Introduction to multivariate data analysis

In this part of the course we shall use both Matlab and R. For your convenience a review of Matlab basics is given in the appendix. Most commands used in this text work similarly in Octave or FreeMat which you can freely download. If you don't have Matlab, use either Octave or FreeMat for practising the elements of the language. From Chapter 3 onwards you need the so-called DA-toolbox which can be copied from

http://users.metropolia.fi/~velimt/TG08S/Environmetrics_II/

Remember to give an addpath command that sets the folder you put files into in your Matlab folder path. These functions are not guaranteed to work in Octave or FreeMat, but you can always try.

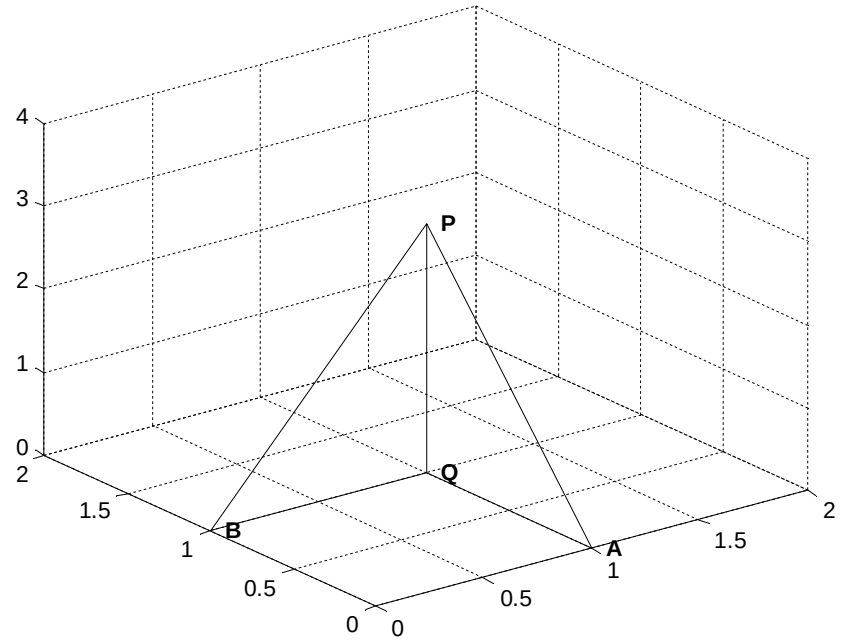
1. PCA and PLS

1.1 PCA

Mathematically principal component analysis (PCA) is a very simple thing. It simply means finding those orthogonal directions on which the projected data gives maximum variance. This also means that we decompose a $n \times m$ data matrix X into a matrix product **TP'** (**'** means the matrix transpose, i.e. rows and columns are interchanged), where **P** is $p \times m$ and contains m orthogonal direction vectors (so called loading vectors) and **T** is $n \times m$ and contains the scalar projections (so called score vectors) on each of the loading vectors. According to basic linear algebra, the PCA-decomposition is most easily done using so called singular value decomposition (SVD). We will use this result without going to the mathematical details. But let us first try to understand the basic ideas behind PCA.

The first projection vector, i.e. the first principal component (PC), in PCA is chosen in a way that the variance of the points projected onto the projection vector, i.e. the variance of the scalar projections, is maximized. The line defined by the first PC vector is called the first principal axis. After this, the next PC is chosen so that is orthogonal (perpendicular) to the first one and again maximizing the variance of the scalar projections of the original points. This procedure is continued till all dimensions of the original space are used, and as a result we obtain a new orthogonal coordinate system that is a rotation aof the original one. The new system gives us a rotated view of the original space.

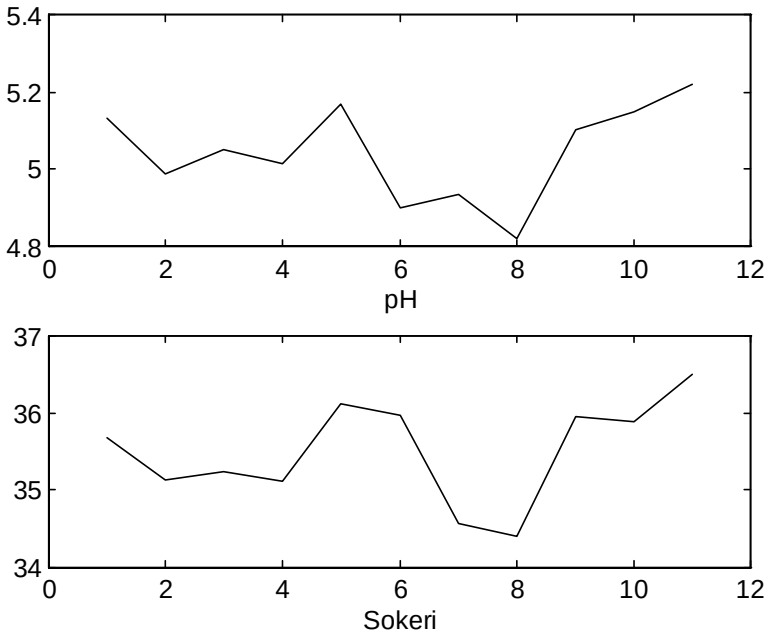
The following figure depicts the idea of orthogonal projections in 3D.



The original point is P, A is the orthogonal projection on a line that represents the first principal axes, and B is the orthogonal projection on a line that represents the second principal axes. The points A and B give the coordinates of the point Q that is the orthogonal projection onto the plane defined by the first two principal axes. A typical PCA visualization of some data is a graph where all points are projected onto the plane formed by the first two principal components.

