Larisa Beilina, Evgenii Karchevskii,
and Mikhail Karchevskii

Numerical Linear Algebra: Theory and Applications

– Appendix with programs –

July 5, 2017

Springer

Contents

Chapter 1
Programs

Matlab Programs

All programs which are located in the archive file Matlab_code.zip are implemented in Matlab and correspond to the following chapters in the Appendix:

- A.1.1: Matlab Programs for Gaussian Elimination using LU Factorization: the main program is
  Poisson2D_LU.m (Example 8.2)
  and functions which are used by this program are located in files:
  DiscretePoisson2D.m (Example 8.2)
  LU_PP.m (Algorithm 8.1)
  ForwSub.m (Algorithm 8.2)
  BackSub.m (Algorithm 8.3)
- A.1.2: Matlab programs for Cholesky decomposition: the main program is
  Poisson2D_Chol.m (Example 8.4.4)
  and functions which are used by this program are located in files:
  DiscretePoisson2D.m (Example 8.2)
  Cholesky.m (Algorithm 8.10)
  ForwSub.m (Algorithm 8.2)
  BackSub.m (Algorithm 8.3)
- A.1.3: Matlab Programs testing Hagers condition estimator: the main program is
  TestHagersCondAlg.m (Example 8.4)
  and function which is used by this program is located in the file:
  HagersAlg.m (Algorithm 8.7)
- A.1.4: Matlab Program FitFunctionNormaleq.m (Example 9.2) to test fitting to a polynomial using method of normal equations.
- A.1.5: Matlab Program FitFunctionQRCGS.m to test fitting to a polynomial using QR decomposition via classical Gram-Schmidt (CGS) orthogonalization procedure (Algorithm 9.4).
- A.1.6: Matlab Program CGS.m performing QR decomposition via classical Gram-Schmidt (CGS) orthogonalization procedure (Algorithm 9.4).

- A.1.7: Matlab Programs to fit a function using linear splines: the main program is
  MainHatFit.m (Example 9.3)
  and functions which are used by this program are in files:
  fihatt.m (Example 9.3)
  LLSChol.m (Algorithm 8.10)
  LLSQR.m (Algorithm 9.4)
  LLSSVD.m
  newtonIR.m (Algorithm 8.8)
- A.1.8: Matlab Programs to fit a function using bellsplines. The main program is MainBellspline.m (Example 9.4), and functions which are used by this program are located in files:
  LLSChol.m (Algorithm 8.10)
  LLSQR.m (Algorithm 9.4)
  LLSSVD.m
  newtonIR.m (Algorithm 8.8)
- A.1.9: Matlab Program PowerM.m (Example 10.1) to test Power Method (Algorithm 10.1).
- A.1.10: Matlab Program InverseIteration.m (Examples 10.5-10.8) to test Inverse Iteration Method (Algorithm 10.2).
- A.1.11: Matlab Program MethodOrtIter.m (Examples 10.9-10.14) to test Method of Orthogonal Iteration (Algorithm 10.3)
- A.1.12: Matlab Program MethodQR_iter.m (Example 10.15) to test Method of QR Iteration (Algorithm 10.4).
- A.1.13: Matlab Program MethodQR_shift.m (Example 10.16) to test Method of QR Iteration with Shifts (Algorithm 10.5).
- A.1.14: Matlab Program MethodQR_Wshift.m (Example 10.16) to test Method of QR Iteration with shifts (Algorithm 10.5) using Wilkinsons Shift.
- A.1.15: Matlab Program HessenbergQR.m (Example 10.17): first is used Hessenberg Reduction (Algorithm 10.6) and then the Method of QR Iteration (Algorithm 10.4).
- A.1.16: Matlab Program testRayleigh.m (Example 11.1) for computation the Rayleigh Quotient (Algorithm 11.1).
  Function which is used by the main program testRayleigh .m is located in the file:
  RayleighQuotient.m (Algorithm 11.1)
- A.1.17: Matlab Program for computation of the algorithm of Divide-and-Conquer: the main program is testDC.m (Example 11.2)
  and function which is used by this program is in the file:
  DivideandConq.m (Algorithm 11.2)
- A.1.18: Matlab Program Bisection.m (Example 11.3, Algorithm 11.4) which finds all eigenvalues of the matrix A ion the input interval [a,b).
  Function which is used by the main program Bisection.m is in the file:
  Negcount.m

- A.1.19: Matlab Program testClassicalJacobi.m (Example 11.4).
  Function which is used by the main program testClassicalJacobi.m is in the file:
  RunJacobi.m (Algorithm 11.7)
- A.1.20: Matlab Program testSVDJacobi.m (Example 11.5). Function which is used by the main program testSVDJacobi.m is located in the file:
  RunSVDJacobi.m (Algorithm 11.14)
- A.1.21: Matlab Programs for solution of the Dirichlet problem for the Poisson's equation in 2D on a square using iterative Jacobi method: the main program is
  Poisson2D_Jacobi.m (Example 12.1, Algorithms 12.1, 12.2)
  and function which is used by this program is located in the file:
  DiscretePoisson2D.m
- A.1.22: Matlab Programs for solution of the Dirichlet problem for the Poisson's equation in 2D on a square using iterative Gauss-Seidel method: the main program is
  Poisson2D_Gauss_Seidel.m (Example 12.2, Algorithms 12.3)
  and function which is used by this program is in the file:
  DiscretePoisson2D.m
- A.1.23: Matlab Programs for solution of the Dirichlet problem for the Poisson's equation in 2D on a square using iterative Gauss-Seidel method with red-black ordering: the main program is
  Poisson2D_Gauss_SeidelRedBlack.m (Example 12.2, Algorithm 12.4)
  and function which is used by this program is:
  DiscretePoisson2D.m
- A.1.24: Matlab Programs for solution of the Dirichlet problem for the Poisson's equation in 2D on a square using iterative SOR method: the main program is
  Poisson2D_SOR.m (Example 12.3, Algorithms 12.5, 12.6)
  and function which is used by this program is in the file:
  DiscretePoisson2D.m
- A.1.25: Matlab Programs for solution of the Dirichlet problem for the Poisson's equation in 2D on a square using Conjugate Gradient method: the main program is
  Poisson2D_ConjugateGrad.m (Example 12.4, Algorithm 12.13)
  and function which is used by this program is in the file:
  DiscretePoisson2D.m
- A.1.26: Matlab Programs for solution of the Dirichlet problem for the Poisson's equation in 2D on a square using Preconditioned Conjugate Gradient method: the main program is
  Poisson2D_PrecConjugateGrad.m (Example 12.6, Algorithm 12.14)
  and function which is used by this program is in the file:
  DiscretePoisson2D.m

C++ and PETSc Programs

All programs which are located in the archive file PETSc_code.zip are implemented in C++ and the software package PETSc (http://www.mcs.anl.gov/petsc/), and correspond to the Chapter 1.27 in Appendix.

PETSc programs implement solution of the Poissons equation using finite difference discretization of the Laplace operator in two dimensions. These programs illustrate Example 12.5: solution of the Dirichlet problem for the Poisson's equation in 2D on a square using different iterative methods for the obtained linear system of equations. These different iterative methods are encoded by numbers 1-7 in the main program Main.cpp in the following order:

- 1 - Jacobis method,
- 2 - Gauss-Seidel method,
- 3 - Successive Overrelaxation method (SOR),
- 4 - Conjugate Gradient method,
- 5 - Conjugate Gradient method (Algorithm 12.13),
- 6 - Preconditioned Conjugate Gradient method,
- 7 - Preconditioned Conjugate Gradient method (Algorithm 12.14).

Methods 1-5 use embedded PETSc functions, and methods 6,7 implement algorithms 12.13, 12.14, respectively, via C++ commands. For example, we can run the program Main.cpp using SOR method as follows:

```
> nohup Main 3 > result.m
```

We have executed programs by running the main program `Main.cpp` using version of PETSc *petsc* − 3.7.4 on 64 bits Red Hat Linux Workstation.

Below we present an example of Makefile used for compilation of PETSc programs:

```
PETSC_ARCH=/sup64/petsc-3.7.4
include ${PETSC_ARCH}/lib/petsc/conf/variables
include ${PETSC_ARCH}/lib/petsc/conf/rules
MPI_INCLUDE = ${PETSC_ARCH}/include/mpiuni
CXX=g++
CXXFLAGS = -Wall -Wextra -g -O0 -c -Iinclude
 -I${PETSC_ARCH}/include -I${MPI_INCLUDE}
LD=g++
LFLAGS=
OBJECTS=Main.o CG.o Create.o DiscretePoisson2D.o
 GaussSeidel.o Jacobi.o PCG.o Solver.o SOR.o
Run=Main
all: $(Run)
$(CXX) $(CXXFLAGS) -o $@ $<
$(Run): $(OBJECTS)
$(LD) $(LFLAGS) $(OBJECTS) $(PETSC_LIB) -o $@
```

After running the program Main.cpp, results will be printed in the file result.m and can be viewed in Matlab using the command surf(result).

1.1 Matlab Programs for Gaussian Elimination using LU Factorization

```
% Program LU_PP.m
% LU factorization with partial pivoting
% This function calculates the permutation matrix P,
% the unit lower triangular matrix L,
% and the nonsingular upper triangular matrix U
% such that LU = PA for a given nonsingular A.


function [L,U,P]=LU_PP(A)

  [n,n]=size(A);
  P=eye(n); L=eye(n); U=A;
  for i=1:n-1
    [pivot m]=max(abs(U(i:n,i)));
    m=m+i-1;
    if m~=i
      % swap rows m and i in P
      temp=P(i,:);
      P(i,:)=P(m,:);
      P(m,:)=temp;
      % swap rows m and i in U
      temp=U(i,:);
      U(i,:)=U(m,:);
      U(m,:)=temp;
      % swap elements L(m,1:i-1) and L(i,1:i-1) in L
      if i >= 2
        temp=L(i,1:i-1);
        L(i,1:i-1)=L(m,1:i-1);
        L(m,1:i-1)=temp;
      end
    end
    L(i+1:n,i)=U(i+1:n,i)/U(i,i);
    U(i+1:n,i+1:n)=U(i+1:n,i+1:n)-L(i+1:n,i)*U(i,i+1:n);
    U(i+1:n,i)=0;
  end
```

```
 % Program ForwSub.m
 % This function computes the vector x, of length n,
 % given Lx = b where L is an n×n nonsingular lower triangular matrix
 % and b is a known vector of length n, by using forward substitution.

function x=ForwSub(L,b)

  s=size(L);
  n=s(1);
  x=zeros(n,1);
  % First, set $x(i)=b(i)$, then subtract the known values.
  % Lastly, divide by diagonal entry $L(i,i)$
  x(1)=b(1)/L(1,1);
  for i=2:n
    x(i)=(b(i)-L(i,1:(i-1))*x(1:(i-1)))/L(i,i);
  end
end
```

```
  % Program BackSub.m
  % This function computes the vector $x$ by backward substitution.
  % We solve $Ux=b$, where $U$ is an $n \times n$ nonsingular upper triangular matrix
  % and $b$ is a known vector of the length $n$, finding the vector $x$.


function x=BackSub(U,b)

  s=size(U);
  n=s(1);
  x=zeros(n,1);

  x(n)=b(n)/U(n,n);
  for i=n-1:-1:1
    x(i)=(b(i)-U(i,(i+1):n)*x((i+1):n))/U(i,i);
  end
end
```

```
% Program Poisson2D_LU.m
% This is the main program for  solution of Poisson's equation
% $-a\triangle u=f$ with Dirichlet b.c. condition $u=0$ in 2D.

close all
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; % 1, 50, 100 choose const. f value
x_0=0.5;
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% -----------------------------------------
% Computing all matrices and vectors
% -----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

% factorize A using LU decomposition with pivoting
[L,U,P]=LU_PP(S);
```

```
%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

%% calculate load vector f

% If f is constant.
% f = f_amp*ones(n^2,1);

% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)=f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

% ----------------------------------------
% Solving the linear system of equations using Gaussian elimination
% ----------------------------------------
% We have system A u = 1/h^2 D L U u = f

% 1. Compute vector of right hand side
%  b = D^(-1)*f given by b(i,j)=f(i,j)/a(i,j)

b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)=f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)
  end
end

% We now have system to solve: 1/h^2 A u = b
% Use first LU decomposition: 1/h^2 L U u = b
% 2. Compute v = L^(-1)*b by forward substitution.

v=ForwSub(L,P*b);

% We now have system 1/h^2 U u = v
% 3. Compute w = U^(-1)*v by backward substitution.
```

```
w=BackSub(U,v);

% 4. We now have system 1/h^2 u = w
% Compute finally solution as:  u=h^2*w
u=h^2*w;

% ----------------------------------------
% Plots and figures.
% ----------------------------------------

% sort the data in u into the mesh-grid, the boundary nodes are zero.
Z = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    Z(i+1,j+1) = u(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

figure(1)
surf(x1,y1,Z) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2) with A = ',num2str(a_amp),...
',  N = ',num2str(n)])

figure(2)
surf(x1,y1,Z) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2) with A = ',num2str(a_amp),...
', N = ',num2str(n)])

% Plotting a(x,y)
Z_a= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
figure(3)
surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
```

```
zlabel('a(x_1,x_2)')
title( ['a(x_1,x_2) with A = ',num2str(a_amp)])

%  plott the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end
figure(4)
surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])
```

```
  % Program DiscretePoisson2D.m
  % The function for 2D discretization  of the Laplace operator △
  % on a square with sign minus: −△
  % Input parameters:
  % n - number of inner nodes, which is assumed to be the same in both
  % the x1- and x2 directions.
```

```
function A=DiscretePoisson2D(n)

  A = zeros(n*n,n*n);

  % Main diagonal
  for i=1:n*n
    A(i,i)=4;
  end

  % 1st and 2nd off-diagonals
  for k=1:n % go through block 1 to n
    for i=1:(n-1)
      A(n*(k-1)+i,n*(k-1)+i+1)=-1; %
      A(n*(k-1)+i+1,n*(k-1)+i)=-1;
    end
  end

  % 3rd and 4th off-diagonals
  for i=1:n*(n-1)
    A(i,i+n)=-1;
    A(i+n,i)=-1;
  end

end
```

