UCD Michael Smurfit
Graduate Business School

Declaration of Authorship

I declare that all material in this assessment is my own work except where there is clear acknowledgement and appropriate reference to the work of others.

Signed:

Cathal Horan

Cathal Horan

Cián Mc Leod

Cian Mc Leod

[illegible]

Stephen O'Reilly

UCD Michael Smurfit Graduate Business School

Msc. Business Analytics (Part Time 2016-2018)

Numerical Analytics and Software Programming Assignment Two

Monday 21st November 2016

Mr. Cathal Horan	03783821	Msc. Business Analytics (Part Time 2016-2018)	cathal.horan@ucdconnect.ie
Mr. Cian Mc Leod	16200385	Msc. Business Analytics (Part Time 2016-2018)	cian.mcleod@ucdconnect.ie
Stephen O'Reilly	16201212	Msc. Business Analytics (Part Time 2016-2018)	stephen.oreilly@ucdconnect.ie

1. Introduction

Successive over relaxation (SOR) is an iterative approach to solving equations of the form Ax = B, where A is a square matrix, and x and B are column vectors, both of size n. In this paper, the implementation of successive over relaxation in a Python environment is discussed; concentrating on the data input, calculation, and stopping phase followed by unit testing and some particularly interesting test cases. The SOR method is applied in a financial context to solve the Black Scholes Merton set of partial difference equations. Finally, some closing remarks and reflections on the assignment are provided.

2. Successive over relaxation implementation

2.1. Data input and conversion

One of the first decisions to be made in implementing this method is the format of the input file. The input file consists of the matrix and the vector B which contains $n^2 + n$ floating point numbers. With a potentially very large n, it's vital to ensure that the input method is both efficient and useable from a client perspective. In an ideal scenario, the matrix would be provided in a compressed row storage format which would shrink the input file to a maximum of 3k+1+n floating point numbers, where k is the number of nonzero elements of the matrix A. Assuming that A is sparse (relatively few non-zero elements), 3k+1+n is a significant improvement on n^2.

However, for ease of the client or user of the program, the choice was made to use the suggested input format: a file of length n+2 with:

- The first row of the file containing a single number equating to n, the size of the matrix.
- n rows each of length n containing the entries of the rows of the matrix A, each entry being separated with a comma.
- The final row containing n numbers corresponding to the elements of the B column vector.

This input method was chosen as it was assumed that many enterprise programs will provide by default, or be able to provide, the matrix in the standard elementary format.

A function *row_iterator* reads over the rows of the input file one by one. The program checks the file to ensure that the matrix provided is valid: ensuring that n is strictly greater than zero and that each row from 2 to n+2 contain exactly n elements. As discussed in class, for the method to converge to an end solution, the spectral radius of the decomposed matrix C must be strictly less than one. It can be shown that a sufficient condition for this is that the matrix is strictly row diagonally dominant, the diagonal element on each row is strictly greater than the sum of the absolute values of the off-diagonal elements. This means that the diagonal element must be non-zero which is also necessary in the calculation itself due to the division by the diagonal element.

As each row is iterated over by the program, the aforementioned checks are performed. The conditions must hold for each row of the matrix which means that the program is aborted when any row fails the checks. As each row passes the check, it is converted to compressed sparse row (CSR) format (a dictionary consisting of the following lists: val, col and row_start). As both the checks and conversion to CSR format can be completed independently on each row, it is not necessary to store the rows of the input file once converted. This minimises any potential loss of efficiency by designing the input file to store the matrix in natural format.

The current program has a number of limitations. The matrix is only tested for strict row diagonal dominance meaning that any matrix A which is strictly column diagonally dominant would fail the row checks. In order to test for column diagonal dominance, each row of the file could not be checked in isolation; either the matrix must be stored in a way that allows a column check or the matrix in the input file must be first transposed and then checked for row dominance. This eliminates the efficiency savings gained by examining the input file on a row by row basis.

The format of the input file could be adjusted in several ways as long as the user of the program can provide an input to meet this condition. Strictly speaking, requiring the size of the matrix n on the first row is not necessary. It could be assumed that the matrix is of size f-1 where f is the number of rows in the file of size f by f-1.

