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.,

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

where

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

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));
```