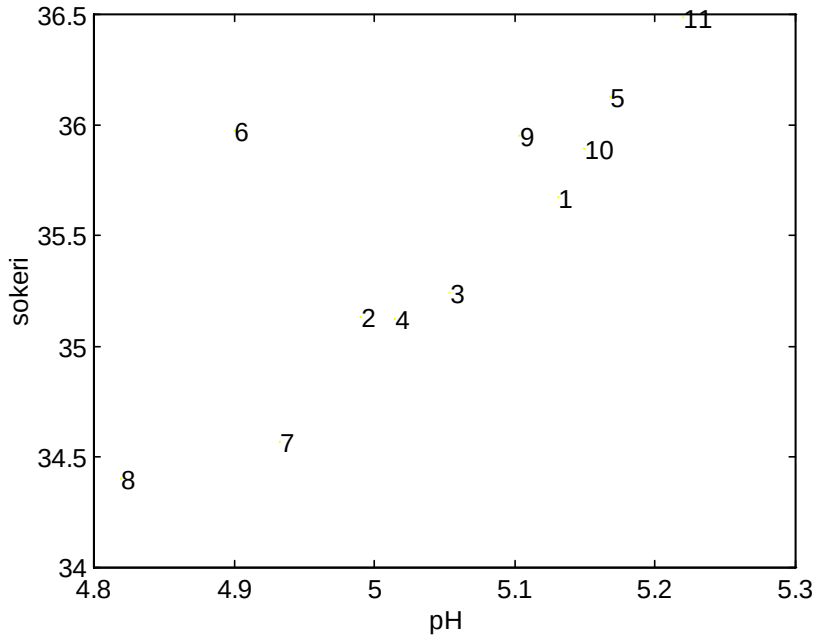
Example 1

Let us consider the following 11 samples taken in time order from a process where pH and sugar content are relevant quantities from the product quality point of view.



Can you detect an observation that is significantly different from the others? You should try it before reading the text any further!

It is rather obvious that there is such an observation but it is not easy to detect by a quick glimpse on the graph. However, is you notice the clear correlation between the two variables, you will certainly also notice that this correlation is broken in the 6'th observation. Now, the question is could there be some other way of graphing the data, in the way that the outlying observation would immediately stand out. Let us consider the observations as points in a two-dimensional space, and plot them accordingly:



Though this is not a PC plot it is easy to see how this plot should be rotated in order to get the PC plot.

Almost always the data is centered, i.e. the mean value of each variable is subtracted from each column, before PCA, and quite often the data is also scaled by dividing the columns with the standard deviations of each variable. Many other data preprocessing alternatives exist as well, and especially the choosing the method of scaling is strongly data dependent.

The scores and loadings are constructed in decreasing order of variance so that we can approximate **X** by taking only *a* first columns of the matrices **T** ($\mathbf{T}_a$) and **P** ($\mathbf{P}_a$), i.e. $\mathbf{X} \approx \mathbf{T}_a\mathbf{P}_a'$.

The usefulness of PCA is based on such approximations, because it means that **X** can be replaced by $\mathbf{T}_a$, which has only few columns (variables) compared to **X**. In addition, experience has shown, that plotting usually the two first scores against each other reveals most of the structure of **X**. Such plots are simply projections of the original data onto a two dimensional plane of the two first loading vectors and they are called score-plots.

Writing down a function for PCA in Matlab is fairly simple (the version in the DA toolbox is a more complicated one):

```
 function [T,P,R2] = pca(X);
%function [T,P,R2] = pca(X);
%the function computes the PCA-decomposition of the matrix 'X'
%for all dimensions.
%
%INPUT:      X     The data matrix. Each observations is a row in X.
%
%OUTPUT:     T     The 'scores' of the observations, i.e. the
%                  projections
%                  of the observations on the principal axes.
%           P      The  principal axes
%           R2     The coefficient of determination
%
%NOTE  To get the correct pca interpretation, center 'X' before
%      calling the function.

 [m,n] = size(X);
 if m < n
    [P,S,U] = svd(X',0);
 else
    [U,S,P] = svd(X,0);
 end

 ssum = cumsum(diag(S).^2); nn = length(ssum);
 R2   = 100*ssum/ssum(nn);
 T    = U*S;
```

The vector of length *a* R2 above gives the percentage of explained variance in X using 1 to *a* principal components.

Example

The following data are from a granulation process and we use a data matrix (f) consisting of 161 particle size distributions of the product. Let us open a new m-file and we shall write down the following commands (using the DA version of pca):

```
load TR %load the data

[T,P,stats] = pca(F); %the PCA-decomposition

figure(1)
plotpca(T,1,2,1:161,1:161,'-') %the score-plot
```

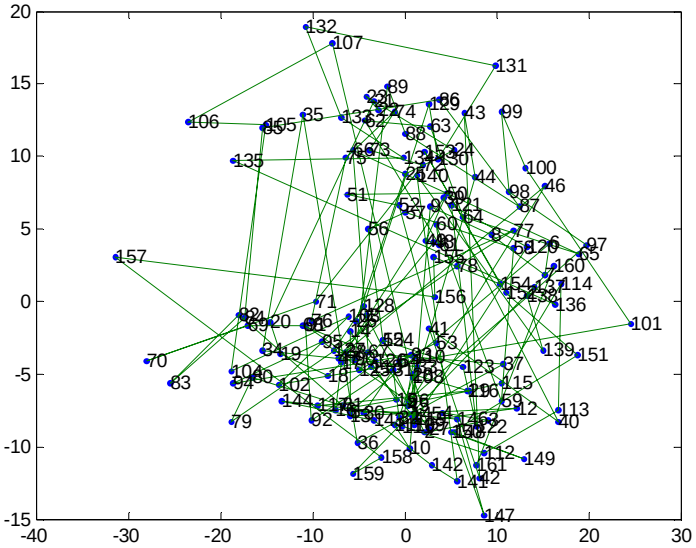
The stats struct contains an approximation of **X** if the desired dimension is given as the second argument. The commands below shows explicitly how such an approximation can be calculated.

```
F2 = T(:,1:2)*P(:,1:2)'; %an approximation using 2 PC's
f2 = center(F2,mf,-1); %uncentering
r2(f,f2) %column-wise explained variances
```

As an output we'll get

```
>> PCA_example
ans =
        22.56       52.698       98.779        93.07       89.373
        89.201      47.611       4.2582
```

We notice that distributions are well explained by two PC's except for the tails. We also get the figure:



We shall play more with this interesting data interactively during the course.

