MODIFIED VERSION OF: An introduction to **Matlab** for dynamic modeling Last compile: February 3, 2016

Stephen P. Ellner[1] and John Guckenheimer[2**]
[1]Department of Ecology and Evolutionary Biology, and
[2]Department of Mathematics
Cornell University

**** Modified and added to by Eric Shea-Brown for AMATH 342:**
errors and other unpleasantries may have been introduced, unless you're in this class refer to www.cam.cornell.edu/ ∼ dmb/DMBsupplements.html for the original!!
With additional material from Prof. Mark Goldman, UC Davis

Introduction

These notes for computer labs accompany our textbook *Dynamic Models in Biology* (Princeton University Press 2006). They are based in part on course materials by former TAs Colleen Webb, Jonathan Rowell and Daniel Fink at Cornell, Professors Lou Gross (University of Tennessee) and Paul Fackler (NC State University), and on the book *Getting Started with Matlab* by Rudra Pratap (Oxford University Press). So far as we know, the exercises here and in the textbook can all be done using the Student Edition of Matlab, or a regular base license without additional Toolboxes.

Thinking about MATLAB

- Very sophisticated "graphing calculator"
- High-level programming language for scientific computing
- vs. Python, similar "level" (widely used in science + eng. but not open source)
- vs. c or FORTRAN, many built-in commands, less-complex syntax
- vs. c or FORTRAN, Interpreted language (code translated during run), with some pre-compiled functions
- IF used carefully, good balance of speed and ease of use

1 Interactive calculations

The MATLAB interface is a set of interacting windows. In some of these you "talk" to MATLAB, and in others MATLAB "talks" to you. Windows can be closed (the $\times$ button) or detached to become free-floating (curved arrow button). To restore the original layout, use **View/Desktop Layout/Default** in the main menu bar.

Two important windows are **Help window** and **Command**. Launch Pad is the online Help system. To open this, type `doc` in the command window and hit enter. The Command window is for *interactive commands*, meaning that the command is executed and the result is displayed as soon as you hit the Enter key.

1.1 Graphing calculator mode – working in the command window

For example, at the command prompt >>, type in 2+2 and hit Enter; you will see

```
>> 2+2
ans =
     4
```

Suppressing output: Now type in 2+2; (including the semicolon) – what happens? A semicolon at the end of a line tells MATLAB **not** to display the results of the calculation. This lets you do a long calculation (e.g. solve a model with a large number of intermediate results) and then display only the final result.

1.1.1 Variables

To do anything complicated, the results have to be stored in variables. For example, type `a=2+2` in the Command window and you see

```
>>  a=2+2
a =
    4
```

The variable `a` has been created, and assigned the value 4. By default, a variable in MATLAB is a matrix (a rectangular array of numbers); in this case `a` is a matrix of size 1$\times$1 (one row, one column), which acts just like a number.

Variable names must begin with a letter, and followed by up to 30 letters, numbers, or underscore characters. **Matlab is case sensitive**: `Abc` and `abc` are not the same variable. In contrast to some other languages, a period (.) *cannot* be part of a variable name.

```
>> a1=2
>> my_favorite_variable_number_2=4
```

VALID: Letter followed by letters, numbers, underscore character.

NOTE! capitalization matters. $A \neq a$!!

NOT VALID:

```
>> my_favorite_variable#2=4
??? my_favorite_variable#2=1
>> 2a=1
Error: Unexpected MATLAB expression.
>> my.favorite.variable.name=1
my =
    favorite: [1x1 struct]
```

Exercise 1.1 Here are some variable names that cannot be used in Matlab; explain why: `cell.maximum.size; 4min; site#7` .

1.1.2 FUNDAMENTAL PROGRAMMING SYNTAX:Assigning VALUES to VARIABLES

- LHS = RHS
- value RHS assigned to LHS, NOT other way around

```
>> c=2
c =
     2

>> 2=c
??? 2=c
Error: The expression to the left of the equals sign is not a valid target
```

Another example:

```
>> a=2;b=4;
>> a=b;
>> a,b
```

Exercise 1.2 What values do you get for a and b in the above? Explain why.

```
>> a=2;b=4;
>> b=a;
>> a,b
```

Exercise 1.3 What values do you get for a and b in the above? Explain why.

