Blake Van Dyken

Assignment 3

I AM USING MY 1-WEEK NO LATE PENALTY ON THIS ASSIGNMENT

Methods Used:

- Python's time.time() from time module
    - Used time.time() to calculate the runtime of each matrix multiplication by subtracting the end time=time.time() from the start time=time.time()
    - time.time() is how many seconds have passed since epoch
- Jacobi's iterative method

Jacobi Pseudocode

choose an initial guess x_0
for $k = 0,1,2, \ldots$
    for $i = 1{:}n$
        $sum = 0.0$
        for $j - 1{:}n,\ j \neq i$
            $sum = sum + A(i,j) * x_k(j)$
        end for
        $x_{k+1}(i) = (b(i) - sum)/A(i,i)$
    end for
    check convergence and continue if needed
end for

- GaussSeidel iterative method

choose an initial guess x_0
for $k = 0,1,2,...$
    for $i = 1:n$
        $sum = 0.0$
        for $j = 1:i-1$
            $sum = sum + A(i,j) * x_{k+1}(j)$
        end for
        for $j = i+1:n$
            $sum = sum + A(i,j) * x_k(j)$
        end for
        $x_{k+1}(i) = (b(i) - sum)/A(i,i)$
    end for
    check convergence and continue if needed
end for

- Successive over-relaxation method
    - Same as Gauss-Siedel however includes scalar w=1.05 with equation to help it converge faster if it is chosen correctly

The SOR method extends Gauss-Seidel to converge more quickly by including a scalar relaxation factor ω:

$$\boldsymbol{Ax} = b$$
$$(\boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U})x = b$$

$$\omega \boldsymbol{L}x + \omega \boldsymbol{D}x + \omega \boldsymbol{U}x = \omega b$$
$$\omega \boldsymbol{L}x + \omega \boldsymbol{D}x + \omega \boldsymbol{U}x + \boldsymbol{D}x = \omega b + \boldsymbol{D}x$$
$$\omega \boldsymbol{L}x + \boldsymbol{D}x = \omega b + \boldsymbol{D}x - \omega \boldsymbol{D}x - \omega \boldsymbol{U}x$$

Results and Figures:

```
n =  128
Python implementation: 0.00298333 seconds
Matlab's implementation (matrix_direct.py): 1.04600024 seconds
Optimized inner most loop of Matlab's (matrix_dot.py): 0.02599859 seconds
Outer products method (matrix_outer): 0.00800204 seconds
Saxpy operation method (matrix_saxpy.py): 0.06099844 seconds
Matrix vector products method (matrix_vector.py): 0.00700045 seconds
n =  512
Python implementation: 0.00300241 seconds
Matlab's implementation (matrix_direct.py): 69.57814312 seconds
Optimized inner most loop of Matlab's (matrix_dot.py): 0.61098456 seconds
Outer products method (matrix_outer): 1.08399940 seconds
Saxpy operation method (matrix_saxpy.py): 1.69000220 seconds
Matrix vector products method (matrix_vector.py): 0.03699946 seconds
n =  1024
Python implementation: 0.02400446 seconds
Matlab's implementation (matrix_direct.py): 600.00000000 seconds
Optimized inner most loop of Matlab's (matrix_dot.py): 5.62400007 seconds
Outer products method (matrix_outer): 13.48499990 seconds
Saxpy operation method (matrix_saxpy.py): 27.78621173 seconds
Matrix vector products method (matrix_vector.py): 0.48400044 seconds
n =  4096
Python implementation: 0.95404100 seconds
Matlab's implementation (matrix_direct.py): 600.00000000 seconds
Optimized inner most loop of Matlab's (matrix_dot.py): 600.00000000 seconds
Outer products method (matrix_outer): 600.00000000 seconds
Saxpy operation method (matrix_saxpy.py): 600.00000000 seconds
Matrix vector products method (matrix_vector.py): 38.78702450 seconds
```

