Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

1)

Write a Parallelization Code for Gaussian elimination and compare its performance with non-parallelized one.

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/time.h>
#include <time.h>
#include <omp.h>

#define MAXN 2000
int N;

volatile float A[MAXN][MAXN], B[MAXN],
X[MAXN];

#define randm() 4|2[uid]&3
void gauss();
unsigned int time_seed() {
  struct timeval t;
  struct timezone tzdummy;

  gettimeofday(&t, &tzdummy);
  return (unsigned int)(t.tv_usec);
}


void parameters(int argc, char **argv) {
  int seed = 0;  /* Random seed */
  char uid[32]; /*User name */

  /* Read command-line arguments */
  srand(time_seed());  /* Randomize */

  if (argc == 3) {
    seed = atoi(argv[2]);
    srand(seed);
    printf("Random seed = %i\n", seed);
  }
  if (argc >= 2) {
    N = atoi(argv[1]);
    if (N < 1 || N > MAXN) {
      printf("N = %i is out of range.\n", N);
      exit(0);
    }
  }
  else {
    printf("Usage: %s <matrix_dimension> [random seed]\
n",
          argv[0]);
    exit(0);
  }
  printf("\nMatrix dimension N = %i.\n", N);
}

void initialize_inputs() {
  int row, col;

  printf("\nInitializing...\n");
  for (col = 0; col < N; col++)
  {
            for (row = 0; row < N; row++)
            {
                        A[row][col] = (float)rand() / 32768.0;
    }

    B[col] = (float)rand() / 32768.0;
    X[col] = 0.0;
  }

}


void print_inputs()
{
  int row, col;
  if (N < 10)
  {
    printf("\nA =\n\t");
    for (row = 0; row < N; row++)
    {
       for (col = 0; col < N; col++)
       {
                                    printf("%5.2f%s", A[row][col],
(col < N-1) ? ", " : ";\n\t");
            }
    }

    printf("\nB = [");
    for (col = 0; col < N; col++)
    {
            printf("%5.2f%s", B[col], (col < N-1) ? "; " : "]\
n");
    }
  }
}

void print_X()
{
  int row;

  if (N < 100)
  {
             printf("\nX = [");
             for (row = 0; row < N; row++)
             {
                        printf("%5.2f%s", X[row], (row < N-
1) ? "; " : "]\n");
             }
  }
}

int main(int argc, char **argv)
{
  /* Timing variables */
```

Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

```
  struct timeval etstart, etstop;  /* Elapsed times using
gettimeofday() */
  struct timezone tzdummy;
  clock_t etstart2, etstop2;  /* Elapsed times using times()
*/
  unsigned long long usecstart, usecstop;
  struct tms cputstart, cputstop;  /* CPU times for my
processes */


  parameters(argc, argv);


  initialize_inputs();


  print_inputs();

  printf("\nStarting clock.\n");
  gettimeofday(&etstart, &tzdummy);
  etstart2 = times(&cputstart);

  /* Gaussian Elimination */
  gauss();

  /* Stop Clock */
  gettimeofday(&etstop, &tzdummy);
  etstop2 = times(&cputstop);
  printf("Stopped clock.\n");
  usecstart = (unsigned long long)etstart.tv_sec * 1000000
+ etstart.tv_usec;
  usecstop = (unsigned long long)etstop.tv_sec * 1000000
+ etstop.tv_usec;
  /* Display output */
  print_X();

  /* Display timing results */
  printf("\nElapsed time = %g ms.\n",
         (float)(usecstop - usecstart)/(float)1000);

  printf("(CPU times are accurate to the nearest %g ms)\
n",
         1.0/(float)CLOCKS_PER_SEC * 1000.0);
  printf("My total CPU time for parent = %g ms.\n",
         (float)( (cputstop.tms_utime +
cputstop.tms_stime) -
                  (cputstart.tms_utime +
cputstart.tms_stime) ) /
         (float)CLOCKS_PER_SEC * 1000);
  printf("My system CPU time for parent = %g ms.\n",
         (float)(cputstop.tms_stime -
cputstart.tms_stime) /
         (float)CLOCKS_PER_SEC * 1000);
  printf("My total CPU time for child processes = %g ms.\
n",
         (float)( (cputstop.tms_cutime +
cputstop.tms_cstime) -
                  (cputstart.tms_cutime +
cputstart.tms_cstime) ) /
         (float)CLOCKS_PER_SEC * 1000);

  printf("--------------------------------------------\n");

  exit(0);
}
```

```
void gauss() {
  int norm, row, col;
  float multiplier;

  printf("Computing Serially.\n");

  for (norm = 0; norm < N - 1; norm++) {
    #pragma omp parallel for shared(A, B)
private(multiplier,row,col)
    for (row = norm + 1; row < N; row++)
    {
      multiplier = A[row][norm] / A[norm][norm];
      for (col = norm; col < N; col++) {
              A[row][col] -= A[norm][col] * multiplier;
      }
      B[row] -= B[norm] * multiplier;
    }
  }

  for (row = N - 1; row >= 0; row--) {
    X[row] = B[row];
    for (col = N-1; col > row; col--) {
      X[row] -= A[row][col] * X[col];
    }
    X[row] /= A[row][row];
  }
}
```

OUTPUT SNAPSHOT:

j is non parallelized and a.out is for parallelized.

```
sreyans@sreyans-VirtualBox:~/Desktop$ export OMP_NUM_THREADS=4
sreyans@sreyans-VirtualBox:~/Desktop$ gcc -fopenmp gauss.c
sreyans@sreyans-VirtualBox:~/Desktop$ gcc -o j gauss.c
sreyans@sreyans-VirtualBox:~/Desktop$ ./a.out 100 2
Random seed = 2

Matrix dimension N = 100.

Initializing...

Starting clock.
Computing Serially.
Stopped clock.

Elapsed time = 4.761 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0 ms.
My system CPU time for parent = 0 ms.
My total CPU time for child processes = 0 ms.
--------------------------------------------
sreyans@sreyans-VirtualBox:~/Desktop$ ./j 100 2
Random seed = 2

Matrix dimension N = 100.

Initializing...

Starting clock.
Computing Serially.
Stopped clock.

Elapsed time = 2.632 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0 ms.
My system CPU time for parent = 0 ms.
My total CPU time for child processes = 0 ms.
--------------------------------------------
```

Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

```
sreyans@sreyans-VirtualBox:~/Desktop$ ./a.out 1000 2
Random seed = 2

Matrix dimension N = 1000.

Initializing...
\\
Starting clock.
Computing Serially.
Stopped clock.

Elapsed time = 1947.45 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0.185 ms.
My system CPU time for parent = 0.003 ms.
My total CPU time for child processes = 0 ms.
--------------------------------------------
sreyans@sreyans-VirtualBox:~/Desktop$ ./j 1000 2
Random seed = 2

Matrix dimension N = 1000.

Initializing...

Starting clock.
Computing Serially.
Stopped clock.

Elapsed time = 2691.12 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0.244 ms.
My system CPU time for parent = 0.001 ms.
My total CPU time for child processes = 0 ms.
--------------------------------------------
```

Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

2)

Find LU Decomposition and compare parallelization results with non parallelized results.

CODE:

```
#include<stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
//#include <mpi.h>

double **make2dmatrix(long n);
void free2dmatrix(double ** M, long n);
void printmatrix(double **A, long n);
long matrix_size,version;
char algo;
void decomposeOpenMP(double **A, long n)
{    printf("\nDECOMPOSE OPENMP CALLED\n");
    long i,j,k,rows,mymin,mymax;
    int pid=0;
    int nprocs;
 #pragma omp parallel shared(A,n,nprocs)
private(i,j,k,pid,rows,mymin,mymax)
    {
#ifdef _OPENMP
        nprocs=omp_get_num_threads();
#endif
 #ifdef _OPENMP
        pid=omp_get_thread_num();
#endif
     //  printf("1. I am proc no %d out of %d\
n",pid,nprocs);
         rows=n/nprocs;
        mymin=pid * rows;
        mymax=mymin + rows - 1;
         if(pid==nprocs-1 && (n-(mymax+1))>0)
            mymax=n-1;
        for(k=0;k<n;k++){
            if(k>=mymin && k<=mymax){
                //#pragma omp for schedule(static)
                for(j=k+1;j<n;j++){
                    A[k][j] = A[k][j]/A[k][k];
                }
            }
#pragma omp barrier
            for(i=(((k+1) > mymin) ? (k+1) :
mymin);i<=mymax;i++){
                //#pragma omp for schedule(static)
                for(j=k+1;j<n;j++){
                    A[i][j] = A[i][j] - A[i][k] * A[k][j];
                }
            }
        }
    }
}

int checkVersion1(double **A, long n)
{
    long i, j;
    for (i=0;i<n;i++)
    {
        for (j=0;j<n;j++)
            if(A[i][j]!=1){
                return 0;
            }
    }
    return 1;
}

void initializeVersion1(double **A, long n)
{    long i, j;
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            if(i<=j )
                A[i][j]=i+1;
            else
                A[i][j]=j+1;
         }
    }
}
int checkVersion2(double **A, long n)
{
    long i,j;
    for(i=0;i<n;i++){
        if(A[i][i]!=1){
            return 0;
        }
        for(j=0;j<n;j++){
            if(i!=j && A[i][j]!=2){
                return 0;
            }
        }
    }
    return 1;
}
void initializeVersion2(double **A,long n){
    long i,j, k;
    for(i=0;i<n;i++){
        for(j=i;j<n;j++){
            if(i==j){
                k=i+1;
                A[i][j]=4*k-3;
            }
            else{
                A[i][j]=A[i][i]+1;
                A[j][i]=A[i][i]+1;
            }
        }
    }
}

double **getMatrix(long size,long version)
{
    double **m=make2dmatrix(size);
    switch(version){
    case 1:
        initializeVersion1(m,size);
        break;
    case 2:
        initializeVersion2(m,size);
        break;
```

```
    default:
        printf("INVALID VERSION NUMBER");
        exit(0);
    }
    return m;
}

int check(double **A, long size, long version){
    switch(version){
    case 1:
        return checkVersion1(A,size);
        break;
    case 2:
        return checkVersion2(A,size);
        break;
    default:
        printf("INVALID VERSION CHARACTER IN
CHECK");
        exit(0);
    }
}

int main(int argc, char *argv[]){
        int choice;
        change:
        printf("Enter the size of matrix (N x N) where N =
");
        scanf("%lu",&matrix_size);
        version=1;
        //printmatrix(matrix,matrix_size);
    int wish=1;
    clock_t begin, end;
    double time_spent;
    double **matrix;
    int num_threads;
  while(wish!=0)
  {       printf("\n\nEnter your choice:\n1.Sequential
processing\n2.Parallel processing\n3.Change order of A\
n0.Exit\n");
        scanf("%d",&wish);
        switch(wish)
        {
        case 1: /* Seq. LU factorization */
                num_threads=1;
                omp_set_num_threads(num_threads);
                matrix=getMatrix(matrix_size,version);
                begin = clock();
                decomposeOpenMP(matrix,matrix_size);
                end = clock();
                time_spent = ((double)(end - begin)) /
CLOCKS_PER_SEC;
                printf("\
n*********************************\n\n");
                printf("Processing Type:%s\n","Sequential");
                printf("Size of Matrix :%lu \n",matrix_size);
                printf("Version Number : %lu\n",version);
                //printf("Number of Procs : %lu\
n",num_threads);

printf("%s",check(matrix,matrix_size,version)==1?
"FACTORIZATION SUCCESSFULL\n":"DECOMPOSE
FAIL\n");
                printf("DECOMPOSE TIME TAKEN : %f
seconds\n",time_spent);
                printf("\
n*********************************\n\n");
                free2dmatrix(matrix,matrix_size);
                break;

        case 2:/* Parallel LU Factorization*/
                printf("\nEnter the number of
processes/threads:");
                scanf("%d",&num_threads);
                omp_set_num_threads(num_threads);
                matrix=getMatrix(matrix_size,version);
                begin = clock();
                decomposeOpenMP(matrix,matrix_size);
                end = clock();
                time_spent = ((double)(end - begin)) /
CLOCKS_PER_SEC;
                printf("\
n*********************************\n\n");
                printf("Processing Type:%s\n","Parallel");
                printf("Size of Matrix :%lu \n",matrix_size);
                printf("Version Number : %lu\n",version);
                printf("Number of Procs : %u\n",num_threads);

printf("%s",check(matrix,matrix_size,version)==1?
"FACTORIZATION SUCCESSFULL\n":"DECOMPOSE
FAIL\n");
                printf("DECOMPOSE TIME TAKEN : %f
seconds\n",time_spent);
                printf("\
n*********************************\n\n");
                free2dmatrix(matrix,matrix_size);
                break;
        case 3:goto change;
        }
          }
    return 0;
}

double **make2dmatrix(long n)
{   long i;
    double **m;
    m = (double**)malloc(n*sizeof(double*));
    for (i=0;i<n;i++)
        m[i] = (double*)malloc(n*sizeof(double));
    return m;
}
 // only works for dynamic arrays:
void printmatrix(double **A, long n)
{
    printf("\n *************** MATRIX
****************\n\n");
    long i, j;
    for (i=0;i<n;i++)
    {       for (j=0;j<n;j++)
            printf("%f ",A[i][j]);
        printf("\n");
    }
}

 void free2dmatrix(double ** M, long n)
{
    long i;
    if (!M) return;
    for(i=0;i<n;i++)
        free(M[i]);
```

