MATLAB array manipulation tips and tricks

Peter J. Acklam
Statistics Division
Department of Mathematics
University of Oslo
Norway

E-mail: jacklam@math.uio.no
WWW URL: http://www.math.uio.no/~jacklam/

5 May 2000

Abstract

This document is intended to be a compilation tips and tricks mainly related to efficient ways of performing low-level array manipulation in MATLAB. Here, "manipulate" means replicating and rotating arrays or parts of arrays, inserting, extracting, permuting and shifting elements, generating combinations and permutations of elements, run-length encoding and decoding, multiplying and dividing arrays and calculating distance matrics and so forth. A few other issues regarding how to write fast MATLAB code is also covered.

This document was produced with $\mathcal{AMS}$-LaTeX.
The PS (PostScript) version was created with `dvips` by Tomas Rokicki.
The PDF (Portable Document Format) version was created with `ps2pdf`, a part of Aladdin Ghostscript by Aladdin Enterprises.

The PS and PDF version may be viewed with software available at the Ghostscript, Ghostview and GSview Home Page at http://www.cs.wisc.edu/~ghost/index.html.
The PDF version may also be viewed with Adobe Acrobat Reader available at http://www.adobe.com/products/acrobat/readstep.html.

Contents

1 Introduction

1.1 Background

Since the early 1990's I have been following the discussions in the main MATLAB newsgroup on Usenet, `comp.soft-sys.matlab`. I realized that many postings there were about how to ma-

nipulate arrays efficiently. I decided to start collecting what I thought was the most interestings solutions and see if I could compile them into one document. Well, this is it.

1.2 Vectorization

The term "vectorization" is frequently associated with MATLAB. Strictly speaking, it means to rewrite code so that, in stead of using a for-loop iterating over each scalar in an array, one takes advantage of MATLAB's vectorization capabilities and does everything in one go. For instance, the 5 lines

```
x = [ 1 2 3 4 5 ];
y = zeros(size(x));
for i = 1:5
   y(i) = x(i)^2;
end
```

may be written in the vectorized fashion

```
x = [ 1 2 3 4 5 ];
y = x.^2;
```

which is faster, most compact, and easier to read. With this rather strict definition of "vectorization", vectorized code is always faster than non-vectorized code.

Some people use the term "vectorization" in the sense "removing any for-loop", but I will stick to the former, more strict definition.

1.3 About the examples

All arrays in the examples are assumed to be of class double and to have the logical flag turned off unless it is stated explicitly or it is apparent from the context.

1.4 Credit where credit is due

As far as possible, I have given credit to what I believe is the author of a particular solution. In many cases there is no single author, since several people have been tweaking and trimming each others solutions. If I have given credit to the wrong person, please let me know.

Note especially that I do not claim to be the author of a solution even though there is no other name mentioned.

1.5 Errors/Feedback

If you find errors or have suggestions for improvements or if there is anything you think should be here but is not, please mail me and I will see what I can do. My address is on the front page of this document.

2 Operators, functions and special characters

Clearly, it is important to know the language one intends to use. The language is described in the manuals so I won't repeat here what they say, but I strongly encourage the reader to type

```
help ops
```

Operators and special characters.

at the command prompt and take a look at the list of operators, functions and special characters, and look at the associated help pages.

When manipulating arrays in MATLAB there are some operators and functions that are particularely useful.

2.1 Operators

`:`	The colon operator. Type `help colon` for more information.
`.'`	Non-conjugate transpose. Type `help transpose` for more information.
`'`	Complex conjugate transpose. Type `help ctranspose` for more information.

2.2 Built-in functions

`find`	Find indices of nonzero elements.
`all`	True if all elements of a vector are nonzero.
`any`	True if any element of a vector is nonzero.
`logical`	Convert numeric values to logical.
`end`	Last index in an indexing expression.
`sort`	Sort in ascending order.
`diff`	Difference and approximate derivative.
`sum`	Sum of elements.
`prod`	Product of elements.
`cumsum`	Cumulative sum of elements.
`permute`	Permute array dimensions.
`reshape`	Change size.

2.3 M-file functions

`sub2ind`	Linear index from multiple subscripts.
`ind2sub`	Multiple subscripts from linear index.
`ipermute`	Inverse permute array dimensions.
`shiftdim`	Shift dimensions.
`squeeze`	Remove singleton dimensions.
`repmat`	Replicate and tile an array.
`kron`	Kronecker tensor product.

3 Creating vectors, matrices and arrays

3.1 Special vectors

3.1.1 Uniformly spaced elements

To create a vector of uniformly spaced elements, use the `linspace` function or the `:` (colon) operator:

```
X = linspace(lower, upper, n);        % row vector
X = linspace(lower, upper, n).';      % column vector
```

```
X = lower : step : upper;            % row vector
X = ( lower : step : upper )';       % column vector
```

If `step` is not a multiple of the difference `upper-lower`, the last element of `X`, `X(end)`, will be less than `upper`. So the condition `A(end) <= upper` is always satisfied.

4 Shifting

4.1 Vectors

To shift and rotate the elements of a vector, use

```
X([ end 1:end-1 ]);              % shift right/down 1 element
X([ end-k+1:end 1:end-k ]);      % shift right/down k elements
X([ 2:end 1 ]);                  % shift left/up 1 element
X([ k+1:end 1:k ]);              % shift left/up k elements
```

Note that these only work if `k` is non-negative. If `k` is an arbitrary integer one may use something like

```
X( mod((1:end)-k-1, end)+1 );    % shift right/down k elements
X( mod((1:end)+k-1, end)+1 );    % shift left/up k element
```

where a negative `k` will shift in the opposite direction of a positive `k`.

