Imperial College
London

COURSEWORK 2

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Computational Linear Algebra

Author:
Richard Fu (CID: 01852979)

Date: December 8, 2022

1 QUESTION 1

1.1 Part a

The function that generates the matrix $\boldsymbol{A}^{(n)}$ as described is located in cw2/q1.py, and is called **generate_A()**. In the same file, there is a function called **get_rho()** which can be called on a matrix to find the growth factor ρ as defined in section 4.3.
Below these functions is the code that creates $\boldsymbol{A}^{(6)}$, and prints the corresponding growth factor ρ, which has a numerical value of 32.0.

1.2 Part b

The LU factorisation of $\boldsymbol{A}^{(n)}$ is given by $\boldsymbol{A}^{(n)} = \boldsymbol{LU}$, where:

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ -1 & 1 & \ddots & \vdots & \vdots \\ -1 & -1 & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ -1 & -1 & \dots & -1 & 1 \end{bmatrix}, \quad \boldsymbol{U} = \begin{bmatrix} 1 & 0 & \dots & 0 & 2^0 \\ 0 & 1 & \ddots & \vdots & 2^1 \\ 0 & 0 & \ddots & 0 & \vdots \\ \vdots & \vdots & \ddots & 1 & 2^{n-2} \\ 0 & 0 & \dots & 0 & 2^{n-1} \end{bmatrix}. \tag{1}$$

To see this, first notice that the first $(n-1)$ columns of $\boldsymbol{U}$ are $(\boldsymbol{e}_1, \boldsymbol{e}_2, \dots \boldsymbol{e}_{n-1})$, where $\boldsymbol{e}_i$ is the i-th canonical basis vector. This means that the first $(n-1)$ columns of $\boldsymbol{LU}$ are just the first $(n-1)$ columns of L.
Now, let us look at the entries in the n-th column of $\boldsymbol{LU}$.

$$(\boldsymbol{LU})_{in} = \begin{cases} 2^0 & i = 1 \\ 2^{i-1} - \sum_{k=0}^{i-2} 2^k & i \geq 2. \end{cases} \tag{2}$$

Trivially $2^0 = 1$, and also $2^{i-1} - \sum_{k=0}^{i-2} 2^k = 2^{i-1} - \frac{1-2^{i-1}}{1-2} = 2^{i-1} + 1 - 2^{i-1} = 1$.

Therefore, $\boldsymbol{LU}$ is given by $\boldsymbol{L}$ with the last column changed to a vector of ones, which is precisely $\boldsymbol{A}^{(n)}$.

$\rho = \frac{max_{ij}|u_ij|}{max_{ij}|a_ij|} = \frac{2^{n-1}}{1} = 2^{n-1}$, so the growth factor for $\boldsymbol{A}^{(n)}$ does depend on n.

1.3 Part c

The functions **error_LUP()** and **error_solve_LUP()** located in cw2/q1.py help to calculate the errors from the LUP factorisation, and the errors in the least squares solution to $\boldsymbol{Ax} = \boldsymbol{b}$ respectively. Underneath we call the functions on $\boldsymbol{A}^{(60)}$, and we see that the errors are quite large for both.

1.4 Part d

The function **random_matrix()** in cw2/q1.py helps to generate random $n \times n$ matrices with entries sampled from a Uniform$\left(-\frac{1}{n}, \frac{1}{n}\right)$ distribution.
Located in cw2/plotting_scripts/q1_plots.py is the code that calculates the growth factor ρ of a random matrix created by **random_matrix()**, for a range of n values, and then generates the plot with log-scales we can see in Figure 1. We can see that the black scatter points seem to follow a straight line quite well, which suggests that there is a power relationship between ρ and n; i.e. $\rho = an^k$. In fact, the red line we see is actually the curve $\rho = \frac{1}{3}n^{\frac{3}{4}}$, and this appears to be a good fit for the scatter points for the random matrices generated.