Row diagonal dominance is a sufficient but not necessary condition for this method to converge. As seen in the test cases in section 3.2, matrices do exist with a spectral radius less than one which are not diagonally dominant. In these cases, the program will abort prematurely without solving the system. To resolve this limitation, it would be necessary to find the eigenvalues of the matrix A: a less than trivial problem. To allow for testing and calculation a command line argument (--check_diag) can be called to skip the row diagonal check.

2.2. Successive over relaxation

Assuming the input file provided passes the checks outlined in section 2.1 and is converted into CSR format, the csr dictionary is passed into the *sor_calc* function which carries out the SOR method using the corrective form:

$$X_i^{k+1} = X_i^k + \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}X_j^{k+1} - \sum_{j=i}^{n} a_{ij}X_j^k)$$

A static value for omega, ω, is chosen. As discussed, the optimal omega is often found to be between 1.2 and 1.4 so a value of 1.2 was picked. The optimal value of omega is found to depend on the spectral radius and as mentioned, this can often be more difficult than solving the system so a static value is chosen. If in future iterations, this program was to find the eigenvalues of the matrix A to determine whether the system would converge, it would be possible to analytically calculate the optimal parameter value.

As is usual with iterative methods, it is necessary to provide a starting vector X_o for the method to initialise from. Here, the zero vector of length n is given as the starting vector for simplicity.

2.3. Stopping rules

In all iterative methods, a value of maximum iterations, maxits, is provided to ensure that the method aborts even if it does not converge. This is helpful in the case of cycling/divergence of solutions or even in cases of programmer human error. In the program, the default value of maxits is 100 iterations, although this can be modified when running the program at the command line.

A test for X vector convergence and divergence is carried out at each iteration, K. The sequence was declared to converge if:

$$\left| \|X^{K+1}\| - \|X^K\| \right| < \varepsilon$$

Where || || is the traditional Euclidean norm of a vector. The difference between the two vector norms is stored in a variable *current_difference* for the purpose of divergence testing. The value of ε is calculated as:

$$\varepsilon = tolerance + (4 * \varepsilon_m * \| X^{K+1} \|)$$

This adjustment to ε ensures that the overall factor used takes account for the floating point number spacing around the norm of the vector. The default value of *tolerance* used is 10^{-3}, however this can be adjusted at the command line when running the program. When the difference between the two vector norms is sufficiently close, the program aborts stating that the X sequence has converged.

The test for divergence checks whether the *current_difference* is greater than the *previous_difference* in absolute value, the program aborts with the reason X sequence divergence. In order to assure that the program does not abort on the first iteration due to divergence, the *previous_difference* variable need to be initialised to a value equal to the X^1 norm, the norm of X^0 is 0 so the difference is just the X^1 norm.

The convergence and divergence tests of the X vectors posed the most difficulties in the design phase. Any one of a number of norms could be chosen, however as specified the Euclidean norm was chosen. The X sequence convergence is not affected by the choice of norm as the norm value is not used in the SOR calculation. However, this does influence the stopping rule and hence the number of iterations the program will run for, resulting in slightly different end X vectors.

It was decided to test for residual convergence as well as X convergence for completion of the program. The overall aim of SOR is to solve the system of linear equations AX = B. At each iteration, we have a candidate solution $\widehat{X}$. If $\widehat{X}$ is a close enough to fit to solve A$\widehat{X}$= B then it is an acceptable solution. At each iteration, the residual *r* is calculated according to the current iterations $\widehat{X}$ as

$$r = B - A\widehat{X}$$

As the matrix A is stored in CSR format, with matrix multiplication of a square matrix and column vector of same length defined in row format, it is computationally simple to calculate the product of these two. The norm of the residual is tested for closeness to zero. A tolerance of 10^{-3} is taken which is adjusted for machine epsilon to account for the number spacing around 0. Overall this tolerance is lower than the tolerance used to test the convergence of the X vector as the representable numbers in the floating point system are much denser in the interval [0,1]. If the norm of the residual is less than the prescribed tolerance, the program aborts taken the solution X* as $\widehat{X}$.