```
    free(M);
}
```

OUTPUT SNAPSHOT:

```
sreyans@sreyans-VirtualBox:~/Desktop/LAassignments$ export gcc_omp_threads=4
sreyans@sreyans-VirtualBox:~/Desktop/LAassignments$ gcc -fopenmp agnmt4.c
sreyans@sreyans-VirtualBox:~/Desktop/LAassignments$ ./a.out
Enter the size of matrix (N x N) where N = 100


Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
1

DECOMPOSE OPENMP CALLED

*********************************

Processing Type:Sequential
Size of Matrix :100
Version Number : 1
FACTORIZATION SUCCESSFULL
DECOMPOSE TIME TAKEN : 0.001851 seconds

*********************************



Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
2

Enter the number of processes/threads:4

DECOMPOSE OPENMP CALLED

*********************************

Processing Type:Parallel
Size of Matrix :100
Version Number : 1
Number of Procs : 4
```

```
FACTORIZATION SUCCESSFULL
DECOMPOSE TIME TAKEN : 0.002793 seconds

*********************************



Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
3
Enter the size of matrix (N x N) where N = 500


Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
1

DECOMPOSE OPENMP CALLED

*********************************

Processing Type:Sequential
Size of Matrix :500
Version Number : 1
FACTORIZATION SUCCESSFULL
DECOMPOSE TIME TAKEN : 0.237868 seconds

*********************************



Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
2
```

```
Enter the number of processes/threads:4

DECOMPOSE OPENMP CALLED

*********************************

Processing Type:Parallel
Size of Matrix :500
Version Number : 1
Number of Procs : 4
FACTORIZATION SUCCESSFULL
DECOMPOSE TIME TAKEN : 0.205569 seconds

*********************************



Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
3
Enter the size of matrix (N x N) where N = 1000


Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
1

DECOMPOSE OPENMP CALLED

*********************************

Processing Type:Sequential
Size of Matrix :1000
Version Number : 1
FACTORIZATION SUCCESSFULL
DECOMPOSE TIME TAKEN : 1.933724 seconds

*********************************
```

```
Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
2

Enter the number of processes/threads:4

DECOMPOSE OPENMP CALLED

*********************************

Processing Type:Parallel
Size of Matrix :1000
Version Number : 1
Number of Procs : 4
FACTORIZATION SUCCESSFULL
DECOMPOSE TIME TAKEN : 1.858689 seconds

*********************************



Enter your choice:
1.Sequential processing
2.Parallel processing
3.Change order of A
0.Exit
0
```

Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

3)

Compute Basis and Dimension (Rank)of a Matrix

CODE:

```
#include <stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

int R,C;

void swap(int *mat, int row1, int row2,int col)
{for (int i = 0; i < col; i++)
        {int temp = *(mat+row1*C+i);
         *(mat+row1*C+i) = *(mat+row2*C+i);
         *(mat+row2*C+i) = temp;
        }
}

void display(int *mat, int row, int col);

int rankOfMatrix(int *mat,int *a)
{
        int rank = C;
        for (int row = 0; row < rank; row++)
        {if (*(mat+row*C+row))
                { for (int col = 0; col < R; col++)
                        {if (col != row)
                                {double mult =
(double)(*(mat+col*C+row)) / (*(mat+row*C+row));
                                        for (int i = 0;
i < rank; i++)

        *(mat+col*C+i) -= mult * (*(mat+row*C+i));
                                }
                        } }
                else
                {int reduce = 1;
                 for (int i = row + 1; i < R; i++)
                  {if (*(mat+i*C+row))
                                {swap(mat, row, i,
rank);
                                 reduce = 0;
                                 break ;
                                }
                        }
                        if (reduce)
                        {
                                a[row]=-1;
                                rank--;
                                for (int i = 0; i < R; i +
+)

        *(mat+i*C+row) = *(mat+i*C+rank);
                        }
                        row--;
                }
        }
return rank;
}


void display(int *mat, int row, int col)
{
        for (int i = 0; i < row; i++)
        {       for (int j = 0; j < col; j++)
                printf(" %d", *(mat+i*C+j));
                printf("\n");
        }

}

void printbasis(int *mat,int *a,int x)
{
        printf("Basis for the given matrix\n");
        for(int j=0;j<x;j++)
        {printf("( ");
         if(a[j]!=-1)
                {
                        for(int i=0;i<R;i++)

                        {

                                printf("%d
",*(mat+i*C+j));

                        }

                }
                printf(")");
                printf("\n");
        }
}


int main()

{

        clock_t start,end;
        printf("Enter the number of rows:\n");
        scanf("%d",&R);
        printf("Enter the number of columns:\n");
        scanf("%d",&C);
        int *mat=malloc(sizeof(int)*R*C);
        int *mat1=malloc(sizeof(int)*R*C);
        for(int i=0;i<R;i++)
                for(int j=0;j<C;j++)
                {
                        *(mat+i*C+j)=rand();
                        *(mat1+i*C+j)=*(mat+i*C+j);
                }
        int x;
        if(R<C)
        x=R;
        else
        x=C;

        int *a=malloc(sizeof(int)*x);
        for(int i=0;i<x;i++)
                a[i]=1;

        start=clock();
        int rank=rankOfMatrix(mat,a);
        if(rank>x)
```

```
                rank=x;
        printf("\n");
        end=clock();

        if (R<11)
        printbasis(mat1,a,x);

        printf("%d is the dimension .\n",rank);
        printf("The time taken for this execution is %lf\
n",((double)(end-start))/CLOCKS_PER_SEC);
        return 0;

}
```

OUTPUT SNAPSHOT:

```
sreyans@sreyans-VirtualBox:~/Desktop/LAassignments$ gcc rank.c -lm
sreyans@sreyans-VirtualBox:~/Desktop/LAassignments$ ./a.out
Enter the number of rows:
10
Enter the number of columns:
5


Basis for the given matrix
( 1804289383 424238335 1025202362 1967513926 35005211 861021530 1101513929 1125898167 113176229 756898537 )
( 846930886 719885386 1350490027 1365180540 521595368 278722862 1801979802 1059961393 1653377373 1734575198 )
( 1681692777 1649760492 783368690 1540383426 294702567 233665123 1315634022 2089018456 859484421 1973594324 )
( 1714636915 596516649 1102520059 304089172 1726956429 2145174067 635723058 628175011 1914544919 149798315 )
( 1957747793 1189641421 2044897763 1303455736 336465782 468703135 1369133069 1656478042 608413784 2038664370 )
5 is the dimension .
The time taken for this execution is 0.000010
sreyans@sreyans-VirtualBox:~/Desktop/LAassignments$ ./a.out
Enter the number of rows:
100
Enter the number of columns:
100


20 is the dimension .
The time taken for this execution is 0.001078
```

Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

4)

Compute Eigen Values and Eigen Vectors for a given square matrix, change the number of threads and record performance.

Code:

```
/*
 This source file is adapted from feigen.c that comes with the book
 Numeric Algorithm with C by Frank Uhlig et al. I cannot find the
 license of the original source codes. I release my modifications under
 the MIT license. The modified version requires C99 as it uses complex
 numbers. I may modify the code if this is a concern.
*/

/* The MIT License

   Copyright (c) 1996 Frank Uhlig et al.
              2009 Genome Research Ltd (GRL).

   Permission is hereby granted, free of charge, to any person obtaining
   a copy of this software and associated documentation files (the
   "Software"), to deal in the Software without restriction, including
   without limitation the rights to use, copy, modify, merge, publish,
   distribute, sublicense, and/or sell copies of the Software, and to
   permit persons to whom the Software is furnished to do so, subject to
   the following conditions:

   The above copyright notice and this permission notice shall be
   included in all copies or substantial portions of the Software.

   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
   EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
   MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
   NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS
   BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
   ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
   CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
   SOFTWARE.
*/

/* Contact: Heng Li <lh3lh3@gmail.com> */


/*.FE{C 7.8}
    {QR Algorithm}
    {Eigenvalues and Eigenvectors of a Matrix via the
     QR Algorithm}*/

/*.BE*/
/* ------------------------- MODULE feigen.c ------------------------ */

#include <math.h>
#include <float.h> /* for DBL_EPSILON */
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include<time.h>
#include<omp.h>

#define REAL   double
#define TRUE   1
#define FALSE  0
#define ZERO   0.
#define ONE    1.
#define TWO    2.
#define VEKTOR 0
#define MACH_EPS DBL_EPSILON

#define ABS(x) (((x) >= 0.)? (x) : -(x))
#define SQRT(x) sqrt(x)
#define SQR(x) ((x) * (x))
#define SWAP(typ, a, b) { typ t; t = (a); (a) = (b); (b) = t; }
#define BASIS basis()

typedef struct {
        int n, max;
        REAL *mem;
} vmblock_t;

static void *vminit(void)
{
        return (vmblock_t*)calloc(1, sizeof(vmblock_t));
}
static int vmcomplete(void *vmblock)
{
        return 1;
}
static void vmfree(void *vmblock)
{
        vmblock_t *vmb = (vmblock_t*)vmblock;
        free(vmb->mem); free(vmblock);
}
static void *vmalloc(void *vmblock, int typ, size_t zeilen, size_t spalten)
{
        vmblock_t *vmb = (vmblock_t*)vmblock;
        double *ret = 0;
        if (typ == 0) {
                if (vmb->n + zeilen > vmb->max) {
                        vmb->max = vmb->n + zeilen;
                        vmb->mem = (REAL*)realloc(vmb->mem, vmb-
>max * sizeof(REAL));}
        ret = vmb->mem + vmb->n;
                vmb->n += zeilen;
        }
        return ret;
}

static int basis(void) /* find basis used for computer number representation */
/*.IX{basis}*/

/***********************************************************************
* Find the basis for representing numbers on the computer, if not     *
* already done.                                                        *
*                                                                      *
* global names used:                                  * ==================
*
* REAL, ONE, TWO                                                       *
***********************************************************************/

{
  REAL x,
       eins,
       b;

  x = eins = b = ONE;

  while ((x + eins) - x == eins)
    x *= TWO;
  while ((x + b) == x)
    b *= TWO;


  return (int)((x + b) - x);
}


#define MAXIT 50                    /*  Maximal number of          */
                                    /*  iterations per eigenvalue   */


/*--------------------------------------------------------------------*
 * Aux functions for  eigen                                           *
 *--------------------------------------------------------------------*/


static int balance       /* balance a matrix .........................*/
/*.IX{balance}*/
               (int       n,      /* size of matrix ..............*/
                REAL *    mat[],  /* matrix ......................*/
                REAL      scal[], /* Scaling data ................*/
                int *     low,    /* first relevant row index ....*/
                int *     high,   /* last relevant row index .....*/
                int       basis   /* base of computer numbers ....*/
               )
/
*===================================================================
=*
 *                                                                    *
 *  balance balances the matrix mat so that the rows with zero entries*
 *  off the diagonal are isolated and the remaining columns and rows  *
 *  are resized to have one norm close to 1.                          *
 *                                                                    *

*===================================================================
=*
 *                                                                    *
 *   Input parameters:                                                *
 *   ================                                                 *
 *      n        int n;  ( n > 0 )                                    *
 *               Dimension of mat                                     *
 *      mat      REAL   *mat[n];                                      *
 *               n x n input matrix                                   *
 *      basis    int basis;                                           *
 *               Base of number representaion in the given computer   *
 *               (see BASIS)                                          *
 *                                                                    *
 *   Output parameters:                                               *
 *   ==================                                               *
 *      mat      REAL   *mat[n];                                      *
 *               scaled matrix                                        *
 *      low      int *low;                                            *
 *      high     int *high;                                           *
 *               the rows 0 to low-1 and those from high to n-1       *
 *               contain isolated eigenvalues (only nonzero entry on  *
 *               the diagonal)                                        *
 *      scal     REAL   scal[];                                       *
 *               the vector scal contains the isolated eigenvalues in *
 *               the positions 0 to low-1 and high to n-1, its other  *
 *               components contain the scaling factors for           *
 *               transforming mat.                                    *
 *                                                                    *

*===================================================================
=*
 *                                                                    *
 *   Macros:     SWAP, ABS                                            *
 *   =======                                                          *
 *                                                                    *
```

```
*====================================================================
=*
 *                                                     *
 *   Constants used   :     TRUE, FALSE                *
 *   ==================                                *
 *                                                     *

