Introduction to Matlab
+
Mathematical aspects of bilinear factor models (PCA and PLS)

Frans van den Berg (last modifications September 22, 2007)
fb@life.ku.dk
University of Copenhagen, Faculty of Life Sciences
Department of Food Science
Quality and Technology, Spectroscopy and Chemometrics Group
Rolighedsvej 30 (Room T 447)
DK-1958 Frederiksberg-C
Denmark

This document + programming code is available form www.models.life.ku.dk

Contents

Introduction to basic Matlab with computer code

Transparencies (+ some telegram-style explanation)

Practicing Matlab with factor models

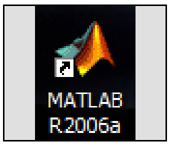
Introduction

Two titles = two aims: 1) to get a quick introduction to the computer program Matlab (see www.mathworks.com); 2) to get some insight into the bilinear factor models Principal Component Analysis (PCA) and Partial Least Squares (PLS) regression, focusing on the mathematics and numerical aspects rather than how's and why's of data analysis practice. For the latter part it is assumed (but not absolutely necessary) that the reader is already familiar with these methods. It also assumes you have had some preliminary experience with linear/matrix algebra.

This introduction is based on Matlab releases 7/14 (but the difference with older releases is insignificant for the scope of this introductory; exercises will work with any of the resent Matlab releases). The computer exercise material can be found at www.models.life.ku.dk.

Matlab environment

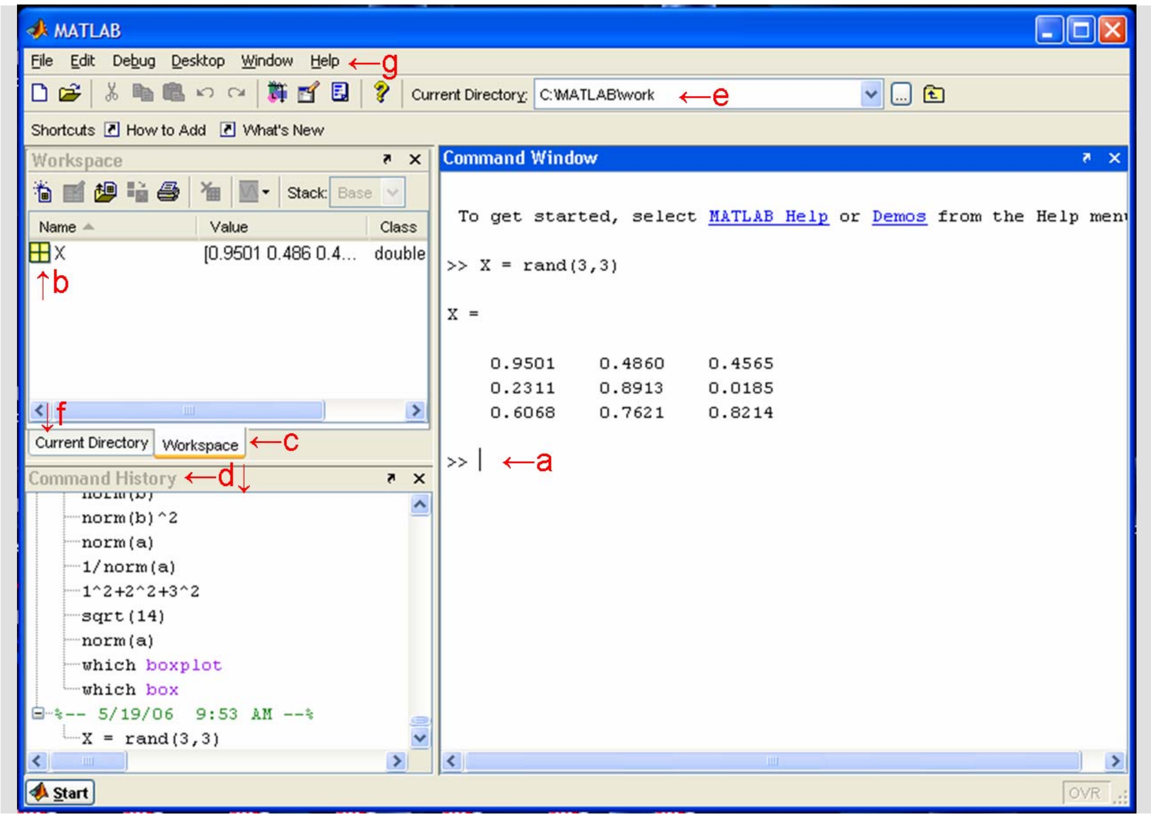
Let us start up the Matlab environment. This will open the Matlab command editor, with the ">>" command prompt. Behind this prompt you can type in a huge number of commands that will perform a multitude of different (mainly numerical and graphical) operations, some of which will be explained below. This same prompt will be used in this document to mark a Matlab statement. Matlab commands are further "coded" by the *italic* font.



Thus, to get help, go to the Matlab windows and type
>> *help*

If you end this line by pressing [enter], Matlab will respond with some lines of information popping up in the Matlab window (never mind what-is-what for now).

The figure below shows the Matlab program workspace or environment (a similar "control panel" should have appeared on your computer screen already, with slight differences dependent on the version/release you are working in).


MATLAB
File Edit Debug Desktop Window Help
g
Current Directory: C:\MATLAB\work
e
Shortcuts How to Add What's New
Workspace
Stack: Base
Command Window
b
f
Current Directory Workspace
c
Command History
d
a
Start
OVR

Name	Value	Class
X	[0.9501 0.486 0.4...	double

```
 To get started, select MATLAB Help or Demos from the Help men

>> X = rand(3,3)

X =

    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214

>> |
```

```
norm(b)
norm(b)^2
norm(a)
1/norm(a)
1^2+2^2+3^2
sqrt(14)
norm(a)
which boxplot
which box
%-- 5/19/06   9:53 AM --%
X = rand(3,3)
```

What do we have?

a) This is the window where Matlab commands are executed. You can type many different commands (e.g. "*help*") and Matlab will react with a response by showing the result, opening a new figure window, showing an error message if the command was incorrect, etc.
b) In Matlab information is stored in so-called data arrays. In the example figure a 3x3-matrix called "X" with random numbers between 0 and 1 is generated (command "*rand*" in the window).
c) The Workspace window shows all the data you have stored in memory (in this case only "X").
d) The Command History window shows all the previously entered commands. Here: first we cleared the command window ("*clc*" = clear console), then we created the "X"-matrix.
e) When you open the program you jump to the directory "work" on the hard disk. This is where the files and data will be stored. When you start working on different projects it is a good idea to organize these as subdirectories in this "work"-root.
f) You can change subdirectories (projects!) in the Current Directory window.
g) Besides the options already mentioned there are many more possibilities which you can find in the (default computer) menu structure on top of the program window.

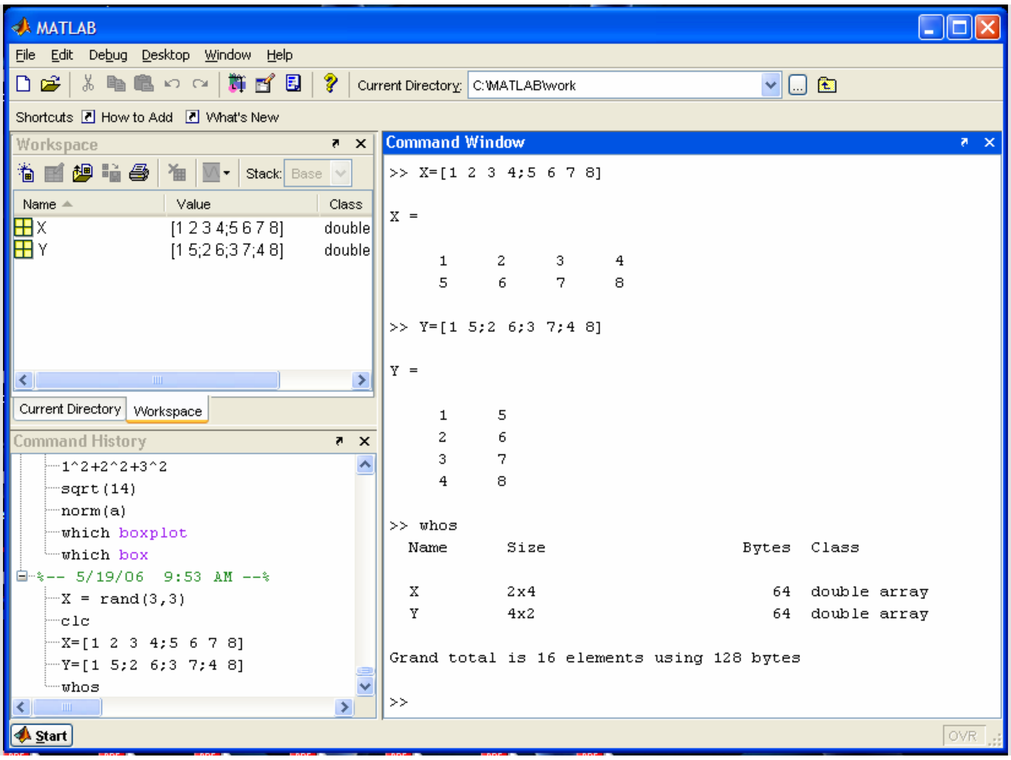
So working with Matlab is really simple: you just type in the right commands! Now all you have to figure out is which of the hundreds of commands you can use when, how and in what order. And that is the subject of the rest of this document.

Matlab commands

Matlab stands for Matrix Laboratory. Hence, all the data handling is matrix-oriented. To enter the simple data table X
>> X = [1 2 3 4; 5 6 7 8]

You now have a matrix X with two rows and four columns (2x4) in the workspace (";" here means new line/row). If you enter
>> Y = [1 5; 2 6; 3 7; 4 8]
you will have a second data table Y (4x2) – four "new lines" of two numbers each = four rows, two columns.

To see the results type
>> who
or
>> whos


MATLAB
File Edit Debug Desktop Window Help
Current Directory: C:\MATLAB\work
Shortcuts How to Add What's New
Workspace
Stack: Base
Current Directory Workspace
Command History
Command Window
Start
OVR

Name	Value	Class
X	[1 2 3 4;5 6 7 8]	double
Y	[1 5;2 6;3 7;4 8]	double

```
1^2+2^2+3^2
sqrt(14)
norm(a)
which boxplot
which box
%-- 5/19/06  9:53 AM --%
X = rand(3,3)
clc
X=[1 2 3 4;5 6 7 8]
Y=[1 5;2 6;3 7;4 8]
whos
```

```
>> X=[1 2 3 4;5 6 7 8]

X =

     1     2     3     4
     5     6     7     8

>> Y=[1 5;2 6;3 7;4 8]

Y =

     1     5
     2     6
     3     7
     4     8

>> whos
  Name      Size                    Bytes  Class

  X         2x4                        64  double array
  Y         4x2                        64  double array

Grand total is 16 elements using 128 bytes

>>
```

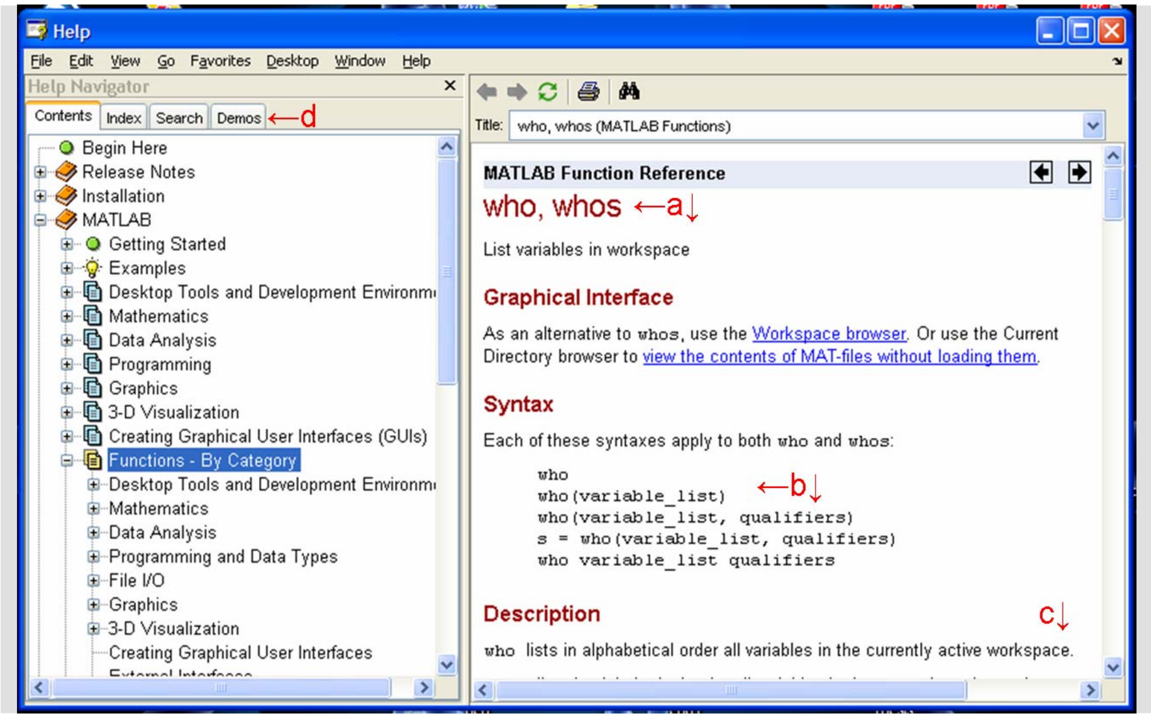
The figure shows the workspace including the dimensions of the matrices (The change in layout is a results from changing operating system).
Verify the sizes of "X" and "Y" from the "*whos*" command, and make sure you understand why the two matrices get this size from the commands shown above.

Two things: a) By simply typing the commands above you generate the new variables. In the same way you can "recycle" variables to contain different data (e.g. "*Y = 3*" will change "Y" from a matrix (4x2) into a scalar value 3 - or special *matrix* of size (1x1). Notice that we did this already with X, just look at the two figures above. b) Matlab is case sensitive, meaning "X" and "x" are not the same variable names.

Here are three ways to get information for (almost) any Matlab-command (here we use "*who*" as an example):

>> *help who*	(short description of the command in the work environment)
>> *helpwin who*	(description in a separate window)
>> *doc who*	(longer description in a separate window)

The figure shows the window using the "*doc*"-command on "*who*".


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Contents Index Search Demos
←d
Begin Here
Release Notes
Installation
MATLAB
Getting Started
Examples
Desktop Tools and Development Environm
Mathematics
Data Analysis
Programming
Graphics
3-D Visualization
Creating Graphical User Interfaces (GUIs)
Functions - By Category
Desktop Tools and Development Environm
Mathematics
Data Analysis
Programming and Data Types
File I/O
Graphics
3-D Visualization
Creating Graphical User Interfaces
Title:
who, whos (MATLAB Functions)
←a↓
←b↓
c↓

MATLAB Function Reference

who, whos

List variables in workspace

Graphical Interface

As an alternative to whos, use the Workspace browser. Or use the Current Directory browser to view the contents of MAT-files without loading them.

Syntax

Each of these syntaxes apply to both who and whos:

```
who
who(variable_list)
who(variable_list, qualifiers)
s = who(variable_list, qualifiers)
who variable_list qualifiers
```

Description

who lists in alphabetical order all variables in the currently active workspace.

a) The function you wanted explained is shown together with a brief description.
b) Then a series of examples shows its possible use.
c) A more official description of the do-and-don'ts for a function follows (often more text than you hope for!).
d) Like for most software an extensive search mechanism is available. This can be very helpful if you know what you want to do with your data, but you cannot remember what the right command was.

Back to the command prompt; we could have created the matrix Y (size 4x2) from before a lot quicker

```
>> Y = X';
```

where the '-addition means transpose-operation.
The ";" at the end of the command-line means: don't show the output on the screen. But if you want to see the new Y anyway, just type

```
>> Y
```

If at some later stage Y becomes really big you can terminate the output to the screen with [ctrl]-[c] or [crtl]-[Break]. (If you are not convinced of its importance just type *"rand(5000,100)"* on the command prompt, and see what happens)

Lets do some numeric operations with these two data tables

```
>> X-Y          (subtraction & your very first error message in Matlab!)
>> X-Y'         (subtraction - the right way - leaving you with all zeros)
>> X+Y'         (addition)
>> X*Y          (inner product multiplication, 2x2 result)
>> Y*X          (outer product multiplication, 4x4 result)
>> Y.*X'        (the element-wise multiplication, 4 x 2 result)
>> Y+3.5        (add 3.5 units to each element in "Y")
```

The examples above showed how to work with entire data matrices/tables. The next examples illustrate how to extract part of a table. To retrieve e.g. the second-row first-column element from table X, and store it in a new variable named "c", enter the following line

```
>> c = X(2,1);
```

In detail: we have our matrix X (2x4), we pick out the element second row (the row-index = first-index between brackets) and first column (column-index = second-index between brackets), and we store the result in "c" (which should contain a 5; check)

E.g. if X = [1 2 3 4
5 6 7 8] → X(1,1), X(2,1) and X(2,4)

Matlab handles scalars as matrices of size (1x1) (use the "*whos*" command to verify this). A variable named "*ans*" might have popped up. It contains the answer of the last question you asked Matlab without explicitly specifying where to store the result. Thus entering

```
>> X(2,3)
```

will show *ans* = 7; it is bad practice to rely on this variable! Making ones – with a meaningful name – can save you a lot of trouble!

To extract a full (row or column) vector from a table you can use the colon ":"-command. E.g.

```
>> y1 = Y(:,1); y2 = Y(:,2);
```

to store the first and second column of table Y in (column vectors) y1 and y2.

How does it work? The ":"-separator is used to automatically generate a list of numbers, e.g.

```
>> 1:4
```

You will see the list of numbers (1 to 4, counter 1). Also try the following command

```
>> 1:3:16
>> 1:2:16                 (Where did 16 go?)
>> 15:-1:1
```

So if we want to have the first three rows from Y, second column

```
>> Y(1:3,2)
```

There are two special versions of this command: ":" by itself means take all entries in this direction (thus entire row or column, like we did in the example); you don't have to specify the length. The command "*end*" can be used to go to the end of a row or column, thus

```
>> Y(2:end,1)
```

shows the rows from 2 until the last one, first column in table Y.

One last method to get elements from a matrix by defining new variables/vectors

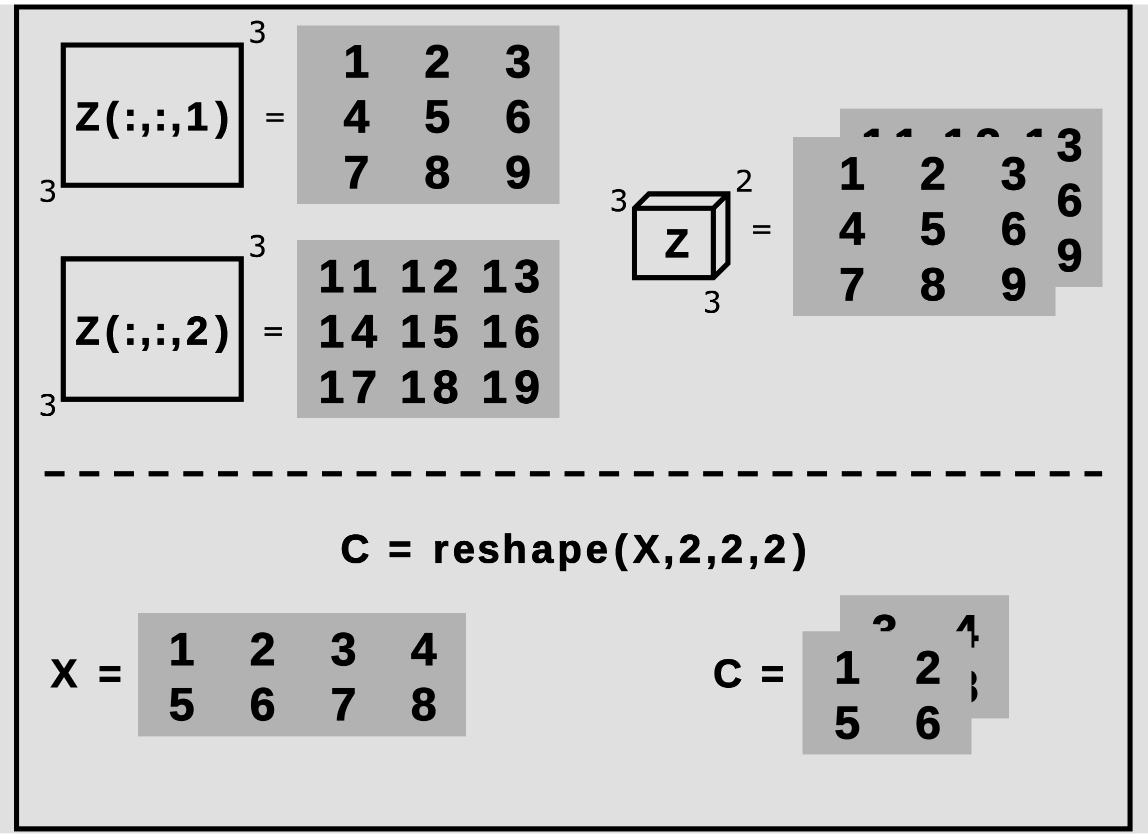
```
>> row_index = [1 3]; col_index = [1 2];        (Make new variables)
>> Y(row_index,col_index)                    (and use them as indexes)
```

Notices: a) you can use every variable name you like – including numbers – but smart names make Matlab-live a lot easier; b) a variable name cannot start with a number, and you should not give it the name of a build-in Matlab command such as "*whos*" (this will override the command, so you cannot use it any longer).

Higher dimensional data arrays in Matlab are created and addressed similar to vectors and matrices. To produce a (3x3x2) cube Z enter the following code

```
>> Z(:,:,1)=[1 2 3;4 5 6;7 8 9];                (first slab)
>> Z(:,:,2)=[11 12 13;14 15 16;17 18 19];       (second slab)
```

The cube "Z" is symbolized in the figure.


Z(:,:,1) = 1 2 3 4 5 6 7 8 9
Z(:,:,2) = 11 12 13 14 15 16 17 18 19
Z =
C = reshape(X,2,2,2)
X = 1 2 3 4 5 6 7 8
C = 1 2 5 6

Indexing in this 3D data structures follows the same conventions as before: first-index = row-index (x-direction), second-index = column-index (y-direction) and third-index = so-called tube or tubular-index (z-direction; "into the paper"). Higher order data tables (4D, 5D, etc.)

are hard to visualize (or imagine) but generating them in Matlab is simple: just introduce a fourth, fifth, etc. dimension in the indexing

To see the center tube (numbers 5 and 15) in this cube enter

```
>> Z(2,2,:)                 (second row, second column, all slabs)
```

Another important command related to higher order arrays is “*reshape*”. To transform our old matrix X (2x4) into a small cube C (2x2x2) enter the following

```
>> C = reshape(X,2,2,2)
```

This reshaping-operation can only work if the number of entries in the source (X: 4x2=8) and target (C: 2x2x2=8) are the same. Notice that “*reshape*” is a column wise operation

```
>> reshape(X,2,2,2)          (create a cube form X – check the numbers)
>> reshape(X',2,2,2)         (create a cube from its transposed - check)
```

One more useful command for data cubes is “*squeeze*”:

```
>> Z(:,1,:)                  (selects the first slab “in the paper”)
>> squeeze(Z(:,1,:))         (“squeezes” this data in a regular/flat matrix)
```

So, the basic form to store data is in arrays of so-called double precision real numbers: scalar in size (1x1), vectors in size (1xn) for rows or (nx1) for columns, matrices in size (nxm), cubes in size (mxnxo), etc. There are many more forms for storing numerical and non-numerical data, important for computer-internal business, but that is only important for more advanced work.

Briefly three structures you could use or encounter. Keep a close eye on the “Workspace” area of your Matlab surrounding to see if you understand all the symbols appearing. To form a string of characters (so-called char array) enter

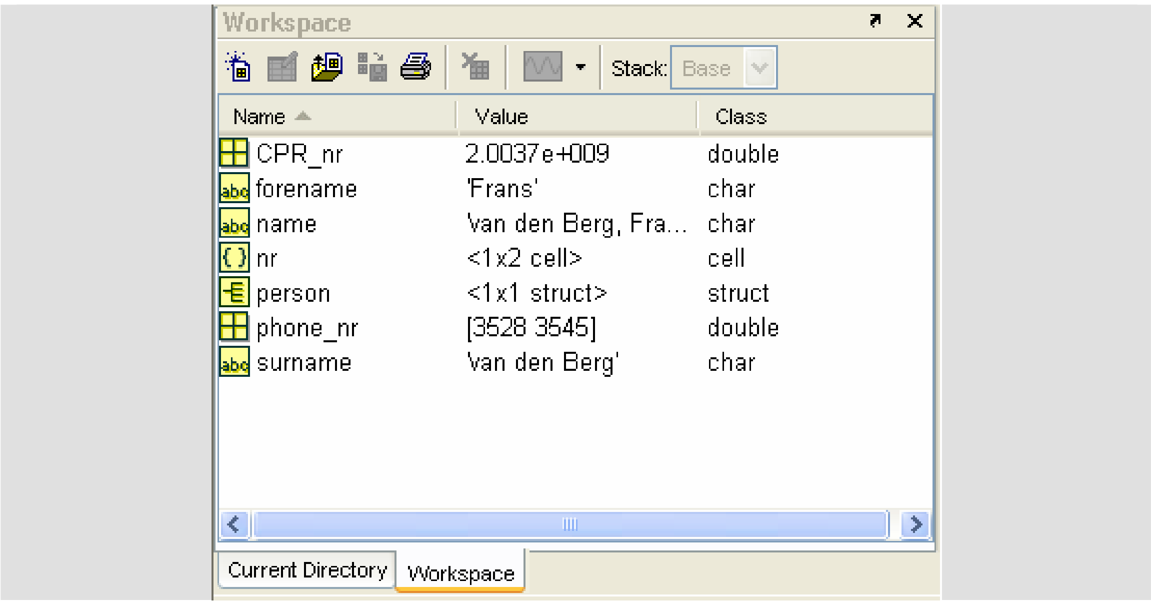
```
>> surname = 'van den Berg'             (make a text string)
>> forename = 'Frans'                   (and another one)
>> name = [surname ', ' forename]       (manipulate strings like arrays)
```

The so-called “cell array” can be used to store both text and numerical arrays with different sizes (this is a rare/advanced format, but notice that e.g. rows of different length cannot be stored in a conventional matrix; also note the curly brackets)

```
>> phone_nrs = [3528 3545]
>> CPR_nr = 2003680000
>> nr = {phone_nrs CPR_nr 'my numbers'}
>> nr{2}                        (this will give the CPR-number)
```

The last data format – "struct" or structure - is convenient to group like-things together (notice the use of "." - dot).

```
>> person.email = 'fb@kvl.dk'   (a new text string)
>> person.name = name           (an existing text string)
>> person.numbers = nr          (cell array in the structure array)
>> person                       (shows the contents of the structure)
>> person.email                 (to address elements use dot-notation)
```



Name	Value	Class
CPR_nr	2.0037e+009	double
forename	'Frans'	char
name	'Van den Berg, Fra...	char
nr	<1x2 cell>	cell
person	<1x1 struct>	struct
phone_nr	[3528 3545]	double
surname	'Van den Berg'	char

Some other useful command

```
>> clear y1 y2         (clear variables y1 and y2 from the workspace)
>> save myfile         (saves all variables in the workspace to a file called
                       "myfile.mat")
>> clear all           (clears all variables)
>> load myfile         (loads all variables from the file "myfile.mat")
```

There is one more convenient feature in the Matlab editor: the arrow-up [↑] of the keyboard will callback all the old command entries, from newest to oldest. So you don't have to type everything over and over again, just call an older version of a similar thing you want to do by repeatedly pushing [↑], and edit it to do the new job. Use [←], [→], [Delete], etc. on the command line like you would in a conventional text editor.

If you know the first letter of the old command, type it at the prompt and push the arrow-up to scroll the commands starting with this letter. E.g. typing "w" and then arrow-up would first show the "*whos*" and then "*who*" for the commands treated so far in this text.

Script, functions and plotting

To fully explore the graphical possibilities of Matlab we first have to get hold of some interesting things to plot. We will generate some Gaussian-curves. Below is the formula for a gauss- or normal-distribution where "*x*" is the running index (a probability index in statistics), "μ" is the mean and "σ" is the standard deviation:

$$y(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{(x-\mu)}{\sigma}\right)^2}$$

We can implement and plot this in Matlab with the following lines of code

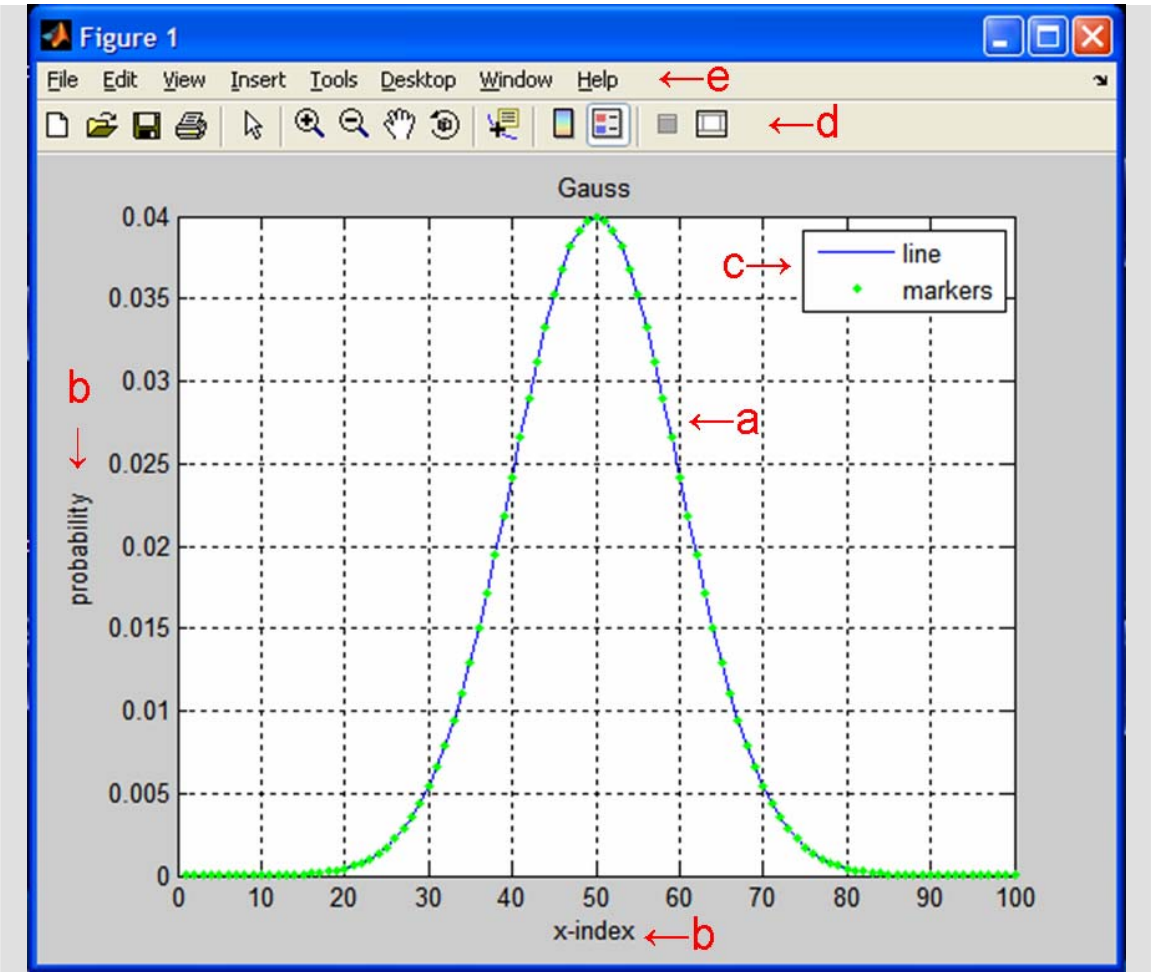
```
>> x = 1:100; mu = 50; sigma = 10;
                              (running index, mean and std. deviation)
>> y = 1/(sigma*sqrt(2*pi)) * exp(-0.5*((x-mu)/sigma).^2);
                              ("sqrt" = square root; "exp" = exponential)
```

Say you would like to plot green dot-markers for the points in our vectors x and y together with a blue line connecting the points (the default plot), enter the following command

```
>> plot(x,y,x,y,'.g')         (plot the results twice: blue line and green dots)
>> grid                       (add some lines to assist in interpretation)
>> xlabel('x-index'); ylabel('probability'); title('Gauss')
>> legend('line','markers')   (add some text and labels for the lines)
```

Notices: a) If you want to pass text to Matlab you have to include it in quotes (e.g. 'x-index') just like we did for string arrays (this is the way the program distinguishes commands and variables names from text-input); b) If you do not specify markers, points will be connected by a line; If you do not specify the color the fixed order of lines will be: blue, green, red, cyan, etc.; c) study the help for the "*plot*" routine to get more info.

You noticed that the plot-command opened a separate window, copied in the figure below.



a) The data is drawn. The first time as a blue line (default since no plotting mode was specified) and second time green "."-symbols specified in the plot command.
b) Labels and title are next to the axis.
c) A legend box is put inside the plotting-paper.
d) A number of operations can be performed on the plot: select objects, add text/arrow/lines, zoom in/out or rotate the figure (we will come back to rotation later on).
e) Under the menu options [Edit] and [Tools] you will find more (albeit more difficult) figure manipulation options.

The commands you typed in to generate the gauss-curves are pretty complicated, and it is easy to make errors when typing complicated commands on the Matlab prompt. To avoid this you can use so-called scripts. Push the [new]-button on the Matlab window (blank in upper left corner). This will open a new window - the script editor.

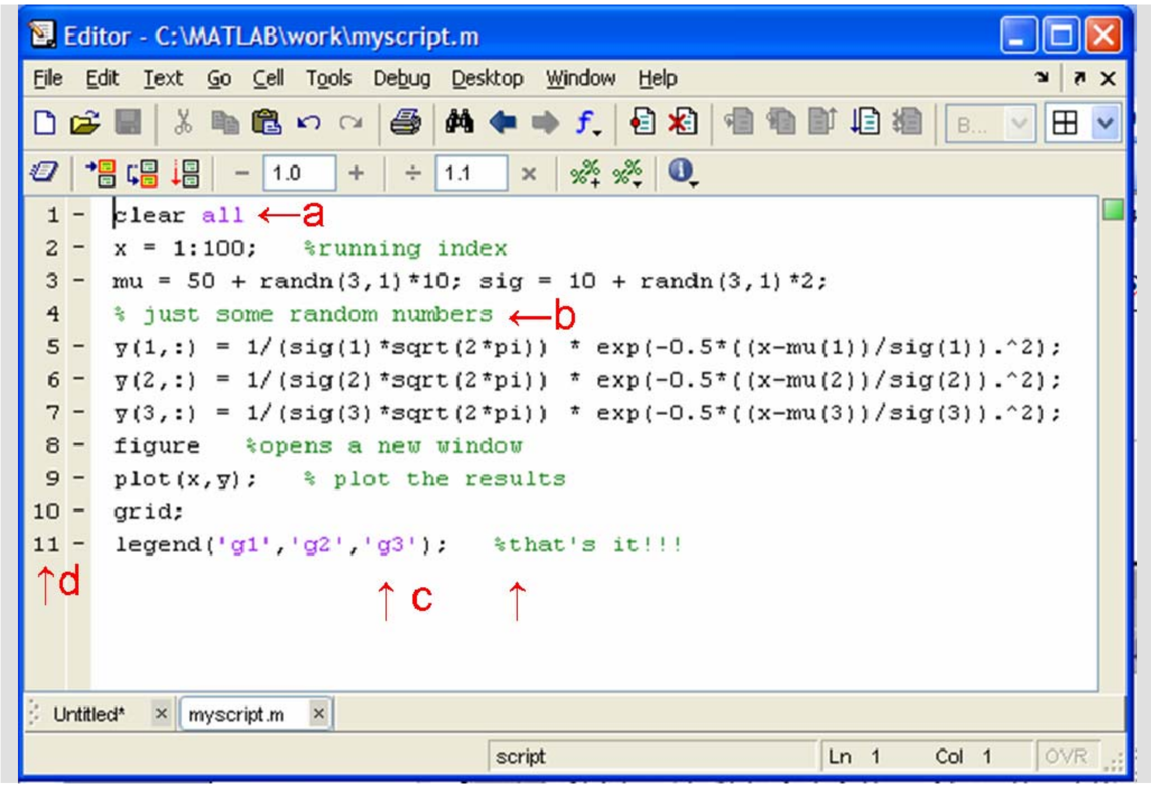
In the editor sheet you can enter the following text (copy-paste, but remove the ">>" signs)

```
>> clear all
>> x = 1:100;   %running index
>> mu = 50 + randn(3,1)*10; sig = 10 + randn(3,1)*2;
>> % just some random numbers
>> y(1,:) = 1/(sig(1)*sqrt(2*pi)) * exp(-0.5*((x-mu(1))/sig(1)).^2);
>> y(2,:) = 1/(sig(2)*sqrt(2*pi)) * exp(-0.5*((x-mu(2))/sig(2)).^2);
>> y(3,:) = 1/(sig(3)*sqrt(2*pi)) * exp(-0.5*((x-mu(3))/sig(3)).^2);
>> figure   %opens a new window
>> plot(x,y);   % plot the results
>> grid;
>> legend('g1','g2','g3');   %that's it!!!
```

The command "*randn*" generates a random number with a so-called normal/Gaussian distribution; see "*helpwin randn*" and "*helpwin rand*" for further information.

Then save this file under the name "*myscript*". Matlab will store it as the file called "myscript.m", and you can execute this series of commands by simply typing

```
>> myscript
```



```
clear all
x = 1:100;    %running index
mu = 50 + randn(3,1)*10; sig = 10 + randn(3,1)*2;
% just some random numbers
y(1,:) = 1/(sig(1)*sqrt(2*pi)) * exp(-0.5*((x-mu(1))/sig(1)).^2);
y(2,:) = 1/(sig(2)*sqrt(2*pi)) * exp(-0.5*((x-mu(2))/sig(2)).^2);
y(3,:) = 1/(sig(3)*sqrt(2*pi)) * exp(-0.5*((x-mu(3))/sig(3)).^2);
figure    %opens a new window
plot(x,y);    % plot the results
grid;
legend('g1','g2','g3');    %that's it!!!
```

Editor - C:\MATLAB\work\myscript.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
a
b
c
d
Untitled*
myscript.m
script
Ln 1
Col 1
OVR

a) Regular Matlab commands within the script file.
b) Comment text lines (always starting with "%") help to explain the intension of the operations. They are neglected by the Matlab command interpreter.
c) Color schemes in the editor improve the readability.
d) Line numbers are useful for the error messages generated in the Workspace Command Window.

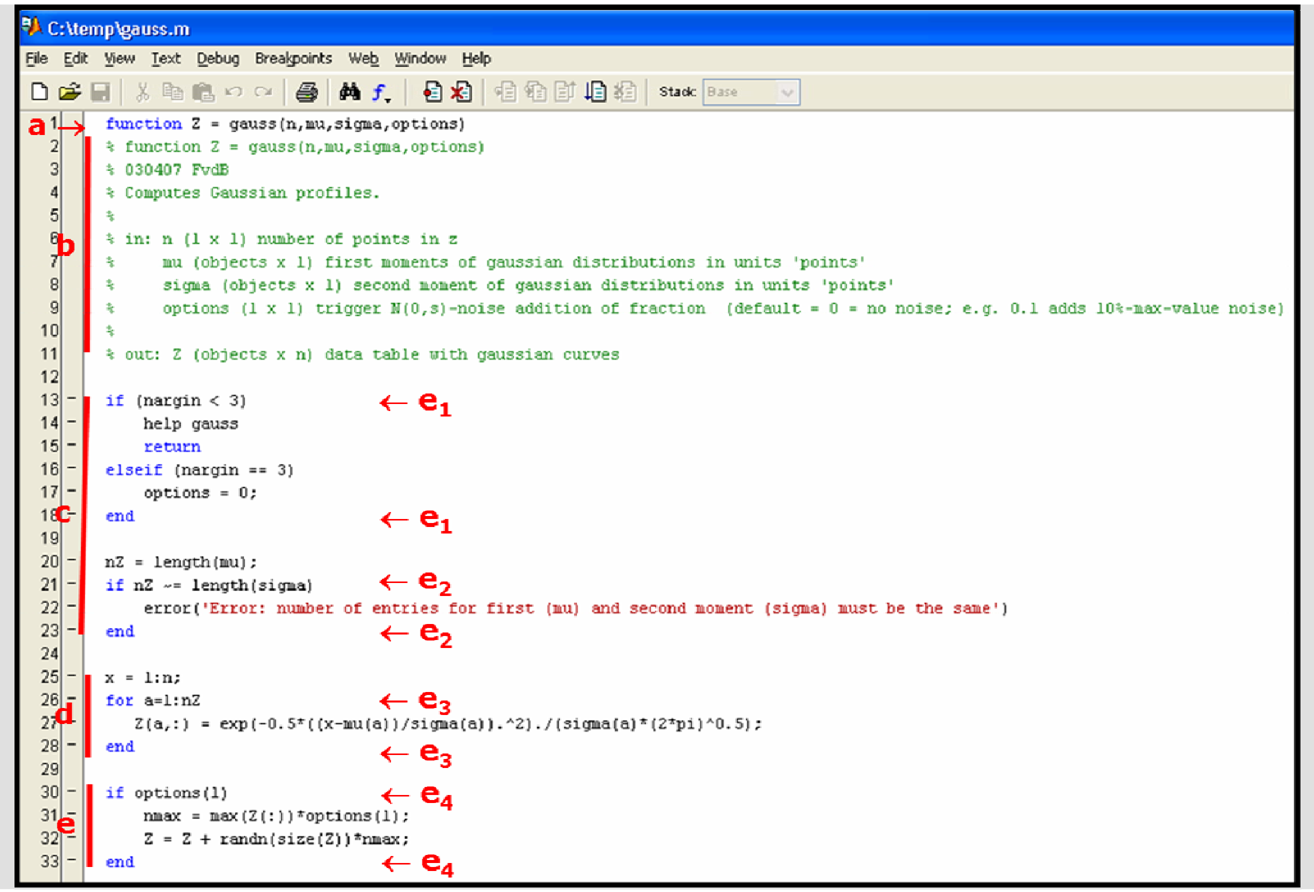
Note: scripts also are a great help and time-saver when the computations of yesterday looked a lot better than the "same" computations today, so use them!

Two related command:
>> type myscript

will print the contents of the file in the command window, and
>> open myscript (first close the editor for the best effect!)

will open a script in the text editor.

Scripts are the perfect tools for work in progress, but for an operation that you do on a daily basis so-called "functions" are more appropriate.
To load the function "gauss.m" into the editor type (make sure the computer exercise material from www.models.life.ku.dk → Teaching → Introduction to Matlab + … - bottom of the page, unzip file - is in the Matlab "work" directory)
>> open gauss



```
function Z = gauss(n,mu,sigma,options)
% function Z = gauss(n,mu,sigma,options)
% 030407 FvdB
% Computes Gaussian profiles.
%
% in: n (1 x 1) number of points in z
%     mu (objects x 1) first moments of gaussian distributions in units 'points'
%     sigma (objects x 1) second moment of gaussian distributions in units 'points'
%     options (1 x 1) trigger N(0,s)-noise addition of fraction  (default = 0 = no noise; e.g. 0.1 adds 10%-max-value noise)
%
% out: Z (objects x n) data table with gaussian curves

if (nargin < 3)
    help gauss
    return
elseif (nargin == 3)
    options = 0;
end

nZ = length(mu);
if nZ ~= length(sigma)
    error('Error: number of entries for first (mu) and second moment (sigma) must be the same')
end

x = 1:n;
for a=1:nZ
   Z(a,:) = exp(-0.5*((x-mu(a))/sigma(a)).^2)./(sigma(a)*(2*pi)^0.5);
end

if options(1)
    nmax = max(Z(:))*options(1);
    Z = Z + randn(size(Z))*nmax;
end
```

In the window you can see the typical structure of a Matlab function:

a) The function description with the function call name ("*gauss*"), input parameters ("n", "mu", "sigma" and "options") and output parameters ("Z"). Input is the data you pass from the Matlab workspace to the function, output is what the function returns to the workspace as answer to the function call. Note that the names proposed in the function header are only for explanation and internal use. You can pass any variable you like. So the call command *X = gauss(hundred,averages,spread)* is perfectly legal. It calls the function gauss, translates the real to virtual variables (e.g. "n" = "hundred"), and executes the commands. Note also that input and output parameters are optional. You could e.g. define no output parameter, so call *gauss(hundred,averages,spread)*. Then the function would compute the curves and the result would go … well nowhere (not very useful, but perfectly legal). We already made use of this for the "*plot*" command: nothing changed in the workspace, but we got a figure, so it can be very useful.

b) The help-text just after the function header should explain what to fill in to make the function perform. This is also the text you get on the screen when you type "*help gauss*".
c) The next step is usually some error tracking. In this example: check if enough input parameters were entered for the function (3), and see if the number of μ's and σ's are the same (every Gaussian distribution requires one of each).
d) Then the actual code – the hart of the function – is executed: compute distributions of length n-points for each entry in "mu" (and thus "sigma"), and store the result in "Z".
e) Archetypal for the internal of functions is computer programming flow-controls. We will not explain these programming tools in detail in this introductory course, but the three flow-controls in the figure are very intuitive: (1) check ("if … end") to see if the user passed the three obligatory input arguments ("*nargin*"), and if not show help end terminate the function ("*break*"); (2) check (second "if … end") to see if "mu" and "sigma" are of equal length, and if not terminate the function showing an error message in the workspace ("error"); (3) loop ("for … end") through the Gaussian distribution computation "nZ" times, and store the result in rows of matrix "Z"; (4) if input argument "options" is activated (fill in "1") some random noise will be added to the gauss-curve.

The important difference between scripts and functions is the way variables are handled. Every time a function call is made a new (small) workspace is created with all the variables necessary to execute all the commands inside the function. Thus, variable names inside the function and in the workspace are separated. If e.g. you have a variable "x" in your Matlab window, and you call the gauss-function, which creates its own "x" (line 25), the contents of "x" in your own workspace will remain the same. This is not the case for script or command-line calls, where the "x"-variables will be overwritten. Functions make a new work space, execute the required operation, clear all the temporary variables, and drop the output variable in the Matlab root-workspace (nice and clean!).

You will probably not write many functions yourself at this stage, but all the computations you will make (and actually many of the Matlab functions used so far, just type at the prompt open('mean')) are function-calls. So, you have to know what to give as input and output.

Another example:

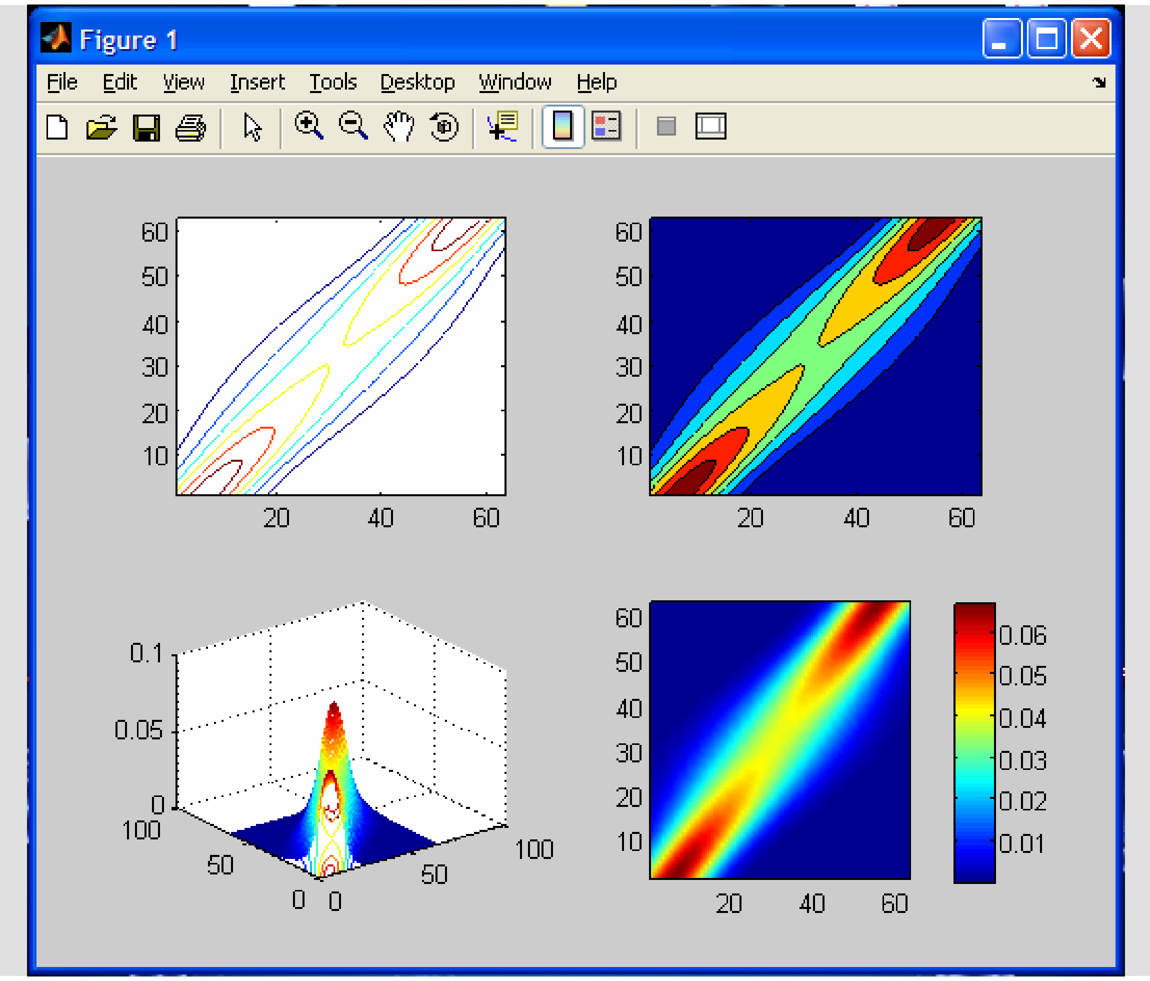
>> myx = -pi:.1:pi; (index running from -π to π with a step

```
                                       size of 0.1)
>> mymu = 31+myx*8;                    (some curve means)
>> mysig = cos(myx)*2+8;               (some standard deviations)
>> myX = gauss(length(mymu),mymu,mysig);        (make a data table)
```

Notices: a) "*length*" returns the length of a vector; b) try to figure out the size of "myX" with the command "*size*"; c) again, Matlab is case sensitive in variable names (myx ≠ myX)!

Now we have the data to do some advanced plotting (just try and see; with the [↻]-button in the figure window you can rotate plots to get a better view):

```
>> figure                              (opens an empty figure window)
>> plot(myx,myX)
>> mesh(myX)
>> surfc(myX); shading interp; colorbar hort
>> subplot(2,2,1)                      ("subplot" = small plots in figure window)
>> contour(myX)
>> subplot(2,2,2)
>> contourf(myX)
>> subplot(2,2,3)
>> meshc(myX)
>> subplot(2,2,4)
>> pcolor(myX); shading interp; colorbar vert
```



Just a few more commands that might come in handy:

>> *what*	(shows the *.mat and *.m file in the directory)
>> *which name*	(tells which file "name" you are using)
>> *dir*	(shows all the files in the working directory)
>> *cd*	(to change directories a-la dos-style)
>> *tic; pause(5); toc*	(these are really three functions: "tic" starts a stopwatch, "pause(5)" holds all operations for 5 seconds, while "toc" stops the stopwatch and shows the time elapse: a convenient way to time how long a Matlab function takes)

>> *x=rand(5,1); mean(x); std(x); var(x); sqrt(x); log(x); log10(x); exp(x);* (Matlab has of course a whole range of arrhythmic operators)

A few special elements:

>> *a = []*	(creates an empty or (0x0) matrix)
>> *b = NaN*	(assigns the "number" not-a-number to b, used to represent missing values inside a Matlab matrix)
>> *c = Inf*	(Assigns the "number" infinite to c, useful in logical operations)

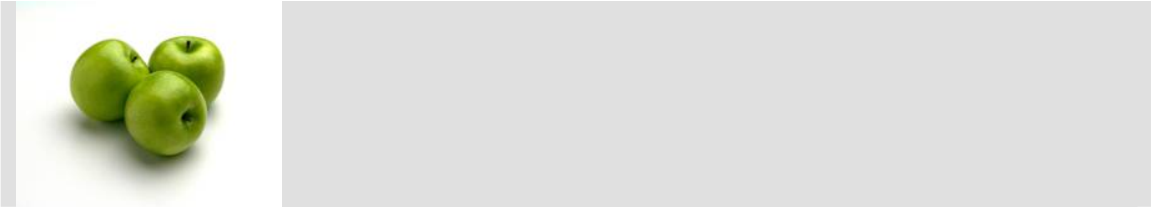
Manipulating Images

To get a little more practice in manipulating data tables we will import an image into matlab. (Notice that an image is nothing more then a multi-way array!)

First import and show the image into a variable called “A” (check the workspace)

```
>> A = imread('apple_green.jpg');
>> image(A);
```

We just loaded an image of green apples, 165 by 213 pixels. Color images are most often coded by three values Red, Green and Blue (so-called RGB-indexing). Hence the data cube is (165x213x3). The class of variables in this image is a special one called “uint8” or unsigned-integer-eight-bits which can hold values between whole 0 and 255.