2.4. Output file

As per the requirements given, once the results from the SOR (both directly from the SOR function and as a part of the Black-Scholes Method) are calculated, the findings are output into a default file entitled 'nas_Sor.out', unless an alternate filename is specified. Contained in this are the outputted headings and the outcome of the attempt to solve the matrix equation as follows:

- Stopping Reason
- The value of maximum Iterations
- Number of Iterations
- Machine Epsilon
- X Sequence Tolerance
- Res. Sequence Tolerance

These will always be outputted regardless of the outcome of the attempt to solve the matrix. It was also decided that for both testing purposes and for ease of use, that the output headings and the outcome of the attempt to the console would be written to console. This gives the user instant feedback, rather than having to locate and open the results file. We found while using the programs that this was a much more user-friendly way and it also sped up our testing and debugging time also.

When we get a successful outcome to our SOR attempt i.e. when we either hit our residual convergence tolerance, our *X* sequence tolerance or our max number of iterations, then we also return the *x* vector in the results file. Unlike the previous two sections of the results file, we found that the outputting of the *x* vector to the console was not very user-friendly and was not beneficial from both a testing/debugging point of view. Furthermore, when the *x* vector is sufficiently large, then it led of the filling up of the entire console buffer and a such the user was unable to view the output headings and outcome of the attempted SOR. We therefore do not output the *x* vector to the console.

3. Testing and experimentation

3.1. Testing Approach

One of the main considerations in the overall program design was to facilitate unit and system testing. During both the input and calculations phases there are multiple complex operations which need to be individually tested. Breaking these sections up into classes and functions allows for easy unit testing of individual operations. For example, row dominance is verified during the input phase. This is a separate function and can be called directly for unit testing. Testing these individual elements via unit testing ensures that any system tests are not verifying multiple separate aspects at one time. The system tests thus focus on the overall interaction of these separate parts and to verify expected results with regard to the test matrices.

The requirement for the results to be output to a file at an exit point which could occur at several different points in the program was also a factor that influenced the test and experimentation phase. The initial approach was to call a function which both wrote to an output file and exited the program after writing. The issue with this approach is that it did not return any state back to the program and testing could only occur via reading of the output file. This introduced further states where an error could occur, e.g. writing or reading the file, instead of checking the returned state from a function called from a main script.

Also, by separating each phase into separate classes which returned some state it made it easier to incorporate the SOR calculation phase in conjunction with the BSM matrix. A separate team member could thus work on the BSM module and know the input and output requirements for that phase. Upon completion it meant the BSM modules could simply be 'plugged' into the overall program and replace the input phase of the original SOR calculation.

To facilitate this approach, we created a global class which contained the output reasons and values such as machine epsilon and the tolerance calculation which would be required at several points in the program. This reduced the requirement to pass multiple variables from function to function when needed.

3.2. Testing

Testing was broken into two separate phases, unit testing and system testing. The unit testing was focused on the individual elements of the program to verify they worked as expected in the context of an isolated single operations. System testing then involved testing the overall program in terms of a number of test matrices whose specifications were defined in the programming assignment document. Both the unit tests and system tests are contained in the /code/tests directory. The test inputs used for testing are contained in the /files directory. The input files used during testing can be found in the appendix.

Test Data	Mocked vector data and input CSR which should result in convergence.
Expected Result	By mocking the input for the convergence check method the expected result from the function should be true indicating convergence occurred.
Actual Result	test_convergence (tests.test_input.TestInput) ... ok

Test Data	Mock row, row number and matrix size data where the row length does not match the matrix size N.
Expected Result	Since the matrix size does not match the row length the row_check method of the input class should return "Invalid Matrix Input" as a stop reason.
Actual Result	test_invalid_row_check (tests.test_input.TestInput) ... ok

Test Data	Mock input file with matrix size N of zero.
Expected Result	This test verifies that a correct or calculable matrix size has been passed in the input file. Since the N size of the matrix is zero the row iterator method of the input class should return "Zero Matrix Size" as a stop reason.
Actual Result	test_zero_matrix (tests.test_input.TestInput) ... ok

