5. LARGE-SCALE LINEAR SYSTEMS (Sparse Methods)

5.1 INTRODUCTION

> *"Indeed, without exploiting sparsity many important problems would be intractable"* ([p. 2] Dahlquist & Björck. 2008).
>
> *"Almost all MATLAB operators and functions work seamlessly on both sparse and full matrices"* ([p. 169] Davis 2006).
>
> *"Finding a good fill-reducing ordering is essential for reducing time and memory requirements for sparse factorization methods."* ([p. 130] Davis 2006).
>
> *"Iterative algorithms for solving Ax = b are used when methods such as Gaussian elimination require too much time or too much space*" ([p. 265] Demmel 1997).
>
> *"Large-scale problems in engineering and science often require solution of sparse linear algebra problems, such as systems of equations. The importance of iterative algorithms in linear algebra stems from the simple fact that a direct approach will require O(N^3) work"* ([p. 22] Dongarra & Sullivan 2000).

LU factorizations, the core topic of chapter 4, cannot be used to solve large ($n \geq 10^4$) n-by-n sized linear systems, $A \cdot x = b$. Such algorithms require far too much work, either in the form of too long execution times or in the form of too large computing memory, both resulting from $\sim n^3$ floating-point arithmetic operations necessary to decompose each matrix ([p. 265] Demmel 1997; [p. 243] Trefethen & Bau 1997). Yet, as noted by Trefethen and Bau, the structure of such large-scale matrices is far from random, since such matrices frequently are created in the discretization of integral and differential equations. They observed that the most noticeable structure that these large matrices possess is sparsity, the great majority of the elements of *A* are zeros. Matrix sparsity is used in solutions of large systems of linear equations in two ways. First, there exist efficient sparse variants of the familiar Gaussian elimination techniques, LU and Cholesky factorizations. Second, there are iterative solution algorithms which generate a sequence of ever more accurate approximations that *involve* their n-by-n sized parent matrix *A* only in context of matrix-vector multiplications, which require at most, $\sim 2n^2$ floating-point arithmentic operations per iteration.

There exist efficient LU, QR, and Cholesky factorizations, which can explot the sparsity patterns of their input matrices ([p. 135] Davis 2006). Such sparse direct methods can factor moderate-sized linear systems that cannot be investigated by the usual dense solvers ([p. 75] Saad 2003). However, as we shall see, the sparse factorization algorithms themselves can create large numbers of nonzero entries, termed *fill-in*, which in turn hinders their use. Consequently, application of reordering algorithms, prior to the direct factorization process, is usually necessary. Thus *fill-in* limits the use of the direct algorithms, and, consequently, largest linear systems must be solved with iterative techniques ([p. 284] Beers 2007). Consequently, it is not surprising that direct procedures are the *method of choice* for solving 2D PDE problems but iterative algorithms are the *method of choice* for investigating 3D PDE problems (e.g. O'Leary 2008).

In iterative methods a solution to a large linear system, $A \cdot x = b$, is calculated as a sequence of increasingly more accurate solutions, x_0, x_1, x_2, …, x_k; the iteration process is halted at the kth iteration step when either the solution meets some user-defined error criterion or a maximum number of iterations is exceeded ([p. 31] Demmel 1997; [p. 323] O'Leary 2009). Unlike sparse direct solvers iterative methods do not modify individual matrix entries and the essential arithmetic operation is simply a $\sim 2n^2$ matrix-vector multiplication ([p. 300] Demmel 1997). However. unlike direct methods, the number of iteration steps required for solution convergence is usually

unknown prior to execution. Furthermore, a common phenomenon, which plagues iterative solvers, is that the more ill-conditioned the problem, more slowly iterative methods converge ([p. 285] Demmell 1997). Note that iterative algorithms are the method of choice for solving *large* linear systems when solver methods based on Gaussian elimination require either too much time or too much memory ([p. 265] Demmel 1997). As Demmel notes ([p. 265] 1997) the creation of better, more efficient iterative algorithms *"involves exploiting the underlying mathematical or physical problem that gives rise to the linear system*." Consequently, for an arbitrary large linear system *"it is not clear which method is best*" ([p. 266] Demmel 1997). The large number of iterative solvers can lead to difficulties for the novice user. For example, MATLAB 2011a has *eleven* iterative solvers!

5.2 PROPERTIES OF SPARSE MATRICES

MATLAB possesses a variety of functions to perform matrix operations on sparse arrays. To access MATLAB help on sparse arrays follow

Help tab ⇒ Product Help ⇒ Functions by Category ⇒ Mathematics ⇒ Sparse Matrices

OR

Help tab ⇒ Demos ⇒ Sparse Matrices

NOTE the following key MATLAB *sparse* functions:

issparse	- indicates if the input matrix is stored in a sparse data format.
sparse	- creates a sparse matrix from a *full* matrix.
full	- converts a sparse matrix to a full matrix.
spdiags	- creates a sparse banded matrix.
speye	- creates a sparse identity matrix.
sprand	- creates a *sparse* uniformly distributed random matrix.
nnz	- indicate the number of nonzero matrix elements.
spy	- visualizes the sparsity pattern. (*an important tool)*

Note that in MATLAB only two-dimensional matrices can be stored in sparse format.

The folllowing shows a simple way to create a sparse matrix in the MATLAB environment -

```
>> A = [0  3  0  0  9; 0 8 4 0 0; 6  2  0  0  0; 0 8 0 0 3; 9  0  0 8 0]

A =  0     3     0     0     9
     0     8     4     0     0
     6     2     0     0     0
     0     8     0     0     3
     9     0     0     8     0
>> nnz(A)
ans = 10
>> numel(A)
ans = 25
>> issparse(A)
ans =  0

>> A = sparse(A)
A= (3,1)        6
   (5,1)        9
```

```
   (1,2)         3
   (2,2)         8
   (3,2)         2
   (4,2)         8
   (2,3)         4
   (5,4)         8
   (1,5)         9
   (4,5)         3
>> issparse(A)
ans = 1
>> nnz(A)
ans = 10
```

Alternative scheme for formation of sparse matrices is the following:

```
>> i_rows = [1 1 2 2 3 3 4 4  5 5];
>> j_cols = [2 5 2 3 1 2 2 5  1 4];
>> values = [3 9 8 4 6 2 8 3  9 8];
>> A = sparse(i_rows, j_cols, values, m_rows, n_cols)
A =(3,1)         6
   (5,1)         9
   (1,2)         3
   (2,2)         8
   (3,2)         2
   (4,2)         8
   (2,3)         4
   (5,4)         8
   (1,5)         9
   (4,5)         3
>> B = full(A)
B =  0     3     0     0     9
     0     8     4     0     0
     6     2     0     0     0
     0     8     0     0     3
     9     0     0     8     0
>> nnz(B)
ans = 10
```

Use of sparse attributes create considerable savings in memory when deal with *large* sparse arrays since sparse array only use 3nz storage locations where nz represents the number of nonzero elements (e.g. O'Leary 2005). For example, note the following

```
>> clear;
>> d = 0:2000; A = diag(d);
>> As = sparse(A);
>> [nnz(A) nnz(As)]
ans =  2000        2000   % same number of nonzero elements
>> whos
  Name         Size                Bytes  Class     Attributes
  A         2001x2001            32032008  double
  As        2001x2001               48016  double    sparse
  d            1x2001               16008  double
```

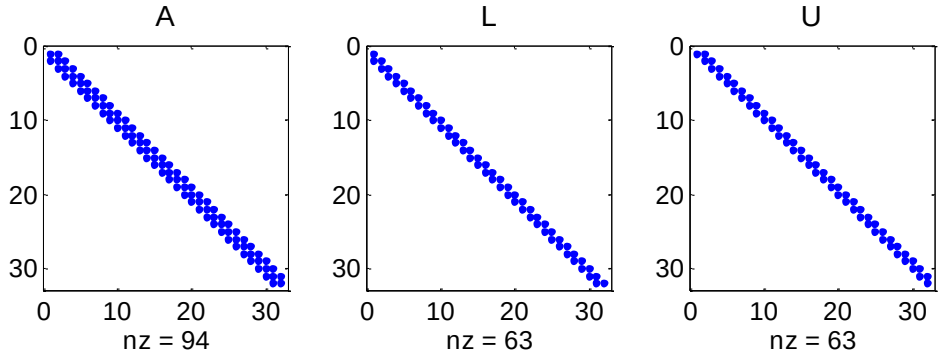
5.3 DIRECT SPARSE FACTORIZATION METHODS

> *"We will just say that direct methods are the methods of choice when the user has no special knowledge about the source of the matirx A or when a solution is required with guarenteed stability and in a guarenteed amount of time"* ([p. 31] Demmel 1997)

Sparse arrays occur in two broad classes, *structured* and *unstructured*. In structured matrices, the nonzero elements possess patterns and often cluster along a number of diagonals (e.g. [p. 75] Saad 2003). The presence or absence of matrix structure affects greatly filllin created by LU factorization.

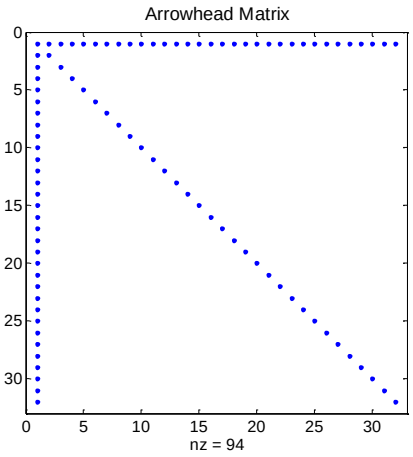
The LU factorization of a simple n-by-n tridiagonal matrix is illustrated below where n = 32. The original matrix has 94 non-zeros (32 along main diagonal, 31 along sub-diagonal, and 31 along super-diagonal) nonzero elements. The corresponding *L* matrix should possess 63 non-zeros (32 along the diagonal and 31 along the sub-diagonal). Similarly, the corresponding *U* matrix should possess 63 non-zeros (32 along the diagonal and 31 along the super-diagonal). The absence of *filllin* is clear in this simple example.

```
>> n = 32; a = 2.0*ones(n, 1); b = -ones(n, 1); c = -ones(n, 1);
>> A = diag(a) + diag(b(2:end), -1) + diag(c(2:end), + 1);
>> [L,U,P] = lu(A);
>> subplot(1,3,1); spy(A); subplot(1,3,2); spy(L); subplot(1,3, 3); spy(U);
>> isequal(P, eye(n,n))
ans = 1
>> [nnz(A) nnz(L) nnz(U) nnz(P)]
ans =  94    63    63    32
```



