CSCI 6906: Fundamentals of Computational Neuroimaging

Thomas P. Trappenberg
Dalhousie University

1 Programming with Matlab

This chapter is a brief introduction to programming with the Matlab programming environment. We assumes thereby little programming experience, although programmers experienced in other programming languages might want to scan through this chapter. MATLAB is an interactive programming environment for scientific computing. This environment is very convenient for us for several reasons, including its interpreted execution mode, which allows fast and interactive program development, advanced graphic routines, which allow easy and versatile visualization of results, a large collection of predefined routines and algorithms, which makes it unnecessary to implement known algorithms from scratch, and the use of matrix notations, which allows a compact and efficient implementation of mathematical equations and machine learning algorithms. MATLAB stands for matrix laboratory, which emphasizes the fact that most operations are array or matrix oriented. Similar programming environments are provided by the open source systems called **Scilab** and **Octave**. The Octave system seems to emphasize syntactic compatibility with MATLAB, while Scilab is a fully fledged alternative to MATLAB with similar interactive tools. While the syntax and names of some routines in Scilab are sometimes slightly different, the distribution includes a converter for MATLAB programs. Also, the Matlab web page provides great videos to learn how to use Matlab at `http://www.mathworks.com/demos/matlab/...`
`...getting-started-with-matlab-video-tutorial.html`.

1.1 The MATLAB programming environment

MATLAB[1] is a programming environment and collection of tools to write programs, execute them, and visualize results. MATLAB has to be installed on your computer to run the programs mentioned in the manuscript. It is commercially available for many computer systems, including Windows, Mac, and UNIX systems. The MATLAB web page includes a set of brief tutorial videos, also accessible from the **demos** link from the MATLAB desktop, which are highly recommended for learning MATLAB.

As already mentioned, there are several reasons why MATLAB is easy to use and appropriate for our programming need. MATLAB is an interpreted language, which means that commands can be executed directly by an interpreter program. This makes the time-consuming compilation steps of other programming languages redundant and allows a more interactive working mode. A disadvantage of this operational mode is that the programs could be less efficient compared to compiled programs. However, there are two possible solution to this problem in case efficiency become a concern. The first is that the implementations of many MATLAB functions is very efficient

[1] MATLAB and Simulink are registered trademarks, and MATLAB Compiler is a trademark of The MathWorks, Inc.

and are themselves pre-compiled. MATLAB functions, specifically when used on whole matrices, can therefore outperform less well-designed compiled code. It is thus recommended to use matrix notations instead of explicit component-wise operations whenever possible. A second possible solutions to increase the performance is to use the MATLAB compiler to either produce compiled MATLAB code in `.mex` files or to translate MATLAB programs into compilable language such as C.

A further advantage of MATLAB is that the programming syntax supports matrix notations. This makes the code very compact and comparable to the mathematical notations used in the manuscript. MATLAB code is even useful as compact notation to describe algorithms, and it is hence useful to go through the MATLAB code in the manuscript even when not running the programs in the MATLAB environment. Furthermore, MATLAB has very powerful visualization routines, and the new versions of MATLAB include tools for documentation and publishing of codes and results. Finally, MATLAB includes implementations of many mathematical and scientific methods on which we can base our programs. For example, MATLAB includes many functions and algorithms for linear algebra and to solve systems of differential equations. Specialized collections of functions and algorithms, called a 'toolbox' in MATLAB, can be purchased in addition to the basic MATLAB package or imported from third parties, including many freely available programs and tools published by researchers. For example, the MATLAB Neural Network Toolbox incorporates functions for building and analysing standard neural networks. This toolbox covers many algorithms particularly suitable for connectionist modelling and neural network applications. A similar toolbox, called NETLAB, is freely available and contains many advanced machine learning methods. We will use some toolboxes later in this course, including the LIBSVM toolbox and the MATLAB NXT toolbox to program the Lego robots.