Test Data	Mock input file with B vector whose element number is smaller than that matrix size.
Expected Result	Test to verify the B vector has same number of values as the matrix size. This is to ensure it is possible to perform a SOR operation since each row has a corresponding B vector element. Since the B vector number does not match the number of rows in the matrix the row iterator method of the input class should return “Invalid Matrix Input” as a stop reason.
Actual Result	test_invalid_vector (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
Invalid Matrix Input	100	None	None	0	0

Test Data	Test 5x5 matrix with zero on (5,5) position.
Expected Result	Since there is a zero on the diagonal the row_check method of the input class will identify this exit condition and return with a stop reason set to “Zero On Diagonal”. A zero diagonal element means that the row cannot be strictly row diagonally dominant which is our sufficient condition for convergence. Furthermore, as detailed in section 2.2, the method depends on the division of omega by the diagonal element a_{ii} which for this row results in a division by zero: an unrecoverable operation.
Actual Result	test_zero_diag_matrix (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
Zero On Diagonal	100	None	None	0	0

Test Data	Test 5x5 row dominant matrix.
Expected Result	The sufficient condition for the method to converge is that the matrix A is strictly diagonally dominant. The strictly row diagonally dominant also excludes the possibility of having a zero as the diagonal element. The method will converge. However, as discussed in section 2.1, the program will reject a matrix which is column instead of row diagonally dominant and abort. If it is row diagonally dominant, it will accept, run the SOR calculation and converge.
Actual Result	test_small_diag_dom (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
x Sequence Convergence	100	4	2.22044604925 e-16	0.001	0.001

Test Data	A small matrix A such that A has no 0 on the main diagonal, is not diagonally dominant and all of the eigenvalues of C have absolute value < 1.
Expected Result	As the matrix has no zero elements on the main diagonal and the eigenvalues are all strictly less than one, the SOR sum will converge as the necessary conditions have been met. A diagonally dominant matrix is the sufficient conditions for the aforementioned properties but is not necessary which means this is not a problem. However, as finding the eigenvalues of the matrix C is computationally difficult, it is not feasible to find these. The program uses the sufficient condition of row diagonally dominance to judge convergence so hence will reject a matrix with zero on the diagonal and abort even though if the SOR sum was to run it would converge.
Actual Result	In this test case a flag was added to allow the row diagonal dominance check to be skipped (Globals.CHECK_DIAG = False). This was added to allow the program to perform a SOR calculation on matrices which do not meet the strict diagonal dominance condition. The test verified that the stop reason was due to “Residual Convergence”: test_sor_eigen_lt_one (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
Residual Convergence	100	2	2.22044604925 e-16	0.001	0.001

Test Data	A small matrix A such that A has no 0 on the main diagonal, is not diagonally dominant and one or more of the eigenvalues of C have absolute value > 1
Expected Result	With at least one eigenvalue of C having absolute value greater than 1, the SOR sum will diverge and no solution for the X vector will be found. Similar to other tests, where the matrix is not row diagonally dominant, we need to override the diagonal check in the input class to allow the SOR calculation to take place.
Actual Result	The test returned the stop reason “x Sequence Divergence”: test_sor_eigen_gt_one (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
x Sequence Divergence	100	2	2.22044604925 e-16	0.001	0.001

Test Data	A small matrix A such that A has no 0 on the main diagonal, is not diagonally dominant and all of C^tC's eigenvalues have absolute value < 1
Expected Result	As the matrix has spectral radius of 1, this acts as an upper bound on the matrix norm. As detailed, $\| \|_2$ norm of C is strictly less than 1 meaning that the SOR sum will converge. However, similar to previous tests which are not row diagonally dominant, the program will abort before calculating the SOR sum as the sufficient condition of row diagonal dominance is not present. As a result, we need to override the diagonal check in the input class to allow the SOR calculation to take place.
Actual Result	The test verified that the stop reason was due to "Residual Convergence": test_sor_transpose_eigen_lt_one (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
Residual Convergence	100	11	2.22044604925 e-16	0.001	0.001

Test Data	A large (1000 < n < 10,000) sparse diagonally dominant matrix
Expected Result	As the matrix is diagonally dominant, it meets the sufficient condition for convergence and excludes the possibility of having a zero on the main diagonal. With a sparse matrix, assuming the input file is valid according to the conditions described in section 1.1, the matrix will be converted to CSR format and the SOR sum will converge.
Actual Result	The test verified that the stop reason was due to "Residual Convergence": test_large_diag_dom (tests.test_input.TestInput) ... ok

