UCD Smurfit Graduate School of Business

Numerical Analytics and Software

Programming Assignment 2

Authors:
Eoghan Aherne (??? - MSc B.A Part-time)
Louis Carnec (??? - MSc B.A Part-time)
Conor Reid (16202630 - MSc B.A Part-time)

Lecturer:
Dr. Sean McGarraghy

November 20, 2016

Contents

1 Introduction

2 Theory

2.1 Overview

2.2 Questions

useful to check for convergence of the residual to zero?
what matrix norms might be useful and why?
scaling be helpful?
which matrices do you get expect convergence and which do not? why?
effect of different values of tolerance?
effect of an ill-conditioned matrix?

2.3 Business Applications

Successive over-relaxation methods were preceded by the Gauss-Seidel algorithm developed in the 19^{th} century which enabled him to solve singular linear systems with equal numbers of unknowns and equations. Over-relaxation techniques later evolved with the work of David Young, these would enable systems of over 20,000 unknowns to be solved by 1965 [1]. Such methods were initially used for systems with elliptic partial differential equations discretised with finite difference techniques (or multi-grid methods).
A range of applications could benefit from the lower computational cost and memory requirements of these iterative methods. Specifically the SOR iterative method is beneficial in circuit analysis, quantitative finance and weather forecasting.

2.4 SOR

The first part of our solution deals with simply applying the successive over-relaxation (SOR) algorithm to a system of linear equations in the form $\mathbf{Ax} = \mathbf{b}$. This is in an effort to solve for $x_{1\times n}$ given $A_{n\times n}$ and $b_{1\times n}$. One approach to solving this type of problem is to decompose A into its diagonal component (D), and upper and lower triangular components (L and U). This yields an expression on which the SOR, an iterative technique, can work and eventually solve. It should be noted that SOR is a modification of the Gauss-Seidel method, increasing the reaction factor from $\omega = 1$ to $\omega > 1$. The purpose of the relaxation factor is to modify the spectral radius (the largest eiganvalue) of the resulting matrix and thus cause convergence more quickly. In theory by increasing this factor, convergence should be positively affected.

It has been shown that the optimal value of ω can be expressed as

$$\omega^* = \frac{2}{1 + \sqrt{1 - r_\theta}(C)} \tag{1}$$

where $C = -(D + L)^{-1}U$ (D, L and U as above). However, in general the finding of this optimal relaxation factor is more challenging and computational expensive than the system of linear equations that are to be solved. For this reason, ω in the range $1 \leq \omega^* \leq 2$ is chosen.

Taking into account the triangular nature of D, L and U x^{k+1} (that is, x at the $k + 1$ iteration of x) can be expressed as

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij}x_j^{k+1} - \sum_{j>i} a_{ij}x_j^k\right), i = 1, 2, ...n \tag{2}$$

Using the equation above it is possible through iteration to approximate the solution of $\mathbf{Ax} = \mathbf{b}$.

2.5 Black-Scholes(-Merton)

The Black-Scholes model provides a way to calculate option prices from stock prices. The Black-Scholes(-Merton) equation, based on the Itô process ($dS = \mu Sdt + S$)
The implementation of the Black-Scholes-Merton pricing problem found in the module `tridiagM.py` takes advantage of the fact that the Black-Scholes differential equation as a system of linear equations ($\mathbf{Ax} = \mathbf{b}$) can be written as a tridiagonal $(N-1)\times(N-1)$ matrix. This tridiagonal matrix, $\mathbf{A}$, which by definition is a sparse matrix, can be solved using the Sparse SOR algorithm which solves for vector $\mathbf{x}$ the system of linear equations $\mathbf{Ax} = \mathbf{b}$ by successive iteration.

We are able to approximate all option prices for a given stock price for all $m \in M$ by approximating the previous time step using a system of linear equations. There are two approximation in finding option prices for all time steps m.
The first of these entails approximating put option prices at time T given Stock prices and a known put option strike price, X. Using this approximation for $f_{n,M}$, we can solve the system of linear equation for option prices at previous time periods $f_{n,m}$.
Given that the matrix $\mathbf{A}$ represents the finite difference constants relating $f_{n,m-1}$ to $f_{n,m-1}$ and that we can approximate the N European put option prices at $f_{n,M}$, where $M = T$ and time at which the option is at maturity, given the fact that the portfolio must
$P = -f + \frac{\partial f}{\partial S} S \left(-\sigma S \frac{\partial f}{\partial S} + \sigma S \frac{\partial f}{\partial S} \right) dz = 0$