1.1.1 Starting a MATLAB session

Starting MATLAB opens the MATLAB desktop as shown in Fig. 1.1 for MATLAB version 7. The MATLAB desktop is comprised of several windows which can be customized or undocked (moving them into an own window). A list of these tools are available under the **desktop menu**, and includes tools such as the **command window**, **editor**, **workspace**, etc. We will use some of these tools later, but for now we only need the **MATLAB command window**. We can thus close the other windows if they are open (such as the **launch pad** or the **current directory window**); we can always get them back from the **desktop** menu. Alternatively, we can undock the command window by clicking the arrow button on the command window bar. An undocked command window is illustrated on the left in Fig. 1.2. Older versions of MATLAB start directly with a command window or simply with a MATLAB command prompt `>>` in a standard system window. The command window is our control centre for accessing the essential MATLAB functionalities.

1.1.2 Basic variables in MATLAB

The MATLAB programming environment is interactive in that all commands can be typed into the command window after the command prompt (see Fig. 1.2). The

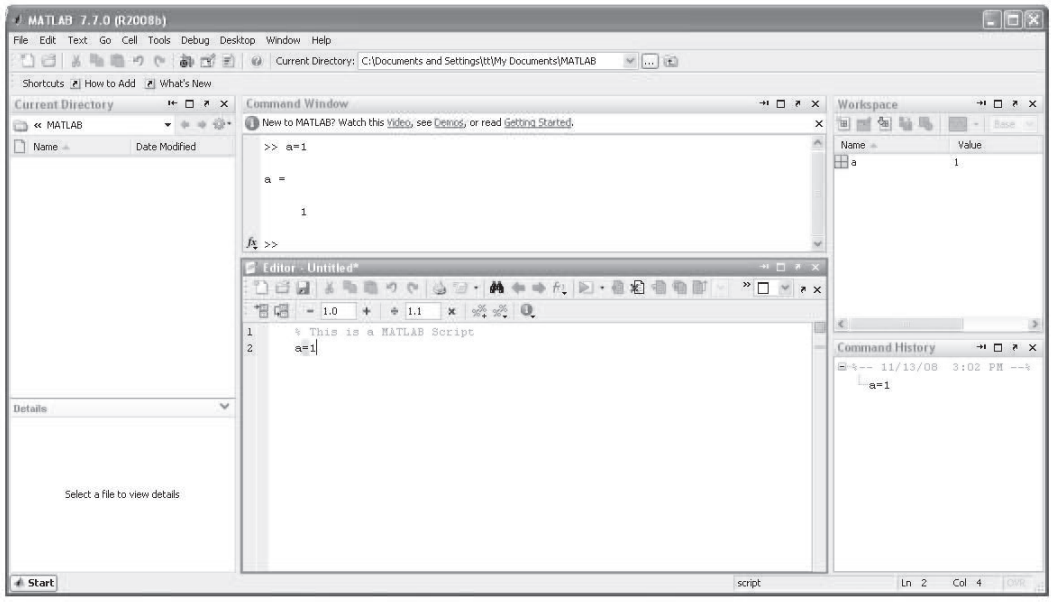

Fig. 1.1 The MATLAB *desktop window* of MATLAB Version 7.