Another way of visualizing results of PCA is to make a so-called bi-plot in which both the scores and the loadings are plotted in the same graph. This is done so that the score plot is supplemented with the loadings, represented as positional vectors for each variable individually. The interpretation is that when a variable has a high value (compared to the average), the scores tend to move into the direction of the corresponding loading vector, and if a variable has a low value (compared to the average), the scores tend to move into the opposite direction of the corresponding loading vector. The biplot technique can used in most projection based multivariate methods, e.g. PCR, PLS or CA.

1.2 PCR (Principal Component Regression)

PCR means simply using ordinary multiple linear regression in explaining **Y** by **X** by replacing **X** with $\mathbf{T}_a$. Sometimes PCA is also done on Y as well before regression analysis. We don't illustrate this more here and, instead, we'll introduce so called partial least squares regression (PLS or PLSR).

1.3 PLS

The basic idea behind PLS is similar to PCA. The difference is in the way how PC's are constructed. In PCA we use the maximum variance criterion, but in PLS we use maximal covariance (between X- and Y-scores) criterion. The mathematics are again simple: the PC's are obtained from the SVD of $\mathbf{Y'X}$. There are alternative ways, most common of which is called the NIPALS algori-

thm, but in Matlab the SVD-based solution is simplest and usually also the fastest. The method is probably best understood by an example:

Example

The data contains of 324 NIR-spectra of 18 different batch reactions (Y) and 324 concentrations of 4 chemical compounds (C). The goal is a calibration model for predicting the concentrations from the spectra.

Let us write down the following commands (you can get help on each of the functions used using the help function-name command)

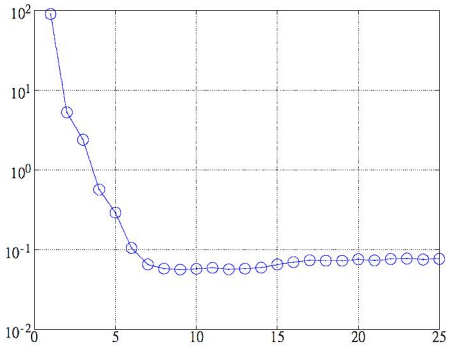
```
load data_for_PLS

X = Y; %rename the matrices according to
Y = C; %normal conventions
n = size(X,1);
i = 1:n;

X1 = X(1:305,:); %the last 19 observations
Y1 = Y(1:305,:); %are left as a test set

Xtest = X(306:end,:);
Ytest = Y(306:end,:);
itest = i(306:end);

%cross-validation in two randomly chosen parts
[stats] = crosplsq(X1,Y1,1:25,2,1);

figure(1)
semilogy(stats.press,'o-') %search for smallest press
grid on

% Make the model
dim = 10;
[b,Yfit,plspar] = plsreg(X1,Y1,dim,1);
% Predict the test set
[b,Ypred] = plsreg(Xtest,[],dim,1,plspar);

r2(Ytest,Ypred)

%plot of the predictions and some enhancements
%to the some features of Matlab-graphics
figure(2)
plot(itest,Ytest,'o',itest,Ypred,'LineWidth',2,'MarkerSize',10)
break
legend('A/data','B/data','C/data','D/data','A/pred',...
     'B/pred','C/pred','D/pred',10)
set(gca,'FontSize',14)
```

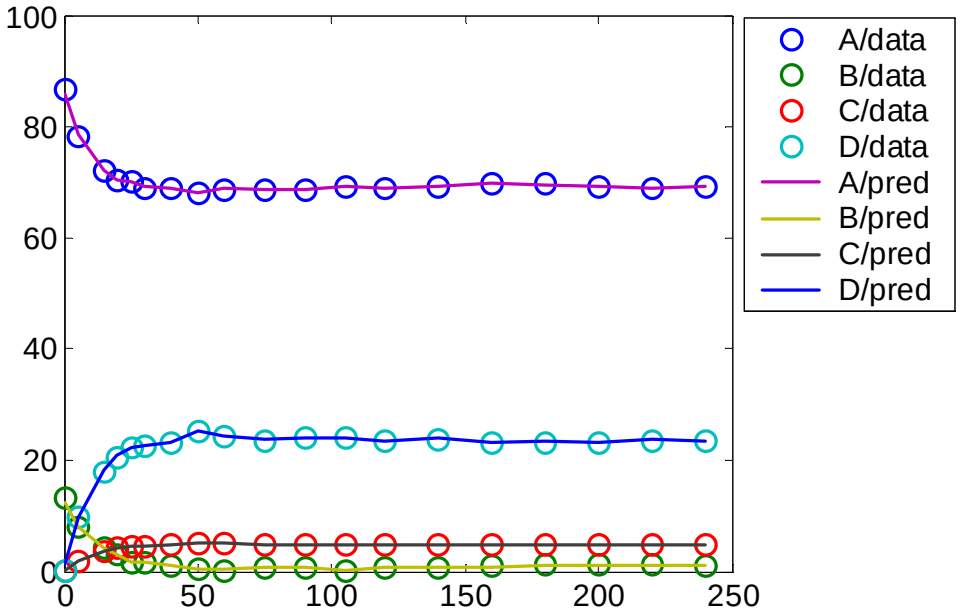
We get

```
>> PLS_example
ans =
```

```
99.774      99.289      99.686      99.663
```

and the figures below





1.4 CA (Correspondence Analysis)

Correspondence analysis (CA) is a popular multivariate method in many ecological and environmental studies. Technically it is PCA with scaling of both rows and columns of the data matrix. In the scaling, the columns are divided by the square roots of their sums, and similarly the rows are divided by the square roots of their sums. There are many different variants of the scaling scheme. In many variants, the data matrix is also centered using so-called expected frequencies as in the classical χ^2- test of independence.

