Programming with Matlab

Eleftherios Gkioulekas
Mathematical Sciences Computing Center
University of Washington

December, 1996

1 Starting Matlab

Matlab is an interactive tool that includes facilities for dealing with numerical analysis, matrix computation, signal processing and graphics. It is meant to be used to understand and test mathematical concepts interactively before coding in a real programming language.

Throughout this tutorial, we will give you an overview of various things and refer you to Matlab's on-line help for more information. The best way to learn is by experimentation, and the best way this tutorial can help you is by telling you the basics and giving you directions towards which you can take off exploring on your own.

To start Matlab type `matlab` on the shell prompt. You get a greeting on your screen, then a window pops up and goes away [1] and finally you get a `>>` prompt. At this point you can start typing in commands on interactive mode or you can quit by typing `quit` at the prompt. You can also navigate around the filesystem, execute the Matlab programs you have written, and get online help. If you want to log your what you do throughout your matlab session type

```
>> diary filename
```

where in `filename` you put the filename you want to use for your log. To stop logging type

```
>> diary off
```

Moreover you can get on-line help by typing

```
>> help
```

on the prompt. A list of topics appears. You can then type `help` again followed with the name of the topic and get more information about what you want to know. For example suppose you type:

```
>> help
[..lots of stuff..]
Matlab/matfun        -  Matrix functions - numerical linear algebra.
[..lots of other stuff..]
```

[1]The reason this happens is because Matlab tries to see if you have an X display. It can do that without popping up a window, but then you wouldn't be as impressed

There you go! Say you want to learn more about the matrix stuff. You type

```
>> help matfun

 Matrix functions - numerical linear algebra.

 Matrix analysis.
   cond        - Matrix condition number.
   norm        - Matrix or vector norm.
[etc...etc...]
```

and you are given a list of commands that are available to you for working with matrices. Then, to learn about the `cond` command you type

```
>> help cond
```

and you will be told how to use it to compute condition numbers. For more information about getting help try:

```
>> help help
```

Another way of getting help is with the `lookfor` command. Suppose you want to see what Matlab can do with "eigenvalues". Typing:

```
>> lookfor eigenvalue
```

will return to you a list of commands in which the word "eigenvalue" occurs. Note that this command is not very artificially intelligent. If you are looking for your son, typing

```
>> lookfor son
```

will not give you what you want

If you are on a machine that can run a web browser, try also:

```
>> doc
```

Finally, if Matlab ever does something you don't understand type:

```
>> why
```

for a succinct explanation.

Like we said, at the prompt you can also execute your programs. So, let's do that!. Bring up an editor [2] and type in the following Matlab program:

[2] On a Unix system you can use the following editors: `emacs`, `vi` and if available `pico`. On DOS there is `edit`. Other systems have (or are supposed to have) a **`text editor`**. Warning: You can NOT use a word-processor to type in Matlab programs, unless the word processor allows you to save your document as pure text.

```
%
% These are comments. In Matlab you can add comments with a % character
% This is the standard hello world program

disp('Hello world!');
```

Save the program with the filename `hello.m`. You want all your Matlab programs to be saved with the extension `.m` at the end. Then start Matlab *under the same directory where you saved the file* and type `hello` at the Matlab prompt. Then you should see the following:

```
>> hello
Hello world!
>>
```

When you terminate a Matlab command with a semicolon, the command will execute silently. When you don't, the command will print back a response. In a matlab program, the former behaviour is desirable. When using Matlab interactively you may want to see these responses. As a general rule of thumb, when you write Matlab programs terminate every statement with a semi-colon and produce the output of interest by invoking the commands whose job is to print things. Such a command would be `disp` for example, which will print `hello` even though it is being "silenced" with a semicolon. We will learn about another printing command later on.

Notice that within the Matlab environment you don't have to type any funny commands to *load* your program. When you start Matlab from your shell, Matlab will load *all* the `*.m` files that happen to be under the present working directory at startup time. So, all you have to do when the prompt shows up is to type the name of the program (*without the* `*.m` *extension*) and it will get executed. Note that Matlab will only look for files with the `*.m` extension, so you are forced to use it. There are no work-arounds for this. Still, Matlab has a provision for the situation where your files are scattered in more than one directory. You can use the Unix `cd, ls, pwd` commands to navigate in the file system and change your current working directory. Also, if you want to be able to have access at the files on two or more seperate directories *simultaneously* type

```
>> help path
```

for more information.

