Math 451: Final Project

Therin Irwin

March 11, 2012

Abstract

Linear systems of equations are notoriously difficult to solve. Most numerical methods for solving linear systems are plagued by inaccuracy in special cases or just being too slow. In this report, four methods will be compared in accuracy when solving this linear system:

$$x_1 + \frac{1}{\varepsilon}x_2 = \frac{1}{\varepsilon}$$
$$x_1 + x_2 = 2$$

where ε is a very small number. The four methods compared are Naïve Gaussian Elimination, the Gauss-Seidel Method, Iterative Refinement, and Scaled Partial Pivoting. Each of these is plagued by their own problems. Naïve Gaussian Elimination, the standard for hand-solving linear systems, can actually return the wrong answer in some cases. The Gauss-Seidel method only works when the matrix representing the linear system is diagonally dominant. Finally, Iterative Refinement adds extra work onto the already slow Gaussian Elimination process.

In addition, this report will attempt to discover ways to determine if a linear system is fit to be solved by one of the methods described above.

1 Hand-Solving the System

To get a feel for how this linear system behaves, we first solve it symbolically by hand and compute its condition number. The Gaussian Elimination method was used for this:

$$\begin{bmatrix} 1 & 1/\varepsilon \\ 1 & 1 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1/\varepsilon \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1/\varepsilon \\ 0 & 1 - 1/\varepsilon \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1/\varepsilon \\ 2 - 1/\varepsilon \end{bmatrix} \rightarrow \begin{cases} x_1 - \dfrac{x_2}{\varepsilon} = \dfrac{1}{\varepsilon} \\ x_2 - \dfrac{x_2}{\varepsilon} = 2 - \dfrac{1}{\varepsilon} \end{cases}$$

$$x_2 = \frac{2 - \frac{1}{\varepsilon}}{1 - \frac{1}{\varepsilon}} \quad \rightarrow \quad \lim_{\varepsilon \to 0} \frac{2 - \frac{1}{\varepsilon}}{1 - \frac{1}{\varepsilon}} = 1$$

$$x_1 = \frac{1 - x_2}{\varepsilon} \quad \rightarrow \quad \lim_{\varepsilon \to 0} \frac{1 - x_2}{\varepsilon} = 1 \qquad (1.1)$$

Therefore, the correct solution to this system is $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ when ε is very small. The condition number can tell us how sensitive the solution to this linear system is. If the condition number is large, the system is very sensitive:

$$\kappa(A) = \|A\|\|A^{-1}\| \qquad (1.2)$$

$$\kappa(A) = \left\| \begin{matrix} 1 & 1/\varepsilon \\ 1 & 1 \end{matrix} \right\| \left\| \begin{matrix} \frac{\varepsilon}{\varepsilon - 1} & \frac{-1}{\varepsilon - 1} \\ \frac{-\varepsilon}{\varepsilon - 1} & \frac{\varepsilon}{\varepsilon - 1} \end{matrix} \right\| = \left(1 + \frac{1}{\varepsilon}\right) * 1 = 1 + \frac{1}{\varepsilon} \qquad (1.3)$$

As we can see, the condition number of the matrix $A = \begin{bmatrix} 1 & 1/\varepsilon \\ 1 & 1 \end{bmatrix}$ gets larger quickly as ε decreases in size. We can say then that as ε gets closer to zero, the system of equations gets more sensitive. Using Mathematica, we can observe this in action as ε goes from 10^{-6} to smaller values:

ε	Solution to System $Ax = b$	Condition Number
10^{-6}	$\{1.000001, 0.999999\}$	1,000,003
10^{-7}	$\{1.0000001, 0.9999999\}$	10,000,003
10^{-8}	$\{1.00000001, 0.99999999\}$	100,000,003
10^{-9}	$\{1.000000001, 0.999999999\}$	1,000,000,003
10^{-10}	$\{1.0000000001, 0.9999999999\}$	10,000,000,003

So, as ε gets smaller, we will have a harder time getting an accurate solution for this matrix.