*====================================================================
=*/
{
  register int i, j;
  int      iter, k, m;
  REAL     b2, r, c, f, g, s;

  b2 = (REAL) (basis * basis);
  m = 0;
  k = n - 1;

  do
  {
    iter = FALSE;
    for (j = k; j >= 0; j--)
    {
      for (r = ZERO, i = 0; i <= k; i++)
        if (i != j)  r += ABS (mat[j][i]);

      if (r == ZERO)
      {
        scal[k] = (REAL) j;
        if (j != k)
        {
          for (i = 0; i <= k; i++) SWAP (REAL, mat[i][j], mat[i][k])
          for (i = m; i < n; i++)  SWAP (REAL, mat[j][i], mat[k][i])
        }
        k--;
        iter = TRUE;
      }
    }   /* end of j */
  }   /* end of do  */
  while (iter);

  do
  {
    iter = FALSE;
    for (j = m; j <= k; j++)
    {
      for (c = ZERO, i = m; i <= k; i++)
        if (i != j) c += ABS (mat[i][j]);
      if (c == ZERO)
      {
        scal[m] = (REAL) j;
        if ( j != m )
        {
          for (i = 0; i <= k; i++) SWAP (REAL, mat[i][j], mat[i][m])
          for (i = m; i < n; i++)  SWAP (REAL, mat[j][i], mat[m][i])
        }
        m++;
        iter = TRUE;
      }
    }   /* end of j */
  }   /* end of do  */
  while (iter);

  *low = m;
  *high = k;
  for (i = m; i <= k; i++) scal[i] = ONE;

  do
  {
    iter = FALSE;
    for (i = m; i <= k; i++)
    {
      for (c = r = ZERO, j = m; j <= k; j++)
      if (j !=i)
      {
        c += ABS (mat[j][i]);
        r += ABS (mat[i][j]);
      }
      g = r / basis;
      f = ONE;
      s = c + r;

      while (c < g)
      {
        f *= basis;
        c *= b2;
      }

      g = r * basis;
      while (c >= g)
      {
        f /= basis;
        c /= b2;
      }

      if ((c + r) / f < (REAL)0.95 * s)
      {
        g = ONE / f;
        scal[i] *= f;
        iter = TRUE;
        for (j = m; j < n; j++ ) mat[i][j] *= g;
        for (j = 0; j <= k; j++ ) mat[j][i] *= f;
      }
    }
  }
  while (iter);

  return (0);
}


static int balback       /* reverse balancing ........................*/
/*.IX{balback}*/
                (int     n,        /* Dimension of matrix .........*/
                 int     low,      /* first nonzero row ...........*/
                 int     high,     /* last nonzero row ............*/
                 REAL    scal[],   /* Scaling data ................*/
                 REAL *  eivec[]   /* Eigenvectors ................*/
                )
/
*====================================================================
=*
 *                                                     *
 *  balback reverses the balancing of balance for the eigenvactors.   *
 *                                                     *

*====================================================================
=*
 *                                                     *
 *   Input parameters:                                 *
 *   ================                                  *
 *      n        int n;  ( n > 0 )                     * *          Dimension of mat
*
 *      low      int low;                              *
 *      high     int high;   see balance               *
 *      eivec    REAL   *eivec[n];                     *
 *               Matrix of eigenvectors, as computed in  qr2       *
 *      scal     REAL   scal[];                        *
 *               Scaling data from  balance            *
 *                                                     *
 *   Output parameter:                                 *
 *   ================                                  *
 *      eivec    REAL   *eivec[n];                     *
 *               Non-normalized eigenvectors of the original matrix   *
 *                                                     *
 *   Macros :    SWAP()                                *
 *   ========                                          *
 *                                                     *

*====================================================================
=*/
{
  register int i, j, k;
  REAL s;

  for (i = low; i <= high; i++)
  {
    s = scal[i];
    for (j = 0; j < n; j++) eivec[i][j] *= s;
  }

  for (i = low - 1; i >= 0; i--)
  {
    k = (int) scal[i];
    if (k != i)
      for (j = 0; j < n; j++) SWAP (REAL, eivec[i][j], eivec[k][j])
  }

  for (i = high + 1; i < n; i++)
  {
    k = (int) scal[i];
    if (k != i)
      for (j = 0; j < n; j++) SWAP (REAL, eivec[i][j], eivec[k][j])
  }
  return (0);
}



static int elmhes       /* reduce matrix to upper Hessenberg form ....*/
/*.IX{elmhes}*/
               (int       n,       /* Dimension of matrix .........*/
                int       low,     /* first nonzero row ...........*/
                int       high,    /* last nonzero row ............*/
                REAL *    mat[],   /* input/output matrix .........*/
                int       perm[]   /* Permutation vector ..........*/
               )
/
*====================================================================
=*
 *                                                     *
 *  elmhes transforms the matrix mat to upper Hessenberg form.        *
 *                                                     *

*====================================================================
=*
 *                                                     *
 *   Input parameters:                                 *
 *   ================                                  *
 *      n        int n;  ( n > 0 )                     *
 *               Dimension of mat                      *
 *      low      int low;                              *
 *      high     int high; see  balance                *
 *      mat      REAL   *mat[n];                       *
 *               n x n matrix                          *
 *                                                     *
 *   Output parameter:                                 *
 *   =================                                 *
 *      mat      REAL   *mat[n];                       *
 *               upper Hessenberg matrix; additional information on   *
 *               the transformation is stored in the lower triangle   *
 *      perm     int perm[];                           *
 *               Permutation vector for elmtrans       *
 *                                                     *

*====================================================================
=*
```

Name: Sreyans Bothra SRN: PES1201802012 CLASS: 4A
Name: Sahith Kurapati SRN: PES1201800032 CLASS: 4A

```
 *                                                                  *
 *   Macros:   SWAP, ABS                                            *
 *   =======                                                        *
 *                                                                  *
*====================================================================
=*/
{
  register int i, j, m;
  REAL   x, y;

  for (m = low + 1; m < high; m++)
  {
    i = m;
    x = ZERO;
    for (j = m; j <= high; j++)
      if (ABS (mat[j][m-1]) > ABS (x))
      {
        x = mat[j][m-1];
        i = j;
      }

    perm[m] = i;
    if (i != m)
    {
      for (j = m - 1; j < n; j++) SWAP (REAL, mat[i][j], mat[m][j])
      for (j = 0; j <= high; j++) SWAP (REAL, mat[j][i], mat[j][m])
    }

    if (x != ZERO)
    {
      for (i = m + 1; i <= high; i++)
      {
        y = mat[i][m-1];
        if (y != ZERO)
        {
          y = mat[i][m-1] = y / x;
          for (j = m; j < n; j++) mat[i][j] -= y * mat[m][j];
          for (j = 0; j <= high; j++) mat[j][m] += y * mat[j][i];
        }
      } /* end i */
    }
  } /* end m */

  return (0);
}


static int elmtrans       /* copy to Hessenberg form .................*/
/*.IX{elmtrans}*/
                  (int     n,       /* Dimension of matrix .........*/
                   int     low,     /* first nonzero row ...........*/
                   int     high,    /* last nonzero row ............*/
                   REAL *  mat[],   /* input matrix ................*/
                   int     perm[],  /* row permutations ............*/
                   REAL *  h[]      /* Hessenberg matrix ...........*/
                  )
/
*====================================================================
=*
 *                                                                  *
 *  elmtrans copies the Hessenberg matrix stored in mat to h.       *
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   Input parameters:                                              *
 *   ================                                               *
 *      n        int n;  ( n > 0 )                                  *
 *               Dimension of  mat and eivec                        *
 *      low      int low;                                           *
 *      high     int high; see  balance                             *
 *      mat      REAL   *mat[n];                                    *
 *               n x n input matrix                                 *
 *      perm     int *perm;                                         *
 *               Permutation data from  elmhes                      *
 *                                                                  *
 *   Output parameter:                                              *
 *   ================                                               *
 *      h        REAL   *h[n];                                      *
 *               Hessenberg matrix                                  *
 *                                                                  *

*====================================================================
=*/
{
  register int k, i, j;

  for (i = 0; i < n; i++)
  {
    for (k = 0; k < n; k++) h[i][k] = ZERO;
    h[i][i] = ONE;
  }

  for (i = high - 1; i > low; i--)
  {
    j = perm[i];
    for (k = i + 1; k <= high; k++) h[k][i] = mat[k][i-1];
    if ( i != j )
    {
      for (k = i; k <= high; k++)
      {
        h[i][k] = h[j][k];
        h[j][k] = ZERO;
      }
      h[j][i] = ONE;
    }
  }

  return (0);
}


/* ------------------------------------------------------------------ */

static int orthes     /* reduce orthogonally to upper Hessenberg form */
/*.IX{orthes}*/
                 (
                  int  n,                  /* Dimension of matrix     */
                  int  low,                /* [low,low]..[high,high]: */
                  int  high,               /* submatrix to be reduced */
                  REAL *mat[],             /* input/output matrix     */
                  REAL d[]                 /* reduction information   */
                 )                         /* error code              */

/***********************************************************************
* This function reduces matrix mat to upper Hessenberg form by         *
* Householder transformations. All details of the transformations are  *
* stored in the remaining triangle of the Hessenberg matrix and in     *
* vector d.                                                            *
*                                                                      *
* Input parameters:                                   ** =================
*
* n        dimension of mat                                            *
* low  \   rows 0 to low-1 and high+1 to n-1 contain isolated          *
* high  >  eigenvalues, i. e. eigenvalues corresponding to             *
*      /   eigenvectors that are multiples of unit vectors             *
* mat      [0..n-1,0..n-1] matrix to be reduced                        *
*                                                                      *
* Output parameters:                                                   *
* ==================                                                   *
* mat      the desired Hessenberg matrix together with the first part  *
*          of the reduction information below the subdiagonal          *
* d        [0..n-1] vector with the remaining reduction information    *
*                                                                      *
* Return value:                                                        *
* =============                                                        *
* Error code. This can only be the value 0 here.                       *
*                                                                      *
* global names used:                                                   *
* ==================                                                   *
* REAL, MACH_EPS, ZERO, SQRT                                           *
*                                                                      *
************************************************************************
* Literature: Numerical Mathematics 12 (1968), pages 359 and 360       *
***********************************************************************/

{
  int  i, j, m;    /* loop variables                                  */
  REAL s,          /* Euclidian norm sigma of the subdiagonal column  */
                   /* vector v of mat, that shall be reflected into a */
                   /* multiple of the unit vector e1 = (1,0,...,0)    */
                   /* (v = (v1,..,v(high-m+1))                        */
       x = ZERO,   /* first element of v in the beginning, then       */
                   /* summation variable in the actual Householder    */
                   /* transformation                                  */
       y,          /* sigma^2 in the beginning, then ||u||^2, with    */
                   /* u := v +- sigma * e1                            */
       eps;        /* tolerance for checking if the transformation is */
                   /* valid                                           */

  eps = (REAL)128.0 * MACH_EPS;

  for (m = low + 1; m < high; m++)
  {
    for (y = ZERO, i = high; i >= m; i--)
      x    = mat[i][m - 1],
      d[i] = x,
      y    = y + x * x;
    if (y <= eps)
      s = ZERO;
    else
    {
      s = (x >= ZERO) ? -SQRT(y) : SQRT(y);
      y    -= x * s;
      d[m] =  x - s;

      for (j = m; j < n; j++)               /* multiply mat from the  */
      {                                     /* left by  (E-(u*uT)/y)  */
        for (x = ZERO, i = high; i >= m; i--)
          x += d[i] * mat[i][j];
        for (x /= y, i = m; i <= high; i++)
          mat[i][j] -= x * d[i];
      }

      for (i = 0; i <= high; i++)           /* multiply mat from the  */
      {                                     /* right by  (E-(u*uT)/y) */
        for (x = ZERO, j = high; j >= m; j--)
          x += d[j] * mat[i][j];
        for (x /= y, j = m; j <= high; j++)
          mat[i][j] -= x * d[j];
      }
    }

    mat[m][m - 1] = s;
  }

  return 0;

}    /* --------------------------- orthes -------------------------- */


/* ------------------------------------------------------------------ */

static int orttrans       /* compute orthogonal transformation matrix */
/*.IX{orttrans}*/
                   (
```

```
          int  n,      /* Dimension of matrix              */
          int  low,    /* [low,low]..[high,high]: submatrix */
          int  high,   /* affected by the reduction        */
          REAL *mat[], /* Hessenberg matrix, reduction inf. */
          REAL d[],    /* remaining reduction information   */
          REAL *v[]    /* transformation matrix            */
         )             /* error code                       */

/***********************************************************************
* compute the matrix v of accumulated transformations from the         *
* information left by the Householder reduction of matrix mat to upper *
* Hessenberg form below the Hessenberg matrix in mat and in the        *
* vector d. The contents of the latter are destroyed.                  *
*                                                                      *
* Input parameters:                                                    *
* =================                                                    *
* n        dimension of mat                                            *
* low  \   rows 0 to low-1 and high+1 to n-1 contain isolated          *
* high  >  eigenvalues, i. e. eigenvalues corresponding to             *
*      /   eigenvectors that are multiples of unit vectors             *
* mat      [0..n-1,0..n-1] matrix produced by `orthes' giving the      *
*          upper Hessenberg matrix and part of the information on the  *
*          orthogonal reduction                                        *
* d        [0..n-1] vector with the remaining information on the       *
*          orthogonal reduction to upper Hessenberg form               *
*                                                                      *
* Output parameters:                                                   *
* ==================                                                   *
* d        input vector destroyed by this function                     *
* v        [0..n-1,0..n-1] matrix defining the similarity reduction    *
*          to upper Hessenberg form                                    *
*                                                                      *
* Return value:                                                        *
* =============                                                        *
* Error code. This can only be the value 0 here.                       *
*                                                                      *
* benutzte globale Namen:                                              *
* =======================                                              *
* REAL, ZERO, ONE                                                      *
*                                                                      *
************************************************************************
* Literature: Numerical Mathematics 16 (1970), page 191                *
***********************************************************************/