2 Matlab variables

Matlab has three basic data types: strings, scalars and matrices. Arrays are just matrices that have only one row. Matlab has also lots of built-in functions to work with these things. You have already seen the `disp` function in our hello-world program.

Starting with strings, you can assign a string to a variable like this:

```
name = 'Indiana Jones';
```

Note that it is a syntax error to quote the string with anything other than the forward quote marks. So, the following are **wrong!**

```
name = "Indiana Jones";      wrong!
name = ‘Indiana Jones‘;      wrong!
```

In a Matlab program you can prompt the user and ask him to enter in a string with the `input` command:

```
% This is a rather more social program
%
yourname = input('Hello! Who are you? ','s');
dadname  = input('What's your daddy name? ','s');
fprintf(1,'Hail oh %s son of %s the Great! \n',yourname,dadname);
```

The `input` command takes two arguments. The first argument is the string that you want the user to be prompted with. You could stick in a variable instead of a fixed string if you wanted to. The second argument tells Matlab to expect the user to enter a string. If you omit the second argument, then Matlab will be expecting a number, and upon you entering your name, Matlab will complain. Finally, it *returns* the value that the user enters, and that value is passed on through assignment to the variable `yourname`.

The `fprintf` command gives you more flexibility in producing output than the `disp` command. There is **a lot** to learn about `fprintf` so please type `help fprintf` to learn all you need to know. Briefly, `fprintf` takes two or three or more arguments. The first argument is a *file descriptor.* File descriptors are integers that reference places where you can send output and receive input from. In Matlab, file descriptor 1 is what you use when you want to send things to the screen. The terminology you may hear is that file descriptor 1 sends things to the *standard output.*

The rest of the arguments depend on what you want to print. If all you want to print is a fixed string, then you just put that in as your second argument. For example:

```
fprintf(1,'Hello world!\n');
```

The \n sequence will switch you over to the next line. `disp` will do this automatically, in `fprintf` you must explicitly state that you wish to go to a new line. This is a feature, since there may be situations where you do *not* want to go to a new line.

If you want to print the values of variables interspersed with your string then you need to put appropriate markers like `%s` to indicate where you want your variables to go. Then, in the subsequent arguments you list the variables in the appropriate order, making sure to match the markers. There are many markers and the Matlab online help will refer you to a C manual. The most commonly used markers are the following:

`%s`	Strings
`%d`	Integers (otherwise you get things like 5.0000)
`%g`	Real numbers in scientific notation.

In our example above, we just used `%s`. You will see further examples later on.

Note that if you merely want to print a variable, it is better to use `disp` since it will format it for you. `fprintf` is more useful for those occasions where you want to do the formatting yourself as well as for sending things to a file.

Scalars can be assigned, inputed and printed in a similar fashion. Here is an example:

```
% yet another one of these happy programs
age = input('Pardon for asking but how old are you?');
if (age < 75)
 life_left = 365.25*24*(75 - age);
 fprintf(1,'You have %g hours left of average life expectancy.\n',life_left);
else
 fprintf(1,'Geez! You are that old?!\n');
end
fprintf(1,'Live long and prosper!\n');
```

Note the following:

- String and numeric variables look the same. You don't have to declare the type of the variable anywhere. Matlab will make sure to do *the right thing* (tm).
- When we use `input` to get the value of a numeric variable we omit the second `'s'` argument. This way, Matlab will do error-checking and complain if you entered something that's not a number.
- You can use `fprintf` to print numeric variables in a similar fashion, but you got to use the `%g` marker. If you are printing an integer you must use the `%d` marker, otherwise Matlab will stick in a few zeroes as decimal places to your integer. It is obvious that you can mix strings and numbers in an `fprintf` command, so long as you don't mix up the order of the variables listed afterwards.
- On line

  ```
  life_left = 365.25*24*(75 - age);
  ```

  we see how you can do simple computations in Matlab. It's very similar to C and Fortran and to learn more about the operators you have available type

  ```
  >> help ops
  >> help relops
  ```
- Finally, we have an example of an `if` statement. We will talk of that more later. The meaning should be intuitively obvious.