Let us examine similarly structure *arrowhead* matrix (O'Leary 2005) -

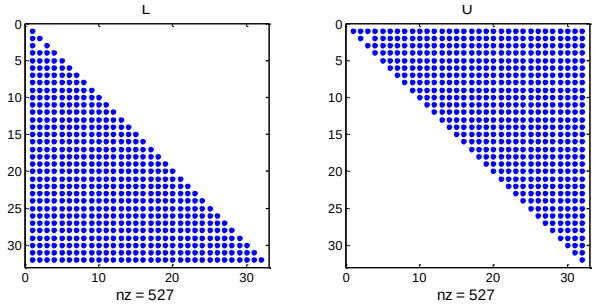
```
>> n = 32; A = eye(n,n); A(1,:) = 1; A(:,1) = 1;
>> [numel(A) nnz(A)]
ans =  1024          94
>> spy(A);
```




```
>> [L, U, P] = lu(A);
```

```
>> [nnz(A) nnz(L) nnz(U) nnz(P)]   % nonzeros increase by ~ 5   → fillin !
ans =   94    527    527     32        % Essentially no zeros in either L or U
>> subplot(1,2,1); spy(sparse(L));  subplot(1,2,2); spy(sparse(U));
```

Now L and U are *fully dense* with n^2 nonzero elements -



5.3.1 Model Poisson Problem

In order to investigate *realistic* large sparse linear systems I present a concrete, but realistic, example. Following the literature, e.g. ([p. 265] Demmel 1997; Persson 2006; [p. 168] Ascher & Greif 2011) I use a finite-difference discretization of a partial differential equation, the Poisson equation, defined over a unit square, $0 \le x \le 1$, $0 \le y \le 1$,

$$-\left(\frac{\partial^2 U}{\partial^2 x} + \frac{\partial^2 U}{\partial^2 y}\right) = g(x, y),$$

where U(x, y) represents the unknown function, which is to be determined, and g(x, y) denotes a *given* source function. Furthermore, one assumes that the unknown function, U(x, y), satisfies *homogeneous* Dirichlet boundary conditions -

$$U(0,y) = 0.0,$$
$$U(1,y) = 0.0,$$
$$U(x,1) = 0.0,$$
$$U(x,0) = 0.0.$$

To solve Poisson's equation, I create a uniform square grid in x-y Cartesian space and identified the spatial grid coordinates as, where the grid width, h = 1.0/(n + 1),

$$(x_i, y_j) = (ih, jh), \quad i, j = 0{:}n + 1.$$

Approximating the partial derivatives by second-order finite differences, one finds that the partial differential equation can modeled by a system of linear equations -

$$4U_{i,j} - U_{i+1,j} - U_{i-1,j} - U_{i,j+1} - U_{i,j-1} = B_{ij}, \quad 1 \le i \le n \quad 1 \le j \le n$$
$$U_{i,j} = 0, \quad \text{otherwise},$$

where U_{ij} represents the value of the unknown function at the ij grid node (x = ih &y = jh) and $B_{ij} = h^2 gij = h^2 g(ih, jh)$. These relations can be represented as a system of linear equations,

$$A \cdot U = b$$

where the variable U_{ij}, (i = 1:n and j = 1:n) is somehow represented as an n^2-by-1 element vector! A simple, but non-unique, approach is stack U_{ij} column by column -

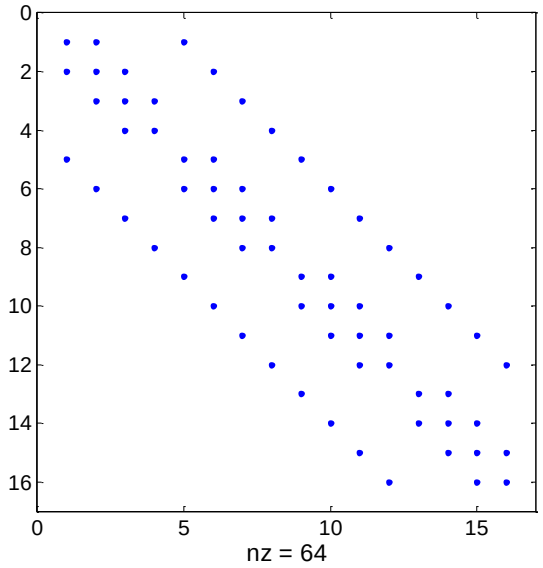
$$u = \begin{pmatrix} U_{11} \\ U_{21} \\ \vdots \\ U_{n1} \\ U_{12} \\ U_{22} \\ \vdots \\ U_{n2} \\ U_{13} \\ \vdots \\ U_{nn} \end{pmatrix}, \quad b = \begin{pmatrix} B_{11} \\ B_{21} \\ \vdots \\ B_{n1} \\ B_{12} \\ B_{22} \\ \vdots \\ B_{n2} \\ B_{13} \\ \vdots \\ B_{nn} \end{pmatrix}.$$

Observe that the matrix A becomes a large n^2-by-n^2 sparse matrix. For example, for n = 4, and g(x, y) = 1, A·u = b, becomes -

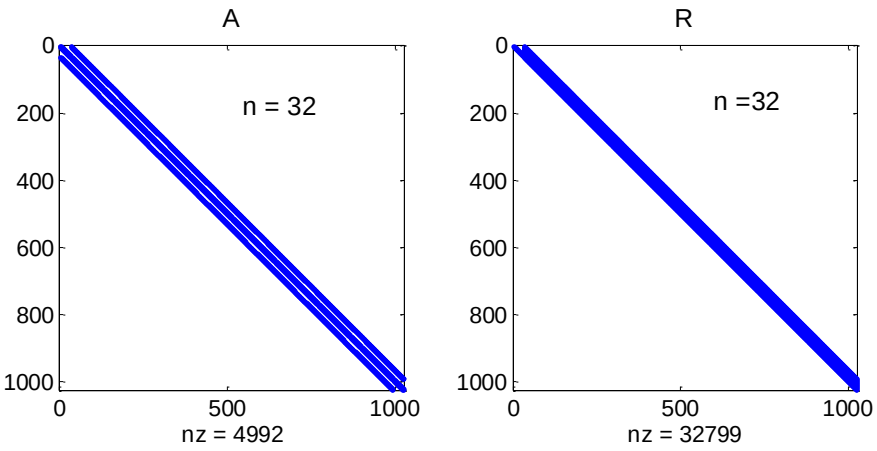
$$\begin{bmatrix} 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \\ u_{13} \\ u_{14} \\ u_{15} \\ u_{16} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \\ b_{10} \\ b_{11} \\ b_{12} \\ b_{13} \\ b_{14} \\ b_{15} \\ b_{16} \end{bmatrix} = \begin{bmatrix} -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \end{bmatrix}$$

Now I create this sparse array using MATLAB -

```
>> n = 4; I = eye(n,n);
>> D = 2*diag(ones(n,1)) - diag(ones(n - 1, 1), -1) - diag(ones(n - 1, 1), +1);
>> A = kron(D, I) + kron(I, D); % Uxx = kron(D, I) and Uyy = kron(I, D)
>> [numel(A) nnz(A)]
ans = 256    64
ans =  9.4721       → % nonsingular
>> isequal(A, transpose(A))
ans = 1             → % symmetric
>> all( eig(A) > 0 )
ans = 1             → % positive definite → DO NOT have to pivot for stability
>> spy(A)
```

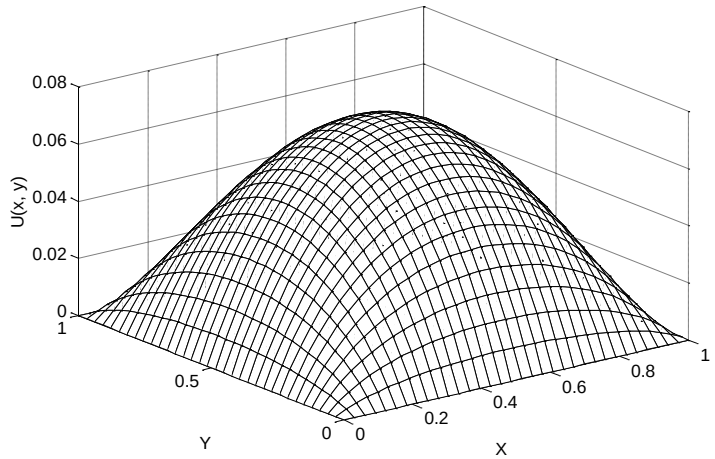



```
function    [A, b] = poisson_model(n)
% Sparse form of the above MATLAB code fragment
% Condition number ~ 4(n + 1)^2/pi^2  (Persson P O 2006)
h  = 1.0/(n + 1);
e  = ones(n,1);
A1 = spdiags([-e,2*e,-e],-1:1,n,n);
A  = kron(A1,speye(n,n))+kron(speye(n,n),A1);
b  = h*h*sparse(ones(n^2, 1));
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


>> n = 32;
>> [A, b] = poisson_model(n);
>> size(A)
ans =  1024        1024  % → Only  0.0047607 of A nonzero!
>> issparse(A)
ans = 1
>> nnz(A)
ans =  4992
>> condest(A)
ans = 640.36              % Condition number
% Remember without keyword 'lower' chol returns UPPER triangle
>> R = chol(A);
% R'*R = A →  A*u = b → R'*R*u = b → y = R'\b  &  u = R\y → u = R(R'\b)
>> issparse(R)
ans = 1
>> nnz(R)
ans = 32799    % fillin increases number of nonzero elements by ~6.5!
>> subplot(1,2, 1); spy(A); subplot(1,2, 2); spy(R)
```



