Alex Merino

MTH 3312

Due: 10/16/2024

All code documented in the python files as well.

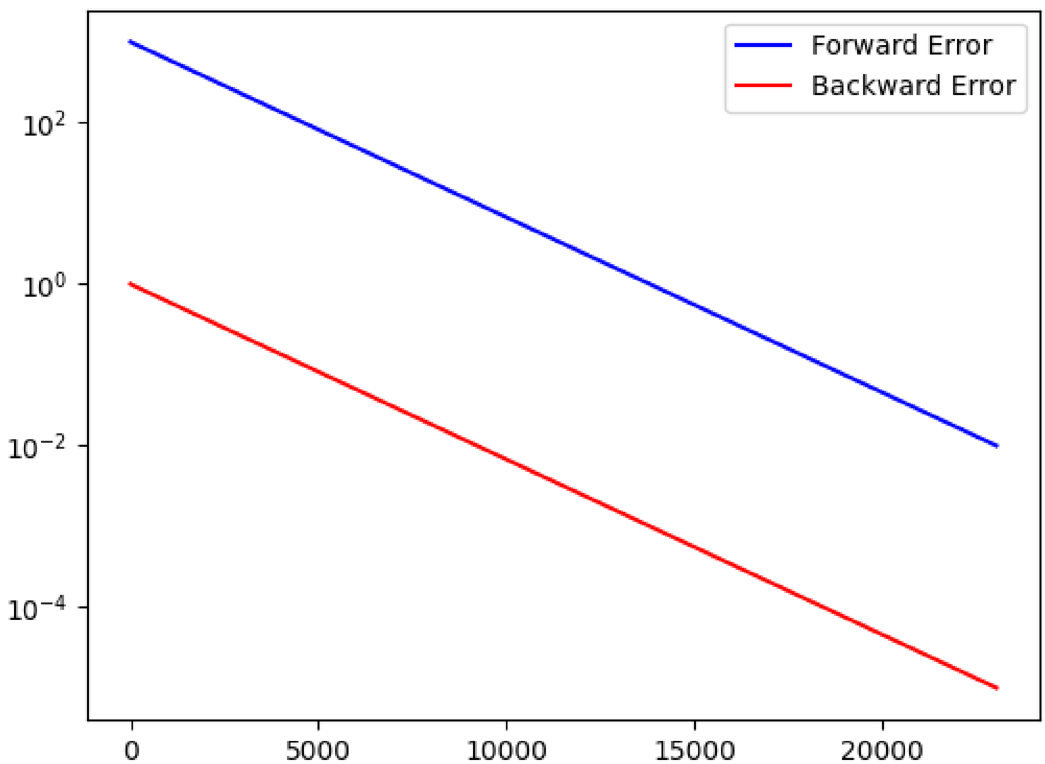
A)

The array below is the output of part A, when multiplied with A, we get the vector of all ones, which was our starting vector. The way the array was solved was from scipy, with the scipy.sparse.linalg.spsolve function. This is Python's equivalent to MATLAB's backslash function.

```
[ 31.1267292   61.28458513  90.50372565 118.81336989 146.2418275
 172.81652694 198.56404291 223.51012292 247.67971305 271.09698289
 293.78534972 315.7675019  337.06542158 357.70040668 377.69309219
 397.06347079 415.83091286 434.01418584 451.63147301 468.70039165
 485.23801069 501.26086773 516.78498564 531.82588854 546.39861733
 560.51774473 574.19738988 587.45123242 600.29252619 612.73411249
 624.7884329  636.46754174 647.78311812 658.74647763 669.36858361
 679.66005817 689.63119279 699.29195861 708.65201638 717.72072617
 726.50715669 735.02009436 743.26805212 751.25927794 759.00176304
 766.5032499  773.77124001 780.81300136 787.63557571 794.24578564
 800.65024135 806.8553473  812.86730861 818.69213722 824.33565796
 829.80351437 835.10117429 840.23393538 845.20693041 850.02513237
 854.69335946 859.21627991 863.59841664 867.84415179 871.95773109
 875.94326812 879.80474842 883.54603347 887.17086455 890.68286649
 894.08555131 897.38232167 900.57647435 903.67120351 906.66960387
 909.57467384 912.38931848 915.11635244 917.75850275 920.31841157
 922.79863879 925.20166466 927.52989218 929.78564961 931.97119268
 934.08870694 936.14030991 938.12805319 940.05392452 941.91984978
 943.72769489 945.47926769 947.17631976 948.82054815 950.41359709
 951.95705963 953.45247922 954.9013513  956.30512472 957.66520327
 958.98294703 960.25967373 961.4966601  962.69514313 963.85632131
 964.98135581 966.07137167 967.12745889 968.15067358 969.14203894
 970.10254633 971.03315628 971.93479938 972.80837728 973.65476355
 974.4748046  975.26932044 976.03910561 976.78492988 977.50753908
 978.20765582 978.88598021 979.54319059 980.17994416 980.79687767
 981.39460806 981.97373305 982.53483178 983.07846534 983.60517737
 984.11549458 984.60992727 985.0889699  985.5531015  986.00278619
 986.43847368 986.86059963 987.26958619 987.66584234 988.04976432
 988.42173607 988.78212956 989.13130517 989.46961209 989.79738862
 990.11496255 990.42265143 990.72076297 991.00959526 991.28943716
 991.56056849 991.82326039 992.07777555 992.32436848 992.56328579
 992.79476637 993.01904173 993.23633613 993.44686686 993.65084446
 993.84847291 994.03994982 994.22546669 994.40520903 994.57935657
 994.74808347 994.91155845 995.069945   995.22340148 995.37208137
 995.51613334 995.65570144 995.79092525 995.92193998 996.04887665
 996.17186219 996.2910196  996.40646803 996.51832293 996.62669615
 996.73169606 996.83342767 996.93199271 997.02748974 997.12001426
 997.2096588  997.29651299 997.3806637  997.46219507 997.54118864
 997.61772339 997.69187587 997.76372023 997.8333283  997.9007697
```

