Chapter 17 C-implementation

PAP' = LDL'

Speaker: Lung-Sheng Chien

Reference: [1] H. M. Deitel, P. J. Deitel, C++ How to Program 5-th edition

OutLine

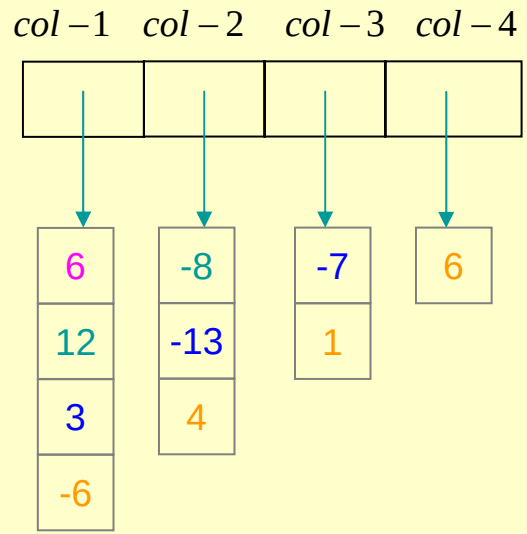
- Data structure of lower triangle matrix
- Function overloading in C++
- Implementation of *PAP' =LDL'*

row-major versus col-major

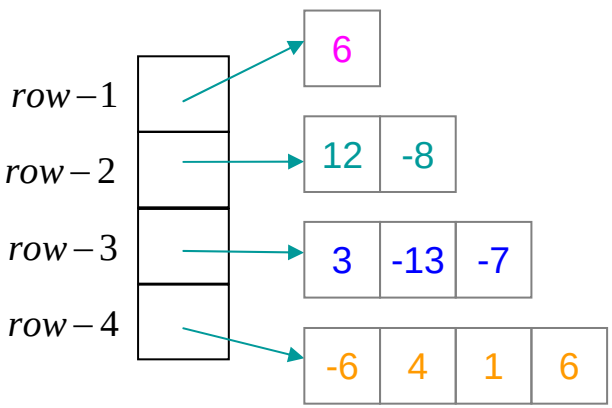
Logical index : 2D

$$A = \begin{pmatrix} 6 & 12 & 3 & -6 \\ 12 & -8 & -13 & 4 \\ 3 & -13 & -7 & 1 \\ -6 & 4 & 1 & 6 \end{pmatrix} = A^T$$

col-major based



row-major based



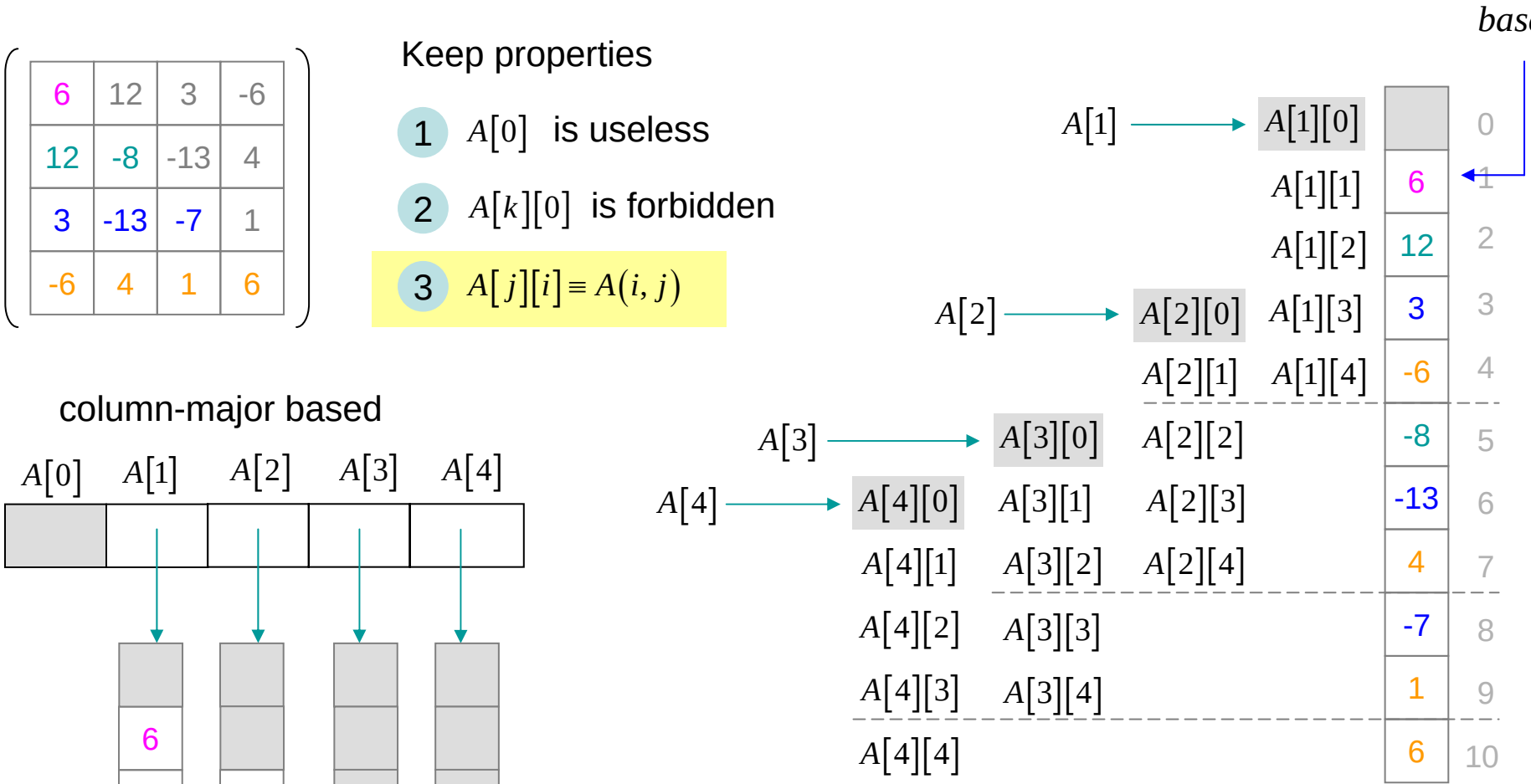
We choose col-major representation

column-major: method 1 (square-matrix based)

6	12	3	-6
12	-8	-13	4
3	-13	-7	1
-6	4	1	6

Keep properties

1. $A[0]$ is useless
2. $A[k][0]$ is forbidden
3. $A[j][i] \equiv A(i,j)$



column-major based

4. $A[j][i]$ is valid only for $i \geq j$ $\left(A[2][1] = -6 = A(4,1)\ ?\right)$
5. $A[1] = base - 1,\ A[2] = A[1] + 3,\ A[3] = A[2] + 2,\ A[4] = A[3] + 1$

column-major: method 2 (triangle-matrix based)

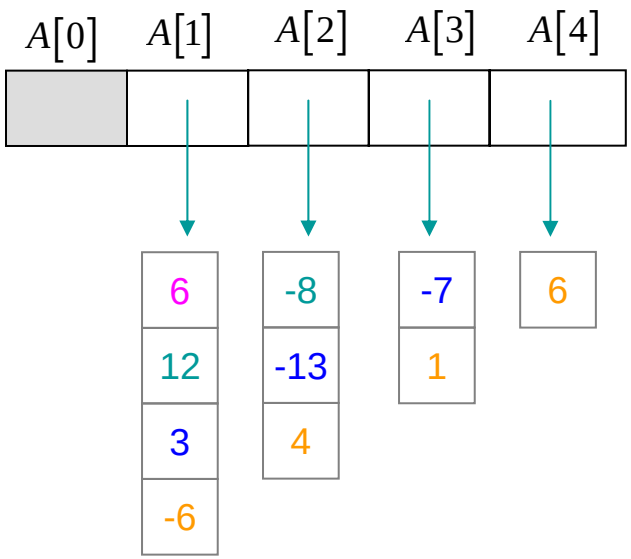
6	12	3	-6
12	-8	-13	4
3	-13	-7	1
-6	4	1	6

