CW2

MATH70024

01495449

Computational Linear Algebra

Imperial College London

December 9, 2022

1 Answer to Question 1a

The resulting matrix after applying LU decomposition to A in place is

$$A_{inplace} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 2 \\ -1 & -1 & 1 & 0 & 0 & 4 \\ -1 & -1 & -1 & 1 & 0 & 8 \\ -1 & -1 & -1 & -1 & 1 & 16 \\ -1 & -1 & -1 & -1 & -1 & 32 \end{bmatrix}$$

As you can see this matrix has 1's across the diagonal and -1's below the diagonal. The last column is powers of 2, starting at 2^0 and ending at 2^5. All the above the diagonal are zeroes except in the last column.

Let U be the upper diagonal matrix of $A_{inplace}$, then the growth factor is calculate as

$$\rho = \frac{max_{ij}|U_{ij}|}{max_{ij}|A_{ij}|} = \frac{32}{1} = 32$$

2 Answer to Question 1b

The general form of LU decomposition for $A_{ij}^{(n)}$ is

$$A_{ij}^{(n)} = LU = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ -1 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -1 & -1 & -1 & \cdots & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & \cdots & 2^0 \\ 0 & 1 & 0 & \cdots & 2^1 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 2^n \end{bmatrix}$$

From the lecture notes,

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & \cdots & \cdots & 0 \\ l_{21} & 1 & 0 & \cdots & 0 & \cdots & \cdots & \vdots & 0 \\ l_{31} & l_{32} & 1 & \cdots & 0 & \cdots & \cdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 & \cdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & l_{k+1,k} & 1 & \cdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & l_{k+2,k} & l_{k+2,k+1} & \ddots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & l_{n-1,k+1} & \cdots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & l_{n,k} & l_{n,k+1} & \cdots & \cdots & 1 \end{bmatrix}$$

We also given that

$$\rho = \frac{U_{ij}}{A_{jj}} = \frac{-1}{1} = -1$$

for $i > j$ and otherwise $l_{ij} = 0$

To prove that U is of the form mentioned above, we know that in LU decomposition

$$U = L_n L_{n-1} ... L_2 L_1 A^{(n)}$$

Where

$$L_k = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & \cdots & \cdots & \vdots & 0 \\ 0 & 0 & 1 & \cdots & 0 & \cdots & \cdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 & 0 & \cdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & -l_{k+1,k} & 1 & \cdots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & -l_{k+2,k} & 0 & \ddots & \vdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots & 0 & \cdots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -l_{n,k} & 0 & \cdots & \cdots & 1 \end{bmatrix}$$

Hence L_k will have ones on the diagonal and ones below the diagonal in the kth column with zeroes everywhere else. This gives

$$L_1 A^{(n)} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 1 & 0 & \cdots & \vdots & 2 \\ \vdots & -1 & \ddots & \ddots & \vdots & 2 \\ \vdots & \vdots & \ddots & \ddots & 0 & 2 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & -1 & \cdots & \cdots & -1 & 2 \end{bmatrix}$$

As you can see applying L_1 to A, removes the minus one entries from below the diagonal on the 1st column and adds 2 to every entry in the last column past the 1st entry. Hence applying $L_k ... L_1$ to the left of A in this order will send all the entries below the diagonal to zeroes, column by column with L_i multiplication sending the values below diagonal in *ith* column to 0. Also each multiplication with L_i will add 2 to all the entries in the last column where the row number is greater than i. Hence the overall effect is that we end up with an upper triangular matrix with 1s on the diagonal and increasing powers of 2 in the last column (from 2^0 to 2^{n-1}).