4.2 Arrays

To shift and rotate the elements of an array `X` along dimension `dim`, first initialize a subscript cell array with

```
idx = repmat({':'}, ndims(X), 1); % initialize subscripts
n   = size(X, dim);               % length along dimension dim
```

then manipulate the subscript cell array as appropriate by using one of

```
idx{dim} = [ n 1:n-1 ];          % shift right/down 1 element
idx{dim} = [ n-k+1:n 1:n-k ];    % shift right/down k elements
idx{dim} = [ 2:n 1 ];            % shift left/up 1 element
idx{dim} = [ k+1:n 1:k ];        % shift left/up k elements
```

finally create the new array

```
Y = X(idx{:});
```

5 Replicating elements and arrays

5.1 Constant array

To create an array whose size is `siz` and where each element has the value `val`, use one of

```
X = repmat(val, siz);
X = val(ones(siz));
```

The repmat solution might in some cases be slighly slower, but it has several advantages. Firstly, it uses less memory. Seconly, it also works if val is a function returning a scalar value, e.g., if val is Inf or NaN:

```
X = NaN(ones(siz));       % won't work unless NaN is a variable
X = repmat(NaN, siz);     % this works
```

Avoid using

```
X = val * ones(siz);
```

since it does unnecessary multiplications and only works if val is of class "double".

5.2 Replicating elements in vectors

5.2.1 Replicate each element a constant number of times

Example Given

```
N = 3; A = [ 4 5 ]
```

create N copies of each element in A, so

```
B = [ 4 4 4 5 5 5 ]
```

Use, for instance,

```
B = A(ones(1,N),:);
B = B(:).';
```

If A is a column-vector, use

```
B = A(:,ones(1,N)).';
B = B(:);
```

Some people use

```
B = A( ceil( (1:N*length(A))/N ) );
```

but this requires unnecessary arithmetic. The only advantage is that it works regardless of whether A is a row or column vector.

6 Reshaping arrays

6.1 Subdividing 2D matrix

Assume X is an m-by-n matrix.

6.1.1 Create 4D array

To create a p-by-q-by-m/p-by-n/q array Y where the i,j submatrix of X is Y(:,:,i,j), use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = permute( Y, [ 1 3 2 4 ] );
```

Now,

```
X = [  Y(:,:,1,1)   Y(:,:,1,2)  ...  Y(:,:,1,n/q)
       Y(:,:,2,1)   Y(:,:,2,2)  ...  Y(:,:,2,n/q)
          ...          ...      ...      ...
       Y(:,:,m/p,1) Y(:,:,m/p,2) ... Y(:,:,m/p,n/q) ];
```

To restore `X` from `Y` use

```
X = permute( Y, [ 1 3 2 4 ] );
X = reshape( X, [ m n ] );
```

6.1.2 Create 3D array (columns first)

Assume you want to create a `p`-by-`q`-by-`m*n/(p*q)` array `Y` where the `i,j` submatrix of `X` is `Y(:,:,i+(j-1)*m/p)`. E.g., if `A`, `B`, `C` and `D` are `p`-by-`q` matrices, convert

```
X = [ A B
      C D ];
```

into

```
Y = cat( 3, A, C, B, D );
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = permute( Y, [ 1 3 2 4 ] );
Y = reshape( Y, [ p q m*n/(p*q) ] )
```

Now,

```
X = [  Y(:,:,1)  Y(:,:,m/p+1) ... Y(:,:,(n/q-1)*m/p+1)
       Y(:,:,2)  Y(:,:,m/p+2) ... Y(:,:,(n/q-1)*m/p+2)
         ...        ...      ...        ...
       Y(:,:,m/p) Y(:,:,2*m/p) ...    Y(:,:,n/q*m/p)    ];
```

To restore `X` from `Y` use

```
X = reshape( Y, [ p q m/p n/q ] );
X = permute( X, [ 1 3 2 4 ] );
X = reshape( X, [ m n ] );
```

6.1.3 Create 3D array (rows first)

Assume you want to create a `p`-by-`q`-by-`m*n/(p*q)` array `Y` where the `i,j` submatrix of `X` is `Y(:,:,j+(i-1)*n/q)`. E.g., if `A`, `B`, `C` and `D` are `p`-by-`q` matrices, convert

```
X = [ A B
      C D ];
```

into

```
Y = cat( 3, A, B, C, D );
```

use

```
Y = reshape( X, [ p m/p n ] );
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ p q m*n/(p*q) ] );
```

Now,

```
X = [       Y(:,:,1)             Y(:,:,2)       ...   Y(:,:,n/q)
          Y(:,:,n/q+1)         Y(:,:,n/q+2)     ...  Y(:,:,2*n/q)
              ...                  ...          ...      ...
      Y(:,:,(m/p-1)*n/q+1) Y(:,:,(m/p-1)*n/q+2) ... Y(:,:,m/p*n/q) ];
```

To restore X from Y use

```
X = reshape( Y, [ p n m/p ] );
X = permute( X, [ 1 3 2 ] );
X = reshape( X, [ m n ] );
```

6.1.4 Create 2D matrix (columns first, column output)

Assume you want to create a m*n/q-by-q matrix Y where the submatrices of X are concatenated (columns first) vertically. E.g., if A, B, C and D are p-by-q matrices, convert

```
X = [ A B
      C D ];
```

into

```
Y = [ A
      C
      B
      D ];
```

use

```
Y = reshape( X, [ m q n/q ] );
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ m*n/q q ] );
```

To restore X from Y use

```
X = reshape( Y, [ m n/q q ] );
X = permute( X, [ 1 3 2 ] );
X = reshape( X, [ m n ] );
```

6.1.5 Create 2D matrix (columns first, row output)

Assume you want to create a p-by-m*n/p matrix Y where the submatrices of X are concatenated (columns first) horizontally. E.g., if A, B, C and D are p-by-q matrices, convert

```
X = [ A B
      C D ];
```

into

```
Y = [ A C B D ];
```

use

```
Y = reshape( X, [ p m/p q n/q ] )
Y = permute( Y, [ 1 3 2 4 ] );
Y = reshape( Y, [ p m*n/p ] );
```

To restore X from Y use

