Assessment Submission Form

Student Name(s)	**Peter Adam, Kieron Ellis, Andy McSweeney**
Student Number(s)	**16201859, 13560567, 16203177**
Assessment Title	**Numerical Analysis and Software – Programming Assignment: Linear Equations**
Module Code	**MIS40530**
Module Title	**Numerical Analysis and Software**
Tutor	**Professor Seán McGarraghy**
Date Submitted	**21-November-2016**
Office use only **Date Received**	
Office use only **Grade/Mark**	

Signed – Peter Adam ……………………… **Date** ……………………………………………

Signed – Kieron Ellis ……………………… **Date** ……………………………………………

Signed – Andy McSweeney ……………………… **Date** ……………………………………………

Numerical Analytics and Software

Programming Assignment: Linear Equations

Successive Over-Relaxation

Black Scholes

Peter Adam – 16201859 – MSc Business Analytics Full Time

Kieron Ellis – 13560567 – MSc Business Analytics Full Time

Andy McSweeney – 16203177 – MSc Business Analytics Full Time

Abstract

The following paper explores the issues present in programming and implementing Successive Over-Relaxation as a method of solving the Black Scholes system of linear equations. The exercise was undertaken by a group of 3 students at the Michael Smurfit Graduate School of Business and code written in Python 3. A discussion of the use of Partial Differential Equations in finance, and the theoretical and mathematic issues of Successive Over-Relaxation (SOR) is presented first. As a result of the issues raised in this discussion, a Detailed Design Document outlines what choices were made in developing the software and how they link into theoretical issues. On top of this, optimization ideas were discussed and tested, both those relating to Python 'Best Practices' and optimization of parameters. Unit and Functional testing strategies, as well the project implementation methodology is also outlined, with additional information on both these topics contained in Appendices A, B and C. Finally, the thoughts on the project from each of the contributing authors are included.

Overview, Discussion & Conclusions

Discussion of PDE's in Finance - Peter

The main mathematical issues we encountered with the SOR algorithm were the strict input matrix parameters that are required for the algorithm to successfully converge. To ensure convergence, the input matrix had to diagonally dominant or not diagonally dominant but with eigenvalues with absolute values less than 1. In both cases, no zeros could be present on the matrix diagonal also. These strict parameters restricted the number of matrices we could solve using our program.

To capture these parameters, we perform a number of input checks on the matrix before it is run through the algorithm. For small matrices, this is a very quick way to ensure the matrix is correctly conditioned for the algorithm. As matrices grow larger however, and with initial conditions unknown, while these checks will still capture ill conditioned matrices, the computational time will grow and may lead to wasted time on matrices that our program cannot solve.

Convergence of the residual to zero is useful to check for because the residual of a well-conditioned matrix will converge to zero when the current x is the solution to $Ax = b$. However, this usefulness doesn't extend to an ill-conditioned matrix which will take many more iterations before it's residual converges to zero. If a matrix is ill-conditioned, a small residual does not mean that the relative error in the solution is small. As such, a residual larger than zero is more appropriate for a more robust algorithm. We decided to use a residual tolerance of the square root of relative condition of the matrix times 1×10^{-10}. As a matrix's condition increases in size, the residual tolerance will increase even faster.

The matrix used in *code/sample_inputs/sample_mtx_diag_dom1.mtx* has a relative condition of $6{,}059{,}745{,}349$. When a residual convergence tolerance of 0 is used, the matrix will reach the maximum iterations before converging sufficiently. But a large residual tolerance allows the algorithm to converge within the maximum number of iterations. A residual tolerance of $\sqrt{6{,}059{,}745{,}349}\times10^{-10}$ causes the algorithm to converge after 31 iterations. However, multiplying the tolerance of 1×10^{-10} by the condition to the power of 0.5 results in a less accurate solution than allowing the algorithm to reach the maximum number of iterations. Taking the condition to the power of 0.4746375 increases the number of iterations from 31 to 34, but with 0.4746376 causing the algorithm to reach the maximum number of iterations, finding a power which brings the number of iterations to 50 or 60 would be a waste of computation time.

We considered using the *code/sor_modules/condition.py* to calculate the relative condition of a matrix and use this information to alter the residual tolerance of our SOR algorithm accordingly, but we felt that the extra computational complexity was too large.

What matrix norm(s) might be useful and why? - Peter

Would scaling be helpful? - Andy

If a matrix A, has a zero on its diagonal it will fail a test in the function *value_checks()* which checks whether A has a zero on the diagonal. A zero on the diagonal would result in SOR dividing by zero, and because of this we use the *value_checks()* function to avoid this error. If a zero on the diagonal is found, the user is alerted and program exited.

We would expect a small diagonally dominant matrix to converge. For example, our test matrix located in *code/sample_inputs/small_diag_dom1.in* converges as shown in the output file located in the same directory, *small_diag_dom1.out*. The convergence of iterative methods for solving $Ax = b$ is usually restricted to diagonally dominant matrices because if T is a contraction mapping equivalent to the spectral radius of the iteration matrix C being less than one, a sufficient condition for this is that for some matrix norm, the size of C is less than 1 and then Banach's Fixed Point Theorem tells us that the sequence x_k defined by $x_{k+1} := T(x_k)$ will converge to a unique limit x, the solution of $Ax = b$. And if C is strictly row diagonally dominant then the row sum norm of C is less than one and if C is strictly column diagonally dominant then the column sum norm of C is less than one. This means that in either of these case the above sufficient condition is satisfied and we get a solution x.

We would expect a small not diagonally dominant matrix, A, with no zero on the main diagonal and the absolute value of its eigenvalues less than one to converge. For example, located in the */sample_inputs/* directory are three test matrices, *not_diag_dom_evls_less_than_one(1, 2 and 3).in* which stop because of x-sequence convergence in less than 30 iterations. When the spectral radius of the iteration matrix, C, is less than one, the error in our solution vector tends towards zero as the iterations tend towards infinity (Anon, [no date]). This error is roughly reduced by a factor of the spectral radius at each iteration, so the smaller the spectral radius, the faster the algorithm converges (Anon, [no date]).

We would expect a small not diagonally dominant matrix, *A*, with no zero on the main diagonal and the absolute value of one of its eigenvalues greater than one to diverge. For example, located in the */sample_inputs/* directory are four test matrices, *not_diag_dom_evls_greater_than_one(1, 2, 3 and 4).in*, these all diverge after 2 iterations. If A is not diagonally dominant and its spectral radius of its iteration matrix, C, is greater than 1 then the error will not tend towards zero as the iterations tend towards infinity which will cause divergence.

We would expect a small not diagonally dominant matrix, A, with no zero on the main diagonal where all of CtC's eigenvalues have the absolute value of less than one to converge. For example, located in the */sample_inputs/* directory are three test matrices, *not_diag_dom_evls_less_than_one(1, 2 and 3).in*, these all have CtC's with a spectral radius between zero and one and all stop because of x-sequence convergence in less than 30 iterations.