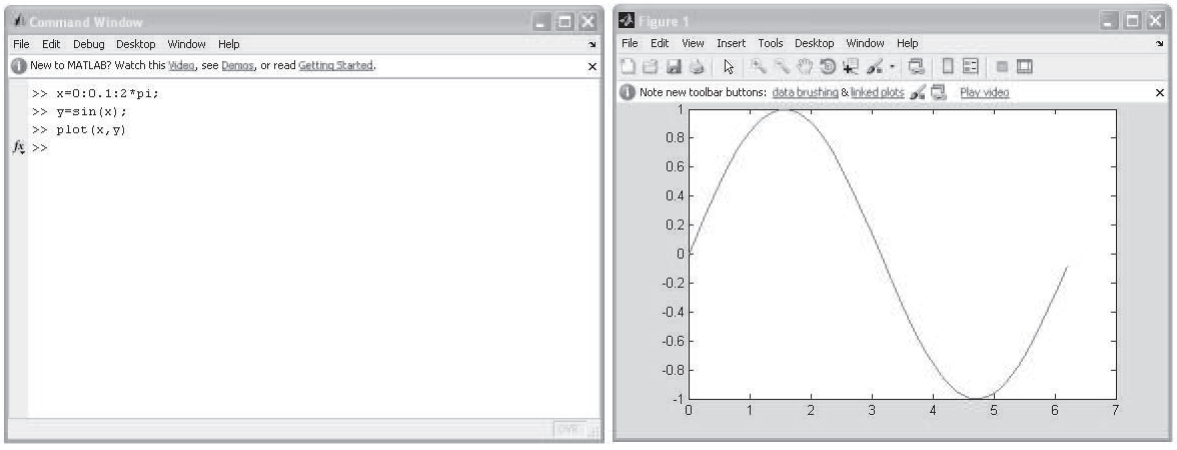

Fig. 1.2 A MATLAB *command window* (left) and a MATLAB *figure window* (right) displaying the results of the function `plot_sin` developed in the text.

commands are interpreted directly, and the result is returned to (and displayed in) the command window. For example, a variable is created and assigned a value with the = operator, such as

```
>> a=3
```

```
a =

     3
```

Ending a command with semicolon (`;`) suppresses the printing of the result on screen. It is therefore generally included in our programs unless we want to view some intermediate results. All text after a percentage sign (`%`) is not interpreted and thus treated as comment,

```
>> b='Hello World!'; % delete the semicolon to echo Hello World!
```

This example also demonstrates that the type of a variable, such as being an integer, a real number, or a string, is determined by the values assigned to the elements. This is called **dynamic typing**. Thus, variables do not have to be declared as in some other programming languages. While dynamic typing is convenient, a disadvantage is that a mistyped variable name can not be detected by the compiler. Inspecting the list of created variables is thus a useful step for debugging.

All the variables that are created by a program are kept in a buffer called **workspace**. These variable can be viewed with the command `whos` or displayed in the **workspace** window of the MATLAB desktop. For example, after declaring the variables above, the `whos` command results in the responds

```
>> whos
  Name      Size             Bytes  Class     Attributes

  a         1x1                  8  double
  b         1x12                24  char
```

It displays the name, the size, and the type (class) of the variable. The size is often useful to check the orientation of matrices as we will see below. The variables in the workspace can be used as long as MATLAB is running and as long as it is not cleared with the command `clear`. The workspace can be saved with the command `save` **filename**, which creates a file **filename**`.mat` with internal MATLAB format. The saved workspace can be reloaded into MATLAB with the command `load` **filename**. The load command can also be used to import data in ASCII format. The workspace is very convenient as we can run a program within a MATLAB session and can then work interactively with the results, for example, to plot some of the generated data.

Variables in MATLAB are generally matrices (or data arrays), which is very convenient for most of our purposes. Matrices include scalars (1×1 matrix) and vectors ($1 \times N$ matrix) as special cases. Values can be assigned to matrix elements in several ways. The most basic one is using square brackets and separating rows by a semicolon within the square brackets, for example (see Fig. 1.2),

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

a =

     1     2     3
     4     5     6
     7     8     9
```

A vector of elements with consecutive values can be assigned by column operators like

```
>> v=0:2:4

v =

     0     2     4
```

Furthermore, the MATLAB desktop includes an **array editor**, and data in ASCII files can be assigned to matrices when loaded into MATLAB. Also, MATLAB functions often return matrices which can be used to assign values to matrix elements. For example, a uniformly distributed random 3×3 matrix can be generated with the command

```
>> b=rand(3)

b =

    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
```

The multiplication of two matrices, following the matrix multiplication rules, can be done in MATLAB by typing

```
>> c=a*b

c =

    3.2329    4.5549    2.9577
    8.5973   10.9730    6.8468
   13.9616   17.3911   10.7360
```

This is equivalent to