{
  int  i, j, m;                        /* loop variables            */
  REAL x,                              /* summation variable in the */
                                       /* Householder transformation */
       y;                              /* sigma  respectively        */
                                       /* sigma * (v1 +- sigma)      */

  for (i = 0; i < n; i++)              /* form the unit matrix in v  */
  {
    for (j = 0; j < n; j++)
      v[i][j] = ZERO;
    v[i][i] = ONE;
  }

  for (m = high - 1; m > low; m--)     /* apply the transformations  */
  {                                    /* that reduced mat to upper  */
    y = mat[m][m - 1];                 /* Hessenberg form also to the */
                                       /* unit matrix in v. This     */
    if (y != ZERO)                     /* produces the desired       */
    {                                  /* transformation matrix in v. */
      y *= d[m];
      for (i = m + 1; i <= high; i++)
        d[i] = mat[i][m - 1];
      for (j = m; j <= high; j++)
      {
        for (x = ZERO, i = m; i <= high; i++)
          x += d[i] * v[i][j];
        for (x /= y, i = m; i <= high; i++)
          v[i][j] += x * d[i];
      }
    }
  }

  return 0;

}    /* -------------------------- orttrans ------------------------- */


static int hqrvec       /* compute eigenvectors ......................*/
/*.IX{hqrvec}*/
              (int     n,          /* Dimension of matrix .......*/
               int     low,        /* first nonzero row .........*/
               int     high,       /* last nonzero row ..........*/
               REAL *  h[],        /* upper Hessenberg matrix ...*/
               REAL    wr[],       /* Real parts of evalues .....*/
               REAL    wi[],       /* Imaginary parts of evalues */
               REAL *  eivec[]     /* Eigenvectors ..............*/
              )
/
*====================================================================
=*
 *                                                                  *
 *  hqrvec computes the eigenvectors for the eigenvalues found in hqr2*
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   Input parameters:                                              *
 *   ================                                               *
 *      n        int n;  ( n > 0 )                                  *
 *               Dimension of  mat and eivec, number of eigenvalues. *
 *      low      int low;                                           *
 *      high     int high; see  balance                             *
 *      h        REAL   *h[n];                                      *
 *               upper Hessenberg matrix                            *
 *      wr       REAL   wr[n];                                      *
 *               Real parts of the n eigenvalues.                   *
 *      wi       REAL   wi[n];                                      *
 *               Imaginary parts of the n eigenvalues.              *
 *                                                                  *
 *   Output parameter:                                              *
 *   ================                                               *
 *      eivec    REAL   *eivec[n];                                  *
 *               Matrix, whose columns are the eigenvectors         *
 *                                                                  *
 *   Return value :                                                 *
 *   =============                                                  *
 *      =  0     all ok                                             *
 *      =  1     h is the zero matrix.                              *
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   function in use  :                                             *
 *   ==================                                             *
*                                                                   *
 *      int   comdiv(): complex division                            *
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   Constants used  :    MACH_EPS                                  *
 *   =================                                              *
 *                                                                  *
 *   Macros :   SQR, ABS                                            *
 *   ========                                                       *
 *                                                                  *

*====================================================================
=*/
{
  int   i, j, k;
  int   l, m, en, na;
  REAL  p, q, r = ZERO, s = ZERO, t, w, x, y, z = ZERO,
        ra, sa, vr, vi, norm;

  for (norm = ZERO, i = 0; i < n; i++)        /* find norm of h      */
  {
    for (j = i; j < n; j++) norm += ABS(h[i][j]);
  }

  if (norm == ZERO) return (1);              /* zero matrix          */
#pragma omp parallel shared(h) private(en,i,j)
{
  #pragma omp for schedule(static)
  for (en = n - 1; en >= 0; en--)            /* transform back       */
  {
    p = wr[en];
    q = wi[en];
    na = en - 1;
    if (q == ZERO)
    {
      m = en;
      h[en][en] = ONE;
      for (i = na; i >= 0; i--)
      {
        w = h[i][i] - p;
        r = h[i][en];
        for (j = m; j <= na; j++) r += h[i][j] * h[j][en];
        if (wi[i] < ZERO)
        {
          z = w;
          s = r;
        }
        else
        {
          m = i;
          if (wi[i] == ZERO)
            h[i][en] = -r / ((w != ZERO) ? (w) : (MACH_EPS * norm));
          else
          {  /* Solve the linear system:            */
             /* | w   x |  | h[i][en]   |   | -r |  */
             /* |       |  |            | = |    |  */
             /* | y   z |  | h[i+1][en] |   | -s |  */

             x = h[i][i+1];
             y = h[i+1][i];
             q = SQR (wr[i] - p) + SQR (wi[i]);
             h[i][en] = t = (x * s - z * r) / q;
             h[i+1][en] = ( (ABS(x) > ABS(z) ) ?
                            (-r -w * t) / x : (-s -y * t) / z);
          }
        }  /* wi[i] >= 0  */
      }  /*  end i     */
    }  /* end q = 0  */

    else if (q < ZERO)
    {
      m = na;
      if (ABS(h[en][na]) > ABS(h[na][en]))
      {
        h[na][na] = - (h[en][en] - p) / h[en][na];
        h[na][en] = - q / h[en][na];
      } else {
                                /* comdiv(-h[na][en], ZERO, h[na][na]-p, q, &h[na][na], &h[na]
[en]); */
                                REAL complex c;
                                c = -h[na][en] / (h[na][na]-p + q * I);
                                h[na][na] = creal(c); h[na][en] = cimag(c);
                }
```

```
    h[en][na] = ONE;
    h[en][en] = ZERO;
    for (i = na - 1; i >= 0; i--)
    {
      w = h[i][i] - p;
      ra = h[i][en];
      sa = ZERO;
      for (j = m; j <= na; j++)
      {
        ra += h[i][j] * h[j][na];
        sa += h[i][j] * h[j][en];
      }

      if (wi[i] < ZERO)
      {
        z = w;
        r = ra;
        s = sa;
      }
      else
      {
        m = i;
        if (wi[i] == ZERO) {
                                        /* comdiv (-ra, -sa, w, q, &h[i][na], &h[i][en]); */
                                        REAL complex c;
                                        c = (-ra - sa * I) / (w + q * I);
                                        h[i][na] = creal(c); h[i][en] = cimag(c);
        } else
        {

       /* solve complex linear system:                          */
       /* | w+i*q     x | | h[i][na] + i*h[i][en]  |   | -ra+i*sa | */
       /* |             | |                        | = |          | */
       /* |   y    z+i*q| | h[i+1][na]+i*h[i+1][en]|   | -r+i*s   | */

          x = h[i][i+1];
          y = h[i+1][i];
          vr = SQR (wr[i] - p) + SQR (wi[i]) - SQR (q);
          vi = TWO * q * (wr[i] - p);
          if (vr == ZERO && vi == ZERO)
            vr = MACH_EPS * norm *
                (ABS (w) + ABS (q) + ABS (x) + ABS (y) + ABS (z));
                                        {
                                                /* comdiv (x * r - z * ra + q * sa, x *
s - z * sa -q * ra,
                                                  vr, vi, &h[i][na], &h[i][en]); */
                                                REAL complex c;
                                                c = (x * r - z * ra + q * sa + I * (x * s
- z * sa -q * ra)) / (vr + I * vi);
                                                h[i][na] = creal(c); h[i][en] =
cimag(c);
                                        }
          if (ABS (x) > ABS (z) + ABS (q))
          {
            h[i+1][na] = (-ra - w * h[i][na] + q * h[i][en]) / x;
            h[i+1][en] = (-sa - w * h[i][en] - q * h[i][na]) / x;
          }
          else {
                                                /* comdiv (-r - y * h[i][na], -s - y *
h[i][en], z, q,
                                                  &h[i+1][na], &h[i+1][en]); */
                                                REAL complex c;
                                                c = (-r - y * h[i][na] + I * (-s - y *
h[i][en])) / (z + I * q);
                                                h[i+1][na] = creal(c); h[i+1][en] =
cimag(c);
                                        }

        }  /* end wi[i] > 0  */
      }  /* end wi[i] >= 0  */
    }  /* end i           */
  }  /*  if q < 0       */
 }  /* end  en          */
}

 for (i = 0; i < n; i++)         /* Eigenvectors for the evalues for */
   if (i < low || i > high)      /* rows < low  and rows  > high     */
     for (k = i + 1; k < n; k++) eivec[i][k] = h[i][k];

 for (j = n - 1; j >= low; j--)
 {
   m = (j <= high) ? j : high;
   if (wi[j] < ZERO)
   {
     for (l = j - 1, i = low; i <= high; i++)
     {
       for (y = z = ZERO, k = low; k <= m; k++)
       {
         y += eivec[i][k] * h[k][l];
         z += eivec[i][k] * h[k][j];
       }

       eivec[i][l] = y;
       eivec[i][j] = z;
     }
   }
   else
     if (wi[j] == ZERO)
     {
       for (i = low; i <= high; i++)
       {
         for (z = ZERO, k = low; k <= m; k++)
           z += eivec[i][k] * h[k][j];
         eivec[i][j] = z;
       }
     }

 }  /*  end j  */

  return (0);
}


static int hqr2        /* compute eigenvalues .......................*/
/*.IX{hqr2}*/
                (int     vec,        /* switch for computing evectors*/
                 int     n,          /* Dimension of matrix .........*/
                 int     low,        /* first nonzero row ...........*/
                 int     high,       /* last nonzero row ............*/
                 REAL *  h[],        /* Hessenberg matrix ...........*/
                 REAL    wr[],       /* Real parts of eigenvalues ...*/
                 REAL    wi[],       /* Imaginary parts of evalues ..*/
                 REAL *  eivec[],    /* Matrix of eigenvectors ......*/
                 int     cnt[]       /* Iteration counter ...........*/
                )
/
*====================================================================
=*
 *                                                                  *
 *  hqr2 computes the eigenvalues and (if vec != 0) the eigenvectors  *
 *  of an  n * n upper Hessenberg matrix.                           *
 *                                                                  *

*====================================================================
=* *                                                                *
 *   Control parameter:                                             *
 *   ==================                                             *
 *      vec      int vec;                                           *
 *       = 0    compute eigenvalues only                            *
 *       = 1    compute all eigenvalues and eigenvectors            *
 *                                                                  *
 *   Input parameters:                                              *
 *   ================                                               *
 *      n        int n;  ( n > 0 )                                  *
 *               Dimension of  mat and eivec,                       *
 *               length of the real parts vector  wr and of the     *
 *               imaginary parts vector  wi of the eigenvalues.     *
 *      low      int low;                                           *
 *      high     int high; see  balance                             *
 *      h        REAL   *h[n];                                      *
 *               upper  Hessenberg matrix                           *
 *                                                                  *
 *   Output parameters:                                             *
 *   ==================                                             *
 *      eivec    REAL   *eivec[n];     ( bei vec = 1 )              *
 *               Matrix, which for vec = 1 contains the eigenvectors  *
 *               as follows  :                                      *
 *               For real eigebvalues the corresponding column      *
 *               contains the corresponding eigenvactor, while for  *
 *               complex eigenvalues the corresponding column contains*
 *               the real part of the eigenvactor with its imaginary  *
 *               part is stored in the subsequent column of eivec.  *
 *               The eigenvactor for the complex conjugate eigenvactor*
 *               is given by the complex conjugate eigenvactor.     *
 *      wr       REAL   wr[n];                                      *
 *               Real part of the n eigenvalues.                    *
 *      wi       REAL   wi[n];                                      *
 *               Imaginary parts of the eigenvalues                 *
 *      cnt      int cnt[n];                                        *
 *               vector of iterations used for each eigenvalue.     *
 *               For a complex conjugate eigenvalue pair the second  *
 *               entry is negative.                                 *
 *                                                                  *
 *   Return value :                                                 *
 *   =============                                                  *
 *      =  0    all ok                                              *
 *      = 4xx    Iteration maximum exceeded when computing evalue xx  *
 *      =  99    zero  matrix                                       *
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   functions in use  :                                            *
 *   ===================                                            *
 *                                                                  *
 *      int hqrvec(): reverse transform for eigenvectors            *
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   Constants used  :   MACH_EPS, MAXIT                            *
 *   =================                                              *
 *                                                                  *
 *   Macros  :   SWAP, ABS, SQRT                                    *
 *   =========                                                      *
 *                                                                  *