Calculations

Calculations are done with variables as if they were numbers

```
+
-
/  (divide)
*  (multiply)
^  (exponent)
abs(x)  (absolute value)
cos(x), sin(x), tan(x)     IMPORTANT:  ANGLE x IS INTERPRETED IN RADIANS
exp(x) exponential function e^x
log(x)  log to base e
log10(x) log to base 10
sqrt(x) square root
```

The latter are built-in functions

```
>>  a = sqrt(2)
a =    1.4142
>> a=2 ; b=4 ; c=a^b
c =    16
```

For example enter

```
>>  x=5; y=2; z1=x*y, z2=x/y, z3=x^y
```

and you should see

```
z1 =
    10
z2 =
    2.5000
z3 =
    25
```

Notice that several commands can go on the same line. The first two were followed by semicolons, so the results were not displayed. The rest were followed by commas, and the results were displayed. A comma after the last statement on the line isn't necessary.

You can do several operations in one calculation, such as

```
    >> A=3; C=(A+2*sqrt(A))/(A+5*sqrt(A))
    C =
        0.5544
```

The parentheses are specifying the order of operations. The command

```
    >> C=A+2*sqrt(A)/A+5*sqrt(A)
```

gets a different result – the same as

```
    >> C=A + 2*(sqrt(A)/A) + 5*sqrt(A).
```

The default order of operations is:

PEMDAS

parenthesis
exponentiation
multiplication
division
addition
subtraction

Say we want: $a = 2$, $b = 4$, $c = \frac{a}{a+b}$

```
>> a=1 ; b=4 ; c=a/a+b
c =     5   .... WRONG OR RIGHT?

>> a=1 ; b=4 ; c=a/(a+b)
c =    0.2000  .... WRONG OR RIGHT?
```

Operations of equal priority are performed left to right.

```
>> b = 12-4/2^3          gives    12 - 4/8 = 12 - 0.5 = 11.5
>> b = (12-4)/2^3        gives    8/8 = 1
```

Exercise 1.4 Compute the values of b below, and explain the results you get.

```
>> b = -1^2
>> b = (-1)^2
```

In complicated expressions it's best to use parentheses to specify explicitly what you want, such as `>> b = 12 - (4/(2^3))` or at least `>> b = 12 - 4/(2^3)`. **Use lots of parentheses** instead of trusting the computer to figure out what you meant.

Exercise 1.5 : Have MATLAB compute the values of

1. $\frac{2^5}{2^5-1}$and compare it with $\left(1 - \frac{1}{2^5}\right)^{-1}$ [answer: 1.0323]
2. $\sin(30), \sin(\pi/6), \cos^2(\pi/8)$ [answers for last two: 0.5, 0.8536. The constant π is a pre-defined variable `pi` in Matlab. So typing `cos(pi/8)` works in Matlab, but note that `cos∧2(pi/8)` won't work!]
3. $\frac{2^5}{2^5-1} + 4\sin(\pi/6)$ [answer: 3.0323].

1.2 Storing + viewing variables and their values

Even though x and y were not displayed, MATLAB automatically “remembers” their values. This is a blessing and a curse. First, three ways to see what values are stored:

(1) Type

```
>> x, y
```

and MATLAB displays the values of x and y.

(2) Variables defined in a session are displayed in the **Workspace** window. Click on the tab to activate it and then double-click on `x` to launch a window summarizing `x′`s properties and entries. Since `x` is a 1×1 matrix, there’s only one value. Getting a bit ahead of ourselves, create a 3×2 matrix of 1’s with the command

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

and then look at what X is using the Workspace window. Clicking on the matrix icon opens a window that displays its values.

(3) Use the `whos` command (MATLAB)

```
>> whos
  Name       Size             Bytes  Class     Attributes

  a          1x1                  8  double
  b          1x1                  8  double
  c          1x1                  8  double
```

Now to the curse: stored values can come back to haunt you, giving unintended output. Thus:

GOOD PRACTICE: clear variables before starting to program.

```
>> clear all
>> whos
```

COMMANDS stored in memory

Commands can be edited, instead of starting again from scratch. There are two ways to do this. In the Command window, the ↑ key recalls previous commands. For example, you can bring back the next-to-last command and edit it to

```
>>  x=5 y=2 z1=x*y z2=x/y z3=x^y
```

so that commands are not separated by either a comma or semicolon. Then press Enter, and you will get an error message. *Multiple commands on a line have to be separated by a comma or a semicolon (no display)*.

