UNIVERSITY OF CAMBRIDGE

Department of Engineering

IT Services

Matlab vectorisation tricks

Some basic tips on speeding up matlab code and on exploiting vectorisation are mentioned in the Optimisation section of our matlab page. Some of the tricks below come from newsreader.mathworks.com and the Mathworks site. They are low-level and capable of delivering order-of-magnitude improvements.

I've added the author names (where known). If you have contributions, mail them to tpl@eng.cam.ac.uk.

Indexing using vectors

Many of these tricks use the fact that there are two ways of accessing matrix elements using a vector as an 'index'.

- If X and V are vectors, then X(V) is [X(V(1)), X(V(2)), ..., X(V(n))]. So for example if X=[2 5 8 11 14 17 20 23 26 29] and V=[4 2 6] then X(V) is [11 5 17]. This idea is often used in association with others. For example, Matlab has a randperm(n) routine that returns a random permutation of the integers from 1 to n. You can use this to produce a random permutation of X by doing X(randperm(length(X)))
- If X and V are the same size and V only consists of true or false elements then MATLAB interprets V as a mask, and returns only the elements of X whose position corresponds to the location of a true in V. For example, if X is an array, then X>6 is an array the same size as X with trues where the corresponding element in X is >6 and falses elsewhere. This array can be used as a "mask" to select only the elements in X which are >6. Try

```
X=1:10
V=X>6
X(V)
```

  or, more succinctly,

```
X=1:10
X(X>6)
```

Using those ideas you can solve this problem - how can you determine the number of items in a 2D array that are greater than 8? Have a go before reading on ...

```
m=magic(5)
```

creates a 2D matrix that you can practice with. Doing

```
m>8
```

is part of the way towards an answer. How can you sum these elements? Try

```
sum(m>8)
```

That's closer - sum has summed the columns. If you sum again

```
sum(sum(m>8))
```

you'll get the answer. An alternative is

```
sum(m(:)>8)
```

m(:) has all the elements of m in a single column

Creating and manipulating Matrixes

To use the indexing ideas effectively you need to be able to create 'mask' matrices efficiently, and manipulate arrays. This requires the use of functions that you may not have used before. Some are listed here -

- Array Manipulation - look at flipud, fliplr, rot90, triu (extracts upper triangle), tril (extracts lower triangle), filter.
- Array Creation - look at hankel (try hankel(1:5), for instance), kron, toeplitz, diag, sparse, repmat (replicates and tiles matrices) and find.

Reshaping matrices can help help when vectorising. Fortunately reshaping is a cheap operation - the underlying data isn't moved. It often helps to be able to locate the original location of an element even if the matrix has changed shape. sub2ind and ind2sub are useful in this regard. The following example creates a 3D matrix where one element is 7, makes that matrix into a vector and finds where the 7 is (it's the 3rd element), then works out where that element is in the original matrix.

```
m=zeros(2,3,4);
m(1,2,1)=7;
a=find(m(:)==7)
[x y z ]=ind2sub(size(m),a)
```

Extra features of common routines

Some fairly commonly used routines have extra features that are especially useful when vectorising

- ismember - Suppose you have a 2D matrix A and another 2D matrix B with the same number of columns. To find which rows of A are in B, you can do ismember(A,B,'rows')
- all - to remove from a 2D matrix M all the rows that contain only zeroes, you can do m(all(m==0,2),:)=[] (the ",2" argument to the all command does a per-row comparison)
- unique - this routine returns an array with duplicates removed, but the returned array is sorted too. If you don't want this, try something like

  ```
  a=[3 2 2 1 7];
  [B,I,J] = unique(a,'first')
  a(sort(I))
  ```

  Note that the final line uses a vector as an 'index'

Vectorising Routines

- bsxfun (Binary Singleton eXpansion FUNction) is quite a new routine. The example that Mathworks offer subtracts the column means from the matrix A

  ```
  A = magic(5);
  A = bsxfun(@minus, A, mean(A))
  ```