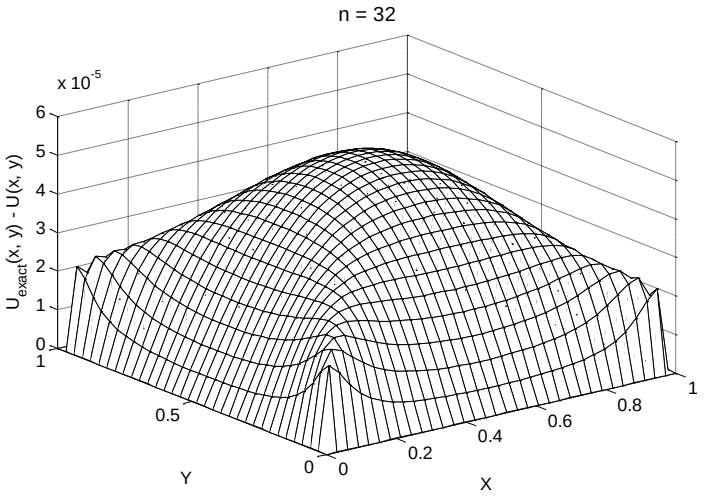
Now solve the sparse system of linear equations with \ operator:

```
% linsolve does NOT work with sparse data type
>> issparse(b)
ans = 1
>> u = R\(R'\b);            % vector u associated in internal grid points
>> issparse(u)              % u does NOT include boundary values where u = 0
ans =  1
>> length(u)
ans = 1024                   % length(u) = n^2 = 32^2 = 1024
>> Ui = reshape(u, n, n);    % Creates a 2D grid of interior values of U
>> dmin = 0.0; dmax = 1.0; h = 1.0/(n + 1);
>> d    = (dmin:h:dmax);
>> [X, Y] = meshgrid(d, d); % Now ncludes x & y values of boundary
>> size(X)
ans =  34    34
>>  U = zeros(size(X));  % All U values (interior & boundary) set to 0
>> U(2:n + 1, 2:n + 1) = reshape(u, n, n);  % Assigns interior U values
>> colormap([0 0 0]);     % Sets mesh colors to be black
>> mesh(X, Y, U);
>> xlabel('X'); ylabel('Y'); zlabel('U(x, y)');
```




```
>> norm_of_residual = norm( A*u - f, 1)
norm_of_residual =   3.421e-014    % norm of residual is very small
>> Ue = exact_solution_poisson01(X, Y);
>> norm( (Ue - U), 1)
ans =    0.0011925
```

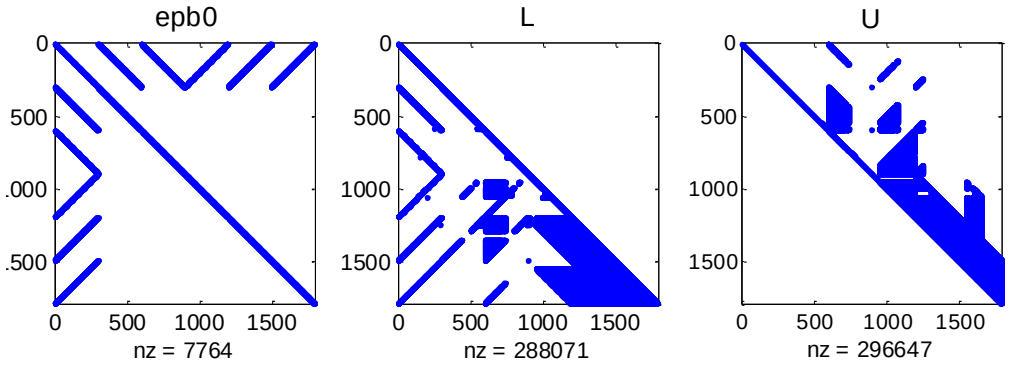
```
>> max(max(abs(Ue - U)))
ans = 5.3164e-005
```



5.3.2 Reordering Algorithms

Note the unusual structure of the following matrix, epb0. Also, it is not symmetric nor is it a positive definite matrix -

```
>> load epb0
>> [size(epb0) nnz(epb0)]
ans = 1794        1794        7764
>> issparse(epb0)
ans = 1
>> tic; [L, U, P] = lu(epb0);  toc
Elapsed time is 0.355890 seconds.
>> [nnz(epb0) nnz(L) nnz(U) nnz(U)/nnz(epb0)]
ans   7764   2.8807e+005   2.9665e+005          38.208
% fillin increased number of nonzeros by 38!
>> subplot(1,3,1); spy(epb0);
>> subplot(1,3,2); spy(L); subplot(1,3,3); spy(U)
```



Now reordering schemes of this of this matrix are examined in an attempt to reduce *fillin* created during factorization. Fill reducing strategies and algorithms are discussed in detail in chapter seven of *Direct Methods for Sparse Linear Systems* (Davis 2006).

Perusal of MATLAB's (R2011a) online help identifies the following *black-box* re-ordering algorithms:

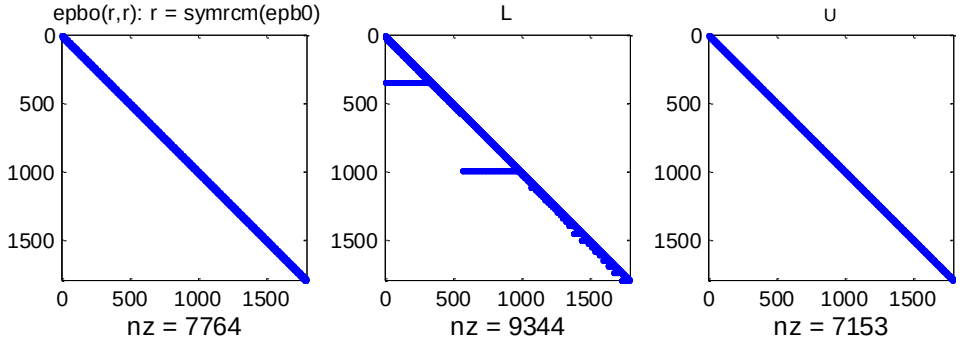
- **amd** Approximate minimum degree permutation.
- **colamd** Column approximate minimum degree permutation.
- **colperm** Sparse column permutation.
- **dmperm** Dulmage-Mendelsohn decomposition.
- **symamd** Symmetric approximate minimum degree permutation.
- **symrcm** Sparse reverse Cuthill-McKee ordering.

a) Reverse Cuthill-McKee Permutation

```
   >> help symrcm  % MATLAB online help
SYMRCM Symmetric reverse Cuthill-McKee permutation.
       p = SYMRCM(S) returns a permutation vector p such that S(p,p)
       tends to have its diagonal elements closer to the diagonal than S.
       This is a good preordering for LU or Cholesky factorization of
       matrices that come from "long, skinny" problems.  It works for
       both symmetric and nonsymmetric S.

  >> tic; r = symrcm(epb0); [L, U, P] = lu(epb0(r, r)); toc
  Elapsed time is 0.005846 seconds.
  >> [ nnz(epb0) nnz(L) nnz(U) nnz(L)/nnz(epb0)]
  ans =   7764         9344         7153       1.2035
```

This is an *impressive* result - the factorization execution speed dropped by factor of ~50 and *fillin* dropped from 38 to 1.2 when reverse Cuthill-McKee reordering is used!

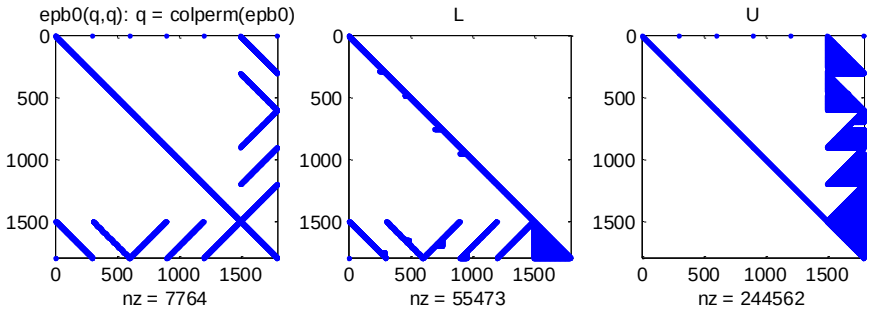
```
>> subplot(1,3,1); spy(epb0(r,r)); subplot(1,3,2); spy(L); subplot(1,3,3); spy(U);
```



b) Column Permutation

```
   >> help colperm % MATLAB online help
COLPERM Column permutation.
   p = COLPERM(S) returns a permutation vector that reorders the
   columns of the sparse matrix S in non-decreasing order of nonzero
   count.  This is sometimes useful as a preordering for LU
   factorization: lu(S(:,p)).

   If S is symmetric, then COLPERM generates a permutation so that
   both the rows and columns of S(p,p) are ordered in nondecreasing
   order of nonzero count.  If S is positive definite, this is
   sometimes useful as a preordering for Cholesky factorization:
   chol(S(p,p)).
```

```
COLPERM is not the best ordering in the world, but it's fast to
compute, and it does a pretty good job.


>> tic; q = colperm(epb0); [L, U, P] = lu(epb0(q, q)); toc
Elapsed time is 0.084831 seconds.
>> [nnz(epb0) nnz(L) nnz(U) nnz(U)/nnz(epb0)]
ans =   7764       55473   2.4456e+005       31.499

>> subplot(1,3,1); spy(epb0(q,q)); subplot(1,3,2); spy(L); subplot(1,3,3); spy(U)
```



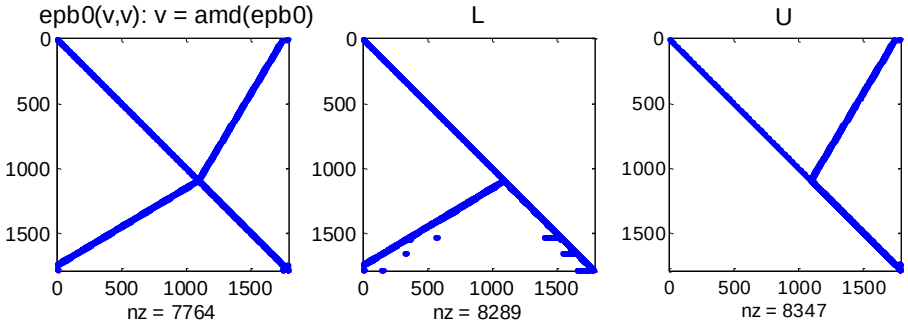
Observe here only a factor of 4 decrease in factorization execution time but only a modest decrease in *fillin.* Similar behavior was found using **dmperm**.

c) Minimum Degree Permutation

```
>> help amd          % MATLAB online help
AMD  Approximate minimum degree permutation.
    P = AMD(A) returns the approximate minimum degree permutation vector
    for the sparse matrix C = A + A'.  The Cholesky factorization of C(P,P),
    or A(P,P), tends to be sparser than that of C or A.  AMD tends to be
    faster than SYMAMD. A must be square. If A is full, AMD(A) is
    equivalent to AMD(SPARSE(A))

    >> tic; v = amd(epb0); [L, U, P] = lu(epb0(v,v)); toc
    Elapsed time is 0.008799 seconds.
    >> [nnz(epb0) nnz(L) nnz(U) nnz(U)/nnz(epb0)]
    ans = 7764         8289         8347       1.0751
```

Observe a factor of almost ~40 decrease in execution time and a decrease in *fillin* to a negligible leve (7.5%)l compared to the corresponding values found in the absence of reordering!