File output:

Stopping Reason	Max. Iterations	No. Iterations	Epsilon	X Seq. Tolerance	Res. Seq. Tolerance
Residual Convergence	100	7	2.22044604925 e-16	0.001	0.001

3.3. Discussion

3.3.1. Sample Test Data

One issue that proved difficult for testing was finding same test matrices for which known SOR results were available. The matrices required significant "tweaking" at times to obtain a matrix which met the required criteria and seemed to match the SOR results expected. However, this leads to a situation where you are changing both the test data and the system under test in parallel. This can result in situation where you are actually changing the test data to engender a result which you expect as opposed to changing an aspect of the system under test to return a different result. It was felt that a standard set of test data could have been provided with expected results. It would be clear then when the program was not performing as expected and when the system needed to be changed to match expected results. Instead, without a verified dataset it was difficult to know if the results were accurate in some cases.

3.3.2. Unit Testing v System Testing

Another aspect of testing that arose during the project was the dynamic between unit and system testing. When creating the initial unit tests it appeared straightforward when a unit test was successful or not. For example, the convergence test was verified in a unit test and appeared to function as expected. However, during system testing the results indicated an issue with the calculation of the convergence. The convergence test was not using the absolute value of the X difference calculation. In the unit tests the values used had not checked for this occurrence. It was only during the end to end system test when results appeared incorrect that this issue was detected. This shows the importance of performing both unit and system tests to ensure all aspects of the program are tested.

3.3.3. Precision

It noticed during testing that when low values were used in the matrix elements it resulted in expected and actual results not matching. For example, the small diagonally dominant matrix initially returned a divergent result. Changes were made to the program and how this was measured to see if this resulted in a convergent output. However, this ended up causing other test cases to fail. Different values were plugged in for Omega and the tolerance factor but this did not change the actual result. The result only changed when the diagonal value were increased to a point where the result converged.

This is not ideal practice, i.e. changing test values to match expected results but it seems that the precision of the results was impacted by the low values during calculations resulting wildly ranging values which triggered the divergent checks. Increasing the diagonal elements resulted in a smoothing of these values and a convergent result. As noted, a standard matrix test example here would have ensured it was clear whether the program was performing as expected in that case. It was unclear whether we needed to investigate altering of the precision to increase the range of matrices that the program could handle. Instead it was decided that this should be noted as part of the testing section to indicate exactly where we encountered issues with testing and the steps taken to try and reach a result which matched the expected values.

4. A financial context

4.1. Implementing the Black-Scholes PDE

As defined in the assignment handout, what we are attempting to do is to implement and solve the Black-Scholes PDE, that is, determining what the fair price is for a European put option given the following parameters: the maturity date of the option, the strike price offered for the stock option until the maturity date, the volatility of the stock and the risk--free interest rate. We also need to define our own segmentation variable i.e. how many time-steps we divide the time between the date of issue of the option and the maturity date of that option. The larger the number of time-steps that we divide the time interval into, the more accurate our solution. The trade-off is that the complexity increases the more time-steps that are calculate.

As the solution of the Black-Scholes PDE is solved by using the SOR method, we also need to provide the user with the ability to alter the parameters of the SOR method. To this end, we also allow the user to amend the maximum number of iterations, the ω value and the tolerance value necessary for the SOR method. The full list of input parameters and their defaults for our experiment can be found in Appendix 6.1.

The program itself takes in these parameters directly from the command line. We have given each of the parameters a unique shortcut also for ease of use.

Our method solves the Black-Scholes PDE using an implicit finite difference method. We attempt to solve this at each of the timestamps of our time interval and use these to give us an approximation of the option price. This involves solving the set of equations:

(a) $\frac{-nk}{2}(n\,\sigma^2 - r)$ for the point $f_{n-11,m}$

(b) $(1 + kr + k\sigma^2 n^2)$ for the point $f_{n,m}$

(c) $\frac{-nk}{2}(n\,\sigma^2 + r)$ for the point $f_{n+1,m}$

We start by solving the equation at time maturity date, as we know the price of the option at this time i.e. the strike price *X*, provided in the function parameters. Thus, the strike price at the maturity date becomes the value $b_{1,m}$ value in the B vector. Using this, we are able to calculate the what the equivalent strike price would have been at the previous time interval i.e. the *k* value, calculated by dividing the time *T* by the size of the matrix generated *M*. We populate each of the above functions using the program parameters and then simply solve these three. Once this value is known, we then divide the strike price at *n,m* by this. This becomes the value of $b_{1,m-1}$ in the *B* vector.