1.2 Matlab programs for Cholesky decomposition

```
% Program Poisson2D_Chol.m
% This is the main program for solution of Poisson's equation
% −a△u = f with Dirichlet b.c. u = 0 in 2D on a square using Cholesky decomposition.

close all
%Define input parameters
n=20; % number of inner nodes in one direction.

A_1 = 10; %  amplitude 1 for the rhs
A_2 = 10; %  amplitude 2 for the rhs

h = 1/(n+1); % define step length

% -----------------------------------------
% Computing all matrices and vectors
% -----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

% factorize A=L*L^T using Cholesky decomposition
[L]=Cholesky(S);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for j=1:n
  for i=1:n
    C(i,j) = 1;
  end
end

%% compute load vector f

f=zeros(n^2,1);
for j=1:n
  for i=1:n
    f(n*(i-1)+j)= A_1*exp(-((i*h-0.25)^2/0.02...
    +(j*h-0.25)^2/0.02))+ A_2*exp(-((i*h-0.75)^2/0.02...
    +(j*h-0.75)^2/0.02));
  end
end

% -----------------------------------------
% Solving the linear system of equations using Gaussian elimination
% -----------------------------------------
% We have system A u = 1/h^2 (C*L*L^T) u = f

% 1. Compute vector of right hand side
% as b(i,j)=f(i,j)/a(i,j)
```

```
b=zeros(n^2,1);
for j=1:n
  for i=1:n
    b(n*(i-1)+j)=f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C

  end
end

% We now have system to solve: 1/h^2 A u = b
% Use first LU decomposition: 1/h^2 (L L^T)  u = b
% 2. Compute v = L^(-1)*b by forward substitution.

v=ForwSub(L,b);

% We now have system 1/h^2 L^T u = v
% 3. Compute w = L^T^(-1)*v by backward substitution.

w=BackSub(L',v);

% 4. We now have system 1/h^2 u = w
% Compute finally solution as:  u=h^2*w
u=h^2*w;

% ----------------------------------------
% Plots and figures.
% ----------------------------------------

% sort the data in u into the mesh-grid, the boundary nodes are zero.
Z = zeros(n+2,n+2);
for j=1:n
  for i=1:n
    Z(i+1,j+1) = u(n*(i-1)+j);
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

figure(1)
surf(x1,y1,Z) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2) with N = ',num2str(n)])

figure(2)
surf(x1,y1,Z) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
```

```
title( ['u(x_1,x_2) with N = ',num2str(n)])


  % Program Cholesky.m
  % This function factorizes square matrix A, assuming that A is s.p.d. matrix,
  % into A=LL', where L' is the transpose
  % of L, and L is non-singular lower triangular matrix.


function L=Cholesky(A)

  s=size(A);
  n=s(1);
  L=zeros(n);

  % diagonal elements i=j
  % a_jj=v_j*v_j'=l_j1^2+l_j2^2+...+l_jj^2 (sum has j-terms)

  % elements below diagonal, i>j
  % a_ij=v_i*v_j'=l_i1 l_j1 + l_i2 l_j2 + ... + l_ij l_jj (sum has j terms)

  for j=1:n % go through column 1 to n
    % Compute diagonal elements, i=j
    L(j,j)=A(j,j);
    for k=1:(j-1)
      L(j,j)=L(j,j)-L(j,k)^2;
    end
    L(j,j)=L(j,j)^(1/2);
    % Compute elements below diagonal, i>j
    for i=(j+1):n
      L(i,j)=A(i,j);
      for k=1:(j-1)
        L(i,j)=L(i,j)-L(i,k)*L(j,k);
      end
      L(i,j)=L(i,j)/L(j,j);
    end
  end
end
```

1.3 Matlab Programs testing Hager's condition estimator

```
% ----------------------------------------
% Program TestHagersCondAlg.m
% Hager's algorithm: for the input matrix A
% the function HagerCond(A) computes
% the lower bound of the one-norm of the matrix A.
% ----------------------------------------

% First we generate some random  symmetric  matrices
```

```
n=5;
A=zeros(n);

for i=1:n
  for j=1:n
    tal = rand*30;
    A(i,i)=rand*20;
    A(i,j)=tal;
    A(j,i)=tal;
  end
end
disp(' The input matrix A  is:');

A

disp(' The computed lower bound of ||A||_1  is:');
HagersEst =  HagersAlg(A)

disp('  result of norm(A,1) is:');
norm(A,1)


% ----------------------------------------
% Program HagersAlg.m
% Run Hager's algorithm.
% ----------------------------------------

function [LowerBound] = HagersAlg(B)

  x=(1/length(B))*ones(length(B),1);

  iter=1;
  while iter < 1000
    w=B*x; xi=sign(w); z = B'*xi;
    if max(abs(z)) <= z'*x
      break
    else
      x= (max(abs(z))== abs(z));
    end
    iter = iter + 1;
  end
  LowerBound = norm(w,1);
end
```

1.4 Matlab Program `FitFunctionNormaleq.m` to test fitting to a polynomial using method of normal equations

```
% ------------------------------------------------------------------------
%   Program FitFunctionNormaleq.m
%   Solution of least squares  problem  $min_x ||Ax - y||_2$
%   using the method of normal equations.
%   Matrix  A is constructed as a Vandermonde matrix.
%   Program performs fitting to the function y = sin(pi*x/5) + x/5
% -----------------------------------------------------------------------

d=5;  % degree of the polynomial
m=10;%number of discretization points or rows in the matrix A

x=zeros(1,m);
y=zeros(1,m);
A=[];
for i=1:1:m
  x = linspace(-10.0,10.0,m);
  %  exact function which should be approximated
  y(i)= sin(pi*x(i)/5) + x(i)/5;
end

% construction of a Vamdermonde matrix

for i=1:1:m
  for j=1:1:d+1
    A(i,j)=power(x(i),j-1);
  end
end

% computing the right hand side in the method of normal equations
c=A'*y';

% computing matrix in the left hand side in the method of normal equations
C=A'*A;

l=zeros(d+1);

% solution of the normal equation using Cholesky decomposition

for j=1:1:d+1
  s1=0;
  for k=1:1:j-1
    s1=s1+l(j,k)*l(j,k);
  end
  l(j,j)=(C(j,j)-s1)^(1/2);
  for i=j+1:1:d+1
    s2=0;
    for k=1:1:j-1
      s2=s2+l(i,k)*l(j,k);
    end
    l(i,j)=(C(i,j)-s2)/l(j,j);
  end
end
for i=1:1:d+1
  for k=1:1:i-1
    c(i)=c(i)-c(k)*l(i,k);
  end
  c(i)=c(i)/l(i,i);
```

```
end
for i=d+1:-1:1
  for k=d+1:-1:i+1
    c(i)=c(i)-c(k)*l(k,i);
  end
  c(i)=c(i)/l(i,i);
end

figure(1)
plot(x,y,'o- r', 'linewidth',1)
hold on

% compute approximation to this exact polynomial with comp. coefficients c

approx = A*c;
plot(x,approx,'*- b', 'linewidth',1)
hold off

str_xlabel = ['poly.degree d=', num2str(d)];

legend('exact  sin(pi*x(i)/5) + x(i)/5',str_xlabel);

xlabel('x')

% computation of the relative error as
%  norm(approx. value - true value) / norm(true value)
e1=norm(y'- approx)/norm(y')
```

1.5 Matlab Program `FitFunctionQRCGS.m` to test fitting to a polynomial using QR decomposition via CGS

```
% ----------------------------------------------------------------------
%   Program FitFunctionQRCGS.m
%   Solution of least squares  problem $min_x ||Ax - y||_2$
%   using QR decomposition. QR decomposition is performed via classical
%   Gram-Schmidt  (CGM) orthogonalization procedure.
%   Matrix  $A$ is constructed as a Vandermonde matrix.
%   Program performs fitting to the function $y = \sin(pi*x/5) + x/5$
% ----------------------------------------------------------------------

d=5;  % degree of polynomial
m=10;%number of discretization points or rows in the matrix A
p=ones(1,d+1);
x=zeros(1,m);
y=zeros(1,m);
A=[];
for i=1:1:m
  x = linspace(-10.0,10.0,m);
  %  exact function which should be approximated
  y(i)= sin(pi*x(i)/5) + x(i)/5;
end
```

```
% construction of a Vamdermonde matrix
for i=1:1:m
  for j=1:1:d+1
    A(i,j)=power(x(i),j-1);
  end
end

q=[];
r=[];

%QR decomposition via CGM

for i=1:1:d+1
  q(:,i)=A(:,i);
  for j=1:1:i-1
    r(j,i)=q(:,j)'*A(:,i);
    q(:,i)=q(:,i)-r(j,i)*q(:,j);
  end
  r(i,i)=norm(q(:,i));
  q(:,i)=q(:,i)/r(i,i);
end
b=[];
b=q'*y';
for i=d+1:-1:1
  for k=d+1:-1:i+1
    b(i)=b(i)-b(k)*r(i,k);
  end
  b(i)=b(i)/r(i,i);
end

figure(1)
plot(x,y,'o- r', 'linewidth',1)
hold on

% compute approximation to this exact polynomial with comp. coefficients b

approx = A*b;
plot(x,approx,'*- b', 'linewidth',1)
hold off

str_xlabel = ['poly.degree d=', num2str(d)];

legend('exact  sin(pi*x(i)/5) + x(i)/5',str_xlabel );

xlabel('x')

% computation of the relative error as
%  norm(approx. value - true value) / norm(true value)
e1=norm(y'- approx)/norm(y')
```

1.6 Matlab Program `CGS.m` performing QR decomposition via CGS

```
% -------------------------------------------------------------
%  Program CGS.m
%  Classical Gram-Schmidt (CGS) orthogonalization process
%  and solution of the linear least square problem  using CGS.
% --------------------------------------------------------------

% size of our matrix A is m-by-n
m= 6;
n=3;

% vector of the right hand side
y=zeros(1,m);

A=[1,0,0;
0,1,0;
0,0,1;
-1, 1,0;
-1,0,1;
0,-1,1];

y = [1237,1941,2417,711,1177,475];

% allocate matrices q and r for QR decomposition

q=[];
r=[];

%QR decomposition using classical Gram-Schmidt orthogonalization
for k=1:1:n
  q(:,k)=A(:,k);
  for j=1:1:k-1
    r(j,k)=q(:,j)'*A(:,k);
    q(:,k)=q(:,k)-r(j,k)*q(:,j);
  end
  r(k,k)=norm(q(:,k));
  q(:,k)=q(:,k)/r(k,k);
end

%compute solution of the system Ax = QR x =   y
% by backward substitution:  R x = Q^T y

b=[];

% compute right hand side Q^T y
b=q'*y';

% perform backward substitution to get solution x = R^(-1)  Q^T y
% obtain solution in b
```

```
for i=n:-1:1
  for k=n:-1:i+1
    b(i)=b(i)-b(k)*r(i,k);
  end
  b(i)=b(i)/r(i,i);
end
```

1.7 Matlab Programs to fit a function using linear splines. The main program is `MainHatFit.m`

```
% --------------------------------------------------------------------------
%   Program MainHatFit.m
%   Solution of the least squares problem $\min_x ||Ax - y||_2$
%   using the method of normal equations, QR decomposition
%   and SVD decomposition.
%   Matrix $A$ is constructed using linear splines.
%   The program performs fitting to the function $y = \sin(\pi * x/5) + x/5$
% ---------------------------------------------------------------------------

clc
clear
clf
format long
close all

% Define the number of measurements or data points.
% It is also the number of columns in matrix $A$.
m = 100;

% the number of junction points
K = 5;

x = linspace(-10, 10.0, m)';
T = linspace(-10, 10.0, K)';

% function which we want to fit
b = sin(pi*x/5) + x/5;

A = zeros(m, K);

% construct matrix A using linear splines
for k = 1:K
  A(:,k) = fihatt(k, x, T);
end
% compute condition number of A
cond(A)

% solution of linear system $Ax = b$ by different methods

% using method of normal equations
xHatChol = LLSChol(A, b);
```

```
% using QR decomposition of A
xHatQR = LLSQR(A, b);

% using SVD decomposition of A
xHatSVD = LLSSVD(A, b);

disp(' Computed relative error ')
disp('      Method of normal eq.           QR               SVD')
disp('')

disp([norm(A*xHatChol-b)/norm(b) norm(A*xHatQR-b)/norm(b)   ...
norm(A*xHatSVD-b)/norm(b)])

% Method of iterative refinement via Newton's method

tol = 0.07;
refinedC = newtonIR(A, xHatChol, b, tol);
refinedQ = newtonIR(A, xHatQR, b, tol);
refinedS = newtonIR(A, xHatSVD, b, tol);

disp('Computed relative error after iterative refinement via Newton method ')
disp('    Method of normal eq.             QR              SVD')
disp('')

disp([norm(A*refinedC-b)/norm(b) norm(A*refinedQ-b)/norm(b) ...
 norm(A*refinedS-b)/norm(b)])

% Plot exact and computed functions

% choose the number of points to plot solution
x = linspace(-10, 10.0, 100)';
b = (sin(pi*x/5) + x/5);
A = zeros(100, K);

for k = 1:K
  A(:,k) = fihatt(k, x, T);
end

% choose method to be plotted

%method = 'cholesky';
%method = 'refinedcholesky';
%method = 'qr';
%method = 'refinedqr';
%method = 'svd';
method = 'refinedsvd';

switch lower(method)
          case 'cholesky'
  % Here, A is constructed by linear splines, approximated function is computed
  % via  the method of normal equations (Cholesky decomposition)
  solution = A*xHatChol;
case 'refinedcholesky'
```

```
  % Here, A is constructed by linear splines, approximated function is computed
  % via iterative refinement of the Cholesky-solution through the Newton method
  solution = A*refinedC;
case 'qr'
  % Here, A is constructed by linear splines, approximated function is computed
  % via QR decomposition
  solution = A*xHatQR;
case 'refinedqr'
  % Here, A is constructed by linear splines, approximated function is computed
  % via iterative refinement of the QR-solution through the Newton method
  solution = A*refinedQ;
case 'svd'
  % Here, A is constructed by linear splines, approximated function is computed
  % via SVD decomposition
  solution = A*xHatSVD;
case 'refinedsvd'
  % Here, A is constructed by linear splines, approximated function is computed
  % via iterative refinement of the SVD-solution through the Newton method
  solution = A*refinedS;
otherwise
  disp('Unknown method')
end

figure (1)
plot(x, b, 'o r', 'linewidth', 1)
hold on
plot(x, solution, ' - b', 'linewidth', 2)
legend('function', 'approx');
figure('Name', 'Hat functions')
plot(x, A, 'k')



% ----------------------------------------------------------------
%   Program fihatt.m
%   Construction of columns in matrix A  using linear splines.
%   Input arguments: T - column vector with junction points,
%   x are measurement ponts (discretization points).
%   Returns column with number k to the matrix A.
% -----------------------------------------------------------------

function f=fihatt(k,x,T)

  h=diff(T);
  N=length(T);
  f=zeros(size(x));
  if k>1
    I=find(x>=T(k-1) & x<=T(k));
    f(I)=(x(I)-T(k-1))/h(k-1);
  end
  if k<N
    I=find(x>=T(k) & x<=T(k+1));
    f(I)=(T(k+1)-x(I))/h(k);
  end
```

```
% ---------------------------------------------------------------------------
%  Program newtonIR.m
%  Iterative refinement using Newton's method.
%   Matrix A is m-by-n, m > n, the rhs vector b is of the size n.
% ----------------------------------------------------------------------------

function w=newtonIR(A,x,b,tol)

  relative_error=1;
  iter = 0;

  while relative_error > tol

    %compute residual
    r = A*x-b;
    d=A\r;
    x=x-d;
    iter = iter+1
    relative_error = norm(A*x - b)/norm(b)

    % here we introduce the maximal number of iterations
    % in Newton's method: if the relative error
    % is not rediced -  we terminate computations

    if iter  > 100
      break
    end
  end
  w=x;



% ------------------------------------------------------------------
%   Program LLSChol.m
%   Solution of the system of linear equations A^T Ax = A^T b
%   using Cholesky factorization of A^T A.
%   Matrix A is m-by-n, m > n, the vector of the rhs b
%   is of the size n.
% -------------------------------------------------------------------

function x=LLSChol(A,b)

  ATb=A'*b;
  ATA=A'*A;
  n=length(A(1,:));
  lowerChol=zeros(n);

  %Cholesky factorization
  for j=1:1:n
    s1=0;
    for k=1:1:j-1
      s1=s1+lowerChol(j,k)*lowerChol(j,k);
    end
    lowerChol(j,j)=(ATA(j,j)-s1)^(1/2);
    for i=j+1:1:n
      s2=0;
      for k=1:1:j-1
        s2=s2+lowerChol(i,k)*lowerChol(j,k);
      end
      lowerChol(i,j)=(ATA(i,j)-s2)/lowerChol(j,j);
    end
  end

  % Solver for LL^T x = A^Tb:
  % Define z=L^Tx, then solve
  % Lz=A^T b to find z.
  % After by known z we get x.

  % forward substitution Lz=A^T b to obtain z

  for i=1:1:n
    for k=1:1:i-1
      ATb(i)=ATb(i)-ATb(k)*lowerChol(i,k);
    end
    ATb(i)=ATb(i)/lowerChol(i,i);
  end

  % Solution of L^Tx=z , backward substitution

  for i=n:-1:1
    for k=n:-1:i+1
      ATb(i)=ATb(i)-ATb(k)*lowerChol(k,i);
    end
    ATb(i)=ATb(i)/lowerChol(i,i);
  end

  % Obtained solution
  x=ATb;


% --------------------------------------------------------------------------
%  Program LLSQR.m
%  Solution of the system of linear equations Ax=b via
%  QR decomposition of a matrix A.
%  Matrix A is m-by-n, m>n, the rhs vector b is of the size n.
%  QR decomposition of A is done via classical
%  Gram-Schmidt  (CGM) orthogonalization procedure.
% ---------------------------------------------------------------------------

function x=LLSQR(A,b)

  n=length(A(1,:));
  q=[];
  r=[];

  for i=1:1:n
    q(:,i)=A(:,i);
    for j=1:1:i-1
```

```
      r(j,i)=q(:,j)'*A(:,i);
      q(:,i)=q(:,i)-r(j,i)*q(:,j);
    end
    r(i,i)=norm(q(:,i));
    q(:,i)=q(:,i)/r(i,i);
  end

  % compute right hand side in the equation
  Rx=q'*b;

  % compute solution via backward substitution
  for i=n:-1:1
    for k=n:-1:i+1
      Rx(i)=Rx(i)-Rx(k)*r(i,k);
    end
    Rx(i)=Rx(i)/r(i,i);
  end

  x = Rx;


% -------------------------------------------------------------------------
%   Program LLSSVD.m
%   Solution of the system of linear equations Ax=b via
%   SVD decomposition of a matrix A.
%   SVD decomposition is done via matlab function svd.
%   Matrix A is m-by-n, m>n, the rhs vector b is of the size n.
% ---------------------------------------------------------------------------

function x=LLSSVD(A,b)

  [U, S, V]=svd(A);

  UTb=U'*b;

  % choose tolerance
  tol=max(size(A))*eps(S(1,1));
  s=diag(S);
  n=length(A(1,:));

  % compute number of singular values > tol
  r=sum(s > tol);

  w=[(UTb(1:r)./s(1:r))' zeros(1,n-r)]';

  x=V*w;
```

