Maxima. Compare your results with those of problem [1]. They should be the same. Recall, the matrix equation is [A]{x} = {b}, so {x} = inv(A)\*{b}.

**matlab:**

>> a = [  
   [1,2,-1]  
   [5,2,2]  
   [-3,5,-1]  
];  
>> b = [  
   [2]  
   [9]  
   [1]  
];  
>> inv(a)\*b  
  
ans =  
  
    1.0000  
    1.0000  
    1.0000  
  
>>

Oh look, it’s the same ☺

Problem 3

**PROBLEM [3]**. Write the code for solving Problem 10.18, and solve the linear system of Problem [1] using this code.

Output:

**a3p3.h:**

#pragma once

#include <vector>

#include "assign3.h"

#include "../shared/matrix.h"

namespace assign3 {

namespace p3 {

void Substitute(Matrix<double>&, std::vector<double>&, std::vector<double>&, std::vector<int>&);

void Pivot(Matrix<double>&, std::vector<double>&, std::vector<int>&, int);

void Decompose(Matrix<double>&, std::vector<double>&, std::vector<int>&, double, MatrixError&);

std::vector<double> LuDecomp(Matrix<double>, std::vector<double>&, double, MatrixError&);

}

void Problem3();

}

**a3p3.cpp:**

#include "a3p3.h"

#include <iostream>

#include <iomanip>

#include <cmath>

#include "../shared/matrix.h"

using namespace assign3::p3;

void assign3::p3::Substitute(

Matrix<double> &a,

std::vector<double> &b,

std::vector<double> &x,

std::vector<int> &o)

{

for (int i = 1; i < a.Rows; ++i) {

double sum = b[o[i]];

for (int j = 0; j <= i - 1; ++j) {

sum -= a[o[i]][j] \* b[o[j]];

}

b[o[i]] = sum;

}

x[a.Rows - 1] = b[o[a.Rows - 1]] / a[o[a.Rows - 1]][a.Rows - 1];

for (int i = a.Rows - 2; i >= 0; --i) {

double sum = 0;

for (int j = i + 1; j < a.Rows; ++j) {

sum += a[o[i]][j] \* x[j];

}

x[i] = (b[o[i]] - sum) / a[o[i]][i];

}

}

void assign3::p3::Pivot(

Matrix<double> &a,

std::vector<double> &s,

std::vector<int> &o,

int k)

{

int p = k;

double temp;

double max = abs(a[o[k]][k] / s[o[k]]);

for (int i = k + 1; i < a.Rows; ++i) {

temp = abs(a[o[i]][k] / s[o[i]]);

if (temp > max) {

max = temp;

p = i;

}

}

int t\_int = o[p];

o[p] = o[k];

o[k] = t\_int;

}

void assign3::p3::Decompose(

Matrix<double> &a,

std::vector<double> &s,

std::vector<int> &o,

double tolerance,

MatrixError &error)

{

for (int i = 0; i < a.Rows; ++i) {

double temp;

o[i] = i;

s[i] = abs(a[i][0]);

for (int j = 1; j < a.Rows; ++j) {

temp = abs(a[i][j]);

if (temp > s[i]) {

s[i] = temp;

}

}

}

int k = 0;

for (; k < a.Rows - 1; ++k) {

Pivot(a, s, o, k);

if (abs(a[o[k]][k] / s[o[k]]) < tolerance) {

error = MatrixError::SINGULAR;

// print a[o[k]][k] / s[o[k]]

return;

}

for (int i = k + 1; i < a.Rows; ++i) {

double factor = a[o[i]][k] / a[o[k]][k];

a[o[i]][k] = factor;

for (int j = k + 1; j < a.Rows; ++j) {

a[o[i]][j] -= factor \* a[o[k]][j];

}

}

}

// Not sure what exactly this is checking

if (abs(a[o[k]][k] / s[o[k]]) < tolerance) {

error = MatrixError::SINGULAR;

// print a[o[k]][k] / s[o[k]]

}

}

std::vector<double> assign3::p3::LuDecomp(

Matrix<double> a,

std::vector<double> &b,

double tolerance,

MatrixError& error)

{

if (a.Rows != a.Cols || a.Rows != b.size()) {

error = MatrixError::INV\_DIM;

return b;

}

std::vector<double> x(a.Rows), s(a.Rows);

std::vector<int> o(a.Rows);