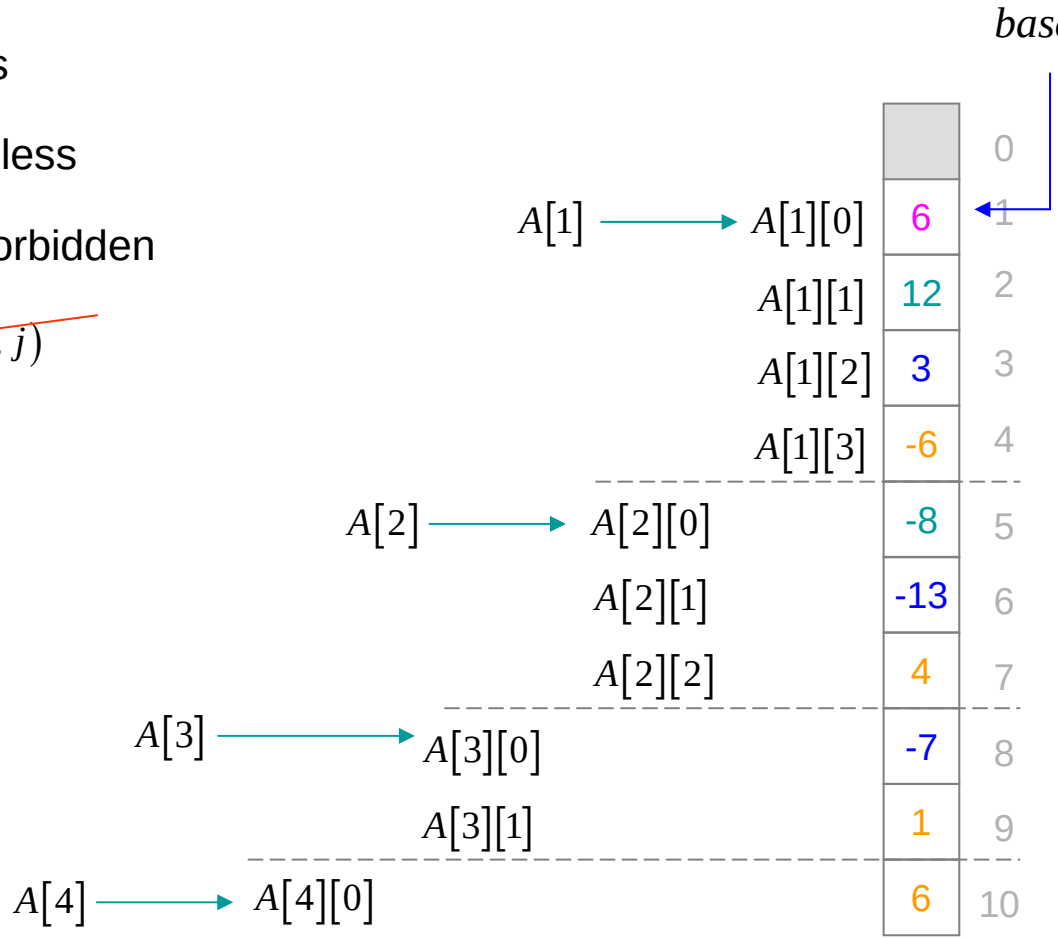
Keep properties

1. $A[0]$ is useless
2. $A[k][0]$ is forbidden
3. ~~$A[j][i] = A(i, j)$~~
4. $A(i, j) = A[j][i-j]$ *for* $i \geq j$

Index rule is more complicated

column-major based





Data structure of matrix and its method

lowerTriangleMatrix.h

```
#ifndef  LOWERTRIANGLEMATRIX_H
#define  LOWERTRIANGLEMATRIX_H

#include "global.h"  // declaration of doublereal
#include "matrix.h"

#include <stdio.h>
#include <iostream> // type of I/O
#include <iomanip>  // manipulate I/O
using namespace std ;

typedef struct lowerTriangleMatrix
{
    integer  m ;
    integer  n ;
    orderVar   sel ; // col-major or row-major
    int  isSym ; // isSym > 0 : A is symmetry
    doublereal **A ;
} lowerTriangleMatrix ;

typedef  lowerTriangleMatrix*   lowerTriangleMatrixHandler ;

// lowerTriangleMatrix constructor
void  zeros( lowerTriangleMatrixHandler* Ah_ptr, integer m, integer n, orderVar sel, int isSym ) ;

// destructor
void dealloc( lowerTriangleMatrixHandler Ah ) ;

// show content of matrix in standard form
void disp( lowerTriangleMatrixHandler Ah,  ostream& out ) ;

doublereal  norm( lowerTriangleMatrixHandler Ah, matrix_keyword p ) ;

// y = A*x
void matvec( lowerTriangleMatrixHandler Ah, matrixHandler xh, matrixHandler yh) ;

// copy A to A_shadow
void duplicate( lowerTriangleMatrixHandler Ah, lowerTriangleMatrixHandler Ah_shadow ) ;

#endif // LOWERTRIANGLEMATRIX_H
```

method 1

A is symmetric, but we only store lower triangle part

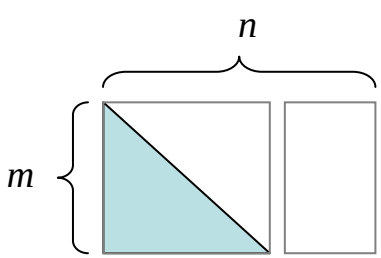
The same name as those in matrix.h

Constructor [1]

lowerTriangleMatrix.cpp

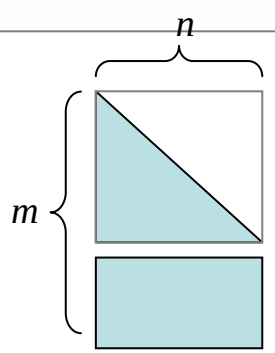
```
// lowerTriangleMatrix constructor
void  zeros( lowerTriangleMatrixHandler* Ah_ptr, integer m, integer n, orderVar sel, int isSym )
{
    integer j ;
    doublereal **A ;
    doublereal *memA ; // contiguous memory block of A
    integer size ; // number of entries in matrix A

    assert( Ah_ptr ) ;
    assert( 0 < m ) ;  assert( 0 < n ) ;

// alolocate an empty handler
    *Ah_ptr = (lowerTriangleMatrixHandler) malloc( sizeof(lowerTriangleMatrix) ) ;
    assert( *Ah_ptr ) ;

    if ( COL_MAJOR == sel ){
// A[0] is useless, A[j] means pointer of column j
        A = (doublereal **) malloc( sizeof(doublereal *)*(n+1) ) ;
        assert( A ) ;

        if ( m < n ){
1           size = m*(m+1) ;
            size = size >> 1 ; // size = m(m+1)/2
        }else{
2           size = n*(n+1) ;
            size = size >> 1 ;
            size += n*(m-n) ;  // size = n(n-1)/2 + n(m-n)
        }
```

1



$$size = \frac{m(m+1)}{2}$$

2



$$size = \frac{n(n+1)}{2} + n(m-n)$$

Constructor [2]

lowerTriangleMatrix.cpp

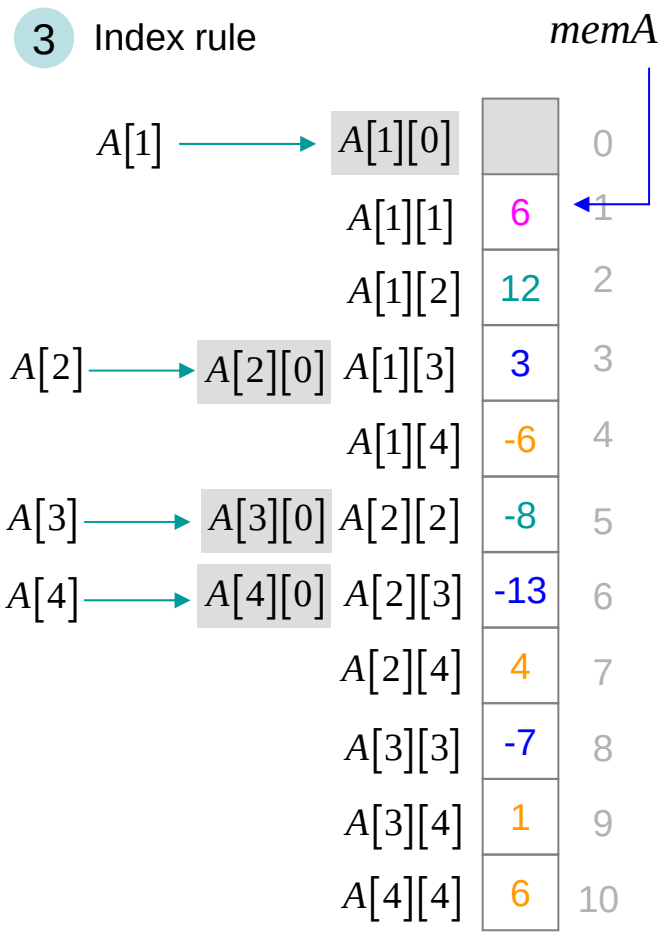
```
#ifdef  HIGH_PRECISION_PACKAGE
        memA = new doublereal [size] ;
#else
        memA = (doublereal*) malloc( sizeof(doublereal)*size ) ;
#endif
        assert( memA ) ;
        for ( j=0 ; j < size ; j++){
            memA[j] = 0.0 ;  // reset matrix A as zero matrix
        }
        A[1] = memA - 1 ; // A[0] is useless
        for ( j=1 ; j < n ; j++){
    3       A[j+1] = (doublereal*)A[j] + m-j ;
        }
    }else{
        printf("Error: we don't support row-major so far\n");
        exit(1) ;
    }
// set parameter of a matrix
    (*Ah_ptr)->m = m  ;  (*Ah_ptr)->n = n ; (*Ah_ptr)->sel = sel ; (*Ah_ptr)->A = A ;
    (*Ah_ptr)->isSym = isSym ;
    if ( 0 != isSym ){
        if ( m != n ){
    4       cerr << "A is symmetry, then A must be square matrix" << endl ;
            exit(1) ;
        }
    }

}
```

4 When matrix A is symmetric, it must be square and we store lower triangle part of A,

Difference from full matrix is 3 4

6	12	3	-6
12	-8	-13	4
3	-13	-7	1
-6	4	1	6

3 Index rule



destructor

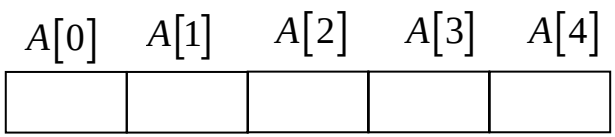
lowerTriangleMatrix.cpp

```
void dealloc( lowerTriangleMatrixHandler Ah )
{
// step 1: dealloc contiguous memory block
#ifdef  HIGH_PRECISION_PACKAGE
    delete [] ( (doublereal*)(Ah->A[1]) + 1 ) ;
#else
    free( (doublereal*)(Ah->A[1]) + 1 ) ;
#endif

// step 2: dealloc column-index array
    free( Ah->A ) ;

// step 3: dealloc matrix handler
    free( Ah ) ;
}
```