As an example, we shall go through a demo (in lab) of an ecological study about Finnish lakes.

2. Clustering and discrimination

In this chapter we shall use R because it has some nice functions for these analyses. Actually, R has also functions for PCA and PLS, but I prefer to use my own functions, because I have data for the examples in Matlab format.

Cluster analysis is a method for finding groups (clusters) in multivariate data. Actually there are many different clustering methods, and we shal get acquainted with the two most popular ones, i.e. hierarchical and K-means clustering. In general, cluster analysis is an *unsupervised* method, which means that there is no prior knowledge about the clusters, or if clusters are found, there is no prior knowledge about the causes. Of course, once clusters have recognized, they are tried to be explained.

The basic idea in hierarchical clustering is link points that are close together into the same 'branch' in a tree representation of the distances. There are many alternative measures of the distance. Let us denote the variables constituting the data matrix **X** by x_1, x_2, ..., x_p . Each row in the data matrix represents a point in an *p*-dimensional space, i.e. the *i*'th point is $(x_{i1}, x_{i2}, ..., x_{ip})$. The most common distances are (the formulae are given the distance between the *i*'th and the *j*'th point).

1. Euclidian distance $d_{ij} = \sqrt{\sum_{k=1}^{p} (x_{ik} - x_{jk})^2}$
2. weighted Euclidian distance $d_{ij} = \sqrt{\sum_{k=1}^{p} w_k (x_{ik} - x_{jk})^2}$
3. Mahalanobis distance $d_{ij} = \sqrt{\sum_{k=1}^{p} \sum_{m=1}^{p} w_{km} (x_{ik} - x_{jk})(x_{im} - x_{jm})}$ where the multipliers w_{km} are the elements of the inverse of the sample covariance matrix of the data.
4. Minkowski distance $d_{ij} = \left(\sum_{k=1}^{p} |x_{ik} - x_{jk}|^{\lambda} \right)^{\frac{1}{\lambda}}$. The special case $\lambda = 1$, is called the Manhattan or the city-block distance (why?).

Instead of distances clustering can be based on *similarities*. A typical measure of similarity is the correlation coefficient between two vectors (points). A similarity can always be transformed into a *dissimilarity* and vice versa, and a distance can always be interpreted as a measure of dissimilarity. For example, a correlation coefficient *R* can be transformed into a distance by the transformation $\sqrt{\frac{R^2 - 1}{R^2}}$.

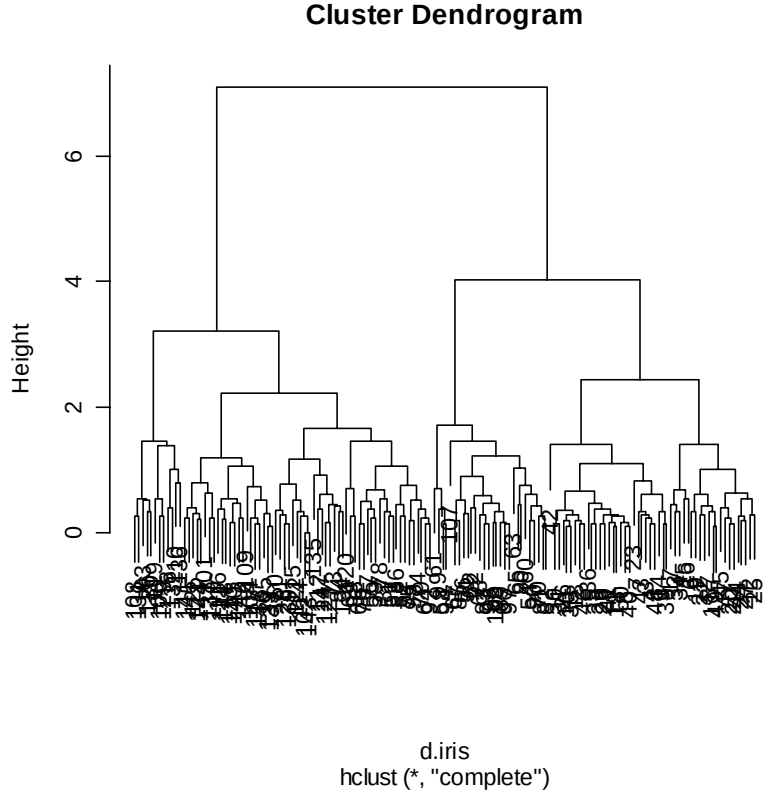
Example

We shall use the classical Fisher's iris data as an example. The data consists of 150 samples of iris flowers whose petal and sepal lengths and widths have been measured. This data can be used as an example for both supervised an unsupervised methods because the variety of each sample is known: the first 50 are of the variety setosa, the next 50 are that of versicolor, and the last 50 belong to the virginica variety.

Hierarchical clustering is fairly simple in R. We simply give the following commands:

```
data(iris)
X  =  as.matrix(iris[,1:4])
d.iris = dist(X)
iris.hclust = hclust(d.iris)
plot(iris.hclust)
```

Note that the data must be transformed into a matrix, because the distances must be first calculated using the function dist which operates only on matrices. The default of dist is that it calculates Euclidian distances. The last two commands carry out the hierarchical clustering and plot the results.



The dendrogram ('tree') shows two clear groups, but the subgroups are nor so clear.

