Matlab Notes for Mathematical Modeling

Lia Vas

Content

1. Review of Matlab in Calculus 1

1.1 Basic Arithmetic

MATLAB can do anything your calculator can do and much more. You can use +, -, *, \ and ^ to add, subtract, multiply, divide or exponentiate, respectively. For example if you enter:
>> 2^3 - 2*2
MATLAB gives you the answer: **ans = 4**

If you want to perform further calculations with the answer, you can type **ans** rather than retyping the whole answer. For example,
>> sqrt(ans) **ans = 2**

To perform symbolic calculations in MATLAB, use **syms** to declare the variables you plan to use. For example, suppose that you need factor x^2-3x+2. First you need
>> syms x (you are declaring that *x* is a variable)

Then, for example, you can use the command **factor**.

```
>> factor(x^2-3*x+2)
ans =  (x-1)*(x-2)
```

Note that we entered **3*x** to represent 3*x* in the command above. **Entering * for multiplication is always necessary in MATLAB.**

1.2 Solving Equations

For solving equations, you can use the command **solve**. The command solve is always followed by parenthesis. After that, you type the equation you would like to solve in single quotes. Then type coma, and variable for which you are solving in (single) quotes. Thus, the command solve has the following form

solve('*equation***', '***variable for which you are solving***')**

For example, to solve the equation x^3-2x-4, you can use:

```
>> solve('x^3-2*x-4=0')
```

and get the following answer:

```
ans =
[   2]
[ -1+i]
[ -1-i]
```

Here *i* stands for the imaginary number $\sqrt{-1}$. This answer tells us that there is just one real solution, 2.

MATLAB can give you both symbolic and numerical answer. For example, to solve the equation 3x^2-8x+2=0, you can use

```
>> solve('3*x^2-8*x+2=0','x')
```

Matlab produces the following

```
ans =
[ 4/3+1/3*10^(1/2)]            [ 4/3-1/3*10^(1/2)]
```

If we want to get the answer in the decimal form with, say, three significant digits, we can use the command **vpa**.

```
>> vpa(ans, 3)
ans =          [ 2.38]          [ 0.28]
```

The command **vpa** has the general form

vpa(*expression you want to approximate***,** *number of significant digits***)**

You can solve more than one equation simultaneously. For example suppose that we need to solve the system x^2+ x+ y^2 = 2 and 2x-y = 2. We can use:

```
>>  [x,y] =solve( 'x^2+ x+ y^2  =  2', '2*x-y  =  2')
```

to get the answer

```
x =     [ 2/5]          [   1]
y =     [ -6/5]         [   0]
```

Note that the **[x,y]=** part at the beginning of the command was necessary since without it MATLAB produces the answer:

```
ans =          x: [2x1 sym]          y: [2x1 sym]
```

This answer tells us just that the solutions are two values of the pair (x,y) but we do not get the solutions themselves. To get the solution vectors displayed, we must use **[x,y]=** before the command **solve**.

You can solve an equation in two variables for one of them. For example the command

```
>> solve('y^2-5*x*y-y+6*x^2+x=2', 'y')
```

solves the given equation for values of *y* in terms of *x*. The answer is:

```
ans =          [ 3*x+2]        [ 2*x-1]
```

1.3 Representing a function

The following table gives an overview of how most commonly used functions or expressions are represented in MATLAB.

function or symbol	representation in MATLAB
e^x	exp(x)
ln x	log(x)
log x	log(x)/log(10)
log. base a of x	log(x)/log(a)
sin x	sin(x)
cos x	cos(x)
arctan(x)	atan(x)
π	pi

To represent a function, use the command **inline**. Similarly to solve, this command is followed by parenthesis and has the following form:

inline('*function*', '*independent variable of the function*')

Here is how to define the function x^2+3x-2:

```
>> f = inline('x^2+3*x-2', 'x')
f =
   Inline function:
   f(x) = x^2+3*x-2
```

After defining a function, we can evaluate it at a point. For example,

```
>> f(2)
ans =     8
```

In some cases, we will need to define function f as a vector. Then we use:

```
>> f = inline(vectorize('x^2+3*x-2'), 'x')
f =     Inline function:
   f(x) = x.^2+3.*x-2
```

