Elementary Matlab

In this handout you will learn the vocabulary and elementary syntax of the Matlab language. We will start by looking at simple mathematical operations, but then move towards more interesting commands that allow you to analyze and understand your data.

Even though we will only start talking in earnest about functions and scripts much later, you need to be aware of some terminology now. I'll use the word command to refer to any of the built-in commands of Matlab. For instance 'help' is a command. Commands come in two types: functions and scripts. The difference is that a script executes without taking any input arguments; you just type the script name on the command line and something happens. An example is `exit`; this command exits the Matlab program.

Functions are more flexible; they take zero or more input arguments, and I'll sometimes talk about those arguments being "passed" to the function. The help function is an example. When you pass the + argument the help function provides information on the + command. Functions usually return arguments as well. In Matlab this is quite flexible as a function can return zero or more arguments. An example is the `sqrt` function, you pass it a number and it returns the square root of that number. So you should think of a function as a room where you pass some information through one or more doors (the input arguments) and some information (that typically depends on that input) is returned back to you through one or more other doors (the output arguments).

The arguments that you pass to a function are placed inside parentheses, like this: `sqrt(4)`; multiple arguments, are separated by commas `plus(1,1)`.

1. Variables

- What are variables?
- How should you name your variables?

Variables are the memory of Matlab (or any other programming language). Variables allow you to store intermediate results of your calculations for later use. For instance:

```
a = exp(10)
a =
     2.2026e+04
```

This stores the result of the simple calculation `exp(10)` in the variable with the name `a`. In a subsequent calculation, you can then use the variable name `a` to refer to the number 2.2026e+004. Hence:

```
log(a)
ans =
     10
```

Variable names in Matlab can be arbitrarily long, cannot start with a number, but can contain any alphanumeric characters, and are case-sensitive. Be careful not to redefine one of the Matlab commands as a variable. Matlab will not give any warnings when you type:

```
print =2
print =
     2
```

Afterwards, however, you will not be able to use the `print` command anymore: in response to `print`, Matlab will just return the contents of the variable `print`, rather than executing the command `print`.

The variable with the name `ans`, short for *answer*, is a special variable that contains the result of the last calculation at the command prmpt that was not assigned to a variable. In other words, if you enter

a mathematical command at the command prompt without specifying a variable for the result, Matlab automatically assigns the result to the variable `ans`. For instance,

```
sqrt(99)
ans =
    9.9499
```

Just as a variable you define, `ans` can be used in further calculations:

```
ans*ans
ans =
    99
```

In Matlab a variable can contain just about anything: a list of numbers (a vector), a bit of text (a string). For instance, you could store a list of reaction times in a variable:

```
reactionTimes = [0.3 0.4 0.2 0.3 0.5]
reactionTimes =
    0.3000    0.4000    0.2000    0.3000    0.5000
```

The variable `reactionTimes` now contains 5 values. Or you can store a subject name:

```
subjectName = 'joe'
subjectName =
joe
```

Variables can be thought of as *boxes* containing data. These can be raw data, acquired from some data-recording program, but also processed data, which are the result of a Matlab calculation with raw data. Another way of thinking about variables is as the *name* or *address* of a chunk of data. Hence, the variable `reactionTimes` tells Matlab where to look for those particular numbers. The variables you define are stored in something called the Matlab workspace. When you start Matlab, a default empty workspace is created. To view the contents of this workspace, you can open the Workspace window in the IDE (In the Ribbon, go to Layout | Workspace), or on the command line: type `who` for a brief description or `whos` for a listing that includes the sizes and formats of the variables.

```
whos
 Name            Size          Bytes        Class        Attributes
 A               1x1           8            Double
 Ans             1x1           8            Double
 print           1x1           8            Double
 reactionTimes   1x5           40           Double
 subjectName     1x3           6            Char
```

This tells us that we have five defined variables.; their properties are listed in the 5 columns.The first variable – `a` – has size 1x1, which means it contains a single number, the result of the calculation `exp(10)`, above. Matlab uses 8 bytes to store this value, and the 'class' of the variable is double. This simply means that it is a numerical value, stored with high (double) precision. The `reactionTimes` variable has a size 1x5, which means that it contains one row of 5 elements. Matlab uses 40 bytes of memory to store the numbers in `reactionTimes`. Finally, the `subjectName` variable is listed with a size of 1x3, and class char; this means it contains 3 characters ('j','o','e') and this requires only 6 bytes of memory.

Once you leave a workspace, which happens among other times when you leave the Matlab program, the variables are lost. In programming terms this is referred to as variables going *out of focus*, or being no longer *visible*. If you want to save your variables for use in a future session of Matlab, you can save them to a file by typing

```
save myDataFile reactionTimes subjectName
```

This will save the listed variables, with their contents, to the file called "myDataFile.mat". In a future session of Matlab, these data can be retrieved with

```
load myDataFile
```

The whole workspace, with all its variables, can be stored by leaving out the list of variable names (`save myWorkspace`) or by using the Save Workspace command on the Ribbon.

When you no longer need a certain variable, you can remove it from the workspace by typing
```
clear reactionTimes
```
This will free up the memory used by this variable. In programs that extensively use intermediate results an occasional clear of superfluous variables can improve performance. The function
```
clear all
```
removes all variables from the workspace. Another way to improve performance is to type the command `pack`. This reorganizes the physical storage of the variables such that more large chunks of memory become available. But unless you are working with very large data sets, or on a computer with extremely limited RAM memory, you should not worry too much about this. Matlab will warn you when it runs out of memory.

1.1.1. Variable Names

With some limitations, the choice of variable names is up to you. It has to start with a character, cannot contain spaces, but can contain numbers. Matlab cares little about your variable names, but you should care a great deal as they are an important aspect of writing reusable code. The primary rule is that you should use variable names that are meaningful: use reactionTime, nrSpikes, voltage, color, and not abstract names such as x,y, v1,v2,etc.

If your variable names consist of two words, make them visibly distinct. I prefer the so-called Camel-Case convention in which each variable starts with a lower-case, but each subsequent word in the variable name starts with an uppercase character (e.g. reactionTime). Another convention separates words by underscores (e.g. reaction_time); this looks ugly to me, but your aesthetic experience may differ.

It is a good idea to abbreviate names that you use frequently. For instance, I use nrSpikes, not numberOfSpikes, and if you always analyse reaction times, then using rt instead of reactionTimes is certainly a time-saver. Just pick a convention and stick to it; it will greatly improve the readability of your code.

The Matlab Style Guide by Richard Johnson is an excellent resource for guidance on how to name your variables and other strategies you can use to improve the readability of your Matlab code.

2. Working with Numbers: Arrays

- Store your data in Matlab arrays
- Perform elementary mathematical calculations
- Linear algebra

To program effectively in Matlab you have to learn to think of your data as arrays. Numeric arrays come in many shapes and sizes in Matlab: from single numbers to tables with rows and columns of numbers and even high dimensional objects that contain mutiple pages of tables. Because each of thse elements can be used in almost identical ways in Matlab I will use the words 'array' and 'matrix' interchangeably. The context will clarify whether we're talking about scalars (single number), vectors (lists of numbers), matrices (tables of numbers), or high-dimensional arrays.

Given a matix you can determine its rows and columns by using the `size` command. For instance:
```
size(m)
ans =
     2     3
```
Which means that the matrix m has 2 rows and 3 columns. I'll refer to this as a 2x3 matrix. For instance, if you performed a reaction time experiment with 5 subjects, who repeated the experiment

10 times, you could store the data in a 10x5 matrix. Or, to represent the number of spikes in 100 miliseconds recorded over 100 seconds from 10 cells, you would use a 1000x10 matrix.

Vectors are just lists of numbers – they are also matrices, but they have just one row (1xN matrix) or just one column (Nx1). The former is called a row vector, the latter a column vector. A vector can, for instance, contain all the times a single cell fired during an extracellular recording, or as in the `reactionTimes` variable above, the reaction times of a subject in a psychophysical detection task.

The decisions you make about how to represent your data are very important as they determine to a large extent whether your code will be simple or cumbersome to write. Keep in mind the mantra to 'keep it together '. This means that data that belong together, because they refer to the same experiment, should be stored together in one matrix or vector.

If your variables differ only by a numeric suffix, your choice of how to represent the data is likely wrong.

