1. Direct Solution Methods: Costs of Basic Storage: 10%

Question 1.1: If we vary the sparsity of sparse matrices from say 1% - 100% for a fixed matrix size do we see a motivation to use CSR storage?

Expectation 1.1: When the matrix size is fixed, CSR storage increases as sparsity increases but can still provide greater memory efficiency than full storage, especially when dealing with highly sparse matrices. While the magnitude of memory savings may decrease, CSR storage is still strongly motivated when considering overall efficiency and space utilisation.

Experiment 1.1: The matrix size N was fixed at 1000. Five experimental points were set on sparsity 1-100% at 1%, 25%, 50%, 75% and 100% to cover different scenarios from very sparse to relatively dense. With all other parameters at default values, for each sparsity set point, the memory usage required using regular storage and CSR storage was recorded in Table 1.1.

Plot graph:

Table 1.1: Memory Usage between Full and CSR Storage Methods under Different Sparsity

Sparsity (%)	Full Storage Memory (MB)	CSR Storage Memory (MB)
1	7.63	0.103
25	7.63	2.67
50	7.63	4.99
75	7.63	6.96
100	7.63	8.57

Figure 1.1 represents the relationship between CSR storage and full storage memory usage. The red line represents CSR storage memory usage, and the green line represents full-storage memory usage. The blue dashed line represents the difference in memory usage between the two methods at the same sparsity, to illustrate the degree of optimisation of CSR. The orange dashed line represents the sparsity at which CSR reaches full-storage memory usage.

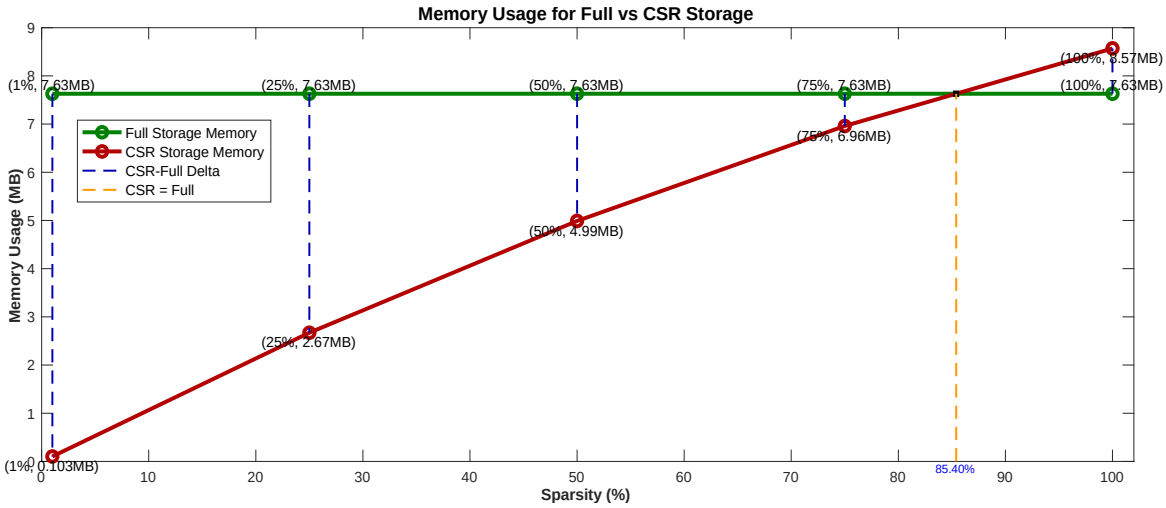

Figure 1.1: Comparative Memory Usage of Full vs CSR Storage under Different Sparsity (N=1000)

Observation and Conclusion:

Figure 1.1 demonstrates that the full-store memory usage is unaffected by matrix sparsity and remains constant because it allocates space for every element in the matrix. In contrast, CSR storage has very low memory usage at low sparsity (1%), which increases linearly with

increasing sparsity, but always stays below full storage, showing high storage efficiency. The blue dashed line clearly shows the amount of memory saved by CSR storage compared to full storage at each sparsity level. When sparsity exceeds 85%, the memory advantage of CSR storage is diminished by the increase in non-zero elements, and the orange dashed line shows the threshold of its memory-saving efficiency. Above this point, CSR is no longer more memorable than the full-store approach.

The expectation of the experiment did not anticipate the phenomenon that occurs after the threshold but was confirmed before the threshold. Given these findings, for matrices where sparsity does not exceed this threshold, CSR storage is recommended to maximize memory efficiency. For matrices exceeding this threshold, further evaluation is recommended to refine the precise threshold, especially for matrices of different sizes or with varying sparsity patterns.

Question 1.2: If for example, we considered three unstructured sparsity cases, 1%, 10% and 100% and then varied the matrix size, what can we deduce about the scalability of normal and CSR storage – i.e. when does it appear that we are likely to hit memory limits on an average 16 GB laptop?

Expectation 1.2: When dealing with matrices with different sparsities (1%, 10%, 100%), as the matrix size increases, it is expected that on a 16 GB standard laptop, using the CSR storage method compared to the normal method can handle larger matrices without reaching the memory limit. Especially at lower sparsities, normal storage may quickly run out of memory as the matrix size increases, whereas CSR storage can support matrix operations with larger sizes until approaching the memory limit of 16 GB.

Experiment 1.2: The memory requirements for five matrices of different sizes (1000, 2000, 3000, 4000, and 5000) at three sparsity levels of 1%, 10%, and 100% using normal and CSR storage methods were recorded in Table 1.2.

Table 1.2: Memory Usage between Full and CSR Storage Methods under Different Sparsity and Matrix Sizes

Matrix Size	Sparsity = 1 %		Sparsity = 10 %		Sparsity = 100 %	
	Normal (MB)	CSR (MB)	Normal (MB)	CSR (MB)	Normal (MB)	CSR (MB)
1000	7.63	0.103	7.63	1.1	7.63	8.57
2000	30.5	0.444	30.5	4.45	30.5	34.3
3000	68.7	1.01	68.7	10	68.7	77.2
4000	122	1.81	122	17.8	122	137
5000	191	2.84	191	27.9	191	215

Plot graph 1.2:

The data in Table 1.2 is plotted in Figure 1.2, where lines of different colours and shapes represent the amount of memory for the two methods, CSR and normal, at different sparsities and with different matrix sizes.