In this case, we can evaluate a function at more than one point at the same time. For example, to evaluate the above function at 1, 3 and 5 we have:

```
>> f([1 3 5])
ans =     2   16   38
```

If a function is short, it might be **faster to evaluate a function at a point simply by typing the value of *x* directly for *x***. For example, to evaluate sin(*x*) at *x*=2, simply type
>> sin(2) and obtain the answer **ans = .909297**

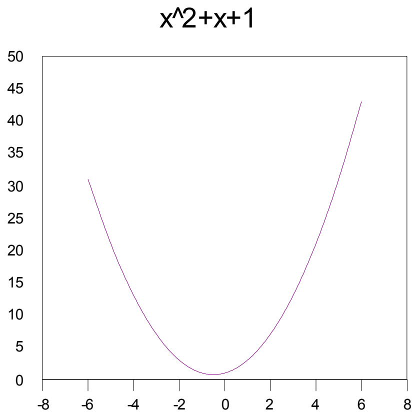
As when using the calculator, one must be careful when representing a function. For example

- $\frac{1}{x(x+6)}$ should be represented as **1/(x*(x+6))** not as **1/x*(x+6)** nor as **1/x(x+6),**
- $\frac{3}{x^2+5x+6}$ should be represented as **3/(x^2+5*x+6)** not as **3/x^2+5*x+6,**
- e^{5x^2} should be represented as **exp(5*x^2)** not as **e^(5*x^2), exp*(5*x^2), exp(5x^2)**

nor as **exp^(5*x^2).**

- Ln(x) should be represented as **log(x)**, not **ln(x).**
- $\log_3(x^2)$ should be represented as **log(x^2)/log(3)** not as **log(x)/log(3)*x^2.**

1.4 Graphic



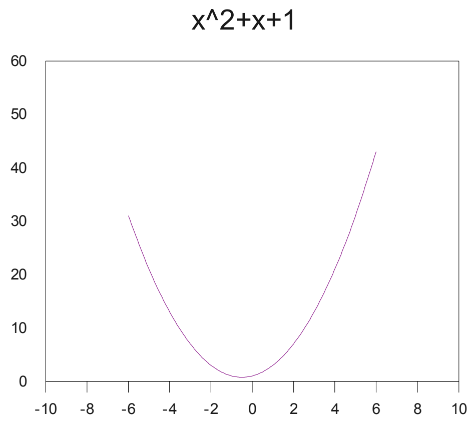
Let us start by declaring that *x* is a variable:
>> syms x

The simplest command in MATLAB for graphing is **ezplot**. The command has the following form

ezplot(*function*)

For example, to graph the function x^2+x+1, you can use
>> ezplot(x^2+x+1)

A new window will open and graph will be displayed. To copy the figure to a text file, go to **Edit** and choose **Copy Figure**. Then place cursor to the place in the word file where you want the figure to be pasted and choose **Edit** and **Paste**.

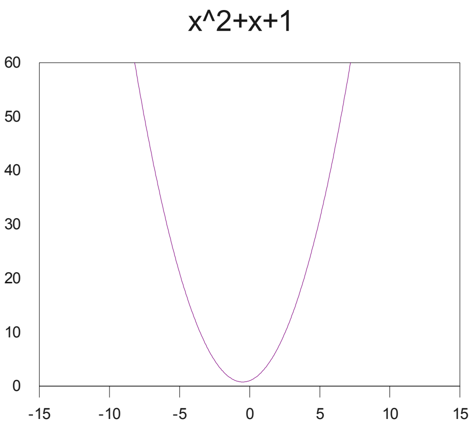


We can specify the different scale on *x* and *y* axis.
To do this, the command **axis** is used.
It has the following form

axis([x_{min}, x_{max}, y_{min}, y_{max}**])**

This command parallels the commands in menu WINDOW on the TI83 calculators.

