# Fast Iterative Solvers

Summer Semester 2023

Project 1

Sreekar Reddy, Sajjala | Matriculation Nu. 415917

## Abstract:

This project focuses on implementation of two iterative solvers, Generalized Minimal Residual (GMRES) and Conjugate Gradient (CG), for solving sparse linear systems. The implementation explores the impact of preconditioning on the convergence behavior and computational efficiency of the GMRES method, considering different restart parameters (m = 30, 50, 100). The sparse linear systems are represented using the Modified Sparse Row (MSR) format for efficient storage and manipulation. Conjugate Gradient method, designed for symmetric positive definite systems, is implemented. Several evaluations are conducted to analyze convergence rate, Krylov vectors, and execution time. The results highlight the effectiveness of GMRES with preconditioning in achieving faster convergence. CG with relative residual and absolute error is compared.

## Methodology:

All the programming has been written in python. Libraries such as Numpy and Matplotlib were employed to read, organize, and manipulate arrays and create plots respectively. The test matrices are processed to extract the symmetry flags, dimension, and the sparsely formatted matrix.

### Matrix-Vector Product

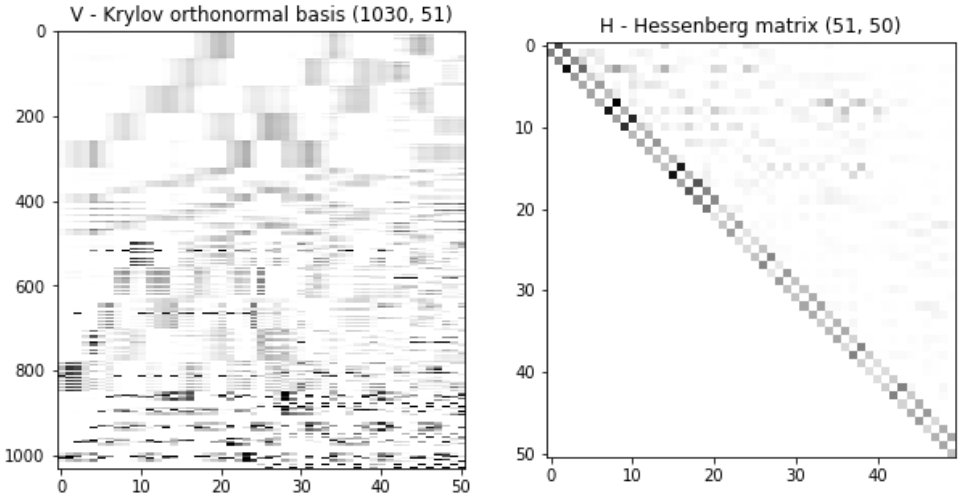
```
def msr_matmul(J      :np.ndarray,
               V      :np.ndarray,
               X      :np.ndarray,
               symm   :bool) -> Ax: np.ndarray:
```

A function (msr_matmul) for matrix-vector product for a matrix stored in MSR format has been implemented for both symmetric and non-symmetric matrices. Within the function, there are three parts: diag_matmul (multiplication for diagonal elements), csr_matmul, and csc_matmul. If the matrix is non-symmetric the function returns the summation of first two parts, else if the matrix is symmetric, since only one half of the matrix is stored, the function returns the summation of all three parts. This function has been called in GMRES and CG implementation when there is a matrix-vector product involving a sparse matrix.

### Krylov Subspace

```
def Arnoldi(msr_mat : np.ndarray,
            symm    : bool,
            b       : np.ndarray,
            m       : int) -> tuple:
-----------------------------------
return      V       : np.ndarray,
            H       : np.ndarray
```

The above function computes the Krylov subspace vectors and Hessenberg matrix for the given inputs in appropriate format. The result from the function for 50 iterations is plotted below indicting that the Hessenberg matrix is mostly upper triangular with one additional element in lower triangle each row.




