HW9 - Math 5610

Andrew Sheridan

December 13, 2016

Problem 1 : Jacobi Iteration

The only "new" code in my main function is the method *JacobiIteration*. The basic Matrix and Vector classes were implemented in HW7, as well as the *GaussianElimination* and *BackSubstitution* methods. To test the algorithm I used a simple sample system found on page 175 of the textbook. Testing with larger systems of equations will be implemented in Problem 3. From our output we can see that the results of the Jacobi Iteration closely match that of the Gaussian Elimination and Back Substitution. Changing our tolerance would improve these results.

```
1  //Andrew Sheridan
2  //Math 5610
3  //Written in C++
4  //main.cpp
5
6  #include "../../../HW7/HW7/Matrix.h"
7  #include "../../../HW7/HW7/MatrixFactory.h"
8  #include "../../../HW7/HW7/MatrixOperations.h"
9  #include "../../../HW7/HW7/Vector.h"
10
11 int main() {
12     //Problem 1 : Jacobi Iteration
13   Matrix m1(3);
14   m1[0][0] = 7;
15   m1[0][1] = 3;
16   m1[0][2] = 1;
17   m1[1][0] = -3;
18   m1[1][1] = 10;
19   m1[1][2] = 2;
20   m1[2][0] = 1;
21   m1[2][1] = 7;
22   m1[2][2] = -15;
23
24   Vector b1(3);
25   b1[0] = 3;
26   b1[1] = 4;
27   b1[2] = 2;
28
29   Vector x1(3);
30
31   int maxIter = 10000;
32   double tolerance = 0.00001;
33   std::cout << "The example matrix given on page 175 of the textbook: "
         << std::endl;
34   m1.PrintAugmented(b1);
35
36   std::cout << "The results of Jacobi Iteration with max iterations of"
         << maxIter << " and tolerance of" << tolerance << std::endl;
```

```
  Vector result1 = JacobiIteration(m1, x1, b1, maxIter, tolerance);
  result1.Print();

  std::cout << "The results of Gaussian Elimination and Back Substitution" << std::endl;
  GaussianElimination(m1, b1);
  Vector actual1 = BackSubstitution(m1, b1);
  actual1.Print();

  return 0;
}
```

```
The example matrix given on page 175 of the textbook:
7           3           1           | 3
-3          10          2           | 4
1           7           -15         | 2

The results of Jacobi Iteration with max iterations of 10000 and tolerance of1e-05
Number of iterations taken in Jacobi Iteration: 51
0.223242    0.448775    0.0909775

The results of Gaussian Elimination and Back Substitution
0.223242    0.448777    0.0909786
```

```
//MatrixOperations.h

///Solves the system of equations using Jacobi Iteration
///A: The Matrix
///x0: The initial guess
///b: The Right-Hand-Side
Vector JacobiIteration(Matrix A, Vector x0, Vector b, int maxIterations, double tolerance) {
  int iterations = 0;
  int n = A.GetRows();
  Vector newX(x0);
  double error = 10 * tolerance;
  while(iterations < maxIterations && tolerance < error){
    Vector oldX(newX);
    for (int i = 0; i < n;  i++) {
      newX[i] = b[i];
      for (int j = 0; j < i; j++) {
        newX[i] = newX[i] - A[i][j] * oldX[j];
      }
      for (int j = i + 1; j < n; j++) {
        newX[i] = newX[i] - A[i][j] * oldX[j];
      }
      newX[i] = newX[i] / A[i][i];
      error = (oldX - newX).L2Norm();
      iterations++;
    }
  }
  std::cout << "Number of iterations taken in Jacobi Iteration: " << iterations << std::endl;
  return newX;
}
```

Problem 2 : Gauss Seidel

The only "new" code in my main function is the method *GaussSeidel*. The basic Matrix and Vector classes were implemented in HW7, as well as the *GaussianElimination* and *BackSubstitution* methods. To test the algorithm I used a simple sample system found on page 175 of the textbook. Testing with larger systems of equations will be implemented in Problem 3. From our output we can see that the results of the Gauss Seidel method closely match that of the Gaussian Elimination and Back Substitution. Changing our tolerance would improve these results.