2 Naïve Gaussian Elimination

Often the first choice when attempting to solve a system of equations, Gaussian Elimination is simple and effective for small matrices with relatively small integer numbers. However, as we shall see, for some matrices this method may return the

wrong answer. To determine whether the method will return the wrong answer, we will use the condition number of the matrix form of a linear system.

Using this method, we can determine at what point does the value of ε return the incorrect answer, and what condition number corresponds to this value of ε.

2.1 Implementation

Gaussian Elimination in essence involves making the coefficient matrix of the linear system into an upper triangular matrix. This makes arriving at the solution easy by using back-substitution. Consider that we start with this matrix, representing a system of 3 equations:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} j \\ k \\ l \end{bmatrix}$$

For each column of the coefficient matrix, we make every element below the diagonal zero by using matrix row operations. These row operations are also multiplied through to the b matrix in $Ax = b$. For our example matrix, we would end up with:

$$\begin{bmatrix} a & b & c \\ 0 & e - \frac{d}{a}b & f - \frac{d}{a}c \\ 0 & h - \frac{g}{a}b & i - \frac{g}{a}c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} j \\ k - \frac{d}{a}j \\ l - \frac{g}{a}j \end{bmatrix}$$

We then repeat the process using the submatrix:

$$\begin{bmatrix} e - \frac{d}{a}b & f - \frac{d}{a}c \\ h - \frac{g}{a}b & i - \frac{g}{a}c \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} k - \frac{d}{a}j \\ l - \frac{g}{a}j \end{bmatrix}$$

Once an upper triangular matrix is computed, the answer vector x is computed using back substitution.

2.2 Analysis of Incorrect Solution

Using the implementation described, a solution was determined from $\varepsilon = 10^{-6}\ to\ 10^{-17}$. The results of these experiments are shown in Fig. 2.1:

```
-6 : ----------------------------
Answer: [1.0000010000076145, 0.999998999999]
Cond #: 1000001.0
```

```
-7 : ----------------------------
Answer: [1.000000100582838, 0.99999989999999]
Cond #: 1.0000001E7
-8 : ----------------------------
Answer: [1.0, 0.99999999]
Cond #: 1.00000001E8
-9 : ----------------------------
Answer: [1.0, 0.999999999]
Cond #: 1.0000000009999999E9
-10 : ----------------------------
Answer: [1.0, 0.9999999999]
Cond #: 1.0000000001E10
-11 : ----------------------------
Answer: [1.0, 0.99999999999]
Cond #: 1.0000000000099998E11
-12 : ----------------------------
Answer: [1.0, 0.999999999999]
Cond #: 1.0000000000009999E12
-13 : ----------------------------
Answer: [1.0, 0.9999999999999]
Cond #: 1.0000000000000998E13
-14 : ----------------------------
Answer: [1.0, 0.99999999999999]
Cond #: 1.0000000000000098E14
-15 : ----------------------------
Answer: [1.0, 0.999999999999999]
Cond #: 1.0000000000000009E15
-16 : ----------------------------
Answer: [0.0, 1.0]
Cond #: 1.0E16
-17 : ----------------------------
Answer: [0.0, 1.0]
Cond #: 1.0E17
```

Fig. 2.1:
Determining when ε produces an incorrect result.

So, the last value of ε for which the solution is correct is 10^{-15}. It is quite easy to see from Fig. 2.1 that when the condition number becomes too large to be accurately represented, the answer becomes wrong.

In terms of our symbolic calculation (formula 1.1) earlier, this also makes sense. The algorithm first solves the equation for x_2 and then works backwards to x_1. In solving for x_2, the algorithm now finds that the solution is *exactly* one. Plugging 1 into the equation for x_1 gets you:

$$x_1 = \frac{1-1}{\varepsilon} = \frac{0}{\varepsilon} = 0$$

So the answer for x_1 is now calculated as zero. For larger values of ε, $1 - x_2$ would have resulted in a very small number. This very small number divided by ε (an equally small number) produced a result of exactly one.