1.8 Matlab Programs to fit a function using bellsplines. The main program is `MainBellspline.m`. Functions `newtonIR.m`, `LLSChol.m`, `LLSQR.m`, `LLSSVD.m` are the same as in section 1.7.

```
% -----------------------------------------------------------------------
%   Program MainBellspline.m
%   Solution of least squares  problem   min_x||Ax-y||_2
%   using the method of normal equations, QR decomposition
%   and SVD decomposition.
%   Matrix  A is constructed using bellsplines.
%   Program performs fitting to the function y = sin(pi*x/5)+x/5.
% ------------------------------------------------------------------------

clc
clear
clf
close all
format short

% input interval on which we fit the function
interval=10;

% junction points

T=linspace(-10,interval,7)';

% Define number of measurement points m
m=30;
x=linspace(-10,interval,m)';

%exact function to be fitted
b=sin(pi*x/5) +x/5;

% construct matrix A with bellsplines
%Number of bellsplines should be number of junction points +2

A=fbell(x,T);

%solution of system Ax = b using different methods for solution
% of least squares problem.
tic
% use method of normal equations
xHatChol = LLSChol(A,b);
toc
tic
%use SVD decomposition of A
xHatSVD = LLSSVD(A,b);
toc
tic
% use QR decomposition of A
xHatQR = LLSQR(A,b);
toc

% compute condition number of A
cond(A)

% use iterative refinement of the obtained solution
%  via Newton's method
% choose tolerance in Newton's method
tol =0.2;

y=  newtonIR(A,xHatChol,b,tol);
y1= newtonIR(A,xHatQR,b,tol);
y2= newtonIR(A,xHatSVD,b,tol);

% compute relative errors

eC=norm(A*xHatChol-b)/norm(b);
eS=norm(A*xHatSVD-b)/norm(b);
eQ=norm(A*xHatQR-b)/norm(b);

disp(' --------------Computed relative errors -------------------- ')
disp('      Method of normal eq.           QR               SVD')
disp('')

disp([eC eS eQ ])

disp('Computed relative errors after iterative refinement via Newton method ')
disp('    Method of normal eq.             QR              SVD')
disp('')

disp([norm(A*y-b)/norm(b) norm(A*y1-b)/norm(b) norm(A*y2-b)/norm(b)])

% Plot results

figure(1)
%plot(t,A,'linewidth',2)

plot(x,A,'linewidth',2)

m =size(A,2);
str_xlabel = [' number of bellsplines=', num2str(m)];
title(str_xlabel)

figure('Name','Cholesky')
title('Cholesky')
plot(x,b,'o- r', 'linewidth',2)
hold on
plot(x,A*xHatChol,' *- b', 'linewidth',2)
legend('exact  ', 'B-spline degree 3, Cholesky');

figure('Name','QR')
plot(x,b,'o- r', 'linewidth',2)
hold on
plot(x,A*xHatQR,'* - b', 'linewidth',2)
legend('exact ', 'B-spline degree 3, QR');

figure('Name','SVD')
title('SVD')
plot(x,b,'o- r', 'linewidth',2)
hold on
plot(x,A*xHatSVD,'*- b', 'linewidth',2)
```

```
legend('exact ', 'B-spline degree 3, SVD');


% ----------------------------------------------------------------------
%   Program fbell.m
%   Matrix  B is constructed using  bellsplines.
%   Input arguments: T - column vector with junction points,
%   x are measurement ponts (discretization points).
% ----------------------------------------------------------------------

function B=fbell(x,T)

  m=length(x);
  N=length(T);
  epsi=1e-14;

  %construct  N+6 column vector
  a=[T(1)*[1 1 1]'; T; T(N)*(1+epsi)*[1 1 1]'];
  n=N+5;
  C=zeros(m,n);
  for k=1:n
    I=find(x>=a(k) & x<a(k+1));
    if ~isempty(I)
      C(I,k)=1;
    end
  end
  for j=1:3
    B=zeros(m, n-j);
    for k=1:n-j
      d1=(a(k+j)-a(k));
      if abs(d1)<=epsi
        d1=1;
      end
      d2=(a(k+j+1)-a(k+1));
      if abs(d2)<=epsi
        d2=1;
      end
      B(:,k)=(x-a(k)).*C(:,k)/d1 + (a(k+j+1)-x).*C(:,k+1)/d2;
    end
    C=B;
  end
```

1.9 Matlab Program `PowerM.m` to Test Power Method

```
% -----------------------------------------
%  Program  Power.m
%  Power method
% -----------------------------------------
clc
clear all
close all
```

```
eps = 1e-7;
fig = figure;

for i =1:4
  if(i==1)
    % Matrix not diagonalizable
    % n=2;
    % A =[0 10;0 0];
    % Matrix has two real eigenvalues with the same sign
    n=3;
    A =[5 0 0;0 2 0;0 0 -5];
  elseif (i==2)
    % Matrix has four real eigenvalues with the same sign
    n =4;
    A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
  elseif (i==4)
    n =2;
    A =[7 -2;3 0];
    n=5;
    A=rand(n);

  end

  % get  reference values of eigenvalues
  exact_lambda = eig(A);

  % set initial guess for the eigenvector x0
  x0=rand(n,1);
  x0=x0/norm(x0);
  lambda0 = inf ;
  % lambda1 = 0;
  lambdavec =[];
  % counter for number of iterations
  count =1;
  % main loop in the power method

  while (count <1000)

    y1=A*x0;

    %  compute approximate eigenvector

    x1=y1/norm(y1);

    %  compute approximate eigenvalue
    lambda1 = transpose(x1)*A*x1;

    lambdavec(count)= lambda1 ;
    x0=x1;
    if(abs(lambda1 - lambda0 )<eps )
```

```
      break ;
    end
    lambda0 = lambda1;
    count = count + 1;

  end

  % Print  computed eigenvalue
  str =['Computed eigenvalue:' num2str(lambda1)];
  str=[str, ', Exact eigenvalues:' num2str(exact_lambda',2)];
  subplot (2 ,2,i)
  plot (lambdavec,  'LineWidth',3)
  xlabel('Number of iterations in Power method')
  ylabel('Computed eigenvalue')
  title (str, 'fontsize',10)

end
```

1.10 Matlab Program `InverseIteration.m` to Test Inverse Iteration Method

```
% ------------------------------------------------------------------------
%  Program InverseIteration.m
%  Implements Inverse Iteration or Inverse Power method.
%  Computes eigenvalue closest to sigma  and corresponding eigenvector.
% ------------------------------------------------------------------------
clc
clear all
close all
eps = 1e-17;
fig = figure;

for i =1:4
  if(i==1)
    % Matrix not diagonalizable
    n=2;
    A =[0 10;0 0];
    % Matrix has two real eigenvalues with the same sign
    % n=3;
    % A =[5 0 0;0 2 0;0 0 -5];
  elseif (i==2)
    % Matrix has four real eigenvalues with the same sign
    n =4;
    A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
  elseif (i==4)
```

```
    % n =2;
    % A =[7 -2;3 0];
    n=5;
    A=rand(5,5);
  end

  % get  reference values of eigenvalues
  exact_lambda = eig(A);

  %make orthogonalization
  Q=orth(rand(n,n));

  A= Q'*A*Q;

  % set initial guess for the eigenvector x0
  x0=rand(n,1);
  x0=x0/norm(x0);
  lambda0 = inf;
  % choose a shift:  should be choosen as closest to the desired eigenvalue
  sigma=10;
  % lambda1 = 0;
  lambdavec =[];
  count =1;
  % main loop in the power method
  while (count <1000)
    A_shift = A - sigma*eye(size(A));
    y1= inv(A_shift)*x0;
    x1=y1/norm(y1);
    lambda1 = transpose(x1)*A*x1;
    lambdavec(count)= lambda1 ;
    x0=x1;
    if(abs(lambda1 - lambda0 )<eps )
      break ;
    end
    lambda0 = lambda1 ;
    count = count + 1;
  end

  % Print  computed and exact  eigenvalue
  str =['Example ' num2str(i)];
  str =[str, '. Comp. eig.:' num2str(lambda1)];
  str=[str, ', Ref. eig.:' num2str(exact_lambda',2)];
  subplot (2 ,2,i)
  plot (lambdavec,  'LineWidth',3)
  xlabel(' Number of iterations in Inverse Power method')
  ylabel('Computed eigenvalues')
  title (str, 'fontsize',12)

end
```

1.11 Matlab Program `MethodOrtIter.m` to Test Method of Orthogonal Iteration

```
% ---------------------------------------------------------
%  Program MethodOrtIter.m
%  Implements method of Orthogonal Iteration.
%  Let initial $Q = I$. Our goal is
%  compute eigenvalues and eigenvectors of the matrix
%  $A$ of the size $n$-by-$n$.
% ---------------------------------------------------------
clc
clear all
close all
eps = 1e-07;
fig = figure;

N=10;
for i =1:6
  if(i==1)
    n=N;
    A=hilb(N);
  elseif (i==2)
    n=20;
    A=hilb(20);
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
  elseif (i==4)
    % Matrix has four real eigenvalues
    n =4;
    A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
  elseif (i==5)
    n =5;
    %
    A=[3,7,8,9,12;5,-7,4,-7,8;1,1,-1,1,-1;4,3,2,1,7;9,3,2,5,4];
  elseif (i==6)
    n=N;
    A= rand(N,N);
  end

  lambda0= inf(n,1);
  count = 1;
  iter =1;

  % get  exact eigenvalues in sorted order
  exact_lambda = sort(eig(A));
  %*********************************************************
  %%% Method of orthogonal iteration

  Q = eye(n);
  for k = 1:100
    Y = A*Q;
    [Q,R] = qr(Y);
    % end
    T=Q'*A*Q;
    %end
    %****************************************

    % %%%%%%%%% Find eigenvalues from Real Schur block
    j =2; count =1;
    eigs = zeros(1,n);
    while (j <=n)
      %real eigenvalues
      if(abs(T(j,j-1)) < 1e-3)
        eigs(j-1) =T(j -1,j -1);
        count= j -1;
      else
        % Complex  eigenvalues
        eigs(j-1: j)= eig(T(j -1:j,j -1:j));
        count =j;
        j=j +1;
      end
      j=j +1;
    end
    if(count < length(eigs))
      eigs(n)=T(n,n);
    end
    %********************************

    computed_lambda = sort(eigs);
    computed_lambda = computed_lambda';

    if(norm(abs(computed_lambda - lambda0 ))<eps )
      break ;
    end
    lambda0 = computed_lambda ;
    iter = iter + 1;

  end

  %****************************************
  str =['Comp. eig.:' num2str(computed_lambda')];
  str=[str, ', Ex. eig.:' num2str(exact_lambda',2)];
  str_xlabel = ['Example ',num2str(i), ...
'.  Nr. of it. in method of Orth. it.:', num2str(iter)];

  subplot (3,2,i)

  plot (exact_lambda,'o b','LineWidth',2,'Markersize',10)
  hold on
  plot (computed_lambda,'+ r','LineWidth',2, 'Markersize',10)
  % xlabel(str, 'fontsize',10)

  xlabel('Real part of eigenvalues');
  ylabel('Imag. part of eigenvalues');

  exact_lambda
  computed_lambda
  legend('Exact eigenvalues','Computed eigenvalues')

  title(str_xlabel,'fontsize',12)

end
```