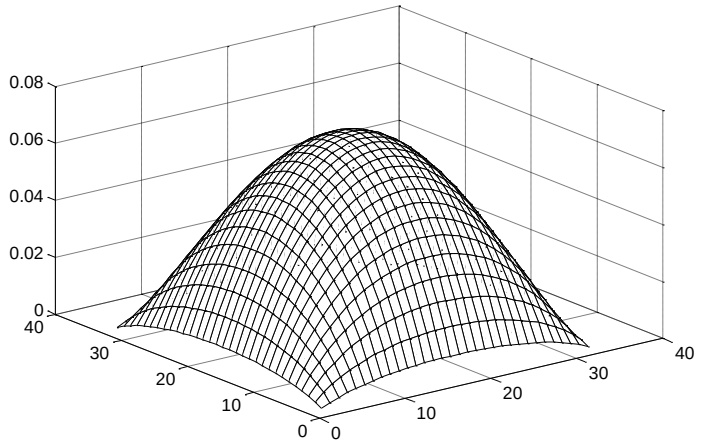
```
>> subplot(1,3,1); spy(epb0(v, v));
>> subplot(1,3,2); spy(L); subplot(1,3,3); spy(U);
```



Let us apply reordering algorithms to poisson example -

```
>> n = 32; [A, b] = poisson_model(n);
>> tic; R = chol(A);  u = R\(R'\b);
 disp([  nnz(A)      nnz(R)      nnz(R)/nnz(A) condest(A, 1)] ); toc
         4992        32799       6.5703        640.36
Elapsed time is 0.022313 seconds.
>> tic; r = amd(A);  R = chol(A(r, r));  u = R\(R'\b(r));
disp([   nnz(A)      nnz(R)      nnz(R)/nnz(A) condest(A, 1)] ); toc
         4992        11900       2.3838        640.36
Elapsed time is 0.028435 seconds
% NOTE use  below of reverse vector in constructing mesh!
>> reverse(r) = (1:length(r)); colormap([0 0]);
mesh( reshape(u(reverse), n, n) ); % Inspect plot below!
```

A *significant* savings in execution time or storage due to reordering is not apparent for n = 32!



Note the following computations as n is raised from 64 to 256 when **amd** reordering is used -

```
>> clear; n = 64; [A, b] = poisson_model(n);
>> tic; R = chol(A);  u = R\(R'\b);
  disp([nnz(A) nnz(R)        nnz(R)/nnz(A) condest(A, 1)] ); toc
        20224  2.6221e+005   12.965        2488.6
Elapsed time is 0.099120 seconds.
>> tic; r = amd(A);  R = chol(A(r, r));  u = R\(R'\b(r));
  disp([nnz(A)       nnz(R)      nnz(R)/nnz(A) condest(A, 1)] ); toc
        20224        67200       3.3228        2488.6
Elapsed time is 0.056089 seconds.

>> clear; n = 256; [A, b] = poisson_model(n);
>> tic; R = chol(A);  u = R\(R'\b);
disp([nnz(A)       nnz(R)         nnz(R)/nnz(A)    condest(A, 1)] ); toc
      3.2666e+005  1.6777e+007       51.361        38926
Elapsed time is 3.845145 seconds.
>> tic; r = amd(A);  R = chol(A(r, r));  u = R\(R'\b(r));
disp([nnz(A)       nnz(R)           nnz(R)/nnz(A) condest(A, 1)] ); toc
      3.2666e+005  1.9714e+006      6.0351        38926
Elapsed time is 1.305669 seconds.

>> clear; n = 512; [A, b] = poisson_model(n);
>> tic; R = chol(A);  u = R\(R'\b);
disp([nnz(A)      nnz(R)         nnz(R)/nnz(A) condest(A, 1)] ); toc
     1.3087e+006  1.3422e+008    102.56        1.551e+005
Elapsed time is 32.836107 seconds.
```

```
>> tic; r = symamd(A);  R = chol(A(r, r));  u = R\(R'\b(r));
disp([nnz(A)       nnz(R)          nnz(R)/nnz(A) condest(A, 1)] ); toc
      1.3087e+006  9.8972e+006     7.5628        1.551e+005
Elapsed time is 6.975859 seconds.
```

These latter computations illustrate key features of employing reordering algorithms in the solution of positive definite linear systems -

- Effects of reordering increase with size of input matrix.
- Reordering dramatically decreases factorizations execution times as matrix size increases.
- Reordering dramatically decreases sizes of factored matrices.

5.4 INTERATIVE SOLUTIONS TO SYSTEMS OF LINEAR EQUATIONS

> *"The term iterative method refers to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step*." ([p. 5] Barrett et al. 1993)
>
> *"For Poisson's equation, there will be a short list of numerical methods that are clearly superior to all others we discuss. But for other linear systems it is not always clear which method is the best (which is why we talk about so many!) "*
> ([p. 266] Demmel 1997).

Sparse *direct* methods, employing reordering strategies, are effective for solving modest-sided, two-dimensional PDEs but the large-scale linear systems created by the discretization of all but the simplest three-dimensional PDEs require the use of iterative techniques e.g. ([p. 68] Elman, Silvester, & Wathen 2005). The properties of iterative solvers are addressed now.

Iterative methods are useful sometimes beyond solving large-scale linear systems ([p. 168] Ascher & Greif 2011). First, sometimes it is not necessary to solve even modest-sized linear systems *exactly*. For example, model linear systems created in the discretization of differential equations are only approximations possessing modest degrees of accuracy. However, iterative solvers can terminated at any level of accuracy. Direct methods cannot achieve this since these methods must solve a linear system down to the float-point machine precision. Second, one has a good estimate of a solution to a modeled linear system. In modeling time-dependent phenomena, an accurate solution at certain time can serve well as initial solution estimate at a subsequent time. With iterative solvers, such estimates can be employed efficiently as initial guesses, yet direct solution algorithms make no use of initial guesses.

An iterative method, starting with an initial guess, $x^{(0)}$, creates a sequence of approximations, $x^{(1)}$, $x^{(2)}$, … which *hopefully* converges to the appropriate solution vector; here $x^{(k)}$ represents the solution vector x at the kth iteration stage. Such an iterative solution requires less memory and may be computed more quickly than solutions generated by direct factorizations ([lecture 18] Persson 2006). There are two types of iterative solvers: stationary and non-stationary, e.g. ([p. 5] Barrett et al. 1993; [p. 323] O'Leary 2009). Stationary algorithms are simple to understand and implement but are ineffective ([p. 5] Barrett et al. 1993. Non-stationary methods differ from stationary methods through the involvement of changing information at every iteration and are based on sequences of orthogonal vector spaces, and, consequently, are more difficult to understand and implement, but are highly effective ([p. 7] Barrett et al. 1993).

Iterative solution algorithms for A·x = b can be represented as

$$x^{(k+1)} = Rx^{(k)} + c.$$

Those with constant *R* and *c* are termed *stationary* iterative methods (SIM).The Jacobi, Gauss-Seidel, and successive overrelaxation (SOR) techniques are the primary stationary methods and the derivation of such stationary methods is based on operator splitting:

$$\begin{aligned} A &= M - K, \\ A \bullet x &= b = (M - K) \cdot x, \\ M \bullet x &= K \bullet x + b, \\ x &= M^{-1} \cdot (K \cdot x + b) = M^{-1} \cdot ((M - A) \cdot x + b), \\ M \cdot x^{(k)} &\cong (M - A)x^{(k-1)} + b, \end{aligned}$$

where $x^{(k)}$ represents the solution vector, *x*, at the iteration count *k*. One investigates operator splitting with the $A \equiv D - L - U$, where *D* is the diagonal part of *A* and -*L* and -*U* represent the strictly upper and strictly lower triangular parts of *A* ([p. 7] Barrett et al. 1993).

5.4.1 Stationary Methods

a) Jacobi Method

The Jacobi method can be derived by isolating a single row of a linear system A· x = b, and solving that the row equation for x_i, assuming that the other values of *x* remain fixed -

$$x_i = (b_i - \sum_{j \neq i}^{n} A_{ij} x_j)/A_{ii}$$

([p. 7] Barrett et al.1993). Such a simple relationship suggests an obvious iterative algorithm,

$$x_i^{(k+1)} \cong (b_i - \sum_{j \neq i}^{n} A_{ij} x_j^{(k)})/A_{ii},$$

which constitutes the Jacobi method. Note that the diagonal elements *must* not be zero. Alternatively, using a matrix format the Jacobi method can be found using matrix splitting where $M = D$ and $M - A = L + U$, and $x^{(k+1)}$ in matrix format can be expressed as e.g. ([p. 7] Barrett et al. 1993; [p. 281] 1997) -

$$D \cdot x^{(k+1)} \cong (L + U) \cdot x^{(k)} + b.$$

The Jacobi method converges if the n-by-n parent matrix, *A*, is *diagonally dominant*, either *strictly* row diagonally dominant,

