Math 110 - Fall 05 - Lectures notes # 20 - Oct 14 (Friday)

We will cover the contents of Chapter 3 slightly differently,
than the textbook, showing how to compute
range(A), rank(A), nullspace(A), nullity(A),
A^{-1} (if it exists)
solve Ax=b (determine if there are no solutions, 1 solution,
or infinitely many solutions).
all by computing a single "matrix factorization" of A, called
the "LU factorization" or "LU decomposition", which is
a formula for A as product of much simpler matrices.
The algorithm for computing this factorization is just
Gaussian elimination, where we keep track of all the
elementary matrix operations we do:
reordering rows and/or columns
adding a multiply of one row to another
multiplying a row or column by a scalar
as a product of simple matrices. When you use computer software to
do matrix operations, it will return your answer as a product of
simpler matrices much like the ones we show here.

LU decomposition plays an important role in government high technology
policy: the largest computers in the world (called "supercomputers") are
regularly measured by how fast they can solve Ax=b using LU decomposition
on very large matrices, with the ratings appearing on the web site
www.top500.org. This web site keeps track of the 500 fastest computers
anywhere in the world (at least those not at secret government sites).
For example, as of June 2005, the top computer was an IBM Blue Gene/L,
consisting of 65,536 parallel processors. The machine solved
a linear system of n = 1,277,951 equations using the LU decomposition
algorithm described below at a speed of 136.8 Teraflops, that is
136.8 trillion = 136.8 x 10^12 arithmetic operations per second.
Since the number of arithmetic operations required by the algorithm
will turn out to be about 2/3*n^3, the time required to solve this
single linear system was
(2/3)*(1,277,951)^3 / 136.8x10^12 = 10161 seconds ~ 2.8 hours
Recall that multiplying 2 matrices takes about 2n^3 operations,
3 times more operations.

A few years ago the fastest machine on the Top 500 list was the
NEC "Earth Simulator", which ran at a speed of 35.9 Teraflops, as fast
as the next four machines combined. This caught the attention of parts

of the US government responsible for national security, because NEC is a Japanese company, and one national security goal of the US government is to have better technology, including faster computers, than anyone else in the world. This is because these computers are used for breaking secret codes, designing weapons, and other national security purposes. Indeed, the largest computers in the US have been built for these purposes for decades. The speed of the NEC machine led to a flurry of government studies and reports to analyze the situation, for example www.sc.doe.gov/ascr.FOSCfinalreport.pdf. More aspects of implementing linear algebra algorithms on parallel machines are discussed in the course CS267, see www.cs.berkeley.edu/~demmel/cs267_Spr05.