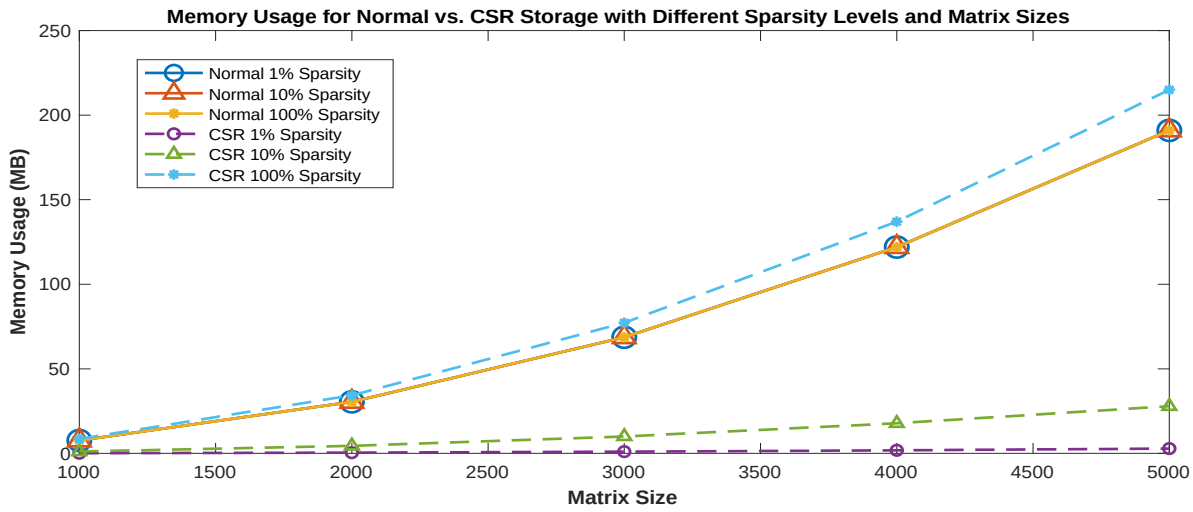

Figure 1.2: Memory Usage of Normal vs CSR Storage under Different Sparsity and Matrix Sizes

Observation and Conclusion 1.2:

From the results in Figure 1.2 and Table 1.2, it can be observed that the memory usage of normal storage increases quadratically with matrix size at all sparsity levels, as evidenced by the steep curve. The memory usage of CSR storage increases much slower, especially at lower sparsity levels, which confirms the expected memory efficiency of CSR in the case of sparse matrices. In addition, the limit of what a 16 GB laptop can achieve with different storage methods at different sparsities can be predicted by the followings:

$$N_{Normal(1\%,10\%,100\%)} = \sqrt{\frac{16\times1024\ \text{MB}}{191}} \times 5000 = 46309 \qquad \textbf{Eqn. 1}$$

$$N_{CSR(1\%)} = \sqrt{\frac{16\times1024\ \text{MB}}{2.84}} \times 5000 = 379770 \qquad \textbf{Eqn. 2}$$

$$N_{CSR(10\%)} = \sqrt{\frac{16\times1024\ \text{MB}}{27.9}} \times 5000 = 121165 \qquad \textbf{Eqn. 3}$$

$$N_{CSR(100\%)} = \sqrt{\frac{16\times1024\ \text{MB}}{215}} \times 5000 = 43648 \qquad \textbf{Eqn. 4}$$

The computed matrix size limit at 16 GB memory capacity further illustrates the excellent scalability of CSR storage, thus validating the initial expectations. In particular, when sparsity is low, CSR can handle much larger matrix sizes than regular storage before reaching the same memory limit. This scalability will be a significant advantage when dealing with large sparse matrices to avoid memory overflow and perform larger computations.

Further research could refine these conclusions by examining the sparsity and matrix size over successive ranges, thus ensuring a comprehensive understanding of memory usage patterns. However, current evidence suggests that there is a clear trend toward choosing storage methods for matrix computations with a limited range of memory.

2. Direct Solution Methods: Costs of "Working" Storage and Run Time Scalability: 30%

Question 2.1: In the above, is there any change in performance between a sparse matrix of size N and a dense matrix (100% sparsity) of the same size?

Expectation 2.1: For a particular decomposition method, it may be designed to be robust enough that its running time and peak memory requirements do not change significantly with

the sparsity of the matrix. Whether the matrix is dense or sparse, the decomposition operation may require a similar number of computational steps and temporary storage space. However, different decomposition methods may utilise sparsity with varying efficiency. Therefore, differences in runtime and peak memory usage can be expected between different decomposition methods, provided that the sparsity is the same.

Experiment 2.1: The matrix size N was fixed to 1000 and three decomposition algorithms were chosen: the QR decomposition, the SVD decomposition, and the DenseLU decomposition. By varying the sparsity of the matrix from 1% to 100%, the running time and peak memory usage of each decomposition method at different sparsity levels were recorded in Table 2.1.

Table 2.1: Performance Metrics of Matrix Decomposition Methods at Varying Sparsity Levels

Sparsity (%)	QR Decomposition		SVD Decomposition		DenseLU Decomposition	
	Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
1	0.61	0.252	3.288	38.2	0.183	0.00381
25	0.623	0.252	3.087	38.2	0.182	0.00381
50	0.589	0.252	3.159	38.2	0.183	0.00381
75	0.596	0.252	3.197	38.2	0.191	0.00381
100	0.59	0.252	3.224	38.2	0.184	0.00381

Plot graph 2.1:

The data in Table 2.1 for running time and peak memory for different decomposition methods at different sparsities are plotted in Figures 2.1 and 2.2, respectively.