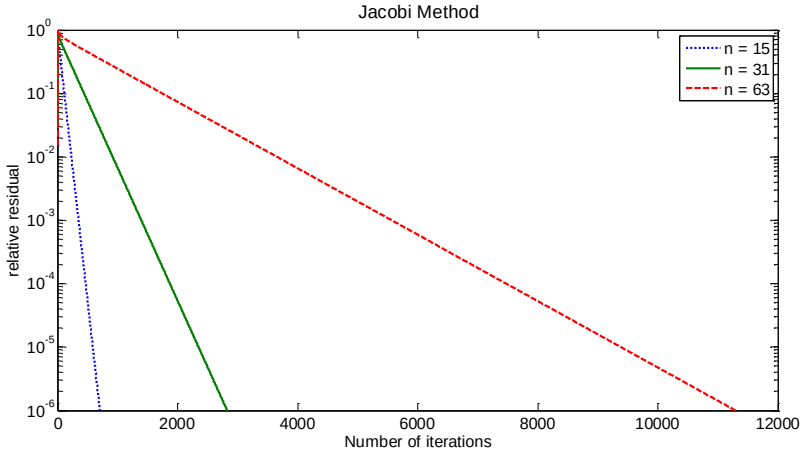
$$|A_{ii}| > \sum_{j \neq i} |A_{ij}|,$$

or *weakly* row diagonally dominant,

$$|A_{ii}| \geq \sum_{j \neq i} |A_{ij}|$$

where the equality applies at least once ([Lecture 17] Persson 2006). The Jacobi method, perhaps not surprisingly, is inefficient. The Jacobi method converges *slowly;* its costs $O(n^2)$ iterations to decrease the error by e^{-1} (or by any constant factor less than 1) for an n-by-n sized matrix *A* ([p. 285] Demmel 1997). Employing the **poisson_model.m** the Jacobi algorithm, listed below, requires 11303 and 45194 iterations respectively to reach a relative residual of 10^{-6} for an *n* of 31 and 63. The next figure illustrates the efficiency of the Jacobi algorithm applied to **poisson_model.m,** quantified as the relative residual norm, ||r||/||b||, computed as a function of the number of required iterations for *n* = 15, 31, and 63.

```
function [x, error, iter, flag, residual, n] = jacobi(A, x, b, maxit, tol)

  M           = diag(diag(A));          % D = diagonal components of A
  N           = diag(diag(A)) - A;      % N = + L + U
                          r            = b - A*x;
  r0          = norm(r);
  normb       = norm(b);
  error       = r0/normb;
  flag        = 0;                      % Set flag to be false, 0
  if (error < tol); return; end
  residual(1) = r0;

     for iter = 2:maxit                 % A*x = b => x = A\b % MATLAB backslash
         x              = M\(N*x + b);  % x = D\((L + U)*x + b) = D\(N*x + b)
         r              = b - A*x;
         residual(iter) = norm(r)/normb;
         n(iter)        = iter;
         error          = norm(r)/normb;
         if (error < tol); flag = 1; break; end
     end
```



b) Gauss Seidel Method

The Gauss-Seidel method is yet another classic iterative method that dates from the mid1800s. It is much too slow to be competitive with modern non-stationary methods and it is only applicable to diagonally dominant or symmetric, positive define matrices ([p. 28] Barrett et al. 1993). It is a simple extension of the Jacobi method. In the Jacobi method the components of $x_i^{(k-1)}$ are employed to compute $x_i^{(k)}$ as noted above. In the Jacobi algorithm at the k^{th} iteration stage of itera-

tion, $x_1^{(k)} \ldots x_{j-1}^{(k)}$ have already computed, and are likely better approximations to $x_1 \ldots x_{i-1}$ than the $x_1^{(k-1)} \ldots x_{i-1}^{(k-1)}$ elements. It would seem plausible to calculate $x_i^{(k)}$ using the most recently calculated values. The recent calculated values are included in the Gauss-Seidel method which in component form is

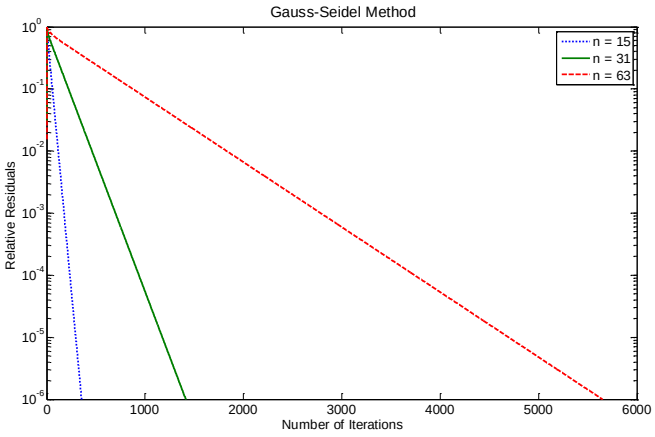
$$x_i^{(k)} = \frac{-\sum_{j=1}^{i-1}(A_{ij}x_j^{(k)}) \quad -\sum_{j=i+1}^{n}(A_{ij}x_j^{(k-1)}) + b_i}{A_{ii}}, \text{ for } i = 1{:}n$$

([p. 9] Barrett et al.1993; [p. 282] Demmel 1997). Using operator splitting the Jacobi method, where $M = D - L$ and $M - A = L + U$, $x^{(k+1)}$ can be represented as e.g. ([p. 9] Barrett et al. 1993) -

$$(D - L) \cdot x^{(k+1)} \cong U \cdot x^{(k)} + b.$$

The Gauss-Seidel method is inefficient also. It converges only twice as fast as the Jacobi method (Persson 2006). Employing the **poisson_model.m** the Gauss-Seidel algorithm, listed below, requires 1415 and 5653 iterations respectively to reach a relative residual of 10^{-6} for an n of 31 and 63. These numbers show that Gauss-Seidel algorithm reaches the relative residual level of 10^{-6} with about a third the number of iterations required by Jacobi method. The next figure illustrates the efficiency of the Jacobi algorithm applied to **poisson_model.m**, quantified as the relative residual norm, ||r||/||b||, computed as a function of the number of required iterations for n = 15, 31, and 63.

```
function [x, error, iter, flag, residual, n] = gauss_seidel(A, x, b, maxit, tol)

  M           =  tril(A);       % lower includes diagonal
  N           = -triu(A, 1);    % strictly upper triangular matrix (no diagonal)
  r           = b - A*x;
  r0          = norm(r); normb  = norm(b); error = r0/normb;
  flag        = 0;              % Set flag to be false, 0
  if (error < tol); return; end
  residual(1) = r0;

     for iter = 2:maxit
         x               = M\(N*x + b);
         r               = b - A*x;
         residual(iter)  = norm(r)/normb;
         n(iter)         = iter;
         error           = residual(iter);
         if (error < tol); flag = 1; break; end
     end
```



c) Successive Overrelaxation Method (SOR) Method

The SOR method is created via extrapolation of the Gauss-Seidel method by computing a weighted average of the present Gauss-Seidel iterate and the past Gauss-Seidel iterate -

$$x_i^{(k)} = \omega \underline{x}_i^{(k)} + (1 - \omega) x_i^{(k-1)},$$

where $\underline{x}$ represents a Gauss-Seidel iterate and ω is the extrapolation factor ([p. 10] Barrett et al. 1993). Using matrix terminology, $x^{(k+1)}$ can be represented here as

$$(D - \omega L) \cdot x^{(k+1)} \cong (\omega U + (1 - \omega) D) \cdot x^{(k)} + \omega b$$

([p. 9] Barrett et al. 1993; [p. 284] Demmel 1997). When $\omega = 1$, the SOR method becomes the Gauss-Seidel method. Unfortunately, it is "*not possible to compute in advance the optimum value of* ω"; moreover, when one *can* compute ω, the cost of computation is "*usually prohibitive*" ([p. 10] Barrett et al. 1993). For symmetric positive definite parent matrices $0 \leq \omega \leq 2$ and some heuristic estimates suggest that $\omega \approx 2 - O(h)$, where h is the mesh spacing of the discretization of the underlying spatial domain ([p. 10] Barrell et al. 1993). At the *optimal* ω value, the SOR algorithm is *n* times faster than Jacobi/Gauss-Seidel methods reaching a constant factor improvement every O(n) iterations (Persson 2006)

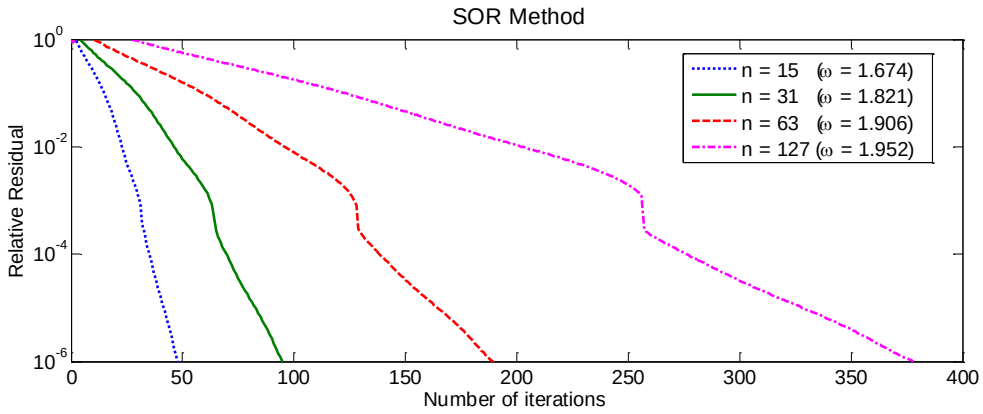
The SOR method is very efficient compared to the Gauss-Seidel algorithm for the optimal ω value. For a square, unit-sized, n-by-n linear system, employing, **`poisson_model.m`**, where h = $(n+1^{)-1}$, and ω can be shown ([p. 178] Ascher & Greif 2011) to be

$$\omega = 2/(1 + \sin(\pi h)),$$

the SOR algorithm, listed below, requires *only* 95 and 190 iterations to reach a relative residual of 10^{-6} for an *n* of 31 and 63 respectively. These numbers show that the optimal SOR algorithm requires factors of n less iterations (actually factors of 15 and 30) than those required by the corresponding Gauss-Seidel algorithm to the Poisson model problem.

```
function [x, error, iter, flag, residual, n] = sor(A, x, b, w, maxit, tol)

  b           = b*w;            % redefined
  M           = +w*tril(A, -1) + diag(diag(A));
  N           = -w*triu(A, +1) + (1.0 - w)*diag(diag(A));
  r           = b/w - A*x;
  r0          = norm(r);
  normb       = norm(b/w);
  error       = r0;
  flag        = 0;
  if (error < tol); return; end
  residual(1) = r0/normb;

     for iter = 2:maxit
         x              = M\(N*x + b);
         r              = (b/w) - A*x;
         residual(iter) = norm(r)/normb;
         n(iter)        = iter;
         error          = norm(r)/normb;
         if (error < tol); flag = 1; break; end
     end
```

The next figure illustrates the efficiency of the SOR algorithm applied to the standard **`poisson_model.m`**, quantified as the relative residual norm, ||r||/||b||, computed as a function of the number of required iterations for *n* = 15, 31, 63, and 127. The above relation (Ascher & Greif 2011) is employed to represent the optimum n-dependent expression for ω.



5.4.2 Non-Stationary Krylov Methods

a) Conjugate Gradient

> *"The conjugate gradient iteration is the original Krylov subspace iteration, the most famous of these methods and the one of the mainstays of scientific computing."* ([p. 293] Trefethen & Bau 1997),

The conjugate gradient method (CG) is the *algorithm of choice* for solving linear systems created by symmetric positive definite matrices ([p. 307] Demmel 1997). This method is one of the ten most important algorithms in scientific computing (Dongarra & Sullivan 2000). Van der Vost (2000) discusses the history and current importance of these Krylov iteration algorithms.