```
Z = reshape( Y, [ p q m/p n/q ] );
Z = permute( Z, [ 1 3 2 4 ] );
Z = reshape( Z, [ m n ] );
```

6.1.6 Create 2D matrix (rows first, column output)

Assume you want to create a `m*n/q`-by-`q` matrix `Y` where the submatrices of `X` are concatenated (rows first) vertically. E.g., if `A`, `B`, `C` and `D` are `p`-by-`q` matrices, convert

```
X = [ A B
      C D ];
```

into

```
Y = [ A
      B
      C
      D ];
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = permute( Y, [ 1 4 2 3 ] );
Y = reshape( Y, [ m*n/q q ] );
```

To restore `X` from `Y` use

```
X = reshape( Y, [ p n/q m/p q ] );
X = permute( X, [ 1 3 4 2 ] );
X = reshape( X, [ m n ] );
```

6.1.7 Create 2D matrix (rows first, row output)

Assume you want to create a `p`-by-`m*n/p` matrix `Y` where the submatrices of `X` are concatenated (rows first) horizontally. E.g., if `A`, `B`, `C` and `D` are `p`-by-`q` matrices, convert

```
X = [ A B
      C D ];
```

into

```
Y = [ A B C D ];
```

use

```
Y = reshape( X, [ p m/p n ] );
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ p m*n/p ] );
```

To restore `X` from `Y` use

```
X = reshape( Y, [ p n m/p ] );
X = permute( X, [ 1 3 2 ] );
X = reshape( X, [ m n ] );
```

7 Rotating matrices and arrays

7.1 Rotating 2D matrices

To rotate an `m`-by-`n` matrix `X`, `k` times 90° counterclockwise one may use

```
Y = rot90(X, k);
```

or one may do it like this

```
Y = X(:,n:-1:1).';        % rotate 90 degrees counterclockwise
Y = X(m:-1:1,:).';        % rotate 90 degrees clockwise
Y = X(m:-1:1,n:-1:1);     % rotate 180 degrees
```

In the above, one may replace `m` and `n` with `end`.

7.2 Rotating ND arrays

Assume `X` is an ND array and one wants the rotation to be vectorized along higher dimensions. That is, the same rotation should be performed on all 2D slices `X(:,:,i,j,...)`.

Rotating 90 degrees counterclockwise

```
s = size(X);                  % size vector
v = [ 2 1 3:ndims(X) ];       % dimension permutation vector
Y = permute( X(:,s(2):-1:1,:), v );
Y = reshape( Y, s(v) );
```

Rotating 180 degrees

```
s = size(X);
Y = reshape( X(s(1):-1:1,s(2):-1:1,:), s );
```

Rotating 90 clockwise

```
s = size(X);                % size vector
v = [ 2 1 3:ndims(X) ];     % dimension permutation vector
Y = reshape( X(s(1):-1:1,:), s );
Y = permute( Y, v );
```

7.3 Rotating ND arrays around an arbitrary axis

When rotating an ND array `X` we need to specify the axis around which the rotation should be performed. In the cases above, the rotation was performed around an axis perpendicular to a plane spanned by dimensions one (rows) and two (columns). To rotate an array around an axis perpendicular to the plane spanned by `dim1` and `dim2`, use first

```
% Largest dimension number we have to deal with.
nd = max( [ ndims(X) dim1 dim2 ] );

% Initialize subscript cell array.
v = {':'};
v = v(ones(nd,1));
```

then, depending on how to rotate, use

Rotate 90 degrees counterclockwise

```
v{dim2} = size(X,dim2):-1:1;
Y = X(v{:});
d = 1:nd;
d([ dim1 dim2 ]) = [ dim2 dim1 ];
Y = permute(X, d);
```

Rotate 180 degrees

```
v{dim1} = size(X,dim1):-1:1;
v{dim2} = size(X,dim2):-1:1;
Y = X(v{:});
```

Rotate 90 degrees clockwise

```
v{dim1} = size(X,dim1):-1:1;
Y = X(v{:});
d = 1:nd;
d([ dim1 dim2 ]) = [ dim2 dim1 ];
Y = permute(X, d);
```

If we want to rotate `n*90` degrees counterclockwise, we may merge the three cases above into

```
% Largest dimension number we have to deal with.
nd = max( [ ndims(A) dim1 dim2 ] );

% Initialize subscript cell array.
v = {':'};
v = v(ones(nd,1));

% Flip along appropriate dimensions.
if n == 1 | n == 2
   v{dim2} = size(A,dim2):-1:1;
end
if n == 2 | n == 3
   v{dim1} = size(A,dim1):-1:1;
end
B = A(v{:});

% Permute dimensions if appropriate.
if n == 1 | n == 3
   d = 1:nd;
   d([ dim1 dim2 ]) = [ dim2 dim1 ];
   B = permute( A, d );
end
```

7.4 Block-rotating 2D matrices

7.4.1 "Inner" vs "outer" block rotation

When talking about block-rotation of arrays, we have to differentiate between two different kinds of rotation. Lacking a better name I chose to call it "inner block rotation" and "outer block rotation".

Inner block rotation is a rotation of the elements within each block, preserving the position of each block within the array. Outer block rotation rotates the blocks but does not change the position of the elements within each block.

An example will illustrate: An inner block rotation 90 degrees counterclockwise will have the following effect

```
[ A B C        [ rot90(A) rot90(B) rot90(C)
  D E F    =>    rot90(D) rot90(E) rot90(F)
  G H I ]        rot90(G) rot90(H) rot90(I) ]
```

However, an outer block rotation 90 degrees counterclockwise will have the following effect

```
[ A B C        [ C F I
  D E F    =>    B E H
  G H I ]        A D G ]
```

In all the examples below, it is assumed that `X` is an `m`-by-`n` matrix of `p`-by-`q` blocks.

7.4.2 "Inner" block rotation 90 degrees counterclockwise

General case To perform the rotation

