HW10 - Math 5610

Andrew Sheridan

December 13, 2016

Preface

The code used in the following problems depends on various header files for structures such as Matrices and Vectors. I will include these files in the appendix at the end of this document. Every method called by the functions written in these problems will be included on a per-problem basis.

Problem 2

My first test matrix was a 2x2 matrix found on the internet, a matrix whose inverse was already given. Once the inverse calculation was working correctly, I wrote the ConditionNumber function, and tested it on a 4x4 SPD matrix as well as a 4x4 random matrix. The SPD matrix had a condition number close to 1, where the condition number of the random matrix was much larger.

I then tested the algorithm on matrices from size 16 to 256, as instructed. I used SPD matrices to assure that the condition number stayed close to 1, which it did. On random matrices of that size, the condition numbers were in the hundreds or thousands.

```
#include "Matrix.h"
#include "MatrixFactory.h"
#include "MatrixOperations.h"
#include "Vector.h"
#include "Error.h"
#include "Complex.h"

int main() {
  Matrix A(2);
  A[0][0] = 4;
  A[0][1] = 7;
  A[1][0] = 2;
  A[1][1] = 6;

  Matrix AI = Inverse(A);
  std::cout << "Matrix A:" << std::endl;
  A.Print();
  std::cout << "Matrix A Inverse:" << std::endl;
  AI.Print();

  double conditionNumber = ConditionNumber(A);
  std::cout << "Condition number of A: " << conditionNumber << std::↩
      endl;

  A = MatrixFactory::Instance()->Random(4, 4);
  conditionNumber = ConditionNumber(A);

```

```
  std::cout << "Condition number of the following matrix: " << ↩
      conditionNumber << std::endl;
  A.Print();

  A = MatrixFactory::Instance()->SPD(4);
  conditionNumber = ConditionNumber(A);

  std::cout << "Condition number of the following matrix: " << ↩
      conditionNumber << std::endl;
  A.Print();

  for (int i = 16; i <= 256; i *= 2) {
    A = MatrixFactory::Instance()->SPD(i);
    conditionNumber = ConditionNumber(A);
    std::cout << "Condition number of matrix size " << i << ": " << ↩
        conditionNumber << std::endl;
  }

  return 0;
}
```

```
Matrix A:
4           7
2           6

Matrix A Inverse:
0.6         -0.7
-0.2        0.4

Condition number of A: 14.3
Condition number of the following matrix: 26.1067
0.712955    0.428471    0.690885    0.71915
0.491119    0.780028    0.410924    0.579694
0.139951    0.401018    0.627317    0.324151
0.244759    0.694755    0.593902    0.631792

Condition number of the following matrix: 1.09246
40.713      0.428471    0.690885    0.71915
0.428471    40.4911     0.780028    0.410924
0.690885    0.780028    40.5797     0.139951
0.71915     0.410924    0.139951    40.401

Condition number of matrix size 16: 1.12642
Condition number of matrix size 32: 1.11086
Condition number of matrix size 64: 1.11418
Condition number of matrix size 128: 1.11113
Condition number of matrix size 256: 1.1087
```

```
///Computes the inifinity norm of an n by n matrix A
double Matrix::InfinityNorm() {
  double rowMax = 0;
  for (unsigned j = 0; j < columns; j++) {
    double rowSum = 0;
    for (unsigned i = 0; i < rows; i++) {
      rowSum += std::abs(entries[i][j]);
```

```
    }
    if (rowSum > rowMax)
      rowMax = rowSum;
  }
  return rowMax;
}

///Estimates the inverse of matrix A by LU Factorization and Forward/Back Substitution
Matrix Inverse(Matrix A) {
  Matrix* LU = LUFactorization(A);
  Matrix L = LU[0];
  Matrix U = LU[1];
  Matrix I = MatrixFactory::Instance()->Identity(L.GetRows(), L.GetColumns());
  Matrix G(L.GetColumns());

  //Calculate the columns of G (currently stored as the rows of G
  for (int i = 0; i < G.GetColumns(); i++) {
    G[i] = BackSubstitution(U, ForwardSubstitution(L, I[i]));
  }
  return G.Transpose(); //Transpose G so the rows become the columns
}

///Multiplies a matrix A by matrix B
Matrix operator* (Matrix A, Matrix B) {
  if (A.columns != B.rows) throw "Incompatible sizes";

  Matrix matrix(A.rows, B.columns);
  Matrix bTranspose = B.Transpose();

  for (unsigned i = 0; i < A.rows; i++) {
    for (unsigned j = 0; j < B.columns; j++) {
      //matrix[i][j] = DotProduct(A[i], bTranspose[j], A.columns);
      matrix[i][j] = A[i] * bTranspose[j];
    }
  }
  return matrix;
}

///Estimates the condition number of a matrix A using the Infinity Norm
double ConditionNumber(Matrix A){
  double aNorm = A.InfinityNorm();
  Matrix aInverse = Inverse(A);
  double aInverseNorm = aInverse.InfinityNorm();
  return aNorm * aInverseNorm;
}
```