In addition to ordinary numbers, you may also have complex numbers. The symbols `i` and `j` are reserved for such use. For example you can say:

```
z = 3 + 4*i;
```

or

```
z = 3 + 4*j;
```

where `i` and `j` represent $\sqrt{-1}$. If you are already using the symbols `i` and `j` as variables, then you can get a new complex unit and use it in the usual way by saying:

```
ii = sqrt(-1);
z = 3 + 4*ii;
```

3 Arrays in Matlab

Next we talk about arrays. In Matlab arrays are dynamic and they are indexed from 1. You can assign them element by element with commands like:

```
a(1) = 23;
a(2) = input('Enter a(2)');
a(3) = a(1)+a(2);
```

It is a syntax error to assign or refer to `a(0)`. This is unfortunate since in some cases the 0-indexing *is* more convenient. Note that you don't have to initialize the array or state it's size at any point. The array will make sure to grow itself as you index higher and higher indices.

Suppose that you do this:

```
a(1) = 10;
a(3) = 20;
```

At this point, `a` has grown to size 3. But `a(2)` hasn't been assigned a value yet. In such situations, during growth any unset elements are set to zero. It is good programming practice however not to depend on this and always initialize all the elements to their proper values.

Notice that for the sake of efficiency you might not like the idea of growing arrays. This is because every time the array is grown, a new chunk of memory must be allocated for it, and contents have to be copied. In that case, you can set the size of the array by initializing it with the `zeros` command:

```
a = zeros(100);
```

This will set `a(1),a(2),...,a(100)` all equal to zero. Then, so long as you respect these boundaries, the array will not have to be grown at any point.

Here are some other ways to make assignments to arrays:

```
x = [3 4 5 6];
```

will set x equal to an array of 4 values. You can recursively add elements to your array x in various ways if you include x on the right hand side. For example, you can make assignments like

```
x = [x 1 2]     % append two elements at end of the array
x = [1 2 x 3 ] % append two elements at front, one at back
```

How about making deletions? Well, first of all notice that we can access parts of the array with the following indexing scheme:

```
y = x(2:4);
```

will return the an array of `x(2), x(3), x(4)`. So, if you want to delete the last element of the array, you just have to find the size of the array, which you can do with the `size` command.

Yet another way to setup arrays is like this:

```
x = 3 : 1 : 6;
```

This will set x equal to an array of equidistant values that begin at 3, end at 6 and are separated from each other by steps of 1. You can even make backwards steps if you provide a negative stepsize, like this:

```
x = 6 : -1 : 3;
```

It is common to set up arrays like these when you want to plot a function whose values are known at equidistant points.

Finally, to conclude, you may want to know how to load arrays from files. Suppose you have a file that contains a list of numbers separated with carriage returns. These numbers could be the values of a function you want to plot on known values of x (presumably equidistant). You want to load all of these numbers on a vector so you can do things to them. Here is a demo program for doing this:

```
filename = input('Please enter filename:','s');
fd = fopen(filename);
vector = fscanf(fd,'%g',inf);
fclose(fd);
disp(vector);
```

Here is how this works:

- The first line, prompts the user for a filename.
- The `fopen` command will open the file for reading and return a file descriptor which we store at variable `fd`.
- The `fscanf` command will read in the data. You really need to read the help page for `fscanf` as it is a very useful command. In principle it is a little similar to `fprintf`. The first argument is the file descriptor from which data is being read. The second argument tells Matlab, what kind of data is being read. The `%g` marker stands for real numbers in scientific notation. Finally the third argument tells Matlab to read in the entire file in one scoop. Alternatively you can stick in an integer there and tell Matlab to load only so many numbers from the file.
- The `fclose` command will close the file descriptor that was opened.
- Finally the `disp` command will show you what has been loaded. At this point you could substitute with somewhat more interesting code if you will.

Another common situation is data files that contain pairs of numbers separated by carriage returns. Suppose you want to load the first numbers onto one array, and the second numbers to another array. Here is how that is done:

```
filename = input('Please enter filename: ','s');
fd = fopen(filename);
A  = fscanf(fd,'%g %g\n',[2,inf]);
x  = A(1,:);
y  = A(2,:);
fclose(fd);
disp('Here comes x:'); disp(x);
disp('Here comes y:'); disp(y);
```

Again, you need to read the help page for `fscanf` to understand this example better. You can use it in your programs as a canned box until then. What we do in this code snippet essentially is to load the file into a two-column matrix, and then extract the columns into vectors. Of course, this example now leads us to the next item on the agenda: matrices.