We would expect a large sparse diagonally dominant matrix to converge or reach our *maxits* of 100. For example *sample_mtx_diag_dom2.out*, a $1{,}922 \times 1{,}922$ sparse diagonally dominant matrix converges after 24 iterations. However, *sample_mtx_diag_dom1.mtx*, a $15{,}439 \times 15{,}439$ sparse diagonally dominant matrix stops because of maximum iterations reached (even when *maxits* is set to $10{,}000$). The second, larger matrix, *sample_mtx_diag_dom1.mtx* has a relative condition of $6{,}059{,}745{,}349$ and the smaller matrix has a condition number of $260{,}000$. The very large condition number of the largest matrix plays a massive role in the huge number of iterations required for it to converge within our tolerances. A larger matrix, *sample_mtx4.mtx*, which has dimension $28{,}924 \times 28{,}924$ and a much larger matrix, *sample_mtx5.mtx*, size 44609×44609, both converge after just 22 iterations due to x-sequence convergence. This shows that the condition number of a matrix has more of an influence on the rate of convergence than the size of a matrix. An ill-conditioned matrix can slow the rate of convergence so much that it will not converge within our tolerances reaching the maximum number of iterations whereas a matrix nearly three time that size will converge within our tolerances.

The higher the tolerance used the less number of iterations needed for convergence, if convergence is possible. For example, when x-sequence tolerance is set to 1 instead of 1×10^{-10}, the $1{,}922 \times 1{,}922$ sparse diagonally dominant matrix, *sample_mtx_diag_dom1.mtx*, converges after just 5 iterations, as seen in *sample_mtx_diag_dom2_tol_equal_one.out*. This is down from 24 iterations when a smaller tolerance is used. Although the higher x-sequence tolerance stops the algorithm much sooner it also results in a loss of accuracy.

Changing the residual tolerance has a similar effect. For example, *small_diag_dom1.in* stops because of x-sequence convergence when the residual tolerance is set to 0, but a residual tolerance of 1×10^{-5} stops the algorithm after 17 iterations, a tolerance of 1×10^{-10} after 29 iterations and a residual tolerance of 1 stops the algorithm after 1 iteration. This is because this matrix is well conditioned, with a relative condition of 1.83.

An Ill-conditioned matrix would be less likely to converge quickly. For example, *code/sample_inputs/Illconditioned_diag_dom1.in* has the relative condition of $19{,}590{,}660$ and takes 21 iterations to converge as seen in *illconditioned_diag_dom1.out*. When the residual tolerance is changed to 1×10^{-1} it converges after 19 iterations and a residual tolerance of 1 causes convergence after 17 iterations. A well-conditioned matrix, for example *small_diag_dom1.in* which has a relative condition of 1.83, converges after 21 iterations with a residual tolerance of 0, but converges after just one iteration when the residual tolerance is set to 1. This happens because this matrix is well conditioned and diagonally dominant.

Results

When the spectral radius of the iteration matrix, C, was greater than 1, the magnitude of the Euclidean vector norm of the difference between solution vectors in successive iterations increased, stopping our SOR algorithm. The stopping reason given in these cases was x-sequence divergence.

When the iteration matrix, C, had a spectral radius less than 1, the magnitude of the Euclidean vector norm of the difference between solution vectors in successive iterations decreased. This either resulted in our SOR algorithm stopping because of residual, x-sequence, or maximum iterations reached. Converging ill-conditioned matrices required more iterations to converge than their well-conditioned counterparts.

When the residual tolerance was set to 0 and a well-conditioned, strictly diagonally dominant matrix was used, our algorithm stopped because of x-sequence convergence, not residual convergence.

Conclusions

The condition of a matrix played a huge role in how many iterations were required for convergence. An ill-conditioned but diagonally dominant matrix, *sample_mtx_diag_dom1.mtx* wouldn't converge within our tolerances even after $10{,}000$ iterations whereas a much larger matrix, *sample_mtx5.mtx*, which has dimension 44609×44609 converged within our tolerances after just 22 iterations.

The spectral radius of the iteration matrix, C, was the key to deciding whether a matrix A would converge or diverge, and it also impacted the condition and rate of convergence. A spectral radius larger than 1 would cause divergence and although the strict diagonal dominance of A was a sufficient condition for convergence, a non-diagonally dominant matrix would converge if its spectral radius was less than one.

A residual tolerance of 0 will result in SOR eventually finding the solution to $Ax = b$, but because we were also testing at every iteration for x-sequence convergence with a tolerance of 1×10^{-10}, the residual convergence was overshadowed by the x-sequence convergence stopping reason. As a result, x-sequence convergence was the stopping reason we constantly saw until we increased the residual tolerance from 0 to 1×10^{-10}.

As a sparse diagonally dominant matrix representing the Black Scholes PDE, the Black Scholes matrix is of the correct form to be solved by Successive Over-Relaxation. Constructing the Black Scholes matrix in Compressed Sparse Row format and solving using each time step using SOR for option values at certain stock prices is a fast and efficient method.

Detailed Design Document:

The initial planning meeting saw the project split into 3 main components:

- Input / Output;

- Successive Over-Relaxation; and
- Black Scholes.

While SOR could not be properly tested without input data, and Black Scholes could not be solved without SOR, we got around this by setting out rigid data structures to span between components. As such, SOR could be completed with test data that was of exactly the same format as would be provided by the input modules. The Black Scholes matrix and solution vector could be tested using traditional matrix multiplication methods until SOR was ready.

This agile approach allowed each member to create their sections simultaneously, and was very successful. When the first working iteration of each component was complete, they were integrated and worked with very little manipulation. A breakdown of the process is in Appendix A.

Input

The input components consisted of reading in files, checking their format, converting to Compressed Sparse Row (CSR) format, and the checking for strict row / column diagonal dominance. A number of standard errors were identified and tested for, and if they were found to occur, we could return advice to the user on how to fix them. Four *numpy* arrays were passed to the SOR section, 3 containing the input matrix in CSR form and 1 containing the solution vector b.

Input and output filenames are specified when calling the function on the command line, but the *check_CM_args()* function exists to check these files exist. If not, the function prompts the user to input another file name until the file is found. This process is robust and ensures that file name errors are caught and the user notified and given an opportunity to correct without exiting the program.

3 file formats are supported. The first, a dense matrix format as described in the assignment outline. The second, a CSR format containing 4 lines of data (the non-zero matrix values, the column indices, the row start indices, and the solution vector b) was included as it is an efficient storage method for sparse matrices. Finally, a *.mtx* format was included to make testing with ‘matrix market’ matrices easier.

A raw input check was conducted on dense and CSR files to ensure that non-digit entries were not present. This was done line by line using Regular Expressions to minimize memory usage. As *.mtx* files were external to the core purpose of the program, explicit testing of their contents was not conducted. Errors were handled using try/except statements when importing data.

Matrix Market data was read and converted to CSR format using the *scipy.io* package. Dense and CSR data was read into memory line by line to minimize the risk of running out of memory when reading in massive matrices. The format of the data was confirmed (correct number of entries, rows and columns) and converted (if necessary) to CSR. This was to minimize the size of *numpy* arrays stored on disk.

While *numpy* arrays have many advantages over traditional python lists, they are memory intensive, especially when appending values to them. As such, the vectors were compiled in a list format, and converted to a *numpy* array when complete.

The final purpose of the input components was to check for zeros on the diagonal, and for row / column diagonal dominance. This is conceptually more difficult with CSR data than with traditional dense data, but actually faster. Any errors found are returned to the user, and an output file generated if required.

The input components pass 4 correctly formatted and tested *numpy* arrays to the SOR section.