2.3 Predicting Inaccuracies

Check for large values. Based on the results received from this linear system, one way to predict inaccuracies is to check whether the matrix representing the linear system has any extremely large or extremely small values. This check would occur after computing the reduced matrix needed to solve the problem, during the calculation of the x vector. The check would be very simple:

$$if\ \left[\ 1 + a_{ij} = a_{ij}\ OR\ 1 + a_{ij} = 1\ \right] then\ Print(\text{"Solution may be inaccurate."})$$

When implemented and tested using the linear system A, the check perfectly indicated when an inaccuracy occurred:

```
-6 : ----------------------------
Answer: [1.0000010000076145, 0.999998999999]
Cond #: 1000001.0
-7 : ----------------------------
Answer: [1.000000100582838, 0.99999989999999]
Cond #: 1.0000001E7
-8 : ----------------------------
Answer: [1.0, 0.99999999]
Cond #: 1.00000001E8
-9 : ----------------------------
Answer: [1.0, 0.999999999]
Cond #: 1.0000000009999999E9
-10 : ----------------------------
Answer: [1.0, 0.9999999999]
Cond #: 1.0000000001E10
-11 : ----------------------------
Answer: [1.0, 0.99999999999]
Cond #: 1.0000000000099998E11
-12 : ----------------------------
Answer: [1.0, 0.999999999999]
Cond #: 1.0000000000009999E12
-13 : ----------------------------
Answer: [1.0, 0.9999999999999]
Cond #: 1.0000000000000998E13
-14 : ----------------------------
Answer: [1.0, 0.99999999999999]
Cond #: 1.0000000000000098E14
-15 : ----------------------------
Answer: [1.0, 0.999999999999999]
Cond #: 1.0000000000000009E15
-16 : ----------------------------
Solution may be inaccurate.
```

```
Answer: [0.0, 1.0]
Cond #: 1.0E16
-17 : ----------------------------
Solution may be inaccurate.
Answer: [0.0, 1.0]
Cond #: 1.0E17
-18 : ----------------------------
Solution may be inaccurate.
Answer: [0.0, 1.0]
Cond #: 9.9999999999999987E17
```

Fig. 2.2: Testing inaccurate solution checks.

The same test was used on several other matrices, including an ill-conditioned Vandermonde matrix, and it performed as it should on these other matrices.

Solving the matrix. Another way that Gaussian Elimination solutions can be checked for inaccuracy is by simply solving the system of equations with the answer just computed. That is, the algorithm is passed matrices A and v, and the algorithm calculates the approximate answer $\tilde{x}$:

$$A\tilde{x} = b$$

But then the algorithm can plug $\tilde{x}$ back in:

$$r = A\tilde{x} - b$$

This will return an r which is close to zero if the answer $\tilde{x}$ is nearly correct. This test was also performed on several different matrices and performed equally well as the previous test.

It is worth noting that this test would take slightly longer than the previous. The difference in computational complexity would be trivial, however, compared to the time it takes to compute the answer using Gaussian Elimination.

3 Iterative Refinement

Iterative Refinement is a very easily implemented algorithm that is able to overcome most of the error involved in solving linear systems with Gaussian Elimination. An Iterative Refinement algorithm generally uses Gaussian Elimination as its first step to get a solution that is “in the ballpark.”

3.1 Implementation

Starting with a (possibly) incorrect answer from Gaussian Elimination (call it our "initial guess"), Iterative Refinement attempts on each iteration to solve the system using the previous guess, and then uses that answer as a basis for a "correction." Assume A is the matrix form of the linear system, and b is the right-hand side of the linear system. For each iteration n of the algorithm, the following happens:

$$r_n = b - Ax_n$$

$$Ad_n = r_n \ \ (solve\ for\ d_n)$$

$$x_{n+1} = x_n + d_n$$

The algorithm continues until the desired accuracy is achieved. The stopping criteria for this algorithm are very similar to that of Newton's Iteration, where the previous correction is compared to the current correction, and the iteration stops if the comparison is within some margin of error.

4 Scaled Partial Pivoting

Scaled Partial Pivoting is similar to Gaussian Elimination, except the pivot row is selected by choosing the maximum pivot entry in the leading column of the submatrix. Like Naïve Gaussian Elimination, it arrives at the correct answer without iteration. However, this method is just as slow, if not slower than, Gaussian Elimination.