The output Krylov vector satisfied the orthogonality condition and also for the following relations when Arnoldi iteration has been executed.

$$A.V = V\_n.H$$
$$H = V\_n * .A.V\_n$$

## Generalized Minimal Residual (GMRES)

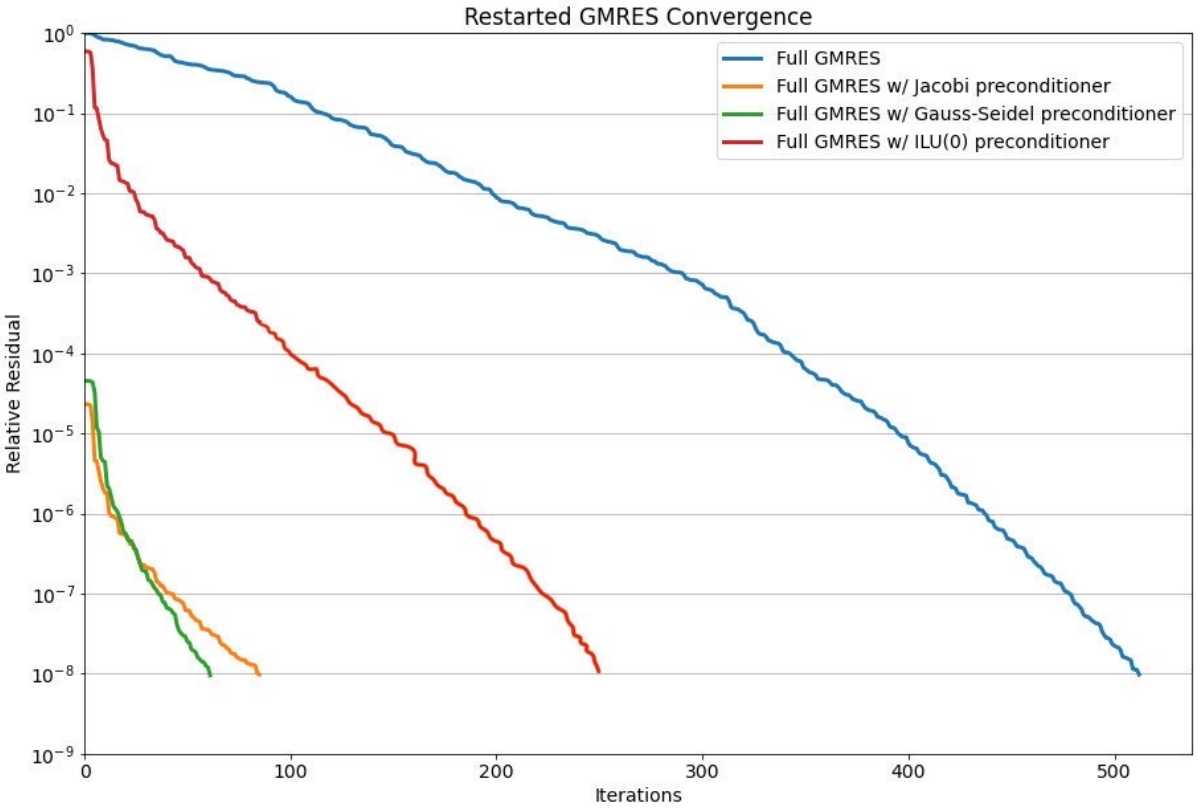
For GMRES two functions run_gmres() and run_restart_gmres() were implemented as suggested in the lecture. The inputs of these functions can be noticed from the following function declarations.

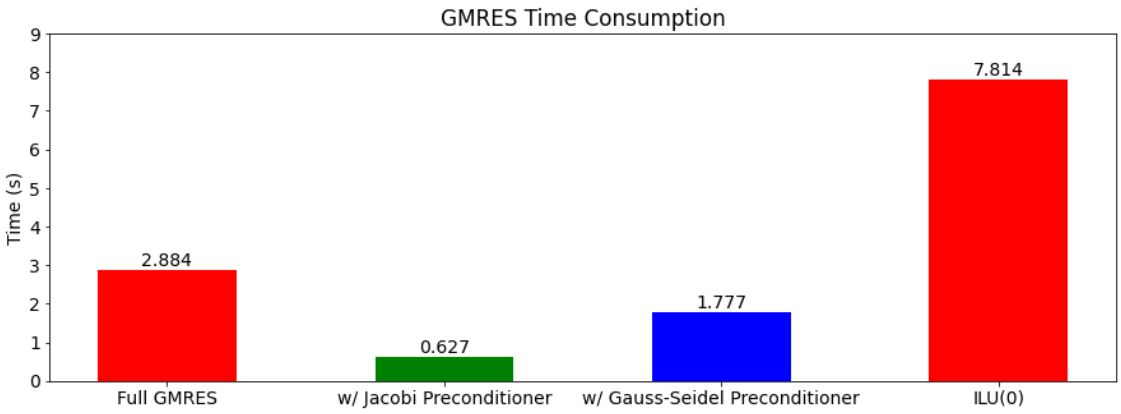
```
def gmres(msr_mat      : np.ndarray,
          symm         : bool,
          x            : np.ndarray,
          b            : np.ndarray,
          max_iter     : int,
          tol          : float = 1e-8,
          precond      : str = None) -> tuple:
-------------------------------------
return    x            : np.ndarray,
          error        : list,
          elapsed_time : float,
          krylov_vector: np.ndarray

def restart_gmres(msr_mat      : np.ndarray,
                  symm         : bool,
                  b            : np.ndarray,
                  restart      : int,
                  tol          : float =1e-8) -> tuple:
-----------------------------------------
return             x_n          : np.ndarray,
                   error        : np.ndarray,
                   elapsed_time : float
```