The other way is to use the **Command History** window, which holds a running history of your commands. You can re-run a command by double-clicking on it.

1.3 Help !!!

MATLAB also has many **built-in mathematical functions** that operate on variables (see Table above). You can get help on any function by entering

```
          help functionname
```

in the console window (e.g., try `help sqrt`).

```
 >> help sqrt
 SQRT   Square root.
    SQRT(X) is the square root of the elements of X. Complex
    results are produced if X is not positive.

    See also sqrtm, realsqrt, hypot.

    Reference page in Help browser
       doc sqrt
```

Even better – a more extensive help browser appears when you type `doc` – try this for sin (`doc sin`) and sqrt.

Thanks anyway, but what SHOULD I be looking up? the lookfor command

```
>> lookfor exponent
EXP    Exponential.
EXPINT Exponential integral function.
EXPM   Matrix exponential.
...
```

Representation of numbers in scientific computing: finite precision

Standard: IEEE floating point arithmetic.

Important feature – finite precision: approx 16 significant digits

Display more digits:

```
 >> a=0.1
```

```
a =
    0.1000
>> format long
>> a
a =   0.100000000000000
```

Roundoff error:

```
  >> a=4/3 ; b=a-1 ; c = (3*b)-1
```

Exercise 1.6 What value does MATLAB return for c above? Does this make sense?

OVERFLOW AND UNDERFLOW:

Maximum number: $\approx 10^{308}$

Minimum number: $\approx 10^{-308}$

Overflow:

```
>> a=10^400
a =   Inf
```

Underflow

```
>> a=10^-400
a =   0
```

Another special number: not defined

```
>> 0/0
ans =
   NaN
```

2 Vectors

MATLAB uses vectors and matrices (1- and 2-dimensional rectangular arrays of numbers) as its primary data types. Starting with the former, a vector value is just a list of scalar values.

Arranged horizontally into a **row vector,:**

$$\vec{x} = (6\ 12\ 5)\,. \tag{1}$$

Or vertically into a **column vector:**

$$\vec{x} = \begin{pmatrix} 6 \\ 12 \\ 5 \end{pmatrix}\,. \tag{2}$$

In MATLAB, row vectors are entered with brackets and spaces or commas:

```
  >>x=[6  12  5]
  x =     6     12     5
```

OR

```
  >>x=[6 ,12 ,5]
  x =     6     12     5
```

In MATLAB, column vectors are entered via semicolons:

```
  >>x=[6 ; 12 ; 5]

x =

     6
    12
     5
```

The **transpose** operation switches between row and column vectors. This is given by dot prime in MATLAB. That is, in MATLAB:

```
>> y=x.'
```

2.1 Computing with vectors

TWO BASIC OPERATIONS ON VECTORS:

Multiplication by scalar

$$c\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} cx_1 \\ cx_2 \\ cx_3 \end{pmatrix}$$; that is, $x_j \to c\,x_j$

Matlab does this via *

```
>> x=[1;2;3] ; 3*x

ans =
     3
     6
     9
```

Addition of two vectors

$$\vec{x} + \vec{y} = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \end{pmatrix}$$

Subtraction is similar, and this works the same for row and column vectors. Matlab implements this via + and - operations:

```
>> x=[1;2] ; y=[0;-2]; z=x+y
z =
     1
     0
```

Matrix dimensions must agree: add row + row or col + col. Here is what happens if this rule is violated:

```
>> z+ [2 2]
??? Error using ==> plus
```

Overall, once a vector has been created it can be used in calculations as if it were a number (more or less).

```
>> initialsize=[1,3,5,7,9,11]
```

```
>>finalsize=initialsize+1
finalsize=
    2 4 6 8 10 12
>> newsize=sqrt(initialsize)
newsize =
    1.0000 1.7321 2.2361 2.6458 3.0000 3.3166
```

Notice that the operations were applied to every entry in the vector. Similarly,

```
     initialsize-5, 2*initialsize, initialsize/10
```

apply subtraction, multiplication by a single (scalar) number, and division to each element of the vector.

Extremely important: element-wise vs array/matrix-wise operations and the "dot" operator.