The k-means clustering method differ from hierarchical clustering in the way that it requires an assumption about the number of groups (clusters), *k*. Then the algorithm finds *k* optimal group centers. The idea is very simple; the following description is taken from Wikipedia:

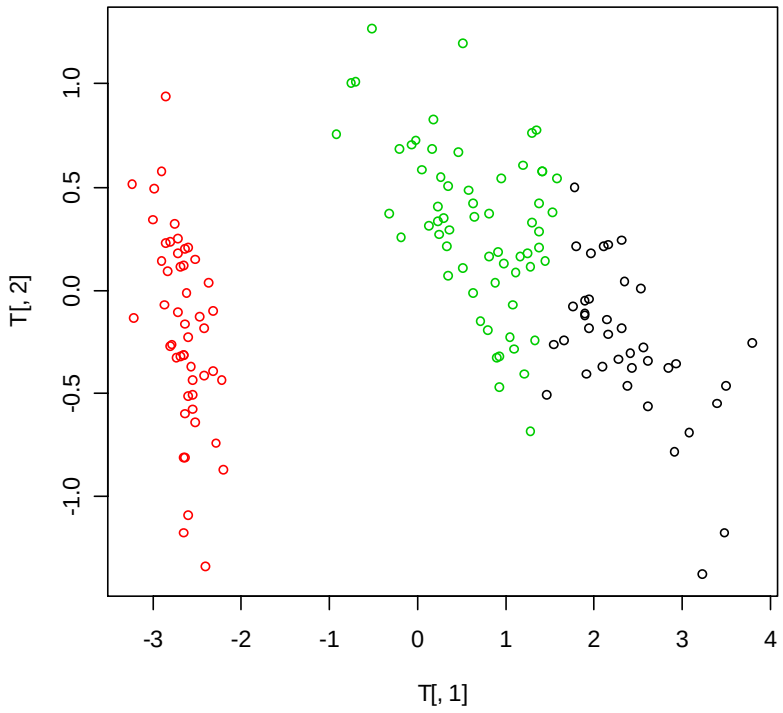
Given a set of observations (x_1, x_2, …, x_n), where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into *k* sets ($k \in n$) $S = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares (WCSS):

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{x_j \in S_i} |x_j - \mu_i|^2$$

where μ_i is the mean of points in S_i. It should be noted that number of groups is often unknown, and one just has give an educated guess.

As an example, let us carry out k-means clustering for the iris data, assuming that number of groups is 3 (which we actually know in this case).

```
data(iris)

#k-means clustering assuming 3 groups
iris.Kmeans = kmeans(iris[,1:4],3)

#in order to see how well the grouping matches
#with the groups seen in the PCA score plot, we
#plot the scores using different colours depending
#on the group

# PCA
iris.pca = prcomp(iris[,1:4])

#the scores
T = iris.pca$x

#the score plot
plot(T[,1],T[,2],col=iris.Kmeans$cluster)
```


T[, 2]
1.0
0.5
0.0
-0.5
-1.0
-3
-2
-1
0
1
2
3
4
T[, 1]

Appendix: Review of Matlab

Matlab is a general purpose mathematical program, especially designed for numerical methods and matrix computations. Matlab is a modular package consisting of the core Matlab and so called toolboxes, which are collections of functions and graphical user interfaces (GUI's) for various fields of applications. An end user of a specific toolbox can use Matlab pretty much like any Windows program, but its real nature is that of a programming language. Therefore, one who really wants to learn how use Matlab efficiently has to learn it in the same way as programming languages are learnt, i.e. one has to learn 1) the language elements, 2) the syntax and control structures, 3) to program new scripts and functions.

As a program language, Matlab is an command interpreter, not a compiler, which however is available. This means that all programming commands can be performed in the command window and, consequently, Matlab can be used similarly to a calculator. This, however, is not efficient use of Matlab and anyone who wants to take the full benefit of Matlab should learn the programming approach to it.

It should be noted that examples in chapter 4 require the DA (Data Analysis) toolbox developed by the author together with prof. Heikki Haario (Lappeenranta University of Technology).

1. The language elements

1.1 Numbers and matrices of numbers

Matlab uses only floating point numbers (except for the Symbolic Math Toolbox). The format command or the function sprintf can be used to change the appearance of the numbers (examples will follow).

The basic computational element in Matlab is a matrix which simply is regular table of numbers. These tables can consist of a single row (a *row vector*), a single column (a *column vector*) or rows and columns (a *matrix*). Matrices can have even more dimensions than just rows and columns, when they actually are *tensors*, but they are not considered in this introduction.

1.2 Variables and rules for constructing matrices

Variables are symbols containing numbers or matrices. There are some rules in naming variable, which are

- small and capital letters are considered different (e.g. ph is not the same variable as pH)
- only English alphabet, *without* any spaces, commas, dots, parentheses, etc., can be used
- the character _ (underscore) can be used

The contents of a variable are given with = so that the variable names is always on the left side of = and the contents (a number, vector or a matrix) on the right.

Examples with numbers:

```
>> a = 1.34;
```

It seems that nothing happens, but now variable named a contains the number 1.34. *The semicolon after any command suppresses the output.* To check the contents, simple type the name of the variable

```
>> a
a =
    1.3400
```

It is very important to understand that = does not mean equality in Matlab (which actually is ==). It is perfectly sensible to give a command like

```
>> a = a+2
```

```
a =
    3.3400
```

In mathematics a = a+2 would imply that a = 0, but not in Matlab. It simple means that 2 is added to a and *the result is the new content of a*. Of course one could give a new name for the result, e.g.

Now let's take another example to illustrate the different formats for numbers:

```
>> pie = pi
pie =
       3.1416
```

Here the value for π is given in the default format (short). If you want more all decimal places that Matlab uses, type

```
>> format long
>> pie
pie =
   3.14159265358979
```

For very large or small numbers the following is better

```
>> format short e
>> pie
pie =
  3.1416e+000
```

You can also get an rational approximation to any floating point number by

```
>> format rat
>> pie
pie =
   355/113
```

Usually the most convenient format is

```
>> format short g
>> pie
pie =
       3.1416
```

For special purposes you have to use the function sprintf, which we shall take up later in connection with function.

