Computational Projects

Lectures 5 & 6:
Gaussian Elimination / LU decomposition

Dr Rob Jack, DAMTP

http://www.maths.cam.ac.uk/undergrad/catam/part-ia-lectures

Motivation

So far we considered algorithms that correspond to very short MATLAB programs, eg the main part of Euler's method is simply

```
x(1) = xstart;
y(1) = ystart;

for i=1:n
    yprime = x(i)*y(i)^2;
    y(i+1) = y(i) + h*yprime;
    x(i+1) = x(i) + h;
end
```

If we want to do something more complicated, we need to understand how to build up a longer program, based on simple ingredients.

In the next 2 lectures, we discuss an extended example of this

The problem

Given an $n \times n$ matrix A and an n-vector b (both with real-valued elements), we want to solve $Ax = b$ to obtain $x = A^{-1}b$

If the inverse does not exist then our method should notice and return an error

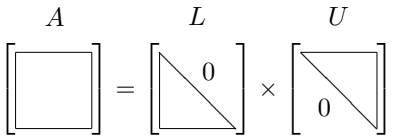
Our method will also work if b is a matrix of size $n \times m$. If we set b to be the identity then we obtain $x = A^{-1}$ (if it exists).

You might remember that this can be done by a method called Gaussian elimination. The method that we use is almost equivalent, it is called LU decomposition.

LU decomposition

Our method is based on a trick, which is to write $A = LU$

. . . where L is a lower triangular matrix and U is upper triangular



$$L_{ij} = 0 \text{ for } i > j$$
$$U_{ij} = 0 \text{ for } i < j$$

We assume (for now) that this is possible. In fact we assume that it is possible with $L_{ii} = 1$ for all i.

. . . cases where this is not possible are discussed later.

Solving Ax = b

Suppose $Ax = LUx = b$. Let $y = Ux$. Then $Ly = b$.

It turns out to be easy to invert triangular matrices. So if we know L and U then we can obtain y as $L^{-1}b$ and then x as $U^{-1}y$.

Plan for writing our program to solve $Ax = b$.

1. Assume that L, U, b are given and write functions for obtaining $y = L^{-1}b$ and $x = U^{-1}y$.
2. Assume that A is given and write a function for obtaining L and U.
3. Deal with cases where A can't be written as LU ("pivoting")

General plan: split a big task into pieces and deal with them one at a time. It's good if the pieces correspond to MATLAB functions

Algorithm for inverting L and U

$Ly = b$ means that $\begin{pmatrix} L_{11} & 0 & \cdots & \cdots & 0 \\ L_{21} & L_{22} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ L_{n-1,1} & \ddots & \ddots & L_{n-1,n-1} & 0 \\ L_{n1} & \cdots & \cdots & \cdots & L_{nn} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{pmatrix}$

First row: $y_1 = b_1/L_{11}$

Second row: $y_2 = (b_2 - L_{21}y_1)/L_{22}$

. . . since we already know y_1 we can compute y_2

kth row: $y_k = \left(b_k - \sum_{j=1}^{k-1} L_{kj}y_j\right)/L_{kk}$

. . . we already know $y_1 \ldots y_{k-1}$ so can compute y_k

We can use a loop to compute $y_1, y_2, \ldots, y_n$

[if $L_{kk} = 0$ for some k then L^{-1} does not exist and this (usually) fails]

MATLAB function

$Ly = b$ $\qquad$ $y_k = \frac{1}{L_{kk}}\left[b_k - \sum_{j=1}^{k-1} L_{kj}y_j\right]$

```
function [ y ] = Lsolve( L,b )
%Lsolve: solve Ly = b (for y) where L is lower triangular and b is a vector
% note: it is not checked that L is actually lower triangular
    [bRows,bCols] = size(b);
    [LRows,LCols] = size(L);

    if LRows ~= LCols || bRows ~= LRows || bCols ~= 1
        error('Either L or b is the wrong size.')
    end
    if any( diag(L) == 0 )
        error('There are zeros on the diagonal of L')
    end

    y = b;
    for k = 1:LRows
      for j = 1:k-1
        y(k) = y(k) - L(k,j)*y(j);
      end
      y(k) = y(k)/L(k,k);
    end
end
```

Example: `Lsolve.m`

Testing

A key advantage of breaking up the complex task into functions is that we can *test each function separately*

Tests for the various functions that we write here are given in `LUtest.m`

It's a good idea to test some easy cases, but also to check what happens in nasty cases (eg if L and b don't have the right size, or L is a singular matrix).