  Here's another example

  ```
  A = magic(5);
  B= A+3;
  % The next line is the equivalent of sqrt(A.^2 + B.^2)
  A = bsxfun(@hypot, A, B)
  ```

Examples

These examples are short, so by reading about the functions used and testing with small matrices, you should be able to discover why they work. More than one way is shown to solve some of these questions so that you can compare methods. When big matrices are used, you might find that some methods are hundreds of times faster than others. Remember that the most elegant-looking way may not be the fastest, and faster methods may use a lot more memory. Use flops or tic and toc to assess performance.

- *Create a 1000 by 1000 vector full of 7s*
  1. X=7*ones(1000,1000);
  2. X=repmat(7,1000,1000);
  3. X(1:1000,1:1000)=7;

  Which method is fastest depends on the version of matlab. For me, currently, method 1 is fastest, with methods 2 and 3 being 50-100% slower. The moral is: test and measure, don't just hypothesise or depend on habit.
- *How can you reverse a row vector?*
  1. X(end:-1:1)
  2. fliplr(X)
- *How can you reverse one column of a matrix*
  1. To reverse column c of a matrix M, try M(:,c)=flipud(M(:,c)).
- *Remove from a vector all the elements that are equal to the biggest element*

```
x(x==max(x)) = []
```

  (Jos)
- *Subtract 3 from each element of x which is greater than 3*

```
x(x>3)=x(x>3)-3;
```

  (Yuri Strukov)
- *A vector x contains some 0s. Create y such that. y[i] is 0 if x[i] is 0, otherwise y[i] is log(x[i])*

```
y=zeros(size(x));
y(find(x))=log(x(find(x)))
```

- *Remove from a 2-D matrix all the rows that contain at least one element less than 3*

```
A=magic(5)
A(ismember(A<3,zeros(size(A,2)),'rows'),:)
```

- *How can you swap rows?*
  1. M([1,5],:) = M([5,1],:) interchanges rows 1 and 5 of M.
  2. More generally, if V has m components and W has n components, then M(V,W) is the m-by-n matrix formed from the elements of M whose subscripts are the elements of V and W. So for example,

```
M=magic(5)
V=[1,2,3]
W=[4,3]
M(V,W)
```

     produces a matrix with 2 columns (values taken from columns 4 and 3) and 3 rows (values taken from rows 1, 2, and 3)
- *Vectorise the following, where data is 200x400*

```
for i = 1:200
   for j = 1:400
     if data(i,j) > 0
           data(i,j) = 0;
     end
   end
end
```

```
data(data > 0) = 0;
```

- *Vectorise the following threshold function*

```
function A = threshold(Matrix, max_value, min_value)
    [a, b] = size(Matrix);
    for i = 1:a
       for j = 1:b
              c =  Matrix(i,j);
              if ( c > max_value )
                 Matrix(i,j) = max_value;
              elseif ( c < min_value )
                 Matrix(i,j) = min_value;
              end
       end
    end
```

By Peter J. Acklam

```
A = min( max( Matrix, min_value ), max_value );
```

- *z is a vector of 100 numbers. Produce a vector p where p(1) sums the 1st 5 elements of z, p(2) sums the next 5 and so on.*
  This can be done using a loop inside a loop, but by reshaping the data to match the way matlab's sum command works, you can avoid explicit loops.

```
p = sum(reshape(z,[5 20]))
```

(Anh Huy Phan)

- *Sum all the elements of a 2D matrix m that are greater than 23*

```
m=magic(5); % create a sample matrix
sum(m(m(:)>23))
```

- *Obtain the mean of each column of x. Ignore elements <=0.*

```
keepers = (x>0);
colSums = sum(x .* keepers);
counts = sum(keepers);
means = colSums ./ counts;
```