Successive Over-Relaxation

SOR was implemented in a similar fashion to that described in the lecture notes. One major design issue was the difference between Python (0-indexing) and the pseudo-code (1-indexing). This was resolved by changing the CSR arrays to 0-indexed and iterating from 0 to $n-1$, and $rowStart[i]\ to\ rowStart[i+1]$.

At the end of each iteration, the residual between the calculated vector x and solution vector b for convergence to within a specified tolerance, and the Euclidean vector norm of the difference between x vectors in successive iterations is checked for divergence.

The residual tolerance was set to 0, as tests showed that ill-conditioned matrices would show x-sequences convergence before residual convergence, and as such this was sufficient for convergence of well-conditioned matrices only.

The SOR function returns the calculated solution vector x (if found), the stopping reason, the iteration limit, the actual number of iterations performed before convergence, the x-sequence tolerance and the residual tolerance.

Output

The output file is created using the x-vector and values returned from SOR, spaced using the *ljust()* function and list comprehension.

Black Scholes

Black Scholes first asks the user for a set of inputs, with a default option, before creating the Black Scholes matrix and initial solution vector b representing the option value on the excise date. The matrix is created directed into CSR format iterating through the $3(N-2)+5$ (2 in first and last rows, 3 in all other rows) non-zero entries and appending them to the val list. The col list is created as $[0,1]$ and appending $i-1, i$ & $i+1$ for each row i from 1 to $N-1$ (except for the last row, which doesn't contain an entry in the $i+1$ column, as that column doesn't exist). Finally, the $rowStart$ list is $[0,2]$ and appending $3*i-1$ for each row i. Once completed, the lists are converted into *numpy* arrays.

The vector b is created with $N-1$ rows, each representing the value of the option at a specified stock price at expiry. The list is created using $\max\{X-nh,0\}$ where X is the strike price, and nh representing the stock price on that day. On the expiration day, option price is equal to payout, so this is simply the value of a put option on that day. The correction value for $f_1, +\frac{k}{2}(\sigma^2-r)X$ is applied before each iteration of SOR.

The maximum stock price is difficult to quantify with no knowledge of historical prices, so a value of 20, 100 or $4\times\max\{stock\ price, exercise\ price\}$ is taken to be sufficiently high, based on the value of the stock and exercise price.

The number of time steps M, price steps N and iterations per round of SOR ($maxits$) represent a trade-off between time and accuracy. After testing, $N=300$, $M=100$ and $maxits=300$ produced reliable results in a reasonable time frame.

The Black Scholes matrix and initial solution vector b are run through SOR $M-1$ times, with the calculated x vector becoming the solution vector b for the subsequent iteration (after the first value f_1, m is corrected $+\frac{k}{2}(\sigma^2-r)X$). After $M-1$ iterations, at the 0^{th} time step (present day) the option value at the specified stock price is calculated and returned to the user.

Optimisation

Throughout the programming, optimisation was at the forefront of our approach, in two ways. The first was to exploit the natural 'fastest method' of operations in Python. One example of this was the decision to build vectors by appending to lists, and then converting to *numpy* arrays. This minimises array creation operations, which are slow and memory intensive. Another was the use of the elementwise operation of *numpy* arrays, which allow fast and complete operations to be performed without iterating through lists. Finally, we use of renaming *numpy* arrays (without creating another in memory) allowed our SOR modules to minimise the number of arrays kept in storage.

The second optimisation approach was algorithm based, around what we could do to help speed up the process. A number of ideas were presented to optimise the relaxation parameter ω and initial guess of the vector . These ideas were harder to quantify, so when the main modules were completed, small test modules were created and run on random a large sparse (20%) dense format file (1000×1000) with a standard (using $seed(123)$) random solution vector .

The two options for initial x optimisation were to create a random numpy array, or create a range of random numpy arrays, check their ‘distance’ from the solution vector b (using the difference of Euclidean norms), and choose the closest. Using random numpy array took an average of 54.7 seconds to converge, whereas testing a number of random arrays for the ‘closest’ too 59.9 seconds. As such, the first method was adopted.

There were two optimisation approaches suggested for ω that were compared to the baseline approach of a guess of 1.3. The first was to run 10 iterations using values of 1.2, 1.3 and 1.4 and then choose the value which showed the greatest convergence. This took an average time of 61.2 seconds and so was not adopted. The second was to compare the convergence of the vector as a ratio of the convergence on the current iteration over the convergence on the previous iteration. If this is approximate faster than the previous iteration, ω is unchanged, otherwise ω is randomly increased or decreased within bounds of 1 and 1.5. This method took an average of 53.2 seconds, an improvement on the baseline, and so was adopted.

A final optimization suggestion was to calculate strict row / column diagonal dominance before the iterations commenced. This would save the calculations of the 2 iterations necessary to identify divergence. This idea was tested but was considerably slower and hence rejected.

Collaboration Issues

A project of this size led to a number of housekeeping issues. Individual coding styles created a challenge to present a cohesive body of code, although this was mostly resolved with careful planning. Separating code out into small modules helped with organisation, but did create issues when testing.

This was also the first time for us working on a collaborative coding project across multiple operating systems. Implementing solutions like ‘*os.path.join*’ to get around the difference between ‘/’ and ‘\’ between Unix and Windows was a key lesson learned. While we had varying individual experience and confidence with Git, the problems presented by local environment files being shared led to issues with our chosen IDE, PyCharm. Research led us to fully utilize the ‘*.gitignore*’ file to ensure that local files were not pushed to GitHub.

Implementing Test Driven Development is an ideal that supposes a certain level of development prowess that was not evident across the whole team. As such, tests were planned but not created before coding commenced. As the project progressed, all team members learned the language of testing, and saw the value in writing tests first.

As modules were separated for housekeeping, our given testing package, NoseTests, started failing to import modules and showing other errors. One unanticipated side-effect of housekeeping was tracking dependencies across all modules. These issues led us to the NoseTests documentation which in general was sparse and unhelpful. A lot of reading and trial and error eventually fixed our problems, but we definitely learned the value of proper documentation.

Testing Approach and Strategy

To ensure good code quality, extensive testing initiatives were undertaken throughout this assignment. As the group were new to software development, we researched best practice development methods and attempted to implement a number of them throughout the project. The methods we used were Test Driven Development, Pair Programming and Continuous Integration. We found that these methods helped us to consistently improve our code and ensure no regression occurred over time.

Test Driven Development
As the team are new to software development, at the beginning of this assignment we were not aware of the test driven development principle. After much research however we eventually adopted it in this project. For unit testing, our team used NoseTests, an extension to the python unit test framework to make testing easier, as our unit testing framework and built test cases for our initial functions. From this point, as we wrote functions, test cases were written in parallel and this helped ensure they were developed correctly. NoseTests also allowed the full suite of test cases to be run at any point, so as functions changed over time, we were able to capture any mistakes made or rewrite the unit tests if the functions now took new inputs or created new outputs. As our test suite grew, we ensured to run the suite before pushing to our master repository on GitHub. This ensured quality code only was sent to this main shared repository.

A detailed list of the unit test cases and results can be found in Appendix B. A small number of functions were excluded from unit testing. This was mainly due to the fact that the functions prompted a number of responses from the command line and we found this very difficult to unit test. All of these functions however were tested during functional testing to ensure their quality.

Functional Testing
In conjunction with unit testing, we also created a number of functional test scenarios to ensure the program functioned as expected for a number of given inputs and outputs. These test scenarios were run towards the end of the project and tested the program as a whole, ensuring the correct inputs were processed correctly and incorrect inputs were captured early and communicated to the user. A detailed list of the functional test cases and results can be found in Appendix C.