Similar algorithm for U

You might imagine that there is a similar method that works for U...

We solve $Ux = y$ (for x)

We consider the rows in turn, but now starting from the *last* (nth) row

Last row: $U_{nn}x_n = y_n$

kth row: $U_{kk}x_k = y_k - \sum_{j=k+1}^{n} U_{kj}x_j$

...we know x_j for $j > k$ so we can compute x_k.

Again, use a loop to compute $x_1, x_2, \ldots, x_n$

Matlab translation

$Ux = y$ $\qquad$ $x_k = y_k - \sum_{j=k+1}^{n} U_{kj}x_j$

```
function [ x ] = Usolve( U,y )
%Usolve: solve Ux = y (for x) where U is upper triangular
% note: it is not checked that U is actually upper triangular
    [yRows,yCols] = size(y);
    [URows,UCols] = size(U);

    if URows ~= UCols || yRows ~= URows || yCols ~= 1
        error('Either L or b is the wrong size.')
    end
    if any( diag(U) ==  0 )
        error('There are zeros on the diagonal of U')
    end

    n = URows;
    x = y;
    for k = n:-1:1  % loop downwards from n to 1
      for j = k+1:n
        x(k) = x(k) - U(k,j)*x(j);
      end
      x(k) = x(k)/U(k,k);
    end
end
```

Example: `Usolve.m`

Combining ingredients

$LUx = b$ $\qquad$ Write $y = Ux$ so $x = U^{-1}y$ and $y = L^{-1}b$

...we have done the hard part so this is easy now(!)...

```
function [ x ] = LUsolve( L,U,b )
%LUsolve: solve LUx=b where U is upper triag and L is lower triag
  y = Lsolve(L,b);
  x = Usolve(U,y);
end
```

Example: `LUsolve.m`

...this is the "philosophy of structured programming"

...break the task into functions and write each one separately, then stick them together at the end

We will see later how to improve our `Lsolve` and `Usolve` functions, but the first job is usually to get something simple that works

LU decomposition

Given an $n \times n$ matrix A, we want to find upper and lower triangular matrices such that $A = LU$, with $L_{ii} = 1$ for all i

Assume that this is possible: $A_{ij} = \sum_{k=1}^{n} L_{ik}U_{kj}$

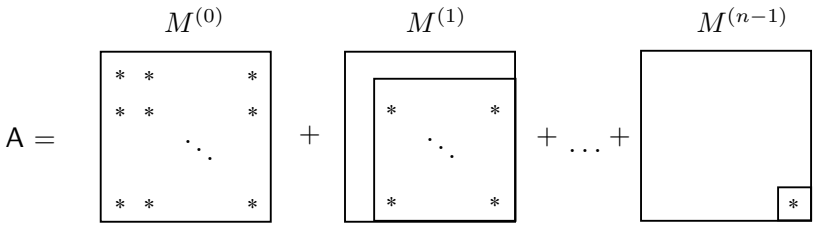
For $k = 0, 1, 2, \ldots, n-1$, define rank-one matrices $M^{(k)}$ with elements

$$M_{ij}^{(k)} = L_{i,k+1}U_{k+1,j}$$

The first k rows of $M^{(k)}$ are full of zeros, as are the first k columns (by the triangular structure of L and U)

Define also $A^{(k)} = A - \sum_{r=0}^{k-1} M^{(r)}$, with $A^{(0)} = A$; also $A^{(n)} = 0$

LU decomposition



$*$ indicates a non-zero element

The first row of $M^{(0)}$ is equal to the first row of A, and similarly for the first column

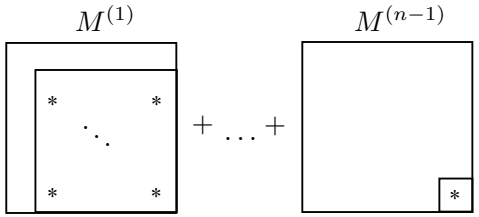
Since $L_{11} = 1$, the first row of $M^{(0)}$ is the first row of U
(since $M^{(0)}_{1j} = L_{11}U_{1j}$)

Similarly the first column of L is U_{11} times the first column of $M^{(0)}$, and U_{11} has already been computed.