```
c=zeros(3);
for i=1:3
   for j=1:3
      for k=1:3
         c(i,j)=c(i,j)+a(i,k)*b(k,j);
      end
   end
end
```

which is the common way of writing matrix multiplications in other programming languages. Formulating operations on whole matrices, rather than on the individual components separately, is not only more convenient and clear, but can enhance the programs performance considerably. Whenever possible, operations on whole matrices should be used. This is likely to be the major change in your programming style when converting from another programming language to MATLAB. The performance disadvantage of an interpreted language is often negligible when using operations on

whole matrices.

The transpose operation of a matrix changes columns to rows. Thus, a row vector such as `v` can be changed to a column vector with the MATLAB transpose operator (`'`),

```
>> v'

ans =

     0
     2
     4
```

which can then be used in a matrix-vector multiplication like

```
>> a*v'

ans =

    16
    34
    52
```

The inconsistent operation `a*v` does produce an error,

```
>> a*v
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Component-wise operations in matrix multiplications (`*`), divisions (`/`) and potentiation `^` are indicated with a dot modifier such as

```
>> v.^2

ans =

     0     4    16
```

The most common operators and basic programming constructs in MATLAB are similar to those in other programming languages and are listed in Table 1.1.

1.1.3 Control flow and conditional operations

Besides the assignments of values to variables, and the availability of basic data structures such as arrays, a programming language needs a few basic operations for building loops and for controlling the flow of a program with conditional statements (see Table 1.1). For example, the **for loop** can be used to create the elements of the vector `v` above, such as

```
>> for i=1:3; v(i)=2*(i-1); end
>> v

v =
```

Table 1.1 Basic programming contracts in MATLAB.

Programming construct	Command	Syntax
Assignment	=	`a=b`
Arithmetic operations	add	`a+b`
	multiplication	`a*b` (matrix), `a.*b` (element-wise)
	division	`a/b` (matrix), `a./b` (element-wise)
	power	`a^b` (matrix), `a.^b` (element-wise)
Relational operators	equal	`a==b`
	not equal	`a~=b`
	less than	`a<b`
Logical operators	AND	`a & b`
	OR	`a|b`
Loop	for	`for` **index=start:increment:end** statement `end`
	while	`while` **expression** statement `end`

Conditional command	if statement	`if` **logical expressions**
		statement
		`elseif` **logical expressions**
		statement
		`else`
		statement
		`end`

Function		`function [x,y,...]=`**name**`(a,b,...)`

```
     0     2     4
```

Table 1.1 lists, in addition, the syntax of a while loop. An example of a conditional statement within a loop is

```
>> for i=1:10; if i>4 & i<=7; v2(i)=1; end; end
>> v2

v2 =

     0     0     0     0     1     1     1
```

In this loop, the statement `v2(i)=1` is only executed when the loop index is larger than 4 and less or equal to 7. Thus, when `i=5`, the array `v2` with 5 elements is created, and since only the elements `v2(5)` is set to 1, the previous elements are set to 0 by default. The loop adds then the two element `v2(6)` and `v2(7)`. Such a vector can also be created by assigning the values 1 to a specified range of indices,

```
>> v3(4:7)=1

v3 =
```

```
     0     0     0     1     1     1     1
```

A 1×7 array is thereby created with elements set to 0, and only the specified elements are overwritten with the new value 1. Another method to write compact loops in MATLAB is to use vectors as index specifiers. For example, another way to create a vector with values such as `v2` or `v3` is

```
>> i=1:10

i =

     1     2     3     4     5     6     7     8     9    10

>> v4(i>4 & i<=7)=1

v4 =

     0     0     0     0     1     1     1
```

1.1.4 Creating MATLAB programs

If we want to repeat a series of commands, it is convenient to write this list of commands into an ASCII file with extension '`.m`'. Any ASCII editor (for example; WordPad, Emacs, etc.) can be used. The MATLAB package contains an editor that has the advantage of colouring the content of MATLAB programs for better readability and also provides direct links to other MATLAB tools. The list of commands in the ASCII file (e.g. **prog1**`.m`) is called a **script** in MATLAB and makes up a MATLAB program. This program can be executed with a run button in the MATLAB editor or by calling the name of the file within the command window (for example, by typing **prog1**). We assumed here that the program file is in the current directory of the MATLAB session or in one of the search paths that can be specified in MATLAB. The MATLAB desktop includes a 'current directory' window (see desktop menu). Some older MATLAB versions have instead a 'path browser'. Alternatively, one can specify absolute path when calling a program, or change the current directories with UNIX-style commands such as `cd` in the command window (see Fig. 1.3).