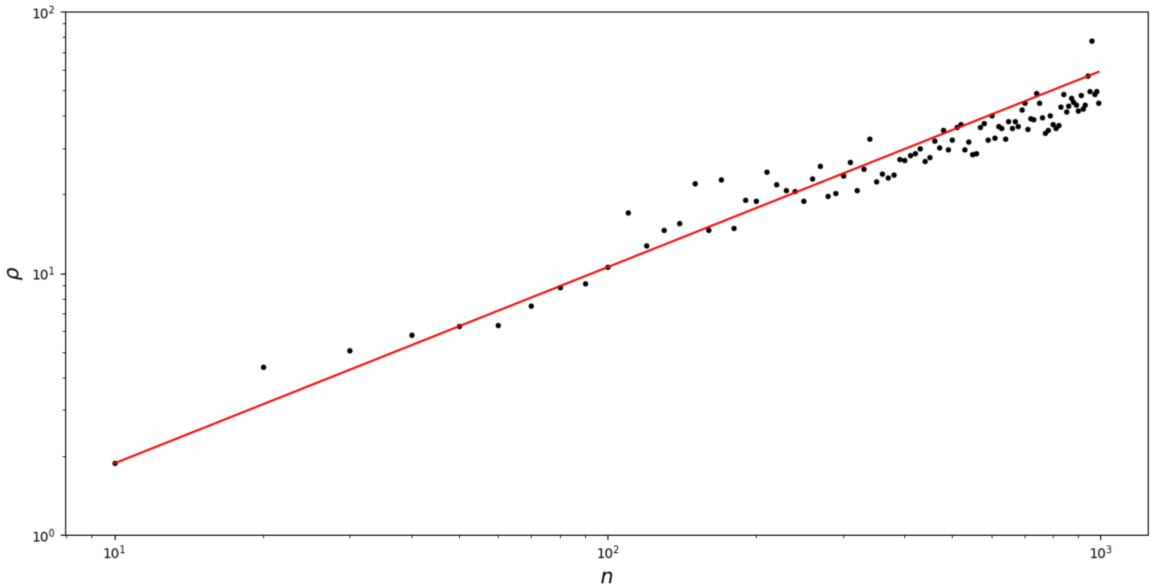

Figure 1: Scatter plot of growth values ρ for varying of n

This investigation shows that in general we will not get an exponential relationship between matrix dimension n and growth factor ρ like we did for matrices $A^{(n)}$, so it is not all bad news for using Gaussian elimination.

2 QUESTION 2

2.1 Part a

The function **MGS_solve_ls()** is located in cw2/q2.py. The pytests corresponding to this function are located in cw2/test2.py and are called by running **test_MGS_solve_ls()**.

2.2 Part b

In cw2/plotting_scripts_q2_plots.py there is code that repeatedly generates random x^* and calculates the numerical error when solving the least squares problem using **MGS_solve_ls()**, and also when using **householder_ls()** from cla_utils. We then plot these errors with log y-axis scale as seen in Figure 2.

Figure 2

Clearly, we can see that the **householder_ls()** function outperforms **MGS_solve_ls()** with respect to error in numerical least squares solution.

2.3 Part c

First,

$$A_+ = \begin{bmatrix} A & | & b \end{bmatrix}. \tag{3}$$

$$A_+ = \begin{bmatrix} \hat{Q} & | & q_{n+1} \end{bmatrix} \left[\begin{array}{c|c} \hat{R} & z \\ \hline \mathbf{0} & \rho \end{array}\right], \tag{4}$$

$$= \begin{bmatrix} \hat{Q}\hat{R} & | & \hat{Q}z + \rho q_{n+1} \end{bmatrix}. \tag{5}$$

$$(3) + (5) \implies b = \hat{Q}z + \rho q_{n+1} \tag{6}$$

Now to find an equation for z, we left multiply (4) by $\left[\hat{Q} \mid q_{n+1}\right]^*$:

$$\left[\hat{Q} \mid q_{n+1}\right]^* \left[A \mid b\right] = \left[\begin{array}{c|c} \hat{R} & z \\ \hline \mathbf{0} & \rho \end{array}\right], \tag{7}$$

$$\implies \left[\begin{array}{c} \hat{Q}^* \\ \hline q_{n+1}^* \end{array}\right] \left[A \mid b\right] = \left[\begin{array}{c|c} \hat{R} & z \\ \hline \mathbf{0} & \rho \end{array}\right], \tag{8}$$

$$\implies \left[\begin{array}{c|c} \hat{Q}^* A & \hat{Q}^* b \\ \hline q_{n+1}^* A & q_{n+1}^* b \end{array}\right] = \left[\begin{array}{c|c} \hat{R} & z \\ \hline \mathbf{0} & \rho \end{array}\right], \tag{9}$$

$$\implies z = \hat{Q}^* b. \tag{10}$$

2.4 Part d

Our modified algorithm proceeds as follows:

1. First form $A_+ = \left[A \mid b\right]$.

2. Find the reduced QR factorisation $\left[\hat{Q} \mid q_{n+1}\right] \left[\begin{array}{c|c} \hat{R} & z \\ \hline \mathbf{0} & \rho \end{array}\right] = A_+$ using our modified Gram-Schmidt implementation **GS_modified()** from cla_utils.