```
X = [ A B ...        [ rot90(A) rot90(B) ...
      C D ...    =>    rot90(C) rot90(D) ...
      ... ... ]          ...      ...    ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = Y(:,:,q:-1:1,:);                % or Y = Y(:,:,end:-1:1,:);
Y = permute( Y, [ 3 2 1 4 ] );
Y = reshape( Y, [ q*m/p p*n/q ] );
```

Special case: `m=p` To perform the rotation

```
[ A B ... ]  =>  [ rot90(A) rot90(B) ... ]
```

use

```
Y = reshape( X, [ p q n/q ] );
Y = Y(:,q:-1:1,:);                  % or Y = Y(:,end:-1:1,:);
Y = permute( Y, [ 2 1 3 ] );
Y = reshape( Y, [ q m*n/q ] );      % or Y = Y(:,:);
```

Special case: `n=q` To perform the rotation

```
X = [  A         [ rot90(A)
       B     =>    rot90(B)
      ... ]          ...    ]
```

use

```
Y = X(:,q:-1:1);                    % or Y = X(:,end:-1:1);
Y = reshape( Y, [ p m/p q ] );
Y = permute( Y, [ 3 2 1 ] );
Y = reshape( Y, [ q*m/p p ] );
```

7.4.3 “Inner” block rotation 180 degrees

General case To perform the rotation

```
X = [ A B ...        [ rot90(A,2) rot90(B,2) ...
      C D ...    =>    rot90(C,2) rot90(D,2) ...
      ... ... ]           ...        ...     ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = Y(p:-1:1,:,q:-1:1,:);            % or Y = Y(end:-1:1,:,end:-1:1,:);
Y = reshape( Y, [ m n ] );
```

Special case: `m=p` To perform the rotation

```
[ A B ... ]  =>  [ rot90(A,2) rot90(B,2) ... ]
```

use

```
Y = reshape( X, [ p q n/q ] );
Y = Y(p:-1:1,q:-1:1,:);              % or Y = Y(end:-1:1,end:-1:1,:);
Y = reshape( Y, [ m n ] );           % or Y = Y(:,:);
```

Special case: `n=q` To perform the rotation

```
X = [  A         [ rot90(A,2)
       B     =>    rot90(B,2)
      ... ]           ...     ]
```

use

```
Y = reshape( X, [ p m/p q ] );
Y = Y(p:-1:1,:,q:-1:1);              % or Y = Y(end:-1:1,:,end:-1:1);
Y = reshape( Y, [ m n ] );
```

7.4.4 "Inner" block rotation 90 degrees clockwise

General case To perform the rotation

```
X = [ A B ...        [ rot90(A,3) rot90(B,3) ...
      C D ...    =>    rot90(C,3) rot90(D,3) ...
      ... ... ]           ...        ...     ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = Y(p:-1:1,:,:,:);                % or Y = Y(end:-1:1,:,:,:);
Y = permute( Y, [ 3 2 1 4 ] );
Y = reshape( Y, [ q*m/p p*n/q ] );
```

Special case: `m=p` To perform the rotation

```
[ A B ... ]  =>  [ rot90(A,3) rot90(B,3) ... ]
```

use

```
Y = X(p:-1:1,:);                    % or Y = X(end:-1:1,:);
Y = reshape( Y, [ p q n/q ] );
Y = permute( Y, [ 2 1 3 ] );
Y = reshape( Y, [ q m*n/q ] );      % or Y = Y(:,:);
```

Special case: `n=q` To perform the rotation

```
X = [  A         [ rot90(A,3)
       B     =>    rot90(B,3)
      ... ]           ...     ]
```

use

```
Y = reshape( X, [ p m/p q ] );
Y = Y(p:-1:1,:,:);                  % or Y = Y(end:-1:1,:,:);
Y = permute( Y, [ 3 2 1 ] );
Y = reshape( Y, [ q*m/p p ] );
```

7.4.5 "Outer" block rotation 90 degrees counterclockwise

General case To perform the rotation

```
X = [ A B ...        [ ... ...
      C D ...    =>    B D ...
      ... ... ]        A C ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = Y(:,:,:,n/q:-1:1);               % or Y = Y(:,:,:,end:-1:1);
Y = permute( Y, [ 1 4 3 2 ] );
Y = reshape( Y, [ p*n/q q*m/p ] );
```

Special case: `m=p` To perform the rotation

```
[ A B ... ]  =>  [ ...
                   B
                   A  ]
```

use

```
Y = reshape( X, [ p q n/q ] );
Y = Y(:,:,n/q:-1:1);                 % or Y = Y(:,:,end:-1:1);
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ m*n/q q ] );
```

Special case: `n=q` To perform the rotation

```
X = [  A
       B     =>  [ A B ... ]
      ... ]
```

use

```
Y = reshape( X, [ p m/p q ] );
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ p n*m/p ] );       % or Y(:,:);
```

7.4.6 "Outer" block rotation 180 degrees

General case To perform the rotation

```
X = [ A B ...        [  ... ...
      C D ...    =>     ... D C
      ... ... ]         ... B A ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = Y(:,m/p:-1:1,:,n/q:-1:1);     % or Y = Y(:,end:-1:1,:,end:-1:1);
Y = reshape( Y, [ m n ] );
```

Special case: `m=p` To perform the rotation

```
[ A B ... ]  =>  [ ... B A ]
```

use

```
Y = reshape( X, [ p q n/q ] );
Y = Y(:,:,n/q:-1:1);                % or Y = Y(:,:,end:-1:1);
Y = reshape( Y, [ m n ] );          % or Y = Y(:,:);
```

Special case: `n=q` To perform the rotation

```
X = [  A         [ ...
       B     =>    B
      ... ]        A  ]
```

use

```
Y = reshape( X, [ p m/p q ] );
Y = Y(:,m/p:-1:1,:);                % or Y = Y(:,end:-1:1,:);
Y = reshape( Y, [ m n ] );
```

7.4.7 "Outer" block rotation 90 degrees clockwise

General case To perform the rotation

