Introduction to Matlab

Nicolas Hudon (nhudon@chee.queensu.ca)
Darryl DeHaan
Pr. Martin Guay
Chemical Engineering, Queen's University

January 17, 2006

Contents

1 Introduction

The purpose of this short monograph is to introduce MATLAB to undergraduate students in chemical engineering (even if the chemical engineering content is kept to a minimum). At Queen's, it is intended mainly for students in CHEE222 - Process Dynamics and Numerical Methods. It is hoped that it will also serve as a reference for other courses :

- CHEE319 - Process Control I
- CHEE321 - Chemical Reaction Engineering
- CHEE434 - Process Control II

Here, we discuss the basic features of programming and good programming practice within the MATLAB environment, therefore SIMULINK and more advanced toolboxes (such as the Control Toolbox) are not covered. However, it is assumed that after this minimal introduction, most students will be comfortable to read by themselves the corresponding help files and the available documentation that can be found online on the MATHWORKS' website (www.mathworks.com).

The next section of the document covers the description of the MATLAB environment and some operations in the WORKSPACE (this term will be defined there). Section 3 presents the two main objects that students should be able to use: SCRIPT and FUNCTION files. Section 4 illustrates the basic vector, matrix and mathematical operations. Section 5 covers the main programming objects in MATLAB - logical operators, loops, etc. The last section will present a broad range of topics (most of it will be used in CHEE222). Those topics will serve mainly as examples of application of MATLAB and as a collection of the main features useful for undergraduate students in chemical engineering. Some plotting capabilities are presented there.

A few references are listed at the end of the document. The best reference for more advanced topics (such as Object Oriented Programming) is the book of Hanselman and Littlefield (2004). For the topics covered in CHEE222 and CHEE319, the books by Bequette (1998, 2003) incorporate MATLAB in a natural way. Finally, for numerical methods in chemical engineering with MATLAB, the book by Constantinides and Mostoufi (1999) is a good reference.

Examples and problems are distributed throughout the text. The solution to the problems may be obtain upon request. Finally, since the present document is a work in progress, some typos and mistakes may remain. Please contact the author in any case.

2 Matlab Interface

2.1 General

Strictly speaking, MATLAB is not a programming language (even if it has his own syntax). It must be seen as a shell (like any UNIX Shell or like DOS). This means that it has some built-in functions that can be called from the prompt line, so that you can write your own functions. Also, you can call a succession of functions by a SCRIPT file. The main advantage of MATLAB over other shells is the possibility to perform mathematical operations (and especially vector and matrix operations). The MATLAB environment is presented in Figure 1.

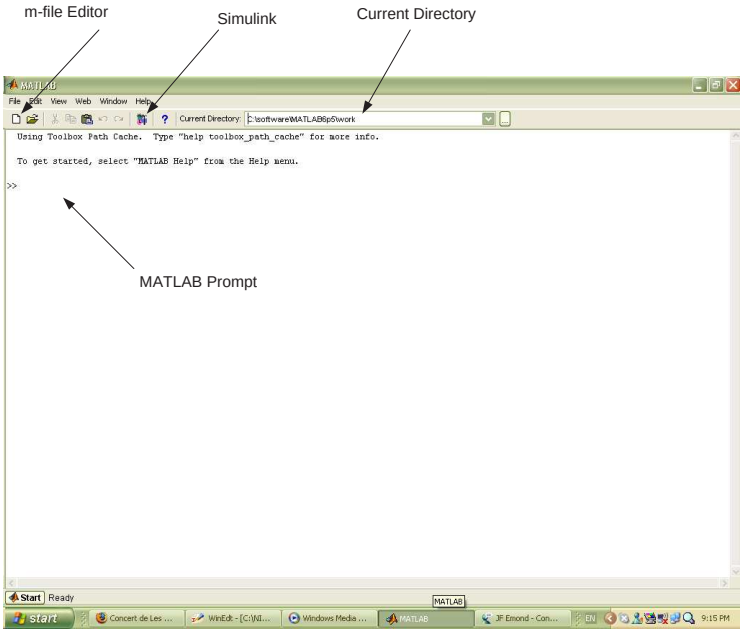

Figure 1: MATLAB Environment

SIMULINK will not be covered here even if it will be used in CHEE319 and CHEE434. For now, we just point out the following: always be sure that the *Current Directory* points to the file folder where your files are located (you can also add your favorite directory to the *path* variable, but we will cover it as an example later on).

For the remainder of this section, we will present elementary manipulations with MATLAB performed on the *prompt line*. Please keep in mind that this is not the best way to use MATLAB; instead, it is preferable to use a SCRIPT or a FUNCTION file - see next Section. However, using the *prompt line*, it is possible to assign values to variables and to perform usual operations:

```
>> x = 4
x =
     4
>> y = 2
y =
     2
>> x + y
ans =
     6
>> x * y
ans =
     8
```

Here, we note that MATLAB assigns the result of the operations to a variable named *ans*. It is possible to know workspace variables and their type using the command *whos*:

```
>> whos
  Name      Size                    Bytes  Class
  ans       1x1                         8  double array
  x         1x1                         8  double array
  y         1x1                         8  double array
Grand total is 3 elements using 24 bytes
```

We can find out the value of a given variable by typing the name on the prompt line:

```
>> ans
ans =
     8
```

The solution to the above addition has been lost. A good way to prevent that is to assign operation results to a given variable:

```
>> x = 4;   % semi-colons prevent the output on the screen
>> y = 2;   % percentage enables comments
>> a = x + y
a =
     6
>> b = x * y
b =
     8
```

```
>> whos
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         1x1                         8  double array
  x         1x1                         8  double array
  y         1x1                         8  double array
Grand total is 4 elements using 32 bytes
>>
```

As noted in the above example, we can avoid screen output using a semi-colon and we can comment our operation using percentage symbol (everything following % won't be read by MATLAB). The function *clear* enables us to delete a given variable or all workspace variables.

```
>> clear x
>> whos
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         1x1                         8  double array
  y         1x1                         8  double array
Grand total is 3 elements using 24 bytes
>> clear all
>> whos         % nothing left in the memory
>>
```

If we look at the *whos* function output, we remark that MATLAB returns the CLASS of a given variable. Most common variable classes in the present context are the *double* - stored in matrix structure, *char*, to handle text (useful for outputs) and *symbolic* variables covered later on. To declare *char* variables, we use apostrophes:

```
>> word_1 = 'hello'
word_1 =
hello
```

It is possible to concatenate words (i.e. to form a new string using simpler strings) using brackets:

```
>> word_1 = 'hello';
>> word_2 = 'world';
>> word_3 = 2005;
>> concatenate = [word_1 word_2 word_3]
concatenate =
helloworldÕ
```

Of course, this is not a neat result. The best way to do it is:

```
>> word_3 = num2str(2005);
>> concatenate = [word_1 ' ' word_2 ' ' word_3]
concatenate =
hello world 2005
```

The function *num2str* enables the user to translate a number to the corresponding char string.

```
>> whos
  Name              Size                    Bytes  Class
  concatenate       1x16                       32  char array
  word_1            1x5                        10  char array
  word_2            1x5                        10  char array
  word_3            1x4                         8  char array
Grand total is 30 elements using 60 bytes
```

It is also possible to interact with MATLAB using the *input* function.

```
>> a = input('Enter a number: '); % User input
   b = sqrt(a); % Square root
   str = ['Square root of ' num2str(a) ' is : ' num2str(b) '.'];
   disp(str) % use display function for output
```

The output is the following:

```
>> Enter a number : 23 % value entered by the user
Square root of 23 is : 4.7958. % Output
>>
```

Of course, the last example makes no sense if we type everything on the screen! For this reason, before covering any vector and matrix operations, we will define the SCRIPT and FUNCTION files and illustrate their use.