One of the most widely used software packages for solving linear algebra problems is called LAPACK (its parallel version is ScaLAPACK); for example it is used within Matlab. This software was produced in a joint project between UC Berkeley, the University of Tennessee, and other groups. We have recently been funded to produce a new version. See www.netlib.org/lapack for the current software.

```
Ex: Suppose L is square and lower triangular:
    L = [ 1  0 0 ]
        [ 1  2 0 ]
        [ 3 -1 3 ]
    Then we can solve Lx = b for x given any b by "substitution"
    Solve 1*x_1 = b_1                 for x_1
    Solve 1*x_1 + 2*x_2 = b_2         for x_2
    Solve 3*x_1 - 1*x_2 + 3*x_3 = b_3 for x_3

    As long as each L_ii is nonzero, so we can divide by it,
    we can solve any Lx=b for x and get a unique answer.
    In particular, the only solution of Lx = 0 is x=0, so that
    L is one-to-one. This implies that L is invertible.
    We can compute L^{-1} by solving for it one column at a time:
    Let X = L^{-1}, so L*X = I, and e_j = j-th standard basis
    vector, so L*X*e_j = I*e_j, or
      L*(X*e_j) = e_j    or    L*(column j of X) = e_j
    so we can solve for X = L^{-1} one column at a time.

Ex: Suppose U is square, upper triangular, and each U_ii is nonzero.
    Then we can similarly solve any U*x=b for a unique x given b by
    substitution, solving first for x_n, then x_{n-1}, and so on.
    We can also compute U^{-1} as above.
```

```
Ex: Suppose A = L*U, where L and U are n by n and nonsingular.
    Then A is invertible (why?), and
      A^{-1} = (L*U)^{-1} = U^{-1} * L^{-1}
    and solving A*x = b for x given b means computing
      x = A^{-1}b
        = (U^{-1} * L^{-1})*b
        = U^{-1} * (L^{-1}*b)    ... by associativity
        = U^{-1} * y
             ... where we compute y = L^{-1}*b i.e. solve L*y=b by substitution
     where we finally compute x = U^{-1}*y i.e. solve U*x = y by substitution
     again. In other words, if we know A = L*U where L and U are invertible
     triangular matrices, we can solve A*x = b by 2 steps:
       (1) solve L*y = b by substitution
       (2) solve U*x = y by substitution
     If we want to compute A^{-1} we use the same idea as above,
     solving A*X = I for X one column of X at a time.

Stop&Ask: Suppose A = U*L where U and L are n by n and invertible.
          How do we solve A*x=b?

It may seem strange to expect to know A in the form A = L*U, but
this is in fact exactly what Gaussian elimination applied to A computes.

Ex: A = [ 3 -1   2 ]
        [ 6  0  10 ]
        [-3 -3 -17 ]

To apply Gaussian elimination, we do the following:
   for i = 1 to n-1 (3-1 = 2 in this case)
     subtract multiples of row i from rows i+1 through n,
      in order to zero out A below the diagonal

  Subtract 2  * row 1 from row 2
  Subtract -1 * row 1 from row 3,
     yielding A' = [ 3 -1 2 ; 0 2 6 ; 0 -4 -15 ]

  Subtract -2 * row 2 from row 2,
     yielding A'' = [ 3 -1 2 ; 0 2 6 ; 0  0 -3 ]

Note that A'' = U is upper triangular. Let's take the multipliers
we computed and put them into a lower triangular matrix L
```

```
   L = [ 1  0 0 ]
       [ 2  1 0 ]
       [-1 -2 1 ]
Then we can confirm that L*U = A. We will see that
this is true in general, as long as we never get a 0 on the diagonal.

ASK & WAIT: What happens if A_11 = 0? What happens at any point if some
            diagonal entry = 0?

If A is m-by-n, we will only get a zero on the diagonal
when rank(A) < min(m,n), as we will see. First we will see what happens
assuming this is not the case, and then show the most general case.

We will use induction. To do so, we need some basic facts
about "block matrices", which are a useful notation to
keep track of what parts of the matrix we're working on:

Def. Let A by m by n, where m = m1 + m2 and n = n1 + n2
Then we can write A as a "block matrix"
     A = [ A_11  A_12 ]
         [ A_21  A_22 ]
where A_11 and A_21 have m1 columns,  A_12 and A_22 have m2 columns
      A_11 and A_12 have n1 rows, and A_21 and A_22 have n2 rows.
We sometimes write the dimensions around the matrix to indicate this:
            m1    m2
     A = [ A_11  A_12 ] n1
         [ A_21  A_22 ] n2

Here is perhaps the most useful fact about block matrices,
that when we multiply them, we can just multiply and add the blocks,
as though they were scalars:

Lemma 1: Let A and B be block matrices, where A's column block
       dimenions m1 and m2 match B's row block dimensions:
            m1    m2                p1    p2
     A = [ A_11  A_12 ] n1    B = [ B_11  B_12]  m1
         [ A_21  A_22 ] n2        [ B_21  B_22]  m2
       This means that matrix products like A_11*B_11 are defined,
       and in fact A*B can be written as the block matrix
                      p1                      p2
     A*B = [ A_11*B_11 + A_12*B_21  A_11*B_12 + A_12*B_22 ] n1
           [ A_21*B_11 + A_22*B_21  A_21*B_12 + A_22*B_22 ] n2
```

```
     Note that this is the usual formula for multiplying 2 by 2
     matrices, when all the block A_ij and B_ij are 1 by 1, i.e. scalars.

Proof: (Picture) When (i,j) is in the top left block, we see
        (A*B)_ij = sum_{k=1 to m} A_ik*B_kj
                 = sum_{k=1 to m1} A_ik*B_kj
                     + sum_{k=m1+1 to m} A_ik*B_kj
                 = sum_{k=1 to m1} A_11_ik*B_11_kj
                     + sum_{k=1 to m2} A_12_ik*B_21_kj
                 = (A_11*B_11)_ij + (A_12*B_21)_ij
       or the leading n1 by p1 block of A*B = A_11*B_11 + A_12*B_21
       The other blocks of A*B are similar.

Corollary 1: Let
            m1   m2
     L = [ I^m1   0  ] m1
         [ L_21 I^m2 ] m2
   be a lower triangular matrix, where I^m1 is an m1 by m1 identity matrix,
   I^m2 is an m2 by m2 identity matrix, and 0 is an m1 by m2 zero matrix.
   Then
                 m1   m2
     L^{-1} = [ I^m1   0  ] m1
              [-L_21 I^m2 ] m2
   Proof: We just confirm L*L^{-1} = I^m = m by m identity matrix:
          The dimensions of the blocks match, letting us use Lemma 1
          to compute
         [ I^m1   0  ] * [ I^m1   0  ]
         [ L_21 I^m2 ]   [-L_21 I^m2 ]
       = [ I^m1 * I^m1 + 0 * (-L_21)    I^m1 * 0 + 0 * I*m2   ]
         [ L_21*I^m1 + I^m2*(-L_21)     L_21* 0 + I^m2 * I^m2 ]
       = [ I^m1 + 0                     0 + 0    ]
         [ L_21 - L_21                  0 + I^m2 ]
       = [ I^m1   0  ]
         [  0   I^m2 ]
       = I^m
```

