Fast Iterative Solvers: Project 1

Jaeyong Jung (359804)

June 21, 2016

1. For the *full* GMRES method: How many Krylov vectors do you need to solve the problem with and without preconditioning?

From the computational experiment, it is observed that it takes **512** Krylov vectors without any preconditioning. On the other hand, only **293** and **162** vectors are necessary when Jacobi and Gauss-Seidel preconditioning are applied.

Options	No preconditioning	Jacobi	Gauss-Seidel
Number of vectors	512	293	162

2. For the restarted GMRES method: Try to find the restart parameter for which the solution is obtained with lowest runtime. (Is this optimal method faster than full GMRES? If yes, why?)

Restart parameters are chosen to be $\frac{3}{4}$, $\frac{2}{3}$, and $\frac{1}{2}$ of the rank of A, 1030. Preconditioning is not done. Average runtime is measured to do more precisely compare.

factor	Full	3/4	2/3	1/2
runtime (s)	0.434	0.423	0.430	0.407

We can see that there is no big difference between runtime of restart method and full GMRES method. The gap between each case's runtime is marginal. Sometimes, runtime of full GMRES method can be less than that of others. Thus, restart method is not always optimal method. Nevertheless, when a size of problem is so big that size of memory is limited, restart method can save memory and while consuming similar runtime.

3. For the preconditioned GMRES method: Plot the relative residual against runtime on a semi-log scale. Which method is the fastest? (Jacobi preconditioning, Gauss-Seidel preconditioning, or no preconditioning.))

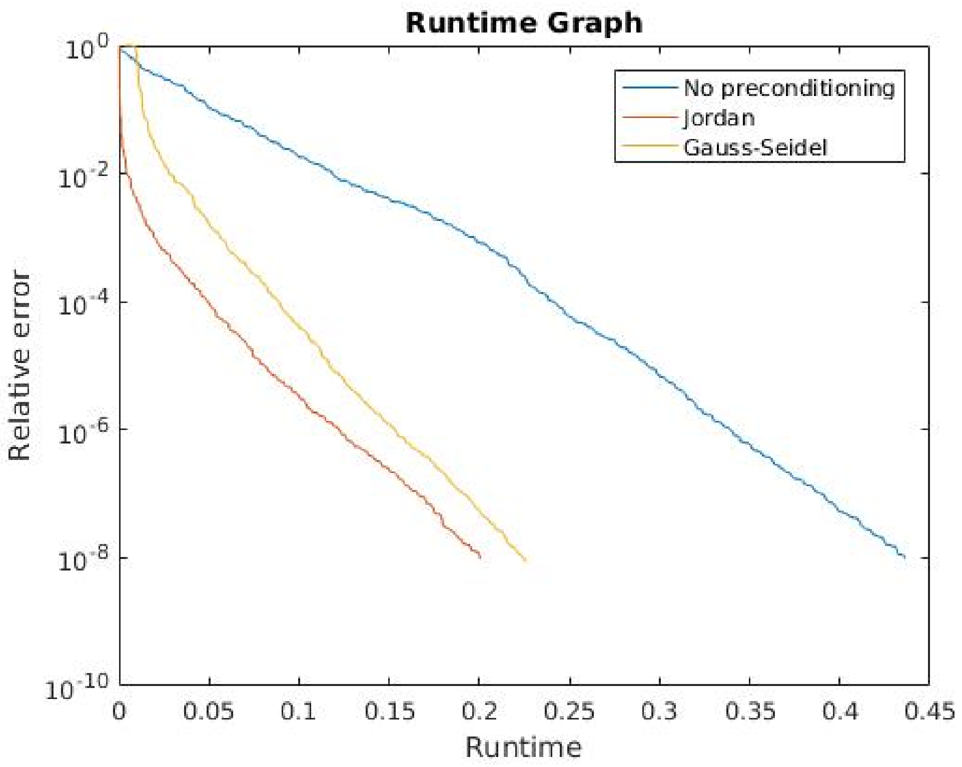

Figure 1:

Obviously, the program is the slowest when there is no preconditioning. The graphs of Jordan and Gauss-Seidel preconditioning seems to take similar amount of time. However, Gauss-Seidel method is supposed to take less time than Jordan graph. The reason is that Gauss-Seidel preconditioning takes 162 steps for convergence but Jacobi takes 293 steps. The runtime of Gauss-Seidel is investigated that 90% percent of total runtime is consumed on iterative procedure. The most time-consuming part of the program is preconditioning part, calculating P^-1w.