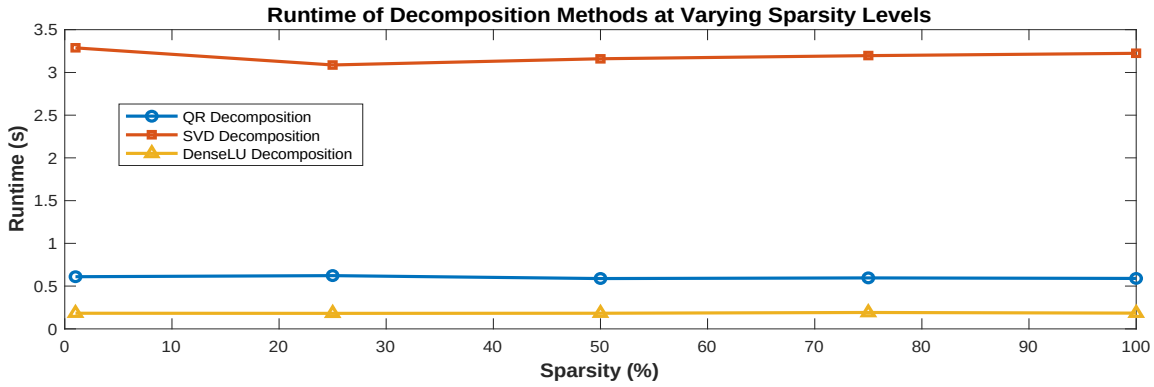

Figure 2.1: Comparison of Running Time of Matrix Decomposition Methods at Different Sparsity Levels

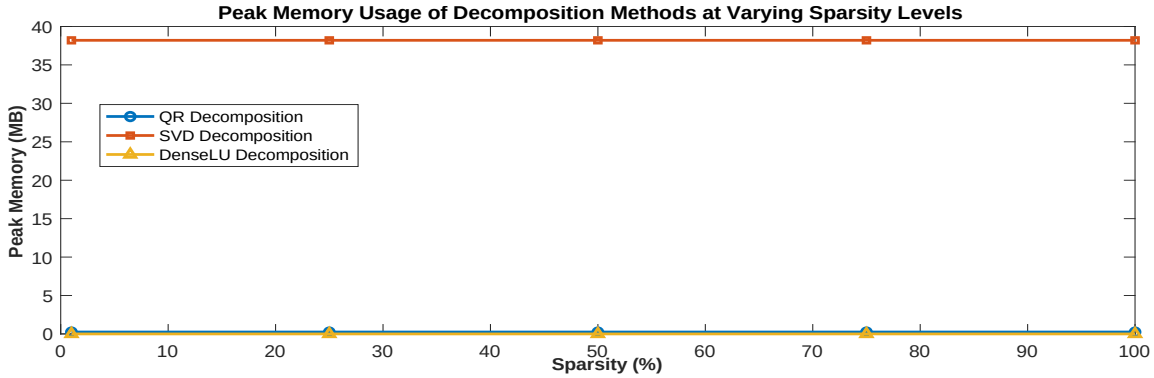

Figure 2.2: Comparison of Peak Memory of Matrix Decomposition Methods at Different Sparsity Levels

Observation and Conclusion 2.1:
According to Fig 2.1, it is observed that the running time of different decomposition methods remains relatively stable at different levels of sparsity. This may be due to the algorithm like QR decomposition which is able to efficiently identify and skip the zero elements and thus maintains a stable running time. In contrast, the slight increase in runtime for SVD may reflect its under-optimisation for sparse matrices, where the additional computational load leads to an increase in time as the number of non-zero elements increases.

According to Fig 2.2, the peak memory usage of the different decomposition methods remains constant for all sparsity levels. QR shows robustness to sparsity, which suggests that its memory requirement is not strongly related to the number of non-zero elements. The peak memory of SVD is consistently higher, implying that it requires a consistently high amount of memory to handle the decomposition of the matrices, which is independent of the sparsity level of the matrices. Whereas DenseLU consistently has the lowest memory usage, probably because it uses efficient data structures and algorithms in processing sparse matrices, thus achieving memory efficiency at various sparsity levels.

Overall, the observed data largely confirms our expectation that the running time and peak memory usage of the different decompositions are not sensitive to changes in sparsity. Given the time budget, further testing may be necessary to test other decomposition methods to see if similar patterns of sparsity insensitivity exist.

Question 2.2: In the above, can you deduce how the run time and storage values scale with matrix size N?

Expectation 2.2: As the matrix size N increases, the runtime and storage need of the various decomposition methods rise. For the QR and DenseLU decompositions, this increase is likely to be largely determined directly by the matrix size, showing a polynomial-level increase. For SVD decomposition, the increase in runtime and storage needs is expected to be even more significant as its algorithms inherently require more internal operations.

Experiment 2.2: The sparsity of the matrix was fixed to 100% and three decomposition algorithms were chosen: the QR decomposition, the SVD decomposition and the DenseLU decomposition. By varying the sparsity of the matrix (increasing from 1000 to 3000), Table 2.2 records the running time and peak memory usage of each decomposition method for different matrix sizes.

Table 2.2: Performance Metrics of Matrix Decomposition Methods at Varying Matrix Sizes

Matrix Size	QR Decomposition		SVD Decomposition		DenseLU Decomposition	
	Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)	Runtime (s)	Peak Memory (MB)
1000	0.596	0.252	3.138	38.2	0.186	0.00381
1500	2.041	0.378	10.3	86	0.635	0.00572
2000	5.128	0.504	21.447	153	1.457	0.00763
2500	9.509	0.629	40.264	239	2.841	0.00954
3000	15	0.755	69.469	344	4.994	0.0114

Plot graph 2.2:

The data in Table 2.2 for running time and peak memory for different decomposition methods at different matrix sizes are plotted in Figures 2.3 and 2.4, respectively. Figure 2.5 is an enlarged version of Figure 2.4 to facilitate observation of the curve characteristics of certain methods.