Try the following commands

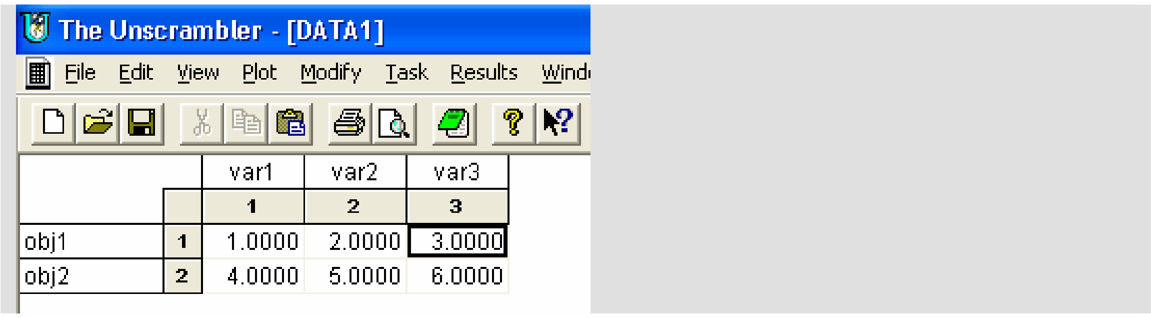
```
>> imagesc(A(:,:,1))          (only show the R-value/slab – notice sc = scale)
>> colormap gray              (use gray-scale colormap)
>> imagesc(A(:,:,2))          (only show the G-value/slab)
>> imagesc(A(:,:,1))          (only show the B-value/slab)
>> image(A(:,:,[3 1 2]))      (“false-images” - rearranging R, G and B )

>> image(A(50:130,80:160,:))          (zoom in on one apple)

>> mesh(double(A(:,:,1)))             (3D relief of the R-value matrix)

>> image(uint8(abs(double(A)-255)))   (never mind, just try it!)
```

From a data point of view images are nothing else then other matrices or data tables/cubes!

Importing data from The Unscrambler, LatentiX and MS-Excel into Matlab

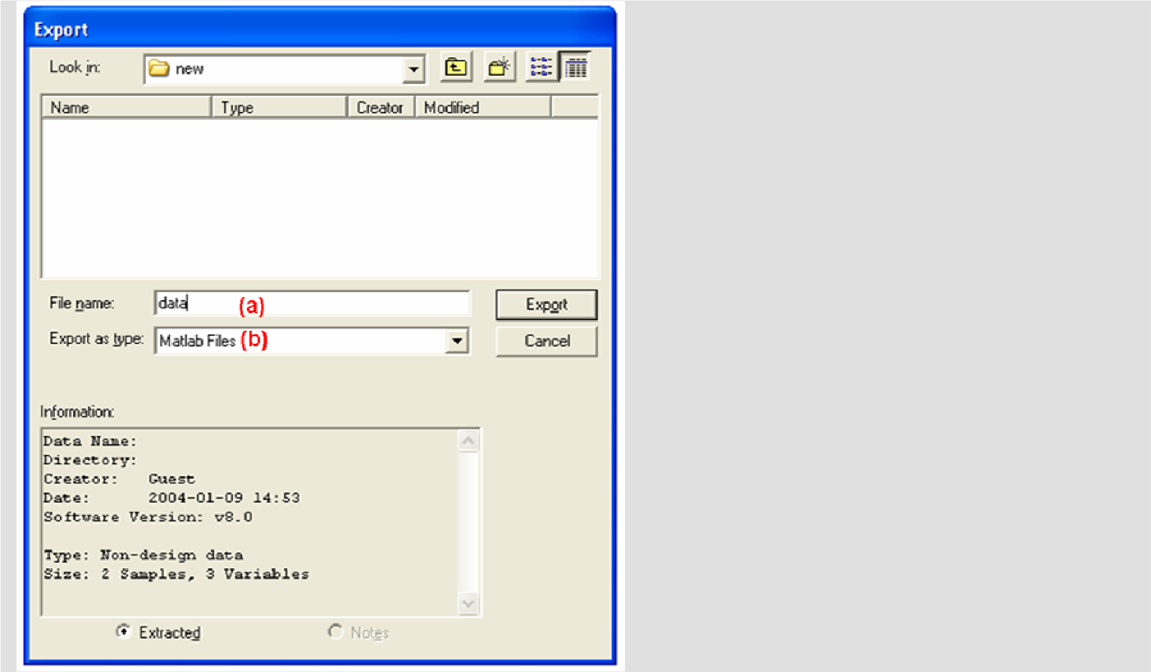
Many advanced ways for importing data into the Matlab exist (including *low-level C-like* programming). Most of these are too complicated for this introductory course, but we will try two routes - importing external information from The Unscrambler and MS-Excel, since experience has shown that many course people have their data in these formats.

Make a small dummy-set that looks like this in The Unscrambler (don't forget the object and variable names):



		var1	var2	var3
		1	2	3
obj1	1	1.0000	2.0000	3.0000
obj2	2	4.0000	5.0000	6.0000

Next, go to the menu [File] → [Export], then select a file-name (a) and "Matlab Export" as type (b):



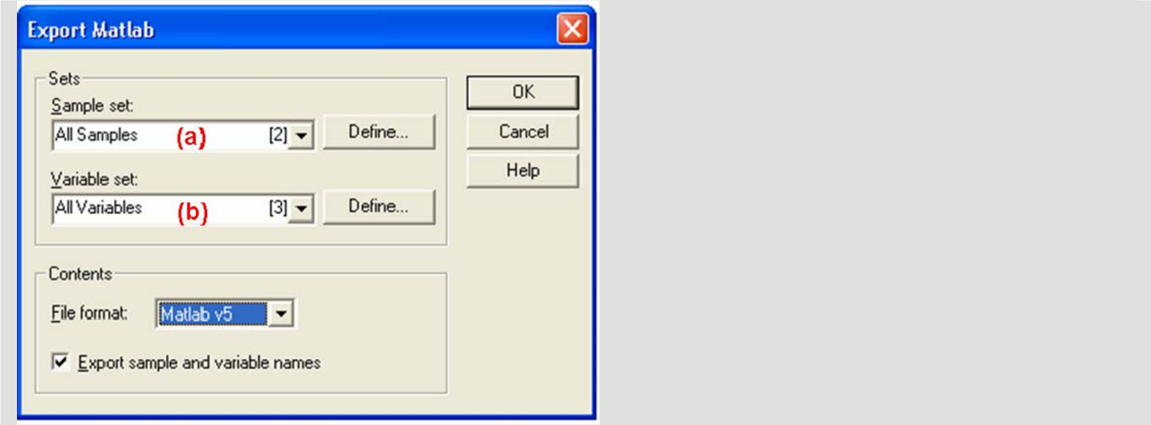
File name: data (a)

Export as type: Matlab Files (b)

```
Data Name:
Directory:
Creator:    Guest
Date:       2004-01-09 14:53
Software Version: v8.0

Type: Non-design data
Size: 2 Samples, 3 Variables
```

● Extracted ○ Notes

If subsets of Samples (a) or Variables (b) have been defined in the Unscrambler analysis, you can select which groups to export:



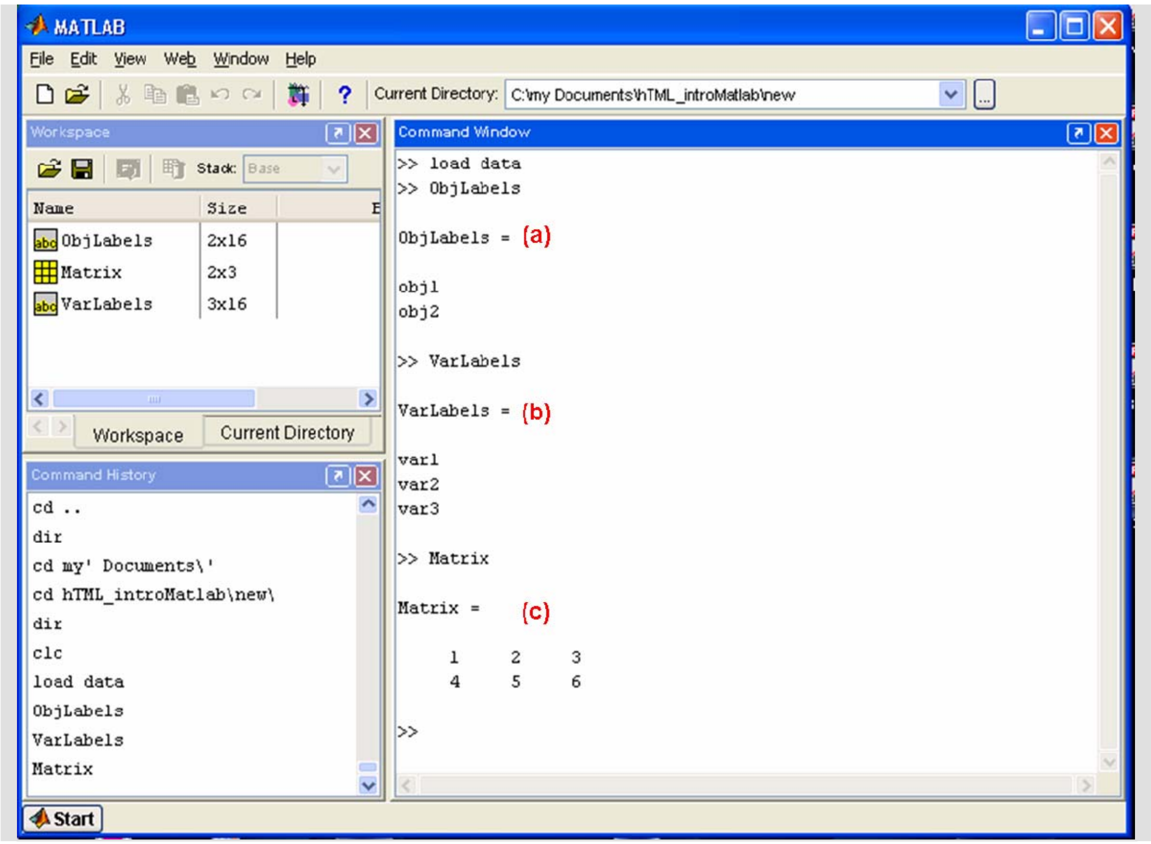
Sample set: All Samples (a) [2]

Variable set: All Variables (b) [3]

File format: Matlab v5

☑ Export sample and variable names

In Matlab the dummy-set will look like this, with (a) objects labels, (b) variable labels and (c) the data:



Current Directory: C:\my Documents\hTML_introMatlab\new

Name	Size
ObjLabels	2x16
Matrix	2x3
VarLabels	3x16

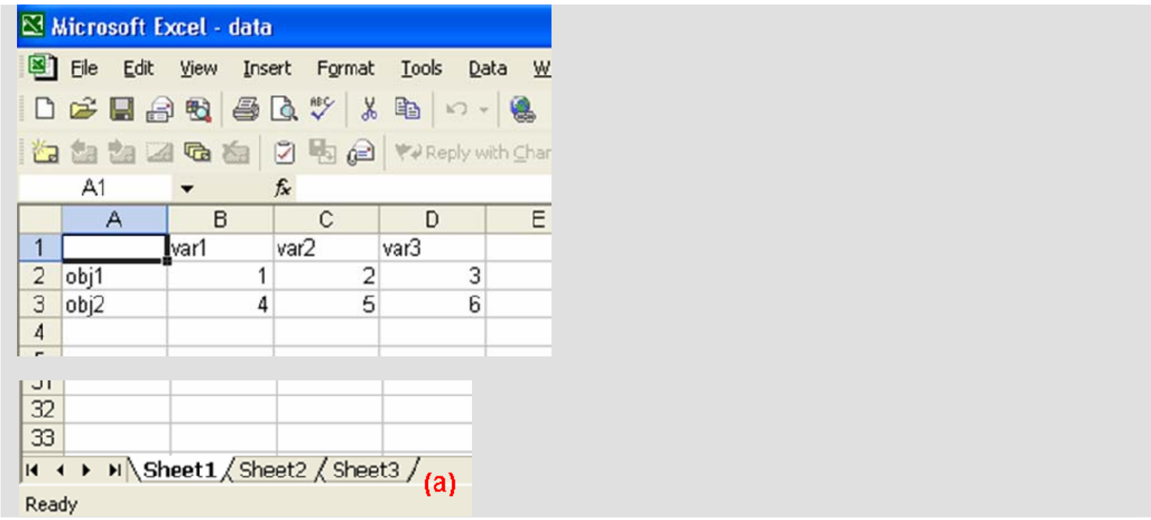
```
cd ..
dir
cd my' Documents\'
cd hTML_introMatlab\new\
dir
clc
load data
ObjLabels
VarLabels
Matrix
```

```
>> load data
>> ObjLabels

ObjLabels = (a)

obj1
obj2

>> VarLabels

VarLabels = (b)

var1
var2
var3

>> Matrix

Matrix =    (c)

     1     2     3
     4     5     6

>>
```

Importing data into Matlab from the data analysis program LatentiX (www.latentix.com) is straightforward since the information is stored in a compatible format with files extension "lxf". So, to read the data from a file called "data.lxf" and store the matrix in a variable "X" enter the following commands

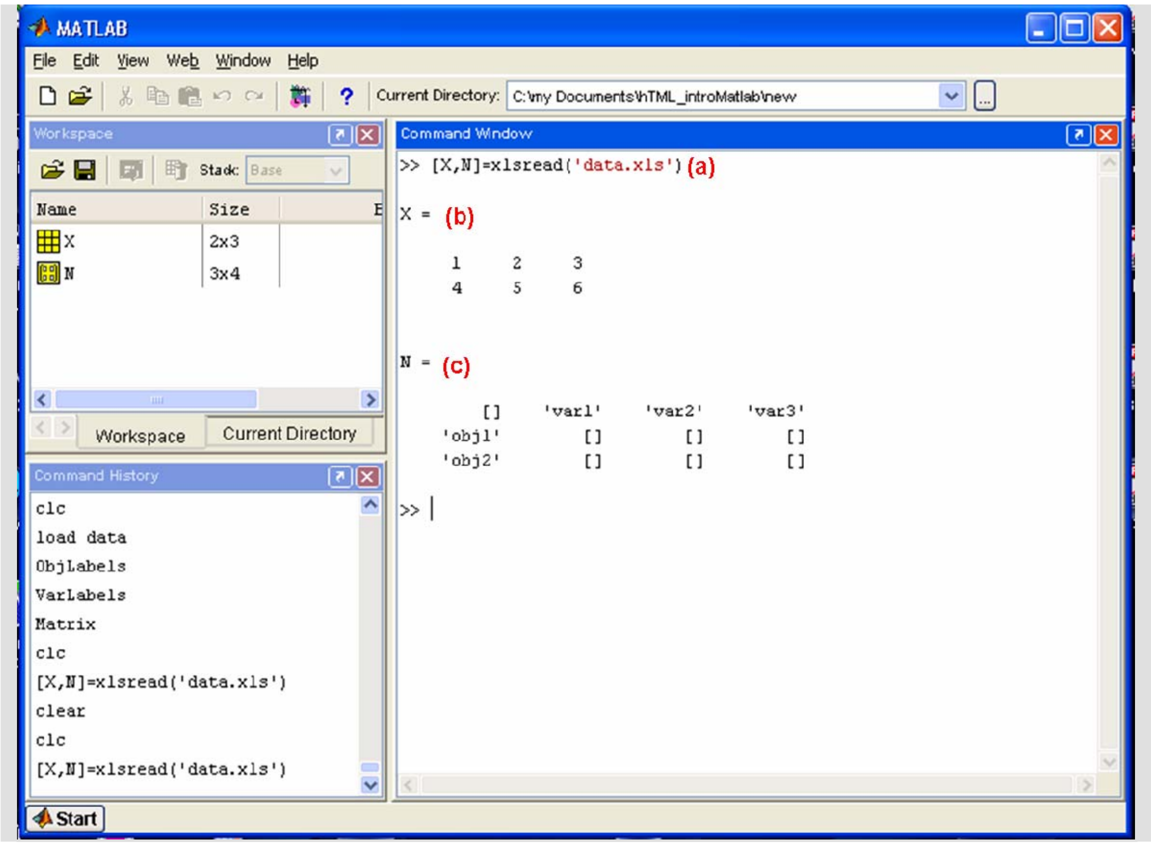
```
>> load -mat data.lxf
>> X = DataSet.data;
>> ObjLabels = DataSet.Labels{1};
>> VarLabels = DataSet.Labels{2};
```

There is a special Matlab function to import MS-Excel spreadsheets. The same dummy-set would look like this (make sure you only have one sheet to avoid problems – (a)):



	A	B	C	D	E
1		var1	var2	var3	
2	obj1	1	2	3	
3	obj2	4	5	6	
4					
32					
33					

In Matlab we can import the spreadsheet-file (call it "data.xls") via the following lines (a), where the data will be stored in **X** (b) and the none-numerical part in **N** (c):



Current Directory: C:\my Documents\hTML_introMatlab\new

Name	Size
X	2x3
N	3x4

```
>> [X,N]=xlsread('data.xls') (a)

X = (b)

     1     2     3
     4     5     6


N = (c)

        []    'var1'    'var2'    'var3'
    'obj1'        []        []        []
    'obj2'        []        []        []

>>
```

```
clc
load data
ObjLabels
VarLabels
Matrix
clc
[X,N]=xlsread('data.xls')
clear
clc
[X,N]=xlsread('data.xls')
```

Be aware that it might take some puzzling to figure out where the numbers are in the Matlab matrices, especially if text and number are intertwined in the spreadsheet. The easiest route is to make new spreadsheets (e.g. one with numbers, one with object labels, one with variable labels, etc.) that are trivial to read.

Notice: starting Matlab 7/R14 there is also a function to write to Excel files called "xlswrite". We will not explain it here, but it might be good to know for later use.

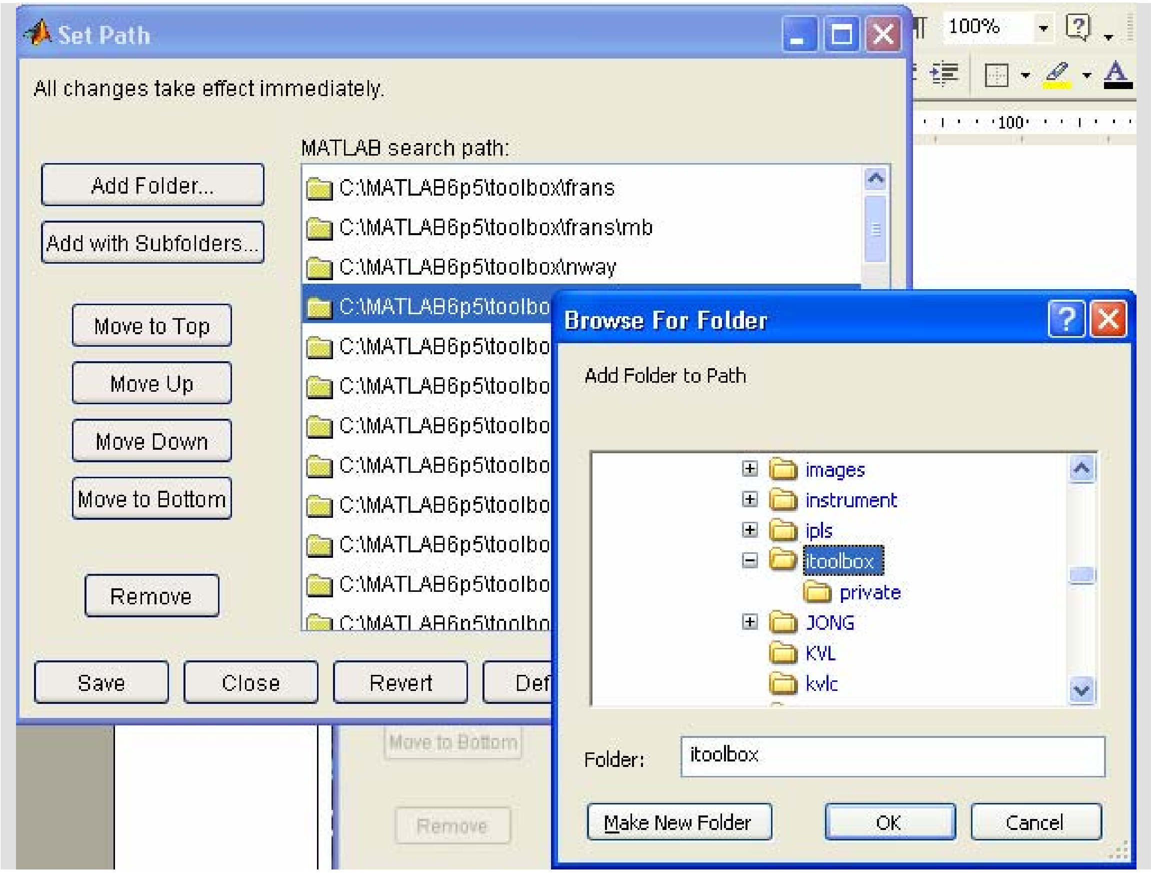
The "path" and third party products

We already have encountered many Matlab commands. A way to see where the source of the command can be found is as follows:
>> which rand
the message on the screen shows that "*rand*" is a so-called build-in function.
>> which surf

now the screen message shows that "*surf*" is a function that can be found in "x:\MATLABXXX\toolbox\matlab\graph3d\surf.m", part of the 3D-graphics function-set.
Next to the internal function Matlab comes with a number of default toolboxes. They consist of hundreds of different functions, organized in conceptually meaningful groups. You can also buy additional toolboxes for special, very diverse purposes (e.g. statistical computations, or instrumental data input-output over the hardware computer interfaces). Even better: many scientists make there code available for free. This means you can do a lot of research by lending other peoples program code, without going into programming details (just search the internet!). However, you still have to understand how to communicate with other people's code via input and output parameters. In this paragraph we will treat organizing new features in the form of toolboxes added to the Matlab search path using the *i*PLS (interval-PLS) from the *i*Toolbox written by Lars Nørgaard.

You can find the source code for the *i*Toolbox at the KVL Quality&Technology homepage under:
www.models.life.ku.dk → Research → Algorithms and code → *i*Toolbox

Here you can find important information on the author, ownership and use. Download the zip-file to your own computer. Make a new directory in the path "Matlab\toolbox" called "itoolbox", and extract the contents of the zip-file to this directory. The next step is to add the new toolbox to the Matlab search path.
In the workspace editor select: [File] → [Set Path] → [Add Folder] → find [toolbox] and then [itoolbox] in the browser → [Save] → [Close] (see figure below for details).



Set Path

All changes take effect immediately.

MATLAB search path:

```
C:\MATLAB6p5\toolbox\frans
C:\MATLAB6p5\toolbox\frans\mb
C:\MATLAB6p5\toolbox\nway
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
C:\MATLAB6p5\toolbo
```

Browse For Folder

Add Folder to Path

Folder: itoolbox

To check if it works type
>> which ipls
>> help ipls

This will write a small help text for the toolbox in the workspace window. Can you figure out what the input and output to this function should be? We will show by illustration. The toolbox comes with an example data file called "nirbeer" (Near Infrared spectra of beer samples):

>> clear	(clear the workspace)
>> load nirbeer	(load data)
>> whos	(to see what we loaded)
>> M=ipls(Xcal,ycal,8,'mean',40,xaxis,'full',40);	
	(compute the model, 8 factors, mean centering, 40 intervals, full cross-validation and store the result in structure "M")
>> iplsplot(M,'intlabel')	(plot the results with numerical labels)

We will not go into the interpretation of this model. If you want to learn more about the *i*PLS method you can run the *i*PLSdemo-routine and study the manual included in the toolbox.

A few important notes: a) You have much liberty in selecting names for variables and functions which sometimes leads to problems; the order is roughly: variables in the workspace, function-files in the working directory and function-files in top-to-bottom order in the directories of the path; b) To keep things in perspective you have to be clear in naming functions and variables; c) If you get strange/unexpected error messages, use the "*which*" command to see which function Matlab is using, if you still get strange errors you can always try the "*why*" command.

As shown above, finding the right input to other peoples functions – of course depending on the help available and the didactic skills of the author – can be pretty complicated. One (very luxurious albeit time consuming way) to circumvent problems is to write a Graphical Users Interface (GUI). In short: GUI's control the user's interaction with a toolbox/set of functions by limiting the input to predefined values and a restricted (graphical) output.

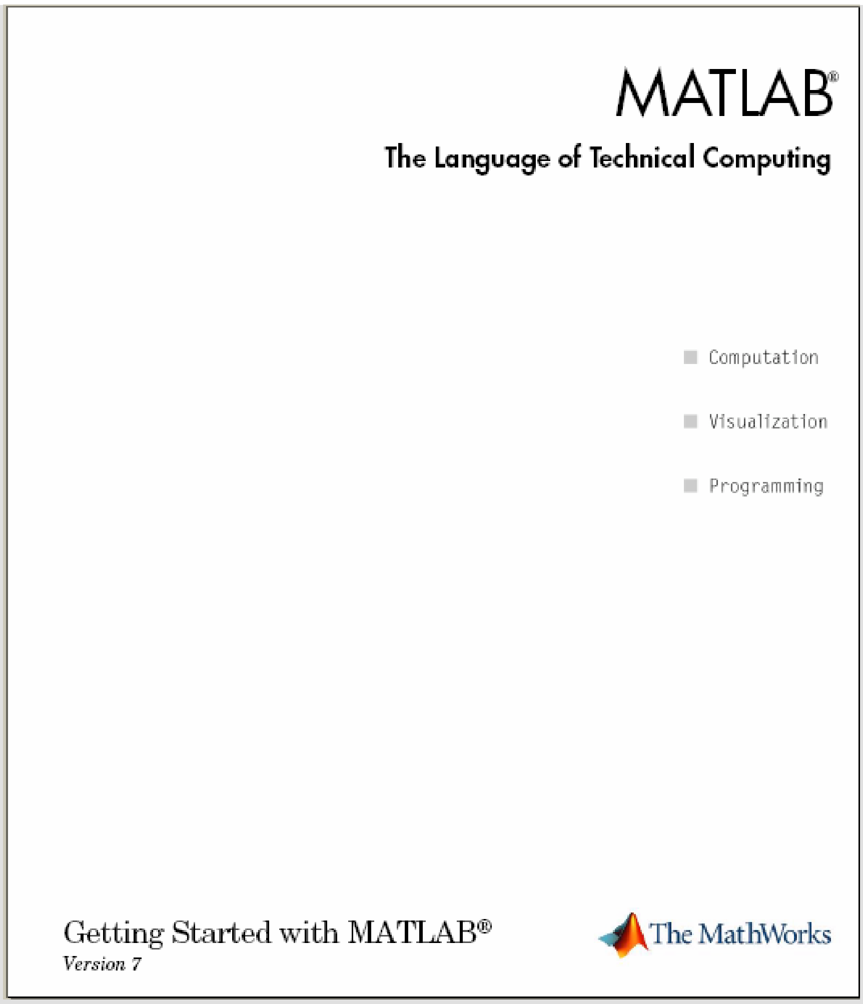
One last helpful command is "*lookfor*". It will scan all the functions on the "path" for a particular text fragment. E.g. if you forgot the command for standard deviation enter the following line (beware, searching all the function files might take some time)

```
>> lookfor deviation
```

More reading material

This is all the basic stuff you need to know about Matlab. If you want more (introductory) information a pdf-file called "Getting Started with MATLAB" (*getstart.pdf*) can help out. The following parts are of interest for this introductory course (see list below for details):

1. What is Matlab (p.12)
2. Matrices and Arrays (p.19-27)
3. Basic Plotting Functions (p.88-111)
4. Flow Control and Other Data Structures (p.126-143, though not every detail!). assigned



MATLAB®

The Language of Technical Computing

Getting Started with MATLAB®

Version 7

In the next part on bilinear factor models we will only use a few specific, additional functions.

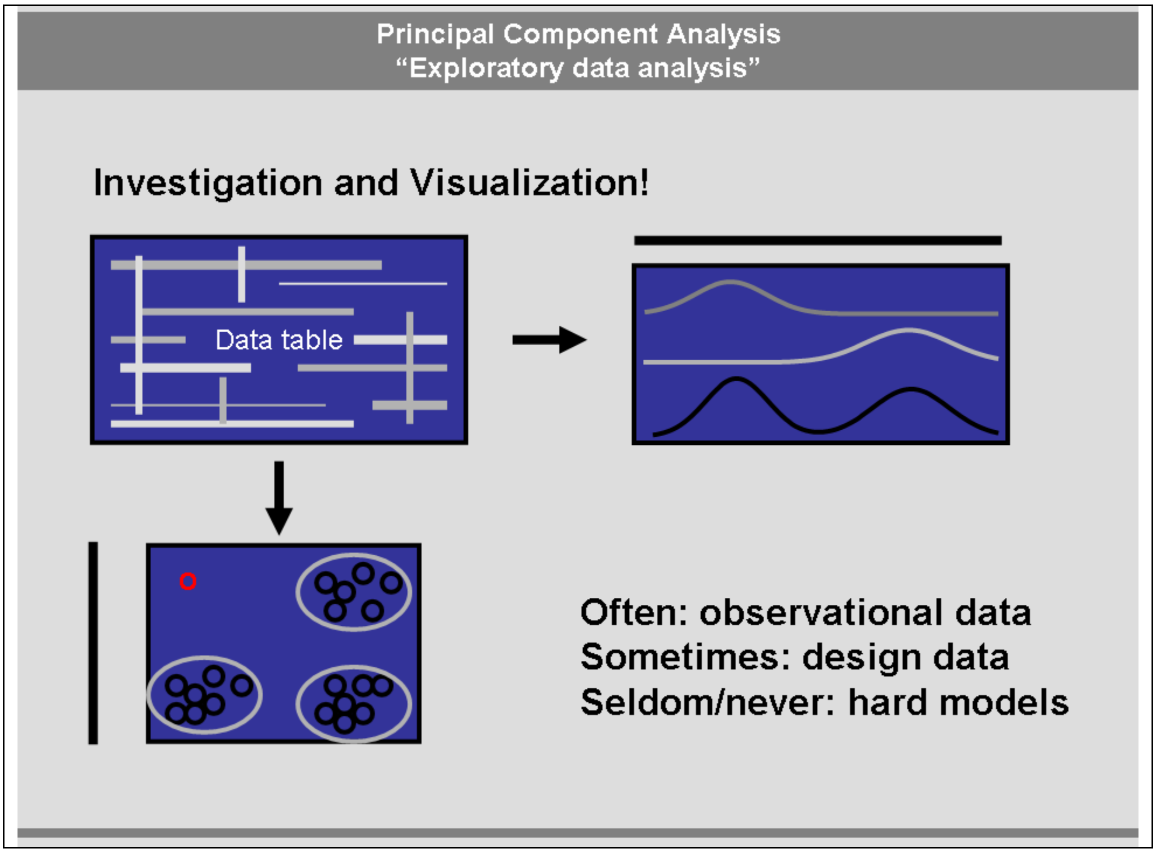
(a : important; b : less important; c : not important for this course)

1 Introduction				
What Is MATLAB?	1-2	a		
The MATLAB System	1-3			c
MATLAB Documentation	1-4			c
MATLAB Online Help	1-4			c
Starting and Quitting MATLAB	1-6	a		
Starting MATLAB	1-6	a		
Quitting MATLAB	1-6	a		
MATLAB Desktop	1-7	a		
2 Matrices and Arrays				
Matrices and Magic Squares	2-2	a		
Entering Matrices	2-3	a		
sum, transpose, and diag	2-4	a		
Subscripts	2-6	a		
The Colon Operator	2-7	a		
The magic Function	2-8	a		
Expressions	2-10	a		
Variables	2-10	a		
Numbers	2-11	a		
Operators	2-11	a		
Functions	2-12	a		
Examples of Expressions	2-13	a		
Working with Matrices	2-14	a		
Generating Matrices	2-14	a		
The load Function	2-15	a		
M-Files	2-15	a		
Concatenation	2-16	a		
Deleting Rows and Columns	2-17	a		
More About Matrices and Arrays	2-18	a		
Linear Algebra	2-18	a		
Arrays	2-21	a		
Multivariate Data	2-24	a		
Scalar Expansion	2-25	a		
Logical Subscripting	2-25	a		
The find Function	2-26	a		
Controlling Command Window Input and Output	2-28	a		
The format Function	2-28	a		
Suppressing Output	2-30	a		
Entering Long Statements	2-30	a		
Command Line Editing	2-30	a		
3 Graphics				
Overview of MATLAB Plotting	3-2		b	
The Plotting Process	3-2			c
Graph Components	3-5			c
Figure Tools	3-7			c
Arranging Graphs Within a Figure	3-13			c
Selecting Plot Types	3-14			c
Editing Plots	3-16			c
Plot Editing Mode	3-16			c
Using Functions to Edit Graphs	3-19			c
Examples — Using MATLAB Plotting Tools	3-20			c
Modifying the Graph Data Source	3-27			c
Preparing Graphs for Presentation	3-29			
Modify the Graph to Enhance the Presentation	3-30			
Printing the Graph	3-33			
Exporting the Graph	3-35			
Basic Plotting Functions	3-38	a		
Creating a Plot	3-38	a		
Multiple Data Sets in One Graph	3-40	a		
Specifying Line Styles and Colors	3-41	a		
Plotting Lines and Markers	3-41	a		
Imaginary and Complex Data	3-43			c
Adding Plots to an Existing Graph	3-44		b	
Figure Windows	3-46	a		
Multiple Plots in One Figure	3-46	a		
Controlling the Axes	3-48	a		
Axis Labels and Titles	3-49	a		
Saving Figures	3-51	a		
Mesh and Surface Plots	3-52	a		
Visualizing Functions of Two Variables	3-52	a		

Images	3-58	a		
Reading and Writing Images	3-59	a		
Printing Graphics	3-60		b	
Handle Graphics	3-62			c
Using the Handle	3-62			c
Graphics Objects	3-63			c
Setting Object Properties	3-65			c
Specifying the Axes or Figure	3-69			c
Finding the Handles of Existing Objects	3-70			c
Animations	3-72			c
Erase Mode Method	3-72			c
Creating Movies	3-74			c
4 Programming				
Flow Control	4-2		b	
if, else, and elseif	4-2		b	
switch and case	4-4		b	
for	4-5		b	
while	4-6		b	
continue	4-6		b	
break	4-7			c
try – catch	4-8			c
return	4-8			c
Other Data Structures	4-9	a		
Multidimensional Arrays	4-9	a		
Cell Arrays	4-11	a		
Characters and Text	4-13	a		
Structures	4-16	a		
Scripts and Functions	4-20	a		
Scripts	4-21	a		
Functions	4-22	a		
Types of Functions	4-24	a		
Global Variables	4-26			c
Passing String Arguments to Functions	4-27			c
The eval Function	4-28			c
Function Handles	4-29			c
Function Functions	4-30			c
Vectorization	4-32			c
Preallocation	4-33			c
5 Creating Graphical User Interfaces				**c**
6 Desktop Tools and Development Environment				**c**

Principal Component Analysis

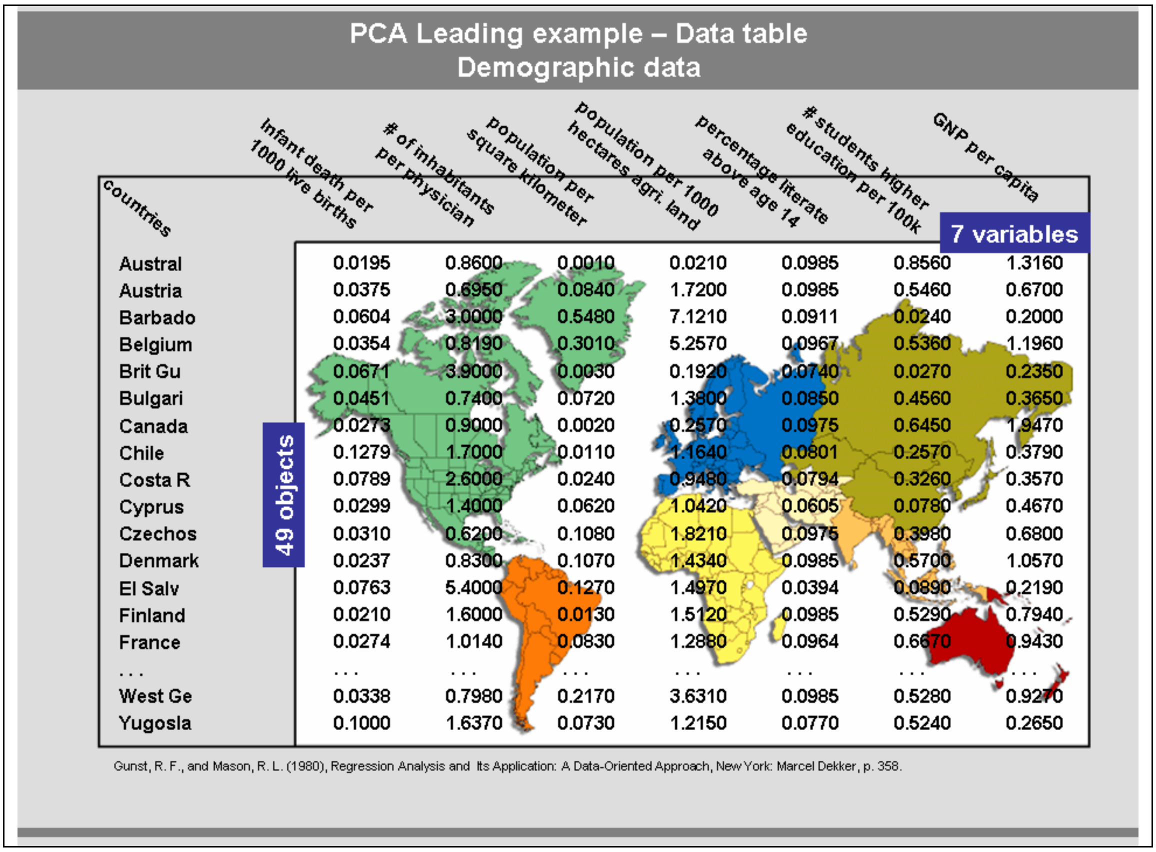
The next part of this document contains the transparencies used in the lectures on PCA, accompanied by some "telegram-style" notes on the main issues.



Principal Component Analysis
"Exploratory data analysis"

Investigation and Visualization!

Often: observational data
Sometimes: design data
Seldom/never: hard models



PCA Leading example – Data table
Demographic data

countries	Infant death per 1000 live births	# of inhabitants per physician	population per square kilometer	population per 1000 hectares agri. land	percentage literate above age 14	# students higher education per 100k	GNP per capita
Austral	0.0195	0.8600	0.0010	0.0210	0.0985	0.8560	1.3160
Austria	0.0375	0.6950	0.0840	1.7200	0.0985	0.5460	0.6700
Barbado	0.0604	3.0000	0.5480	7.1210	0.0911	0.0240	0.2000
Belgium	0.0354	0.8190	0.3010	5.2570	0.0967	0.5360	1.1960
Brit Gu	0.0671	3.9000	0.0030	0.1920	0.0740	0.0270	0.2350
Bulgari	0.0451	0.7400	0.0720	1.3800	0.0850	0.4560	0.3650
Canada	0.0273	0.9000	0.0020	0.2570	0.0975	0.6450	1.9470
Chile	0.1279	1.7000	0.0110	1.1640	0.0801	0.2570	0.3790
Costa R	0.0789	2.6000	0.0240	0.9480	0.0794	0.3260	0.3570
Cyprus	0.0299	1.4000	0.0620	1.0420	0.0605	0.0780	0.4670
Czechos	0.0310	0.6200	0.1080	1.8210	0.0975	0.3980	0.6800
Denmark	0.0237	0.8300	0.1070	1.4340	0.0985	0.5700	1.0570
El Salv	0.0763	5.4000	0.1270	1.4970	0.0394	0.0890	0.2190
Finland	0.0210	1.6000	0.0130	1.5120	0.0985	0.5290	0.7940
France	0.0274	1.0140	0.0830	1.2880	0.0964	0.6670	0.9430
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
West Ge	0.0338	0.7980	0.2170	3.6310	0.0985	0.5280	0.9270
Yugosla	0.1000	1.6370	0.0730	1.2150	0.0770	0.5240	0.2650

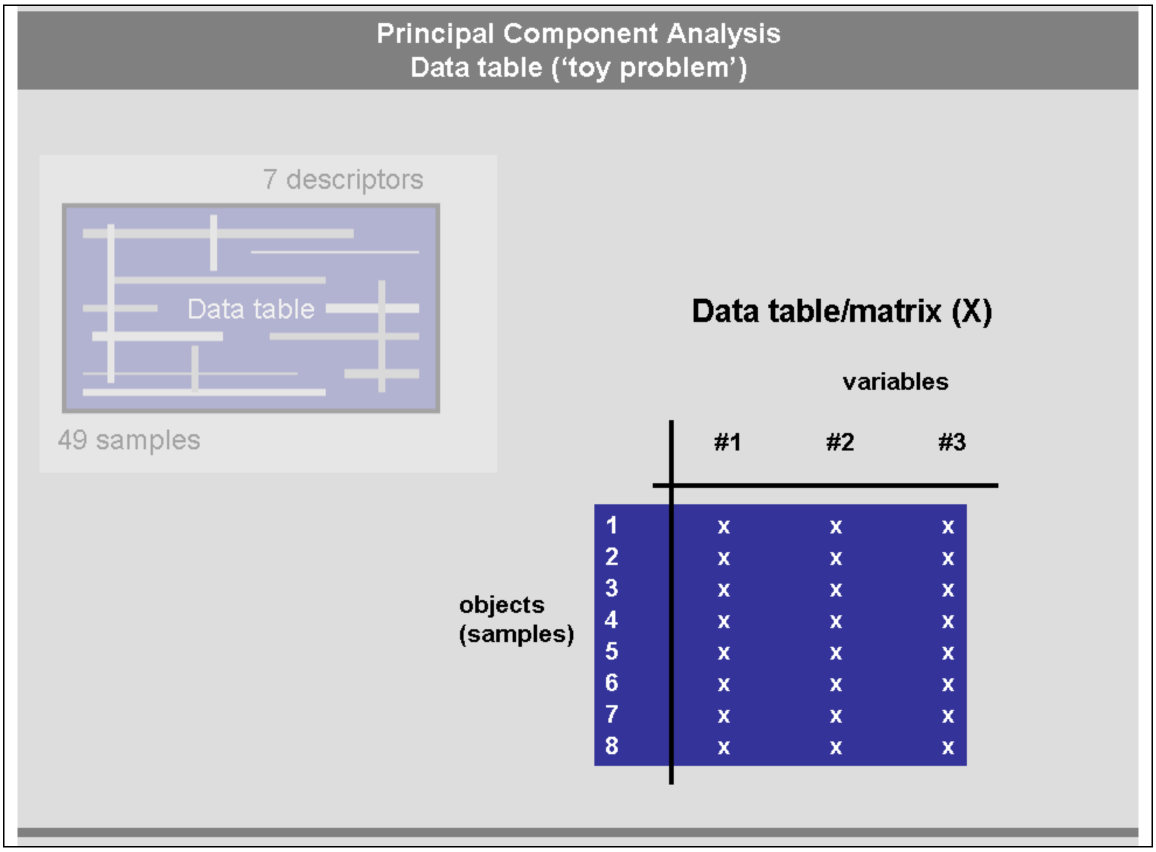
Gunst, R. F., and Mason, R. L. (1980), Regression Analysis and Its Application: A Data-Oriented Approach, New York: Marcel Dekker, p. 358.

We will use a small example as a reminder: socio-economical status of 49 countries world-wide determined by 7 indicator variables/demographic characteristics:

1) Infant death per 1000 live births
2) # of inhabitants per physician
3) population per square kilometer
4) population per 1000 hectares of agricultural land
5) Percentage literate of population aged 15 years and over
6) # of students enrolled in higher education per 100000 population
7) gross national product per capita (US dollars)

Notice that we don't go blind into the data: we have "independent measurements" in the form of numbers, but we also know about the sample (countries). The knowledge is important for interpretation.

R.F. Gunst and R.L. Mason "Regression Analysis and Its Application: A Data-Oriented Approach" Marcel Dekker, New York, p. 358(1980)



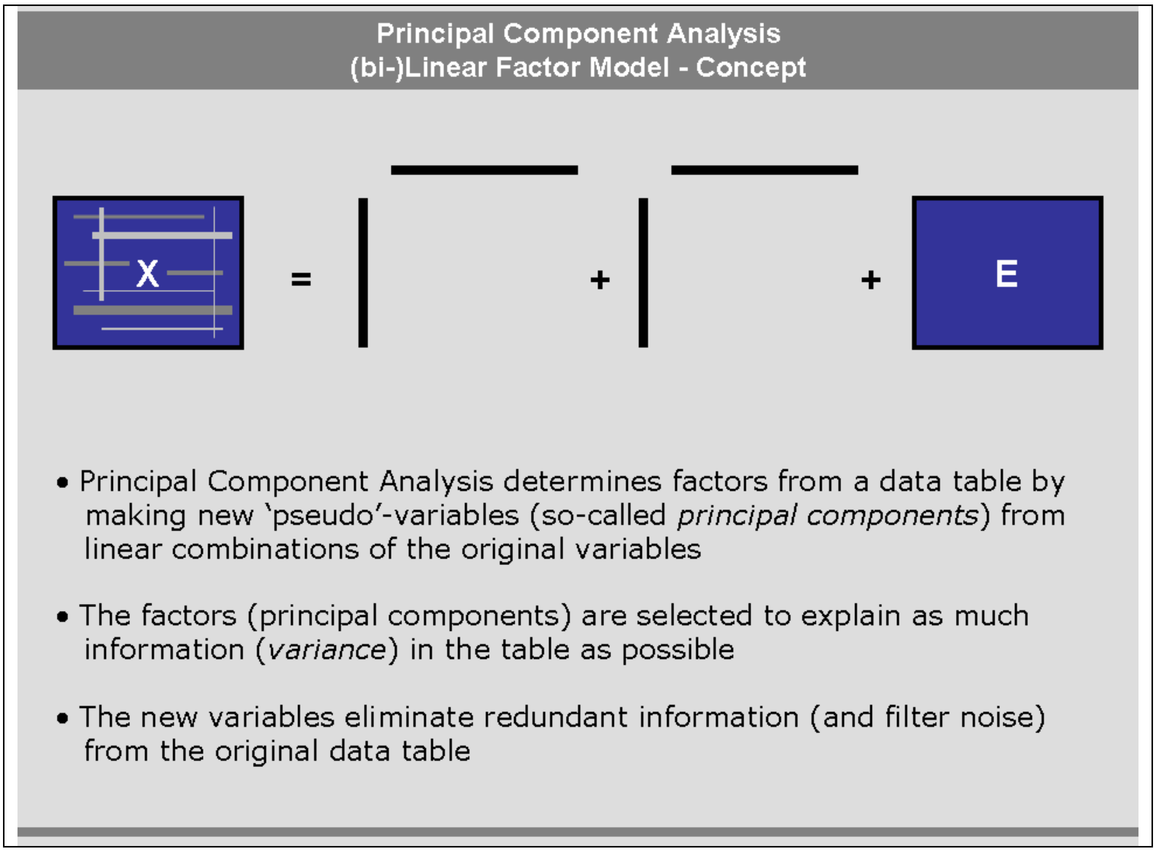
Principal Component Analysis
Data table ('toy problem')

Data table/matrix (X)

variables

objects (samples)	#1	#2	#3
1	x	x	x
2	x	x	x
3	x	x	x
4	x	x	x
5	x	x	x
6	x	x	x
7	x	x	x
8	x	x	x

Principal Component Analysis (PCA) can be used to analyze tables of data, where a number of the same variables are measured/collected on a set of samples (called objects). The aim is to find the important things happening in the table: separate the main directions/principal components form the noise.
Stated differently: extract what objects have in common from what makes objects unique.



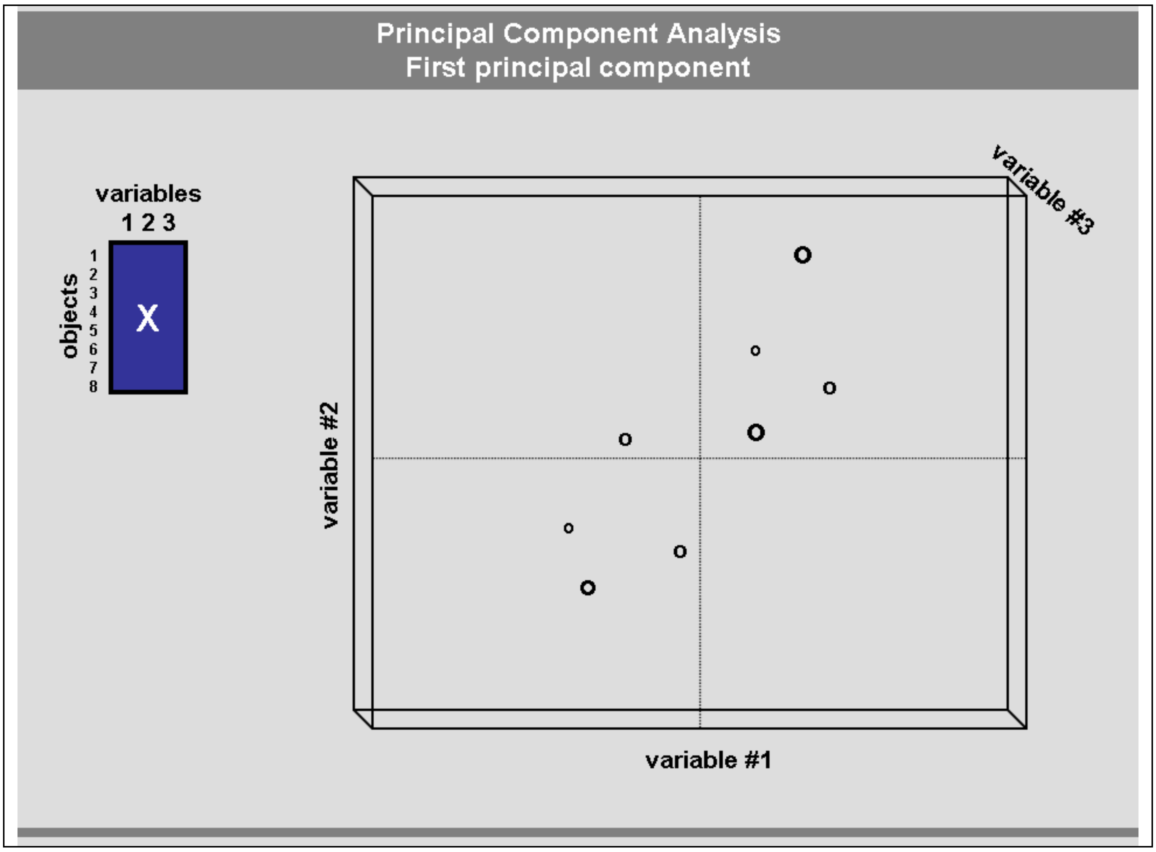
Principal Component Analysis
(bi-)Linear Factor Model - Concept

- Principal Component Analysis determines factors from a data table by making new 'pseudo'-variables (so-called *principal components*) from linear combinations of the original variables

- The factors (principal components) are selected to explain as much information (*variance*) in the table as possible

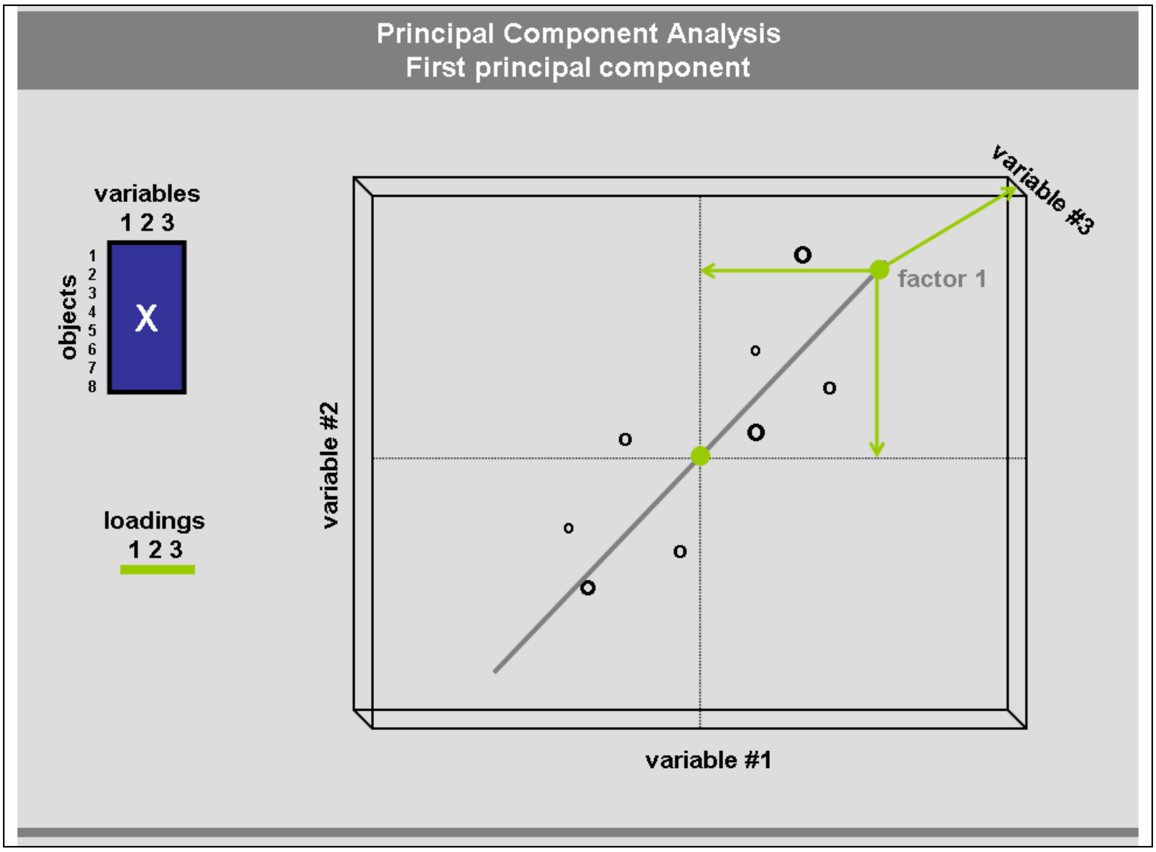
- The new variables eliminate redundant information (and filter noise) from the original data table

There are many different objectives for doing PCA – often different viewpoints associated with the specific fields of science – but will focus our attention on these three.



As a small example we will take a matrix **X** of eight object and three variables (8 x 3). We can visualize this table as eight points in a three dimensional space. The objects have a particular score on the three variables, and the position of an object is fixed by its three values in the Cartesian coordinate system.

In the same line we can picture (in our mind!) an eight dimensional space where the three variables (three points) are expressed in the objects-space (it is a lot harder to visualize this, but will use this mental picture later on!).