4.1 Implementation

Implementing Scaled Partial Pivoting is similar to Gaussian Elimination, except that for Scaled Partial Pivoting a choice must be made about which row to use as the pivot row. (In Naïve Gaussian Elimination, we just choose the top row to be the pivot row). To do this, we first find the largest element in magnitude of each row of the matrix and store it in an array, call it the *scale* array. These are the elements we use to determine which pivot is the largest relative to the other elements in its row.

Next, we begin the Gaussian Elimination algorithm as usual, but instead of choosing the top row as the pivot, we instead search through the first column to find the largest relative pivot value:

$$\max_{j \le i \le n} \frac{|a_{ij}|}{scale[i]}$$

Here, n is the number of equations in the system, and j is the submatrix we are working in. Once the best pivot value is found, the best row is swapped so that it is at the top of the current submatrix. After this occurs, the Gaussian Elimination algorithm continues as normal.

It is time-consuming to actually swap rows of a matrix, especially in the case of a large matrix. Therefore, a *row index array* is used to represent the swapping of rows. At the beginning of the algorithm, this array, call it *rowSwap*, is initialized to contain $[0, 1, 2, \ldots, n]$. This represents the fact that no rows have been swapped yet, so the 0^{th} row corresponds to the 0^{th} stored row. But if we were to swap rows 4 and 0, *rowSwap* would contain $[4, 1, 2, 3, 0, \ldots, n]$, indicating that the 0^{th} row is now located at the 4^{th} storage position.

5 Gauss-Seidel Method

The Gauss-Seidel Method is an iterative method that uses previously computed values of the vector x to approach a better solution. The initial guess is generally not a problem for this method and is usually set to the zero vector. This method is certainly the fastest of those discussed, but it does not converge for all matrices.

5.1 Implementation

This method is fairly easy to implement compared to the others. One would start with a column vector $x = [0, 0, \ldots, 0]^T$ as the initial guess to the system. Then the algorithm computes successive $x_j^{(k)\prime}s$, where k is the current iteration of the vector x, and i is the i^{th} component of x:

$$x_i^{(k)} = \left[- \sum_{j=1<i}^{n} \left(\frac{a_{ij}}{a_{ii}} \right) x_j^{(k)} - \sum_{j>i}^{n} \left(\frac{a_{ij}}{a_{ii}} \right) x_j^{(k-1)} + \frac{b_i}{a_{ii}} \right]$$

Successive $k's$ are computed until the desired accuracy is achieved.

6 Comparison of Methods

Each of the methods examined have their own unique issues.

6.1 Iterative Refinement vs. Scaled Partial Pivoting

Speed of Implementations. Iterative Refinement involves successively computing Gaussian Elimination on a refinement of the linear system's x vector. Although the method converges quickly, this could easily become a problem with a large, very

ill-conditioned matrix, as Gaussian Elimination would be performed on the matrix during each step.

Scaled Partial Pivoting, on the other hand, does only one Gaussian Elimination, but uses techniques to avoid having to recalculate the solution as Iterative Refinement does. Therefore, Scaled Partial Pivoting is certainly the faster choice.

```
Gaussian Elimination:
  Taken: 7700 ns


Scaled Partial Pivoting:
  Taken: 8556 ns

Iterative Refinement:
  Taken: 13262 ns
```

Fig. 6.1: Average Speed of Methods using matrix *A*

As shown in Fig. 6.1, Iterative Refinement takes significantly longer than Scaled Partial Pivoting.

Accuracy of Implementations. Both Iterative Refinement and Scaled Partial Pivoting provide reasonably accurate solutions. Iterative Refinement can obviously be as accurate as the computer system allows by simply doing more iterations. For Scaled Partial Pivoting, however, we do not have this option, so this implementation tends to be slightly less accurate.

Scaled Partial Pivoting does not guarantee accuracy for all matrices. In implementing such an algorithm, one would have to take this into account. Fortunately, Scaled Partial Pivoting is adequate for most linear systems.