3 SCRIPT and FUNCTION Files

So far, we have used MATLAB like a calculator. For repetitive tasks (and especially to plot graphical outputs), it is far more practical to write some short programs. We usually use SCRIPT files that we can launch from the MATLAB *prompt line* to call and save outputs from the most repetitive operations, usually written in FUNCTION files. Both type share the file extension **.m*. We usually write those using the M-file editor (clicking the icon or by typing *edit* from the prompt line) but any ASCII editor (such as Windows Notepad) can also be used.

3.1 SCRIPT Files

We use SCRIPT files to assign values to variables, to call FUNCTION -usually MATLAB built-in functions, perform algebraic operations or operations on *symbolic* variables and plot figures. Basically, a SCRIPT file just performs the same operations we would have performed directly in the Workspace. Note that **variables used in a SCRIPT are available in the Workspace**. For example, we can write the following m-file:

```
% test1.m

clear all % to clear variables in the Workspace
close all % to close all plots windows opened by MATLAB

x = 4;
y = 2;
a = x + y
b = x * y

whos
```

We call this file (with corresponding output) from the Workspace:

```
>> test1
a =
     6
b =
     8
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  b         1x1                         8  double array
  x         1x1                         8  double array
  y         1x1                         8  double array
Grand total is 4 elements using 32 bytes
>>
```

Of course, it is important to check that the file we are calling is located in the *Current Directory*. If not, MATLAB will most likely output an error message like the following:

```
>> test1
??? Undefined function or variable 'test1'.
```

One important feature of MATLAB is that it is not a compiled programming language like C/C++. Each time you call MATLAB, the program reads and performs the program file line by line. If MATLAB detects an error, it will output an error message (with the line where the error might be located). Learning how to interpret error messages from MATLAB is a key to programming efficiently.

3.2 FUNCTION Files

FUNCTION files are usually used to perform repetitive simple mathematical operations. It is also useful if we don't need intermediate values (since values internal to a FUNCTION file are not available from the Workspace). Also, it is a nice way to divide a larger problem in simpler sub-operations. A function is defined by its inputs and outputs, like in the following (already seen example):

```
function a = test2(x,y)

a = x + y;
b = x * y;
```

and the following call:

```
>> value = test2(4,2)
value =
     6
>> whos
  Name        Size                    Bytes  Class
  value       1x1                         8  double array
Grand total is 1 element using 8 bytes
```

The multiplication result is lost (even if we performed it in the FUNCTION file). Since, we can define multiple outputs from a FUNCTION file, this can be fixed using the following:

```
function [a,b] = test3(x,y)

a = x + y;
b = x * y;
```

```
>> [ans1,ans2] = test3(4,2)
ans1 =
     6
ans2 =
     8
>> whos
  Name        Size                    Bytes  Class
  ans1        1x1                         8  double array
  ans2        1x1                         8  double array
Grand total is 3 elements using 24 bytes
>>
```

We will now illustrate the use of SCRIPT and FUNCTION files using more involved operations. The next section covers most of the basic mathematical operations available in MATLAB.

4 Basic Mathematical Operations

4.1 Scalars, Vectors and Matrices

The basic element in MATLAB is a matrix. For example a scalar is stored in a 1×1 matrix, a n column vector is stored in a $n \times 1$ matrix and a n row vector is stored in a $1 \times n$ matrix. Unlike in other programming languages that you might have encountered so far, it is not necessary to declare a variable's type and length in MATLAB, therefore, we must be careful when manipulating those. As seen before, scalars are input in MATLAB directly:

```
>> x = 2;
>> a = x;
>> b = x^2; % square of x
```

Row vectors can be declared using brackets:

```
>> row_v = [0 1 2]
row_v =
   0 1 2
```

To declare column vectors, we use semi-colon at the end of each line:

```
>> col_v = [0;1;2]
col_v =
   0
   1
   2
```

It is also possible to transpose vectors using the apostrophe:

```
>> col_v = row_v'
col_v =
   0
   1
   2
>> row_v = col_v'
row_v =
   0 1 2
```

The colon is the incremental operator in MATLAB (with default value set to 1):

```
>> row_v2 = [0:5] % using default increment
row_v2 =
   0 1 2 3 4 5
>> row_v3 = [0:0.2:1]  % using a different increment
row_v3 =
   0.0000 0.2000 0.4000 0.6000 0.8000 1.0000
```

Note the format of the outputs (the *short* format gives 4 digit). It is possible to change it (since Matlab stores all doubles using 16 digits but usually output only 4):

```
>> format long
>> out = row_v3(3)
out =
   0.40000000000000
>>
```

In the last example, we output only the 3rd value from the vector *row_v3* using the parentheses. Note that it is possible to change a value in a vector using the exact same referencing procedure:

```
>> format short % to recover original display
>> row_v3(2) = 10
row_v3 =
   0.0000 10 0.4000 0.6000 0.8000 1.0000
```

Usual vector operations of addition, substraction and multiplication by a scalar are defined in MATLAB:

```
>> V1 = [1 2];
>> V2 = [3 4];
>> V3 = V1 + V2 % vector addition
V3 =
     4     6
>> V4 = V3 - V2 % vector substraction
V4 =
     1     2
>> V5 = 2*V4   % scalar multiplication
 V5 =
     2     4
```

Dot products for two vectors of the same dimension and cross product (for dimension 3) are also defined:

```
>> dprod= dot(V1,V2)
dprod =
    11
>> cprod = cross([V1 2],[V2 0])
cprod =
    -8     6    -2
```

For vector multiplication and division, we have to be careful with the dimensions. To perform element-by-element multiplication and division, we can use the following dot-operators:

```
>> V6 = V1.*V2   % element-by-element multiplication
V6 =
     3     8
>> V7 = V1./V2   % element-by-element division
V7 =
    0.3333    0.5000
```

However, MATLAB throws an error when dimensions are not matched (please note the error message, quite useful to debug your own code):

```
>> V8 = [1 2 3];
>> V9 = V1.*V8
??? Error using ==> .* Matrix dimensions must agree.
```

It is known from linear algebra that for multiplication, the order is important:

```
>> V9 = V2';   % defining a 2x1 vector
>> MAT = V9*V1
MAT =
     3     6
     4     8
>> SCALAR = V1*V9
SCALAR =
    11
```

As seen before in the case of *char* variables, it is possible to concatenate vectors of proper dimensions:

```
>> V10 = [V1 V2]
V10 =
     1     2     3     4
>> length(V10)
ans =
     4
>> MAT2 = [V1;V2]
MAT2 =
     1     2
     3     4
```

The function *length* returns the length of a vector, which is quite useful when using the *for* loops. We can perform the same operations on column vectors:

```
>> V11 = [1;2];
>> V12 = [3;4];
>> V13 = [V11;V12]
V13 =
     1
     2
     3
     4
>> length(V13)
ans =
     4
>> MAT3 = [MAT2 V11 V12]
MAT3 =
     1    2    1     3
     3    4    2     4
>> size(MAT3)
ans =
     2     4
```