LU decomposition

From previous slide, we know the first column of L and the first row of U. Hence we can compute all elements of $M^{(0)}$ (as $M^{(0)}_{ij} = L_{i1}U_{1j}$)

$$A^{(1)} = A - M^{(0)} =$$



$A^{(1)}$ is a known matrix, and its second row is the same as the second row of $M^{(1)}$... and similarly the second column.

Similar to previous slide, the second row of $M^{(1)}$ is the second row of U, and we can also compute the second column of L

LU algorithm - summary

In this way, we can work out all elements of L and U:

Let $A^{(0)} = A$

Iterate $k = 1, 2, \ldots, n$

kth row of U: $U_{kj} = A^{(k-1)}_{kj}$ $\quad j = k, \ldots, n$

kth column of L: $L_{ik} = A^{(k-1)}_{ik} / A^{(k-1)}_{kk}$ $\quad i = k, \ldots, n$

subtract $M^{(k-1)}$: $A^{(k)}_{ij} = A^{(k-1)}_{ij} - L_{ik}U_{kj}$ $\quad i, j \geq k$

What can go wrong?

This all works (and the decomposition exists) if $A^{(k-1)}_{kk} \neq 0$ for all k, otherwise it fails (see later)

Matlab

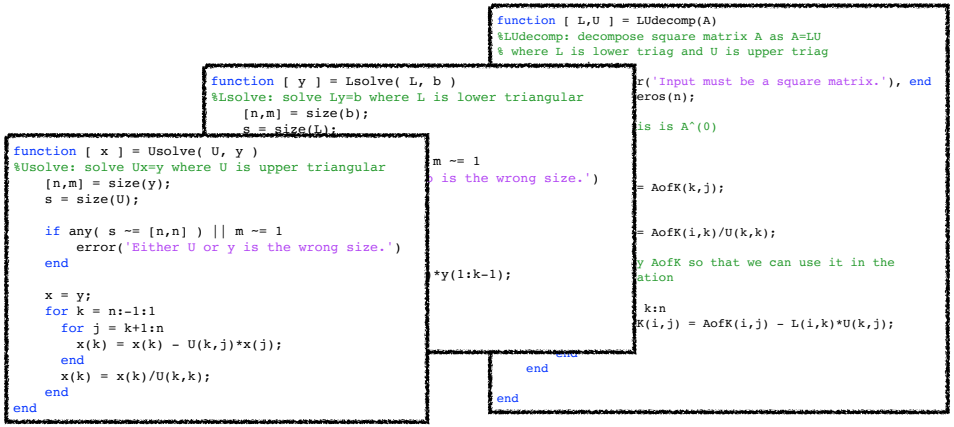
```
function [ L,U ] = LUdecomp(A)
%LUdecomp: decompose square matrix A as A=LU
% where L is lower triag and U is upper triag
    [m, n]=size(A);
    if m ~= n, error('Input must be a square matrix.'), end
    L=zeros(n); U=zeros(n);

    % remember A^(0) is A
    AofK = A;
    for k = 1:n
        % at this point AofK is A^(k-1)
        for j = k:n
            U(k,j) = AofK(k,j);
        end
        % check that we don't divide by zero...
        if U(k,k) == 0
            error('** A^(k-1)_{k,k}==0 in LU decomp')
        end
        for i = k:n
           L(i,k) = AofK(i,k)/U(k,k);
        end
        % now modify AofK so that we can use it in the
        % next iteration
        for i = k:n
            for j = k:n
                AofK(i,j) = AofK(i,j) - L(i,k)*U(k,j);
            end
        end
    end
end
```

Example: `LUdecomp.m`

All together

```
function [ x ] = Asolve( A,b )
%Asolve: solve Ax=b by LU decomposition
  [L,U] = LUdecomp(A)
  y = Lsolve(L,b);
  x = Usolve(U,y);
end
```



Complexity

LU decomposition: we loop over $k = 1, 2, \ldots n$ and for each k we need to compute the elements $A_{ij}^{(k)}$, which requires $(n-k)^2$ multiplication operations

The total number of multiplications in this step is $\sum_{k=1}^{n}(n-k)^2 = O(n^3)$, the other steps in the algorithm are $O(n^2)$

Given L and U, **solving for** x is easily checked to be $O(n^2)$.