Let's define a new vector

```
     >> vect=[1 2 3]
```

Let's say we want to multiply each entry of `initialsize` by itself, returning `[1 4 9]`. Easy, right? Let's type

```
     >> initialsize*initialsize
```

BUT MATLAB responds with

```
Error using  *
Inner matrix dimensions must agree.
```

What's going on? When you use `*` between two arrays of numbers (vectors or matrices), MATLAB interprets this as a MATRIX operation – a kind of multiplication that is very different than the "element-wise" multiplication we are trying to do here. It is only defined the two arrays have the "right" sizes, so MATLAB returned an error here – apparently we failed that test. Of course, it would have been even worse if we had the right size arrays, because then we would have gotten an unintended answer that could have REALLY confused us!

THE SOLUTION IS TO USE THE DOT-TIMES OPERATION. That is

```
>> vect.*vect ans = 1 4 9
```

which implies element-wise multiplication. This is EXTREMELY IMPORTANT and will come up again and again in the class. It applies to matrices of numbers as well. More on that later, but see this example:

```
>> matrix=[1 2 ; 3 4]
```

```
matrix =
     1     2
     3     4

>> matrix*matrix
ans =
     7    10
    15    22

>> matrix.*matrix
ans =
     1     4
     9    16
```

Now try

```
     >> vect^2
```

and MATLAB responds with

```
??? Error using ==> ^
Matrix must be square.
```

Why? Because `vect^2` is interpreted as initialsize*initialsize, and $*$ indicates **matrix multiplication**. It was OK to compute 2*vect because Matlab interprets multiplication with a 1×1 matrix as scalar multiplication.

SUMMARY: Elementwise (Entry-by-entry) operations are indicated by a period before the operation symbol:

```
    >> nextsize=initialsize.^2
    >> x=initialsize.*newsize
    >> x=initialsize./finalsize
```

Note that addition and subtraction are always term-by-term. To be safe, if you want an elementwise operation, always us the dot form of an operator!!!

Functions for vector construction

A set of regularly spaced values can be constructed by
x=start:increment:end

```
>> x=0:1:10
x =
     0  1  2  3  4  5  6  7  8  9  10
```

The increment can be positive or negative. If you omit the increment it is assumed to be 1, hence `x=0:10` gives the same result as `x=0:1:10`.

`x=linspace(start, end, length)` lets you specify the number of steps rather than the increment.

```
>> x=linspace(0,10,5)
x =
         0    2.5000    5.0000    7.5000   10.0000
```

Note that `linspace` requires commas as the separators, instead of colons.

Exercise 2.1 Create a vector v=[1 5 9 13], first using the `v=a:b:c` construction, and then using `v=linspace(a,b,c)`.

Vector addressing

Often it is necessary to extract a specific entry or other part of a vector. This is done using subscripts, for example:

```
>> initialsize(3)
ans =
     5
```

This extracts the third element in the vector. You can also access a block of elements using the functions for vector construction

```
c=initialsize(2:5)
c =
     3     5     7     9
```

This has extracted the 2nd through 5th elements in the vector. If you type in

```
     >> c=initialsize(4:2:6)
```

the values in parentheses are interpreted as in vector creation `x=(a:b:c)`. So what do you think this command will do? Try it and see.

Extracted parts don't have to be regularly spaced. For example

```
>> c=initialsize([1 2 5])
c =
     1     3     9
```

extracts the 1st, 2nd, and 5th elements.

Addressing is also used to **set specific values within a vector**. For example,

```
     >> initialsize(1)=12
```

changes the value of the first entry in initialsize while leaving the rest alone, and
`>> initialsize([1 3 5])=[22 33 44]` changes the 1st, 3rd, and 5th values.

Illustrating a common error:

```
>> x=1:.1:1.5
x =
    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000

>> x(7)
??? Index exceeds matrix dimensions.
```

Moreover: remember (especially if you code in c or python!) that indexing in MATLAB starts with 1; x(0) does not work.

Exercise 2.2 Write a **one-line** command to extract the the second, first, and third elements of initialsize **in that order**.

Exercise 2.3 : Have MATLAB compute the values of

1. Make a list numbers spaced by 0.2, between a minimum value of 1 and a maximum value of 20. Assign that list to the variable name `myvector`.
2. repeat the previous command using `linspace`
3. pick out the 4th value of `myvector` and assign it to the variable name `fourthelement`

3 Time to code! Writing script (.m) files

Small tasks can be done interactively, but modeling or complicated data analysis are done using *programs* – sets of commands stored in a file. MATLAB uses the extension **.m** for program files and refers to them as **M-files**.