Problem 3

To test my implementation of the hybrid method (using secant and bisection methods), I chose functions $f(x)$ and $f'(x)$ which I already knew the solution to ($\sqrt(2)$). I then used relative and absolute error to see how close the approximation came. I also included an output statement which would indicate how many iterations it took to switch to the secant method. The functions $f(x)$ and $f'(x)$ converged quickly using bisection, and after 4 iterations, secant method took over.

```cpp
// Andrew Sheridan
//Math 5610
//Written in C++
//main.cpp
#include "ApproximationMethods.h"

//A test function, f(x)
double f(double x) {
  return std::pow(x, 2) - 2;
}

// A test function, f'(x), to be paired with f(x)
double df(double x) {
  return 2 * x;
}

int main() {
  //Using 1 and 10 as starting points, 4 iterations of bisection per loop, 100 max interations, and 10^-8 as tolerance.
  double result = secant_hybrid_method(f, df, 1, 10, 4, 100, std::pow(10, -8));
  std::cout << "Result of secant hybrid method: " << result << std::endl;
  std::cout << "Relative error: " << realRelative(std::sqrt(2), result) << std::endl;
  std::cout << "Absolute error: " << realAbsolute(std::sqrt(2), result) << std::endl;

  return 0;
}
```

```
Switching to secant method after 4 iterations.
Result of secant hybrid method: 1.41421
Relative error: 1.13047e-14
Absolute error: 1.59872e-14
```

```cpp
///The Hybrid Method, which executes the bisection method until secant method starts to converge
///f: The function f, which takes a double and returns a double
///    Input: double
///    Output: double
///df: The derivative of function f
///    Input: double
///    Output: double
///a: Left bound of initial guess
///b: Right bound of initial guess
///bisection_iterations: The number of bisections executed before attempting secant method
///tol: The algorithm's tolerance
///max_iterations: The maximimum number of loops before exit
```

```cpp
double secant_hybrid_method(double(*f)(double), double(*df)(double), double a, double b, unsigned int bisection_iterations, unsigned int max_iterations, double tol) {

  double fa = f(a);
  double fb = f(b);
  // Validate input
  if (a >= b || (fa * fb >= 0) || tol <= 0 || bisection_iterations < 1 || max_iterations < 1)
  {
    std::cout << "This input is not valid." << std::endl;
    return 0;
  }

  unsigned int totalIterationCount = 0;
  unsigned int bisectionIterationCount = 0;
  bool useSecant = false;
  double p;
  while (totalIterationCount < max_iterations && !useSecant && (b - a) > tol) {
    p = (a + b) / 2;
    double fp = f(p);
    if (fa * fp < 0) {
      b = p;
      fb = fp;
    }
    else {
      a = p;
      fa = fp;
    }

    bisectionIterationCount++;
    totalIterationCount++;

    // If we've done the set number of bisections, try secant method
    if (bisectionIterationCount == bisection_iterations) {
      double secantResult = secant_method(f, a, b, tol, 1);
      // If secant method was more efficient than bisection, start using secant method.
      if (std::abs(secantResult - p) < std::abs(b - a)) {
        useSecant = true;
        p = secantResult;
        std::cout << "Switching to secant method after " << totalIterationCount << " iterations." << std::endl;
        totalIterationCount++;
      }
      //Else, start bisection again.
      else {
        bisectionIterationCount = 0;
      }
    }
  }
  // If we have begun using secant method, iterate through secant method until we've reached the max number of iterations or tolerance is met.
  if (useSecant) {
    double previous = 999999;
    while (totalIterationCount < max_iterations && std::abs(p - previous) > tol) {
      previous = p;
      p = secant_method(f, a, b, tol, max_iterations - totalIterationCount);
      totalIterationCount++;
    }
```

```
    }

    return p;
}
```

