

**Lab Exercise No. 6 : LU Decomposition and Matrix Powers**

OUT: Wed 15 Nov 2006

IN : Wed 22 Nov 2006

---

## 6.1 Purpose

The purpose of this exercise is to (1) gain more experience with MATLAB, matrices, and *LU* decomposition, (2) to show that apparently trivial computational tasks are often not, and (3) that many matrix problems are best done from a **decompositional viewpoint**.

## 6.2 Solving $A^k x = b$

Here are three different algorithms to solve  $A^k x = b$ .

**Algorithm** SolvePow1( $A, b, k$ )

1. Calculate  $R := \underbrace{A \times A \times \dots \times A}_{k \text{ times}}$
  2. Solve  $Rx = b$  for  $x$ .
  3. **return**  $x$
- endalg** SolvePow1

**Algorithm** SolvePow2( $A, b, k$ )

1. If  $k = 2^p$  there is a better way to calculate  $A^k$  :  
$$A^2 = A \times A, A^4 = A^2 \times A^2, \dots, A^{2^p} = A^{2^{p-1}} \times A^{2^{p-1}} \quad \text{and} \quad R := A^{2^p}$$
  2. Solve  $Rx = b$  for  $x$ .
  3. **return**  $x$
- endalg** SolvePow2

This algorithm is from Golub and Van Loan, page 121, who do not give a derivation of it.

<p><b>Algorithm</b> SolvePow3(<math>A, b, k</math>)</p> <hr/> <ol style="list-style-type: none"><li>1. Calculate <math>[L, U, P] := \text{LUdecomp}(A)</math></li><li>2. <b>for</b> <math>m := 1</math> <b>to</b> <math>k</math> <b>do</b><ol style="list-style-type: none"><li>2.1 Solve <math>Ly = Pb</math> for <math>y</math>.</li><li>2.2 Solve <math>Ux = y</math> for <math>x</math>.</li><li>2.3 Set <math>b := x</math></li></ol></li><li><b>endfor</b></li><li>3. <b>return</b> <math>x</math></li></ol> <p><b>endalg</b> SolvePow3</p>
---

### 6.3 Exercise

1. Implement each algorithm as a MATLAB function.
2. Test and time each function on random matrices of size  $n = 10^1, 10^2, 10^3$  and  $k = 2^2, 2^3, 2^4$ .
3. Analyse each algorithm and derive  $T(n, k)$ , the number of ops required as a function of  $n$  and  $k$ . Compare with your timing results.

**Note.** The MATLAB code for all three algorithms, rolled into one function, took me about 15 short lines of code, excluding initialization and plotting.

**Question 1.** How can SolvePow2( $A, b, k$ ) be modified to handle matrices where  $n$  is not a power of 2?

**Question 2.** How does MATLAB do matrix integer powers, i.e.,  $A^k$ ? Does it use the method in the first algorithm or the second?

**Question 3.** In MATLAB, what is the difference between  $A^k$  and  $A.^k$ ? When you find the answer to this question, please write to O-MATRIX and tell them what the difference is.