A researcher has collected data from 3 subjects who pressed a button as soon as they saw a target appear on a computer monitor. You receive the data as three (row) vectors:

```
reactionTime1 = [0.5426 0.7377 0.5107 0.7036 0.6177];
reactionTime2 = [0.5159 0.5486 0.6585 0.7194 0.8976];
reactionTime3 = [0.6385 0.9117 0.9751 0.6908 0.5934];
```

Each of these variables represents the same kind of information, so this should be stored together. Combine the three vectors into one matrix:

```
reactionTime = [reactionTime1' reactionTime2' reactionTime3']
reactionTime =
    0.5426    0.5159    0.6385
    0.7377    0.5486    0.9117
    0.5107    0.6585    0.9751
    0.7036    0.7194    0.6908
    0.6177    0.8976    0.5934
```

Note that I used the apostrophe (') to transpose each of the reactionTime vectors (i.e. turn a row vector into a column vector). Now we have a 5x3 matrix in which each of the three columns represents a subject. This, again, is an important choice of how to represent your data. We could have combined these same vectors in a 3x5 matrix, but this would turn out to be less efficient. The general rule is that the tightest link between the numbers should be along the row dimension. In this case, the five reaction times of a given subject are more tightly linked (they represent measurements of the same underlying quantity) than the reaction times of subject 1 and subject 2. Therefore the individual trials should be along the row dimension, and the subjects along the column dimension.

The scalar, vector, matrix ladder can be extended to N-dimensional arrays. These arrays work just like matrices, except that beyond rows and columns, they also have so-called *pages*. A three dimensional array is, for instance, useful to store all results from 2 groups (let's say young and old) of 5 subjects who did the same reaction time experiment 10 times. You use an array with size 10x5x2. Each column represents the reaction times of a single subject. The first page represents the data from the young subjects, the second page the data from the old subjects.

There is no limit to the number of dimensions in a Matlab array and you should generally match the dimensions of your data to the factors in your experiment. For instance, the previous example had three factors Trials, Subjects, Age, and therefore a three-dimensional array was most appropriate to represent the data. Now if your experiment also tested reaction times before and after performing some pharmacological manipulation, you would have another factor (Drug), and you should store your data in a four-dimensional array.

Arrays (1, 2, 3, or N-dimensional ones) are by far the most appropriate type of variable to store your data for numerical analysis. So, before you start analyzing data, you should put some effort into determining the appropriate shape of your data arrays, and if at all possible, use arrays. There are, however, some situations where arrays are cumbersome. For instance, for a balanced reaction time experiment (each combination of factors has the same number of trials) the array would have numbers in all 10*2*5*2 =200 locations. However, if some subjects only completed 9 trials, and others only 5, then the data don't really fit in a neat array. Or, put differently you'd need an array in which some columns have 9 rows, while others have 5; Matlab arrays, however, do not allow this.

There is a variable type (called a cell-array) that allows you to represent such ragged arrays. However, you should use those only as a last resort as many of the benefits of vectorized computing are lost when you use cell arrays. Often, a better alternative is to use an ND array in which you highlight some data as 'missing'. For instance, if one subject failed to perform the last trial, then you would enter NaN ('Not a Number') in the appropriate location in the ND array. Data functions in Matlab exist (or can be written) to deal with such missing data appropriately. For instance, `nanmean` determines the mean while ignoring the missing (NaN) values in a matrix. These nan-aware functions are part of the statistics toolbox, but if you do not have access to this toolbox, you can easily write such functons yourself (See Exercises).

2.1.1. Creating Matrices

You can always create a matrix by typing in each of its entries. For instance, using a semicolons to separate the rows of a matrix.

```
m = [1 2; 3 4]
m =
     1     2
     3     4
```

Of course this is only feasible for small sets of numbers; there are many useful commands that allow you to create matrices more easily:

For instance, you can generate vectors by iteration. For instance:

```
x = 0:1:10
y = 0:2:10
z = 10:-1:0
x =
     0   1   2   3   4   5   6   7   8   9  10
y =
     0   2   4   6   8  10
z =
    10   9   8   7   6   5   4   3   2   1   0
```

The notation a:c can be read as: 'all integer numbers between a and c'. The notation a:b:c extends this to all numbers between a and c, stepping with a step size equal to b. You can also use the two special functions `linspace(start, stop, number)` and `logspace(start, stop, number)` to create vectors of a specified length in which the entries are spaced linearly or logarithmically between two values:

```
linspace(0,100,11)
ans =
     0  10  20  30  40  50  60  70  80  90 100
```

In logspace, the entries represent the exponents:

```
logspace(0,2,5)
ans =
    1.0000 3.1623 10.0000 31.6228 100.0000
```

You can create matrices by concatenating vectors:

```
a = [1 2 3]; b = [4 5];
c = [a b]
```

```
c =
     1    2    3    4    5
```
or by concatenating other matrices in the column direction:
```
bigM = [m m]
bigM =
     1     2     1     2
     3     4     3     4
```
Or in the row direction:
```
bigM = [m ; m]
bigM =
     1     2
     3     4
     1     2
     3     4
```
Another way to achieve the same result is the `cat` function. The first argument instructs `cat` along which dimension the concatenation should be done (1=rows, 2=columns, etc.) , and the subsequent arguments are the matrices that should be concatenated. Hence `cat(2,m,m)` is the same as `[m;m]`. The cat function is particularly useful for concatenating arrays along the 3rd or higher dimensions (e.g. `cat(3,m,m)`), where no shortcuts like `[m;m]` exist. We'll see examples of this below.

Concatenation can be useful to create long vectors that have a certain kind of regularity with a minimal amount of typing:
```
upAndDown = [(1:2:10) (10:-2:1)]
x =
     1    3    5    7    9   10    8    6    4    2
```

Note the use of square brackets and parentheses. The square brackets are used to group the individual elements of a vector (or matrix). The parentheses are used to group iterations of the `(a:b:c)` type, to group mathematical operations `3*(5+6)`, as well as to denote function arguments as in `exp(10)`.

Of course, you will normally only join numbers in a matrix if these numbers are somehow similar. For instance, all of them represent spike counts, or all of them voltages, or BOLD signals, but you should not store voltages and spike counts together in the same matrix. A different way of putting this is that all entries in a matrix should have the same units. There is nothing in Matlab that enforces this, but it is a good practice that will keep your Matlab code readable.

If you analyze data that could have many different units (e.g. miliseconds, seconds, volt, milivolt, centimeters, inches etc.), you should adopt a convention to use the units in the variable name. For instance `reactionTimeMs` for a variable that contains the reaction time in miliseconds. This is particularly important if your data come from different recording devices that use different units for the raw data. If you think this is cumbersome and too much work, you should read why the Mars Climate Orbiter crashed into Mars in 1998.

The `zeros` function creates a matrix with a specified number of rows and columns, with all entries equal to zero.

```
zeros(1,5)
ans =
     0     0     0     0     0
zeros(2,4)
ans =
     0     0     0     0
     0     0     0     0
```
Similarly, the `ones` command allows you to easily create a matrix whose entries are all 1 (or any other number).
```
ones(1,5)
ans =
```

```
     1     1     1     1     1
4.*ones(3,3)
ans =
     4     4     4
     4     4     4
     4     4     4
```

Finally, the `repmat` command (repeat matrix) can create large matrices from small ones:

```
repmat(1:3,[2 1])
ans =
     1     2     3
     1     2     3
```

Repmat takes the first argument (`1:3`) and repeats it 2 times across the row dimension and once in the column dimension [2 1], hence the `1:3` input is turned into a 2x3 matrix.

A similar function is `meshgrid`: it takes two vector arguments, and repeats the first vector as many time as there are elements in the second, and repeats the second as many times as there are elements in the fits. An example:

```
[X,Y] = meshgrid([1 2],[10 20])
X =
     1     2
     1     2
Y =
    10    10
    20    20
```

This is useful if you have data with two independent variables and you want to look at all possible pairwise combinations.

2.1.2. Element-wise Matrix Calculations

In most programming languages lists of numbers (vectors or matrices) are manipulated sequentially. For instance, to calculate the sine of a long list of angles, you would start at the first in the list, calculate the sine, go to the next etc. In Matlab you can calculate the sine of a list of values with a single command:

```
x = 0:pi/4:pi;
sin(x)
ans =
         0    0.7071    1.0000    0.7071    0.0000
```

Not only does this work for lists of numbers (vectors) but even for whole tables (matrices):

```
cos(m)
ans =
    0.5403   -0.4161   -0.9900
    0.5403   -0.4161   -0.9900
```

In fact, *all* mathematical operations are defined on a matrix basis. You can add and subtract, multiply or divide the elements in a matrix all with a single command, as if you were adding single numbers. This so-called vectorized nature of Matlab commands is very powerful and makes code efficient and compact.