Pair Programming
In conjunction with our test approach, we conducted a number of pair programming sessions throughout the development of our project. Pair programming is an agile software development technique where two programmers work together at the same workstation. One programmer works as the driver, writing the code, while the second programmer observers and navigates, giving instant feedback to the driver.

Our project team implement this technique by meeting up in an on campus syndicate room with one member putting his laptop screen on a projector for the whole group to see. During these sessions, specific goals were laid out and were worked on by the group where one member wrote code and the other members gave instantaneous feedback. This technique allowed us to solve a number of problems we encountered and sped up our development process considerably.

Continuous Integration
Throughout the development of this program the project team utilised GitHub as our code repository. This allowed the group to practice continuous integration, a software engineering technique, where each team member merged their code with the master at least once a day. This reduced the effort for the team members in dealing with merge conflicts, allowed for early detection of code problems and gave an accurate picture to the team on what section of the code each member was working on.

Personal Thoughts:

Peter
I personally thought the assignment was very useful. I was quite competent with python, git and test driven development coming into the project, but teaching these concepts (especially git) to Andy and Kieron helped me solidify my understanding. Development experience in a collaborative environment is difficult to obtain, and this project allowed us to practice agile project management practices and see what worked for us.

Researching the implementation of mathematical concepts is also a new skill, as was structuring and testing such an integrated project. Also, the 4-step process of Test Driven Development, Pair Programming, writing documentation, and writing design criteria was very eye opening. The number of errors, logical fallacies or unjustified decisions caught at each stage was surprising, and the multi-layer structure of this review process worked very well.

As such, I think the most important aspects of the project were teamwork and project management, multi-level and iterative review processes, and the application of mathematical concepts through code. All of these will be critical in our careers, and being able to practice them on such an interesting assignment is invaluable.

Andy

I found this assignment very useful to both put into practice what we have learned in class as well as improve on my development and testing skills. Firstly, putting into practice the theory we learned in class, personally, is the best way to get a full understanding of the content. Learning to break down piece by piece how to create the Black Scholes matrix, and then to build the SOR algorithm to solve it made it very clear to me how and why this is done and the complexity involved. Secondly, before undertaking this assignment I had very limited version control and testing skills. I was aware of both concepts and why they were useful but had no knowledge as to how to use them myself. This assignment forced our team to develop these skills fast and I now feel a lot more comfortable with both which will be vital for my career ahead.

Kieron

I really enjoyed this assignment; I always enjoy programming assignments but this was the first time I had ever collaborated on a programming assignment and I found the experience invaluable. I had never used Git or GitHub before so looking at the assignment guidelines on day one, I thought we would have to take turns working on the master copy of the program. Peter, who had used git and GitHub before, showed Andy and I during our first team meeting how we could use Git and GitHub to work collaboratively, merging our branches to the master without breaking the overall program. The knowledge I have gained through this aspect of the project alone will have a big impact on the rest of my programming career.

I liked the detailed assignment outline which gave us a comprehensive overview of what to think about as we working on this project but also I enjoyed having the freedom to tackle the problems we faced however we saw fit. I would have liked for graphs and visualisation to have been part of the specifications because I feel I still have a lot to learn about plotting with Python.

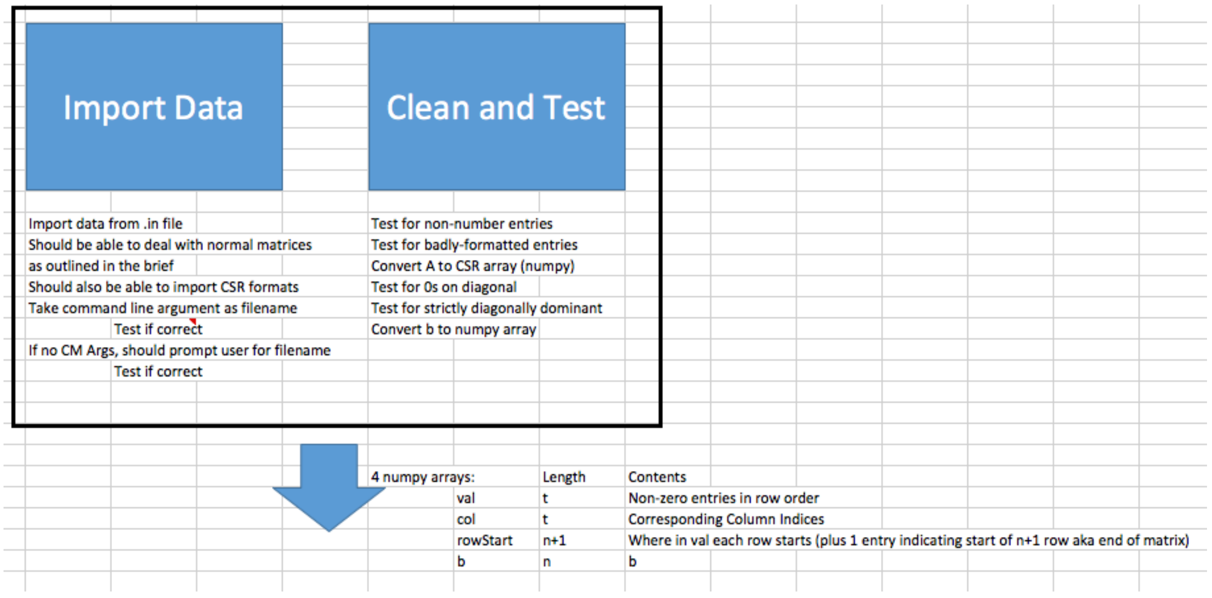
The most important aspects of this assignment for me were learning how to collaborate on a programming assignment, version control skills and learning more of the intricacies of Python.

References

Anon. [no date]. “Introduction to Numerical Methods. Iterative Methods For Matrix Equations”. University of Nottingham, School of Mathematical Sciences. [Online]. [Accessed 19/11/2016]. Available from: http://www.it.uom.gr/teaching/linearalgebra/ExamplesToIterativeMethods.pdf

