Computational Projects

Lecture 5:
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

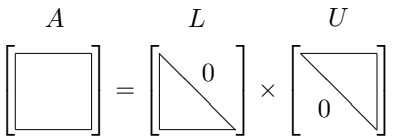
Our method will also work if b is a matrix of size $n \times m$. If we set b to be the identity then we obtain $x = A^{-1}$ (if it exists).

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: $A_{ij} = \sum_{k=1}^{n} L_{ik}U_{kj}$

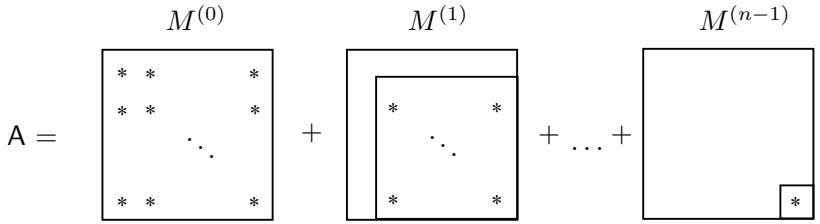
For $k = 0, 1, 2, \ldots, n-1$, define rank-one matrices $M^{(k)}$ with elements

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

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

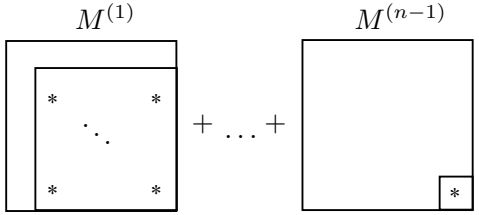
Since $L_{11} = 1$, the first row of $M^{(0)}$ is the first row of U
(since $M^{(0)}_{1j} = L_{11}U_{1j}$)

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
```

```
function [ L,U ] = LUdecomp(A)
%LUdecomp: decompose square matrix A as A=LU
% where L is lower triag and U is upper triag
r('Input must be a square matrix.'), end
eros(n);
is is A^(0)
= AofK(k,j);
= AofK(i,k)/U(k,k);
y AofK so that we can use it in the
ation
 k:n
K(i,j) = AofK(i,j) - L(i,k)*U(k,j);
    end
end
```

```
function [ y ] = Lsolve( L, b )
%Lsolve: solve Ly=b where L is lower triangular
    [n,m] = size(b);
    s = size(L);
m ~= 1
 is the wrong size.')
*y(1:k-1);
```

```
function [ x ] = Usolve( U, y )
%Usolve: solve Ux=y where U is upper triangular
    [n,m] = size(y);
    s = size(U);

    if any( s ~= [n,n] ) || m ~= 1
        error('Either U or y is the wrong size.')
    end

    x = y;
    for k = n:-1:1
      for j = k+1:n
        x(k) = x(k) - U(k,j)*x(j);
      end
      x(k) = x(k)/U(k,k);
    end
end
```


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 lecture

What can we do about $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} x = b$?

What other improvements might be useful to consider?