From the last example, we see that MAT2 $\neq$ MAT3. Also, we use *size* to check the dimension of the resulting matrix (number of lines, number of columns). Note that *length* of a matrix return a value:

```
>> length(MAT3)
ans =
      4
```

Again, we must be careful when manipulating vectors in MATLAB. For example, if we try the following:

```
>> V14 = [V1 V11]
??? Error using ==> horzcat All matrices on a row in the bracketed
expression must have the
 same number of rows.
```

Of course, we can also construct matrices directly:

```
>> M1 = [1 2; 3 4]
M1 =
     1     2
     3     4
```

Which is exactly the same as the MAT2 definition form above. We have access to element of a matrix by the indices of rows and column:

```
>> m21 = M1(2,1) % second line, first column
m21 =
     3
>> M12 = M1(1,2) % first line, second column
M12 =
     2
```

It is also possible to access an element of a matrix using only one indice. Please note that MATLAB counts elements of an entire column first and then proceed to the second column:

```
>> M2 = [1 2 3; 4 5 6; 7 8 9]
M2 =
     1     2     3
     4     5     6
     7     8     9
>> m4 = M2(4)
m4 =
     2
```

We can extract an entire line or column using the column operator as the following:

```
>> v1 = M2(2,:) % all elements from 2nd line
v1 =
     4     5     6
>> v2 = M2(:,3)  % all elements from 3rd column
v2 =
     3
     6
     9
```

Extracting part of a line, column or matrix is also possible:

```
>> v3 = M2(3,2:3)   % 3rd line, 2nd to 3rd elements
v3 =
     8     9
>> M3 = M2(1:2,1:3)  % 1st and 2nd lines, 1st to 2nd columns
M3 =
     1     2     3
     4     5     6
>> length(M3)            % length returns the greatest dimension of a
ans =                    % matrix -- here the number of columns
     3
>> max(dim(M3))          % equivalent statement
ans =
     3
```

We can inverse a square matrix, transpose any matrix using:

```
>> iM1 = inv(M1)         % inverse of a matrix
iM1 =
   -2.0000    1.0000
    1.5000   -0.5000
>> tM1 = transpose(M1)   % transpose of a matrix
tM1 =
     1     3
     2     4
>> tM1 = M1'             % transpose using the apostrophe
tM1 =
     1     3
     2     4
```

Addition, substraction, power, matrix multiplication and division of matrices are defined (be careful with dimensions):

```
>> clear all
>> A = [1 2; 5 6];   % 2x2 matrix
>> B = [3 4; 7 8];   % 2x2 matrix
>> V1 = [1 2];       % 1x2 vector
>> C = A + B        % addition
C =
     4     6
    12    14
>> D = A - B        % substraction
D =
    -2    -2
    -2    -2
>> E = 3*A          % scalar multiplication
E =
     3     6
    15    18
>> F = A*B          % Matrix multiplication
F =
    17    20
    57    68
>> G = A.*B         % element-by-element multiplication
G =
     3     8
    35    48
>> H = A/B           % Matrix division --- right-multiplication by the inverse
H =
    1.5000   -0.5000
    0.5000    0.5000
```

```
>> I = A * inv(B)     % Right multiplication by the inverse
I =
    1.5000   -0.5000
    0.5000    0.5000
>> J = A\B           % Matrix division --- left-multiplication by the inverse
J =
    3.0000    2.0000
   -2.0000   -1.0000

>> K = inv(B)*A        % Left multiplication by the inverse
K =
    3.0000    2.0000
   -2.0000   -1.0000
>> L = A./B            % Element-by-element division (A(i,j) divided by B(i,j))
L =
    0.3333    0.5000
    0.7143    0.7500
>> M = A.\B            % Element-by-element division (B(i,j) divided by A(i,j))
M =
    3.0000    2.0000
    1.4000    1.3333
>> N = A^2                % Power of a matrix
N =
    11    14
    35    46
>> N = A*A
N =
    11    14
    35    46
>> O = A.^2                % element-by-element Power of a Matrix
O =
     1     4
    25    36
```

MATLAB also provides some tools like to compute the exponential (*expm*), logarithm (*logm*) and square root (*sqrtm*) of matrices. Multiplication of matrices by vector is straightforward:

```
>> L = V1*A
L =
    11    14
>> M = A*V1
??? Error using ==> * Inner matrix dimensions must agree.
>> M = A*V1'
M =
     5
    17
```

As we see in the example above, we must be very careful with operations involving matrices, especially division, defined as the right multiplication by the

inverse. Before presenting a table summarizing some of the matrix functions available in MATLAB, we will present two built-in declaration for matrices in MATLAB:

```
>> I = eye(3)        % the identity matrix
I =
     1     0     0
     0     1     0
     0     0     1
>> Z1 = zeros(2,2)     % a zeros 2x2 square matrix
Z1 =
     0     0
     0     0
>> Z2 = ones(2,2)     % a ones 2x2 square matrix
Z2 =
     1     1
     1     1
>> Z1Vc = zeros(2,1)     % a 2x1 zero column vector
Z1Vc =
     0
     0
>> Z2Vc = ones(1,2)     % a 1x2 ones row vector
Z2Vc =
     1      1
```

Table 1: MATLAB Matrix Functions

Function	Description
det(A)	determinant of matrix A
trace(A)	trace of matrix A
rank(A)	rank of the matrix
norm(A)	norm of the matrix (can be applied to vector)
inv(A)	inverse of matrix A
poly(A)	Characteristic polynomial of the matrix A
eig(A)	eigenvalues/eigenvectors of A
svd(A)	singular value decomposition of A
min(V)	minimum value in vector V
max(V)	maximum value in vector V
sum(V)	sum of the elements of vector V

Before using the functions of Table 1, you should always look at the *help* function from MATLAB to check the proper call to it. For example, for the function *eig*

```
>> A = [1 2; 5 6];   % 2x2 matrix
>> eig(A)
ans =
   -0.5311
    7.5311
```

returns only the eigenvalues. To obtain the (normalized) eigenvectors, the proper way to call the function is the following:

```
>> [evalues, evectors] = eig(A)
evalues =
   -0.7940   -0.2928
    0.6079   -0.9562
evectors =
   -0.5311         0
         0    7.5311
```

We now suggest a few exercises to test your ability to perform standard matrix operations.

Ex 1. We want to verify that the multiplication of two matrices is not commutative using :

$$A = \begin{pmatrix} 3 & 4 & 4 \\ 6 & 5 & 3 \\ 10 & 8 & 2 \end{pmatrix}, B = \begin{pmatrix} 4 & 5 & 8 \\ 3 & 11 & 12 \\ 2 & 1 & 7 \end{pmatrix}$$

Write a short MATLAB SCRIPT that check if

$$AB - BA \neq 0$$

Ex 2. Using matrices A and B from number 1, check the following identity:

$$(A + B)^t = A^t + B^t$$

Ex 3. Using the same matrix and function *eye*, check also that

$$BB^{-1} = B^{-1}B = I$$

Ex 4. Using the function *trace* and *eig*, check for matrix A from Exercise 1, that

$$\mathrm{tr}A = \sum_{i=1}^{n} \lambda_i(A)$$

where λ_i are the eigenvalues of the matrix.

Ex 5. We want to solve an algebraic linear system of the form

$$Ax = b$$

where A is an $n \times n$ square matrix, x and b are $n \times 1$ column vectors. The solution is given by:

$$x = A^{-1}b$$

Using the function *inv*, solve the system with A defined in Exercise 1 and b being the second column of the matrix B from the same Exercise.