```
X = [ A B ...        [ ... C A
      C D ...    =>     ... D B
      ... ... ]        ... ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = Y(:,m/p:-1:1,:,:);                % or Y = Y(:,end:-1:1,:,:);
Y = permute( Y, [ 1 4 3 2 ] );
Y = reshape( Y, [ p*n/q q*m/p ] );
```

Special case: `m=p` To perform the rotation

```
[ A B ... ]  =>  [  A
                    B
                   ... ]
```

use

```
Y = reshape( X, [ p q n/q ] );
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ m*n/q q ] );
```

Special case: `n=q` To perform the rotation

```
X = [  A
       B     =>  [ ... B A ]
      ... ]
```

use

```
Y = reshape( X, [ p m/p q ] );
Y = Y(:,m/p:-1:1,:);                  % or Y = Y(:,end:-1:1,:);
Y = permute( Y, [ 1 3 2 ] );
Y = reshape( Y, [ p n*m/p ] );
```

7.5 Blocktransposing a 2D matrix

7.5.1 "Inner" blocktransposing

Assume `X` is an `m`-by-`n` matrix and you want to subdivide it into `p`-by-`q` submatrices and transpose as if each block was an element. E.g., if `A`, `B`, `C` and `D` are `p`-by-`q` matrices, convert

```
X = [ A B ...        [ A.' B.' ...
      C D ...    =>     C.' D.' ...
      ... ... ]        ... ... ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = permute( Y, [ 3 2 1 4 ] );
Y = reshape( Y, [ q*m/p p*n/q ] );
```

7.5.2 "Outer" blocktransposing

Assume X is an m-by-n matrix and you want to subdivide it into p-by-q submatrices and transpose as if each block was an element. E.g., if A, B, C and D are p-by-q matrices, convert

```
X = [ A B ...        [ A C ...
      C D ...    =>    B D ...
      ... ... ]        ... ... ]
```

use

```
Y = reshape( X, [ p m/p q n/q ] );
Y = permute( Y, [ 1 4 3 2 ] );
Y = reshape( Y, [ p*n/q q*m/p] );
```

8 Multiply arrays

8.1 Multiply each 2D slice with the same matrix (element-by-element)

Assume X is an m-by-n-by-p-by-q-by-... array and Y is an m-by-n matrix and you want to construct a new m-by-n-by-p-by-q-by-... array Z, where

```
Z(:,:,i,j,...) = X(:,:,i,j,...) .* Y;
```

for all i=1,...,p, j=1,...,q, etc. This can be done with nested for-loops, or by the following vectorized code

```
sx = size(X);
Z = X .* repmat(Y, [1 1 sx(3:end)]);
```

8.2 Multiply each 2D slice with the same matrix (left)

Assume X is an m-by-n-by-p-by-q-by-... array and Y is a k-by-m matrix and you want to construct a new k-by-n-by-p-by-q-by-... array Z, where

```
Z(:,:,i,j,...) = Y * X(:,:,i,j,...);
```

for all i=1,...,p, j=1,...,q, etc. This can be done with nested for-loops, or by the following vectorized code

```
sx = size(X);
sy = size(Y);
Z = reshape(Y * X(:,:), [sy(1) sx(2:end)]);
```

8.3 Multiply each 2D slice with the same matrix (right)

Assume X is an m-by-n-by-p-by-q-by-... array and Y is an n-by-k matrix and you want to construct a new m-by-n-by-p-by-q-by-... array Z, where

```
Z(:,:,i,j,...) = X(:,:,i,j,...) * Y;
```

for all i=1,...,p, j=1,...,q, etc. This can be done with nested for-loops, or by the following vectorized code

```
sx = size(X);
sy = size(Y);
dx = ndims(X);
Xt = reshape(permute(X, [1 3:dx 2]), [prod(sx)/sx(2) sx(2)]);
Z2 = Xt * Y;
Z2 = permute(reshape(Z2, [sx([1 3:dx]) sy(2)]), [1 dx 2:dx-1]);
```

The third line above builds a 2D matrix which is a vertical concatenation (stacking) of all 2D slices `X(:,:,i,j,...)`. The fourth line does the actual multiplication. The fifth line does the opposite of the third line.

8.4 Multiply matrix with every element of a vector

Assume `X` is an `m`-by-`n` matrix and `v` is a row vector with length `p`. How does one write

```
Y = zeros(m, n, p);
for i = 1:p
   Y(:,:,i) = X * v(i);
end
```

with no for-loop? One way is to use

```
Y = reshape(X(:)*v, [ m n p ]);
```

8.5 Multiply each 2D slice with corresponding element of a vector

Assume `X` is an `m`-by-`n`-by-`p` array and `v` is a row vector with length `p`. How does one write

```
Y = zeros(m, n, p);
for i = 1:p
   Y(:,:,i) = X(:,:,i) * v(i);
end
```

with no for-loop? One way is to use

```
Y = X .* repmat(reshape(v, [1 1 p]), [ m n ]);
```

8.6 Outer product of all rows in a matrix

Assume `X` is an `m`-by-`n` matrix. How does one create an `n`-by-`n`-by-`m` matrix `Y` so that, for all `i` from `1` to `m`,

```
Y(:,:,i) = X(i,:)' * X(i,:);
```

The obvious for-loop solution is

```
Y = zeros(n, n, m);
for i = 1:m
   Y(:,:,i) = X(i,:)' * X(i,:);
end
```

a non-for-loop solution is

```
j = 1:n;
Y = reshape(repmat(X', n, 1) .* X(:,j(ones(n, 1),:)).', [n n m]);
```

Note the use of the non-conjugate transpose in the second factor to ensure that it works correctly also for complex matrices.

8.7 Keeping only diagonal elements of multiplication

Assume `X` and `Y` are two `m`-by-`n` matrices and that `W` is an `n`-by-`n` matrix. How does one vectorize the following for-loop