The complexity of solving $Ax = b$ by this method is $O(n^3)$, and the dominant cost is the LU decomposition.

Tests...

see `LUtest.m`

Improvements

Once we have a method that works, we can think about how to improve it

Eg, currently we can solve $Ax = b$ where b is a vector, but it should be easy to generalise to $n \times m$ matrices b.

We only need to modify `Lsolve` and `Usolve`

Eg, to solve $Ly = b$ we must now compute

$$y_{km} = \frac{1}{L_{kk}}\left[b_{km} - \sum_{j=1}^{k-1} L_{kj} y_{jm}\right]$$

Improvements

Option 1 : use an extra loop to deal with the columns of b

(main part of `Lsolve`, original version)

```
y = b;
for k = 1:Lrows
  for j = 1:k-1
    y(k) = y(k) - L(k,j)*y(j);
  end
  y(k) = y(k)/L(k,k);
end
```

(improved)

```
y = b;
for m = 1:bCols
  for k = 1:Lrows
    for j = 1:k-1
      y(k,m) = y(k,m) - L(k,j)*y(j,m);
    end
    y(k,m) = y(k,m)/L(k,k);
  end
end
```

Note the complexity of this part is now $O(n^2 m)$ where m is the number of columns in b. We expect $m \leq n$ so the LU decomposition is likely still to be the dominant cost.

Improvements

Option 2 : use the fact that MATLAB can deal with whole rows in a simple way (this is sometimes called an "implicit loop")

(original)

```
y = b;
for k = 1:Lrows
  for j = 1:k-1
    y(k) = y(k) - L(k,j)*y(j);
  end
  y(k) = y(k)/L(k,k);
end
```

(improved)

```
y = b;
for k = 1:Lrows
  for j = 1:k-1
    y(k,:) = y(k,:) - L(k,j)*y(j,:);
  end
  y(k,:) = y(k,:)/L(k,k);
end
```

Option 3 (only for the brave...): use another implicit loop

(improved)

```
y = b;
for k = 1:Lrows
  y(k,:) = y(k,:) - L(k, 1:k-1 ) * y( 1:k-1, : );
  y(k,:) = y(k,:)/L(k,k);
end
```

next steps...

We coded up a method for solving $Ax = b$ by writing $A = LU$ where L and U are triangular matrices

Suppose we want to solve $Ax = b$ with

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad b = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Clearly $x = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ but our method can't solve this, because it is not possible to write $A = LU$

Can we implement a method that works for all non-singular matrices?

General idea: we started with a simple algorithm, now we (gradually) improve it

Beyond LU

The solution to our problem is to write

$$A = P^{-1}LU$$

where P is a *permutation matrix*

A permutation matrix is a square matrix where each row and each column contain exactly one '1' and all other entries are zero. They are easy to invert because $P^{-1} = P^T$.

This means that $PA = LU$, where PA is the same as A, but with the rows in a different order.

If A is non-singular then the decomposition $A = P^{-1}LU$ always exists

Beyond LU

If we can obtain $A = P^{-1}LU$ then we can write $Ax = b$ as $LUx = Pb$, which is easy to solve by our original method.

Example:

$$A = \begin{pmatrix} 0 & u \\ v & 1 \end{pmatrix}$$

has no LU decomposition, but

$$PA = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & u \\ v & 1 \end{pmatrix} = \begin{pmatrix} v & 1 \\ 0 & u \end{pmatrix}$$

has a (trivial) LU decomposition where L is the identity

This is called *pivoting*, but how do we find a suitable P?

Does this work?

To show that this method is valid requires some effort
There is some discussion in IB numerical analysis...
... the following 3 slides have an overview

Our main concern here is how to do the programming, not the proof that the algorithm is valid

LU algorithm with pivot

Let $A^{(0)} = A$. Let $P = I$ (identity). Set $L = 0$ and $U = 0$.

Iterate the following for $k = 1, 2, \ldots n$

Find the row r_k of $A^{(k-1)}$ that maximises $|A^{(k-1)}_{r_k,k}|$. (new steps)
Let $P^{(k-1)}$ be the permutation matrix that swaps row k with row r_k.
(We always have $r_k \geq k$, if $r_k = k$ then $P^{(k-1)} = I$.)