4.2 Simple Plots

Before introducing the usual trigonometric functions in MATLAB, we will introduce the basic plotting function. We will cover more advanced plotting functions and how to export those graphics later on.

The simplest way to produce graphical outputs from MATLAB is by using the fonction *plot*, as in the following example:

```
>> t = [0:0.01:2*pi];
>> plot(t,cos(t),'+', t, sin(t),'*')
```

with the following output Figure 2:

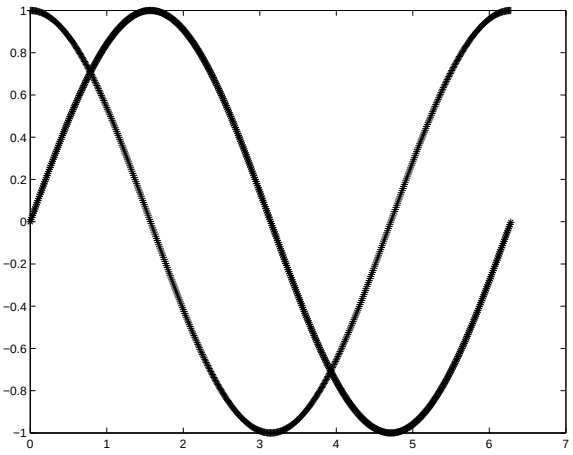

Figure 2: sin(t) and cos(t)

To add elements to the graphic, we can use the following SCRIPT file:

```
% test4.m

clear all
close all

t = [0:0.01:2*pi];
plot(t,cos(t),'+', t, sin(t),'*')
xlabel('t')
ylabel('f(t)')
title('sin(t) and cos(t) for [0, 2pi]')
legend('cos(t)','sin(t)',0)
```

which produces Figure 3:

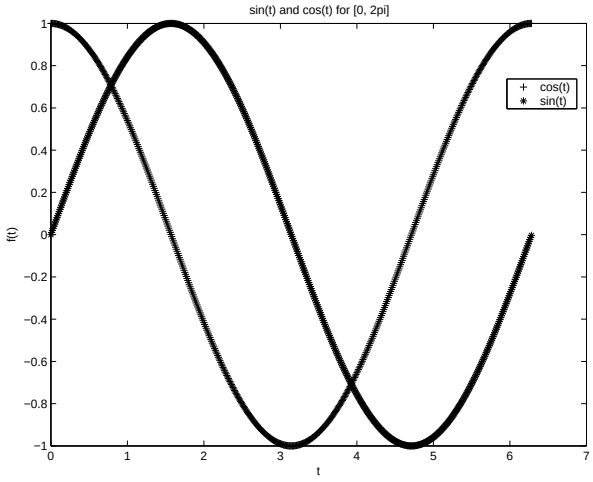

Figure 3: sin(t) and cos(t) - revised

Colors and curves elements are explained better in the help. Just type *help plot* from the MATLAB command line.

4.3 Usual Mathematical Functions

The usual mathematical operations in MATLAB are presented in Table 2. Note that the trigonometric functions can be applied to scalars or vectors.
Inverse trigonometric functions are defined like *asin*, i.e. by adding *a* in front; hyperbolic functions are defined like *sinh*, i.e. by adding *h* after the function name and the same goes for inverse hyperbolic functions.

Table 2: MATLAB Functions

Function	Description
sin(x)	sinus of x ; x in radians
cos(x)	cosinus of x ; x in radians
tan(x)	tangent of x ; x in radians
sec(x)	secant of x ; x in radians
csc(x)	cosecant of x; x in radians
cot(x)	cotangent of x ; x in radians
asin(x)	arc sinus of x
sinh(x)	hyperbolic sinus of x
asinh(x)	inverse hyperbolic sinus of x
exp(x)	exponential of x
log(x)	natural logarithm of x
log10(x)	base 10 logarithm of x
sqrt(x)	square root of x
power(x,a)	x to the power of a (equiv to x^4)
abs(x)	absolute value of x
round(x)	round x towards nearest integer
floor(x)	round x towards minus infinity
ceil(x)	Round x towards infinity
fix(x)	Round x towards zero

Ex 6. Check graphically that

$$\sinh(x) = \frac{e^x - e^{-x}}{2}$$

on the domain $x \in [-2, 2]$. Use increments of 0.1. Label the plots properly.

Ex 7. Check graphically that (using the same domain for x than number 6) that

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$$

Be careful with the division of the two vectors on the right hand side!

5 Programming with MATLAB

This section will review the usual programming loops within MATLAB: IF Statement, FOR and WHILE Loops. First, we begin the discussion with a refresher on logical operators. Students who already had a programming course will only need to familiarize with MATLAB syntax - and some good programming practices in MATLAB.

5.1 Logical Operators

Logical operators are quite important in conditional loops (IF-ELSE and WHILE statements). The first example will illustrate one common mistake done using comparison operator ==:

```
>> a = sin(2*pi);   % should be zero
>> bool = (a==0)    % should be true --> bool = 1?
bool =
     0
>> a
a =
 -2.4493e-016        % Remember precision when using logical operators
```

A better way to do it is to set a precision number (in the following 1×10^{-6}) to check a statement:

```
>> bool = (abs(a)<1e-6)
bool =
     1
```

The standard comparison operators are listed in Table 3.

Table 3: MATLAB Comparison Operators

Operator	Description
$\sim a$	NOT: returns 1 if a = 0 and 1 if a = 0
a==b	returns 1 if a equals b (0 otherwise)
a < b	returns 1 if a is strictly smaller than b
a > b	returns 1 if a is strictly greater than b
a <= b	returns 1 if a is smaller or equal to b
a >= b	returns 1 if a is greater or equal to b
a ~= b	returns 1 if a is different from b
&	AND operator (equiv. to and(bool1,bool2)
|	OR operator (equiv. to or(bool1,bool2)
xor	Exclusive OR operaor

Using the operators AND and OR, it is easy to compute logical tables by the following code:

```
% test5.m - Truth tables for AND and OR

clear all

P = [1 1 0 0];
Q = [1 0 1 0];

% AND Table
for i=1:4
    tableAND(i) = P(i)&Q(i);
end

% OR Table
for i=1:4
    tableOR(i) = P(i)|Q(i);
end

TAB1 = [P' Q' tableAND']

TAB2 = [P' Q' tableOR']
```

With the following output:

```
TAB1 =             % Truth Table for AND
     1     1     1
     1     0     0
     0     1     0
     0     0     0
TAB2 =             % Truth Table for OR
     1     1     1
     1     0     1
     0     1     1
     0     0     0
```

We are now ready to present the elementary loops for programming in MATLAB.