Following LeVeque ([p. 78] 2007) we initiate the CG discussion by interpreting this technique as a *descent method* for solving a minimization problem. Note that a function φ(u) can be defined to represent A·u = b as

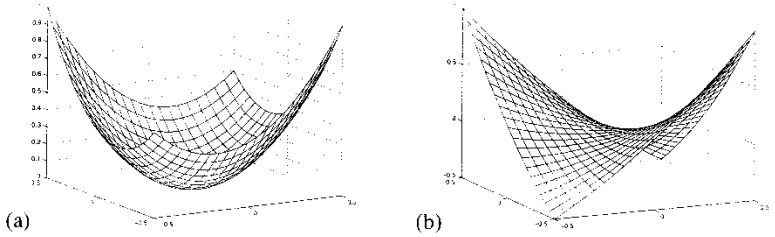
$$\phi(u) = \frac{1}{2} u^T \cdot A \cdot u - u^T \cdot b,$$

which is a quadratic vector function of the vector, $u = u_1 \ldots u_n$. As an example, setting n = 2, thus $u = [u_1\ u_2]$ -

$$\phi(u) = \phi(u_1, u_2) = \frac{1}{2}(A_{11}u_1^2 + 2A_{12}u_1u_2 + A_{22}u_2^2) - u_1b_1 - u_2b_2.$$

Note that since A is symmetric, $A_{12} = A_{21}$. If *A* is *positive definite*, then plotting φ(u) as function of u_1 and u_2 creates a *parabolic* bowl shown in the left panel a in the figure below ([p. 79] LeVeque 2007) shown immediately below. Panel b illustrates the shape of φ(u), a saddle point, created by a matrix which is *not* a symmetric positive definite array. Inspect φ(u) created by symmetric posi-

tive definite arrays, panel a. For such shapes there exists a unique minimum. At such minima, the partial



derivatives of φ with respect to each component of u are zero, which then are

$$\frac{\partial \phi}{\partial u_1} = A_{11}u_1 + A_{12}u_2 - b_1 = 0,$$

$$\frac{\partial \phi}{\partial u_2} = A_{21}u_1 + A_{22}u_2 - b_2 = 0.$$

These are components of A·u = b, the solution we are searching for. Finding u*, which minimizes $\varphi(u)$, is equivalent to solving the n = 2 linear system, A·u = b. For n > 2, the function $\varphi(u)$ still has a unique minimum u*, at the point where $\nabla\phi(u^*) = 0$, and there $\nabla\phi(u^*) = A \bullet u - b = 0$, Consequently, determining the minimum of $\varphi(u)$, associated with symmetric positive definite *A*, solves the linear system, A·u = b! If *A* is *indefinite*, then $\varphi(u)$ still has a minimum at u*, where $\nabla\phi(u^*) = 0$, but this location is a saddle point, such as is illustrated in the right panel b in the figure above.

The CG approach is determine u* in an iterative approach. If one has an estimate of u* at the k -1 iteration stage, the estimate can be improved by noting that ∇u points in the direction of the maximum increase of φ with u. Thus one improves the estimate of u^k via

$$u^k = u^{k-1} - \alpha_{k-1}\nabla\phi(u^{k-1}),$$

by solving for a scalar α_{k-1}. Obviously $\alpha_{k-1} = 0$ only if $u^{k-1} = u^*$. At u^{k-1} the gradient is

$$\nabla\phi(u^{k-1}) = A \cdot u^{k-1} - b \equiv -r^{k-1},$$

where r^{k-1} is the residual vector based on the approximation, u^{k-1}. Now to determine α^{k-1}, we calculate the derivative of $\varphi(u)$, but now with respect to α, and set *this* function equal to 0. Thus

$$\phi(u + \alpha r) \equiv \left(0.5u^T \cdot A \cdot u - u^T \cdot b\right) + \alpha\left(r^T \cdot A \cdot u - r^T \cdot b\right) + 0.5\alpha^2 r^T \cdot A \cdot r,$$

$$\frac{d\phi(u + \alpha r)}{d\alpha} = r^T \cdot A \cdot u - r^T \cdot b + \alpha r^T \cdot A \cdot r = 0, \Rightarrow \text{solving for } \alpha,$$

$$\alpha = \left(r^T \cdot r\right) / \left(r^T \cdot A \cdot r\right).$$

Thus the search algorithm becomes

Chose initial guess, u_0,

while ($\left\| r^k \right\| \leq$ tolerance)

$r^{k-1} = b - A \cdot u^{k-1}$;

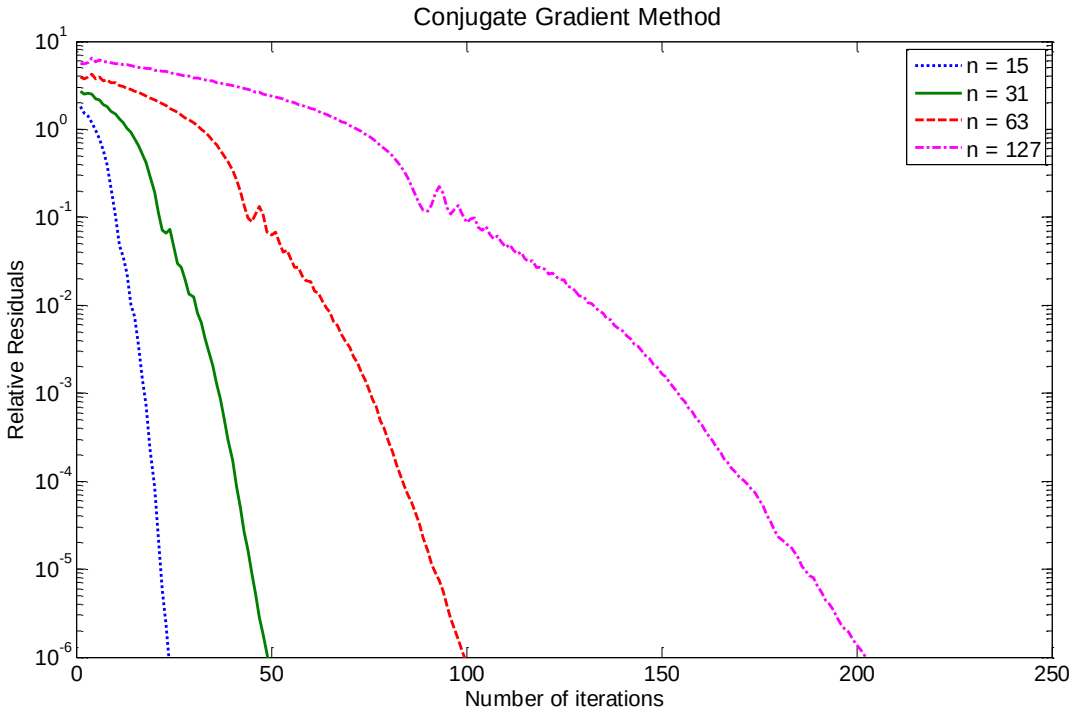
$\alpha^{k-1} = (r^{k-1} \cdot r^{k-1}) / (r^{k-1} \cdot A \cdot r^{k-1})$;

$u^{k-1} = u^{k-1} + \alpha^{k-1} r^{k-1}$;

end

At each iteration step the CG method computes several vector manipulations as well as one matrix-vector product, the calculation of $A \cdot u^{(k-1)}$. If *A* is dense, this matrix-vector product controls the count of arithmetic floating-point operations, ~$2n^2$, per iteration; if *A* is sparse, then the matrix-vector product can be computed in ~ n operations, and thus number of operations per iteration step can be as low as ~ O(n) ([p. 295] Trefethen & Bau 1997).

The conjugate gradient method is efficient, converging quickly *for well-behaved* matrices, employing ~O(n) iterations for n-by-n sized coefficient arrays. Employing the **`poisson_model.m`** the cg algorithm, listed below, requires 50, 100, and 203 iterations respectively to reach a relative residual of 10^{-6} for an *n* of 31, 63, and 127. The next figure illustrates the efficiency of the Jacobi algorithm applied to **`poisson_model.m`**, quantified as the relative residual norm, ||r||/||b||, computed as a function of the number of required iterations for *n* = 15, 31, 63, and 127.



```
function [x, error, k, flag, residual, n] = cg(A, x, b, maxit, tol)
  % Algorithm 38.1 Conjugate Gradient (Trefethen, L N & Bau, D  p. 294
  %   Numerical Linear Algebra  [SIAM; Philadelphia]  (1997)
  % Ascher, U M & Greif C (2011) p. 184
  %   A First Course in Numerical Methods (SIAM: Philadelphia)

  r           = b - A*x;
  p           = r;
  normb       = norm(b);
  error       = norm(r)/normb;
  flag        = 0;
  if (error < tol); return; end
  residual(1) = norm(r)/normb;
  k           = 1;

     while (k < maxit)
         s              = A*p;
         r01            = (r'*r);
         alpha          = (r'*r)/(p'*s);      % step length
         x              = x + alpha*p;        % approximate solution
         r              = r - alpha*s;        % residual
         beta           = (r'*r)/r01;         % improvement this step
         p              = r + beta*p;         % serarch direction
         residual(k)    = norm(r)/normb;
         n(k)           = k;
         error          = norm(r)/normb ;
         k              = k + 1;
         if (error < tol); flag = 1; break; end
     end
```

b) Preconditioners

> *"For most challenging practical problems arising from differential equations more challenging than our model problem, linear systems, $A \cdot x = b$ must be preconditioned or replaced by equivalent systems $M^{1} \cdot Ax = M^{1} \cdot b$, which somehow are easier to solve.* ([p. 266] Demmel 1997).

Heath ([p. 474], 2002) notes that the CG algorithm converges slowly if A is ill conditioned. In such cases convergence can be *substantially accelerated* by preconditioning, a technique which can be understand as multiplying A by an M^{-1}, whose inverse approximates that of A such that the product $A*M^{-1}$ is well conditioned. The choice of the M depends on the usual exchange between better convergence and increased cost per iteration associated with employing a preconditioner; the choice of such preconditioners is an active area of research (Heath 2002). Some common preconditioners are:

- Diagonal matrix with diagonal elements equal to those of A.
- Block diagonals.
- *Incomplete* factorizations. Compute approximate Cholesky factorization of $A \approx LL^{T}$, with little fill and force the nonzero entries of the lower triangular matrix, L, to be in the same positions of the nonzero elements in lower A. Thus $M \approx LL^{T}$.
- M is a polynomial in A.

- M^{-1} is an approximate inverse of A derived from an optimization algorithm designed to minimized the residual, $|| I - A*M^{-1}||$.