This gives the growth factor of $A^{(n)}$ to be:

$$\rho = \frac{U_{ij}}{A_{jj}} = \frac{2^{n-1}}{1} = 2^{n-1}$$

3 Answer to Question 1c

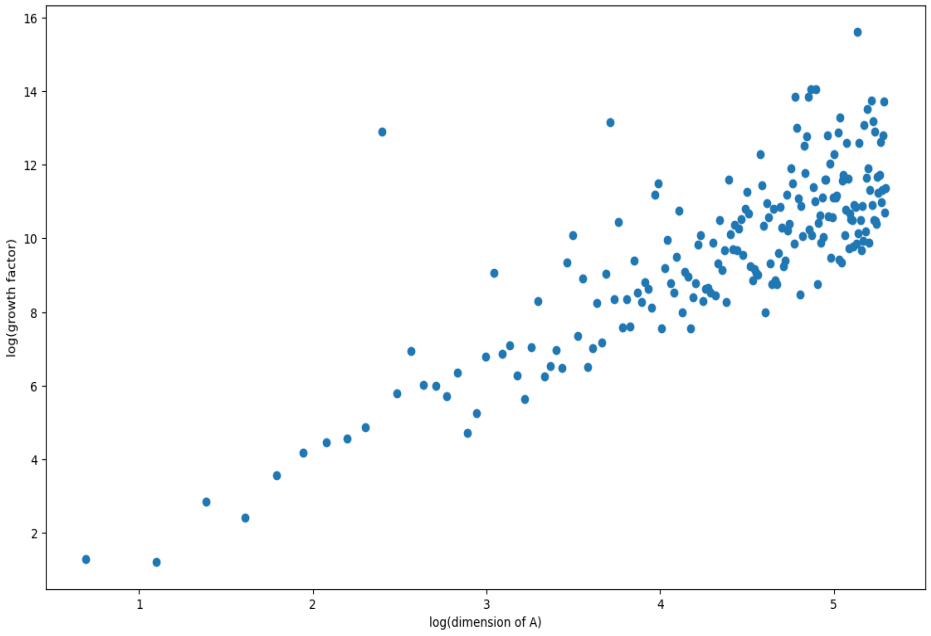
The code outputs:
Growth factor is 5.764607523034235e+17
Backward stability of cla_utils/LUP_inplace for $A^{(60)}$ is 0.05635852861825637
Backward stability of cla_utils/solve_LUP for $A^{(60)}$ is 0.25338741118082025

This shows that the order of error for LUP_inplace and solve_LUP is far greater than the order of machine precision of 10^{-16}, which shows that LUP_inplace is not backward

stable.

4 Answer to Question 1d

Below is a graph that looks to investigate how the growth factor changes as the dimension of the randomly generated matrices changes. I generate matrices with individually independently distributed random variables drawn from the uniform distribution on the interval $[-\frac{1}{n}, \frac{1}{n}]$. In the graph below I calculate the log(growth factor) as dimensions increase from 2 to 200 with steps of 1.



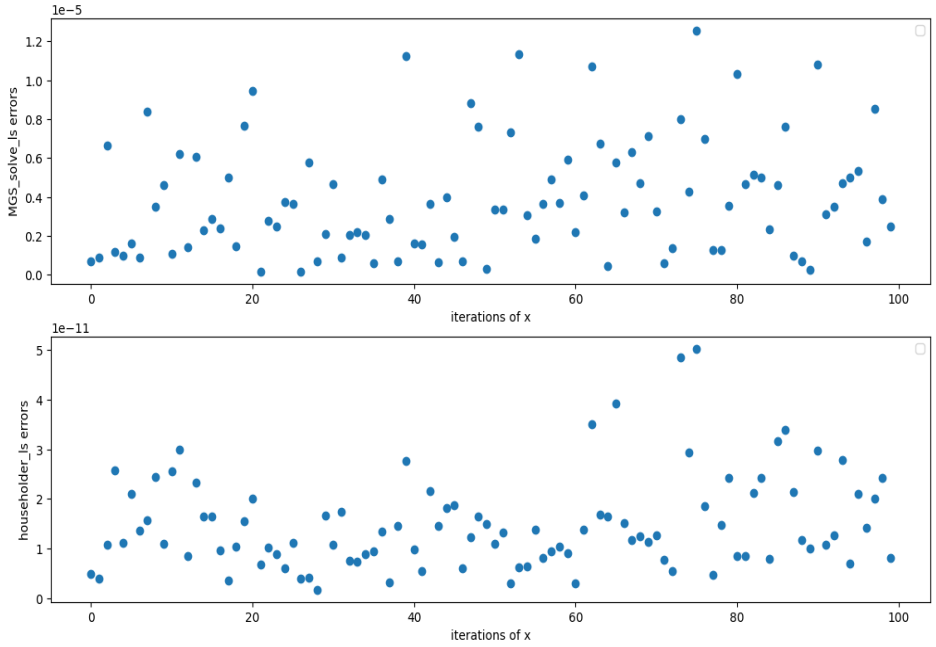
The graph choose a underlying linear relationship (with noise) between the log of growth factor and log of dimension of the matrix after skipping past the first few smaller dimension matrices. This relationship suggests that:

growth factor = $e^a dimension^b$

Where a and b are constants.