After collecting the reaction time data you realize that there was an error in the reaction time clock. All times are 0.1s too long. To correct this, you type `reactionTimes = reactionTimes-0.1.` *After this operation, all reactionTimes have been reduced by 0.1.*

When you add two matrices of the same size, all their corresponding elements are added, and a matrix of the same size results. For instance, add two vectors:

```
x = 1:10;
y = 10:-1:1;
x+y
ans =
    11    11    11    11    11    11    11    11    11    11
```

Or add two matrices:

```
m+m
ans =
     2     4     6
     2     4     6
```

Note that you cannot add or subtract two matrices that have different sizes. You will receive an error message that will soon become very familiar…

```
x=1:2;y=1:10;
x-y
??? Error using -
Matrix dimensions must agree.
```

The only exception to this is the example above where a single number (a 1x1 matrix) is added to a matrix with more than 1 row or column. In this case, the single number is added to each of the elements in the matrix.

The reaction time data were provided in seconds, but you decide that you'd rather work in miliseconds. You multiply each of the numbers with 1000: `reactionTimes = reactionTimes.*1000`*. Now your reaction times data are in miliseconds.*

The general rule for multiplication is the same as for addition/subtraction; you can multiply any two matrices element-wise, using the `.*` operator as long as they have the same size, or one of them has a size of 1x1. For instance:

```
a = [1 0; 0 1]
a =
     1     0
     0     1

b = [10 20; 30  40]
b =
    10    20
    30    40

a.*b
ans =
    10     0
     0    40
```

Matlab handles division just like multiplication. The operator `./` denotes element-by-element division:

```
m./m
ans =
     1     1     1
     1     1     1
```

Powers of the elements of a matrix are calculated with `.^`:

```
b.^2
ans =
```

```
         100         400
         900        1600
```

2.1.3. Matrix Operations & Linear Algebra

It is easy to perform linear algebra in Matlab. For instance matrix multiplication:

```
a= [1 0; 0 1]; b=[1 2 ;3 4];
a*b
ans =
     1     2
     3     4
```

The operator for matrix multiplication is '`*`', the example shows the multiplication of the identity matrix `a` with the matrix `b`, which results in the matrix `b`. If your linear algebra is rusty, the Kahn Academy has some excellent videos that will help you brush up.

The inner product of two vector is also calculated with the '`*`' operator:

```
x = 1:10;
y = 10:-1:1;
x*y'
ans =
   220
```

Note that matrix multiplication is only defined when the number of columns of the first matrix is the same as the number of rows of the second matrix. This is why the vector `y` (a matrix with one row and ten columns) had to be transposed (') for the above example.

Just as you can multiply two matrices, you can also divide them:

```
x= A\B
```

Note that you should read this as 'divide `B` by `A`', because B is above the division line. When you ask Matlab to compute this, you actually ask it to solve the system of linear equations A*x =B. If A is a square matrix (and invertible) then the result will be $x = A^{-1}B$. If A is not square, then Matlab will determine the least-squares solution to the set of linear equations; type `help \` for more details.

2.1.4. Matrix Functions

Above you saw that functions that you would normally apply to a single number (such as `sin,cos, log,exp,round`) are applied to all numbers within a matrix by Matlab. These are called *elementary functions* in matlab and you can get a list of them by typing `help elfun`.

Data functions are Matlab functions that extract information from a collection of numbers. For instance, basic operations such as `mean, sum, median`, but also more complex operations such as `diff`, `gradient`, or `fft`. To list these functions, type `help datafun`. These functions do not distinguish between row-vectors or column-vectors. For instance:

```
sum(1:10) % Supply a row-vector
ans =
    55

mean((1:10)')%Supply a column-vector
ans =
    5.5000
```

However, when you provide a matrix, Matlab calculates the result separately for each column:

```
a=magic(3)
sum(a)
a =
     8     1     6
     3     5     7
```

```
     4     9     2
ans =
    15    15    15
```

This is the main reason why it is important to organize your data matrices in a logical, meaningful manner (see Page 5). In the reaction time experiment, we organized the data in a matrix where each column corresponds to a subject. Because of this choice we can now easily determine the mean reaction time and standard deviation for each subject:

```
meanReactionTime = mean(reactionTimes)
meanReactionTime =
    0.3400
stdReactionTime = std(reactionTimes)
stdReactionTime =
    0.1140
```

If you had set up the reactionTimes matrix with one subject per row then the operation `mean(reactionTimes)` would take the average over subjects per trial, which is much less likely to be meaningful. Of course, you could calculate `mean(reactionTimes')`. This transposes the matrix before passing it to the `mean` function such that each subject's data are, once again, in separate columns. Adding such a transpose makes your code more complex, somewhat less readable, and a little slower. The reason for this is that transposing the matrix takes time. For the small `reactionTimes` matrix in the example, this would not matter much, but for a large data matrix this adds up (especially because you would have to perform this transpose operation every time that you calculate some aggregate (e.g. `max`, `min`, `median`, etc)).

If your code uses the transpose operator (') over and over again, it is time to reconsider whether you chose your columns and rows appropriately.

Many data functions also allow you to specify the dimension along which you want to evaluate the function. For instance, you can calculate `mean(reactionTimes, 2)` to determine the mean along the 2nd (i.e. column) dimension. This is faster than transposing the `reactionTime` matrix, but your code becomes less readable. Especially because the dimension argument is not in the same place for all data functions. To compute the standard deviation across a column dimension you would have to write `stdev(reactionTimes, 0,2)` where the third argument specifies the column direction (the second argument specifies whether you want to normalize the standard deviation by N or N-1). So avoid this use of data functions if you can, by thinking carefully about the appropriate layout of your arrays.

A situtation where it may be difficult to avoid this usage is when you have high dimensional data arrays. With such arrays, averages in one dimension are often as meaningful as averages in another dimension. Or, you may want to average over trials and over subjects:

```
mean(mean(reactionTimes, 2),3)
ans =
    0.3400
```

When you do have to use these kind of calculations in a script, you can greatly improve readability by defininging variables with meaningful names that contain the values 2 and 3. For instance:

```
dimSUBJECT = 2;
dimTRIAL   = 1;
mean(mean(reactionTimes,dimTRIAL),dimSUBJECT);
```

when you come back to this line of code after many months, it is immediately obvious that this is an avereage over trials and subjects, and you would not have to go back to the place where you define the reactionTime matrix to figure out what the cryptic 2 and 3 mean.

2.1.5. Exercises

For most of these exercises there is more than one solution; this is typical for Matlab programs. Whenever you find a solution, do step back for a bit and think whether it could not have been done in a different, simpler, or faster way.

1 Create a 6 by 4 matrix whose first three rows have elements equal to 1, and the last three rows elements equal to 2.
2 Construct an 8 by 8 matrix, consisting of 2 by 2 submatrices with the entries: [1 2; 3 4]
3 During development, a fiber from retinal ganglion cells grows through the optic tract to reach the tectum. Its speed is 50 µm/h. Where is the growth cone after 1,2 ,4, 8, and 10 hours?
4 A hippocampal place cell fires 100 spikes when the rat sits on the left side of a room for 5 seconds, 750 spikes when it passes through the center in 1 second, and then 500 spikes when it sits on the right side of the room for 10 seconds. Calculate the firing rate in each of these locations.
5 Three subjects perform six rapid eye movements to a briefly presented visual target. The time between the presentation of the target and the time that the eye starts to move (the saccade latency) are recorded: Subject A: 0.1, 0.15, 0.02, 0.4, 0.2, 0.1 seconds. Subject B: 0.05, 0.12, 0.04, 0.1, 0.2, 0.5. Subject C: 0.3, 0.2, 0.1, 0.02, 0.02, 0.2. Make a matrix that describes these experimental data, then determine the mean and standard deviation of the saccade latency per subject.

3. Working with Text

- Storing text in arrays
- Search and replace text
- Parse text into its component parts

Matlab distinguishes text from from variable names or code by single (forward) quotes. This line assigns the text string 'john' to the variable firstName:

```
firstName = 'John'
firstName =
John
```

The firstName variable is a vector of characters, and you can treat it pretty much like any vector:

```
length(firstName)
ans =
     4
```

Concatenation :

```
lastName = 'Smith';
fullName = [firstName ' ' lastName]

fullName =
John Smith
```

or, equivalently, you can use the `strcat` function:

```
strcat(firstName, lastName)
ans =
JohnSmith
```

You can also create text matrices:

```
fullName = char(firstName,lastName)
fullName =
John
Smith
```

Let's check to see what kind of matrix this is:

```
size(fullName)
ans =
     2     5
```

So, `fullName` has two rows, each of which has 5 entries (characters). But 'John' only has 4 characters. The `char` function added a space at the end of 'John' so that its length matched up with the

length of 'Smith'. This was necessary because a matrix must have the same number of columns in each row. When working with text matrices you have to be aware of the possibility that there could be extra, padded, spaces in any row. The `isspace` and `deblank` functions will come in handy to find and get rid of such extraneous spaces.