Principal Component Analysis
First principal component

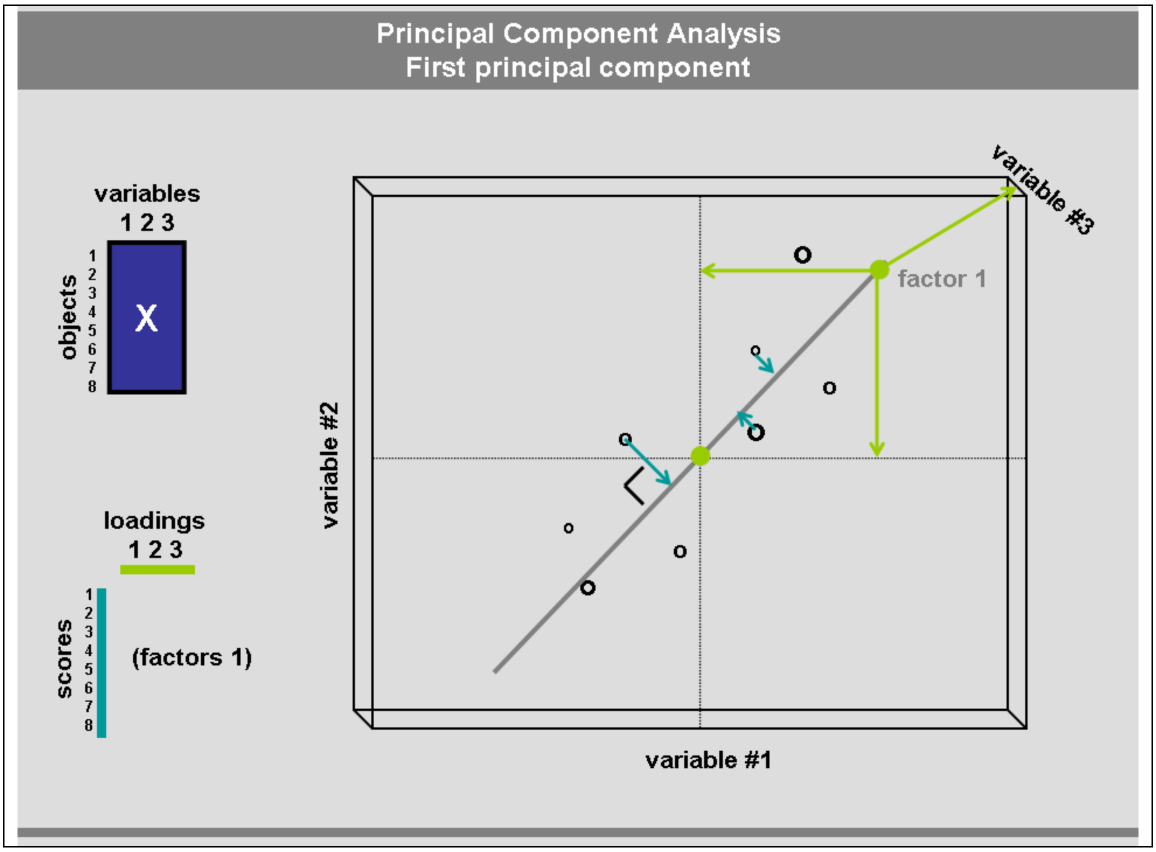
The first factor in PCA is:

- The direction that "spans" the objects the best
- Explains the maximum amount of variance in the data set
- Finds the common ("most common") direction in the set

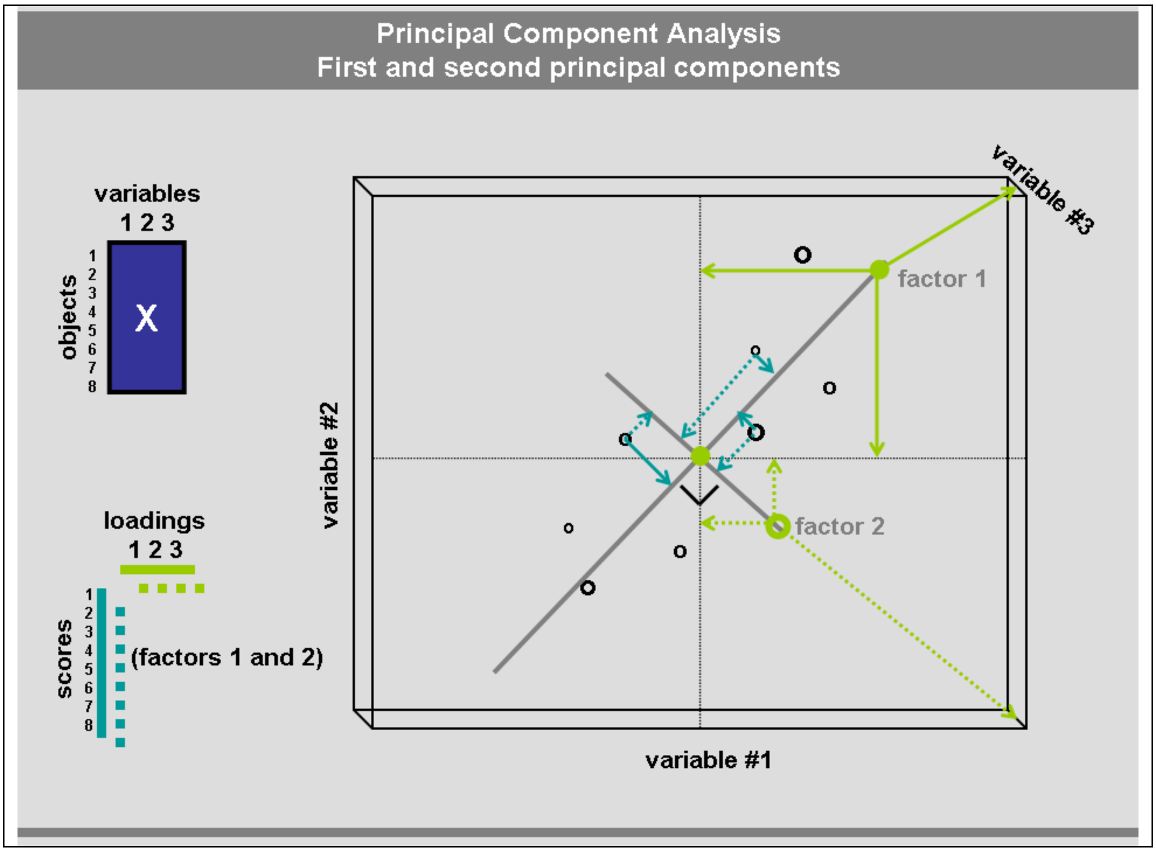
(Three times the same thing!).

To fixate the new direction in the original variable space we define three loading-values (traditionally called **p**) plus the origin: two points to fix a line.

Principal Component Analysis
First principal component

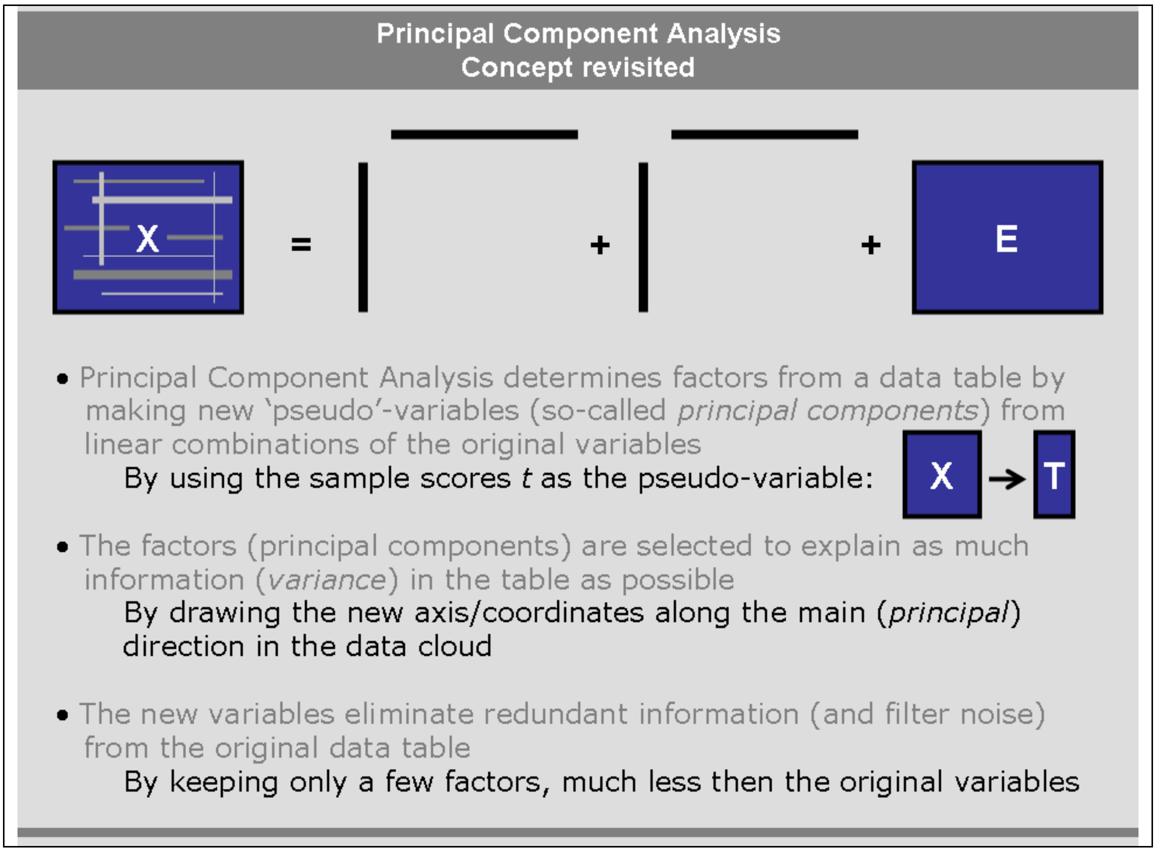


By projecting the sample-points on the new coordinate line (determined by the loadings), every object gets a score value (traditionally called **t**). The **t** and **p** vector pair form the first factor (or principal component). The outer product of **t** and **p** has the same number of entries as the original matrix, and this "reconstruction" forms the best rank one (outer product) approximation of **X**.



In the same way a second factor can be determined, where we select the new coordinate perpendicular to the first one (and again going through the origin).

We can continue in extracting factors until all the systematic variance is captured and only noise/information on the individual object level remains.



Principal Component Analysis
Concept revisited

- Principal Component Analysis determines factors from a data table by making new 'pseudo'-variables (so-called *principal components*) from linear combinations of the original variables

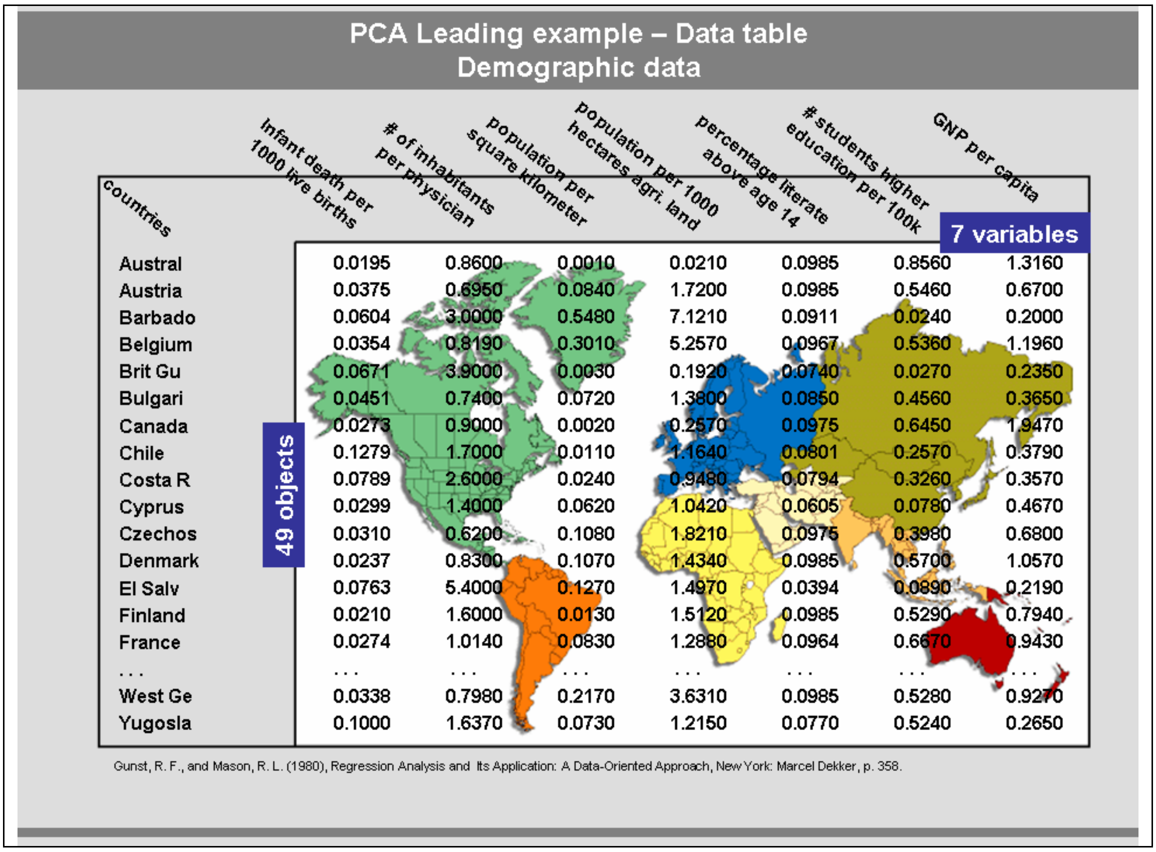
  By using the sample scores *t* as the pseudo-variable: X → T

- The factors (principal components) are selected to explain as much information (*variance*) in the table as possible

  By drawing the new axis/coordinates along the main (*principal*) direction in the data cloud

- The new variables eliminate redundant information (and filter noise) from the original data table

  By keeping only a few factors, much less then the original variables



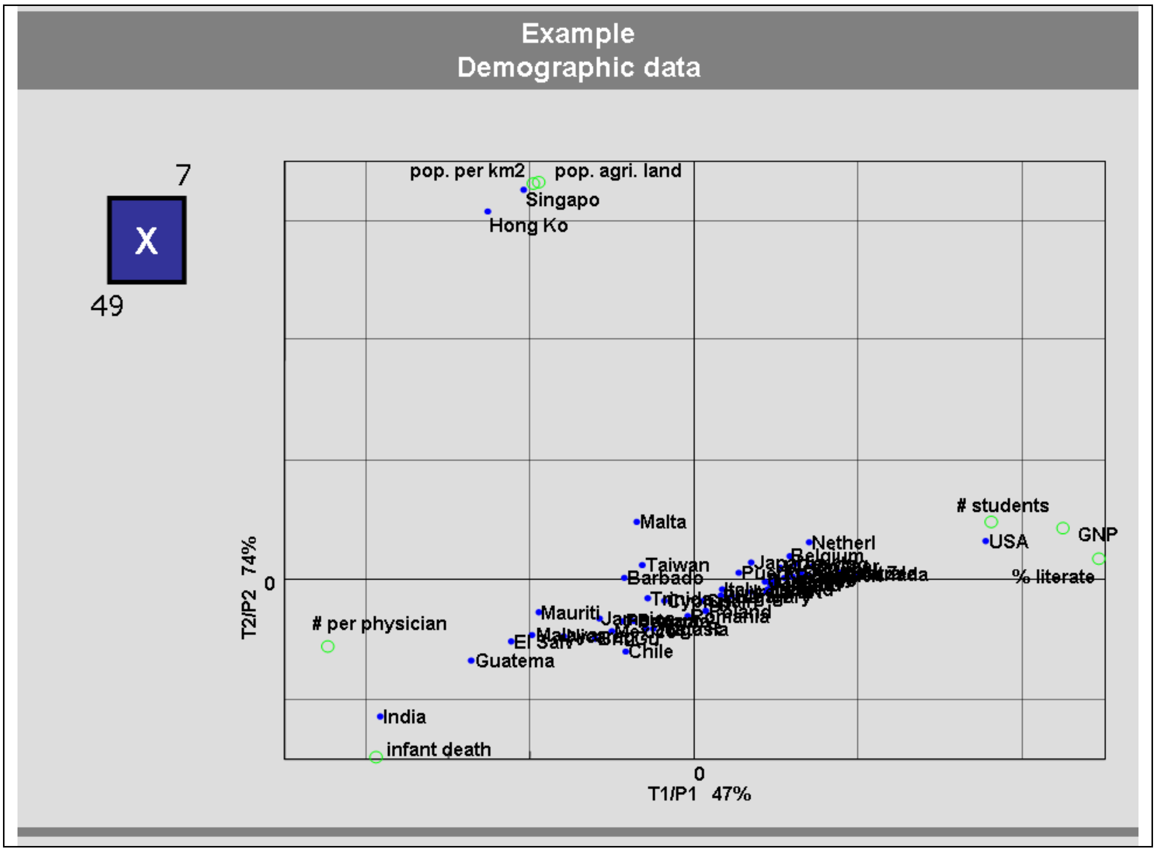
PCA Leading example – Data table
Demographic data

7 variables

49 objects

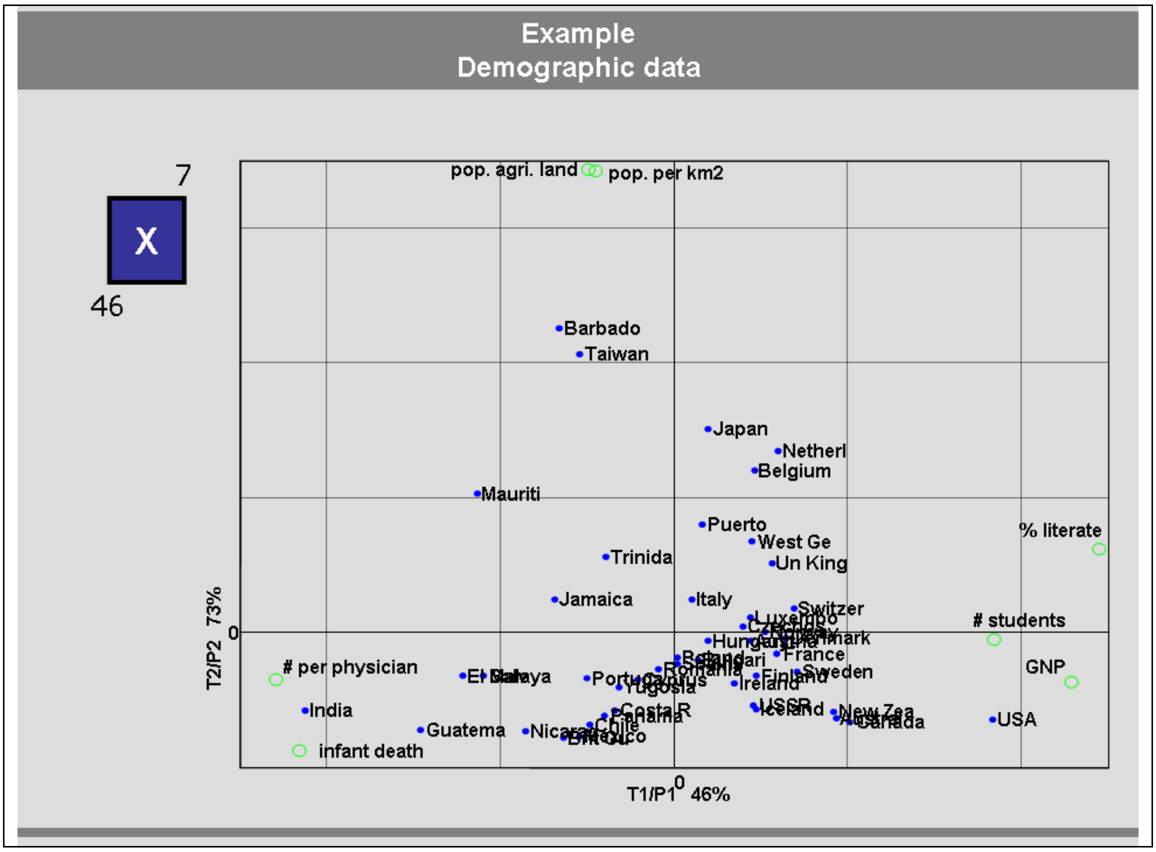
countries	Infant death per 1000 live births	# of inhabitants per physician	population per square kilometer	population per 1000 hectares agri. land	percentage literate above age 14	# students higher education per 100k	GNP per capita
Austral	0.0195	0.8600	0.0010	0.0210	0.0985	0.8560	1.3160
Austria	0.0375	0.6950	0.0840	1.7200	0.0985	0.5460	0.6700
Barbado	0.0604	3.0000	0.5480	7.1210	0.0911	0.0240	0.2000
Belgium	0.0354	0.8190	0.3010	5.2570	0.0967	0.5360	1.1960
Brit Gu	0.0671	3.9000	0.0030	0.1920	0.0740	0.0270	0.2350
Bulgari	0.0451	0.7400	0.0720	1.3800	0.0850	0.4560	0.3650
Canada	0.0273	0.9000	0.0020	0.2570	0.0975	0.6450	1.9470
Chile	0.1279	1.7000	0.0110	1.1640	0.0801	0.2570	0.3790
Costa R	0.0789	2.6000	0.0240	0.9480	0.0794	0.3260	0.3570
Cyprus	0.0299	1.4000	0.0620	1.0420	0.0605	0.0780	0.4670
Czechos	0.0310	0.6200	0.1080	1.8210	0.0975	0.3980	0.6800
Denmark	0.0237	0.8300	0.1070	1.4340	0.0985	0.5700	1.0570
El Salv	0.0763	5.4000	0.1270	1.4970	0.0394	0.0890	0.2190
Finland	0.0210	1.6000	0.0130	1.5120	0.0985	0.5290	0.7940
France	0.0274	1.0140	0.0830	1.2880	0.0964	0.6670	0.9430
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
West Ge	0.0338	0.7980	0.2170	3.6310	0.0985	0.5280	0.9270
Yugosla	0.1000	1.6370	0.0730	1.2150	0.0770	0.5240	0.2650

Gunst, R. F., and Mason, R. L. (1980), Regression Analysis and Its Application: A Data-Oriented Approach, New York: Marcel Dekker, p. 358.



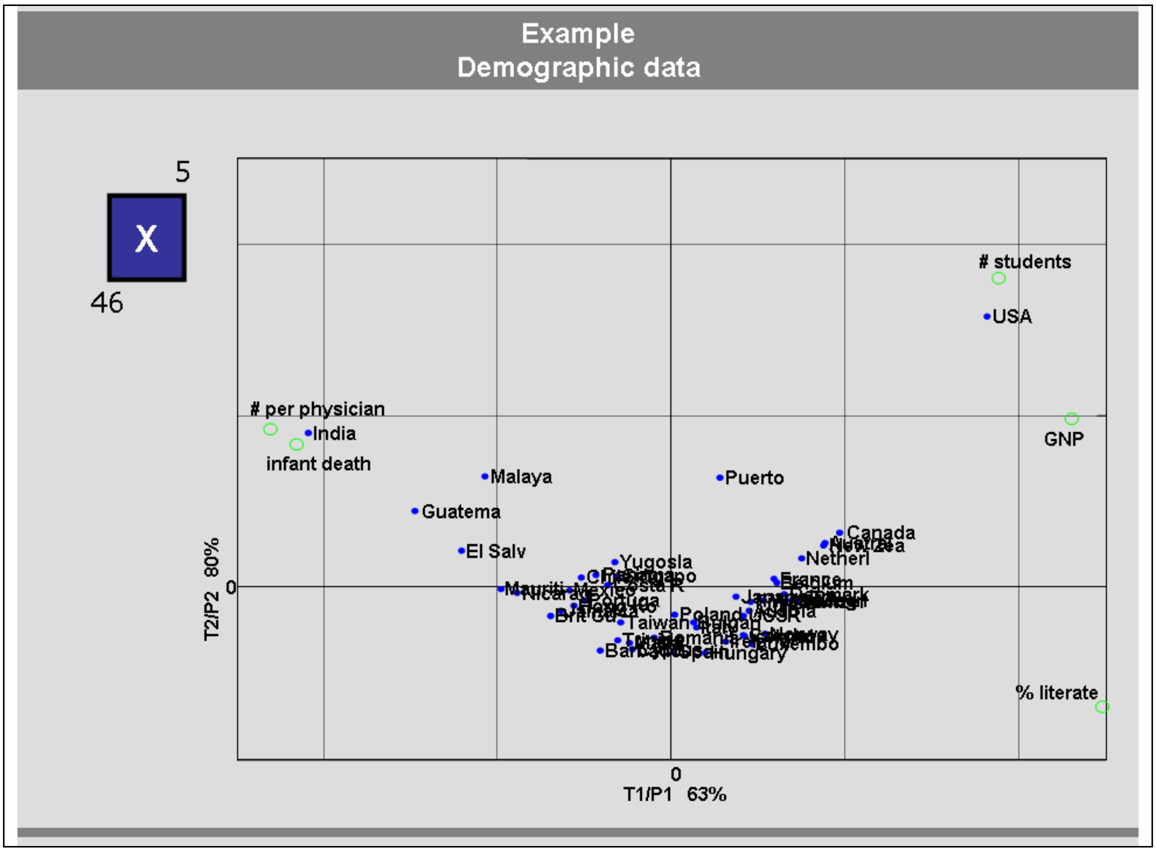
Example
Demographic data

A bi-plot of the two main directions/principal components (total variation explained 74%): size of the country and Poor versus Rich "Outliers" are objects that do not fit the general behavior due to errors, or in this case, due to abnormal behavior ("city-states"). Hong Kong and Singapore dominate the solution. The user often has to verify what is abnormal!



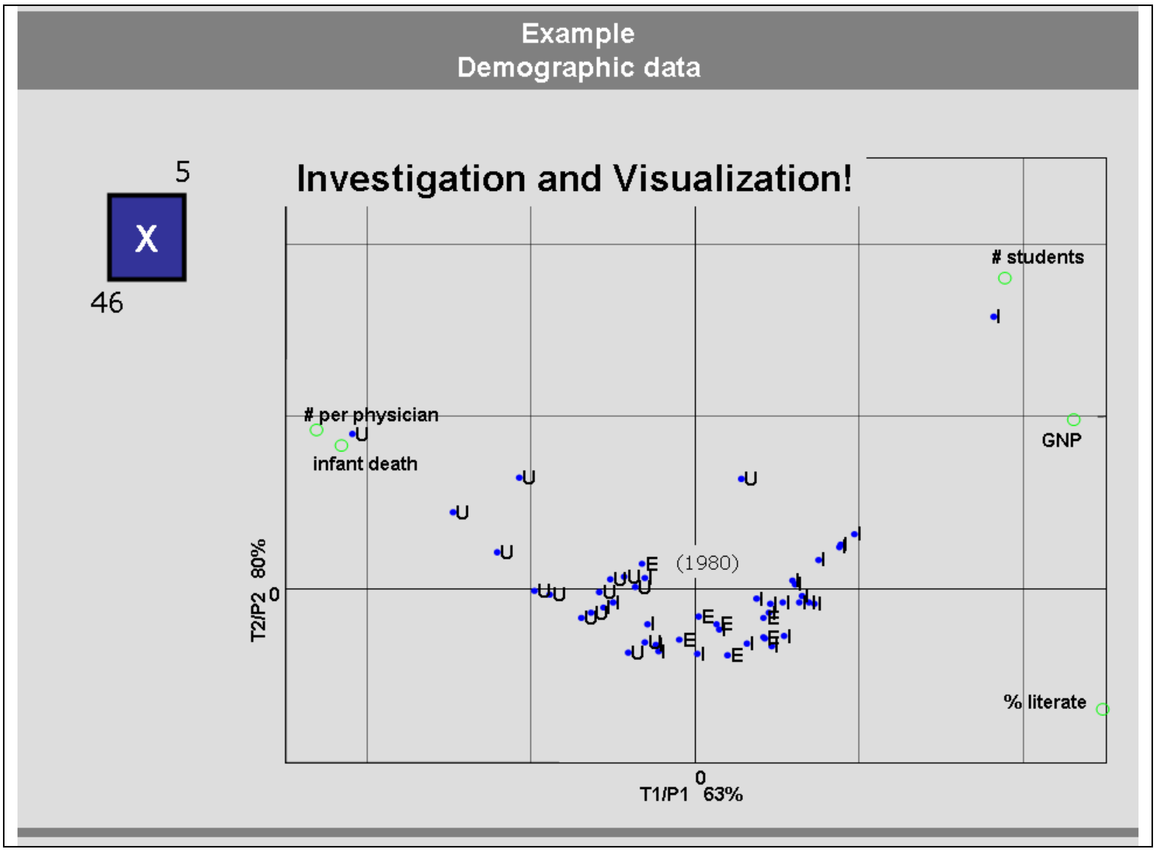
Example
Demographic data

By removing the two outliers we get a better view of the same split-up: small but rich or rather, crowded but rich.



Example
Demographic data

We can also remove the "crowdedness" direction by removing the appropriate variables from the data table. The new split: poor (health related) versus rich based on (American) education system.

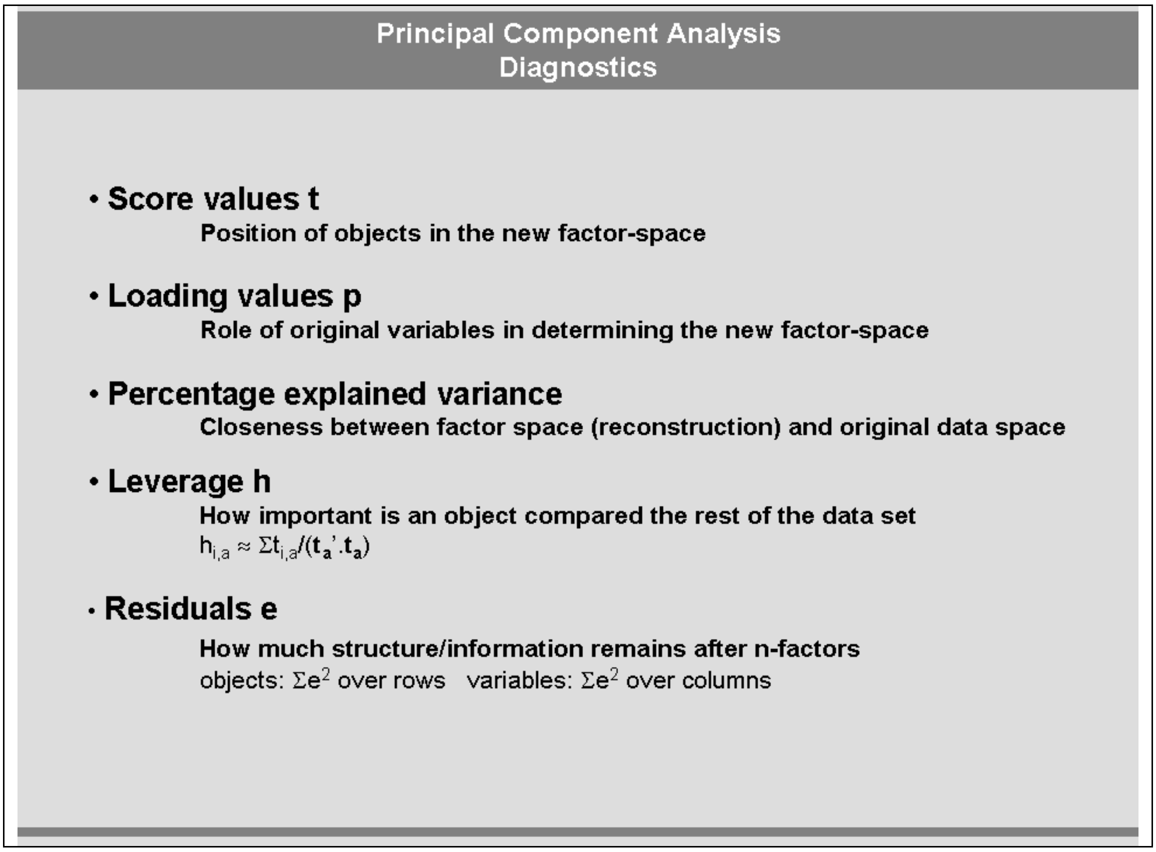


Example
Demographic data

Investigation and Visualization!

Important: the interpretation is for the user. The labeling/coding of objects is very important (visualization!). E.g. code in Industrial, Unindustrialized and Eastern Europe (remember: the data is from 1980).

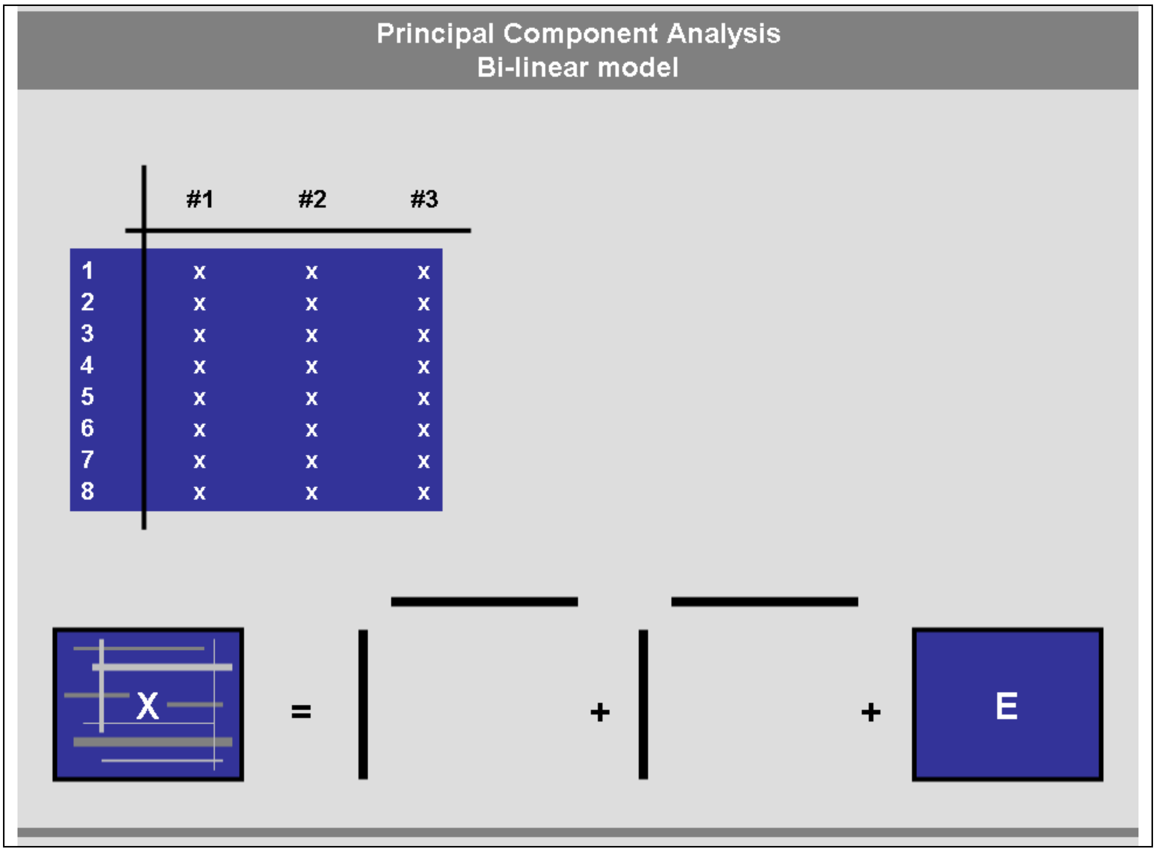
Another important issue: random reordering of rows or columns does not affect the results (only look at variances). As a user you have to keep track of the "secondary" structure (e.g. a time series, geographical or political ordering).



Principal Component Analysis
Diagnostics

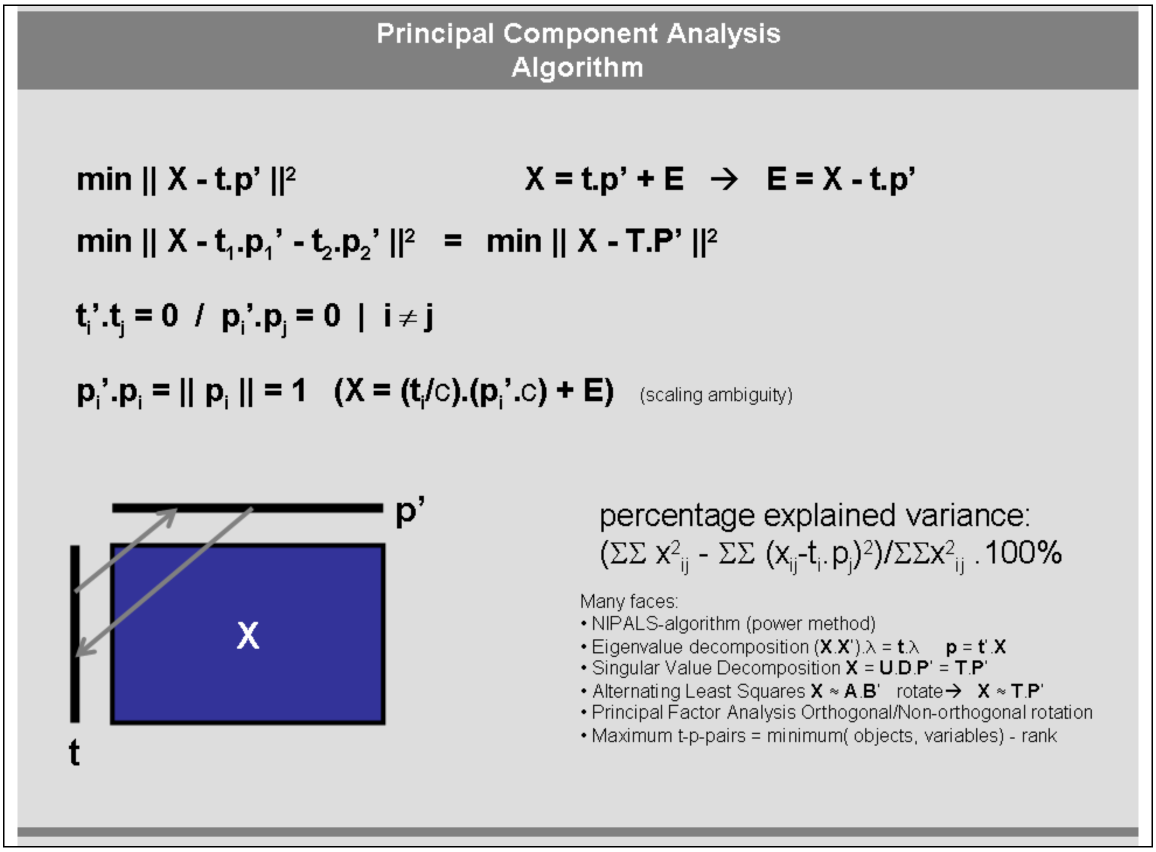
- **Score values t**
  Position of objects in the new factor-space
- **Loading values p**
  Role of original variables in determining the new factor-space
- **Percentage explained variance**
  Closeness between factor space (reconstruction) and original data space
- **Leverage h**
  How important is an object compared the rest of the data set
  $h_{i,a} \approx \Sigma t_{i,a}/(\mathbf{t_a}'.\mathbf{t_a})$
- **Residuals e**
  How much structure/information remains after n-factors
  objects: Σe^2 over rows   variables: Σe^2 over columns

Of the many diagnostics that become available from a PC analysis these five are the most used. Plotting them, keeping in mind the data its "secondary" structure mentioned before, forms the basis of exploratory data analysis.



Principal Component Analysis
Bi-linear model

	#1	#2	#3
1	x	x	x
2	x	x	x
3	x	x	x
4	x	x	x
5	x	x	x
6	x	x	x
7	x	x	x
8	x	x	x



Principal Component Analysis
Algorithm

$\min \| X - t.p' \|^2$ $\qquad X = t.p' + E \rightarrow E = X - t.p'$

$\min \| X - t_1.p_1' - t_2.p_2' \|^2 = \min \| X - T.P' \|^2$

$t_i'.t_j = 0 \; / \; p_i'.p_j = 0 \; | \; i \neq j$

$p_i'.p_i = \| p_i \| = 1 \quad (X = (t_i/c).(p_i'.c) + E)$ (scaling ambiguity)

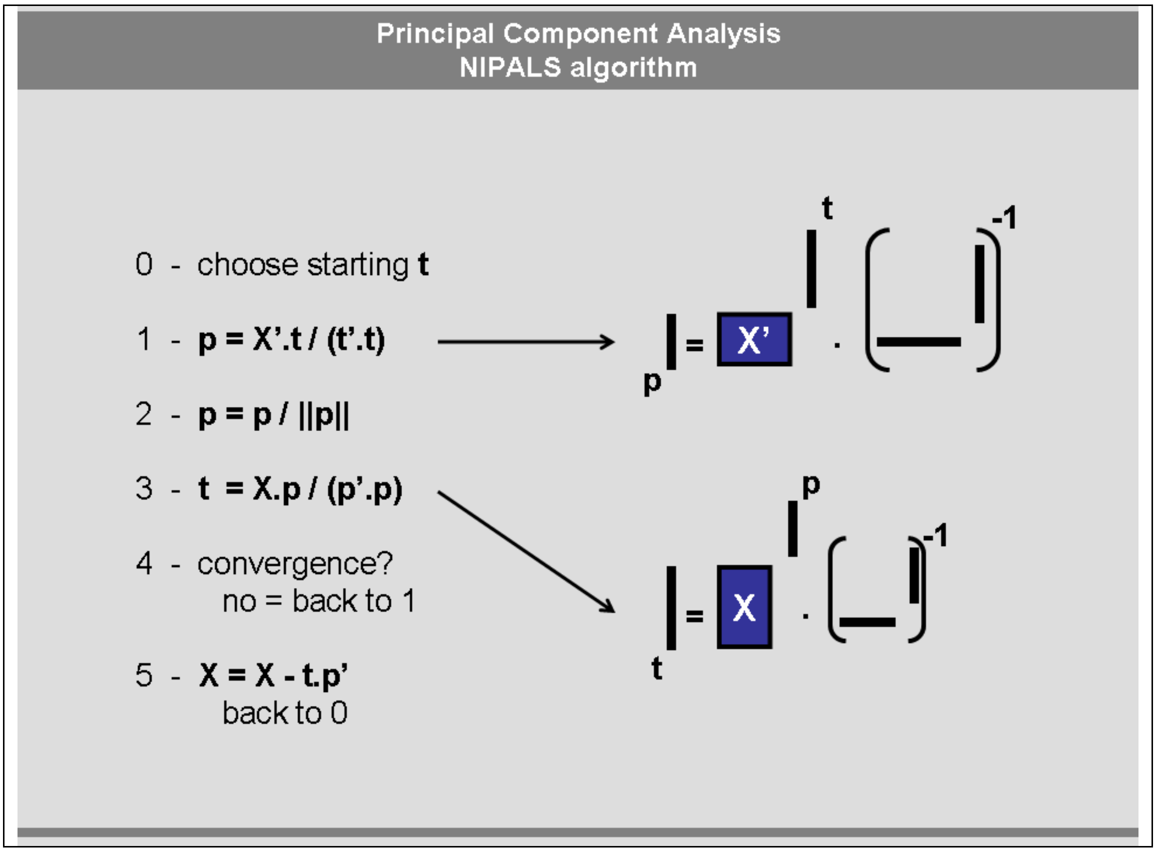
percentage explained variance:
$(\Sigma\Sigma\, x^2_{ij} - \Sigma\Sigma\, (x_{ij}-t_i.p_j)^2)/\Sigma\Sigma x^2_{ij}\,.100\%$

Many faces:
- NIPALS-algorithm (power method)
- Eigenvalue decomposition $(\mathbf{X}.\mathbf{X}').\lambda = \mathbf{t}.\lambda \quad \mathbf{p} = \mathbf{t}'.\mathbf{X}$
- Singular Value Decomposition $\mathbf{X} = \mathbf{U}.\mathbf{D}.\mathbf{P}' = \mathbf{T}.\mathbf{P}'$
- Alternating Least Squares $\mathbf{X} \approx \mathbf{A}.\mathbf{B}'$ rotate→ $\mathbf{X} \approx \mathbf{T}.\mathbf{P}'$
- Principal Factor Analysis Orthogonal/Non-orthogonal rotation
- Maximum t-p-pairs = minimum(objects, variables) - rank

In mathematical terminology the first factor is the least squares rank one approximation of **X**. The second factor is the least squares approximation of all that remains (the residuals **E**).
Furthermore we want the scores and loadings (impose one and the other one comes for free) of successive factors to be orthogonal (inner product zero).

All that remains is to remove the so-called scaling ambiguity: if we divide the score-values by a constant *c* and multiply the loadings by this same *c*, we get the same solution. This holds for any *c*, thus we have infinitely many solutions! To solve this inconvenience we specify the length of loading vector **p** to one, thereby fixing *c* (but any other choice would have done).

Some terminology:
Bilinear modeling: PCA determines two least squares models, one in variable space and one in object space (also see NIPALS algorithm below).
Soft modeling: well… you impose the least squares criterion, but that is all. Not physical, chemical, psychological, whatever-al functions are predefined to get a particular solution.



Principal Component Analysis
NIPALS algorithm

0 - choose starting **t**

1 - **p = X'.t / (t'.t)**

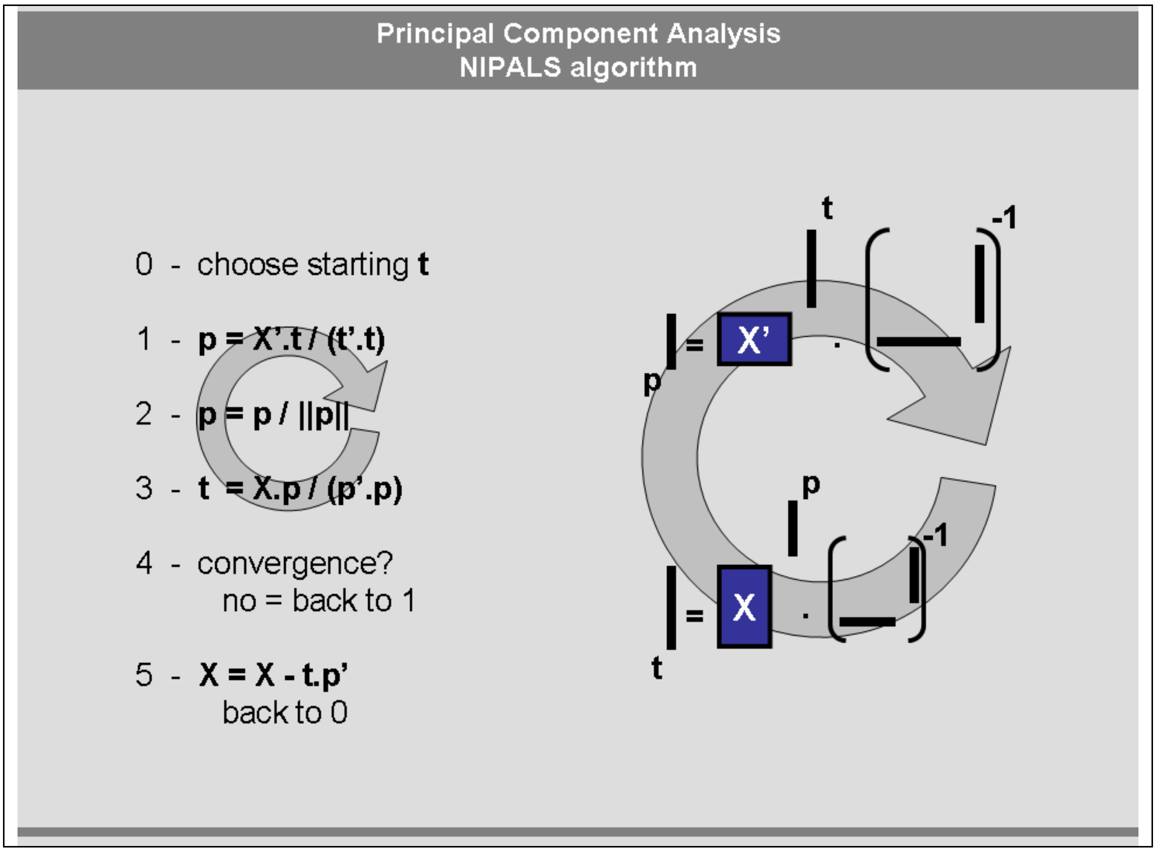
2 - **p = p / ||p||**

3 - **t = X.p / (p'.p)**

4 - convergence?
no = back to 1

5 - **X = X - t.p'**
back to 0

The NIPALS algorithm (Non-linear Iterative PArtial Least Squares) is given to illustrate how to determine PCA factors. It consists of two least squares steps on the data table **X**: one to compute the loadings, one to compute the scores. These steps are repeated in sequence until no more changes in **t** and/or **p** are observed. In the middle we have to norm the loading **p** to get ride of the scaling ambiguity.



Principal Component Analysis
NIPALS algorithm

0 - choose starting **t**

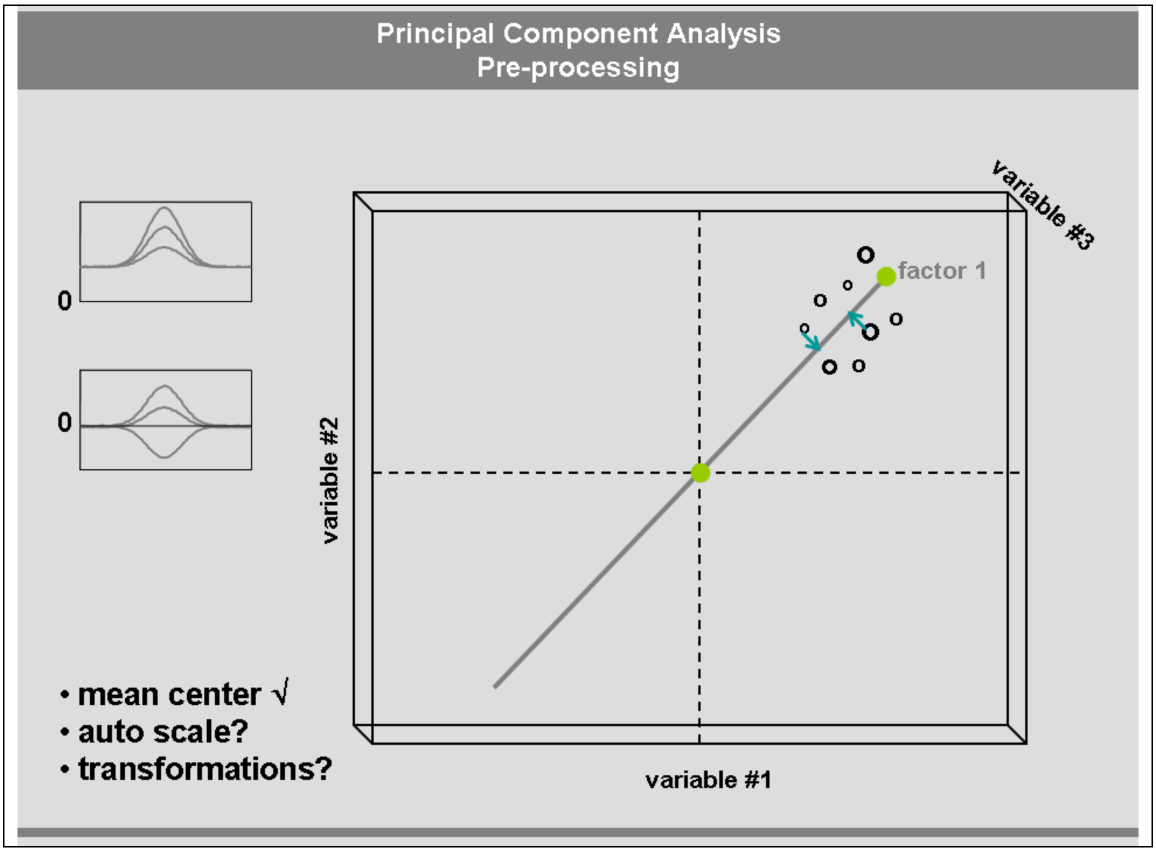
1 - **p = X'.t / (t'.t)**

2 - **p = p / ||p||**

3 - **t = X.p / (p'.p)**

4 - convergence?
    no = back to 1

5 - **X = X - t.p'**
    back to 0



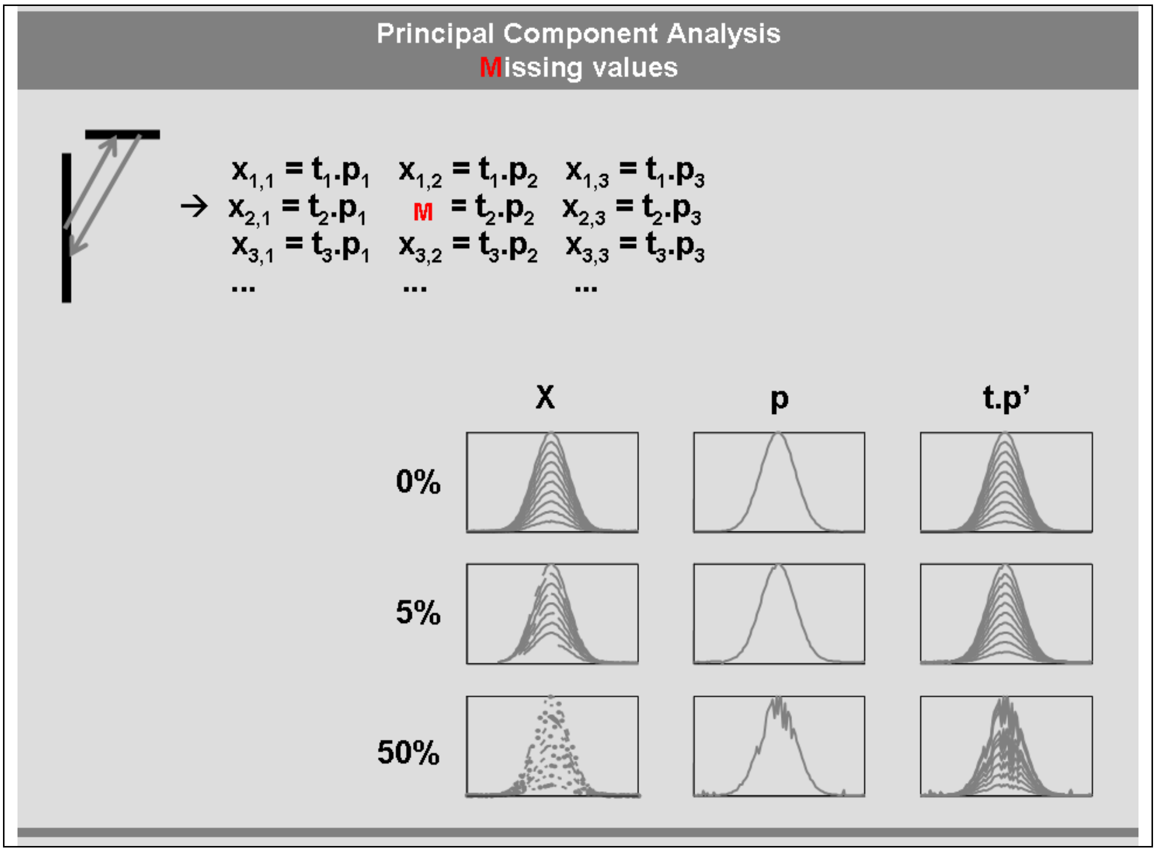
Principal Component Analysis
Pre-processing

- mean center √
- auto scale?
- transformations?