Now we can show how to compute a factorization A = LU assuming we never get a zero on the diagonal. We present it for A rectangular.

Thm 1 (Gaussian Elimination, assuming no 0s appear on the diagonal) Let A be m by n. Then there is a lower triangular matrix L and upper triangular matrix U such that A = L*U:

(1) When A is square (m=n) then L and U are also n by n
(2) When A has more rows than columns (m>n), then L is m by n and U is n by n
    (picture)
(3) When A has more columns than rows (m<n), then L is m by m and U is m by n
    (picture)
In fact, we can pick L to have ones on its diagonal.
This is also called "LU factorization".

Proof:
Base cases: When min(m,n)=1, we pick L and U so A=L*U as follows:

```
(1) When A = [ a_11 ] is 1 by 1, choose L = 1 and U = A
(2) When m > n = 1, so A = [a_11] with a_11 neq 0, pick L = [     1     ]
                           [a_21]                           [ a_21/a_11 ]
                           [... ]                           [    ...    ]
                           [a_m1]                           [ a_m1/a_11 ]
    and U = a_11
(3) When m = 1 < n, so A = [a_11 a_12 ... a_1n ] with a_11 neq 0,
    pick L = 1 and U = A
```

Now we do induction. We assume the theorem is true for m-1 by n-1 matrices,
and prove it is true for an m by n matrix A. By assumption a_11 neq 0.
We write A as a block matrix (note that A_11 is just a_11):

```
             1      n-1
      A = [ A_11   A_12 ] 1
          [ A_21   A_22 ] m-1
```

