M4N9 PROJECT 3

Isa Majothi
CID: 00950286

The contexts of this report and associated codes are my own work unless otherwise stated

January 2018

1. Schrodinger Equation as an Eigenvalue Problem

Part 1

When seeking the eigenvalues of the symmetric matrix $A := -D + V_N$, alternatively finding the eigenvalues of $L := P^T AP$ is a valid way of finding the eigenvalues of A, where P is the permutation matrix as defined in the task. First recall that two matrices A, $B \in \mathbb{C}^{m\times m}$ are *similar* if there exists a non-singular matrix $X \in C^{m\times m}$ such that $B = X^{-1}AX$. Secondly, a theorem stated in lectures says that if X is a non-singular matrix, then A and $X^{-1}AX$ have the same characteristic polynomial, meaning their eigenvalues and the respective multiplicities of these eigenvalues are the same.

Upon closer examination of the definition of L above, we can conclude that L and A are similar matrices. For any permutation matrix P, we have that $P^{-1} = P^T$ (meaning P is certainly non-singular since P^{-1} exists), and this must clearly hold for the P mentioned above. Hence $L = P^{-1}AP$, and L is similar to A; by the theorem stated above, L and A have the same eigenvalues, therefore it is valid to use the permuted problem to find the eigenvalues of A.

Part 2

In order to form the pentadiagonal matrix, L, I adapted the file `Schrodinger.m` into a function called `schrodingerMatrix`, which takes as its input a value of N for the number of equispaced points in the system. Following the formation of $L \in \mathbb{C}^{N\times N}$, the `MATLAB` function `tridiagMatrix` transforms L into a tridiagonal matrix, T, using Givens rotations. Note that as well as returning T, `tridiagMatrix` also outputs the matrix $Q \in \mathbb{C}^{N\times N}$, which is the Givens rotation matrix such that $T = Q^TLQ$ (see (2) below), since it will be useful in Part 4 when computing the eigenvectors of A.

For any value of N, L is of the form

$$L = \begin{pmatrix} a_1 & b_1 & c_1 & & & & & \\ b_1 & a_2 & 0 & c_2 & & & & \\ c_1 & 0 & a_3 & \ddots & \ddots & & & \\ & c_2 & \ddots & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & a_{N-2} & 0 & c_{N-2} \\ & & & & \ddots & 0 & a_{N-1} & b_2 \\ & & & & & c_{N-2} & b_2 & a_N \end{pmatrix} \tag{1}$$

We've already established that L is pentadiagonal, but since A is symmetric we also have that L is symmetric, as is shown in (1). Since we are aiming to transform L into a tridiagonal matrix, it is only necessary to apply Givens rotations beneath the subdiagonal of L and above the superdiagonal of L i.e. to transform the $\{c_i\}_{i=1}^N$ to zero and any entries that may become non-zero as a result of applying Givens rotations. Furthermore, the symmetry of L enables us to, for any Givens rotation G_{ij} with $i < j$ that sets L_{ji} to zero via multiplication by G_{ij}^T on the left, multiply by G_{ij} on the right so that $L_{ij} = 0$ also - see (2) below.

$$\begin{aligned} T &= G_{N-2,N}^T \dots G_{2N}^T \dots G_{24}^T G_{1N}^T \dots G_{13}^T \, L \, G_{13} \dots G_{1N} G_{24} \dots G_{2N} \dots G_{N-2,N} \\ &:= Q^T L Q \end{aligned} \tag{2}$$

where

$$Q := G_{13} \dots G_{1N} G_{24} \dots G_{2N} \dots G_{N-2,N} \tag{3}$$

In `tridiagMatrix`, I initialise $T = L$ and $Q = I_N$, and these are updated as we apply each successive Givens rotation. I then iterate over the first $N - 2$ columns, using the current entry of T which is in that column and on the subdiagonal (see a in the code). `tridiagMatrix` works its way down the column, and the function `givensMatrix` computes the sine and cosine values - s and c respectively - associated with the Givens rotation that will set the the entry further down the column to zero (initially b in the code). Note that the algorithm found in `givensMatrix` is from Chapter 5 of *Matrix Computations* by Golub and Van Loan, and I implemented it as it was presented; therefore, I hereby do not claim the code in `givensMatrix` to be my own.

These values s and c are used to form the matrix $G \in \mathbb{C}^{2\times 2}$, and since this Givens rotation will only affect 2 rows and 2 columns of our current version of T, it is only these 2 particular rows and columns that are updated by multiplication by G^T on the left and G on the right respectively. This avoids large matrix-matrix multiplications, since forming G in its actual $N \times N$ form would require two matrix-matrix multiplications of $N \times N$ matrices, as we'd have $T^{(k+1)} = G^T T^{(k)} G$ where $T^{(k)}$ is our current version of T. Since each Givens rotation requires 5 flops and a square root, this would result in approximately $4N^3 - 2N^2 + 5$ flops per Givens rotation, whereas storing G as a 2×2 matrix only requires approximately $12N + 5$ flops per Givens rotation. Furthermore, storing G in this way requires significantly less memory, particularly for large N.

After completing the necessary number of Givens rotations, the result is a tridiagonal matrix, T, as required. Since we need to iterate over the first $N-2$ columns and the last $N-2$ rows as well, we'd expect the operation count of my algorithm would be of $O(N^3)$ for large N. To confirm this, I timed how long the process of transforming L into T took for different values of N, since there is clearly a relationship between the operation count and the time taken. Starting from $N = 1000$ until $N = 2500$ in increments of 100 (see the vector *NValuesQ12* in my code), I formed T from L using `tridiagMatrix` in each case, storing the resulting times in the vector *timesQ12*.

To assess how the time taken to form T from L changes as N becomes large, I considered $\log(t)$ against $\log(N)$ so that if $t \sim N^k$ for large N, then we would expect a linear relationship between $\log(t)$ and $\log(N)$, where the constant of proportionality is k. According to the log laws, this would mean

$$\begin{aligned} \log(t) &= k \cdot \log(N) + c \\ \Rightarrow t &\sim N^k \quad \text{for large}\, N \end{aligned} \tag{4}$$

Consequently, to determine the behaviour of t for large N, I plotted the values of *NValuesQ12* against their respective times in *timesQ12* on a log-log scale alongside the function $t = e^c N^k$, where c and k can be found in the vector *coeffQ12* in my `MATLAB` code. Note that c and k are obtained using the `MATLAB` function `polyfit`.