4 Matrices in Matlab

In Matlab, arrays are matrices that have only one row. Like arrays, matrices can be defined element by element like this:

```
a(1,1) = 1; a(1,2) = 0;
a(2,1) = 0; a(2,2) = 1;
```

Like arrays, matrices grow themselves dynamically as needed when you add elements in this fashion. Upon growth, any unset elements default to zero just like they do in arrays. If you don't want that, you can use the `zeros` command to initialize the matrix to a specific size and set it equal to zero, and then take it from there. For instance, the following example will create a zero matrix with 4 rows and 5 columns:

```
A = zeros(4,5);
```

To get the size of a matrix, we use the `size` command like this:

```
[rows,columns] = size(A);
```

When this command executes, the variable `rows` is set equal to the number of rows and `columns` is set equal to the number of columns. If you are only interested in the number of rows, or the number of columns then you can use the following variants of `size` to obtain them:

```
rows = size(A,1);
columns = size(A,2);
```

Since arrays are just matrices with one row, you can use the `size(array,2)` construct to get hold of the size of the array. Unfortunately, if you were to say:

```
s = size(array);   % wrong!
```

it would be wrong, because this returns both the number of rows and columns and since you only care to pick up one of the two numbers, you pick up the number of rows, which for arrays is always equal to 1. Not what you want!

Naturally, there are a few other ways to assign values to a matrix. One way is like this:

```
A = [ 1 0 0 ; 0 1 0 ; 0 0 1]
```

This will set `A` equal to the 3 by 3 identity matrix. In this notation you list the rows and separate them with semicolons.

In addition to that you can extract pieces of the matrix, just like earlier we showed you how to extract pieces of the arrays. Here are some examples of what you can do:

```
a = A(:,2);        % this is the 2nd column of A
b = A(3,:);        % this is the 3rd row of A
c = A(1:4,3);      % this is a 4 by 1 submatrix of A
d = A(:,[2 4 10]); % this is the 2nd, 4th and 10th columns of A stacked
```

In general, if `v` and `w` are arrays with integer components, then `A(v,w)` is the matrix obtained by taking the elements of A with row subscripts from `v` and column subscripts from `w`. So:

```
n = size(A,2);
A = A(:,n:-1:1);
```

will reverse the columns of A. Moreover, you can have **all** of these constructs appear on the left hand side of an assignment and Matlab will do the right thing. For instance

```
A(:,[3 5 10]) = B(:,1:3)
```

replaces the third, fifth and tenth columns of A with the first three columns of B.

In addition to getting submatrices of matrices, Matlab will allow you to put together block matrices from smaller matrices. For example if `A,B,C,D` are a square matrices of the same size, then you can put together a block matrix like this:

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

Finally, you can get the transpose of matrix by putting a `'` mark next to it. For example:

```
A = [ 2 4 1 ; 2 1 5 ; 4 2 6];
Atrans = A';
```

Matlab provides functions that return many special matrices. These functions are listed in Figure 1 and we urge you to look these up with the `help` command and experiment.

To display the contents of matrices you can simply use the `disp` command. For example, to display the 5 by 5 Hilbert matrix you want to say:

`zeros`	Returns the zero matrix
`ones`	Returns a matrix in which all entries are set equal to one
`rand`	A matrix with uniformly distributed random elements
`randn`	A matrix with normally distributed random elements
`eye`	The identity matrix
`compan`	Computes the companion matrix
`diag`	Extract one of the diagonals of a matrix
`gallery`	Returns a couple of small test matrices. See help page.
`hadamard`	Returns a Hadamard matrix of any order where N, N/12 or N/20 is a power of 2
`hilb`	Returns the Hilbert matrices
`invhilb`	Returns the inverse of Hilbert matrices
`pascal`	Returns Pascal's triangle
`toeplitz`	Returns Toeplitz matrices
`vander`	Returns Vandermonde matrices.