For example, to see the above graph between x-values -10 and 10 and y-values 0 and 60, you can enter
>> axis([-10 10 0 60])



Note that the domain of function did not change by command axis. To see the graph on the entire domain (in this case [-10, 10]), add that domain after the function in the command ezplot:

ezplot(*function*, [x_{min}, x_{max}**])**

In this case,
>> ezplot(x^2+x+1, [-10, 10])
will give you the desired graph.

For the alternative command for graphics, **plot,** you can find more details by typing **help**.

To plot multiple curves on the same window, you can also use the **ezplot** command. For example:
>> ezplot(sin(x))
>> hold on
>> ezplot(exp(-x^2))

>> hold off

1.5 Solving equations using **fzero**

In some cases, the command **solve** may fail to produce all the solutions of an equation. In those cases, you can try to find solutions using **fzero** (short for "find zero") command. In order to use the command, first you need to write equation in the form

$$f(x)=0.$$

Thus, <u>put all the terms of th equations on one side</u> leaving just zero on the other. To find a solution near the *x*-value x=*a*, you can use

fzero('*left side of the equation***',** *a***)**

The command **fzero**, similarly as **solve** is always followed by parenthesis. The equation should be in single quotes.

If it is not clear what a convenient *x*-value *a* should be, you may want to graph the function on the left side of the equation first, check where it intersects the *x*-axis (alternatively graph left and right side of the equation if it is not in *f(x)=0* form and see where the two functions intersect) and then decide which solution you need to find.

For example, to solve the equation $e^{x^2}-2=x+4$, we can first graph the functions on the left and right side of the equation using

syms x   ezplot(exp(x^2)-2)   hold on   ezplot(x+4)   hold off

From the graph, we can see that the two functions intersect at a number near -1 and at a number near 1. To use **fzero**, we need to represent the equation in the form $e^{x^2}-2-(x+4)=0$ (or simplified form $e^{x^2}-x-6=0$). Then, we can find the positive solution by using **fzero** to find a zero near 1 and then to find the negative solution near -1, for example. Thus, both solutions can be obtained by:

```
>> fzero('exp(x^2)-2-(x+4)', -1)          ans = -1.248
>> fzero('exp(x^2)-2-(x+4)', 1)           ans = 1.415
```

Note also that the command **solve('exp(x^2)-2=x+4', 'x')** returns just the positive solution. Thus, knowing how to use **fzero** command may be really useful in some cases.

1.6 Differentiation

Start by declaring x for a variable. The command for differentiation is **diff**. It has the following form

diff(*function***)**

For example,

```
>> syms x
>> diff(x^3-2*x+5)
```

gives us the answer **ans = 3*x^2-2**

To get n-th derivative use

diff(*function, n***)**

For example, the 23rd derivative of sin(x) is obtained as follows.

```
>> diff(sin(x), 23)                 ans =-cos(x)
```

1.7 Integration

We can use MATLAB for computing both definite and indefinite integrals using the command **int**. For the indefinite integrals, start with **syms x** followed by the command

int(*function***)**

For example, the command
>> int(x^2)
evaluates the integral $\int x^2 dx$ and gives us the answer **ans = 1/3*x^3**

For definitive integrals, the command is

int(*function, lower bound, upper bound***)**

For example,
>> int(x^2, 0, 1)
evaluates the integral $\int_0^1 x^2 dx$ The answer is **ans = 1/3**

In MATLAB **Inf** stands for positive infinity. MATLAB can evaluate the (convergent) improper integrals as well. For example:
>> int(1/x^2, 1, Inf) **ans = 1**
For the divergent integrals, MATLAB gives us the answer infinity. For example:
>> int(1/x, 0, 1) **ans = inf**

1.8 Limits

You can use **limit** to compute limits, left and right limits as well as infinite limits. For example, to evaluate the limit when $x \to 2$ of the function $\frac{x^2-4}{x-2}$, we have:

>> syms x
>> limit((x^2-4)/(x-2), x, 2) **ans = 4**

For left or right limits:
>> limit(abs(x)/x, x, 0, 'left') **ans = -1**
>> limit(abs(x)/x, x, 0, 'right') **ans = 1**
Limits at infinity:
>> limit(exp(-x^2-5)+3, x, Inf) **ans = 3**