As we saw before: the first factor finds all the variation that the objects have in common, and goes through the origin. If there is an offset ("baseline") - like in the picture – this will often be the main variance. To get a more parsimonious model the data is almost always mean centered before analysis by subtracting the mean of a column from the entries in that column.

Other scaling-operations are sometimes used, where auto-scaling is the most famous one. In auto-scaling each variable get the same importance by setting the mean to zero (subtract the column average) and variance to one (divide by the column standard deviation).

Alternative transformations can lead to completely other model outcomes. The correct choice for transformations has to come from the problem domain (e.g. knowledge on the types of measurements preformed).

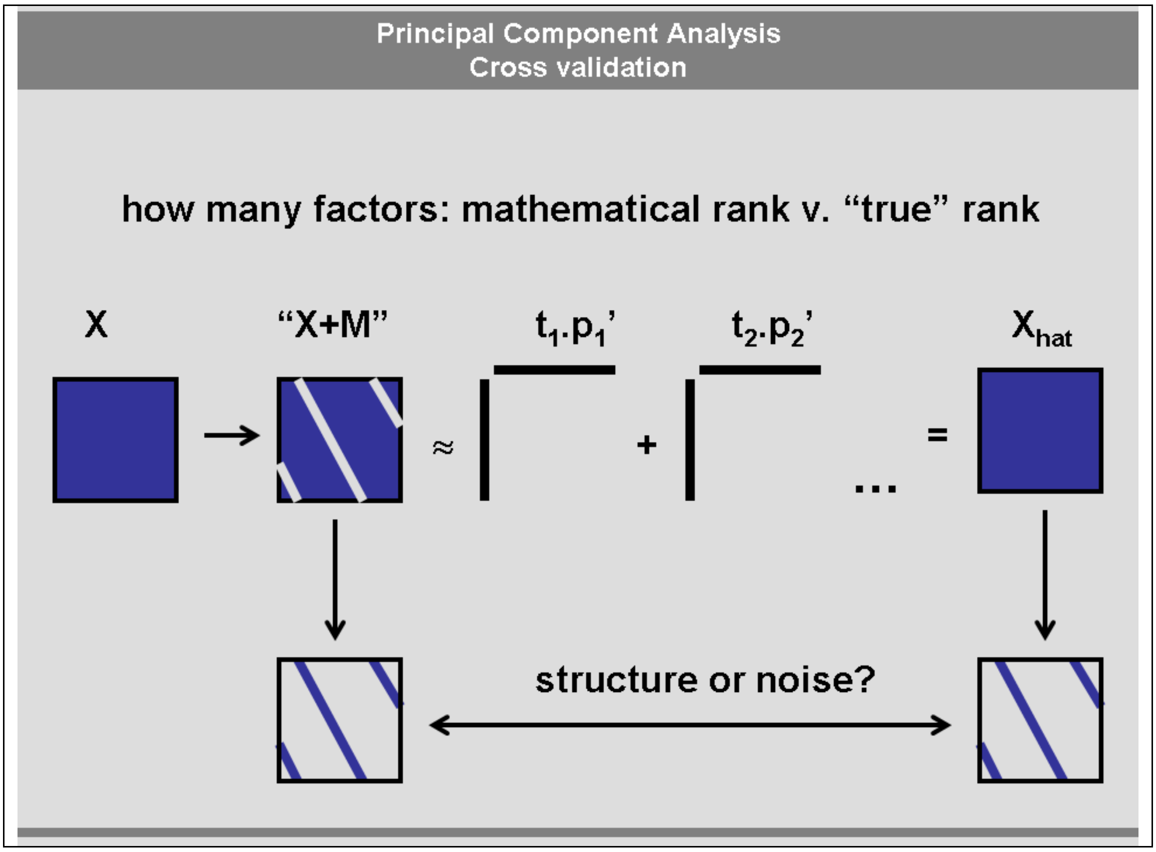


Principal Component Analysis
Missing values

$$
\begin{array}{llll}
 & x_{1,1} = t_1.p_1 & x_{1,2} = t_1.p_2 & x_{1,3} = t_1.p_3 \\
\rightarrow & x_{2,1} = t_2.p_1 & M = t_2.p_2 & x_{2,3} = t_2.p_3 \\
 & x_{3,1} = t_3.p_1 & x_{3,2} = t_3.p_2 & x_{3,3} = t_3.p_3 \\
 & \dots & \dots & \dots
\end{array}
$$

PCA can handle a modest amount of missing data. The model parameters (**t** and **p**) are estimated from a least squares fit by solving many simple equations. If there is a missing value in the table we just estimate the parameters from the equation that are complete. From the estimates we can than reconstruct the missing entry.

Whether this trick works depends on the amount of missing data and the structure in the matrix.

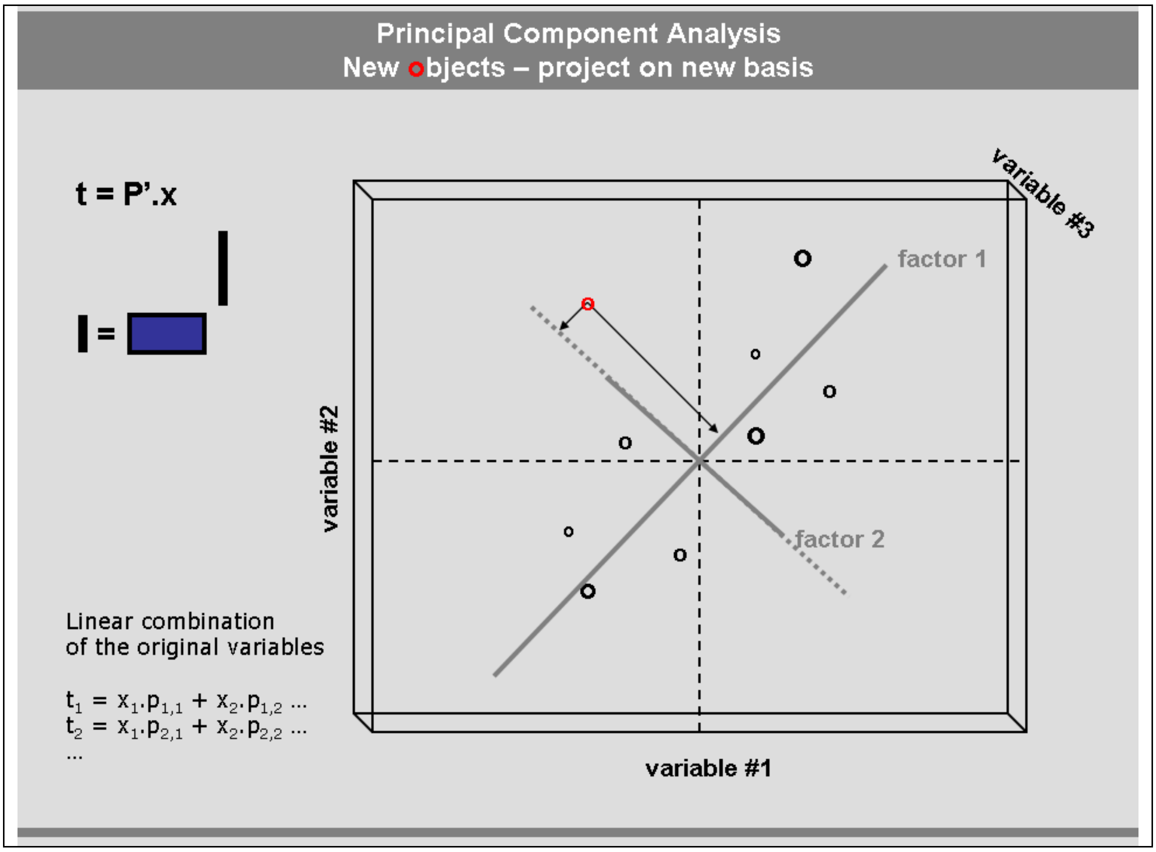


Principal Component Analysis
Cross validation

how many factors: mathematical rank v. “true” rank

The important question in PCA-modeling is how many factors are relevant. The maximum mathematical rank of a matrix is the minimum of the number of rows and columns. But often the true rank of a system is much lower because the variables form (near) linear combinations (co-linearity). An important indicator is the percentage variance explained. This will always grow when increasing the model complexity. But if the increase from one factor to the next is small it is most likely noise you are fitting.

Another way to estimate the model complexity is cross validation. In PCA this is achieved by setting (non-overlapping) diagonals to missing. Next, the data table is reconstructed for an increasing number of factors, and the reconstructed diagonals are compared to the original entries. This procedure is repeated for all the diagonals. When the reconstruction error drops from one factor to the next, the extra component models real structure. When the reconstruction error flattens out, the factor describes primarily noise.



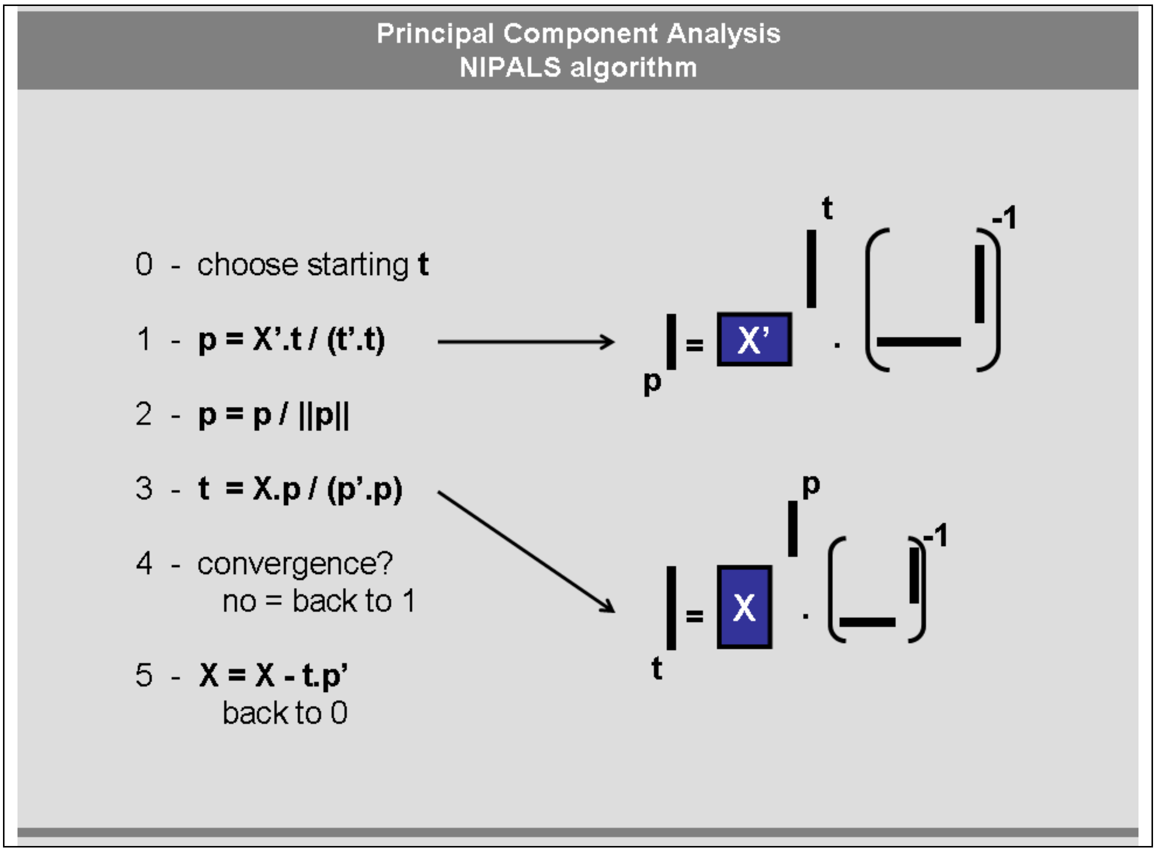
Principal Component Analysis
New objects – project on new basis

$t = P'.x$

Linear combination of the original variables

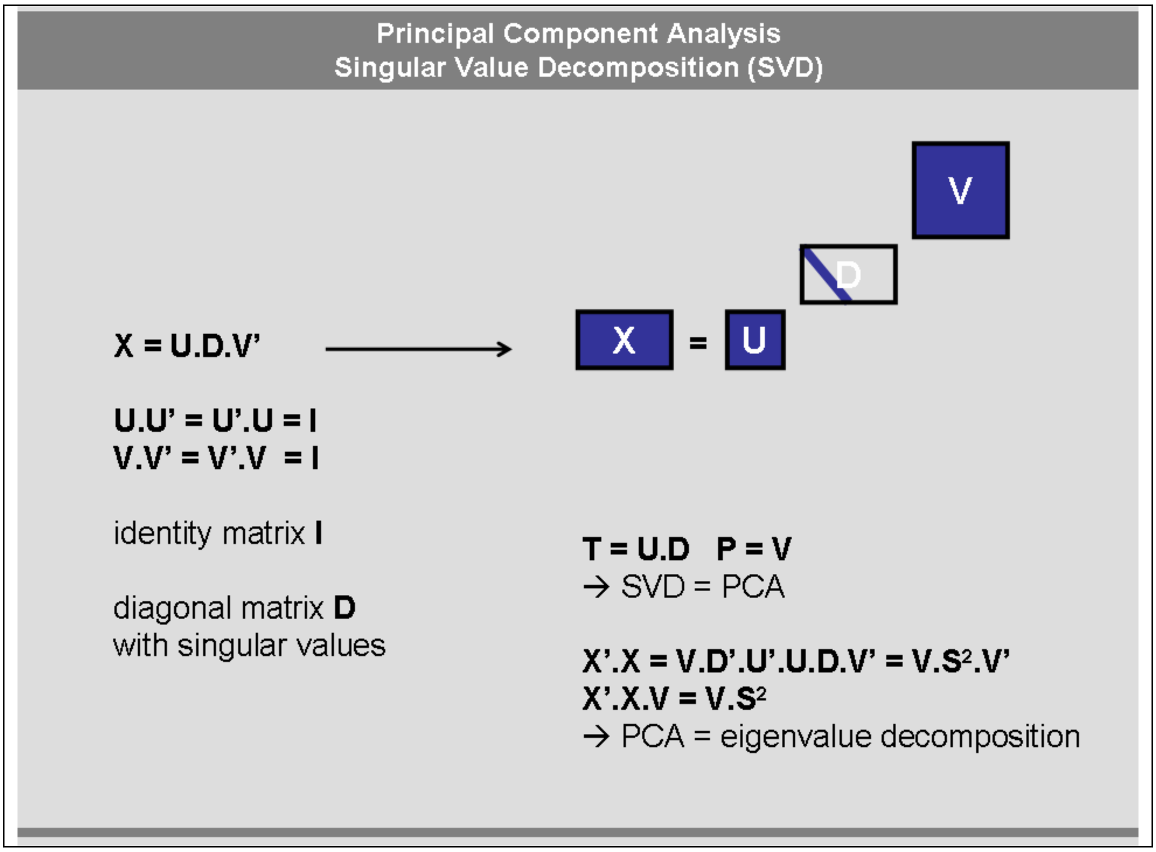
$t_1 = x_1.p_{1,1} + x_2.p_{1,2}$ ...
$t_2 = x_1.p_{2,1} + x_2.p_{2,2}$ ...
...

Ones you have a PCA model (= the correct number of loadings **p**) you can project new objects (= a series of new measurements **x**) and get a new set of "principal" variables (= the score values **t**).
In short: PCA defines coordinates/loadings **p** to filter the noise from the interesting directions and expresses the objects/samples in a new set of variables/scores **t**.



Principal Component Analysis
NIPALS algorithm

```
0 - choose starting t
1 - p = X'.t / (t'.t)
2 - p = p / ||p||
3 - t = X.p / (p'.p)
4 - convergence?
      no = back to 1
5 - X = X - t.p'
      back to 0
```



Principal Component Analysis
Singular Value Decomposition (SVD)

$\mathbf{X = U.D.V'}$ →

$\mathbf{U.U' = U'.U = I}$
$\mathbf{V.V' = V'.V = I}$

identity matrix **I**

diagonal matrix **D**
with singular values

$\mathbf{T = U.D \quad P = V}$
→ SVD = PCA

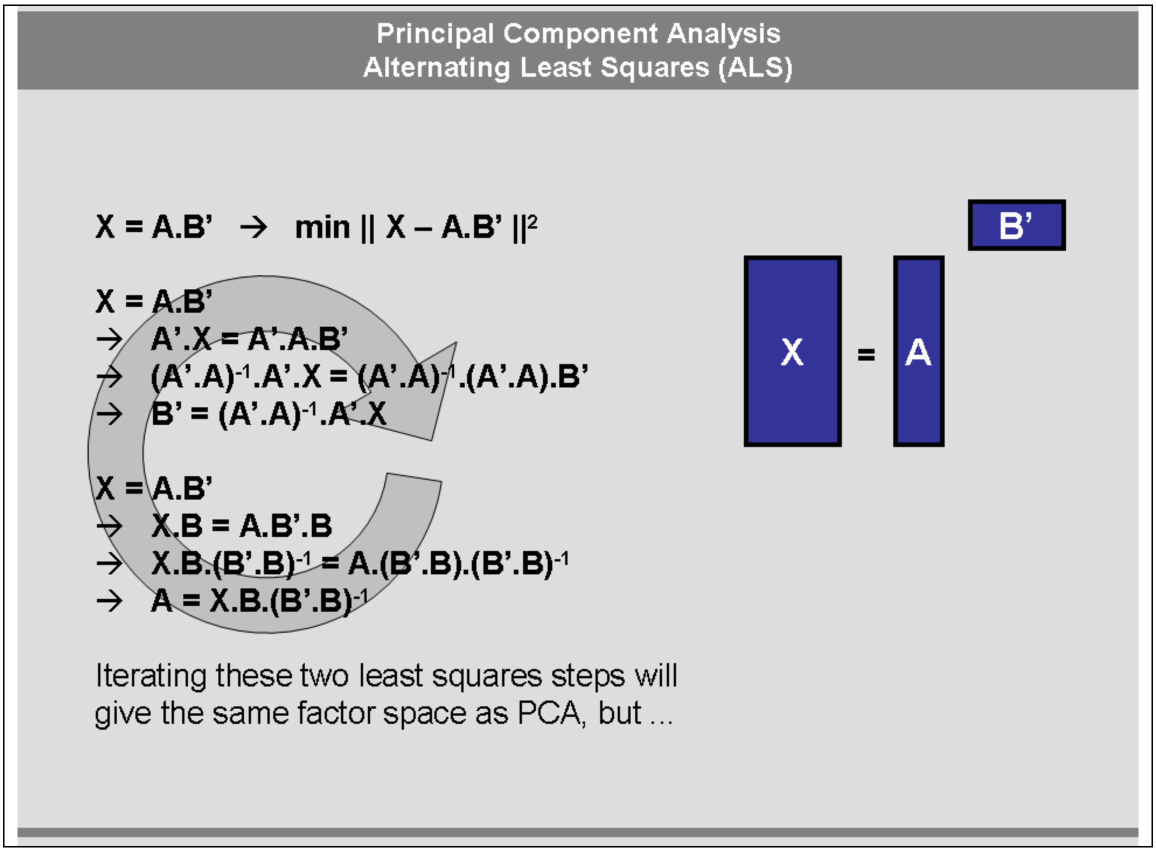
$\mathbf{X'.X = V.D'.U'.U.D.V' = V.S^2.V'}$
$\mathbf{X'.X.V = V.S^2}$
→ PCA = eigenvalue decomposition

Another interesting algorithm closely related to PCA is the Singular Value Decomposition (SVD). This method (frequently used inside PCA-software) decomposes the matrix **X** in three parts: the left singular vectors **U** (the object space), the right singular vectors **V** (the variable space) and the diagonal matrix **D** that has the singular values on it diagonal, sorted in decreasing order.

The diagonal matrix immediately shows the structure of the matrix: a few large singular values in the upper part represent the main phenomena, while the (near) zero diagonal elements below are the noise dimensions.

The left and right singular value matrices are both column-wise orthonormal, which has a nice property for matrix inversion we will use later on ($\mathbf{U}^{-1} = \mathbf{U}'$).

The SVD-decomposition can easily be turned into the PCA-solution. For two-dimensional data (so-called first order data) the two representations are equivalent. From SVD we also learn that PCA is an eigenvalue decomposition of the association matrix **X'.X** (or **X.X'**).



Principal Component Analysis
Alternating Least Squares (ALS)

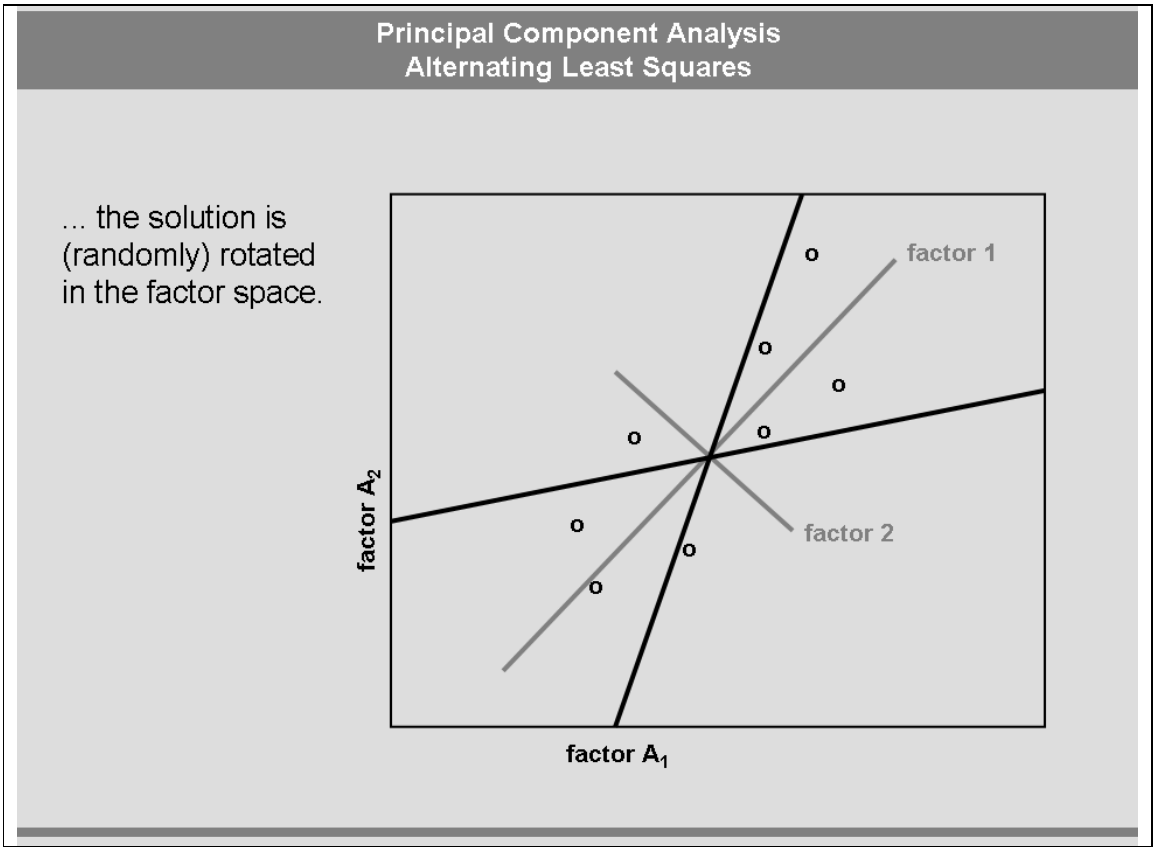
$X = A.B' \rightarrow \min \| X - A.B' \|^2$

$X = A.B'$
$\rightarrow A'.X = A'.A.B'$
$\rightarrow (A'.A)^{-1}.A'.X = (A'.A)^{-1}.(A'.A).B'$
$\rightarrow B' = (A'.A)^{-1}.A'.X$

$X = A.B'$
$\rightarrow X.B = A.B'.B$
$\rightarrow X.B.(B'.B)^{-1} = A.(B'.B).(B'.B)^{-1}$
$\rightarrow A = X.B.(B'.B)^{-1}$

Iterating these two least squares steps will give the same factor space as PCA, but …

So why not go between NIPALS (one factor at the time) and SVD (all factors at ones)? That is the Alternating Least Squares (ALS) algorithm, computing the bilinear models for the number of factors *N* you think are interesting. You start by picking a random **A** (the scores matrix) or **B** (the loading matrix), and iterate until you find the solution. The solution will span the same *N*-dimensional space as the first *N*-PCA factors (the least squares solution), but …

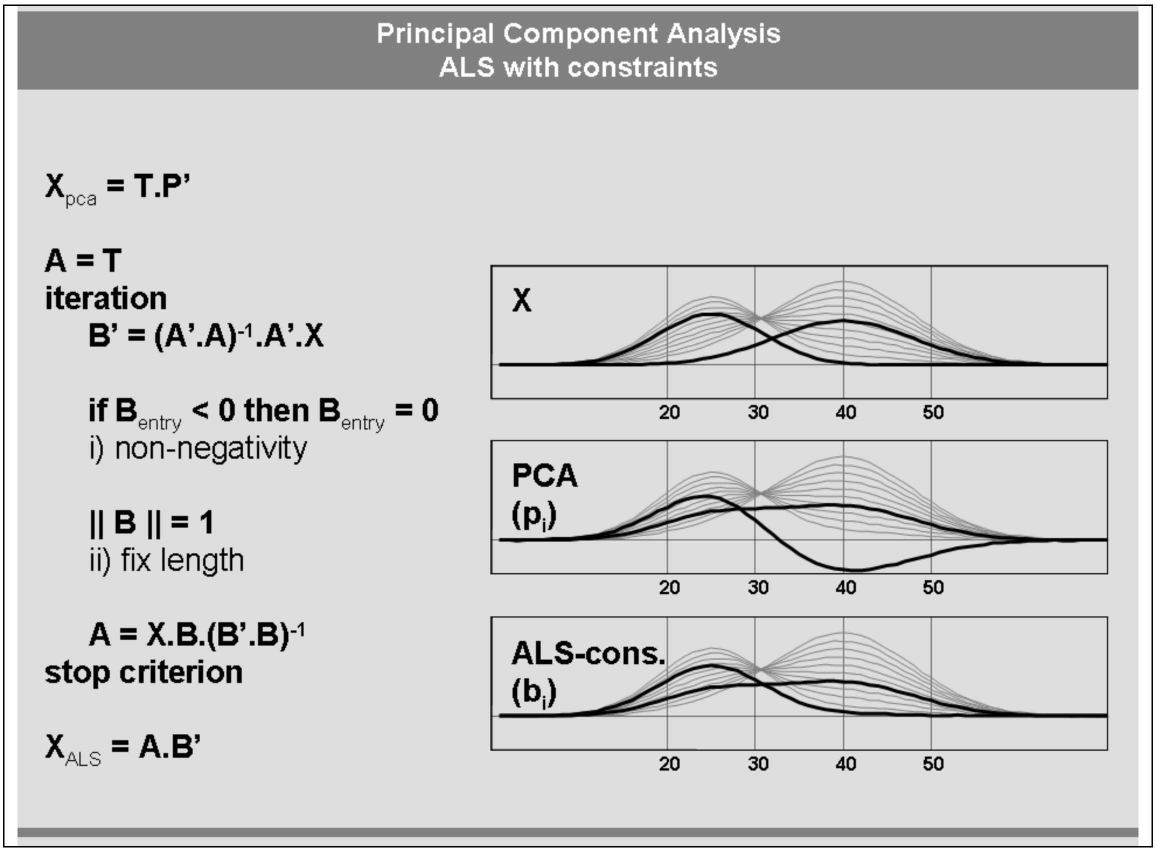


Principal Component Analysis
Alternating Least Squares

... the solution is (randomly) rotated in the factor space.

… in this ALS we did not impose the additional criteria from PCA (orthogonal score/loading vectors, correct for scaling ambiguity), so our solution is in the same space but rotated.

This is however no problem. You are free to select a new base within the new factor space and any pair of axis will span the two-dimensional plane (as long as they are not the same!).



Principal Component Analysis
ALS with constraints

```
$X_{pca} = T.P'$

$A = T$
iteration
    $B' = (A'.A)^{-1}.A'.X$

    if $B_{entry} < 0$ then $B_{entry} = 0$
    i) non-negativity

    $|| B || = 1$
    ii) fix length

    $A = X.B.(B'.B)^{-1}$
stop criterion

$X_{ALS} = A.B'$
```

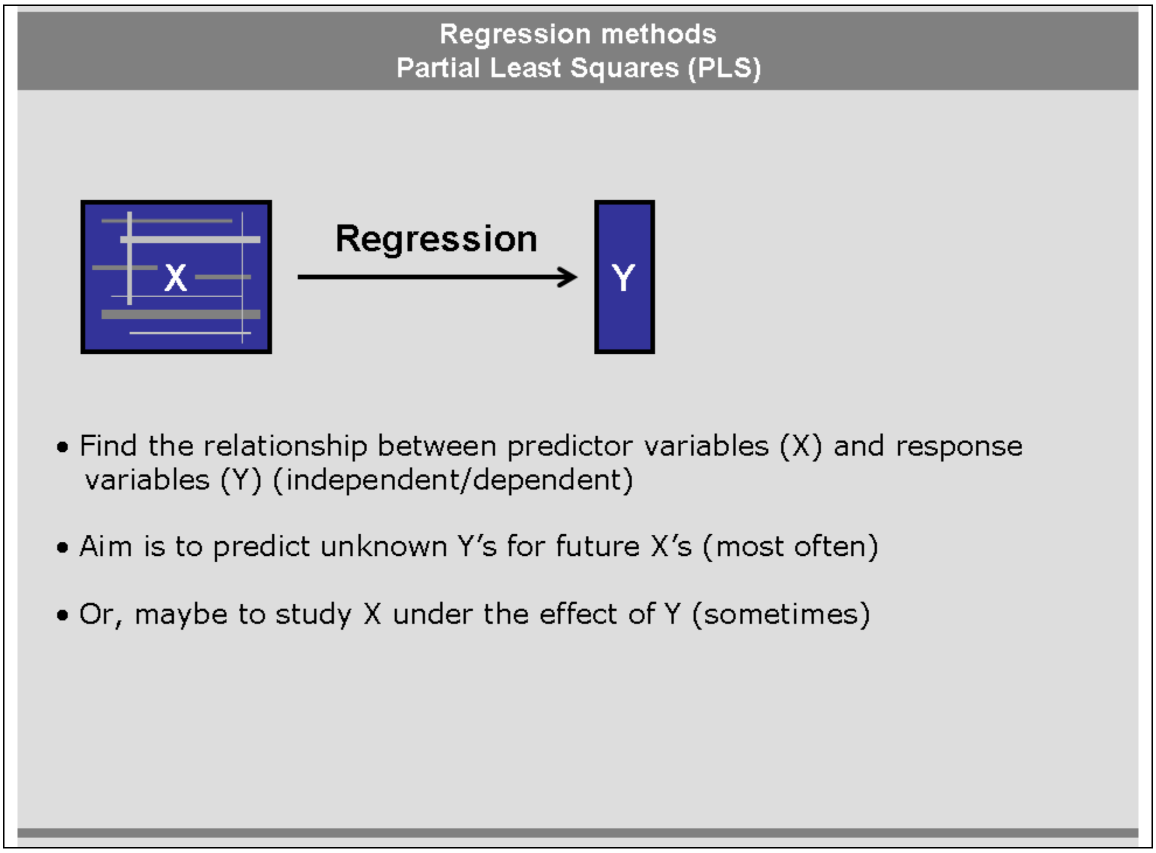
The simple ALS algorithm opens the possibility to introduce (additional) constraints in a factor model. In this example the object-scores **A** are initialized as PCA scores **T**. Then an iterative ALS algorithm is used to find a constraint factor model. The two constraints are: all entries in the variable-loadings larger than zero (non-negativity), and fix the length of the loading vectors to one (eliminate scaling ambiguity). Constrains can assist in the interpretation of assumed physical phenomena in a data analysis (e.g. spectra in curve resolution problem).

By imposing the appropriate constrains (orthogonal solution vectors, sorted by size), the relation between ALS, NIPALS and SVD becomes obvious.

Notice that the two subspaces in this example (PCA versus ALS + constrains) are not the same. We chose to alter the model (and sacrifices fit) in favor of a non-negative loading vectors **B**.

Partial Least Squares regression

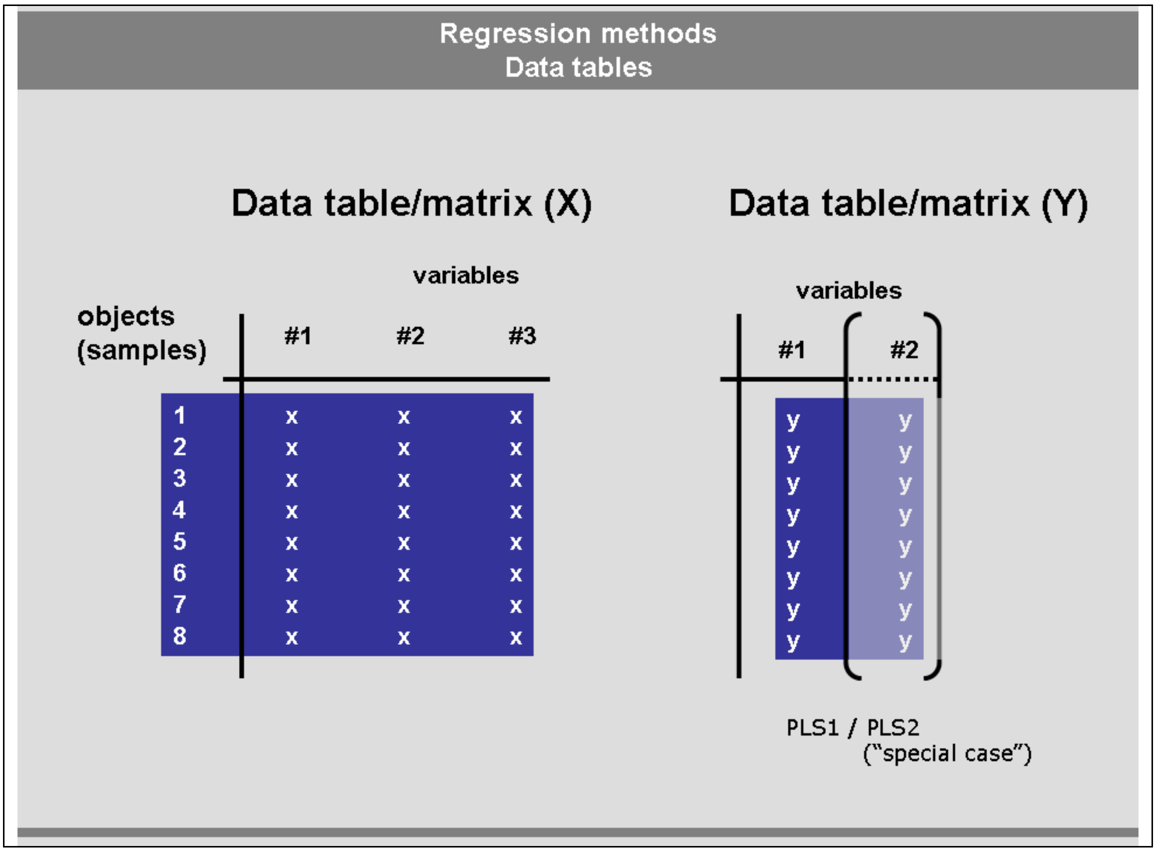
The next part of this document contains the transparencies used in the lectures on PLS, accompanied by some "telegram-style" notes on the main issues.



Regression methods
Partial Least Squares (PLS)

X
Regression
Y

- Find the relationship between predictor variables (X) and response variables (Y) (independent/dependent)
- Aim is to predict unknown Y's for future X's (most often)
- Or, maybe to study X under the effect of Y (sometimes)



Regression methods
Data tables

Data table/matrix (X)

variables

objects (samples)	#1	#2	#3
1	x	x	x
2	x	x	x
3	x	x	x
4	x	x	x
5	x	x	x
6	x	x	x
7	x	x	x
8	x	x	x

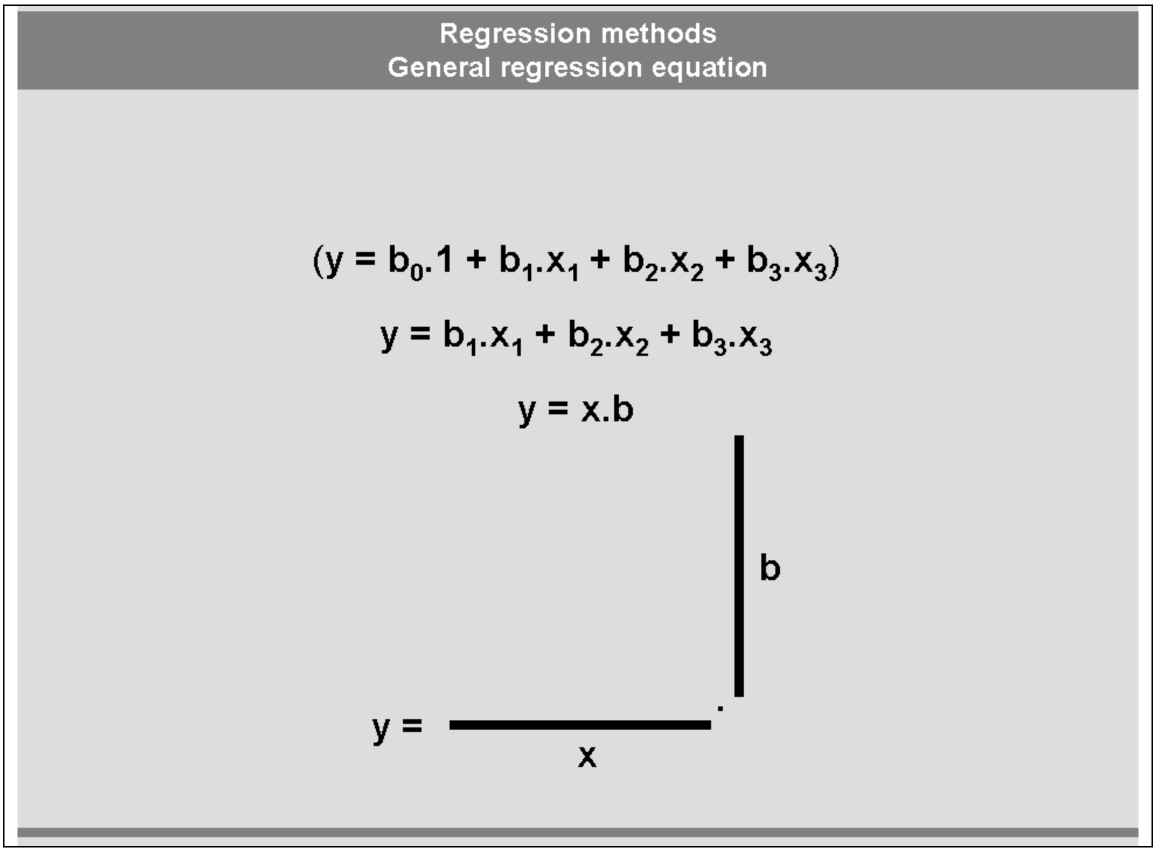
Data table/matrix (Y)

variables

#1	#2
y	y
y	y
y	y
y	y
y	y
y	y
y	y
y	y

PLS1 / PLS2
("special case")

Partial Least Squares (PLS) can be used to build regression models between a predictor block **X** and a response vector **y**. PLS has the possibility to form a regression model with a table **Y** (so-called PLS2), by determining a bilinear model for the response block and computing with the pseudo variables/scores **u**. It is however not always favorable to do so, and since PLS is rather unique in this capability we will focus on the one response variable situation.



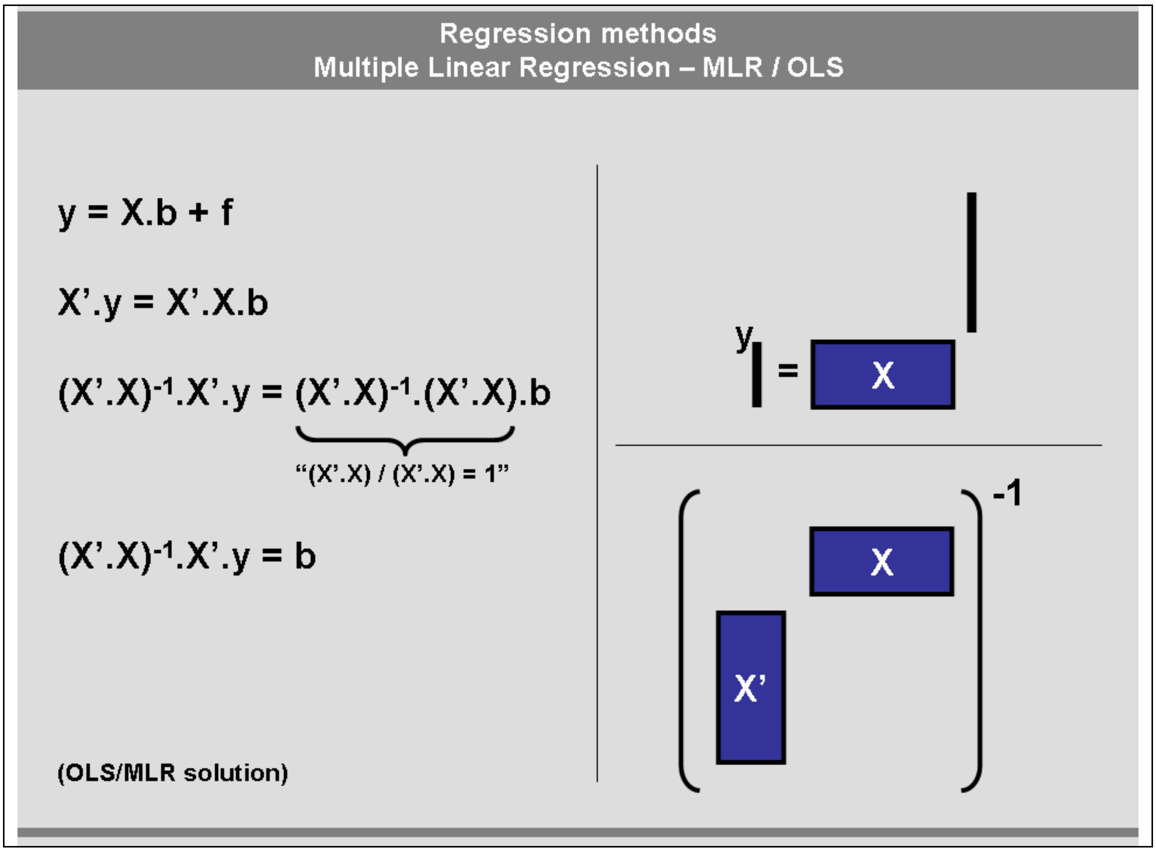
Regression methods
General regression equation

$$(y = b_0.1 + b_1.x_1 + b_2.x_2 + b_3.x_3)$$

$$y = b_1.x_1 + b_2.x_2 + b_3.x_3$$

$$y = x.b$$

In its most simple form we are solving for **b**, a vector of regression coefficients, and one coefficient for each variable in **X**. If **y** is not mean centered an offset b_0 can included.



Regression methods
Multiple Linear Regression – MLR / OLS

$$y = X.b + f$$

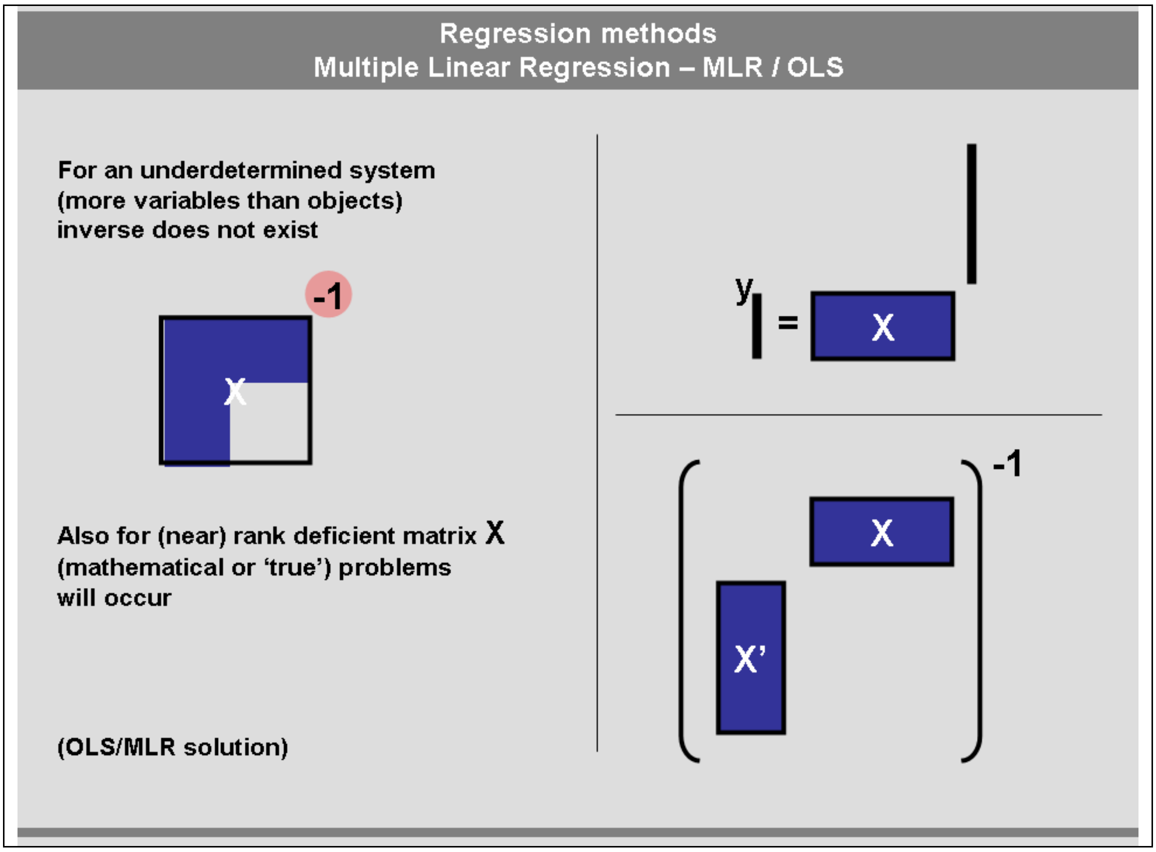
$$X'.y = X'.X.b$$

$$(X'.X)^{-1}.X'.y = \underbrace{(X'.X)^{-1}.(X'.X)}_{\text{“(X'.X) / (X'.X) = 1”}}.b$$

$$(X'.X)^{-1}.X'.y = b$$

(OLS/MLR solution)

In a regression problem we are looking for a linear combination of the variables in **x** that can predict the response **y**. This linear combination is defined by the regression vector **b**, found from a least squares solution. This Multiple Linear Regression (MLR) solution can have some serious problems: is the predictor matrix **X** is (near) rank deficient the inverse is very hard to compute (unstable solution), and if the system is undetermined (more variables than objects) the solution is even undefined.



Regression methods
Multiple Linear Regression – MLR / OLS

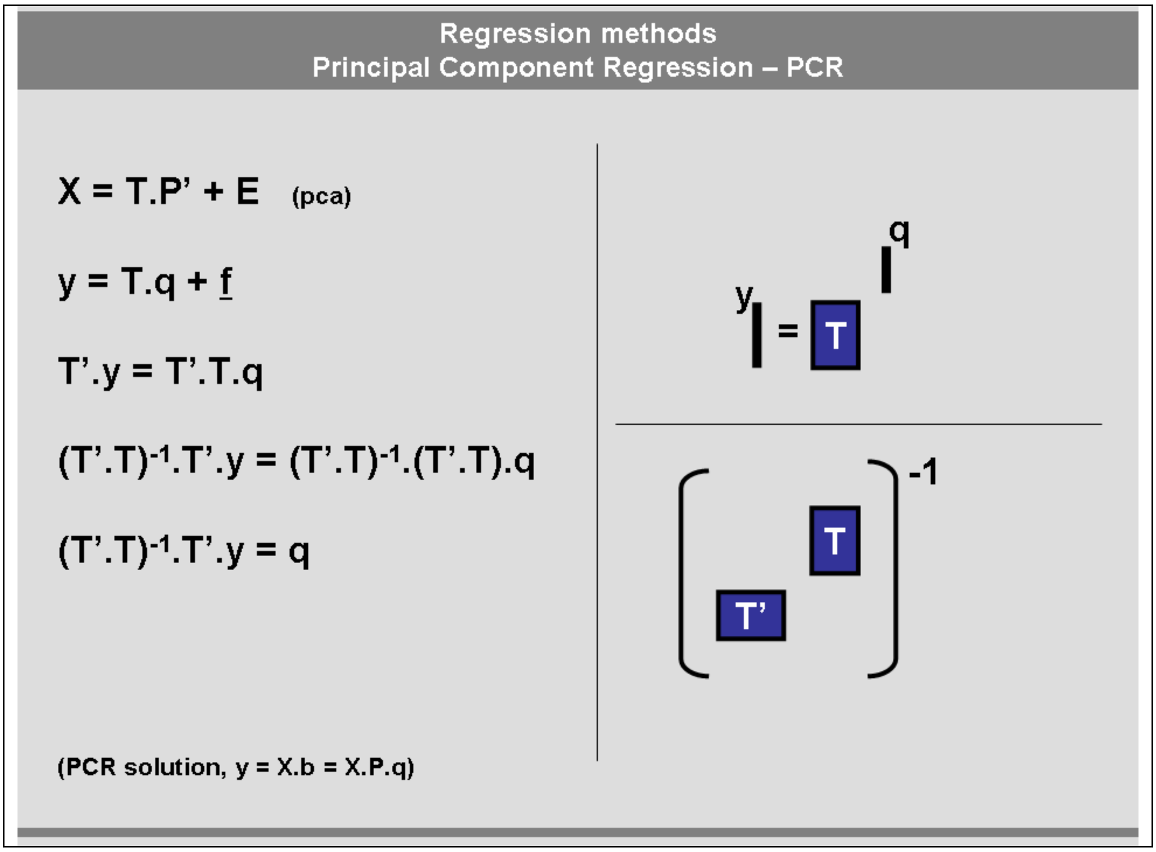
For an underdetermined system (more variables than objects) inverse does not exist

Also for (near) rank deficient matrix X (mathematical or ‘true’) problems will occur

(OLS/MLR solution)

If the matrix **X** (“the system”) is underdetermined (more variables/measurements than objects/samples) the inverse does not exist. If two or more variables are collinear the inverse of a over-determined (more or equal number of observations than variables) might also not exist (singular or rank deficient system).

In practice many systems are “near rank deficient” – due to noise, rounding errors, etc. – but computing inverse will be very unreliable.



Regression methods
Principal Component Regression – PCR

$X = T.P' + E$ (pca)

$y = T.q + \underline{f}$

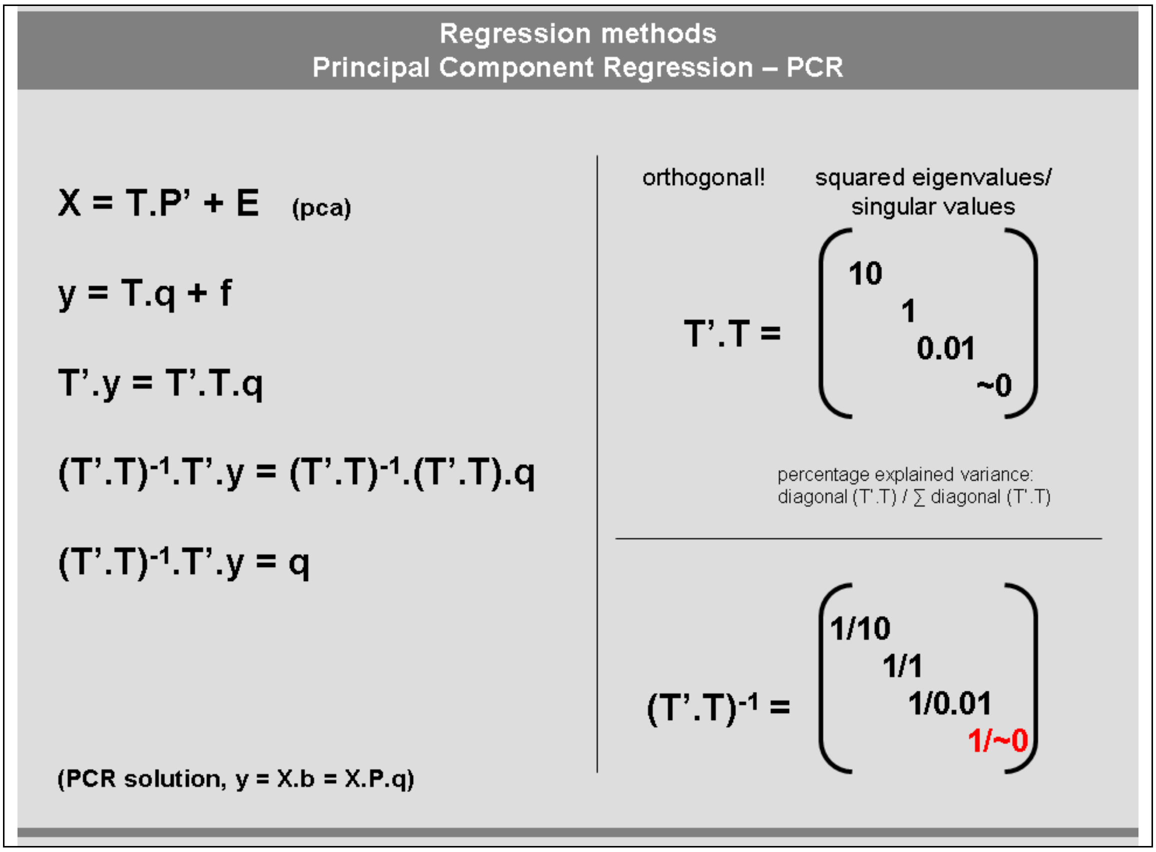
$T'.y = T'.T.q$

$(T'.T)^{-1}.T'.y = (T'.T)^{-1}.(T'.T).q$

$(T'.T)^{-1}.T'.y = q$

(PCR solution, $y = X.b = X.P.q$)

To overcome the problems in MLR we need to regularize the regression solution. One obvious candidate is to substitute the original variables by a smaller set of more relevant variables: the scores from a PCA analysis (the Principal Component Regression model). The score vectors are by definition orthogonal, leading to a well-conditioned inverse, and PCA can never extract more factors then the maximum rank of **X**, so the inverse is never underdetermined.