2.5.1 Creating the Grid

To solve for option prices at all time steps we need create a grid of stock prices (S, the stock price vector) and of M time steps until maturity (T). Options have expiration dates or maturity. For example, an option expires in 12 days (T=12) and since our time steps is days ($k = T/M = 1$), there are $M = 12$ grid steps. If our time step was half a day (k=0.5) for the same expiry date (12 days), the grid would have 24 time steps ($M = 12 \times 0.5$). Stock prices can be any non-negative real number. However the grid must start at 0 since the put option price is the strike price when the stock price is zero ($f_{0,m} = X$). To use the Sparse SOR algorithm we must have a square matrix A. Therefore there must be the same number of grid steps for the time (M) and stock price (N) vectors (N=M).
A function (`createSandT`) was created to return two vectors for stock prices and times. The grid step lengths, k (step length between time steps) and h (step length between stock prices) are calculated from input parameters. h is the length of the stock price divided by the number of time steps (M), $\frac{S_{max} - S_0}{M}$.

2.5.2 Components of the system

The triadiagonal matrix $\mathbf{A}$ requires the length of S and t to initiate the length of the three diagonals which make up the matrix. The function `abcArrays` creates three arrays for the diagonals using a `for` loop. The arrays are referred to as; b (the main diagonal), a (the first diagonal below b), and c (the first diagonal above b). They are then passed to the `tridiag` function which returns the tridiagonal matrix $\mathbf{A}$. We are able to build this matrix solely based on vectors S and t which gives us the number of steps in each direction since the components of the diagonals increase linearly with each time step.
The first b vector of future prices at time T needs to be approximated. `vectorB` creates an array of zeros of length equal to the number of points in the S vector. The array is filled using a `for` loop which approximates the n option prices at M ($f_{n,M}$) by calculating $X - ih$ where i is looped, the option price is equal to $X - ih$ if it is positive and 0 if not, $max\{X - nh, 0\}$.

2.5.3 Approximating 'past' option prices

From the first option price approximation we can approximate all 'past' n option prices by looping through each time step backwards from $t = T$ to $t = 0$ and solving the system of linear equations for f at each step. The b vector approximated with `vectorB` is passed to the function `findPastOptionPrices`, and at each step the vector f ($f_{n,m}$) is approximated. This vector f is then assigned to b and the next time-step ($m - 1$) is approximated.
The issue with this last function is that the Sparse SOR function is ran T times to calculate x for each time step. For large grids (large M) this results in long runtime.

2.5.4 Finite Difference Method

2.6 Iterative algorithms / SOR

3 Code Base and Solution Design

3.1 SOR

There are a number of notable features of the solution given that are discussed here. An important design decision that was made here was to use compressed row storage (csr) for the input matrices. This is chosen as the optimal storage method, due to the nature of the input matrices having a small amount of non-zero entries. In traditional storage methods, each value in the matrix would be stored in memory making for quick access times for any one specific value. This requires $\Theta(n^2)$ storage space, which can cause issues with particularly large matrices. In contrast, csr only stores the non-zero entries, along with the location of that entity. While accessing the value of a given matrix coordinate is not as easy as with the traditional method, the storage required is $O(2t + n)$. This is significantly less, and thus clearly an optimal design choice.

The `main.py` module, from where the program as a whole is instigated and controlled, first handles file input and output. By default, an `input.in` file is read from. This file is stored in the **docs**/ folder, and contains the dimension of the input A, A itself and b all in plain text in array format. On read, these matrices are converted to compressed row storage (csr) for optimal storage and performance. A python dictionary is created containing these elements. Also by default, a `nas_SOR.out` file is used for logging purposes, stored too in the **docs** folder.

It is possible however, to change these defaults and input another file stored locally elsewhere. This is achieved by specifying the path to this file as the first argument to the `main.py` module. Note, that both files with the '.in' and '.mtx' are acceptable input to the program. It is expected that the former be in the same format as that of the default input. However, '.mtx' files can contain a matrix already in csr format. This facilitates the use of matrices sourced from `http://math.nist.gov/MatrixMarket/`. In addition, it is possible to provide a second argument to the `main.py` module specifying the location of the new log file. This will have no effect on the contents of the logs, only the location where they are saved.

Once the input and output files have been handled, a number of checks are carried out on the input matrices. Firstly, a check is made for any zeros on the main diagonal of the **A** matrix. From equation (1), it is clear that a divide-by-zero would occur should there be any zeros on the main diagonal of A, in the $\frac{\omega}{a_{ii}}$ term.