Problem 4

To test my algorithms, I calculated the difference between the solutions given by Gaussian Elimination and Gaussian Elimination With Pivoting, and then calculated the Infinity Norm, Manhattan Norm, and L2 Norm of that difference. I expected there to be a much larger difference in a system of equations with such small variations in the values, but the calculations ended up being closer than I'd expected. Still, the difference is present, as can be seen in the results below. The one thing that did result as expected is that the difference became larger as the systems of equations grew.

```cpp
// Andrew Sheridan
//Math 5610
//Written in C++
//main.cpp
#include "Matrix.h"
#include "MatrixFactory.h"
#include "MatrixOperations.h"
#include "Vector.h"
#include "Error.h"

int main() {
  for (int i = 16; i <= 256; i *= 2) {
    Matrix A1 = MatrixFactory::Instance()->Random(i, i); //Every entry is a random number between 0 and 1
    Matrix A2 = A1; //Copy matrix A1 into A2
    Vector b1(i);
    b1.InitializeAllOnes();
    Vector b2(i);
    b2.InitializeAllOnes();

    Matrix U1 = GaussianElimination(A1, &b1); //Vector passed by reference so it may be modified.
    Matrix U2 = GaussianEliminationWithScaledPivoting(A2, &b2);

    Vector x1 = BackSubstitution(U1, b1);
    Vector x2 = BackSubstitution(U2, b2);

    Vector diff = x1 - x2;

    std::cout << "Size: " << i << std::endl;
    std::cout << "Infinity Norm: " << diff.InfinityNorm() << std::endl;
    std::cout << "Manhattan Norm: " << diff.ManhattanNorm() << std::endl;
    std::cout << "L2 Norm: " << diff.L2Norm() << std::endl << std::endl;
  }

  return 0;
}
```

```
Size: 16
Infinity Norm: 4.60254e-12
Manhattan Norm: 2.80924e-11
L2 Norm: 9.09482e-12

Size: 32
Infinity Norm: 4.60965e-13
Manhattan Norm: 3.17426e-12
L2 Norm: 7.93738e-13
```

```
Size: 64
Infinity Norm: 3.73617e-11
Manhattan Norm: 8.72267e-10
L2 Norm: 1.28949e-10

Size: 128
Infinity Norm: 4.78762e-11
Manhattan Norm: 1.82054e-09
L2 Norm: 1.96125e-10

Size: 256
Infinity Norm: 6.39797e-10
Manhattan Norm: 3.81e-08
L2 Norm: 2.99137e-09
```

```
/// Reduces a matrix right-hand-side b to upper-triangular form using Gaussian Elimination
//A: The matrix to be reduced
// b: Right-Hand-Side
Matrix GaussianElimination(Matrix A, Vector* b) {
  if (A.GetRows() != b->GetSize()) return NULL;
  for (unsigned k = 0; k < A.GetRows(); k++) {
    for (unsigned i = k + 1; i < A.GetRows(); i++) {
      double factor = A[i][k] / A[k][k];
      for (unsigned j = 0; j < A.GetColumns(); j++) {
        A[i][j] = A[i][j] - factor*A[k][j];
      }
      A[i][k] = 0;
      b->entries[i] = b->entries[i] - factor*(b->entries[k]);
    }
  }
  return A;
}

/// Reduces an matrix A and right-hand-side b to upper-triangular form using Gaussian Elimination
// A: The coefficient matrix
// b: Right-Hand-Side
Matrix GaussianEliminationWithScaledPivoting(Matrix A, Vector* b) {
  if (A.GetRows() != b->GetSize()) return NULL;

  int n = b->GetSize();

  for (int k = 0; k < n; k++) {
    double* ratios = new double[n - k]; // New vector of size n - k to store the ratios
    for (int i = k; i < n; i++) {
      double rowMax = A[i].FindMaxMagnitudeStartingAt(k);
      ratios[i - k] = rowMax / A[i][k];
    }
    int newPivot = FindMaxIndex(ratios, n - k) + k; //Find the best row for this iteration

    Vector temp = A[k]; //
    A[k] = A[newPivot]; // Switch the current row with the best row for this iteration
    A[newPivot] = temp; //