```
Z = zeros(m, 1);
for i = 1:m
   Z(i) = X(i,:)*W*Y(i,:)';
end
```

Two solutions are

```
Z = diag(X*W*Y');                    % (1)
Z = sum(X*W.*conj(Y), 2);            % (2)
```

Solution (1) does a lot of unnecessary work, since we only keep the `n` diagonal elements of the `n^2` computed elements. Solution (2) only computes the elements of interest and is significantly faster if `n` is large.

9 Divide arrays

9.1 Divide each 2D slice with the same matrix (element-by-element)

Assume `X` is an `m`-by-`n`-by-`p`-by-`q`-by-... array and `Y` is an `m`-by-`n` matrix and you want to construct a new `m`-by-`n`-by-`p`-by-`q`-by-... array `Z`, where

```
Z(:,:,i,j,...) = X(:,:,i,j,...) ./ Y;
```

for all `i=1,...,p`, `j=1,...,q`, etc. This can be done with nested for-loops, or by the following vectorized code

```
sx = size(X);
Z = X./repmat(Y, [1 1 sx(3:end)]);
```

9.2 Divide each 2D slice with the same matrix (left)

Assume `X` is an `m`-by-`n`-by-`p`-by-`q`-by-... array and `Y` is an `m`-by-`m` matrix and you want to construct a new `m`-by-`n`-by-`p`-by-`q`-by-... array `Z`, where

```
Z(:,:,i,j,...) = Y \ X(:,:,i,j,...);
```

for all `i=1,...,p`, `j=1,...,q`, etc. This can be done with nested for-loops, or by the following vectorized code

```
Z = reshape(Y\X(:,:), size(X));
```

9.3 Divide each 2D slice with the same matrix (right)

Assume `X` is an `m`-by-`n`-by-`p`-by-`q`-by-... array and `Y` is an `m`-by-`m` matrix and you want to construct a new `m`-by-`n`-by-`p`-by-`q`-by-... array `Z`, where

```
Z(:,:,i,j,...) = X(:,:,i,j,...) / Y;
```

for all `i=1,...,p`, `j=1,...,q`, etc. This can be done with nested for-loops, or by the following vectorized code

```
sx = size(X);
dx = ndims(X);
Xt = reshape(permute(X, [1 3:dx 2]), [prod(sx)/sx(2) sx(2)]);
Z = Xt/Y;
Z = permute(reshape(Z, sx([1 3:dx 2])), [1 dx 2:dx-1]);
```

The third line above builds a 2D matrix which is a vertical concatenation (stacking) of all 2D slices `X(:,:,i,j,...)`. The fourth line does the actual division. The fifth line does the opposite of the third line.

The five lines above might be simplified a little by introducing a dimension permutation vector

```
sx = size(X);
dx = ndims(X);
v = [1 3:dx 2];
Xt = reshape(permute(X, v), [prod(sx)/sx(2) sx(2)]);
Z = Xt/Y;
Z = ipermute(reshape(Z, sx(v)), v);
```

If you don't care about readability, this code may also be written as

```
sx = size(X);
dx = ndims(X);
v = [1 3:dx 2];
Z = ipermute(reshape(reshape(permute(X, v), ...
       [prod(sx)/sx(2) sx(2)])/Y, sx(v)), v);
```

10 Calculating distances

10.1 Euclidean distance

The Euclidean distance from $\mathbf{x}_i$ to $\mathbf{y}_j$ is

$$\mathbf{d}_{ij} = ||\mathbf{x}_i - \mathbf{y}_j|| = \|\mathbf{x}_i - \mathbf{y}_j\| = \sqrt{(x_{1i} - y_{1j})^2 + \cdots + (x_{pi} - y_{pj})^2}$$

10.2 Distance between two points

To calculate the Euclidean distance from a point represented by the vector `x` to another point represeted by the vector `y`, use one of

```
d = norm(x-y);
d = sqrt(sum(abs(x-y).^2));
```

10.3 Euclidean distance vector

Assume `X` is an `m`-by-`p` matrix representing `m` points in `p`-dimensional space and `y` is a `1`-by-`p` vector representing a single point in the same space. Then, to compute the `m`-by-`1` distance vector `d` where `d(i)` is the Euclidean distance between `X(i,:)` and `y`, use

```
d = sqrt(sum(abs(X - repmat(y, [m 1])).^2, 2));
d = sqrt(sum(abs(X - y(ones(m,1),:)).^2, 2));   % inline call to repmat
```

10.4 Euclidean distance matrix

Assume X is an m-by-p matrix representing m points in p-dimensional space and Y is an n-by-p matrix representing another set of points in the same space. Then, to compute the m-by-n distance matrix D where D(i,j) is the Euclidean distance X(i,:) between Y(j,:), use

```
D = sqrt(sum(abs(   repmat(permute(X, [1 3 2]), [1 n 1]) ...
                  - repmat(permute(Y, [3 1 2]), [m 1 1]) ).^2, 3));
```

The following code inlines the call to repmat, but requires to temporary variables unless one doesn't mind changing X and Y

```
Xt = permute(X, [1 3 2]);
Yt = permute(Y, [3 1 2]);
D = sqrt(sum(abs(   Xt( :, ones(1, n), : ) ...
                  - Yt( ones(1, m), :, : ) ).^2, 3));
```

10.5 Special case when both matrices are identical

If X and Y are identical one may use the following, which is nothing but a rewrite of the code above

```
D = sqrt(sum(abs(   repmat(permute(X, [1 3 2]), [1 m 1]) ...
                  - repmat(permute(X, [3 1 2]), [m 1 1]) ).^2, 3));
```

One might want to take advantage of the fact that D will be symmetric. The following code first creates the indexes for the upper triangular part of D. Then it computes the upper triangular part of D and finally lets the lower triangular part of D be a mirror image of the upper triangular part.

