HW10 - Math 5610

Andrew Sheridan

December 13, 2016

Problem 1: Power Method

I first sought out an example to make sure that the steps were being executed correctly. Once the example yielded the correct eigenvalue and eigenvector, I tested it on diagonally dominant systems from size 5 to 160. The only new code here is the PowerMethod, which I will include. Other functions were implemented in a modular nature in previous assignments. The eigenvalues obtained seem to be what we'd expect of these system, given that the entries are rather predictable. The algorithm used to generate these matrices is that which was assigned us in a previous assignment.

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

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

int main() {
    //problem 1: Power Method
  Matrix A(2);
  A[0][0] = 2;
  A[0][1] = -12;
  A[1][0] = 1;
  A[1][1] = -5;

  Vector x_0(2);
  x_0[0] = 1;
  x_0[1] = 1;

  double lambda = PowerMethod(A, x_0, 0.0001, 1000);
  std::cout << "Lambda: " << lambda << std::endl;
  double relativeError = realRelative(-2.0, lambda);
  std::cout << "Relative Error: " << relativeError << std::endl;
  double absoluteError = realAbsolute(-2.0, lambda);
  std::cout << "Absolute Error: " << absoluteError << std::endl;

  for (int i = 5; i <= 160; i *= 2) {
    Matrix m1 = MatrixFactory::Instance()->DiagonallyDominant(i, i);
    Vector onesVector(i);
    onesVector.InitializeAllOnes();
    Vector v1 = m1 * onesVector;

    Vector zeroes(i);
```

```
    double result1a = PowerMethod(m1, onesVector, 0.000001, 10000);
    std::cout << "Result of Power Method on DD matrix size " << i << ↩
        std::endl;
    std::cout << result1a << std::endl;
  }

  return 0;
}
```

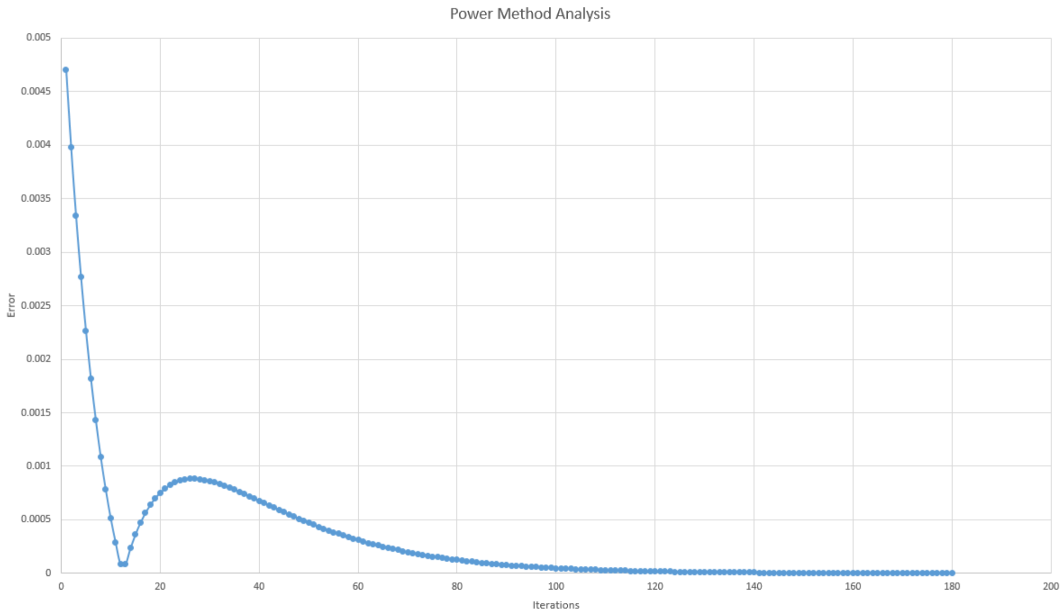
```
Test Matrix:
2             -12           | 1
1             -5            | 1

Lambda: -2.00005
Relative Error: 2.645e-05
Absolute Error: 5.28999e-05

Result of Power Method on DD matrix size 5
52.5177
Result of Power Method on DD matrix size 10
104.828
Result of Power Method on DD matrix size 20
209.826
Result of Power Method on DD matrix size 40
420.242
Result of Power Method on DD matrix size 80
840.329
Result of Power Method on DD matrix size 160
1680.25
```