```

```cpp
    double tempEntry = b->entries[k];
    b->entries[k] = b->entries[newPivot];
    b->entries[newPivot] = tempEntry;

    for (int i = k + 1; i < n; i++) {
      double factor = A[i][k] / A[k][k];
      for (int j = 0; j < n; j++) {
        A[i][j] = A[i][j] - factor*A[k][j];
      }
      b->entries[i] = b->entries[i] - (factor*b->entries[k]);
    }
  }
  return A;
}

/// Solves a set of linear equations using back substitution
/// Note: Does not reduce matrix A
//A: The upper-triangular matrix
// b: Right-Hand-Side
inline Vector BackSubstitution(Matrix A, Vector b) {
  if (A.GetRows() != b.GetSize()) return NULL;

  Vector x(b.GetSize());

  for(int i = b.GetSize() - 1; i >= 0; i--)
  {
    x[i] = b[i];
    for (int j = i + 1; j < b.GetSize(); j++) {
      x[i] -= A[i][j] * x[j];
    }
    x[i] /= A[i][i];
  }
  return x;
}

/// Initializes the entries to 1
void Vector::InitializeAllOnes() {
  for (unsigned i = 0; i < size; i++) {
    entries[i] = 1; //Assign each entry in entries to 1
  }
}

///Finds the entry with the largest magnitude, starting with entry "start".
double Vector::FindMaxMagnitudeStartingAt(unsigned start) {
  double max = 0;
  for (unsigned i = start; i < size; i++) {
    double value = std::abs(entries[i]);
    if (value > max) max = value;
  }
  return max;
}

///Finds the index of the value with the largest magnitude
unsigned Vector::FindMaxIndex() {
  double max = 0;
  unsigned index = -1;
  for (unsigned i = 0; i < size; i++) {
    double value = std::abs(entries[i]);
    if (value > max) {
      max = value;
      index = i;
    }
  }
```

```
  return index;
}

///Computes the L2 norm of the vector
double Vector::L2Norm() {
  double sum = 0;
  for (int i = 0; i < size; i++) {
    sum += (entries[i] * entries[i]);
  }
  return std::sqrt(sum);
}

///Computes the Manhattan norm of the vector
double Vector::ManhattanNorm() {
  double sum = 0;
  for (unsigned i = 0; i < size; i++) {
    sum += std::abs(entries[i]);
  }
  return sum;
}

///Computes the Infinity norm of the vector
double Vector::InfinityNorm() {
  double max = 0;
  for (unsigned i = 0; i < size; i++) {
    if (std::abs(entries[i]) > max) {
      max = std::abs(entries[i]);
    }
  }
  return max;
}
```