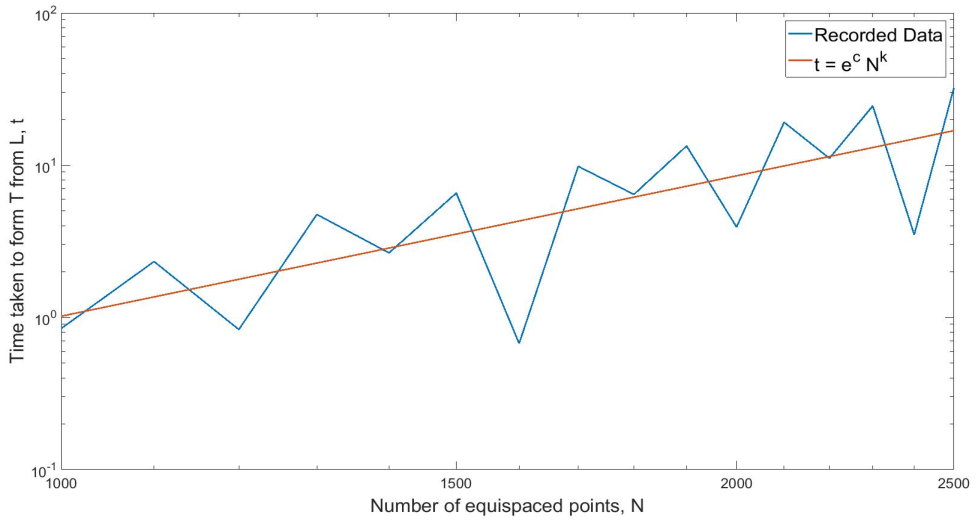

Figure 1: log-log plot of t against N when using `tridiagMatrix`

The value of k obtained using `polyfit` was 3.0639×10^0 in standard form to to 4 decimal places (d.p), and c was -2.1147×10^1 in standard form to 4 d.p. Making reference to (4), this supports our claim that the operation count of applying Givens rotations to form T from L does scale like $O(N^3)$ as N becomes large, since $k = 3$ to 1 d.p. It is worth noting that this is still relatively inexpensive when compared to the method previously discussed of forming each Givens rotation matrix G as an $N \times N$ matrix, since we'd expect this method to cause the operation count to scale like $O(N^4)$ for large N.

It is worth noting that in general, using Householder reflections to transform L into T results in approximately $\frac{4}{3}N^3$ flops for large N, whereas alternatively using Givens rotations requires approximately $2N^3$ for large N. However, in this case we find that it is more efficient to obtain T using Givens rotations since L is pentadiagonal, meaning it is sparse and therefore already has many zeros present (see (1)). Givens rotations enable us to ignore entries that are equal to zero, meaning no operation cost is incurred. This happens on a large scale when applying the method to L, so we can take advantage of the fact that so many zeros are already present in L to obtain T much more efficiently using Givens rotations rather than Householder reflectors.

Part 3

Recall from Part 1 that if $A, B \in \mathbb{C}^{m\times m}$ are similar, then they have the same eigenvalues. Using this fact, we established that A and L have the same eigenvalues. Yet also note that $T = Q^T LQ$, where Q is the Givens rotation matrix as defined in (3). Since all Givens rotations are orthogonal, we have $Q^{-1} = Q^T$, giving $T = Q^{-1}LQ$; by definition, L and T are similar matrices, so they have the same eigenvalues which must also be the same eigenvalues as A. Consequently, finding the eigenvalues of A can be done by finding the eigenvalues of T.

The function `eigenvalueFinder` takes as its inputs an $N \times N$ tridiagonal matrix, T, and a tolerance, *tol*, and performs the QR algorithm with shifts as many times as is necessary until it finds each eigenvalue of T to a degree of accuracy dependent on *tol*. For each eigenvalue, the function `qrShifts` is used to compute it, and once it is stored the last row and column of the most recently updated matrix are deleted; `qrShifts` is then reapplied to successively smaller matrices until the matrix is merely a single element, this being the final eigenvalue.

The function `qrShifts` initialises $\mu^{(0)}$ (see mu in the code) to be the bottom-right most entry of T i.e. $\mu^{(1)} = T^{(0)}_{NN}$, and $tmm1$ is the entry to the immediate left of this i.e. $tmm1 = T^{(0)}_{N,N-1}$; the QR algorithm with shifts is reapplied until $tmm1 < tol$, and at this point we take the bottom-right most entry to be an eigenvalue of T. At iteration k, we compute a QR factorisation of the shifted matrix using the function `tridiagQR` such that

$$T^{(k-1)} - \mu^{(k)} I = Q^{(k)} R^{(k)} \tag{5}$$

where $\mu^{(k)}$ is computed using the Wilkinson Shift as provided in lectures.

We then obtain $T^{(k)}$ by

$$T^{(k)} = R^{(k)} Q^{(k)} + \mu^{(k)} I \tag{6}$$

and this is repeated until *tmm1 < tol*.

Note that this is a valid algorithm for computing eigenvalues of T because after each iteration k we have that $T^{(k-1)}$ and $T^{(k)}$ are similar, meaning that their eigenvalues are the same. This is proven below, using the fact that $[Q^{(k)}]^{-1} = [Q^{(k)}]^T$, since $Q^{(k)}$ is orthogonal.

$$\begin{aligned}
[Q^{(k)}]^T T^{(k-1)} Q^{(k)} &= [Q^{(k)}]^T \left(Q^{(k)} R^{(k)} + \mu^{(k)} I \right) Q^{(k)} \\
&= R^{(k)} Q^{(k)} + \mu^{(k)} I \\
&= T^{(k)}
\end{aligned}$$

In order to exploit the tridiagonal structure of T when computing the QR factorisation at each step, I made use of the fact that T already resembles an upper-triangular matrix with exception of its subdiagonal not being equal to zero. By (5), we are seeking the QR factorisation of $T^{(k-1)} - \mu^{(k)}I$, which is clearly tridiagonal. So conceptually, my algorithm involves applying a sequence of Givens rotations to the matrix $T^{(k-1)} - \mu^{(k)}I$ such that the resulting matrix is upper-triangular, namely R.