Subtracting a multiple of row 1 from row i to zero out a_i1 means the
multiplier must be a_i1/a_11, since this changes the (i,1) entry from a_i1
to a_i1 - (a_i1/a_11)*a_11 = 0. We express this by premultiplying A by

```
                                      1            n-1
   L' = [ 1          0 0 0 ... 0 ]  = [ 1             0      ]  1
        [ -a_21/a_11 1 0 0 ... 0 ]    [-A_21/A_11    I^{m-1} ]  m-1
        [ -a_31/a_11 0 1 0 ... 0 ]
        [           ...          ]
        [ -a_m1/a_11 0   ... 0 1 ]
```

where -A_21/A_11 is m-1 by 1, and 0 is 1 by m-1. Lemma 1 lets us multiply

```
   L' * A = [      1        0     ] * [ A_11  A_12 ]
            [ -A_21/A_11  I^{n-1} ]   [ A_21  A_22 ]
          = [ 1*A_11 + 0*A_21                  1*A_12 + 0*A_22                 ]
            [ -A_21/A_11*A_11 + I^{n-1}*A_21  -A_21/A_11*A_12 + I^{n-1}*A_22 ]
```

```
         = [ A_11         A_12              ]
           [  0      A_22 - A_21*A_12/A_11 ]

confirming we have zeroed out the first column beneath the diagonal as desired.

Another way to see what it means to multiply L'*A is to look at it row by row:
   row j of L'*A = e_j^t *(L'*A)   ... e_j = j-th standard basis vector
                 = (e_j^t*L')*A    ... by associativity
                 = (row j of L')*A
When j=1, row j of L' = e_1^t so row 1 of L'*A = row 1 of A
When j=2, row j of L' = [-a_21/a_11, 1, 0, ..., 0]
          so row 2 of L'*A = (-a_21/a_11)*(row 1 of A) + (1)*(row 2 of A)
Similary, for other j, row j of L'*A = (-a_j1/a_11)*(row 1 of A) + (row j of A)

The quantity S = A_22 - A_21 * A_12 / A_11 is called a "Schur complement".
Note that L'*A is closer to upper triangular, because it is zero below
the first diagonal. Now  S is m-1 by n-1 so we can apply induction to write
S = L_S * U_S (where we again assume no zeros appear on the diagonal):

                                 1     n-1
   L' * A = [ A_11   A_12 ] = [ A_11  A_12     ]  1
            [  0      S   ]   [  0    L_S*U_S ]  m-1

          = [ 1    0  ] * [ A_11  A_12 ]
            [ 0   L_S ]   [  0    U_S  ]

Note that the second matrix above is upper triangular; we will call it U. Then
   A = L'^{-1} * [ I  0  ] * U
                 [ 0 L_S ]
     = [ 1    0     ]^{-1}  * [ 1  0  ] * U
       [ X  I^{n-1} ]         [ 0 L_S ]
             ... where X = -A_21/A_11 for short
     = [ 1    0     ] * [ 1  0  ] * U
       [-X  I^{n-1} ]   [ 0 L_S ]
             ... by Corollary 1
     = [ 1*1 + 0*0          1*0 + 0*L_S       ] * U
       [-X*1 + I^{n-1}*0   -X*0 + I^{n-1}*L_S ]
     = [  1  0  ] * U
       [ -X L_S ]
     = L*U
where L = [ 1  0  ] is lower triangular with ones on the diagonal as desired.
          [-X L_S ]
```

and with the multipliers from Gaussian elimination below the diagonal.
This completes the proof of Thm 1.

Now we generalize this result to the case where zeros may appear on the diagonal.
For example, if A_11 had been 0 above, we would have tried to divide by
zero, and it would not have worked. But as long as there is a nonzero somewhere
in the matrix, we can always reorder the rows and columns to put that nonzero
in the (1,1) entry. If, after the first step, the (2,2) entry is zero, we
again permute rows and columns to put a nonzero there if one exists.
We will also use matrix-multiplication, by "permutation matrices",
to keep track of this bookkeeping.

```
Ex: If A = [ 0 1 ], then we can't write
           [ 1 2 ]
    A = LU = [ l_11   0  ] * [ u_11 u_12 ] = [ l_11*u_11  l_11*u_12                ]
             [ l_21 l_22 ]   [  0   u_22 ]   [ l_21*u_11  l_21*u_12 + l_22*u_22 ]
    unless l_11 = 0 or u_11 = 0 to match a_11 = 0.
    But this would mean either a_12 = 0 (if l_11=0) or a_21 = 0 (if u_11=0)
    a contradiction. But if we permute A's rows to get [ 1 2 ; 0 1 ],
    or A's columns to get [ 1 0 ; 2 1 ]; then we can do LU factorization.
```