Figure 1: Matrix commands

```
>> disp(hilb(5))
    1.0000    0.5000    0.3333    0.2500    0.2000
    0.5000    0.3333    0.2500    0.2000    0.1667
    0.3333    0.2500    0.2000    0.1667    0.1429
    0.2500    0.2000    0.1667    0.1429    0.1250
    0.2000    0.1667    0.1429    0.1250    0.1111
```

The Hilbert matrix is a famous example of a badly conditioned matrix. It is also famous because the exact inverse of it is known analytically and can be computed with the `invhilb` command:

```
>> disp(invhilb(5))
            25          -300          1050         -1400           630
          -300          4800        -18900         26880        -12600
          1050        -18900         79380       -117600         56700
         -1400         26880       -117600        179200        -88200
           630        -12600         56700        -88200         44100
```

This way you can interactively use these famous matrices to study concepts such as ill-conditioning.

5 Matrix/Array Operations

- **Matrix addition/subtraction**: Matrices can be added and subtracted like this:

  ```
  A = B + C;
  A = B - C;
  ```

  It is necessary for both matrices `B` and `C` to have the same size. The exception to this rule is when adding with a scalar:

```
A = B + 4;
```

In this example, all the elements of `B` are increased by 4 and the resulting matrix is stored in `A`.

- **Matrix multiplication**: Matrices `B` and `C` can be multiplied if they have sizes $n \times p$ and $p \times m$ correspondingly. The product then is evaluated by the well-known formula

  $$A_{ij} = \sum_{k=1}^{p} B_{ik} C_{kj},\ \forall i \in \{1, \dots, n\},\ \forall j \in \{1, \dots, m\}$$

  In Matlab, to do this you say:

  ```
  A = B*C;
  ```

  You can also multiply all elements of a matrix by a scalar like this:

  ```
  A = 4*B;
  A = B*4;  % both are equivalent
  ```

  Since vectors are matrices that are $1 \times n$, you can use this mechanism to take the dot product of two vectors by transposing one of the vectors. For example:

  ```
  x = [2 4 1 5 3];
  y = [5 3 5 2 9];
  p = x'*y;
  ```

  This way, `x'` is now a $n \times 1$ "matrix" and `y` is $1 \times n$ and the two can be multiplied, which is the same as taking their dot product.

  Another common application of this is to apply $n \times n$ matrices to vectors. There is a catch though: In Matlab, vectors are defined as matrices with one row, i.e. as $1 \times n$. If you are used to writing the matrix product as Ax, then you have to transpose the vector. For example:

  ```
  A = [1 3 2; 3 2 5; 4 6 2];   % define a matrix
  x = [2 5 1];                 % define a vector
  y = A*x;                     % This is WRONG!
  y = A*x';                    % This is correct :)
  ```

- **Array multiplication:** This is an alternative way to multiply *arrays*:

  $$C_{ij} = A_{ij} B_{ij},\ \forall i \in \{1, \dots, n\},\ \forall j \in \{1, \dots, m\}$$

  This is not the traditional matrix multiplication but it's something that shows up in many applications. You can do it in Matlab like this:

```
C = A.*B;
```

- **Array division:** Likewise you can divide arrays in matlab according to the formula:

$$C_{ij} = \frac{A_{ij}}{B_{ij}},\ \forall i \in \{1, \ldots, n\},\ \forall j \in \{1, \ldots, m\}$$

  by using the `./` operator like this:

```
C = A./B;
```

- **Matrix division:** There are two *matrix division* operators in Matlab: `/` and `\`. In general

```
X = A\B   is a solution to   A*X = B
X = A/B   is a solution to   X*A = B
```

  This means that `A\B` is defined whenever `B` has as many rows as `A`. Likewise `A/B` is defined whenever `B` has as many columns as `A`. If `A` is a square matrix then it is factored using Gaussian elimination. Then the equations `A*X(:,j) = B(:,j)` are being solved for every column of `B`. The result is a matrix with the same dimensions as `B`. If `A` is not square, it is factored using the Householder orthogonalization with column pivoting. Then the corresponding equations will be solved in the least squares fit sense. Right division `A/B` in Matlab is computed in terms of left division by `A/B = (A'\B')'`. For more information type

```
>> help slash
```

- **Matrix inverse:** Usually we are not interested in matrix inverses as much as applying them directly on vectors. In these cases, it's best to use the matrix division operators. Nevertheless, you can obtain the inverse of a matrix if you need it by using the `inv` function. If `A` is a matrix then `inv(A)` will return it's inverse. If the matrix is singular or close to singular a warning will be printed.