Replace $A^{(k-1)}$ by $P^{(k-1)}A^{(k-1)}$; replace L by $P^{(k-1)}L$; replace P by $P^{(k-1)}P$.
This ensures that $A^{(k-1)}_{kk} \neq 0$, except if the k column of $A^{(k-1)}$ is all zeros.
(In fact it ensures $A^{(k-1)}_{kk} \neq 0$ whenever A is non-singular.)

Let $U_{kj} = A^{(k-1)}_{kj}$ for $j = k \ldots n$.
Let $L_{ik} = A^{(k-1)}_{ik}/A^{(k-1)}_{kk}$ for $j = k \ldots n$.
Let $A^{(k)}_{ij} = A^{(k-1)}_{ij} - L_{ik}U_{kj}$ for $i, j = k \ldots n$. (same as regular LU)

As long as A is non-singular, we have finally $PA = LU$ where P is a permutation, L is lower triangular, and U is upper triangular.

mathematical formulation of LU algorithm

Let $A^{(0)} = A$. We compute matrices $A^{(k)}$ with elements

$$A^{(k)}_{ij} = A^{(k-1)}_{ij} - \frac{A^{(k-1)}_{ik}A^{(k-1)}_{kj}}{A^{(k-1)}_{kk}}$$

Then the first k rows of $A^{(k)}$ are all zeros, as are the first k columns. Compute L and U with elements

$$L_{ik} = A^{(k-1)}_{ik}/A^{(k-1)}_{kk}, \qquad U_{kj} = A^{(k-1)}_{kj}.$$

Then $A = LU$, where L is lower triangular and U is upper triangular.

To see that $A = LU$ note that $A^{(n)} = 0$ (all rows are zero) so that

$$\begin{aligned} 0 = A^{(n)}_{ij} &= A^{(n-1)}_{ij} - \frac{A^{(n-1)}_{in}A^{(n-1)}_{nj}}{A^{(n-1)}_{nn}} \\ &= A^{(n-2)}_{ij} - \frac{A^{(n-2)}_{i,n-1}A^{(n-2)}_{n-1,j}}{A^{(n-2)}_{n-1,n-1}} - \frac{A^{(n-1)}_{in}A^{(n-1)}_{nj}}{A^{(n-1)}_{nn}} \\ &= \cdots \\ &= A^{(0)}_{ij} - \sum_{k=1}^{n} \frac{A^{(k-1)}_{ik}A^{(k-1)}_{kj}}{A^{(k-1)}_{kk}} \end{aligned}$$

Use the formula for $A^{(k)}$, n times

Hence $A^{(0)}_{ij} = \sum_{k=1}^{n} L_{ik}U_{kj}$ so $A = LU$

LU **with pivot**

Notation: if A, B are matrices then $(AB)_{ij}$ indicates element ij of the matrix AB.

For $k = 1, 2, \ldots n$, let $P^{(k-1)}$ be a permutation matrix that swaps row k with some row $r_k \geq k$.
Let $A^{(0)} = P^{(0)}A$. Also let $P^{(n)} = I$, the identity.
For $1, 2, \ldots n$, compute matrices $A^{(k)}$ with elements

$$A^{(k)}_{ij} = (P^{(k)}A^{(k-1)})_{ij} - \frac{(P^{(k)}A^{(k-1)})_{ik}A^{(k-1)}_{kj}}{A^{(k-1)}_{kk}}$$

Then the first k rows of $A^{(k)}$ are all zeros, as are the first k columns. Compute L and U with elements

$$L_{ik} = \left(P^{(n-1)}P^{(n-2)}\cdots P^{(k)}A^{(k-1)}\right)_{ik} \times \frac{1}{A^{(k-1)}_{kk}}, \qquad U_{kj} = A^{(k-1)}_{kj}.$$

Finally, let $P = P^{(n-1)}P^{(n-2)}\cdots P^{(0)}$. Then $PA = LU$, also L is lower triangular and U is upper triangular.