Functions are another way to encapsulate code. They have the additional advantage that they can be pre-compiled with the MATLAB Compiler$^{\mathrm{TM}}$ available from MathWorks, Inc. Functions are kept in files with extension '`.m`' which start with the command line like

```
function y=f(a,b)
```

where the variables `a` and `b` are passed to the function and `y` contains the values returned by the function. The return values can be assigned to a variable in the calling MATLAB

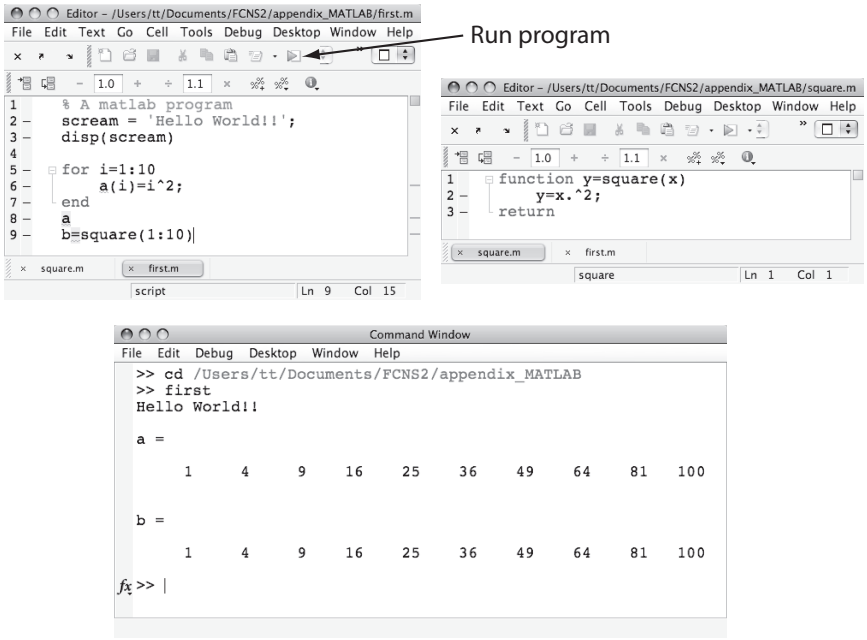

Fig. 1.3 Two editor windows and a command window.

script and added to the workspace. All other internal variables of the functions are local to the function and will not appear in the workspace of the calling script.

MATLAB has a rich collection of predefined functions implementing many algorithms, mathematical constructs, and advanced graphic handling, as well as general information and help functions. You can always search for some keywords using the useful command `lookfor` followed by the keyword (search term). This command lists all the names of the functions that include the keywords in a short description in the function file within the first **comment lines** after the function declaration in the function file. The command `help`, followed by the function name, displays the first block of comment lines in a function file. This description of functions is usually sufficient to know how to use a function. A list of some frequently used functions is listed in Table 1.1.4.

1.1.5 Graphics

MATLAB is a great tool for producing scientific graphics. We want to illustrate this by writing our first program in MATLAB: calculating and plotting the sine function. The program is

```
x=0:0.1:2*pi;
y=sin(x);
plot(x,y)
```

The first line assigns elements to a vector $\mathbf{x}$ starting with $x(1) = 0$ and incrementing the value of each further component by 0.1 until the value 2π is reached (the variable

