PYU44C01 Assignment 2

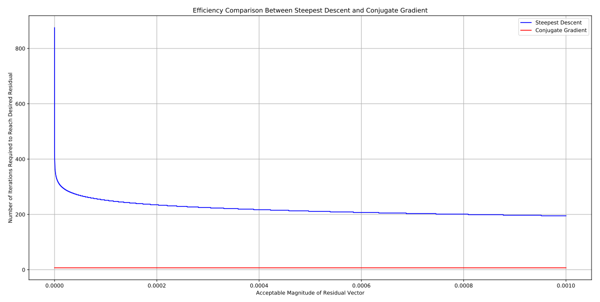
Alexander Hackett
15323791
Prof. Charles Patterson

Methodology

The script *ahackett_Assignment_3.py* made use of the steepest descent and conjugate gradient methods in order to determine a solution for the system of linear equations $A \cdot x = b$, where A was the 13×13 matrix provided in the file *A_matrix.txt* and b was the vector [1 2 3 4 5 6 7 8 9 10 11 12 13]. The purpose of both algorithms was to find the value of x that gives the minimum value for $f(x) = \frac{1}{2}x^T \cdot A \cdot x - b \cdot x$. This value is the solution to the matrix equation $A \cdot x = b$. The script takes an 'exact' solution in the method solveExact(), by inverting A with the standard linalg.inv() function, and then solving $x = A^{-1} \cdot b$ directly. This solution is used to confirm that the two iterative solutions produce the correct answer. The steepest descent algorithm is implemented in the method steepestDescent(). This method works by taking an initial estimate for x: x_0, the 13 length zero vector (the same initial guess for both algorithms), and computing an initial residual vector $r = b - A \cdot x_0$. The magnitude of this vector $\Delta = r^T \cdot r$ is utilized as a measurement of the error, or 'distance' to the solution, and the direction of the residual vector is the direction of (initial) steepest descent. While the current iteration is below the maximum (set to 10^6), and while the magnitude of the residual is above the error tolerance, the algorithm enters the following loop:
First, determine α as $\frac{\Delta}{r^T \cdot A \cdot r}$. Then determine $\alpha \cdot r$ and add this to x_0 to produce x_1, the first 'step' towards the solution. Next, update the residual vector so that $r = b - A \cdot x$. Finally, determine the new error/magnitude of the residual, $\Delta = r^T \cdot r$. Repeat this loop until Δ is at or below the tolerance desired, and x will contain the solution vector, and r will contain the residual vector.
The conjugate gradient algorithm, implemented in the method conjGrad() is largely similar. First, determine the initial residual as before, and then set the direction vector, d to r. Then enter a similar loop as before, with the same exit conditions. Determine the descent direction as $\alpha = \frac{\Delta}{d^T \cdot d}$, hence ensuring that each search direction is conjugate to the last one. Then, update x as $x_1 = x_0 + \alpha \cdot d$, as the first conjugate 'step' towards the solution. Recompute the residual vector r as $b - A \cdot x$ as before, and compute its new magnitude, Δ_i as $\Delta_i = r^T \cdot r$, were r is the updated residual vector. Then determine β_i as $\frac{\Delta_i}{\Delta_{i-1}}$. Analogously to updating the solution vector, now update the direction vector d_i as $d_i = r + \beta \cdot d_{i-1}$. Again, continue this loop until the maximum number of iterations is reached, or until the error, Δ_i reaches the desired tolerance.

Results

The **MatEqnSol** class containing the methods discussed in the methodology section were utilized in order to compare the efficiency of the steepest descent algorithm and the conjugate gradient method. Since direct timing would depend on exact implementation and hardware, the number of steps (number of times the main loop as described) required to reach the desired accuracy (magnitude of the residual vector) in each algorithm was utilized as a metric of efficiency. The following plot shows the number of steps of the main loop of each algorithm required to reach the desired accuracy as a function of that desired accuracy.



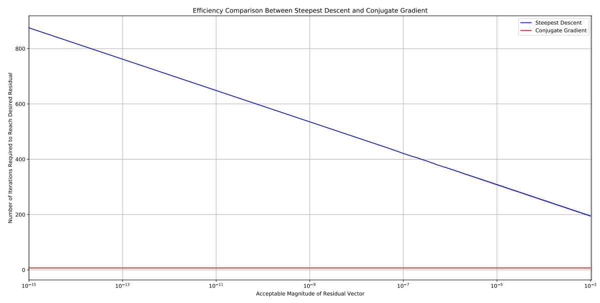


Figure 1: Linear and Semi-Log Plots of the Efficiency of the Steepest Descent and Conjugate Gradient Algorithms. Even at low tolerances, the conjugate gradient algorithm outperforms the steepest descent algorithm, but as the tolerance is tightened further towards machine precision the performance of the steepest descent dis-improves rapidly, while the conjugate gradient's performance is constant.

Utilizing an initial $x_0 = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]$ with a requirement to reach a magnitude of the residual of 10^{-16}, the steepest descent algorithm produced a solution vector:

```
x = [ 1.39139776 10.00790465 4.83404805 5.75386526 6.67368248 -46.80162505 -20.90228751 -21.55535754 -22.20842757 47.12994921 27.96943296 29.21609042 30.46274788]
```

requiring 931 steps.
The conjugate gradient method, with the same x_0 and the same residual vector magnitude requirement, produced the same solution vector:

```
x = [ 1.39139776 10.00790465 4.83404805 5.75386526 6.67368248 -46.80162505 -20.90228751 -21.55535754 -22.20842757 47.12994921 27.96943296 29.21609042 30.46274788]
```

requiring just 7 steps in order to reach the required tolerance.
Finally, utilizing the direct inverse produced the same solution vector:

```
x = [ 1.39139776 10.00790465 4.83404805 5.75386526 6.67368248 -46.80162505 -20.90228751 -21.55535754 -22.20842757 47.12994921 27.96943296 29.21609042 30.46274788]
```

Thus showing the ability of these algorithm in solving these systems of linear equations, as well as demonstrating the improved efficiency of the conjugate gradient method compared to the steepest descent algorithm.