Making reference to (2), transforming L into T via Givens rotations entailed multiplication on the left by G_{ij}^T and on the right by G_{ij}. However, we only require an upper-triangular matrix, meaning multiplying $T^{(k-1)} - \mu^{(k)}I$ on the left by the appropriate G_{ij}^T's is sufficient to turn the entries beneath the leading diagonal of $T^{(k-1)} - \mu^{(k)}I$ to zero. This gives

$$\begin{aligned} R^{(k)} &= [G_{N-1,N}^{(k)}]^T[G_{N-2,N-1}^{(k)}]^T \dots [G_{34}^{(k)}]^T[G_{23}^{(k)}]^T[G_{12}^{(k)}]^T \left(T^{(k-1)} - \mu^{(k)}I\right) \qquad (7) \\ &= [Q^{(k)}]^T \left(T^{(k-1)} - \mu^{(k)}I\right), \quad \text{using (3)} \end{aligned}$$

Making use of the fact that $[Q^{(k)}]$ is orthogonal since it is a Givens rotation, we arrive at the desired result

$$T^{(k-1)} - \mu^{(k)}I = Q^{(k)}R^{(k)}$$

In `qrShifts`, at iteration k I initialise $R^{(k)} = T^{(k-1)} - \mu^{(k)}$ and $Q^{(k)} = I_N$, and these are updated accordingly via the appropriate Givens rotation. Using the concept illustrated in (7), $R^{(k)}$ is multiplied by G^T on the left, whilst $Q^{(k)}$ is multiplied by G on the right. Each of these processes requires $6N$ flops, so the total for each Givens rotation is $12N + 5$ flops since one Givens rotation needs 5 flops. Since we need to iterate over the first $N - 1$ columns of $T^{(k-1)} - \mu^{(k)}I$, completing one iteration per column, the operation count of computing the QR factorisation will scale like $O(N^2)$.

Similarly to Part 2, to confirm this I timed how long it took to find the QR factorisation of a tridiagonal matrix $T \in \mathbb{C}^{N\times N}$ for different values of N. Again, starting with $N = 1000$ and ending at $N = 2500$ in increments of 100, I used `schrodingerMatrix` and `tridiagMatrix` to form L and T respectively. I then timed how long it took `tridiagQR` to find the QR factorisation of the tridiagonal matrix for each value of N, and stored these values in the vector *timesQ13*. As before, I plotted these times on a log-log scale (see Figure 2) and found $k = 2.0095 \times 10^0$, and $c = -1.6909 \times 10^1$, where k and c are defined as in (4). This is resounding evidence that the operation count scales like $O(N^2)$ for large N, as $k = 2$ when rounded to 1 d.p.

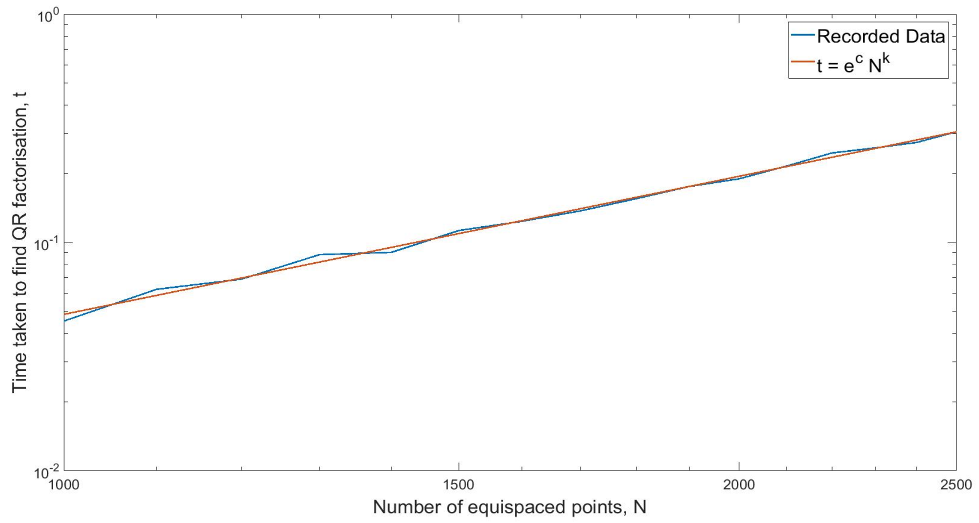

Figure 2: log-log plot of t against N when using `tridiagQR`

For the case where $N = 1024$, I used `eigenvalueFinder` to compute the eigenvalues of T, and stored them in ascending order in the vector *orderedEValues*. A plot of E_n against n can be found in Figure 3, whilst Table 1 consists of the 10 lowest energy levels, which correspond to the 10 smallest eigenvalues, in standard form when rounded to 4 d.p.

n	E_n
1	-1.5906×10^2
2	-8.5625×10^1
3	-3.0858×10^1
4	-1.2203×10^0
5	3.2089×10^{-1}
6	2.1454×10^0
7	2.8690×10^0
8	7.1873×10^0
9	7.8826×10^0
10	1.4690×10^1

Table 1: 10 lowest energy levels, E_n, when $N = 1024$

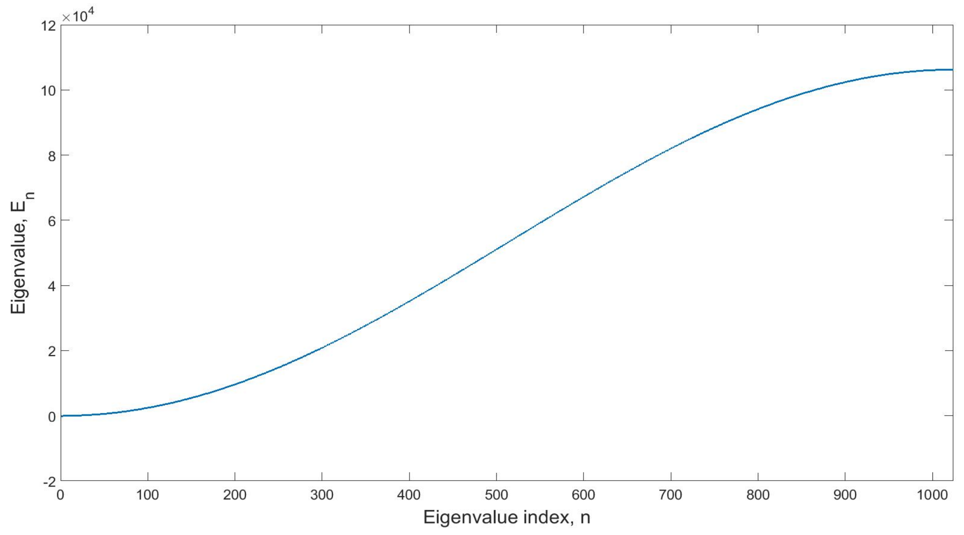

Figure 3: Plot of energy levels, E_n against n for the eigenvalues of A when $N = 1024$

In trying to obtain the eigenvalues of the symmetric $N \times N$ matrix, A, recall we first transformed it into a pentadiagonal matrix, L, before transforming L into a tridiagonal matrix, T; the QR algorithm with shifts was then applied repeatedly to T in order to obtain all of the eigenvalues of T, and hence L and A. The cost of forming T from L was of $O(N^3)$, and computing the QR factorisation of T was of $O(N^2)$. Whilst it is true that the QR algorithm is much less expensive when applied to T as opposed to L, this comes at a cost of $O(N^3)$ since we have to form T in the first place. We also observe that when using Givens rotations as was done in Parts 2 and 3 i.e. by using G as a 2×2 matrix, the cost will not exceed $O(N^3)$.