1.12 Matlab Program `MethodQR_iter.m` to Test Method of QR Iteration

```
% -----------------------------------------------------------------------
%  Program MethodQR_iter.m
%  Method of  QR  iteration. This method reorganizes the method of
%  orthogonal iteration and is more efficient since it does not requires
%  assumption about distinct absolute eigenvalues of A.
% ------------------------------------------------------------------------
clc
%  clear all
%  close all
eps = 1e-07;
fig = figure;
N=10;
for i =1:6
  if(i==1)
    n=N;
    A=hilb(N);
  elseif (i==2)
    n=20;
    A=hilb(20);
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
  elseif (i==4)
    % Matrix has four real eigenvalues
    n =4;
    A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
  elseif (i==5)
    n =5;
    %
    A=[3,7,8,9,12;5,-7,4,-7,8;1,1,-1,1,-1;4,3,2,1,7;9,3,2,5,4];
  elseif (i==6)
    n=N;
    A= rand(N,N);
  end
  lambda0= inf(n,1);
  count = 1;
  iter =1;

  % get  exact eigenvalues in sorted order
  exact_lambda = sort(eig(A));
  %*********************************************************
  %%% Method of QR iteration

  for k = 1:100

    [Q,R] = qr(A);
    A= R*Q;

    %****************************************
    % %%%%%%%%% Find eigenvalues from Real Schur block
    j =2; count =1;
    eigs = zeros(1,n);
    while (j <=n)
      %real eigenvalues
      if(abs(A(j,j-1)) < 1e-10)
        eigs(j-1) =A(j -1,j -1);
        count= j -1;
      else
        % Complex  eigenvalues
        eigs(j-1: j)= eig(A(j -1:j,j -1:j));
        count =j;
        j=j +1;
      end
      j=j +1;
    end
    if(count < length(eigs))
      eigs(n)=A(n,n);
    end
    %********************************

    computed_lambda = sort(eigs)';

    if(norm(abs(computed_lambda - lambda0 ))<eps )
      break ;
    end
    lambda0 = computed_lambda ;
    iter = iter + 1;
  end

  %****************************************
  str =['Comp. eig.:' num2str(computed_lambda')];
  str=[str, ', Ex. eig.:' num2str(exact_lambda',2)];
  str_xlabel = ['Example ',num2str(i), ...
'.  Nr. of it. in method of QR it.', num2str(iter)];

  subplot (3,2,i)

  plot (exact_lambda,'o b','LineWidth',2,'Markersize',10)
  hold on
  plot (computed_lambda,'+ r','LineWidth',2, 'Markersize',10)

  % xlabel(str, 'fontsize',10)

  xlabel('Real part of eigenvalues');
  ylabel('Imag. part of eigenvalues');

  exact_lambda
  computed_lambda
  legend('Exact eigenvalues','Computed eigenvalues')

  title(str_xlabel,'fontsize',12)
end
```

1.13 Matlab Program `MethodQR_shift.m` to Test Method of QR Iteration with Shift $\sigma = A(n,n)$

```
% ---------------------------------------------------------------------
%  Program MethodQR_shift.m
%  Implements method of  QR iteration with  shift   σ = A(n,n).
% ---------------------------------------------------------------------
clc
%clear all
%close all
eps = 1e-09;
fig = figure;
N=10;
for i =1:6
  if(i==1)
    n=N;
    A=hilb(N);
  elseif (i==2)
    n=20;
    A=hilb(20);
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
  elseif (i==4)
    % Matrix has four real eigenvalues
    n =4;
    A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
  elseif (i==5)
    n =5;
    %
    A=[3,7,8,9,12;5,-7,4,-7,8;1,1,-1,1,-1;4,3,2,1,7;9,3,2,5,4];
  elseif (i==6)
    n=N;
    A= rand(N,N);
  end

  lambda0= inf(n,1);
  count = 1;
  iter =1;
  %choose shift
  %sigma=1.0;
  sigma=A(n,n);
  %sigma=A(1,1);

  % get  exact eigenvalues in sorted order
  exact_lambda = sort(eig(A));
  %%% Method of QR iteration with shift

  for k = 1:100

    A = A - sigma*eye(n);

    [Q,R] = qr(A);
    % end

    A = R*Q + sigma*eye(n);

    %compute shift
    sigma=A(n,n);

    % %%%%%%%%% Find eigenvalues from Real Schur block
    j =2; count =1;
    eigs = zeros(1,n);
    while (j <=n)
      %real eigenvalues
      if(abs(A(j,j-1)) < 1e-7)
        eigs(j-1) =A(j -1,j -1);
        count= j -1;
      else
        % Complex  eigenvalues
        eigs(j-1: j)= eig(A(j -1:j,j -1:j));
        count =j;
        j=j +1;
      end
      j=j +1;
    end
    if(count < length(eigs))
      eigs(n)=A(n,n);
    end
    %********************************

    computed_lambda = sort(eigs)';

    if(norm(abs(computed_lambda - lambda0 ))<eps )
      break ;
    end
    lambda0 = computed_lambda ;
```

```
    iter = iter + 1;
  end
  %****************************************
  str =['Comp. eig.:' num2str(computed_lambda')];
  str=[str, ', Ex. eig.:' num2str(exact_lambda',2)];
  str_xlabel = ['Example ',num2str(i), ...
 '.  Nr.it. in  QR it. with shift:', num2str(iter)];

  subplot (3,2,i)

  plot (exact_lambda,'o b','LineWidth',2,'Markersize',10)
  hold on
  plot (computed_lambda,'+ r','LineWidth',2, 'Markersize',10)

  % xlabel(str, 'fontsize',10)

  xlabel('Real part of eigenvalues');
  ylabel('Imag. part of eigenvalues');

  exact_lambda
  computed_lambda
  legend('Exact eigenvalues','Computed eigenvalues')

  title(str_xlabel,'fontsize',12)

end
```

1.14 Matlab Program `MethodQR_Wshift.m` to Test Method of QR Iteration with Wilkinson's Shift

```
% ----------------------------------------------------
%  Program MethodQR_Wshift.m
%  Method of QR iteration with Wilkinson shift.
% ----------------------------------------------------
clc
%clear all
%close all
eps = 1e-09;
fig = figure;
N=10;
for i = 1:6
  if(i==1)
    n=N;
    A=hilb(N);
  elseif (i==2)
    n=20;
    A=hilb(20);
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
```

```
elseif (i==4)
  % Matrix has four real eigenvalues
  n =4;
  A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
elseif (i==5)
  n =5;
  %
  A=[3,7,8,9,12;5,-7,4,-7,8;1,1,-1,1,-1;4,3,2,1,7;9,3,2,5,4];
elseif (i==6)
  n=N;
  A= rand(N,N);
end

lambda0 = inf(n,1);
count = 1;
iter = 1;

% Wilkinson shift
eig_w = eig([A(n-1,n-1),A(n-1,n); A(n,n-1),A(n,n)]);

abs_eig1 = abs(A(n,n) - eig_w(1));
abs_eig2 = abs(A(n,n) - eig_w(2));

if abs_eig1 < abs_eig2
  sigma =  eig_w(1);
else
  sigma =  eig_w(2);
end

% get exact eigenvalues in sorted order
exact_lambda = sort(eig(A));
%**************************************************
%% Method of QR iteration with shift

for k = 1:100

  A = A - sigma*eye(n);

  [Q,R] = qr(A);
  % end

  A = R*Q + sigma*eye(n);

  % compute Wilkinson shift

  eig_w = eig([A(n-1,n-1),A(n-1,n); A(n,n-1),A(n,n)]);

  abs_eig1 = abs(A(n,n) - eig_w(1));
  abs_eig2 = abs(A(n,n) - eig_w(2));

  if abs_eig1 < abs_eig2
    sigma =  eig_w(1);
  else
    sigma =  eig_w(2);
    end

    % %%%%%%%%% Find eigenvalues from Real Schur block
    j =2; count =1;
    eigs = zeros(1,n);
    while (j <=n)
      %real eigenvalues
      if(abs(A(j,j-1)) < 1e-7)
        eigs(j-1) =A(j -1,j -1);
        count= j -1;
      else
        % Complex  eigenvalues
        eigs(j-1:j) = eig(A(j -1:j,j -1:j));
        count = j;
        j = j + 1;
      end
      j = j + 1;
    end
    if(count < length(eigs))
      eigs(n)=A(n,n);
    end
    %*******************************

    computed_lambda = sort(eigs)';

    if(norm(abs(computed_lambda - lambda0 ))<eps)
      break ;
    end
    lambda0 = computed_lambda;
    iter = iter + 1;
  end

  %***************************************
  str = ['Comp. eig.:' num2str(computed_lambda')];
  str = [str, ', Ex. eig.:' num2str(exact_lambda',2)];
  str_xlabel = ['Example ',num2str(i),  ...
 '.  Nr.it. in QR it. with W.shift:', num2str(iter)];

  subplot (3,2,i)

  plot (exact_lambda,'o b','LineWidth',2,'Markersize',10)
  hold on
  plot (computed_lambda,'+ r','LineWidth',2, 'Markersize',10)

  % xlabel(str, 'fontsize',10)

  xlabel('Real part of eigenvalues');
  ylabel('Imag. part of eigenvalues');

  exact_lambda
  computed_lambda
  legend('Exact eigenvalues', 'Computed eigenvalues')

  title(str_xlabel,'fontsize',12)
```

```
end
```

1.15 Matlab Program `HessenbergQR.m`: First we Use Hessenberg Reduction and then the Method of QR Iteration

```
% ----------------------------------------------------------------
%  Program HessenbergQR.m
%  We will first reduce the matrix A to the upper
%  Hessenberg matrix and then compute it's QR factorization.
% ----------------------------------------------------------------
clc
clear all
close all
eps = 1e-07;
fig = figure;
N=10;
for i =1:6
  if(i==1)
    n=N;
    A=hilb(N);
  elseif (i==2)
    n=20;
    A=hilb(20);
  elseif (i ==3)
    % Largest eigenvalue is complex
    n =3;
    A =[0 -5 2; 6 0 -12; 1 3 0];
  elseif (i==4)
    % Matrix has four real eigenvalues
    n =4;
    A=[3,7,8,9;5,-7,4,-7;1,-1,1,-1;9,3,2,5];
  elseif (i==5)
    n =5;
    %
    A=[3,7,8,9,12;5,-7,4,-7,8;1,1,-1,1,-1;4,3,2,1,7;9,3,2,5,4];
  elseif (i==6)
    n=N;
    A= rand(N,N);

  end

  lambda0= inf(n,1);
  count = 1;
  iter =1;

  % get  exact eigenvalues in sorted order
  exact_lambda = sort(eig(A));
  % First we reduce matrix A to upper Hessenberg

  for k=1:n - 2
    x= A(k+1:n,k);
    u=x;
    u(1) = u(1)+ sign(x(1))*norm(x);
    u=u/norm (u);
    P= eye(n-k) - 2*(u*u') ;
    A(k +1:n,k:n) =P*A(k +1:n,k:n) ;
    A(1:n,k +1:n)=A(1:n,k+1:n)*P;
  end

% ----------------------------------------

  for k = 1:1000
    [Q,R] = qr(A);
    A = R*Q;
  end

  %%%%%%%%%% Find eigenvalues from Real Schur block
  j =2; count =1;
  eigs = zeros(1,n);
  while (j <=n)
    %real eigenvalues
    if(abs(A(j,j-1)) < 1e-3)
      eigs(j-1) =A(j -1,j -1);
      count= j -1;
    else
      % Complex  eigenvalues
      eigs(j-1: j)= eig(A(j -1:j,j -1:j));
      count =j;
      j=j +1;
    end
    j=j +1;
  end
  if(count < length(eigs))
    eigs(n)=A(n,n);
  end
% ----------------------------------------

  computed_lambda = sort(eigs)';

  if(norm(abs(computed_lambda - lambda0 ))<eps )
    break ;
  end
  lambda0 = computed_lambda ;
  iter = iter + 1;

  str =['Comp. eig.:' num2str(computed_lambda')];
  str=[str, ', Ex. eig.:' num2str(exact_lambda',2)];
  str_xlabel = ['Example ',num2str(i),  ...
'. Nr. of it. in method of QR it.:', num2str(iter)];

  subplot (3,2,i)
  plot (exact_lambda,'o b','LineWidth',2,'Markersize',10)
  hold on
  plot (computed_lambda,'+ r','LineWidth',2, 'Markersize',10)

  % xlabel(str, 'fontsize',10)

  xlabel('Real part of eigenvalues');
  ylabel('Imag. part of eigenvalues');

  exact_lambda
  computed_lambda
  legend('Exact eigenvalues','Computed eigenvalues')

  title(str_xlabel,'fontsize',12)

end
```

1.16 Matlab Program RayleighQuotient.m for computation the Rayleigh Quotient

```
% ------------------------------------------------------------------
% Program testRayleigh.m
% Program which generates predefined random tridiagonal matrices
% A of dim(A) = n
% and then calls the function   RayleighQuotient.m
% -------------------------------------------------------------------

n=10;
A=zeros(n);

for i=2:n
  tal = rand*30;
  A(i,i)=rand*20;
  A(i,i-1)=tal;
  A(i-1,i)=tal;
end
A(1,1)=22*rand;

%run algorithm of Rayleigh Quotient Iteration

[rq]=RayleighQuotient(A);

disp('Computed  Rayleigh Quotient is:')
disp(rq)


% ------------------------------------------------------------------
% Program RayleighQuotient.m
% Computes value of  Rayleigh Quotient rq which is in the tolerance
% tol from an eigenvalue of A.
% ------------------------------------------------------------------

function rq = RayleighQuotient(A)

  [n,~]=size(A);
  x0=zeros(n,1);

  % initialize initial vector x0 which  has norm 1
  x0(n)=1;

  tol = 1e-10;
  xi = x0/norm(x0,2);

  i=0;
  % initialize  Rayleigh Quotient for x0
  rq = (xi'*A*xi)/(xi'*xi);

  while norm((A*xi-rq*xi),2) > tol
    yi = (A-rq*eye(size(A)))\xi;
    xi=yi/norm(yi,2);
    rq = (xi'*A*xi)/(xi'*xi)
    i=i+1;
  end

end

% ----------------------------------------
```