Fig. 6.2 shows the solutions for Iterative Refinement and Scaled Partial Pivoting for values of $\varepsilon < 10^{-6}$

```
-6 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.000001000001, 0.999998999999]
Iterative Refinement:
  Answer: [1.000001000001, 0.999998999999]
-7 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.00000010000001, 0.99999989999999]
Iterative Refinement:
  Answer: [1.00000010000001, 0.99999989999999]
-8 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.00000001, 0.99999999]
Iterative Refinement:
  Answer: [1.0000000100000002, 0.9999999899999998]
-9 : ----------------------------
```

```
Scaled Partial Pivoting:
  Answer: [1.000000001, 0.999999999]
Iterative Refinement:
  Answer: [1.000000001, 0.999999999]
-10 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.0000000001, 0.9999999999]
Iterative Refinement:
  Answer: [1.0000000001, 0.9999999999]
-11 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.00000000001, 0.99999999999]
Iterative Refinement:
  Answer: [1.00000000001, 0.99999999999]
-12 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.000000000001, 0.999999999999]
Iterative Refinement:
  Answer: [1.000000000001, 0.999999999999]
-13 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.0000000000001, 0.9999999999999]
Iterative Refinement:
  Answer: [1.0000000000001, 0.9999999999999]
-14 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.00000000000001, 0.99999999999999]
Iterative Refinement:
  Answer: [1.00000000000001, 0.99999999999999]
-15 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.0000000000000009, 0.999999999999999]
Iterative Refinement:
  Answer: [1.0000000000000009, 0.999999999999999]
-16 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.0, 1.0]
Iterative Refinement:
  Answer: [1.0000000000000004, 0.9999999999999998]
-17 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.0, 1.0]
Iterative Refinement:
  Answer: [1.0, 1.0]
-18 : ----------------------------
Scaled Partial Pivoting:
  Answer: [1.0, 1.0]
Iterative Refinement:
  Answer: [1.0, 1.0]
```

Fig. 6.2: Results for Iterative Refinement and Scaled Partial Pivoting.

6.2 Gauss-Seidel Implementation

When it converges, the Gauss-Seidel method is definitely the fastest method available for solving a linear system. Using a randomly generated 10x10 matrix, Gauss-Seidel performed on average 3 times faster than any of the other methods when it converged.

Unfortunately, this method did not converge for the matrix A discussed in Section 1. This is because the Gauss-Seidel method is only guaranteed to converge for diagonally dominant or symmetric positive definite matrices.

Therefore, unless the linear system being solved is guaranteed to be positive definite or diagonally dominant, Gauss-Seidel should not be used.

7 Conclusion

Based on the results reported above, if we were looking to solve a very ill-conditioned linear system, we would use either the Scaled Partial Pivoting or the Gauss-Seidel method. If the matrix representing the linear system is guaranteed to converge for the Gauss-Seidel method, then this would be the method to use as it is much faster than Scaled Partial Pivoting.