Examples with matrices:

```
row_vector_1 =
     1     2     3
>> row_vector_1 = [1, 2, 3]
row_vector_1 =
     1     2     3
```

Note that both commas or blanks can be used as separators between numbers which given inside brackets.

```
>> column_vector_1 = [1; 2; 3]
column_vector_1 =
     1
     2
     3
>> column_vector_1 = [1 2 3]'
column_vector_1 =
     1
     2
     3
```

Note that semicolon *inside* brackets starts a new row. In the latter case the row vector in turned into a column vector by using the *transpose operator* ‘.

```
>> A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
A =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

This example constructs a 4x3 matrix called A.

If you have to make large matrices the easiest way is to type it down in the following way

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

You can also copy tables from e.g. Excel and make Matlab-matrices of them in the same manner.

Referring to matrix elements

It is very easy to refer to vector or matrix elements. The rules are most easily learnt by the following examples. We shall use the matrix B above and some comments following the %-sign (which is the of commenting commands in Matlab):

```
>> B(1,1)
ans =
     1
>> B(3,4)=-9 %we shall change the element on 3rd row
             %and 4th column
B =
     1     2     3     4     5
     2     3     4     5     6
     3     4     5    -9     7
     4     5     6     7     8
     5     6     7     8     9
>> B(2,:) %the 2nd row
ans =
     2     3     4     5     6
>> B(:,3) %the 3rd column
ans =
     3
     4
     5
     6
     7
>> B(2:end,2:end) %B without the 1st row and column
ans =
     3     4     5     6
     4     5    -9     7
     5     6     7     8
     6     7     8     9
>> B(3:end-1,3:end-1) %rows and columns from the 3rd to the
                      %second last
ans =
     5    -9
     6     7

>> B([1 4 5],[2 4]) %selected rows and columns
ans =
     2     4
     5     7
     6     8
```

In referring to vectors, only one index is needed, e.g.

```
>> row_vector_1(3)
ans =
     3
>> row_vector_1(1:3)
ans =
     1     2     3
```

Note the short hand notation 1:3 for [1 2 3].

Some useful matrix functions

The following functions are explained solely by examples:

```
>> zeros(3,2)
ans =
     0     0
     0     0
     0     0

>> ones(3,8)
ans =
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1
     1     1     1     1     1     1     1     1

>> eye(5)
ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

>> P = diag([2 2 5 5 3 3])
P =
     2     0     0     0     0     0
     0     2     0     0     0     0
     0     0     5     0     0     0
     0     0     0     5     0     0
     0     0     0     0     3     0
     0     0     0     0     0     3

>> diag(P)
ans =
     2
     2
     5
     5
     3
```

```
     3
>> I = eye(10);
>> size(I)
ans =
    10    10
```

1.3 Character strings

Strings in Matlab are character arrays which are given by enclosing the text in apostrophes. Examples:

```
>> name = 'My name is Veli-Matti';
>> name(1:5)
ans =
My na
>> name(11:end)'
ans =

V
e
l
i
-
M
a
t
t
i
```

1.4 Cell arrays

Cell arrays are very useful structures for advanced Matlab-users. They are not considered in this introduction, but we give one example (note the braces!):

```
>> array_1 = {'hello' [1 2;3 4]; pi 'good bye'}
array_1 =
    'hello'     [2x2 double]
    [3.1416]    'good bye'
>> array_1{2,2}
ans =
good bye
>> array_1{1,2}
ans =
     1     2
     3     4
```

1.5 Structs

Structs are useful in organizing data in a logical way. Structs are constructed by adding field names to a variable. Examples:

```
>> person.name = 'Veli-Matti';
>> person.date_of_birth = [16 8 1953];
>> person
person =
             name: 'Veli-Matti'
    date_of_birth: [16 8 1953]
```

1.6 Operators

Matlab uses ordinary notation for arithmetic operations: +,-,*,/,^ for scalar variables. *, / and ^ mean corresponding matrix operations if variables are vectors or matrices. Element-wise multiplication, division and powers are performed by .*, ./ and .^. The transpose of a matrix is performed by ‘. Examples:

```
>> A = [1 2;3 4]
A =
     1     2
     3     4
>> b = [7;8]
b =
     7
     8
>> A*b %matrix multiplication
ans =
    23
    53
>> A(:,1).*b %element-wise multiplication (1. column of A
            %with b)
ans =
     7
    24
>> A(:,2).*b %element-wise multiplication (2. column of A
              %with b)
ans =
    14
    32
>> A^2 %matrix power (= A*A)
ans =
     7    10
    15    22
```

```
>> A.^2 %element-wise power
ans =
     1     4
     9    16
>> A/A %matrix division (= A*A^(-1) = A*inv(A))
ans =
     1     0
     0     1
>> A./A %element-wise division
ans =
     1     1
     1     1
>> A' %transpose of A
ans =
     1     3
     2     4
>> b' %transpose of b
ans =
     7     8
>> b'*b
ans =
   113
```

Logical operations in Matlab are element-wise operations whose results are true (= 1) or false (= 0). ‘And’ is &, ‘or’ is | and ‘equal to’ is ==. Examples:

```
>> A == 1
ans =
     1     0
     0     0
>> A > 1
ans =
     0     1
     1     1


>> A >= 1
ans =
     1     1
     1     1
>> A > 1 & A > 2
ans =
     0     0
     1     1
>> A > 1 | A > 2
ans =
```

```
     0     1
     1     1
```

If you want to find elements of a matrix or a string which satisfy given conditions, functions ‘find’ and ‘findstr’ are extremely useful, but we will illustrate their use later.

1.7 Functions

There are three kinds of functions in Matlab: built-in, inline and m-file functions. Built-in functions are the core of Matlab and they cannot be viewed or changed. If you try to type a built-in function, you will only get a message that the function is built-in:

```
>> type norm
norm is a built-in function.
```

Most Matlab-functions are so called m-file functions that can be viewed or edited. The user can also easily make his own m-file functions. An example:

```
>> type factorial

function p = factorial(n)
%FACTORIAL Factorial function.
%   FACTORIAL(N) for scalar N, is the product of all the integers from 1 to N,
%   i.e. prod(1:N). When N is an N-D matrix, FACTORIAL(N) is the factorial for
%   each element of N.  Since double precision numbers only have about
%   15 digits, the answer is only accurate for N <= 21. For larger N,
%   the answer will have the correct order of magnitude, and is accurate for
%   the first 15 digits.
%
%   See also PROD.