- *Vectorise*

```
for i = 1:100
  for j = 1:100
    r(i,j) = sqrt(i^2+j^2);
  end
end
```

```
[i,j]=meshgrid(1:100,1:100);
r = sqrt(i.^2+j.^2);
```

The following alternative works with new versions

```
[i,j]=meshgrid(1:100,1:100);
r=bsxfun(@hypot, i, j)
```

- *Vectorise the following, where elements depend on previous ones*

```
n=1000;
x(1)=1;
for j=1:n-1,
   x(j+1) = x(j) + n - j;
end
```

```
n=1000;
x(1)=1;
j=1:n-1;
x(j+1) = n - j;
cumsum(x);
```

- *From A = [3,4,3] and B = [1,2,3,4,5,6,7,8,9,10] produce a vector C where C(1) is the sum of the first A(1) elements of B, C(2) is the sum of the next A(2) elements of B, etc.*
  By Ulrich Elsner

```
foo=cumsum(B);
C=diff([0 foo(cumsum(A))])
```

- *Find the maximum of sin(x) * cos(y) where x is 1,2..7 and y is 1,2 .. 5.*

```
x=1:7
[rx,cx]=size(x);

y=1:5
[ry,cy]=size(y);

X=repmat(x,cy,1)
Y=repmat(y',1,cx)

% Doing max(max( sin(X).*cos(Y))) would find the max in one
% line. To know the location of the maximum too, use
XY=sin(X).*cos(Y)
[r,c]=max(XY);
[r2,c2]=max(max(XY));
sprintf('The biggest element in XY is %d at XY(%d,%d)',r2,c(c2), ...
        c2)
```

- *Count how many times the components of a matrix are repeated.*
  1.
  ```
  bins = min(min(X)):max(max(X));
  [numTimesInMatrix, Number] = hist(X(:),bins);
  ```
  From Mathworks - "Hist can only partially vectorize this problem, so it uses a loop as well. Therefore, the algorithm runs with order O(length(bins)). Also, this algorithm reports the number of occurrences of all numbers in bin, including those that appear zero times. To get rid of these, add this":
  ```
  ind=find(~numTimesInMatrix);
  Number(ind)=[];numTimesInMatrix(ind)=[];
  ```
  2.
  ```
  x = sort(X(:));
  difference = diff([x;max(x)+1]);
  count = diff(find([1;difference]));
  y = x(find(difference));
  ```
  Note that we use the (:) operation to ensure that x is a vector.
  3.
  ```
  count = sparse(1,X,1);
  ```
  (read about sparse if, like me, you were surprised by this)

- *Retrieve those elements that are shared in matrices A and B*
  1.
  ```
  intersect(A(:),B(:))
  ```
  (Sijmen-de.Jong)
  2. For *positive* integers:
  ```
  a=A(:); % make A a row
  b=B(:);
  m=max([a;b]);
  x1=zeros(m,1);
  x2=zeros(m,1);
  x1(a)=ones(length(a),1);
  x2(b)=ones(length(b),1);
  find(x1&x2)
  ```
  3. For equally sized A and B:
  ```
  a = A(:)'+1;
  b = B(:)'+1;
  simple = [find(sparse(1,a,1)>0);find(sparse(1,b,1)>0)];
  values = find(full(sparse(1,simple',1))>1)-1
  ```

- *Scale all the rows of each column by the data in col 300.*

```
[nr,nc] = size(A);
B = A(:,300);
A = A ./ B( :, ones(nc,1) );
```

- *Write an m-file which will replace each element of a matrix with a 4x4 matrix of that element.*
  1. From pete@electrosystems.com

```
x=[1 2 3; 4 5 6];
y=kron(x,ones(4))
```

  2.

```
A=randn(100);
A(ceil((1:(size(A,1)*4))/4),ceil((1:(size(A,2)*4))/4));
```