Def: A permutation matrix P is an n by n matrix with exactly one 1 in each
row, one 1 in each column, and the other entries equal to 0.

An equivalent definition is that a permutation matix P is gotten by taking the
identity matrix, and permuting its rows (or its columns).

```
Ex: P1 = [ 1 0 ],  P2 = [ 0 1 ],  P3 = [ 0 0 1 ]
         [ 0 1 ]        [ 1 0 ]        [ 1 0 0 ]
                                       [ 0 1 0 ]
```

Stop&Ask: How do we construct P2 and P3 by permuting rows or columns of I?

Lemma 2: Let P be a permutation matrix. Then
(1) If y = P*x, then y has the same entries as x but in a permuted order
(2) If Y = P*X, then Y has the same rows as X but in a permuted order
(3) If Y = X*P, then Y has the same columns as X but in a permuted order
(4) If P1 and P2 are permutation matrices, so is P1*P2.
(5) If P is a permutation matrix, so is P^t = P^{-1}

```
Ex: P1*[x1] = [x1],  P2*[x1] = [x2],  P3*[x1] = [x3]
       [x2]   [x2]      [x2]   [x1]      [x2]   [x1]
```

```
                                        [x3]   [x2]

Proof:
  (1) Recall that y=Px -> y_i = e_i^t*y = e_i^t*(P*x) = (e_i^t*P)*y
      = (row i of P)*y.
      Since row i of P is a row of the identity matrix, say e_j^t, then
      y_i = e_j^t * x = x_j. Since each row of P is a different row of the
      identity, each y_i is a different component of x.
  (2) Apply (1) to each column of Y and X, since Y = P*X means that for all j,
      column j of Y = Y*e_j = (P*X)*e_j = P*(X*e_j) = P*(column j of X)
  (3), (4) and (5):  homework!

Finally, we can do the general case of Gaussian elimination:



Thm 2 (Gaussian Elimination, or LU decomposition in general case)
     Let A be m by n and nonzero. Then we can write
     A = P_L * L * U * P_R
where, for some r with 1 <= r <= min(m,n)
     P_L is an m by m permutation matrix (needed to permute A's rows)
     L   is an m by r lower triangular matrix,
          (with ones on the diagonal)
     U   is an r by n upper triangular matrix
          (with nonzero diagonal)
     P_R is an n by n permutation matrix (needed to permute A's columns)

We can write A = P_L*L*U*P_R equivalently as P_L^t*A*P_R^t = L*U,
or that A's rows and columns can be reordered (by P_L^t and P_R^t,
respectively), so that the reordered matrix has an LU decomposition.

Proof: We use induction as before. This time the base case is more complicated:

Base cases: When min(m,n)=1, we factor A as follows:

(1) When A = [ a_11 ] is 1 by 1 (and so a_11 neq 0), pick r=1 and
       write A = 1   * 1 *a_11*  1
               = P_L * L * U  * P_R

(2) When m > n = 1, so A = [a_11] with a_11 neq 0, pick r = 1 and write
                           [a_21]
                           [ ...]
```

```
                          [a_m1]
    A = I^m * [    1    ] * a_11 * 1
              [a_21/a_11]
              [   ...   ]
              [a_m1/a_11]
      = P_L *      L      *  U   * P_R
```

When m > n = 1, with a_11 = 0 but some other a_ii neq 0,
then pick r = 1 and P_L to swap entries 1 and i