Regression methods
Principal Component Regression – PCR

$X = T.P' + E$ (pca)

$y = T.q + f$

$T'.y = T'.T.q$

$(T'.T)^{-1}.T'.y = (T'.T)^{-1}.(T'.T).q$

$(T'.T)^{-1}.T'.y = q$

(PCR solution, y = X.b = X.P.q)

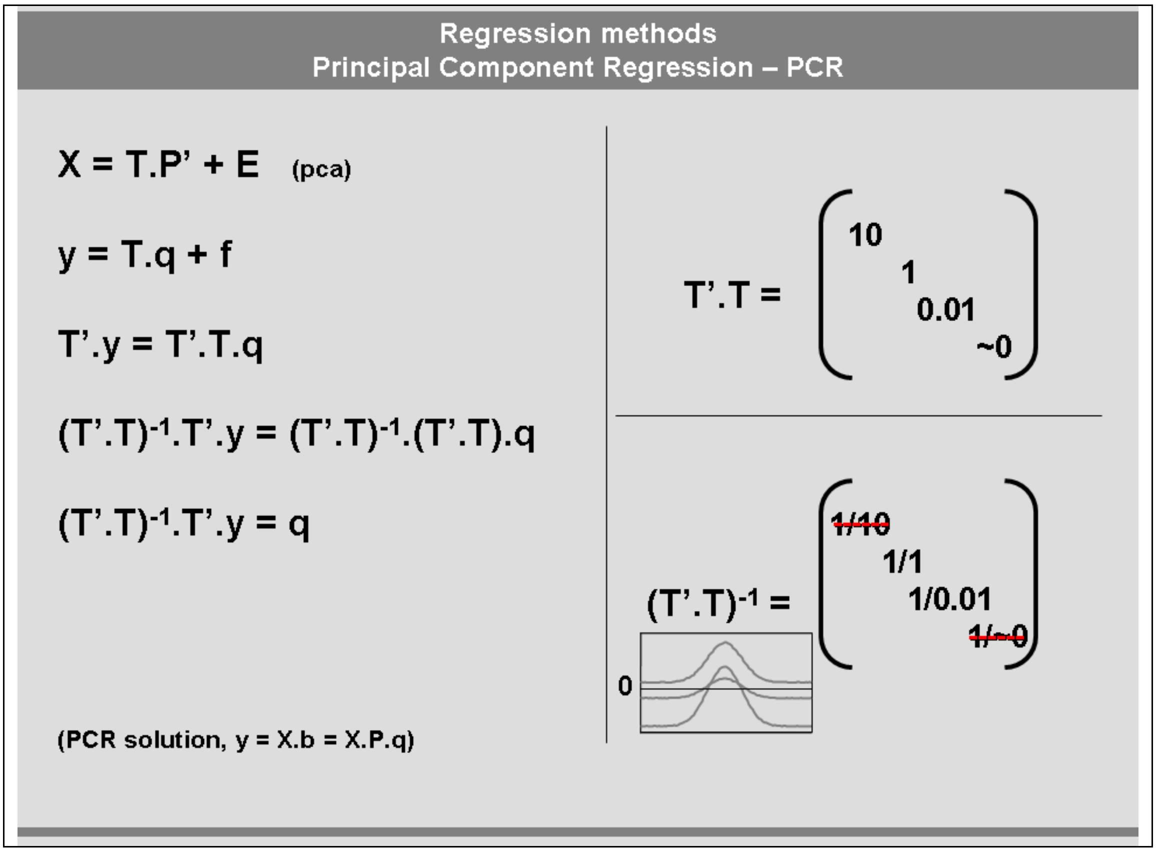
orthogonal! squared eigenvalues/ singular values

$$T'.T = \begin{pmatrix} 10 & & & \\ & 1 & & \\ & & 0.01 & \\ & & & \sim 0 \end{pmatrix}$$

percentage explained variance:
diagonal (T'.T) / ∑ diagonal (T'.T)

$$(T'.T)^{-1} = \begin{pmatrix} 1/10 & & & \\ & 1/1 & & \\ & & 1/0.01 & \\ & & & 1/\sim 0 \end{pmatrix}$$

The concept of a regularized inverse is best illustrated using the SVD to compute a PCR solution. We end up with computing the inverse of the diagonal matrix **D**, which is easy because the inverse of a diagonal matrix is just the inverse of the elements on the diagonal.
Remember that **D** has the singular values (squared eigenvalues) in decreasing order as its entries. For the interesting part (upper corner) we have large singular values, and the inverse can be determined relatively easy. For the small singular values (lower corner; the noise) the inverse can "blow up": you end up with enormous numbers that completely dominate the inverse and thus the regression solution. If the last singular values are zero (something that seldom happens for real data) the inverse is undefined altogether.



Regression methods
Principal Component Regression – PCR

$X = T.P' + E$ (pca)

$y = T.q + f$

$T'.y = T'.T.q$

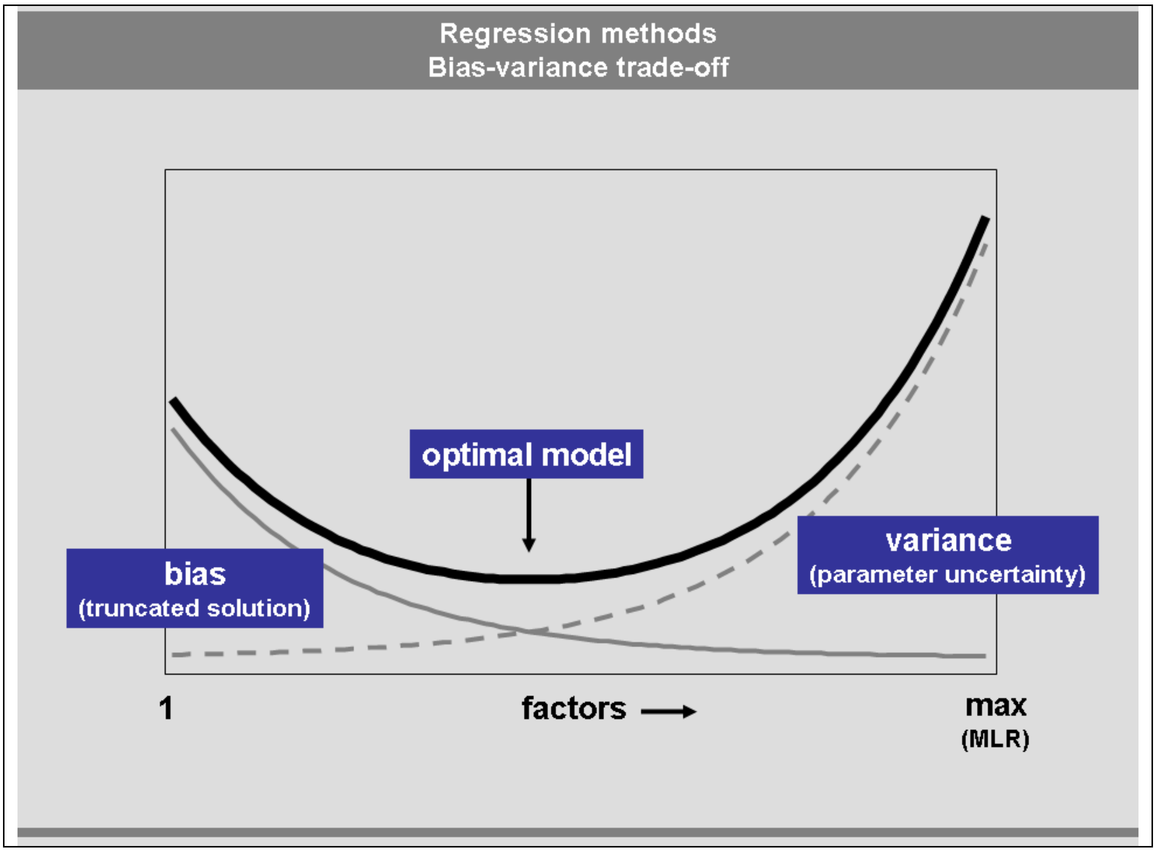
$(T'.T)^{-1}.T'.y = (T'.T)^{-1}.(T'.T).q$

$(T'.T)^{-1}.T'.y = q$

(PCR solution, y = X.b = X.P.q)

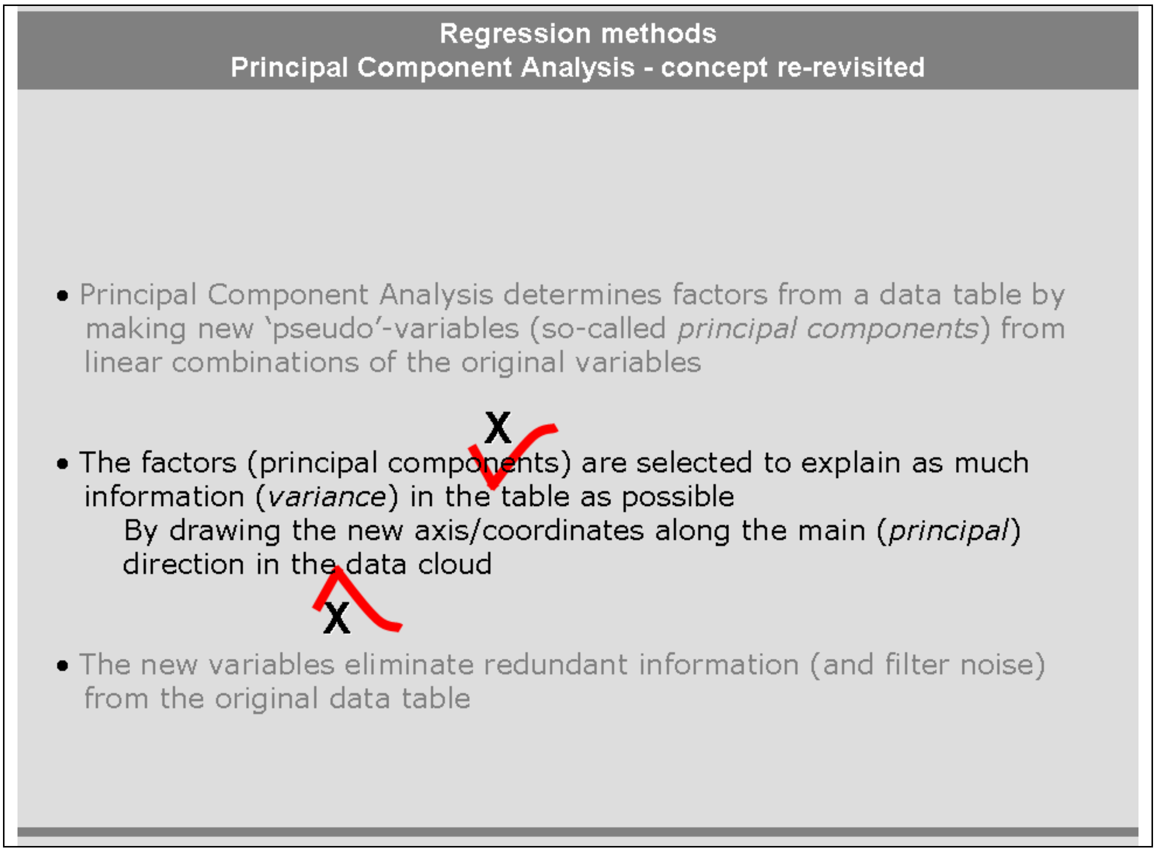
$$T'.T = \begin{pmatrix} 10 & & & \\ & 1 & & \\ & & 0.01 & \\ & & & \sim 0 \end{pmatrix}$$

$$(T'.T)^{-1} = \begin{pmatrix} \cancel{1/10} & & & \\ & 1/1 & & \\ & & 1/0.01 & \\ & & & \cancel{1/\sim 0} \end{pmatrix}$$



Regression methods
Bias-variance trade-off

The observation on the other page is the heart of factor-models: you want to include the factors/principal components/latent variables that are significant but in doing so you disregard smaller dimensions, introducing a bias in the parameters/solution. The gain is more stable parameter estimation, because the difficulty in estimating factors in the smaller/unstable dimensions is avoided. Somewhere there is an optimal model complexity in number of factors: the bias-variance trade-off.



Regression methods
Principal Component Analysis - concept re-revisited

- Principal Component Analysis determines factors from a data table by making new 'pseudo'-variables (so-called *principal components*) from linear combinations of the original variables

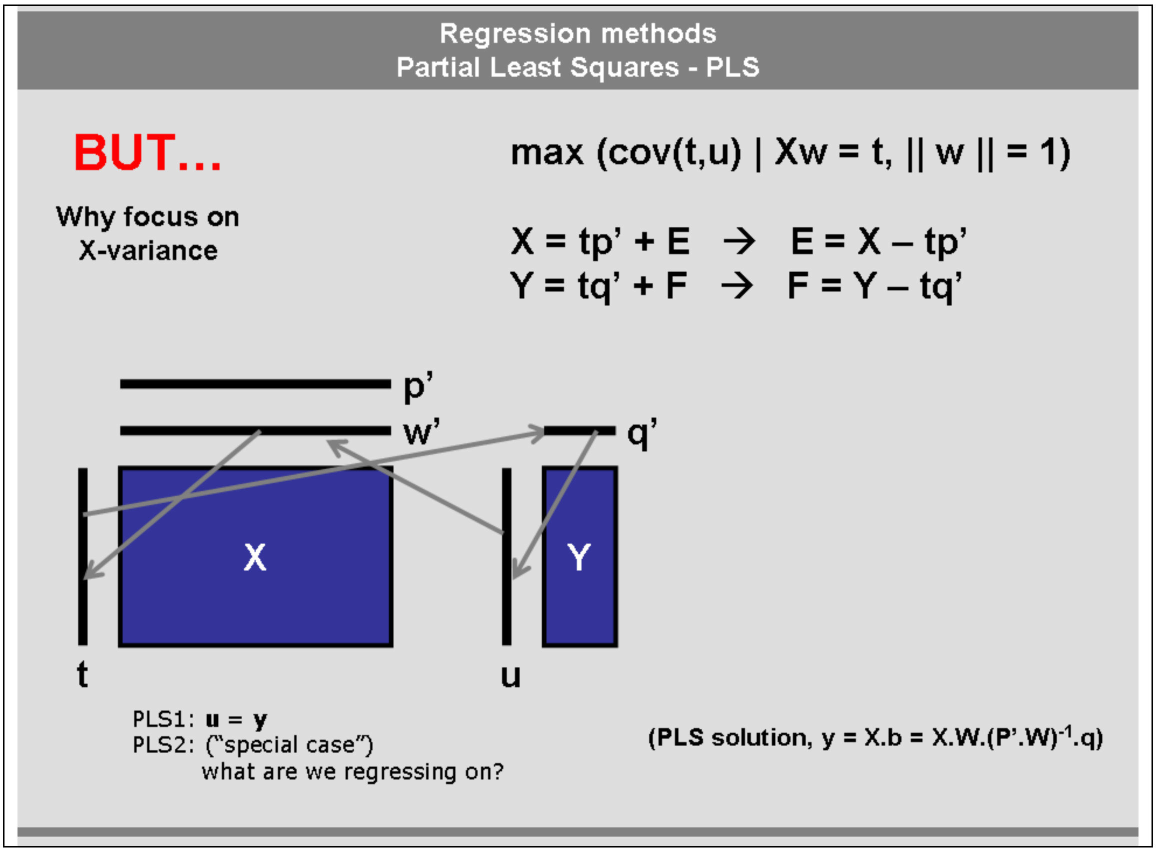
X ✓

- The factors (principal components) are selected to explain as much information (*variance*) in the table as possible

  By drawing the new axis/coordinates along the main (*principal*) direction in the data cloud

X

- The new variables eliminate redundant information (and filter noise) from the original data table



Regression methods
Partial Least Squares - PLS

BUT…

Why focus on X-variance

max (cov(t,u) | Xw = t, || w || = 1)

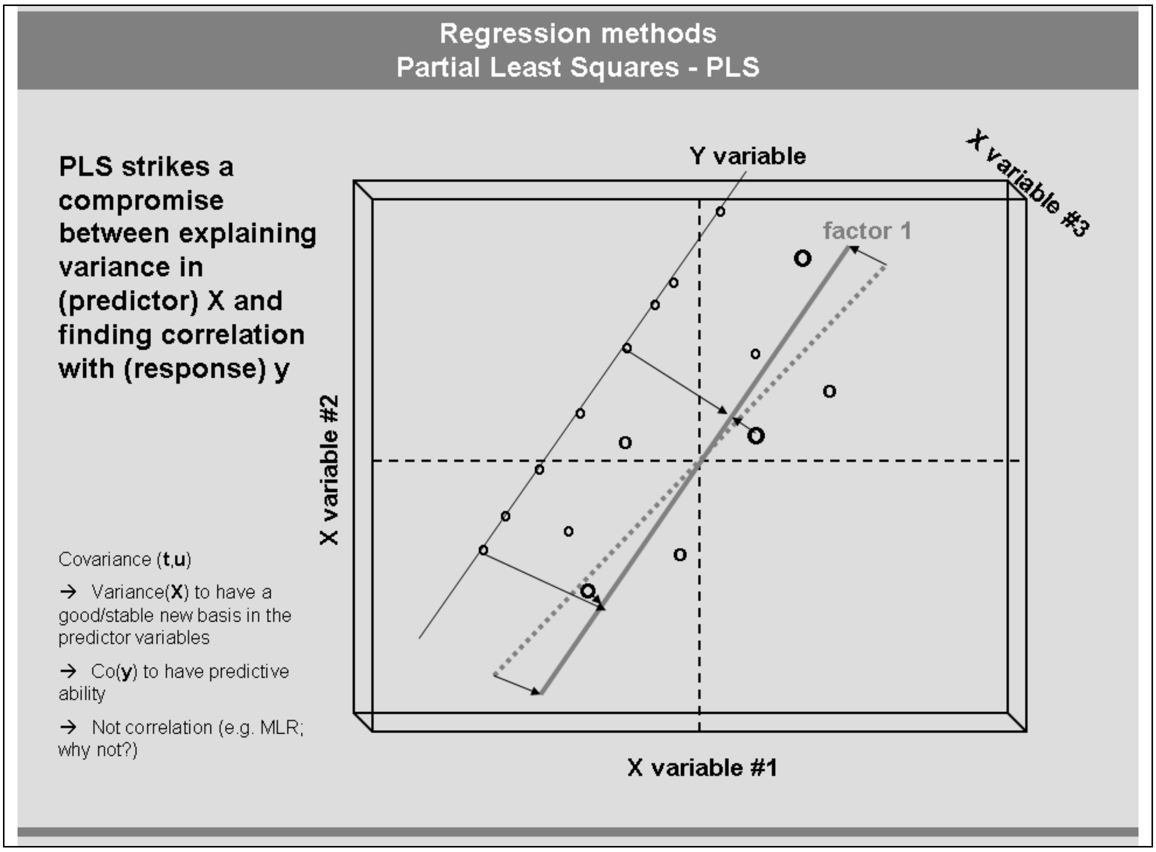
X = tp' + E → E = X – tp'
Y = tq' + F → F = Y – tq'

PLS1: **u** = **y**
PLS2: ("special case")
what are we regressing on?

(PLS solution, y = X.b = X.W.$(P'.W)^{-1}$.q)

So far the new factor-coordinate system was selected to explain as much of the **X**-variance as possible. There is however no guarantee that this set of loadings (or rather the scores formed by projection) is suitable for predicting **y**.

In PLS an alternative set of coordinates in the form of loading-weights **W** is defined. The criteria of selection are to both explain variance in **X** (to get a stable inverse) and correlated the direction with **y** (or **u** in the case of multiple **y**'s; to get good predictions): maximize the covariance. This more-or-less heuristic decision to make the two criteria equally important works well in practice and often leads to simpler models than e.g. PCR.



Regression methods
Partial Least Squares - PLS

PLS strikes a compromise between explaining variance in (predictor) X and finding correlation with (response) y

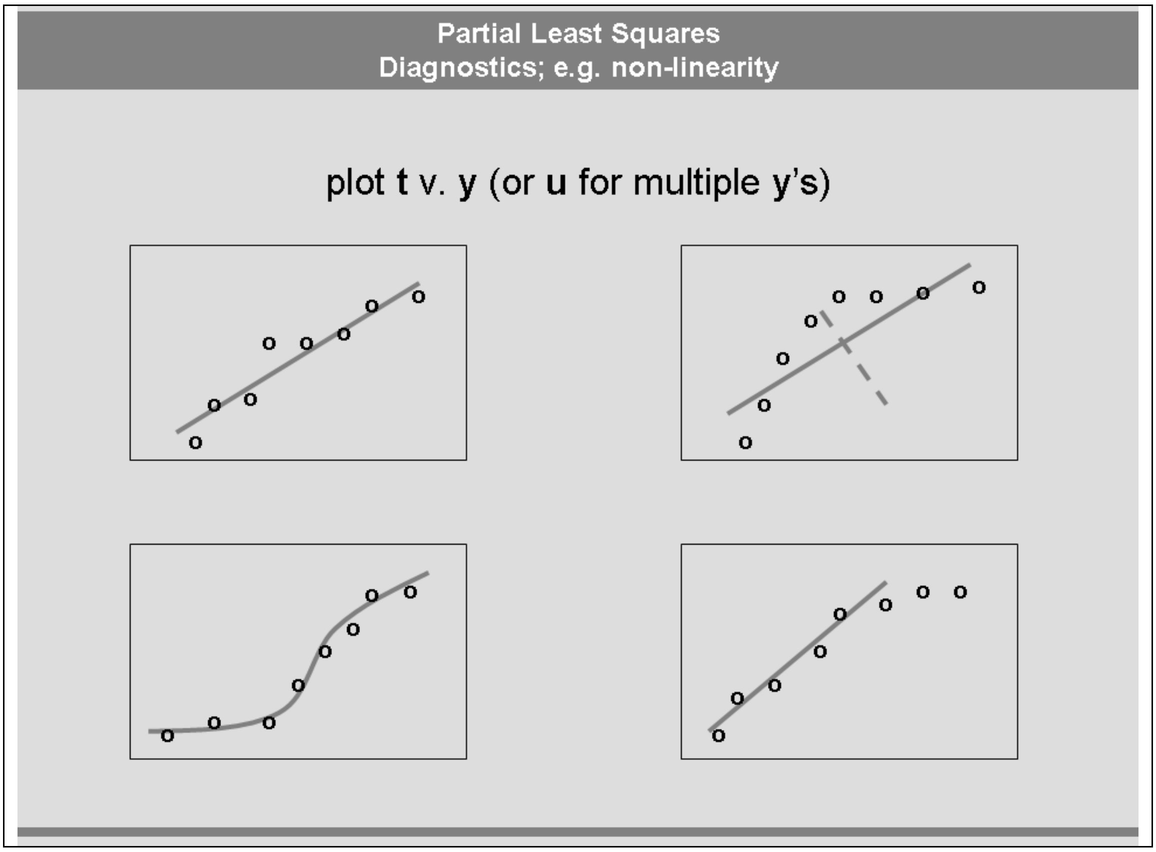
Covariance (**t**,**u**)

→ Variance(**X**) to have a good/stable new basis in the predictor variables

→ Co(**y**) to have predictive ability

→ Not correlation (e.g. MLR; why not?)

A (futile) attempt to visualize PLS: next to the three variable space of our **X**-block we have a one variable space of the **y**-block (non- overlapping spaces!). The original factor 1 is tilted (out of the plane!) in the direction of the y-space. Hence, PLS is a skewed projection. It is easier to imagine this in the object space (three **x**- and one **y**-variable both in eight dimensional spaces for this example) but that is impossible to draw.



Partial Least Squares
Diagnostics; e.g. non-linearity

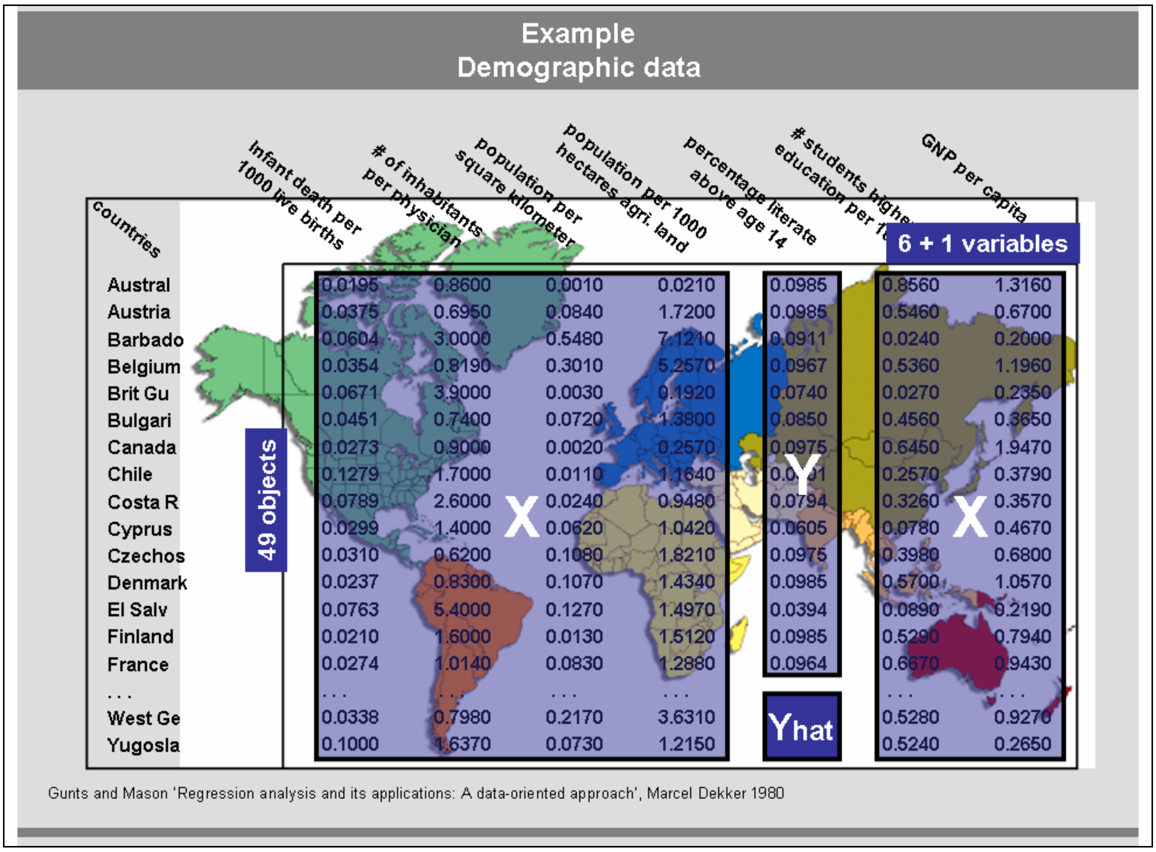
plot **t** v. **y** (or **u** for multiple **y**'s)

The actual regression part inside PLS is a simple, univariate, linear regression between the predictor/**X** pseudo variables **t**, and the responses/**y** or the pseudo variables **u** in case of multiple y's.

A point on regression factor-models: non-linearity in the parameters. A good way to detect non-linearity is to plot the two participants in the regression equation: **t** versus **y**.

PLS is able handle modest non-linearity by simply including more factors. Many authors have attempted to tackle linearity issue by defining more complex regression equation inside the PLS algorithm (the so-called inner relation between **t** and **u**/**y**). Unfortunately the relations are often too complex (e.g. detector saturation) to find an easy/general solution.

A good tip when you expect non-linearity is to augment the data-block **X** with its squared form $\mathbf{X}^2$.



Example
Demographic data

49 objects
6 + 1 variables
X
Y
X
Yhat

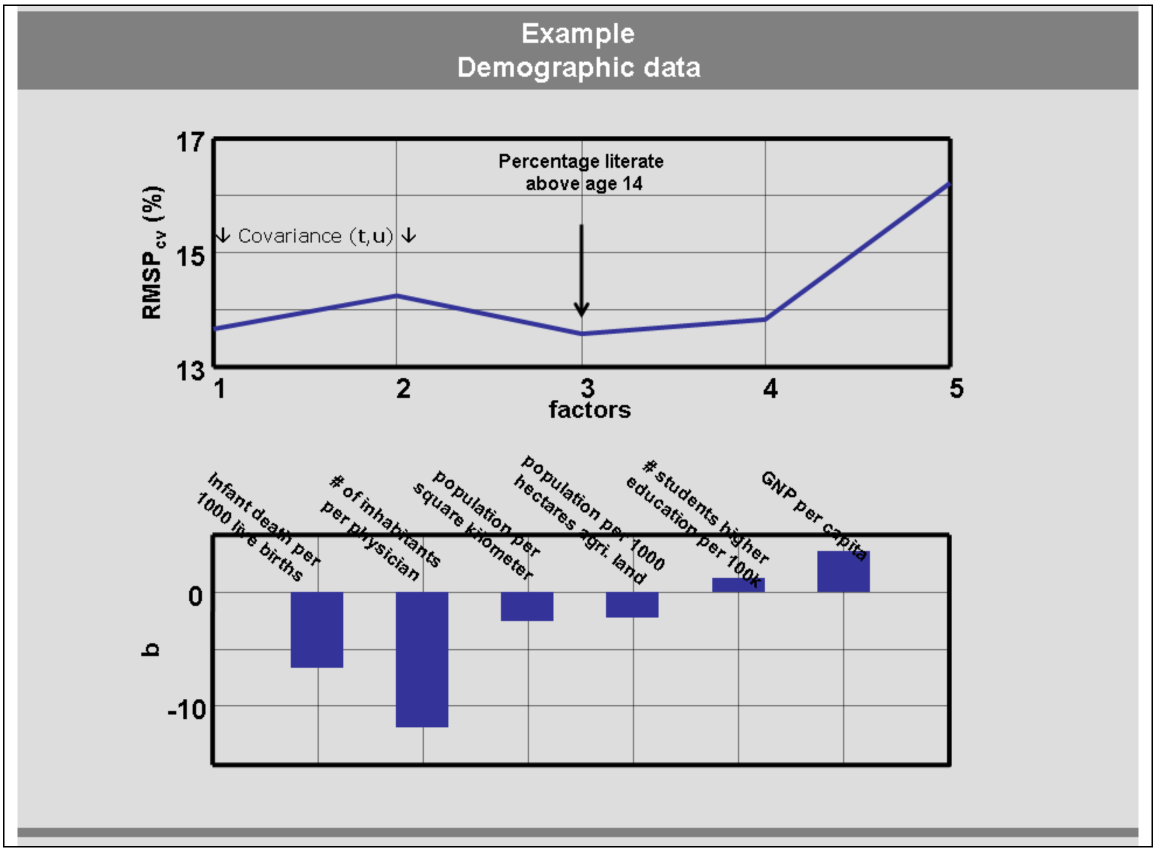
countries	Infant death per 1000 live births	# of inhabitants per physician	population per square kilometer	population per 1000 hectares agri. land	percentage literate above age 14	# students highe education per 1	GNP per capita
Austral	0.0195	0.8600	0.0010	0.0210	0.0985	0.8560	1.3160
Austria	0.0375	0.6950	0.0840	1.7200	0.0985	0.5460	0.6700
Barbado	0.0604	3.0000	0.5480	7.1210	0.0911	0.0240	0.2000
Belgium	0.0354	0.8190	0.3010	5.2570	0.0967	0.5360	1.1960
Brit Gu	0.0671	3.9000	0.0030	0.1920	0.0740	0.0270	0.2350
Bulgari	0.0451	0.7400	0.0720	1.3800	0.0850	0.4560	0.3650
Canada	0.0273	0.9000	0.0020	0.2570	0.0975	0.6450	1.9470
Chile	0.1279	1.7000	0.0110	1.1640	[illegible]	0.2570	0.3790
Costa R	0.0789	2.6000	0.0240	0.9480	0.0794	0.3260	0.3570
Cyprus	0.0299	1.4000	0.0620	1.0420	0.0605	0.0780	0.4670
Czechos	0.0310	0.6200	0.1080	1.8210	0.0975	0.3980	0.6800
Denmark	0.0237	0.8300	0.1070	1.4340	0.0985	0.5700	1.0570
El Salv	0.0763	5.4000	0.1270	1.4970	0.0394	0.0890	0.2190
Finland	0.0210	1.6000	0.0130	1.5120	0.0985	0.5290	0.7940
France	0.0274	1.0140	0.0830	1.2880	0.0964	0.6670	0.9430
. . .	. . .	. . .	. . .	. . .		. . .	. . .
West Ge	0.0338	0.7980	0.2170	3.6310		0.5280	0.9270
Yugosla	0.1000	1.6370	0.0730	1.2150		0.5240	0.2650

Gunts and Mason 'Regression analysis and its applications: A data-oriented approach', Marcel Dekker 1980

We use again the same small example as a reminder: socio-economical status of 49 countries world-wide determined by 7 indicator variables/demographic.

In the regression problem we try to predict "percentage literate above age 14" for the last 9 countries using the first 40 as training set.

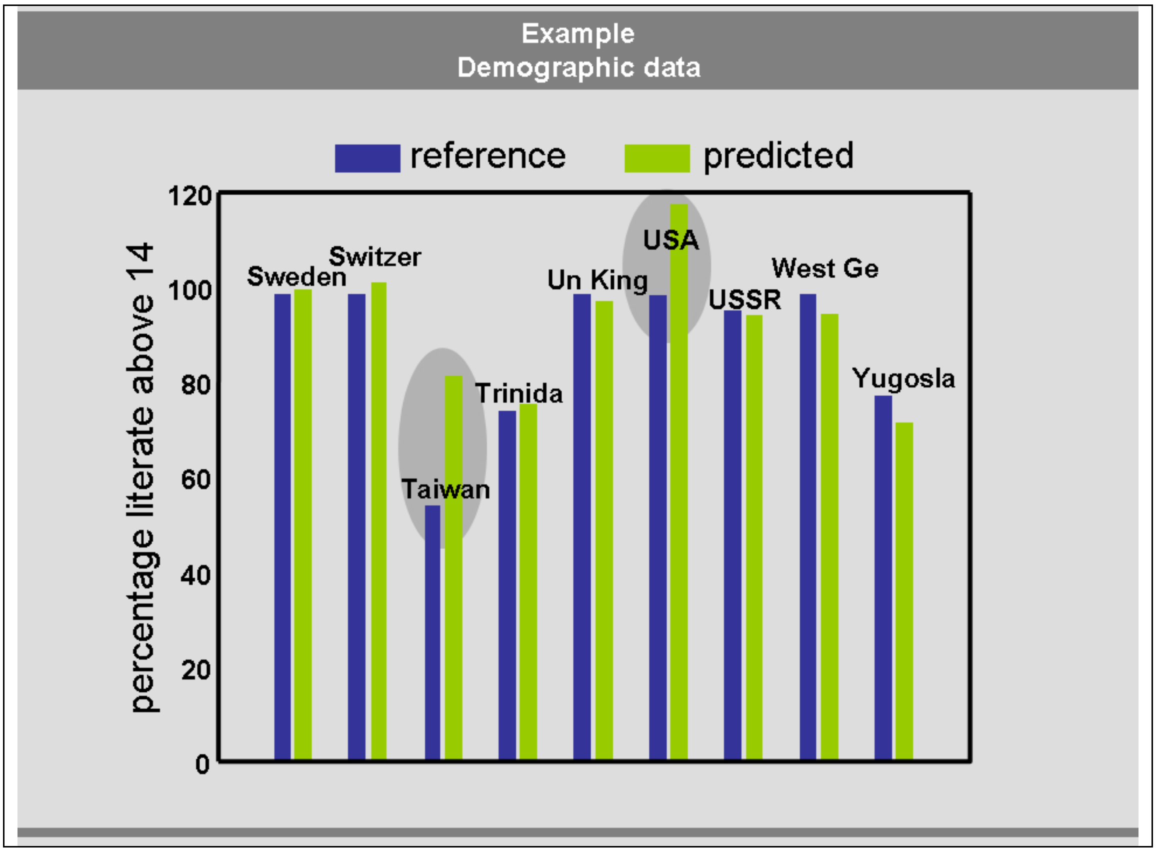
R.F. Gunst and R.L. Mason "Regression Analysis and Its Application: A Data-Oriented Approach" Marcel Dekker, New York, p. 358(1980)



Example
Demographic data

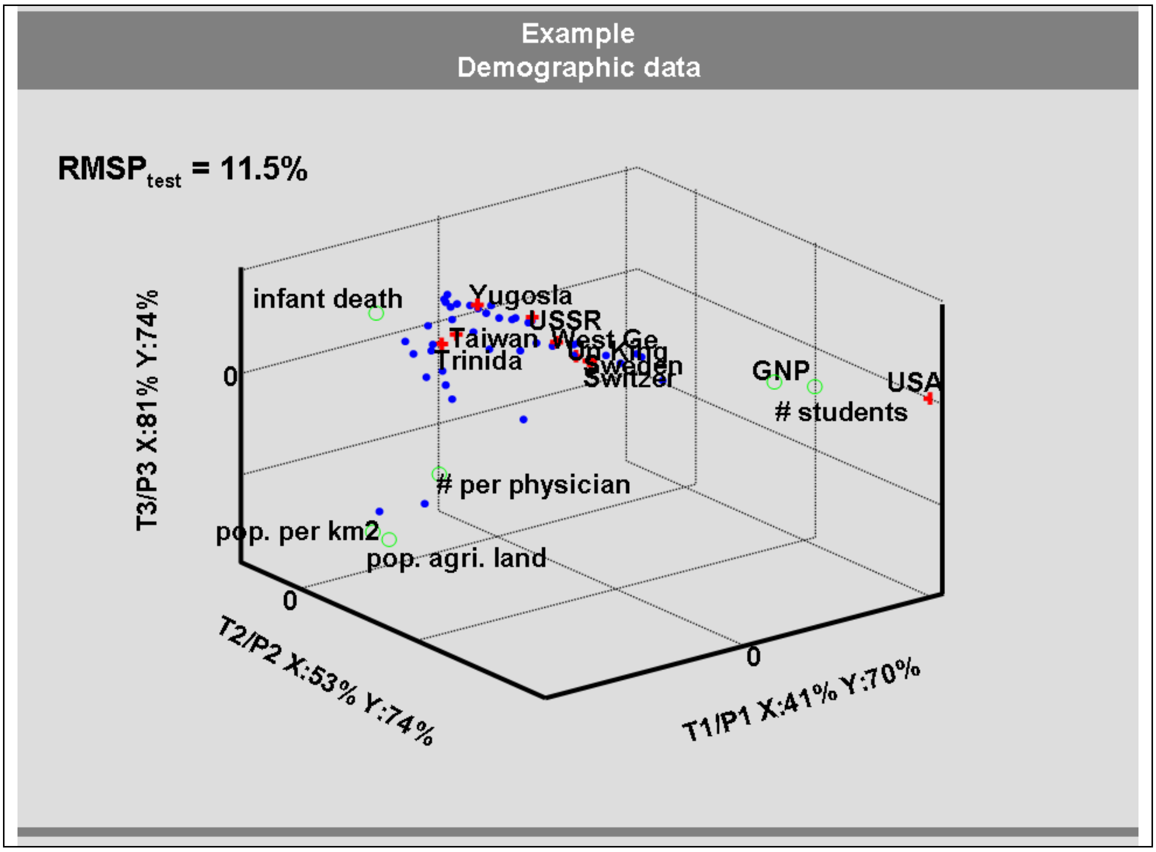
A three factor model gives slightly better RMSP$_{cv}$ than a one factor model, but maybe only one factor is sufficient.

Regression coefficients show that "percentage literate above age 14" negatively related to physicians and infant deaths. This same response variable is positively related to predictor variable GNP. This is in good agreement with the findings in the Principal Component Analysis.



Example
Demographic data

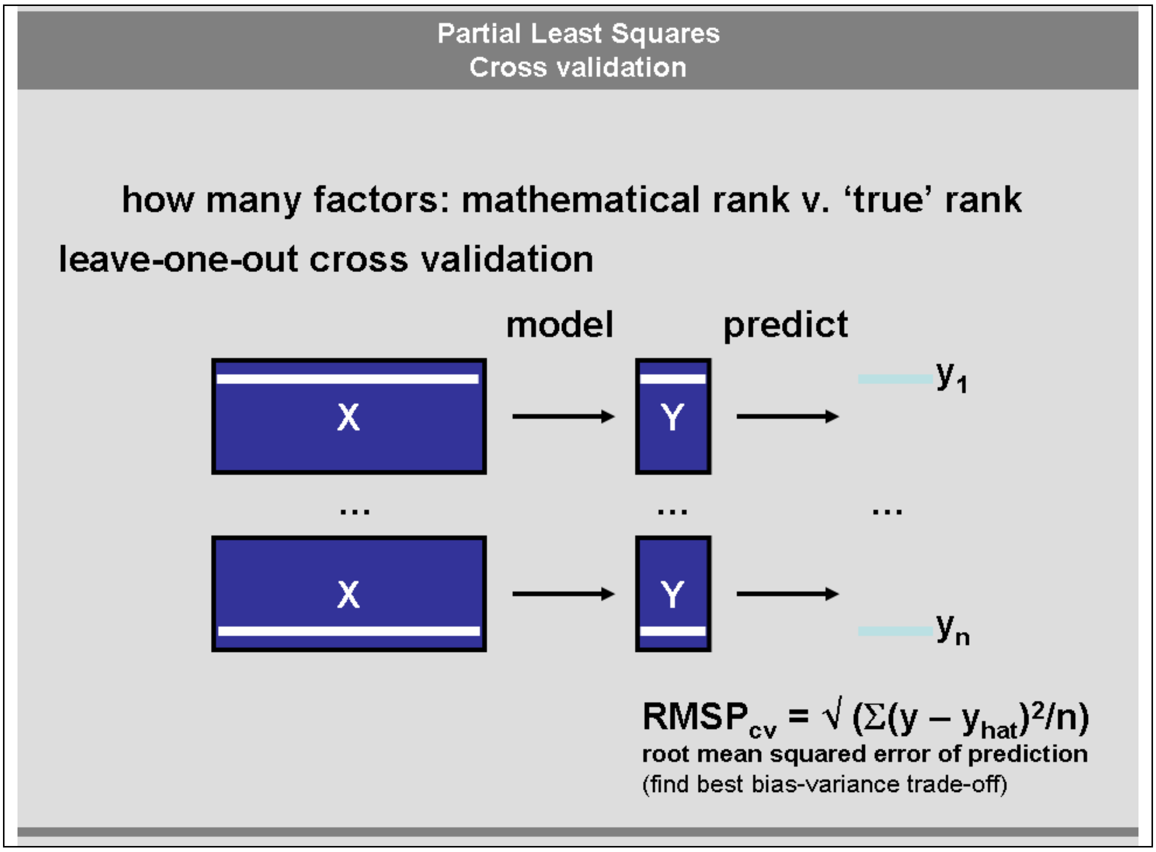
Overall the predictions of percentage literate from the other five socio-economical descriptors are good. Exceptional countries found in the PCA are also predicted relatively bad (Taiwan as a typical island state and USA which has (or had in 1980) a different university educational system).



Example
Demographic data

RMSP$_{test}$ = 11.5%

The exceptional position of the USA also becomes clear by looking at the bi-plot for the first three PLS factors.



Partial Least Squares
Cross validation

how many factors: mathematical rank v. ‘true’ rank

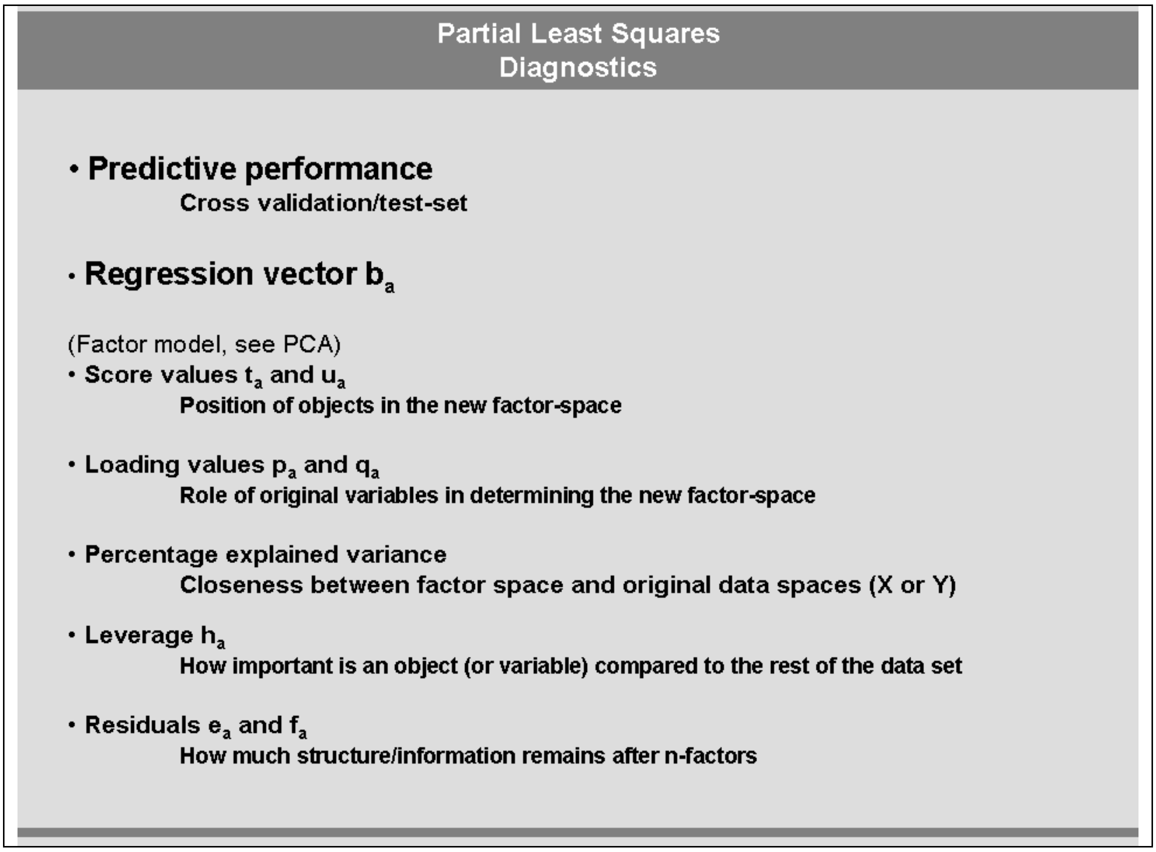
leave-one-out cross validation

$\mathrm{RMSP}_{cv} = \surd\,(\Sigma(y - y_{hat})^2/n)$

root mean squared error of prediction

(find best bias-variance trade-off)

In factor regression problems we have to determine the optimal model complexity. This can be done by leave-one-object-out (or leave-a-particular-set-out) cross validation. We are looking for the minimum of the bias-variance RMSP-plot (in practice a no-longer-a-significant-drop, to get parsimonious models).



Partial Least Squares
Diagnostics

- **Predictive performance**

  Cross validation/test-set
- **Regression vector b_a**

(Factor model, see PCA)

- Score values t_a and u_a

  Position of objects in the new factor-space
- Loading values p_a and q_a

  Role of original variables in determining the new factor-space
- Percentage explained variance

  Closeness between factor space and original data spaces (X or Y)
- Leverage h_a

  How important is an object (or variable) compared to the rest of the data set
- Residuals e_a and f_a

  How much structure/information remains after n-factors

In PLS regression modeling we have the same + some more diagnostic tools as in PCA.

Linear Algebra / PCA

In this part of the text we will use linear algebra methods and SVD/PCA to get some more experience in using the Matlab program. Let's start by generating some data. First we define two “true” loadings (Gaussian-curves of length 100) and two times ten “true” scores

```
>> p1 = gauss(100,65,8)';
>> p2 = gauss(100,35,6)';
>> t1 = [1:10]';
>> t2 = [3:-.1:2.1]' + randn(10,1)*0.5;
```

From this we generate our data matrix X, adding some noise, and plot the results

```
>> Xclean = t1*p1' + t2*p2';
>> X = Xclean + randn(size(t2*p2'))*0.003;
>> plot(1:100,Xclean,'b',1:100,X,'g');
>> grid;
```

We can define labels for the ten objects/samples in our set

```
>> ObjLab = ['#0';'#1';'#2';'#3';'#4';'#5';'#6';'#7';'#8';'#9']
```

To compute an SVD on this data table enter (function “diag” selects diagonal entries)

```
>> [U,D,V] = svd(X);
>> d = diag(D)
>> plot(1:10,diag(D));
```

Check to see if the criteria defined in the figure on SVD are fulfilled: UU’ = U’U = $_{10}I^{10}$, VV’ = V’V = $_{100}I^{100}$, D = diagonal and sorted according to size, etc.

How much of the variation in X, in percentage, is explained by the first factor? We can compute this directly from the singular values in “D”

```
>> d(1)^2/sum(d.^2)*100
```

And how about the cumulative percentage variation explained for the first two factors? Try to dissect this Matlab command line

```
>> sum(d(1:2).^2)/sum(d.^2)*100
```

To get the first three PCA scores and loadings from the SVD decomposition simply compute

```
>> T = U(:,1:3)*D(1:3,1:3);
>> P = V(:,1:3);
>> plot(1:100,P(:,1:3));
>> legend('P1','P2','P3')
```

Now try the same SVD decomposition for the clean data set (without noise, “Xclean” from above). What is the effect of adding noise? Try the Matlab commands “rank” and “cond” (= condition number).

To see e.g. the relation between the first true scores “t1” and the scores on the first PCA factor “T” (“text” puts the strings in “ObjLab” on position x,y)

```
>> plot(t1,T(:,1),'.');
>> xlabel('t1');
>> ylabel('T1')
>> text(t1,T(:,1),ObjLab)
```

The correlation coefficient between “t1” and the estimate “T(:,1)” can be computed as follows (on the diagonal is the correlation coefficient of a vectors with itself, which is of coarse always 1)

```
>> R = corrcoef(t1,T(:,1))
```

The squared correlation coefficient (= explained variance) is

```
>> R(1,2).^2
```

How about the second set of true scores “t2”?

To study the alternating least squares (ALS) algorithm we need some more code (assign a random starting point in “A”, “inv” computes the inverse of the matrix)

```
>> A = randn(10,2);
>> for a=1:100, B = (inv(A'*A)*A'*X)'; A = X*B*inv(B'*B); end
```

This script does not specify a real convergence criterion. It just iterates/loops true the two steps one hundred times (the “for … end” Matlab command), assuming the minimum in the least squares solution is found by then.

Lets plot the results

```
>> figure
>> subplot(2,1,1); plot(1:100,B); title('loadings')
>> subplot(2,1,2); plot(t1,A(:,1),'.-',t2,A(:,2),'.-'); title('scores')
```

Try this ALS+plot procedure a couple of times (open a new figure window each time) with new random starting values in “A”. What are the differences and similarities between solutions? How are these solutions different from the SVD/PCA results found earlier?

Regression / PCR

In this part we will look at some regression equations in Matlab. Let's start by generating some data (same as in “PCA” part).

```
>> p1 = gauss(100,65,8)';
>> p2 = gauss(100,35,6)';
>> t1 = [1:10]';
>> t2 = [3:-.1:2.1]' + randn(10,1)*0.5;
```

Then we generate our data matrices plus some noise

```
>> Xclean = t1*p1' + t2*p2';
>> X = Xclean + randn(size(t2*p2'))*0.003;
>> y = t1 + t2 + randn(size(t2))*0.1;
>> ObjLab = ['#0';'#1';'#2';'#3';'#4';'#5';'#6';'#7';'#8';'#9']
```

To compute an SVD on this data enter

```
>> [U,D,V] = svd(X);
```

From this decomposition we compute the Multiple Linear Regression (MLR) solution (10 object by 100 variables, so the maximum number of factors is 10) and plot it

```
>> b10 = V(:,1:10)*inv(D(1:10,1:10))*U(:,1:10)'*y
>> plot(1:100,b10);
```

This will look like a lot of noise! Why is that? We included too many factors. Let’s look at the 1, 2 and 3-factor Principal Component Regression solutions instead

```
>> b1 = V(:,1:1)*inv(D(1:1,1:1))*U(:,1:1)'*y
>> b2 = V(:,1:2)*inv(D(1:2,1:2))*U(:,1:2)'*y
>> b3 = V(:,1:3)*inv(D(1:3,1:3))*U(:,1:3)'*y
>> plot(1:100,[b1 b2 b3]);
>> legend('b1','b2','b3')
```

We can also predict (or rather “fit” in this case) the y-values

```
>> yp1 = X*b1;
>> yp2 = X*b2;
>> yp3 = X*b3;
>> plot(y,yp1,'.-', y,yp2,'.-', y,yp3,'.-'); grid
>> legend('1 factor','2 factors','3 factors');
>> line([2 14],[2 14])
```

The “line” command plots a line: offset 0, slope 45°.