(see next slide)

This method is valid for any choice of the row indices $r_1, r_2, \ldots, r_n$, as long as $r_k \geq k$, and $A^{(k-1)}_{kk} \neq 0$ for all k. We can choose r_k at the same time as we compute $A^{(k-1)}$, aiming to ensure that $A^{(k-1)}_{kk} \neq 0$.

LU **with pivot -- check**

To check that $PA = LU$, follow the same method as before: note that $A^{(n)} = 0$ (all rows are zero), and use our formula for $A^{(k)}$ repeatedly: then

$$\begin{aligned}
0 = A^{(n)}_{ij} &= \left(P^{(n)}A^{(n-1)}\right)_{ij} - \frac{(P^{(n)}A^{(n-1)})_{in}A^{(n-1)}_{nj}}{A^{(n-1)}_{nn}} \\
&= \left(P^{(n)}P^{(n-1)}A^{(n-2)}\right)_{ij} - \frac{\left(P^{(n)}P^{(n-1)}A^{(n-2)}\right)_{i,n-1}A^{(n-2)}_{n-1,j}}{A^{(n-2)}_{n-1,n-1}} - \frac{(P^{(n)}A^{(n-1)})_{in}A^{(n-1)}_{nj}}{A^{(n-1)}_{nn}} \\
&= \cdots \\
&= \left(P^{(n)}P^{(n-1)}\cdots P^{(0)}A\right)_{ij} - \sum_{k=1}^{n} \frac{\left(P^{(n)}P^{(n-1)}P^{(n-2)}\cdots P^{(k)}A^{(k-1)}\right)_{ik}A^{(k-1)}_{kj}}{A^{(k-1)}_{kk}}
\end{aligned}$$

Hence from the formulae for L, U, P, we have $(PA)_{ij} = \sum_{k=1}^{n} L_{ik}U_{kj}$ so $PA = LU$

We should also check that the first k rows (and columns) of $A^{(k)}$ are always zero. This can be done, it relies on the fact that row k of $P^{(k)}A^{(k-1)}$ is the same as row k of $A^{(k-1)}$.

... writing the program

The algorithm is not too simple, can we write the program?

Break up the problem into manageable pieces...

- Write a function to swap two rows of a matrix
  (this is the same as multiplying from the left by some P)
- Write a function that finds the largest element in a given column, and outputs the row in which that element appears.
  (this is needed to work out r_k)

Test these functions carefully before combining them into our program

... finally

```
function [ P,L,U ] = PALUdecomp( A )
%PALUdecomp decompose A = P^{-1}LU
    [m, n]=size(A);
    if m ~= n, error('Input must be a square matrix.'), end
    P=eye(n); L=zeros(n); U=zeros(n);

    % remember A^(0) is A
    AofK = A;
    for k = 1:n

        % this is the new part, do the pivot...
        rk = findLargestinCol(AofK,k);
        if rk ~= k
            AofK = swapRows( AofK, rk, k );
            L = swapRows( L, rk, k );
            P = swapRows( P, rk, k );
        end

        % from here it is the same as the OLD algorithm
        for j = k:n
            U(k,j) = AofK(k,j);
        end
        % check that we don't divide by zero(!)
        if U(k,k) == 0
            error('** A^(k-1)_{k,k}==0 in PALU decomp')
        end
        for i = k:n
           L(i,k) = AofK(i,k)/U(k,k);
        end
        for i = k:n
            for j = k:n
                AofK(i,j) = AofK(i,j) - L(i,k)*U(k,j);
            end
        end
    end  % of the loop over k

end % of the function
```

... the only new part...

```
rk = findLargestinCol(AofK,k);
if rk ~= k
    AofK = swapRows( AofK, rk, k );
    L = swapRows( L, rk, k );
    P = swapRows( P, rk, k );
end
```

Example: `PALUdecomp.m`, also `pivotTest.m`

... finally

```
function [ rk ] = findLargestinCol( A,k )
% findLargestinCol: find the row index of the large element in column k
%    (of some matrix A)
  [value,index] = max( abs( A(:,k) ) );  % max is built-in for matlab
                                         % don't forget to take absolute
                                         % value
  rk = index;
end
```

```
function [ A ] = swapRows( A,u,v )
%swapRows: swap rows u and v of matrix A and send the answer as output
  storeMe = A( v, : );     % store row v of A
  A( v , : ) = A( u, : );  % copy row u of A into row v of A
  A( u , : ) = storeMe;    % copy the stored row into row u of A
end
```

`findLargestinCol.m , swapRows.m`

round-off (again)

The problematic part of (normal) LU decomposition was

$$A_{ij}^{(k)} = A_{ij}^{(k-1)} - \frac{A_{ik}^{(k-1)} A_{kj}^{(k-1)}}{A_{kk}^{(k-1)}}$$

when $A_{kk}^{(k-1)} = 0$.