3. Slice out both the vector $z = \hat{Q}^* b$, and the matrix $\hat{R}$.

4. Solve $\hat{R}x = z$ using our backward substitution implementation **solve_U()** from cla_utils.

The operation count for modified Gram-Schmidt is $\mathcal{O}(mn^2)$, and for backward substitution it is $\mathcal{O}(n^2)$. Therefore, the operation count for our modified algorithm is dominated by modified Gram-Schmidt, and the operation count is $\mathcal{O}(mn^2)$.
The function **MGS_solve_ls_modified()** is located in cw2/q2.py, and the corresponding pytests are called by running **test_MGS_solve_ls_modified()** in cw2/test2.py.

2.5 Part e

The code applying **MGS_solve_ls_modified()** to the provided matrix A_2 is also located in cw2/plotting_scripts/q2_plots.py. We then plot the errors in the numerical least squares solution using all three algorithms, which can be seen in Figure 3. Again, we have used a log y-axis scale.

Figure 3

From Figure 3 we can clearly see that **MGS_solve_ls_modified()** performs very similarly to the **householder_ls()** and is a major improvement from **MGS_solve_ls()**, with respect to error in numerical least squares solution.

2.6 Part f

The code that generates $\boldsymbol{b}$ outside the range space of A_2, and calculates the errors in numerical least squares solution using this $\boldsymbol{b}$ is located in cw2/plotting_scripts/q2_plots.py. We then plot the errors in the numerical least squares solution using this $\boldsymbol{b}$ using all three algorithms, which can be seen in Figure 4. Again, we have used a log y-axis scale.

Figure 4

From Figure 4, it appears that the improvement in **MGS_solve_ls_modified()** is evident even in this case of $\boldsymbol{b}$ outside the range space of $\boldsymbol{A}_2$, as it performs better than both **MGS_solve_ls()** and **householder_ls()** with respect to error in numerical least squares solution.

3 QUESTION 3

3.1 Part a

The formula for the $N^2 \times N^2$ matrix $\boldsymbol{D}$ is given below:

$$\boldsymbol{D}_{ij} = \begin{cases} 1+4s^2 & i=j \\ -s^2 & |j-i|=N \\ -s^2 & j-i=1, \text{and } i \not\equiv 0 \pmod{N} \\ -s^2 & i-j=1, \text{and } j \not\equiv 0 \pmod{N} \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

The bandwidth of matrix $\boldsymbol{D}$ is N.

3.2 Part b

The function **LU_inplace()** in cla_utils/exercises6.py has been updated to support the banded version of LU factorisation. Lower bandwidth can be specified by keyword argument *bl*, and upper bandwidth can be specified by keyword argument *bu*. The corresponding pytests that test this implementation are located in cw2/test3.py, and can be called by running **test_LU_inplace_banded()**.

The function **solve_L()** has been updated to support the more efficient forward substitution algorithm for banded lower triangular matrices, and this lower bandwidth can be specified by keyword argument *bl*. The corresponding pytests that test this implementation are located in cw2/test3.py, and can be called by running **test_solve_L_banded()**.

The function **solve_U()** has been updated to support the more efficient backward substitution algorithm for banded upper triangular matrices, and this upper bandwidth can be specified by keyword argument *bu*. The corresponding pytests that test this implementation are located in cw2/test3.py, and can be called by running **test_solve_U_banded()**.

Our function **solve_LU()** is located in cw2/q3.py, and can take keyword arguments *bl* (lower bandwidth), and *bu* (upper bandwidth). If we are solving a banded matrix system, we can specify the bandwidths to utilise the bandwidth support added to the functions. The corresponding pytests that test this implementation are located in cw2/test3.py, and can be called by running **test_solve_LU_banded()**.