The most probable factor for making Gauss-Seidel is utilization of ”Cache Memory”. Terms of Jacobi preconditioner is simply stored sequentially in 1-D array. On the other hand, Gauss-Seidel preconditioner's terms are aligned in column direction, CSC. To elaborate, terms are saved in series like, (1,1), (2,1), (2,2), (3,1), (3,2), (3,3) ...., (M,M-1), (M,M). As applying preconditioner is to calculate $y, P^-1y = x$, back-substitution is done in row direction. As neighbour terms in row directions are far away to each other in memory structure, ”Cache Memory” is poorly utilized. To solve this problem, it is necessary to store preconditioner's terms in CSR format. I have tried to fix it but I could not finish it because of deadline.

Thus, it is hard to compare the result of Jordan and Gauss-Seidel graph because of difference in ”Cache-Memory” utilization. Although, Gauss-Seidel graph is derived with poor cache utilization, it is close to Jordan graph. Therefore, Gauss-Seidel runtime will be faster if Gauss-Seidel preconditioner is implemented in CSR format.

4. For the preconditioned GMRES method: Compare the true absolute resid-

ual $r := b - Ax$ with the residual of preconditioned system. Does the relative residual reduction depend on which residual you monitor? (Why would one rather not monitor the true residual during the computation of the solution?)

I tried to calculate true norm $r_m = ||b - A \times x_m||$. However, the program came up with error message so that I could not finish it. Following statements are based on my guess.

It might be independent to monitor any residual because non-preconditioned and preconditioned problems have the same solution. It may be possible to observe true norm while running the program. However, it takes more computational effort and harms computational performance. If it is independent to choose true norm or norm of preconditioned system, observing g of preconditioned system will be more efficient method.

5. For full GMRES: check the orthogonality of the Krylov vectors! Plot the computed values of (v_1, v_k) against k on a semi-log scale.

Orthogonality of the Krylov vecotrs is proven by calculating dot product between vectors, (v_1, v_k).