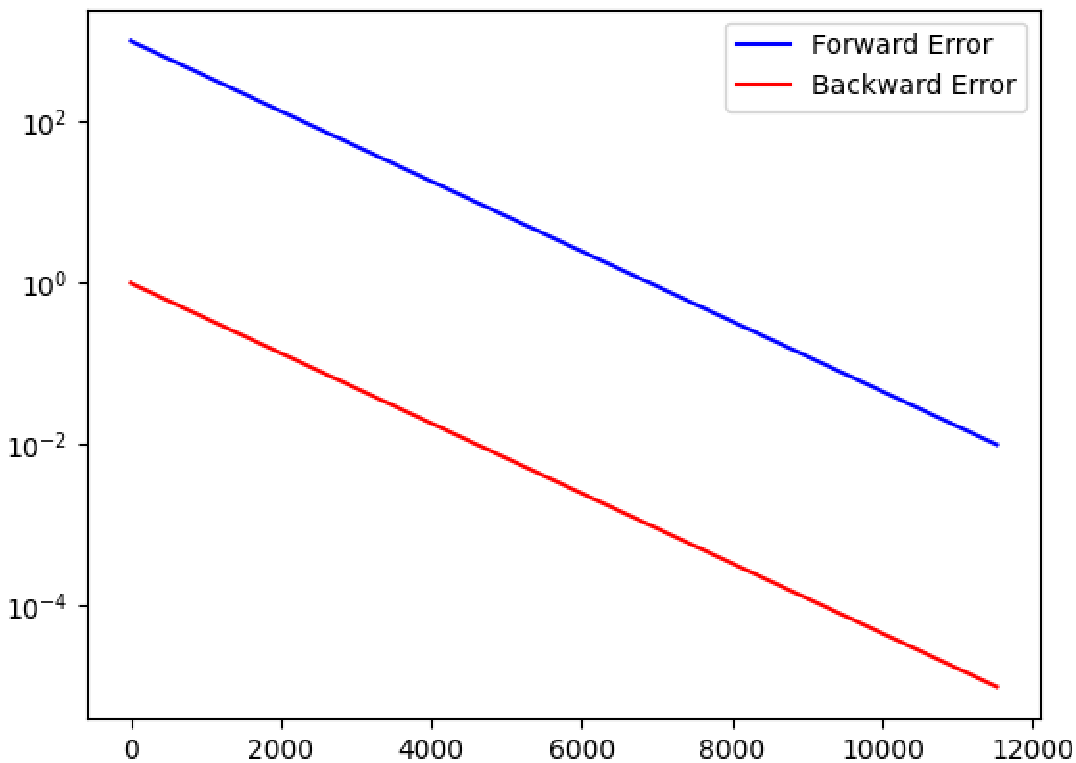
B)

For part B, I started with creating the upper triangle, lower triangle, and diagonal matrices. I then, used a while loop to iterate until the backward error of 10^{-5} was reached. In the while loop, I solve for the next iteration of x using Jacobi's method then calculate the errors. Looking at the graph of the errors, the slope of the line tells us that Jacobi has a linear convergence speed. The exact number is dependent on the spectral radius, but we cannot calculate that. It took around 24000 iterations to converge though, which is good to note when comparing to other methods.



C)

For the Gauss-Seidel Method, I created a matrix that included the diagonal and lower triangle, which simulates the matrix D + L. I then created the upper triangle matrix. Then starting with x = 0, I used a while loop to iteratively solve for x using the Gauss-Seidel method. The resulting graph shows how the convergence rate of Gauss-Seidel is also linear. However, compared to Jacobi's method, it converged in under 12,000 iterations which is twice as fast as Jacobi's method.



D)

The graph below plots the iterations it took to converge based on the omegas given. When omega was 0.1 and 0.2, after 10^5 iterations, it still did not converge. In the code, I created the lower, upper, and diagonal matrices. Then created a list of all omegas we want to test. Then in a loop for each omega, we iteravely solve for x using the SOR method with that specified omega. When omega =1 the method is just Gauss-Seidel, and converges after around 12,000 iterations just like in part C.

The graph also falls in line with how SOR theoretically convergenes. When omega < 1, it takes longer to convergce since we are shirnking the value every time we solve for x, and when omega > 1, it converges quicker since we are increasing the value of x by a factor of omega. When omega = 1.9 it converged the fastest, in only 607 steps. The convergence speed is about linearly getting better when increasing omega by 0.1.