The linear relationship isn't linear for matrices with smaller dimensions, this is because for smaller dimensions the LU_inplace doesn't explode.

5 Answer to Question 2b



From the graphs above we can see that the errors for MGS_solve_ls are of the order of magnitude 1e-5, where as errors for householder_ls are of the order of magnitude 1e-11. This shows that the householder_ls algorithm is more accurate, which is makes sense as its backward stable.

6 Answer to Question 2c

We are given that:

$$A_+ = \begin{bmatrix} \hat{A} & | & b \end{bmatrix}$$

$$A_+ = \begin{bmatrix} \hat{Q} & | & q_{n+1} \end{bmatrix} \begin{bmatrix} \hat{R} & z \\ 0 & \rho \end{bmatrix}$$

This implies:

$$\begin{bmatrix} \hat{A} & | & b \end{bmatrix} = \begin{bmatrix} \hat{Q} & | & q_{n+1} \end{bmatrix} \begin{bmatrix} \hat{R} & z \\ 0 & \rho \end{bmatrix}$$

Expanding the above gives:

$$A = \hat{Q}\hat{R}$$

$$b = \hat{Q}z + q_{n+1}\rho$$

We know $\hat{Q}^*\hat{Q} = I$ as Q is a orthonormal matrix. Hence,

$$z = \hat{Q}^* b - \hat{Q}^* q_{n+1}\rho$$

Since q_{n+1} is produced by the reduced QR factorisation of A_+ and is added as an additional column to $\hat{Q}$. We know q_{n+1} is orthogonal to all the columns of $\hat{Q}$ and hence the rows of $\hat{Q}^*$. Giving,

$$\hat{Q}^* q_{n+1} = 0$$

So,

$$z = \hat{Q}^* b$$

$$b = \hat{Q}z + q_{n+1}\rho$$

7 Answer to Question 2d

MGS_solve_ls_modified is a combination of modified GS and of the function solve_U. Therefore we need to workout the N_{FLOPS} for both these algorithm and sum them up to get the N_{FLOPS} for MGS_solve_ls_modified. Lets first work out N_{FLOPS} for modified Gram-Schmidt.

We concentrate on the operations happening inside the inner j loop of the psuedocode of the Gram-Schmidt Algorithm. Inside the loop there are two operations.

1) $r_{ij} = q_i^* v_i$. This is the inner product of two vectors in $|R^m$, which requires m multiplications and m−1 additions, so we count 2m−1 FLOPS per inner iteration.

2)$v_j = v_j - r_{ij}q_i$. This requires m multiplications and m subtractions, so we count 2m FLOPS per inner iteration.

The input for the modified GS algorithm is a augmented matrix A with m rows and n+1 columns hence at each iteration we require a combined operation count of 4m FLOPS. There are n+1 outer iterations over i, and n−i inner iterations over j, which we can estimate by approximating the sum as an integral,

$$N_{FLOPS} \approx \sum_{i=1}^{n+1}\sum_{j=1}^{n+1} 4m \approx 4m \int_0^{n+1}\int_0^{n+1} dx'dx = 4m\frac{(n+1)^2}{2} = 2m(n+1)^2$$

For solve_U, the procedure is called backward substitution which we summarise in the following code:
1)$x_m = \frac{y_m}{R_{mm}}$ 2)FOR $i = m-1$ TO 1 (BACKWARDS): $x_i = \frac{(y_i - \sum_{k=i+1}^{m} R_{ik}x_k)}{R_{ii}}$

From 1) we are just dividing one number by another, hence the operation count for this is 1. Now for 2), i will split the operation count into:

a) multiplication within the sum for all iterations b) addition within the sum for all iterations c) subtraction within the brackets for all iterations d)division

For a) there are m-i multiplications and to get the operations for all iterations, we do

$$\sum_{1}^{m-1} m - i = m(m-1) - \frac{1}{2}m(m-1) = \frac{m(m-1)}{2}$$

For b) there are m-i-1 multiplications and to get the operations for all iterations, we do

$$\sum_{1}^{m-1} m - i - 1 = m(m-1) - \frac{1}{2}m(m-1) - (m-1) = \frac{(m-1)(m-2)}{2}$$

For c), we know that only one operation count per iteration as subtraction only happens once each iteration. Hence the operation count is m-1 for c).