3.1.1. Searching and replacing text

A common task related to strings is to find specific characters or substrings inside a larger string. Many functions are available for this purpose.

The simplest function determines whether two strings are the same:

```
strcmp('Jack','Jill')
ans =
     0
```

Or, to compare only the first N characters in a string:

```
strncmp('John','Joe',2)
ans =
     1
```

By default string comparisons are case sensitive, but you can make your comparisons case-insensitive with the `strcmpi` function:

```
strcmpi('John','john')
ans =
     1
```

The next step are functions that allow you to find whether a large string contains a smaller string.

```
text = 'Do you speak Matlab?';
strfind(text,'a')
ans =
    11    15    18
```

The answer means that the character "a' was found in three locations (the 11th, the 15th and the 18th spot) in the first string. If the smaller string (the second argument) cannot be found, `strfind` returns an empy vector:

```
strfind(text,'q')
ans =
     []
```

The function `strrep` allows you to replace a substring in a string with another string:

```
strrep(text,'Matlab','English')
ans=
Do you speak English?
```

Several other functions will come in handy when manipulating characters in a string: `lower` and `upper` convert strings to all lower- or uppercase. This is useful when you want to make your code case-insensitive; you simply convert string arguments all to the same case. For instance, because `strfind` is case-sensitive, it will report that there is no 'A' in the `text` string.

```
strfind(text,'A')
ans =
     []
```

To make this search case-insensitive, use:

```
strfind(upper(text),'A')
ans =
    11    15    18
```

The `fliplr` function reverses the order of the characters in a string, which is rarely useful in natural languages, but can be useful with data strings when you are looking not for the first, but the last occurrence of some string.

3.1.2. Parsing Text

Text strings, especially those used in the context of data analysis, are rarely just random collection of words. They often follow some regular pattern, or they are constructed from elementary building blocks according to some more or less logical rule. Parsing refers to the process of extracting the elements or building blocks from a longer string. A simple example would be to extract all the words from a sentence; all you need to know for this is that the building blocks (words) are always separated by spaces. To parse a relatively simple string into its building blocks, you could use the `strtok` function, for more complex strings, you'll likely need the `regexp` function. We'll discuss both here.

The `strtok` function allows you to divide a long string with elements separated by specific tokens (characters) into its constituent parts. For instance a sentence in this book is a long string of words (elements) separated by the space characeter (the token). Another example is the full name of a path:

```
pathToData = 'c:/data/2012/10/10/';
```

Here the elements are folder names, and the '/' character is the token. To extract the first element (i.e. the element before the first token):

```
strtok(pathToData','/')
ans =
c:
```

If you want more than just the first element you can ask strtok to return the remainder (i.e everything *after* the first token) and call strtok repeatedly on the remainder:

```
[element,remainder] = strtok(pathToData,'/')
element =
c:
remainder =
/data/2012/10/10/
[element,remainder] = strtok(remainder,'/')
element =
data
remainder =
/2012/10/10/
[element,remainder] = strtok(remainder,'/')
element =
2012
remainder =
/10/10/
```

To extract all elements from a path with unknown length, you will have to use a loop or the regular expression matching technique discussed next.

Regular expression matching is a fairly complicated but very flexible method to parse strings. Essentially, you tell the command which pattern you are looking for, and the `regexp` function tries to match the pattern agains the string. For example we could describe the pattern of the `pathToData` string as "numbers representing the year, month, and day, each separated by a forward slash ('/')". In Matlab this pattern is described by the following regular expression: '/\d{4}/\d{2}/\d{2}/'. \d means digits, and \d{4} means 4 digits. If the year could have 2 or 4 digits, you could specifuy \d{2,4} or you could specify that there should be at least one digit with \d+. For a complete list of the possible building blocks (characters, non-characters, digits) and the ways in which you can qualify an element (one only, one or more, zero or more) read the help documentation on `regexp`.

```
regexp(pathToData,'/\d{4}/\d{2}/\d{2}/')
ans =
     8
```

the number 8 means that this expression was matched at the 8th position in the `pathToData` string. That is good to know, but it would be more useful to extract the year, month, and day. To achieve this, we give names to each of the subexpressions, using the (?<name>expression) syntax. For instance to give the name 'year' to the \d{4} expression, you use: (?<year>\d{4}). Note how the parenthesis () group the relevant expression, and the ?<…> notaton is merely a way to specify a name ('year') for this group. The full expression becomes:

```
regexp(pathToData,'(?<year>\d{4})/(?<month>\d{2})/(?<day>\d{2})/','names')

ans =
     year: '2012'
    month: '10'
      day: '10'
```

The `'names'` option tells the regexp command to return the information as a structure where each field contains the matched value. These kinds of variables are discussed in Chapter 4.1.1 (Structures). You can ignore this for now, and just note that the `regexp` function can parse a long string and split it up into its constituent, named parts.

Constructing regular expressions is not trivial, and you should build such expressions methodically by first just matching a few of the elements of a complex string, then slowly building up by adding more and more complexity or detail. That said, mastering this technique is really useful as it is a powerful way to extract information from structured text. Once you've mastered it you may also want to look into `regexprep` which allows you to find and replace text based on regular expressions.

3.1.3. String Cell Arrays

By default Matlab stores text strings as arrays, but arrays must have the same number of columns for each row and this is not always convenient for text. You may want to store multiple experiment names in a single variable, but not all experiment names have the same length. You could use padding with spaces (like the `char` function does), but this can be inefficient.

Cell arrays are the answer to this problem: they are like arrays (they can have 1, 2, or more dimensions) but they do not require all elements to have the same size. This allows you to store a short string in one cell element and a long string in another. To distinguish cell arrays from numerical arrays Matlab uses curly brackets:

```
experiments ={'attention','move left','move right','go-nogo'}
experiments =
    'attention'    'intention'    'move left'    'move right'    'go-nogo'
```

Just like standard arrays, cell arrays can be concatenated:

```
experiments = [experiments ,'attention right']
experiments =
  Columns 1 through 5
    'attention'    'intention'    'move left'    'move right'    'go-nogo'
  Column 6
    'attention right'
```

or using the `cat` function which specifies the dimension along which we wish to concatenate. Because the example `experiments` array is a row array, we need to concatenate in the column dimension (2).

```
experiments = cat(2,experiments,'attention left')
experiments =
  Columns 1 through 5
    'attention'    'intention'    'move left'    'move right'    'go-nogo'
  Columns 6 through 7
    'attention right'    'attention left'
```

The string search and replace functions we talked about in the previous section also apply to string cell arrays. For instance, to find which of the elements of a `experiments` array are equal to the word 'attention':

```
strcmp(experiments,'attention')
ans =
     1     0     0     0     0     0     0
```

The answer means that the first entry in the cell array is a match, but the others are not. Or, to find all occurences of the word 'left':

```
strfind(experiments,'left')
ans =
    []    []    [6]    []    []    []    [11]
```

This means that the word 'left' was not found in the first two entries (empty vectors), but it was found in location 6 in the third element, and in location 11 of the last element.

Finally, you can replace strings across a whole cell array:

```
strrep(experiments,'attention','intention')
ans =
  Columns 1 through 5
    'intention'    'intention'    'move left'    'move right'    'go-nogo'
  Columns 6 through 7
    'intention right'    'intention left'
```

Cell arrays are used here to store strings, but they are in fact much more general and can contain any object or collection of objects. You could, for instance, store a string in the first row, a number in the second and a matrix in the third. While some of this flexibility can be useful at times, it violates the 'keep like with like' principle, and is likely to lead to code that is difficult to understand. So don't.

3.1.4. Excercises

1. Find the *last* token separated by '\' in 'c:\programs\matlab\bin\matlab.exe' *without using* a loop.
2. Count the characters in the variable `sentence`. Count them again, without counting the spaces.
   `sentence = 'How many characters are there in this sentence?'`
3. Count how often the character ‘e’ occurs in the `sentence` string.