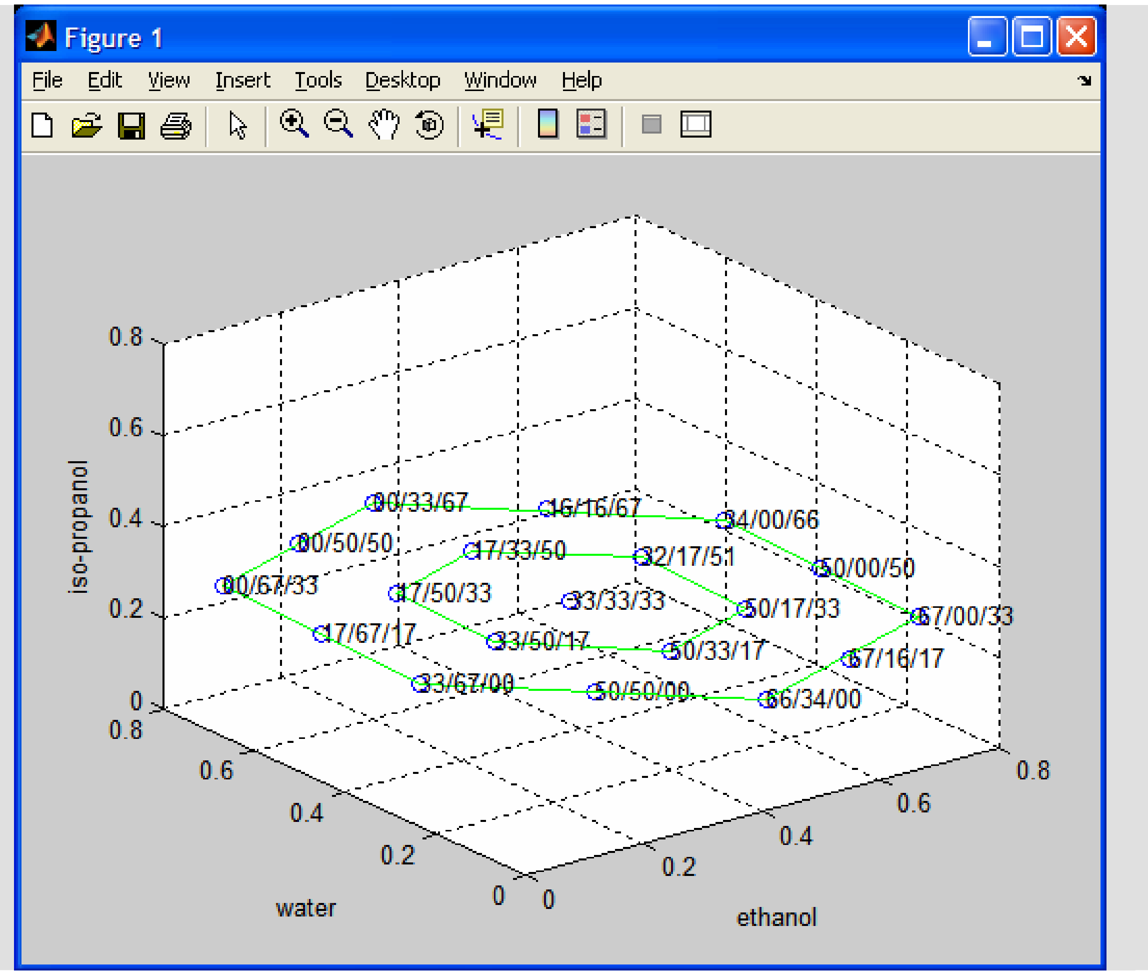
NIR with temperature effects

A set of Near InfraRed (NIR)*) spectra is available in the Matlab data-file “NIRdata.mat”. This set (three X-blocks “specXX”: 19 x 512; spectra recorded at 30°C, 50°C and 70°C) is part of a larger triangular design formed by mixtures of ethanol (first column in “conc”: the Y-block with mass fractions of the constituents), water (second column) and iso-propanol (third column). The corner points of the design (the pure components/spectra) are removed from the main set, but are available in the data-file. The Matlab variable “ObjLab” contains appropriate name-labels for the objects (see first figure below for the design).

*) Florian Wülfert, Wim. Th. Kok and Age K. Smilde “Influence of temperature on vibrational spectra and consequences for the predictive ability of multivariate models” Anal.Chem.(1998)1761-1767

To plot the mixture design points:

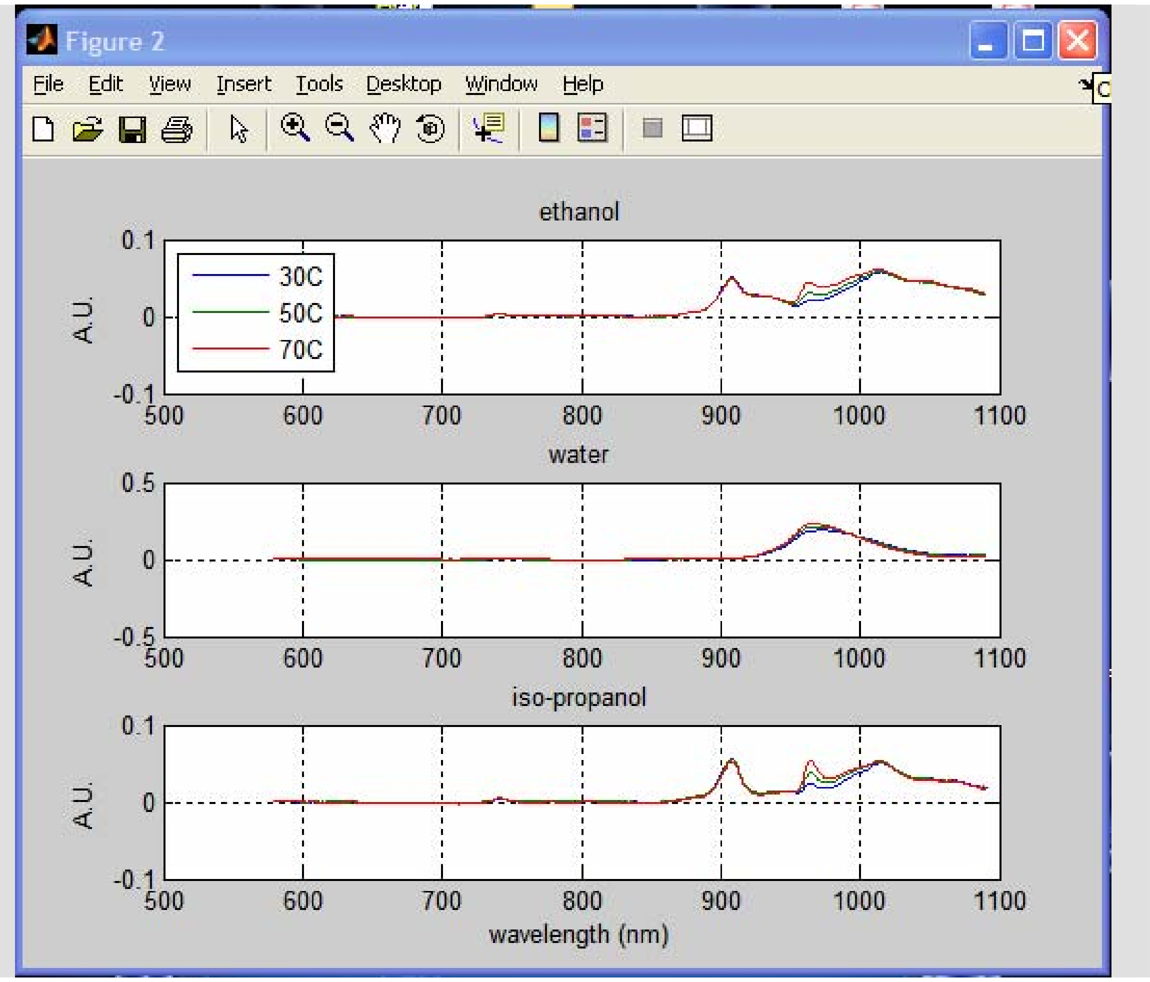
```
>> clear all
>> close all
>> load NIRdata
>> plotNIRDesign(conc)
>> text(conc(:,1),conc(:,2),conc(:,3),ObjLab)
>> xlabel('ethanol')
>> ylabel('water')
>> zlabel('iso-propanol')
```



Use the rotate button to get a better view of the design!

Next, have a look at the temperature influence by plotting the pure components of the mixtures:

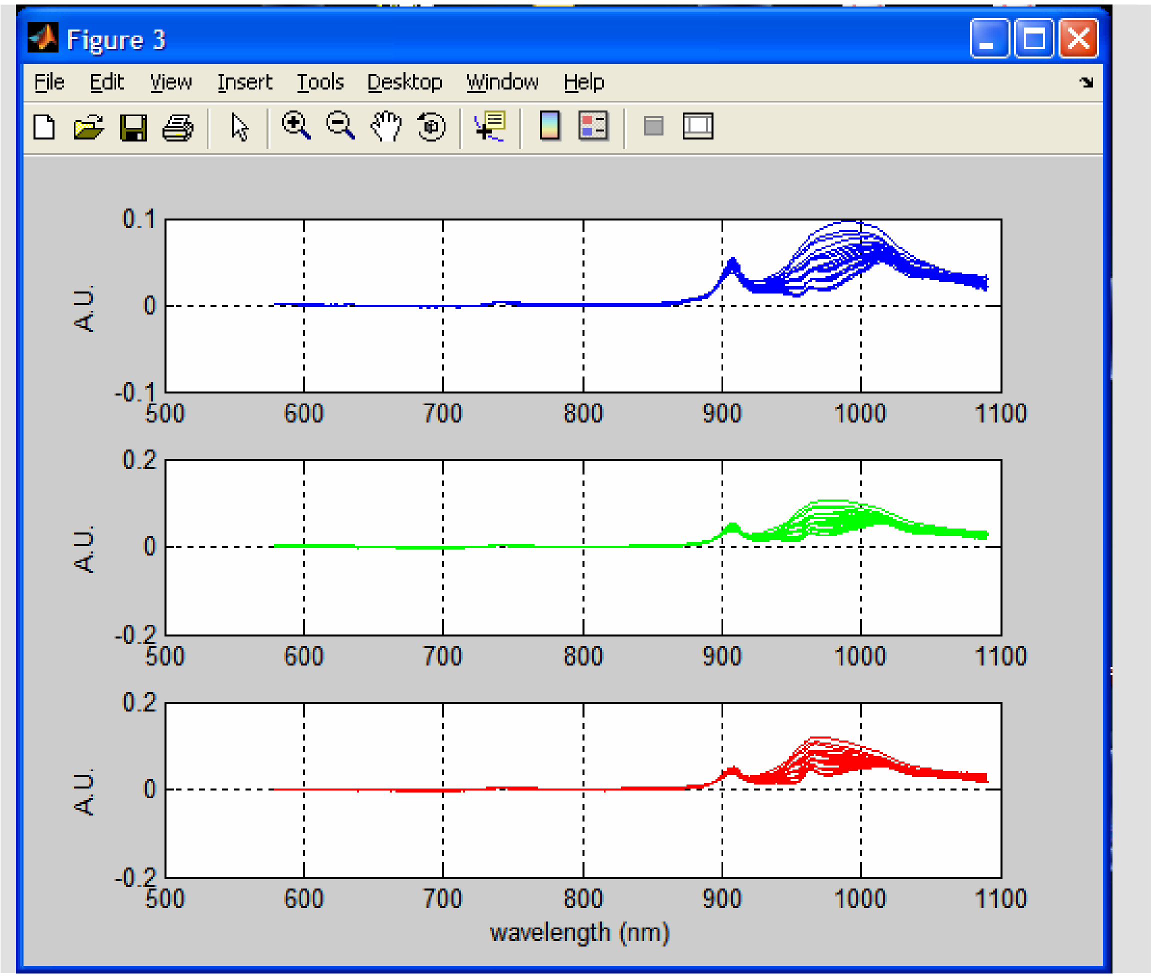
```
>> figure
>> subplot(3,1,1)
>> plot(wavelen,[ethanol30; ethanol50; ethanol70])
>> legend('30C','50C','70C',2)
>> title('ethanol')
>> ylabel('A.U.')
>> grid
>> subplot(3,1,2)
>> plot(wavelen,[water30; water50; water70])
>> title('water')
>> ylabel('A.U.')
>> grid
>> subplot(3,1,3)
>> plot(wavelen,[isopropanol30; isopropanol50; isopropanol70])
>> xlabel('wavelength (nm)')
>> ylabel('A.U.')
>> grid
>> title('iso-propanol')
```



Use the zoom button for a detailed view. What is the chemical nature of the changes?

The different mixture-spectra look as follows:

```
>> figure
>> subplot(3,1,1)
>> plot(wavelen,spec30,'b')
>> ylabel('A.U.')
>> grid
>> subplot(3,1,2)
>> plot(wavelen,spec50,'g')
>> ylabel('A.U.')
>> grid
>> subplot(3,1,3)
>> plot(wavelen,spec70,'r')
>> xlabel('wavelength (nm)')
>> ylabel('A.U.')
>> grid
```



A SVD/PCA analysis of the 30°C data-set could go as follows:

```
>> [U30,D30,P30] = svd(spec30);
>> T30 = U30*D30;
```

Have a look at the loading-vectors (pseudo spectra); how do they compare to the raw spectra? How much variance is explained by the first four factors (use D30)? What is the rank of this system? What should the rank from a chemical point of view? And what about closure?

Can you still find the design (use rotate) plotting the scores of the first three factors?

```
>> figure
>> plotNIRDesign(T30(:,1:3))
>> xlabel('PC1'); ylabel('PC2'); zlabel('PC3');
```

If you rotate this plot you will see it looks like a plane (representing the mixture design) in a 3D room. What we did not do in our SVD/PCA analysis above is mean centering the data before decomposition. But this is by now "easy" in Matlab (hints: first we determine the size of the matrix, then we compute the mean spectrum – thus over the columns in the row direction; finally a trick: we subtract a matrix from the data build from the outer product of a vector of ones and the mean spectrum. What happens here? Check the outer product.)

```
>> [N,M] = size(spec30);
>> mean_spec30 = mean(spec30,1);
>> spec30c = spec30 - ones(N,1)*mean_spec30;
```

Check the result of the mean centering pre-processing

```
>> figure
>> subplot(2,1,1);
>> plot(wavelen,spec30,'b',wavelen,mean_spec30,'g'); grid;
>> subplot(2,1,2); plot(wavelen,spec30c,'r'); grid;
```

And try SVD/PCA on this

```
>> [U30c,D30c,P30c] = svd(spec30c);
>> T30c = U30c*D30c;
>> figure
>> plotNIRDesign(T30c(:,1:3))
>> xlabel('PC1'); ylabel('PC2'); zlabel('PC3');
```

What happend to the rank and the singular values in "D30c"? What happened to the image of the design in the score plot?

Try the same for the mixture-sets recorded at 50°C and 70°C. What effect does temperature have on the results?

Basic Statistics

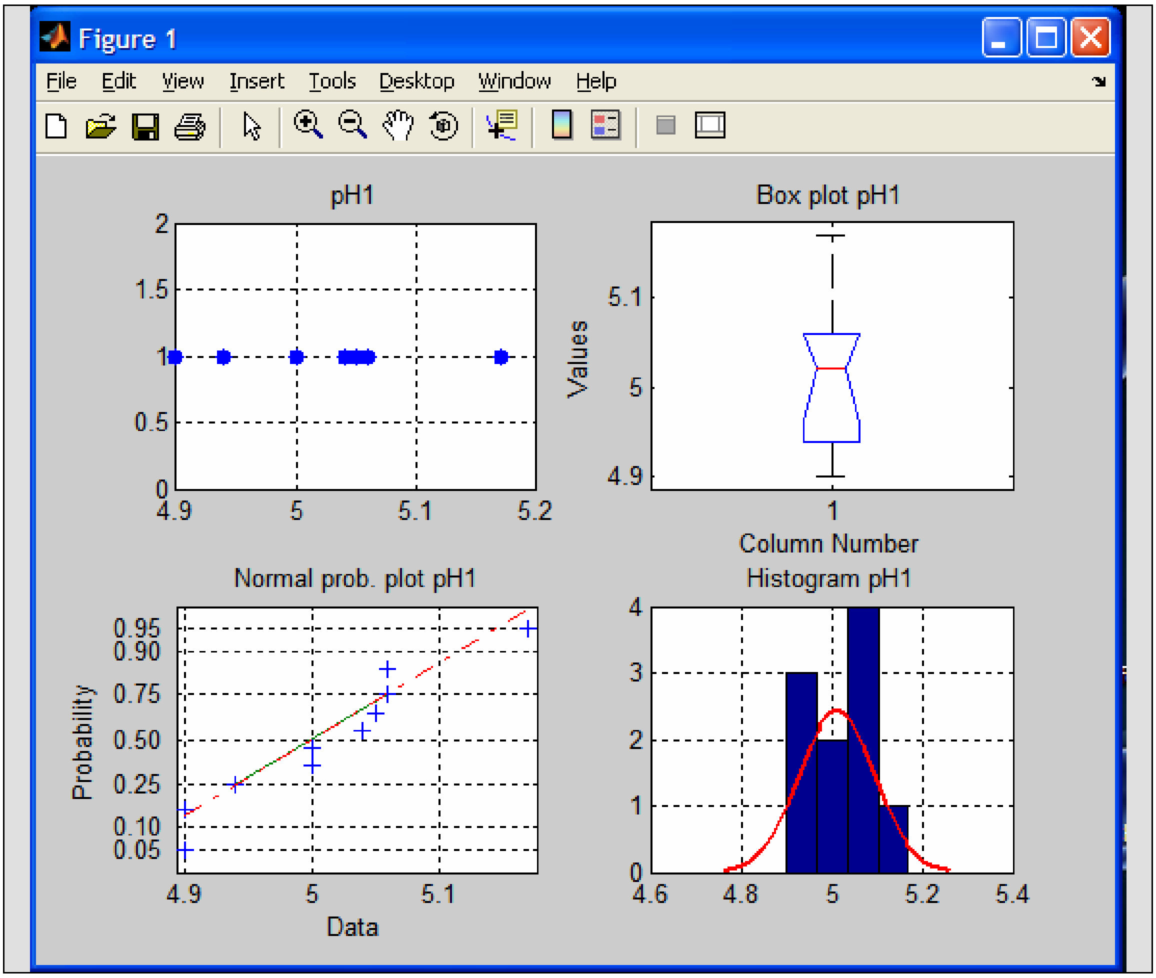
To study some basic we will use a small example of 10 pH measurement from one solution. The data is as follows

```
>> clear all
>> close all
>> pH1 = [4.90 5.06 5.05 5.17 5.06 4.94 5.04 4.90 5.00 5.00]';
```

The first thing to do when collecting new data is to plot it!!!

In this case we have the 10 observations, a so-called box-and-whiskers plot, the histogram (with a normal or Gaussian distribution imposed) and the normal probability plot

```
>> figure
>>subplot(2,2,1);
>> plot(pH1,ones(1,length(pH1)),'.','MarkerSize',20);
>> grid; title('pH1');
>> subplot(2,2,2);
>> boxplot(pH1,'notch','on');
>> title('Box plot pH1');
>> subplot(2,2,3);
>> normplot(pH1);
>> title('Normal prob. plot pH1');
>> subplot(2,2,4);
>> histfit(pH1); grid;
>> title('Histogram pH1')
```



This data-set is a little small to study the normal probability plot and the histogram function. Try the following code which generates to vectors with 200 random numbers each

```
>> figure
>> xn = randn(1,200); x = rand(1,200);
>> subplot(2,2,1); histfit(xn);
>> subplot(2,2,2); histfit(x);
>> subplot(2,2,3); normplot(xn);
>> subplot(2,2,4); normplot(x);
```

What is the difference between random numbers generated by "rand" and "randn" (use the help function)?

Compute the important statistics for this pH data-set: mean standard deviation, relative standard deviation and standard error of the mean

(make sure you understand both the meaning of these numbers AND the Matlab commands to calculate them)

```
>> m_pH1 = mean(pH1)
>> s_pH1 = std(pH1)
>> RSD_pH1 = s_pH1/m_pH1*100
>> SE_m_pH1 = std(pH1)/sqrt(10)
```

The 95% confidence interval can be formulated as follows ("tinv" is a function equivalent to the student-t lookup table; note 5% two-sided it 2.5% or 0.25 for each side)

```
>> t95p = tinv(0.975,length(pH1)-1)
>> L = m_pH1 - t95p*SE_m_pH1;
>> U = m_pH1 + t95p*SE_m_pH1;
>> disp([num2str(L,3) ' < mu pH1 < ' num2str(U,3)])
```

Next step is the comparison of two data-sets. We make a second vector with pH values and plot the result

```
>> pH2 = [5.10 5.07 5.21 4.91 5.14 5.19 5.17 5.16 5.10 5.17]';
>> figure
>> subplot(2,1,1);
>> plot(pH1,ones(1,length(pH1)),'.b',pH2,2*ones(1,length(pH2)),'.g','MarkerSize',20);
>> grid; title('pH1 and pH2');
>> subplot(2,1,2);
>> boxplot([pH1 pH2],'notch','on'); title('Box plot pH1 and pH2');
```

Are the two pH-series the same? To compare them we can use the so-called Fisher F-test, based on the ration between Treatments (= the two pH series) and the total error/uncertainy ("finv" is a function equivalent to the F lookup table; again, make sure you understand both the meaning of these numbers and the Matlab commands to calculate them)

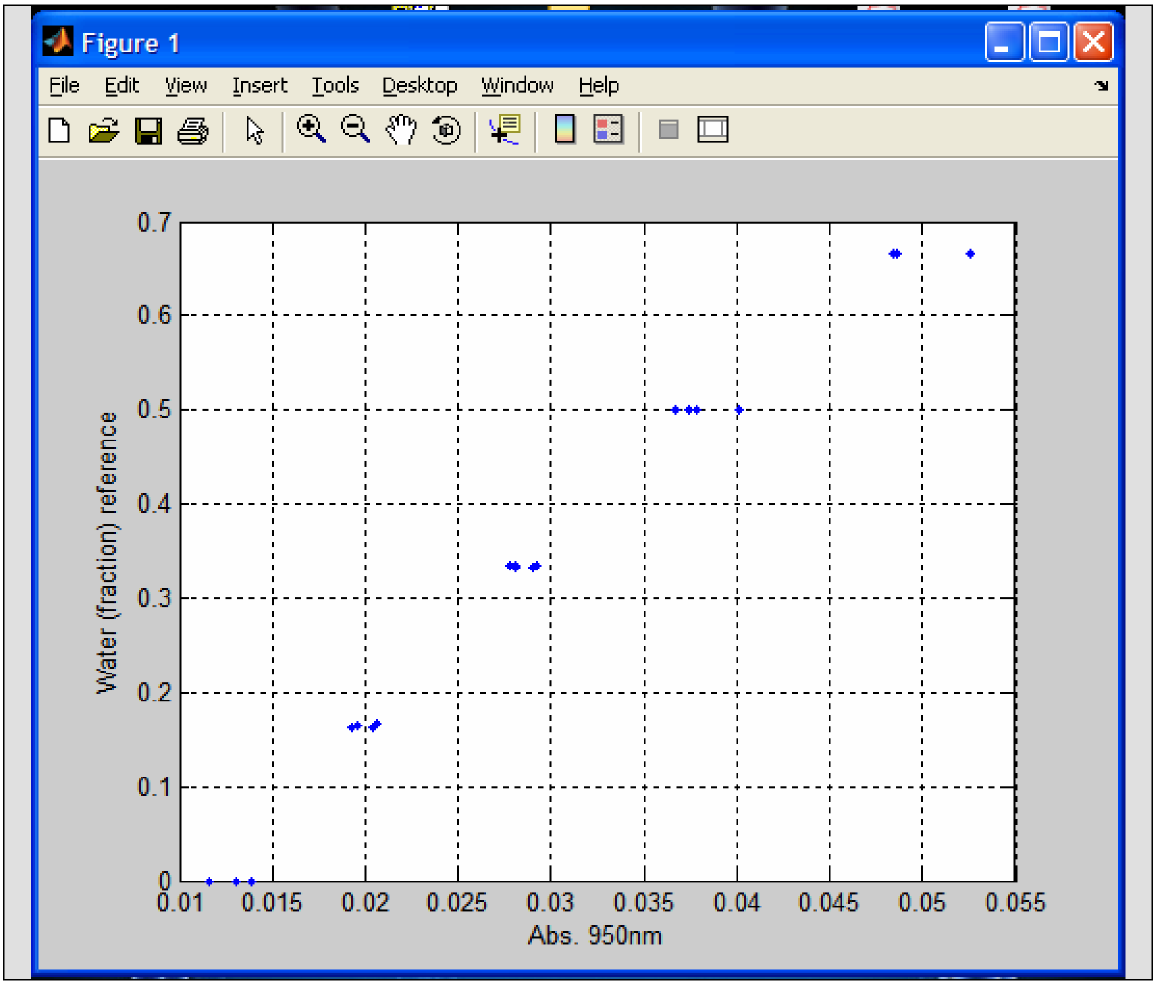
```
>> MStreatment = sum(10*([mean(pH1);mean(pH2)]-mean([pH1;pH2])).^2)/(2-1)
>> MSerror = sum([(pH1-mean(pH1)).^2;(pH2-mean(pH2)).^2])/(20-2)
>> F = MStreatment/MSerror
>> c = [0.75 0.90 0.95 0.975 0.99];
>> [100-c*100;finv(c,1,18)]
```

From the list we see that finding these 2 x 10 numbers, under the assumption that the pH's of the two series where the same, is smaller then 1%.

Linear Regression

We will use the NIR spectra from above to implement univariate linear regression in Matlab. If we want to predict the water contents in our samples we could use e.g. the absorbance value at 950nm (confirm this from the spectral plots above; closely study the way we find 950nm in the variable “wavelen”; remember water is the second column in “conc”)

```
>> clear all
>> close all
>> load NIRdata
>> y = conc(:,2);
>> [dummy,index950] = min(abs(wavelen-950));
>> wavelen(index950)
>> X = spec30(:,index950);
```



A plot of water fraction as a function of the absorbance at 950nm looks as follows

```
>> figure
>> plot(X,y,'.')
>> grid; xlabel('Abs. 950nm'); ylabel('Water (fraction) reference');
```

Next we form our matrix of independent **X**, where we add a column of ones for the offset (the b_0 term). From this matrix and the dependent **y** we can estimate our unknowns in the regression equation $y_{water} = b_0 + x_{950nm}.b_1$;

```
>> X = [ones(size(X)) X];
>> b = inv(X'*X)*X'*y
```

To see how our model performs we estimate our water fractions and make the regular predicted-versus-reference plot (red line is the ideal line: offset zero, slope 1), the Root Mean Square error of Prediction for fit (Why is this fit, and nor real prediction? Check the formula for RMSP, working from the inside to the outside), and the correlation coefficient

```
>> yhat = X*b;
>> figure
>> plot(y,yhat,'.',[0 1],[0 1],'r')
>> grid; axis square; xlabel('Water (fraction) reference');
>> ylabel('Water (fraction) predicted/fitted');
>> rmsp_fit = sqrt(mean((y-yhat).^2))
>> corrcoef(y,yhat)
```

Alternatively we can plot our estimates together with the original absorbances

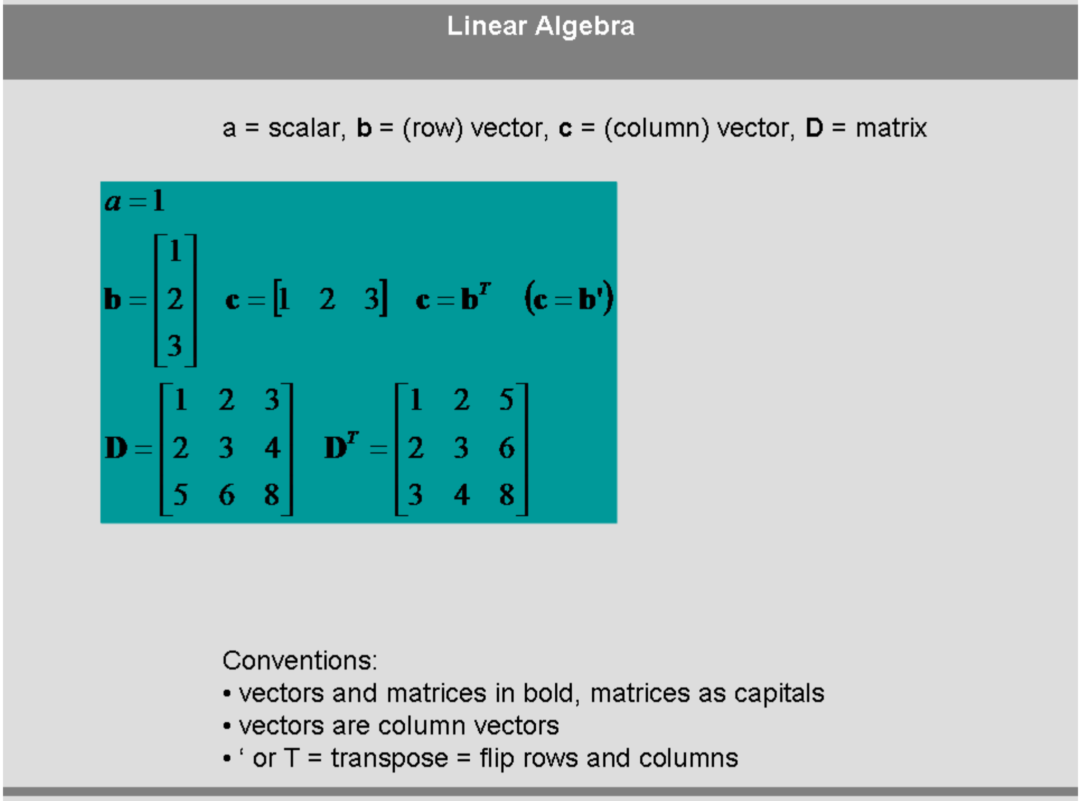
```
>> figure
>> plot(X(:,2),y,'.',X(:,2),yhat,'+g',[0 max(X(:,2))],[[1 0]*b [1 max(X(:,2))]*b],'b')
>> grid; xlabel('Abs. 950nm'); ylabel('Water (fraction) reference');
```

Why is this regression model of NIR spectra to predict water not very good (hint: have another look at the spectra)? And how could we improve it (hint: what about switching to MLR or PLS including e.g. absorbance wavelength 908nm – now we are back at the origin of Chemometrics again)?

Here is some “quick and dirty” steps to achieve MLR

```
>> [dummy,index908] = min(abs(wavelen-908));
>> X = [X spec30(:,index908)];
>> b_mlr = inv(X'*X)*X'*y
>> yhat_mlr = X*b_mlr;
```

Linear Algebra



Linear Algebra

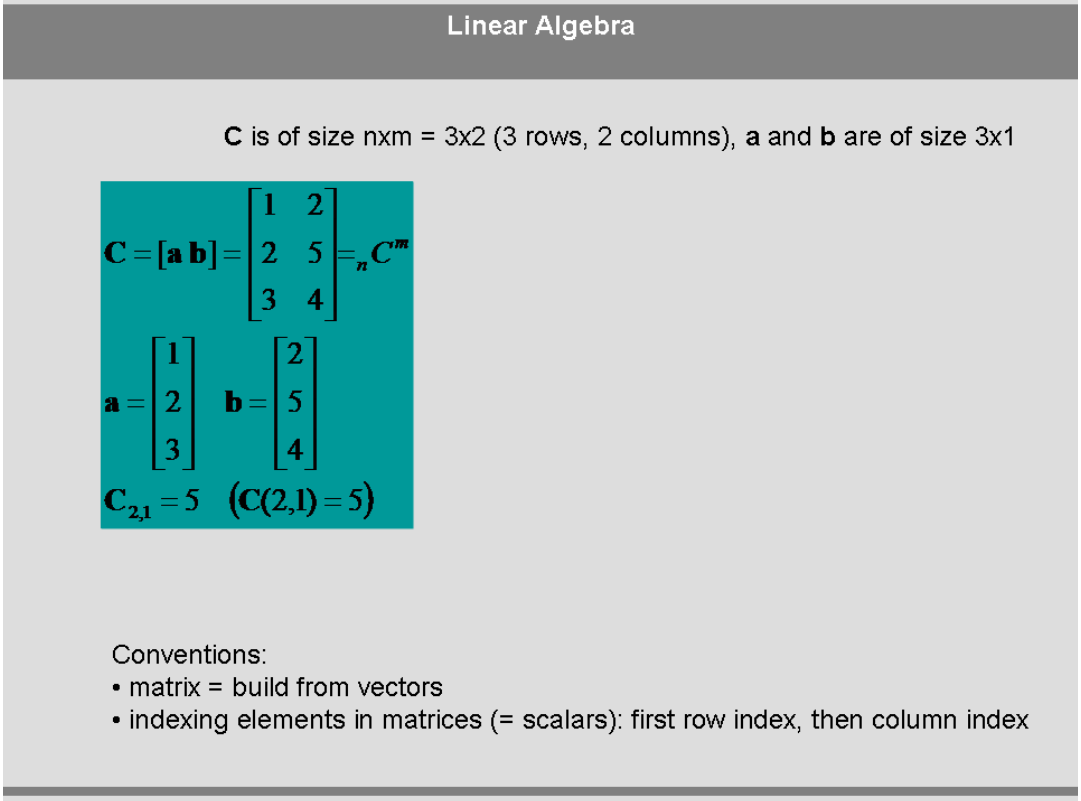
a = scalar, **b** = (row) vector, **c** = (column) vector, **D** = matrix

$$a = 1$$

$$\mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \mathbf{c} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \quad \mathbf{c} = \mathbf{b}^T \quad (\mathbf{c} = \mathbf{b}')$$

$$\mathbf{D} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 5 & 6 & 8 \end{bmatrix} \quad \mathbf{D}^T = \begin{bmatrix} 1 & 2 & 5 \\ 2 & 3 & 6 \\ 3 & 4 & 8 \end{bmatrix}$$

Conventions:
- vectors and matrices in bold, matrices as capitals
- vectors are column vectors
- ‘ or T = transpose = flip rows and columns



Linear Algebra

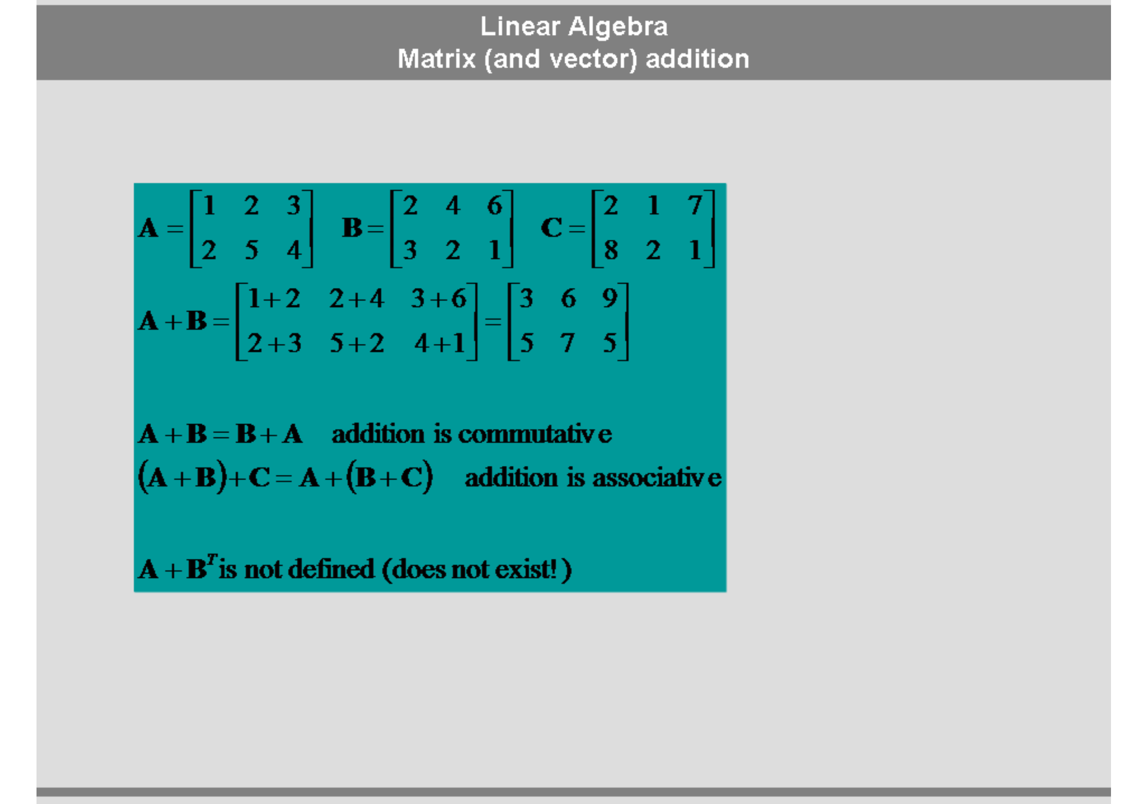
C is of size nxm = 3x2 (3 rows, 2 columns), **a** and **b** are of size 3x1

$$\mathbf{C} = [\mathbf{a}\,\mathbf{b}] = \begin{bmatrix} 1 & 2 \\ 2 & 5 \\ 3 & 4 \end{bmatrix} = {}_n C^m$$

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 2 \\ 5 \\ 4 \end{bmatrix}$$

$$\mathbf{C}_{2,1} = 5 \quad (\mathbf{C}(2,1) = 5)$$

Conventions:
- matrix = build from vectors
- indexing elements in matrices (= scalars): first row index, then column index



Linear Algebra
Matrix (and vector) addition

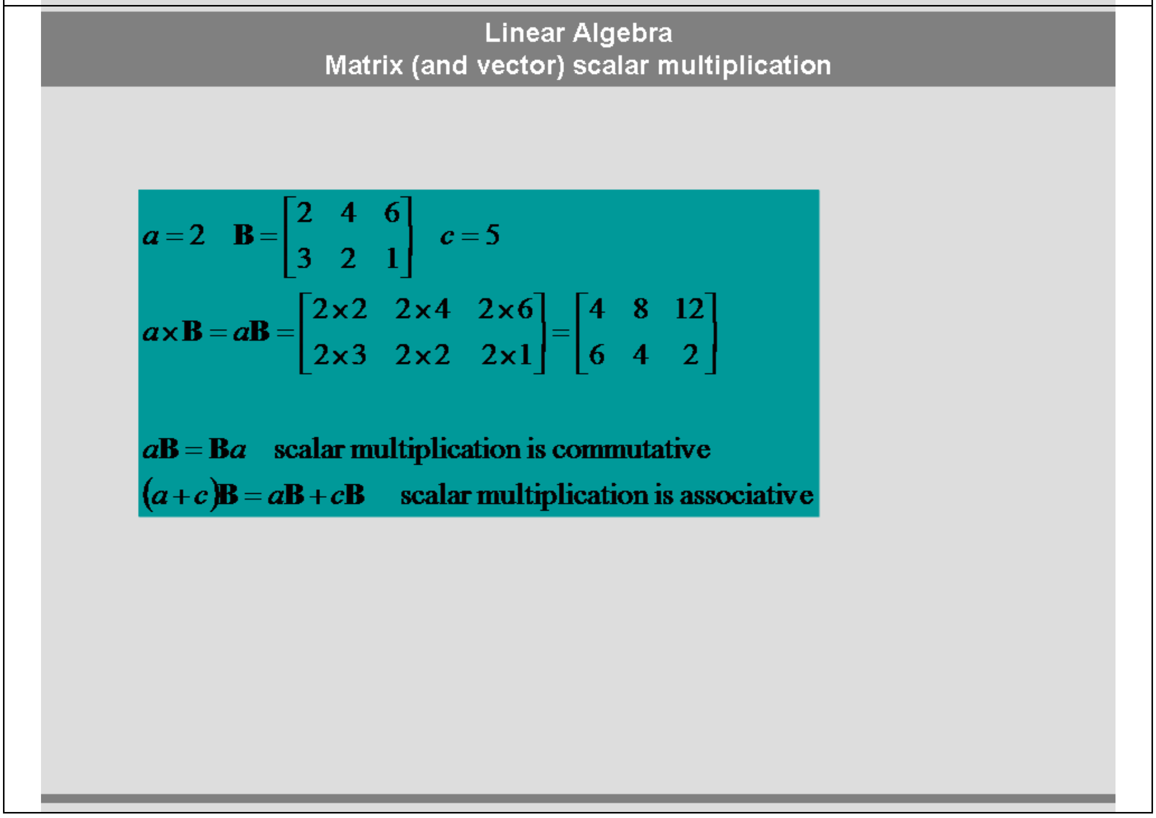
$$\mathbf{A}=\begin{bmatrix}1 & 2 & 3\\ 2 & 5 & 4\end{bmatrix}\quad \mathbf{B}=\begin{bmatrix}2 & 4 & 6\\ 3 & 2 & 1\end{bmatrix}\quad \mathbf{C}=\begin{bmatrix}2 & 1 & 7\\ 8 & 2 & 1\end{bmatrix}$$

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix}1+2 & 2+4 & 3+6\\ 2+3 & 5+2 & 4+1\end{bmatrix}=\begin{bmatrix}3 & 6 & 9\\ 5 & 7 & 5\end{bmatrix}$$

$\mathbf{A}+\mathbf{B}=\mathbf{B}+\mathbf{A}$ addition is commutative

$(\mathbf{A}+\mathbf{B})+\mathbf{C}=\mathbf{A}+(\mathbf{B}+\mathbf{C})$ addition is associative

$\mathbf{A}+\mathbf{B}^T$ is not defined (does not exist!)



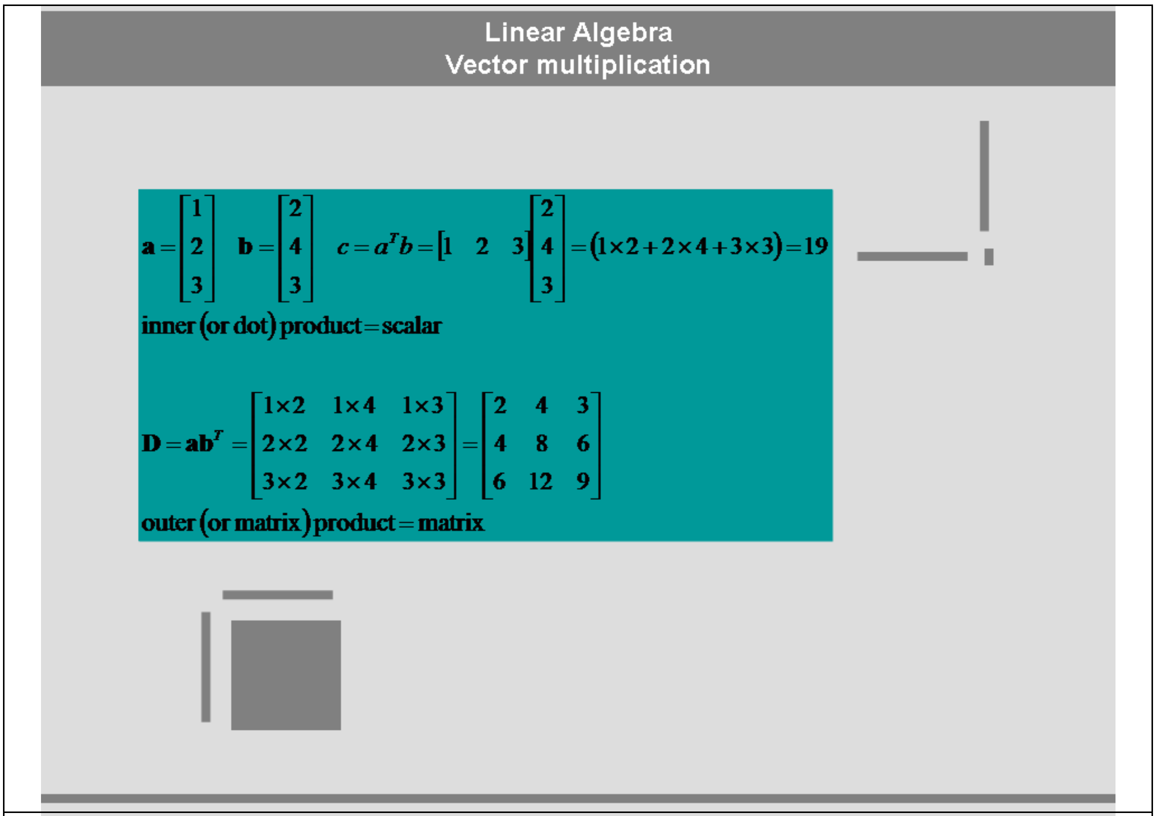
Linear Algebra
Matrix (and vector) scalar multiplication

$$a=2\quad \mathbf{B}=\begin{bmatrix}2 & 4 & 6\\ 3 & 2 & 1\end{bmatrix}\quad c=5$$

$$a\times\mathbf{B}=a\mathbf{B}=\begin{bmatrix}2\times 2 & 2\times 4 & 2\times 6\\ 2\times 3 & 2\times 2 & 2\times 1\end{bmatrix}=\begin{bmatrix}4 & 8 & 12\\ 6 & 4 & 2\end{bmatrix}$$

$a\mathbf{B}=\mathbf{B}a$ scalar multiplication is commutative

$(a+c)\mathbf{B}=a\mathbf{B}+c\mathbf{B}$ scalar multiplication is associative



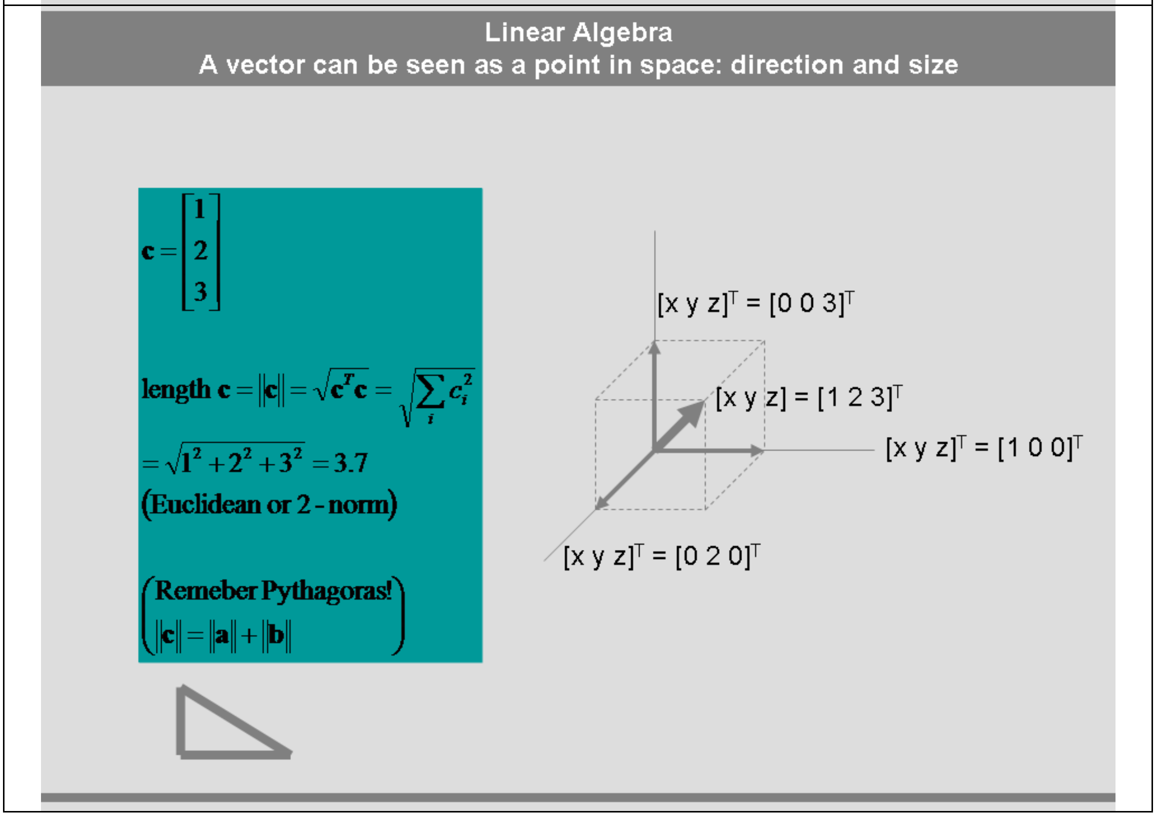
Linear Algebra
Vector multiplication

$$\mathbf{a}=\begin{bmatrix}1\\2\\3\end{bmatrix}\quad \mathbf{b}=\begin{bmatrix}2\\4\\3\end{bmatrix}\quad c=a^Tb=\begin{bmatrix}1 & 2 & 3\end{bmatrix}\begin{bmatrix}2\\4\\3\end{bmatrix}=(1\times2+2\times4+3\times3)=19$$

inner (or dot) product = scalar

$$\mathbf{D}=\mathbf{ab}^T=\begin{bmatrix}1\times2 & 1\times4 & 1\times3\\2\times2 & 2\times4 & 2\times3\\3\times2 & 3\times4 & 3\times3\end{bmatrix}=\begin{bmatrix}2 & 4 & 3\\4 & 8 & 6\\6 & 12 & 9\end{bmatrix}$$

outer (or matrix) product = matrix



Linear Algebra
A vector can be seen as a point in space: direction and size

$$\mathbf{c}=\begin{bmatrix}1\\2\\3\end{bmatrix}$$

$$\text{length } \mathbf{c}=\|\mathbf{c}\|=\sqrt{\mathbf{c}^T\mathbf{c}}=\sqrt{\sum_i c_i^2}$$

$$=\sqrt{1^2+2^2+3^2}=3.7$$

(Euclidean or 2 - norm)

$$\left(\begin{matrix}\text{Remeber Pythagoras!}\\ \|\mathbf{c}\|=\|\mathbf{a}\|+\|\mathbf{b}\|\end{matrix}\right)$$

$[x\ y\ z]^T = [0\ 0\ 3]^T$
$[x\ y\ z] = [1\ 2\ 3]^T$
$[x\ y\ z]^T = [1\ 0\ 0]^T$
$[x\ y\ z]^T = [0\ 2\ 0]^T$

Linear Algebra
Matrix multiplication

$${}_2\mathbf{A}^2 = \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} \quad {}_2\mathbf{B}^3 = \begin{bmatrix}2 & 4 & 6\\ 3 & 2 & 1\end{bmatrix}$$

$${}_2\mathbf{A}^2\,{}_2\mathbf{B}^3 = \begin{bmatrix}1\times2+2\times3 & 1\times4+2\times2 & 1\times6+2\times1\\ 3\times2+4\times3 & 3\times4+4\times2 & 3\times6+4\times1\end{bmatrix} = \begin{bmatrix}8 & 8 & 8\\ 18 & 20 & 22\end{bmatrix} = {}_2\mathbf{C}^3$$

$\mathbf{D}(\mathbf{A}+\mathbf{B}) = \mathbf{DA}+\mathbf{DB}$ **multiplication is distributive**

$(\mathbf{DA})\mathbf{B} = \mathbf{D}(\mathbf{AB})$ **multiplication is associative**

$\mathbf{AB} \neq \mathbf{BA}$ **multiplication is generally NOT commutative**

$$(\mathbf{A}^T)^T = \mathbf{A} \quad (\mathbf{A}+\mathbf{B})^T = \mathbf{A}^T+\mathbf{B}^T \quad (\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

Linear Algebra
Orthogonal and orthonormal vectors

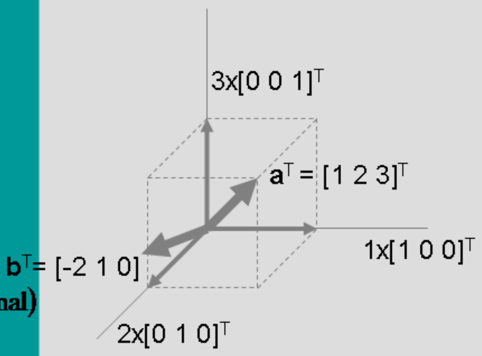
$\mathbf{a}^T = [1 \;\; 2 \;\; 3] \quad \mathbf{b}^T = [-2 \;\; 1 \;\; 0]$

$\mathbf{a}^T\mathbf{b} = 0 \quad \mathbf{a}^T\mathbf{a} = \sqrt{14} \quad \mathbf{b}^T\mathbf{b} = \sqrt{5}$ **a and b orthogonal**

$$\tilde{\mathbf{a}}^T = \frac{\mathbf{a}}{\|\mathbf{a}\|} = \begin{bmatrix}\frac{1}{\sqrt{14}} & \frac{2}{\sqrt{14}} & \frac{3}{\sqrt{14}}\end{bmatrix} \quad (\|\tilde{\mathbf{a}}\| = 1)$$

$$\tilde{\mathbf{b}}^T = \frac{\mathbf{b}}{\|\mathbf{b}\|} = \begin{bmatrix}\frac{-2}{\sqrt{5}} & \frac{1}{\sqrt{5}} & 0\end{bmatrix} \quad (\|\tilde{\mathbf{b}}\| = 1)$$

$\tilde{\mathbf{a}}^T\tilde{\mathbf{b}} = 0 \quad \tilde{\mathbf{a}}^T\tilde{\mathbf{a}} = \tilde{\mathbf{b}}^T\tilde{\mathbf{b}} = 1$ **$\tilde{\mathbf{a}}$ and $\tilde{\mathbf{b}}$ orthonormal**

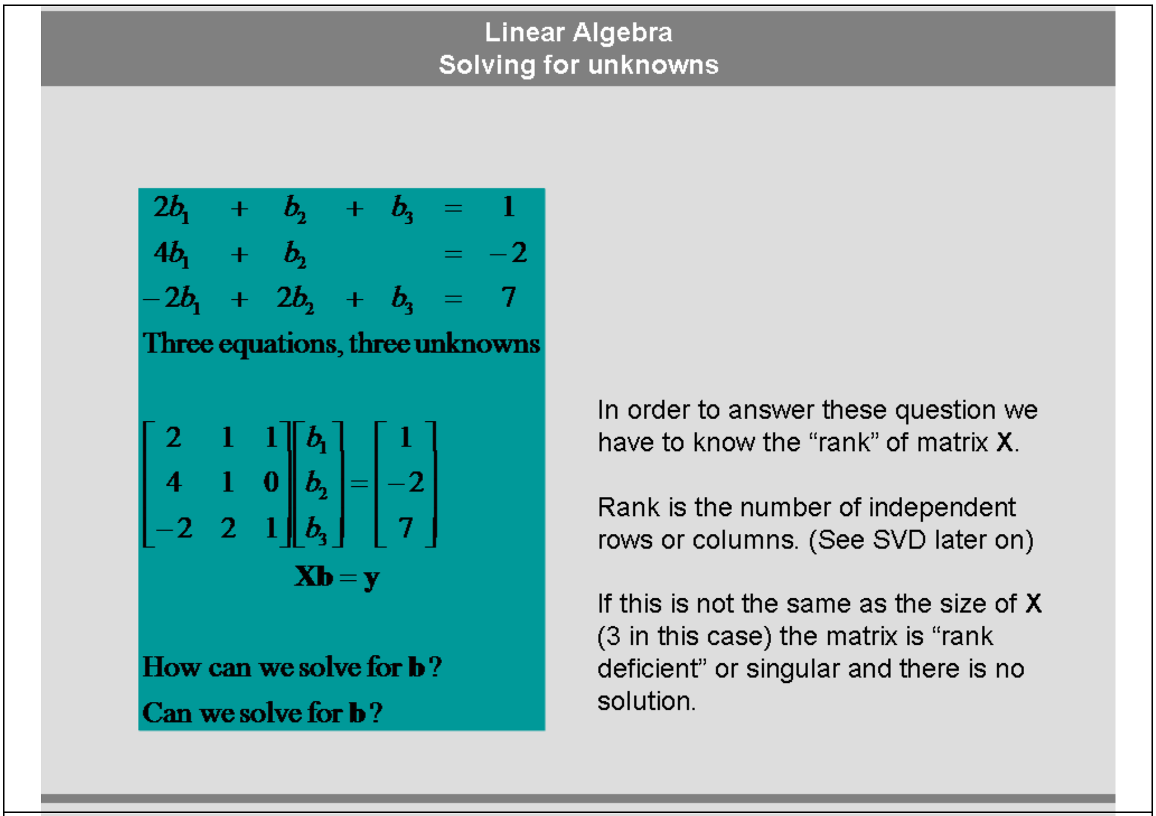


$$\mathbf{I} = \begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix} \quad \text{identity matrix (diagonal, orthogonal)}$$

$\mathbf{a}^T\mathbf{I} = \mathbf{a}^T \quad \mathbf{Ia} = \mathbf{a}$

$$\left(\mathbf{D} = \begin{bmatrix}5 & 0 & 0 & 0\\ 0 & 3 & 0 & 0\\ 0 & 0 & 1 & 0\end{bmatrix} \quad \text{diagonal, orthogonal}\right)$$

$d_{i,j} \neq 0$ iff I = j



Linear Algebra
Solving for unknowns

$$\begin{array}{rcrcrcr} 2b_1 & + & b_2 & + & b_3 & = & 1 \\ 4b_1 & + & b_2 & & & = & -2 \\ -2b_1 & + & 2b_2 & + & b_3 & = & 7 \end{array}$$

Three equations, three unknowns

$$\begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 0 \\ -2 & 2 & 1 \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}=\begin{bmatrix} 1 \\ -2 \\ 7 \end{bmatrix}$$

$$\mathbf{Xb}=\mathbf{y}$$

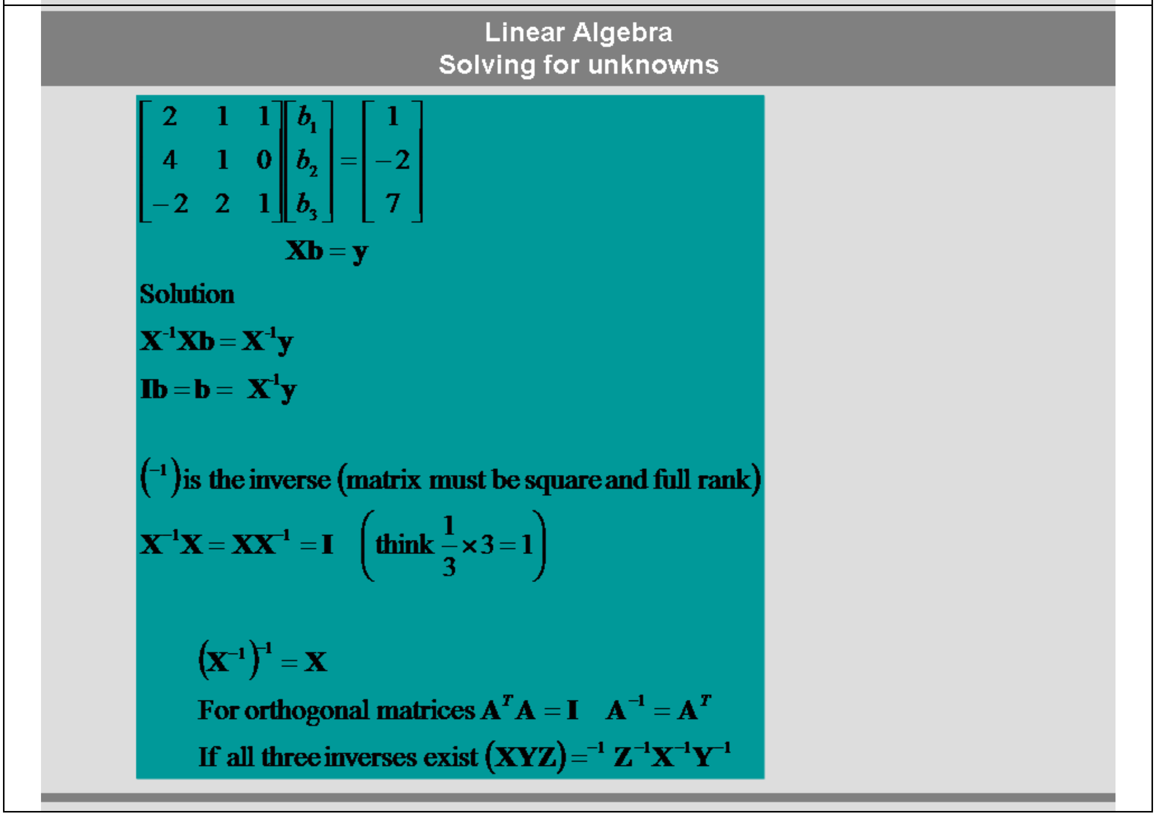
How can we solve for $\mathbf{b}$?