ASK & WAIT: What does P_L look like?
and write

```
         A = P_L * [     1      ] * a_ii * 1
                   [ a_21/a_ii  ]
                   [ a_31/a_ii  ]
                   [    ...     ]
                   [ a_11/a_ii  ]  <- i-th location
                   [    ...     ]
                   [ a_m1/a_ii  ]
           = P_L *       L       *   U  * P_R
```

(3) When m = 1 < n, so A = [a_11 a_12 ... a_1m] with a_11 neq 0,
then pick r = 1 and write

```
       A =  1  * 1 * A * I^n
         = P_L * L * U * P_R
```

When m = 1 < n, so A = [a_11 a_21 ... a_m1]
with a_11 = 0 but some other a_ii neq 0
then pick r = 1 and P_R (as described above) to swap entries 1 and i so

```
       A =  1 * 1 * [a_ii a_22 ... a_11 ... a_m1] * P_R
                                   ^^^^
                                 location i

         = P_L* L *                U             * P_R
```

Now for the induction. Let A be m by n and assume Theorem 2 is
true for (m-1) by (n-1) matrices. If A_11 neq 0,
let P'_L = I^m and P'_R = I^n. Otherwise, if some other A_ij is nonzero,
pick P'_L (to swap rows 1 and i) and P'_R (to swap columns 1 and j)
so that P'_L*A*P'_R has a nonzero entry (A_ij) in the (1,1) position.
Then as above write

```
                     1    n-1
  P'_L*A*P'_R = [ A_11 A_12 ]  1
                [ A_21 A_22 ]  n-1
```

```
           = [ 1         0       ] * [ A_11 A_12                   ]
             [ A_21/A_11 I^{n-1} ]   [  0   A_22 - A_21/A_11*A_12 ]

           = [ 1 0       ] * [ A_11 A_12 ]
             [ X I^{n-1} ]   [  0    S   ]
where X = A_21/A_11 and S = A_22 - A_21/A_11*A_12 for short.
There are 2 cases:
  (1) If S = 0, then we can confirm that we can write
        P'_L*A*P'_R = [ 1 0       ] * [ A_11 A_12 ]
                      [ X I^{n-1} ]   [  0    S   ]
                    = [ 1 0       ] * [ A_11 A_12 ]
                      [ X I^{n-1} ]   [  0    0   ]
                    = [ A_11   A_12   ]
                      [ X*A_11 X*A_12 ]
                    = [ 1 ] * [ A_11 A_12 ]   = L * U
                      [ X ]
      Thus A = (P'_L)^{-1} * L * U * (P'_R)^{-1}
             = P_L * L * U * P_R
      where P_L = P'_L^t and P_R = P'_R^t are permutations by Lemma 2 part (5),
      is the desired LU factorization.
  (2) If S is nonzero, we can apply the induction hypothesis to S to
      get S = P_LS * L_S * U_S * P_RS, where L_S (U_S) has r' columns (rows)
      Plugging this in we get
      A = (P'_L)^t * [ 1 0 ] * [ A_11 A_12              ] * (P'_R)^t
                     [ X I ]   [  0   P_LS*L_S*U_S*P_RS ]
            ... substituting for S
        = (P'_L)^t * [ 1 0 ] * [ 1 0        ]  * [ A_11 A_12     ] * (P'_R)^t
                     [ X I ]   [ 0 P_LS*L_S ]    [  0   U_S*P_RS ]
            ... factoring the 3rd matrix.
            ... Note that P_LS*L_S is m-1 by r' and U_S*P_RS is r' by n-1
        = (P'_L)^t * [ 1 0        ] * [ A_11 A_12     ] * (P'_R)^t
                     [ X P_LS*L_S ]   [  0   U_S*P_RS ]
            ... multiplying factors 2 and 3
        = (P'_L)^t * [ 1 0    ] * [ 1          0   ] * [ A_11 A_12     ] * (P'_R)^t
                     [ 0 P_LS ]   [ P_LS^t*X L_S ]   [  0   U_S*P_RS ]
            ... factoring the 2nd matrix
            ... Note that L_S is m-1 by r' and P_LS^t*X is m-1 by 1