```
///Finds an approximation of the largest Eigenvalue of a matrix
///A : The square matrix
///x0 : The initial guess vector
///tol : The tolerance of the algorithm
///maxIter : The maximum number of iterations to be executed by the ↩
    method
double PowerMethod(Matrix A, Vector x0, double tol, int maxIter) {
  double error = 10 * tol;
  int k = 0;
  Vector y = A * x0;
  Vector xk = x0;
  double lambda_k = 0;

  while (error > tol && k < maxIter) {
    Vector xkp1 = y / y.L2Norm();
    y = A * xkp1;
    double lambda_kp1 = xkp1 * y;
    error = abs(lambda_kp1 - lambda_k);

    lambda_k = lambda_kp1;
    k++;
  }

  /*std::cout << "Eigenvector approximation: " << std::endl;
  y.Print();*/
```

```

  return lambda_k;
}
```

Problem 2: Power Method Testing

Inside the PowerMethod function I added some output during each loop, by finding the error and outputting it with the iteration count. This imported to Excel gave us the following graph. Note: The error of the first entry was much larger than the scale of the other entries, so it was removed to provide a more meaningful visualization. This particular graph was gathered from the 80x80 system from problem 1.




```
///Finds an approximation of the largest Eigenvalue of a matrix
///A : The square matrix
///x0 : The initial guess vector
///tol : The tolerance of the algorithm
///maxIter : The maximum number of iterations to be executed by the method
double PowerMethod(Matrix A, Vector x0, double tol, int maxIter) {
  double error = 10 * tol;
  int k = 0;
  Vector y = A * x0;
  Vector xk = x0;
  double lambda_k = 0;

  //Output for problem 10.2. Comment out if not needed.
  std::ofstream output("powerMethod.txt");
  output << "Iterations \t Error " << std::endl;

  while (error > tol && k < maxIter) {
    Vector xkp1 = y / y.L2Norm();
    y = A * xkp1;
    double lambda_kp1 = xkp1 * y;
    error = abs(lambda_kp1 - lambda_k);

    //Output for problem 10.2. Comment out if not needed.
    output << k << "\t" << error << std::endl;

    lambda_k = lambda_kp1;
    k++;
  }

```

```
  output.close();

  /*std::cout << "Eigenvector approximation: " << std::endl;
  y.Print();*/

  return lambda_k;
}
```

Problem 3: Inverse Power Method

Using the same system as part 1, I tested to make sure the correct smallest eigenvalue was returned. The final error shows that our approximation is close, and certainly if we decrease the tolerance of the function, we could increase the accuracy further (to a certain point.) Larger systems will be tested in Problem 4.

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

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

int main() {
    //problem 3: Inverse Power Method
  Matrix A3(2);
  A3[0][0] = 2;
  A3[0][1] = -12;
  A3[1][0] = 1;
  A3[1][1] = -5;

  Vector x3_0(2);
  x3_0[0] = 1;
  x3_0[1] = 1;

  double lambda3 = InversePowerMethod(A3, x3_0, 0.000001, 1000);
  std::cout << "Lambda: " << lambda3 << std::endl;
    double relativeError3 = realRelative(-1.0, lambda3);
  std::cout << "Relative Error: " << relativeError3 << std::endl;
  double absoluteError3 = realAbsolute(-1.0, lambda3);
  std::cout << "Absolute Error: " << absoluteError3 << std::endl;
  return 0;
}
```

```
Lambda: -0.999999
Relative Error: 7.29279e-07
Absolute Error: 7.29279e-07
```