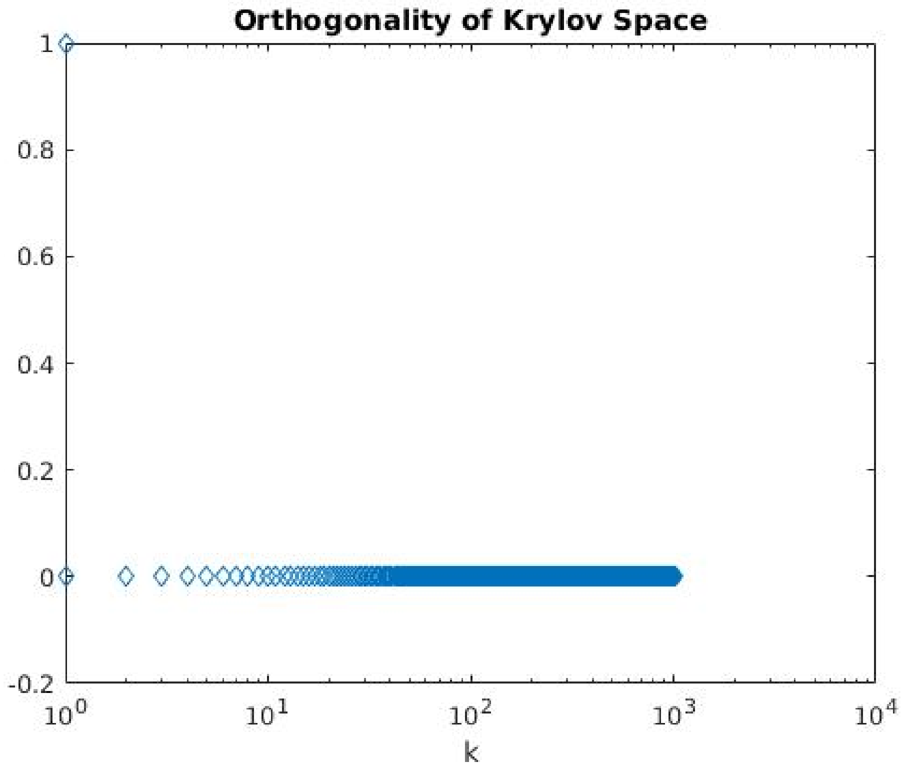

Figure 2: Orthogonality

This graph's x axis is made in log-scale. The starting point on the x-axis is 1. One point on (0, 1) is observed. It is calculated from (v_1, v_1). Otherwise, dot product values are zero, meaning that v_1 is orthogonal to rest of all.

6. For the conjugate gradient method: Plot both the error in A-norm, i.e. $||e||_A = (Ae, e)$, and the residual in standard 2-Norm, i.e. $||r||_2 = (r, r)$ against the iteration index on a semi-log scale. Compare qualitatively the difference in convergence behaviour. (i.e. the difference between the two norms). Is there an explanation for what you observe?

In order to investigate, how those two norms look different from each other, the graph of norms with respect to iteration number is described below, Figure 2. It is seen that norm of residual oscillates. Moreover, it seems to increase in some interval, such as x region from 4×10^4 to 5×10^4. On the other hand, error in A norm is monotonically decreasing when number of iteration increases. That is because Conjugate Gradient Method approximates x_m that makes A-norm of the error to be smaller.

A-norm to decrease monotonically is guaranteed according to the Theorem 1 from the course material.

$$Theorem 1.$$

$$||x_m - x||_A \leqq 2\left(\frac{\sqrt(k) - 1}{\sqrt(k) + 1}\right)^m ||x_0 - x||_A$$

k is condition number of A. since coefficient of$||x_0 - x||_A$ is always smaller than 1, $||x_m - x||_A$ has to decrease. It is known that consecutive residuals are orthonormal but it does not mean monotonically decreasing residual.