The outputs for the functions can be noticed from the return values of the respective functions.

The solution vector $x$ is defined as all ones, through which the $b$ vector is computed using msr_matmul function. The initial guess for $x$ is set to zeros and the function has a condition to break the loop early when the relative residual $(\|r_k\|_2/\|r_0\|_2)$ reaches less than or equal to the tolerance ($1e$-8).

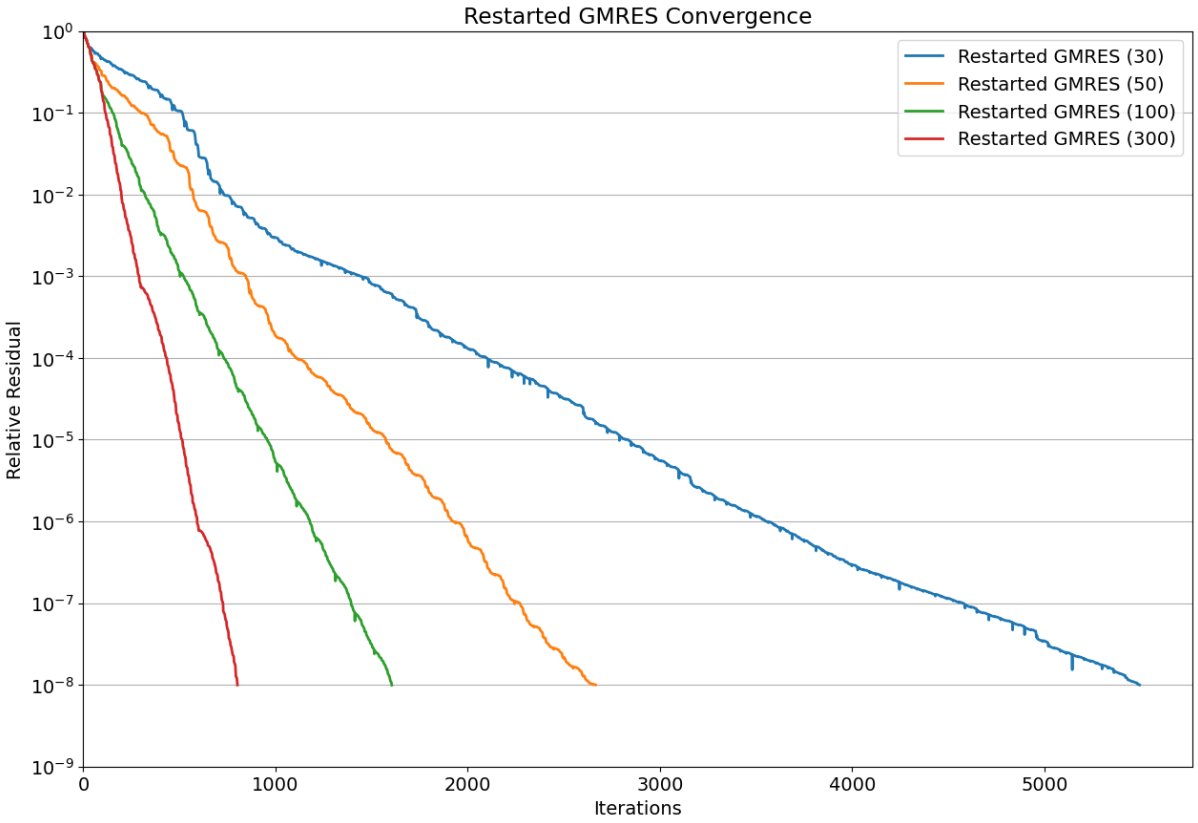
Full GMRES has been evaluated with restart_gmres where the restart parameter is 600. The solution converged to 9.760 E-09 and required 512 Krylov vectors. Preconditioned GMRES have also been executed with the same restart parameter and convergence is achieved much quicker in contrast to the former run. GMRES with Jacobi preconditioner reached convergence with 85 Krylov vectors, with Gauss-Seidel preconditioner reached convergence with only 62 Krylov vectors, and ILU(0) preconditioner reached convergence with 268 Krylov vectors. The following convergence plot shows the comparison between these three runs and the bar plot shows the time consumed.