As stated above, however, Scaled Partial Pivoting does not necessarily produce the correct answer for all matrices, so it is still beneficial to use the inaccuracy checks described in Section 2.2 to make sure the method is returning the correct answer.

```java
import java.util.Arrays;
import java.util.Scanner;

public class GaussianElimination {

    public static MatrixOps.Answer solve(double[][] mx, double[] vx) {
        boolean inacc = false;
        MatrixOps.Answer a = new MatrixOps.Answer();
        a.ans = new double[vx.length];
        double[][] m = new double[mx.length][];
        double[] v = new double[vx.length];

        MatrixOps.array2DCopy(mx, m);
        System.arraycopy(vx, 0, v, 0, vx.length);

        // Reduce the Matrix.
        for (int i = 0; i < m[0].length; i++) {
            double pivot = m[i][i];

            // Check if pivot value is 1, if not, divide row by pivot.
            if (pivot != 1) {
                MatrixOps.rowDiv(m, i, pivot);
                v[i] /= pivot;
            }

            // Zero all values below the pivot.
            for (int j = i + 1; j < m[0].length; j++) {
                double fix = m[j][i];
                if (fix == 0)
                    continue;

                MatrixOps.rowMultSub(m, i, j, fix);
                v[j] -= v[i] * fix;
            }
        }

        // Compute the x vector.
        for (int i = m[0].length - 1; i >= 0; i--) {
            double subAns = v[i];

            for (int j = i + 1; j < m[0].length; j++) {

                // Check if the solution will be inaccurate.
                inacc = inacc ||
                 (2 + m[i][j] == 1 + m[i][j] || 1 + m[i][j] == 1);

                subAns -= m[i][j] * a.ans[j];
                a.mults++;
            }

            a.ans[i] = subAns;
        }
```

```
        double[] chk = MatrixOps.mult(mx, a.ans);
        chk = MatrixOps.sub(chk, vx);

        for (int i = 0; i < chk.length; i++)
            inacc = inacc || Math.abs(chk[i]) > MatrixOps.epsilon;

        if (inacc)
            System.out.println("Solution may be inaccurate.");

        return a;
    }

    public static MatrixOps.Answer iterativeRefinement(double[][] m,
     double[] v, MatrixOps.Answer a) {
        boolean stop = false;
        double[] ax = new double[m.length], r = new double[m.length];
        double[] temp = new double[m.length];

        for (int x = 0; x < 10; x++) {
            // Compute Ax
            ax = MatrixOps.mult(m, a.ans);

            // Compute r = v - Ax
            r = MatrixOps.sub(v, ax);

            // Solve Ar = v; Add d to x
            temp = solve(m, r).ans;
            a.ans = MatrixOps.add(a.ans, temp);

            stop = true;
            for (int i = 0; i < r.length; i++)
                stop = stop && !(Math.abs(temp[i]) > MatrixOps.epsilon);

            if (stop) break;
        }

        return a;
    }

    public static void main(String[] args) {
        int size = 2, value = -6;
        double e = Math.pow(10, -6);
        long time;
        MatrixOps.Answer a;
        double[] vc = {1, 2}, v = new double[size];
        double[][] mc = {{1, 1}, {1, 1}}, m = new double[size][];
        Scanner sc = new Scanner(System.in);

        while (value > -32) {

            for (int i = 0; i < vc.length; i++)
                v[i] = vc[i];
```

```
            for (int i = 0; i < mc.length; i++) {
                m[i] = new double[size];

                for (int j = 0; j < mc.length; j++)
                    m[i][j] = mc[i][j];
            }

            System.out.println(value + " : ------------------------------");

            System.out.println("\nGaussian Elimination:");
            time = System.nanoTime();
            a = GaussianElimination.solve(m, v);
            time = System.nanoTime() - time;
            System.out.println("  Taken: " + time);

            System.out.print("  Answer: ");
            System.out.println(Arrays.toString(a.ans));

            System.out.println("Scaled Partial Pivoting:");
            a = ScaledPartialPivoting.solve(m, v);

            System.out.print("  Answer: ");
            System.out.println(Arrays.toString(a.ans));

            System.out.println("Iterative Refinement:");
            a = GaussianElimination.iterativeRefinement(m, v, a);

            System.out.print("  Answer: ");
            System.out.println(Arrays.toString(a.ans));

            System.out.println("\nGauss-Seidel:");
            time = System.nanoTime();
            a = GaussSeidel.solve(m, v);
            time = System.nanoTime() - time;
            System.out.println("  Taken: " + time);

            System.out.print("  Answer: ");
            System.out.println(Arrays.toString(a.ans));

            e /= 10;
            value--;
        }
    }
}
```

```
public class MatrixOps {