3.3 Part c

The function **image_simulator()** is located in cw2/q3.py, and there is also a helper function called **construct_D()** that returns $\boldsymbol{D}$ as defined above, and this is called in the **image_simulator()** function. There is also a keyword argument *banded_solve*

which specifies whether to use the banded solver.
In cw2/plotting_scripts/q3_plots.py is the code that draws the plots in Figure 5.

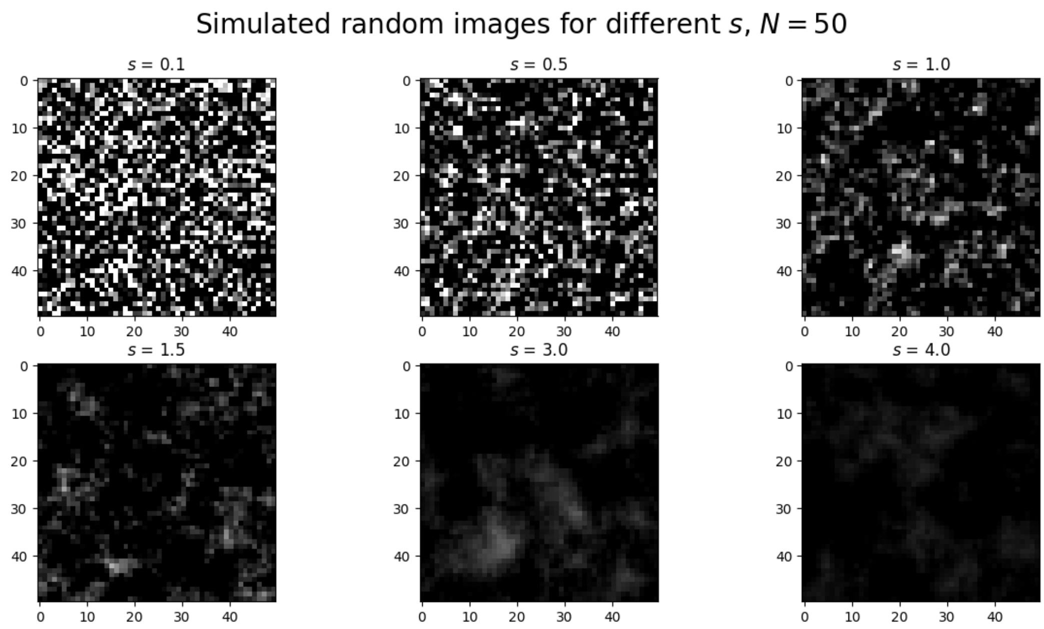

Figure 5

In Figure 5, we can see that the random simulated images become darker as s increases. s is a parameter that governs the strength of correlations between nearby points, so this behaviour is as we expect since we assume the points outside of the grid are black.

3.4 Part d

The function **image_simulator()** in cw2/q3.py has a *return_time* keyword argument that specifies whether to return the runtime of the simulation rather than the actual array. We utilise this in the code in cw2/plotting_scripts/q3_plots.py to obtain our runtimes for different N, and draw the plot in Figure 6.

