Five Algorithms for Finding Matrix Equation Solutions

An Honors Project

Tyler Allen

1 Introduction

Systems of linear equations appear frequently in mathematics. Matrices have associated properties and algorithms that simplify handling systems of linear equations dramatically. However, matrix equations become increasingly difficult to solve as the matrix increases in size. Thankfully, computers can perform these calculations significantly faster than a human can by hand. I have implemented five different algorithms for solving matrix equations: Gaussian Elimination, Gaussian Elimination with Partial Pivoting, Jacobi's Method, The Gauss-Seidel Method, and a Successive Over-relaxation (SOR) Method. We will discuss the details of these algorithms and their implementation.

2 General Implementation Information

These algorithms for solving matrix equations were implemented in Python 3.4, due to some improvements in Python's looping and printing methods over Python 2. It is likely that the program will not execute succesfully or efficiently in Python 2. The implementation provides a single interface for all of the provided methods, and uses command line arguments to accept the input.

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py
Invalid number of arguments.
Usage: python /home/tyler/dev/math362/MatrixSolvers/src/main.py solve_method [file_input] [max_iterations] [tolerance] [sor_w]
 max_iterations, tolerance required for iterative methods.sor_w value required for SOR methods.

Valid Solve Methods:
gaussian: Basic Gaussian Elimination using Backwards Substitution
gaussian_pivot: Gaussian Elimination using Partial Pivots
jacobi: Jacobi approximation method
gauss_seidel: Gause-Seidel approximation, improvement on Jacobi
sor: Successive Over-Relaxtion approximation method

Valid File Format: {{a11, a12, ...}, {a21, a22, ...}, ...}
Any whitespace is accepted.

Process finished with exit code 1
```

Figure 1: The usage message printed in the event of incorrect arguments or missing files.

Figure 1 contains a message shown to the user if a user inputs an incorrect number of arguments or the file containing their matrix is unreadable or does not exist. Due to the complexity of matrix input, it was decided that using files for input would provide an easier experience to the user than allowing the user to enter the matrix at program execution.

```
{ { 4, 3, 0, 24}, {3, 4, -1, 30}, {0, -1, 4, -24} }
```

Figure 2: An example input file.

```
{
{1, -1, 2, -1, -8},
{2, -2, 3, -3, -20},
{1,  1, 1,  0, -2},
{1, -1, 4,  3,  4}
}
```

Figure 3: Another example input file.

Figures 2, 3 show different formats for an input file. Some test files are included with the source code. Any amount of whitespace may be used. The matrix parsing algorithm is not advanced enough to detect most errors in user input. A usage message will be displayed in the event that an input file contains syntax errors. The matrix entry format is standard; outer brackets indicate the a matrix, inner brackets indicate a row, and commas seperate elements (values AND rows). It was decided that this format is more flexible than entering matrices directly into source code before executing the program, even with the possibility of user error. The algorithms in the referenced textbook specify nxn matrices[1]. The program will only function correctly when given an $nxn + 1$ matrix, where

the additional vector is the $\bar{b}$ vector in $A\bar{x} = \bar{b}$[1]. This is because the algorithms expect the matrices to have single solutions, $rank(n)$. These implementations are also undefined in the event of inconsistency.

3 Exact Methods

Exact methods, such as Gaussian Elimination, give an exact solution. These exact solutions can be computationally expensive, but attempt to give solutions without approximation[1]. Some inconsistency is inherent in these algorithms when executed by a computer due to computer roundoff error[1]. This error can be exaggerated greatly; a value with roundoff error will propogate the error throughout every calculation where the value appears[1]. This can create exceedingly large error, but is a casualty of limited storage for floating point values. Some exact methods attempt to avoid this roundoff error[1].

3.1 Gaussian Elimination with Backwards Substitution

Gaussian Elimination is a standard method for solving matrix equations by-hand. If a matrix can be reduced to its Reduced Row Echelon form, the solution is easily obtained. Each iteration of this method attempts to find a row to become the pivot row for the curret pivot column. This is done by finding the first non-zero row. Once a pivot row and column are determined, the other values in that column are eliminated using elementary row operations.

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py gaussian solve2.txt
Original Matrix:
[[1, -1, 2, -1, -8], [2, -2, 3, -3, -20], [1, 1, 1, 0, -2], [1, -1, 4, 3, 4]]

Matrix:

[[1, -1, 2, -1, -8], [0.0, 2.0, -1.0, 1.0, 6.0], [0.0, 0.0, -1.0, -1.0, -4.0], [0.0, 0.0, 0.0, 2.0, 4.0]]

Solution Vector:

[-7.0, 3.0, 2.0, 2.0]
```

Figure 4: Gaussian Elimination

Figure 4 displays a usage of Gaussian Elimination. This method can be accessed by using "gaussian" as the solution method within the program. Example equations were pulled from the text[1].

3.2 Gaussian Elimination with Partial Pivoting

Gaussian Elimination with Partial Pivoting was created to handle the computational roundoff error issue[1]. The difference between Gaussian Elimination with Backwards Substitution and this method is in the choice of pivot rows; rather than use the first available non-zero entry in the appropriate column as the pivot row, we use the entry in the pivot column with the highest magnitude (i.e. highest absolute value)[1]. This will produce a more accurate result with less roundoff error[1].