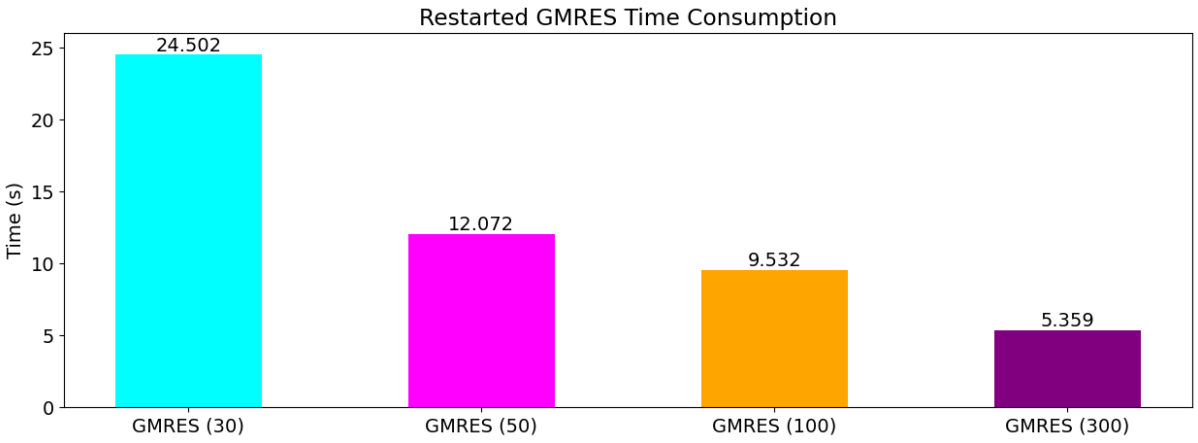


It can be seen that the time consumed for preconditioned GMRES is significantly lower for Jacobi in particular while ILU(0) preconditioned GMRES took significantly longer. It is not certain if the ILU(0) implementation is accurate for this project.

GMRES function has been executed with different restart parameters i.e., 30, 50, 100, and 300. It is evident from the following plots that the GMRES(30) required more than 5000 iterations to reach convergence and for others it is gradually lowering. The time consumption is also significantly higher for restarted GMRES compared to full GMRES. It can be seen that as the restart parameter is increased the time consumption is decreased. GMRES(300) is executed for exploratory purposes.