$$(r_m, r_{m+1}) = 0$$

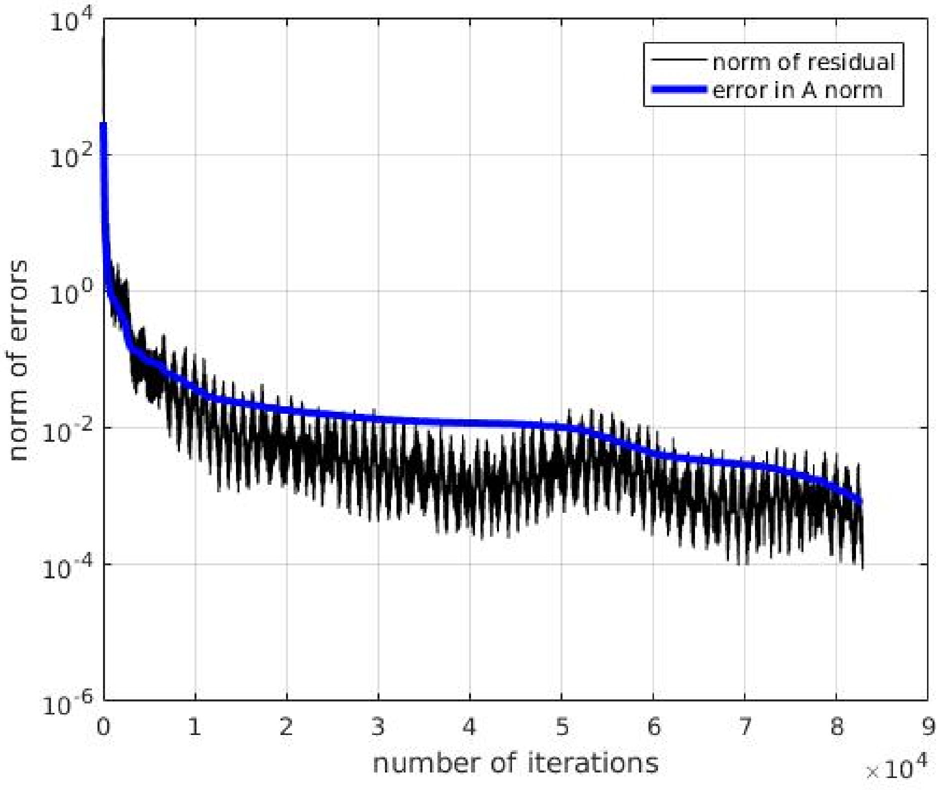

Figure 3: Two types of norms

Implementation of parallel algorithm is done on main function. C code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mmio.h"
#include "operations.h"
#include <time.h>
#include <assert.h>



double tol = 1e-08;



int main(int argc, char *argv[])
{
        int ret_code, precondition = 0, precond_count, flag = 1,
           restart_param;
```

```
MM_typecode matcode;
FILE *f,*out;
int M, N, nz, symm = 0;
int i, j, *I, *J, *Ic, *Jc, *J_csr;
double *val, *Xv, *valc, *y, str_t, end_t, precond_i,
   precond_f, run_i;


if (argc < 2)
{
        fprintf(stderr, "Usage: %s [matrix-market-filename
           ]\n", argv[0]);
        exit(1);
}
else
{
        if ((f = fopen(argv[1], "r")) == NULL)
                exit(1);
}

if (mm_read_banner(f, &matcode) != 0)
{
        printf("Could not process Matrix Market banner.\n"
           );
        exit(1);
}


/*  This is how one can screen matrix types if their
   application */
/*  only supports a subset of the Matrix Market data types
   .      */

if (mm_is_complex(matcode) && mm_is_matrix(matcode) &&
                mm_is_sparse(matcode) )
{
        printf("Sorry, this application does not support "
           );
        printf("Market Market type: [%s]\n",
           mm_typecode_to_str(matcode));
        exit(1);
}

/* find out size of sparse matrix .... */
```

```
    if ((ret_code = mm_read_mtx_crd_size(f, &M, &N, &nz)) !=0)
            exit(1);

    if (argc == 2)
    {
            printf("Preconditioner not defined but still it
               will run. 0:No, 1: Jordan, 2: Gauss-Seidel.\n")
               ;
            precondition = 0;
    }
    else if (argc == 3)
    {
            precondition = atoi(argv[2]);
    }


    printf("Input check is finished. \n");
    str_t=clock();
    /* reserve memory for matrices in CSR format */


    if (mm_is_general(matcode) == 1)
    {
            I = (int *) calloc(nz, sizeof(int));
            Ic = (int *) calloc(nz, sizeof(int));
            J_csr = (int *) calloc((M+1), sizeof(int));
            J = (int *) calloc(nz, sizeof(int));
            Jc = (int *) calloc(nz, sizeof(int));
            val = (double *) calloc(nz, sizeof(double));
            valc = (double *) calloc(nz, sizeof(double));
            Xv = (double *) calloc(N, sizeof(double));
            y = (double *) calloc(N, sizeof(double));
    }

    if (mm_is_symmetric(matcode) == 1)
    {
            I = (int *) calloc(nz, sizeof(int));
            Ic = (int *) calloc(nz, sizeof(int));
            J_csr = (int *) calloc((M+1), sizeof(int));
            J = (int *) calloc(nz, sizeof(int));
            Jc = (int *) calloc(2*nz, sizeof(int));
            val = (double *) calloc(nz, sizeof(double));
            valc = (double *) calloc(2*nz, sizeof(double));
            Xv = (double *) calloc(N, sizeof(double));
            y = (double *) calloc(N, sizeof(double));
```