If our matrices are not specially constructed to be "nasty" we might expect this to be rare...

...but if $A_{kk}^{(k-1)}$ is a small number, we can expect large numbers to start appearing in our matrices...

...this is not so rare, and it can cause round-off problems when we start taking differences of these large numbers...

`pivotTest.m`

"More general" algorithm (with pivot) is also more accurate...

further improvement...

Now we have a working program, we can solve equations...

However, there are still a few things that we can think about, especially if we want to work with large matrices

Can we make our program more flexible?

(eg can we get solutions where b is *n* x *m*? ... see earlier)

Can we make our program run faster?

(eg can we reduce the total number of operations?)

also, does our program use up a lot of memory?

"Efficiency"

Our current algorithm keeps track of a permutation matrix P.

This is an $n \times n$ matrix in which there are n ones and all other elements are zero.

...perhaps it would be more efficient to just keep track of the ones, and let the zeros look after themselves...

If we do this, we can store all information about the matrix by keeping track of n integers, instead of n^2 real numbers.

(Eg, for row j, keep keep track of the position x_j of the "one" that appears in that row.)

compact version

```
% main body of PALUdecompV2

L=zeros(n); U=zeros(n);
P = 1:n;  % P starts as a vector with elements 1,2,... n

% remember A^(0) is A
AofK = A;
for k = 1:n

    % this is the new part, do the pivot...
    rk = findLargestinCol(AofK,k);
    if rk ~= k
        AofK = swapRows( AofK, rk, k );
        L = swapRows( L, rk, k );
        % swap two elements of P (instead of swapping rows)
        storeMe = P(rk);
        P(rk) = P(k);
        P(k) = storeMe;
    end

    % from here it is the same as the OLD algorithm
    [snipped to save space]

end  % of the loop over k
```

Example: `PALUdecompV2.m`

compact version

. . . P is not stored as a matrix, how do I compute (for example) PA?

```
function [ A ] = PTimesMat( P,M )
% compute P.M where P is a "permutation vector" and M is a matrix

    [Mrows,Mcols] = size(M);
    A = zeros(Mrows,Mcols);

    % should check here that P is vector of size Mrows

    for i=1:Mrows
        A(i,:) = M( P(i),: );
    end
end
```

. . . a further advantage is that this “matrix multiplication” is now an $O(n^2)$ operation, instead of $O(n^3)$

Lesson from P matrix

If we think carefully about the "information content" of our data, we can reduce the amount of data that we store

This can also help to reduce the number of operations in our computation, although our code might be a bit more complicated

In addition, the computer has a finite amount of memory: by storing less data, we reduce the demand on resources, this helps when solving large problems...

Saving more memory

For LU decomposition, our final output is an upper triangular matrix and a lower triangular matrix (with 1s on the diagonal)

You can see that there are only n^2 non-trivial numbers that we have to compute. . . in fact we scan store all of these in one $n \times n$ matrix, instead of two

It is possible to do LU decomposition (with pivot) in which we end up with a final matrix $A^{(n)}$ whose elements are those of L and U (instead of zeros)

If you really care about speed and memory usage, this is a good idea. But always remember: if you just want something that works, the simplest method can still be best. . .

Memory -- one last thing

MATLAB is very good at dealing with vectors, which we can use to make lists

However, dealing with very large lists (millions or billions of entries) can be slow

It's good to ask: do I really need to store every element in the list?

Eg, for ODE solving, we computed and stored the whole sequence of y_n. But if the step h is small then for any practical purpose (eg plotting a graph), we can just as well store every 10th point, or every 100th

If you keep this in mind, it will help to avoid "bad programming habits"

... today

we showed how to build up a reasonably complicated algorithm (LU decomposition with pivot)

started to think about what makes a "good program"

... next lecture

programs that use random numbers
(in particular for computing high-dimensional integrals)