Heath observes that conjugate gradient algorithms are *rarely used* without some form of preconditioner. Since diagonal preconditioners require minimal amounts of work and storage, Heath recommends that their use is *always advisable*.

The conjugate gradient method is very efficient. For an n-by-n sized linear system, employing, **`poisson_model.m`**, the conjugate gradient algorithm **`cg_jch.m`**, listed below, requires *only* 100 and 202 iterations to reach a relative residual of 10^{-6} for an *n* of 31 and 63 respectively.

they are discussed in the appendix. Their convergence rates are slow. Hence they are used primarily as preconditioners to Krylov subspace methods (e.g. [p. 323] O'Leary 2009). Iterative methods usually converge faster using a preconditioner. A preconditioner maps the original linear system, A*x = b, onto a *different* system

$$\bar{A}*x = \bar{b},$$

which has the same solution, but which hopefully exhibits better convergence characteristics. A poorly chosen preconditioner or no preconditioner at all may result in a very slow rate or lack of convergence. Note that preconditioning involves a trade-off between the reduction in the number of iterations required for convergence and additional computational costs per iteration.

All the MATLAB solvers accept arguments M_1 and M_2 or $M = M_1 \cdot M_2$ and effectively solve the preconditioned system

$$(M_1^{-1} \bullet A \bullet M_2^{-1}) \bullet (M_2 \bullet x) = M_1^{-1} \bullet b \quad \text{or} \quad M^{-1} \bullet A \bullet x = M^{-1} \bullet b.$$

The aim is to choose M_1 and M_2 so $M^{-1}{}_1 \cdot A \cdot M^{-1}{}_2$ sometimes a difficult task that usually requires knowledge of the application from which the linear system came. The MATLAB functions **`ilu`**

and **ichol** compute incomplete factorizations that provide at least one of constructing preconditioners.

MATLAB possesses a variety of functions to perform matrix operations on sparse arrays. To access MATLAB help on sparse arrays follow

Help tab ⇒ Product Help ⇒ Functions by Category ⇒ Mathematics ⇒
Sparse Matrices ⇒ Linear Equations (Iterative Methods)

OR

Help tab ⇒ Demos ⇒ Sparse Matrices

MATLAB's iterative Krylov algorithms for solving sparse linear systems, A·x = b, break into two broad classes –

Symmetric Linear Systems:

- **minres** Minimum residual method:
  A must be symmetric but need *not* be positive definite.
  A should be large and sparse.
- **pcg** Preconditioned conjugate gradients method:
  A must be symmetric *and* positive definite.
  A should be large and sparse.
- **symmlq** Symmetric LQ method:
  A must be symmetric but need *not* be positive definite.

Non-symmetric Linear Systems:

- **bicg** Biconjugate gradients method:
  A must be square.
  A should be large and sparse.
- **bicgstab** Biconjugate gradients stabilized method:
  A must be square.
  A should be large and sparse.
- **bicgstabl** Biconjugate gradients stabilized (l) method:
  A must be square.
- **cgs** Conjugate gradients squared method:
  A must be square.
  A should be large and sparse.
- **gmres** Generalized minimum residual method (with restarts):
  A must be square.
  A should be large and sparse.
- **lsqr** LSQR method:
  A need *not* be square.
  A should be large and sparse.
- **qmr** Quasi-minimal residual method:
  A must be square.
  A should be large and sparse.
- **tfqmr** Transpose-free quasi-minimal residual method:
  A must be square.

These *eleven* built-in Krylov solvers attempt to solve the system of linear equations, A·x = b, where the coefficient matrix, *A*, has an n-by-n size and the corresponding vector, *b*, must have length n.

Note that the matrix *A*[1] can be a function handle, *A_fn*, such that **A_fn(x)** accepts as input the vector, *x*, as returns as output the matrix-vector product, **A*x**.

If a solver converges, a message is printed in the command window. If the solver fails to converge after the maximum number of iterations or halts, a warning message is printed displaying relevant relative residual value as well as the iteration number at which the algorithm terminated.

These solvers have consistent interfaces:

Input Arguments:

```
x = XXX(A, b);
x = XXX(A, b);
x = XXX(A, b, tol, maxit);
x = XXX(A, b, tol, maxit, M);
x = XXX(A, b, tol, maxit, M1, M2);
x = XXX(A, b, tol, maxit, M1, M2, x0);
```

tol	specifies the tolerance of the method. If it is [], the method uses a default value of 10^{-6}.
maxit	specifies the maximum number of iterations. If it is [], the method employs a default value of **min(n, 20)**.
M	represents a preconditioner[2,3] M, or M = $M_1 \cdot M_2$ and the algorithm solves $M^{-1}A \cdot x = M^{-1} \cdot b$. If M = [], no preconditioner is used. M can also be a function, **M_fn**[4], such that **M_fn(x)** returns **M\x**.
x0	specifies the initial guess. If x_0 = [], the method employs a default of an all-zero vector.

For **gmres** an *additional* input variable can be supplied:

```
x = gmres(A, b, restart, tol, maxit, M1, M2, x0);
```

restart	restarts **gmres** every **restart** *inner* iterations. The maximum number of outer iterations is **min(n/restart, 10)**. The maximum number of total iterations is **restart* min(n/restart, 10)**. If **restart** is **n** or [], **gmres** does not restart and the maximum number of total iterations is **min(n, 10)**.

[1] In using the **qmr, lsqr,** and **bicg** algorithms the function, **A_fn(x, 'notransp')**, must return **A*x** and **A_fn(x,'transp')** must return **A'*x**.

[2] The preconditioner, **M**, must be a symmetric positive definite matrix for **minres, pcg**, and **symmlq**.

[3] In **symmlq** and **minres** the preconditioner, **M**, effectively solves **inv(sqrt(M))*A*inv(sqrt(M))*y = inv(sqrt(M)*b** for **y** and then returns **x = inv(sqrt(M))*y.**

[4] In using the **qmr, lsqr,** and **bicg** algorithms the function, **M_fn(x, 'notransp'),** must return **M\x** and **M_fn(x,'transp')** must return **M'\x**.

maxit specifies the maximum number of *outer* iterations. The total number of iterations does not exceed **restart*maxit**. If **maxit** is [], **gmres** employs the default, **min(n/restart, 10)**. If **restart** is *n* or [], then the number of total iterations is **maxit**.

Output Arguments:

```
[x, flag]                          = XXX(A, b, …);
[x, flag, relres]                  = XXX(A, b, …);
[x, flag, relres, iter]            = XXX(A, b, …);
[x, flag, relres, iter, resvec]    = XXX(A, b, …);
```

flag outputs an exit flag:

0	converged to given tolerance **tol** with **maxit** iterations.
1	iterated **maxit** times but did not converge.
2	preconditioner **M** was ill conditioned.
3	method stagnated.
4	one of the calculated scalars during method became too small or too large to continue calculation.
5	preconditioner, M, is not symmetric, positive definite[5].

relres outputs relative residual norm, **relres = norm(b-A*x)/norm(b)**. If **flag = 0**, **relres <= tol**.

iter outputs iteration value at which x was calculated, where **0<= iter <= maxit.**

relres outputs a vector of the residual norms at each iteration including the initial value for **norm(b - A*x0)**.

For **minres** and **symmlq** an *additional* output variable:

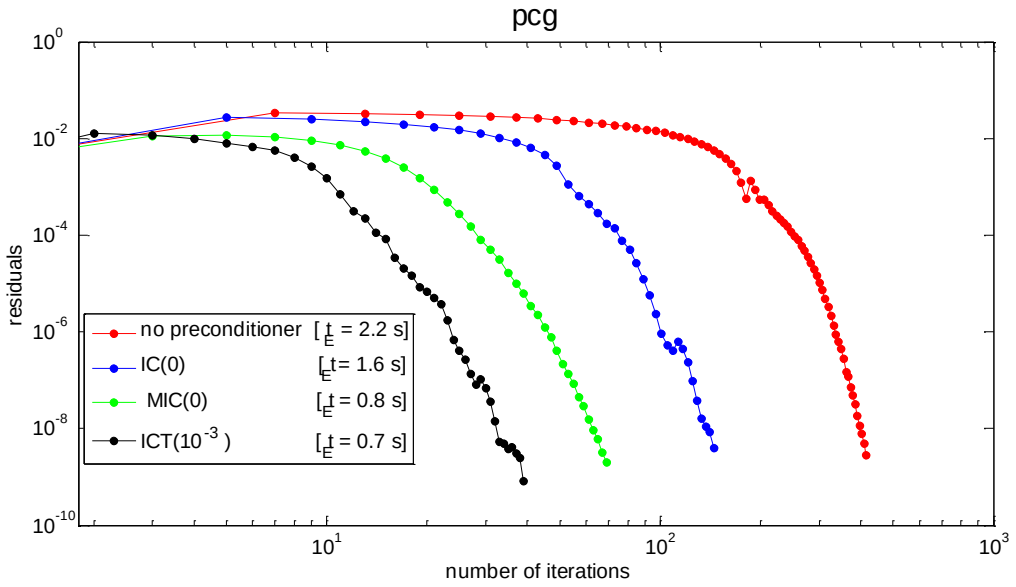
```
[x, flag, relres, iter, resvec, resveccg]        = ZZZ(A, b, …);
```

resveccg outputs a vector of the *conjugate gradients* residual norms at each iteration.

```
>> n = 256; [A, b] = poisson_model(n);
>> tol = 5e-07; maxit = 1000;
>> tic;  [x, flag, relres, iter0, residual0] = pcg(A, b, tol, maxit); toc
Elapsed time is 2.151028 seconds.
>> disp([flag iter0, relres])
         0    419   4.8868e-007

>> tic; L = ichol(A);
  [x, flag, relres, iter1, residual1] = pcg(A, b, tol, maxit, L, L'); toc
Elapsed time is 1.652525 seconds.
>> disp([flag iter1, relres])
```

[5] Error flag 5 applies only to **symmlq**.

```
         0    147    4.9358e-007

>> tic; opts.type = 'nofill'; opts.michol = 'on'; L = ichol(A, opts);
   [x, flag, relres, iter, residual2] = pcg(A, b, tol, maxit, L, L'); toc
Elapsed time is 0.767630 seconds.
>> disp([flag iter2, relres])
         0    69     3.8041e-007

>> tic; opts.type = 'ict'; opts.droptol = 1.0e-03; L = ichol(A, opts);
   [x, flag, relres, iter, residual3] = pcg(A, b, tol, maxit, L, L'); toc
Elapsed time is 0.677365 seconds.
>>  disp([flag iter3, relres])
          0    38    2.1135e-007
```



Appendix – Partial Differential Equation, Poisson Equation

A numerical solution to a two-dimensional elliptic PDE, the Poisson equation, is developed here. The form of this PDE is the following:

$$U_{xx} + U_{yy} = f(x, y) \quad \text{in } \Omega(x, y),$$
$$U = g_D(x, y) \text{ in } \partial\Omega,$$

where f(x,y) defines the source term, $\Omega(x, y)$ represents the spatial domain of the problem, and $\partial\Omega(x, y)$ defines the region's boundary. The PDE is the following -

$$\frac{\partial^2 U}{\partial^2 x} + \frac{\partial^2 U}{\partial^2 y} = 1.0, \quad 0 \le x \le 1, \ 0 \le y \le 1,$$
$$U(0,y) = 0.0,$$
$$U(1,y) = 0.0,$$
$$U(x,1) = 0.0,$$
$$U(x,0) = 0.0.$$

To do this, we will discretize the elliptic PDE in space:

- Replace the continuous domain of the PDE by a discrete mesh of points.
- Approximate the derivatives by second-order finite differences.
- Evaluate the solution at previously selected points in space.