5.2 IF-ELSE Loops

The prototype of and IF-ELSE Statement is the following:

```
if CONDITION 1
   ACTION 1;
elseif CONDITION 2 % cond. 1 is not fulfilled but cond. 2 is
   ACTION 2;        % neither cond. 1 or 2 is fulfilled
else
   ACTION 3;
end
```

For example, we want to write a function that checks wether a number is even positive or not. The following function does the computation and it returns a boolean 0 or 1. Note that we use the function *modulo* to check if the input number to the function is divisible by 2.

```
function bool = test6(a)

if a <= 0
    bool = 0;
elseif mod(a,2) ~= 0  % modulo operator
    bool = 0;
else
    bool = 1;
end
```

Successive calls to the functions are illustrate here.

```
>> test6(-3)
ans =
     0
>> test6(18)
ans =
     1
>> test6(13)
ans =
     0
```

5.3 FOR Loops

FOR Loops are very useful to perform successive operations on elements of a vector. The main element here is to use an incremental value. The model for that in MATLAB is the following:

```
for increment = initial_value:final_value
  ACTION 1;
  ACTION 2;
  ...
  ACTION N;
end
```

In this prototype, we used the unitary increment. But we can use any increment, for example, if we want to calculate square roots of odd integers between 1 and 25:

```
for i = 1:2:25
  SQRT = sqrt(i)
end
```

Please note that if you are dealing with elements in a vector, indice 0 can not be used (take note of the error message):

```
>> for i=0:10; v(i) =i; end
??? Subscript indices must either be real positive integers or
logicals.
```

FOR loops are also quite time consuming (as demonstrate in Exercise 9). It is usually better in MATLAB to deal with the vector dot-operations as defined earlier. In some cases (for example, when searching an element in a vector), it is usually better to use a WHILE loop.

5.4 WHILE Loops

WHILE loops enable us to repeat an action until a condition is matched (for example, convergence). The MATLAB prototype for this type of loops is the following:

```
initialisation of CONDITION
while CONDITION is TRUE
    ACTION 1;
    ACTION 2;
    ...
    ACTION N;
end
```

For example, if we want to know the number of positive integer that we need to sum up before reaching a given number x, we can use the following function:

```
function n = test7(x)

n = 0;
total = 0;
while total < x
    n = n + 1;   % incrementing the number of integer
    total = total + n;  % new sum
end
```

with the following usage (for x = 1000):

```
>> test7(1000)
ans =
    45
```

MATLAB allows the user to insert breakpoints within loops to terminate it (it by might useful for example if convergence is not achieved - see exercise 11. The prototype within a while loop would be the following:

```
WHILE CONDITION1 is TRUE
    ACTION1...N
    IF CONDITION2 is TRUE
        BREAK
    END
END
```

Use of the break point can be avoid by using CONDITION1 and CONDITION2 in the WHILE STATEMENT:

```
WHILE CONDITION1 is TRUE AND CONDITION2 is FALSE
    ACTION1...N
END
```

This will cause to terminate the loop whenever the condition 2 is met even if condition 1 is not.

5.5 SWITCH Loops

SWITCH loops are a simpler version of the IF-ELSEIF-ELSE loops presented earlier. They are quite useful for user input treatment and when cases can be foreseen easily. The prototype in MATLAB is the following:

```
switch (CASE)
case{CASE 1}
    ACTION 1;
case{CASE 2}
    ACTION 2;
case{CASE 3}
    ACTION 3;
...
case{CASE N}
    ACTION N;
otherwise
   ACTION N+1;
end
```

For example, the following code computes the inverse of a matrix if it is non-singular and throws an error if the matrix is closed to be singular.

```
function iA = test8(A)

a = det(A);

bool = (abs(a) < 1e-6);

switch bool
    case 0
        iA = inv(A);
    case 1
        error('Matrix might be singular');
end
```

Testing for a non-singular matrix and a singular matrix:

```
>> M = [1 2;3 4];
>> invM = test8(M)
invM =
   -2.0000    1.0000
    1.5000   -0.5000
>> M = [1 1; 1 1];
>> invM = test8(M)
??? Error using ==> test8 Matrix might be singular
```

We now suggest a few examples related to those simple loops before turning our attention to most useful capabilities of MATLAB.

Ex 8. Check, using the logical operators, the De Morgan's rules:

$$\begin{aligned} \sim (p \ \& \ q) &= \sim p \mid \sim q \\ \sim (p \mid q) &= \sim p \ \& \ \sim q \end{aligned}$$

Ex 9. We want to compare the computing time necessary for a FOR loop versus the time necessary for a dot-vectorial approach. In order to do so, we use the tic/toc functions in MATLAB:

```
>> tic
>> operation
>> t = toc
```

MATLAB will then output the time used by the operation. In a SCRIPT file, define the vector: $t = [0 : 0.0001 : 1]$ and compute the following function:

$$x(t) = \frac{e^t}{1 + e^t}$$

First, use the vector operation "./" and then compare the value with the following FOR loop:

```
clear x
t = [0:0.0001:1];
tic
for i = 1:length(t)
   x(i) = exp(t(i))/(1+exp(t(i)));
end toc
```

Try also the following code and compare the results:

```
clear x
t = [0:0.0001:1];
x = zeros(1,length(t));
tic
for i = 1:length(t)
   x(i) = exp(t(i))/(1+exp(t(i)));
end
toc
```

Ex 10. Write a function that decides if the roots of the quadratic equation

$$ax^2 + bx + c = 0$$

are distinct reals, identical reals or complex. Do it so that MATLAB returns you a message like: " The roots for the equation are...".

Ex 11. $\arctan(x)$ can be approximated by:

$$\arctan(x) \approx \sum_{m=0}^{M} \frac{(-1)^m x^{2m+1}}{2m+1}$$

for $|x| < 1$. We want to know the number of terms in the series that enable us to approximate the true value with an error less than 10^{-4} at a given x. Use a SWITCH loop (since this series expansion is not really good for some values, you may want to use a break point if the solution converged away from the true value).

Ex 12. To calculate the roots of a non-linear algebraic equation, we usually use Newton-Raphson method. The algorithm is the following:

```
0. pose x(0)  % initial estimate
1. while f(x_k) > given criteria
  1.1 Compute x(k+1) = x(k) - f(x_k)/f'(x_k)
  1.2. k = k+1
  1.3. evaluate f(k)
2. exit
```

Using this algorithm, find the roots of

$$2x - 2^x = 0$$

Check the effect of the initial estimate on the number of iterations needed to converge.

6 Advanced Functions in MATLAB

The different topics treated in this last section are quite specific. Students should consult each subsections independently, following their needs.

6.1 Advanced Graphics

We already covered the basic MATLAB graphical function *plot*. Here, we will cover 3 main topics: how to plot multiple graphs in the same figure (using different windows); how to use different axis and how to export a figure to a file.

Figure 4 presents a typical bad application of plotting in MATLAB. Plotting reactor Temperature and Concentration on the same figure.

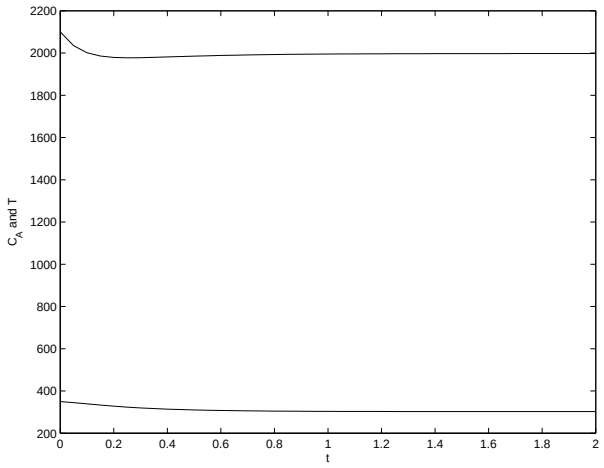

Figure 4: C(t) and T(t) on the same plot

Figure 5 presents a convenient way to present both graphs on the same figure:

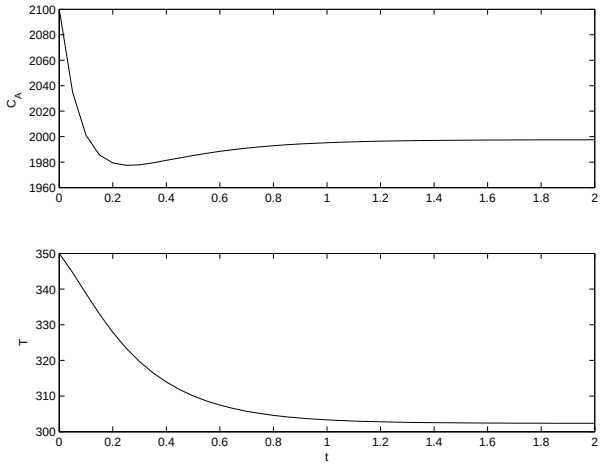

Figure 5: C(t) and T(t) on same figure, different plots

The code needed to do so in the following:

```
figure(1) % declare a figure

subplot(2,1,1) % first digit:  number of rows in the figure
               % second digit: number of columns in the figure
               % third digit: the window where we will plot

plot(t,C)      % Plot Concentration
ylabel('C [=] mol/m^3')

subplot(2,1,2)  % third digit we will now plot in the second window
plot(t,T)      % Plot Temperature
ylabel('T [=] K')
xlabel('t [=] hr')
```

For some applications -i.e. to compute kinetic rate laws, we might need to use different axis type. Calls different from *plot* are listed in Table 4.

When using *subplot*, the iteration on the last digit follows the way we counted elements in matrix. For example, *subplot(2,2,4)* refers to the 4th matrix (bottom right) window of a 2×2 figure.

Finally, we mention that the best way to save a given figure is not by using copy and paste. You can always use the FILE-EXPORT button in the figure window or, directly in the code, using the command *print*:

Table 4: MATLAB plotting types

Function	Description
semilogx	x axis is logarithmic
semilogy	y axis logarithmic
loglog	x and y axis are log
polar	plot in polar coord.
mesh	3 dimensional plots

```
print -deps solfig2.eps
```

Here, we used the Encapsulated Postcript Driver (eps), but others are available, as describe if you type *help print* from the command line. We now suggest you a few exercises.

Ex 13. Datas from a batch experiment are listed in Table 5.

Table 5: Batch Datas for Ex. 13.

t(min)	C_A (mol L^{-1})
1	1.00
2	0.90
3	0.80
4	0.73
5	0.67
6	0.62
7	0.58
8	0.55
9	0.53
10	0.52

On the same figure (i.e. using *subplot*), plot the following graphs:
a - $C_A(t)$ as a function of t.
b - $\frac{-dC_A(t)}{dt}$ as a function of C_A.
c - $\ln \frac{-dC_A(t)}{dt}$ as a function of $\ln(C_A)$ on a *loglog* graph.

Note that you can evaluate the derivatives numerically for i= 1,...,10 using:

$$\begin{aligned}\left(\frac{dC_A}{dt}\right)_{i=1} &\approx \frac{-3C_{A1}+4C_{A2}-C_{A3}}{2\Delta t}\\ \left(\frac{dC_A}{dt}\right)_{i=2,\ldots,9} &\approx \frac{C_{A(i+1)}-C_{A(i-1)}}{2\Delta t}\\ \left(\frac{dC_A}{dt}\right)_{i=10} &\approx \frac{C_{A8}-4C_{A9}+3C_{A10}}{2\Delta t}\end{aligned}$$

Ex 14. We want to plot, using MATLAB, the following function:

$$z(x,y) = e^{-0.5[x^2+0.5(x-y)^2]}$$

for $x \in [-4,4]$ and $y \in [-4,4]$. Using two FOR loops, calculate the function values using increment of 0.05 (don't forget, the result should be a Matrix). Plot the resulting surface using the MATLAB function *mesh(xvector,yvector,zmatrix)* (check the call of the function if you are not sure). Compare the resulting output with the similar functions *surf(xvector,yvector,zmatrix)* and *contour(xvector,yvector,zmatrix)*.

6.2 Importing and Exporting Data

We will now show how to export and import data from and to MATLAB. The simplest way to save data from MATLAB is using the command *save*. MATLAB then saves all the variables stored in the Workspace in a *.mat file (that includes, of course, data generated from a SCRIPT, since those variables "live" in the Worspace). The following short example illustrates the way to use *save* and *load*.

```
>> A = ones(3);     % 3x3 matrix
>> b = zeros(3,1);  % 3x1 vector
>> save my_session  % save the Workspace variables in my_session.mat
>> ls               % use to list the files in the current directory
.
..
my_session.mat
>> clear            % clear all the variables
>> whos             % to show that A, b are not in the workspace anymore
>> load my_session  % re-load the file
>> whos
  Name      Size                    Bytes  Class
  A         3x3                        72  double array
  b         3x1                        24  double array
Grand total is 12 elements using 96 bytes
```

The main drawback of *.mat files is that they are not recognized by other softwares (such as Excel, for example). The best way to save data in a general

fashion is by forcing MATLAB to save them in a ASCII file, easily readable from Excel. To save the matrix A from the last example, we would use the following lines:

```
>> save my_session.dat -ASCII A
>> load my_session.dat
```

The format name ".dat" is not really important (but it is the usual name for those files). It can be read by Notepad as well.

Another useful function to import data to MATLAB is the function *xlsread* that enables the user to import the content of an excel spreadsheet to MATLAB directly (without going through the Excel - export to .dat procedure, which also works).

```
>> xlsread my_xls_spreadsheet.xls
```

To have more information, you can read the information from *help fileformats*. The next exercises will illustrate the interaction with Excel and how to use structures to save more complicated sets of data.

Ex 15. Save the matrix $z(x, y)$ calculated in exercise 14 in an ASCII file (.dat). Open it with Excel.

Ex 16. MATLAB enables the user to save more complicated structures using variable cast as *cells*. For example, consider the function given in exercise 14:

$$z(x, y) = e^{-a[x^2+b(x-y)^2]}$$

with $a = 0.5$ and $b = 0.5$ over the domain $x = -2 : 0.05 : 2$ and $y = -2 : 0.05 : 2$. We can stock all this information in one structured variable and save different trials as above using *save*. To define this structure, we can use the following:

```
>> trial1 = struct('x',x,'y',y,'z',z,'a',a,'b',b);
trial1 =
    x: [161x1 double]
    y: [161x1 double]
    z: [161x161 double]
    a: 0.5000
    b: 0.5000
```

We can access elements of this structure by:

```
>> trial1.a
ans =
       0.5000
```

Write a code that varies the parameter a between 0 and 1 by increment of 0.1 and save the result in 11 different structures as described above (you may also plot the result using your knowledge from exercise 12). Save all resulting structures in a cell array (Cstruct$\{i\}$ = struct(...)) . Save the result in a .mat file.

6.3 Roots of an Algebraic Equation

This section covers two functions: the function *roots* to calculate the roots of a polynomial expression and the function *fzero* that computes the solutions of a nonlinear algebraic expression using the Newton-Raphson method.