Can we solve for $\mathbf{b}$?

In order to answer these question we have to know the “rank” of matrix **X**.

Rank is the number of independent rows or columns. (See SVD later on)

If this is not the same as the size of **X** (3 in this case) the matrix is “rank deficient” or singular and there is no solution.



Linear Algebra
Solving for unknowns

$$\begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 0 \\ -2 & 2 & 1 \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}=\begin{bmatrix} 1 \\ -2 \\ 7 \end{bmatrix}$$

$$\mathbf{Xb}=\mathbf{y}$$

Solution

$$\mathbf{X}^{-1}\mathbf{Xb}=\mathbf{X}^{-1}\mathbf{y}$$

$$\mathbf{Ib}=\mathbf{b}=\ \mathbf{X}^{-1}\mathbf{y}$$

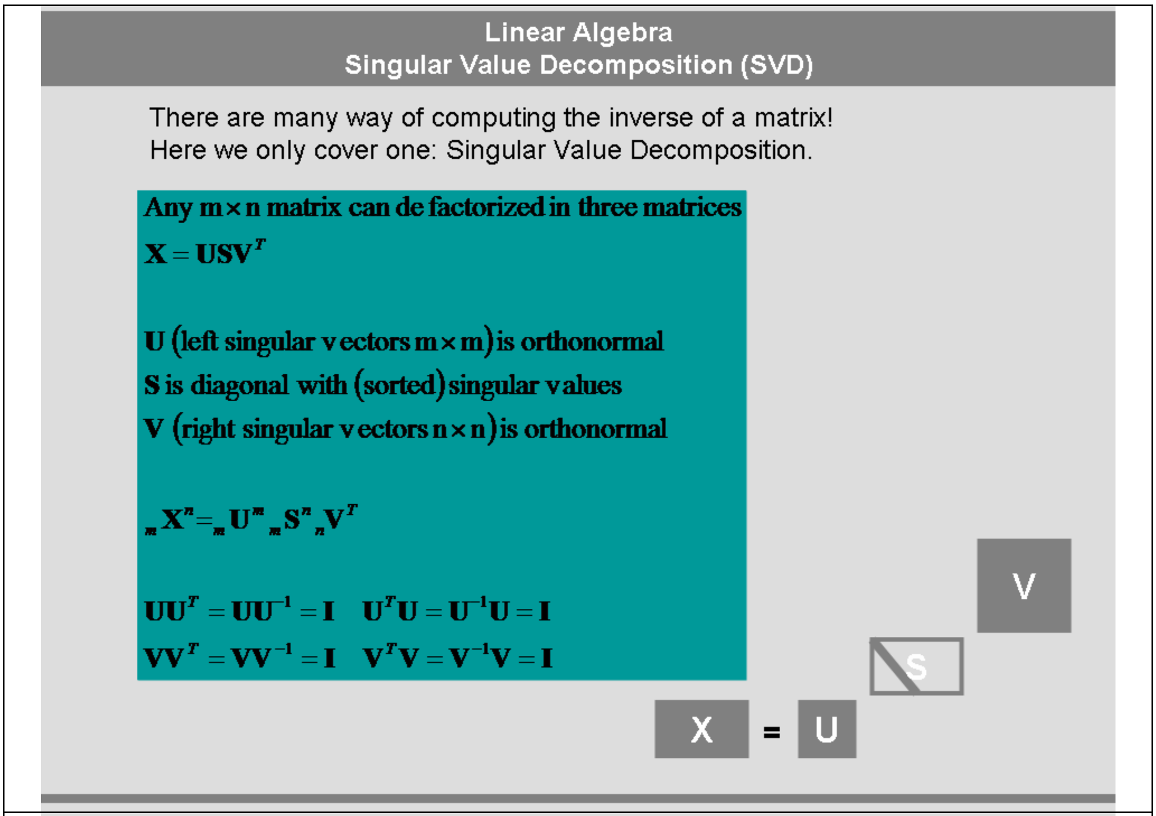
$\left(^{-1}\right)$ is the inverse (matrix must be square and full rank)

$$\mathbf{X}^{-1}\mathbf{X}=\mathbf{XX}^{-1}=\mathbf{I}\quad\left(\text{think }\frac{1}{3}\times 3=1\right)$$

$$\left(\mathbf{X}^{-1}\right)^{-1}=\mathbf{X}$$

For orthogonal matrices $\mathbf{A}^T\mathbf{A}=\mathbf{I}\quad \mathbf{A}^{-1}=\mathbf{A}^T$

If all three inverses exist $(\mathbf{XYZ})=^{-1}\mathbf{Z}^{-1}\mathbf{X}^{-1}\mathbf{Y}^{-1}$



Linear Algebra
Singular Value Decomposition (SVD)

There are many way of computing the inverse of a matrix!
Here we only cover one: Singular Value Decomposition.

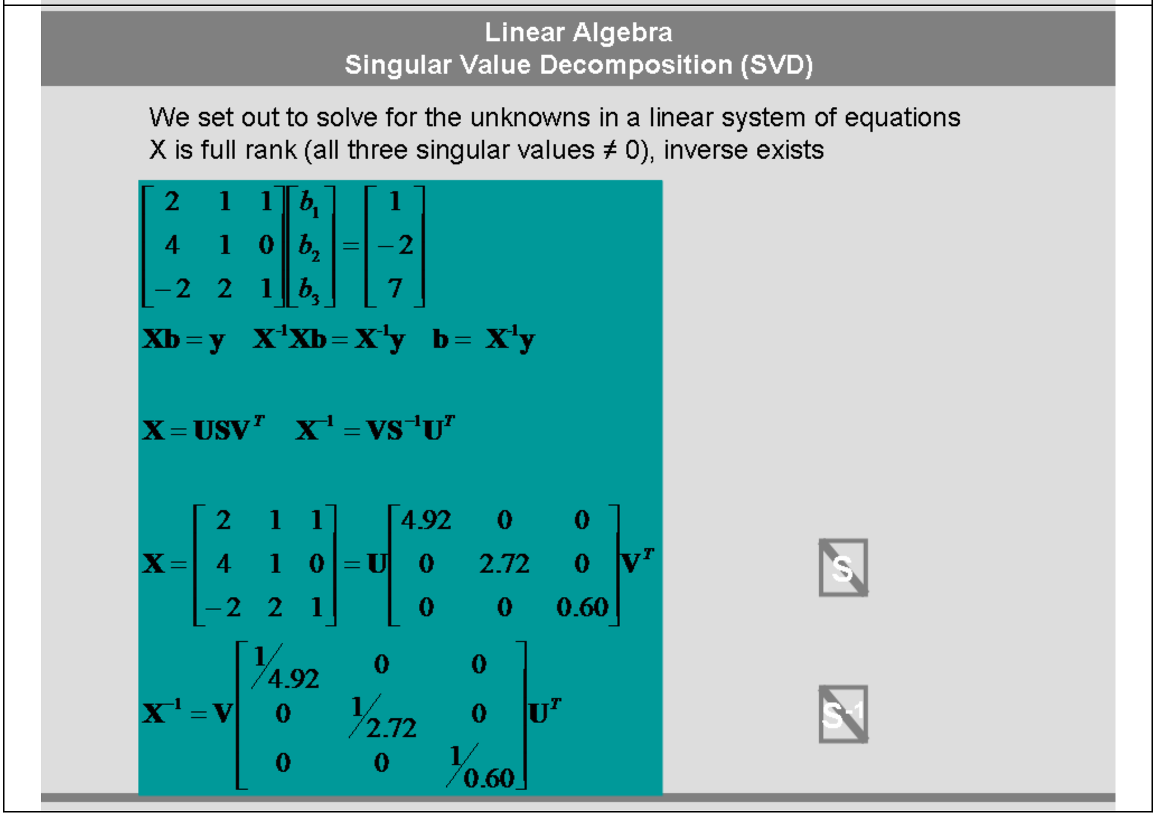
Any m × n matrix can de factorized in three matrices

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T$$

U (left singular vectors m × m) is orthonormal
S is diagonal with (sorted) singular values
V (right singular vectors n × n) is orthonormal

$${}_m\mathbf{X}^n = {}_m\mathbf{U}^m\,{}_m\mathbf{S}^n\,{}_n\mathbf{V}^T$$

$$\mathbf{U}\mathbf{U}^T = \mathbf{U}\mathbf{U}^{-1} = \mathbf{I} \quad \mathbf{U}^T\mathbf{U} = \mathbf{U}^{-1}\mathbf{U} = \mathbf{I}$$
$$\mathbf{V}\mathbf{V}^T = \mathbf{V}\mathbf{V}^{-1} = \mathbf{I} \quad \mathbf{V}^T\mathbf{V} = \mathbf{V}^{-1}\mathbf{V} = \mathbf{I}$$



Linear Algebra
Singular Value Decomposition (SVD)

We set out to solve for the unknowns in a linear system of equations
X is full rank (all three singular values ≠ 0), inverse exists

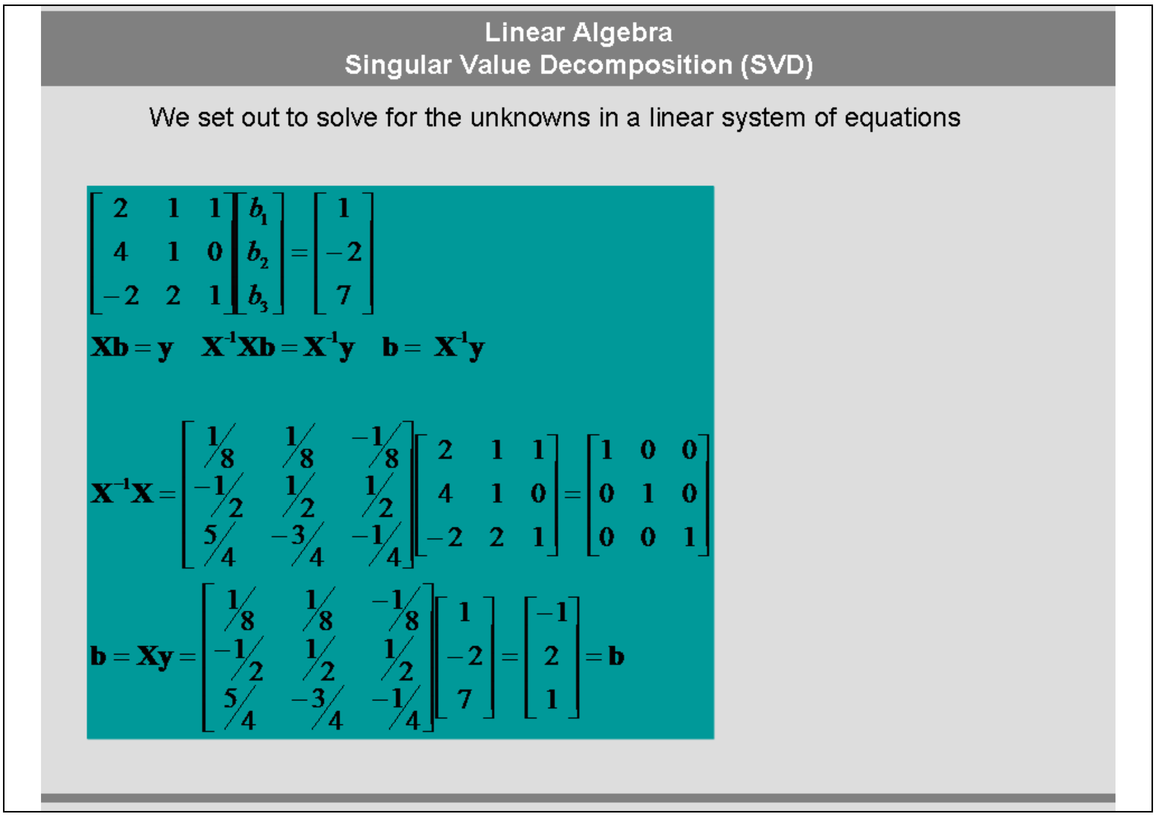
$$\begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 0 \\ -2 & 2 & 1 \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 7 \end{bmatrix}$$

$$\mathbf{X}\mathbf{b} = \mathbf{y} \quad \mathbf{X}^{-1}\mathbf{X}\mathbf{b} = \mathbf{X}^{-1}\mathbf{y} \quad \mathbf{b} = \mathbf{X}^{-1}\mathbf{y}$$

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T \quad \mathbf{X}^{-1} = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}^T$$

$$\mathbf{X} = \begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 0 \\ -2 & 2 & 1 \end{bmatrix} = \mathbf{U}\begin{bmatrix} 4.92 & 0 & 0 \\ 0 & 2.72 & 0 \\ 0 & 0 & 0.60 \end{bmatrix}\mathbf{V}^T$$

$$\mathbf{X}^{-1} = \mathbf{V}\begin{bmatrix} 1/4.92 & 0 & 0 \\ 0 & 1/2.72 & 0 \\ 0 & 0 & 1/0.60 \end{bmatrix}\mathbf{U}^T$$



Linear Algebra
Singular Value Decomposition (SVD)

We set out to solve for the unknowns in a linear system of equations

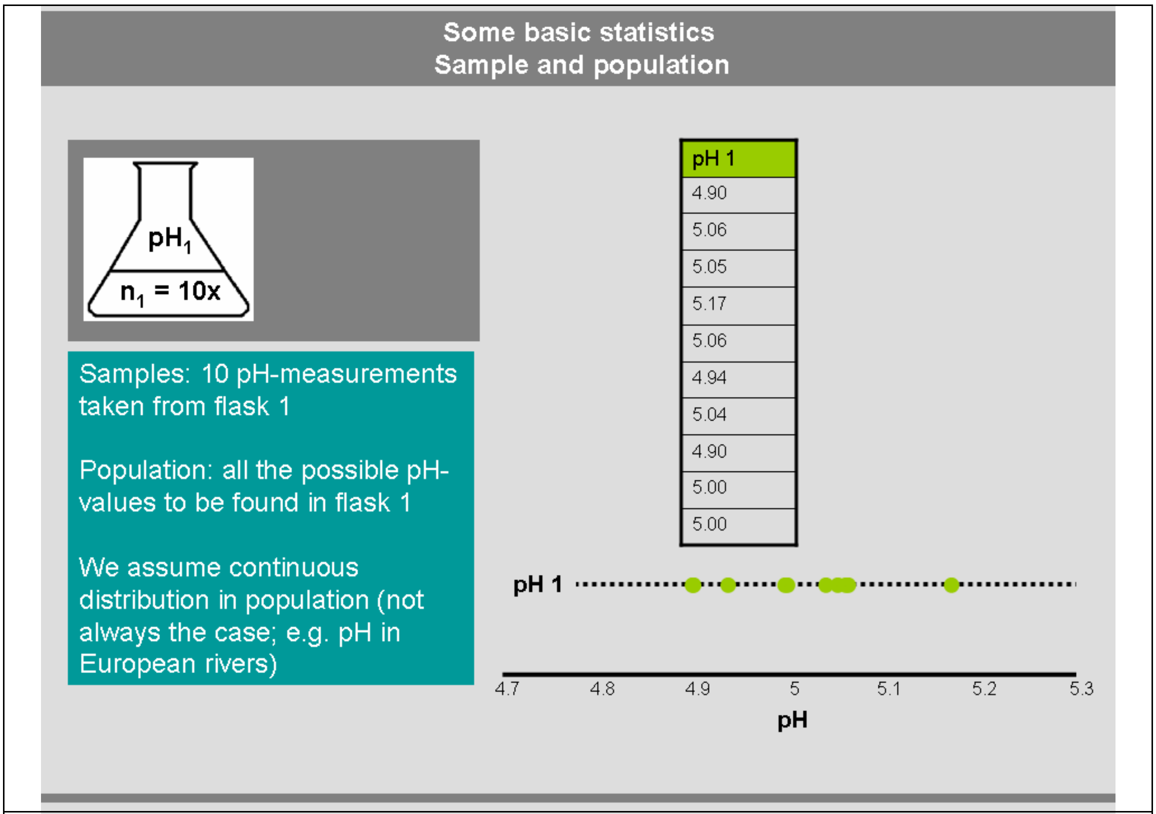
$$\begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 0 \\ -2 & 2 & 1 \end{bmatrix}\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 7 \end{bmatrix}$$

$$\mathbf{Xb} = \mathbf{y} \quad \mathbf{X}^{-1}\mathbf{Xb} = \mathbf{X}^{-1}\mathbf{y} \quad \mathbf{b} = \mathbf{X}^{-1}\mathbf{y}$$

$$\mathbf{X}^{-1}\mathbf{X} = \begin{bmatrix} 1/8 & 1/8 & -1/8 \\ -1/2 & 1/2 & 1/2 \\ 5/4 & -3/4 & -1/4 \end{bmatrix}\begin{bmatrix} 2 & 1 & 1 \\ 4 & 1 & 0 \\ -2 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{b} = \mathbf{Xy} = \begin{bmatrix} 1/8 & 1/8 & -1/8 \\ -1/2 & 1/2 & 1/2 \\ 5/4 & -3/4 & -1/4 \end{bmatrix}\begin{bmatrix} 1 \\ -2 \\ 7 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix} = \mathbf{b}$$

Basic Statistics



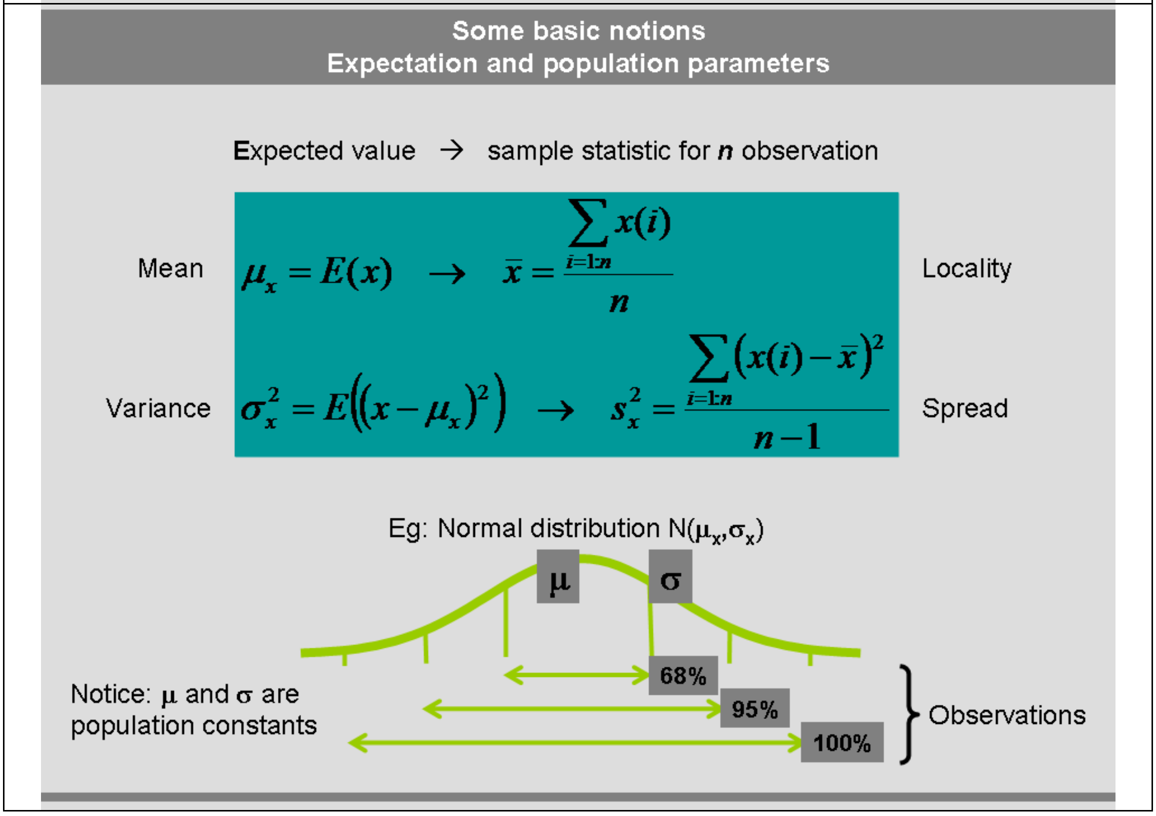
Some basic statistics
Sample and population

Samples: 10 pH-measurements taken from flask 1

Population: all the possible pH-values to be found in flask 1

We assume continuous distribution in population (not always the case; e.g. pH in European rivers)

pH 1
4.90
5.06
5.05
5.17
5.06
4.94
5.04
4.90
5.00
5.00



Some basic notions
Expectation and population parameters

Expected value → sample statistic for *n* observation

Mean $\mu_x = E(x) \rightarrow \bar{x} = \dfrac{\sum_{i=1:n} x(i)}{n}$ Locality

Variance $\sigma_x^2 = E\left((x-\mu_x)^2\right) \rightarrow s_x^2 = \dfrac{\sum_{i=1:n} (x(i)-\bar{x})^2}{n-1}$ Spread

Eg: Normal distribution N(μ_x,σ_x)

Notice: μ and σ are population constants

Some basic notions

$$\bar{x} = \frac{\sum_{i=1:n} x(i)}{n}$$

Mean

$$E(\bar{x}) = \mu$$

$$s_x^2 = \frac{\sum_{i=1:n} \left(x(i) - \bar{x}\right)^2}{n-1}$$

Variance

$$s_x = \sqrt{s_x^2}$$

Standard deviation

$$SE_{\bar{x}} = \frac{s_x}{\sqrt{n}}$$

Standard error
of the mean

$$s_r = \frac{s}{\bar{x}}$$

Relative SD

$$s_r(\%) = 100 s_r$$

RSD in %
(coefficient of variation)

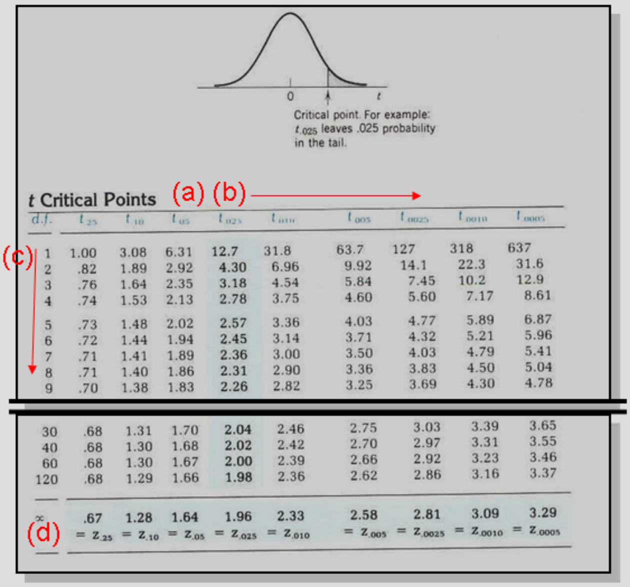
$$\Pr\left(\mu - t_{n-1}SE_{\bar{x}} < \bar{x} < \mu + t_{n-1}SE_{\bar{x}}\right) = 95\%$$

$$\Pr\left(\bar{x} - t_{n-1}SE_{\bar{x}} < \mu < \bar{x} + t_{n-1}SE_{\bar{x}}\right) = 95\%$$

95% confidence interval/level; *n* large or s_x given $t_{n-1} = z_0 = 1.96$

Some basic notions
Critical t-values

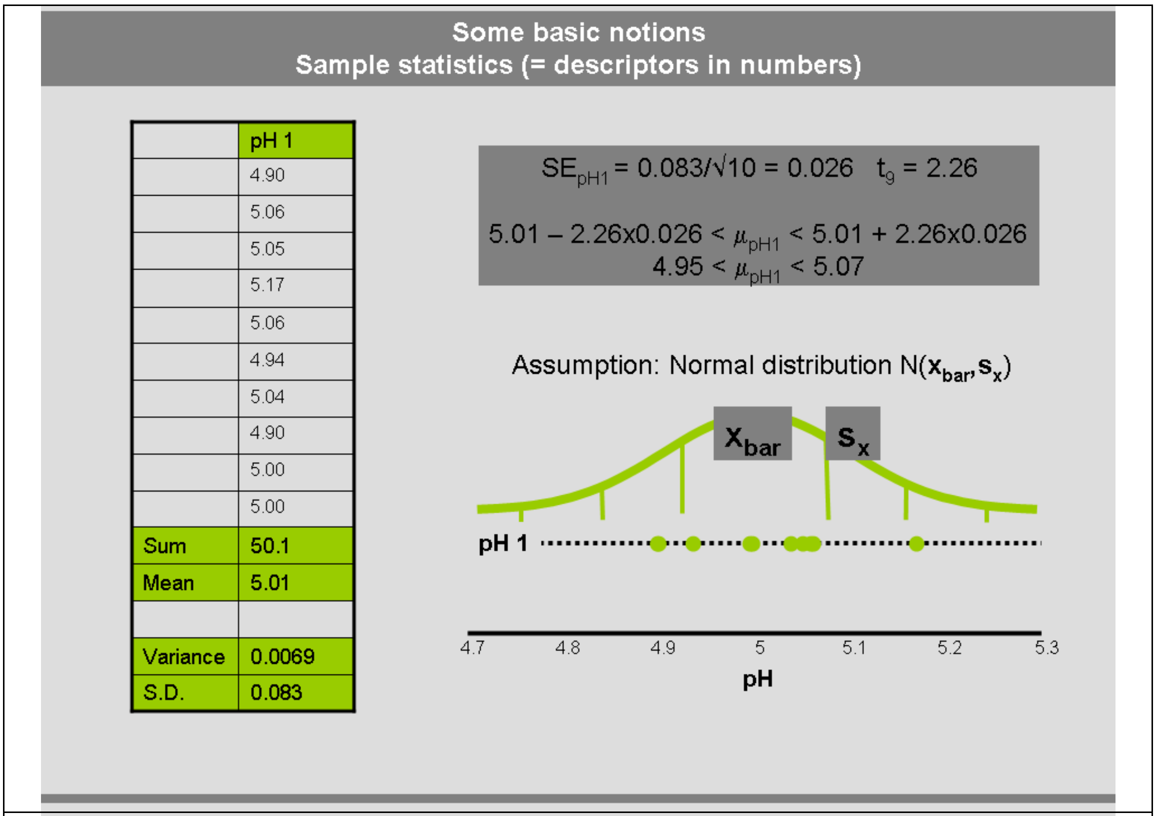
(a) α is users choice
(b) Increasing for α
(c) Decreasing for *n*
(d) *n* large (or σ known)



Critical point. For example: $t_{.025}$ leaves .025 probability in the tail.

t Critical Points

d.f.	$t_{.25}$	$t_{.10}$	$t_{.05}$	$t_{.025}$	$t_{.010}$	$t_{.005}$	$t_{.0025}$	$t_{.0010}$	$t_{.0005}$
1	1.00	3.08	6.31	12.7	31.8	63.7	127	318	637
2	.82	1.89	2.92	4.30	6.96	9.92	14.1	22.3	31.6
3	.76	1.64	2.35	3.18	4.54	5.84	7.45	10.2	12.9
4	.74	1.53	2.13	2.78	3.75	4.60	5.60	7.17	8.61
5	.73	1.48	2.02	2.57	3.36	4.03	4.77	5.89	6.87
6	.72	1.44	1.94	2.45	3.14	3.71	4.32	5.21	5.96
7	.71	1.41	1.89	2.36	3.00	3.50	4.03	4.79	5.41
8	.71	1.40	1.86	2.31	2.90	3.36	3.83	4.50	5.04
9	.70	1.38	1.83	2.26	2.82	3.25	3.69	4.30	4.78
30	.68	1.31	1.70	2.04	2.46	2.75	3.03	3.39	3.65
40	.68	1.30	1.68	2.02	2.42	2.70	2.97	3.31	3.55
60	.68	1.30	1.67	2.00	2.39	2.66	2.92	3.23	3.46
120	.68	1.29	1.66	1.98	2.36	2.62	2.86	3.16	3.37
∞	.67 $= z_{.25}$	1.28 $= z_{.10}$	1.64 $= z_{.05}$	1.96 $= z_{.025}$	2.33 $= z_{.010}$	2.58 $= z_{.005}$	2.81 $= z_{.0025}$	3.09 $= z_{.0010}$	3.29 $= z_{.0005}$



Some basic notions
Sample statistics (= descriptors in numbers)

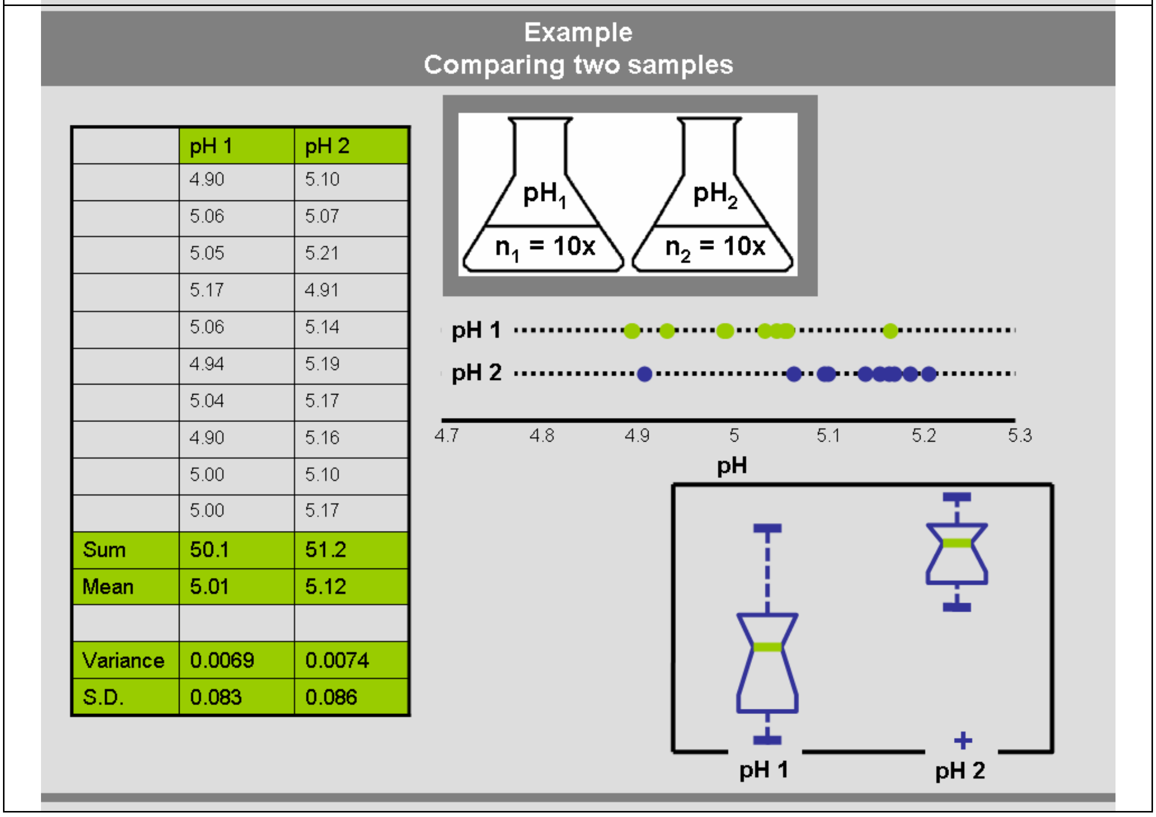
	pH 1
	4.90
	5.06
	5.05
	5.17
	5.06
	4.94
	5.04
	4.90
	5.00
	5.00
Sum	50.1
Mean	5.01
Variance	0.0069
S.D.	0.083

$SE_{pH1} = 0.083/\sqrt{10} = 0.026 \quad t_9 = 2.26$

$5.01 - 2.26x0.026 < \mu_{pH1} < 5.01 + 2.26x0.026$

$4.95 < \mu_{pH1} < 5.07$

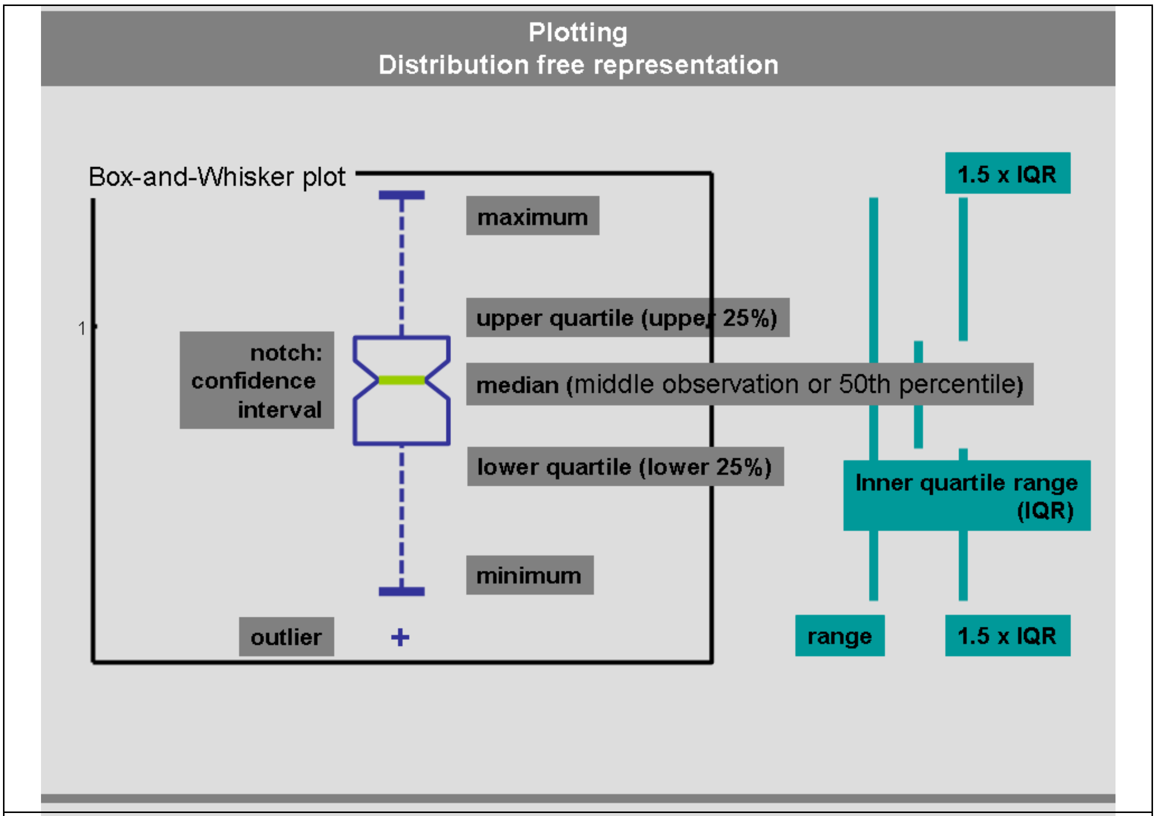
Assumption: Normal distribution $N(\mathbf{x}_{\mathbf{bar}}, \mathbf{s}_{\mathbf{x}})$



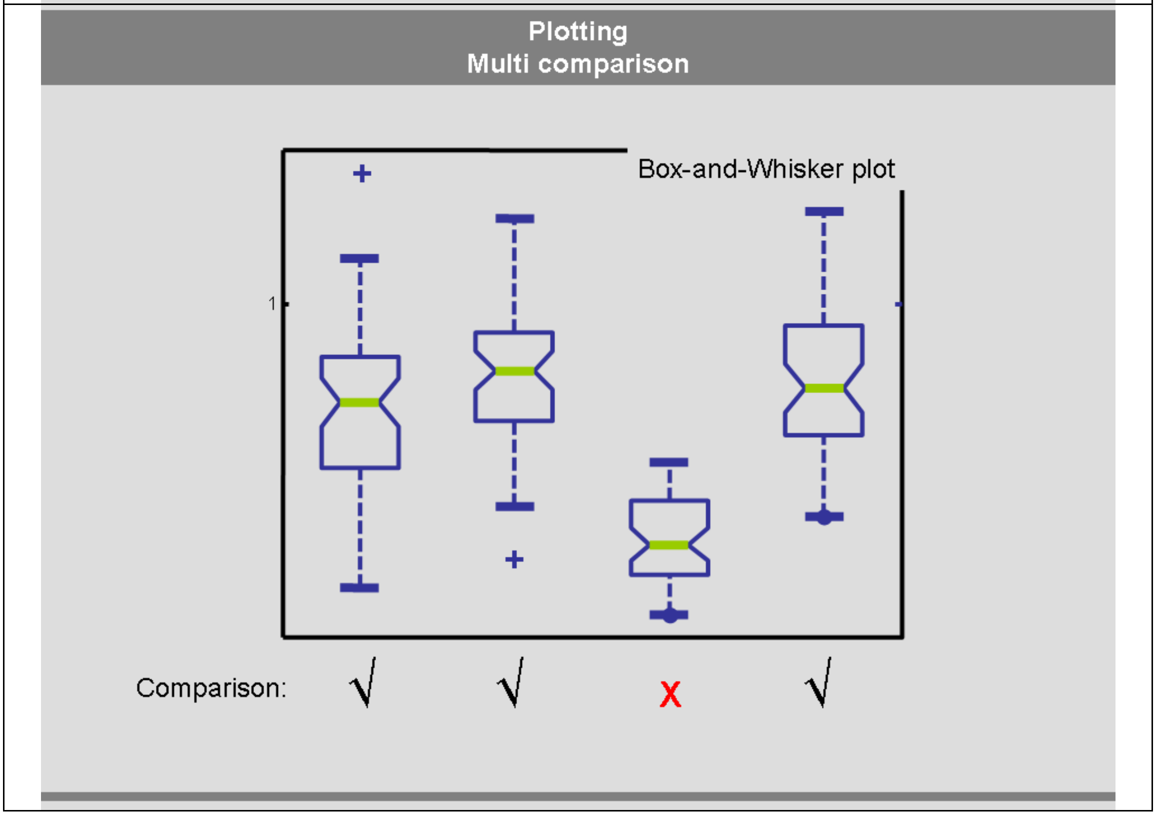
Example
Comparing two samples

	pH 1	pH 2
	4.90	5.10
	5.06	5.07
	5.05	5.21
	5.17	4.91
	5.06	5.14
	4.94	5.19
	5.04	5.17
	4.90	5.16
	5.00	5.10
	5.00	5.17
Sum	50.1	51.2
Mean	5.01	5.12
Variance	0.0069	0.0074
S.D.	0.083	0.086

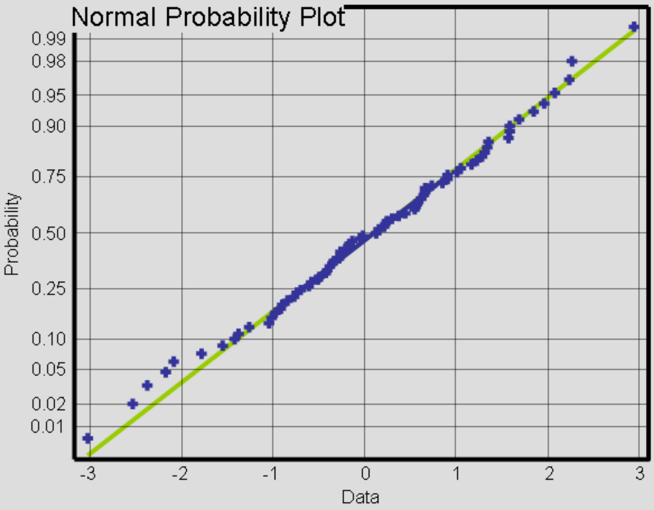
Plotting
Distribution free representation



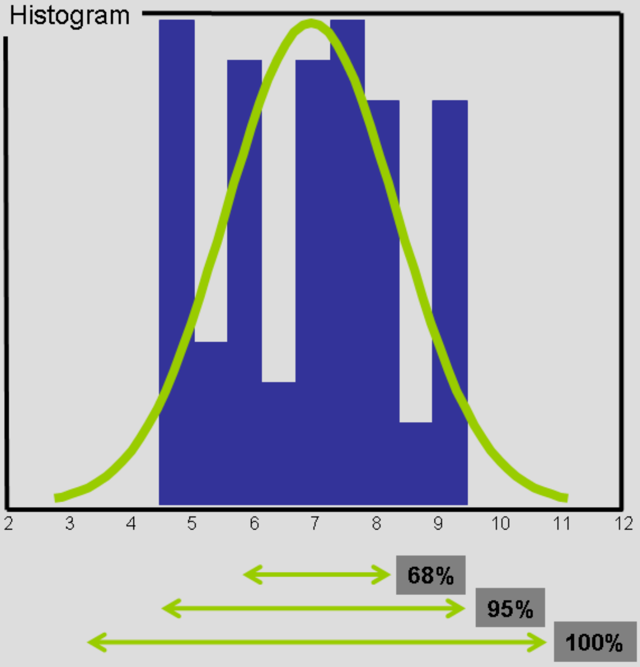
Plotting
Multi comparison

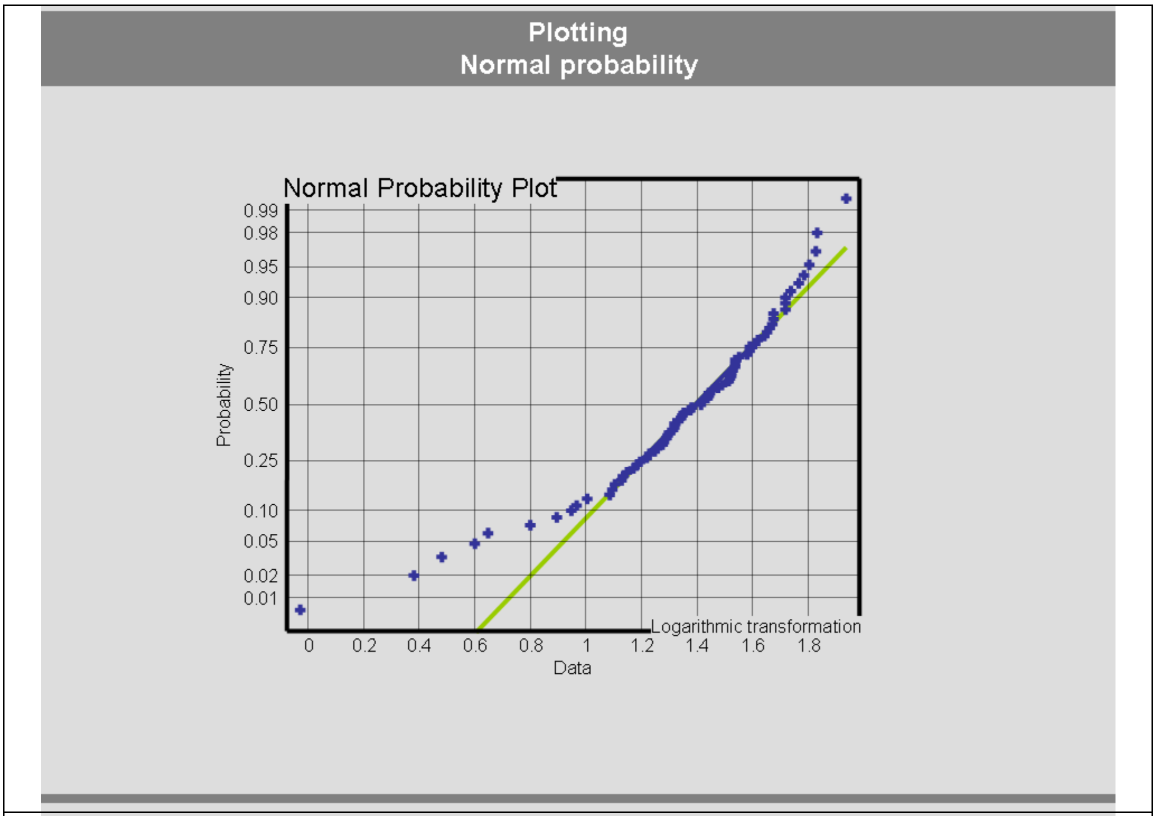


Plotting
Normal probability

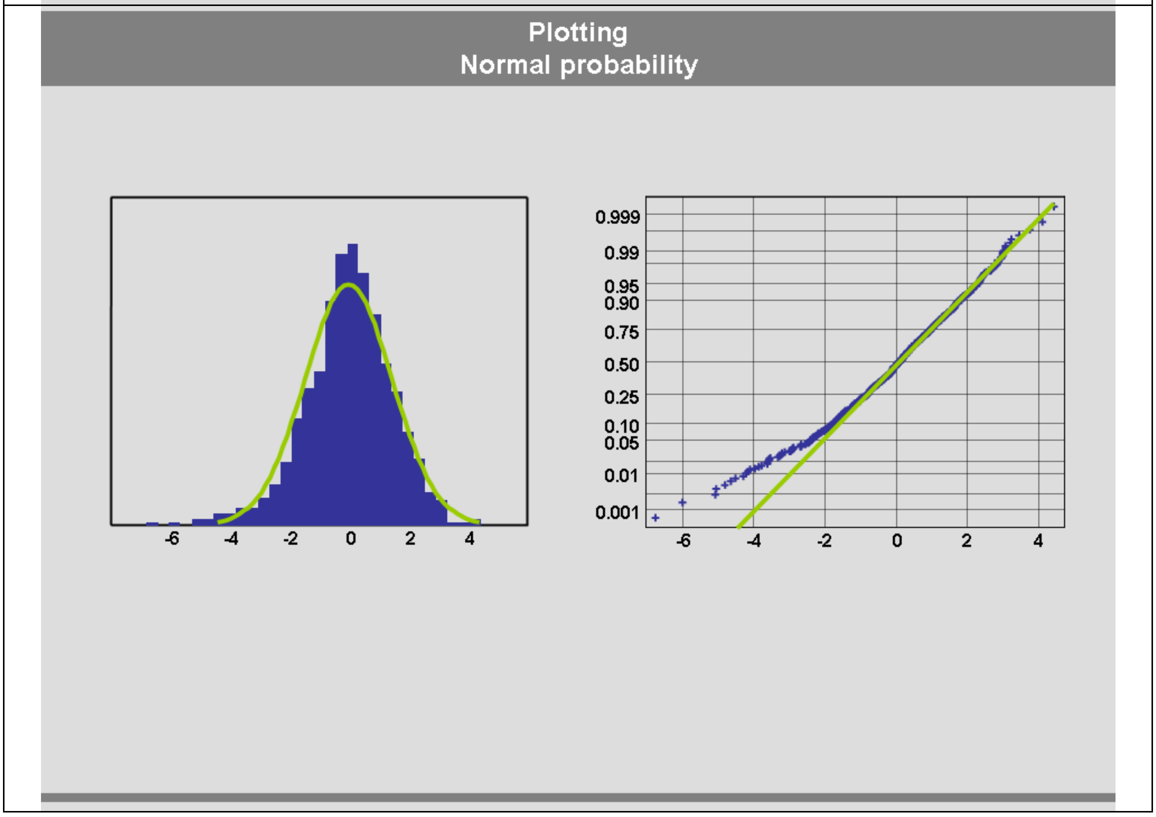


Plotting
Normal distribution

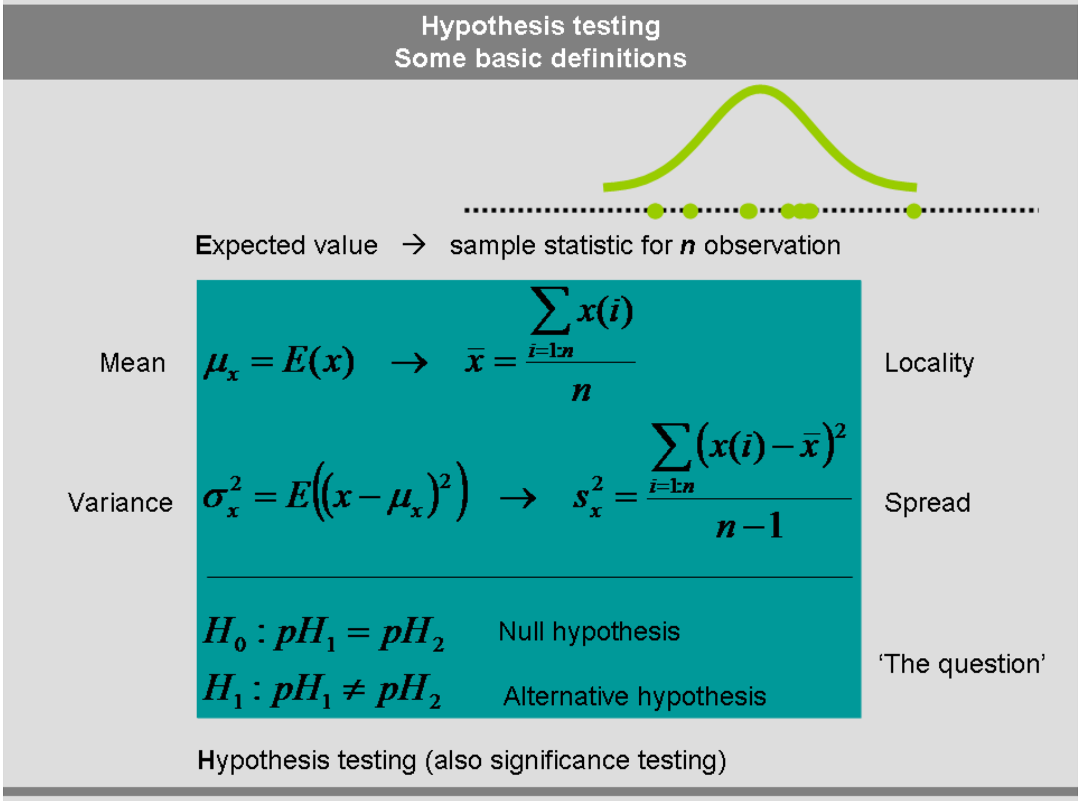




Plotting
Normal probability



Plotting
Normal probability



Hypothesis testing
Some basic definitions

Expected value → sample statistic for ***n*** observation

Mean $$\mu_x = E(x) \rightarrow \bar{x} = \frac{\sum_{i=1:n} x(i)}{n}$$ Locality

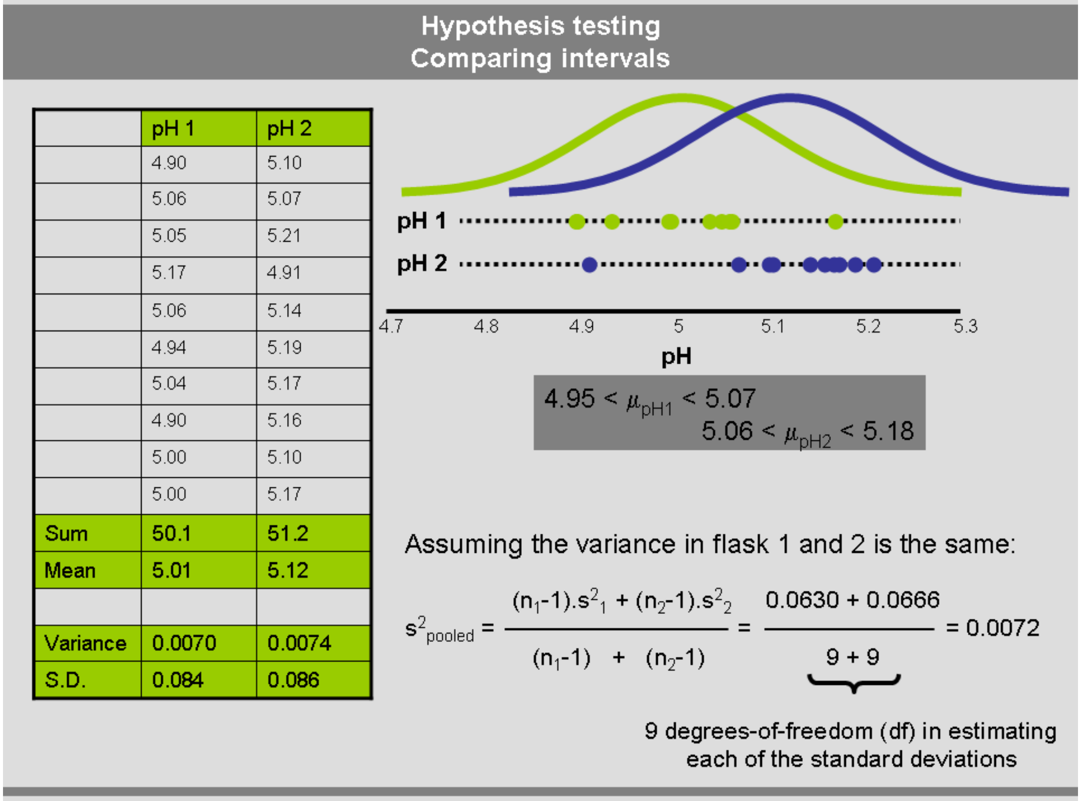
Variance $$\sigma_x^2 = E\left((x-\mu_x)^2\right) \rightarrow s_x^2 = \frac{\sum_{i=1:n} \left(x(i)-\bar{x}\right)^2}{n-1}$$ Spread

$H_0 : pH_1 = pH_2$ Null hypothesis

$H_1 : pH_1 \neq pH_2$ Alternative hypothesis

'The question'

Hypothesis testing (also significance testing)



Hypothesis testing
Comparing intervals

	pH 1	pH 2
	4.90	5.10
	5.06	5.07
	5.05	5.21
	5.17	4.91
	5.06	5.14
	4.94	5.19
	5.04	5.17
	4.90	5.16
	5.00	5.10
	5.00	5.17
Sum	50.1	51.2
Mean	5.01	5.12
Variance	0.0070	0.0074
S.D.	0.084	0.086

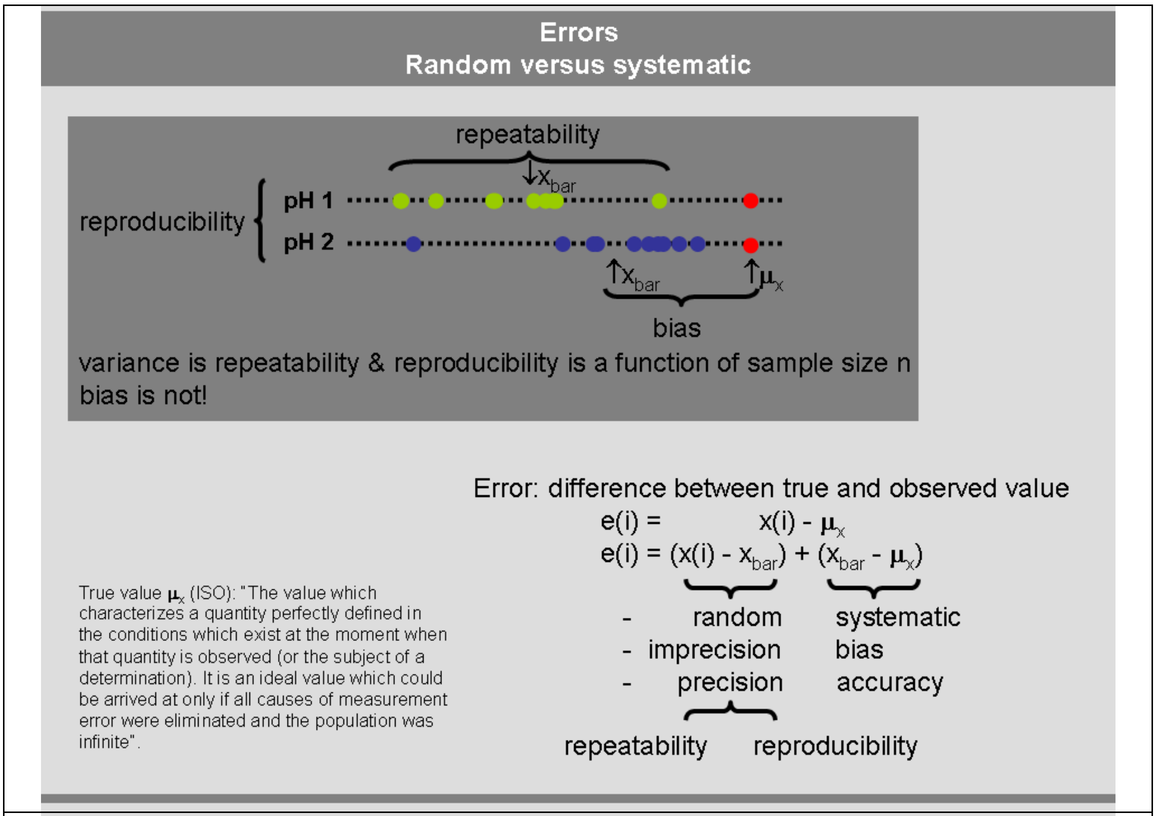
$4.95 < \mu_{pH1} < 5.07$

$5.06 < \mu_{pH2} < 5.18$

Assuming the variance in flask 1 and 2 is the same:

$$s^2_{pooled} = \frac{(n_1-1).s^2_1 + (n_2-1).s^2_2}{(n_1-1) + (n_2-1)} = \frac{0.0630 + 0.0666}{\underbrace{9+9}} = 0.0072$$

9 degrees-of-freedom (df) in estimating each of the standard deviations



Errors
Random versus systematic

variance is repeatability & reproducibility is a function of sample size n
bias is not!