1

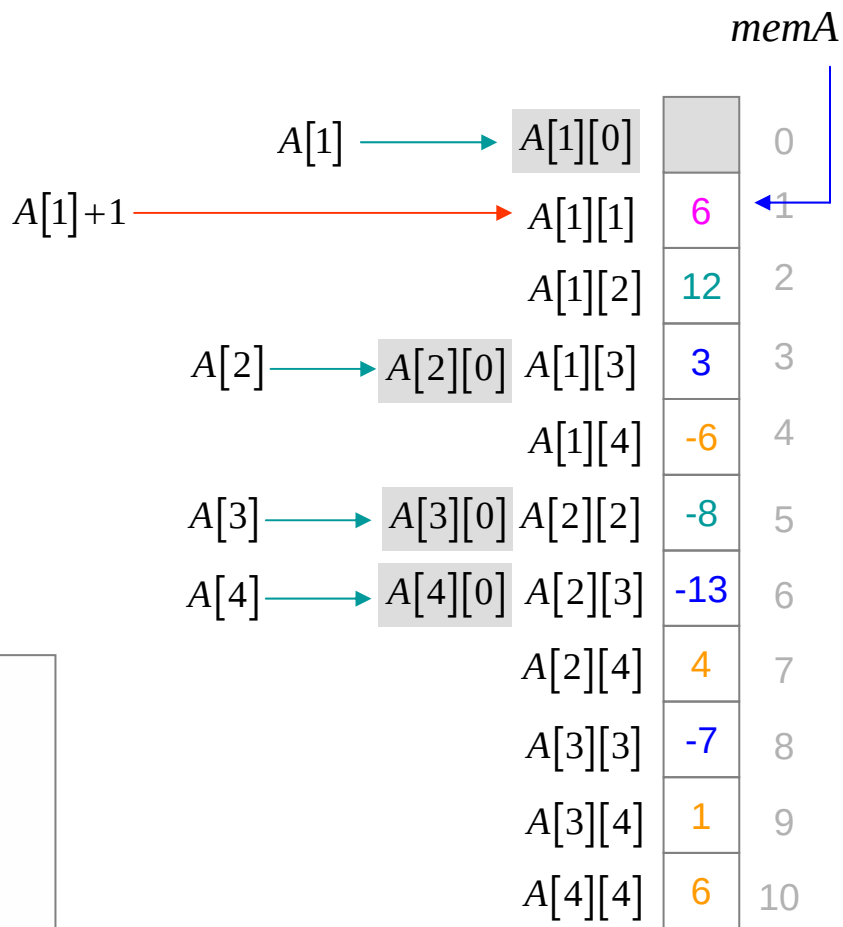
Pointer arithmetic, remember to do casting

2 free pointer array



3 free lowerTriangleMatrix handler

```
typedef struct lowerTriangleMatrix
{
    integer  m ;
    integer  n ;
    orderVar   sel ; // col-major or row-major
    int  isSym ; // isSym > 0 : A is symmetry
    doublereal **A ;
} lowerTriangleMatrix ;

typedef  lowerTriangleMatrix*   lowerTriangleMatrixHandler ;
```



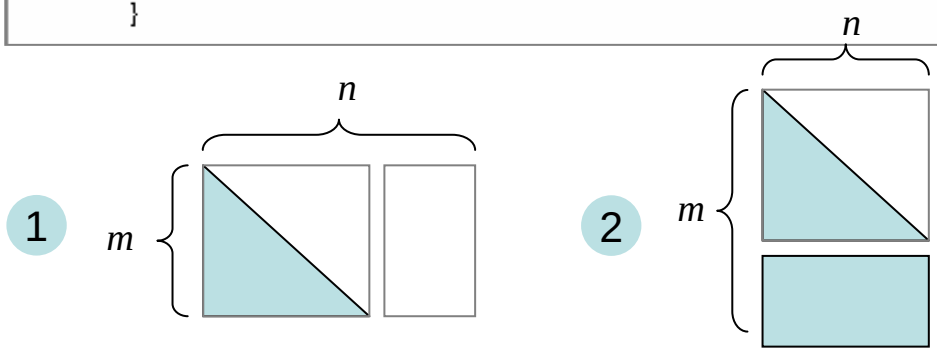
Input/Output

lowerTriangleMatrix.cpp

```
// show content of matrix in standard form
void disp( lowerTriangleMatrixHandler Ah,  ostream& out )
{
    int i, j ;
    doublereal **A ;
    integer  m, n ;

    assert( Ah ) ;
    assert( COL_MAJOR == Ah->sel ) ;

    out.precision(4) ;
    out << std::fixed ;

    if ( Ah->isSym ){
        out << "dimension of symmetric matrix is (" << Ah->m  << ","
            << Ah->n << ") with col-major" << endl ;
    }else{
        out << "dimension of lower triangle matrix is (" << Ah->m  << ","
            << Ah->n << ") with col-major" << endl ;
    }

    A = Ah->A ;  m = Ah->m ; n = Ah->n ;
    if ( m < n ){
        for( i=1 ; i <= m ; i++ ){
            for( j=1 ; j <= i ; j++ ){
                out << setw(10) << A[j][i]  ;
            }
            out << endl ;
        }
    }else{ // m >= n
        for( i=1 ; i <= n ; i++ ){
            for( j=1 ; j <= i ; j++ ){
                out << setw(10) << A[j][i]  ;
            }
            out << endl ;
        }
        for ( i=n+1 ; i <= m ; i++ ){
            for( j=1 ; j <= n ; j++ ){
                out << setw(10) << A[j][i]  ;
            }
            out << endl ;
        }
    }// if ( m < n )
    out.precision(16) ;
    out << std::scientific ;
}
```

1

2



No matter what A is symmetric or not, we only report lower triangle part

Simple driver

main.cpp

```
#include "global.h"
#include <stdio.h>
#include "lowerTriangleMatrix.h"

void  test_matrix( void ) ;

int main( int argc, char* argv[])
{
#ifdef  HIGH_PRECISION_PACKAGE
    unsigned int old_cw;
    fpu_fix_start(&old_cw);
#endif

#ifdef DO_ARPREC
    mp::mp_init(ARPREC_NDIGITS);
#endif

// ---------- main code ----------

    test_matrix() ;

// ---------- end main code ----------

#ifdef DO_ARPREC
    mp::mp_finalize();
#endif

#ifdef  HIGH_PRECISION_PACKAGE
    fpu_fix_end(&old_cw);
#endif
    return 0 ;
}
```

```
void  test_matrix( void )
{
    integer m = 4 ;
    integer n = 4 ;
    lowerTriangleMatrixHandler Ah ;
    doublereal **A ;

    zeros( &Ah, m, n, COL_MAJOR, 1 ) ;
    A  = Ah->A ;

    A[1][1] =  6. ; A[1][2] = 12. ; A[1][3] =  3.  ; A[1][4] = -6.  ;
                    A[2][2] = -8. ; A[2][3] = -13. ; A[2][4] =  4.  ;
                                    A[3][3] = -7.  ; A[3][4] =  1.  ;
                                                     A[4][4] =  6.  ;

    disp( Ah, cout ) ;

    dealloc( Ah ) ;
}
```

```
C:\WINDOWS\system32\cmd.exe
dimension of symmetric matrix is (4,4) with col-major
    6.0000
   12.0000   -8.0000
    3.0000  -13.0000   -7.0000
   -6.0000    4.0000    1.0000    6.0000
請按任意鍵繼續 . . .
```

Advantage of method 1: index rule is the same as that of full matrix, one only notices

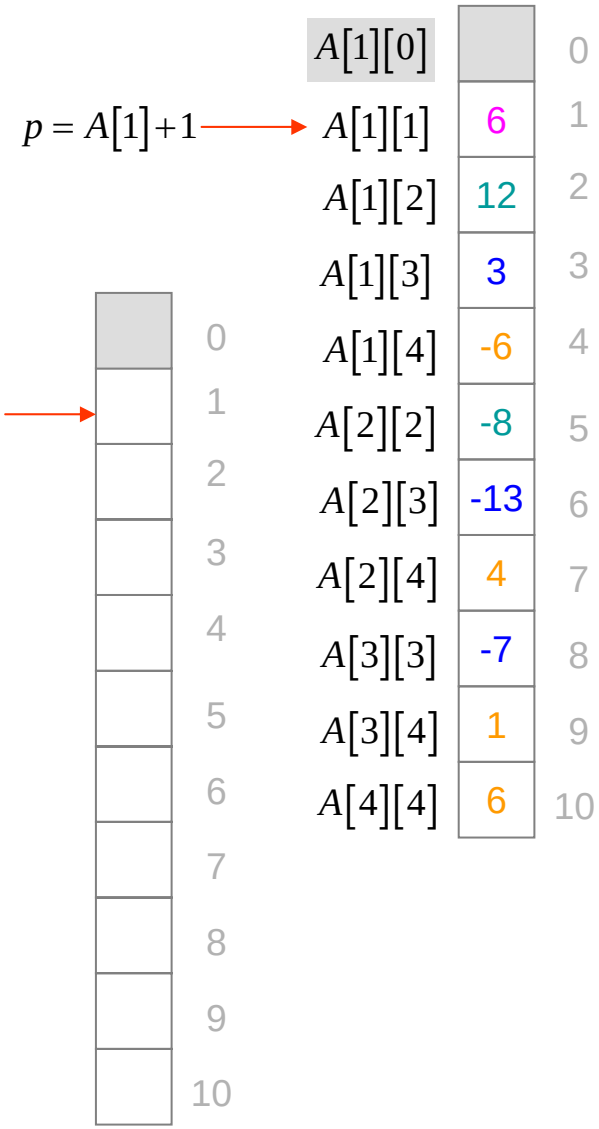
$A[j][i]$ is valid only for $i \geq j$

Question: how can we verify $A[j][i]$ is valid only for $i \geq j$ automatically?