2. Review of the use of Matlab in Calculus 2 and 3

2.1 Parametric Plots

We can use the command **ezplot** to graph a parametric curve as well. For example, to graph a circle $x = \cos t$, $y = \sin t$ for $0 \le t \le 2\pi$, we have:
>> ezplot('cos(t)', 'sin(t)', [0, 2*pi])

2.2 Special effects

You can change the title above the graph by using the command **title**. For example,
>> ezplot('x^2+2*x+1')
>> title 'a parabola'

You can add labels to *x* and *y* axis by typing **xlabel** and **ylabel.** You can also add text to your picture. For example, suppose that we want to add a little arrow pointing to the minimum of this function, the point (-1, 0). We can do that with the command:

>> text(-1, 0, '(-1, 0) \leftarrow a minimum')

2.3 Derivatives of Multivariable Functions

The command **diff** can be used to compute **partial derivatives**. For example, to find the first partial derivative with respect to y of the function x^2y^3, use

```
>> syms x y
>> diff(x^2*y^3, y)                ans =   3*x^2*y^2
```

To find the second partial derivative with respect to *x*, use

```
>> diff(x^2*y^3, x, 2)             ans =   2*y^3
```

For the mixed second partial derivative, one can use:

```
>> diff( diff(x^2*y^3, x), y)   ans =   6*x*y^2
```

2.4 Graphing in Three-Dimensional Space

The command for drawing 3D curves is **ezplot3**. For example, suppose that we need to graph the helix $x = \cos t$, $y = \sin t$, $z = t$, for $-10 \le t \le 10$.

>> ezplot3('cos(t)', 'sin(t)', 't', [-10, 10])

There are two commands for graphing surfaces: **ezmesh** and **ezsurf**. The first produces a transparent wired plot, a mesh surface. The second produces a shaded, nontransparent surface. Both commands can be used for graphing surfaces that are given in the form $z = f(x, y)$. For example, the graph of the cone $z = \sqrt{x^2+y^2}$ over the square $-7 \le x \le 7$, $-7 \le y \le 7$, can be obtained by **ezmesh('sqrt(x^2+y^2)', [-7, 7], [-7, 7])** or by **ezsurf('sqrt(x^2+y^2)', [-7, 7], [-7, 7]).**

Ezmesh and ezsurf are also used when graphing the surfaces that are given in parametric form

$$x = f(s, t) \qquad y = g(s, t) \qquad z = h(s, t)$$

The cone $z = \sqrt{x^2+y^2}$ can be represented by parametric equations as $x = r\cos t$, $y = r\sin t$, $z = r$. Using this representation, we can get the graph by **ezmesh('r*cos(t)', 'r*sin(t)', 'r', [0 , 10, 0, 2*pi])** or by **ezsurf('r*cos(t)', 'r*sin(t)', 'r', [0 , 10, 0, 2*pi]).**

We can plot the vector field **F**$(x, y) = P(x, y)$ **i** + $Q(x, y)$ **j** by using the command **quiver(x, y, P(x,y), Q(x,y))**. For example, we plot the vector field **F**$(x, y) = x$ **i** - y **j** as follows:

```
>> [x,y]=meshgrid(-1:.2:1, -1:.2:1);
>> quiver(x, y, x, -y);
>> axis equal
```

In 3D, we can also plot the vector field **F**$(x, y, z) = P(x, y, z)$ **i** + $Q(x, y, z)$ **j** + $R(x, y, z)$ **k** by using the command **quiver3(x, y, z, P(x,y,z), Q(x,y,z), R(x,y,z)).** For example, we plot the field **F**$(x, y, z) = x$ **i** + y **j** + z **k** as follows:

```
>> [x,y,z]=meshgrid(-3:1:3, -3:1:3, -3:1:3);
>> quiver3(x, y, z, x, y, z);
>> axis equal
```

3. Differential Equations

3.1 Basics of Differential Equations

We can use MATLAB to solve differential equations. The command for finding the symbolic solution is **dsolve**. For that command, the derivative of the function *y* is represented by **Dy**. The command has the following form:

dsolve('*equation***', '***independent variable***')**

For example, suppose that we want to find the **general solution** of the equation $x\,y' - y = 1$. You can do that by:

```
>> dsolve('x*Dy-y=1', 'x')                    ans = -1+x*C1
```

This means that the solution is any function of the form $y = -1 + cx$, where *c* is any constant.