Combining all of these facts together, it can be argued that forming T at this cost is not necessary for the purpose of computing eigenvalues. This is because if we had applied the QR algorithm with shifts to L directly, this would have also resulted in an operation count of $O(N^3)$, the same operation cost as forming T. However, forming a tridiagonal matrix has its advantages, as will be seen in Part 4 when computing the eigenvectors of the system using inverse iteration. Solving a linear system $Tv = w$, where T is tridiagonal, can be solved at an operation cost of $O(N)$. In short, if we only require the eigenvalues of the system, forming T doesn't appear to be necessary; however, if we also wish to obtain eigenvalues then there is clearly an advantage to forming T, even if it comes at an initial cost.

Part 4

After having obtained the eigenvalues of T, inverse iteration can be used to find the eigenvectors of A, namely the wave functions, Ψ_n. Whilst the matrices A and T being similar may mean that their eigenvalues are the same, this cannot be said for their eigenvectors.

Let λ be an eigenvalue of T, and v be its corresponding eigenvector. Then $Tv = \lambda v$. Recall from (2) that $Q^T LQ = T$, hence $Q^T LQv = \lambda v$, and using the fact that $Q^{-1} = Q^T$ we get $L(Qv) = \lambda(Qv)$; so Qv is an eigenvector of L. Finally, recall that $L = P^T AP = P^{-1}AP$, giving $P^{-1}AP(Qv) = \lambda(Qv)$, and multiplying both sides on the right by P yields $A(PQv) = \lambda(PQv)$; so we see that to obtain the eigenvectors of A from the eigenvectors of T, we merely have to multiply them by PQ. It is for this reason that the function `tridiagMatrix` outputs Q, so it can be later used to obtain the eigenvectors of A as was just described.

The function `eigenvectorFinder` finds the eigenvectors of T using inverse iteration, which is an adaptation of the power iteration method; it works on the basis that for any $\mu \in \mathbb{R}$ such that μ is not an eigenvalue of T, then the eigenvectors of the matrix $(T - \mu I)^{-1}$ are the same as those of T. For each eigenvalue, λ_i, of T, we pick μ_i to be *close* to λ_i i.e. such that $|\lambda_i - \mu_i|^{-1} \gg |\lambda_j - \mu_i|^{-1}$ for $i \neq j$. Note that in `eigenvectorFinder`, μ_i is set to λ_i + *diff*, where *diff* is a small number that is an input of the function; I set *diff* $= 10^{-8}$ to find the eigenvectors when $N = 1024$.

After initialising a random unit vector $v^{(0)}$, we solve the system $(T - \mu_i I)w = v^{(k-1)}$, and set $v^{(k)} = w/||w||$ before proceeding to the next iteration. This is repeated until convergence is reached, where this is measured by how $|\lambda^{(k)} - \lambda_i|$ compares to $\left|\frac{\mu_i - \lambda_i}{\mu_i - \lambda_J}\right|^k$ at each iteration k, because from lectures we know $|\lambda^{(k)} - \lambda_i| = O\left(\left|\frac{\mu_i - \lambda_i}{\mu_i - \lambda_J}\right|^{2k}\right)$. Once the convergence criteria has been satisfied, we take the most recently computed $v^{(k)}$ to be the eigenvector associated with λ_i, and this process is repeated for each eigenvalue of T until all eigenvectors have been found.

As previously touched upon in Part 3, it is in solving $(T - \mu_i I)w = v^{(k-1)}$ that we see forming the tridiagonal matrix, T, becomes advantageous. Once fixing a value of μ_i, the matrix $(T - \mu_i I)$ does not change, and it is tridiagonal. This is beneficial because when iteratively solving the linear system $(T - \mu_i I)w = v^{(k-1)}$, the only thing that changes is the vector on the right-hand side. The function `tridiagSolver` takes as inputs vectors a, b, c and v, where these are the subdiagonal, diagonal and superdiagonal of the tridiagonal matrix, and the right-hand side of the linear system, respectively. In finding the solution, w, the tridiagonal system is modified to transform it into one that can be solved using backwards substitution. Note that the algorithm in `tridiagSolver` is an implementation of the Thomas algorithm for solving tridiagonal linear systems [1], and I do not claim it to be my own.

The benefit of forming T is that it allows inverse iteration to be implemented such that solving the linear system only comes at an operation cost of $8N$ flops, where our tridiagonal matrix is an $N \times N$ matrix. For the majority of linear system solvers, the operation count is of $O(N^3)$ for large N, examples being using an LU factorisation followed by one forward substitution and one backward substitution. Even for matrices with *nice* properties, such as otrhogonal matrices, solving such linear systems scales like $O(N^2)$ for large N; to have a solver where the operation count scales like $O(N)$ for large N is clearly a massive computational saving, and it justifies the initial operation cost of forming T from L.

To confirm that this was indeed the case, I timed how long `tridiagSolver` took to solve a tridiagonal linear system 10000 times for the values of N used in Parts 2 and 3, before taking the average of these times for each respective N (see *NValuesQ14* and *timesQ14* in my code). As before, I plotted these times, t, against N on a log-log scale in Figure 4, to determine the value of k for large N. Using `polyfit`, I found $k = 9.7389e-01$, which is strong evidence to support the claim that `tridiagSolver` solves linear systems in $O(N)$ operations since, $k = 1$ to 1 d.p.