duplicate of a lower triangle matrix

lowerTriangleMatrix.cpp

```
// copy A to A_shadow
void duplicate( lowerTriangleMatrixHandler Ah, lowerTriangleMatrixHandler Ah_shadow )
{
    integer  m, n, size ;
    integer  i, j ;
    doublereal **A ;
    doublereal **A_shadow ;
    doublereal *p ;
    doublereal *q ;

    assert( Ah ) ;  assert( Ah_shadow ) ;
    assert( COL_MAJOR == Ah->sel ) ;
    assert( COL_MAJOR == Ah_shadow->sel ) ;

    m = Ah->m ; n = Ah->n ;
// check Ah_shadow has the same configuration as Ah
    assert( m == Ah_shadow->m ) ;
    assert( n == Ah_shadow->n ) ;
    Ah_shadow->isSym = Ah->isSym ;

    if ( m < n ){
        size = m*(m+1) ;
        size = size >> 1 ; // size = m(m+1)/2
    }else{
        size = n*(n+1) ;
        size = size >> 1 ;
        size += n*(m-n) ;  // size = n(n-1)/2 + n(m-n)
    }

    A = Ah->A ;   A_shadow = Ah_shadow->A ;
    p = (doublereal*) A[1] + 1 ;
    q = (doublereal*) A_shadow[1] + 1 ;

    for ( i = 0 ; i < size ; i++ ){
        *q++ = *p++ ;
    }
}
```



Question: This copy routine is different from that in matrix.cpp, which one is better?

Matrix norm [1]

lowerTriangleMatrix.cpp

```
doublereal  norm_sym( lowerTriangleMatrixHandler Ah, matrix_keyword p )
{
    integer m, n ;
    doublereal  sum ;
    doublereal  sumMax ;
    doublereal  **A ;
    int i, j ;

    m = Ah->m ; n = Ah->n ;
    A = Ah->A ;
    if ( INF_NORM == p ){
        sumMax = 0.0 ;
        for ( i = 1 ; i <= m ; i++ ){
            sum = 0.0 ;
            for ( j = 1 ; j <= i ; j++){
                sum += fabs( A[j][i] ) ;
            }
            for( j = i+1 ; j <= n ; j++ ){
                sum += fabs( A[i][j] ) ;
            }
            if ( sum > sumMax ){
                sumMax = sum ;
            }
        }// for each row
    }else if ( ONE_NORM == p ){
        sumMax = 0.0 ;
        for ( j = 1 ; j <= n ; j++ ){
            sum = 0.0 ;
            for ( i = 1 ; i < j ; i++){
                sum += fabs( A[i][j] ) ; // A[i][j] = A[j][i]
            }
            for ( i = j ; i <= m ; i++){
                sum += fabs( A[j][i] ) ;
            }
            if ( sum > sumMax ){
                sumMax = sum ;
            }
        }// for each col
    }else{
        printf("TWO_NORM is NOT implement\n");
        exit(1) ;
    }
    return sumMax ;
}
```

lowerTriangleMatrix.cpp

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

#include "lowerTriangleMatrix.h"

// private supplement function
doublereal  norm_sym( lowerTriangleMatrixHandler Ah, matrix_keyword p ) ;
doublereal  norm_nonsym( lowerTriangleMatrixHandler Ah, matrix_keyword p ) ;

// implement p = 1, 2 and inf
doublereal  norm( lowerTriangleMatrixHandler Ah, matrix_keyword p )
{
    assert( Ah ) ;
    assert( COL_MAJOR == Ah->sel ) ;

    doublereal  val ;
    if ( Ah->isSym ){
        val = norm_sym( Ah, p ) ;
    }else{
        val = norm_nonsym( Ah, p ) ;
    }
    return val ;
}
```

"norm_sym" and "norm_nonsym" are private, means that user don't need to know them and we declare them in lowerTriangleMatrix.cpp, NOT in lowerTriangleMatrix.h which user looks at

Matrix norm [2]

lowerTriangleMatrix.cpp

```
doublereal  norm_nonsym( lowerTriangleMatrixHandler Ah, matrix_keyword p )
{
    integer m, n ;
    doublereal  sum ;
    doublereal  sumMax ;
    doublereal  **A ;
    int i, j ;

    m = Ah->m ; n = Ah->n ;
    A = Ah->A ;
    if ( INF_NORM == p ){
        sumMax = 0.0 ;
        for ( i = 1 ; i <= m ; i++ ){
            sum = 0.0 ;
            for ( j = 1 ; j <= i ; j++){
                sum += fabs( A[j][i] ) ;
            }
            if ( sum > sumMax ){
                sumMax = sum ;
            }
        }// for each row

    }else if ( ONE_NORM == p ){
        sumMax = 0.0 ;
        for ( j = 1 ; j <= n ; j++ ){
            sum = 0.0 ;
            for ( i = j ; i <= m ; i++){
                sum += fabs( A[j][i] ) ;
            }
            if ( sum > sumMax ){
                sumMax = sum ;
            }
        }// for each col
    }else{
        printf("TWO_NORM is NOT implement\n");
        exit(1) ;
    }
    return sumMax ;
}
```

symmetric : norm_sym

6	12	3	-6
12	-8	-13	4
3	-13	-7	1
-6	4	1	6

nonsymmetric : norm_nonsym

6			
12	-8		
3	-13	-7	
-6	4	1	6

$y = Ax$

only A is lower trianlge matrix (either symmetric or non-symmetric), x and y are full matrices.

```
// y = A*x
void matvec( lowerTriangleMatrixHandler Ah, matrixHandler xh,
             matrixHandler yh)
{
    integer  m, n, s ;

    assert( Ah ) ;  assert( xh ) ;  assert( yh ) ;
    assert( COL_MAJOR == Ah->sel ) ;
    assert( COL_MAJOR == xh->sel ) ;
    assert( COL_MAJOR == yh->sel ) ;

    m = Ah->m ; n = Ah->n ;  s = xh->n ;

    assert(n == xh->m) ;
    assert(m == yh->m) ; assert(s == yh->n) ;

    if ( Ah->isSym ){
        matvec_sym( Ah, xh, yh) ;
    }else{
        matvec_nonsym( Ah, xh, yh) ;
    }
}
```

```
void matvec_nonsym( lowerTriangleMatrixHandler Ah, matrixHandler xh,
                    matrixHandler yh)
{
    integer  m, n, s ;
    integer  i, j, k ;
    doublereal **A ;
    doublereal **x ;
    doublereal **y ;

    m = Ah->m ; n = Ah->n ; s = xh->n ;
    A = Ah->A ; x = xh->A ; y = yh->A ;
// y = x(1)*A(:,1) + sum_{j}( x(j)*A(:,j) )
    for ( k=1 ; k <= s ; k++){
        for ( i=1 ; i <= m ; i++ ){
            y[k][i] = A[1][i] * x[k][1] ;
        }
        for ( j=2 ; j <= n ; j++){
// A is lower triangle part of A
            for ( i=j ; i <= m ; i++ ){
                y[k][i] += A[j][i] * x[k][j] ;
            }
        }// for each col-j
    }// for each vector x(:,k)
}
```

```
void matvec_sym( lowerTriangleMatrixHandler Ah, matrixHandler xh,
                 matrixHandler yh)
{
    integer  m, n, s ;
    integer  i, j, k ;
    doublereal **A ;
    doublereal **x ;
    doublereal **y ;

    m = Ah->m ; n = Ah->n ; s = xh->n ;
    A = Ah->A ; x = xh->A ; y = yh->A ;
// y = x(1)*A(:,1) + sum_{j}( x(j)*A(:,j) )
    for ( k=1 ; k <= s ; k++){
        for ( i=1 ; i <= m ; i++ ){
            y[k][i] = A[1][i] * x[k][1] ;
        }
        for ( j=2 ; j <= n ; j++){
            for ( i=1 ; i < j ; i++){
// i < j, A(i,j) = A(j,i)
                y[k][i] += A[i][j] * x[k][j] ;
            }
            for ( i=j ; i <= m ; i++ ){
                y[k][i] += A[j][i] * x[k][j] ;
            }
        }// for each col-j
    }// for each vector x(:,k)
}
```

OutLine

- Data structure of lower triangle matrix
- Function overloading in C++
- Implementation of *PAP' =LDL'*

Function overloading in C++

- C++ enables several functions fo the same name to be defined, as long as these functions have different sets of parameter
  (1) parameter type
  (2) number of parameters
  (3) order of parameter type

- when an overloaded function is called, the C++ compiler selects the proper function by examining (1) number (2) types (3) order of the arguments in the call.

- the only one function which cannot be overloaded is "main" since it is the entry point.

- Advantage: overloading functions that perform closely related tasks can make programs more readable

Function overloading of matrix data type

1. constructor