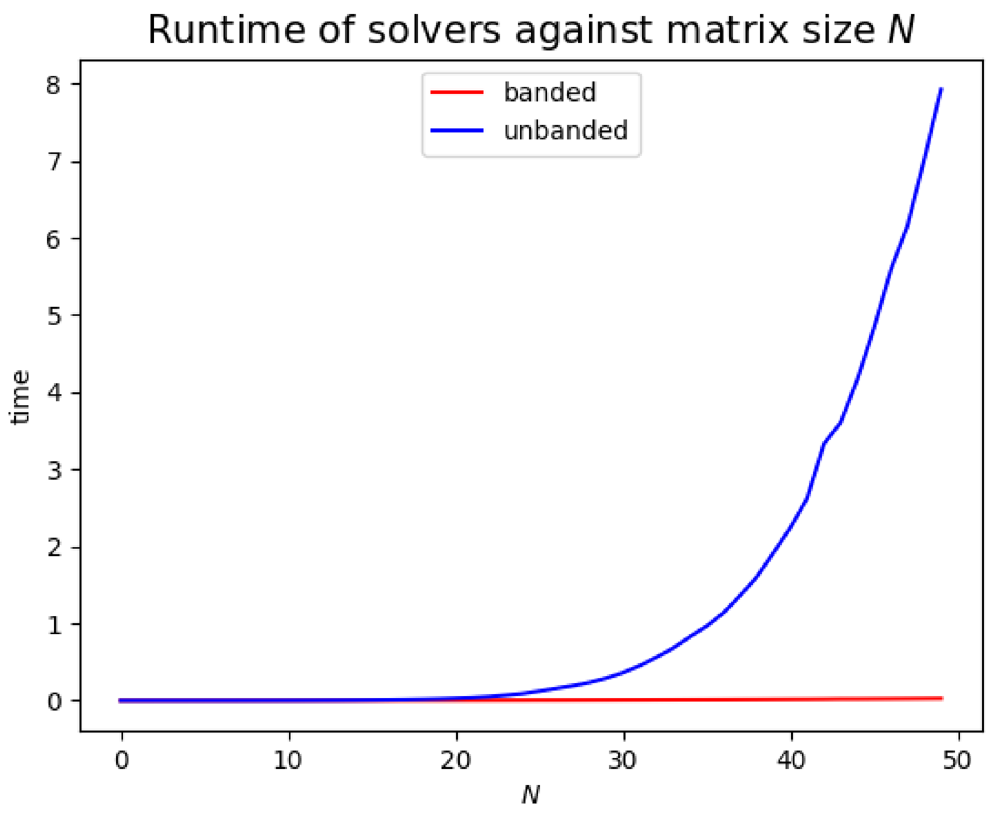

Figure 6

From Figure 6, we can see that the runtime for the unbanded solver seems to grow exponentially with N, whilst the runtime for the banded solver is very fast for all values of N. $\boldsymbol{D}$ is size N^2, and has lower and upper bandwidth size equal to N. This means that the operation count for our banded solver is $\mathcal{O}(N^4)$, whilst the operation count for our unbanded solver is $\mathcal{O}(N^6)$. This is consistent with the observed timings.

3.5 Part e

Our iterative equations are:

$$((1+\rho)\boldsymbol{I} + s^2\boldsymbol{H})\hat{\boldsymbol{u}}^{n+\frac{1}{2}} = (\rho\boldsymbol{I} - s^2\boldsymbol{V})\hat{\boldsymbol{u}}^n + \boldsymbol{w}, \tag{12}$$

$$((1+\nu)\boldsymbol{I} + s^2\boldsymbol{V})\hat{\boldsymbol{u}}^{n+1} = (\nu\boldsymbol{I} - s^2\boldsymbol{H})\hat{\boldsymbol{u}}^{n+\frac{1}{2}} + \boldsymbol{w}. \tag{13}$$

Let us assume the algorithm converges to a limit $\boldsymbol{u}^*$, i.e. $\hat{\boldsymbol{u}}^{n+1} = \hat{\boldsymbol{u}}^{n+\frac{1}{2}} = \hat{\boldsymbol{u}}^n = \boldsymbol{u}^*$. Our iterative equations now become:

$$((1+\rho)\boldsymbol{I} + s^2\boldsymbol{H})\boldsymbol{u}^* = (\rho\boldsymbol{I} - s^2\boldsymbol{V})\boldsymbol{u}^* + \boldsymbol{w}, \tag{14}$$

$$((1+\nu)\boldsymbol{I} + s^2\boldsymbol{V})\boldsymbol{u}^* = (\nu\boldsymbol{I} - s^2\boldsymbol{H})\boldsymbol{u}^* + \boldsymbol{w}. \tag{15}$$

$$(14) \implies (\boldsymbol{I} + s^2\boldsymbol{V} + s^2\boldsymbol{H})\boldsymbol{u}^* = \boldsymbol{w}. \tag{16}$$

$$(15) \implies (\boldsymbol{I} + s^2\boldsymbol{V} + s^2\boldsymbol{H})\boldsymbol{u}^* = \boldsymbol{w}. \tag{17}$$