For example, if we want to solve the following degree 5 polynomial with MATLAB

$$x^5 + 3x^4 - 8x^3 + 12x^2 - x + 4 = 0$$

We will use the following MATLAB command:

```
>> coeff = [1 3 -8 12 -1 4];
>> roots(coeff)
ans =
  -5.0623
   1.1051 + 1.1056i
   1.1051 - 1.1056i
  -0.0739 + 0.5638i
  -0.0739 - 0.5638i
```

Note that the coefficients are entered as a vector of decreasing power. Another example would be the following:

$$x^2 + 1 = 0$$

and the corresponding solution is:

```
>> coeff = [1 0 1];
>> roots(coeff)
ans =
        0 + 1.0000i
        0 - 1.0000i
```

If we now want to solve the equation of Exercise 12:

$$2x - x^2 = 0$$

we will use the function *fzero*. The argument for that functions are the function to be evaluated and an initial condition. The function might be an usual FUNCTION file:

```
function f = testroot(x)
f = 2*x - x^2;
```

The call to *fzero* would be:

```
>> sol = fzero('testroot',0.5)
sol =
     2;
```

Or we can declare an *inline* function and solve everything in the worskspace:

```
>> f = inline('2*x - 2^x')  % inline function declaration
f =

     Inline function:
     f(x) = 2*x - 2^x
>> f(3)                 % Evaluate the function at x = 3
\ans =
    -2
```

Then, the call to *fzero* would be:

```
>> sol = fzero(f,0.5)
sol =
     1
```

Ex 17. Write a program that tests the solution to the expression

$$2x - x^2 = 0$$

for different values of initial condition $x0$ between $[-2, 2]$ by successive calls to *fzero*. Report those values in a simple plot.

Ex 18. Write a short program that calculates the number of real roots of a polynomial expression. To do so, you might use the function *isreal* from MATLAB.

6.4 FSOLVE Function

If we want to solve a set of nonlinear algebraic equations (such as the steady state solution to a non-isothermal Continuous Stirred Tank Reactor), the MATLAB function *fsolve* is to be used. *fsolve* uses a quasi-Newton method. For example, if we want to solve the following set of nonlinear equations:

$$\begin{aligned} f_1(x_1, x_2) &= x_1 - 4x_1^2 - x_1 x_2 \\ f_2(x_1, x_2) &= 2x_2 - x_2^2 - 3x_1 x_2 \end{aligned}$$

The first thing to do is to write a FUNCTION file:

```
function f = nle(x)
f(1) = x(1) - 4*x(1)^2 - x(1)*x(2);
f(2) = 2*x(2) - x(2)^2 + 3*x(1)*x(2);
```

The call to *fsolve* would be the following (using $[1; 1]$ as initial guess):

```
>> x = fsolve('nle',[1;1])
x =
    0.2500
    0.0000
```

NOTE that *fsolve* is not included in the basic MATLAB installation. In exercise 20, you will be asked to program yourself a Newton method to solve exercise 19 covering the steady-state solution of non-isothermal CSTR.

Ex 19. Consider the model of a non-isothermal CSTR with a 1st order exothermic reaction $A \rightarrow B$:

$$\begin{aligned} \frac{dC_A}{dt} &= \frac{F}{V}(C_{A0}(t) - C_A(t)) - k_0 e^{\frac{-E}{RT(t)}} C_A(t) \\ \frac{dT}{dt} &= \frac{F}{V}(T_0(t) - T(t)) + \frac{-\Delta H}{\rho Cp} k_0 e^{\frac{-E}{RT(t)}} C_A(t) - UA\frac{(T(t) - T_J}{\rho Cp} \end{aligned}$$

and parameters listed in Table 6:

Table 6: Parameters for Ex. 19.

Parameters	Values
E	9×10^4 J/mol
k_0	17×10^{12} hr^{-1}
$-\Delta H$	9×10^4 J/mol
ρCp	1.1×10^6 J/(m^3 K)
R	8.314 J/(mol K)
U	4.75×10^5 J/(hr K m^2)
A	8.2 m^2
V	15 m^3
F	60 m^3/hr
C_{A0}	2000 mol/m^3
T_0	303 K
T_J	288 K

Using *fsolve*, compute the solution of the system at steady-state (i.e. when both derivatives with respect to time are zero). Use the following initial guess:

a - $(C_A(0), T(0)) = (2100, 350)$;
b - $(C_A(0), T(0)) = (200, 450)$;
c - $(C_A(0), T(0)) = (1100, 375)$;

Ex 20. Solve Exercise 19 using the Newton method:

$$\begin{pmatrix} x_1(k+1) \\ x_2(k+1) \end{pmatrix} = \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} - J(x(k))^{-1} \begin{pmatrix} f_1(x(k)) \\ f_2(x(k)) \end{pmatrix}$$

where J is the jacobian of the system evaluated at $x(k)$. You may want to use symbolic manipulation (presented below) to compute the Jacobian. Use a while loop with a norm on a vector criteria to check convergence.

6.5 Regression

In this subsection, we present an easy way to do regression using MATLAB and to compute the resulting model. Let's suppose we have the following two data vectors:

```
>> x = [0 1 2 3 4 5]; % independent variable
>> y = [0 2.2 4.5 5.4 5.5 5.5];  % dependent variable
```

The following lines of commands show how to construct a 3rd order and a second order polynomials for this data.

```
>> poly_model3 = polyfit(x,y,3) % 3rd order polynomial
poly_model3 =
    0.0074   -0.4091    2.9636   -0.0865
>> poly_model2 = polyfit(x,y,2)
poly_model2 =
   -0.3536    2.8621   -0.0643
```

Both models are thus given by:

$$\begin{aligned} y_{m,O3} &= 0.0074x^3 - 0.4091x^2 + 2.9636x - 0.0865 \\ y_{m,O2} &= -0.3536x^2 + 2.8621x - 0.0643 \end{aligned}$$

Figure 6 shows the simple plot obtain after reconstruction of both polynomial models:

```
>> ym3 = polyval(poly_model3,x);
>> ym2 = polyval(poly_model2,x);
>> plot(x, y,'x', x, ym2,'*', x, ym3,'+')
```

The following example illustrates the use of a random vector generated by MATLAB.

Ex 21. We want to approximate the function

$$y(t) = 10 * e^{-t} + k(t)$$

for $0 \leq t \leq 10$ (using increments of 0.1) by an order 5 polynomial. Parameter $k(t)$ is generated by the function *rand*:

```
k = rand(length(t),1) % t is a column vector
```

Using a simple plot, compare the resulting model and the true function.

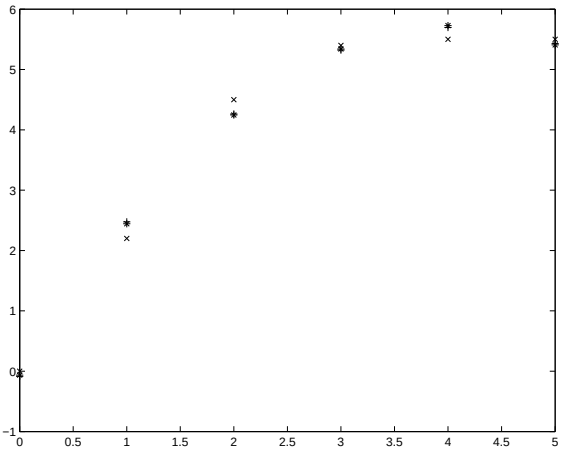

Figure 6: Regression example

6.6 Numerical Integration

Numerical integration is quite important in chemical engineering (i.e. in Reactor design, in separation principles). There exists several numerical methods to approximate an integral, but we will limit ourselves to the most common integration method in MATLAB, the function *quad*. It is an adaptive Simpson quadrature method (refer to a numerical analysis book for more info). For example if we want to integrate the following function:

$$\int_0^1 \frac{\sin(x)}{x} dx$$

We can use the following code:

```
>> S = quad('sin(x)./x', 0, 1)
S =
    0.9461
>>
```

In the following exercise, we will compare the performance of *quad* with a Simpson 3/8 quadrature.