```
}

// To make it sure, I set large size on below arrays.

int H_size = (M * (M+1))/2;
int *precond_csr = (int *) calloc((M+1), sizeof(int));
int *precond = (int *) calloc(H_size, sizeof(int));
double *precond_val = (double *) calloc(H_size, sizeof(
   double));

// I: X, J: Y reading data
for (i=0; i<nz; i++)
{
        fscanf(f, "%d %d %lg\n", &I[i], &J[i], &val[i]);
        I[i]--;  /* adjust from 1-based to 0-based */
        J[i]--;
}

if (f !=stdin) fclose(f);

// remind that the initial data are arranged in columns
// I will compress J


if (mm_is_symmetric(matcode) == 1)
{
        symm = 1;
}

printf("Converting COO into COO2. \n");
nz = coo_to_csr(Ic, I, Jc, J_csr, J, valc, val, M, nz,
   symm);
printf("Finished converting COO2 into CSR. \n");

if (argc == 3)
{
        printf("By default, restart parameter is set to be
            rank of A\n");
        precondition = atoi(argv[2]);
        restart_param = M;
}
else if (argc == 4)
{
precondition = atoi(argv[2]);
restart_param = atoi(argv[3]);
```

```
}

/* Random numbers in a vector X*/


/* Start GMRES method from now */


/* The real GMRES codes starts from here*/


double *Y = (double *) calloc(M, sizeof(double)); //
   original equ's output vector
double *Y_true = (double *) calloc(M, sizeof(double)); //
   original equ's output vector
double *r = (double *) calloc(M, sizeof(double)); //
   initial residual and residual
double *V = (double *) calloc((M+1)* (M+1), sizeof(double)
   ); // basis matrix, normal form
double *c = (double *) calloc(M, sizeof(double)); //
   cosine
double *s = (double *) calloc(M, sizeof(double)); // sine
double *x = (double *) calloc(N, sizeof(double)); //
   initial guess & solution
double *x_temp = (double *) calloc(N, sizeof(double)); //
   initial guess & solution
double *x_r = (double *) calloc(N, sizeof(double)); //
   Known real solution
double *H = (double *) calloc( H_size, sizeof(double) );
int *H_csc = (int *) calloc ((H_size+1), sizeof (int)); //
    CSC format for H
double *temp_arr = (double *) calloc(M, sizeof(double));
   // temporary array
double *temp_arr2 = (double *) calloc(M, sizeof(double));
   // temporary array
double *temp_arr3 = (double *) calloc(M, sizeof(double));
   // temporary array
double *temp_arr4 = (double *) calloc(M*2, sizeof(double))
   ; // temporary array for calculating real value
double *temp_arr5 = (double *) calloc(M*2, sizeof(double))
   ; // temporary array for calculating real value
double *temp_arr6 = (double *) calloc(M*2, sizeof(double))
   ; // temporary array for calculating real value
double *temp_arr7 = (double *) calloc(M*2, sizeof(double))
   ; // temporary array for calculating real value
```

```
double *g = (double *) calloc((M+1), sizeof(double)); //
    transformed result
double *e1 = (double *) calloc((M+1), sizeof(double)); //
    basis vector
double *w = (double *) calloc(M, sizeof(double));
double *V_c = (double *) calloc(M, sizeof(double));
double *g_norm_save = (double *) calloc(M*2, sizeof(double
    ));
double *true_norm_save = (double *) calloc(M*2, sizeof(
    double));
double true_residual = 0.0223412;
int g_norm_count = 0;
double norm_r;


// Initializing initial guess, known solution and krylov
    space;
for (i=0; i<N; i++)
{
        x[i] = 0;
        x_r[i] = 1;
}
printf("Real solution and initial guess initialized \n");

precond_i = clock();