4. Working with Multi-Faceted Data

- Store numbers, text, and other information, all in one place
- Keep related informaton together
- Use structures for multi-faceted data
- Use cell arrays only as a last resort

4.1.1. Structures

Multi-dimensional arrays are useful to keep data together, especially data that you may want analyze in the same way (i.e. all reaction times, all spike counts, all voltages, all BOLD signals). However, you should not use such arrays to store different kinds of data from the same experiment. For instance, if you recorded both spike counts and local field potentials recorded from multiple electrodes, then you could store spike times in page 1 of a 3D array, and local field potentials in page 2. However, this can quickly become confusing as you'd have to remember carefully which page contains what. For instance, `data(1,2,1)` represents a spike count (1st time bin, 2nd electrode) , but `data(1,2,2)` represents the local field potential.

There is nothing in this notation that makes it clear what you are accessing and hence it leads to the same problems as the use of cryptic variable names.When you write this code, you still know that '1'

represents spikes and '2' local fields, but you will not remember when you come back to this code a few months later to add a new analysis. In fact, if you have to resort to a generic variable name such as `data` then you're probably combining measurements that should not be combined. (Using a variable name `spikesAndLocalFields` would not help much…).

A better way to keep different measurements from the same experiment together is by using *structures*. A structure contains elements that can be referenced by a meaningful name. For the example above, we could define a structure `data`, which has elements (called fields) `spikeCount` and `lfp` (short for local field potential). These fields can be defined by assigning data to them:

```
data.spikeCount         = [2 3 4 10 12];
data.lfp                = [0.3 0.4 0.1 0.4 0.2];
```

Now the structure `data` contains two vectors that can be referenced by their names and treated as any ordinary vector. To determine the mean spike count, for instance:

```
mean(data.spikeCount)
ans =
    6.2000
```

In this example we have stored only the measurements in the `data` structure, but we could also add the name of the subject, the day when the experiment was done and any other information that you may need at some point during the analysis:

```
data.subject   = 'Mo';
data.date      = '01-Apr-12';
```

The information stored in the variable `data` can be retrieved just like any other variable, by typing it at the command prompt. This will, however, only show those elements that are easy to display: strings and vectors, but not matrices. For matrices contained in the structure, only the size is shown.

```
data
data =
    spikeCount: [2 3 4 10 12]
           lfp: [0.3000 0.4000 0.1000 0.4000 0.2000]
       subject: 'Mo'
          date: '01-Apr-12'
```

To take this one step further, we can now store the information on many subjects' performance in the experiment in a *structure array*:

```
data(2).subject                 = 'Yo';
data(2).date                    = '02-Apr-12'
data(2).spikeCount              = [1 1 3 100 132];
data(2).localField    = [0.1 0.2 0.3 0.2 0.3];
data =
1x2 struct array with fields:
    spikeCount
    lfp
    subject
    date
```

Now, `data` is a struct array. The first element of the array contains the information from the experiment with subject 'Mo', the second from subject 'Yo'.

Structures allow us to store all information that is relevant to a particular data analysis in one place, in one variable. This makes your code more transparent, because you always know where the information is and where it comes from. Compare for instance the notation `data(2).reactionTime` with `data(2,1)`. Both could store the same information, but the former, by using sensible variable names and the power of structures is much easier to understand than the cryptic `data(2,1)`.

The variable name `data` is still somewhat generic. In the current example, each element of the data array actually refers to a subject (each element contains all the data for a sinlge subject). So, yould improve readability by using a variable name `subject`, or `dataForSubject`. With variables like

`subject(2).spikeCount` your code becomes almost like a natural language your code almost documents itself.

There is some tension between the advice to store measurements that belong together in a single array, and the advice to use structures to combine all information on a single subject (or cell, or electrode). The best data representation depends on what kind of data you have, how many different kinds of measurement, and what you want to do with those data. For instance, if your experiments provides measurements of a single quantity (e.g. reaction times) but for many subjects, then you should store all of these data together in a single matrix. This will allow you to write vectorized code to analyze the data for all subjects at the same time. If, however, your experiment collects a large number of measurements (e.g. not just reaction times, but also skin conductance, pupil size, & heart rate), then you should probably create a structure to store the data for one subject, and store the data for all subjects in one struct array. When you are getting ready to analyze the reaction times, it is easy to create a matrix with all the subjects' reaction times from the struct array. Assume the following data set:

```
subject(1).reactionTime        = [ 100 ; 200 ;220; 190; 220];
subject(1).heartRate           = [ 1.2; 1.1 ;1.3 ;1.4 ; 1.0];
subject(1).pupilSize           = [ 0.3; 0.3; 0.4; 0.3 ; 0.3];
subject(1).skinConductance     = [ 2.3; 2.3; 2.4; 2.0; 2.8];

subject(2).reactionTime        = [ 200; 190; 240; 170; 125];
subject(2).heartRate           = [ 1.2; 1.2; 1.2; 1.1;  1.2];
subject(2).pupilSize           = [ 0.2; 0.4; 0.2; 0.2; 0.4];
subject(2).skinConductance     = [ 2.1; 2.1; 2.2; 2.1; 2.9];
```

To start analyzing the reaction times, you simply concatenate all the indivudual vectors from the structs with the command:

```
reactionTime = cat(2,subject.reactionTime)
reactionTime =
   100   200
   200   190
   220   240
   190   170
   220   125
```

What is going on here is that the expression subject.reactionTime returns a (comma separated) list of column vectors. This list is passed to the `cat` function, which combines them along the 2nd dimension. In other words the line above is equivalent to `cat(2,subject(1).reactionTime, subject(2).reactionTime)`, but it works for any number of subjects not just two. As long as you use column vectors for your data there is a nice shorthand notation for this:

```
reactionTime = [subject.reactionTime];
```

So, with this procedure you have the best of both worlds. All of the data for a single subject are stored together in a struct, all the subjects are in a single struct array, and when you get ready to analyze one aspect of the data (e.g. reactionTimes) one line of code puts that subset into a matrix. When you're ready to start analyzing the pupilSize, you just create a new variable:

```
pupilSize = [subject.pupilSize]
pupilSize =
    0.3000    0.2000
    0.3000    0.4000
    0.4000    0.2000
    0.3000    0.2000
    0.3000    0.4000
```

Note that if I had not used column but row vectors in the structure I would first concatenate them along the row dimension, and then transpose the result to get to the same starting point

```
subject(1).initials            = 'JS';
subject(1).reactionTime        = [ 100  200 220 190 220];
subject(1).heartRate           = [ 1.2 1.1 1.3 1.4  1.0];
subject(1).pupilSize           = [ 0.3 0.3 0.4 0.3  0.3];
subject(1).skinConductance     = [ 2.3 2.3 2.4 2.0 2.8];

subject(2).initials            = 'DA';
subject(2).reactionTime        = [ 200 190 240 170 125];
subject(2).heartRate           = [ 1.2 1.2 1.2 1.1  1.2];
subject(2).pupilSize           = [ 0.2 0.4 0.2 0.2 0.4];
subject(2).skinConductance   = [ 2.1 2.1 2.2 2.1 2.9];

reactionTime = cat(1,subject.reactionTime)'
reactionTime =
   100   200
   200   190
   220   240
   190   170
   220   125
```

4.1.2. Cell Arrays

As we discussed above, cell arrays are useful to store elements of unequal size. For instance, list of experiment names, subject names, paradigm names. Cell arrays, however, can do more than that. In fact, they can store any other variable (including cell arrays). Matlab will let you store the data from the example in the structure chapter as follows:

```
subjects = {'JS', [100   200   220   190   220],[ 1.2000    1.1000
1.3000    1.4000    1.0000],[0.3000    0.3000    0.4000    0.3000
0.3000],[ 2.3000    2.3000    2.4000    2.0000    2.8000];'DA',[200   190
240   170   125],[ 1.2000    1.2000    1.2000    1.1000    1.2000],[ 0.2000
0.4000    0.2000    0.2000    0.4000],[ 2.1000    2.1000    2.2000
2.1000    2.9000]}
subjects =
    'JS'    [1x5 double]    [1x5 double]    [1x5 double]    [1x5 double]
    'DA'    [1x5 double]    [1x5 double]    [1x5 double]    [1x5 double]
```

This is a cell array with 2 rows (one per subject) and 5 columns (1st colum has the initials, the following four have the values of the reaction time, heart rate, pupil size and, skin conductance, respectively. From my point of view this is a very poor data representation. The main reason is that there is no way to know what the second column means; you would have to look up the documentation of the code (ahum…) to know what those numbers mean. Using structures avoids this, and creates self-documenting code. If you are given such a structure to analyze, use the `cell2struct` command to convert it into a much more readable data representation.