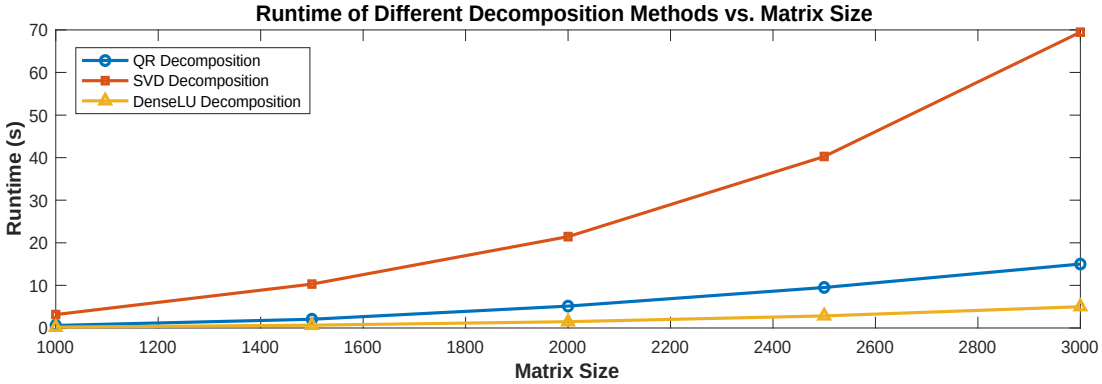

Figure 2.3: Comparison of Running Time of Matrix Decomposition Methods at Different Matrix Sizes

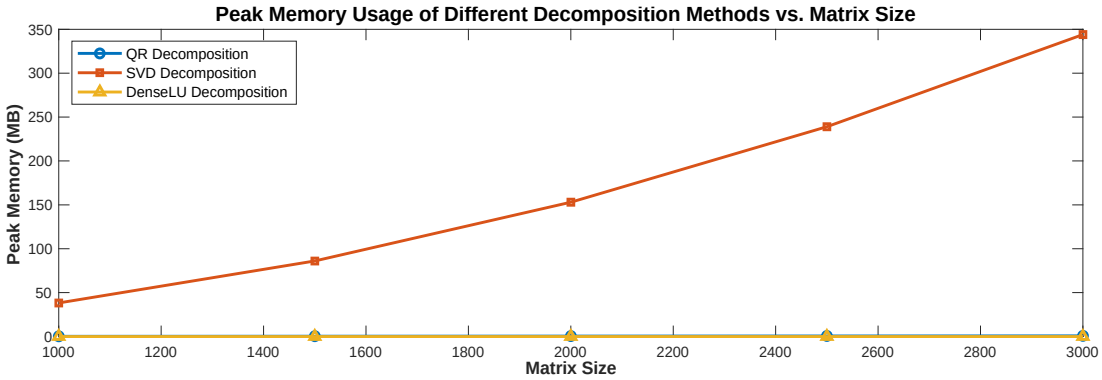

Figure 2.4: Comparison of Peak Memory of Matrix Decomposition Methods at Different Matrix Sizes

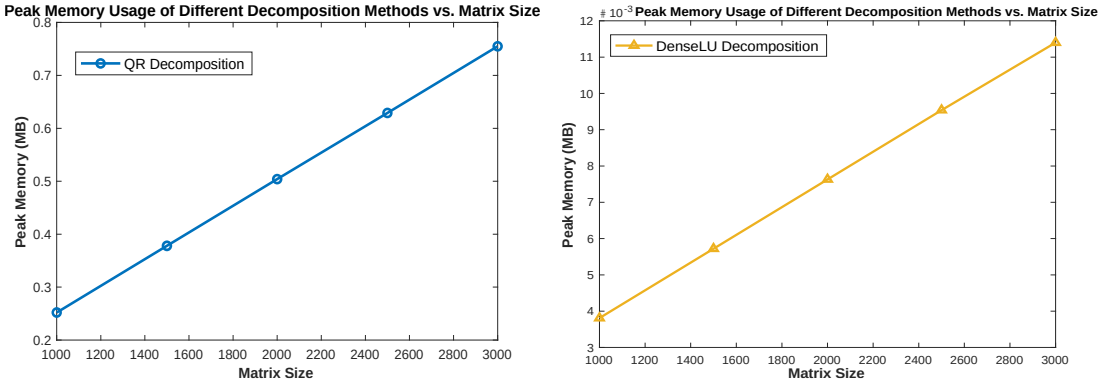

Figure 2.5: Enlarged version of a section for Fig 2.4 (Left for QR, Right for DenseLU)

Observation and Conclusion 2.2:

From Figure 2.3, it is observed that the running time of all the decomposition methods tends to increase as the matrix size N increases. The running time of QR and DenseLU decomposition grows more moderately, showing a nearly linear trend, which may be attributed to these two algorithms are relatively simple to operate, and their complexity is directly related to the number of elements in the matrix. On the contrary, SVD has a much steeper runtime growth, which implies that its algorithmic complexity may be super-linear, especially when it comes to singular value decomposition, which is a much more computationally intensive process.

In Figures 2.4 and 2.5, the peak memory usage of the three methods increases with the matrix size, reflecting the fact that the storage requirement increases with the matrix size. The peak memory usage of QR and DenseLU decomposition is approximately linear with the matrix size, which may be due to these methods are able to handle the increased amount of data with lower complexity. The SVD decomposition, on the other hand, requires significantly more peak memory and exhibits a faster growth rate with increasing matrix size, possibly because the SVD algorithm needs to allocate more temporary storage space for its complex internal operations.

The above observations are in line with expectations and show that the matrix size indeed has a significant impact on the runtime and peak memory usage of the decomposition methods. These emphasise the importance of choosing an appropriate decomposition method when dealing with large matrix problems. Especially when computational resources are limited, SVD decomposition may not be an efficient choice. Finally, to gain insight into the performance of these decomposition methods, further tests can be carried out over a wider range of matrix sizes and different sparse structure patterns can be considered.