- *Duplicate each element of a vector.*
  1. By Ulrich Elsner

```
rep=2;
foo=repmat(b,rep,1);
b=foo(:)'              % reshape into row
```

- *Multiply every column of matrix X of size (1000,500) by another vector V which has dimension (1000).*
  1. A non-vectorised way to do this is:

```
X2 = zeros(1000,500)
for k = 1:500
   X2(:,k) = V.*X(:,k);
end
```

  2.

```
X2 = diag(V)*X;
```

  3.

```
rows=1000;
columns = 500;
Gcolvec = [1:rows]'; % Be sure that G is a column vector
Gmatrix = repmat(Gcolvec,1,columns);
R = ones(rows, columns);
tic, GxR = Gmatrix.*R; toc
```

  4. By marco@chinook.physics.utoronto.ca

```
Ra = R .* (G * ones(1,500));
Rb = R .* repmat(G,1,500);
Rc = R .* G(:,ones(1,500));
```

  5. The solution proposed in the Mathworks "How Do I Vectorize My Code?" is the fastest

```
X2 = diag(sparse(V))*X;
```

- *If you have two matrices, which together give x and y coordinates into another matrix, eg.*

```
xc = [1 2 1; 3 2 4 ];
yc = [1 1 1; 2 2 1; ];

M = [ 6  7  8  9;
     10 11 12 13];
```

  produce the 2x3 matrix R which contains the values of M at the locations given in xc and yc. Thus:

```
R = [ 6  7 6;
     12 11 9];
```

  1.

```
R=M(sub2ind(size(M),yc,xc));
```

  2.

```
a = [1 4 3 2]
b = [2 3 1 4]
ind = sub2ind([4 4],a,b)
A = rand(4,4)
A(ind)
```

- *Create a matrix X, where each column is a shifted copy of the vector v*

```
v = (1:5)';
X = toeplitz(v, v([1,length(v):-1:2]))
```

  To shift up rather than down, use

```
% By Robert Richter
hankel(v,v([end,1:end-1]))
```

- *Unique sort (i.e. duplicates removed)*
  -
    ```
    x = sort(x(:));
    unique(x)
    ```
  -
    ```
    x = sort(x(:));
    difference = diff([x;NaN]);
    y = x(difference~=0);
    ```
    From Mathworks - "You may be wondering why we didn't use the find function, rather than ~=0 - especially since this function is specifically mentioned above. This is because the find function does not return indices for NaN elements, and the last element of difference as we defined it is a NaN."
- *Given a 2D matrix remove all duplicate rows (where a row is a considered a duplicate of another if it has the same elements)*

```
a=[1 2 5
     2 6 7
     1 2 9
     1 3 6
     5 1 2
     3 6 7
     7 2 6]
[result_sorted,INX] = unique(sort(a,2),'rows');
a(INX,:)
```

(by Jos)

- *Create a vector vi which is made by interleaving elements from an array v1 of length N and an array v2 of length N-1*

```
% By Peter J. Acklam
N=10000
v1 = 1:N;
v2 = 1:N-1;

   vi = zeros(1, 2*N-1);
   vi(1:2:end) = v1;
   vi(2:2:end) = v2;
% or
   vi = [ v1 ; v2 0 ];
   vi = vi(1:end-1);
```

- *Write a function d = nearney(x, y) where x, y and d are column vectors of the same length and where d(i) is the smallest distance on a plane from the point [x(i) y(i)] to [x(j) y(j)] for all i ~= j; that is the distance from [x(i) y(i)] to its nearest neighbor.*
  1. From moler@mathworks.com - "This gets good marks for terseness, but has lousy time and space complexity"

     ```
     function d = nearney(x,y)
     [X,Y] = meshgrid(x+i*y);
     d = min(abs(X-Y) + realmax*eye(length(x)))';
     ```
  2. From Sijmen de Jong