Name	Brief description	Name	Brief description
abs	absolute functions	mod	modulus function
axis	sets axis limits	num2str	converts number to string
bar	produces bar plot	ode45	ordinary differential equation solver
ceil	round to larger interger	ones	produces matrix with unit elements
colormap	colour matrix for surface plots	plot	plot lines graphs
cos	cosine function	plot3	plot 3-dimensional graphs
diag	diagonal elements of a matrix	prod	product of elements
disp	display in command window	rand	uniformly distributed random variable
errorbar	plot with error bars	randn	normally distributed random variable
exp	exponential function	randperm	random permutations
fft	fast Fourier transform	reshape	reshaping a matrix
find	index of non-zero elements	set	sets values of parameters in structure
floor	round to smaller integer	sign	sign function
hist	produces histogram	sin	sine function
int2str	converts integer to string	sqrt	square root function
isempty	true if array is empty	std	calculates standard deviation
length	length of a vector	subplot	figure with multiple subfigures
log	logarithmic function	sum	sum of elements
lsqcurevfit	least mean square curve	surf	surface plot
	fitting (statistics toolbox)	title	writes title on plot
max	maximum value and index	view	set viewing angle of 3D plot
mix	minimum value and index	xlabel	label on x-axis of a plot
mean	calculates mean	ylabel	label on y-axis of a plot
meshgrid	creates matrix to plot grid	zeros	creates matrix of zero elements

Table 1.2 MATLAB functions used in this course. The MATLAB command `help cmd`, where `cmd` is any of the functions listed here, provides more detailed explanations.

`pi` has the appropriate value in MATLAB). The last element is $x(63) = 6.2$. The second line calls the MATLAB function `sin` with the vector `x` and assigns the results to a vector `y`. The third line calls a MATLAB plotting routine. You can type these lines into an ASCII file that you can name `plot_sin.m`. The code can be executed by typing `plot_sin` as illustrated in the **command window** in Fig. 1.2, provided that the MATLAB session points to the folder in which you placed the code. The execution of this program starts a **figure window** with the plot of the sine function as illustrated on the right in Fig. 1.2.

The appearance of a plot can easily be changed by changing the attributes of the plot. There are several functions that help in performing this task, for example, the function `axis` that can be used to set the limits of the axis. New versions of MATLAB provide window-based interfaces to the attributes of the plot. However, there are also two basic commands, `get` and `set`, that we find useful. The command `get(gca)` returns a list with the axis properties currently in effect. This command is useful for finding out what properties exist. The variable `gca` (get current axis) is the **axis handle**, which is a variable that points to a memory location where all the attribute variables are kept. The attributes of the axis can be changed with the `set` command. For example, if we want to change the size of the labels we can type `set(gca,'fontsize',18)`.

There is also a handle for the current figure `gcf` that can be used to get and set other attributes of the figure. MATLAB provides many routines to produce various special forms of plots including plots with error-bars, histograms, three-dimensional graphics, and multi-plot figures.

1.2 A first project: modelling the world

Suppose there is a simple world with a creature that can be in three distinct states, sleep (state value 1), eat (state value 2), and study (state value 3). An agent, which is a device that can sense environmental states and can generate actions, is observing this creature with poor sensors, which add white (Gaussian) noise to the true state. Our aim is to build a model of the behaviour of the creature which can be used by the agent to observe the states of the creature with some accuracy despite the limited sensors. For this exercise, the function `creature_state()` is available on the course page on the web. This function returns the current state of the creature. Try to create an agent program that predicts the current state of the creature. In the following we discuss some simple approches.

A simulation program that implements a specific agent a with simple world model (a model of the creature), which also evaluates the accuracy of the model, is given in Table 1.3. This program, also available on the web, is provided in file `main.m`. This program can be downloaded into the working directory of MATLAB and executed by typing `main` into the command window, or by opening the file in the MATLAB editor and starting it from there by pressing the icon with the green triangle. The program reports the percentage of correct perceptions of the creature's state.

Line 1 of the program uses a comment indicator (%) to outline the purpose of the program. Line 2 clears the workspace to erase all eventual existing variables, and sets a counter for the number of correct perceptions to zero. Line 4 starts a loop over 1000 trials. In each trial, a creature state is pulled by calling the function `creature_state()` and recording this state value in variable `x`. The sensory state `s` is then calculated by adding a random number to this value. The value of the random number is generated from a normal distribution, a Gaussian distribution with mean zero and unit variance, with the MATLAB function `randn()`.