```
void zeros( matrixHandler * , integer m, integer n, orderVar sel ) ;
void zeros( int_matrixHandler * , integer m, integer n, orderVar sel ) ;
void zeros( lowerTriangleMatrixHandler* , integer m, integer n, orderVar sel, int isSym ) ;
```

2. destructor

```
void dealloc( matrixHandler Ah ) ;
void dealloc( int_matrixHandler Ah ) ;
void dealloc( lowerTriangleMatrixHandler Ah ) ;
```

3. I/O

```
void disp( matrixHandler Ah    , ostream& out ) ;
void disp( int_matrixHandler Ah, ostream& out ) ;
void disp( lowerTriangleMatrixHandler Ah,  ostream& out ) ;
```

4. metric

```
doublereal  norm( matrixHandler Ah, matrix_keyword p ) ;
doublereal  norm( lowerTriangleMatrixHandler Ah, matrix_keyword p ) ;
```

•
•
•

OutLine

- Data structure of lower triangle matrix
- Function overloading in C++
- Implementation of *PAP' =LDL'*

Algorithm (PAP' = LDL', partial pivot) [1]

Given symmetric indefinite matrix $A \in R^{n \times n}$, construct initial lower triangle matrix $L = I$

use permutation vector P to record permutation matrix $P^{(k)}$

let $A^{(1)} := A$, $L^{(0)} = I$, $P^{(0)} = (1, 2, 3, \cdots, n)$ and $pivot = zero(n)$, $\alpha = \frac{1+\sqrt{17}}{8} \approx 0.6404$

verification

```
int  bunch_kaufman( lowerTriangleMatrixHandler Ah, int_matrixHandler Ph,
                    int_matrixHandler pivoth, doublereal alpha )
{
    .
    .
    .
    assert( Ah ) ;  assert( Ph ) ; assert( pivoth ) ;
    assert( COL_MAJOR == Ah->sel ) ;
    assert( COL_MAJOR == Ph->sel ) ;
    assert( COL_MAJOR == pivoth->sel ) ;

    m = Ah->m ; n = Ah->n ;
    assert( m == n ) ;
    assert( 0 != Ah->isSym ) ; // A must be symmetric matrix

    assert( 1 == Ph->n ) ; assert( m <= Ph->m ) ;
    assert( 1 == pivoth->n ) ; assert( m <=  pivoth->m ) ;

    assert( (0 <= alpha) && (1.0 >= alpha) ) ;

    A = Ah->A ; P = Ph->A ; pivot = pivoth->A ;
    for ( i = 1 ; i <= n ; i++){
        P[1][i] = i ;
    }
    for ( i = 1 ; i <= n ; i++){
        pivot[1][i] = 0 ;
    }
```

Algorithm (PAP' = LDL', partial pivot) *[2]*

(1) we store lower triangle matrix *L* and block diagonal matrix *D* into storage of matrix *A*

(2) permutation matrix *P* and pivot sequence *pivot* are of type *int_matrix*

$k = 1$

while $k \le (n-1)$

we have compute $P^{(k-1)} A \left(P^{(k-1)} \right)^T = L^{(k-1)} A^{(k)} \left(L^{(k-1)} \right)^T$

$$A^{(k)} = \left(\begin{array}{ccc|ccc} D_1 & 0 & \cdots & 0 & \cdots & 0 \\ 0 & \ddots & & & & \\ \vdots & & D_s & \cdots & \cdots & 0 \\ \hline \vdots & & 0 & a_{k,k}^{(k)} & \cdots & a_{n,k}^{(k)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{n,k}^{(k)} & \cdots & a_{nn}^{(k)} \end{array}\right)$$

update original matrix A , where $D_i : 1\times 1$ or 2×2

$$L^{(k-1)} = \overbrace{\left(\begin{array}{c|c} W & O \\ \hline M & I \end{array}\right)}^{k-1} \Big\}k-1$$

combines all lower triangle matrix and store in L

Algorithm (PAP' = LDL', partial pivot) [3]

1 compute $\lambda_1 = \max_{k+1 \le j \le n} |A_{j1}| = |A_{r1}|, \quad r \ge k$

```
//
// step 1: compute  lambda_1 = max{|a(j,k)|: k+1<=j<=n}
        lambda_1 = 0.0 ;
        for( i = k ; i <= n ; i++ ){
            tmp = fabs( A[k][i] ) ;
            if ( tmp >  lambda_1 ){
                lambda_1 = tmp ;
                r = i ;
            }
        }// for each row
```

```
53 % step 1: compute  lambda_1 = max(|a(j,k)|: k+1<=j<=n)
54 %
55   [lambda_1, j] = max( abs(A(k:n,k)) ) ;
56   r = k + j - 1 ;
```

Case 1: $|a_{kk}| \ge \alpha\lambda_1$ 1×1 pivot no interchange

2 do 1x1 pivot : $A^{(k)} = \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & c^T \\ & c & B \end{pmatrix} = \begin{pmatrix} I & & \\ & 1 & \\ & c/a_{kk}^{(k)} & I \end{pmatrix} \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & \\ & & B - cc^T/a_{kk}^{(k)} \end{pmatrix} \left(L^{(k)}\right)^T$

$$\begin{cases} L(k+1:n,k) \leftarrow c/a_{kk}^{(k)} \\ A(k+1:n,k+1:n) -= cc^T/a_{kk}^{(k)} \end{cases}$$

then $A = L^{(k-1)}L^{(k)}A^{(k+1)}L^{(k)}\left(L^{(k-1)}\right)^T$

Technique problem: if we store *L* into memory block of *A*, then $L(k+1:n,k)$ will overwrite $c = A(k+1:n,k)$ such that $A(k+1:n,k+1:n) -= cc^T/a_{kk}^{(k)}$ fails.

We use a temporary storage $L(1:n,2)$

Algorithm (PAP' = LDL', partial pivot) [4]

do 1x1 pivot :

$$A^{(k)} = \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & c^T \\ & c & B \end{pmatrix} = \begin{pmatrix} I & & \\ & 1 & \\ & c/a_{kk}^{(k)} & I \end{pmatrix} \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & \\ & & B - cc^T/a_{kk}^{(k)} \end{pmatrix} \left(L^{(k)}\right)^T$$

$$\begin{cases} L(k+1:n,k) \leftarrow c/a_{kk}^{(k)} \\ A(k+1:n,k+1:n) -= cc^T/a_{kk}^{(k)} \end{cases}$$

```
78 % step 2: do 1x1 pivot,
79 %    L(k+1:n, k ) <-- c/A(k,k)
80 %    A(k+1:n, k+1:n) -= c*c'/A(k,k)
81 %
82      L(k+1:n, k    ) = A(k+1:n,k)/A(k,k) ;
83      A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - L(k+1:n,k)*A(k+1:n,k)' ;
84      A(k+1:n, k    ) = zero_vec(k+1:n) ;
85
```

```
    matrixHandler Lh ;
    doublereal **L ;

    zeros( &Lh, n, 2, COL_MAJOR  ) ;
    L = Lh->A ;
```

```
//      L(k+1:n, k    ) = A(k+1:n,k)/A(k,k) ;
            a_kk = A[k][k] ;
            for( i = k+1 ; i <= n ; i++){
                L[1][i] = A[k][i] / a_kk ;
            }
//      A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - L(k+1:n,k)*A(k+1:n,k)'
// remember: only update lower triangle part
            for (j = k+1 ; j <= n ; j++ ){
                tmp = A[k][j] ; // A(k,j) = A(j,k)
                for( i = j ; i <= n ; i++){
                    A[j][i] -= L[1][i] * tmp ;
                }// for each row ;
            }// for each column
// store L into A
            for( i = k+1 ; i <= n ; i++){
                A[k][i] = L[1][i] ;
            }
```

We must update lower triangle part of matrix *A* only.

Algorithm (PAP' = LDL', partial pivot) *[5]*

3 $k \leftarrow k+1$ and $pivot(k)=1$

```
// step 3: update k <- k+1 and pivot(k) = 1
            pivot[1][k] = 1 ;
            k = k+1 ;
```

```
86 %
87 % step 3: update k <- k+1 and pivot(k) = 1
88 %
89      pivot(k) = 1 ;
90      k = k+1 ;
```

When *case 1* is not satisfied, we compute

$$\lambda_r \equiv \max_{j \neq r} \left| a_{j,r} \right| =$$ maximum of off-diagonal elements in column ***r***

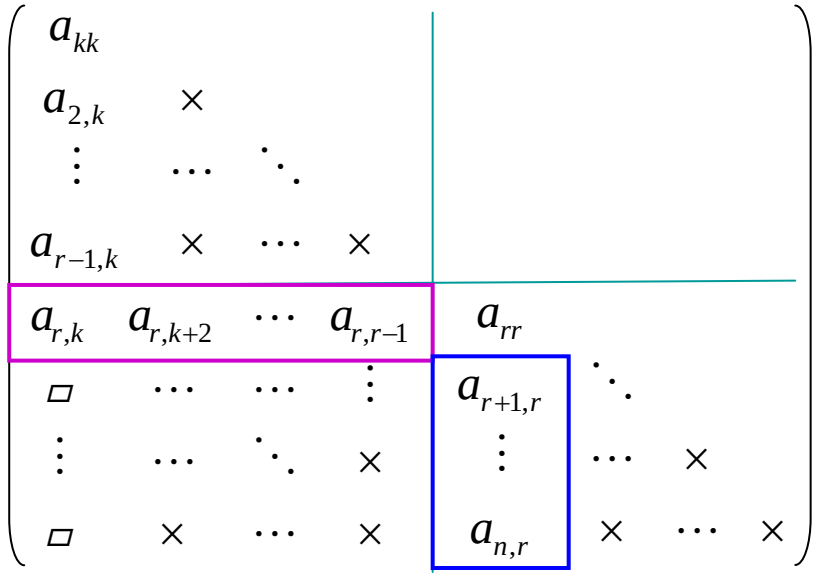
```
// step 4: find lambda_r = maxmum of off-diagonal of col-r
 // r < n : lambda_r = max( max(abs(A(r,k:r-1))), max(abs(A(r+1:n,r))) )
// r = n : lambda_r = max(abs(A(r,k:r-1)))
            lambda_r = 0.0 ;
            for( j = k ; j < r ; j++ ){
                tmp = fabs( A[j][r] ) ;
                if ( tmp > lambda_r ){
                    lambda_r = tmp ;
                }
            }// for each column
            if ( r < n ){
                for( i = r+1 ; i <= n ; i++ ){
                    tmp = fabs( A[r][i] ) ;
                    if ( tmp > lambda_r ){
                        lambda_r = tmp ;
                    }
                }// for each column
            }// if (r < n)
```