For d, the operation count is same as for c) as division happens also once every iteration.

Hence the total N_{FLOPS} for solve_U is $\frac{m(m-1)}{2} + \frac{(m-1)(m-2)}{2} + 2 * (m-1) = (m-1)^2$

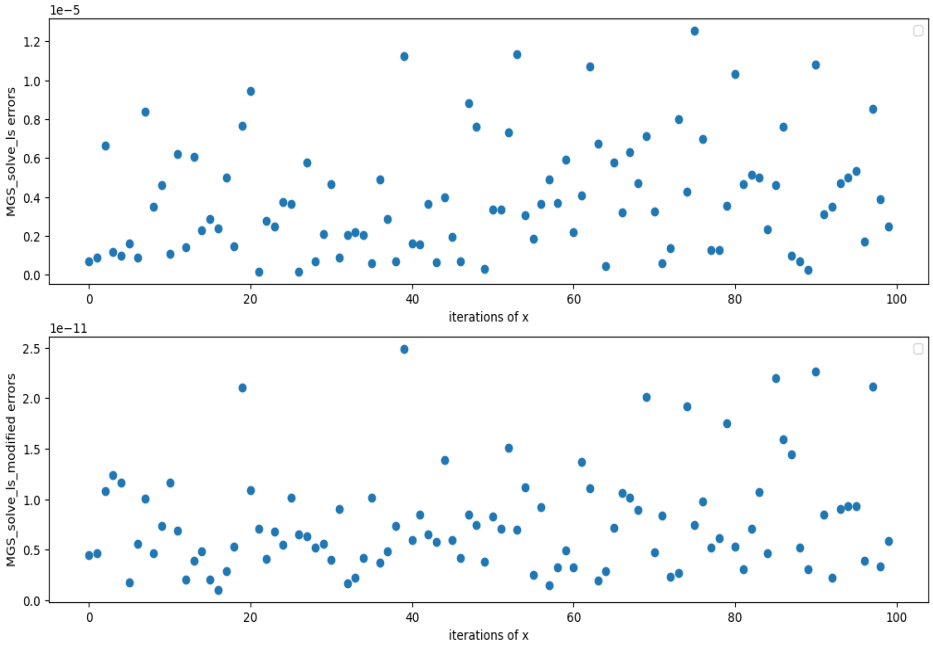
Therefore the N_{FLOPS} of MGS_solve_ls_modified is

$$2m(n+1)^2 + (m-1)^2$$

Where the operation count is $O(mn^2)$

8 Answer to Question 2e

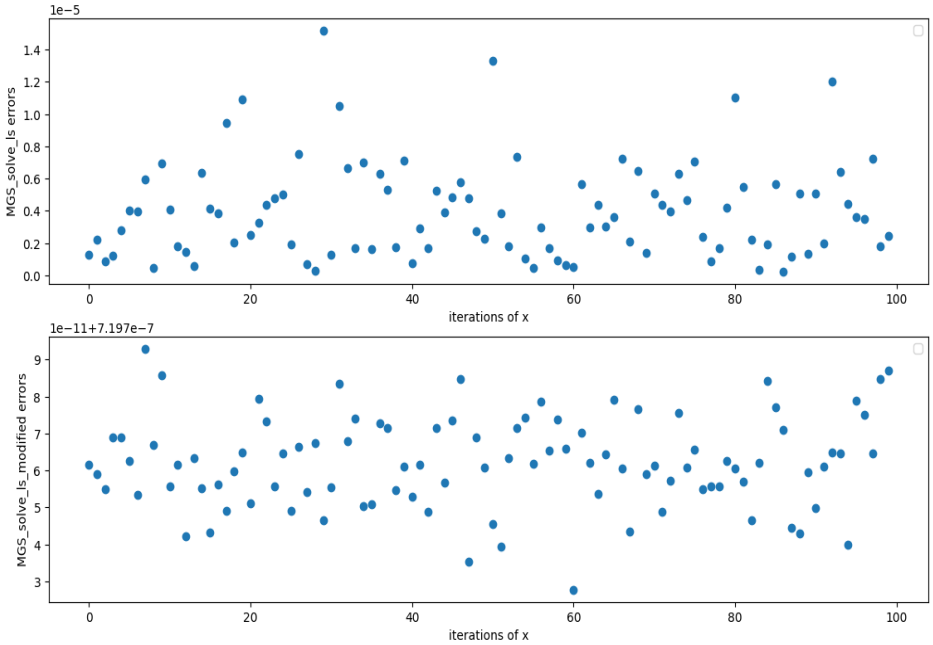
The following graphs are generated when b is inside the range of A.