1.17 Matlab Program `DivideandConq.m`

```
% ------------------------------------------------------------------
% Program testDC.m
% Program which generates predefined random tridiagonal matrices
% A of dim(A) = n
% and then calls the function   DivideandConq.m
% -------------------------------------------------------------------

%Program which generates some random  symmetric tridiagonal matrices

n=5;
A=zeros(n);

for i=2:n
  tal = rand*30;
  A(i,i)=rand*20;
  A(i,i-1)=tal;
  A(i-1,i)=tal;
end
A(1,1)=22*rand;

%run Divide-and-Conquer  algorithm
[Q,L]=DivideandConq(A)
```

```
% ------------------------------------------------------------------
% Program DivideandConq.m
% Computes algorithm of Divide-and-Conquer:
% eigenvalues will be roots of the secular equation  f(λ) = 0
% and will lie on the diagonal of the output matrix L.
% The roots λ of the secular equation f(λ) = 0 are
%  computed using Newton's method.
% In the output matrix Q will be corresponding eigenvectors.
% ------------------------------------------------------------------

function [Q,L] = DivideandConq(T)
  % Compute size of input matrix T:
  [m,n] = size(T);

  % here we will divide the matrix
  m2 = floor(m/2);

  %if m=0 we shall return
  if m2 == 0 %1 by 1
    Q = 1; L = T;
    return;
    %else we perform recursive computations
  else
    [T,T1,T2,bm,v] = formT(T,m2);

    %recursive computations
    [Q1,L1] = DivideandConq(T1);
    [Q2,L2] = DivideandConq(T2);

    %pick out the last and first columns of the transposes:
    Q1T = Q1';
    Q2T = Q2';
    u = [Q1T(:,end); Q2T(:,1)];

    %Creating the D-matrix:
    D = zeros(n);
    D(1:m2,1:m2) = L1;
    D((m2+1):end,(m2+1):end) = L2;

    % The Q matrix (with Q1 and Q2 on the "diagonals")
    Q = zeros(n);
    Q(1:m2,1:m2) = Q1;
    Q((m2+1):end,(m2+1):end) = Q2;

    %Creating the matrix B, which determinant is the secular equation:
    % det B = f(\lambda)=0
    B = D+bm*u*u';

    % Compute eigenvalues as roots of the secular equation
    %  f(\lambda)=0  using Newton's method
    eigs = NewtonMethod(D,bm,u);
    Q3 = zeros(m,n);

    % compute eigenvectors for corresponding eigenvalues
    for i = 1:length(eigs)
      Q3(:,i) = (D-eigs(i)*eye(m))\u;
      Q3(:,i) = Q3(:,i)/norm(Q3(:,i));
    end

    %Compute  eigenvectors of the original input matrix T
    Q = Q*Q3;

    % Present eigenvalues  of the original matrix input T
    %(they will be on diagonal)
    L = zeros(m,n);
    L(1:(m+1):end) = eigs;

    return;
  end

end

% Compute T1, T2  constant bm  and the vector v
%from the input matrix A.

function [T,T1,T2,bm,v] = formT(A,m)

  T1 = A(1:m,1:m);
  T2 = A((m+1):end,(m+1):end);
  bm = A(m,m+1);

  T1(end) = T1(end)-bm;
  T2(1) = T2(1)-bm;

  v = zeros(size(A,1),1);
  v(m:m+1) = 1;

  T = zeros(size(A));
  T(1:m,1:m) = T1;
  T((m+1):end,(m+1):end) = T2;

end

% compute eigenvalues in the secular equation
% using the Newton's method

function eigs = NewtonMethod(D,p,u)
  [m,n] = size(D);

  %The initial guess in  the Newton's method
  % will be the numbers d_i
  startingPoints = sort(diag(D));

  %if p > 0 we have an eigenvalue on the right, else on the left
  if p >= 0
    startingPoints = [startingPoints; startingPoints(end)+10000];
  elseif p < 0
    startingPoints = [startingPoints(1)-10000; startingPoints];
```

```
end

eigs = zeros(m,1);

% tolerance in Newton's method
convCriteria = 1e-05;

% step in the approximation of the derrivative
% in Newton's method
dx = 0.00001;

%plot the secular equation
X = linspace(-3,3,1000);
for t = 1:1000
  y(t) =SecularEqEval(D,p,u,X(t),m,n);
end
plot(X,y, 'LineWidth',2)
axis([-3 3 -5 5])
legend('graph of the secular equation f(λ)=0')

%Start  Newton's method
for i = 1:m
  %the starting value of lambda
  currentVal = (startingPoints(i)+startingPoints(i+1) )/ 2;

  % this value is used inthe stoppimg criterion below
  currentVal2 = inf;
  %  computed secular equation for \lambda=currentVal
  fCurr = SecularEqEval(D,p,u,currentVal,m,n);

  rands = 0;
  k =0;
  j = 0;

  if  ~((startingPoints(i+1)-startingPoints(i)) < 0.0001)
    while ~(abs(fCurr) < convCriteria)

      %compute value of the function  dfApprox with small step dx to
      %approximate derivative
      fval2 = SecularEqEval(D,p,u,currentVal+dx,m,n);
      fval1 = SecularEqEval(D,p,u,currentVal,m,n);
      dfApprox = (fval2-fval1)/dx;

      % compute new value of  currentVal in Newton's method,
      % or perform one iteration in Newton's method
      currentVal = currentVal - fCurr/dfApprox;

      % check: if we are  outside of the current range, reinput inside:
      if currentVal <= startingPoints(i)
        currentVal= startingPoints(i)+0.0001;
        k=k+1;
      elseif currentVal >= startingPoints(i+1);
        currentVal= startingPoints(i+1)-0.0001;
        k=k+1;
    elseif dfApprox == Inf || dfApprox == -Inf
     currentVal= startingPoints(i) + ...
     rand*(startingPoints(i+1)-startingPoints(i));
     rands = rands+1;
    end

    j=j+1;

    fCurr = SecularEqEval(D,p,u,currentVal,m,n);

    if k > 10 || j > 50;
     tempVec = [startingPoints(i),startingPoints(i+1)];
     [val,ind] = min(abs([startingPoints(i),startingPoints(i+1)]-currentVal));
     if ind == 1
      currentVal = tempVec(ind)+0.00001;
     else
      currentVal = tempVec(ind)-0.00001;
     end
     break;
    elseif currentVal2 == currentVal || rands > 5 || isnan(currentVal) || isnan(fCurr)
     currentVal = currentVal2;
     break;
    end
    %save last value:
    currentVal2 = currentVal;
   end
  end

  %assigning eigenvalue in the right order
  eigs(i) = currentVal;

 end

end

% evaluate the  secular equation in Newton's method for the computed
% eigenvalue x
function fVal = SecularEqEval(D,p,u,x,m,n)

 fVal = 1+p*u'*inv((D-x*eye(m,n)))*u;

end
```

1.18 Matlab Program `Bisection.m`

```
% ----------------------------------------------------------
% Program Bisection.m
% This program computes  all eigenvalues of the matrix
% A on the input interval [a,b).
% ----------------------------------------------------------

% define size n of the  n-by-n matrix A
```

```
n=5;

% Generate the symmetric tridiagonal  matrix A
A=randomTridiag(n);

% Set bounds for the interval [a,b) in the algorithm and the tolerance
a=-100;b=100;
tol=0.000001;

%Define functions for the worklist

DeleteRowInWorklist=@(Worklist,linenr) ChangeRowInWorklist(Worklist,linenr,'delete');
InsertRowInWorklist=@(Worklist,LineToAdd)...
ChangeRowInWorklist(Worklist,LineToAdd,'add');

% Set the info for the first worklist
na=Negcount(A,a);
nb=Negcount(A,b);
Worklist=[];

%If no eigenvalues are found on the interval [a,b) then save an empty worklist
if na~=nb
  Worklist=InsertRowInWorklist(Worklist,[a,na,b,nb]);
end

while numel(Worklist)~=0
  [Worklist, LineToWorkWith ]= DeleteRowInWorklist(Worklist,1);

  low=LineToWorkWith(1);
  n_low=LineToWorkWith(2);
  up=LineToWorkWith(3);
  n_up=LineToWorkWith(4);

  % if the upper and lower bounds are close enough we  print out this interval
  if (up-low)< tol
    NrOfEigVal = n_up-n_low;
    fprintf('We have computed %3.0f eigenvalues in the interval [%4.4f,%4.4f) \n', ...
    NrOfEigVal,low,up);
  else
    % Perform the bisection step
    mid= (low+up)/2;
    n_mid= Negcount(A,mid);
    if n_mid > n_low
      Worklist = InsertRowInWorklist(Worklist,[low,n_low,mid,n_mid]);
    end
    if n_up>n_mid
      Worklist = InsertRowInWorklist(Worklist,[mid,n_mid,up,n_up]);
    end
  end
end


% ----------------------------------------
% Add or remove rows to the  WorkList
% If action = 'add' then add a line to the Worklist, return Worklist and
%               new line
%   If action = 'delete' then delete the given line from  the Worklist, return
%               Worklist and deleted line
% ----------------------------------------

function [ Worklist , LineInQuestion] = ChangeRowInWorklist(Worklist,LINE,action)

  if strcmp(action,'delete')
    if (length(Worklist(:,1)) == 1)
      LineInQuestion=Worklist;
      Worklist=[];
    elseif (LINE==length(Worklist(:,1)))
      LineInQuestion = Worklist(LINE,:);
      Worklist=Worklist(1:(end-1),:);
    elseif (LINE==1)
      LineInQuestion = Worklist(LINE,:);
      Worklist=Worklist(2:end,:);
    else
      LineInQuestion = Worklist(LINE,:);
      Worklist=[Worklist(1:(LINE-1),:);Worklist((LINE+1):end,:)];
    end

  elseif strcmp(action,'add')
    LineInQuestion = LINE;
    if (length(Worklist) == 0)
      Worklist=LINE;
    else
      Worklist = [Worklist;LINE];
    end

  else
    fprintf('The third argument must be either delete or add!')

  end
end


% ------------------------------------------------
% Program Negcount.m
% This program computes number of eigenvalues of
% a tridiagonal matrix A
%(without pivoting)  which are less then z.
% ------------------------------------------------

function [ neg ] = Negcount( A,z )

  d=zeros(length(A),1);
  d(1)=A(1,1)-z;
  for i = 2:length(A)
    d(i)=(A(i,i)-z)-(A(i,i-1)^2)/d(i-1);
  end

  %compute number of negative eigenvalues of A
  neg=0;
  for i = 1:length(A)
    if d(i)<0
      neg = neg+1;
    end
  end

end


% -----------------------------------------
% generation of the random  tridiagonal symmetric matrix
% -----------------------------------------

function [A] = randomTridiag(n)

  A=zeros(n);

  for i=2:n
    num = rand*30;
    A(i,i)=rand*20;
    A(i,i-1)=num;
    A(i-1,i)=num;
  end
  A(1,1)=22*rand;
end

% -----------------------------------------
```

1.19 Matlab Program `testClassicalJacobi.m`

```
% --------------------------------------------
% Program testClassicalJacobi.m
% Program which generates predefined random
% tridiagonal matrices A
% and the calls the function RunJacobi.m
% --------------------------------------------
n=5;
A=zeros(n);

for i=2:n
  tal = rand*30;
  A(i,i)=rand*20;
  A(i,i-1)=tal;
  A(i-1,i)=tal;
end
A(1,1)=22*rand;

% initialization of matrix
%A=rand(5,5)*10;
```

```
Ainit=A
%Ainit =A*A'

% run classical Jacobi algorithm
A= RunJacobi(Ainit)

%Print out computed by Jacobi algorithm eigenvalues
disp('computed by Jacobi algorithm eigenvalues:');
eig(A)

% Print out eigenvalues of the initial matrix A using eig(Ainit)
disp('eigenvalues of the initial matrix Ainit using eig(Ainit):');
eig(Ainit)


% -----------------------------------------
% Run Classical Jacobi rotation algorithm.
% until the matrix A is sufficiently diagonal or off(A) < tol
% -----------------------------------------

function [A] = RunJacobi(A)

  tol=0.005;

  iter=1;

  %compute initial off's
  [sum,v]=off(A);

  while sum >tol && iter<100000
    % search for maximal values of off's
    j=v(2,max(v(1,:)) == v(1,:)); %get index j
    k=v(3,max(v(1,:)) == v(1,:)); %get index k

    %perform Jacobi rotation for indices (j,k)
    A=jacobiRot(A,j,k);
    [sum,v]=off(A);
    iter=iter+1;
  end

end

% Run one Jacobi rotation

function [A] = jacobiRot( A,j,k )
  tol=0.0001;

  if abs(A(j,k))>tol
    tau=(A(j,j)-A(k,k))/(2*A(j,k));
    t=sign(tau)/(abs(tau)+sqrt(1+tau^2));
    c=1/(sqrt(1+t^2));
    s=c*t;

    R=eye(length(A));
    R(j,j)=c;
    R(k,k)=c;
    R(j,k)=-s;
    R(k,j)=s;

    A=R'*A*R;
  end

end

% Compute  off's:  the square root of the sum of squares
% of the upper off-diagonal elements.
% v is a matrix that holds the information needed.

function [sum,v] = off(A)

  sum=0;
  %create array v for off's:
  % in the first row  will be sum of square root of the squares of computed off's
  % in the second row: the index  j
  % in the third row: the index k

  v=[0;0;0];
  for i=1:(length(A)-1)
    for j=(i+1):length(A)
      sum=sum+A(i,j)*A(i,j);
      v=[v,[sqrt(A(i,j)*A(i,j));i;j]];
    end
  end
  sum=sqrt(sum);
  v=v(:,2:end);

end
```

1.20 Matlab Program testSVDJacobi.m

```
% -----------------------------------------
% Program testSVDJacobi.m
% Program which generates predefined random
% tridiagonal matrices A
% and the calls the function RunSVDJacobi.m
% -----------------------------------------

n=5;
A=zeros(n);

for i=2:n
  tal = rand*30;
  A(i,i)=rand*20;
  A(i,i-1)=tal;
  A(i-1,i)=tal;
end
```

```
A(1,1)=22*rand;

Ainit=A

disp('computed by one-sided Jacobi algorithm  SVD decomposition:');
[U,S,V]= RunSVDJacobi(Ainit)

disp('computed  SVD decomposition using svd command (for comparison):');
[u,sigma,v]=svd(Ainit)


% -------------------------------------------------
% Program RunSVDJAcobi.m
% Computes the SVD decomposition of the matrix G
%  using the one-sided Jacobi rotation.
% -------------------------------------------------

function [U,S,V] = RunSVDJacobi(G)

  % input tolerance
  tol=0.005;

  J=eye(length(G));
  iter=1;

  [sum,v]=off(G'*G);

  while sum>tol && iter<1000
    for j=1:(length(G)-1)
      for k=j+1:length(G)
        [G,J]=oneSidedJacobiRot(G,J,j,k);
      end
    end

    [sum,v]=off(G'*G);
    iter=iter+1;
  end

  % elements in the matrix sigma will be the two-norm
  % of i-column of the matrix G

  for i=1:length(G)
    sigma(i)=norm(G(:,i));
  end

  U=[];

  for i=1:length(G)
    U=[U,G(:,i)/sigma(i)];
  end

  V=J;

  S=diag(sigma);
```

```
end

% compute one-sided Jacobi rotation for G

function [G,J] = oneSidedJacobiRot(G,J,j,k )

  tol=0.0001;
  A=(G'*G);
  ajj=A(j,j);
  ajk=A(j,k);
  akk=A(k,k);

  if abs(ajk)>tol
    tau=(ajj-akk)/(2*ajk);
    t=sign(tau)/(abs(tau)+sqrt(1+tau^2));
    c=1/(sqrt(1+t^2));
    s=c*t;

    R=eye(length(G));
    R(j,j)=c;
    R(k,k)=c;
    R(j,k)=-s;
    R(k,j)=s;

    G=G*R;

    % if eigenvectors are desired
    J=J*R;
  end
end

% Compute  off's:  the square root of the sum of squares
% of the upper off-diagonal elements.
% v is a matrix that holds the information needed.

function [sum,v] = off(A)

  sum=0;
  %create array v for off's:
  % in the first row  will be sum of square root of the squares of computed off's
  % in the second row: the index  j
  % in the third row: the index k

  v=[0;0;0];
  for i=1:(length(A)-1)
    for j=(i+1):length(A)
      sum=sum+A(i,j)*A(i,j);
      v=[v,[sqrt(A(i,j)*A(i,j));i;j]];
    end
  end
  sum=sqrt(sum);
  v=v(:,2:end);
```

```
end
```