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

#include "Matrix.h"
#include "Vector.h"
#include "MatrixOperations.h"
#include "MatrixFactory.h"
#include <iostream>

int main() {
//Problem 2 : Gauss-Seidel
  Matrix m2(3);
  m2[0][0] = 7;
  m2[0][1] = 3;
  m2[0][2] = 1;
  m2[1][0] = -3;
  m2[1][1] = 10;
  m2[1][2] = 2;
  m2[2][0] = 1;
  m2[2][1] = 7;
  m2[2][2] = -15;

  Vector b2(3);
  b2[0] = 3;
  b2[1] = 4;
  b2[2] = 2;

  Vector x2(3);
  std::cout << "The example matrix given on page 175 of the textbook: " << std::endl;
  m2.PrintAugmented(b2);

  std::cout << "The results of Gauss Seidel with max iterations of " << maxIter << " and tolerance of" << tolerance << std::endl;
  Vector result2 = GaussSeidel(m2, x2, b2, maxIter, tolerance);
  result2.Print();

  std::cout << "The results of Gaussian Elimination and Back Substitution" << std::endl;
  GaussianElimination(m2, b2);
  Vector actual2 = BackSubstitution(m2, b2);
  actual2.Print();

  return 0;
}
```

```
The example matrix given on page 175 of the textbook:
7           3           1           | 3
-3          10          2           | 4
```

```
1           7           -15          | 2

The results of Gauss Seidel with max iterations of 10000 and tolerance of1e-05
Number of iterations taken in Gauss Seidel: 51
0.223242     0.448775     0.0909775

The results of Gaussian Elimination and Back Substitution
0.223242     0.448777     0.0909786
```

```
//MatrixOperations.h

///Solves the system of equations using Gauss-Seidel
///A: The Matrix
///x0: The initial guess
///b: The Right-Hand-Side
Vector GaussSeidel(Matrix A, Vector x0, Vector b, int maxIterations, double tolerance) {
  int iterations = 0;
  int n = A.GetRows();
  Vector newX(x0);
  double error = 10 * tolerance;
  while (iterations < maxIterations && tolerance < error) {
    Vector oldX(newX);
    for (int i = 0; i < n; i++) {
      newX[i] = b[i] / A[i][i];
      for (int j = 0; j < i; j++) {
        newX[i] = newX[i] - ((A[i][j] / A[i][i]) * oldX[j]);
      }
      for (int j = i + 1; j < n; j++) {
        newX[i] = newX[i] - ((A[i][j] / A[i][i]) * oldX[j]);
      }
      error = (oldX - newX).L2Norm();
      iterations++;
    }
  }
  std::cout << "Number of iterations taken in Gauss Seidel: " << iterations << std::endl;
  return newX;
}
```

Problem 3 : Comparing Gauss-Seidel and Jacobi Iteration

As has been done in previous assignments, I have made matrices of increasingly large sizes and have tested my new algorithms with larger and larger systems of equations. My intentions were to assure that these functions work with larger systems of equations, and to compare the number of iterations required for each method. Turns out, the number of iterations were exactly the same for each method. Also, the results obtained were vectors of ones, which were the desired result.