We are now ready to build a model for the agent to interpret the sensory state. In the example shown, this model is given in Lines 8–12. This model assumes that a sensory value below 1.5 corresponds to the state of a sleeping creature (Line 9), a sensory value between 1.5 and 2.5 corresponds to the creature eating (Line 10), and a higher value corresponds to the creature studying (Line 11). Note that we made several assumptions by defining this model, which might be unreasonable in real-world applications. For example, we used our knowledge that there are three states with ideal values of 1, 2, and 3 to build the model for the agent. Furthermore, we used the knowledge that the sensors are adding independent noise to these states in order to come up with the decision boundaries. The major challenge for real agents is to build models without this explicit knowledge. When running the program we find that a little bit over 50% of the cases are correctly perceived by the agent. While this is a good start, one could do better. Try some of your own ideas . . .

Table 1.3 Program main.m

```
% Project 1: simulation of agent which models simple creature
clear; correct=0;

for trial=1:1000
    x=creature_state();
    s=x+randn();

    %% perception model
    if (s<1.5) x_predict=1;
    elseif (s<2.5) x_predict=2;
    else x_predict=3;
    end

    %% calculate accuracy
    if (x==x_predict) correct=correct+1; end
 end

disp(['percentage correct: ',num2str(correct/1000)]);
```

. . . Did you succeed in getting better results? It is certainly not easy to guess some better model, and it is time to inspect the data more carefully. For example, we can plot the number of times each state occurs. For this we can write a loop to record the states in a vector,

```
>> for i=1:1000; a(i)=creature_state(); end
```

and then plot a histogram with the MATLAB function `hist()`,

```
>> hist(a)
```

The result is shown in Fig. 1.4. This histogram shows that not all states are equally likely as we implicitly assumed in the above agent model. The third state is indeed much less likely. We could use this knowledge in a modified model in which we predict that the agent is sleeping for sensory states less than 1.5 and is eating otherwise. This modified model, which completely ignores study states, predicts around 65% of the states correctly. Many machine learning methods suffer from such 'explaining away' solutions for imbalanced data, as further discussed in Chapter **??**.

It is important to recognize that 100% accuracy is not achievable with the inherent limitations of the sensors. However, higher recognition rates could be achieved with better world (creature + sensor) models. The main question is how to find such a model. We certainly should use observed data in a better way. For example, we could use several observations to estimate how many states are produced by function `creature_state()` and their relative frequency. Such parameter estimation is a basic form of learning from data. Many models in science take such an approach by proposing a parametric model and estimating parameters from the data by model fitting. The main challenge with this approach is how complex we should make the model. It is much easier to fit a more complex model with many parameters to example data, but the

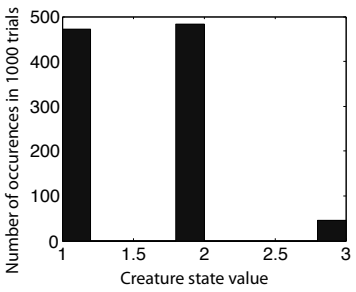

Fig. 1.4 The MATLAB *desktop window* histogram of states produced by function `creature_state()` from 1000 trials.

increased flexibility decreases the prediction ability of such models. Much progress has been made in machine learning by considering such questions, but those approaches only work well in limited worlds, certainly much more restricted than the world we live in. More powerful methods can be expected by learning how the brain solves such problems.

1.3 Alternative programming environments: Octave and Scilab

We briefly mention here two programming environments that are very similar to Matlab and that can, with certain restrictions, execute Matlab scripts. Both of these programming systems are open source environments and have general public licenses for non-commercial use.

The programming environment called **Octave** is freely available under the GNU general public license. Octave is available through links at http://www.gnu.org/software/octave/. The installation requires the additional installation of a graphics package, such as gnuplot or Java graphics. Some distributions contain the SciTE editor which can be used in this environment. An example of the environment is shown in Fig. 1.5

Scilab is another scientific programming environment similar to MATLAB. This software package is freely available under the CeCILL software license, a license compatible to the GNU general public license. It is developed by the Scilab consortium, initiated by the French research centre INRIA. The Scilab package includes a MATLAB import facility that can be used to translate MATLAB programs to Scilab. A screen shot of the Scilab environment is shown in Fig. 1.6. A Scilab script can be run from the execute menu in the editor, or by calling `exec("filename.sce")`.