    public static final double epsilon = Math.pow(10, -10);

    static class Answer {
        public int mults = 0;
        public double[] ans;
        public double cond;
    }

    public static void rowDiv(double[][] m, int row, double scalar) {
        for (int j = 0; j < m[0].length; j++)
            m[row][j] /= scalar;
    }

    public static void rowMultSub(double[][] m, int rowSrc, int rowDest,
     double scalar) {
        for (int k = 0; k < m[0].length; k++)
            m[rowDest][k] -= m[rowSrc][k] * scalar;
    }

    public static double innerProduct(double[] a, double[] b) {
        double ans = 0d;

        for (int i = 0; i < a.length; i++)
            ans += a[i] * b[i];

        return ans;
    }

    public static double[] mult(double[][] m, double[] v) {
        double[] ans = new double[v.length];

        for (int i = 0; i < v.length; i++)
            ans[i] = innerProduct(m[i], v);

        return ans;
    }

    public static double[] sub(double[] v1, double[] v2) {
        double[] r = new double[v1.length];

        for (int i = 0; i < r.length; i++)
            r[i] = v1[i] - v2[i];

        return r;
    }

    public static double[] add(double[] v1, double[] v2) {
        double[] r = new double[v1.length];

        for (int i = 0; i < r.length; i++)
```

```
            r[i] = v1[i] + v2[i];

        return r;
    }

    public static void array2DCopy(double[][] src, double[][] dest) {
        for (int i = 0; i < src.length; i++) {
            dest[i] = new double[src.length];

            for (int j = 0; j < src.length; j++)
                dest[i][j] = src[i][j];
        }
    }
}
```

```java
public class ScaledPartialPivoting {

    public static MatrixOps.Answer solve(double[][] mx, double[] vx) {
        MatrixOps.Answer a = new MatrixOps.Answer();
        a.ans = new double[vx.length];
        double[][] m = new double[mx.length][];
        double[] v = new double[vx.length];

        double[] scaled = new double[vx.length];
        int[] rp = new int[vx.length];
        double pivot = 0d;
        int bestRow = 0, rowTemp = 0;

        MatrixOps.array2DCopy(mx, m);
        System.arraycopy(vx, 0, v, 0, vx.length);

        // Find the infinity norm of each row.
        for (int i = 0; i < vx.length; i++) {
            double max = 0d;
            for (int j = 0; j < vx.length; j++)
                max = max > Math.abs(m[i][j]) ? max : Math.abs(m[i][j]);

            scaled[i] = max;
        }

        // Set the inital row locations.
        for (int i = 0; i < vx.length; i++)
            rp[i] = i;

        // Iterate through each pivot element.
        for (int i = 0; i < vx.length; i++) {
            pivot = 0d;
            bestRow = i;

            // Find best pivot element.
            for (int j = i; j < vx.length; j++)
                if (pivot < Math.abs(m[rp[j]][i]) / scaled[rp[j]]) {
                    pivot = Math.abs(m[rp[j]][i]) / scaled[rp[j]];
                    bestRow = j;
                }

            pivot = m[rp[bestRow]][i];
            rowTemp = rp[i];
            rp[i] = rp[bestRow];
            rp[bestRow] = rowTemp;

            // Check if pivot value is 1, if not, divide row by pivot.
            if (pivot != 1) {
                MatrixOps.rowDiv(m, rp[i], pivot);
                v[rp[i]] /= pivot;
            }
```

```
            // Zero all values below the pivot.
            for (int j = i + 1; j < m[0].length; j++) {
                double fix = m[rp[j]][i];
                if (fix == 0)
                    continue;

                MatrixOps.rowMultSub(m, rp[i], rp[j], fix);
                v[rp[j]] -= v[rp[i]] * fix;
            }
        }

        // Compute the x vector.
        for (int i = m[0].length - 1; i >= 0; i--) {
            double subAns = v[rp[i]];

            for (int j = i + 1; j < m[0].length; j++) {
                subAns -= m[rp[i]][j] * a.ans[j];
                a.mults++;
            }

            a.ans[i] = subAns;
        }

        return a;
    }
}
```

```java
public class GaussSeidel {

    public static MatrixOps.Answer solve(double[][] mx, double[] vx) {
        MatrixOps.Answer ans = new MatrixOps.Answer();
        double[] xAns = new double[vx.length];
        double[] xPrevAns = new double[vx.length];

        for (int k = 0; k < 120; k++) {
            for (int i = 0; i < vx.length; i++) {
                double subAns = 0d;

                for (int j = 0; j < i; j++)
                    subAns += mx[i][j] * xAns[j];
                for (int j = i + 1; j < vx.length; j++)
                    subAns += mx[i][j] * xPrevAns[j];

                xAns[i] = (-subAns + vx[i]) / mx[i][i];
            }

            System.arraycopy(xAns, 0, xPrevAns, 0, xAns.length);
        }

        ans.ans = xAns;
        return ans;
    }
}
```