1.21 Matlab Program `Poisson2D_Jacobi.m`. The function `DiscretePoisson2D.m` is given in section 1.1.

```
% ---------------------------------------------------------
% Program Poisson2D_Jacobi.m
% This is the main program to solve  Poisson's equation
%   -a△u = f with Dirichlet b.c.  u = 0
%   in 2D  on a square using iterative Jacobi method.
% ---------------------------------------------------------

close all
clc
clear
clf
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; %  we can choose f=1, 50, 100
x_0=0.5;
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% -----------------------------------------
% Computing all matrices and vectors
% -----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

% If f is constant.
% f = f_amp*ones(n^2,1);
% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)= f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

%  Compute vector of right hand side
%  b = D^(-1)*f   computed as b(i,j)=f(i,j)/a(i,j)

b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)= f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)
  end
end

% ----------------------------------------
% ---  Solution of 1/h^2 S*u = b using Jacobi's method, version I
% ----------------------------------------

err = 1;  k=0;  tol=10^(-9);

w_old = ones(length(S),1);
L=tril(S,-1);
U=L';
Dinv=diag(diag(S).^(-1));
R=Dinv*(-L-U);
c=Dinv*h^2*b;

while(err>tol)
  w_new = R*w_old +c;
  k=k+1;

  % stopping criterion: choose one of two
  err = norm(w_new-w_old);
  %  err = norm(S*w_new - h^2*b);
  w_old = w_new;
end

disp('-- Number of iterations in the version I of Jacobi method ----------')
k

% ----------------------------------------
% Plots and figures for version I
% ----------------------------------------

% sort the data in u into the mesh-grid, the boundary nodes are zero.
V_new = zeros(n+2,n+2);
for i=1:n
```

```
  for j=1:n
    V_new(i+1,j+1) = w_new(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

figure(1)

subplot (2,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) Jacobi version I ',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (2,2,2)

surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2) Jacobi version I',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

% Plotting a(x,y)
Z_a= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,3)
surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
zlabel('a(x_1,x_2)')
title( ['coefficient a(x_1,x_2) with A = ',num2str(a_amp)])

%  plott the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,4)
surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])

% ----------------------------------------
% ---  Jacobi's method, version II --------------------------
% ----------------------------------------

k=0; err = 1;
V_old = zeros(n,n);
V_new = zeros(n,n);
F=vec2mat(b,n)';
X=diag(ones(1,n-1),-1);
X=X+X';

while(err>tol)
  V_new = (X*V_old + V_old*X' + h^2*F)/4;
  k=k+1;
  err = norm(V_new-V_old);
  V_old = V_new;
end

%apply boundary conditions
V_new = [zeros(1,n+2); zeros(n,1) V_new zeros(n,1);zeros(1,n+2)]

disp('-- Number of iterations in the version II of Jacobi method ----------')
k

figure(2)
% ----------------------------------------
% Plots and figures for version II
% ----------------------------------------

%% plotting
x1=0:h:1;
y1=0:h:1;

subplot (1,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) Jacobi version II ',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (1,2,2)
```

```
surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2)  Jacobi version II',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

% ----------------------------------------
% ---  Jacobi's method, version III --------------------------
% ----------------------------------------

err = 1;  k=0;  tol=10^(-9);
% Initial guess
uold = zeros(n+2, n+2);
unew= uold;

% counter for iterations
k = 0;

while(err > tol)
  for i = 2:n+1
    for j = 2:n+1
      unew(i, j) = (uold(i-1, j) + uold(i+1, j) + uold(i, j-1) + uold(i, j+1)
+ h^2*b(n*(i-2)+j-1))/4.0;
    end
  end
  k = k+1;
  err = norm(unew-uold);
  uold = unew;
end

u = reshape(unew(2:end-1, 2:end-1)', n*n, 1);

disp('-- Number of iterations in the version III of Jacobi method ----------')

k

figure(3)
% ----------------------------------------
% Plots and figures for version III
% ----------------------------------------

% sort the data in u into the mesh-grid, the boundary nodes are zero.
V_new = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    V_new(i+1,j+1) = u(j+n*(i-1));
  end
end

%% plotting
```

```
x1=0:h:1;
y1=0:h:1;

subplot (1,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) Jacobi version III ',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (1,2,2)

surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2)  Jacobi version III',...
',  N = ',num2str(n),', iter. = ',num2str(k)])
```

1.22 Matlab Program **Poisson2D_Gauss_Seidel.m**. The function **DiscretePoisson2D.m** is given in section 1.1.

```
% ------------------------------------------------------------
% Program Poisson2D_Gauss_Seidel.m
% This is the main program to solve  Poisson's equation
%  −a△u = f with Dirichlet b.c. u = 0
%  in 2D  on a square using iterative   Gauss-Seidel method.
% ------------------------------------------------------------

close all
clc
clear
clf
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; %  we can choose f=1, 50, 100
x_0=0.5;
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% ----------------------------------------
% Computing all matrices and vectors
% -----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

% If f is constant.
% f = f_amp*ones(n^2,1);

% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)=f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

%  Compute vector of right hand side
%  b = D^(-1)*f   computed as b(i,j)=f(i,j)/a(i,j)

b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)=  h^2*(f(n*(i-1)+j))/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)

  end
end

% -----------------------------------------
% Solution of  S*u = b  using iterative Gauss-Seidel method
% -----------------------------------------

residual = 1;  k=0;  tol=10^(-9);

u = zeros(n^2,1);
u_old = u;

% use  Gauss-Seidel  algorithm without red-black ordering:
```

```
% values u(1:(j-1)) are already updated, and u_old((j+1):n^2)
% are older, computed on the previous iteration

while (norm(residual)> tol)
  for j = 1:n^2
    u(j) = 1/S(j,j) * (b(j)  ...
    - S(j,1:(j-1))*u(1:(j-1)) - S(j,(j+1):n^2)*u_old((j+1):n^2));
  end
  u_old = u;
  residual = S*u- b;
  k = k+1;
end

disp('-- Number of iterations in Gauss-Seidel method ----------')
k

% ----------------------------------------
% Plots and figures for Gauss-Seidel method
% ----------------------------------------
% sort the data in u into the mesh-grid, the boundary nodes are zero.
Z = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    Z(i+1,j+1) = u(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

subplot (2,2,1)
surf(x1,y1, Z) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) Gauss-Seidel method ',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (2,2,2)

surf(x1,y1, Z) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2)  Gauss-Seidel method',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

% Plotting a(x,y)
Z_a= zeros(n+2);
```

```
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,3)
surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
zlabel('a(x_1,x_2)')
title( ['coefficient a(x_1,x_2) with A = ',num2str(a_amp)])

%  plott the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,4)
surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])
```

1.23 Matlab Program Poisson2D_Gauss_SeidelRedBlack.m. The function DiscretePoisson2D.m is given in section 1.1.

```
% ---------------------------------------------------------------------
% Program Poisson2D_Gauss_SeidelRedBlack.m
% This is the main program for the solution of Poisson's equation
%  -aΔu=f with Dirichlet b.c. u=0 in 2D on a square
%  using iterative   Gauss-Seidel method
%  with Red-Black ordering.
% ---------------------------------------------------------------------

close all
clc
clear
clf
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; %  we can choose f=1, 50, 100
x_0=0.5;
```

```
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% ----------------------------------------
% Computing all matrices and vectors
% ----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

% If f is constant.
% f = f_amp*ones(n^2,1);

% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)=f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

%  Compute vector of right hand side
%  b = D^(-1)*f   computed as b(i,j)=f(i,j)/a(i,j)

b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)=f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)
  end
end

% ----------------------------------------
% Solution of 1/h^2 S u = b using iterative Gauss-Seidel method
% with red-black ordering, version I
% ----------------------------------------

err = 1;  k=0;  tol=10^(-9);
V = zeros(n,n);
V_old = zeros(n,n);
F=vec2mat(b,n)';
X=diag(ones(1,n-1),-1);
X=X+X';

blackindex = invhilb(n) < 0;
redindex = fliplr(blackindex);
B=V;
V(redindex)=0;

R=V;
V(blackindex)=0;

redF = F; redF(blackindex)=0;
blackF = F; blackF(redindex)=0;

while(err>tol)
  R = (X*B + B*X + h^2*redF)/4;
  B = (X*R + R*X + h^2*blackF)/4;
  k=k+1;

  V_new =R+B;
  err = norm(V_new - V_old);
  V_old = V_new;
end

V_new = [zeros(1,n+2); zeros(n,1) V_new zeros(n,1);zeros(1,n+2)]

disp('-- Number of iterations in Gauss-Seidel method  ----------')
k

% ----------------------------------------
% Plots and figures for Gauss-Seidel method
% ----------------------------------------

figure(1)
%% plotting
x1=0:h:1;
y1=0:h:1;

subplot (2,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) in Gauss-Seidel Red-Black ordering',...
',  N = ',num2str(n),', iter. = ',num2str(k)])
```

```
subplot (2,2,2)

surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2) in Gauss-Seidel Red-Black ordering',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

% Plotting a(x,y)
Z_a= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,3)
surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
zlabel('a(x_1,x_2)')
title( ['coefficient a(x_1,x_2) with A = ',num2str(a_amp)])

%  plott the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,4)
surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])

% ----------------------------------------
% Solution of 1/h^2 S u = b using iterative Gauss-Seidel method
% with red-black ordering, version II
% ----------------------------------------

err = 1;  k=0;  tol=10^(-9);
% Initial guess
uold = zeros(n+2, n+2);
unew= uold;

while(err > tol)
```

```
  % Red nodes
  for i = 2:n+1
    for j = 2:n+1
      if(mod(i+j,2) == 0)
        unew(i, j) = (uold(i-1, j) + uold(i+1, j) + uold(i, j-1) + uold(i, j+1)
+ h^2*b(n*(i-2)+j-1))/4.0;
        % for computation of residual
        u(j-1 + n*(i-2)) = unew(i,j);
      end
    end
  end

  % Black nodes
  for i = 2:n+1
    for j = 2:n+1
      if(mod(i+j,2) == 1)
        unew(i,j) = 0.25*(unew(i-1,j) + unew(i+1,j) ...
        + unew(i,j-1) + unew(i,j+1) + h^2*b(n*(i-2)+j-1));
        % for computation of residual
        u(j-1 + n*(i-2)) = unew(i,j);
      end
    end
  end

  k = k+1;

  % different stopping  rules
  err = norm(unew-uold);
  %computation of residual
  %  err = norm(S*u' - h^2*b);
  uold = unew;
end

u = reshape(unew(2:end-1, 2:end-1)', n*n, 1);

disp('-- Number of iterations in the version II of Gauss-Seidel method----------')

k

% ----------------------------------------
% Plots and figures for version II
% ----------------------------------------

figure(2)
% sort the data in u into the mesh-grid, the boundary nodes are zero.
V_new = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    V_new(i+1,j+1) = u(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
```

```
y1=0:h:1;

subplot (1,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) in Gauss-Seidel Red-Black ordering, version II',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (1,2,2)

surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2)   in Gauss-Seidel Red-Black ordering, version II',...
',  N = ',num2str(n),', iter. = ',num2str(k)])
```

1.24 Matlab Program Poisson2D_SOR.m. The function DiscretePoisson2D.m is given in section 1.1.

```
% ----------------------------------------
% Program Poisson2D_SOR.m
% This is the main program for the solution of
% Poisson's equation
%  −aΔu = f with Dirichlet b.c. u = 0
% in 2D using iterative  SOR method.
% ----------------------------------------

close all
clc
clear
clf
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; %  we can choose f=1, 50, 100
x_0=0.5;
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% ----------------------------------------
% Computing all matrices and vectors
% ----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

% If f is constant.
% f = f_amp*ones(n^2,1);

% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)=f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

%  Compute vector of right hand side
%  b = D^(-1)*f   computed as b(i,j)=f(i,j)/a(i,j)

b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)=f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)
  end
end

% ----------------------------------------
% Solution of 1/h^2 S u = b using  SOR method
% with red-black ordering, version I
% ----------------------------------------

err = 1;  k=0; sch = 0;  tol=10^(-9);
V = zeros(n,n);
V_old = zeros(n,n);
F=vec2mat(b,n)';
X=diag(ones(1,n-1),-1);
X=X+X';
```

```
%arrange red-black indexing

blackindex = invhilb(n) < 0;
redindex = fliplr(blackindex);
B=V;
V(redindex)=0;

R=V;
V(blackindex)=0;

redF = F; redF(blackindex)=0;
blackF = F; blackF(redindex)=0;

% extract matrices L and U for matrix RSOR

L=tril(S,-1);
U=L';
Dinv=diag(diag(S).^(-1));
L  = Dinv*(-L);
U =  Dinv*(-U);
D=diag(ones(1,n*n));

omegas = 1.05:0.05:1.95;
for omega = omegas
  k=0;
  err =1;
  B=V;
  V(redindex)=0;

  R=V;
  V(blackindex)=0;

  % counter for omega
  sch = sch+1;

  while(err>tol)
    R = (1 - omega)*R + omega*(X*B + B*X + h^2*redF)/4;
    B = (1- omega)*B + omega*(X*R + R*X + h^2*blackF)/4;
    k=k+1;

    V_new =R+B;
    err = norm(V_new - V_old);
    V_old = V_new;
  end

  % the matrix RSOR in the method SOR: x_m+1 = RSOR*x_m + c_SOR
  RSOR = inv(D - omega*L)*((1-omega)*D + omega*U);

  lambda = max(abs(eig(RSOR)));

  mu =  (lambda + omega -1)/(sqrt(lambda)*omega);

  disp('-- Relaxation parameter in  SOR method  ----------')
  omega
  disp('-- Computed optimal relaxation parameter  ----------')

  omega_opt = 2/(1 +   sqrt(1 - mu^2))

  if (omega <= 2.0 && omega  >=omega_opt )
    disp('-- omega_opt < omega < 2.0 ----------')
    radius = omega -1
  elseif(omega <= omega_opt  && omega > 0)
    disp('-- omega < omega_opt ----------')
    omega_tail = -omega +0.5*omega^2*mu^2 ...
    + omega*mu*sqrt(1 - omega + 0.25*omega^2*mu^2)
    radius = 1 + omega_tail
  end

  disp('-- Number of iterations in  SOR method ----------')
  k

  iterations(sch) = k;
  spectral_radius(sch)= radius;
  omega_optimal(sch) = omega_opt;
end

% apply zero boundary conditions
V_new = [zeros(1,n+2); zeros(n,1) V_new zeros(n,1);zeros(1,n+2)];

% ----------------------------------------
% Plots and figures for SOR method, version I
% ----------------------------------------

figure(1)
%% plotting
x1=0:h:1;
y1=0:h:1;

subplot (2,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) in SOR  method ',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (2,2,2)

surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2) in SOR method',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

% Plotting a(x,y)
Z_a= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,3)
surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
zlabel('a(x_1,x_2)')
title( ['coefficient a(x_1,x_2) with A = ',num2str(a_amp)])

%  plot the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end

subplot (2,2,4)
surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])

% plot  convergence of SOR depending on omega
figure(2)

plot(omegas, iterations,'b o-', 'LineWidth',2)
hold on
plot(omega_optimal, iterations,'r o ', 'LineWidth',2)

xlabel('Relaxation parameter \omega')
ylabel('Number of iterations in SOR')
legend('SOR(\omega)','Computed optimal \omega')
title(['Mesh:  ',num2str(n),' by ',num2str(n),' points'])

% plot  convergence of SOR depending on omega
figure(3)
plot(omegas, spectral_radius,'b o-', 'LineWidth',2)

xlabel('Relaxation parameter \omega')
ylabel(' Spectral radius \rho(R_{SOR(\omega)})')
legend('\rho(R_{SOR(\omega)})')
title(['Mesh:  ',num2str(n),' by ',num2str(n),' points'])
```

```
% ----------------------------------------
% Solution of 1/h^2 S u = b using iterative SOR
% with red-black ordering, version II
% ----------------------------------------

disp('--  Works SOR method, version II ----------')

err = 1;  k=0;  tol=10^(-9);

% choose relaxation parameter  0 < omega < 2
% optimal omega can be computed as
omega_opt = 2/(1 + sin(pi/(n+1)))
% Initial guess
uold = zeros(n+2, n+2);
unew= uold;

while(err > tol)
  % Red nodes
  for i = 2:n+1
    for j = 2:n+1
      if(mod(i+j,2) == 0)
        unew(i, j) =   (1-omega)*unew(i,j) + ...
        omega*(uold(i-1, j) + uold(i+1, j) + uold(i, j-1) + uold(i, j+1) ...
        + h^2*b(n*(i-2)+j-1))/4.0;
        % for computation of residual
        u(j-1 + n*(i-2)) = unew(i,j);
      end
    end
  end

  % Black nodes
  for i = 2:n+1
    for j = 2:n+1
      if(mod(i+j,2) == 1)
        unew(i,j) =  (1-omega)*unew(i,j)  + ...
        omega*0.25*(unew(i-1,j) + unew(i+1,j) + unew(i,j-1) + unew(i,j+1) + ...
        h^2*b(n*(i-2)+j-1));
        % for computation of residual
        u(j-1 + n*(i-2)) = unew(i,j);
      end
    end
  end

  k = k+1;

  % different stopping  rules
  err = norm(unew-uold);
  %computation of residual
  %  err = norm(S*u' - h^2*b);
  uold = unew;
end

u = reshape(unew(2:end-1, 2:end-1)', n*n, 1);
```

```
disp('-- Number of iterations in the version II of   SOR ----------')

k

% ----------------------------------------
% Plots and figures for version II
% ----------------------------------------

figure(4)
% sort the data in u into the mesh-grid, the boundary nodes are zero.
V_new = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    V_new(i+1,j+1) = u(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

subplot (1,2,1)
surf(x1,y1,V_new) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['solution u(x_1,x_2) in SOR with Red-Black ordering, version II',...
',  N = ',num2str(n),', iter. = ',num2str(k)])

subplot (1,2,2)

surf(x1,y1,V_new) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')

title( ['solution u(x_1,x_2)   in  SOR with Red-Black ordering, version II',...
',  N = ',num2str(n),', iter. = ',num2str(k)])
```