// I have no idea what this does anymore; it's someone else's code

Decompose(a, s, o, tolerance, error);

if (error == MatrixError::GOOD) {

Substitute(a, b, x, o);

}

return x;

}

void assign3::Problem3() {

Matrix<double> a{

{1, 2, -1},

{5, 2, 2},

{-3, 5, -1},

};

std::vector<double> b{2, 9, 1};

std::vector<double> x;

MatrixError error = MatrixError::GOOD;

x = LuDecomp(a, b, .0001, error);

// Output the result of LU Decomposition

if (error == MatrixError::GOOD) {

std::cout << "The result is: " << x << std::endl;

} else if (error == MatrixError::INV\_DIM) {

std::cout << "The matrix a and the solution vector b have invalid dimensions..." << std::endl;

} else if (error == MatrixError::SINGULAR) {

std::cout << "The matrix a is singular, and ax = b has no solution..." << std::endl;

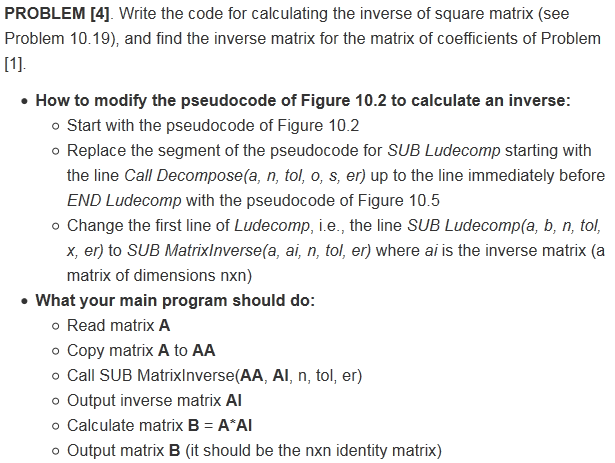
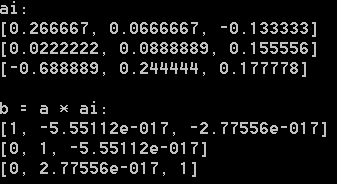
} else {

std::cout << "An unkown error has occured..." << std::endl;

}

}

Problem 4

 Output:

**a3p4.h:**

#pragma once

#include <vector>

#include "assign3.h"

#include "../shared/matrix.h"

namespace assign3 {

namespace p4 {

Matrix<double> MatrixInverse(Matrix<double>, double, MatrixError&);

}

void Problem4();

}

**a3p4.cpp:**

#include "a3p4.h"

#include <iostream>

#include <iomanip>

#include <cmath>

#include "../shared/matrix.h"

// No need to duplicate code

#include "a3p3.h"

using namespace assign3::p4;

Matrix<double> assign3::p4::MatrixInverse(

Matrix<double> a, // copy by value

double tolerance,

MatrixError& error)

{

if (a.Rows != a.Cols) {

error = MatrixError::INV\_DIM;

return a;

}

int n = a.Rows;

Matrix<double> ai(a);

std::vector<double> b(n), x(n), s(n);

std::vector<int> o(n);

// I have no idea what this does anymore; it's someone else's code

assign3::p3::Decompose(a, s, o, tolerance, error);

if (error == MatrixError::GOOD) {

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

b[j] = i == j ? 1 : 0;

}

assign3::p3::Substitute(a, b, x, o);

for (int j = 0; j < n; ++j) {

ai[j][i] = x[j];

}

}

}

return ai;

}

void assign3::Problem4() {

Matrix<double> a{

{1, 2, -1},

{5, 2, 2},

{-3, 5, -1},

};

MatrixError error = MatrixError::GOOD;

Matrix<double> ai = MatrixInverse(a, .0001, error);

Matrix<double> b = a \* ai;

// Output the inverse and product of the two matricies (identity)

if (error == MatrixError::GOOD) {

std::cout << "ai: " << ai << "b = a \* ai:" << b;

} else if (error == MatrixError::INV\_DIM) {

std::cout << "The matrix a and the solution vector b have invalid dimensions..." << std::endl;

} else if (error == MatrixError::SINGULAR) {

std::cout << "The matrix a is singular, and ax = b has no solution..." << std::endl;

} else {

std::cout << "An unkown error has occured..." << std::endl;

}

}