```
%
% step 4: find lambda_r = maxmum of off-diagonal of col-r
%
  if ( r < n )
    lambda_r = max( max(abs(A(r,k:r-1))), max(abs(A(r+1:n,r))) ) ;
  else
    lambda_r = max(abs(A(r,k:r-1))) ;
  end
```

Algorithm (PAP' = LDL', partial pivot) [6]

4 $\lambda_r = \max\left(\max\left|A(r,k:r-1)\right|, \max\left|A(r+1:n,r)\right|\right)$

$\lambda_r \geq \lambda_1 > 0$?



Case 2: $|a_{kk}|\lambda_r \geq \alpha\lambda_1^2$ 1×1 pivot no interchange

5 do 1x1 pivot : $A^{(k)} = \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & c^T \\ & c & B \end{pmatrix} = \begin{pmatrix} I & & \\ & 1 & \\ & c/a_{kk}^{(k)} & I \end{pmatrix} \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & \\ & & B - cc^T/a_{kk}^{(k)} \end{pmatrix} \left(L^{(k)}\right)^T$

$$\begin{cases} L(k+1:n,k) \leftarrow c/a_{kk}^{(k)} \\ A(k+1:n,k+1:n) -= cc^T/a_{kk}^{(k)} \end{cases}$$

then $A = L^{(k-1)}L^{(k)}A^{(k+1)}\left(L^{(k)}\right)^T\left(L^{(k-1)}\right)^T$

6 $k \leftarrow k+1$ and $pivot(k) = 1$

The same as code in case 1

Algorithm (PAP' = LDL', partial pivot) [7]

Case 3: $|a_{rr}| \geq \alpha \lambda_r$ 1×1 pivot do interchange

define permutation $P = (1:k-1, r, k+1:r-1, k, r+1:n)$ to do symmetric permutation

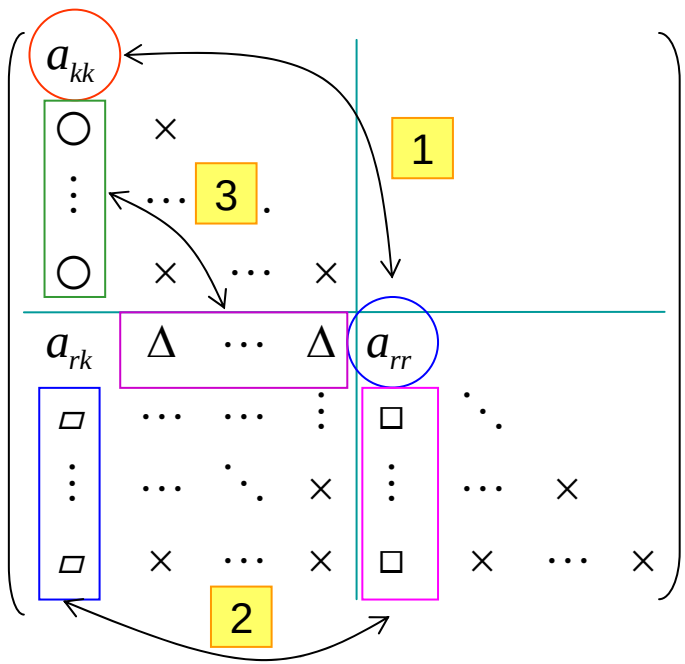
7 $P(k) \leftrightarrow P(r)$

```
int_tmp = P[1][k] ;
P[1][k] = P[1][r] ;
P[1][r] = int_tmp ;
```

```
tmp = P(k) ;
P(k) = P(r) ;
P(r) = tmp ;
```

To compute $\tilde{A}^{(k)} = P_k A^{(k)} P_k^T$, we only update lower triangle of $A^{(k)}$

8
1. $A(k,k) \leftrightarrow A(r,r)$
2. $A(r+1:n,k) \leftrightarrow A(r+1:n,r)$
3. $A(k+1:r-1,k) \leftrightarrow A(r,k+1:r-1)$



Algorithm (PAP' = LDL', partial pivot) [8]

8

1. $A(k,k) \leftrightarrow A(r,r)$
2. $A(r+1:n,k) \leftrightarrow A(r+1:n,r)$
3. $A(k+1:r-1,k) \leftrightarrow A(r,k+1:r-1)$

```
//
// step 8: compute A_tilda_{k} = P_{k} * A_{k} * (P_{k})'
//   (1) A(k,k) <--> A(r, r)
                    tmp      = A[k][k] ;
                    A[k][k]  = A[r][r] ;
                    A[r][r]  = tmp ;
//   (2) A(r+1:n, k) <--> A(r+1:n, r)
                    for ( i = r+1 ; i <= n ; i++ ){
                        tmp      = A[r][i] ;
                        A[r][i]  = A[k][i] ;
                        A[k][i]  = tmp ;
                    }
//   (3) A(k+1:r-1, k) <--> A(r, k+1:r-1)
                    for ( i = k+1 ; i < r ; i++){
                        tmp      = A[k][i] ;
                        A[k][i]  = A[i][r] ;
                        A[i][r]  = tmp ;
                    }
```

```
145 %
146 % step 8: compute A_tilda_(k) = P_(k) * A_(k) * (P_(k))'
147 %    (1) A(k,k) <--> A(r, r)
148 %    (2) A(r+1:n, k) <--> A(r+1:n, r)
149 %    (3) A(k+1:r-1, k) <--> A(r, k+1:r-1)
150 %
151        tmp     = A(k,k) ;
152        A(k,k) = A(r,r) ;
153        A(r,r) = tmp ;
154
155        tmp(r+1:n)   = A(r+1:n, r) ;
156        A(r+1:n, r) = A(r+1:n, k) ;
157        A(r+1:n, k) = tmp(r+1:n) ;
158
159        tmp_vec(k+1:r-1) = A(k+1:r-1, k) ;
160        A(k+1:r-1, k)    = A(r, k+1:r-1)';
161        A(r, k+1:r-1)    = tmp_vec(k+1:r-1)' ;
```

then $\tilde{A}^{(k)} = \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & c^T \\ & c & B \end{pmatrix}$, $a_{kk}^{(k)} := a_{rr}$

Algorithm (PAP' = LDL', partial pivot) [9]

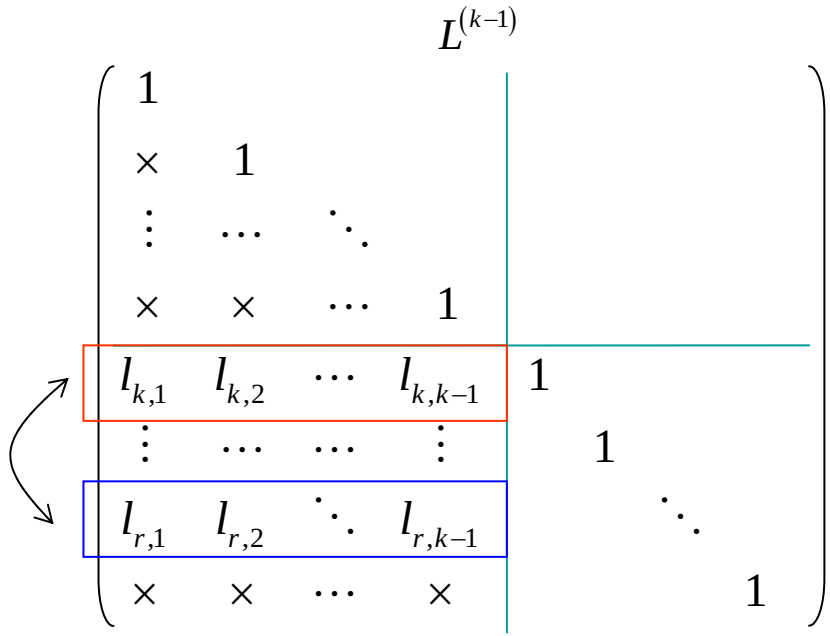
To compute $\tilde{L}^{(k-1)} = P_k L^{(k-1)} P_k^T$

9 We only update lower triangle matrix L

$$L(k,1:k-1) \leftrightarrow L(r,1:k-1)$$

then

$$P^{(k)} A \left(P^{(k)}\right)^T = \tilde{L}^{(k-1)} \tilde{A}^{(k)} \left(\tilde{L}^{(k-1)}\right)^T$$