- **Matrix determinants:** Matrix determinants are defined by

$$\det(A) = \sum_{\sigma \in S_n} \left\{ \operatorname{sign}(\sigma) \prod_{i=1}^{n} A_{i\sigma(i)} \right\}$$

  where S_n is the set of permutations of the ordered set $(1, 2, \ldots, n)$, and $\operatorname{sign}(\sigma)$ is equal to $+1$ if the permutation is even and -1 if the permutation is odd. Determinants make sense only for square matrices and can be computed with the `det` function:

```
a = det(A);
```

- **Matrix exponential function:** These are some very fascinating functions. The *matrix exponential* function is defined by

  $$\exp(A) = \sum_{k=0}^{+\infty} \frac{A^k}{k!}$$

  where the power A^k is to be evaluated in the *matrix product* sense. Recall that your ordinary exponential function is defined by

  $$e^x = \sum_{k=0}^{+\infty} \frac{x^k}{k!}$$

  which converges for all x (even when they are complex). It is **not** obvious from this that the corresponding matrix expression also converges. But it does, and the result is the *matrix exponential.* The matrix exponential can be computed in Matlab with the `expm` function. It's usage is as simple as:

  ```
  Y = expm(X);
  ```

  Matrix exponentials show up in the solution of systems of differential equations.

Matlab has a *plethora* of commands that do almost anything that you would ever want to do to a matrix. And we have only discussed a subset of the operations that are permitted with matrices. The following `help` calls should be helpful in exploring what Matlab offers:

```
>> help elmat
>> help matfun
>> help sparfun
```

We will go into a detailed discussion about matrices and linear algebra in Matlab, on a separate tutorial.

6 Flow control in Matlab

So far, we have spent most of our time discussing the data structures available in Matlab, and how they can be manipulated, as well as inputted and outputed. Now we continue this discussion by discussing how Matlab deals with *flow control.*

- **For loops:** In Matlab, a for-loop has the following syntax:

  ```
  for v = matrix
   statement1;
   statement2;
   ....
  end
  ```

The columns of the matrix are stored one at a time in the variable, and then the statements up to the `end` statement are executed. If you wish to loop over the rows of the matrix then you simply transpose it and plug it in. It is recommended that the commands between the `for` statement and the `end` statement are indented by one space so that the reader of your code can visually see that these statements are enclosed in a loop.

In many cases, we use arrays for matrices, and then the for-loop reduces to the usual for-loop we know in languages like Fortran and C. In particular using expressions of the form `X:Y` will effectively make the loop variable be a scalar that goes from `X` to `Y`. Using an expression of the form `start:step:stop` will allow you to loop a scalar from value `start` all the way to value `stop` with stepsize `step`. For instance the Hilbert matrix is defined by the equation:

$$A_{ij} = \frac{1}{i + j + 1}$$

If we didn't have the `hilb` command, then we would use *for-loops* to initialize it, like this:

```
N = 10;
A = zeros(N,N);
for i = 1:N
 for j = 1:N
  A(i,j) = 1/(i+j-1);
 end
end
```

The same code can be rewritten more consisely like this:

```
N = 10; A = zeros(N,N);
for i = 1:N , for j = 1:N , A(i,j) = 1/(i+j-1); , end, end
```

Note that the `for` and `end` statements in the long-winded version of the example are being separated by *newlines* and not semicolons. To obtain the consise version, we merely substitute those newlines with *commas*.

- **While loops**: In Matlab while loops follow the following format:

  ```
  while variable
   statement1;
   statement2;
   ....
   statementn;
  end
  ```

  where `variable` is almost always a *boolean expression* of some sort. In Matlab you can compose boolean expressions as shown in Figure 2.

  Here is an example of a Matlab program that uses the while loop:

`a == b`	True when `a` equals `b`
`a > b`	True when `a` is greater than `b`
`a < b`	True when `a` is smaller than `b`
`a <= b`	True when `a` is smaller or equal to `b`
`a >= b`	True when `a` is greater or equal to `b`
`a  = b`	True when `a` is not equal to `b`
`a & b`	True when both boolean expressions `a` and `b` are true
`a | b`	True when at least one of `a` or `b` is true.
`a xor b`	True only when only one of `a` or `b` is true.
`a`	True when `a` is false.