*====================================================================
=*/
{
  int  i, j;
  int  na, en, iter, k, l, m;
  REAL p = ZERO, q = ZERO, r = ZERO, s, t, w, x, y, z;

  for (i = 0; i < n; i++)
    if (i < low || i > high)
    {
      wr[i] = h[i][i];
      wi[i] = ZERO;
      cnt[i] = 0;
    }

  en = high;
  t = ZERO;

  while (en >= low)
```

```
{
 iter = 0;
 na = en - 1;

 for ( ; ; )
 {
  for (l = en; l > low; l--)           /* search for small      */
   if ( ABS(h[l][l-1]) <=              /* subdiagonal element   */
        MACH_EPS * (ABS(h[l-1][l-1]) + ABS(h[l][l])) )  break;

  x = h[en][en];
  if (l == en)                         /* found one evalue     */
  {
   wr[en] = h[en][en] = x + t;
   wi[en] = ZERO;
   cnt[en] = iter;
   en--;
   break;
  }

  y = h[na][na];
  w = h[en][na] * h[na][en];

  if (l == na)                         /* found two evalues    */
  {
   p = (y - x) * 0.5;
   q = p * p + w;
   z = SQRT (ABS (q));
   x = h[en][en] = x + t;
   h[na][na] = y + t;
   cnt[en] = -iter;
   cnt[na] = iter;
   if (q >= ZERO)
   {                                   /* real eigenvalues     */
    z = (p < ZERO) ? (p - z) : (p + z);
    wr[na] = x + z;
    wr[en] = s = x - w / z;
    wi[na] = wi[en] = ZERO;
    x = h[en][na];
    r = SQRT (x * x + z * z);

    if (vec)
    {
     p = x / r;
     q = z / r;
     for (j = na; j < n; j++)
     {
      z = h[na][j];
      h[na][j] = q * z + p * h[en][j];
      h[en][j] = q * h[en][j] - p * z;
     }

     for (i = 0; i <= en; i++)
     {
      z = h[i][na];
      h[i][na] = q * z + p * h[i][en];
      h[i][en] = q * h[i][en] - p * z;
     }

     for (i = low; i <= high; i++)
     {
      z = eivec[i][na];
      eivec[i][na] = q * z + p * eivec[i][en];
      eivec[i][en] = q * eivec[i][en] - p * z;
     }
    }  /* end if (vec) */
   }  /* end if (q >= ZERO) */
   else                                /* pair of complex      */
   {                                   /* conjugate evalues    */
    wr[na] = wr[en] = x + p;
    wi[na] =   z;
    wi[en] = - z;
   }

   en -= 2;
   break;
  }  /* end if (l == na) */

  if (iter >= MAXIT)
  {
   cnt[en] = MAXIT + 1;
   return (en);                        /* MAXIT Iterations     */
  }

  if ( (iter != 0) && (iter % 10 == 0) )
  {
   t += x;
   for (i = low; i <= en; i++) h[i][i] -= x;
   s = ABS (h[en][na]) + ABS (h[na][en-2]);
   x = y = (REAL)0.75 * s;
   w = - (REAL)0.4375 * s * s;
  }

  iter ++;

  for (m = en - 2; m >= l; m--)
  {
   z = h[m][m];
   r = x - z;
   s = y - z;
   p = ( r * s - w ) / h[m+1][m] + h[m][m+1];
   q = h[m + 1][m + 1] - z - r - s;
   r = h[m + 2][m + 1];
   s = ABS (p) + ABS (q) + ABS (r);
   p /= s;
   q /= s;
   r /= s;
   if (m == l) break;
   if ( ABS (h[m][m-1]) * (ABS (q) + ABS (r)) <=
        MACH_EPS * ABS (p)
        * ( ABS (h[m-1][m-1]) + ABS (z) + ABS (h[m+1][m+1])) )
     break;
  }


  for (i = m + 2; i <= en; i++) h[i][i-2] = ZERO;
  for (i = m + 3; i <= en; i++) h[i][i-3] = ZERO;

  for (k = m; k <= na; k++)
  {
   if (k != m)            /* double  QR step, for rows l to en  */
   {                      /* and columns m to en                */
    p = h[k][k-1];
    q = h[k+1][k-1];
    r = (k != na) ? h[k+2][k-1] : ZERO;
    x = ABS (p) + ABS (q) + ABS (r);
    if (x == ZERO) continue;                /*  next k        */
    p /= x;
    q /= x;
    r /= x;
   }
   s = SQRT (p * p + q * q + r * r);
   if (p < ZERO) s = -s;

   if (k != m) h[k][k-1] = -s * x;
    else if (l != m)
         h[k][k-1] = -h[k][k-1];
   p += s;
   x = p / s;
   y = q / s;
   z = r / s;
   q /= p;
   r /= p;

   for (j = k; j < n; j++)              /* modify rows          */
   {
    p = h[k][j] + q * h[k+1][j];
    if (k != na)
    {
     p += r * h[k+2][j];
     h[k+2][j] -= p * z;
    }
    h[k+1][j] -= p * y;
    h[k][j]   -= p * x;
   }

   j = (k + 3 < en) ? (k + 3) : en;
   for (i = 0; i <= j; i++)             /* modify columns       */
   {
    p = x * h[i][k] + y * h[i][k+1];
    if (k != na)
    {
     p += z * h[i][k+2];
     h[i][k+2] -= p * r;
    }
    h[i][k+1] -= p * q;
    h[i][k]   -= p;
   }

   if (vec)      /* if eigenvectors are needed ..................*/
   {
    for (i = low; i <= high; i++)
    {
     p = x * eivec[i][k] + y * eivec[i][k+1];
     if (k != na)
     {
      p += z * eivec[i][k+2];
      eivec[i][k+2] -= p * r;
     }
     eivec[i][k+1] -= p * q;
     eivec[i][k]   -= p;
    }
   }
  }    /* end k          */

 }    /* end for ( ; ;) */

}    /* while (en >= low)                 All evalues found    */

 if (vec)                               /* transform evectors back  */
  if (hqrvec (n, low, high, h, wr, wi, eivec)) return (99);
 return (0);
}

static int norm_1       /* normalize eigenvectors to have one norm 1 .*/
/*.IX{norm\unt 1}*/
          (int     n,      /* Dimension of matrix ...........*/
           REAL *  v[],    /* Matrix with eigenvektors ......*/
           REAL    wi[]    /* Imaginary parts of evalues ....*/
          )
/
*====================================================================
=*
 *                                                                  *
 *  norm_1 normalizes the one norm of the column vectors in v.      *
 *  (special attention to complex vectors in v  is given)           *
 *                                                                  *

*====================================================================
=*
 *                                                                  *
 *   Input parameters:                                              *
 *   ================                                               *
 *      n        int n; ( n > 0 )                                   *
 *               Dimension of matrix v                              *
 *      v        REAL   *v[];                                       *
 *               Matrix of eigenvectors                             *
 *      wi       REAL   wi[];                                       *
```

```
 *          Imaginary parts of the eigenvalues             *
 *                                                          *
 *  Output parameter:                                       *
 *  ================                                        *
 *     v       REAL   *v[];                                 *
 *             Matrix with normalized eigenvectors          *
 *                                                          *
 *  Return value :                                          *
 *  ============                                            *
 *     = 0     all ok                                       *
 *     = 1     n < 1                                        *
 *                                                          *

*==================================================================
=*
 *                                                          *
 *  functions used  :                                       *
 *  ================                                        *
 *     REAL   comabs():  complex absolute value             *
 *     int    comdiv():  complex division                   *
 *                                                          *
 *  Macros :   ABS                                          *
 *  =======                                                 *
 *                                                          *

*==================================================================
=*/
{
  int  i, j;
  REAL maxi, tr, ti;

  if (n < 1) return (1);
#pragma omp parallel shared(v) private(i,j)
{
  #pragma omp for schedule(static)
  for (j = 0; j < n; j++)
  {

    if (wi[j] == ZERO)
    {
      maxi = v[0][j];
      for (i = 1; i < n; i++)
        if (ABS (v[i][j]) > ABS (maxi))  maxi = v[i][j];

      if (maxi != ZERO)
      {
        maxi = ONE / maxi;
        for (i = 0; i < n; i++) v[i][j] *= maxi;
      }
    }

    else
    {
      tr = v[0][j];
      ti = v[0][j+1];
      for (i = 1; i < n; i++)
                        /* if ( comabs (v[i][j], v[i][j+1]) > comabs (tr, ti) ) */
                        if (cabs(v[i][j] + I * v[i][j+1]) > cabs(tr + I * ti))
                        {
                                tr = v[i][j];
                                ti = v[i][j+1];
                        }

      if (tr != ZERO || ti != ZERO)
                        for (i = 0; i < n; i++) {
                                /* comdiv (v[i][j], v[i][j+1], tr, ti, &v[i][j], &v[i]
[j+1]); */
                                REAL complex c;
                                c = (v[i][j] + I * v[i][j+1]) / (tr + I * ti);
                                v[i][j] = creal(c); v[i][j+1] = cimag(c);
                        }

      j++;                          /* raise j by two */
    }
  }
}
  return (0);
}

/*.BA*/

static int eigen        /* Compute all evalues/evectors of a matrix ..*/
/*.IX{eigen}*/
         (
          int     vec,           /* switch for computing evectors ...*/
          int     ortho,         /* orthogonal Hessenberg reduction? */
          int     ev_norm,       /* normalize Eigenvectors? .........*/
          int     n,             /* size of matrix ..................*/
          REAL *  mat[],         /* input matrix ....................*/
          REAL *  eivec[],       /* Eigenvectors ....................*/
          REAL    valre[],       /* real parts of eigenvalues .......*/
          REAL    valim[],       /* imaginary parts of eigenvalues ..*/
          int     cnt[]          /* Iteration counter ...............*/
         )
/
*==================================================================
=*
 *                                                          *
 *  The function  eigen  determines all eigenvalues and (if desired)  *
 *  all eigenvectors of a real square  n * n  matrix via the QR method*
 *  in the version of  Martin, Parlett, Peters, Reinsch and Wilkinson.*
 *                                                          *

*==================================================================
=*
.BE*)
 *                                                          *
 *  Literature:                                             *
 *  ==========                                              *
 *     1) Peters, Wilkinson: Eigenvectors of real and complex    *
 *        matrices by LR and QR triangularisations,              *
 *        Num. Math. 16, p.184-204, (1970); [PETE70]; contribution  *
 *        II/15, p. 372 - 395 in [WILK71].                       *
 *     2) Martin, Wilkinson: Similarity reductions of a general  *
 *        matrix to Hessenberg form, Num. Math. 12, p. 349-368,(1968)*
 *        [MART 68]; contribution II,13, p. 339 - 358 in [WILK71].   *
 *     3) Parlett, Reinsch: Balancing a matrix for calculations of  *
 *        eigenvalues and eigenvectors, Num. Math. 13, p. 293-304,  *
 *        (1969); [PARL69]; contribution II/11, p.315 - 326 in      *
 *        [WILK71].                                              *
 *                                                          *

*==================================================================
=*
 *                                                          *
 *  Control parameters:                                     *
 *  ===================                                     *
 *     vec      int vec;                                    *
 *              call for eigen :                            *
 *      = 0     compute eigenvalues only                    *
 *      = 1     compute all eigenvalues and eigenvectors    *
 *     ortho    flag that shows if transformation of mat to *
 *              Hessenberg form shall be done orthogonally by  *
 *              `orthes' (flag set) or elementarily by `elmhes'  *
 *              (flag cleared). The Householder matrices used in  *
 *              orthogonal transformation have the advantage of  *
 *              preserving the symmetry of input matrices.   *
 *     ev_norm  flag that shows if Eigenvectors shall be    *
 *              normalized (flag set) or not (flag cleared)  *
 *                                                          *
 *  Input parameters:                                       *
 *  ================                                        *
 *     n        int n;  ( n > 0 )                           *
 *              size of matrix, number of eigenvalues       *
 *     mat      REAL   *mat[n];                             *
 *              matrix                                      *
 *                                                          *
 *  Output parameters:                                      *
 *  ==================                                      *
 *     eivec    REAL   *eivec[n];     ( bei vec = 1 )       *
 *              matrix, if  vec = 1  this holds the eigenvectors   *
 *              thus :                                      *
 *              If the jth eigenvalue of the matrix is real then the *
 *              jth column is the corresponding real eigenvector;   *
 *              if the jth eigenvalue is complex then the jth column *
 *              of eivec contains the real part of the eigenvector   *
 *              while its imaginary part is in column j+1.   *
 *              (the j+1st eigenvector is the complex conjugate     *
 *              vector.)                                    *
 *     valre    REAL   valre[n];                            *
 *              Real parts of the eigenvalues.              *
 *     valim    REAL   valim[n];                            *
 *              Imaginary parts of the eigenvalues          *
 *     cnt      int cnt[n];                                 *
 *              vector containing the number of iterations for each  *
 *              eigenvalue. (for a complex conjugate pair the second *
 *              entry is negative.)                         *
 *                                                          *
 *  Return value :                                          *
 *  ============                                            *
 *     =   0    all ok                                      *
 *     =   1    n < 1 or other invalid input parameter      *
 *     =   2    insufficient memory                         *
 *     = 10x    error x from balance()                      *
 *     = 20x    error x from elmh()                         *
 *     = 30x    error x from elmtrans()   (for vec = 1 only)  *
 *     = 4xx    error xx from hqr2()                        *
 *     = 50x    error x from balback()    (for vec = 1 only)  *
 *     = 60x    error x from norm_1()     (for vec = 1 only)  *
 *                                                          *