```
//
// step 9: compute L_tilda_{k} = P_{k} * L_{k} * (P_{k})'
//    L(k, 1:k-1) <--> L(r, 1:k-1)
                  if ( 1 < k ){
                      for ( j = 1 ; j < k ; j++){
                          tmp     = A[j][k] ;
                          A[j][k] = A[j][r] ;
                          A[j][r] = tmp ;
                      }
                  }// if ( k > 1)
```

```
163 %
164 % step 9: compute L_tilda_(k) = P_(k) * L_(k) * (P_(k))'
165 %    L(k, 1:k-1) <--> L(r, 1:k-1)
166 %
167       if ( k > 1 )
168         tmp_vec(1:k-1) = L(k,1:k-1)' ;
169         L(k, 1:k-1)    = L(r, 1:k-1) ;
170         L(r, 1:k-1)    = tmp_vec(1:k-1)' ;
171       end
```

we store lower triangle matrix *L* into storage of matrix *A*

Algorithm (PAP' = LDL', partial pivot) [10]

10 do 1x1 pivot : $\tilde{A}^{(k)} = \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & c^T \\ & c & B \end{pmatrix} = \begin{pmatrix} I & & \\ & 1 & \\ & c/a_{kk}^{(k)} & I \end{pmatrix} \begin{pmatrix} D_{k-1} & & \\ & a_{kk}^{(k)} & \\ & & B - cc^T / a_{kk}^{(k)} \end{pmatrix} \left(L^{(k)}\right)^T$

$$\begin{cases} L(k+1:n,k) \leftarrow c / a_{kk}^{(k)} \\ A(k+1:n,k+1:n) -= cc^T / a_{kk}^{(k)} \end{cases}$$

then $P^{(k)} A \left(P^{(k)}\right)^T = \tilde{L}^{(k-1)} L^{(k)} A^{(k+1)} \left(L^{(k)}\right)^T \left(\tilde{L}^{(k-1)}\right)^T$

11 $k \leftarrow k+1$ and $pivot(k) = 1$

The same as code in case 1

Case 4: $|a_{rr}| < \alpha \lambda_r$ 2×2 pivot do interchange

define permutation $P = (1:k, r, k+2:r-1, k+1, r+1:n)$, change a_{rk} to $a_{k+1,k}$

12 $P(k+1) \leftrightarrow P(r)$

```
int_tmp   = P[1][k+1] ;
P[1][k+1] = P[1][r] ;
P[1][r]   = int_tmp ;
```

```
tmp     = P(k+1) ;
P(k+1)  = P(r) ;
P(r)    = tmp ;
```

Algorithm (PAP' = LDL', partial pivot) [11]

To compute $\tilde{A}^{(k)} = P_k A^{(k)} P_k^T$, we only update lower triangle of $A^{(k)}$

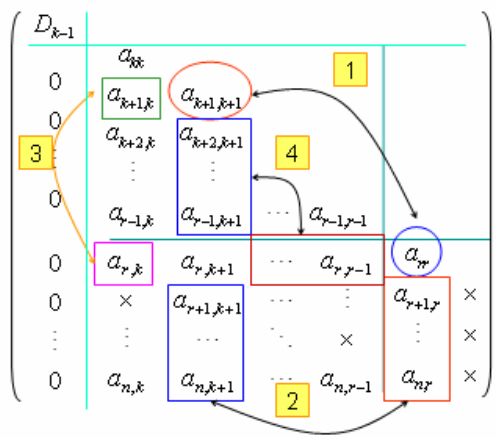
13 do interchange row/col $k+1 \leftrightarrow r$

1 $A(k+1,k+1) \leftrightarrow A(r,r)$

2 $A(r+1:n,k+1) \leftrightarrow A(r+1:n,r)$

3 $A(k+1,k) \leftrightarrow A(r,k)$

4 $A(k+2:r-1:k+1) \leftrightarrow A(r,k+2:r-1)$




```
//
// step 13: do interchange row/col k+1 <--> r
//  (1) A(k+1,k+1) <--> A(r,r)
                      tmp         = A[k+1][k+1] ;
                      A[k+1][k+1] = A[r][r] ;
                      A[r][r]     = tmp ;
//   (2) A(r+1:n, k+1 ) <--> A(r+1:n, r)
                      for ( i = r+1 ; i <= n ; i++){
                          tmp       = A[k+1][i] ;
                          A[k+1][i] = A[r][i] ;
                          A[r][i]   = tmp ;
                      }
//   (3) A(k+1,k) <--> A(r,k)
                      tmp       = A[k][k+1] ;
                      A[k][k+1] = A[k][r] ;
                      A[k][r]   = tmp ;
//   (4) A(k+2:r-1,k+1) <--> A(r, k+2:r-1)
                      for ( i = k+2 ; i < r ; i++){
                          tmp       = A[k+1][i] ;
                          A[k+1][i] = A[i][r] ;
                          A[i][r]   = tmp ;
                      }
```

```
% step 13: do interchange row/col k+1 <--> r
%    (1) A(k+1,k+1) <--> A(r,r)
%    (2) A(r+1:n, k+1 ) <--> A(r+1:n, r)
%    (3) A(k+1,k) <--> A(r,k)
%    (4) A(k+2:r-1,k+1) <--> A(r, k+2:r-1)
%
      tmp         = A(k+1,k+1) ;
      A(k+1,k+1)  = A(r,r) ;
      A(r,r)      = tmp ;

      tmp_vec(r+1:n) = A(r+1:n, k+1 ) ;
      A(r+1:n, k+1 ) = A(r+1:n, r) ;
      A(r+1:n, r)    = tmp_vec(r+1:n) ;

      tmp       = A(k+1,k) ;
      A(k+1,k)  = A(r,k) ;
      A(r,k)    = tmp ;

      tmp_vec(k+2:r-1) = A(k+2:r-1 , k+1) ;
      A(k+2:r-1 , k+1) = A(r , k+2:r-1)' ;
      A(r, k+2:r-1)    = tmp_vec(k+2:r-1)' ;
```

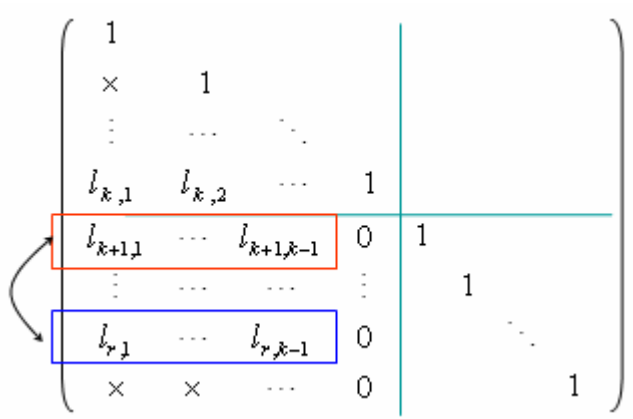
Algorithm (PAP' = LDL', partial pivot) [12]

14 do interchange row $k+1 \leftrightarrow r$

$$L(k+1,1:k-1) \leftrightarrow L(r,1:k-1)$$

then

$$P^{(k)} A \left(P^{(k)}\right)^T = \tilde{L}^{(k-1)} \tilde{A}^{(k)} \left(\tilde{L}^{(k-1)}\right)^T$$



```
// to compute L_tilda_{k} = P_{k} * L_{k} * (P_{k})'
                        if ( k > 1){
// step 14: do interchange row k+1 <--> r
//          L(k+1, 1:k-1) <--> L(r, 1:k-1)
                            for ( j = 1 ; j < k ; j++ ){
                                tmp       = A[j][k+1] ;
                                A[j][k+1] = A[j][r] ;
                                A[j][r]   = tmp ;
                            }
                        }// if ( k > 1)
```

```
      if ( k > 1 )
%
% step 14: do interchange row k+1 <--> r
%     L(k+1, 1:k-1) <--> L(r, 1:k-1)
%
        tmp_vec(1:k-1) = L(k+1,1:k-1)' ;
        L(k+1 , 1:k-1) = L(r  ,1:k-1) ;
        L(r   , 1:k-1) = tmp_vec(1:k-1)' ;
      end % if ( k > 1 )
```

15 do 2x2 pivot : $\tilde{A}^{(k)} = \begin{pmatrix} D_{k-1} & & \\ & E & c^T \\ & c & B \end{pmatrix} = \begin{pmatrix} I & & \\ & I & \\ & cE^{-1} & I \end{pmatrix} \begin{pmatrix} D_{k-1} & & \\ & E & \\ & & B - cE^{-1}c^T \end{pmatrix} \left(L^{(k)}\right)^T$

$$\begin{cases} L(k+2:n,k:k+1) \leftarrow cE^{-1} \\ A(k+2:n,k+2:n) -= cE^{-1}c^T \end{cases}$$

then $P^{(k)} A \left(P^{(k)}\right)^T = \tilde{L}^{(k-1)} L^{(k)} \; A^{(k+2)} \; \left(L^{(k)}\right)^T \left(\tilde{L}^{(k-1)}\right)^T$

Algorithm (PAP' = LDL', partial pivot) [13]

do 2x2 pivot :

$$\begin{cases} L(k+2:n, k:k+1) \leftarrow cE^{-1} \\ A(k+2:n, k+2:n) -= cE^{-1}c^T \end{cases}$$