- Type edit at command line
- Put a few of your favorite commands in the editor
- Make sure `clear all` is the first line in your program
- save it as `myprogram.m` Remember the folder where you saved it.
- You've made a .m file — that is, a MATLAB program!
- Navigate (click on the ..., or use the `cd` command) in the command window to the location where you stored the program
- To run it, type `myprogram` at the command line

Most programs for working with models or analyzing data follow a simple pattern:

1. “Setup” statements.
2. Input some data from a file or the keyboard.
3. Carry out the calculations that you want.
4. Print the results, graph them, or save them to a file.

Exercise 3.1 Write a program `integrator_1.m`.
First define the discrete vector `signal_vector=sin(t)` where `t=[1:10]` represents time. Write code that computes `I_vector` as the cumulative sum of `signal_vector`. (Hint: use doc and help). Make a plot of `I_vector` vs.`t` using $*$ markers for the data. Have your .m file automatically label the axes. (Hint: use doc plot to learn all about plotting). Is it clear that `I_vector` represents a “discrete” integral of the signal?

The for loop

Loops make it easy to do the same operation over and over again, for example:

- make population forecasts 1 year ahead, then 2 years ahead, then 3, ...
- update the state of every neuron based on the inputs it received in the last time interval.

The basic structure can be understood via this MATLAB code.

```
for n= 1:9
    disp(n)
end
```

Use edit to code this into a program `myloop.m`, save it and run it!

COMPONENTS OF THE FOR LOOP:

- n loop variable
- 1:9 loop vector
- disp(n) or print(n) command

HOW IT WORKS:

- Code starts with n equal to first element in loop vector
- runs command
- advances n to next element
- repeats
- quits when have covered all elements of loop vector

In more detail:

- in any for loop, we define a list of numbers, here 1 through 9. think of this as the ”loop vector.” The loop vector can be **any list of numbers** – it does not have to be “integers counting up.”
- the loop variable, n, starts at the first number in the list. it is set equal to that value
- the commands (here, just printing the value of n to the screen) are run
- then when end is reached, n is reset to the NEXT number in the list, the commands are run, and the process is repeated
- it terminates when all entries of the loop vector have been used.

Please type in, and make sure you understand what happens, for these example as well:

```
for p=[4 6 67 -1]
    disp(p)
end
```

```
sum=0
for k=1:5
    sum=sum+k
end
```

Exercise 3.2

- Imagine that you have a giant neural network, and each cell is either firing (“on”) or not (“off”). Each second, for every neuron that is already on, two more switch on. This is a model of EXCITATORY SYNAPTIC COMMUNICATION from the “on” neurons. At time $t = 0$ seconds, 1 neuron is “on.” Write a program, called `neural_explosion.m` that does the following:
  - using a for loop, compute a vector `number_on` that is the number of neurons on at each second, from $t = 0$ to $t = 30$ seconds.
  - Make a plot of the number of neurons on vs. time. Label the axes “time” and “number on.” Hint: type `help plot`!

==	Equal to
∼=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
&	AND
|	OR
∼	NOT

Table 1: Comparison and logical operators in Matlab.

4 IF STATEMENTS

Logical conditions also allow the rules for “what happens next” in a model to be affected by the current values of state variables. The `if` statement lets us do this; the basic format is

```
if(condition);
    commands
else;
    other commands
end;
```

Here’s a simple example or two. Code this into MATLAB:

```
x=2
if (x==2);
    disp('OK, x is 2')
else;
    disp('umm, x is not 2')
end;


x=3
if (x<2);
    disp('OK, x is less than 2')
else;
    disp('umm, x is not less than 2')
end;
```

You get the picture: the condition is ANYTHING you want that is true or false. Of course, for this to make sense, we need MATLAB operations that return true or false values after doing a logical test. We’ve seen two of these above, but and a complete list is in Table 1.

If the “else” is to do nothing, you can leave it out:

```
if(condition);
    commands
end;
```

As in

```
x=3
if (x<2);
    disp('OK, x is less than 2')
end;
```

Exercise 4.1 INTEGRATOR STAGE 2: Write a code `integrator_2.m` which

- defines a signal `signal_vector` (a vector of signal values at different timepoints).
- defines a threshold `thresh`

and computes as its answer the first time that the cumulative sum of the signal crosses the threshold.