## 6.4 Solution

Here are the MATLAB implementations of these functions.

```
function [T1,T2,T3,TM,x1,x2,x3,x4] = LabExer606(Name);
% Lab Exer 6 Solving  $A^k x = b$ 
% Derek O'Connor, Nov 2006
now = date;
titlestr = ['Lab Exer 6 : ' Name now];
disp(' ')
disp('=====')
disp(titlestr);
disp('=====')
kvals = [1 2 4 8 16];
nvals = [100 250 500 750 1000];
T1 = zeros(length(nvals),length(kvals));
T2 = zeros(length(nvals),length(kvals));
T2 = zeros(length(nvals),length(kvals));
TM = zeros(length(nvals),length(kvals));
for nn = 1:length(nvals)
    n = nvals(nn);
    A = randn(n,n)+10^10*eye(n,n);
    x = ones(n,1);
    b = A*x;
    for kk = 1:length(kvals)
        k = kvals(kk);
        % --- Algorithm 1. -----
        tic;
        R = A;
        for m = 2:k
            R = A*R;
        end;
        x1 = R\b;
        T1(nn,kk) = toc;
        % --- Algorithm 2 -----
        tic;
        R = A;
        for m = 1:log2(k)
            R = R*R;
        end
        x2 = R\b;
        T2(nn,kk) = toc;
        % --- Algorithm 3 -----
        tic;
        [L,U,P] = lu(A);
        for m = 1:k
            y = L\(P*b);
            x3 = U\y;
            b = x3;
        end;
        T3(nn,kk) = toc;
        % --- Algorithm M -----
        tic;
        R = A^k;
        x4 = R\b;
        TM(nn,kk) = toc;
    end; % for kk
end; % for nn
subplot(2,2,1);plot(nvals,T1);
subplot(2,2,2);plot(nvals,T2);
subplot(2,2,3);plot(nvals,T3);
subplot(2,2,4);plot(nvals,TM);
```

Table 6.1: Time(secs) to solve  $A^k x = b$

Algorithm	$n$	$k = 2^0$	$k = 2^1$	$k = 2^2$	$k = 2^3$	$k = 2^4$
1	100	0.000	0.000	0.015	0.031	0.063
	250	0.047	0.281	0.203	0.453	0.890
	500	0.281	0.750	1.672	3.531	7.235
	750	0.812	2.328	5.391	11.515	23.765
	1000	1.750	5.375	12.656	27.266	56.375
2	100	0.015	0.000	0.016	0.015	0.015
	250	0.047	0.188	0.156	0.218	0.282
	500	0.297	0.734	1.219	1.672	2.140
	750	0.813	2.328	3.875	5.391	6.922
	1000	1.719	5.344	8.984	12.609	16.281
3	100	0.000	0.016	0.000	0.000	0.000
	250	0.047	0.062	0.063	0.063	0.109
	500	0.265	0.297	0.328	0.406	0.547
	750	0.797	0.844	0.906	1.063	1.375
	1000	1.719	1.781	1.938	2.219	2.765
MATLAB	100	0.000	0.000	0.016	0.219	0.016
	250	0.109	0.313	0.219	0.265	0.344
	500	0.782	1.234	1.719	2.203	2.656
	750	2.390	3.906	5.438	6.953	8.485
	1000	5.468	9.110	12.703	16.344	20.000

Dell Precision 620, 800 MHz.

### 6.4.1 Analysis

- SolvePow1 requires  $k - 1$  matrix multiplications or  $O(n^3 k)$  ops. This is followed by an  $O(n^3)$  solve. Total  $O(n^3 k)$ .
- SolvePow2 requires  $\log_2 k$  matrix multiplications or  $O(n^3 \log_2 k)$  ops. This is followed by an  $O(n^3)$  solve. Total  $O(n^3 \log_2 k)$ .
- SolvePow3 requires an  $O(n^3)$  LU decomposition of  $A$ . This followed by  $k$  Forward and Back Substitutions at a cost of  $O(n^2 k)$ . Total  $O(n^3 + n^2 k)$ .
- MATLAB was implemented as  $R = A^k$ ;  $x = R \backslash b$  and if it uses repeated squaring for  $A^k$  then it is the same as SolvePow2. Total  $O(n^3 \log_2 k)$ .

Golub & Van Loan's SolvePow3 is truly clever and beats the others by a factor of  $\log_2 k$  because its complexity,  $O(n^3 + n^2 k) \approx O(n^3)$  for any reasonable  $k$ . It takes only 2.765 secs for  $n = 1000$  and  $k = 16$ , whereas SolvePow1 is 20 times slower.

Golub & Van Loan's method uses LU decomposition on the original matrix at a cost of  $O(n^3)$ . Solving  $A^k x = b$  is then replaced by solving  $(LU)^k x = b$ ,<sup>1</sup> which can be done by repeated Forward and

<sup>1</sup>We ignore the  $P$  matrix for simplicity.