There are some situtations where a cell array is useful. For instance, if you have multiple electrodes recording signals with different sampling frequencies then the data vectors for each of those electrodes will have different lengths. This prevents you from storing them together in a single array. So instead, you use a cell array:

```
lfpPerElectrode = {[0.1; 0.2; 0.1; 0.3; 0.2;0.3],[0.2; 0.1; 0.1; 0.2; 0.1;
0.3; 0.2; 0.1; 0.3; 0.1; 0.1; 0.2]}
electrode =
    [6x1 double]    [12x1 double]
```

here elecrode one had half the sample rate, hence we have only half the number of values compared to electrode 2. With this data representation we can still write reasonably compact and readable code thanks to the `cellfun` command. This function applies a function that you specify to each of the elements in a cell array. To determine the mean of the local fields of each electrode:

```
cellfun(@mean,lfpPerElectrode)
ans =
```

```
    0.2000    0.1667
```

(The @ sign creates a function handle, which is basically the name of the function, you can ignore it for now).

While `cellfun` is useful at times and creates compact code, internally, it uses loops to evaluate your function for each of the elements of the cell array. This is much slower than using vectorized code. For instance, evaluating the mean of 1000 vectors with 1000 numbers, is 240 times faster with vectorized code than using `cellfun`.

Cell arrays are definitely useful to store text and to specify graphical properties, but they are really only a last resort for numerical data. If you do end up having to use cell arrays for your data, get familiar with the `cellfun` command.

4.1.3. Sparse Arrays

If you use very large matrices whose entries are almost all zero, it is worthwhile to delve into `sparse` matrices. These matrices represent their contents in a different way than normal matrices. I.e. they assume that an entry is zero unless it is explicitly set to something else. Clearly, if you have a 1000*1000 matrix in which only the diagonal entries are non-zero, this saves a lot of storage space. In this case 1000*1000*8 bytes ~ 8Mb whereas the diagonal elements need only 100*8 ~ 800 bytes. Although the internal representation of these matrices is different, they can be used just as any other matrices (i.e. for multiplication, addition etc).

5. Manipulating Arrays

- Extract and change elements from arrays
- Reshape data arrays

This section discusses the nuts and bolts of Matlab array manipulation. You learn how to enter or change numbers in a matrix and how to extract elements of a matrix. Because scalars, vectors, matrices, high-dimensional arrays, and even text strings are all just arrays in Matlab, this manipulation works for all of them in the same way. Being able to mould matrices and to think in terms of matrices is crucial for efficient Matlab programming. The possibilities are almost limitless, and creating and manipulating matrices is almost certainly faster than enumerating entry-by-entry. Before writing a for-loop to fill or extract the elements of a matrix, make sure that it cannot be done in any of the ways described in this section.

5.1.1. Subscript Addressing

Each element of a matrix has a row number (1 or larger) and a column number (1 or larger). Elements of matrices and vectors can be identified by providing subscripts or ranges of subscripts. This is called subscript addressing. If you want to know the value of the element in the first column of the first row of a matrix `m`:

```
m= [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16; 17 18 19 20];
```

Just provide the row subscript 1 and column subscript 1:

```
m(1,1)
```

If you want to extract the first two rows of the first two columns, type:

```
m(1:2,1:2)
ans =
     1     2
     5     6
```

You might have expected that this would extract only two elements: (1,1) because the first row subscript is 1 and the first column script is 1, and then the element (2,2) because the second row subscript is 2 and the second column script is 2. Clearly Matlab does not pair-up the subscripts.

Instead, you should think of the subscripts as selecting the entire row: row subscript (1:2) selects the entire first and second row. Column subscript (1:2) selects the entire first and second column. Hence together these two subscripts select the block (submatrix) of four numbers shown above.

Sometimes, you may actually want to pair subscripts to extract only specific elements (and not submatrices) ; index addressing is one way to achieve this. We'll discuss this in the next section.
To extract *all* columns in a particular row or *all* rows in a particular column, use the ':' operator that can be read as 'all elements'. All columns in the second row are extracted by:

```
m(2,:)
ans =
     5     6     7     8
```

The operator 'end' denotes the last element in a subscript range. This allows you to extract for instance, the 2 by 2 sub-matrix in the lower right corner of m:

```
m(4:end,3:end)
ans =
    15    16
    19    20
```

Legal subscripts for matrices are between 1 and the size of the matrix in that particular dimension, otherwise there are no restrictions on the order or even on the number of times that a subscript appears in the subscript vector. This makes subscript addressing a very flexible way to manipulate your matrices. You could, for instance, take all columns (:), but only the odd rows of a matrix

```
m (1:2:5,:)
ans =
     1     2     3     4
     9    10    11    12
    17    18    19    20
```

or, you could take the third row, but r*everse* its columns:

```
m(3,4:-1:1)
ans =
    12    11    10     9
```

To expand a matrix, you can use an index multiple times. To take each element of the fourth row twice:

```
m(4,[1 1 2 2 3 3 4 4] )
ans =
    13    13    14    14    15    15    16    16
```

You may want to analyze only the later half of the trials in the reaction time experiment. For instance to determine the mean over those trials, you use:
`mean(reactionTimes(4:end,:))`
The row subscript runs from the 4th trial until the last trial (end), the column subscript (:) specifies that each subject should be included.

Just as you can extract elements with subscript addressing, you can also assign values to an element of an array.

```
m=zeros(3,4);
m(3,4)= 5
m =
     0     0     0     0
     0     0     0     0
     0     0     0     5
```

or

```
m(3:4,5) = [1 2]
m =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     5     1
     0     0     0     0     2
```

Of course the number of values you assign (the right hand side of the equation) has to match the number of positions you identify in the matrix (the left hand side). This, for instance, won't work:

```
m(3:4,5:6) = [1 2]
??? Subscripted assignment dimension mismatch.
```

because the left hand side has size 2x2, while the right hand side is only 1x2. The only exception to this rule is when you assign a scalar value. In that case every element identified on the left hand side is assigned that scalar value:

```
m(3:4,5:6) = [1]
m =
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     5     1     1
     0     0     0     0     1     1
```

5.1.2. Index Addressing

A subscript indentifies an element in an array by specifying one number for each of the dimensions of the array. So, you need 1 number for a vector (the row or the column), two for a matrix (the row and the column) and three for a 3D array (row, column, and page), etc. An index also identifies an element in an array, but does so with a single number. For this purpose, the elements in a matrix are numbered from 1 to N where N is the total number of elements. For a matrix with four rows and 5 columns, the index runs from 1 to 20. The indices 1 to 4 identify the first column, the indices 5 to 8 the second column, etc. Hence, Matlab indices snake row-first through a matrix. An example will clarify this:

```
m= [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16; 17 18 19 20]
m =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
    17    18    19    20
m(1:8)
ans =
     1     5     9    13    17     2     6    10
```

I.e. the first 8 entries in `m` are four from the first column then another four from the second column. Note that the answer no longer is a 2D matrix, but a vector.

You can easily transform from index to subscript addresses and vice versa with the functions `index = sub2ind(matrixSize, rowSubscript,columnSubscript)` and `[rowSubscript, columnSubscript] = ind2sub(matrixSize,index)`. For instance, if you want to know the index number corresponding to the fourth element in the third row of a matrix `m`, type:

```
index = sub2ind(size(m),3,4)
index =
    18
```

Index addressing is useful in combination with the `find` function (see below), or whenever the elements you want to extract from a matrix do not form a matrix themselves. For example, returning to the discussion about subscript addressing on page 19, let's say you have a list elements, identified by their row and column subscripts. In the reaction time example, this would be a list of trial numbers and subject numbers:

```
trial     = [1  3 4];
condition  = [1 1 1];
subject   = [1  2  3];
```

In words, we want only three reaction times: the one for subject 1 in trial 1 (of condition 1), subject 2 in trial 3, and subject 3 in trial 4. Subscript addressing will not work:

```
reactionTimes(trial,condition,subject)
ans(:,:,1) =
  486.8834  486.8834  486.8834
  347.0577  347.0577  347.0577
  503.1087  503.1087  503.1087
```

```
ans(:,:,2) =
  412.6024  412.6024  412.6024
  506.1127  506.1127  506.1127
  599.2746  599.2746  599.2746
ans(:,:,3) =
  411.9153  411.9153  411.9153
  437.5645  437.5645  437.5645
  463.2557  463.2557  463.2557
```

But if we first convert the subscripts to indices:

```
ix = sub2ind(size(reactionTimes),trial,condition,subject)
ix =
     1    93   184
```

then we can use these indices to get the three reaction times:

```
reactionTimes(ix)
ans =
  486.8834  506.1127  463.2557
```

Whenever the result you want is still a matrix with columns and rows that match those of the original matrix (in meaning, not number), you should probably use subscript addressing, but if the result is just a list of numbers, you need index addressing.