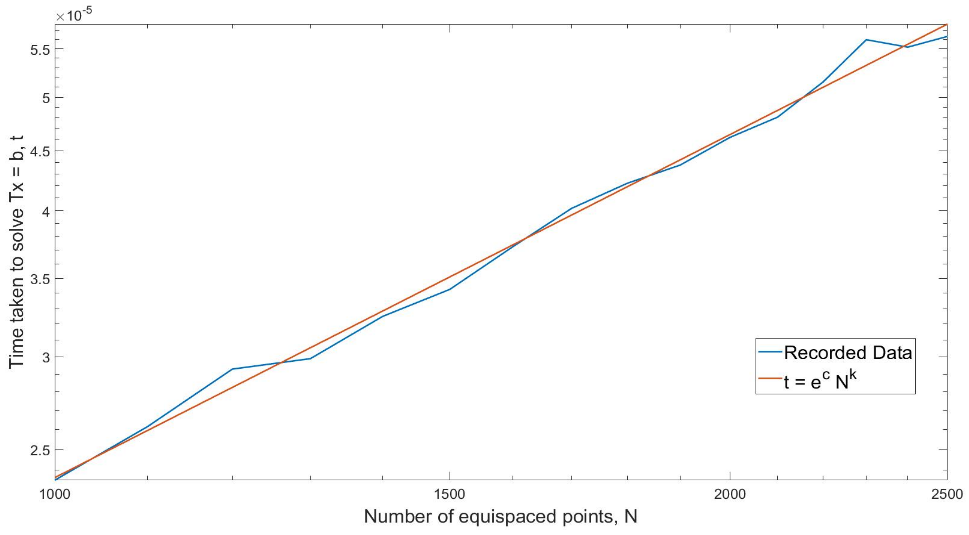

Figure 4: log-log plot of t against N when using `tridiagSolver`

For the case where $N = 1024$, I used `eigenvectorFinder` to compute the eigenvectors of T, and these were stored in the $N \times N$ matrix $eVectors$. To then obtain the eigenvalues of A, I multiplied $eVectors$ by PQ so that the N eigenvectors of A formed the columns of this matrix (see Psi in code). For the negative eigenvalues i.e. $E_n < 0$, I have included a plot of $|\Psi_n|^2$ against x in Figure 5, as well as the case when $n = 10$.

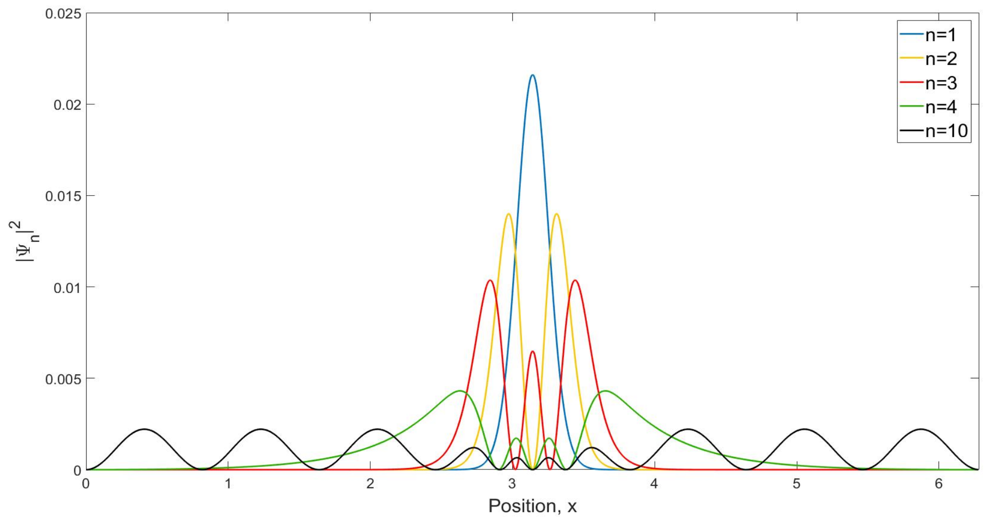

Figure 5: Plot of wave functions $|\Psi_n|^2$ against x for different values of n when $N = 1024$

It is clear that $|\Psi_n|^2$ is symmetric about $x = \pi$ over the domain of x in all five cases, but after analysing these five cases in Figure 5, it appears that as E_n increases there are a few noticeable trends that occur. For any given n, is seems as though the number of maxima of $|\Psi_n|^2$ is also n, and the number of minima $n-1$. Furthermore, as n increases the amplitude of the wave functions $|\Psi_n|^2$ decreases, because for $n = 1$ the peak value is approximately 0.02, whilst this decreases to approximately 0.015, 0.01, 0.005 and 0.0025 for $n = 2, 3, 4$ and 10 respectively. Both the increase in the number of maxima and the decrease in peak value as n increases are to be expected, because `eigenvectorFinder` normalises the computed eigenvectors. Hence the sum of $|\Psi_n|^2$ over the domain would be 1, meaning if the peak value of $|\Psi_n|^2$ is higher for smaller n then the function will decrease at a much faster rate than in other cases in which there are more turning points but where $|\Psi_n|^2$ at these maxima is lower in magnitude.

2. Portfolio Optimisation with Conjugate Gradients

Part 1

Using the code provided in the file `PortfolioOpt.m`, I modified this into a function called `portOptimiseLDLT`, which takes as its inputs a parameter $\gamma \in [0, 1]$, the symmetric positive definite (spd) matrix Σ and the vector $\bar{p}$ as they are given to us in the file `PortfolioData.m`. With regards to solving the optimisation problem, `PortfolioOpt.m` was missing the part solving the linear system

$$A \begin{bmatrix} \Delta x_n \\ \lambda \end{bmatrix} = \begin{bmatrix} g_n \\ 0 \end{bmatrix} \tag{8}$$

In `portOptimiseLDLT`, I implement the LDL^T factorisation of A to solve the linear system. Note that the code found in `portOptimiseLDLT` which computes the LDL^T factorisation of A is the algorithm from Chapter 4 of *Matrix Computations* by Golub and Van Loan as it was presented, and I hereby do not claim it to be my own.

In order to apply this LDL^T algorithm to A, we require that A is symmetric. By definition, we have

$$A = \begin{bmatrix} M & \mathbf{1} \\ \mathbf{1}^T & 0 \end{bmatrix}$$

where $M = 2\gamma\Sigma + \Phi(x_n)$ and $\mathbf{1}$ is the vector of ones of appropriate size. By definition, both Σ and $\Phi(x_n)$ are symmetric matrices, since Σ is spd and therefore obviously symmetric, and $\Phi(x_n)$ is a diagonal matrix which is must also be symmetric. Hence M is symmetric, for it is the sum of 2 symmetric matrices, and A must also be symmetric by its construction. Thus, using the LDL^T is an appropriate choice of algorithm.

The advantage of using this method to solve (8) is that once $A = LDL^T$ has been obtained, the system can be solved using one forward substitution of N^2 flops, followed by one diagonal solver requiring N flops, and finally one backward substitution also of N^2 flops. Computing the LDL^T algorithm requires $N^3/3$ flops, but this is half the operation count of using Gaussian elimination to solve (8); for large N, it is evident that this method is less expensive than using Gaussian elimination.

After varying γ from 0.1 to 0.9 with steps of $\Delta = 0.1$, I computed the risk and return in each case, storing them in the vectors *returnValuesQ21* and *riskValuesQ21* respectively, which are reported in Table 2. I have also included the two largest entries of x^* for each value of γ, where x^* is the optimal solution of where the portfolio should be invested. All values are reported in standard form to 4 d.p.