```
1  //Andrew Sheridan
2  //Math 5610
3  //Written in C++
4  //main.cpp
5
6  #include "Matrix.h"
7  #include "Vector.h"
8  #include "MatrixOperations.h"
9  #include "MatrixFactory.h"
10 #include <iostream>
11
12 int main() {
13     int maxIter3 = 10000;
14   double tol3 = 0.00001;
15   for (int i = 4; i <= 160; i *= 2) {
16     Matrix m3 = MatrixFactory::Instance()->DiagonallyDominant(i, i);
17     Vector onesVector(i);
18     onesVector.InitializeAllOnes();
19     Vector v3 = m3 * onesVector;
20
21     Vector zeroes(i);
22     Vector result3 = JacobiIteration(m3, zeroes, v3, maxIter3, tol3);
23     std::cout << "Result of Jacobi iteration on DD matrix size " << i ↩
           << std::endl;
24     result3.Print();
25
26     Vector result3b = GaussSeidel(m3, zeroes, v3, maxIter3, tol3);
27     std::cout << "Result of Gauss Seidel on DD matrix size " << i << ↩
           std::endl;
28     result3b.Print();
29   }
30
31   return 0;
32 }
```

Here are the results gathered for the number of iterations required for the Gauss-Seidel and Jacobi Iteration methods for matrices of various sizes.

Size	4	8	16	32	64	128
Gauss-Seidel	20	48	96	192	384	768
Jacobi	20	48	96	192	384	768

Problem 4 : Comparing LU Factorization to Iterative Methods

As has been done in past assignments, I created a loop which would test the Gauss-Seidel and Jacobi Iteration methods and compare them to the LU Factorization method in terms of operations required. The number of iterations required in the Gauss-Seidel and Jacobi Iteration methods were smaller than that of the LU Factorization method for larger systems.

Once the size of our systems reached 160, the number of operations required by the iterative methods was around 450000 (for tolerance of 0.000000001). The operations required by LU factorization (without including the operation count for back substitution) was about 4,000,000, around ten times as many. When compared to the operation counts required for smaller systems, LU factorization seemed to be more efficient.

I modified these various methods to count their operations and then print them to the console before returning their results. I have included these operations counts in the *verbatim* section.

Note that I've only done these comparisons with Diagonally Dominant matrices. Random matrices were not well conditioned for the iterative methods, and these methods would go on until reaching their max iteration count, and return garbage results.

```
1  //Andrew Sheridan
2  //Math 5610
3  //Written in C++
4  //main.cpp
5
6  #include "Matrix.h"
7  #include "Vector.h"
8  #include "MatrixOperations.h"
9  #include "MatrixFactory.h"
10 #include <iostream>
11
12 int main() {
13     //Problem 4: LU Factorization vs Iterative Methods
14   int maxIter4 = 10000;
15   double tol4 = 0.000000001;
16   for (int i = 5; i <= 160; i *= 2) {
17     Matrix m4 = MatrixFactory::Instance()->DiagonallyDominant(i, i);
18     Vector onesVector(i);
19     onesVector.InitializeAllOnes();
20     Vector v4 = m4 * onesVector;
21
22     Vector zeroes(i);
23     Vector result4 = JacobiIteration(m4, zeroes, v4, maxIter4, tol4);
24     std::cout << "Result of Jacobi iteration on DD matrix size " << i ↩
           << std::endl;
25     result4.Print();
26
27     Vector result4b = GaussSeidel(m4, zeroes, v4, maxIter4, tol4);
28     std::cout << "Result of Gauss Seidel on DD matrix size " << i << ↩
           std::endl;
29     result4b.Print();
30
31     LUFactorization(m4, v4);
32     Vector result4c = BackSubstitution(m4, v4);
33     std::cout << "Result of LU Factorization and Back Substitution on ↩
           DD matrix size " << i << std::endl;
34     result4c.Print();
35   }
36
37   return 0;
38 }
```