Next, a check is made for diagonal dominance of the **A** matrix. A condition for the convergence on a solution of **x** is that that matrix have a $r_\sigma(C) < 1$ for

$$C = -(D + L)^{-1}U \tag{3}$$

where D, L and U are as described earlier. Having said this, it is appropriate to check for row or column diagonal dominance, and thus sufficiently meet the above criteria. There are two separate methods in the `doMath.py` module to achieve this, using differing methodologies. The `isStrictlyDiagDom` method uses *for* loops, and returns the count of instances where diagonal dominance is violated. The `isStrictlyDiagonallyDominant` method on the other hand, takes advantage of the ability in *numpy* to find the sum of a matrix along each axis. The value on the diagonal is then removed from this sum, and checked against the new sum to check for dominance of that diagonal value.

3.2 Black-Scholes(-Merton)

3.3 Version Control

It was decided to make use of source control, specifically GitHub, to manage the collaboration and multi-developer aspect of this project. The public repository for the current solution is available at the following url `https://github.com/ReidConor/Sparse-SOR/tree/master/`. This contains just just the code used to achieve the solution outlined here, but also the source code for this paper as well as the test matrices and resulting logs used throughout. In addition, this repository provides posterity and a way to retrieve the source code in addition to the files provided in the submission of this paper.

3.4 Project Structure

Here, a brief discussion of the source code structure is included for reference. Spare-SOR is the name of the project, with the `main.py` module directly underneath. This ensures ease of running, as well as better handling dependancies throughout the project.

```
Sparse-SOR
```

```
├── Report
│   ├── (latex files)..
│   └── Report.pdf
├── core
│   ├── doMath.py
│   ├── fileIO.py
│   ├── tridiagM.py
│   └── visualize.py
├── docs
│   ├── Various .in files..
│   ├── Various .mtx files..
│   └── Plots if generated..
├── logs
│   ├── BSM_logs.out
│   ├── nas_SOR.out
│   └── Any other .out generated by providing an argument at runtime
├── tests
│   ├── outfile.txt
│   ├── TestingC.py
│   ├── tests.doc
│   └── unitTests.py
├── main.py
└── README.md
```

The *'Report'* folder obviously contains this paper. The *core* folder contains much of the logic for SOR, as well as file input and output logic and methods for visualising results. The methods for solving the BSM equation is also found here. The *docs* folder contains the input matrices that were tested, and also provides a location to write plots out to if necessary. The *logs* folder obviously contains output files describing what happened in the latest run, and *tests* contains both unit testing and also methods used in the construction and running of tests for the SOR algorithm itself.

3.5 Run Time

As mentioned previously, the `main.py` module controls the flow of the program, calling methods on the adjacent packages. Thus, it is this module that is run to invoke the functionality of the solution. Firstly, it is possible to run this module with no arguments. This will run the SOR algorithm against a default *'.in'* file containing a simple 5×5 matrix A and output the solution to the default logging location, a file called *nas_SOR.out* in the `docs` folder.

Secondly, one can provide an argument when running the module. This would be the path to another data input file to solve using the SOR algorithm. Note, that this file can be in one of two forms; plain text or compress row storage. The former should be held in a *'.in'* file, the latter a *'.mtx'* file. This aids the program in deciding how best to treat the input, while allowing a diverse set of allowable input data. Passing a data file with a different file extension will cause the program to fail, and output 'input file error' in the logs.

Thirdly, it is possible to pass a second parameter to the `main.py` module, which specifies a location in which to save the log file. This log file has been mentioned earlier, and simply provides a way to see what happened in the latest run of Sparse-SOR. Note, it is only possible to provide this alternative location if a path to the input file is also specified.

3.6 Testing

Why testings important, how we found it really useful all the way through the assignment (lol). Brief overview of what tests we have included, maybe some specific problem we meet and how testing helped us solve it?