3. Direct Solution Methods: Impact of Data Reordering: 10%

Question: Re-run some of the SVD, QR and denseLU decompositions above using each reordering possibility recording the essential results and required comment if there is any impact on the engineering measures of interest and if the incurred cost of performing the reordering means it is worth doing it here.

Expectation: For different decomposition methods, data reordering does not significantly affect their runtime and memory usage when dealing with sparse and dense matrices.

Experiment: In the first experiment, the changes in the running time and extra memory requirements of the QR, SVD and DenseLU decomposition methods are recorded for 1000 size matrices with sparsity of 1%, 10%, and 100% after reordering using the Metis library and the Reverse Cuthill-McKee method. The second experiment was similarly evaluated for matrices of size 1000, 1500, and 2000 with sparsity at 100%. The data are recorded in Table 3.1 and 3.2. The data in the table are plotted in Figures 3.1 and 3.2 to facilitate observation of the results.

Plot graph:

Table 3.1: Time and Memory Impact of Data Reordering Methods Across Various Matrix Sparsities

Reordering Type	**Decomposition Type**	**QR**			**SVD**			**DenseLU**		
	Sparsity (%)	1%	10%	100%	1%	10%	100%	1%	10%	100%
Metis Library	**Decomposition Time (s)**	0.584	0.776	0.838	3.244	3.143	3.193	0.185	0.24	0.189
	Additional Memory (MB)	0.252	0.252	0.252	38.2	38.2	38.2	0.00381	0.00381	0.00381
Reverse Cuthill McKee	**Decomposition Time (s)**	0.584	0.606	0.649	3.199	3.175	3.207	0.183	0.181	0.185
	Additional Memory (MB)	0.252	0.252	0.252	38.2	38.2	38.2	0.00381	0.00381	0.00381

Table 3.2: Time and Memory Impact of Data Reordering Methods Across Various Matrix Sizes

Reorder Type	Decomposition Type	QR			SVD			DenseLU		
	Matrix Size	1000	1500	2000	1000	1500	2000	1000	1500	2000
Metis Library	Decomposition Time (s)	0.616	2.092	4.967	3.153	10.512	21.409	0.185	0.623	1.455
	Additional Memory (MB)	0.252	0.378	0.504	38.2	86	153	0.0038	0.0057	0.0076
Reverse Cuthill McKee	Decomposition Time (s)	0.586	2.037	4.991	3.164	10.66	21.057	0.181	0.626	1.544
	Additional Memory (MB)	0.252	0.378	0.504	38.2	86	153	0.0038	0.0057	0.0076

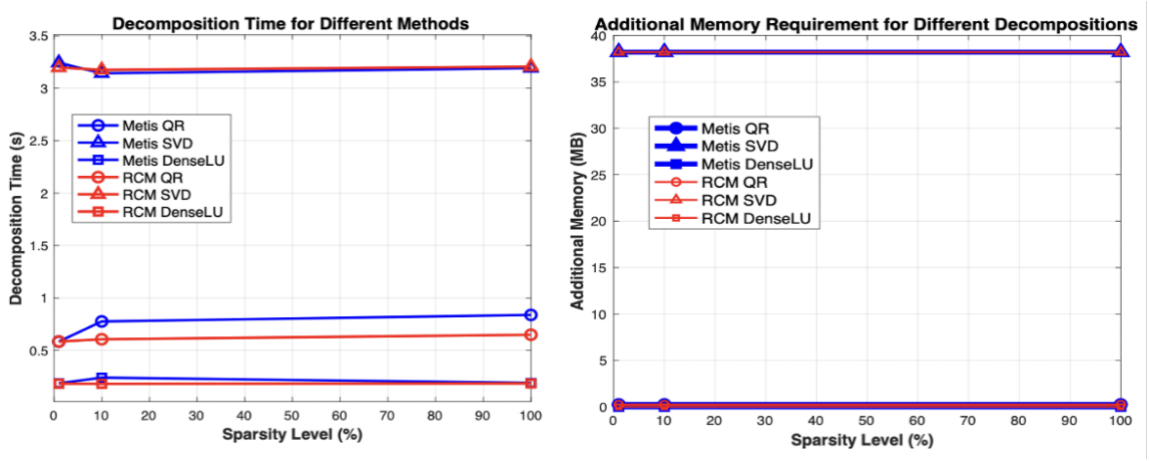

Figure 3.1: Reordering Impact on Decomposition Time (Left) and Memory (Right) Across Sparsities

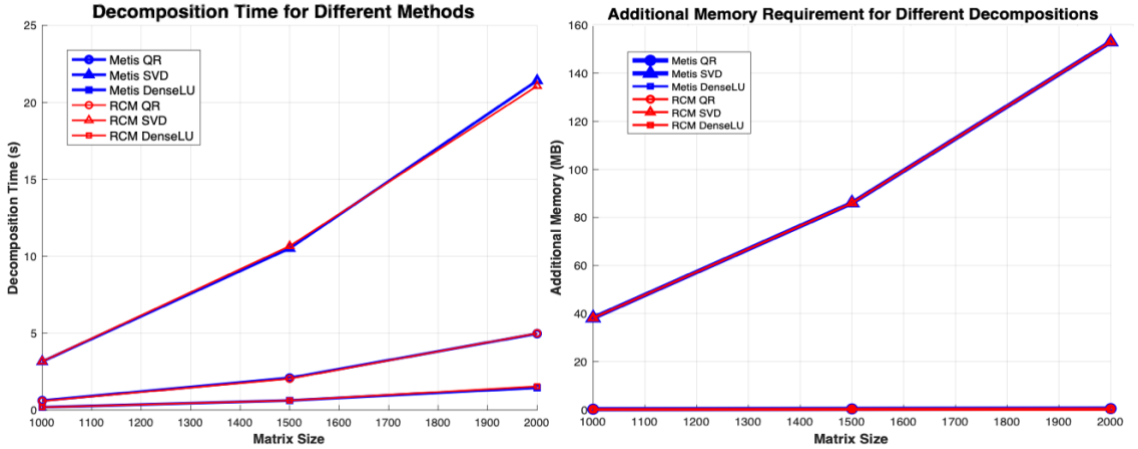

Figure 3.2: Reordering Impact on Decomposition Time (Left) and Memory (Right) Across Matrices

Observation and Conclusion:

In Fig 3.1, the decomposition time and additional memory required for different rearrangement methods appear to be constant for different sparsities. This suggests that reordering has a negligible effect on the computation time, which may be because the decomposition algorithm handles sparsity efficiently or the overhead of reordering is small compared to the total decomposition time. The memory requirement remains constant possibly because the memory footprint is mainly determined by the size of the data structures used in the decomposition.