Error: difference between true and observed value

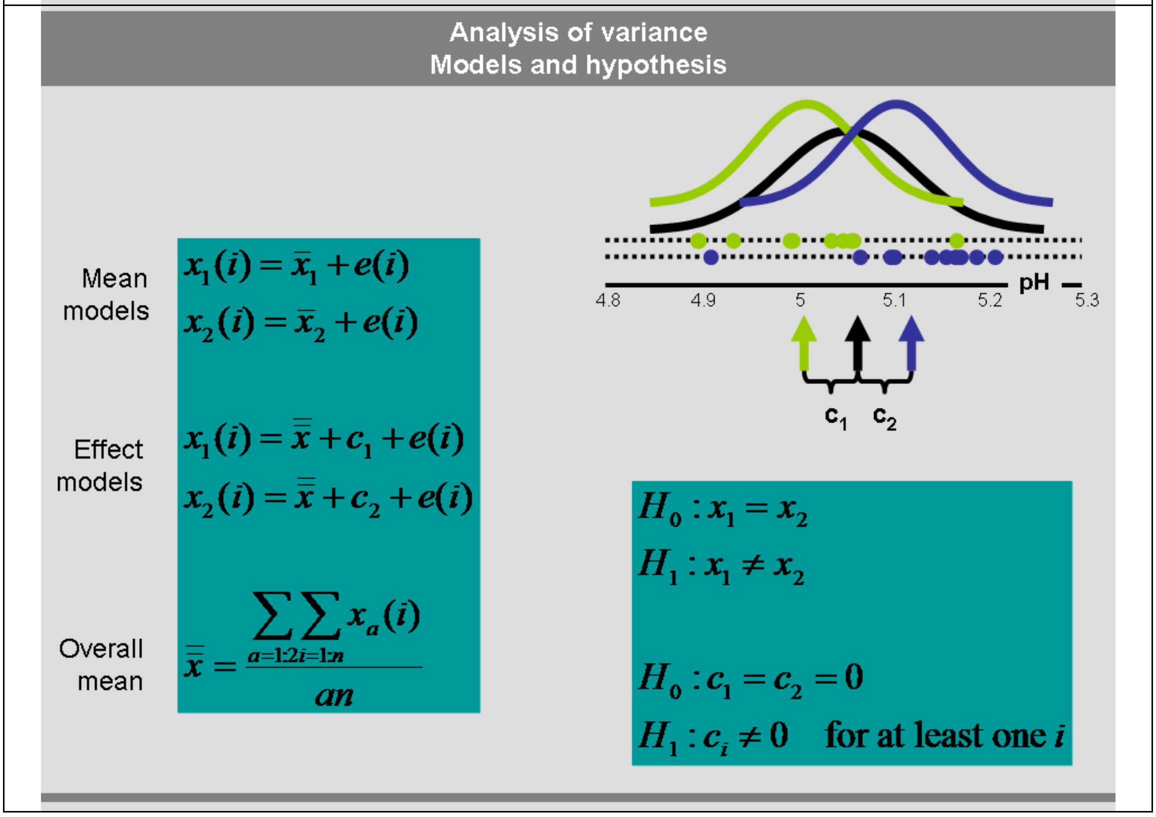
$$e(i) = x(i) - \mu_x$$

$$e(i) = \underbrace{(x(i) - x_{bar})}_{\text{random}} + \underbrace{(x_{bar} - \mu_x)}_{\text{systematic}}$$

- random — systematic
- imprecision — bias
- precision — accuracy

$\overbrace{\text{repeatability} \quad \text{reproducibility}}$

True value μ_x (ISO): "The value which characterizes a quantity perfectly defined in the conditions which exist at the moment when that quantity is observed (or the subject of a determination). It is an ideal value which could be arrived at only if all causes of measurement error were eliminated and the population was infinite".



Analysis of variance
Models and hypothesis

Mean models

$$x_1(i) = \bar{x}_1 + e(i)$$
$$x_2(i) = \bar{x}_2 + e(i)$$

Effect models

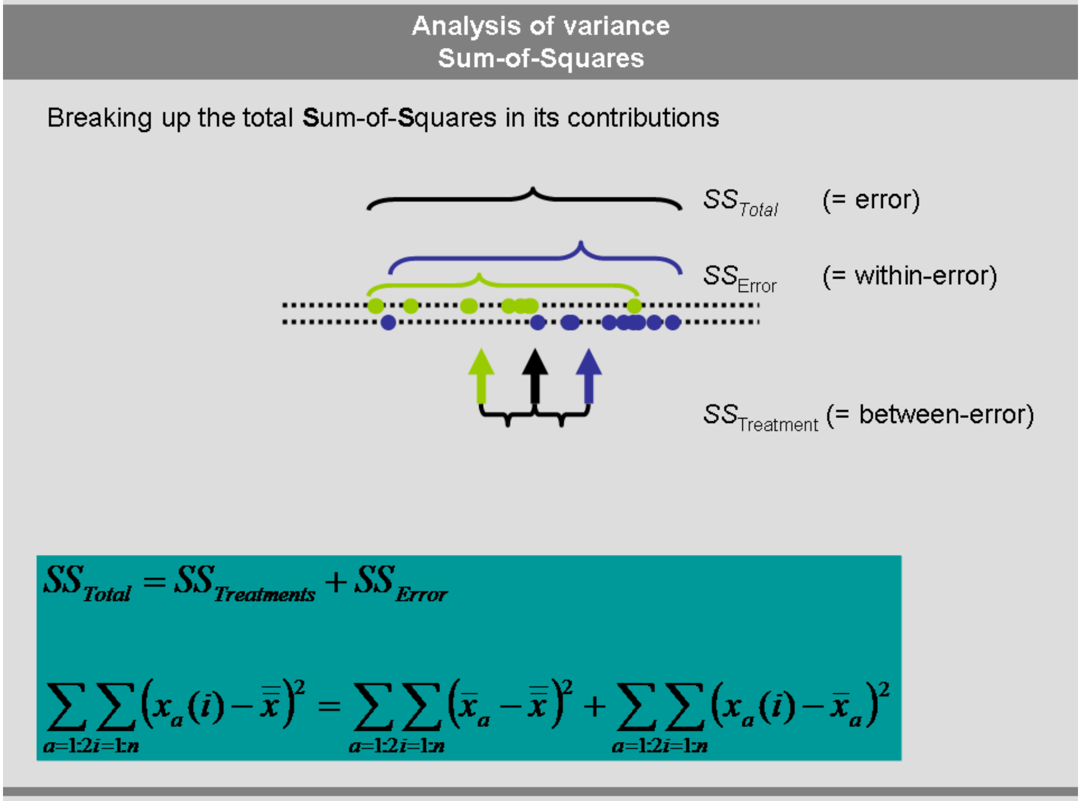
$$x_1(i) = \bar{\bar{x}} + c_1 + e(i)$$
$$x_2(i) = \bar{\bar{x}} + c_2 + e(i)$$

Overall mean

$$\bar{\bar{x}} = \frac{\sum_{a=1:2}\sum_{i=1:n} x_a(i)}{an}$$

$$H_0 : x_1 = x_2$$
$$H_1 : x_1 \neq x_2$$

$$H_0 : c_1 = c_2 = 0$$
$$H_1 : c_i \neq 0 \quad \text{for at least one } i$$



Analysis of variance
Sum-of-Squares

Breaking up the total **S**um-of-**S**quares in its contributions

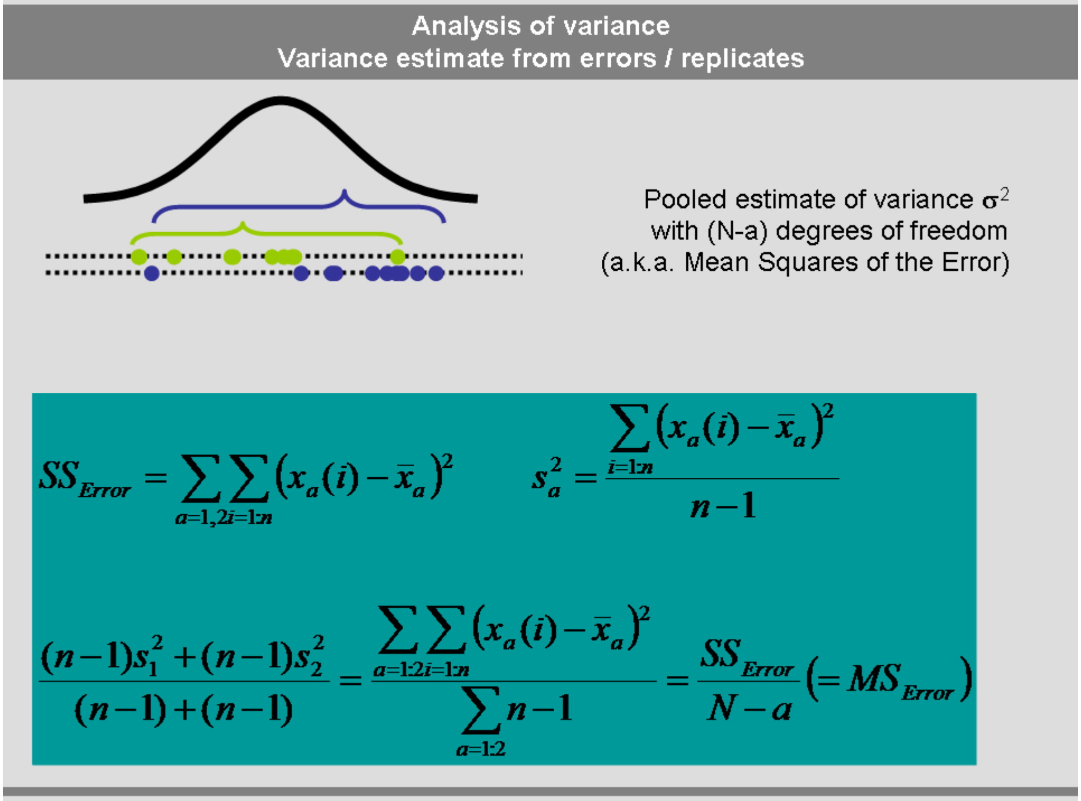
SS_{Total} (= error)

SS_{Error} (= within-error)

$SS_{Treatment}$ (= between-error)

$$SS_{Total} = SS_{Treatments} + SS_{Error}$$

$$\sum_{a=1:2}\sum_{i=1:n}\left(x_a(i)-\bar{\bar{x}}\right)^2 = \sum_{a=1:2}\sum_{i=1:n}\left(\bar{x}_a-\bar{\bar{x}}\right)^2 + \sum_{a=1:2}\sum_{i=1:n}\left(x_a(i)-\bar{x}_a\right)^2$$



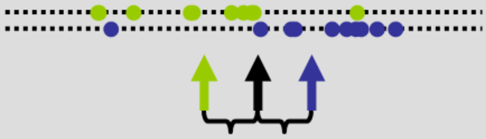
Analysis of variance
Variance estimate from errors / replicates

Pooled estimate of variance σ^2
with (N-a) degrees of freedom
(a.k.a. Mean Squares of the Error)

$$SS_{Error} = \sum_{a=1,2}\sum_{i=1:n}\left(x_a(i)-\bar{x}_a\right)^2 \qquad s_a^2 = \frac{\sum_{i=1:n}\left(x_a(i)-\bar{x}_a\right)^2}{n-1}$$

$$\frac{(n-1)s_1^2+(n-1)s_2^2}{(n-1)+(n-1)} = \frac{\sum_{a=1:2}\sum_{i=1:n}\left(x_a(i)-\bar{x}_a\right)^2}{\sum_{a=1:2} n-1} = \frac{SS_{Error}}{N-a}\left(=MS_{Error}\right)$$

Analysis of variance
Variance estimate from treatment

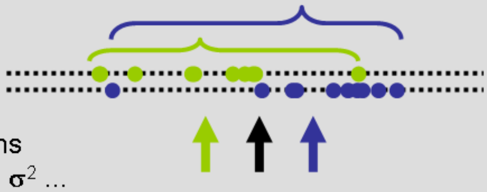


Variance estimate σ^2 from treatment averages
with (a-1) degrees of freedom

$$SS_{Treatment} = \sum_{a=1:2}\sum_{i=1:n}\left(\bar{x}_a - \bar{\bar{x}}\right)^2 = n\sum_{a=1:2}\left(\bar{x}_a - \bar{\bar{x}}\right)^2$$

$$\frac{n\sum_{a=1:2}\left(\bar{x}_a - \bar{\bar{x}}\right)^2}{a-1} = \frac{SS_{Treatment}}{a-1}\left(= MS_{Treatment}\right)$$

Analysis of variance
The test statistic



If there is no difference in the treatment means
we have two estimates of the model variance σ^2 …

$$MS_{Treatment} = \frac{SS_{Treatment}}{a-1} = \sigma^2 = \frac{SS_{Error}}{N-a} = MS_{Error}$$

The answer is derived from comparing experimental with tabulated F-values

… and from these two estimates
we derive the test statistic (F-test)

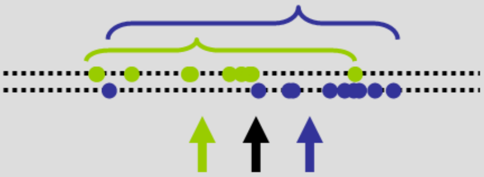
$$F_0 = \frac{SS_{Treatment}/(a-1)}{SS_{Error}/(N-a)} = \frac{MS_{Treatment}}{MS_{Error}}$$

'The question'

$$H_0 : F_0 < F_{\alpha,a-1,N-a}$$

$$H_1 : F_0 > F_{\alpha,a-1,N-a}$$

Example
The test statistic



‘F’-test to compare the two means:

$$F_0 = \frac{SS_{Treatment}/(a-1)}{SS_{Error}/(N-a)} = \frac{0.060/(2-1)}{0.130/(20-2)}$$

$$= 8.39 > F_{\alpha=0.05/1,18} = 4.41$$

Classical: ‘H_0 rejected at α = 5% level’

‘F’-distribution lookup table:

α (%)	25	10	5.0	2.5	1.0
F	1.41	3.01	4.41	5.98	8.29

Better: ‘p < 1%’

Even better: ‘Assuming these two treatments had the same pH, the probability of finding these particular 2 x 10 numbers is smaller than 1 percent!’

Analysis of variance
Critical F-values

(a) $\qquad SS_{Treatment}/(a\text{-}1)$

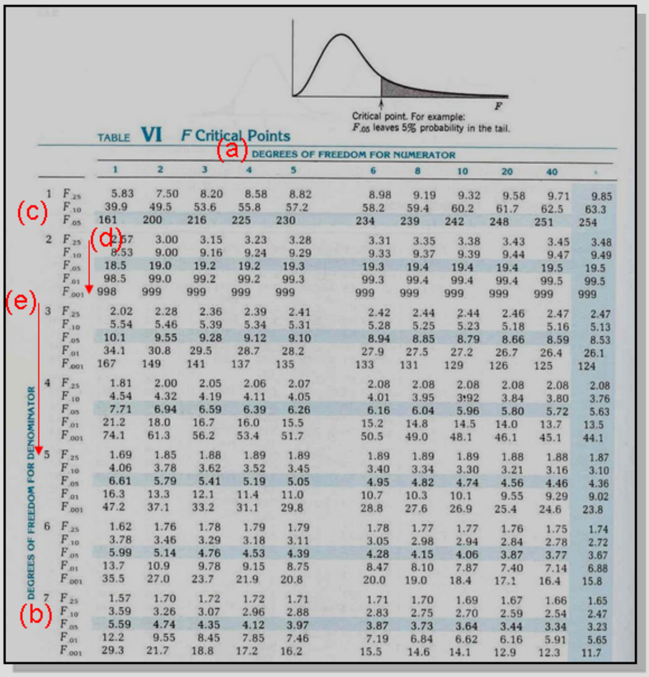
$$F0 = \frac{SS_{Treatment}/(a-1)}{SS_{Error}/(N-a)}$$

(b) $\qquad SS_{Error}/(N\text{-}a)$

(c) α is users choice

(d) Increasing for α

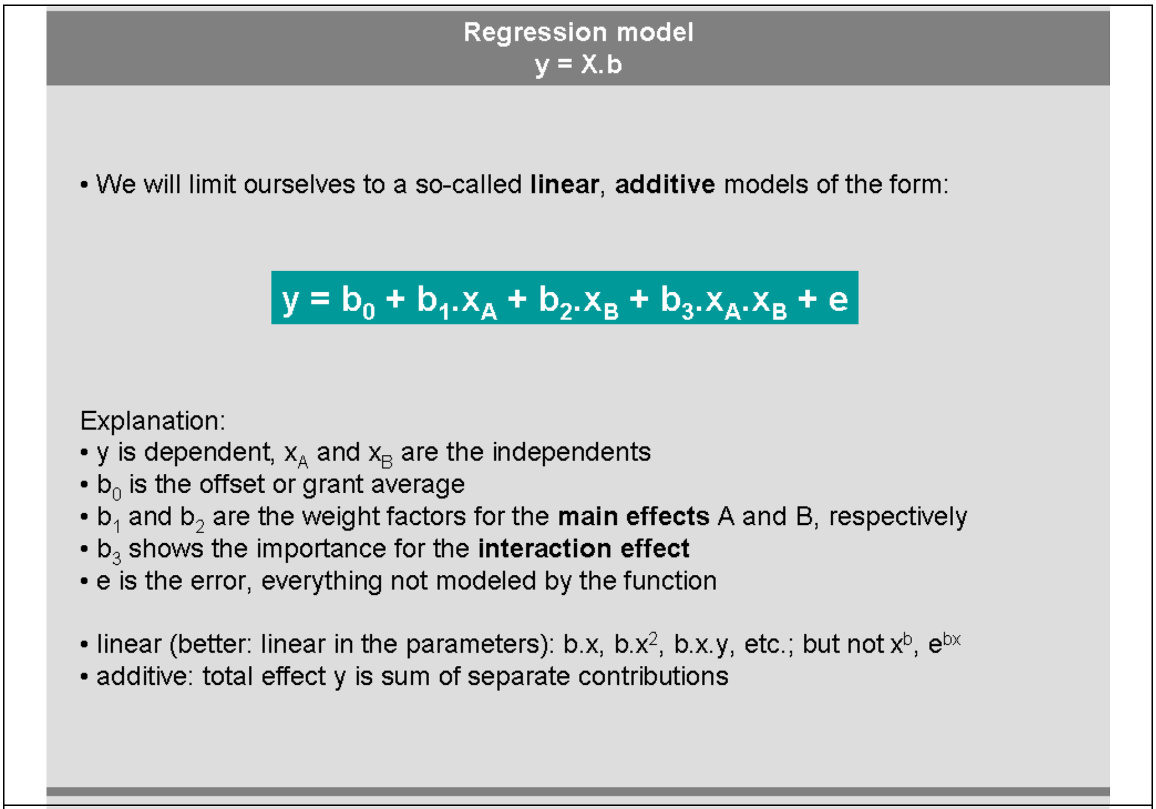
(e) Decreasing for *N*



Critical point. For example: $F_{.05}$ leaves 5% probability in the tail.

TABLE VI *F* Critical Points

DEGREES OF FREEDOM FOR DENOMINATOR		DEGREES OF FREEDOM FOR NUMERATOR										
		1	2	3	4	5	6	8	10	20	40	∞
1	$F_{.25}$	5.83	7.50	8.20	8.58	8.82	8.98	9.19	9.32	9.58	9.71	9.85
	$F_{.10}$	39.9	49.5	53.6	55.8	57.2	58.2	59.4	60.2	61.7	62.5	63.3
	$F_{.05}$	161	200	216	225	230	234	239	242	248	251	254
2	$F_{.25}$	2.57	3.00	3.15	3.23	3.28	3.31	3.35	3.38	3.43	3.45	3.48
	$F_{.10}$	8.53	9.00	9.16	9.24	9.29	9.33	9.37	9.39	9.44	9.47	9.49
	$F_{.05}$	18.5	19.0	19.2	19.2	19.3	19.3	19.4	19.4	19.4	19.5	19.5
	$F_{.01}$	98.5	99.0	99.2	99.2	99.3	99.3	99.4	99.4	99.4	99.5	99.5
	$F_{.001}$	998	999	999	999	999	999	999	999	999	999	999
3	$F_{.25}$	2.02	2.28	2.36	2.39	2.41	2.42	2.44	2.44	2.46	2.47	2.47
	$F_{.10}$	5.54	5.46	5.39	5.34	5.31	5.28	5.25	5.23	5.18	5.16	5.13
	$F_{.05}$	10.1	9.55	9.28	9.12	9.10	8.94	8.85	8.79	8.66	8.59	8.53
	$F_{.01}$	34.1	30.8	29.5	28.7	28.2	27.9	27.5	27.2	26.7	26.4	26.1
	$F_{.001}$	167	149	141	137	135	133	131	129	126	125	124
4	$F_{.25}$	1.81	2.00	2.05	2.06	2.07	2.08	2.08	2.08	2.08	2.08	2.08
	$F_{.10}$	4.54	4.32	4.19	4.11	4.05	4.01	3.95	3:92	3.84	3.80	3.76
	$F_{.05}$	7.71	6.94	6.59	6.39	6.26	6.16	6.04	5.96	5.80	5.72	5.63
	$F_{.01}$	21.2	18.0	16.7	16.0	15.5	15.2	14.8	14.5	14.0	13.7	13.5
	$F_{.001}$	74.1	61.3	56.2	53.4	51.7	50.5	49.0	48.1	46.1	45.1	44.1
5	$F_{.25}$	1.69	1.85	1.88	1.89	1.89	1.89	1.89	1.89	1.88	1.88	1.87
	$F_{.10}$	4.06	3.78	3.62	3.52	3.45	3.40	3.34	3.30	3.21	3.16	3.10
	$F_{.05}$	6.61	5.79	5.41	5.19	5.05	4.95	4.82	4.74	4.56	4.46	4.36
	$F_{.01}$	16.3	13.3	12.1	11.4	11.0	10.7	10.3	10.1	9.55	9.29	9.02
	$F_{.001}$	47.2	37.1	33.2	31.1	29.8	28.8	27.6	26.9	25.4	24.6	23.8
6	$F_{.25}$	1.62	1.76	1.78	1.79	1.79	1.78	1.77	1.77	1.76	1.75	1.74
	$F_{.10}$	3.78	3.46	3.29	3.18	3.11	3.05	2.98	2.94	2.84	2.78	2.72
	$F_{.05}$	5.99	5.14	4.76	4.53	4.39	4.28	4.15	4.06	3.87	3.77	3.67
	$F_{.01}$	13.7	10.9	9.78	9.15	8.75	8.47	8.10	7.87	7.40	7.14	6.88
	$F_{.001}$	35.5	27.0	23.7	21.9	20.8	20.0	19.0	18.4	17.1	16.4	15.8
7	$F_{.25}$	1.57	1.70	1.72	1.72	1.71	1.71	1.70	1.69	1.67	1.66	1.65
	$F_{.10}$	3.59	3.26	3.07	2.96	2.88	2.83	2.75	2.70	2.59	2.54	2.47
	$F_{.05}$	5.59	4.74	4.35	4.12	3.97	3.87	3.73	3.64	3.44	3.34	3.23
	$F_{.01}$	12.2	9.55	8.45	7.85	7.46	7.19	6.84	6.62	6.16	5.91	5.65
	$F_{.001}$	29.3	21.7	18.8	17.2	16.2	15.5	14.6	14.1	12.9	12.3	11.7



Regression model
y = X.b

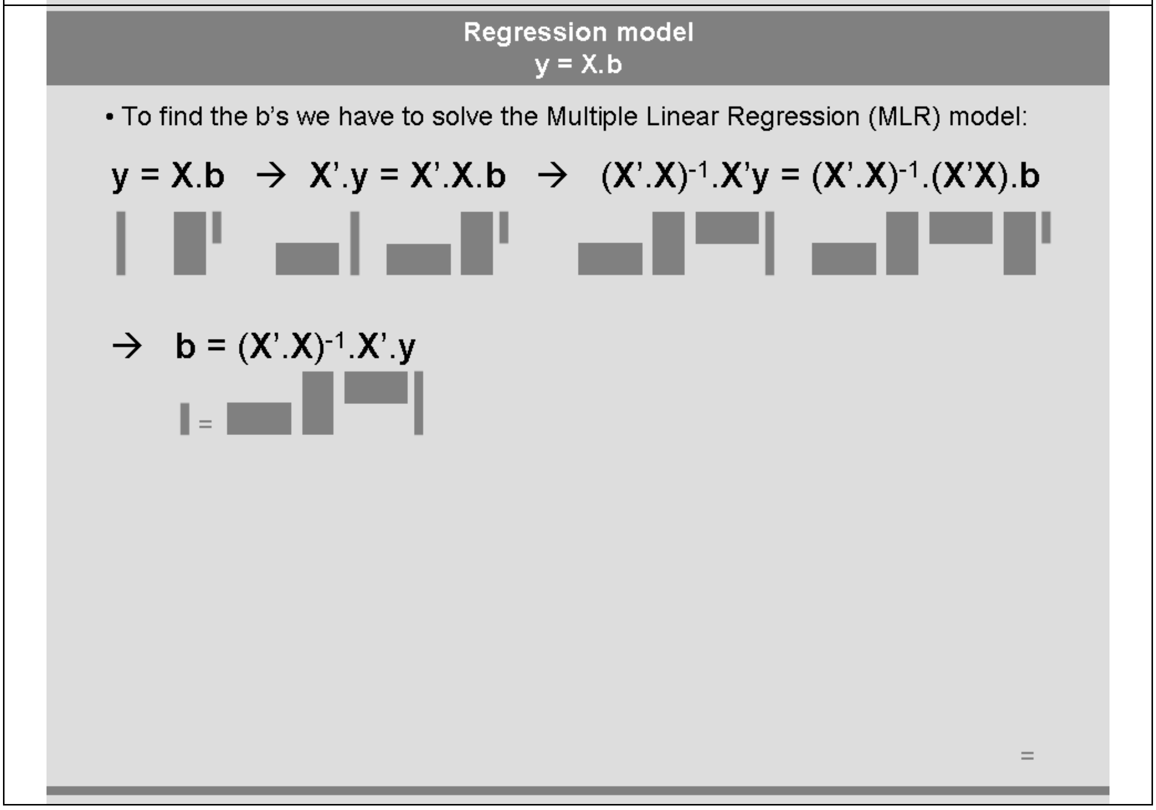
- We will limit ourselves to a so-called **linear**, **additive** models of the form:

$$y = b_0 + b_1.x_A + b_2.x_B + b_3.x_A.x_B + e$$

Explanation:

- y is dependent, x_A and x_B are the independents
- b_0 is the offset or grant average
- b_1 and b_2 are the weight factors for the **main effects** A and B, respectively
- b_3 shows the importance for the **interaction effect**
- e is the error, everything not modeled by the function

- linear (better: linear in the parameters): b.x, $b.x^2$, b.x.y, etc.; but not x^b, e^{bx}
- additive: total effect y is sum of separate contributions

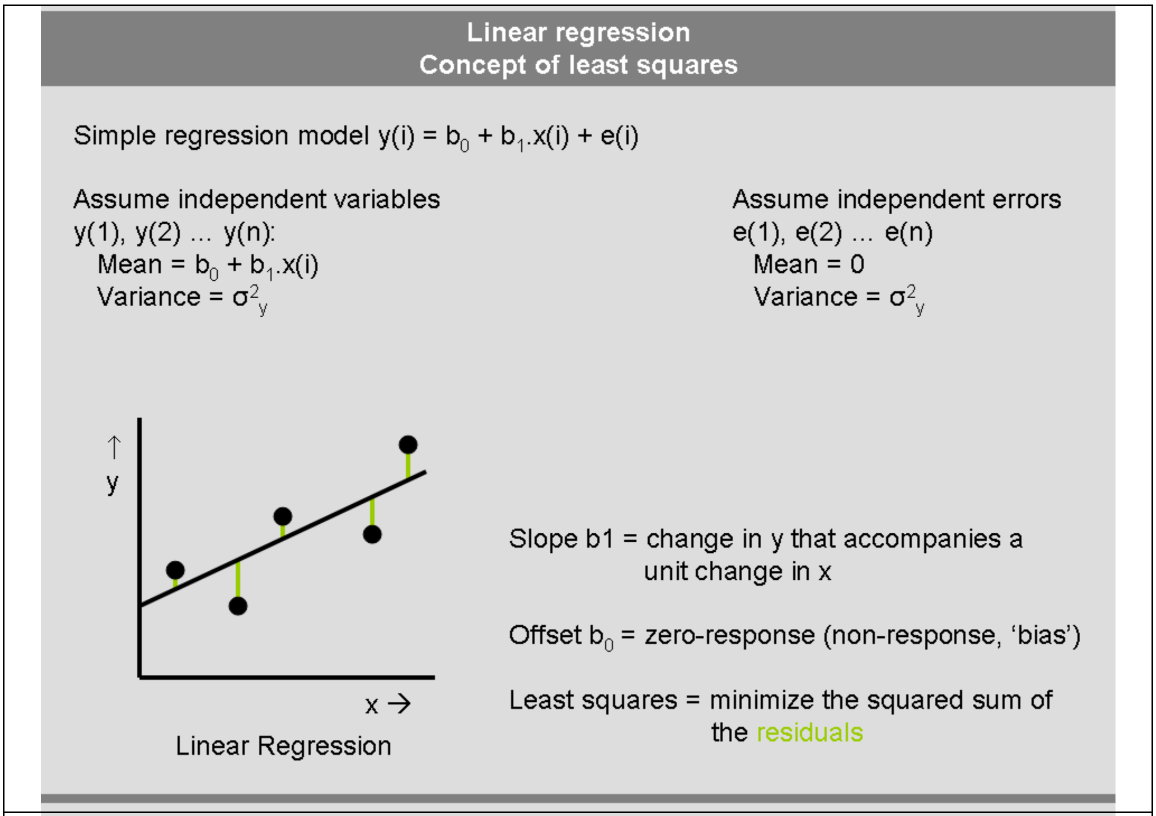


Regression model
y = X.b

- To find the b's we have to solve the Multiple Linear Regression (MLR) model:

$$y = X.b \rightarrow X'.y = X'.X.b \rightarrow (X'.X)^{-1}.X'y = (X'.X)^{-1}.(X'X).b$$

$$\rightarrow b = (X'.X)^{-1}.X'.y$$



Linear regression
Concept of least squares

Simple regression model $y(i) = b_0 + b_1.x(i) + e(i)$

Assume independent variables
$y(1)$, $y(2)$ … $y(n)$:
Mean = $b_0 + b_1.x(i)$
Variance = σ^2_y

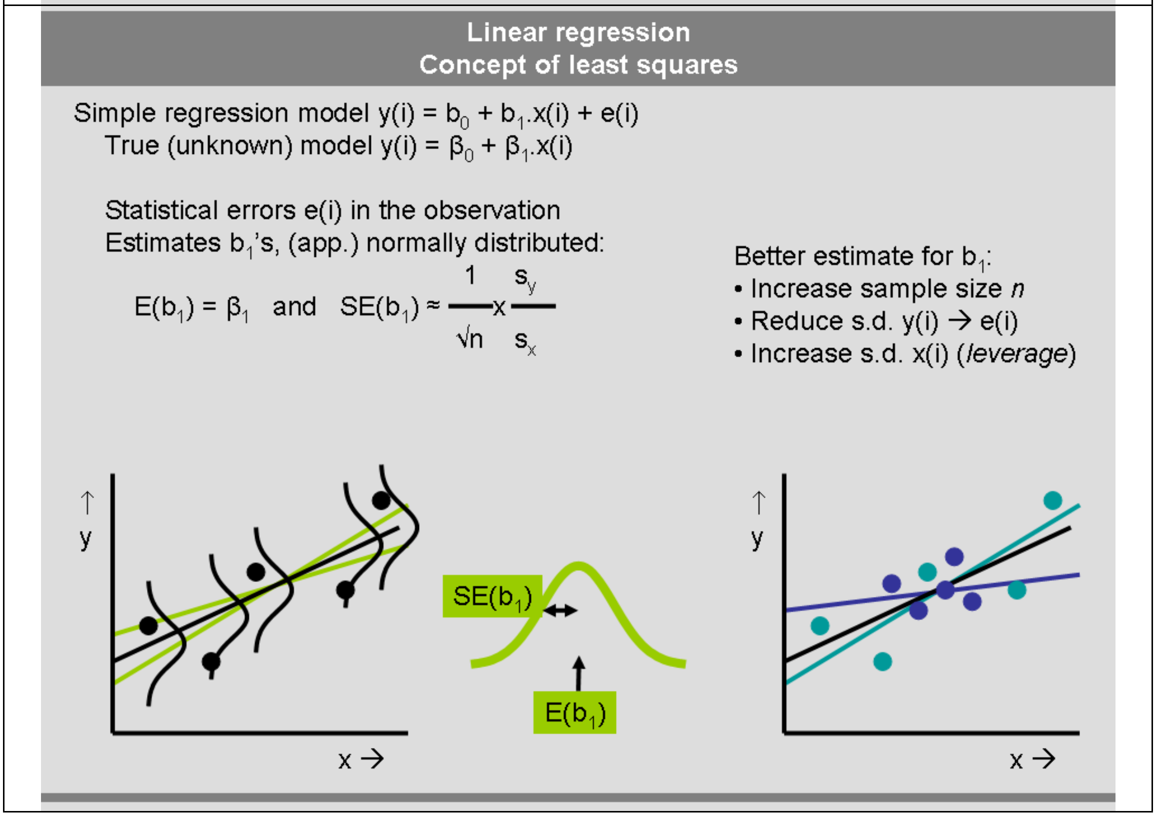
Assume independent errors
$e(1)$, $e(2)$ … $e(n)$
Mean = 0
Variance = σ^2_y

Linear Regression

Slope b1 = change in y that accompanies a unit change in x

Offset b_0 = zero-response (non-response, 'bias')

Least squares = minimize the squared sum of the residuals



Linear regression
Concept of least squares

Simple regression model $y(i) = b_0 + b_1.x(i) + e(i)$
True (unknown) model $y(i) = \beta_0 + \beta_1.x(i)$

Statistical errors $e(i)$ in the observation
Estimates b_1's, (app.) normally distributed:

$$E(b_1) = \beta_1 \quad \text{and} \quad SE(b_1) \approx \frac{1}{\sqrt{n}} x \frac{s_y}{s_x}$$

Better estimate for b_1:
- Increase sample size *n*
- Reduce s.d. $y(i)$ → $e(i)$
- Increase s.d. $x(i)$ (*leverage*)

Linear regression
Equations

$$y(i) = b_0 + b_1 x(i) + e(i) \qquad \hat{y}(i) = b_0 + b_1 x(i) \quad \rightarrow \quad y(i) - \hat{y}(i) = e(i)$$

$$\text{least squares : minnimize } \sum e(i)^2$$

$$b_1 = \frac{\sum (x(i) - \bar{x})(y(i) - \bar{y})}{\sum (x(i) - \bar{x})^2} = \frac{s_{xy}}{s_x^2} \qquad b_0 = \bar{y} - b_1 \bar{x}$$

(x_{bar},y_{bar}) is mass center of data cloud!

$$r = \frac{\sum (x(i) - \bar{x})(y(i) - \bar{y})}{\sqrt{\sum (x(i) - \bar{x})^2} \sqrt{\sum (y(i) - \bar{y})^2}} \qquad b_1 = r \frac{s_y}{s_x}$$

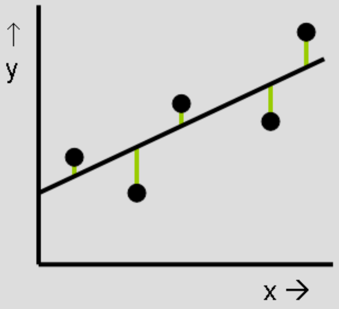
Sample correlation/determination coefficient

Multiple linear regression
Least squares model fitting

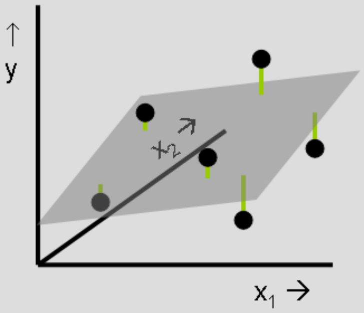
$y(i) = b_0 + b_1.x_1(i) + b_2.x_2(i) + b_3.x_1(i).x_2(i) + e(i)$

Least squares model fit

$\min \Sigma(y(i) – (b_0 + b_1.x_1(i) + b_2.x_2(i) + b_3.x_1(i).x_2(i)))^2 = \min (\Sigma e(i)^2)$

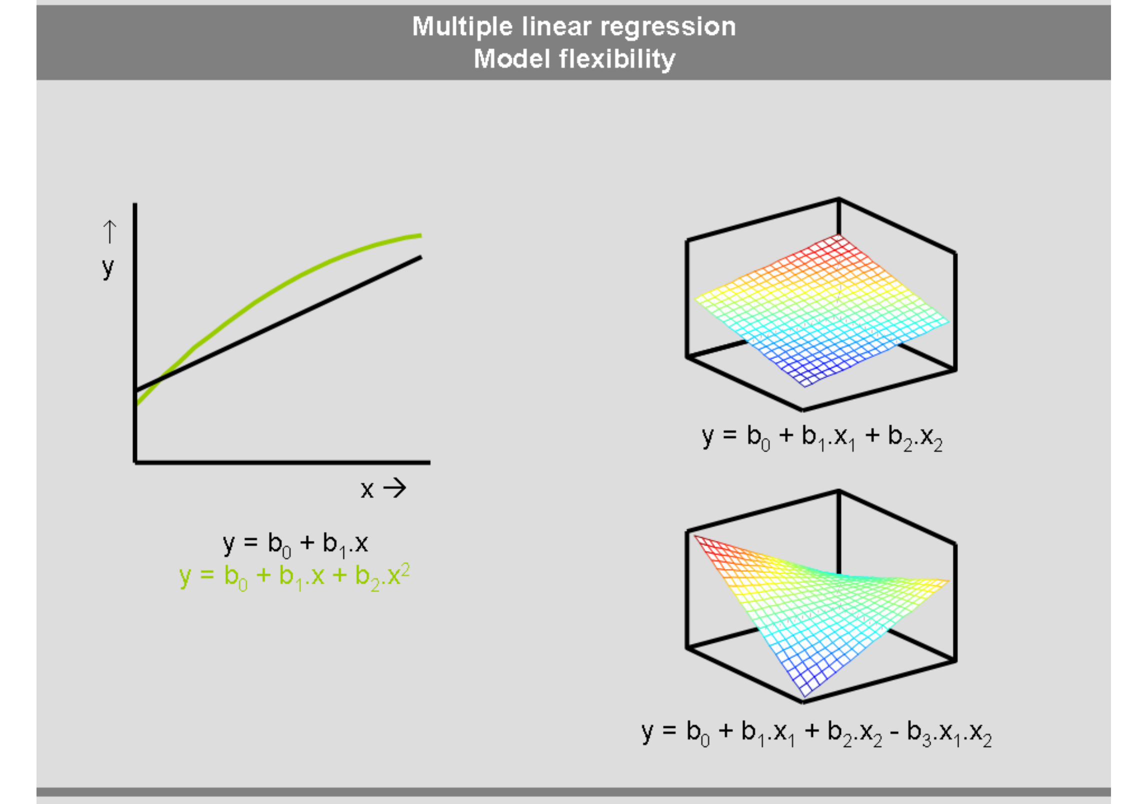


Linear Regression
(univariate)



Multiple Linear Regression
(MLR; multivariate)

More points (replicates) = better estimates



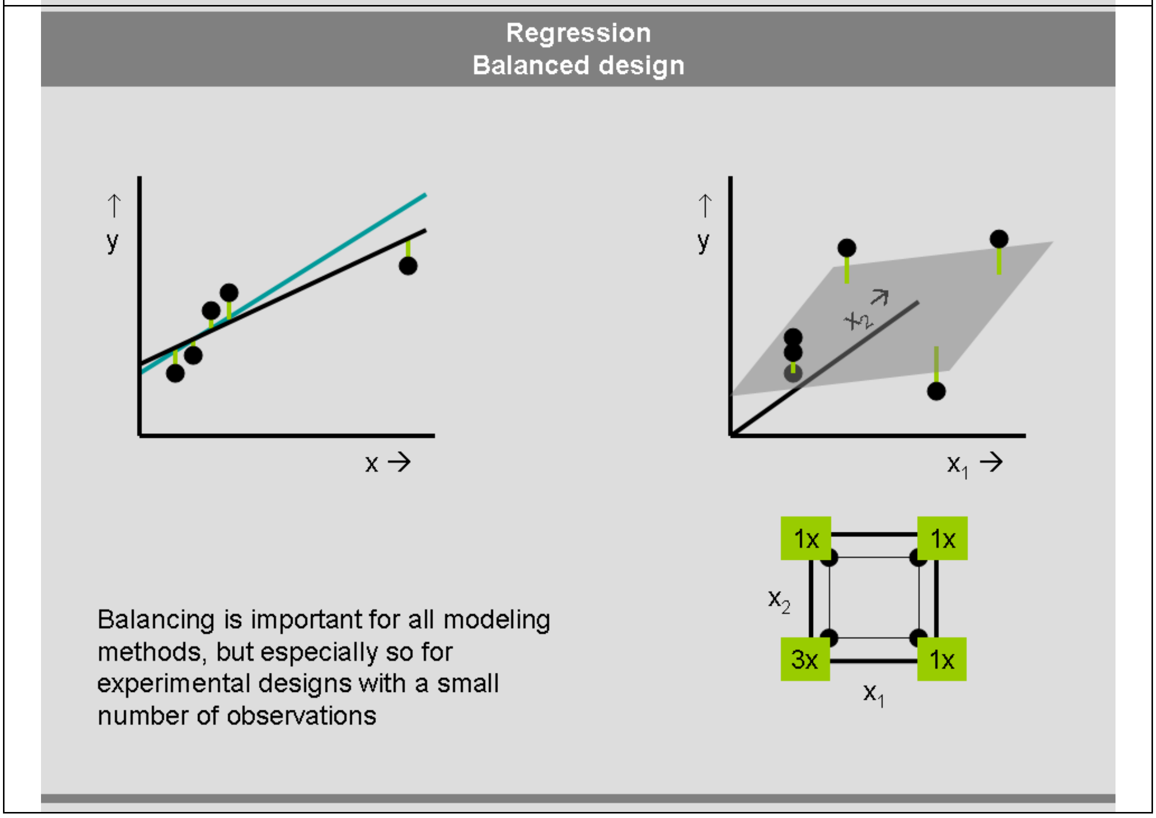
Multiple linear regression
Model flexibility

$y = b_0 + b_1.x$

$y = b_0 + b_1.x + b_2.x^2$

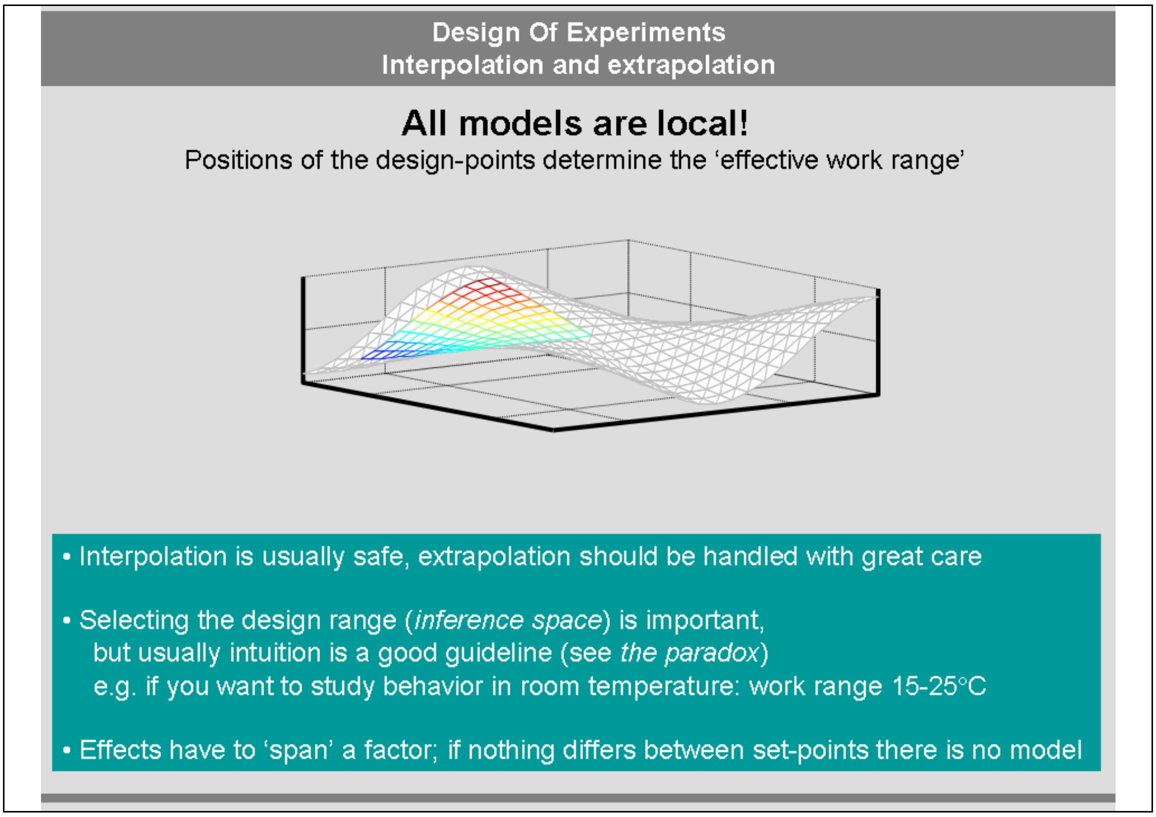
$y = b_0 + b_1.x_1 + b_2.x_2$

$y = b_0 + b_1.x_1 + b_2.x_2 - b_3.x_1.x_2$



Regression
Balanced design

Balancing is important for all modeling methods, but especially so for experimental designs with a small number of observations



Design Of Experiments
Interpolation and extrapolation

All models are local!

Positions of the design-points determine the ‘effective work range’

- Interpolation is usually safe, extrapolation should be handled with great care
- Selecting the design range (*inference space*) is important,
  but usually intuition is a good guideline (see *the paradox*)
  e.g. if you want to study behavior in room temperature: work range 15-25°C
- Effects have to ‘span’ a factor; if nothing differs between set-points there is no model