From the graphs above we can see that the errors for MGS_solve_ls are of the order of magnitude 1e-5, where as errors for MGS_solve_ls_modified are of the order of magnitude 1e-11. This shows that the MGS_solve_ls_modified algorithm is more precise.

9 Answer to Question 2f

The following graphs are generated when b is outside the range of A.



Comparing the above graphs with the graphs in answer to question 2e, we can see that MGS_solve_ls performs similarly with b being inside or outside the range of A. The magnitude of the errors for this remain at 1e-5. However the performance of MGS_solve_ls_modified has decreased for b outside the range of A from 1e-11 for b inside the range of A to 1e-7 for b outside the range of A. The important takeaway is that MGS_solve_ls_modified is still better than MGS_solve_ls.

10 Answer to Question 3a

We are given that $(D\hat{u})_{ij} = u_{ij} = u_{(i-\hat{1})N+j}$.
Using the above we can write: $(PD\hat{u})_{ij} = (D\hat{u})_{(i-1)N+j}$.

We are given,

$$(PD\hat{u})_{ij} = u_{i,j} + s^2(-u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} + 4u_{i,j})$$

After substitution: We are given,

$$(PD\hat{u})_{ij} = u_{(i-\hat{1})N+j} + s^2(-\hat{u}_{(i-2)N+j} - \hat{u}_{iN+j} - \hat{u}_{(i-1)N+j-1} - \hat{u}_{(i-1)N+j+1} + 4\hat{u}_{(i-1)N+j})$$

we can write

$$(D\hat{u}_{(i-1)N+j}) = \sum_{k=1}^{N^2} D_{(i-2)N+j,k}\hat{u_k}$$

.

This gives a formula for D where,

$$D_{(i-1)N+j,(i-1)N+j} = 1 + 4s^2$$

$$D_{(i-1)N+j,(i-2)N+j} = -s^2$$

$$D_{(i-1)N+j,iN+j} = -s^2$$

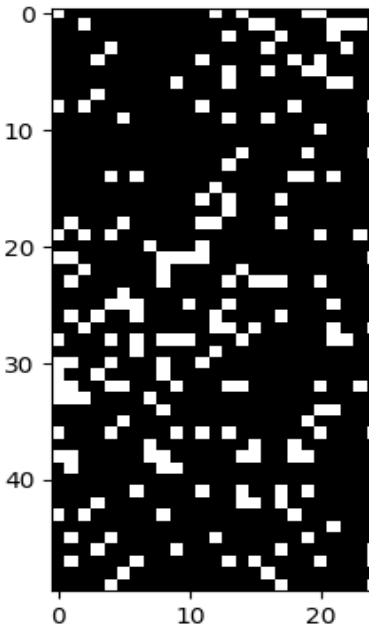
$$D_{(i-1)N+j,(i-1)N+j-1} = -s^2$$

$$D_{(i-1)N+j,(i-1)N+j+1} = -s^2$$

This is a banded matrix with bandwidth N.