Appendix A – Project Management & Module Breakdown

Successive Over-Relaxation



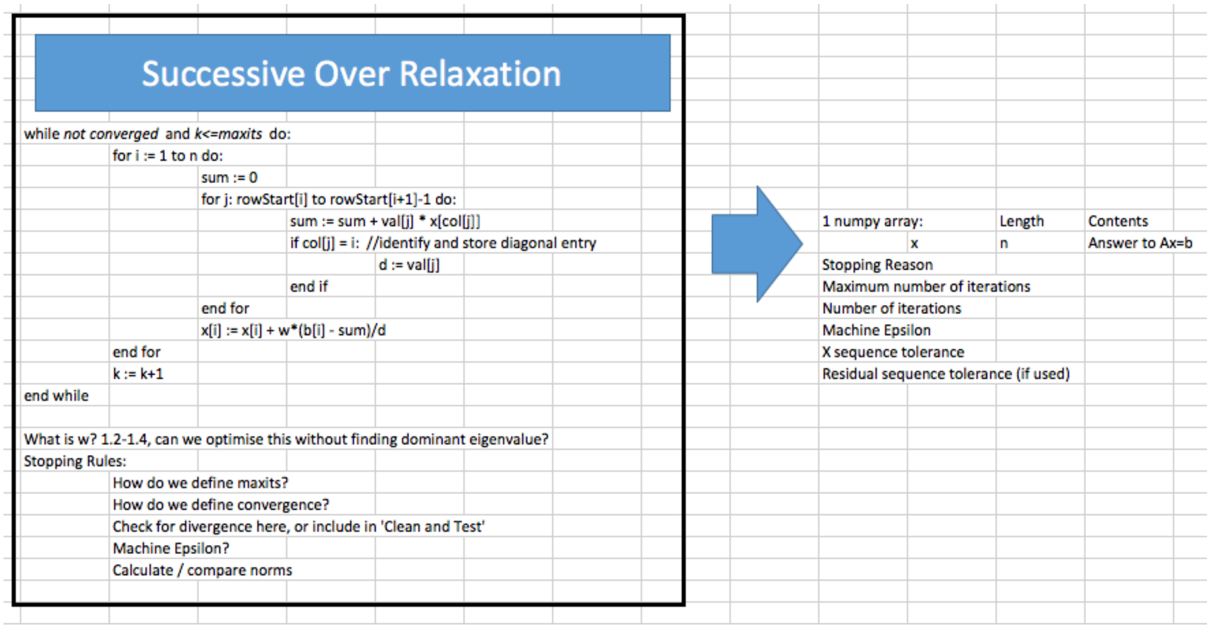
Import Data

Import data from .in file
Should be able to deal with normal matrices
as outlined in the brief
Should also be able to import CSR formats
Take command line argument as filename
Test if correct
If no CM Args, should prompt user for filename
Test if correct

Clean and Test

Test for non-number entries
Test for badly-formatted entries
Convert A to CSR array (numpy)
Test for 0s on diagonal
Test for strictly diagonally dominant
Convert b to numpy array

4 numpy arrays:	Length	Contents
val	t	Non-zero entries in row order
col	t	Corresponding Column Indices
rowStart	n+1	Where in val each row starts (plus 1 entry indicating start of n+1 row aka end of matrix)
b	n	b

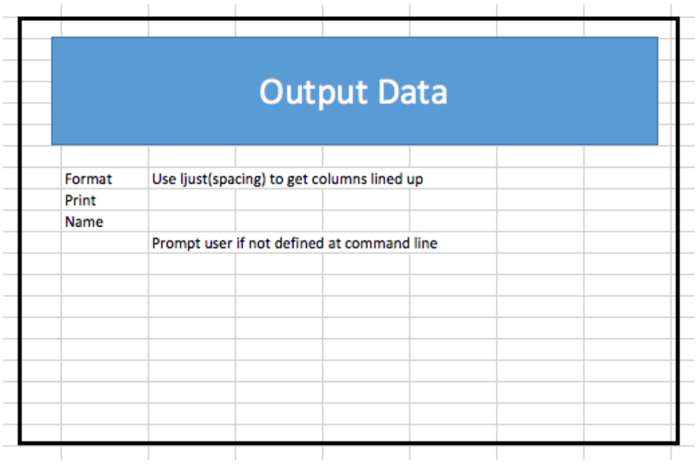


Successive Over Relaxation

```
while not converged and k<=maxits do:
    for i := 1 to n do:
        sum := 0
        for j: rowStart[i] to rowStart[i+1]-1 do:
            sum := sum + val[j] * x[col[j]]
            if col[j] = i:  //identify and store diagonal entry
                d := val[j]
            end if
        end for
        x[i] := x[i] + w*(b[i] - sum)/d
    end for
    k := k+1
end while
```

What is w? 1.2-1.4, can we optimise this without finding dominant eigenvalue?
Stopping Rules:
How do we define maxits?
How do we define convergence?
Check for divergence here, or include in 'Clean and Test'
Machine Epsilon?
Calculate / compare norms

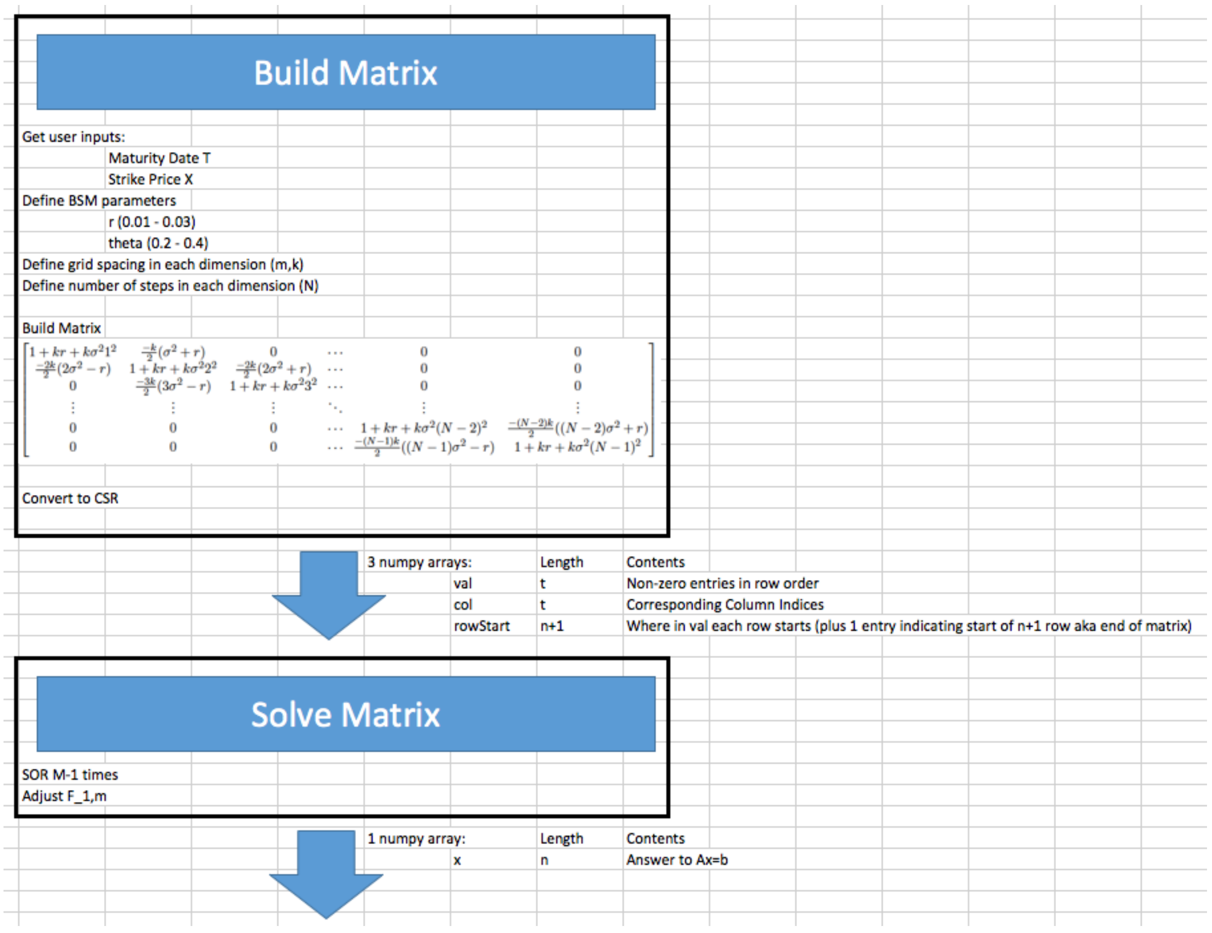
1 numpy array:	Length	Contents
x	n	Answer to Ax=b
Stopping Reason		
Maximum number of iterations		
Number of iterations		
Machine Epsilon		
X sequence tolerance		
Residual sequence tolerance (if used)		