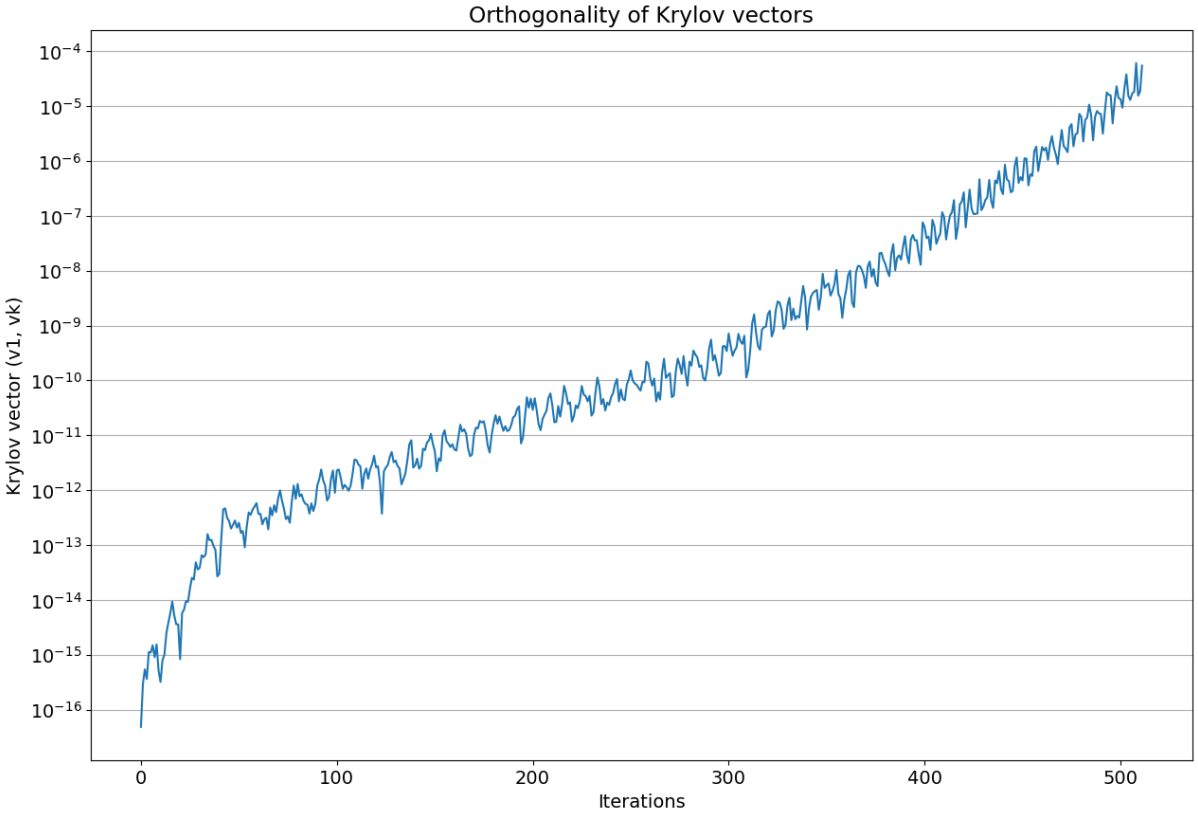


Tests have been performed with restart paramters 10, 20, 60, 70, 80, 90, 110, … 400, … until 500 but no time advantage is observed compared to the Full GMRES. Ignoring runtime, restarted GMRES has some advantages over Full GMRES:

1. *Memory Efficiency:* Restarted GMRES uses a fixed amount of memory, regardless of the total number of iterations. It only keeps the most recent basis vectors, reducing the memory requirements compared to full GMRES, which retains all the basis vectors.
2. *Computational Efficiency:* Restarted GMRES can be more computationally efficient since it performs iterations in smaller subspaces. This can result in faster convergence for certain problems, especially when the solution is dominated by a few dominant eigenvalues.
3. *Potential for Improved Convergence:* By resetting the Krylov subspace, restarted GMRES can potentially overcome stagnation or slow convergence issues that may occur in full GMRES.