     ```
     z = [x, y];
     d  = sum(z'.^2);
     [m,n] = size(z);
     d = sqrt(min(d(ones(m,1),:)+d(ones(m,1),:)'-2*z*z'+realmax*eye(m)));
     ```
- *Another 'find the nearest' question Write the code to do this*

```
function out = nearest(y,x1,x2);
% function out = nearest(y,x1,x2);
% y,x1,x2, and out are row vectors of the same length
% out(i) = x1(i) or x2(i) depending on which one is closer to y(i)
%
% for example if:
%        y = [3    5    7  9    11]
%       x1 = [3.1  6    7  8.9  12]
%       x2 = [3.2  5.9  0  0    10.9]
% then out = [3.1  5.9  7  8.9  10.9]
```

1.
```
a = [x1; x2];
[c d] = min(abs([y; y] - a));
out = diag(a(d,:))';
```
2.
```
out = x1;
i2 = find(abs(x1-y) > abs(x2-y));
out(i2) = x2(i2);
```
3. More generally

```
function out = nearest(y,x)
% function out =nearest(y,x)
% y is a row-vector and x is composed of row-vectors of the same length as y.
%
% out(i)=the element of x(:,i) closest to y(i)
%
% For example if:
%      y= [3    5    7  9    11]
%      x=[[3.1  6    7  8.9  12];
%         [3.2  5.9  0  0    10.9]]
% then out=[3.1  5.9  7  8.9  10.9]
%
[c d]=min(abs(y(ones(size(x,1),1),:)-x));
out=x(d+(0:size(x,2)-1)*size(x,1));
```

- *Given a matrix of points, find the distance between each pair of points*

```
points = [ 3 4 5;
           6 5 7;
           6 7 1;
           7 6 0;
           3 2 4;
           1 0 3;
           5 6 2;
           9 0 1]

m=8;
p=3;

% Here's a one-liner by Peter Acklam. Results are in D
D= sqrt(sum(abs(repmat(permute(points, [1 3 2]), [1 m 1]) ...
   - repmat(permute(points, [3 1 2]), [m 1 1]) ).^2,3));

% Here's another solution by Peter Acklam which exploits symmetry
[ i j ] = find(triu(ones(m),1));
E= zeros(m,m);
E(i+m*(j-1) ) = sqrt(sum(abs(points(i,:) -points(j,:)).^2,2));
E(j + m*(i-1)) = E(i+m*(j-1));

% And here's a solution by Jyri Kivinen based on the notes in
% Borg and Groenen's "Modern Multidimensional Scaling: Theory and
% Applications"
K=points*points';
```

```
d=diag(K);
one=ones(length(d),1);
D=sqrt(d*one'+one*d'-2*K);
```

- *Suppose have a 4-column matrix and a row vector [0 1 0 1]. How can I create new matrix containing the same column data as the original data when the corresponding row vector element is 1, and a column of 0s otherwise?*
  Given

```
A= [1 2 3 4
6 7 1 2
1 2 4 3
1 2 3 1];

v=[0 1 0 1];
```

1.
```
[M,N] =size(A);
B=A.*repmat(v,M,1)
```
2.
```
B=A*diag(v)
```
(by /PB)

3.
```
B=A; B(:,~logical(v))=0;
```
(by Fabio Gori)

- *Given a vector how can each sequence of more than 1 zero be reduced to a single zero? E.g. given [0 0 1 5 6 0 7 0 7 0 0 9 8 0 0] produce [0 1 5 6 0 7 0 7 0 9 8 0]*

```
a = [0 0 1 5 6 0 7 0 7 0 0 9 8 0 0]
a( (a(1:end-1)==0) & (a(2:end)==0) )=[]
```
(by Alan B)

- *You have a logical vector l1 with n true values, and another logical vector l2 of length n. For all the positions i s.t l2(i) == false, set the i-th true value of l1 to false. E.g. given* l1 =[0 0 1 0 1 0 0 1 1 1]*and* l2 =[0 0 1 1 1]*, l1 should end up as* 0 0 0 0 0 0 0 1 1 1.

```
v=find(l1); v=v(l2); l1=false(size(l1)); l1(v)=true;
```
(by Fabio Gori)