Output Data

Format	Use ljust(spacing) to get columns lined up
Print	
Name	
	Prompt user if not defined at command line

Black Scholes



Build Matrix

Get user inputs:

- Maturity Date T
- Strike Price X

Define BSM parameters

- r (0.01 - 0.03)
- theta (0.2 - 0.4)

Define grid spacing in each dimension (m,k)

Define number of steps in each dimension (N)

Build Matrix

$$\begin{bmatrix} 1+kr+k\sigma^2 1^2 & \frac{-k}{2}(\sigma^2+r) & 0 & \cdots & 0 & 0 \\ \frac{-2k}{2}(2\sigma^2-r) & 1+kr+k\sigma^2 2^2 & \frac{-2k}{2}(2\sigma^2+r) & \cdots & 0 & 0 \\ 0 & \frac{-3k}{2}(3\sigma^2-r) & 1+kr+k\sigma^2 3^2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1+kr+k\sigma^2(N-2)^2 & \frac{-(N-2)k}{2}((N-2)\sigma^2+r) \\ 0 & 0 & 0 & \cdots & \frac{-(N-1)k}{2}((N-1)\sigma^2-r) & 1+kr+k\sigma^2(N-1)^2 \end{bmatrix}$$

Convert to CSR

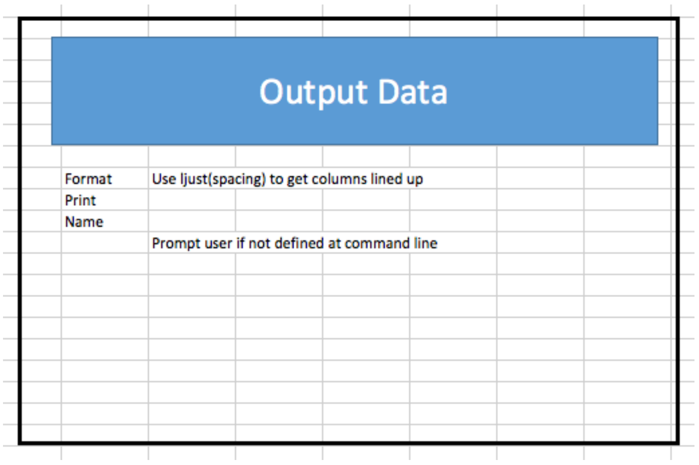
3 numpy arrays:	Length	Contents
val	t	Non-zero entries in row order
col	t	Corresponding Column Indices
rowStart	n+1	Where in val each row starts (plus 1 entry indicating start of n+1 row aka end of matrix)

Solve Matrix

SOR M-1 times

Adjust F_1,m

1 numpy array:	Length	Contents
x	n	Answer to Ax=b



Output Data

Format	Use ljust(spacing) to get columns lined up
Print	
Name	
	Prompt user if not defined at command line

Appendix B - Unit Test Cases

Calculate_residual.py

residual()

Input	val, col, rowstart, b (all originating from matrix in file *nas_Sor.in*) and randomly generated x-vector
Expected Output	$1.1532562570790883e-07$
Actual Output	$1.1532562570790883e-07$

convert_to_csr.py

con_to_csr()

Input	Four separate values each for vector, matrix size and rowStart:
	vector1 = np.array($[13., 0., 0., 4.]$)
	vector2 = np.array($[4., 11., 0., 4.]$)
	vector3 = np.array($[7., 8., 20., 4.]$)
	vector4 = np.array($[1., 0., 1., 14.]$)
	
	matrix_size = 4
	
	rowStart1 = 0
	rowStart2 = 2
	rowStart3 = 5
	rowStart4 = 9

Expected Output	Val = np.array($[13]$), np.array($[4]$), Col =$[0, 3]$, RowStart = 2
	Val = np.array($[4]$), np.array($[11]$), np.array($[4]$), Col = $[0, 1, 3]$, RowStart = 5
	Val = np.array($[7]$), np.array($[8]$), np.array($[20]$), np.array($[4]$), Col = $[0, 1, 2, 3]$, RowStart = 9
	Val = np.array($[1]$), np.array($[1]$), np.array($[14]$), Col = $[0, 2, 3]$, RowStart = 12

Actual Output	[np.array($[13]$), np.array($[4]$)], $[0, 3]$, 2)) [np.array($[4]$), np.array($[11]$), np.array($[4]$)], $[0, 1, 3]$, 5)) [np.array($[7]$), np.array($[8]$), np.array($[20]$), np.array($[4]$)], $[0, 1, 2, 3]$, 9)) [np.array($[1]$), np.array($[1]$), np.array($[14]$)], $[0, 2, 3]$, 12))

create_BS_b.py

create_BS_b()

Input	N = 20
	X = 12
	S_max = 20
	h = (S_max/N)
	k = 5
	sigma = 0.02
	r = 0.3

Expected Output	np.array($[11., 10., 9., 8., 7., 6., 5., 4., 3., 2., 0., 0., 0., 0., 0., 0., 0., 0., 0.]$)
Actual Output	np.array($[11., 10., 9., 8., 7., 6., 5., 4., 3., 2., 0., 0., 0., 0., 0., 0., 0., 0., 0.]$)

create_BS_matrix.py

create_BS_matrix()

Input	Test Case 1: M = 2, k = 5, r = 0.02, sigma = 0.3 Test Case 2: M = -2, k = 5, r = 0.02, sigma = 0.3 Test Case 3: M = 5, k = 5, r = 0.02, sigma = 0.3 Test Case 4: M = 5, k = 'string', r = 0.02, sigma = 0.3
Expected Output	Test Case 1: SystemError("There must be at least 3 intervals") Test Case 2: M = 2, k = 5, r = 0.02, sigma = 0.3 Test Case 3: Val = np.array($[1.55, -0.275, -0.8, 2.9, -1., -1.875, 5.15, -2.175, -3.4, 8.3]$ Col = np.array($[0, 1, 0, 1, 2, 1, 2, 3, 2, 3]$ rowStart = np.array($[0, 2, 5, 8, 10]$) Test Case 4: TypeError
Actual Output	Test Case 1: SystemError("There must be at least 3 intervals") Test Case 2: M = 2, k = 5, r = 0.02, sigma = 0.3 Test Case 3: Val = np.array($[1.55, -0.275, -0.8, 2.9, -1., -1.875, 5.15, -2.175, -3.4, 8.3]$ Col = np.array($[0, 1, 0, 1, 2, 1, 2, 3, 2, 3]$ rowStart = np.array($[0, 2, 5, 8, 10]$) Test Case 4: TypeError

get_bsm_inputs.py
As this function prompts to the command line it was not unit tested. It was tested during functional testing.

output_bsm.py
As this function writes the Black Scholes matrix results to an output file it was not unit tested. It was fully tested during functional testing.

get_filename.py

check_CM_args()

As this function prompts to the command line it was not unit tested. It was tested during functional testing.

check_file_exists()