1.25 Matlab Program Poisson2D_ConjugateGrad.m. The function DiscretePoisson2D.m is given in section 1.1.

```
% ------------------------------------------------
% Program Poisson2D_ConjugateGrad.m
% This is the main program for the solution of
% Poisson's equation  $-a\triangle u = f$
% with Dirichlet b.c. $u = 0$ in 2D on a square
% using  Conjugate Gradient Method.
% -------------------------------------------------

close all
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; % 1, 50, 100 choose const. f value
x_0=0.5;
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% ----------------------------------------
% Computing all matrices and vectors
% ----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

%% calculate load vector f

% If f is constant.
% f = f_amp*ones(n^2,1);

% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)=f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

%  Compute vector of right hand side
%  b = D^(-1)*f given by b(i,j)=f(i,j)/a(i,j)
```

```
b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)=f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)
  end
end
% ----------------------------------------
% ----------- Conjugate gradient method
% ----------------------------------------
% We should solve: 1/h^2 S u = b

k=0;
err = 1; x=0; r0= h^2*b; p= h^2*b; tol=10^(-9);
while(err>tol)
  k=k+1;
  z = S*p;
  nu = (r0'*r0)/(p'*z);
  x = x + nu*p;
  r1 = r0 - nu*z;
  mu = (r1'*r1)/(r0'*r0);
  p = r1 + mu*p;
  r0=r1;
  err = norm(r0);
end

disp('-- Number of iterations in Conjugate gradient method ----------')
k

% ----------------------------------------
% Plots and figures.
% ----------------------------------------

% sort the data in u into the mesh-grid, the boundary nodes are zero.
Z = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    Z(i+1,j+1) = x(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

subplot(2,2,1)

surf(x1,y1,Z) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2) in Conjugate gradient method ',...
',  N = ',num2str(n)])

subplot(2,2,2)
surf(x1,y1,Z) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2)  in Conjugate gradient method ', ...
', N = ',num2str(n)])

% Plotting a(x,y)
Z_a= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end

subplot(2,2,3)

surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
zlabel('a(x_1,x_2)')
title( ['a(x_1,x_2) with A = ',num2str(a_amp)])

%  plott the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end

subplot(2,2,4)

surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])
```

1.26 Matlab Program `Poisson2D_PrecConjugateGrad.m`. The function `DiscretePoisson2D.m` is given in section 1.1

```
% ------------------------------------------------------------
% Program Poisson2D_PrecConjugateGrad.m
% This is the main program for the solution of
% Poisson's equation −a△u = f with
% Dirichlet b.c. u = 0 in 2D on a square
% using  Preconditioned Conjugate Gradient Method.
% ----------------------------------------------------------

close all
%Define input parameters
n=20; % number of inner nodes in one direction.
a_amp = 12; %  amplitude for the function a(x_1,x_2)
f_amp = 1; % we can set f = 1, 50, 100
x_0=0.5;
y_0=0.5;
c_x=1;
c_y=1;

h = 1/(n+1); % define step length

% ----------------------------------------
% Computing all matrices and vectors
% ----------------------------------------
% Generate a n*n by n*n stiffness matrix
S = DiscretePoisson2D(n);

%% generate coefficient matrix of a((x_1)_i,(x_2)_j) = a(i*h,j*h)
C = zeros(n,n);
for i=1:n
  for j=1:n
    C(i,j) = 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
% create diagonal matrix from C
D = zeros(n^2,n^2);
for i=1:n
  for j=1:n
    D(j+n*(i-1),j+n*(i-1)) = C(i,j);
  end
end

%% calculate load vector f

% If f is constant.
% f = f_amp*ones(n^2,1);

% If f is Gaussian function.
f=zeros(n^2,1);
for i=1:n
  for j=1:n
    f(n*(i-1)+j)=f_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
```

```
% 1. Compute vector of right hand side
%  b = D^(-1)*f given by b(i,j)=f(i,j)/a(i,j)

b=zeros(n^2,1);
for i=1:n
  for j=1:n
    b(n*(i-1)+j)=f(n*(i-1)+j)/C(i,j); % Use coefficient matrix C or
    % diagonal matrix D to get a(i,j)
  end
end

% ----------------------------------------
% --- Preconditioned conjugate gradient method (PCGM):
%  choose  different preconditioners:
% Cholesky factorization, Jacobi  preconditioner, block Jacobi preconditioner
% ----------------------------------------
% We now have system to solve: 1/h^2 S u = b

%initialize preconditioner
Ssparse = sparse(S);

% Preconditioner: preconditioner matrix  here is incomplete
% Cholesky factorization of S

cond = ichol(Ssparse); cond=cond*cond'; cond=full(inv(cond));

% Preconditioner: preconditioner matrix  here is
% Jacobi preconditioner.
% Results are the same as in usual conjugate gradient update

%M = diag(diag(S));
%cond = diag(1.0./diag(M));

% Preconditioner: preconditioner matrix  here is
%Block Jacobi Preconditioner
%blockSize = 2; % size of blocks
%cond = zeros(n^2);
%Iinds = ceil( (1:(blockSize*n^2))/blockSize);
%Jinds = blockSize*ceil( (1:(blockSize*n^2))/blockSize^2)-(blockSize-1) ...
%       + repmat%(0:blockSize-1,1,n^2);
%vecInds = sub2ind(size(S),Iinds, Jinds);
%cond(vecInds)  = S(vecInds);

%initialize parameters in the method
err = 1; x=0; r0= h^2*b; p=cond*h^2*b; y0=cond*r0; tol=10^(-9);
k=0;

while(err>tol)
  z = S*p;
  nu = (y0'*r0)/(p'*z);
  x = x + nu*p;
  r1 = r0 - nu*z;
  y1 = cond*r1;
  mu = (y1'*r1)/(y0'*r0);
  p = y1 + mu*p;
  r0=r1;
  y0=y1;
  err = norm(r0);
  k=k+1;
end

disp('-- Number of iterations in  Preconditioned conjugate gradient method (PCGM) ')
k

% ----------------------------------------
% Plots and figures.
% ----------------------------------------

% sort the data in u into the mesh-grid, the boundary nodes are zero.
Z = zeros(n+2,n+2);
for i=1:n
  for j=1:n
    Z(i+1,j+1) = x(j+n*(i-1));
  end
end

%% plotting
x1=0:h:1;
y1=0:h:1;

subplot(2,2,1)
surf(x1,y1,Z) % same plot as above, (x1, y1 are vectors)
view(2)
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2) in PCGM',...
',  N = ',num2str(n)])

subplot(2,2,2)
surf(x1,y1,Z) % same plot as above
colorbar
xlabel('x_1')
ylabel('x_2')
zlabel('u(x_1,x_2)')
title( ['u(x_1,x_2) in PCGM ',...
', N = ',num2str(n)])

% Plotting a(x,y)
Z_a= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_a(i,j)= 1 + a_amp*exp(-((i*h-x_0)^2/(2*c_x^2)...
    +(j*h-y_0)^2/(2*c_y^2)));
  end
end
```

```
subplot(2,2,3)
surf(x1,y1,Z_a)
xlabel('x_1')
ylabel('x_2')
zlabel('a(x_1,x_2)')
title( ['a(x_1,x_2) with A = ',num2str(a_amp)])

%  plott the function f(x,y)
Z_f= zeros(n+2);
for i=1:(n+2)
  for j=1:(n+2)
    Z_f(i,j)=f_amp*exp(-((x1(i)-x_0)^2/(2*c_x^2)...
    +(y1(j)-y_0)^2/(2*c_y^2)));
  end
end

subplot(2,2,4)
surf(x1,y1,Z_f)
xlabel('x_1')
ylabel('x_2')
zlabel('f(x_1,x_2)')
title( ['f(x_1,x_2) with A_f = ',num2str(f_amp)])
```