*==================================================================
=*
 *                                                          *
 *  Functions in use   :                                    *
 *  ===================                                     *
 *                                                          *
 *  static int balance (): Balancing of an  n x n  matrix   *
 *  static int elmh ():    Transformation to upper Hessenberg form   *
 *  static int elmtrans(): intialize eigenvectors           *
 *  static int hqr2 ():    compute eigenvalues/eigenvectors *
 *  static int balback (): Reverse balancing to obtain eigenvectors  *
 *  static int norm_1 ():  Normalize eigenvectors           *
 *                                                          *
 *  void *vmalloc():       allocate vector or matrix        *
 *  void vmfree():         free list of vectors and matrices  *
 *                                                          *

*==================================================================
=*
 *                                                          *
 *  Constants used   :     NULL, BASIS                      *
 *  ===================                                     *
 *                                                          *

*==================================================================
=*/
{
  int i;
  int      low, high, rc;
  REAL     *scale,
           *d = NULL;
  void     *vmblock;

  if (n < 1) return (1);                      /*  n >= 1 ............*/

  if (valre == NULL || valim == NULL || mat == NULL || cnt == NULL)
    return (1);
```

```
 for (i = 0; i < n; i++)
   if (mat[i] == NULL) return (1);

 for (i = 0; i < n; i++) cnt[i] = 0;

 if (n == 1)                                  /*  n = 1 ............*/
 {
   eivec[0][0] = ONE;
   valre[0]    = mat[0][0];
   valim[0]    = ZERO;
   return (0);
 }

 if (vec)
 {
   if (eivec == NULL) return (1);
   for (i = 0; i < n; i++)
     if (eivec[i] == NULL) return (1);
 }

 vmblock = vminit();
 scale = (REAL *)vmalloc(vmblock, VEKTOR, n, 0);
 if (! vmcomplete(vmblock))                /* memory error         */
   return 2;

 if (vec && ortho)                         /* with Eigenvectors     */
 {                                         /* and orthogonal        */
                                           /* Hessenberg reduction? */
   d = (REAL *)vmalloc(vmblock, VEKTOR, n, 0);
   if (! vmcomplete(vmblock))
   {
     vmfree(vmblock);
     return 1;
   }
 }

                                           /* balance mat for nearly */
 rc = balance (n, mat, scale,              /* equal row and column   */
               &low, &high, BASIS);        /* one norms              */
 if (rc)
 {
   vmfree(vmblock);
   return (100 + rc);
 }

 if (ortho)
   rc = orthes(n, low, high, mat, d);
 else
   rc = elmhes (n, low, high, mat, cnt);   /*  reduce mat to upper   */
 if (rc)                                   /*  Hessenberg form       */
 {
   vmfree(vmblock);
   return (200 + rc);
 }

 if (vec)                                  /*  initialize eivec      */
 {
   if (ortho)
     rc = orttrans(n, low, high, mat, d, eivec);
   else
     rc = elmtrans (n, low, high, mat, cnt, eivec);
   if (rc)
   {
     vmfree(vmblock);
     return (300 + rc);
   }
 }

 rc = hqr2 (vec, n, low, high, mat,        /*  execute Francis QR    */
            valre, valim, eivec, cnt);     /*  algorithm to obtain   */
 if (rc)                                   /*  eigenvalues           */
 {
   vmfree(vmblock);
   return (400 + rc);
 }

 if (vec)
 {
   rc = balback (n, low, high,             /*  reverse balancing if  */
                 scale, eivec);            /*  eigenvaectors are to  */
   if (rc)                                 /*  be determined         */
   {
     vmfree(vmblock);
     return (500 + rc);
   }
   if (ev_norm)
     rc = norm_1 (n, eivec, valim);        /* normalize eigenvectors */
   if (rc)
   {
     vmfree(vmblock);
     return (600 + rc);
   }
 }

 vmfree(vmblock);                          /* free buffers          */


 return (0);
}

/* -------------------------- END feigen.c -------------------------- */

/* _a[0..n^2-1] is a real general matrix. On return, evalr store the
   real part of eigenvalues and evali the imgainary part. If _evec is
   not a NULL pointer, eigenvectors will be stored there. */
int n_eigeng(double *_a, int n, double *evalr, double *evali, double *_evec, double *b)
{
            double **a, **evec = 0;
            int i, j, *cnt;
            a = (double**)calloc(n, sizeof(void*));
            if (_evec) evec = (double**)calloc(n, sizeof(void*));
            cnt = (int*)calloc(n, sizeof(int));
    #pragma omp parallel shared(evec) private(i)
      {
      #pragma omp for schedule(static)
      for (i = 0; i < n; ++i) {
         a[i] = _a + i * n;
         if (_evec) evec[i] = _evec + i * n;
      }
   }
            eigen(_evec? 1 : 0, 0, 1, n, a, evec, evalr, evali, cnt);
            if (_evec) {
                        double tmp;                   for (j = 0; j < n; ++j) {
                                    tmp = 0.0;
         #pragma omp parallel shared(evec) private(i)
          {
         #pragma omp for schedule(static)
                                    for (i = 0; i < n; ++i) tmp += SQR(evec[i][j]);
         }
                                    tmp = SQRT(tmp);
          #pragma omp parallel shared(evec) private(i)
          {
         #pragma omp for schedule(static)
                                    for (i = 0; i < n; ++i) evec[i][j] /= tmp;
          }
                        }
            }
            free(a); free(evec); free(cnt);
            return 0;
}


static int hqrvec1       /* compute eigenvectors ......................*/
/*.IX{hqrvec}*/
        (int     n,          /* Dimension of matrix .......*/
         int     low,        /* first nonzero row .........*/
         int     high,       /* last nonzero row ..........*/
         REAL *  h[],        /* upper Hessenberg matrix ...*/

         REAL    wr[],       /* Real parts of evalues .....*/
         REAL    wi[],       /* Imaginary parts of evalues */
         REAL *  eivec[]     /* Eigenvectors ..............*/
        )

{
  int   i, j, k;
  int   l, m, en, na;
  REAL  p, q, r = ZERO, s = ZERO, t, w, x, y, z = ZERO,
        ra, sa, vr, vi, norm;

  for (norm = ZERO, i = 0; i < n; i++)       /* find norm of h       */
  {
    for (j = i; j < n; j++) norm += ABS(h[i][j]);
  }

  if (norm == ZERO) return (1);              /* zero matrix          */

  for (en = n - 1; en >= 0; en--)            /* transform back       */
  {
    p = wr[en];
    q = wi[en];
    na = en - 1;
    if (q == ZERO)
    {
      m = en;
      h[en][en] = ONE;
      for (i = na; i >= 0; i--)
      {
        w = h[i][i] - p;
        r = h[i][en];
        for (j = m; j <= na; j++) r += h[i][j] * h[j][en];
        if (wi[i] < ZERO)
        {
          z = w;
          s = r;
        }
        else
        {
          m = i;
          if (wi[i] == ZERO)
            h[i][en] = -r / ((w != ZERO) ? (w) : (MACH_EPS * norm));
          else
          {  /* Solve the linear system:            */
             /* | w   x |  | h[i][en]   |   | -r |  */
             /* |       |  |            | = |    |  */
             /* | y   z |  | h[i+1][en] |   | -s |  */

             x = h[i][i+1];
             y = h[i+1][i];
             q = SQR (wr[i] - p) + SQR (wi[i]);
             h[i][en] = t = (x * s - z * r) / q;
             h[i+1][en] = ( (ABS(x) > ABS(z) ) ?
                          (-r -w * t) / x : (-s -y * t) / z);
          }
        }  /* wi[i] >= 0  */
      }  /*  end i     */
    }  /* end q = 0  */

    else if (q < ZERO)
    {
      m = na;
      if (ABS(h[en][na]) > ABS(h[na][en]))
      {
        h[na][na] = - (h[en][en] - p) / h[en][na];
        h[na][en] = - q / h[en][na];
      } else {
```

```
                    /* comdiv(-h[na][en], ZERO, h[na][na]-p, q, &h[na][na], &h[na]
[en]); */
                    REAL complex c;
                    c = -h[na][en] / (h[na][na]-p + q * I);
                    h[na][na] = creal(c); h[na][en] = cimag(c);
          }

      h[en][na] = ONE;
      h[en][en] = ZERO;
      for (i = na - 1; i >= 0; i--)
      {
        w = h[i][i] - p;
        ra = h[i][en];
        sa = ZERO;
        for (j = m; j <= na; j++)
        {
          ra += h[i][j] * h[j][na];
          sa += h[i][j] * h[j][en];
        }

        if (wi[i] < ZERO)
        {
          z = w;
          r = ra;
          s = sa;
        }
        else
        {
          m = i;
          if (wi[i] == ZERO) {
                              /* comdiv (-ra, -sa, w, q, &h[i][na], &h[i][en]); */
                              REAL complex c;
                              c = (-ra - sa * I) / (w + q * I);
                              h[i][na] = creal(c); h[i][en] = cimag(c);
          } else
          {

         /* solve complex linear system:                    */
         /* | w+i*q     x | | h[i][na] + i*h[i][en]  |   | -ra+i*sa | */
         /* |             | |                        | = |          | */
         /* |   y    z+i*q| | h[i+1][na]+i*h[i+1][en]|   | -r+i*s   | */

            x = h[i][i+1];
            y = h[i+1][i];
            vr = SQR (wr[i] - p) + SQR (wi[i]) - SQR (q);
            vi = TWO * q * (wr[i] - p);
            if (vr == ZERO && vi == ZERO)
              vr = MACH_EPS * norm *
                (ABS (w) + ABS (q) + ABS (x) + ABS (y) + ABS (z));
                              {
                                        /* comdiv (x * r - z * ra + q * sa, x *
s - z * sa -q * ra,

                                           vr, vi, &h[i][na], &h[i][en]); */
                                        REAL complex c;
                                        c = (x * r - z * ra + q * sa + I * (x * s
- z * sa -q * ra)) / (vr + I * vi);
                                        h[i][na] = creal(c); h[i][en] =
cimag(c);
                              }
            if (ABS (x) > ABS (z) + ABS (q))
            {
              h[i+1][na] = (-ra - w * h[i][na] + q * h[i][en]) / x;
              h[i+1][en] = (-sa - w * h[i][en] - q * h[i][na]) / x;
            }
            else {
                                        /* comdiv (-r - y * h[i][na], -s - y *
h[i][en], z, q,

                                           &h[i+1][na], &h[i+1][en]); */
                                        REAL complex c;
                                        c = (-r - y * h[i][na] + I * (-s - y *
h[i][en])) / (z + I * q);

                                        h[i+1][na] = creal(c); h[i+1][en] =
cimag(c);
                              }

          }   /* end wi[i] > 0  */
        }   /* end wi[i] >= 0  */
      }   /* end i            */
    }    /*  if q < 0       */
  }    /* end  en           */

  for (i = 0; i < n; i++)         /* Eigenvectors for the evalues for */
    if (i < low || i > high)      /* rows < low  and rows  > high     */
      for (k = i + 1; k < n; k++) eivec[i][k] = h[i][k];

  for (j = n - 1; j >= low; j--)
  {
    m = (j <= high) ? j : high;
    if (wi[j] < ZERO)
    {
      for (l = j - 1, i = low; i <= high; i++)
      {
        for (y = z = ZERO, k = low; k <= m; k++)
        {
          y += eivec[i][k] * h[k][l];
          z += eivec[i][k] * h[k][j];
        }

        eivec[i][l] = y;
        eivec[i][j] = z;
      }
    }
    else
      if (wi[j] == ZERO)
      {
        for (i = low; i <= high; i++)
        {
          for (z = ZERO, k = low; k <= m; k++)
            z += eivec[i][k] * h[k][j];
          eivec[i][j] = z;
        }
      }