```
[ i j ] = find(triu(ones(m), 1));     % Trick to get indices.
D = zeros(m, m);                      % Initialise output matrix.
D( i + m*(j-1) ) = sqrt(sum(abs( X(i,:) - X(j,:) ).^2, 2));
D( j + m*(i-1) ) = D( i + m*(j-1) );
```

10.6 Mahalanobis distance

The Mahalanobis distance from a vector $\mathbf{y}_j$ to the set $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{n_x}\}$ is the distance from $\mathbf{y}_j$ to $\bar{\mathbf{x}}$, the centroid of $\mathcal{X}$, weighted according to C_x, the variance matrix of the set $\mathcal{X}$. I.e.,

$$\mathbf{d}_j^2 = (\mathbf{y}_j - \bar{\mathbf{x}})' \mathbf{C}_\mathbf{x}^{-1} (\mathbf{y}_j - \bar{\mathbf{x}})$$

where

$$\bar{\mathbf{x}} = \frac{1}{n_x} \sum_{i=1}^{n} \mathbf{x}_i \qquad \text{and} \qquad \mathbf{C}_\mathbf{x} = \frac{1}{n_x - 1} \sum_{i=1}^{n_x} (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})'$$

Assume Y is an ny-by-p matrix containing a set of vectors and X is an nx-by-p matrix containing another set of vectors, then the Mahalanobis distance from each vector Y(j,:) (for j=1,...,ny) to the set of vectors in X can be calculated with

```
nx = size(X, 1);              % size of set in X
ny = size(Y, 1);              % size of set in Y
m = mean(X);
C = cov(X);
d = zeros(ny, 1);
for j = 1:ny
   d(j) = (Y(j,:) - m) / C * (Y(j,:) - m)';
end
```

which is computed more efficiently with the following code which does some inlining of functions (`mean` and `cov`) and vectorization

```
nx = size(X, 1);                    % size of set in X
ny = size(Y, 1);                    % size of set in Y

m  = sum(X, 1)/nx;                  % centroid (mean)
Xc = X - m(ones(nx,1),:);           % distance to centroid of X
C  = (Xc' * Xc)/(nx - 1);           % variance matrix
Yc = Y - m(ones(ny,1),:);           % distance to centroid of X
d  = sum(Yc/C.*Yc, 2));             % Mahalanobis distances
```

In the complex case, the last line has to be written as

```
d  = real(sum(Yc/C.*conj(Yc), 2));    % Mahalanobis distances
```

The call to `conj` is to make sure it also works for the complex case. The call to `real` is to remove "numerical noise".

The Statistics Toolbox contains the function `mahal` for calculating the Mahalanobis distances, but `mahal` computes the distances by doing an orthogonal-triangular (QR) decomposition of the matrix `C`. The code above returns the same as `d = mahal(Y, X)`.

Special case when both matrices are identical If `Y` and `X` are identical in the code above, the code may be simplified somewhat. The for-loop solution becomes

```
n = size(X, 1);              % size of set in X
m = mean(X);
C = cov(X);
d = zeros(n, 1);
for j = 1:n
   d(j) = (Y(j,:) - m) / C * (Y(j,:) - m)';
end
```

which is computed more efficiently with

```
n = size(x, 1);
m = sum(x, 1)/n;                    % centroid (mean)
c = x - m(ones(n,1),:);             % distance to centroid of X
C = (c' * c)/(n - 1);               % variance matrix
d = sum(c/C.*c, 2);                 % Mahalanobis distances
```

again, to make it work in the complex case, the last line must be written as

```
d = real(sum(c/C.*conj(c), 2));       % Mahalanobis distances
```

11 Statistics, probability and combinatorics

11.1 Discrete uniform sampling with replacement

To generate an array `X` with size vector `s`, where `X` contains a random sample from the numbers `1,...,n` use

```
X = ceil(n*rand(s));
```

To generate a sample from the numbers `a,...,b` use

```
X = a + floor((b-a+1)*rand(s));
```

11.2 Discrete weighted sampling with replacement

Assume `p` is a vector of probabilities that sum up to `1`. Then, to generate an array `X` with size vector `s`, where the probability of `X(i)` being `i` is `p(i)` use

```
m = length(p);                % number of probabilities
c = cumsum(p);                % cumulative sum
R = rand(s);
X = ones(s);
for i = 1:m-1
   X = X + (R > c(i));
end
```

Note that the number of times through the loop depends on the number of probabilities and not the sample size, so it should be quite fast even for large samples.

11.3 Discrete uniform sampling without replacement

To generate a sample of size `k` from the integers `1,...,n`, one may use

```
X = randperm(n);
x = X(1:k);
```

although that method is only practical if `N` is reasonably small.

11.4 Combinations

"Combinations" is what you get when you pick `k` elements, without replacement, from a sample of size `n`, and consider the order of the elements to be irrelevant.

11.4.1 Counting combinations

The number of ways to pick `k` elements, without replacement, from a sample of size `n` is $\binom{n}{k}$ which is calculate with

```
c = nchoosek(n, k);
```

one may also use the definition directly

```
k = min(k, n-k);      % use symmetry property
c = round(prod( ((n-k+1):n) ./ (1:k) ));
```

which is safer than using

```
k = min(k, n-k);      % use symmetry property
c = round( prod((n-k+1):n) / prod(1:k) );
```

which may overflow. Unfortunately, both `n` and `k` have to be scalars. If `n` and/or `k` are vectors, one may use the fact that

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}$$

and calculate this in with

```
round(exp(gammaln(n+1) - gammaln(k+1) - gammaln(n-k+1)))
```

where the `round` is just to remove any "numerical noise" that might have been introduced by `gammaln` and `exp`.

11.4.2 Generating combinations

To generate a matrix with all possible combinations of `n` elements taken `k` at a time, one may use the MATLAB function `nchoosek`. That function is rather slow compared to the `choosenk` function which is a part of Mike Brookes' Voicebox (Speech recognition toolbox) whose homepage is at http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html

For the special case of generating all combinations of *n* elements taken 2 at a time, there is a neat trick