4 Technical Challenges and Limitations

What were the main technical challenges we faced doing this. What are the limitations of our brilliant solution

5 Results

Our fantastic results

5.1 Black-Scholes-Merton Results

The number of grid points and thus the shape of the matrix depend on the values of M given. The program was run using a maturity of 12 months, a strike price of 100 and three different M values; M=12 for months, arbitrary M=40, and M=365 for days. The first three figures in Appendix 4.2 demonstrate the relationship between stock prices and option prices that satisfy the Black-Scholes equation. Each line from top to bottom represent a time step from T to 0 respectively. We can see that the distribution of the the put option prices vary greatly among the three figures. As k, the length between each time step decreases and the number of steps increases, the range of put option prices increases; the further the option is from maturity the lower its price. There is a reversal point in each figure which can only be spotted looking at the data in 2d, this is most obvious in the first two figures, the yellow line (option prices at time T) goes from being on top (option price is highest for some stock price p) to being on the bottom (option price is lowest for some stock price p). Therefore we can say that there is some threshold option price ($f^*_{n,m}$) and stock price ($p^*_{n,m}$) dependent on the time step (m) at which the slope of the curve decreases. From the 3D plots (figures 4,5 and 6), we can observe that the lower the value for k (the more steps maturity), the faster the fall in option price the further away from maturity and the greater the stock price.

6 Conclusions

Appendix

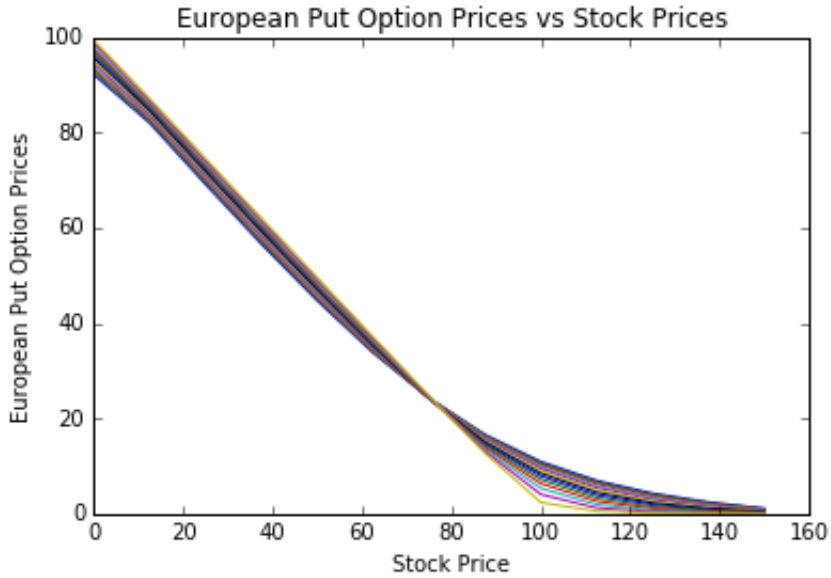

Figure 1: Distribution of put option prices against stock prices for m $T = 12$, $M = 12$, $k = 1$

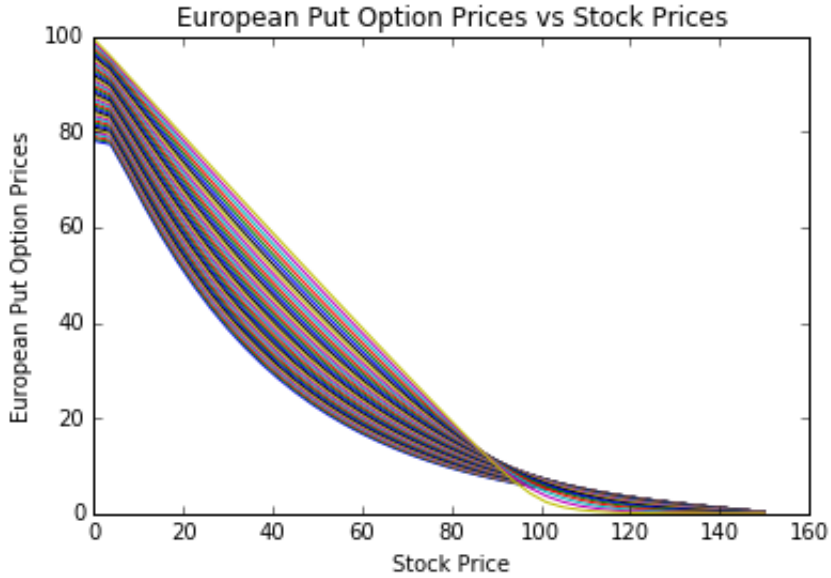

Figure 2: Distribution of put option prices against stock prices for m $T = 12$, $M = 40$, $k = 0.3$