5.1.3. Logical Addressing

A third and very powerful method of accessing elements of matrices is called *logical addressing*. A logical address for a matrix m is another matrix, of the same size, in which each element has either the value true or false. If the logical address value is true, then the corresponding element from the matrix m is kept, if the logical address is false, the value from m is discarded. An example:

```
m
m =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
    17    18    19    20
```

Create a logical address

```
address =  [true true true false;true true true false; false false false
false; false false false false; false false false false;]
address =
     1     1     1     0
     1     1     1     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

And use this address as a logical index into `m`. Note that the `''` operator transposes the answer; this is done for cosmetic reasons only (it turns the answer into a row that will fit on a single line in this book).

```
m(address)'
ans =
     1     5     2     6     3     7
```

This shows that only the top-left corner of the matrix `m` is returned as the answer. The structure of this sub matrix is lost; the result of a logical address operation always is a vector of numbers that satisfy the conditions of the logical address. The elements in this vector are ordered by the index they had in the matrix.

Rarely will you create a logical address by typing it in. Instead, you use relational operators (see `help relop` for a complete list). For instance, extract all values in `m` larger than 10.

```
m(m>10)'
```

```
ans =
    13    17    14    18    11    15    19    12    16    20
```

To determine the mean reaction times of only the female particpants:
```
mean(reactionTimes(:,:,SUBJECTGENDER=='f'))
```
Here we extract all trials (row subscript ':'), all conditions (column subscript ':') but only those pages where the SUBJECTGENDER vector equals the character 'f'.

Assignment with logical addresses works the same way as with subscript addresses. However, because you typically do not know how many entries will be 'true' in your logical address, this is really only useful when assigning scalar values. For instance:
```
m(m<5) = 0
m =
     0     0     0     0
     5     6     7     8
     9    10    11    12
    13    14    15    16
    17    18    19    20
```
This will change all entries in `m` that are less than 5 to 0. Note that – even though `m(m<5)` would return a vector, the matrix form of `m` is maintained in this assignment.

You want to exclude reaction times that are unrealistic (for instance longer than 1000ms or shorter than 100ms).
```
reactionTimes(reactionTimes >1000 | reactionTimes <100) = NaN;
```
Note that I assigned a NaN (not-a-number), many of the data functions in the statistics toolbox are nan-aware (i.e. they will treat such entries in an array as 'missing values' and simply ignore them).

This example also shows that you can combine multiple logical adddresses into one using the logical operators OR '|' , AND ('&'), NOTt ('~') , and exclusive or (xor(a,b)). This allowed me to remove data that were too short OR too long.

Other operators that are useful in the context of logical addressing are the relational operators (help relop) and the two vector operators `any` and `all`. The `any` function returns true if any element of a vector is non-zero. Just as all other mathematical operations, the `any` function is applied to each column of a matrix. Hence, the `any` function returns `true` for each column in which any entry is non-zero.
```
any(address)
ans =
     1     1     1     0
```
As you might expect, the `all` function returns true only when all entries of the vector are non-zero:
```
all(address)
ans =
     0     0     0     0
all(1:5)
ans =
     1
```

With the `any` and `all` functions the per-column operation may not always be what you want (i.e. you may just want to know whether there is any element at all in the matrix that is non-zero). This is easily achieved by converting the matrix to a vector on the fly, using the ':' notation. For instance:
```
any(address(:))
ans =
     1
```

What happens here is that the index address (`:`) converts the 2D matrix into a single column vector (try it!), and `any` operates on that vector, so it returns a single number that corresponds to true.

Finally, there are the functions `isnan`, `isinf`, `isfinite`, which return logical matrices the same size as the argument with true at the position of NaN (Not a Number = 0/0), Infinite (1/0) and finite numbers, respectively.

```
x =[0 1 2 0 4 5 ]
nanVector = isnan(0./x)
infiniteVector = isinf(1./x)
finiteVector = isfinite(1./x)
x =
     0     1     2     0     4     5
nanVector =
     1     0     0     1     0     0
infiniteVector =
     1     0     0     1     0     0
finiteVector =
     0     1     1     0     1     1
```

These operations are useful to throw out nonsensical data points from matrices. If you read in the following vector from a data file:

```
data = [0.5 0.5 0.6 0.8 Inf Inf Inf ];
```

You could clean the data with:

```
cleanData =data(isfinite(data))
cleanData =
    0.5000    0.5000    0.6000    0.8000
```

Or, using the composition of multiple logical addresses you could decide to use only those data points that are finite and have a value less than 0.75:

```
cleanData = data(isfinite(data) & data<=0.75)
cleanData =
    0.5000    0.5000    0.6000
```

This selection can actually be done in a slightly quicker way:

```
cleanData= data(data<=0.75)
cleanData =
    0.5000    0.5000    0.6000
```

This works because Matlab knows that `Inf` is larger than 0.75. Note, however, that comparisons with NaN always fail; NaNs are not larger, not smaller, nor equal to zero or any other number.

The `isnan` function is particularly useful to find missing values in a data set and to make your own scripts nan-aware.

5.1.4. find

Logical addressing return the values in the matrix that pass some criterion (or you can use it to assign values to elements in the matrix that pass some criterion), but sometimes you need to know subscripts or indices of the relevant matrix elements. For instance, if a reaction time matrix has trials along the row dimension, and subjects along the columns, then logical addressing can extract all reaction times above a certain minimum reaction time. But it is often just as important to know which subjects had those reaction times, and in which trials.

You should use the `find` function for this; it can return an index, or the row and column subscripts for which the matrix m is non-zero.

If you ask `find` to return a single variable (by specifying a single output argument), it will return an index. For instance

```
m=magic(4)
ix = find(m>14)
m =
    16     2     3    13
```

```
     5    11    10     8
     9     7     6    12
     4    14    15     1
ix =
     1
    12
```

This means that the 1st and 12th element in the matrix are larger than 14. But if you ask `find` to return *two* variables:

```
[rowNr,colNr] = find(m>14)
rowNr =
     1
     4
colNr =
     1
     3
```

It returns the row subscript and the column subscript for each element. So this means that `m(1,1)` and `m(4,3)` are larger than 14.

Suppose you have a matrix with spike counts for electrodes located in a rectangular array and you want to know which of these electrodes recorded any spikes at all. The position in the matrix corresponds to position in your electrode array. For an electrode array with 12 electrodes arranged in a 3 by 4 rectangle you could have the following data:

```
spikeCount = [100 0 123 120;  50 90 100 0; 0  0 0 0]
spikeCount =
   100     0   123   120
    50    90   100     0
     0     0     0     0
```

To find the electrode locations where spikes were recorded:

```
[rowNr,colNr] = find(spikeCount);
rowNr'
ans =
     1     2     2     1     2     1
colNr'
ans =
     1     1     2     3     3     4
```

The variables rowNr and colNr now contain the locations of electrodes with firing cells. This technique becomes even more powerful in combination with logical operators. For instance, find all those electrodes where the firing rate is above 100.

```
[rowNr,colNr] = find(spikeCount>100);
rowNr'
ans =
     1     1
colNr'
ans =
     3     4
```

Which means that the electrode at (1,3) and the electrode at (1,4) have spike counts above 100.

You should be careful when using `find` with higher dimensional arrays. Importantly, it is *not* the case that `[i,j,k] = find(m)` returns the column, row, and page subscript in `i,j,k` respectively. Instead, the third element returned by `find` is always the non-zero value that `find` found. Practically, for any multi-dimensional array you will have to use `find` with one output argument and, if needed, convert this index into multiple subscripts using `ind2sub`.

Find all trials in which a subject had a reaction time above a minimal value.