1.4 Exercises

1. Write a Matlab function that takes a character string and prints out the character string in reverse order. reverses program
2. Write a Matlab program that plots a two dimensional gaussian function.

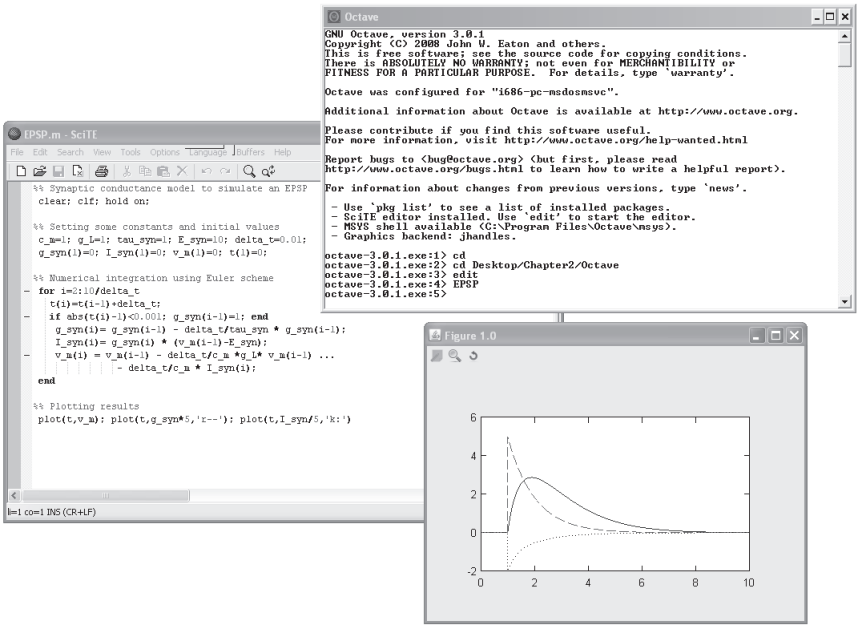

Fig. 1.5 The Octave programming environment with the main console, and editor called *SciTE*, and a graphics window.

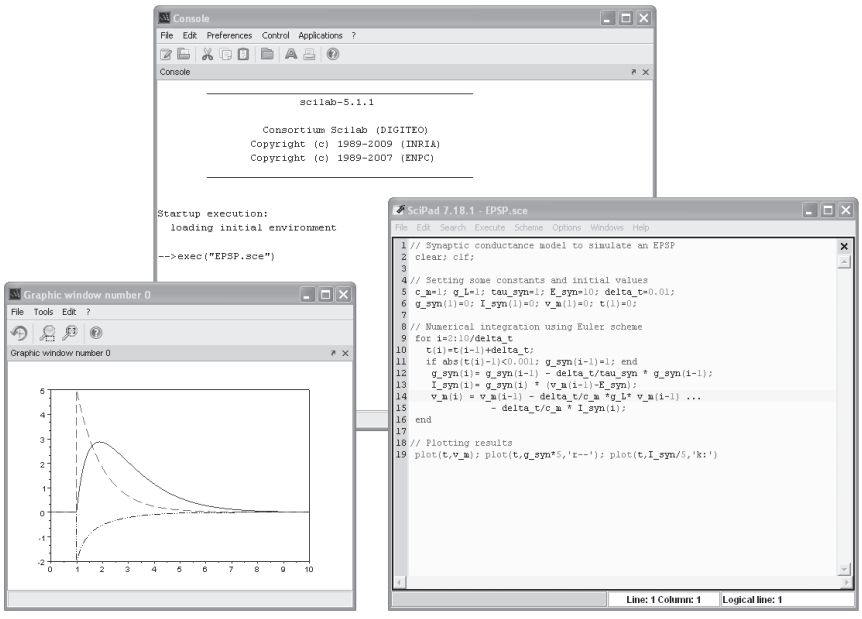

Fig. 1.6 The Scilab programming environment with *console*, and editor called *SciPad*, and a graphics window.