More complicated decisions can be built up using `elseif`. The basic format for that is

```
if(condition);
   commands
elseif(condition);
   other commands
else;
other commands
end;
```

5 Creating new functions

M-files can be used to create new functions, which then can be used in the same way as Matlab's built-in functions. Function m-files are often **useful** because they let you break a big program into a series of steps that can be written and tested one at a time. They are also sometimes **necessary**. For example, to solve a system of differential equations in Matlab, you have to write a function m-file that calculates the rate of change for each state variable.

5.1 Simple functions

Function m-files have a special format. Here is an example, `mysum.m` that calculates the sum of the entries in a matrix [the `sum` function applied to a matrix calculates the sum of each column, and then a second application of `sum` gives the sum of all column sums].

```
function f=mysum(A);
    f=sum(sum(A));
return;
```

It works like this:

```
>> mysum([1 2])
ans =
3
```

This example illustrates the rules for writing function m-files:

1. The first line must begin with the word `function`, followed by an expression of the form: `variable_name = function_name(function arguments)`
2. The function name **must** be the same as the name of the m-file.

3. The last line of the file is `return;` (this is not required in the current version of Matlab, but is useful for compatibility with older versions).
4. In between are commands that calculate the function value, and assign it to the variable `variable_name` that appeared in the first line of the function.

5.2 Functions with multiple arguments or returns

To have more than one returned value, the first line in the m-file is slightly different: the various quantities to be returned are enclosed in [ ]. An example is **stats.m**:

```
function [mean_x,var_x,median_x,min_x,max_x]=stats(x);
    mean_x=mean(x); var_x=var(x);
    median_x=median(x);
    min_x=min(x); max_x=max(x);
return;
```

Function m-files can contain *subfunctions*, which are functions called by the main function (the one whose name appears in the m-file name). Subfunctions are "visible" only within the m-file where they are defined. In particular, you cannot call a subfunction at the Command line, or in another m-file. For example, create an m-file `Sumgeseries.m` with the following commands:

```
function f=Sumgseries(r,n);
    u=gseries(r,n); f=sum(u);
return;
function f=gseries(r,n);
   f=r.^(0:n);
return;
```

Only the first of the two functions – the one with the same name as the m-file will be 'visible' to Matlab. That is:

```
>> Sumgseries(0.1,500)
ans =
    1.1111
>> gseries(0.1,500)
??? Undefined command/function 'gseries'.
```

Exercise 5.1 . Write a function m-file `rmatrix.m` which takes as arguments 3 matrices A, S, Z, and returns the matrix $B = A + S. * Z$. When it's working you should be able to do:

```
>> A=ones(2,2); S=0.5*eye(2); Z=ones(2,2); B=rmatrix(A,S,Z)
B =
    1.5000    1.0000
    1.0000    1.5000
```

`zeros(n,m)`	$n \times m$ matrix of zeros
`ones(n,m)`	$n \times m$ matrix of ones
`rand(n,m)`	$n \times m$ matrix of Uniform(0,1) random numbers
`randn(n,m)`	$n \times m$ matrix of Normal($\mu = 0, \sigma = 1$) random numbers
`eye(n)`	$n \times n$ identity matrix
`diag(v)`	diagonal matrix with vector `v` as its diagonal
`linspace(a,b,n)`	vector of n evenly spaced points running from a to b
`length(v)`	length of vector v
`size(A)`	dimensions of matrix A [# rows, # columns]
`find(A)`	locate indices of nonzero entries in A
`min(A), max(A), sum(A)`	minimum, maximum, and sum of entries

Table 2: Some important functions for creating and working with vectors and matrices; many more are listed in the Help system, Functions by Category:Mathematics:Arrays and Matrices. Many of these functions have additional optional arguments; use the Help system for full details.

6 Matrices

A matrix is a two-dimensional array of numbers. Matrices are entered as if they were a column vector whose entries are row vectors. For example:

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

The values making up a row are entered with white space or commas between them. A semicolon indicates the end of one row and the start of the next one. The same process lets you combine vectors to make matrices. For example

```
     >> L=1:3; W=2*L; B=[L;W;L]
```

creates `B` as a 3-row matrix whose 1^{st} and 3^{rd} rows are L=[1 2 3] and the 2^{nd} row is W=[2 4 6]. Similarly

`>> C=[A, B]` or

`>> C=[A B]`

creates a matrix with 3 rows and 6 columns. As with vector creation, the comma between entries is optional.