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py gaussian_pivot solve4.txt
Original Matrix:
[[0.003, 59.14, 59.17], [5.291, -6.13, 46.78]]

Matrix:

[[5.291, -6.13, 46.78], [0.0, 59.143475713475716, 59.143475713475716]]

Solution Vector:

[10.0, 1.0]
```

Figure 5: Gaussian Elimination with Partial Pivoting. Example was pulled from the text[1].

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py gaussian solve4.txt
Original Matrix:
[[0.003, 59.14, 59.17], [5.291, -6.13, 46.78]]

Matrix:

[[0.003, 59.14, 59.17], [0.0, -104309.37666666668, -104309.37666666668]]

Solution Vector:

[10.000000000000378, 1.0]
```

Figure 6: Gaussian Elimination produces a less accurate result due to roundoff error. Example was pulled from the text[1].

Figure 5 displays one usage of this method. The same matrix equation is solved using Gaussian Elimination with Backwards Substitution in Figure 6. The solution provided using the Partial Pivoting method provides a far more precise result. This method can be accessed by using "gaussian_pivot" as the solution method within the program.

4 Iterative Methods

Iterative methods provide approximate solutions to matrix equations. For large matrices, these methods are far more computationally efficient than their exact counterparts[1]. However, accuracy is lost without a sufficient number of iterations and a sufficiently lower tolerance.

4.1 Jacobi's Method

Jacobi's method is one such approximate method. It requires a candidate solution to be provided as a guess[1]. This implementation assumes the guess to be $\bar{0}$ in every case. Jacobi's method then uses the provided candidate to approach the actual solution[1]. After an iteration, if the maximum number of iterations is exceeded or the current solution's distance from the actual solution is within the

tolerance level, we use the current solution[1]. Otherwise, the current solution is the new candidate for the next iteration[1].

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py jacobi jacobi.txt 25 .003
Original Matrix:
[[10, -1, 2, 0, 6], [-1, 11, -1, 3, 25], [2, -1, 10, -1, -11], [0, 3, -1, 8, 15]]

Finished in 8 iterations
Matrix:

[[10, -1, 2, 0, 6], [-1, 11, -1, 3, 25], [2, -1, 10, -1, -11], [0, 3, -1, 8, 15]]

Solution Vector:

[0.9996741452148707, 2.0004476715450092, -1.0003691576845712, 1.0006191901399695]
```

Figure 7: Jacobi's Method. Example was pulled from the text[1].

Figure 7 displays an example usage of Jacobi's method. This method can be accessed by using "jacobi" as the solution method within the program.

4.2 The Gauss-Seidel Method

The Gauss-Seidel method is an improvement on Jacobi's method[1]. During Jacobi's method the candidate's values are used to calculate a new candidate[1]. The values of the new candidate are created in order from $1...n$[1]. For the calculation of the later values, using the already calculated values of the current candidate allows us to converge on the true solution faster[1].

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py gauss_seidel jacobi.txt 25 .003
Original Matrix:
[[10, -1, 2, 0, 6], [-1, 11, -1, 3, 25], [2, -1, 10, -1, -11], [0, 3, -1, 8, 15]]

Finished in 4 iterations
Matrix:

[[10, -1, 2, 0, 6], [-1, 11, -1, 3, 25], [2, -1, 10, -1, -11], [0, 3, -1, 8, 15]]

Solution Vector:

[1.000091280285995, 2.000021342246459, -1.0000311471834449, 0.9999881032596473]
```

Figure 8: The Gauss-Seidel Method. Example matrix was pulled from the text[1].

Figure 8 shows the usage of the Gauss-Seidel method. This method can be accessed by using "gauss_seidel" as the solution method within the program.

4.3 Successive Over-Relaxation Method

The SOR method is a modification of the Gauss-Seidel method[1]. The SOR method is an over-relaxtion method[1]. This means that, by using some additional operations involving some constant $1 < \omega < 2$, we will converge faster[1]. With a properly chosen ω, convergence can be sped up greatly over the Gauss-Seidel method[1]. However, choosing an appropriate ω depends greatly on the matrix itself and can be difficult[1]. This implementation leaves the value of ω up to the user.

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py sor sor.txt 25 .0003 1.25
Original Matrix:
[[4, 3, 0, 24], [3, 4, -1, 30], [0, -1, 4, -24]]

Finished in 8 iterations
Matrix:

[[4, 3, 0, 24], [3, 4, -1, 30], [0, -1, 4, -24]]

Solution Vector:

[3.000002459267881, 4.00001497819195, -5.0000222146211835]
```

Figure 9: The SOR Method. Example matrix pulled from the text[1].

```
/usr/bin/python3.4 /home/tyler/dev/math362/MatrixSolvers/src/main.py gauss_seidel sor.txt 25 .0003
Original Matrix:
[[4, 3, 0, 24], [3, 4, -1, 30], [0, -1, 4, -24]]

Finished in 18 iterations
Matrix:

[[4, 3, 0, 24], [3, 4, -1, 30], [0, -1, 4, -24]]

Solution Vector:

[3.000254109884176, 3.999788241763187, -5.000052939559203]
```

Figure 10: The Gauss-Seidel Method performing slower than the SOR method. Example matrix pulled from the text[1].

Figure 9 shows the usage of the SOR method. In comparison to 10, it is apparent how a well-chosen ω can have significant impact on the speed of convergence. This method can be accessed by using "sor" as the solution method within the program.

5 Conclusion

In conclusion, we have examined the implementation of five algorithms for solving matrix equations: Gaussian Elimination, Gaussian Elimination with Partial Pivoting, Jacobi's Method, The Gauss-Seidel Method, and a Successive Over-relaxation (SOR) Method. We have also looked at some example usage of this implementation. Future improvements to this work would include adding additional solution methods, finding a way to calculate optimal values for some user-provided constants, such as ω, adding consistency checks, and improving the matrix parsing algorithm.

References

[1] Faires J.D. Burden R.L., *Numerical analysis*, 9ed., Brooks Cole, 2010.