Problem 5

Using the method found in the textbook, I have created the following loop to solve a system of equations using QR Factorization via Gram Schmidt. I tested the method by taking the difference of the result and the ones vector, as has been done in previous homework assignments. The methods for computing the vector norms, as well as initializing vectors and matrices, are either already contained in previous problems or in the appendix. The only new method contained in this section is GramSchmidt, which computes the QR factorization.

```
// Andrew Sheridan
//Math 5610
//Written in C++
//main.cpp
#include "Matrix.h"
#include "MatrixFactory.h"
#include "MatrixOperations.h"
#include "Vector.h"
#include "Error.h"

int main() {
  for (int i = 16; i <= 256; i *= 2) {
    //Creates a diagonally dominant matrix
    Matrix matrix = MatrixFactory::Instance()->DiagonallyDominant(i, i);
    Vector onesVector(i);
    onesVector.InitializeAllOnes();
    Vector b = matrix * onesVector;

    //The first entry in the result is Q, the second, R
    Matrix* QR = GramSchmidt(matrix);
    Vector QTy = QR[0].Transpose() * b;
    Vector x = BackSubstitution(QR[1], QTy);

    Vector difference = onesVector - x;
    std::cout << "Accuracy of formula for system size " << i << std::endl;
    std::cout << "L2 Norm: " << difference.L2Norm() << std::endl;
    std::cout << "Manhattan Norm: " << difference.ManhattanNorm() << std::endl;
    std::cout << "Infinity Norm: " << difference.InfinityNorm() << std::endl <<std::endl;
  }

  return 0;
}
```

```
Accuracy of formula for system size 16
L2 Norm: 1.50598e-15
Manhattan Norm: 5.10703e-15
Infinity Norm: 6.66134e-16

Accuracy of formula for system size 32
L2 Norm: 2.48253e-15
Manhattan Norm: 1.11022e-14
Infinity Norm: 1.33227e-15

Accuracy of formula for system size 64
L2 Norm: 4.43256e-15
```

```
Manhattan Norm: 2.64233e-14
Infinity Norm: 1.55431e-15

Accuracy of formula for system size 128
L2 Norm: 9.7219e-15
Manhattan Norm: 8.88178e-14
Infinity Norm: 2.44249e-15

Accuracy of formula for system size 256
L2 Norm: 1.95718e-14
Manhattan Norm: 2.49911e-13
Infinity Norm: 3.55271e-15
```

```cpp
///Computes the QR factorization of Matrix A
///Returns a pair of matrices in an array. The first is Q, the second, R.
Matrix* GramSchmidt(Matrix A) {
  if (A.GetRows() != A.GetColumns()) return NULL;

  Matrix r(A.GetRows(), A.GetColumns());
  Matrix q(A.GetRows(), A.GetColumns());

  for (int k = 0; k < A.GetRows(); k++) {
    r[k][k] = 0;
    for (int i = 0; i < A.GetRows(); i++)
      r[k][k] = r[k][k] + A[i][k] * A[i][k];

    r[k][k] = sqrt(r[k][k]);

    for (int i = 0; i < A.GetRows(); i++)
      q[i][k] = A[i][k] / r[k][k];

    for (int j = k + 1; j < A.GetColumns(); j++) {
      r[k][j] = 0;
      for (int i = 0; i < A.GetRows(); i++)
        r[k][j] += q[i][k] * A[i][j];

      for (int i = 0; i < A.GetRows(); i++)
        A[i][j] = A[i][j] - r[k][j] * q[i][k];
    }
  }
  Matrix* QR = new Matrix[2];
  QR[0] = q;
  QR[1] = r;
  return QR;
}
```

Appendix

Here are the headers files for the Matrix, Vector, and MatrixFactory classes. The full methods used in the problems of this exam were included on a per-problem basis, except for the methods for the MatrixFactory, which have been included here.

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
//Matrix.h

#pragma once
#include "Vector.h"

class Matrix{
public:
  //Initialization and Deconstruction
  Matrix() = default;
  Matrix(unsigned size);
  Matrix(unsigned rowCount, unsigned columnCount);
  Matrix(const Matrix &m);
  Matrix operator = (const Matrix& m);
  ~Matrix();

  void InitializeIdentityMatrix();
  void InitializeRandom();
  void InitializeRange(double minValue, double maxValue);
  void InitializeDiagonallyDominant();

  //Overloaded Operators
  Vector &operator[] (unsigned row) { return entries[row]; }
  friend bool operator == (const Matrix& A, const Matrix& B);
  friend bool operator != (const Matrix& A, const Matrix& B);
  friend Vector operator * (const Matrix& A, Vector& x);
  friend Vector operator / (const Matrix& A, Vector& x);
  friend Matrix operator * (Matrix A, Matrix B);
  friend Matrix operator - (Matrix A, Matrix B);

  //Basic Algorithms
  bool IsSymmetric();
  Matrix Transpose();
  double OneNorm();
  double InfinityNorm();