In Fig 3.2, time, and memory increase with the matrix size. All methods show a consistent trend that the larger the matrix, the more time and memory is required.

In summary, the data presented in Figures 3.1 and 3.2 support the expectation, future surveys will expand the matrix size and employ other reordering techniques that will be valuable in confirming trends and revealing nuances not observed in the current dataset.

4. Iterative Solution Methods: Convergence of the Basic Methods: 20%

Question: Record and present the convergence versus iteration, noting the run time to achieve convergence to a residual error of 10-6. Comment on the performance and costs of the four approaches.

Expectation: All four iterative methods are theoretically capable of being used to solve large sparse matrix problems when the maximum number of iterations is large enough so that the final residuals should be able to be controlled within the required limits. This predicts that if a method does not achieve the expected accuracy within a reasonable number of iterations under resource-constrained conditions, then the method may not be applicable to large problems or problems with specific sparsity. The increase in sparsity and matrix size, it may theoretically require more iterations and longer running time to reach the convergence condition.

Experiment 1:

The matrix size was fixed to 1000. Four iterative solving methods (Steepest Descent, Minimal Residual, GMRES, BiCGSTAB) were tested for different sparsity levels (1%, 10%, and 100%). The maximum number of iterations was set to 10000, and the target residual threshold was 1×10^{-6}. The number of iterations required for each method, the final residual value, the running time, and whether it was successfully reduced to the threshold value are recorded in Table 4.1. The condition number is an important measure of the sensitivity of the matrix numerical stability solution. A method is considered unsuccessful in solving the problem if it fails to reduce the residuals to the target threshold before the maximum number of iterations is reached.

Plot graph 1:

Table 4.1: Comparison of Iterative Methods for Solving Matrixs at Different Sparsities

Methods:		**Steepest descent**	**Minimal residual**	**GMRES**	**BiCGSTAB**
Sparsity: 1%	**Run time (s)**	2.306	2.189	0.003	0.003
	Iterations	9999	9999	11	6
	Residual	3.40171e+307	0.428694	1.28069e-13	3.73466e-09
	Condition	Unsuccessful	Unsuccessful	Successful	Successful
Sparsity: 10%	**Run time (s)**	3.838	3.654	0.014	0.009
	Iterations	9999	9999	34	13
	Residual	3.9725e+306	407.779	7.46611e-07	5.24036e-07
	Condition	Unsuccessful	Unsuccessful	Successful	Successful
Sparsity: 100%	**Run time (s)**	14.57	14.8	1.671	4.31
	Iterations	9999	9999	960	1675
	Residual	2.54202e+306	95364.5	3.50305e-07	5.61202e-07
	Condition	Unsuccessful	Unsuccessful	Successful	Successful

Observation and Conclusion 1:

From Table 4.1, it can be observed that Steepest Descent and Minimal Residual, methods do not converge to the required residual values at all sparsity levels and very high when the maximum limit of the number of iterations is reached, which indicates a failure of convergence. GMRES and BiCGSTAB methods successfully converge at all sparsity levels and significantly require fewer number of iterations and running time compared to the other two methods, BiCGSTAB required the least number of iterations and running time among all the tested methods, which indicates that the method is more efficient. As the sparsity increases, the running time of all methods increases, and the number of iterations required for convergence increases for both GMRES and BiCGSTAB methods.

These findings are consistent with the expectation that not all iterative methods are suitable for all types of problems and that the efficiency of iterative methods depends on the degree of sparsity. Given the poor and uncompetitive performance of the steepest descent and minimum residual methods, they were not considered in subsequent tests. It is worth noting that although BiCGSTAB has a higher iteration at 100% sparsity compared to GMRES, BiCGSTAB still has a significantly shorter runtime, suggesting that it may be more efficient per iteration. A detailed comparison of these two methods is presented in following.

Experiment 2:

The matrix sparsity was fixed at 10%. Two iterative solving methods (GMRES and BiCGSTAB) were tested for different matrix sizes (1000, 2500 and 5000). The maximum number of iterations was set to 10000 and the target residual threshold was 10^{-6}. Table 4.2 records the number of iterations required for each method, the final residual value, the running time, and whether the threshold was successfully reduced.

Plot graph 2:

Table 4.2: Comparison of Iterative Methods for Solving Matrixs at Different Matrix sizes

Methods:		**GMRES**	**BiCGSTAB**
Matrix size: 1000	**Run time (s)**	0.014	0.009
	Iterations	34	13
	Residual	7.46611e-07	5.24036e-07
	Condition	Successful	Successful
Matrix size: 2500	**Run time (s)**	0.042	0.042
	Iterations	37	14
	Residual	9.84609e-07	5.93757e-07
	Condition	Successful	Successful
Matrix size: 5000	**Run time (s)**	0.163	0.221
	Iterations	39	16
	Residual	9.40415e-07	1.07435e-07
	Condition	Successful	Successful

Observation and Conclusion 2:

For the data in Table 4.2, both GMRES and BiCGSTAB are robust iterative methods that can effectively handle the increase in matrix size without substantial performance loss in terms of final residual values. In terms of the number of iterations, BiCGSTAB appears to be more

efficient, requiring fewer iterations even with increasing matrix size, and is still competitive although its runtime is longer with larger matrices.