γ	Return, $\bar{p}^T x^*$	Risk, $x^* \Sigma x$	Largest Entry Of x^*	Second Largest Entry Of x^*
0.1	3.9710×10^{-3}	1.6485×10^{-3}	1.0000×10^{0}	2.7570×10^{-7}
0.2	3.8952×10^{-3}	1.2048×10^{-3}	6.8542×10^{-1}	3.1458×10^{-1}
0.3	3.8596×10^{-3}	1.0933×10^{-3}	5.4177×10^{-1}	4.5298×10^{-1}
0.4	3.7326×10^{-3}	8.5516×10^{-4}	3.9333×10^{-1}	3.7500×10^{-1}
0.5	3.6309×10^{-3}	7.2830×10^{-4}	3.3712×10^{-1}	2.9798×10^{-1}
0.6	3.5597×10^{-3}	6.6937×10^{-4}	3.6699×10^{-1}	2.9869×10^{-1}
0.7	3.4729×10^{-3}	6.2236×10^{-4}	3.8673×10^{-1}	2.6492×10^{-1}
0.8	3.3720×10^{-3}	5.8813×10^{-4}	3.8001×10^{-1}	2.3329×10^{-1}
0.9	2.3149×10^{-3}	4.2129×10^{-4}	2.8230×10^{-1}	1.0539×10^{-1}

Table 2: Values of expected return, expected risk and the two largest values of x^* for different values of γ

Part 2

Assuming there are N potential investments, by inspection this means that $\mathbf{1} \in \mathbb{C}^N$. Now for any given vector $v \in \mathbb{C}^N$, the null space of v^T is an $(N-1)$-dimensional hyperplane, H, such that $H \cong \mathbb{C}^{N-1}$ i.e. H is isomorphic to $\mathbb{C}^{N-1}$. Hence there exists a linearly independent basis of H, which must be of size $N-1$ by results in linear algebra.

Since we are looking for the null space of $v = \mathbf{1}^T$, we require $N-1$ linearly independent vectors $\{\mathbf{z}_i\}_{i=1}^{N-1}$ such that $\mathbf{1}^T \mathbf{z}_i = 0$ for $i = 1, \dots, N-1$. If we let $\mathbf{z}_i = (z_{i1}, \dots, z_{iN})^T$, then this means that $\mathbf{1}^T \mathbf{z}_i = \sum_{j=1}^{N} z_{ij} = 0$.

In order to satisfy $\mathbf{1}^T Z = 0$, where $Z \in C^{N \times N}$, such that the columns of Z span the null space of $\mathbf{1}^T$, we simply need to pick $\{\mathbf{z}_i\}_{i=1}^{N}$ to be the columns of Z such that the sum of each column adds to zero and the first $N-1$ columns are linearly independent. An obvious choice for such a $Z = (z_{kl}) \in \mathbb{C}^{N \times N}$ is

$$Z = \begin{cases} (z_{k,k}) = 1 & \text{for} \quad k = 1, \cdots, N \\ (z_{k+1,k}) = -1 & \text{for} \quad k = 1, \cdots, N-1 \\ (z_{1,N}) = -1 & \\ 0 & \text{otherwise} \end{cases}$$

Demonstrated visually, this looks like

$$Z = \begin{pmatrix} 1 & & & & & & & -1 \\ -1 & 1 & & & & & & \\ & -1 & 1 & & & & & \\ & & \ddots & \ddots & & & & \\ & & & \ddots & \ddots & & & \\ & & & & \ddots & 1 & & \\ & & & & & -1 & 1 & \\ & & & & & & -1 & 1 \end{pmatrix}$$

with any blank entries being zeros.

After analysing this illustration of Z, it is evident that the first $N-1$ columns are linearly independent. This is because given any one of these $\mathbf{z}_i$, it is impossible to express it as a linear combination of the other $\mathbf{z}_j$ with $j \neq i$. Furthermore, the condition $\mathbf{1}^T Z = 0$ is satisfied since the sum of the entries in each column is clearly zero, meaning this Z is suitable for dealing with the equality constraint.

Part 3

Recall that $M = 2\gamma\Sigma + \Phi(x_n)$, where Σ is the spd covariance matrix of the investment prices and $\Phi(x_n)$ is a diagonal matrix, with diagonal entries $\Phi_{ii}(x_n) = t/(x_{n,i}^2) > 0$ since $t > 0$. To prove that $\tilde{M} = Z^T M Z$ is positive semidefinite i.e. $y^T \tilde{M} y \geq 0$ for all y, I will first show that M is spd. Note that Z here is as it was defined in Part 2.

By Part 1, we know that M is symmetric, so it only remains to show that M is positive definite i.e. $y^T M y > 0$ for all $y \neq 0$. Using the definition of M, we observe that

$$\begin{aligned} y^T M y &= y^T \left(2\gamma\Sigma + \Phi\right) y \\ &= 2\gamma(y^T \Sigma y) + y^T \Phi y \end{aligned}$$

For $y \neq 0$, $y^T \Sigma y > 0$ because Σ is spd, and since $\gamma \in [0, 1]$ this means $2\gamma(y^T \Sigma y) \geq 0$. Also note that Φ is a diagonal matrix whose entries are *all* positive; this means

$$y^T \Phi y = \sum_{i=1}^{N} \Phi_{ii}\, y_i^2 > 0$$

Combining these two results gives us the desired result, namely that $y^T M y > 0$. Hence M is spd.

Given that M is spd, we can certainly say that M is positive semidefinite. Now using a result about positive semidefinite matrices, namely if M is positive semidefinite then $Q^T M Q$ is also positive semidefinite [2], we can conclude that $\tilde{M}$ is positive semidefinite.

Rather than using the LDL^T factorisation of A to solve the linear system in (8), alternatively we can implement the conjugate gradient (CG) method to solve the system $\tilde{M}\Delta\tilde{x}_n = \tilde{g}_n$, where $\tilde{M} = Z^T M Z$, $\Delta x_n = Z\Delta\tilde{x}_n$ and $\tilde{g}_n = Z^T g_n$. Again using the code provided in `PortfolioOpt.m`, I modified this into a function called `portOptimiseCG`, which performs exactly the same task as `portOptimiseLDLT` with the exception that the CG method is used to solve the linear system $\tilde{M}\Delta\tilde{x}_n = \tilde{g}_n$, whereas `portOptimiseLDLT` used the LDL^T of the matrix A to solve (8). Note that the CG algorithm implemented in `portOptimiseCG` is from lectures, and I hereby do not claim it to be my own.