The individual results of the three equations become the row values $a_{n-1,m}$, $a_{n,m}$ and $a_{n+1,m}$ in the A matrix. There are two cases where this does not hold true, however. We can't populate equation (c) in the initial iteration of the matrix population, due to the fact the that value refers to a point in time beyond the maturity date i.e. the next time interval after maturity date and, as such, does not fall into the bounds of the calculated matrix. We do still need to calculate this value, however, as it is necessary for the calculation of the $b_{1,m-1}$ value in the B matrix.

The second case is in the final iteration, whereby we can't populate equation (a) in the matrix, due to the fact the that value refers to a point in time before the option date arose and, as again, does not fall into the bounds of the calculated matrix. This time, we do not need to calculate this value, as in this last iteration, there is no call for the calculation of the $b_{1,m-1}$ value in the B matrix, as we are already have the $b_{1,1}$ value and thus a fully populated B matrix to pass into our SOR function for solving.

One of the difficult parts with having to start the equation calculations at the maturity date, is that we have to inversely populate our CSR matrix for the Black-Scholes calculation. This meant that for the *rowstart*, *B*, *val* and *col* variables, having to calculate the last values in each of the arrays, populate these values into the CSR object and then iteratively fill the arrays until we reach the final iteration of the population loop i.e. the first row of the matrix. The inverse nature of this calculation did take some extra thought, but once we factored in the two extraordinary boundary cases, mentioned in the previous paragraph, our CSR object was populated accurately.

Once solved, we write our results out to the command line as per the SOR function. The only difference is that we write them out to the defaulted 'nas_BSM.out' file, instead of the nas_SOR.out file, such that they don't overwrite each other when called consecutively.

4.2. Interpreting the results

Once we have generated the output matrix from the SOR calculations, we still need to interpret the results coming from it i.e. we still need to know from this matrix whether an option is worth considering, given the maturity date, strike price etc. The matrix that we initially generated gave us the linear equations for timesteps from maturity date to time equal to zero i.e. the present day. Therefore, if we need to know if an option offered to us is a valuable asset or not, then we concentrate on the initial value of the *x* vector.

Take as an example, the default values for the Black-Scholes PDE listed in the appendix i.e. a strike price of $10 with a maturity date of 5 months. When calculated, the x_1 value in the matrix gives us an approximate value $9.04, a difference from the strike price of $0.96. With this knowledge, we can say that if the option is offered to us at a price of $0.96 or less, then it is worth taking up this option on the stock. However, if the option price is higher that that value, then we deem the option to be not worthy of taking.

5. Closing Remarks

By separating into three main modules (input, SOR & output), the program's structure allowed comprehensive unit testing to be performed, as well as the use of the code by the Black Scholes program. The lack of restrictions on the format of the program allowed a lot of designer discretion into how it should be structured and a lot of choices had to be made on key elements of the program. The choices made have been justified throughout the design document and any potential impact, whether negative or positive, highlighted. Different design choices may impact the results received due to the number of iterations the program carries out but overall, systems will still converge or diverge regardless.

The program was robustly tested as detailed in section three. This involved identifying key unit tests to verify functional testing and choosing test matrices to verify end to end system testing. As noted in the testing section, problems arose when expected results were not initially observed. The problem related to the fact that we could not be sure the issue was with the matrix chosen as the test input or the program itself. What followed was a series of changes to both the test input and the program to achieve the expected result. It also resulted in the reduction of size of some of the test matrices to 2x2 or 3x3 to make it easier to ensure they met the eigenvalue requirements noted in the specification. While not ideal this did result in significant discussion among the group as to why this may be occurring. This was helpful in better understanding the theory behind the SOR iterative method.