```
[ x(:,2) x(:,1) ] = find(tril(ones(n), -1));
```

11.5 Permutations

11.5.1 Counting permutations

```
p = prod(n-k+1:n);
```

11.5.2 Generating permutations

To generate a matrix with all possible permutations of `n` elements, one may use the function `perms`. That function is rather slow compared to the `permutes` function which is a part of Mike Brookes' Voicebox (Speech recognition toolbox) whose homepage is at
http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html

12 Miscellaneous

This section contains things that don't fit anywhere else.

12.1 Creating index vector from index limits

Given two index vectors `lo` and `hi`. How does one create another index vector

```
x = [lo(1):hi(1) lo(2):hi(2) ...]
```

A straightforward for-loop solution is

```
m = length(lo);       % length of input vectors
x = [];               % initialize output vector
for i = 1:m
   x = [ x lo(i):hi(i) ];
end
```

which unfortunately requires a lot of memory copying since a new `x` has to be allocated each time through the loop. A better for-loop solution is one that allocates the required space and then fills in the elements afterwards. This for-loop solution above may be several times faster than the first one

```
m = length(lo);       % length of input vectors
d = hi - lo + 1;      % length of each "run"
n = sum(d);           % length of output vector
c = cumsum(d);        % last index in each run

x = zeros(1, n);      % initialize output vector
```

```
for i = 1:m
   x(c(i)-d(i)+1:c(i)) = lo(i):hi(i);
end
```

Neither of the for-loop solutions above can compete with the the solution below which has no for-loops. It uses `cumsum` rather than the `:` to do the incrementing in each run and may be many times faster than the for-loop solutions above.

```
m = length(lo);       % length of input vectors
d = hi - lo + 1;      % length of each "run"
n = sum(d);           % length of output vector

x = ones(1, n);
x(1) = lo(1);
x(1+cumsum(d(1:end-1))) = lo(2:m)-hi(1:m-1);
x = cumsum(x);
```

If fails, however, if `lo(i)>hi(i)` for any `i`. Such a case will create an empty vector anyway, so the problem can be solved by a simple pre-processing step which removing the elements for which `lo(i)>hi(i)`

```
i = lo <= hi;
lo = lo(i);
hi = hi(i);
```

There also exists a one-line solution which is clearly compact, but not as fast as the no-for-loop solution above

```
x = eval(['[' sprintf('%d:%d,', [lo ; hi]) ']']);
```

12.2 Matrix with different incremental runs

Given a vector of positive integers

```
a = [ 3 2 4 ];
```

How does one create the matrix where the `i`th column contains the vector `1:a(i)` possibly padded with zeros:

```
b = [ 1 1 1
      2 2 2
      3 0 3
      0 0 4 ];
```

One way is to use a for-loop

```
n = length(a);
b = zeros(max(a), n);
for k = 1:n
   t = 1:a(k);
   b(t,k) = t(:);
end
```

and here is a way to do it without a for-loop

```
[bb aa] = ndgrid(1:max(a), a);
b = bb .* (bb <= aa)
```

or the more explicit

```
m = max(a);
aa = a(:)';
aa = aa(ones(m, 1),:);
bb = (1:m)';
bb = bb(:,ones(length(a), 1));
b = bb .* (bb <= aa);
```

To do the same, only horizontally, use

```
[aa bb] = ndgrid(a, 1:max(a));
b = bb .* (bb <= aa)
```

or

```
m = max(a);
aa = a(:);
aa = aa(:,ones(m, 1));
bb = 1:m;
bb = bb(ones(length(a), 1),:);
b = bb .* (bb <= aa);
```

12.3 Finding indexes

How does one find the index of the last non-zero element in each row. That is, given

```
x = [ 0 9 7 0 0 0
      5 0 0 6 0 3
      0 0 0 0 0 0
      8 0 4 2 1 0 ];
```

how dows one obtain the vector

```
j = [ 3
      6
      0
      5 ];
```

One way is of course to use a for-loop

```
m = size(x, 1);
j = zeros(m, 1);
for i = 1:m
   k = find(x(i,:) ~= 0);
   if length(k)
      j(i) = k(end);
   end
end
```

or

```
m = size(x, 1);
j = zeros(m, 1);
for i = 1:m
   k = [ 0 find(x(i,:) ~= 0) ];
   j(i) = k(end);
end
```

but one may also use

```
j = sum(cumsum((x(:,end:-1:1) ~= 0), 2) ~= 0, 2);
```

To find the index of the last non-zero element in each column, use

```
i = sum(cumsum((x(end:-1:1,:) ~= 0), 1) ~= 0, 1);
```

12.4 Run-length encoding and decoding

12.4.1 Run-length encoding

Assuming `x` is a vector

```
x = [ 4 4 5 5 5 6 7 7 8 8 8 8 ]
```

and one wants to obtain the two vectors

```
l = [ 2 3 1 2 4 ];        % run lengths
v = [ 4 5 6 7 8 ];        % values
```

one can get the run length vector `l` by using

```
l = diff([ 0 find(x(1:end-1) ~= x(2:end)) length(x) ]);
```

and the value vector `v` by using one of

```
v = x([ find(x(1:end-1) ~= x(2:end)) length(x) ]);
v = x(logical([ x(1:end-1) ~= x(2:end) 1 ]));
```

These two steps can be combined into

```
i = [ find(x(1:end-1) ~= x(2:end)) length(x) ];
l = diff([ 0 i ]);
v = x(i);
```

12.4.2 Run-length decoding

Given the run-length vector `l` and the value vector `v`, one may create the full vector `x` by using

```
i = cumsum([ 1 l ]);
j = zeros(1, i(end)-1);
j(i(1:end-1)) = 1;
x = v(cumsum(j));
```