Input	a = '*nas_Sor.in*'
	b = '*nas_Sor*'
	c = '*/nas_Sor10.in*'

Expected Output	True, False, False
Actual Output	True, False, False

con_filename()

Input	a = *'sample_inputs/nas_Sor2.in'*
	b = *'Input_descriptions.txt'*
	c = *'san_Ros.ni'*
	d = *'sample_inputs/nas_Sor3.in'*
	e = *'input.txt'*

Expected Output	"*sample_inputs/nas_Sor2.in*", True "*Input_descriptions.txt*", True “*san_Ros.ni*", False "*sample_inputs/nas_Sor3.in*", True "*input.txt*", False
Actual Output	"*sample_inputs/nas_Sor2.in*", True "*Input_descriptions.txt*", True “*san_Ros.ni*", False "*sample_inputs/nas_Sor3.in*", True "*input.txt*", False

import_mtx.py

import_mtx()

Input	Test Case 1: *'sample_inputs/sample_mtx.mtx*', *'nas_Sor.out*' Test Case 2: *'sample_inputs/s3dkt3m2.mtx'*, '*nas_Sor.out'*

Expected Output	Test Case 1:
	val = np.array([1,2,3,4,5,6,7,8,9])
	col = np.array([0,1,2,0,1,2,0,1,2])
	rowStart = np.array([0,3,6,9])
	Test Case 2:
	SystemExit("Unable to import the *.mtx* file. Please check it and try again")

Actual Output	Test Case 1:
	val = np.array([1,2,3,4,5,6,7,8,9])
	col = np.array([0,1,2,0,1,2,0,1,2])
	rowStart = np.array([0,3,6,9])
	Test Case 2:
	SystemExit("Unable to import the *.mtx* file. Please check it and try again")

get_mtx_b()

As this function prompts to the command line it was not unit tested. It was tested during functional testing.

input_checks.py

csr_input_checks()

Input	val1 = np.array([13.0, 4.0, 4.0, 11.0, 4.0, 7.0, 8.0, 20.0, 4.0, 1.0, 1.0, 14.0])
	val2 = np.array([13.0, 4.0, 4.0, 11.0, 4.0, 7.0, 8.0, 4.0, 1.0, 1.0, 14.0])
	val3 = np.array([3.0, 4.0, 4.0, 11.0, 4.0, 7.0, 8.0, 20.0, 4.0, 1.0, 1.0, 14.0, 15.0])
	col1 = np.array([0, 3, 0, 1, 3, 0, 1, 2, 3, 0, 2, 3])
	col2 = np.array([0, 3, 0, 1, 3, 0, 1, 3, 0, 2, 3, 4])
	col3 = np.array([0, 3, 0, 3.5, 3, 0, 1, 2, 3, 0, 2, 3])
	rowStart1 = np.array([0, 2, 5, 9, 12])
	rowStart2 = np.array([0, 2, 5, 8, 10])
	rowStart3 = np.array([1, 2, 5, 9, 12])
	rowStart4 = np.array([0, 2.5, 5, 9, 12])
	rowStart5 = np.array([0, 2, 5, 9, 13])
	b1 = np.array([2, 3, 4, 5])
	b2 = np.array([2, 3, 4, 5, 6])
	Test Case 1: val1, col1, rowStart1, b1
	Test Case 2: val1, col1, rowStart1, b2
	Test Case 3: val1, col2, rowStart1, b1

	Test Case 4: val1, col1, rowStart2, b1 Test Case 5: val1, col1, rowStart3, b1 Test Case 6: val1, col1, rowStart4, b1 Test Case 7: val3, col1, rowStart5, b1 Test Case 8: val1, col3, rowStart1, b1
Expected Output	Test Case 1: **[]** Test Case 2: Number of columns in matrix is not the same as the " "number of rows in Vector b" Test Case 3: “Uneven number of rows and columns" Test Case 4: “Last entry of RowStart vector is equivalent to the " "nth entry of val + 1" Test Case 5: “First entry of RowStart vector is not 0" Test Case 6: “RowStart vector contains non-integer entries" Test Case 7: “Value and column vectors do not have the same number " "of entries" Test Case 8: "Column vector contains non-integer entries", "Uneven number of rows and columns"
Actual Output	Test Case 1: **[]** Test Case 2: Number of columns in matrix is not the same as the " "number of rows in Vector b" Test Case 3: “Uneven number of rows and columns" Test Case 4: “Last entry of RowStart vector is equivalent to the " "nth entry of val + 1" Test Case 5: “First entry of RowStart vector is not 0" Test Case 6: “RowStart vector contains non-integer entries" Test Case 7: “Value and column vectors do not have the same number " "of entries" Test Case 8: "Column vector contains non-integer entries", "Uneven number of rows and columns"

condition.py

condition()

Input	row = np.array($[0, 0, 1, 2, 2, 2]$)
	col = np.array($[0, 2, 1, 0, 1, 2]$)
	data = np.array($[1, 2, 3, 4, 5, 6]$)
	row = row.astype(float)
	col = col.astype(float)
	data = data.astype(float)
	n = 3

Expected Output	26.4622102617
Actual Output	26.4622102617

optimise_w.py

op_w()

Input	list = $[0.6, 0.7, 0.79, 0.7, 0.8]$
	w = 1.2

Expected Output	1.2
Actual Output	1.2

raw_input_checks.py

read_raw_inputs()

Input	Test Case 1: '*sample_inputs/nas_Sor2.in*', '*nas_Sor.out*' Test Case 2: '*sample_inputs/nas_Sor3.in*', '*nas_Sor.out*'
Expected Output	Test Case 1: False Test Case 2: SystemExit("First line contains non-digit entries. Please fix and try " "again")
Actual Output	Test Case 1: False Test Case 2: SystemExit("First line contains non-digit entries. Please fix and try " "again")

read_input.py

read_inputs()

Input	'*sample_inputs/nas_Sor.in*', '*nas_Sor.out*'

Expected Output	val = np.array($[12, 1, 4, 11, 3, 7, 8, 16, 1, 3]$)
	col = np.array($[0, 3, 0, 1, 3, 0, 1, 2, 0, 3]$)
	rowStart = np.array($[0, 2, 5, 8, 10]$)
	vector_b = np.array($[1, 2, 3, 4]$)

Actual Output	val = np.array($[12, 1, 4, 11, 3, 7, 8, 16, 1, 3]$)
	col = np.array($[0, 3, 0, 1, 3, 0, 1, 2, 0, 3]$)
	rowStart = np.array($[0, 2, 5, 8, 10]$)
	vector_b = np.array($[1, 2, 3, 4]$)

solve_sor.py

sor()

Input	val = $[13.0, 4.0, 4.0, 11.0, 4.0, 7.0, 8.0, 20.0, 4.0, 1.0, 1.0, 14.0]$
	col = np.array($[0, 3, 0, 1, 3, 0, 1, 2, 3, 0, 2, 3]$)
	rowStart = np.array($[0, 2, 5, 9, 12]$)
	b = np.array($[1, 2, 3, 4]$)
	n = 4
	maxits = 100
	w = 1.2
	x = np.array($[-0.00979992, 0.08289098, 0.06390363, 0.28184973]$)
	e = $2.22044604925e-16$
	tol = $1e-10$

Expected Output	np.array($[-0.00979992,\ 0.08289098,\ 0.06390363,\ 0.28184973]$)
	np.array($[5]$)
	np.array($[1.040731881863575e-10]$)