Looking at the console output from my program above, any program that says 600 seconds are the calculations that took more than 10 minutes. I calculated these on my **HP Pavilion** laptop which has a processor that is an **Intel Core i5-1035G1 CPU @ 1GHz**. Looking at each n iteration, we can see that python's implementation always performs the fastest in each case, I think this is because it is biased toward my machine, and python's method is more integrated with what I am running it on thus it is faster. When n is small, it does not really matter which method you use as they all run in under a second. When n gets larger, we see that matlab's implementation is the one that starts to perform the worst compared to everything else. When n is 4096, we see

that the only other viable method besides pythons built in one is the matrix-vector products. I speculate that since all these methods are not as integrated as python's, they all should perform worse as python has access to my computer resources and such to make its computations faster.

```
direct method solution(numpy):  [ 1.00000000e+01  5.83683362e+00  3.99661568e+00  3.012
59585e+00
  2.30084885e+00  1.75550515e+00  1.30837574e+00  9.33349212e-01
  6.09218500e-01  3.24528687e-01  7.45530902e-02 -1.49562855e-01
 -3.51769691e-01 -5.30987429e-01 -6.91432722e-01 -8.34469577e-01
 -9.61301741e-01 -1.07212966e+00 -1.16814199e+00 -1.24877626e+00
```

```
jacobi converged after 100 iterations and 104.19898653030396  seconds
jacobi matrix:  [ 1.00000000e+01  5.88952884e+00  4.08209728e+00  3.13420319e+00
  2.46679035e+00  1.97300626e+00  1.58389881e+00  1.27464085e+00
  1.02265119e+00  8.16035510e-01  6.48016230e-01  5.09291893e-01
  3.95133603e-01  3.05021571e-01  2.32592994e-01  1.74416291e-01
  1.27435594e-01  8.98737996e-02  5.97570214e-02  3.67075724e-02
  1.96496624e-02  7.32899523e-03 -8.43658454e-04 -5.73002568e-03
```

```
guass-seidel converged after 70 iterations and 72.33157753944397  seconds
guass-seidel matrix:  [ 1.00000000e+01  3.60342527e+00  1.19825143e+00  5.02028758e-01
  1.03397815e-01  4.31902391e-02  2.53656549e-02  1.17375522e-02
  6.82770972e-03  2.33145987e-03  8.83066151e-04  4.37408830e-04
  9.34566134e-05  3.95520894e-05  2.32296473e-05  1.10019086e-05
  4.26951086e-06  2.40153287e-06  1.17495923e-06  4.99647334e-07
```

```
SOR with w=1.05 converged after 100 iterations and 104.4592227935791  seconds
SOR w=1.05 matrix:  [ 1.05000000e+01  4.39649484e+00  2.00312731e+00  1.03877479e+00
  2.98785929e-01  1.64470723e-01  1.16576584e-01  6.79871252e-02
  4.57566452e-02  1.88961263e-02  8.65571084e-03  4.83039644e-03
  1.34069242e-03  7.43048651e-04  5.20727346e-04  2.72298066e-04
  1.00202171e-04  3.15675291e-05 -2.10119167e-05 -5.33305091e-05
 -7.08177615e-05 -8.86440436e-05 -8.06646064e-05 -1.24857580e-04
```

My tolerance for the methods was set to tol=0.0001, however, I tried changing it a little bit and got similar results where the SOR performed worse than the gauss-seidel method. First, looking at the Jacobi method, it converged after 100 iterations and 104 seconds, which was the slowest of the three methods. The gauss-seidel method

converged after 70 iterations, **which was the fastest of the three in iterations and CPU time**, but it should be faster than Jacobi because it uses half as much storage to compute the values. However, the SOR method improves upon the Gauss method and is faster if you choose a good w value, but in my code, it converged slower at 100 iterations and 104 seconds because I timed it out as it would have taken a little bit longer. This is possible because of a bad w value or because the formula is implemented possibly in the wrong way.

Lastly, looking at the direct method of solving, it solved the problem very very fast, only in a couple of seconds. This method is the fastest because it has already been refined by someone else and also is integrated with python to perform better. Finally, it looks like the Jacobi method got the closest to converging to it possibly because I need a lower tolerance value, but my computer would start to slow down and it was not reasonably possible to do it with a tolerance of 1e-10.

Sources:

All my code is submitted with the project and on GitHub here:

https://github.com/iPupkin/Theory-3200