```
// compute inv(E)
            detE = A[k][k] * A[k+1][k+1] - A[k][k+1]*A[k][k+1] ;
            invE_11 = A[k+1][k+1] / detE ;
            invE_22 = A[k][k] / detE ;
            invE_12 = -A[k][k+1] / detE ;
            invE_21 = invE_12 ;
// L(k+2:n, k:k+1) = A(k+2:n, k:k+1)*inv(E) ;
            for (i = k+2 ; i <= n ; i++ ){
                L[1][i] = A[k][i] * invE_11 + A[k+1][i]* invE_21 ;
                L[2][i] = A[k][i] * invE_12 + A[k+1][i]* invE_22 ;
            }
// A(k+2:n, k+2:n) = A(k+2:n, k+2:n) - L(k+2:n, k:k+1)* (A(k+2:n, k:k+1)') ;
            for ( j = k+2 ; j <= n ; j++){
                for ( i = j ; i <= n ; i++){
                    A[j][i] -=  L[1][i] * A[k][j] + L[2][i] * A[k+1][j] ;
                }// for each row
            }// for each col
// store L into A
            for (i = k+2 ; i <= n ; i++ ){
                A[k  ][i] = L[1][i] ;
                A[k+1][i] = L[2][i] ;
            }
```

$$E = \begin{pmatrix} a_{k,k} & a_{k+1,k} \\ a_{k+1,k} & a_{k+1,k+1} \end{pmatrix}$$

$$E^{-1} = \frac{1}{\det E} \begin{pmatrix} a_{k+1,k+1} & -a_{k+1,k} \\ -a_{k+1,k} & a_{k,k} \end{pmatrix}$$

```
%
% step 15: do 2x2 pivoting
%
    E(1,1) = A(k  ,  k) ;
    E(2,1) = A(k+1,  k) ;
    E(1,2) = E(2,1) ;
    E(2,2) = A(k+1,k+1) ;

    L(k+2:n, k:k+1) = A(k+2:n, k:k+1)*inv(E) ;
    A(k+2:n, k+2:n) = A(k+2:n, k+2:n) - L(k+2:n, k:k+1)* (A(k+2:n, k:k+1)') ;
    A(k+2:n, k  )   = zero_vec(k+2:n) ;
    A(k+2:n, k+1)   = zero_vec(k+2:n) ;
```

Algorithm (PAP' = LDL', partial pivot) [14]

16 $k \leftarrow k+2$ and $pivot(k) = 2$

```
// step 16: update pivot(k) = 2 and k <-- k+2
            pivot[1][k] = 2 ;
            k = k + 2 ;
```

```
%
% step 16: update pivot(k) = 2 and k <-- k+2
%
    pivot(k) = 2 ;
    k = k + 2 ;
```

endwhile

```
// terminatin condition for pivot array
// when k = n-1, do 1x1 pivoting, after leaving while loop, pivot(n) = 0,
// we must correct pivot(n) = 1 (implicit 1x1 pivoting)
    if ( 0 == pivot[1][n] ){
        if ( 1 == pivot[1][n-1] ){
            pivot[1][n] = 1 ;
        }
    }
```

```
% terminatin condition for pivot array
% when k = n-1, do 1x1 pivoting, after leaving while loop, pivot(n) = 0,
% we must correct pivot(n) = 1 (implicit 1x1 pivoting)
%
if ( 0 == pivot(n) )
  if ( 1 == pivot(n-1) )
    pivot(n) = 1 ;
  end
end
```

Example: partial pivot

A

6	12	3	-6
12	-8	-13	4
3	-13	-7	1
-6	4	1	6

$$\alpha = \frac{1+\sqrt{17}}{8}$$

Bunch-Kaufman (partial pivot)

$P =$

1	2	4	3

$pivot =$

2	0	1	1

$L =$

1			
	1		
0	-0.5	1	
-0.6875	0.5938	-0.6875	1

$D =$

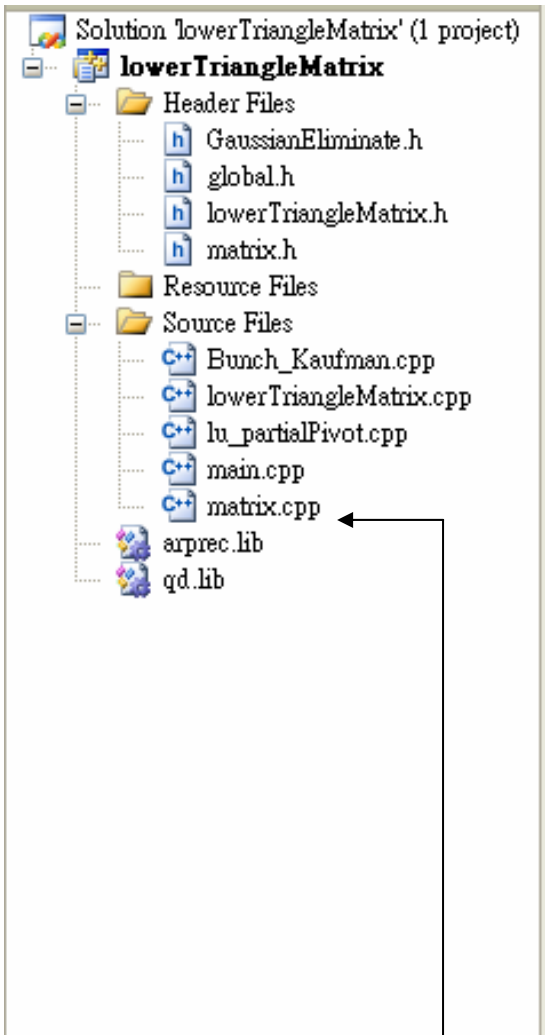
6			
12	-8		
		8	
			-1

$L + D \rightarrow mem(A) =$

6			
12	-8		
0	-0.5	8	
-0.6875	0.5938	-0.6875	-1

driver for example [1]

main.cpp




```
#include "global.h"
#include <stdio.h>
#include "lowerTriangleMatrix.h"
#include "GaussianEliminate.h"

void  test_BunchKaufman( void ) ;

int main( int argc, char* argv[])
{
#ifdef  HIGH_PRECISION_PACKAGE
    unsigned int old_cw;
    fpu_fix_start(&old_cw);
#endif

#ifdef DO_ARPREC
    mp::mp_init(ARPREC_NDIGITS);
#endif

// ---------- main code ---------------

    test_BunchKaufman() ;

// ---------- end main code ------------

#ifdef DO_ARPREC
    mp::mp_finalize();
#endif

#ifdef  HIGH_PRECISION_PACKAGE
    fpu_fix_end(&old_cw);
#endif
    return 0 ;
}
```

Test example

We need full matrix representation of x, b of linear system $Ax = b$ and permutation matrix *P*, pivot sequence *pivot*

driver for example [2]

```
void  test_BunchKaufman( void )
{
    integer m = 4 ;
    integer n = 4 ;
    lowerTriangleMatrixHandler Ah ;
    lowerTriangleMatrixHandler Ah_dup;
    int_matrixHandler Ph ;
    int_matrixHandler pivoth ;
    matrixHandler bh ;
    matrixHandler xh ;  // x = inv(A)*b
    matrixHandler bh_hat ; // b_hat = A*x
    matrixHandler rh ; // residual r = b - Ax
    doublereal  r_supnorm ;
    int isSingular ;
    doublereal **A ;
    doublereal **b ;
    doublereal alpha ;

    zeros( &Ah, m, n, COL_MAJOR, 1 ) ;
    A  = Ah->A ;

    A[1][1] =  6. ; A[1][2] = 12. ; A[1][3] =  3.  ; A[1][4] = -6.  ;
                    A[2][2] = -8. ; A[2][3] = -13. ; A[2][4] =  4.  ;
                                    A[3][3] = -7.  ; A[3][4] =  1.  ;
                                                     A[4][4] =  6.  ;

    cout << "configuration of matrix A" << endl ;
    disp( Ah, cout ) ;

    zeros( &Ah_dup, m, n, COL_MAJOR, 1 ) ;
    duplicate( Ah, Ah_dup ) ;

    zeros( &Ph    , m, 1, COL_MAJOR ) ;
    zeros( &pivoth, m, 1, COL_MAJOR ) ;
    alpha = (1.0 + sqrt(17.0))/8.0 ;
// step 2: factorization, PAP' = LDL'
    cout.precision(4) ;
    cout << std::fixed ;
    isSingular = bunch_kaufman_v2( Ah, Ph, pivoth, alpha ) ;

        cout << "\nPAP' = LDL' : store L and D into A: with alpha = " << alpha << endl ;
        disp( Ah, cout ) ;

        cout << "\npermutation matrix P:\n" ;
        disp( Ph, cout ) ;

        cout << "\npivot matrix piovt:\n" ;
        disp( pivoth, cout ) ;
```

6			
12	-8		
0	-0.5	8	
-0.6875	0.5938	-0.6875	-1

```
C:\WINDOWS\system32\cmd.exe
configuration of matrix A
dimension of symmetric matrix is (4,4) with col-major
    6.0000
   12.0000   -8.0000
    3.0000  -13.0000   -7.0000
   -6.0000    4.0000    1.0000    6.0000

PAP' = LDL' : store L and D into A: with alpha = 0.6404
dimension of symmetric matrix is (4,4) with col-major
    6.0000
   12.0000   -8.0000
   -0.0000   -0.5000    8.0000
   -0.6875    0.5938   -0.6875   -1.0000
```

```
permutation matrix P:
dimension of matrix is (4,1) with col-major
         1
         2
         4
         3

pivot matrix piovt:
dimension of matrix is (4,1) with col-major
         2
         0
         1
         1
```

Report

- Implement PAP’ = LDL’ and linear solver in C/C++
- describe program structure
- How to verify your program
- Comparison with MATLAB implementation
- Memory usage (do you need extra storage?)
- Speedup strategy