To perform this task, one creates a uniform square grid in x-y Cartesian space and identifies grid spatial coordinates as

$$(x_i, y_j) \equiv (ih, jh), \quad i, j = 0{:}n + 1,$$

where the relevant spatial coordinates of the *boundary* region are identified by the indices, 0, (N +1)h. Thus the unknown solution function at grid points, $U(x_i, y_j)$ is represented as $U_{i,j}$. The same step size is used in both the x and y directions.

A finite difference approximation is derived via a Taylor expansion. Consider a simple smooth function, g(x), which depends (for the moment) only on x, Perform Taylor expansions around x employing h as a step size –

NOTE: the symbol $O(h^\alpha)$ means that the terms discarded are proportional to h^α .

$$g(x - h) = g - hg_x + \frac{h^2}{2}g_{xx} - \frac{h^3}{6}g_{xxx} + \frac{h^4}{24}g_{xxxx} + O(h^5) \quad \cdots \text{ [I]}$$
$$g(x + h) = g + hg_x + \frac{h^2}{2}g_{xx} + \frac{h^3}{6}g_{xxx} + \frac{h^4}{24}g_{xxxx} + O(h^5) \quad \cdots \text{ [II]}$$

Adding equations I and II one finds that

$$g_{xx}(x) \cong \frac{g(x+h) - 2g(x) + g(x-h)}{h^2} - \frac{h^2}{12} g_{xxxx} \simeq \frac{g(x+h) - 2g(x) + g(x-h)}{h^2} + O(h^2).$$

Now if one assumes that g is a smooth function of both x and y. Thus the second x derivative on a uniform Cartesian grid, where $g_{i,j} = g(x_j, y_i)$ $gi_{,j} \rightarrow g$ at ith row and jth column, where x = jh and y = ih, and $x \pm h$ is represented by $(j \pm 1)h$, then g_{xx} becomes

$$g_{xx}(i, j) = \frac{g_{i,j+1} - 2g_{i,j} + g_{i,j-1}}{h_x^2},$$

and, similarly, the second y derivative of g(x, y) becomes

$$g_{yy}(i,j) = \frac{g_{i+1,j} - 2g_{i,j} + g_{i-1,j}}{h_y^2}.$$

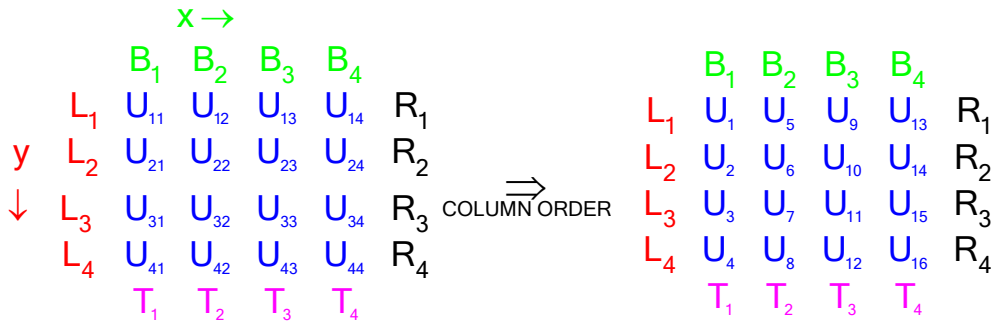
Here we express the quantity of interest as U(x, y), and not as g(x, y),

,

$$\frac{U_{i,j+1} - 2U_{i,j} + U_{i,j-1}}{h_x^2} + \frac{U_{i+1,j} - 2U_{i,j} + U_{i-1,j}}{h_y^2} = f_{ij}, \Rightarrow \quad h_x = h_y$$

$$4U_{ij} - U_{i+1,j} - U_{i-1,j} - U_{i,j+1} - U_{i,j-1} = -h^2 f_{ij}.$$

To begin the discussion of the Poisson equation one uses a 4-by-4 grid–



Here the symbols L, B, R, and T represent respectively, left, bottom, right and top. In MATLAB columns increase from left to right but rows increase from top to bottom. The "bottom" vector is above the first row and the "top" vector appears below the last row.

To create the linear system the n-by-n unknown solution *matrix* $U_{i,j}$ is represented by a *vector* of size n^2-by-1, constructed by concatenating all the columns consecutively starting with the first column. Creating a solution vector in row element order, top to bottom, or bottom to top, or alternatively in column major order, right to left or left to right is *completely* arbitrary ([p. 116], Iserles 1996; [p. 272], Demmel 1997). Since f(x, y) = 1, T(x,y) = 0, B(x, y) = 0, L(x, y) = 0,and R(x, y) = 0, the system of linear equations becomes the following –

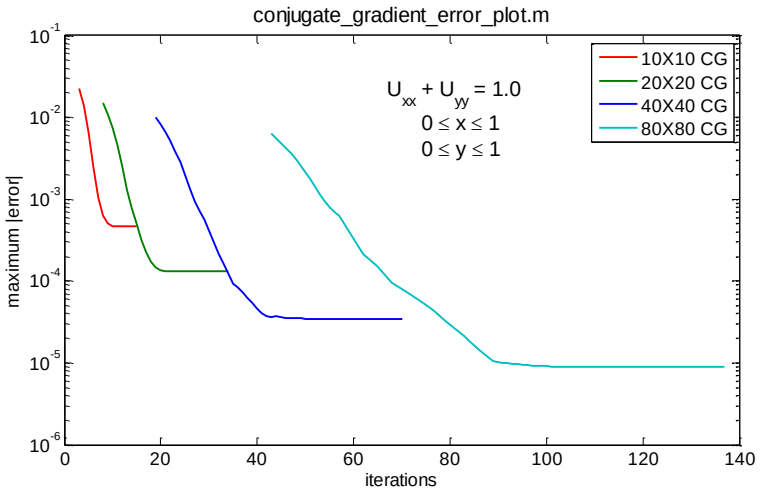
$$A*U = c.$$

In component form, the matrix A and vectors U and c have the following form, where by convention the main A diagonal is defined to be positive -

Using a 10-by-10 grid requires a solution of 10^4-by-10^4 linear system with only 460 nonzero terms.

Conjugate Gradient Method

In the Figure below, **`conjugate_gradient_example.m,`** we apply the MATLAB built-in conjugate-gradient algorithm, **`pcg,`** to a symmetric positive-definite Poisson problem. The number of calculated iterations was unaffected when either a diagonal matrix or an incomplete Cholesky factorizations were used as preconditioners. Such behavior is expected since even the $(80X80)^2$ matrix is relatively well conditioned with a condition number of only 861. As earlier we plot the maximum of the difference between the model calculation and the actual U(x, y) as a function of the number of iterations. At large iteration numbers, the difference between the exact solution and the model solution is the discretization error. As can be seen from the comparison with the corresponding error plots created by the Jacobi and Gauss-Seidel algorithms, the present conjugate-gradient algorithm is significantly more efficient than those iterative methods.



As noted above by Heath the conjugate gradient method is applicable only to symmetric, positive-definite linear systems. If the matrix A is either indefinite or non-symmetric, the conjugate gradient method is inapplicable both in theory and in practice. However, the conjugate-gradient method have been applied to symmetric, *indefinite* linear systems in the form of the algorithm, *SYMMLQ*; also, a number of algorithms, based on the concept of Krylov subspaces, such as *GMRES, QMR, CGS, BiCG, Bi-STAB*, have been developed to solve iteratively non-symmetric linear systems (Heath 2002). However, these solvers for non-symmetric linear systems are inclined to be less robust or require more work than the conjugate-gradient method (Heath 2002). Please note that these alternative iterative solvers exist as built-in functions in MATLAB: **`symmlq, gmres, qmr, cgs, bicg,`** and **`bicgstab.`**

6 REFERENCES

Barrett, R. et al. 1993, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods* (SIAM: Philadelphia).

Beers, K. J. 2007, *Numerical Methods in Chemical Engineering, Applications in MATLAB®*, (Cambridge: Cambridge UK).

Dahlquist, G. and Björck A. 2008, *Numerical Methods in Scientific Computing, Volume 2* (SIAM: Philadelphia, PA).

Davis, T. A. 2006, *Direct Methods for Sparse Linear Systems* (SIAM: Philadelphia).

Demmel, J. W. 1997, *Applied Numerical Linear Algebra* (SIAM: Philadelphia).

Dongarra, J., & Sullivan, F 2000, *Computing in Science & Engineering*, **2**, 22.

Elman, H., Silvester, D., & Wathen A. 2005 *Finite Elements and Fast Iterative Solvers* (Oxford University Press; Oxford UK).

Heath, M. T. 2002, *Scientific Computing: An Introductory Survey, 2nd Ed*. (McGraw-Hill: NY, NY).

Iserles, A., 2009, *A First Course in the Numerical Analysis of Differential Equations*, 2nd Ed. (Cambridge University Press: Cambridge, UK).

LeVeque, R. J. 2007, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems* (SIAM: Philadelphia).

O'Leary, D. P. 2009, *Scientific Computing with Case Studies* (SIAM: Philadelphia).

Saad. Y. 2003, *Iterative Methods for Sparse Linear Systems* (SIAM: Philadelphia).

Trefethen, L. N. and Bau, D. 1997, *Numerical Linear Algebra* (SIAM: Philadelphia).

Van der Vorst, H. A. 2000, *Computing in Science & Engineering*, **2**, 32.