- *Suppose you have a vector x=[2 6 99 3 10] and a vector y=['a' 'b' 'c' 'd' 'e']. How could you re-order y so that if the elements of x were re-ordered in the same way, the elements would be in order? (in this example the re-ordered y would be "adbec")*

```
[dummy I]=sort(x);
y=y(I)
```

- *For each row of an array how do you find the longest sequence of numbers that are less than the mean for that row? (the actual question involved analysing an image of a vertical crack, measuring the width of the crack).*

```
% create a test array
m=rand(10,10)

% find mean of each row
means=mean(m,2);
for row=1:10
  [Y,I] =max(diff([0 (find(~(m(row,:)<means(row)))) numel(m(1,:))+1]) -1);
  disp(sprintf('Row %d biggest crack (width %d) at %d ',row, Y, I))
end
```
This uses code that "kinor" on a discussion board described as follows. Suppose you wanted to find out about sequences of numbers less than 1.

```
diff(find(~(A<1)) )
```

will give you the number of steps from one element >=1 to the next element >=1, the number of steps being one plus the number of elements being <1 on the way, so the number you are interested in is

```
diff(find(~(A<1)) )-1
```

You then have to take account of the first and final values, where you'll need to decide whether sequences can begin at an edge, etc.

A worked example

You have 2 sets of (x,y,z) coordinates. Find the distance from each of the points in the 1st set to each of the points in the 2nd set
The following worked example may not be the most elegant solution, but I hope it's educational. Let's start with 2 little datasets.

```
one=[1 2 3 ; 6 5 9; 8 1 3]
two=[1 2 3 ; 4 5 6]
```

Clearly one could use nested loops to solve this, but we're going to sacrifice space in the hope that we'll gain speed. First note that if we had 2 points a = [1 2 3] and b = [1 4 7] we can find the distance between them using sqrt(sum((a-b).^2)). How can we use this formula to find more distances at once? Suppose we have

```
aa = [ 1 2 3 ;          bb = [1 4 7 ;
       1 2 3 ]                1 2 3 ]
```

- how much do we need to adjust the formula we used for a and b? sqrt(sum((aa-bb).^2)) gives 0 2 4 because sum has summed the columns, but sqrt(sum((aa-bb).^2,2)) sums the rows and gives [4.4721 ; 0], which is what we want in this case. So if we could take the initial matrices one and two and replicate their rows to form the matrices below, we could get all the distances we need by using the revised formula.

```
bigone =                    bigtwo =
   [ 1     2     3;             [ 1     2     3;
     6     5     9;               1     2     3;
     8     1     3;               1     2     3;
     1     2     3;               4     5     6;
     6     5     9;               4     5     6;
     8     1     3]               4     5     6]
```

Getting bigone is easy enough. First we need to know the array sizes

```
[r1 c1]=size(one);
[r2 c2]=size(two);
```

Now bigone=repmat(one,r2,1). There's probably an easy way to get bigtwo as well, but at the moment all I can see is

```
foo=repmat(two,1,r1)
g=reshape(foo',3,r1*r2)
bigtwo=g'
```

Now

```
a= sqrt(sum((bigone-bigtwo).^2,2))
```

will give us 6 distances as a column vector. We can put them into a more convenient shape by using reshape(a,r1,r2). Compressing these lines of code to avoid too many temporary variables (they slow things down) gives us

```
one=[1 2 3 ; 6 5 9; 8 1 3]
two=[1 2 3 ; 4 5 6]
[r1 c1]=size(one);
[r2 c2]=size(two);
bigone=repmat(one,r2,1);
```

```
bigtwo=reshape(repmat(two,1,r1)',3,r1*r2)'
reshape(sqrt(sum((bigone-bigtwo).^2,2)),r1,r2)
```

and the answer

```
     0    5.1962
8.3666    3.6056
7.0711    6.4031
```

The code above when using matrices with 5000 and 100 points (created using one=rand([5000,3]); two=rand([100,3]);) was 30 times faster than the following code

```
[r1 c1]=size(one);
[r2 c2]=size(two);
for i=1:r1
  for j=1:r2
    answer(i,j)=sqrt(sum((one(i,:)-two(j,:)).^2));
  end
end
```