1.27 PETSc programs for solution of the Poisson's equation in two dimensions.

```
// Program Main.cpp
// This is the Main program
// for the solution of
// Poisson's equation −a△u = f with
// Dirichlet b.c. u = 0 in 2D on a square
// using C++ and PETSc.

static char help[] ="";
#include <iostream>
#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
#include <cmath>
#include <time.h>
#include "Poisson.h"

const PetscInt n = 20;
const PetscScalar h = 1 / (PetscScalar)(n + 1);

const bool VERBOSE = true;

using namespace std;

char METHOD_NAMES[8][70] = {
    "invalid method",
    "Jacobi's method",
    "Gauss-Seidel method",
    "Successive Overrelaxation method (SOR)",
    "Conjugate Gradient method",
    "Conjugate Gradient method (Algorithm 12.13)",
    "Preconditioned Conjugate Gradient method",
    "Preconditioned Conjugate Gradient method (Algorithm 12.14)"};

char *GetMethodName(PetscInt method) {
    if (method < 0 || method > 7)
        return METHOD_NAMES[0];
    else
        return METHOD_NAMES[method];
}

int main(int argc, char **argv) {
  PetscErrorCode ierr;
  ierr = PetscInitialize(&argc, &argv,(char *)0, help);CHKERRQ(ierr);

  PetscInt method =  atoi(argv[1]);
  PetscBool methodSet = PETSC_FALSE;
  Mat S;
  Vec h2b, u;


    ierr = PetscOptionsGetInt(NULL, NULL, "-m", &method, &methodSet);
    if (method < 1 || method > 7) {
        cout << "Invalid number of the selected method: "
         << method << ".\nExiting..." << endl;
        exit(-1);
    }

    // To use SOR with omega != 1, we need to disable inodes
    if (method == METHOD_SOR)
        PetscOptionsSetValue(NULL, "-mat_no_inode", NULL);

    ierr = CreateMatrix(&S, n*n, n*n); CHKERRQ(ierr);
    ierr = CreateVector(&h2b, n*n); CHKERRQ(ierr);
    ierr = CreateVector(&u, n*n); CHKERRQ(ierr);

    // create discrete Laplacian
    ierr = DiscretePoisson2D(n, &S);

    // create right hand side
    ierr = DiscretePoisson2D_coeffs(n, h, &h2b);

    ierr = MatAssemblyBegin(S, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatAssemblyEnd(S, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(h2b); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(h2b); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(u); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(u); CHKERRQ(ierr);
    /*
      Below we solve system  S*u= h2b
     */
    if (VERBOSE)
        PetscPrintf(PETSC_COMM_WORLD, "Using %s\n", GetMethodName(method));

    if (method == METHOD_CG_FULL)
        ConjugateGradient_full(S, h2b, u, VERBOSE);
    else if (method == METHOD_PCG_FULL)
        PreconditionedConjugateGradient_full(S, h2b, u, VERBOSE);
    else
        Solve(S, h2b, u, method, VERBOSE);

    // Print out  solution
 FILE* resultfile = fopen("solution.m", "w");

    if (VERBOSE) {
      PetscInt i, j, matsize, *idx = new PetscInt[n*n];
      PetscScalar *vecu = new PetscScalar[n*n];
      matsize = n*n;

      for (i = 0; i <  matsize; i++)
    idx[i] = i;

      ierr = VecGetValues(u, matsize, idx, vecu);

      for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      PetscPrintf(PETSC_COMM_WORLD, "%.12e ", vecu[n*i + j]);
      fprintf(resultfile, "%.12e ", vecu[n*i + j]);
    }
    PetscPrintf(PETSC_COMM_WORLD, "\n");
    fprintf(resultfile, "\n");
      }

      delete [] vecu;
      delete [] idx;
    }
    fclose(resultfile);
    ierr = PetscFinalize(); CHKERRQ(ierr);
    return 0;
}


#ifndef _CE3_H
#define _CE3_H

#include <petsc.h>

PetscErrorCode CreateMatrix(Mat*, PetscInt, PetscInt);
PetscErrorCode CreateVector(Vec*, PetscInt);

PetscErrorCode DiscretePoisson2D(PetscInt, Mat*);
//PetscErrorCode DiscretePoisson2D_coeffs(PetscInt, PetscScalar, Mat*, Mat*, Vec*);
```

```
PetscErrorCode DiscretePoisson2D_coeffs(PetscInt, PetscScalar, Vec*);

/**********************/
/* METHODS OF SOLUTION */
/**********************/
PetscErrorCode Solve(Mat, Vec, Vec, PetscInt, bool);
PetscErrorCode Jacobi(PC);
PetscErrorCode GaussSeidel(PC);
PetscErrorCode SOR(PC);
PetscErrorCode ConjugateGradient(KSP, PC);
PetscErrorCode ConjugateGradient_full(Mat, Vec, Vec, bool);
PetscErrorCode ConjugateGradient_inner(Mat, Vec, Vec, Mat, bool);
PetscErrorCode PreconditionedConjugateGradient(KSP, PC);
PetscErrorCode PreconditionedConjugateGradient_full(Mat, Vec, Vec, bool);
PetscErrorCode PreconditionedConjugateGradient_inner(Mat, Vec, Vec, Mat, bool);

enum SolverMethod {
    METHOD_INVALID=0,
    METHOD_JACOBI=1,
    METHOD_GAUSS_SEIDEL=2,
    METHOD_SOR=3,
    METHOD_CG=4,
    METHOD_CG_FULL=5,
    METHOD_PCG=6,
    METHOD_PCG_FULL=7
};

#endif/*_CE3_H*/
```

```
//  Program Create.cpp
// This  program   creates PETSc-matrix  and PETSc-vector.


#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>

PetscErrorCode CreateMatrix(Mat *A, PetscInt rows, PetscInt cols) {
    PetscErrorCode ierr;
    ierr = MatCreate(PETSC_COMM_WORLD, A); CHKERRQ(ierr);
    ierr = MatSetSizes(*A, PETSC_DECIDE, PETSC_DECIDE, rows, cols); CHKERRQ(ierr);
    ierr = MatSetFromOptions(*A); CHKERRQ(ierr);
    ierr = MatSetUp(*A); CHKERRQ(ierr);

    return 0;
}

PetscErrorCode CreateVector(Vec *v, PetscInt N) {
    PetscErrorCode ierr;

    ierr = VecCreate(PETSC_COMM_WORLD, v); CHKERRQ(ierr);
    ierr = VecSetSizes(*v, PETSC_DECIDE, N); CHKERRQ(ierr);
    ierr = VecSetFromOptions(*v); CHKERRQ(ierr);
    return 0;
}


// Program DiscretePoisson2D.cpp
// This program  generates  the discretized Laplacian.

#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
#include <cmath>

const PetscScalar A_amplitude = 12.;
const PetscScalar f_amplitude = 1.;
const PetscScalar c_x = 1.;
const PetscScalar c_y = 1.;
const PetscScalar poisson_x0 = 0.5;
const PetscScalar poisson_y0 = 0.5;

/**
 * Compute coefficient matrices.
 *
 * n: Number of rows of matrices
 * h: Timestep length
 * C: n-by-n matrix
 * D: (n*n)-by-(n*n) matrix
 * f:
 **/
PetscErrorCode DiscretePoisson2D_coeffs(PetscInt n, PetscScalar h, Vec *h2b) {
    PetscErrorCode ierr;
    PetscInt i, j, idx2[n*n];
    PetscScalar *vecb = new PetscScalar[n*n];

    // Compute C, D and f
    PetscScalar xarg, yarg, expfunc, a, f;
    for (i = 0; i < n; i++) {
        xarg = (((i+1) * h - poisson_x0)) / c_x;

        for (j = 0; j < n; j++) {
            idx2[i*n + j] = i*n + j;

            yarg = (((j+1) * h - poisson_y0)) / c_y;
            expfunc = exp(-(xarg*xarg/2 + yarg*yarg/2));

            f = f_amplitude * expfunc;
            a = 1 + A_amplitude * expfunc;

            vecb[i*n + j] = h*h * f / a;
        }
    }

    ierr = VecSetValues(*h2b, n*n, idx2, vecb, INSERT_VALUES); CHKERRQ(ierr);
    delete [] vecb;

    return 0;
}
PetscErrorCode DiscretePoisson2D(PetscInt n, Mat *A) {
    PetscErrorCode ierr;
    PetscInt i, k, curr, next, matsize = n*n, idx[matsize];
    PetscScalar *matrep = new PetscScalar[matsize*matsize];

    // Initialize all elements to 0
    for (i = 0; i < matsize; i++) {
        // Create index vectors
        idx[i] = i;

        for (k = 0; k < matsize; k++) {
            matrep[i*matsize + k] = 0;
        }
    }

    // Set main diagonal
    for (i = 0; i < matsize; i++)
        matrep[i*matsize + i] = 4.;

    // 1st and 2nd off-diagonals
    for (k = 0; k < n; k++) {
        for (i = 0; i < n-1; i++) {
            curr = (n*k + i);
            next = (n*k + i + 1);

            matrep[curr*matsize + next] = -1;
            matrep[next*matsize + curr] = -1;
        }
    }

    // 3rd and 4th off-diagonals
    for (i = 0; i < n*(n-1); i++) {
        matrep[i*matsize + (i+n)] = -1;
        matrep[(i+n)*matsize + i] = -1;
    }

    ierr = MatSetValues(*A, matsize, idx, matsize, idx, matrep, INSERT_VALUES);
    CHKERRQ(ierr);

    delete [] matrep;

    return 0;
}


// Program Solver.cpp
// This program chooses  different PETSc    preconditioners
// for different iterative methods.

#include <petsc.h>
```

```
#include <petscmat.h>
#include <petscvec.h>
#include <petscksp.h>
#include <cmath>
#include "Poisson.h"

PetscErrorCode Solve(Mat S, Vec h2b, Vec u, PetscInt method, bool VERBOSE) {
    PetscErrorCode ierr;
    KSP ksp;
    KSPConvergedReason convergedReason;
    PC preconditioner;
    PetscInt number_of_iterations;

    ierr = KSPCreate(PETSC_COMM_WORLD, &ksp); CHKERRQ(ierr);
    ierr = KSPSetOperators(ksp, S, S); CHKERRQ(ierr);
    //ierr = KSPSetOperators(ksp, S, S, DIFFERENT_NONZERO_PATTERN); CHKERRQ(ierr);

    ierr = KSPGetPC(ksp, &preconditioner); CHKERRQ(ierr);
    if (method == METHOD_JACOBI) {
        ierr = Jacobi(preconditioner); CHKERRQ(ierr);
    } else if (method == METHOD_GAUSS_SEIDEL) {
        ierr = GaussSeidel(preconditioner); CHKERRQ(ierr);
    } else if (method == METHOD_SOR) {
        ierr = SOR(preconditioner); CHKERRQ(ierr);
    } else if (method == METHOD_CG) {
        ierr = ConjugateGradient(ksp, preconditioner); CHKERRQ(ierr);
    } else if (method == METHOD_PCG) {
        ierr = PreconditionedConjugateGradient(ksp, preconditioner); CHKERRQ(ierr);
    }

    ierr = KSPSetFromOptions(ksp); CHKERRQ(ierr);

    ierr = KSPSolve(ksp, h2b, u); CHKERRQ(ierr);
    ierr = KSPGetIterationNumber(ksp, &number_of_iterations); CHKERRQ(ierr);

    ierr = KSPGetConvergedReason(ksp, &convergedReason); CHKERRQ(ierr);

    if (convergedReason < 0) {
        PetscPrintf(PETSC_COMM_WORLD,
       "KSP solver failed to converge! Reason: %d\n", convergedReason);
    }

    if (VERBOSE) {
        PetscPrintf(PETSC_COMM_WORLD, "Number of iterations: %d\n", number_of_iterations);
    }

    ierr = KSPDestroy(&ksp); CHKERRQ(ierr);

    return 0;
}



// Program Jacobi.cpp
// This program implements Jacobi's method.


#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
#include <petscksp.h>
#include <cmath>

/**
 * Returns the preconditioner used for Jacobi's method
 */
PetscErrorCode Jacobi(PC preconditioner) {
    PetscErrorCode ierr;
    ierr = PCSetType(preconditioner, PCJACOBI); CHKERRQ(ierr);

    return 0;
}


// Program GaussSeidel.cpp
// This program implements  Gauss-Seidel method.


#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
#include <petscksp.h>
#include <cmath>
#include "Poisson.h"

PetscErrorCode GaussSeidel(PC preconditioner) {
    PetscErrorCode ierr;
    ierr = PCSetType(preconditioner, PCSOR); CHKERRQ(ierr);

    /**
     * To use the Gauss-Seidel method we set
     * omega = 1.
     */
    // By default, omega = 1, so the below line is not necessary
    //ierr = PCSORSetOmega(preconditioner, 1.0); CHKERRQ(ierr);

    return 0;
}


// Program SOR.cpp
// This program computes SOR(ω)
// for ω = 1.5

#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
```

```
#include <petscksp.h>
#include <cmath>
#include "Poisson.h"

const PetscScalar omega = 1.5;

PetscErrorCode SOR(PC preconditioner) {
    PetscErrorCode ierr;

    ierr = PCSetType(preconditioner, PCSOR); CHKERRQ(ierr);
    ierr = PCSORSetOmega(preconditioner, omega); CHKERRQ(ierr);

    return 0;
}


// Program CG.cpp
// This program  implements  two versions
// of the Conjugate gradient method.


#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
#include <petscksp.h>
#include <cmath>
#include "Poisson.h"
/**
 * Conjugate gradient method using  inbuilt PETSc functions.
 */

PetscErrorCode ConjugateGradient(KSP ksp, PC preconditioner) {
    PetscErrorCode ierr;

    ierr = KSPSetType(ksp, KSPCG);
    ierr = PCSetType(preconditioner, PCNONE); CHKERRQ(ierr);

    return 0;
}

/**
 * An implementation of the conjugate gradient method
 * not utilizing the PETSc KSP interface, but
 * implementing the matrix/vector operations directly.
 */
PetscErrorCode ConjugateGradient_full(Mat A, Vec b, Vec x, bool VERBOSE) {
    PetscErrorCode ierr;
    PetscInt k=0, n;
    PetscScalar mu, nu, rTr, pTz, rNorm, tol = 1e-12;
    Vec p, r, z;

    ierr = MatGetSize(A, &n, NULL); CHKERRQ(ierr);

    CreateVector(&p, n);
    CreateVector(&r, n);
    CreateVector(&z, n);

    VecCopy(b, p);
    VecCopy(b, r);

    ierr = VecAssemblyBegin(p); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(p); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(r); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(r); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(z); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(z); CHKERRQ(ierr);

    ierr = VecZeroEntries(x);

    // Pre-compute first (r^T r)
    ierr = VecDot(r, r, &rTr); CHKERRQ(ierr);

    do {
        k++;

        // z = A * p_k
        ierr = MatMult(A, p, z); CHKERRQ(ierr);

        // nu_k = r_{k-1}^T r_{k-1} / p_k^T z
        ierr = VecDot(p, z, &pTz); CHKERRQ(ierr);
        nu = rTr / pTz;

        // x_k = x_{k-1} + nu_k p_k
        ierr = VecAXPY(x, nu, p); CHKERRQ(ierr);

        // r_k = r_{k-1} - nu_k z
        ierr = VecAXPY(r, -nu, z); CHKERRQ(ierr);

        // r_k^T r_k
        mu = 1 / rTr;
        ierr = VecDot(r, r, &rTr); CHKERRQ(ierr);

        // mu_{k+1}
        mu = rTr * mu;

        // p_{k+1} = r_k + mu_{k+1} p_k
        ierr = VecAYPX(p, mu, r);

        // || r_k ||_2
        ierr = VecNorm(r, NORM_2, &rNorm);
    } while (rNorm > tol);

    if (VERBOSE) {
        PetscPrintf(PETSC_COMM_WORLD, "Number of iterations: %d\n", k);
    }

    return 0;
}
```

```
// Program PCG.cpp
// This program implements Preconditioned Conjugate gradient method.

#include <petsc.h>
#include <petscmat.h>
#include <petscvec.h>
#include <petscksp.h>
#include <cmath>
#include "Poisson.h"


PetscErrorCode PreconditionedConjugateGradient(KSP ksp, PC preconditioner) {
    PetscErrorCode ierr;

    ierr = KSPSetType(ksp, KSPCG);

    //ierr = PCSetType(preconditioner, PCJACOBI); CHKERRQ(ierr);
    ierr = PCSetType(preconditioner, PCCHOLESKY); CHKERRQ(ierr);

    return 0;
}

/**
 * Implements the preconditioned conjugate gradient
 * method with Jacobi preconditioning.
 */
PetscErrorCode PreconditionedConjugateGradient_full(Mat A, Vec b, Vec x,
                            bool VERBOSE) {
    PetscErrorCode ierr;
    Mat Minv;
    Vec diagonal, unity;
    PetscInt n;

    ierr = MatGetSize(A, &n, NULL); CHKERRQ(ierr);
    ierr = CreateMatrix(&Minv, n, n); CHKERRQ(ierr);
    ierr = CreateVector(&diagonal, n); CHKERRQ(ierr);
    ierr = CreateVector(&unity, n); CHKERRQ(ierr);

    ierr = MatAssemblyBegin(Minv, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = MatAssemblyEnd(Minv, MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(diagonal); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(diagonal); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(unity); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(unity); CHKERRQ(ierr);

    // We use the diagonal preconditioner for simplicity
    ierr = MatGetDiagonal(A, diagonal); CHKERRQ(ierr);

    // Compute inverse of all diagonal entries
    ierr = VecSet(unity, 1.0); CHKERRQ(ierr);
    ierr = VecPointwiseDivide(diagonal, unity, diagonal);

    // Create M^{-1}
    ierr = MatDiagonalSet(Minv, diagonal, INSERT_VALUES); CHKERRQ(ierr);
    return PreconditionedConjugateGradient_inner(A, b, x, Minv, VERBOSE);
}
PetscErrorCode PreconditionedConjugateGradient_inner(Mat A, Vec b, Vec x,
                                Mat Minv, bool VERBOSE) {
    PetscErrorCode ierr;
    PetscInt k=0, n;
    PetscScalar mu, nu, yTr, pTz, rNorm, tol = 1e-12;
    Vec p, r, y, z;

    ierr = MatGetSize(A, &n, NULL); CHKERRQ(ierr);

    CreateVector(&p, n);
    CreateVector(&r, n);
    CreateVector(&y, n);
    CreateVector(&z, n);

    VecCopy(b, r);
    ierr = MatMult(Minv, b, p); CHKERRQ(ierr);
    VecCopy(p, y);

    ierr = VecAssemblyBegin(p); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(p); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(r); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(r); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(y); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(y); CHKERRQ(ierr);
    ierr = VecAssemblyBegin(z); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(z); CHKERRQ(ierr);

    ierr = VecZeroEntries(x);

    // Pre-compute first (y^T r)
    ierr = VecDot(y, r, &yTr); CHKERRQ(ierr);

    do {
        k++;

        // z = A * p_k
        ierr = MatMult(A, p, z); CHKERRQ(ierr);

        // nu_k = y_{k-1}^T r_{k-1} / p_k^T z
        ierr = VecDot(p, z, &pTz); CHKERRQ(ierr);
        nu = yTr / pTz;

        // x_k = x_{k-1} + nu_k p_k
        ierr = VecAXPY(x, nu, p); CHKERRQ(ierr);

        // r_k = r_{k-1} - nu_k z
        ierr = VecAXPY(r, -nu, z); CHKERRQ(ierr);

        // y_k = M^{-1} r_k
        ierr = MatMult(Minv, r, y); CHKERRQ(ierr);
        // y_kˆT r_k
        mu = 1 / yTr;
        ierr = VecDot(y, r, &yTr); CHKERRQ(ierr);

        // mu_{k+1}
        mu = yTr * mu;

        // p_{k+1} = r_k + mu_{k+1} p_k
        ierr = VecAYPX(p, mu, y);

        // || r_k ||_2
        ierr = VecNorm(r, NORM_2, &rNorm);
    } while (rNorm > tol);

    if (VERBOSE) {
        PetscPrintf(PETSC_COMM_WORLD, "Number of iterations: %d\n", k);
    }

    return 0;
}
```