MATLAB has many functions for creating and working with matrices (Table 2; Uniform(0,1) means that all values between 0 and 1 are equally likely; Normal(0,1) means the bell-shaped Normal (also called Gaussian) distribution with mean 0, standard deviation 1).

Matrix addressing

Matrix addressing works like vector addressing except that you have to specify both row and column, or a range of rows and columns. For example `q=A(2,3)` sets q equal to 6, which is the (2nd row, 3rd column) entry of the matrix **A**, and

```
>> v=A(2,2:3)
v =
     5     6
>> B=A(2:3,1:2)
B =
     4     5
     7     8
```

The Matlab Workspace shows that `v` is a row vector (i.e. the orientation of the values has been preserved) and `B` is a 2×2 matrix.

There is a useful shortcut to extract entire rows or columns, a colon with the limits omitted

```
>> firstrow=A(1,:)
firstrow =
     1     2     3
```

The colon is interpreted as “all of them”, so for example `A(3,:)` extracts all entries in the 3^{rd} row, and `A(:,3)`extracts everything in the 3^{rd} column of A.

As with vectors, addressing works in reverse to assign values to matrix entries. For example,

```
>> A(1,1)=12
A =
    12     2     3
     4     5     6
     7     8     9
```

The same can be done with blocks, rows, or columns, for example

```
A(1,:)=rand(1,3)
A =
    0.9501    0.2311    0.6068
    4.0000    5.0000    6.0000
    7.0000    8.0000    9.0000
```

A numerical function applied to a matrix acts element-by-element:

```
>> A=[1 4; 9 16]; A, sqrt(A)
A =
     1     4
     9    16
ans =
     1     2
     3     4
```

The same is true for scalar multiplication and division. Try

```
     >> 2*A, A/3
```

and see what you get.

If two matrices are the same size, then you can do element-by-element addition, subtraction, multiplication, division, and exponentiation:

```
     A+B, A-B, A.*B, A./B, A.^B
```

Exercise 6.1 Use `rand` to construct a 5×5 matrix of random numbers with a uniform distribution on $[0, 1]$, and then (a) Extract from it the second row, the second column, and the 3×3 matrix of the values that are not at the margins (i.e. not in the first or last row, or first or last column). (b) Use `linspace` to replace the values in the first row by `2 5 8 11 14`.

Matrices and loading data

Another key step is often loading data from a text file. From our website download **ChlorellaGrowth.txt** and save it in your working directory to see how this is done. First, instead of having to type in the numbers at the Command line, the command

```
X=load('ChlorellaGrowth.txt')
```

reads the numbers in **ChlorellaGrowth.txt** and puts them into variable **X**. We extract them with the commands

```
Light=X(:,1); rmax=X(:,2);
```

As you know by now, these are shorthand for 'Light=everything in column 1 of X', and 'rmax=everything in column 2 of X'.

Type whos to confirm what you now have.

7 Matrix computations

Matrix-vector multiplication

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = x_1 \begin{pmatrix} A_{1,1} \\ A_{2,1} \end{pmatrix} + x_2 \begin{pmatrix} A_{1,2} \\ A_{2,2} \end{pmatrix}$$

e.g.

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

In general,

$$\begin{pmatrix} | & \cdots & | \\ a_1 & \cdots & a_n \\ | & \cdots & | \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \sum_j x_j \begin{pmatrix} | \\ a_j \\ | \end{pmatrix}$$

Also useful to think of equivalent "ROW-WISE" form of matrix multiplication.

MATLAB performs matrix-vector multiplication via the * operator.

```
>> A=[1 1 ; 1 2] ; A*[1 ; 2]

ans =

     3
     5
```

In $y = Ax$, A must have same number of columns as x has rows.

Nonsense:

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 4 \end{pmatrix}$$

```
>> A=[1 1 ; 1 2] ; A*[1 ; 2 ; 4]
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Exercise 7.1 : Compute the below by hand and check your result in MATLAB

$$\begin{pmatrix} 2 & -3 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$