Without the use of third party test data, it became difficult to robustly test the program within the limited time frame. Depending on the test data used, different results may be obtained. For example, while testing a small diagonally dominant matrix, it would be possible to test using a matrix which is either strongly diagonally dominant such that the absolute value of the diagonal element is greater than the sum of the absolute values of the off diagonals by a large value λ.

$$|a_{ii}| > \lambda \left(\sum_{j,\, i \neq j}^{n} |a_{ij}| \right)$$

For a large value λ, the system will converge much quicker as little weight is placed on the corrective sum. For a matrix with a small λ, the corrective element of the sum carries a lot of weight such as the system will not converge as quickly. Given this choice in test data could have such a drastic impact on the testing results, a standard set of test matrices would have been beneficial to objectively test the program.

One aspect that we did not fully test was changes to the precision used in the program and the impact this may have on certain calculation. We could have used higher levels of precisions (i.e. using software packages such as decimal) and investigated the trade-off in terms of improvement in accuracy versus the increased time incurred when implementing extended precision calculations.

The interpretation of the results for the Black Scholes PDE proved difficult for a time during this assignment. Once we had generated the results vector, we were initially unsure as to how to interpret this. While understanding that the vector was a regression (in a literal sense of the word) of the risk-free value of the stock from the maturity date back in time towards the present value, our understanding of what this represented with regards the option price varied over time. While there are existing formulae from the financial realm for solving Black Scholes, these factor in the present value of the stock price, which is not available to us.

6. Appendices

6.1. Default Parameters for Successive Over Relaxation

Option	Parameter	Shortcut	Default
File to Read	--infile	-i	nas_SOR.in
File to Write	--outfile	-ou	nas_SOR.out
Max. No.of Iterations	--maxits	-m	100
Omega Value	--omega	-o	1.2
Tolerance	--tol	-t	0.001

6.2. Default Parameters for Black Scholes PDE

Option	Parameter	Shortcut	Default
`Strike Price	--strikeprice	-s	10
Time	--time	-ti	5
Volatility	--volatility	-v	0.2
Risk-Free Rate	--rate	-r	0.02
No. of Intervals	--intervals	-i	10
Max. No.of Iterations	--maxits	-m	100
Omega Value	--omega	-o	1.2
Tolerance	--tol	-t	0.001

6.3. Input files

6.3.1. nas_SOR_invalid_n.in (test_zero_matrix)

```
0
12, 0, 3, 4
0, 10, 8, 1
1, 0, 7, 0
0, 0, 1, 4
```

6.3.2. nas_SOR_invalid_b.in (test_invalid_vector)

```
5
12, 0, 3, 4, 0
0, 10, 8, 1, 0
1, 0, 7, 0, 3
0, 0, 1, 5, 0
1, 4, 5, 0, 11
1, 2, 3
```

6.3.3. nas_SOR.in (test_invalid_row_check)

```
3
-2, 1, 0
1, -2, 1
0, 1, -2
```

6.3.4. nas_SOR_small_zero_diag.in (test_zero_diag_matrix)

```
5
12, 0, 3, 4, 0
0, 10, 8, 1, 0
1, 0, 7, 0, 0
0, 0, 1, 4, 0
1, 2, 3, 4, 0
1, 4, 5, 9, 3
```

6.3.5. nas_SOR_small_diag_dom.in (test_small_diag_dom)

```
5
20, 0, 3, 4, 0
0, 40, 8, 1, 0
1, 0, 50, 0, 3
0, 0, 1, 80, 0
1, 4, 1, 0, 91
1, 2, 3, 4, 5
```

6.3.6. nas_SOR_Eigen_gt_1.in (test_sor_eigen_gt_one)

```
3
6, 1, 1
3, 5, 4
2, 5, 6
1, 2, 3
```

6.3.7. nas_SOR_Eigen_lt_1.in (test_sor_eigan_lt_one)

```
3
0.3, 0.4, 0
0.7, 0.5, 0
0, 0.8, 0.5
0, 0, 0.1
```

6.3.8. nas_SOR_transpose.in (test_sor_transpose_eigen_lt_one)

```
2
0.5, -0.3
-0.3, 0.25
0.1, 0.2
```