// calculating preconditioner
if (precondition == 1)
{
        precond_count = Precondition(Jc, J_csr, valc, M,
            nz, precond, precond_csr, precond_val,
            precondition);
        printf("Preconditioning 1 applied\n");
}
else if (precondition == 2)
{
        precond_count = Precondition(I, J, val, M, nz,
            precond, precond_csr, precond_val, precondition
            );
        printf("Preconditioning 2 applied\n");
}


// temp_arr = A*x
if (precondition == 0)
```

```
        {
                matrix_vector(J_csr, Jc, valc, x, temp_arr, M);
        }
        else if (precondition == 1)// temp_arr = precond *
           temp_arr3, temp_arr3 = A*x
        {
                matrix_vector(J_csr, Jc, valc, x, temp_arr3, M);
                matrix_vector(precond_csr, precond, precond_val,
                   temp_arr3, temp_arr, M);
        }
        else if (precondition == 2)// temp_arr = precond *
           temp_arr3, temp_arr3 = A*x
        {
                matrix_vector(J_csr, Jc, valc, x, temp_arr3, M);
                // I use H_inv_g to apply preconditioning
                H_inv_g(precond_val, precond_csr, M, precond_count
                   , temp_arr3, temp_arr);
        }


        // Initializing Y = Ax or Y = P^-1 * A*x; (know solution
           is x[i] = 1)
        if (precondition == 0)
        {
                matrix_vector(J_csr, Jc, valc, x_r, Y, M);
        }
        else if (precondition == 1)
        { // temp_arr3 = A*x_r
                matrix_vector(J_csr, Jc, valc, x_r, temp_arr3, M);
        for (i=0;i<M;i++)
        {
        Y_true[i] = temp_arr3[i];
        }
                matrix_vector(precond_csr, precond, precond_val,
                   temp_arr3, Y, M);
        }
        else if (precondition == 2)
        { // temp_arr3 = A*x_r
                matrix_vector(J_csr, Jc, valc, x_r, temp_arr3, M);
        for (i=0;i<M;i++)
{
Y_true[i] = temp_arr3[i];
}
                // Y = P^-1*(A-x_r)
                H_inv_g(precond_val, precond_csr, M, precond_count
                    , temp_arr3, Y);
        }



        precond_f = clock();


        for (i=0; i<M; i++)
        {
                r[i] = Y[i] - temp_arr[i];
                V[i*N] = r[i];
        }


        norm_r = norm(r, M);
        g[0] = norm_r;
        printf("\nInitial value of norm_r %e \n\n", norm_r);

        for (i=0; i<M; i++)
        {
                V[i*N] = V[i*N] / norm_r;
        }

        // H_csc initialization
        H_csc[0] = 0;
        for (i=1; i <= M; i++)
        {
                H_csc[i] = H_csc[i-1] + i;
        }



// measure runtime
run_i = clock();
        /* Iteration to do GMRES starts now */
        printf("norm_r = %e \n",norm_r);
        double g_norm = norm_r;

        while( ((fabs(g_norm)/norm_r) > tol) && (flag ==1) )
        {
                for (j=0; j< restart_param; j++)
                {
                        g_norm = getkrylov(I, J, val, Jc, J_csr,
                           valc, H, H_csc, V, V_c, w, j, M, N, nz,
                g, c, s, symm, precond, precond_csr,
                   precond_val, precondition);
            g_norm_save[g_norm_count] = fabs(g_norm/
                   norm_r);

            if (precondition >0) // saving true norm
                   for comparision
            {
            H_inv_g(H, H_csc, (g_norm_count+1), (
                   g_norm_count+1)*(g_norm_count+2)/2, g,
                   temp_arr4);
            dense_matrix_vector(V, (g_norm_count+1), (
                   g_norm_count+1), temp_arr4, temp_arr5);