% Copyright 1998-2004 The MathWorks, Inc.

N = n(:);
if any(fix(N) ~= N) || any(N < 0) || ~isa(N,'double') || ~isreal(N)
  error('MATLAB:factorial:NNegativeInt', ...
        'N must be a matrix of non-negative integers.')
end
n(N>170) = 171;
m = max([1;n(:)]);
N = [1 1 cumprod(2:m)];
p = N(n+1);
```

As an example of a user-defined function we will show how to make the sine function for angles in degrees. Matlab’s function atan2 gives only angles in radians.

```
function y = atan2d(y,x)
% atan2d(y,x) calculates the phase angle of
% a point whose coordinates are (x,y)
% Note the order of the arguments!
y = atan2(y,x)*180/pi;
```

After writing down this function, it works as any function in Matlab:

```
>> atan2d(1,1)
ans =
    45.000
```

The comment lines after the function definition serve as help:

```
>> help atan2d
  atan2d(y,x) calculates the phase angle of
  a point whose coordinates are (x,y)
  Note the order of the arguments!
```

Another useful way of defining functions, especially for temporary use, are so called in-line functions. They are defined by the inline command. As an example, we'll show how to define a function that gives the pH by giving the H_3O^+ concentration:

```
>> pH = inline('-log10(x)')
pH =
     Inline function:
     pH(x) = -log10(x)
```

Now, to see what is pH if H_3O^+ concentration is 0.00001, you simply type

```
>> pH(0.00001)
ans =
     5
```

2. Operating with files and some plotting examples

2.1 Command files (scripts, m-files)

The most effective way to use Matlab is to write commands into command files (also called as scripts or m-files) whose extension is .m. Scripts are ordinary text files which can be edited with any text editor (e.g. Notepad). Most convenient, however, is to use Matlab's own editor. To start to editor, simply click the New M-file icon in the toolbar. Before opening the file, it is advisable to change the current directory (= folder) to be the one where you want your application to be located. This can be done either in the current directory window in the toolbar or by applying the cd-command (cd('directory path')).

After changing the current directory and opening a new m-file, simple type down the commands you want Matlab to perform. It is advisable to add clarifying comments, anything following the %-character will be interpreted as a comment.

Example

```
%this m-file plots the exponential function
%y = a*exp(-b*x) on the interval [0,x_max]

x = linspace(0,x_max); %this command produces 100 values
                       %of x between 0 and x_max
figure
plot(x,a*exp(-b*x))
xlabel('x'), ylabel('y')
title('y = ae^-^b^x')
```

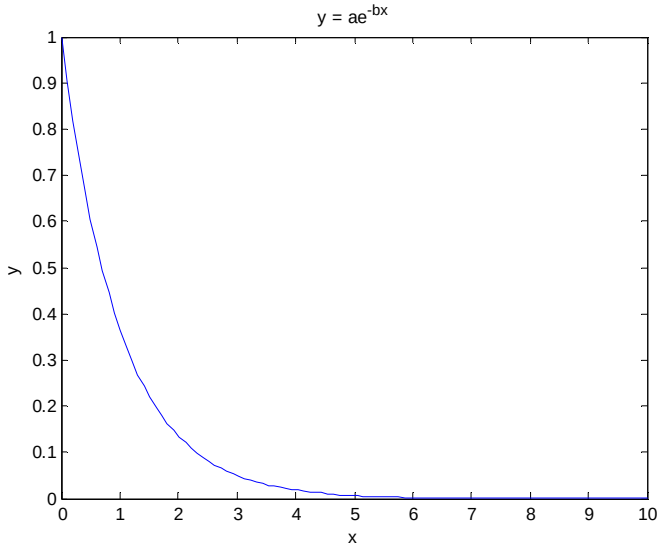
You can run the commands by typing the name of the file or by pressing F5:

```
>> example1
??? Undefined function or variable "x_max".

Error in ==> example1 at 4
x = linspace(0,x_max);
```

The error message tells that the variable x_max is undefined. It is important to understand that any variable on the right hand side of = sign should have a numerical value (or values). Let's have a new trial:

```
>> a=1;b=1;x_max=10;
>> example1
```

Now we get the figure below.



If you now would like to plot the same function with different parameter values or on a different interval, all you need is to retype the commands with different values, or to edit the command history:

```
>> a=1;b=.5;x_max=10;
>> example1
```

and you would get a new figure. If you would rather see the new curve in the same figure, just type the command >> hold on before the commands above. There other ways to plot several curves onto the same figure, but we'll come to that later on.

2.2 Data files

2.2.1 Text files

In many applications, you need data from external sources. There are many alternative ways to do get data from outside Matlab, but we'll show only the simplest, i.e. retrieving text-files. Suppose the file reg_ex.txt contains the following data:

```
%x and y
   3.1   5.5
   2.3   4.8
   3.0   4.7
   1.9   3.9
   2.5   4.5
   3.7   6.2
   3.4   6.0
   2.6   5.2
   2.8   4.7
   1.6   4.3
   2.0   4.9
   2.9   5.4
   2.3   5.0
   3.2   6.3
   1.8   4.6
   1.4   4.3
   2.0   5.0
   3.8   5.9
   2.2   4.1
   1.5   4.7
```

The data can be loaded and plotted using the following commands