```
 MINREACTIONTIME = 950;
[ix]=find(reactionTimes>MINREACTIONTIME )
ix =
   786
   879
```
This means that entry number 786 and 879 in the reactionTimes matrix had a value above `MINREACTIONTIME`*. If you need to know which trial, condition, and subject this index corresponds to, use the ind2sub function:*
```
[trialNr,conditionNr,subjectNr] = ind2sub(size(reactionTimes),ix)
trialNr =
     6
     9
conditionNr =
     3
     3
subjectNr =
     9
    10
```

Which means that subjects 9 and 10 each had a single trial in which their reaction time was above MINREACTIONTIME.

5.1.5. Reshaping matrices

Sometimes you will have to reshape a matrix. The transpose operation (`'`) which exchanges rows and columns is probably the simplest example.Other examples are functions that can flip a matrix upside down: `flipud` (i.e. the last row becomes the first row), or left to right: `fliplr` (the rightmost column becomes the leftmost column).

More generally, you can use the `reshape` function to change any array into an array with a different number of rows and columns as long as the total number of elements stays the same.

You want to analyze the reaction time data without taking the different conditions into account.
```
[nrTrials, nrConditions, nrSubjects] = size(reactionTimes)
nrTrials =
    30
nrConditions =
     3
nrSubjects =
    10
```
In other words, you want to reshape this reactionTimes matrix such that the rows represent each trial (regardless of condition) and the columns represent the subject.
```
reactionTimesAllConditions = reshape(reactionTimes, nrTrials*nrConditions,
nrSubjects);
size(reactionTimesAllConditions)
ans =
    90    10
```

What happens behind the scenes is that Matlab takes the first `nrTrials*nrConditions` (90) entries in the array, places them in one column, and then takes the next 90 entries, places them in the next column etc. In essence this uses index-addressing to change the matrix.

An extreme variant of reshape uses the colon operator, which reshapes any matrix into a single column vector. Hence `m(:)` is equivalent to `reshape(m,numel(m),1)`. Applied to vectors, this operator transforms the vector into a column vector. This can be a useful first operation in a function M-file. By applying this operator to a vector that was passed as an argument, you can be sure that the

vector in the function file is a column vector, even if the person using this function passed a row vector.

To eliminate parts of a matrix you assign the empty matrix. For instance, you discover that the second row of electrodes in your array was broken. You use the empty matrix [ ] to eliminate parts of a matrix.

```
spikeCount(2,:) =[]
spikeCount =
   100     0   123   120
     0     0     0     0
```

Deletion is only allowed if the resulting output would still be a matrix. Eliminating a single element therefore leads to an error:

```
spikeCount(1,2) =[]
??? Subscripted assignment dimension mismatch.
```

5.1.6. Cell Arrays

Cell arrays can be manipulated in very similar ways as arrays: subscripts, indices, and logical addresses all work in essentially the same way. The only thing that is different is the use of the brackets. When you use curly brackets, you extract the data stored inside the cell. For instance, from a 2x2 cell array, you can use subscripting to extract the data stored in the 1st row, 2nd column:

```
experiments ={'attention','move left'; 'move right','go-nogo'}
experiments{1,2}
experiments =
    'attention'     'move left'
    'move right'    'go-nogo'
ans =
move left
```

The `ans` variable now contains the string that was stored in the cell array at position (1,2). This is what you typically use if you want to work with one or more elements of the cell array. Note that if you ask for multiple elements to be returned, they are returned as a list:

```
experiments{1:2}
ans =
attention
ans =
move right
```

When passed to a function Matlab interprets such a list as a comma separated list of arguments. In other words, fun(experiments{1},experiments{2}) is the same as fun(experiments{1:2}). This is a convenient way to pass many arguments to a function:

```
char(experiments{1:2})
ans =
attention
move right
```

This is particularly useful if you want to pass all elements of a cell array to a function, but the number of elements is not always the same:

```
char(experiments{:})
ans =
attention
move right
move left
go-nogo
```

5.1.6.1. Addressing Cell Arrays

In the examples above we extracted data stored in a cell array. This requires the use of curly brackets:

```
experiments{1,1}
ans =
attention
```

The result is a string, because that is what was stored in the first element of the cell array. Sometimes, however, you may not want to extract the data from the cell array but manipulate the array (remove an element, or create an array that consists of a subset of elements). For this purpose you use standard parentheses:

```
experiments(1,2)
ans =
    'move left'
```

This looks almost the same as the result of `experiments{1,2}` above, but here `ans` actually is a 1x1 cell array whose content is a string ('move left').

You also use parentheses to *remove* elements from a cell array. For instance, to remove the first row of elements:

```
experiments(1,:) = []
experiments =
    'move right'    'go-nogo'
```

If you use a curly bracket instead, then the element of the cell array will have an empty matrix as its *content*.

```
experiments{1,1} = []
experiments =
   []    'go-nogo'
```

5.1.7. Exercises

6 When analyzing saccade latency data you realize that some of the latencies seem rather unlikely: throw out those trials for which the saccade latencies is larger than 0.5 s and those smaller than 0.05s. Now determine the mean again. Did you determine the mean per subject?

7 You perform a reaction time experiment and represent your data in a matrix in which each column represents the results of a single subject. Each row represents a repeated trial. We want to find out when the reaction time is shortest. First, you decide that reaction times shorter than 0.2s are not trustworthy; find them, and set them to NaN. Then, for each subject find the trial at which the reaction time is shortest, and determine the mean trial number (across subjects) in which the shortest reaction time occurs (if this number is large, this suggests some kind of learning effect). You can generate fake date for this exercise using data = rand(100,5); which fakes 100 trials each for 5 subjects.

8 We have a vector with all reaction times recorded during an experiment. There were 5 conditions in the experiment, with 3 trials each. The conditions were recorded consecutively (subject 1 trial 1 for condition 1 then trial 2 for condition 1 etc.). The data for each of the 4 subjects is specified in a single list of numbers. Reshape this vector into a 3D matrix whose rows represent trials, each column is a condition, and pages are subjects. To generate fake date for this assignment, use `vector = rand(1,75);`

For instance, for a smaller set of 2 trials, 2 conditions, and 2 subjects the data would be

```
vector =[0.1233    0.1839    0.2400    0.4173    0.0497    0.9027
0.9448    0.4909];
```

and the data matrix would be

```
    0.1233    0.2400
    0.1839    0.4173
```

for the first page and

```
    0.0497    0.9448
    0.9027    0.4909
```

for the second page.

9 Create a 4D array with all 1's.

10 Create a 3D array with one 3x2 page of 1's, one 3x2 page of 2's, and one 3x2 page of 3's. Functions you could use are `repmat`, `ones` and `cat`. Then try again using only `repmat` and `permute`.

11 You performed an experiment in which you measured the detection performance of 5 subjects in 3 trials for each of 4 conditions. The data are:

```
m=rand(3,5,4)
```

Find the largest number in m.

Determine which subject has the overall largest value. You'll need the `max` or `find`, and `ind2sub` functions.

In a second analysis you want to know in which condition each subject performed best. (I.e. find the condition per subject). Use the `max` and `ind2sub` functions.

12 You did an experiment in which you recorded from 5 cells, during 10 trials, and there were 10 time bins (of 10ms each) in each trial.

Generate fake data for a neuron that spiked with poisson probability with a mean of 3 spikes per 10ms bin (see `help poissrnd`)

Store the data in an appropriate array for analysis.

Determine the mean firing rate per trial.
Determine the mean firing rate per 20ms time bin (use `reshape`).

13 You recorded extracellular action potentials from 3 cells simultaneously, each cell has a unique ID, a recording date, some comments about the quality of isolation, and your data acquisition program provides you with the firing rates for each of the 5 bins for each of the 10 trials that you recorded in each of the 3 conditions.

Create a struct to store the data for one cell and a struct array to combine the data from all three cells. Fill the structure with made up numbers (use `rand`) and text.

From the struct array, extract/create a multi-dimensional array with just the firing rates. Determine the mean firing rate across cells, for each condition. And determine the mean firing rate across trials, for each cell and condition.

14 Create a structure named Experiment. It should have subjectName, intensity and response as fields. This is a detection experiment in whicht the intensity of a stimulus was varied. Make up subject names.

Each subject runs 50 trials and in each trial the response (0: not detected, 1: detected) is stored. Generate the response field with randomly chosen 0/1, using the `randi` function. Each trial has an intensity between 1 and 10. Generate the intensity field such that each value between 1 and 10 is repeated 5 times in random order. (use `randperm`)

Now we'll create some errors in the data:

- Introduce a ';' character in one of the subject names.

- Assign 'NaN' to any two 'intensity' values for each subject.

Once you created this structure:

- Identify the 'response' that has NaN as its corresponding 'intensity' and set it to NaN.

- Identify the subject name with a ‘;’ and remove the ‘;’ from the name

- Extract an appropriate arrray from the struct array such that you can easily compute the proportion correct for each condition (and per subject).