  }  /*  end j  */

  return (0);
}


static int hqr21         /* compute eigenvalues .......................*/
/*.IX{hqr2}*/
                (int     vec,         /* switch for computing evectors*/
                 int     n,           /* Dimension of matrix .........*/
                 int     low,         /* first nonzero row ...........*/
                 int     high,        /* last nonzero row ............*/
                 REAL *  h[],         /* Hessenberg matrix ...........*/
                 REAL    wr[],        /* Real parts of eigenvalues ...*/                 REAL    wi[],        /*
Imaginary parts of evalues ..*/
                 REAL *  eivec[],     /* Matrix of eigenvectors ......*/
                 int     cnt[]        /* Iteration counter ...........*/
                )
{
  int  i, j;
  int  na, en, iter, k, l, m;
  REAL p = ZERO, q = ZERO, r = ZERO, s, t, w, x, y, z;

  for (i = 0; i < n; i++)
    if (i < low || i > high)
    {
      wr[i] = h[i][i];
      wi[i] = ZERO;
      cnt[i] = 0;
    }

  en = high;
  t = ZERO;

  while (en >= low)
  {
    iter = 0;
    na = en - 1;

    for ( ; ; )
    {
      for (l = en; l > low; l--)             /* search for small      */
        if ( ABS(h[l][l-1]) <=               /* subdiagonal element   */
             MACH_EPS * (ABS(h[l-1][l-1]) + ABS(h[l][l])) )  break;

      x = h[en][en];
      if (l == en)                           /* found one evalue     */
      {
        wr[en] = h[en][en] = x + t;
        wi[en] = ZERO;
        cnt[en] = iter;
        en--;
        break;
      }

      y = h[na][na];
      w = h[en][na] * h[na][en];

      if (l == na)                           /* found two evalues    */
      {
        p = (y - x) * 0.5;
        q = p * p + w;
        z = SQRT (ABS (q));
        x = h[en][en] = x + t;
        h[na][na] = y + t;
        cnt[en] = -iter;
        cnt[na] = iter;
        if (q >= ZERO)
        {                                    /* real eigenvalues     */
          z = (p < ZERO) ? (p - z) : (p + z);
          wr[na] = x + z;
          wr[en] = s = x - w / z;
          wi[na] = wi[en] = ZERO;
          x = h[en][na];
          r = SQRT (x * x + z * z);

          if (vec)
          {
            p = x / r;
            q = z / r;
            for (j = na; j < n; j++)
            {
              z = h[na][j];
              h[na][j] = q * z + p * h[en][j];
              h[en][j] = q * h[en][j] - p * z;
            }

            for (i = 0; i <= en; i++)
            {
              z = h[i][na];
              h[i][na] = q * z + p * h[i][en];
              h[i][en] = q * h[i][en] - p * z;
            }

            for (i = low; i <= high; i++)
            {
              z = eivec[i][na];
              eivec[i][na] = q * z + p * eivec[i][en];
              eivec[i][en] = q * eivec[i][en] - p * z;
            }
          }  /* end if (vec) */
        }  /* end if (q >= ZERO) */
        else                                 /* pair of complex      */
        {                                    /* conjugate evalues    */
          wr[na] = wr[en] = x + p;
```

```
      wi[na] =   z;
      wi[en] = - z;
    }

    en -= 2;
    break;
  }  /* end if (l == na) */

  if (iter >= MAXIT)
  {
    cnt[en] = MAXIT + 1;
    return (en);                         /* MAXIT Iterations     */
  }

  if ( (iter != 0) && (iter % 10 == 0) )
  {
    t += x;
    for (i = low; i <= en; i++) h[i][i] -= x;
    s = ABS (h[en][na]) + ABS (h[na][en-2]);
    x = y = (REAL)0.75 * s;
    w = - (REAL)0.4375 * s * s;
  }

  iter ++;

  for (m = en - 2; m >= l; m--)
  {
    z = h[m][m];
    r = x - z;
    s = y - z;
    p = ( r * s - w ) / h[m+1][m] + h[m][m+1];
    q = h[m + 1][m + 1] - z - r - s;
    r = h[m + 2][m + 1];
    s = ABS (p) + ABS (q) + ABS (r);
    p /= s;
    q /= s;
    r /= s;
    if (m == l) break;
    if ( ABS (h[m][m-1]) * (ABS (q) + ABS (r)) <=
            MACH_EPS * ABS (p)
            * ( ABS (h[m-1][m-1]) + ABS (z) + ABS (h[m+1][m+1])) )
      break;
  }

  for (i = m + 2; i <= en; i++) h[i][i-2] = ZERO;
  for (i = m + 3; i <= en; i++) h[i][i-3] = ZERO;

  for (k = m; k <= na; k++)
  {
    if (k != m)             /* double  QR step, for rows l to en  */
    {                       /* and columns m to en                */
      p = h[k][k-1];
      q = h[k+1][k-1];
      r = (k != na) ? h[k+2][k-1] : ZERO;
      x = ABS (p) + ABS (q) + ABS (r);
      if (x == ZERO) continue;                   /*  next k        */
      p /= x;
      q /= x;
      r /= x;
    }
    s = SQRT (p * p + q * q + r * r);
    if (p < ZERO) s = -s;

    if (k != m) h[k][k-1] = -s * x;
      else if (l != m)
            h[k][k-1] = -h[k][k-1];
    p += s;
    x = p / s;
    y = q / s;
    z = r / s;
    q /= p;
    r /= p;

    for (j = k; j < n; j++)               /* modify rows          */
    {
      p = h[k][j] + q * h[k+1][j];
      if (k != na)
      {
        p += r * h[k+2][j];
        h[k+2][j] -= p * z;
      }
      h[k+1][j] -= p * y;
      h[k][j]   -= p * x;
    }

    j = (k + 3 < en) ? (k + 3) : en;
    for (i = 0; i <= j; i++)              /* modify columns       */
    {
      p = x * h[i][k] + y * h[i][k+1];
      if (k != na)
      {
        p += z * h[i][k+2];
        h[i][k+2] -= p * r;
      }
      h[i][k+1] -= p * q;
      h[i][k]   -= p;
    }

    if (vec)      /* if eigenvectors are needed ..................*/
    {
      for (i = low; i <= high; i++)
      {
        p = x * eivec[i][k] + y * eivec[i][k+1];
        if (k != na)
        {
          p += z * eivec[i][k+2];
          eivec[i][k+2] -= p * r;
        }
        eivec[i][k+1] -= p * q;
        eivec[i][k]   -= p;
      }
    }
  }    /* end k          */

 }    /* end for ( ; ;) */

 }    /* while (en >= low)                     All evalues found    */

  if (vec)                            /* transform evectors back  */
    if (hqrvec1 (n, low, high, h, wr, wi, eivec)) return (99);
  return (0);
}

static int norm_11       /* normalize eigenvectors to have one norm 1 .*/
/*.IX{norm\unt 1}*/
                (int     n,       /* Dimension of matrix ...........*/
                 REAL *  v[],     /* Matrix with eigenvektors ......*/
                 REAL    wi[]     /* Imaginary parts of evalues ....*/
                )

{
  int  i, j;
  REAL maxi, tr, ti;

  if (n < 1) return (1);

  for (j = 0; j < n; j++)
  {
    if (wi[j] == ZERO)
    {
      maxi = v[0][j];
      for (i = 1; i < n; i++)
        if (ABS (v[i][j]) > ABS (maxi))  maxi = v[i][j];

      if (maxi != ZERO)
      {
        maxi = ONE / maxi;
        for (i = 0; i < n; i++) v[i][j] *= maxi;
      }
    }
    else
    {
      tr = v[0][j];
      ti = v[0][j+1];
      for (i = 1; i < n; i++)
                                        /* if ( comabs (v[i][j], v[i][j+1]) > comabs (tr, ti) ) */
                                        if (cabs(v[i][j] + I * v[i][j+1]) > cabs(tr + I * ti))
                                        {
                                                        tr = v[i][j];
                                                        ti = v[i][j+1];
                                        }

      if (tr != ZERO || ti != ZERO)
                                        for (i = 0; i < n; i++) {
                                                        /* comdiv (v[i][j], v[i][j+1], tr, ti, &v[i][j], &v[i]
[j+1]); */
                                                        REAL complex c;
                                                        c = (v[i][j] + I * v[i][j+1]) / (tr + I * ti);
                                                        v[i][j] = creal(c); v[i][j+1] = cimag(c);
                                        }

      j++;                                        /* raise j by two */
    }
  }
  return (0);
}



static int eigen11        /* Compute all evalues/evectors of a matrix ..*/
/*.IX{eigen}*/
         (
          int     vec,           /* switch for computing evectors ...*/
          int     ortho,         /* orthogonal Hessenberg reduction? */
          int     ev_norm,       /* normalize Eigenvectors? .........*/
          int     n,             /* size of matrix ..................*/
          REAL *  mat[],         /* input matrix ....................*/
          REAL *  eivec[],       /* Eigenvectors ....................*/
          REAL    valre[],       /* real parts of eigenvalues .......*/
          REAL    valim[],       /* imaginary parts of eigenvalues ..*/
          int     cnt[]          /* Iteration counter ...............*/
         )

{
  int i;
  int      low, high, rc;
  REAL     *scale,
           *d = NULL;
  void     *vmblock;

  if (n < 1) return (1);                       /*  n >= 1 ............*/

  if (valre == NULL || valim == NULL || mat == NULL || cnt == NULL)
    return (1);

  for (i = 0; i < n; i++)
    if (mat[i] == NULL) return (1);

  for (i = 0; i < n; i++) cnt[i] = 0;

  if (n == 1)                                  /*  n = 1 ............*/
  {
    eivec[0][0] = ONE;
    valre[0]    = mat[0][0];
    valim[0]    = ZERO;
    return (0);
  }
```

```
 if (vec)
 {
  if (eivec == NULL) return (1);
  for (i = 0; i < n; i++)
    if (eivec[i] == NULL) return (1);
 }

 vmblock = vminit();
 scale = (REAL *)vmalloc(vmblock, VEKTOR, n, 0);
 if (! vmcomplete(vmblock))                /* memory error        */
  return 2;

 if (vec && ortho)                         /* with Eigenvectors    */
 {                                         /* and orthogonal       */
                                           /* Hessenberg reduction? */
  d = (REAL *)vmalloc(vmblock, VEKTOR, n, 0);
  if (! vmcomplete(vmblock))
  {
    vmfree(vmblock);
    return 1;
  }
 }

                                           /* balance mat for nearly */
 rc = balance (n, mat, scale,              /* equal row and column   */
        &low, &high, BASIS);               /* one norms              */
 if (rc)
 {
  vmfree(vmblock);
  return (100 + rc);
 }
 if (ortho)
  rc = orthes(n, low, high, mat, d);
 else
  rc = elmhes (n, low, high, mat, cnt);    /* reduce mat to upper   */
 if (rc)                                   /*  Hessenberg form      */
 {
  vmfree(vmblock);
  return (200 + rc);
 }

 if (vec)                                  /*  initialize eivec     */
 {
  if (ortho)
    rc = orttrans(n, low, high, mat, d, eivec);
  else
    rc = elmtrans (n, low, high, mat, cnt, eivec);
  if (rc)
  {
    vmfree(vmblock);
    return (300 + rc);
  }
 }

 rc = hqr21 (vec, n, low, high, mat,       /*  execute Francis QR   */
      valre, valim, eivec, cnt);           /*  algorithm to obtain  */
 if (rc)                                   /*  eigenvalues          */
 {
  vmfree(vmblock);
  return (400 + rc);
 }

 if (vec)
 {
  rc = balback (n, low, high,              /*  reverse balancing if  */
         scale, eivec);                    /*  eigenvaectors are to  */
  if (rc)                                  /*  be determined         */
  {
    vmfree(vmblock);
    return (500 + rc);
  }
  if (ev_norm)
    rc = norm_11 (n, eivec, valim);        /* normalize eigenvectors */
  if (rc)
  {
    vmfree(vmblock);
    return (600 + rc);
  }
 }
 vmfree(vmblock);                          /* free buffers          */
 return (0);
}

int n_eigeng1(double *_a, int n, double *evalr, double *evali, double *_evec, double *b)
{
            double **a, **evec = 0;
            int i, j, *cnt;
            a = (double**)calloc(n, sizeof(void*));
            if (_evec) evec = (double**)calloc(n, sizeof(void*));
            cnt = (int*)calloc(n, sizeof(int));
    {
    for (i = 0; i < n; ++i) {
      a[i] = _a + i * n;
      if (_evec) evec[i] = _evec + i * n;
    }
  }
            eigen11(_evec? 1 : 0, 0, 1, n, a, evec, evalr, evali, cnt);
            if (_evec) {
                        double tmp;
                        for (j = 0; j < n; ++j) {
                                    tmp = 0.0;

       {

                                    for (i = 0; i < n; ++i) tmp += SQR(evec[i][j]);

       }
                                    tmp = SQRT(tmp);

       {
                                                for (i = 0; i < n; ++i) evec[i][j] /= tmp;
      }
                                    }
                        }
                        free(a); free(evec); free(cnt);
                        return 0;
}
int main(void)
{   int n;
    printf("Enter the order of the matrix:");
    scanf("%d",&n);
            /*static double   a[5][5] = {{1.0, 6.0, -3.0, -1.0, 7.0},

              {8.0, -15.0, 18.0, 5.0, 4.0},

              {-2.0, 11.0, 9.0, 15.0, 20.0},

              {-13.0, 2.0, 21.0, 30.0, -6.0},

              {17.0, 22.0, -5.0, 3.0, 6.0}};*/

            double *b=malloc(n*n*sizeof(double));
            double *b1=malloc(n*n*sizeof(double));

    for(int i=0 ; i<n;i++)
    {
      for(int j=0;j<n;j++)
      {
        *(b+i*n+j)=rand();
*(b1+i*n+j)=*(b+i*n+j);
      }
    }
            double *mem, *evalr, *evali, *evec;
            int i, j;
            mem = (double*)calloc(n * n + 2*n , sizeof(double));
            evec = mem;
            evalr = evec + n*n;
            evali = evalr + n;
//          n_eigeng(a[0], 5, evalr, evali, evec);
            /*n_eigeng(a[0], 5, evalr, evali, 0);
            for (i = 0; i < 5; ++i) {
                        printf("%le + %le J\n", evalr[i], evali[i]);
            }*/
            printf("\n\n");
    clock_t start,end;
    start = clock();
            n_eigeng(b+0, n, evalr, evali, evec,b);
    end=clock();