Actual Output	np.array($[-0.00979992,\ 0.08289098,\ 0.06390363,\ 0.28184973]$)
	np.array($[5]$)
	np.array($[1.040731881863575e-10]$)

Value_checks.py

value_tests():

Input	val1 = np.array([$13.0, 4.0, 4.0, 11.0, 4.0, 7.0, 8.0, 4.0, 1.0, 1.0, 14.0$])
	col1 = np.array([$0, 3, 0, 1, 3, 0, 1, 3, 0, 2, 3$])
	rowStart1 = np.array([$0, 2, 5, 8, 11$])

Expected Output	SystemExit("There are zeros on the diagonal")
Actual Output	SystemExit("There are zeros on the diagonal")

vector_norm.py

vectornorm()

Input	Test Case 1: np.array([$1, 1, 1, 1, 1$]) Test Case 2: np.array([$-1, 3, 5, -7$])
Expected Output	Test Case 1: sqrt(5) Test Case 2: sqrt(84)
Actual Output	Test Case 1: sqrt(5) Test Case 2: sqrt(84)

write_output.py

output_text_file()
As this function writes the SOR algorithm results to an output file it was not unit tested. It was fully tested during functional testing.

Appendix C – Functional Test Cases

Case 1 - A small diagonally dominant matrix A

Input file	*small_diag_dom1.in* - contains a diagonally dominant 4×4 matrix with no 0s on the diagonal
Output file	*small_diag_dom1.out*
Expected Result	*small_diag_dom1.out* file with the following results: • Stopping reason: x-sequence convergence • Max Number of Iterations: 100 • Number of Iterations: 36 • Machine Epsilon: $2.22044604925e - 16$ • X Sequence Tolerance: $1e - 10$ • Residual Sequence Tolerance: $1.0137282758487921e - 10$ • x-vector: $-1.39963413362e - 11$ 0.5 0.500000000011 0.499999999993
Actual Output	*small_diag_dom1.out* file with details above created. Test Successful.

Case 2 - A small matrix A having a 0 on the diagonal

Input file	*nas_Sor_zero_on_diag.in* - contains a 3×3 matrix with one 0 on the diagonal
Output file	*nas_Sor_zero_on_diag.out*
Expected Result	Error printed to the terminal: There are zeros on the diagonal *nas_Sor_zero_on_diag.out* file with the following results: • Stopping reason: Zero on diagonal • Max Number of Iterations: 100 • Number of Iterations: 0 • Machine Epsilon: $2.22044604925e - 16$ • X Sequence Tolerance: $1e - 10$ • Residual Sequence Tolerance: 0
Actual Output	Error printed successfully. *nas_Sor_zero_on_diag.out* file with details above created. Test Successful.

Case 3 - A small matrix A (such that A has no 0 on the main diagonal, A is not diagonally dominant and all of the eigenvalues of C have absolute value < 1)

Input file	*not_diag_dom_evls_less_than.in* - contains a non-diagonally dominant 2×2 matrix with all eigenvalues with absolute value less than 1.
Output file	*not_diag_dom_evls_less_than.out*
Expected Result	*not_diag_dom_evls_less_than.out* file with the following results: • Stopping reason: x-Sequence Convergence • Max Number of Iterations: 100 • Number of Iterations: 23 • Machine Epsilon: $2.22044604925e - 16$ • X Sequence Tolerance: $1e - 10$ • Residual Sequence Tolerance: $4.589902317696708e - 12$

Actual Output	*not_diag_dom_evls_less_than.out* file with details above created. Test Successful.

Case 4 - A small matrix A (such that A has no $\mathbf{0}$ on the main diagonal, A is not diagonally dominant and all of eigenvalues have absolute value < 1)

Input file	*not_diag_dom_evls_greater_than.in* - contains a non-diagonally dominant 3×3 matrix with all eigenvalues with absolute value greater than 1.
Output file	*not_diag_dom_evls_greater_than4.out*
Expected Result	*not_diag_dom_evls_greater_than4.out* file with the following results: • Stopping reason: Divergence • Max Number of Iterations: 100 • Number of Iterations: 2 • Machine Epsilon: $2.22044604925e-16$ • X Sequence Tolerance: $1e-10$ Residual Sequence Tolerance: 0
Actual Output	*not_diag_dom_evls_greater_than.out* file with details above created. Test Successful.

Case 5 - A large sparse diagonally dominant matrix.

Input file	*large_matrix.in* – contains a diagonally dominant 1000×1000 matrix
Output file	*large_matrix.out*
Expected Result	*large_matrix.out* file with the following results: • Stopping reason: x-Sequence Convergence • Max Number of Iterations: 100 • Number of Iterations: 33 • Machine Epsilon: $2.22044604925e-16$ • X Sequence Tolerance: $1e-10$ Residual Sequence Tolerance: $1.7386968486732083e-08$ x-vector with 1000 values
Actual Output	*large_matrix.out* file with details above created. Test Successful.

Case 6 - A large matrix with file format *.mtx*.

Input file	Test Case 1: *sample_mtx1.mtx* – contains a large matrix in *.mtx* file format (94.8MB) Test Case 2: *sample_mtx1.mtx* – contains a large matrix in *.mtx* file format (94.8MB)
Output file	*nas_Sor_mtx1.out* *nas_Sor_mtx1_maxits200.out*
Expected Result	Test Case 1: *nas_Sor_mtx1.out* file with the following results: • Stopping reason: Max Iterations Reached • Max Number of Iterations: 100 • Number of Iterations: 100 • Machine Epsilon: $2.22044604925e-16$ • X Sequence Tolerance: $1e-10$ Residual Sequence Tolerance: 11091314.622381652

	x-vector values Test Case 2: *nas_Sor_mtx1.out* file with the following results: • Stopping reason: Max Iterations Reached • Max Number of Iterations: 200 • Number of Iterations: 200 • Machine Epsilon: $2.22044604925e-16$ • X Sequence Tolerance: $1e-10$ Residual Sequence Tolerance: 8389635.509502143 x-vector values
Actual Output	*nas_Sor_mtx1.out* file with details above created. Test Successful.

Case 7 – *bsm.py* default input tests

Input file	Python *bsm.py* prompted at the command line When prompted, default values are chosen: Stock Price Today: $\$40.00$ Strike Price: $\$42.00$ Days to Maturity: 90 Risk Free Rate: 2% Volatility: 30%
Output file	*bsm_solution.out*
Expected Result	*bsm_solution.out* file with the following results: Option Value: $\$0.51$ Stock Price Today: $\$40.00$ Strike Price: $\$42.00$ Days to Maturity: 90 Risk Free Rate: 2% Volatility: 30%
Actual Output	*bsm_solution.out* file with details above created. Test Successful.

Case 8 – *bsm.py* input tests

Input file	Python *bsm.py* prompted at the command line When prompted, default values are chosen: Stock Price Today: $\$54.00$ Strike Price: $\$60.00$ Days to Maturity: 60 Risk Free Rate: 25% Volatility: 15%
Output file	*bsm_solution.out*
Expected Result	*bsm_solution.out* file with the following results: Option Value: $\$0.24$ Stock Price Today: $\$54.00$ Strike Price: $\$60.00$ Days to Maturity: 60 Risk Free Rate: 25% Volatility: 15%
Actual Output	*bsm_solution.out* file with details above created. Test Successful.