  //Getters and Setters
  unsigned GetRows() { return rows; }
  unsigned GetColumns() { return columns; }
  void SetRows(unsigned r) { rows = r; }
  void SetColumns(unsigned c) { columns = c; }

  //Output
  void Print();
  void PrintAugmented(Vector v);

private:
  Vector* entries; //The entries of the matrix

  unsigned rows;
  unsigned columns;
};
```

```
#pragma once
//Andrew Sheridan
//Math 5610
//Written in C++
//Vector.h

class Vector {
public:
  //Initialization and Destruction
  Vector();
  Vector(unsigned n);
  Vector(const Vector &v);
  Vector(double* v, unsigned size);
  Vector operator=(const Vector& v);
  ~Vector();

  void InitializeRandomEntries();
  void InitializeAllOnes();

  //Overloaded Operators
  double& operator[] (unsigned x) { return entries[x]; }
  friend double operator*( Vector& a, Vector& b);
  friend Vector operator*(Vector& a, double constant);
  friend Vector operator*(double constant, Vector& a);
  friend Vector operator+(Vector& a, Vector& b);
  friend Vector operator-(Vector& a, Vector& b);
  friend Vector operator/(Vector& a, double constant);
  friend Vector operator+(Vector& a, Vector& b);

  //Basic Algorithms
  double FindMaxMagnitudeStartingAt(unsigned start);
  unsigned FindMaxIndex();
  double L2Norm();

  //Accessing Data
  void Print();
  unsigned GetSize() { return size; }
  void SetSize(unsigned newSize) { size = newSize; }

protected:
  unsigned size;

private:
  double* entries; //The stored values of the vector
};
```

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
//MatrixFactory.h
#pragma once
#include "Matrix.h"

///A singleton which creates new matrices
class MatrixFactory {
public:
  static MatrixFactory* Instance();
  Matrix Identity(unsigned rows, unsigned columns);
  Matrix Ones(unsigned rows, unsigned columns);
  Matrix UpperTriangular(unsigned rows, unsigned columns);
  Matrix LowerTriangular(unsigned rows, unsigned columns);
  Matrix Random(unsigned rows, unsigned columns);
  Matrix RandomRange(unsigned rows, unsigned columns, double min, double max);
  Matrix DiagonallyDominant(unsigned rows, unsigned columns);
  Matrix Symmetric(unsigned size);
  Matrix SPD(unsigned size);

private:
  MatrixFactory() {};
  MatrixFactory(MatrixFactory const&) {};
  MatrixFactory& operator=(MatrixFactory const&) {};
  static MatrixFactory* m_instance;
};
```

```cpp
//Andrew Sheridan
//Math 5610
//Written in C++
//MatrixFactory.cpp

#include "MatrixFactory.h"
#include <random>

MatrixFactory* MatrixFactory::m_instance = nullptr;

///Returns the instance of the MatrixFactory singleton
MatrixFactory* MatrixFactory::Instance() {
  if (!m_instance)
    m_instance = new MatrixFactory();

  return m_instance;
}

///Creates an identity matrix
Matrix MatrixFactory::Identity(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  m.InitializeIdentityMatrix();
  return m;
}

///Creates a matrix where every entry is a value between 0 and 1
Matrix MatrixFactory::Random(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  m.InitializeRandom();
  return m;
}

///Creates a Diagonally Dominant matrix
```

```
Matrix MatrixFactory::DiagonallyDominant(unsigned rows, unsigned columns) {
  Matrix m(rows, columns);
  m.InitializeDiagonallyDominant();
  return m;
}

///Creates a symmetric positive definite matrix
Matrix MatrixFactory::SPD(unsigned size) {
  std::mt19937 generator(123); //Random number generator
  std::uniform_real_distribution<double> dis(0.0, 1.0); //Desired distribution

  Matrix matrix(size);

  for (unsigned i = 0; i < size; i++) {
    for (unsigned j = i; j < size; j++) {
      double value = dis(generator); //Assign each entry in matrix to random number between 0 and 1
      matrix[i][j] = value;
      matrix[j][i] = value;
    }
  }

  for (unsigned k = 0; k < size; k++) {
    matrix[k][k] += 10 * size; //Add 10*n to all diagonal entries
  }

  return matrix;
}
```