            for (i=0; i<(g_norm_count+1); i++)
            {
                    x_temp[i] = x[i] + temp_arr5[i];
            }
            // temp_arr 4= true A*x
            matrix_vector(J_csr, Jc, valc, x_temp,
                   temp_arr6, M);
            for (i=0; i<(g_norm_count+1); i++)
            {
                    temp_arr7[i] = Y_true[i] -
                       temp_arr6[i];
            }

                    true_norm_save[g_norm_count] =
                       norm(temp_arr7,(g_norm_count+1)
                       );
            }

            g_norm_count++;

            if (g_norm_save[g_norm_count-1] < tol)
            {
                    flag = 0;
                    printf("I am breaking the loop
                       since the relative norm is %e \
                       n", g_norm_save[g_norm_count
                       -1]);
                    break;
            }
    }
```

```
// We go for one more set of iterations.
if ((j == restart_param) && (fabs(g_norm/norm_r) >
    tol) && (flag == 1))
{
        printf("\n It is not yet converged \n");
           // temp_arr2 = V*R^-1*g
        H_inv_g(H, H_csc, g_norm_count, (
           g_norm_count)*(g_norm_count+1)/2, g,
           temp_arr);
        dense_matrix_vector(V, M, N, temp_arr,
           temp_arr2);

        for (i=0; i<N; i++)
        {
                x[i] = x[i] + temp_arr2[i];
        }

        // calculate Ax firstly
        if (precondition == 0)
        {
                matrix_vector(J_csr, Jc, valc, x,
                   temp_arr, M);
        }
        else if (precondition ==1)
        { // temp_arr3 = A*x
                matrix_vector(J_csr, Jc, valc, x,
                   temp_arr3, M);
                matrix_vector(precond_csr, precond
                   , precond_val, temp_arr3,
                   temp_arr, M);
        }
        else if (precondition ==2)
        {
                matrix_vector(J_csr, Jc, valc, x,
                   temp_arr3, M);
                H_inv_g(precond_val, precond_csr,
                   M, precond_count, temp_arr3,
                   temp_arr);
        }
        // calculate residual again
        for (i=0; i<M; i++)
        {
                r[i] = Y[i] - temp_arr[i];
                V[i*N] = r[i];
                }

                norm_r = norm(r, M);
                g[0] = norm_r;

                for (i=0; i<M; i++)
                {
                        V[i*N] = V[i*N] / norm_r;
                }
        } // end of re-initializing x, r for restarted
           method.
}// end of while loop

H_size = ((j+1) * (j+2))/2;

printf("g_norm %e\n",g_norm);
H_inv_g(H, H_csc, j+1, H_size, g, temp_arr);

dense_matrix_vector(V, M, N, temp_arr, x);
end_t=clock();
printf("Total runtime is %e \n", (end_t-str_t)/
   CLOCKS_PER_SEC);
printf("Preconditioner is %d. 0:No, 1: Jordan, 2: Gauss-
   Seidel.\n", precondition);
printf("Preconditioning time is %e \n", (precond_f-
   precond_i)/CLOCKS_PER_SEC);
printf("Pure runtime is %e \n", (end_t-run_i)/
   CLOCKS_PER_SEC);
printf("Restart parameter is %d \n\n",restart_param);
printf("true residual norm = %e\n", true_residual);

free(I);
free(J_csr);
free(J);
free(val);
free(Xv);
free(y);
free(Ic);
free(Jc);
free(valc);
free(precond);
free(precond_csr);
free(precond_val);

free(Y);
```

```
        free(Y_true);
        free(r);
        free(V);
        free(c);
        free(H);
        free(H_csc);
        free(s);
        free(x);
        free(x_temp);
        free(x_r);
        free(temp_arr);
        free(temp_arr2);
        free(temp_arr3);
        free(temp_arr4);
        free(temp_arr5);
        free(temp_arr6);
        free(temp_arr7);
        free(g);
        free(e1);
        free(w);
        free(V_c);
        free(g_norm_save);
        free(true_norm_save);

        return 0;
}
```