Since the N columns of Z are not linearly independent, Z is not of full rank and is therefore singular. Therefore, $\tilde{M}$ is also singular and the system $\tilde{M}\Delta\tilde{x}_n = \tilde{g}_n$ does not have a unique solution. However, this does not imply that it has no solution; if $\text{rank}(\tilde{M}|\tilde{g}_n) = \text{rank}(\tilde{M}) < N$, then the system has infinitely many solutions, whereas if $\text{rank}(\tilde{M}|\tilde{g}_n) > \text{rank}(\tilde{M})$ then there are no solutions [3]. Since $\text{rank}(Z) = N - 1$, we have that $\text{rank}(\tilde{M}) = N - 1$. Consequently, $\text{rank}(\tilde{M}|\tilde{g}_n) > \text{rank}(\tilde{M}) = N - 1$ is impossible, because if it were true then we could only have that $\text{rank}(\tilde{M}|\tilde{g}_n) = N$, but this would imply that there is a unique solution; this is a contradiction because $\tilde{M}$ is singular, and a unique solution doesn't exist. Hence the system $\tilde{M}\Delta\tilde{x}_n = \tilde{g}_n$ is of the former case, namely $\text{rank}(\tilde{M}|\tilde{g}_n) = \text{rank}(\tilde{M}) < N$; therefore, there are infinitely many solutions and we can use the CG method to solve this system despite $\tilde{M}$ not being of full rank, because the existence of solutions is guaranteed - in fact, there are infinitely many of them.

To compare the performance of the CG method relative the LDL^T method, I timed how long each method took to solve the optimisation problem; the vectors *timesQ21* and *timesQ23* contain the times taken for `portOptimiseLDLT` and `portOptimiseCG` to run, respectively, for each value of γ. I also counted the number of iterations required, namely how many LDL^T factorisations were needed in total as well as the number of times the CG method was used. All results on timings and iteration counts can be found in Table 3, in standard form rounded to 4 d.p where appropriate.

γ	Run Time For `portOptimiseLDLT`	Run Time For `portOptimiseCG`	Iteration Count For `portOptimiseLDLT`	Iteration Count For `portOptimiseCG`
0.1	3.7561×10^0	1.4528×10^1	673	291465
0.2	4.1348×10^0	2.0247×10^1	742	425380
0.3	5.0454×10^0	3.4727×10^1	908	744968
0.4	4.7678×10^0	3.7559×10^1	865	789006
0.5	4.6987×10^0	3.5329×10^1	835	725177
0.6	5.0663×10^0	4.2007×10^1	913	903872
0.7	4.9529×10^0	4.2704×10^1	899	895009
0.8	4.9023×10^0	4.4485×10^1	893	944336
0.9	5.1231×10^0	4.8881×10^1	839	1027064

Table 3: Run times and iteration counts obtained when running `portOptimiseLDLT` and `portOptimiseCG` for different values of γ

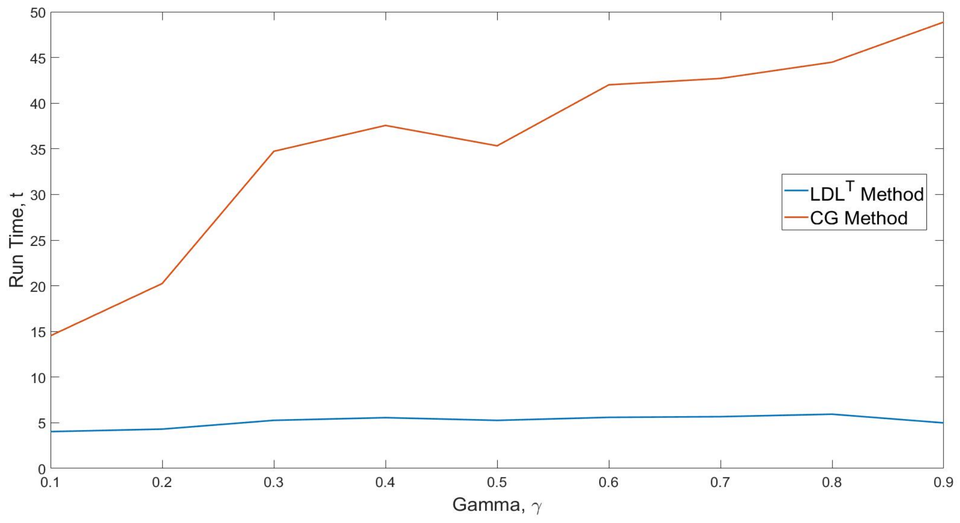

Figure 6: Graph of run times of LDL^T and CG methods for different values of γ

It is clear from the data in both Table 3 and Figure 6 that using the CG method to solve the linear system performs worse than the LDL^T method, and this was the case for every value of γ. The run time when using LDL^T factorisation was of $O(10^0)$ for all values of γ, whereas it was of $O(10^1)$ when using the CG method. This stark difference is reflected visually in Figure 6, as the run time, t, when using the CG method is evidently substantially larger than the t obtained using the LDL^T method for its respective value of γ.

The readings in Table 3 provide some insight as to why this was the case. For all values of γ, we only required $O(10^2)$ LDL^T factorisations to find the solution to the optimisation problem. However, with the exception of $\gamma = 0.9$, the number of times CG was applied was of $O(10^5)$, an order difference of $O(10^3)$. The case where $\gamma = 0.9$ performed particularly badly, as the number of applications of the CG method was of $O(10^6)$. Other notable instances in which the CG method performed particularly badly are $\gamma = 0.6, 0.7$ and 0.8.

It would also appear that, in general, increasing the value of γ leads to in increase in the number of applications of the CG method, and consequently an increase in the run time to find the optimal solution; the red line in Figure 6 clearly shows. On the other hand, varying γ doesn't appear to make any particular difference in the performance of `portOptimiseLDLT`, as for all values of γ the run times and iteration counts for this method were of the same order, and didn't change drastically or with any obvious trend. The blue line in Figure 6 supports this, as the plot of run times for the LDL^T method appears fairly constant around $t = 4$

Part 4

It is evident from Part 3 that using the CG method as a means to solve the linear systems that appear in the optimisation process is highly inefficient. In order to combat this, using a suitable preconditioner on the system $\tilde{M}\Delta\tilde{x}_n = \tilde{g}_n$ will hopefully lead to a reduction in the number of iterations carried out using the CG method, and thus also a lesser computation run time. Rather than solving $\tilde{M}\Delta\tilde{x}_n = \tilde{g}_n$, using a preconditioning matrix, C, gives us the system $C^{-1}\tilde{M}\Delta\tilde{x}_n = C^{-1}\tilde{g}_n$, or $\bar{M}\Delta x_n = \bar{g}_n$, where $\bar{M} := C^{-1}\tilde{M}$ and $\bar{g}_n := C^{-1}\tilde{g}_n$. Clearly, the two linear systems have the same solution, namely Δx_n, but by incorporating C into the linear system we hope that it becomes *easier* to solve.