Solving this equation:

$$\begin{aligned}
(P(\boldsymbol{I} + s^2\boldsymbol{V} + s^2\boldsymbol{H})\boldsymbol{u}^*)_{i,j} &= (P\boldsymbol{u}^*)_{i,j} + s^2(P\boldsymbol{V}\boldsymbol{u}^*)_{i,j} + s^2(P\boldsymbol{H}\boldsymbol{u}^*)_{ij}, && (18)\\
&= u^*_{i,j} + s^2(2u^*_{i,j} - u^*_{i,j-1} - u^*_{i,j+1}) + s^2(2u^*_{i,j} - u^*_{i-1,j} - u^*_{i+1,j}), && (19)\\
&= u^*_{i,j} + s^2(-u^*_{i-1,j} - u^*_{i+1,j} - u^*_{i,j-1} - u^*_{i,j+1} + 4u^*_{i,j}), && (20)\\
&= (P\boldsymbol{D}\boldsymbol{u}^*)_{ij}, && (21)\\
\implies (\boldsymbol{I} + s^2\boldsymbol{V} + s^2\boldsymbol{H}) &= \boldsymbol{D}, && (22)\\
\implies \boldsymbol{D}\boldsymbol{u}^* &= \boldsymbol{w}. && (23)
\end{aligned}$$

3.6 Part f

First, it is important to note that $\boldsymbol{H}$ is an $N^2 \times N^2$ banded matrix with 2's on its main diagonal, and -1's on the N-th super and sub diagonals.

Let us introduce the notation $\hat{\boldsymbol{x}}_{perm}$ to denote the order 2 permutation of an N^2-length vector $\hat{\boldsymbol{x}}$, such that $(P\hat{\boldsymbol{x}}) = \left(P\hat{\boldsymbol{x}}_{perm}\right)^T$, and $\left(P\hat{\boldsymbol{x}}_{perm}\right) = (P\hat{\boldsymbol{x}})^T$, where P is the mapping defined in the question.

Now note that:

$$\boldsymbol{H}\hat{\boldsymbol{x}} = \left(\boldsymbol{V}\hat{\boldsymbol{x}}_{perm}\right)_{perm} \iff (\boldsymbol{H}\hat{\boldsymbol{x}})_{perm} = \boldsymbol{V}\hat{\boldsymbol{x}}_{perm}, \tag{24}$$

$$\implies \boldsymbol{H}\hat{\boldsymbol{x}} = \boldsymbol{b} \iff \boldsymbol{V}\hat{\boldsymbol{x}}_{perm} = \boldsymbol{b}_{perm}. \tag{25}$$

It then follows:

$$((1+\rho)\boldsymbol{I} + s^2\boldsymbol{H})\hat{\boldsymbol{u}}^{n+\frac{1}{2}} = (\rho\boldsymbol{I} - s^2\boldsymbol{V})\hat{\boldsymbol{u}}^n + \boldsymbol{w}, \tag{26}$$

$$\iff \underbrace{((1+\rho)\boldsymbol{I} + s^2\boldsymbol{V})}_{\boldsymbol{A}_1}\underbrace{\hat{\boldsymbol{u}}^{n+\frac{1}{2}}_{perm}}_{\boldsymbol{x}_1} = \underbrace{\left((\rho\boldsymbol{I} - s^2\boldsymbol{V})\hat{\boldsymbol{u}}^n + \boldsymbol{w})\right)_{perm}}_{\boldsymbol{b}_1}. \tag{27}$$

$$((1+\nu)\boldsymbol{I} + s^2\boldsymbol{V})\hat{\boldsymbol{u}}^{n+1} = (\nu\boldsymbol{I} - s^2\boldsymbol{H})\hat{\boldsymbol{u}}^{n+\frac{1}{2}} + \boldsymbol{w}, \tag{28}$$

$$\iff \underbrace{((1+\nu)\boldsymbol{I} + s^2\boldsymbol{V})}_{\boldsymbol{A}_2}\underbrace{\hat{\boldsymbol{u}}^{n+1}}_{\boldsymbol{x}_2} = \underbrace{\left((\nu\boldsymbol{I} - s^2\boldsymbol{V})\hat{\boldsymbol{u}}^{n+\frac{1}{2}}_{perm}\right)_{perm} + \boldsymbol{w}}_{\boldsymbol{b}_2}. \tag{29}$$

Our systems (27) and (29) both involve banded matrices $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ with bandwidths 1, which can be used to our advantage in terms of efficiency.
Also, $\boldsymbol{A}_1$ is a block diagonal matrix with N repeated identical blocks of size N that also must have bandwidth 1, which we will denote $\boldsymbol{A}_1'$.