Backward substitution, or

$$\begin{aligned} \underbrace{LULU \cdots LULU}_k x &= b \\ \underbrace{LULU \cdots LULU}_{k-1} x &= U^{-1}L^{-1}b \\ &\vdots \\ LUx &= \underbrace{U^{-1}L^{-1} \cdots U^{-1}L^{-1}}_{k-1} b \\ x &= \underbrace{U^{-1}L^{-1} \cdots U^{-1}L^{-1}}_k b \end{aligned}$$

Thus we see that once we have the LU decomposition, we perform a sequence of  $k$  Forward and Back substitutions, each at a low cost of  $O(n^2)$ .

### 6.4.2 The Main Point

The main point of this exercise was to emphasize that thinking about and solving numerical linear algebra or matrix problems is best done from a **decompositional viewpoint**. In this case we use *LU Decomposition* and solve  $(LU)^k x = b$  rather than solving  $A^k x = b$  directly. Other linear algebra problems use different decompositions. For example, the general linear least squares problem is best solved using  $A = QR$  factorization.

The **worst way** of thinking about and solving linear algebra or matrix problems is **using the inverse**. As children we are taught that the solution to  $Ax = b$  is  $x = A^{-1}b$ . This true but trivial fact is computationally useless.

### 6.5 Answers to the Questions.

The following are fairly extended answers. I did not expect you to go into such detail.

**Question 1.** How can  $\text{SolvePow2}(A, b, k)$  be modified to handle matrices where  $n$  is not a power of 2?

**Solution 1.** We can generalize the repeated squaring recurrence as follows

$$A^k = \begin{cases} A & \text{if } k = 1, \\ A^{k/2} * A^{k/2}, & \text{if } k \text{ is even,} \\ A * A^{k-1} & \text{if } k \text{ is odd.} \end{cases}$$

Translating this recurrence into our algorithmic language we get

```

function RMatPow (A, k)
  if k = 1 then return A
  else if k is even then
    return RMatPow(A, k/2)*RMatPow(A, k/2)
  else
    return A*RMatPow(A, k - 1)
endfunc RMatPow
    
```

The iterative version of RMatPow is as follows:<sup>2</sup>

```

function IMatPow (A, k)
    R := I
    S := A
    while k > 0 do
        if k is odd then R := R*S
        S := S*S
        k := k ÷ 2
    endwhile
    return R
endfunc IMatPow
    
```

Note that ÷ means integer division, and that a **while** – loop is needed here because we do not know the prime decomposition of *k*. This algorithm, although superficially simple, is not easy to derive from RMatPow.

Here are the MATLAB implementations of these functions.

```

function R = RMatPow(A,k);
    if k == 1
        R=A;
    elseif mod(k,2) == 0
        R = RMatPow(A,k/2)*RMatPow(A,k/2);
    else
        R = A*RMatPow(A,k-1);
    end;

function R = IMatPow(A,k);
    R = eye(size(A)); S = A;
    while k > 0
        if mod(k,2) == 1
            R = R*S;
        end;
        S = S*S;
        k = fix(k/2);
    end;
    
```

**Question.** Two of all the mults that IMatPow performs are redundant. Can you find them?

**Question 2.** How does MATLAB do matrix integer powers, i.e.,  $A^k$ ? Does it use the method in the first algorithm or the second?

**Solution 2.** Table 6.2 suggests that MATLAB is using the repeated squaring method to calculate  $A^k$ . This method is  $O(n^3 \log_2 k)$ , which accounts for the slow growth of  $T(k)$  for fixed  $n$ .

Table 6.2: Times for  $A^k$  where  $A$  is  $500 \times 500$

<i>k</i>	6	7	8	9	10	11	12	13	14	15	16
$T_{ML}(k)$	1.922	2.375	1.907	2.359	2.360	2.828	2.375	2.828	2.859	3.312	2.391
$T_{RS}(k)$	1.906	2.359	1.937	2.375	2.406	2.844	2.391	2.844	2.829	3.297	2.375

**Question 3.** In MATLAB, what is the difference between  $A^k$  and  $A.^k$ ? When you find the answer to this question, please write to O-MATRIX and tell them what the difference is.