        = P_L * [ 1          0   ] * [ A_11 A_12     ] * (P'_R)^t
                [ P_LS^t*X L_S ]   [  0   U_S*P_RS ]
            ... multiplying factors 1 and 2, both permutation matrices,
            ... to get the final permutation matrix P_L, by Lemma 2 part (4)
```

```
        = P_L * [ 1        0   ] * [ A_11 A_12*P_RS^t ] * [ 1 0    ] * (P'_R)^t
                [ P_LS^t*X L_S ]   [  0   U_S         ]   [ 0 P_RS ]
            ... factoring the 3rd matrix
            ... Note that U_S is r' by n-1 and A_12*P_RS^t is 1 by n-1
        = P_L * [ 1        0   ] * [ A_11 A_12*P_RS^t ] * P_R
                [ P_LS^t*X L_S ]   [  0   U_S         ]
            ... multiplying factors 4 and 5, both permutation matrices,
            ... to get the final permutation matrix P_R, by Lemma 2 part (4)
        = P_L * L * U * P_R
            ... the final factorization, by noticing that the 2nd matrix
            ... is lower triangular with 1s on the diagonal as desired,
            ... and the 3rd matrix is upper triangular with nonzeros
            ... on the diagonal, as desired.
            ... Note that L has r = r'+1 columns and U has r rows.


Corollary 2: Let A = P_L * L * U * P_R be the above factorization
of the m by n nonzero matrix A, where L (U) has r columns (rows). Then
   (1) range(A) = span(P_L*L)
   (2) rank(A) = r
   (3) nullity(A) = n-r
   (4) nullspace(A) may be computed as follows.
       Write U = [U1 , U2] where U1 is r by r (and so upper triangular)
       and U2 is r by n-r. Then
       nullspace(A) = span( P_R^t * [ -U1^{-1}*U2 ] )
                                    [    I^{n-r}  ]
   (5) If A is invertible if and only if m = n = r, in which case
       A^{-1} = P_R^t * U^{-1} * L^{-1} * P_L^t
   (6) The complete solution set of Ax = b may be computed from the entries
       of the LU factorization of A (see below).

                  r                    r    n-r
Proof: Write L = [ L1 ] r     and U = [U1 , U2 ] r
                 [ L2 ] m-r
       so that L1 is square and lower triangular
       and U1 is square and upper triangular. Since the diagonal entries
       of L1 and U1 are nonzero, L1 and U1 are invertible, and have rank = r.
       Thus L has nullity(L) = 0 and so rank(L) = r, since
       L*x = 0 => [L1*x] = 0 => L1*x=0 => x=0 => L is one-to-one
                  [L2*x]

       (1): range(A) = {A*x, all x in F^n}
```

```
      = {P_L*L*U*P_R*x, all x in F^n}
      = {P_L*L*U*x, all x in F^n}
         ... since {x: x in F^n} = {P_R*x: x in F^n}
      = {P_L*L*U*[ x1 ], all x1 in F^r and x2 in F^{n-r}}
                 [ x2 ]
      = {P_L*L*(U1*x1 + U2*x2), all x1 in F^r and x2 in F^{n-r}}
         ... by multiplying out
      = {P_L*L*(x1 + U2*x2), all x1 in F^r and x2 in F^{n-r}}
         ... since U1 is invertible, {x1: x1 in F^r} = {U1*x1: x1 in F^r}
      = {P_L*L*x1,  all x1 in F^r}
         ... since F^r = {x1: x1 in F^r}
         ...           = {x1 + U2*x2: x1 in F^r and x2 in F^{n-r}}
      = range(P_L*L)

(2) rank(A) = dim(range(A))
            = dim(range(P_L*L))
            = dim(P_L*range(L))
            = dim(range(L))   ... since P_L is invertible
            = rank(L)
            = r

(3) By the dimension theorem, nullity(A) = #columns(A) - rank(A) = n-r