Figure 2: Some boolean expressions in Matlab

```
n = 1;
while prod(1:n) < 1.e100
 n = n + 1;
end
disp(n);
```

This program will display the first integer for which $n!$ is a 100-digit number. The `prod` function takes an array (or matrix) as argument and returns the product of it's elements. In this case, `prod(1:n)` is the factorial $n!$.

- **If and Break statements:** The simplest way to set-up an `if` branch is like this:

```
if variable
 statement1;
 ....
 statementn;
end
```

The statements are executed only if the real part of the `variable` has all non-zero elements. [3] Otherwise, the program continues with executing the statements right after the `end` statement. The variable is usually the result of a boolean expression. The most general way to do if-branching is like this:

```
if variable
 statement1;
 ......
```

[3]Note that in the most general case, `variable` could be a complex number. The `if` statement will only look into its real part

```
 statementn;
elseif variable2
 statement1;
 ......
 statementn;
[....as many elseifs as you want...]
else
 statement1;
 ......
 statementn;
end
```

In this case, if `variable` is true, then the statements right after it will execute until the first `else` or `elseif` (whichever comes first), and then control will be passed over to the statements after the `end`. If `variable` is not true, then we check `variable2`. Now, if that one is true, we do the statements following thereafter until the next `else` or `elseif` and when we get there again we jump to `end`. Recursively, upon consecutive failures we check the next `elseif`. If all the `elseif` variables turn out to be false, then we execute the statements after the `else`. Note that the `elseif`s and/or the `else` can be omitted all together.

Here is a general example that illustrates the last two methods of flow control.

```
% Classic 3n+1 problem from number theory
while 1
 n = input('Enter n, negative quits. ');
 if n <= 0, break, end
 while n > 1
  if rem(n,2) == 0 , n = n/2;
  else, n = 3*n+1;
  end
 end
end
```

This example involves a fascinating problem from number theory. Take any positive integer. If it is even, divide it by 2; if it is odd, multiply it by 3 and add 1. Repeat this process until your integer becomes a 1. The fascinating unsolved problem is: Is there any integer for which the process does not terminate? The conjecture is that such integers do not exist. However nobody has managed to prove it yet.

The `rem` command returns the remainder of a Euclidean division of two integers. (in this case `n` and 2)

7 Functions in Matlab

Matlab has been written so that it can be extended by the users. The simplest way to do that is to write *functions* in Matlab code. We will illustrate this with an example: suppose you want to create a function called `stat` which will take an array and return it's mean and standard deviation. To do that you must create a file called `stat.m`. *The file has to have the same name as the function you are defining, and the function definition is the only thing you can put in that file!.* Then in that file, say the following:

```
function [mean, stdev] = stat(x)
% stat -- mean and standard deviation of an array
%   The stat command returns the mean and standard deviation of the
%   elements of an array. Typical syntax is like this:
%   [mean,dev] = stat(x);
%
% See also: foo, gleep, bork
[m,n] = size(x);
if m == 1
 m = n;
end
mean = sum(x)/m;
stdev = sqrt(sum(x.^2)/m - mean.^2);
```

The first line of the file should have the keyword `function` and then the syntax of the function that is being implemented. Notice the following things about the function syntax:

- Functions can have an arbitrary number of arguments. In this case there is only one such argument: `x`. The arguments are being passed by *value*: The variable that the calling code passes to the function is *copied* and the copy is being given to the function. This means that if the function internally changes the value of x, the change will **not** reflect on the variable that you use as argument on your main calling code. Only the copy will be changed, and that copy will be discarded as soon as the function completes it's call.
- Of course it is not desirable to only pass things by value. The function has to communicate some information back to the calling code. In Matlab, the variables on the right hand side, listed in brackets, are also being passed to the function, but this is done **by reference**: The function is not given a *copy* of the variables but it is actually given the variables *themselves*. This means that that any changes made to those variables while inside the function will reflect on the corresponding variables on the calling code. So, if one were to call the function with:

  ```
  a = 1:20;
  m = 0;
  s = 0;
  [m,s] = stat(a);
  ```

then the values of the variables `m` and `s` would change after the call to `stat`.