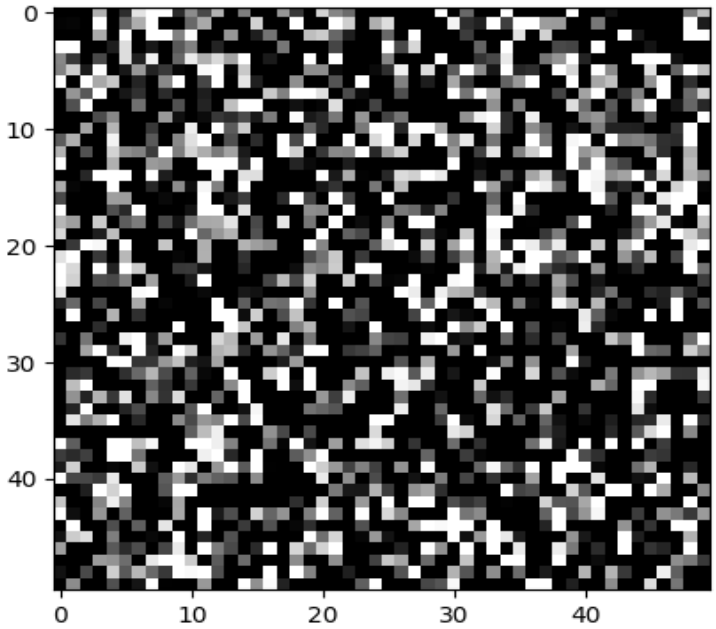
11 Answer to Question 3c

I did a sense check to compare the performance of solve_banded_LU to solve LU unbanded matrix. The performance f the unbanded matrix version is extremely slow whereas the banded version is fast as we would expect.

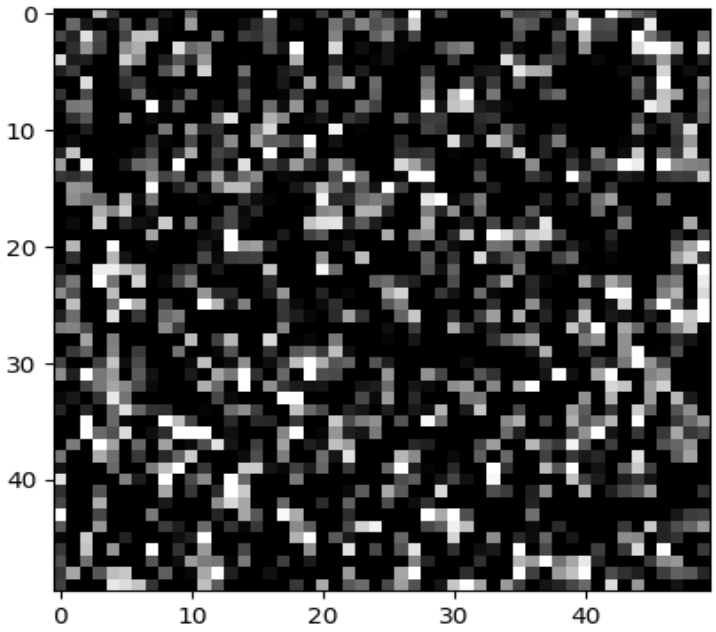


Below is graph for N=50 in the banded matrix case, where s=0

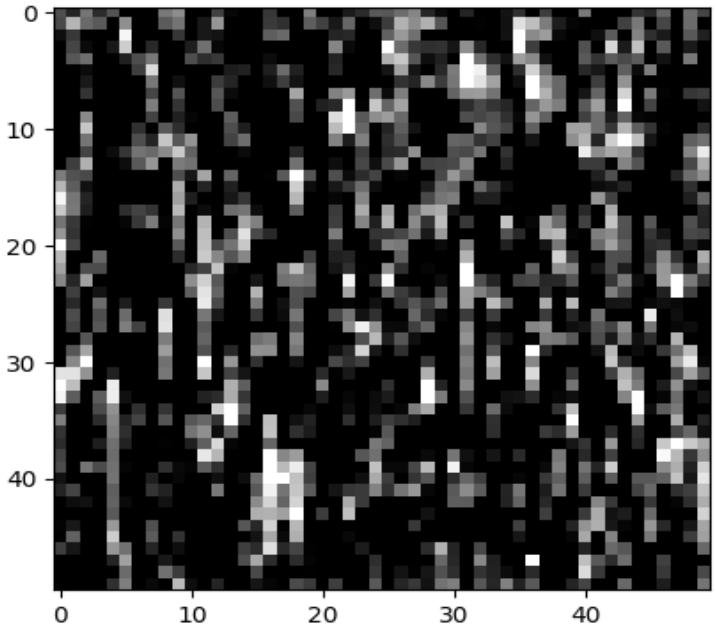
Below is graph for N=50 in the banded matrix case, where s=0.25