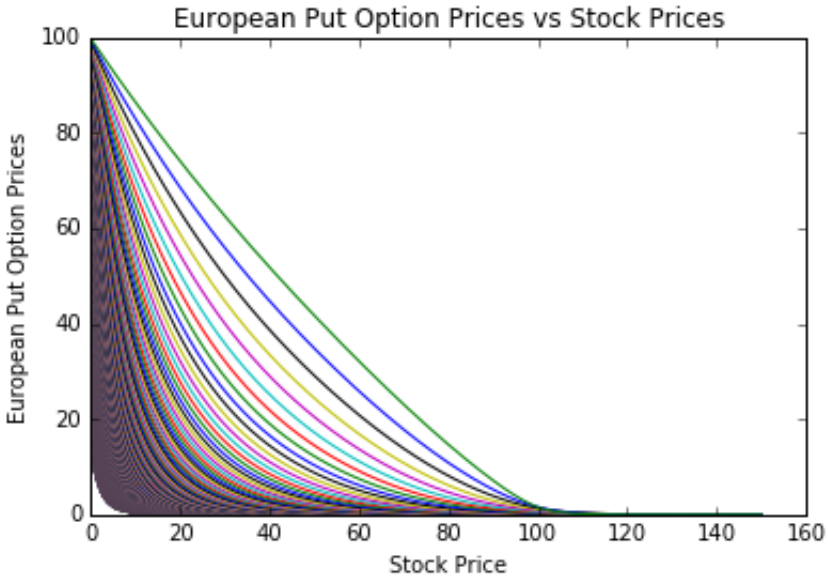

Figure 3: Distribution of put option prices against stock prices for m $T = 12$, $M = 365$, $k = 0.033$

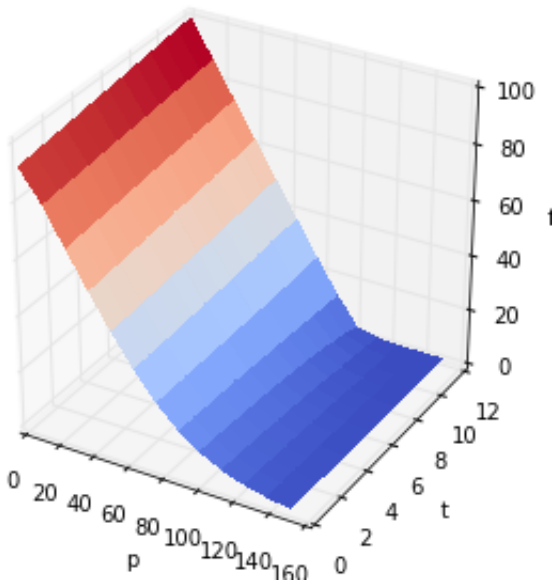

Figure 4: 3D Plot of European Put Option Prices for $T = 12$, $M = 12$, $k = 1$

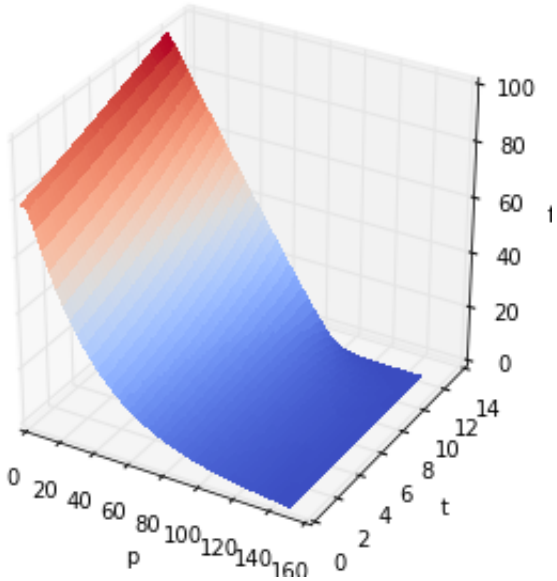

Figure 5: 3D Plot of European Put Option Prices for $T = 12$, $M = 40$, $k = 0.3$

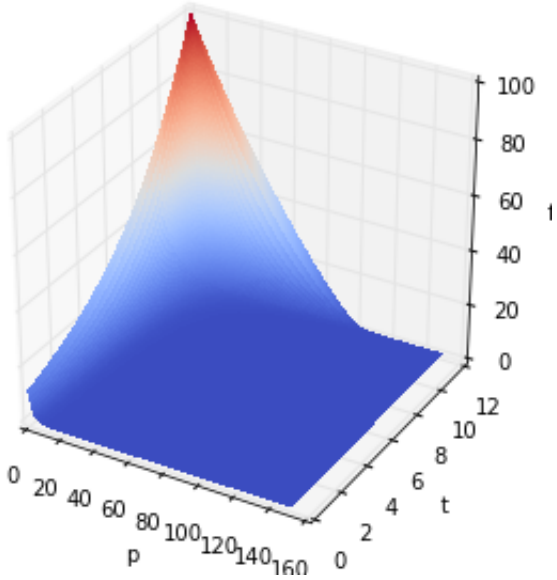

Figure 6: 3D Plot of European Put Option Prices for $T = 12$, $M = 365$, $k = 0.033$