Ex 22. We want to evaluate the following integral:

$$\int_0^1 \frac{1}{1+\sqrt{(}x)} dx$$

First, write a MATLAB function that implement the SIMPSON 3/8 quadrature:

$$\int_0^1 f(x)dx \quad \approx \quad \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3)$$

where the indices $0, 1, 2, 3$ refers to the equidistant points where the function should be evaluate (i.e. $f_0 = f(0)$, $f_1 = f(0.33)$, $f_2 = f(0.66)$ and $f_3 = f(1)$). Compare the solution to the one obtain using *quad*.

6.7 Solving ODEs with Initial Conditions

One of the most useful feature of MATLAB for chemical engineers is the the ode solvers suite. One usually learns the standard numerical algorithm to compute the solution of non-linear differential equations: Euler methods (*ode23*) and Runge-Kutta (*ode45*). The methods built-in MATLAB are more advance than the classical scheme we learn (mainly because they adapt the grid of integration with respect of the system stiffness). Here, we won't present how to program your own odesolver (even if it is a good exercise) since ode solvers from MATLAB are now widely accepted. The prototype for a call the a ode solver (here ode45) is the following:

```
>> [t, x] = ode45('diff_fun', [tspan vect], [init. cond. vect])
```

where the outputs are time (t) and the states (x). The inputs are the function name (can be inline or in a function file), the vector of integration time (can be of the form of $[t0\ tf]$ for initial time $t0$ and final time tf or $[t0 : incr : tf]$ where you fix the increment of the output) and the column vector of initial conditions.

For example, we want to compute numerically the following system of differential equations:

$$\begin{aligned} \frac{dx_1}{dt} &= a\sin(x_1) + \cos(x_2) \\ \frac{dx_2}{dt} &= a\cos(x_1) + b\sin(x_2) \end{aligned}$$

with initial conditions $x_1(0) = 0$ and $x_2(0) = 1$ and parameters $a = 1$ and $b = 0.5$. Using that information, we can build the FUNCTION file (that will be called by the ode solver:

```
function dxdt = diff_fun(t,x)

% parameters
a = 1;
b = 0.5;

% differential equations
dxdt(1,1) = a*sin(x(1)) + cos(x(2));
dxdt(2,1) = cos(x(1)) + b*sin(x(2));
```

We can of course call ode45 from the prompt line as above:

```
>> [t,x] = ode45('diff_fun', [0 10], [0;1]);
```

or using the SCRIPT (usually a good choice for plotting):

```
clear all
close all
clc        % clc clears the output from the command window

tspan = [0 10]; % we could also have used [0:0.01:1]
init = [0;1];

[t,x] = ode45('diff_fun', tspan, init);

figure(1)
subplot(2,1,1)
plot(t,x(:,1))    % note that x1 is returned as the first column vector
ylabel('x_1')
subplot(2,1,2)
plot(t,x(:,2))    % note that x2 is returned as the second column vector
ylabel('x_2')
xlabel('t')
```

The graphical output will be as shown in Figure 7.

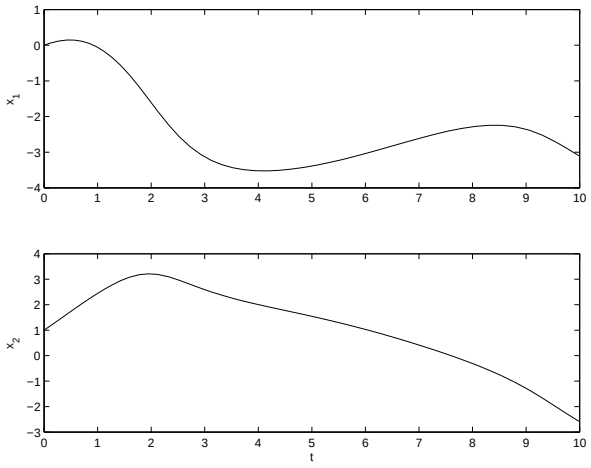

Figure 7: ODE solver output example

The following example covers the isothermal CSTR.

Ex 23. Write a SCRIPT and a FUNCTION that solve the following system of non-linear differential equations:

$$\begin{aligned}\frac{dC_A(t)}{dt} &= 0.5714(10 - C_A(t)) - \left(\frac{5}{6}\right) C_A(t) - \left(\frac{1}{6}\right) C_A^2(t) \\ \frac{dC_B(t)}{dt} &= -0.5714 C_B(t) + \left(\frac{5}{6}\right) C_A(t) - \left(\frac{5}{3}\right) C_B(t)\end{aligned}$$

With initial conditions $C_A(0) = 5$ mol/L and $C_B(0) = 0$ mol/L. Plot both $C_A(t)$ and $C_B(t)$ as a function of time for 10 time units (use subplot structure).

6.8 Symbolic Variables

MATLAB enables manipulation of symbolic variables (just as Maple). In fact, the symbolic toolbox in MATLAB is based on some old Maple kernel. Since MATLAB usually uses numerical variables, we have to declare symbolic variables before manipulating them. Here, we present an example on how to compute the Jacobian of a nonlinear 2×2 algebraic system in a SCRIPT file.

```
clear all
close all
clc

syms z1 z2 real % symbolic variables declaration

f1 = z1 - z2^2 + 4;      % first function declaration
f2 = 5*(1-exp(-z1/66));  % second function declaration

A = jacobian([f1;f2],[z1 z2])
```

with the following output:

```
A = [               1,              -2*z2]
    [5/66*exp(-1/66*z1),0]

>> whos
  Name              Size                Bytes          Class
  A                 2x2                   354          sym object
  f1                1x1                   142          sym object
  f2                1x1                   158          sym object
  z1                1x1                   128          sym object
  z2                1x1                   128          sym object
Grand total is 63 elements using 910 bytes
```

We can now manipulated the symbolic object A. For example, if we want to evaluate the Jacobian at $(z1, z2) = (0, 0)$, we use the function *feval*:

```
>> z1 = 0;
>> z2 = 0;
>> A1 = eval(A)
A1 =
    1.0000      0
    0.0758      0
>>
```

Another important symbolic manipulation in CHEE222 involves the Laplace transform. We will not cover it here, but it mainly involves the usual matrix operation (i.e. *det*,...) with symbolic expressions. The following examples cover the most simple symbolic operations for CHEE222.

Ex 24. Using the function *solve* and using only symbolic variables, find roots to the equation

$$x^2 + 1 = 0$$

Ex 25. Consider the following expression:

$$y(t) = t^3 + cos(t)$$

Using symbolic variables and the function *diff*, compute the expression of $\frac{dy}{dt}$ and evaluate it at $t = 5$ using function *eval*.

References

B.W. Bequette. *Process Dynamics - Modeling, Analysis, and Simulation*. Prentice-Hall, Upper Saddle, NJ, 1998. ISBN 0132068893.

B.W. Bequette. *Process Control - Modeling, Design and Simulation*. Prentice-Hall, Upper Saddle, NJ, 2003. ISBN 0133536408.

A. Constantinides and N. Mostoufi. *Numerical Methods for Chemical Engineers with MATLAB Applications*. Prentice-Hall, Upper Saddle, NJ, 1999. ISBN 0130138517.

D. Hanselman and B. Littlefield. *Mastering MATLAB 7*. Prentice-Hall, Upper Saddle, NJ, 2004. ISBN 0131430181.