If we have the initial condition, we can get the **particular solution** on the following way:

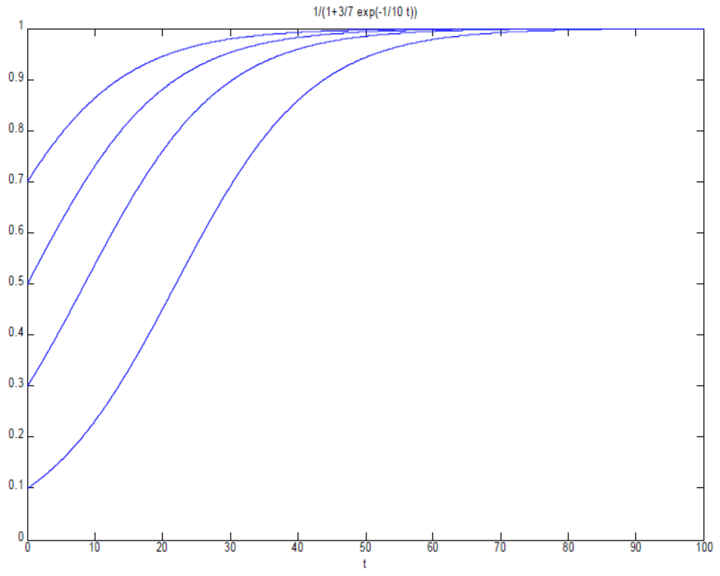
dsolve('*equation***', '***initial condition***', '***independent variable***')**

For example, to solve $x\,y' - y = 1$ with the initial condition $y(1)=5$, we can use:

```
>> dsolve('x*Dy-y=1', 'y(1)=5', 'x')          ans = -1+6*x
```

To graph this solution, we can simply type:

```
>> ezplot(ans)
```



We can graph a couple of different solutions on the same chart by using **hold on** and **hold off** commands. For example, to graph the solutions of differential equation y'=0.1y(1-y) for several different initial conditions, y(0)=0.1, y(0)=0.3 y(0)=0.5 and y(0)=0.7, first find the four particular solutions, then graph them using **hold on** and **hold off** commands. To observe the limiting behavior of the solution, in examples below the interval [0, 100] is used as the domain.

Finding the four solutions:

```
>> dsolve('Dy=0.1*y*(1-y)','y(0)=0.1', 'x')   ans = 1/(1+9*exp(-1/10*x))
>> dsolve('Dy=0.1*y*(1-y)','y(0)=0.3' 'x')    ans = 1/(1+7/3*exp(-1/10*x))
>> dsolve('Dy=0.1*y*(1-y)','y(0)=0.5' 'x')    ans = 1/(1+exp(-1/10*x))
>> dsolve('Dy=0.1*y*(1-y)','y(0)=0.7' 'x')    ans = 1/(1+3/7*exp(-1/10*x))
```

Graphing the fours solutions:

```
>> ezplot('1/(1+9*exp(-1/10*x))',[0,100])     >> hold on
>> ezplot('1/(1+7/3*exp(-1/10*x))',[0,100]) >> ezplot('1/(1+exp(-1/10*x))',[0,100])
>> ezplot('1/(1+3/7*exp(-1/10*x))',[0,100]) >> hold off
>>axis([0,100,0,1])
```

3.2 Numerical Solutions

Many differential equations cannot be solved explicitly in terms of elementary functions. for those equations, we will need the numerical methods to get the approximate solution. The approximate solutions of the equations can be found by using the command **ode45**.

We can illustrate the command ode45 on the initial value problem y' = e^{-x^2} , y(0)=1. The command ode45 requires that the function on the right side of the equation is represented as a vector. So, we can start with the command: **>> f=inline(vectorize('exp(-x^2)'),'x','y');**

The command **ode45** plots the graph of the initial value problem on the specified interval. Suppose that we want to graph the solution on the interval [0, 2]. We use **>> ode45(f, [0 2], 1)** Note that 1 at the end of the command represents the *y* value at *x*=0 from the given initial condition y(0)=1.