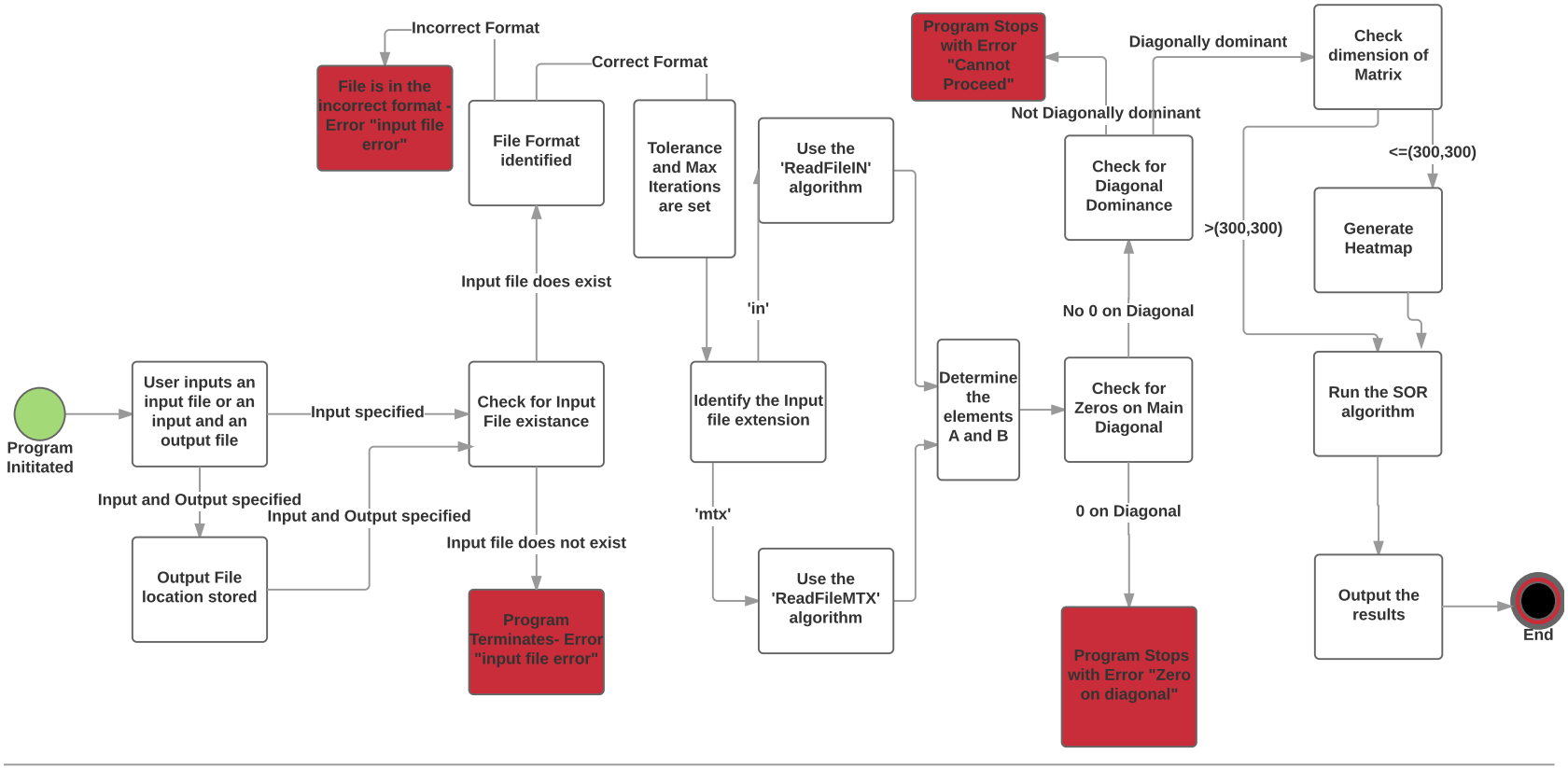

Figure 7: Sparse SOR Application - Program Map

References

[1] Saad, Y., and Van Der Vorst, H. A. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics 123*, 1 (2000), 1–33.