We can view our system involving $\boldsymbol{A}_1$ and $\boldsymbol{b}_1$ as equivalent to solving $\boldsymbol{A}_1'\boldsymbol{x}_1^{(i)} = \boldsymbol{b}_1^{(i)}$, for $i = 1, \dots, N$, where $\boldsymbol{b}_1 = \begin{bmatrix} \boldsymbol{b}_1^{(1)} \\ \vdots \\ \boldsymbol{b}_1^{(N)} \end{bmatrix}$, $\boldsymbol{x}_1 = \begin{bmatrix} \boldsymbol{x}_1^{(1)} \\ \vdots \\ \boldsymbol{x}_1^{(N)} \end{bmatrix}$.

An identical argument applies to the system involving $\boldsymbol{A}_2$, $\boldsymbol{b}_2$, and $\boldsymbol{x}_2$.

Our algorithm for each iteration is as follows:

1. Evaluate the RHS of equation (27), $\boldsymbol{b}_1$, and use **numpy.reshape()** to cast this to the form $\boldsymbol{B}_1 = \begin{bmatrix} \boldsymbol{b}_1^{(1)} & \dots & \boldsymbol{b}_1^{(N)} \end{bmatrix}$.
2. Generate $\boldsymbol{A}_1'$, the size N diagonal block of $\boldsymbol{A}_1$.
3. Use our **solve_LU()** function from cw2/q3.py to solve $\boldsymbol{A}_1'\boldsymbol{x}_1^{(i)} = \boldsymbol{b}_1^{(i)}$, for $i = 1, \dots, N$ simultaneously, with keyword arguments $bl = 1$ and $bu = 1$. This returns $\boldsymbol{x}_1$ cast to a matrix.
4. Evaluate the RHS of equation (29), $\boldsymbol{b}_2$, and again use **solve_LU()** function from cw2/q3.py to solve $\boldsymbol{A}_2'\boldsymbol{x}_2^{(i)} = \boldsymbol{b}_2^{(i)}$, for $i = 1, \dots, N$ simultaneously, with keyword arguments $bl = 1$ and $bu = 1$. This returns $\boldsymbol{x}_2$ cast to a matrix.
5. Use **numpy.flatten()** to flatten this matrix column-wise to obtain $\boldsymbol{x}_2$.

In steps 1 and 4, when performing the multiplication of the bandwidth 1 matrix multiplied by another matrix, we can take advantage of the bandedness so that the operation count is $\mathcal{O}(N^2)$ rather than the standard $\mathcal{O}(N^3)$. The operation counts for solving the two band-1, size N systems are both $\mathcal{O}(N^2)$, so the whole algorithm is $\mathcal{O}(N^2)$.
This is less than the operation count for the direct solving of the band-N system, which is $\mathcal{O}(N^4)$.
It requires less memory, as utilising the block diagonal properties of $\boldsymbol{A}_1$ and $\boldsymbol{A}_2$ means that we only have to store $N \times N$ matrices rather than $N^2 \times N^2$ matrices.

3.7 Part g

All the functions mentioned are located in cw2/q3.py. The function **construct_V_block()** is used to help construct one of the blocks of $\boldsymbol{V}$.
The function **mat_mul_banded()** is an implementation of the optimised matrix multiplication involving a banded matrix as described above. The corresponding pytests that make sure this is working properly are located in cw2/test3.py, and are called by running **test_mat_mul_banded()**.
There is also a **run_iter()** function that performs the algorithm described in 3.6, and this is called by the **iterative_solve()** function iteratively until our stopping condition $||\boldsymbol{D}\hat{\boldsymbol{u}}^n - \boldsymbol{w}|| < \epsilon||\boldsymbol{w}||$ is met. The corresponding pytests are called by running **test_run_iter()** and **test_iterative_solve()**, which are also located in cw2/test3.py.

3.8 Part h

All the code for this question is located in cw2/plotting_scripts/q3_plots.py In order to explore the effects of ρ and ν on the number of iterations we calculate the number of iterations for combinations of ρ and ν between 0.2 and 2 at 0.1 intervals. We sort these values into a matrix, and produce a heap mat for various N values. The heat map should be darker for lower iteration counts, and lighter for higher iteration counts.