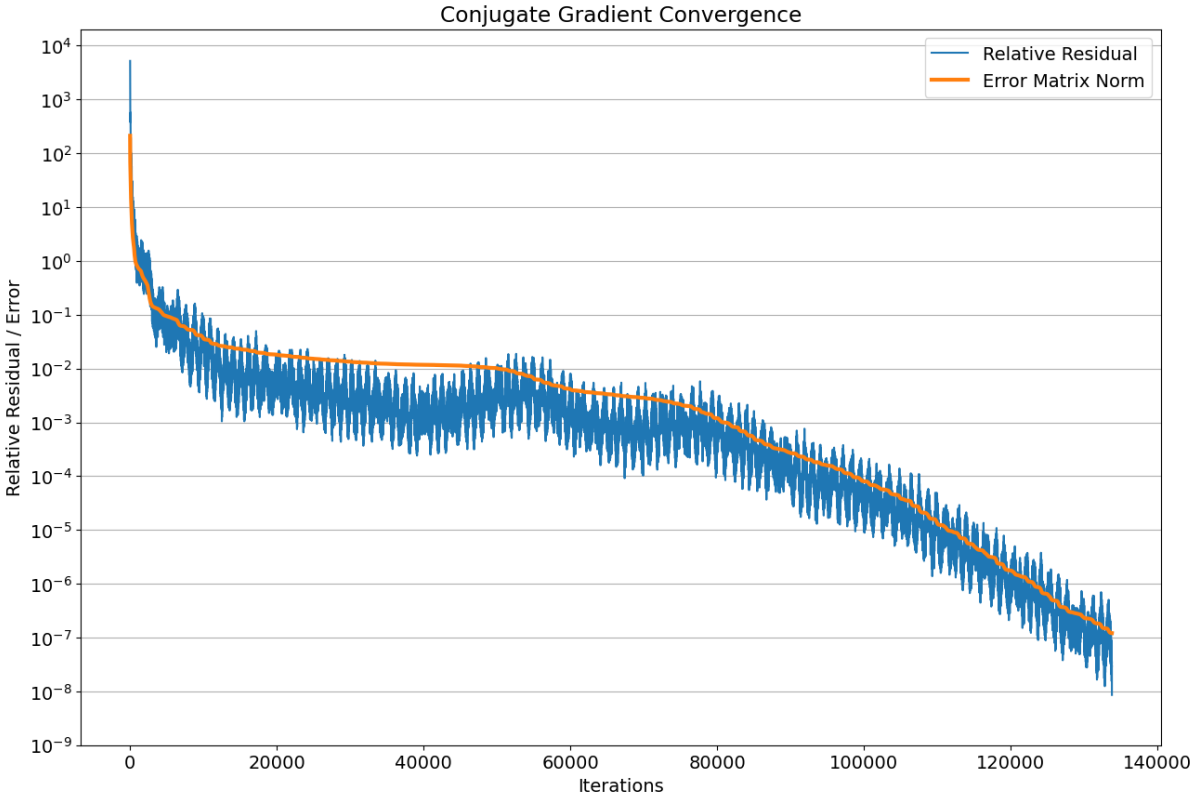
The krylov subspace vectors for full GMRES without preconditioning are checked for orthogonality and the computed values $(V_1, V_k)$ are plotted below.



Conjugate Gradient (CG)

```
def conjGrad(msr_mat     : np.ndarray,
             b           : np.ndarray,
             symm        : bool,
             iter        : int,
             tol         : float=1e-8) -> tuple:
-------------------------------------------
return       x           : np.ndarray,
             residual    : np.ndarray,
             error       : np.ndarray,
             round(end-start, 3): float
```

The above function implements the CG algorithm discussed in the lecture. The function outputs relative residual and matrix norm $\|e\|_A = \sqrt{(Ae, e)}$ as residual and error, respectively. The relative residual of the solver converged to 8.57e-09 after 133,765 iterations. The following semilogy plot shows the residual and error both follow similar trend throughout the runtime.




The difference in convergence behavior between the error norm $\|e\|_A$ and the residual norm $\|r\|_2$ arises from their distinct measures of accuracy. The error norm $\|e\|_A$ captures the approximation quality of the solution in the norm defined by matrix A. It assesses how well the CG method converges in the problem-specific norm, which may be influenced by the eigenvalues and eigenvectors of matrix A. On the other hand, the residual norm $\|r\|_2$ focuses on the accuracy of satisfying the linear system equation. It measures the reduction in the residual error throughout the iterations. Initially, the rapid convergence of the residual norm $\|r\|_2$ indicates that the CG method effectively reduces the mismatch between the

approximate solution and the right-hand side. However, as the iterations progress, the convergence of the residual norm $\|r\|_2$ tends to slow down, while the error norm $\|e\|_A$ exhibits smoother convergence behavior. This observation suggests that the CG method achieves a good approximation of the true solution early on but requires more iterations to refine the approximation according to the problem-specific norm defined by matrix A.

## Appendix

The functions mentioned in the report are written for better readability, but the actual functions are implemented in OOPS architecture; file name: "**proj1_oop.py**" for the classes and "**proj1_main.ipynb**" for the main commands.

Conjugate gradient is executed on RWTH HPC cluster due to the resource's shortage on local PC, the SLURM script is also submitted (slurm_submit.py).

To visualize the test matrices a function msr_to_mat has been implemented and the following figure shows the non-zero values in the matrices for CG and GMRES.