```
Jacobi Iteration size 5: 529
Gauss Seidel size 5: 488
LU Factorization size 5: 130
Jacobi Iteration size 10: 1849
Gauss Seidel size 10: 1768
LU Factorization size 10: 1035
Jacobi Iteration size 20: 7750
Gauss Seidel size 20: 7569
LU Factorization size 20: 8170
Jacobi Iteration size 40: 29890
Gauss Seidel size 40: 29529
LU Factorization size 40: 64740
Jacobi Iteration size 80: 117370
Gauss Seidel size 80: 116649
LU Factorization size 80: 515080
Jacobi Iteration size 160: 465130
Gauss Seidel size 160: 463689
LU Factorization size 160: 4108560
```

Problem 5: Conjugate Gradient Method

In order to test the conjugate gradient method in a manner I could step through and debug, I first used Example 7.9 found on page 184 of the textbook. Once I got the expected result and the steps worked correctly, I tested the algorithm on systems sizes 10, 20, 40, 80, and 160, as instructed. The matrices tested were all symmetric positive definite.

Using the method implemented in various other assignments, we set up the system with a matrix and right-hand-side where we know the computed solutions will be the ones vector. For all sizes, the computed result was, in fact, the ones vector. The printed values below are my results from Example 7.9.

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

#include "Matrix.h"
#include "Vector.h"
#include "MatrixOperations.h"
#include "MatrixFactory.h"
#include <iostream>

int main() {
    //Problem 5 : Conjugate Gradient Method
  Matrix m5(3);
  m5[0][0] = 7;
  m5[0][1] = 3;
  m5[0][2] = 1;
  m5[1][0] = 3;
  m5[1][1] = 10;
  m5[1][2] = 2;
  m5[2][0] = 1;
  m5[2][1] = 2;
  m5[2][2] = 15;

  Vector b5(3);
  b5[0] = 28;
  b5[1] = 31;
  b5[2] = 22;

  Vector x5(3);

  std::cout << "Matrix and RHS given on page 184." << std::endl;
  m5.PrintAugmented(b5);
  Vector result5 = ConjugateGradient(m5, b5, x5, 0.000001);
  std::cout << "Result of conjugate gradient method: " << std::endl;
  result5.Print();

  for (int i = 5; i <= 160; i *= 2) {
    Matrix m5 = MatrixFactory::Instance()->SPD(i);
    Vector ones(i);
    ones.InitializeAllOnes();
    Vector v5 = m5 * ones;
    Vector zeroes(i);

    Vector result5 = ConjugateGradient(m5, v5, zeroes, 0.0000001);
    result5.Print();
  }

  return 0;
}
```

```
Matrix and RHS given on page 184.
7            3            1            | 28
3            10           2            | 31
1            2            15           | 22

Result of conjugate gradient method:
3            2            1
```

```
///Solves a matrix and RHS using Conjugate Gradient Method
///A : The matrix to be solved
///b : The RHS vector
///x0 : The initial guess vector
///tol : The tolerance of our method
Vector ConjugateGradient(Matrix A, Vector b, Vector x0, double tol) {
  Vector rk = b - (A * x0);
  double dk = rk * rk;
  double bd = b * b;
  int k = 0;
  Vector pk = rk;
  Vector xk = x0;
  while (dk > tol * tol * bd) {
    Vector sk = A * pk;
    double ak = dk / (pk * sk);
    Vector xkp1 = xk + (ak * pk);
    Vector rkp1 = rk - (ak * sk);
    double dkp1 = rkp1 * rkp1;
    Vector pkp1 = rkp1 + ((dkp1 / dk) * pk);
    k++;
    //All values have been computed, set all kth values to equal the k+1 value in preparation for next iteration
    xk = xkp1;
    rk = rkp1;
    pk = pkp1;
    dk = dkp1;
  }
  return xk;
}
```

Problem 6: Textbook 7.7

Problem 7 : Textbook 7.15

If the associated matrix of an energy norm $||x||_A = \sqrt{x^T Ax}$ is symmetric positive definite, the energy norm is in fact a norm. Because the matrix is symmetric positive definite, as we do matrix-vector multiplication, it is almost as though we are multiplying x by a scalar vector, because the diagonal entries of matrix A are larger in magnitude than the sum of the other entries in each corresponding row and column. Hence, the energy norm is similar to the L2 norm, except that we are using the SPD matrix A to scale the result in terms of A.