/*
            for (i = 0; i <= n-1; i++)
                        printf("%13.7e + %13.7e J\n", evalr[i], evali[i]);
            printf("\n");
            for (i = 0; i <= n-1; i++) {
                        for (j = 0; j <= n-1; j++)
                                    printf("%12.6e ", evec[i*5 + j]);
                        printf("\n");
            }
            printf("Values and vectors computed\n");

    printf("\n");*/
    printf("Time taken with parallelization:%f\n",((float)(end - start))/CLOCKS_PER_SEC);
double *mem1, *evalr1, *evali1, *evec1;
            mem1 = (double*)calloc(n * n + 2*n , sizeof(double));
            evec1 = mem;
            evalr1 = evec + n*n;
            evali1= evalr + n;
            start = clock();
            n_eigeng1(b1+0, n, evalr1, evali1, evec1,b1);
            end=clock();
    printf("Time taken without parallelization:%f\n",((float)(end - start))/CLOCKS_PER_SEC);
            free(mem);
            return 0;
}
```

OUTPUT SNAPSHOT:

5)

Solve the normal equation. Change the number of threads and record performance.

CODE:

```
#include<stdio.h>
#include<math.h>
#include<omp.h>
#include<time.h>
#define MAXN 25

float determinant(float [][MAXN], float);
void cofactor(float [][MAXN], float,float []
[MAXN],float [][MAXN],float);
void transpose(float [][MAXN], float [][MAXN],
float,float [][MAXN],float [][MAXN],float);

int findmultiply(float a[][MAXN],float b[]
[MAXN],float c[][MAXN],int m,int n,int p)
{
  int i,j,k;
#pragma omp parallel shared(a,b,c) private(i,j,k)
   {
#pragma omp for  schedule(static)
   for (i=0; i<m; ++i){
      for (j=0; j<n; ++j){
         a[i][j]=0.;
         for (k=0; k<p; ++k){
            a[i][j]=(a[i][j])+((b[i][k])*(c[k][j]));
         }
      }
   }
   }
   return 0;
}

int main()
{
  float a[MAXN][MAXN],b[MAXN]
[MAXN],c[MAXN][MAXN],e[MAXN]
[MAXN],m,n,d;
  int i, j;
  printf("Enter the number of the rows of the Matrix
(that is m) : ");
  scanf("%f", &m);
  printf("Enter the number of the columns of the
Matrix (that is n) : ");
  scanf("%f", &n);


  printf("Enter the elements of A %.0fX%.0f
Matrix : \n", m, n);
  for (i = 0;i < m; i++)
    {
     for (j = 0;j < n; j++)
       {
        scanf("%f", &a[i][j]);
        c[j][i]=a[i][j];//this is the transpose;
       }
    }
  printf("Enter the elements of B %.0fX%.0f Matrix
that is 'm' elements : \n", m,1.0);
  for (i = 0;i < m; i++)
{
scanf("%f",&b[i][0]);
    }
    //m=n;n=n;p=m
clock_t start,end;

start=clock();
  findmultiply(e,c,a,n,n,m);

  d = determinant(e, n);
  if (d == 0)
   printf("\nInverse of Entered Matrix is not possible\
n");
  else
   cofactor(e,n,c,b,m);
end=clock();
printf("Time taken:%f\n",((float)(end -
start))/CLOCKS_PER_SEC);
}

/*For calculating Determinant of the Matrix */
float determinant(float a[MAXN][MAXN], float k)
{
  float s = 1, det = 0, b[MAXN][MAXN];
  int i, j, m, n, c;
  if (k == 1)
    {
     return (a[0][0]);
    }
  else
    {
     det = 0;

       for (c = 0; c < k; c++)
       {
        m = 0;
        n = 0;
        for (i = 0;i < k; i++)
          {
            for (j = 0 ;j < k; j++)
              {
               b[i][j] = 0;
               if (i != 0 && j != c)
                {
                 b[m][n] = a[i][j];
                 if (n < (k - 2))
                  n++;
                 else
```

```
              {
               n = 0;m++;
                }
              }
           }
         }
     det = det + s * (a[0][c] * determinant(b, k - 1));
     s = -1 * s;
       }

   }

   return (det);
}

void cofactor(float num[MAXN][MAXN], float
f,float c[MAXN][MAXN],float  b1[][MAXN],float
border)
{
 float b[MAXN][MAXN], fac[MAXN][MAXN];
 int p, q, m, n, i, j;

 for (q = 0;q < f; q++)
 {
  for (p = 0;p < f; p++)
   {
    m = 0;
    n = 0;
    for (i = 0;i < f; i++)
    {
     for (j = 0;j < f; j++)
      {
       if (i != q && j != p)
       {
        b[m][n] = num[i][j];
        if (n < (f - 2))
         n++;
        else

         {
          n = 0;
          m++;
          }
        }
      }
     }

    fac[q][p] = pow(-1, q + p) * determinant(b, f -
1);
   }
  }
  transpose(num, fac, f,c,b1,border);

}
/*Finding transpose of matrix*/
void transpose(float num[MAXN][MAXN], float
fac[MAXN][MAXN], float r,float atrans[MAXN]
[MAXN],float b1[][MAXN],float border)
{
  int i, j;
  float b[MAXN][MAXN], inverse[MAXN]
[MAXN], d;

  for (i = 0;i < r; i++)
   {
    for (j = 0;j < r; j++)
     {
      b[i][j] = fac[j][i];
      }
   }
  d = determinant(num, r);
  for (i = 0;i < r; i++)
   {
    for (j = 0;j < r; j++)
     {
      inverse[i][j] = b[i][j] / d;
      }
   }
   float midmul[MAXN][MAXN];
   findmultiply(midmul,inverse,atrans,r,border,r);
   float finalans[MAXN][MAXN];
   findmultiply(finalans,midmul,b1,r,1,border);
   for (int i=0;i<r;++i)
    printf("%f\n",finalans[i][0]);
}
```

OUTPUT SNAPSHOT:

```
sreyans@sreyans-VirtualBox:~/Desktop$ gcc agnmt2.c -lm
sreyans@sreyans-VirtualBox:~/Desktop$ ./a.out
Enter the number of the rows of the Matrix (that is m) : 4
Enter the number of the columns of the Matrix (that is n) : 2
Enter the elements of A 4X2 Matrix :
1 -4 1 1 1 2 1 3
Enter the elements of B 4X1 Matrix that is 'm' elements :
4 6 10 8
6.655172
0.689655
Time taken:0.000066
sreyans@sreyans-VirtualBox:~/Desktop$ export omp_num_threads=4
sreyans@sreyans-VirtualBox:~/Desktop$ gcc -fopenmp agnmt2.c -lm
sreyans@sreyans-VirtualBox:~/Desktop$ ./a.out
Enter the number of the rows of the Matrix (that is m) : 4
Enter the number of the columns of the Matrix (that is n) : 2
Enter the elements of A 4X2 Matrix :
1 -4 1 1 1 2 1 3
Enter the elements of B 4X1 Matrix that is 'm' elements :
4 6 10 8
6.655172
0.689655
Time taken:0.000445
```

6)

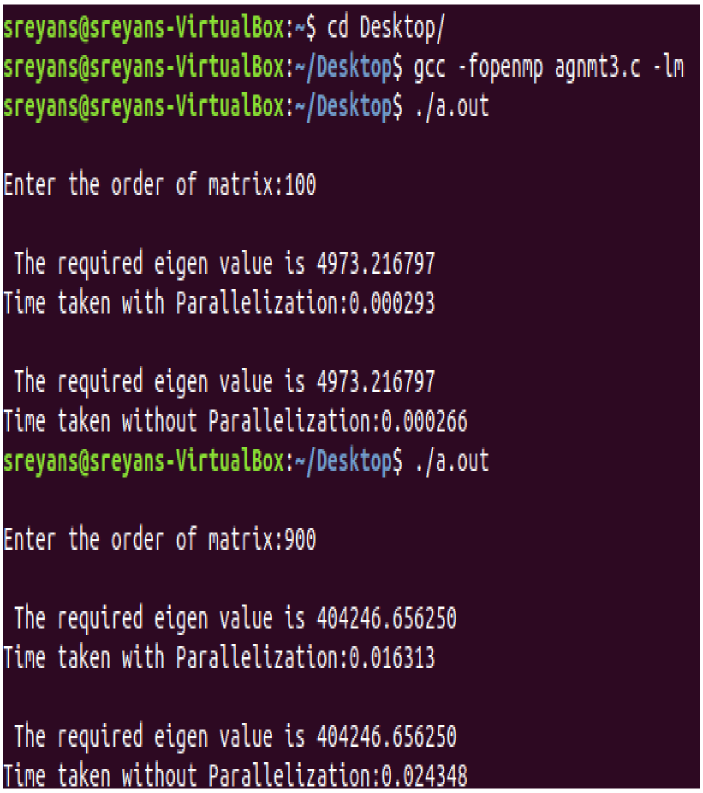
Compute largest Eigen value and corresponding eigen vector using power method.

CODE:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>


void main()
{
   clock_t start,end;
   int i,j,n;
   printf("\nEnter the order of matrix:");
   scanf("%d",&n);
   int A[n+1][n+1]; float
x[n+1],z[n+1],e[n+1],zmax,emax;
   for(i=1; i<=n; i++)
   {
      for(j=1; j<=n; j++)
      {
         A[i][j]=rand()%n;
      }
   }

   x[1]=1;
   for(i=2; i<=n; i++)
   {
      x[i]=0;
   }

   start = clock();
   do
   {
      #pragma omp parallel shared(A,z,x) private(i,j)
      {
         #pragma omp for schedule(static)
         for(i=1; i<=n; i++)
         {
            z[i]=0;
            for(j=1; j<=n; j++)
            {
               z[i]=z[i]+A[i][j]*x[j];
            }
         }
      }
      zmax=fabs(z[1]);
      for(i=2; i<=n; i++)
      {
         if((fabs(z[i]))>zmax)
            zmax=fabs(z[i]);
      }
      #pragma omp parallel shared(z) private(i)
      {
      #pragma omp for schedule(static)
      for(i=1; i<=n; i++){
           z[i]=z[i]/zmax;
      }
      }
      #pragma omp parallel shared(e,z,x) private(i)
      {
      #pragma omp for schedule(static)
      for(i=1; i<=n; i++)
      {
         e[i]=0;
         e[i]=fabs((fabs(z[i]))-(fabs(x[i])));
      }
      }
      emax=e[1];
      for(i=2; i<=n; i++)
      {
         if(e[i]>emax)
            emax=e[i];
      }
      #pragma omp parallel shared(e,z,x) private(i)
      {
         #pragma omp for schedule(static)
         for(i=1; i<=n; i++)
         {
            x[i]=z[i];
         }
      }
   }
   while(emax>0.001);
   end = clock();
   printf("\n The required eigen value is %f",zmax);
if (n<=10){
   printf("\n\nThe required eigen vector is :\n");
   for(i=1; i<=n; i++)
   {
      printf("%f ",z[i]);
   }}
   printf("\n");
   printf("Time taken with Parallelization:%f\n",
((float)(end - start))/CLOCKS_PER_SEC);

x[1]=1;
   for(i=2; i<=n; i++)
   {
      x[i]=0;
   }

   start = clock();
   do
   {


      {


         for(i=1; i<=n; i++)
```

```
        {
          z[i]=0;
          for(j=1; j<=n; j++)
          {
            z[i]=z[i]+A[i][j]*x[j];
          }
        }
      }
      zmax=fabs(z[1]);
      for(i=2; i<=n; i++)
      {
        if((fabs(z[i]))>zmax)
          zmax=fabs(z[i]);
      }
      {

      for(i=1; i<=n; i++)
      {
        z[i]=z[i]/zmax;
      }
      }

      {

      for(i=1; i<=n; i++)
      {
        e[i]=0;
        e[i]=fabs((fabs(z[i]))-(fabs(x[i])));
      }
      }
      emax=e[1];
      for(i=2; i<=n; i++)
      {
        if(e[i]>emax)
          emax=e[i];
      }

      {

        for(i=1; i<=n; i++)
        {
          x[i]=z[i];
        }
      }
    }
    while(emax>0.001);
    end = clock();
    printf("\n The required eigen value is %f",zmax);
if (n<=10){
    printf("\n\nThe required eigen vector is :\n");
    for(i=1; i<=n; i++)
    {
      printf("%f ",z[i]);
    }}
    printf("\n");
    printf("Time taken without Parallelization:%f\n",
((float)(end - start))/CLOCKS_PER_SEC);}
```

OUTPUT SNAPSHOT:



```
sreyans@sreyans-VirtualBox:~$ cd Desktop/
sreyans@sreyans-VirtualBox:~/Desktop$ gcc -fopenmp agnmt3.c -lm
sreyans@sreyans-VirtualBox:~/Desktop$ ./a.out

Enter the order of matrix:100

 The required eigen value is 4973.216797
Time taken with Parallelization:0.000293

 The required eigen value is 4973.216797
Time taken without Parallelization:0.000266
sreyans@sreyans-VirtualBox:~/Desktop$ ./a.out

Enter the order of matrix:900

 The required eigen value is 404246.656250
Time taken with Parallelization:0.016313

 The required eigen value is 404246.656250
Time taken without Parallelization:0.024348
```