As the matrix size increases, the number of iterations increases slightly while the runtime increases more, which is consistent with the expectation that more computational resources are typically required to solve larger problems. Future experiments may focus on optimizing these methods, especially BiCGSTAB, for larger matrix sizes, as well as investigating their behavior under different sparsity patterns and non-zero element distributions.

5. Iterative Solution Methods: The Impact of Data Reordering: 10%

Question: Reassess (some of !) the four iterative schemes using (i) Reverse Cuthill-Mckee reordering, (ii) the Metis library reordering.

Expectation: Either the Reverse Cuthill-Mckee or the Metis library, these two reordering methods may not have a significant effect on the results of the four iterative methods.

Experiment: Mainly, it is still performed in two parts, like what was tested in question 4, except that two rearrangement methods (Reverse Cuthill-Mckee and Metis library) are used in each part separately to observe the impact.

First, the matrix size was fixed to 1000. Four iterative methods of solving (Steepest Descent, Minimum Residual, GMRES, BiCGSTAB) using the two reordering methods were tested for different levels of sparsity (1%, 10% and 100%). Tables 5.1 and 5.2 record the number of iterations required, the final residual value, the running time, and whether the reduction to the threshold was successful for each reordering method.

After that, the matrix sparsity was fixed at 10%. Two iterative methods solving (GMRES and BiCGSTAB) using two reordering methods were tested for different matrix sizes (1000, 2500 and 5000). Tables 5.3 and 5.4 record the data under the two reordering methods.

Plot graph:

Table 5.1: Comparison of Iterative Methods for Solving Matrixs at Different Sparsities (Metis library)

Methods:		**Steepest descent**	**Minimal residual**	**GMRES**	**BiCGSTAB**
Sparsity: 1%	**Run time (s)**	3.904	3.817	0.006	0.005
	Iterations	9999	9999	11	6
	Residual	7.16437e+307	0.0625623	6.81675e-15	6.66093e-13
	Condition	Unsuccessful	Unsuccessful	Successful	Successful
Sparsity: 10%	**Run time (s)**	5.937	6.432	0.029	0.023
	Iterations	9999	9999	35	14
	Residual	4.19843e+306	385.927	3.93802e-07	1.74879e-07
	Condition	Unsuccessful	Unsuccessful	Successful	Successful
Sparsity: 100%	**Run time (s)**	20.597	19.833	2.57	8.685
	Iterations	9999	9999	950	1985
	Residual	6.6888e+306	19640.6	3.92982e-07	7.39115e-07
	Condition	Unsuccessful	Unsuccessful	Successful	Successful

Table 5.2: Comparison of Iterative Methods for Solving Matrixs at Different Sparsities (Reverse Cuthill-Mckee)

Methods:		Steepest descent	Minimal residual	GMRES	BiCGSTAB
Sparsity: 1%	Run time (s)	3.863	3.777	0.006	0.004
	Iterations	9999	9999	11	6
	Residual	7.10325e+307	0.0305808	5.71801e-14	5.38094e-10
	Condition	Unsuccessful	Unsuccessful	Successful	Successful
Sparsity: 10%	Run time (s)	5.878	6.388	0.028	0.022
	Iterations	9999	9999	35	13
	Residual	3.35127e+306	353.595	3.90167e-07	1.08426e-07
	Condition	Unsuccessful	Unsuccessful	Successful	Successful
Sparsity: 100%	Run time (s)	0.493	20.049	1.671	6.656
	Iterations	194	9999	950	1521
	Residual	nan	117692	4.23676e-07	6.62878e-07
	Condition	Unsuccessful	Unsuccessful	Successful	Successful

Table 5.3: Comparison of Iterative Methods for Solving Matrixs at Different Matrix sizes (Metis library)

Methods:		GMRES	BiCGSTAB
Matrix size: 1000	Run time (s)	0.006	0.005
	Iterations	11	6
	Residual	6.81675e-15	6.66093e-13
	Condition	Successful	Successful
Matrix size: 2500	Run time (s)	0.1	0.103
	Iterations	37	14
	Residual	8.26061e-07	1.94611e-07
	Condition	Successful	Successful
Matrix size: 5000	Run time (s)	0.378	0.441
	Iterations	39	16
	Residual	9.1664e-07	9.77665e-08
	Condition	Successful	Successful

Table 5.4: Comparison of Iterative Methods: Solving Matrixs at Different Matrix (Reverse Cuthill-Mckee)

Methods:		GMRES	BiCGSTAB
Matrix size: 1000	Run time (s)	0.006	0.004
	Iterations	11	6
	Residual	5.71801e-14	5.38094e-10
	Condition	Successful	Successful
Matrix size: 2500	Run time (s)	0.098	0.096
	Iterations	37	13
	Residual	8.82708e-07	7.28526e-07
	Condition	Successful	Successful
Matrix size: 5000	Run time (s)	0.416	0.472
	Iterations	39	15

	Residual	8.85385e-07	1.47425e-07
	Condition	Successful	Successful

Observation and Conclusion:

As can be seen from the data in the tables above reordering methods such as Reverse Cuthill-McKee and Metis library may not have a significant impact on the iterative solver. The matrices used may already be optimally structured for the solver, so reordering is redundant. Some iterative methods are robust to matrix ordering, thus undermining the potential benefits of reordering. Furthermore, if the computational cost of reordering outweighs its benefits, the overall performance improvement may be minimal. It is worth noting that the steep slope descent method under the Reverse Cuthill-Mckee reordering has a residual value of "nan" at the 100% sparsity level, which may indicate a computational problem during the iteration process. This could be a numerical instability or an overflow error that causes the solver to fail, thus failing to provide a valid numerical residual value. This could be due to the conditions of the matrix making the problem unfavorable for this type of solver.

In conclusion, the results of this experiment were generally in line with expectations and additional reordering methods could be used in the future to determine this conclusion.