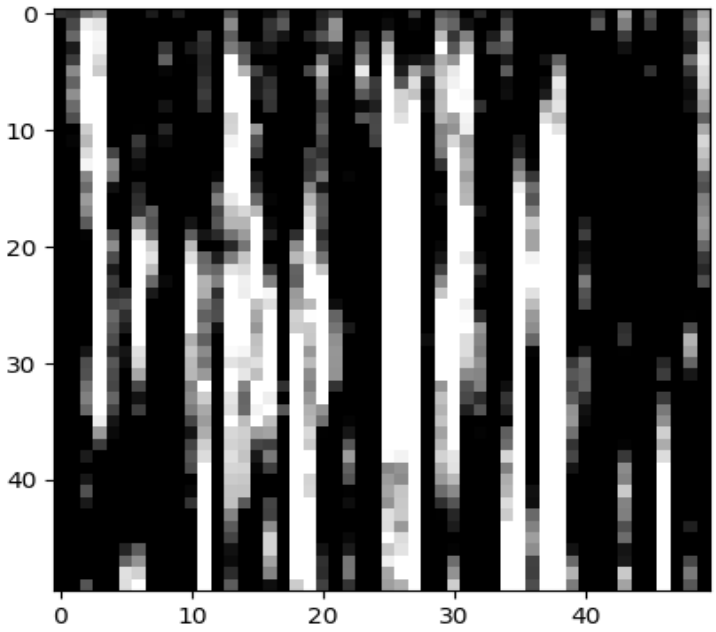
Below is graph for N=50 in the banded matrix case, where s=0.5



Below is graph for N=50 in the banded matrix case, where s=0.75



Below is graph for N=50 in the banded matrix case, where s=1



12 Answer to Question 3d

I used the time.time() function to find out how long the banded and the non banded case take to compute in seconds. I fixed s to be $\frac{1}{2}$. The results are displayed below:

N	Time for banded	Time for unbanded
10	0.04	0.04
20	0.521	2.635
30	0.377	24.838
40	1.0	94.963
50	2.134	409.744

From the table above you can see that both algorithms take the same time for N=10. However as N increases, the time for the banded algorithm increases slowly whereas the rate of increase in time for the unbanded algorithm increases rather drastically for each increase of 10 in N. This is inline with what I expected. The operation count for the banded matrix algorithm is O(mpq) as mentioned in notes, which is linear in m instead of the cubic in the non banded matrix algorithm. Even the *solve_L* and *solve_U* operation counts have reduced to O(mp) and O(mq) from O(m^2) and O(m^2) respectively.

13 Answer to Question 3e

When the algorithm converges to a limit u^*, at this point $\hat{u}^{\frac{n+1}{2}} = \hat{u}^{n+1} = \hat{u}^n$. Hence we can rearrange the iterative algorithm equation given in the question to

$$w = (I + s^2(V + H))u^*$$

We also know that:

$$\begin{aligned} P(I + s^2(V + H))u^* &= u^*_{i,j} + s^2 P(V + H)u^* \\ &= u^*_{i,j} + s^2(4u^*_{i,j} - u^*_{i-1,j} - u^*_{i+1,j} - u^*_{i,j-1} - u^*_{i,j+1}) \\ &= (PDu^*)_{i,j} \end{aligned} \tag{1}$$

Hence clearly

$$\begin{aligned} Du^* &= (I + s^2(V + H))u^* \\ &= w \end{aligned} \tag{2}$$

14 Answer to Question 3f

Parallelism is a good idea when the task can be divided into sub-tasks that can be executed independent of each other without communication or shared resources. This algorithm can be divided as such which means running it using parallel computing will be fast than sequential computing as we won't need to wait for one task to finish before the next task can start.