**Solution 3.**  $A^2 = A \times A$ , and  $A.^2 = [a_{ij}^2]$ . O-MATRIX uses  $A^2 = [a_{ij}^2]$ . Perverse!

**Note** [4 Feb 2009] : Perhaps this is not perverse. Fortran calculates  $C = A*B$  as  $c_{ij} = a_{ij} * b_{ij}$

<sup>2</sup>From Brassard & Bratley, *Fundamentals of Algorithmics*, Prentice-Hall, 1996, page 247.

## 6.6 Matlab Notes

### 6.6.1 Matlab's Backslash command to solve $Ax = b$

The statement  $x=A\backslash b$  for dense  $A$  performs these steps (stopping when successful):

1. If  $A$  is upper or lower triangular, solve by back/forward substitution
2. If  $A$  is permutation of triangular matrix, solve by permuted back substitution (useful for  $[L,U]=lu(A)$  since  $L$  is permuted)
3. If  $A$  is symmetric/hermitian
  - Check if all diagonal elements are positive
  - Try Cholesky, if successful solve by back substitutions
4. If  $A$  is Hessenberg (upper triangular plus one subdiagonal), reduce to upper triangular then solve by back substitution
5. If  $A$  is square, factorize  $PA = LU$  and solve by back substitutions
6. If  $A$  is not square, run Householder  $QR$ , solve least squares problem

### Left or Right Matrix Division

Here is the official word from MATHWORKS

**Syntax:** `mldivide(A,B)` or `A\B`.     `mrdivide(B,A)` or `B/A`.

#### Description

`mldivide(A,B)` and the equivalent `A\B` perform matrix left division (back slash).  $A$  and  $B$  must be matrices that have the same number of rows, unless  $A$  is a scalar, in which case `A\B` performs element-wise division — that is,  $A\backslash B = A.\backslash B$ .

If  $A$  is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way. If  $A$  is an  $n \times n$  matrix and  $B$  is a column vector with  $n$  elements, or a matrix with several such columns, then  $X = A\backslash B$  is the solution to the equation  $AX = B$  computed by Gaussian elimination with partial pivoting (see Algorithm for details). A warning message is displayed if  $A$  is badly scaled or nearly singular.

If  $A$  is an  $m \times n$  matrix with  $m \neq n$  and  $B$  is a column vector with  $m$  components, or a matrix with several such columns, then  $X = A\backslash B$  is the solution in the least squares sense to the under- or overdetermined system of equations  $AX = B$ . In other words,  $X$  minimizes  $\|AX - B\|$ , the length of the vector  $AX - B$ . The rank  $k$  of  $A$  is determined from the  $QR$  decomposition with column pivoting (see Algorithm for details). The computed solution  $X$  has at most  $k$  nonzero elements per column. If  $k < n$ , this is usually not the same solution as `x = pinv(A)*B`, which returns a least squares solution.

`mrdivide(B,A)` and the equivalent `B/A` perform matrix right division (forward slash).  $B$  and  $A$  must have the same number of columns.

If  $A$  is a square matrix, `B/A` is roughly the same as `B*inv(A)`. If  $A$  is an  $n \times n$  matrix and  $B$  is a row vector with  $n$  elements, or a matrix with several such rows, then  $X = B/A$  is the solution to the equation  $XA = B$  computed by Gaussian elimination with partial pivoting. A warning message is displayed if  $A$  is badly scaled or nearly singular.

If  $B$  is an  $m \times n$  matrix with  $m \neq n$  and  $A$  is a column vector with  $m$  components, or a matrix with several such columns, then  $X = B/A$  is the solution in the least squares sense to the under- or over-determined system of equations  $XA = B$ .

**Note:** Matrix right division and matrix left division are related by the equation  $B/A = (A'\backslash B)'$ .