(4) Ax=0 implies that P_L*L*U*P_R*x = 0, and that U*P_R*x = 0,
    since nullity(L) = 0.  Let P_R*x = [ x1 ]  r
                                       [ x2 ]  n-r
    so that Ax = 0 =>  0 = [U1 U2]*[x1] = U1*x1+U2*x2.
                                   [x2]
    Since U1 is invertible, this means x1 = -U1^{-1}*U2*x2.
    In other words, a vector x is in nullspace(A) iff
    x = P_R^t * [x1] = P_R^t * [-U1^{-1}*U2] * x2, for any x2 in F^{n-r}
                [x2]           [  I^{n-r}  ]
    or x in span( P_R^t * [-U1^{-1}*U2] )
                          [  I^{n-r}  ]
(5) A is invertible iff it is square (m=n) and has rank(A) = r = n.
    In this case A^{-1} = (P_L * L * U * P_R)^{-1}
                        = P_R^{-1} * U^{-1} * L^{-1} * P_L^{-1}
                        = P_R^t * U^{-1} * L^{-1} * P_L^t


(6) We can solve Ax=b if b in range(A) = range(P_L*L), i.e.
    if P_L^t*b in range(L) = range([L1]) or P_L^t*b = [L1]*x' = [L1*x']
```

```
                                [L2])               [L2]      [L2*x']
for some x'. Write P_L^t*b = [b1] r    so b1 = L1*x' and b2 = L2*x'
                             [b2] n-r
Thus x' = L1^{-1}*b1 and b2 = L2*L1^{-1}*b1.
In other words, we can solve Ax=b if and only if b2 = L2*L1^{-1}*b1
where P_L^t * b = [b1]
                  [b2]
If this condition holds, we can find all the solutions as follows.
Suppose there are two solutions, x and y. Then
  A*(x-y) = A*x -A*y = b - b = 0
so x-y must be in nullspace(A). In other words, if we find one
solution x, all the others can be found by taking x plus any member
of nullspace(A) from part (4). To find one solution x, we solve
   x' = U * P_R * x
for x or
   x' = [U1,U2] * P_R * x = [U1,U2] * [x1] = U1*x1 + U2*x2
                                      [x2]
One solution is gotten by setting x2 = 0, and solving x'=U1*x1
for x1 = U1^{-1}*x' = U1^{-1}*L1^{-1}*b1. Thus the
complete solution set of Ax = b, if a solution exists at all, is
{x = P_R^t * [ U1^{-1}*L1^{-1}*b1 ] + z, for all z in nullspace(A) }
             [         0          ]
```

```
Finally, we consider the number of arithmetic operations
required to compute the LU factorization. We note that multiplying
by permutation matrices does not require any arithmetic operations,
only reordering rows (or columns). We assume for simplicity that
A is n by n with rank n (if the rank is lower the number of operations
is lower too). Let C(n) be the "cost", i.e. number of operations. Then
by the proof of Theorem 2
   C(n) = number of operations to compute X = A_21/a11
          + number of operations to compute S = A_22 - X*A_12
          + number of operations to compute the LU factorization of S
        = n-1 divisions
          + (n-1)^2 multiplications to form X*A_12
            + (n-1)^2 substractions to subtract A_22 - X*A_12
          + C(n-1)
This gives us the recurrence
   C(n) = 2*(n-1)^2 + (n-1) + C(n-1)
along with the initial value C(1) = 0, or
   C(n) = sum_{j=1 to n-1} (2*j^2 + j)
        = 2/3*n^3 + "lower order terms"
```

where "lower order terms" means a quadratic function of n.
When n is large, C(n) is well approximated by 2/3*n^3.
It is interesting to note that solving a system of linear
equations costs 2/3*n^3 + lower order terms, a third as
much as multiplying two matrices of the same size, 2*n^3.