```
>> edit reg_ex.txt
>> load reg_ex.txt
>> x = reg_ex(:,1);
>> y = reg_ex(:,2);
>> figure(1),plot(x,y,'*')
```

2.2.2 Mat-files

Any variable in Matlab can be save in Matlab's own mat-format by the command save file-name variable-list. If you omit the variable list, all variables in the work-space are saved. The save variables can be retrieved by the command load file-name (the command clear all clears the work-space and the command whos shows the contents of the work-space)

Example

```
>> save reg_ex x y
>> clear all
>> load reg_ex
>> whos
  Name      Size                    Bytes  Class

  x        20x1                      160  double array
  y        20x1                      160  double array

Grand total is 40 elements using 320 bytes
```

3. Spectral data and basic plotting commands

The simplest form of spectral data is a single spectrum which is a single vector, if wave-lengths (or wave-numbers) are not included, or two vectors, if wave-lengths are included. A single spectrum can be plotted against the wave-lengths or the sequential number of the wave-lengths

For the following examples, you need to load the file spec1.mat in the following way

```
>> load spectra.mat
>> whos
Variables in the current scope:

  Attr Name        Size                     Bytes  Class
  ==== ====        ====                     =====  =====
       C          17x4                       544  double
       spec     7601x17                  1033736  double
       t          17x1                       136  double

Total is 129302 elements using 1034416 bytes
```

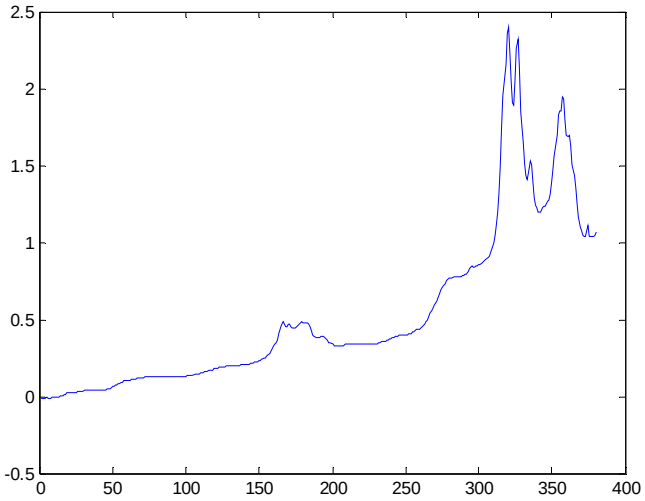
The command whos tells you the names of all defined variables and their sizes and types. We see that, for example, **spec** is a matrix whose columns

represent the spectra of 17 samples. **t** is a vector of sampling times and **C** is a matrix containing all 17 concentrations of 4 compounds.

To plot the first spectrum in **spec** we do the following

```
>> i = 1:7601; %sequential numbers
>> figure(1),plot(i,spec(:,1))
```

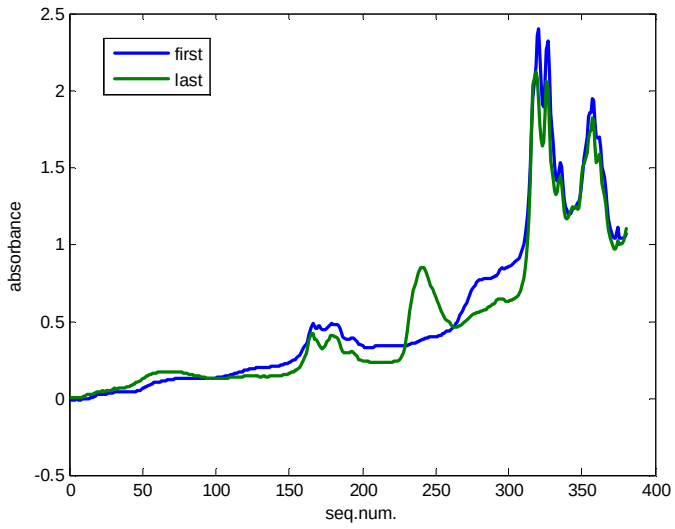
These commands will produce a figure resembling the one below



To plot the first and the last spectrum on the same plot and to add some enhancements, we could do the following

```
>> figure(1),plot(i,spec(:,1),i,spec(:,end),'LineWidth',2)
>> xlabel('seq.num.'),ylabel('absorbance')
>> legend('first','last')
```

and we would get something like

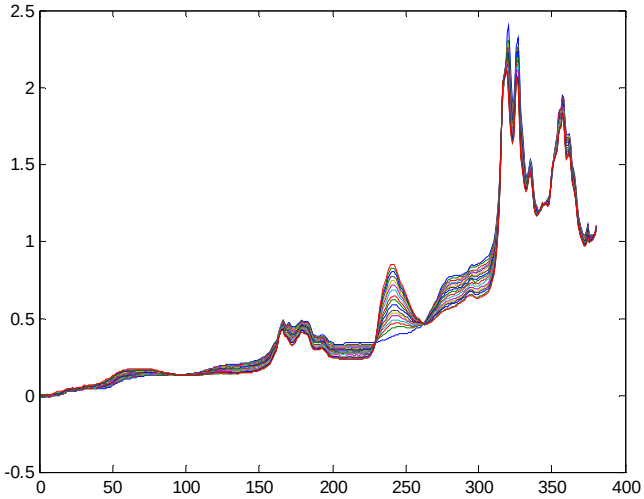


Note that all such features as ‘LineWidth’ above, can also be changed interactively from the toolbar by selecting Edit/Figure properties... It is also easy to add text, arrows etc. on the figure, or to zoom in and out, but such operations are most easily learned in an interactive session.

If we would want see all spectra of the batch, we could do the following

```
>> figure(1),plot(i,spec)
```

giving us



We might also be interested in time evolution which is not easily seen from the figure above. If we choose time instead of the sequential number for the x-axis and take, for example, the 250th absorbance, we have to give following commands