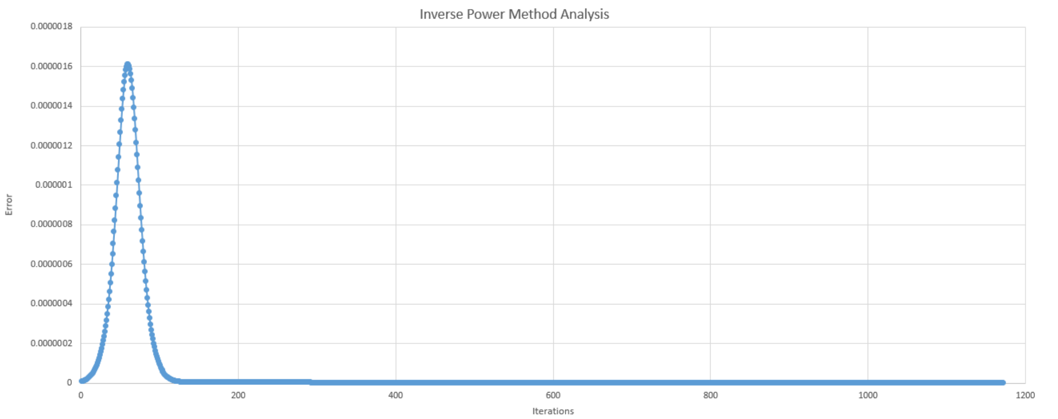
```cpp
///Finds an approximation of the smallest Eigenvalue of a matrix
///A : The square matrix
///x0 : The initial guess vector
///tol : The tolerance of the algorithm
///maxIter : The maximum number of iterations to be executed by the method
double InversePowerMethod(Matrix A, Vector x0, double tol, int maxIter) {
  double error = 10 * tol;
  int k = 0;
  Matrix U = A;
  Matrix L = LUFactorization(U, x0);// L is returned, U is modified to be upper triangular
  Vector y = BackSubstitution(U, x0);// Solve for y by doing back substitution.
  double lambda_x = 0;
```

```
  while (error > tol && k < maxIter) {
    Vector x = y / y.L2Norm();
    y = SolveSystem(A, x); //Does Gaussian Elimination then Back ↩
        Substitution
    double lambda_xp1 = x * y;
    error = abs(lambda_xp1 - lambda_x);

    lambda_x = lambda_xp1;
    k++;
  }

  return lambda_x;
}
```

Problem 4: Inverse Power Method Testing

The particular graph below was taken from the 80 by 80 system created in our main loop. I added some output to the original InversePowerMethod function from problem 3 to output the error and iteration count at each iteration.

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

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

int main() {
    //Problem 4 : Inverse Power Method Analysis
  for (int i = 5; i <= 80; i *= 2) {
    Matrix m4 = MatrixFactory::Instance()->DiagonallyDominant(i, i);
    Vector onesVector(i);
    onesVector.InitializeAllOnes();
    Vector v4 = m4 * onesVector;

    Vector zeroes(i);
    std::cout << "Test matrix: " << std::endl;
    double result4a = InversePowerMethod(m4, onesVector, 0.000000001, 10000);
    std::cout << "Result of Inverse Power Method on DD matrix size " << i << std::endl;
    std::cout << result4a << std::endl;
  }

  return 0;
}
```




```cpp
///Finds an approximation of the smallest Eigenvalue of a matrix
///A : The square matrix
///x0 : The initial guess vector
///tol : The tolerance of the algorithm
///maxIter : The maximum number of iterations to be executed by the ↩
```

```
    method
double InversePowerMethod(Matrix A, Vector x0, double tol, int maxIter↩
    ) {

  //Output for problem 10.5. Comment out if not needed.
  std::ofstream output("inversePowerMethod.txt");
  output << "Iterations \t Error " << std::endl;

  double error = 10 * tol;
  int k = 0;
  Matrix U = A;
  Matrix L = LUFactorization(U, x0);// L is returned, U is modified to↩
       be upper triangular
  Vector y = BackSubstitution(U, x0);// Solve for y by doing back ↩
      substitution.
  double lambda_x = 0;
  while (error > tol && k < maxIter) {
    Vector x = y / y.L2Norm();
    y = SolveSystem(A, x); //Does Gaussian Elimination then Back ↩
        Substitution
    double lambda_xp1 = x * y;
    error = abs(lambda_xp1 - lambda_x);
    /*std::cout << "lambda_k: " << lambda_x << std::endl;
    std::cout << "lambda_k+1: " << lambda_xp1 << std::endl;*/
    //std::cout  << std::endl;
    output << k << "\t" << error << std::endl;

    lambda_x = lambda_xp1;
    k++;
  }

  output.close();
  return lambda_x;
}
```