- The lines afterwords are comments. However, these comments are what will be spit out if you type

  ```
  >> help stat
  ```

  on your prompt. You usually want to make the first comment line be a summary of what the function does because in some instances only the first line will get to be displayed, so it should be complete. Then in the lines afterwords, you can explain how the function is meant to be used.

- After you are done with documentation you type in your usual Matlab code that will implement the function.

The main problem with Matlab function definitions is that you are forced to put every function in a separate file, and are even restricted in what you can call that file. Another thing that could cause you problems is *name collision*: What if the name you choose for one of your functions happens to be the name of an obscure built-in Matlab function? Then, your function will be completely ignored and Matlab will call up the built-in version instead. To find out if this is the case use the `which` command to see what Matlab has to say about your function.

Many of the examples we saw earlier would be very useful if they were to implemented as functions. For instance, if you commonly use Matlab to manipulate data that come out in (x, y) pairs you can make our earlier example into a function like this:

```
function [x,y] = load_xy(filename)
% load_xy  -- Will allow you to load data stored in (x,y) format
% Usage:
% Load your data by saying
%   [x,y] = load_xy(filename)
% where 'filename' is the name of the file where the data is stored
% and 'x' and 'y' are the vectors where you want the data loaded into
fd = fopen(filename);
A  = fscanf(fd,'%g %g\n',[2,inf]);
x  = A(1,:);
y  = A(2,:);
fclose(fd);
```

You would have to put this in a file called `load_xy.m` of course. Suppose that after making some manipulations you want Matlab to save your data on file again. One way to do it is like this:

```
function save_xy(filename,x,y)
% save_xy  -- Will let your save data in (x,y) format.
% Usage:
% If x and y are vectors of equal length, then save them in (x,y)
```

```
% format in a file called 'filename' by saying
% save_xy(filename,x,y)
fd = fopen(filename,'w');
A(1,:) = x;
A(2,:) = y;
fprintf(fd,'%g %g\n',A);
fclose(fd);
```

Notice that it is not necessary to use a `for` loop to `fprintf` or `fscanf` the data one by one. This is explained in detail in the on-line help pages for these two commands. In many other cases Matlab provides ways to eliminate the use of `for`-loops and when you make use of them, your programs will generally run faster. A typical example is *array promotion.* Take a very simple function

```
function y = f(x)
```

which takes a number `x` and returns another number `y`. Typical such functions are `sin, cos, tan` and you can always write your own. Now, if instead of a number you plug in an array or a matrix, then the function will be applied on every element of your array or matrix and an array of the same size will be returned. This is the reason why you don't have to specify in the function definition whether `x` and `y` are simple numbers, or arrays in the first place! To Matlab everything is a matrix as far as functions are concerned. Ordinary numbers are seen as `1x1` matrices rather than numbers. You should keep that in mind when writing functions: sometimes you may want to multiply your `x` and `y` with the `.*` operator instead of the `*` to handle array promotion properly. Likewise with division. Expect to be surprised and be careful with array promotion.

Let's look at an example more closely. Suppose you write a function like this:

```
function x = foo(y,z)
x = y+z;
```

Then, you can do the following on the Matlab prompt:

```
>> disp(foo(2,3))
     5
>> a = 1:1:10;
>> b = 1:2:20;
>> disp(foo(a,b))
     2     5     8    11    14    17    20    23    26    29
```

What you will *not* be allowed to do is this:

```
>> a = 1:1:10
>> b = 1:1:20
>> disp(foo(a,b))
??? Error using ==> +
Matrix dimensions must agree.
```

```
Error in ==> /home/lf/mscc/matlab/notes/foo.m
On line 2  ==> x = y+z;
```

The arguments `a` and `b` can not be added because they don't have the same size. Notice by the way that we used addition as our example on purpose. We challenge you to try `*` versus `.*` and see the effects!

One use of functions is to build complex algorithms progressively from simpler ones. Another use is to automate certain commonly-used tasks as we did in the example of loading and saving (x, y) pairs. Functions do not solve all of the worlds problems, but they can help you a lot and you should use them when you have the feeling that your Matlab program is getting too long and complicated and needs to be broken down to simpler components.