For any choice of C, we require that it is invertible, otherwise C^{-1} will not exist. However, in addition to this, we would also desire that C has other properties so that convergence to the solution is more rapid since this is the very purpose of introducing C. Firstly, C must be somewhat similar to $\tilde{M}$, and by this we mean that we pick C such that $\tilde{M}^{-1} \approx C^{-1}$ i.e. $||I - C^{-1}\tilde{M}|| < 1$. In addition to this, any linear system $Cx = b$ should be easy to solve, since we will be required to solve such linear systems at every iteration in the preconditioned CG method.

Upon inspection, I noticed that the spd matrix $\tilde{M}$ is largely dominated by its leading diagonal; in order to fulfil the criteria of picking C such that it speeds up the rate of convergence towards the solution, I selected C to be the diagonal matrix whose diagonal entries are those of $\tilde{M}$ i.e. $C_{ii} = \tilde{M}_{ii}$ for $i = 1, \ldots, N$. Since there is a cost associated with implementing a preconditioning matrix, I chose this C since it is readily obtainable from $\tilde{M}$. More significantly, I chose it because to solve $Cx = b$ for a diagonal matrix C is notably simple, and can be done in $O(N)$ flops; this is not the case when choosing C to be lower-triangular, for this would require $O(N^2)$ flops.

To implement this preconditioner, C, I defined the function `portOptimisePCCG` which again finds the optimal solution to the optimisation problem, but now by applying the CG method to the preconditioned system. Note that the preconditioned CG algorithm was provided to us in lectures and I hereby do not claim it to be my own. After using `portOptimisePCCG` to find the solution for the values of γ mentioned previously, I stored the run time of each case in the vector *timesQ24*, as well as the iteration counts in the vector *iterCountsQ24*. These values are reported in Table 4, in standard form rounded to 4 d.p where appropriate. I have also included the data in Table 3 that was obtained by using `portOptimiseCG`, so that a direct comparison can be made between both the run times and iteration counts as to see whether implementing C led to an improvement in performance. Furthermore, Figure 7 represents these run times graphically.

γ	Run Time For `portOptimisePCCG`	Run Time For `portOptimiseCG`	Iteration Count For `portOptimisePCCG`	Iteration Count For `portOptimiseCG`
0.1	7.0469×10^0	1.4528×10^1	147024	291465
0.2	7.0699×10^0	2.0247×10^1	150443	425380
0.3	9.0010×10^0	3.4727×10^1	191845	744968
0.4	9.1571×10^0	3.7559×10^1	197686	789006
0.5	7.7903×10^0	3.5329×10^1	165248	725177
0.6	8.6864×10^0	4.2007×10^1	186927	903872
0.7	9.6397×10^0	4.2704×10^1	204518	895009
0.8	9.8273×10^0	4.4485×10^1	209092	944336
0.9	8.8439×10^0	4.8881×10^1	185883	1027064

Table 4: Run times and iteration counts obtained when running `portOptimisePCCG` and `portOptimiseCG` for different values of γ

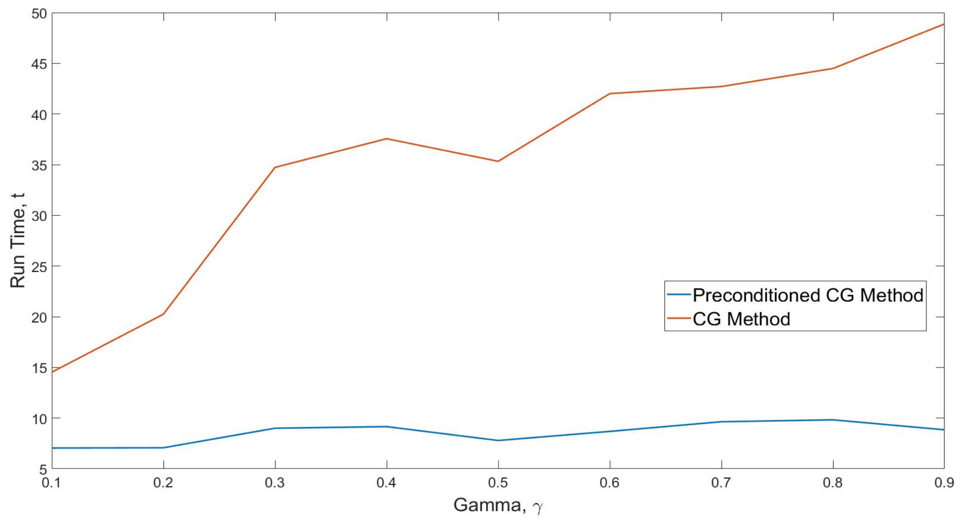

Figure 7: Graph of run times of CG and preconditioned CG methods for different values of γ

It is clear that the preconditioned CG method results in a massive improvement in performance relative to the standard CG method. Whilst the iteration counts of `portOptimisePCCG` were of $O(10^5)$ for all values of γ, preconditioning clearly led to a massive reduction in the number of iterations. The factor of this reduction appears to grow as γ increases, because for $\gamma = 0.1$ the number of iterations approximately halved as a result of preconditioning, whilst the effect for $\gamma = 0.7, 0.8$ and 0.9 is approximately 20% less iterations needed.

However, it is in the run times of `portOptimisePCCG` that the benefit is really highlighted. Not only was the run time of `portOptimisePCCG` less than that of `portOptimiseCG` for every value of γ (see Figure 6), but we also observe that the run time never exceeded $O(10^1)$ for `portOptimisePCCG`, whereas the run time appears to grow for `portOptimiseCG` as γ increases. This further highlights the advantage of using the preconditioned CG method, since at a relatively low cost of solving the linear system for my diagonal matrix, C, we observe a much more rapid rate of convergence.

Comparing the performance of `portOptimisePCCG` to `portOptimiseLDLT` shows that the run time of `portOptimiseLDLT` was less for all values of γ. However, using `portOptimisePCCG` led to a massive improvement compared to `portOptimiseCG`, so much so that both `portOptimiseLDLT` and `portOptimisePCCG` had run times of $O(10^0)$ for all γ. It is possible that by using an alternative C that we may find that the performance of `portOptimisePCCG` trumps that of `portOptimiseLDLT`.

Bibliography

[1] www.quantstart.com/articles/Tridiagonal-Matrix-Solver-via-Thomas-Algorithm

[2] en.wikipedia.org/wiki/Positive-definite_matrix

[3] math.stackexchange.com/questions/1721893/in-ax-b-if-a-is-not-invertible-there-are-no-solutions-or-infinity-how-to-d