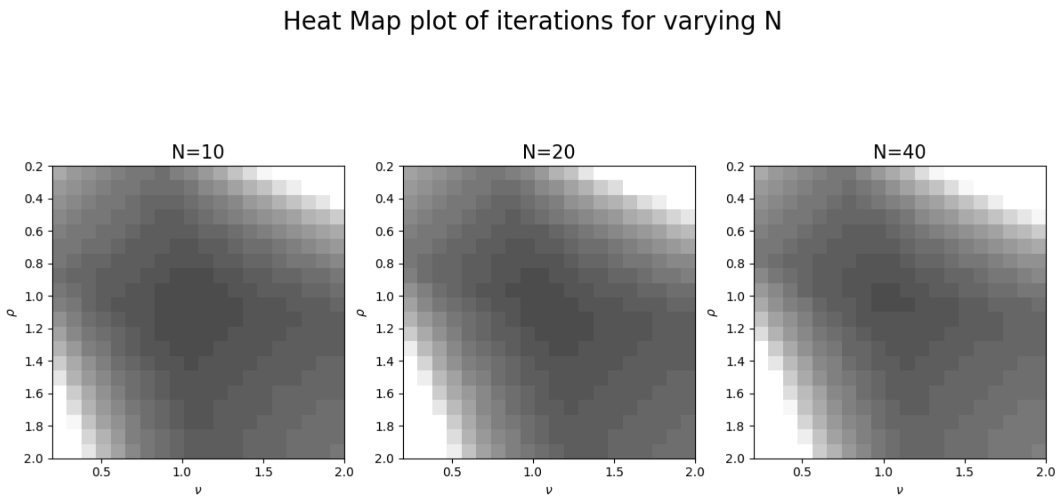

Figure 7

In Figure 7 we see that for all N, the optimal values regions for ρ and ν both seem to be centered around 1. This optimal value region appears to converge to $\rho = 1$, $\nu = 1$ as N increases, so it suggests we should take these our optimal values.

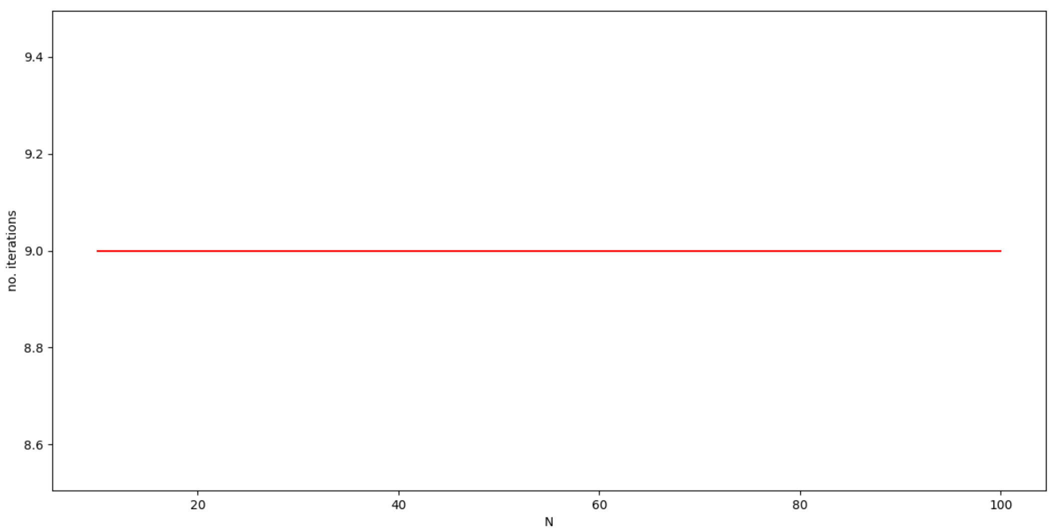

Figure 8

In Figure 8 we see our typical iteration count is 9 using our optimal values and $\epsilon = 1.0e - 6$, and this does not appear to depend on N. This suggests our operation count is still just $\mathcal{O}(N^2)$.

3.9 Part i

With a parallel computer that can execute multiple operations at once (multi-core), we can execute the solving of the set of systems $\boldsymbol{A}_1'\boldsymbol{x}_1^{(i)} = \boldsymbol{b}_1^{(i)}$, for $i = 1, \ldots, N$ simultaneously rather than one after the other as our current implementation does, and similarly for the second set of systems $\boldsymbol{A}_2'\boldsymbol{x}_2^{(i)} = \boldsymbol{b}_2^{(i)}$, for $i = 1, \ldots, N$. As long as the computer performance is unaffected by the simultaneous operations, this will accelerate the runtime of algorithm.