6. Iterative Solution Methods: Impact of Preconditioning: 20%

Question 6.1: Select just the "best" one or two of the iterative methods and select a few particular matrices (i.e. sparsity and size) as the test cases for further exploration.

Expectation 6.1: The best iterative method for dealing with the performance of large sparse matrices is the BiCGSTAB method, which has a fast run time, fewer iterations, and successful convergence.

Experiment 6.1: BiCGSTAB iterative method was tested for three matrix sizes (1000, 5000, and 7000) at 1%, 10%, and 100% sparsity, respectively, and the test results were recorded in Table 6.1 for run time, iterations, and whether the iteration was successful.

Plot graph 6.1:

Table 6.1: Simulation Result Using BiCGSTAB iterative method

Sparsity (%)	Matrix Size	Run time (s)	Iterations	Residual	Condition
1	1000	0.003	6	3.73466E-09	Successful
	5000	0.027	9	5.82272E-07	Successful
	7000	0.055	10	4.50999E-07	Successful
10	1000	0.016	13	5.24036E-07	Successful
	5000	0.35	16	1.07435E-07	Successful
	7000	0.642	15	5.01195E-07	Successful
100	1000	7.297	1675	5.61202E-07	Successful
	5000	735.534	9999	0.033148	Unsuccessful
	7000	368.63	4999	0.0369407	Unsuccessful

Observation and Conclusion 6.1:
At lower sparsity levels (1% and 10%), BiCGSTAB has a fast runtime and a low number of iterations. The method successfully converges to the desired accuracy within a reasonable number of iterations. However, as the sparsity increases to 100%, the ability of BiCGSTAB to handle large matrices diminishes. At matrix sizes of 5000 and 7000, the method fails to converge within the set iteration limits, suggesting that BiCGSTAB is limited when dealing with fully dense matrices or matrices with high sparsity, and that preprocessing methods such as ILUT may need to be used to improve the convergence of high-density matrices.

Question 6.2: Introduce the incomplete LU with Thresholding, ILUT, decomposition to see if whether the use of thresholding and reordering combined allows solution of generally larger sparse matrix problems?

Expectation 6.2: Incorporating ILUT and reordering into the BiCGSTAB method is expected to significantly improve its efficiency in solving large sparse matrices. It is expected that this approach will speed up convergence and reduce the number of iterations required, thus allowing the solver to handle large matrices that were previously too demanding due to computational and memory constraints.

Experiment 6.2: ILUT with thresholds was introduced based on the experiments done in problem 6.1 for various scenarios, and three thresholds of 0.01, 0.1, and 1 were selected for testing. In addition to this, two reordering methods (Reverse Cuthill-Mckee and Metis library) were introduced and tested separately. The matrix size was fixed at 1000 and the sparsity value was fixed at 1% to make it easier to determine whether the introduced ILUT or in combination with the reordering methods could improve the efficiency. The data were recorded in Table 6.2. The better performing reordering method was tested with a matrix size of 7000 and a sparsity value of 10% at three different thresholds to test whether the introduction of ILUT or the combination with the rearrangement method could solve larger sparse matrices. The data was recorded in Table 6.3.

Plot graph 6.2:

Table 6.2: Results of the BiCGSTAB combining ILUT at different thresholds and reordering methods

Threshold	Reorder Type	Run time (s)	Iterations	Residual	Condition	Peak Memory (MB)
0.01	None	0.145	13	3.94067E-07	Successful	2.14
	Reverse	0.017	7	2.85312E-07	Successful	0.17
	Metis	0.004	2	19.0544E-12	Successful	0.0953
0.1	None	18.838	4999	3928.5	Unsuccessful	8.47
	Reverse	0.006	4	1.60922E-07	Successful	0.0587
	Metis	0.005	3	8.01359E-07	Successful	0.0786
1	None	0.038	74	5.5477E-08	Successful	0.085
	Reverse	0.926	1573	5.49912E-07	Successful	0.243
	Metis	0.006	5	5.39197E-02	Successful	0.0238

The data in Table 6.3 are rearranged using the Metis library because of its better performance results.

Table 6.3: Results of the BiCGSTAB combining ILUT at different thresholds and Metis reordering methods

Threshold	Run time (s)	Iterations	Residual	Condition	Peak Memory (MB)
0.01	2.155	7	8.42792e-09	Successful	50.2
0.1	2.543	20	4.33858e-07	Successful	41.8
1	3.079	63	1.56327e-07	Successful	16

Observation and Conclusion 6.2:

As can be seen from the data in Table 6.2, despite the use of ILUT preprocessing, this approach sometimes leads to an increase in runtime, a higher number of iterations, and a rise in memory usage when dealing with sparse matrices. This may be due to the high computational overhead of the preprocessing itself, especially when many additional non-zero elements are introduced during the preprocessing.

When combining ILUT with the reordering method, the processing matrices become somewhat better. This is because this combination reduces the memory requirement and accelerates the convergence of the iterative solver by reducing the padding of non-zero elements, increasing the locality of the data, improving the condition number of the matrix, and optimizing the structure of the parallel computation. Among them, the Metis library method works better than Reverse Cuthill-McKee. Since Metis reordering focuses on graph partitioning to reduce padding and improve cache efficiency. This typically leads to better parallel performance and less padding of non-zero elements, which reduces computation and memory requirements. RCM reordering optimizes performance by reducing matrix bandwidth, which may be effective for some problems, but it is not as nuanced as Metis to optimize memory access patterns and reduce padding of non-zero elements.

The choice of threshold value also affects performance to some extent, with 0.1 being a better choice overall, but the effect is not significant. A lower threshold (0.01) may cause the algorithm to perform the preprocessing steps more accurately, which in turn may increase memory usage and computation time. Conversely, a higher threshold (1) may reduce the accuracy of the preprocessing and decrease memory requirements and computation time, but possibly at the expense of solution accuracy and numerical stability.

The data in Table 6.3 can show that the use of thresholding and reordering can solve generally larger sparse matrix problems, as expected.