A bubblesort using 1 loop

Each time the program goes round the loop it will look at the (1st 2nd), (3rd 4th) , ... pairs, swapping the numbers if necessary, then it will look at the (2nd 3rd), (4th 5th) ... pairs, swapping the numbers if necessary. The resulting code might not be very fast ...

```
a=[ 1 2 5 4 8 10 3 6 9 7 0];
len=length(a);
evenmask= repmat([0 1], 1, fix(len/2));
oddmask = repmat([1 0], 1, fix(len/2));
if rem(len,2)==1,
   evenmask(len)=0;
   oddmask(len)=1;
end
changed=1;

while changed==1,
  changed=0;
  %swap odd-end pairs if necessary
  odds_shifted_right=[0 a(1:len-1)].*evenmask;
  evens=a.*evenmask;
  change = evens<odds_shifted_right;
  if length(find(change)) ~= 0,
    changed=1;
    swopped=odds_shifted_right+[a(2:len) 0].*oddmask;
    mask=change+ [change(2:len) 0];
    a= ~mask.*a + mask.*swopped
  end
  %swap even-odd pairs if necessary
  odds_shifted_left=[a(2:len) 0].*evenmask;
  evens=a.*evenmask;
  change = evens>odds_shifted_left;
  change(len)=0;
  if length(find(change)) ~= 0,
    changed=1;
    swopped=odds_shifted_left+[0 a(1:len-1)].*oddmask;
```

```
    mask=change+ [0 change(1:len-1)];
    a= ~mask.*a + mask.*swopped
  end
end
```

A Lesson

A user sent this code to the newgroup, asking how to optimise it

```
N = 10;
delt = 5/10;
A = rand(N,N,N);
L = zeros(N^3,4);
ind = -250*delt:delt:250*delt;

  for di=1:N
    for dj=1:N
      for dk=1:N
        L((N^2)*(di-1)+N*(dj-1)+dk,:) = [ind(di),ind(dj),ind(dk),A(di,dj,dk)];
      end
    end
  end
```

Having a complicated line within 3 levels of loops is likely to be costly. Matt Fig suggested

```
...
X = repmat({1:N},3,1);
[X Y Z] = ndgrid(X{:});
L = [ind([Z(:) Y(:) X(:)]) reshape(permute(A,[3 2 1]),[],1)];
```

which gives the same answer without using loops. Ingenious, hard to understand, but if speed is important, who care? He then offered

```
...
cnt = 0;
for di=1:N
  for dj=1:N
      for dk=1:N
          cnt = cnt + 1;
          L(cnt,1) = ind(di);
          L(cnt,2) = ind(dj);
          L(cnt,3) = ind(dk);
          L(cnt,4) = A(di,dj,dk);
      end
  end
end
```

which is nowhere near as impressive. When I tried these out, the non-looping solution was about 10% faster than the original code whereas the last solution was 10 times faster. The moral of the story is - measure. Matlab can be good at optimising code with loops as long as the code is simple. Vectorised code sometimes has hidden costs in terms of time or memory.

See Also

- How Do I Vectorize My Code? (from Mathworks)
- Peter Acklam's MATLAB array manipulation tips and tricks

© Cambridge University, Engineering Department, Trumpington Street, Cambridge CB2 1PZ, UK (map)
Tel: +44 1223 332600, Fax: +44 1223 332662
Contact: helpdesk