We can obtain the numerical values of the solution as well by using >> **[x, y] = ode45(f, [0 2], 1).** The values between 0 and 2 at which **ode45** calculates the solution are stored in *x*, and the value of solution at these values is stored in *y*. To obtain the graph of these values use **>> plot(x, y).**

3.3 Second Order Equations

The second order linear equations can be solved similarly as the first order differential equations by using **dsolve** or **ode45**. For the command **dsolve**, recall that we represent the first derivative of the function *y* with **Dy**. The second derivative of *y* is represented with **D2y**. For example, the command for solving *y*''-3*y*'+2*y* = sin *x*.

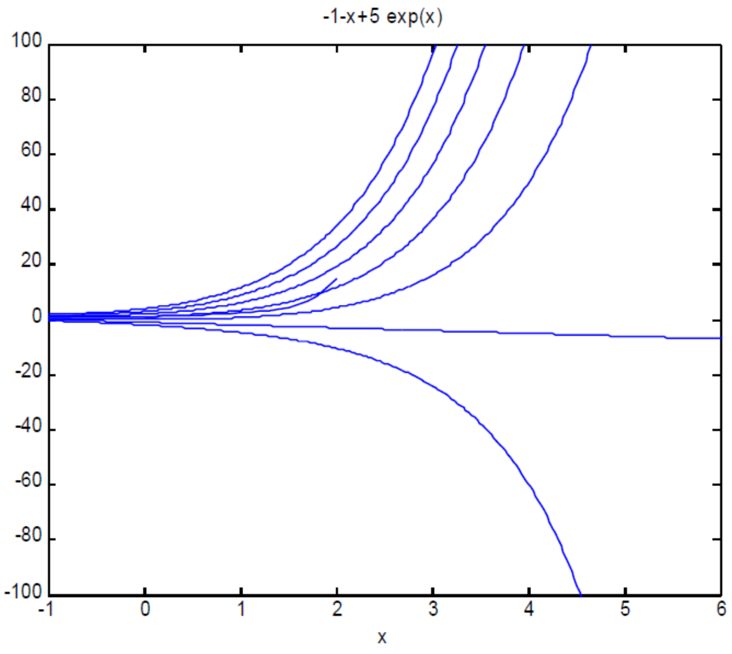
```
>> dsolve('D2y-3*Dy+2*y=sin(x)', 'x')
ans = 3/10*cos(x)+1/10*sin(x)+C1*exp(x)+C2*exp(2*x)
```

If we have the initial conditions y(0) = 10, y'(0)=-10, we would have:

```
>> dsolve('D2y-3*Dy+2*y=sin(x)', 'y(0)=1', 'Dy(0)=-1', 'x')
ans = 3/10*cos(x)+1/10*sin(x)+5/2*exp(x)-9/5*exp(2*x)
```

Practice problems 1

1. a) Find the general solution of the differential equation *y'-2y=6x*.
   b) Find the particular solution with initial condition y(0)=3.
   c) Plot the particular solution on interval [0,2] and find the value of this solution at 2.
2. Graph the solutions of the differential equation *y'* = *x*+*y* for the y-values of the initial condition y(0) taking integer values between -2 and 4.
3. Find the general solution of the equation *y*''-4 *y*'+4 *y*= e^x +*x²*. Then find the particular solution with the initial values *y*(0)=8, *y*'(0)=3.



Solutions.

1. a) General solution: **dsolve('Dy-2*y=6*x', 'x')**
   ans= (-6x-3)/2+C1*exp(2x)

b) Paticular solution: **dsolve('Dy-2*y=6*x', 'y(0)=3', 'x') ans= (-6x-3+9*exp(2x))/2**

c) **syms x ezplot((-6x-3+9*exp(2x))/2, [0 2])**

To find the value at 2: **f=inline((-6x-3+9*exp(2x))/2, x) f(2)**
ans=238.19

2. First, find the seven particular solutions of *y' = x+y* with initial conditions y(0)=-2,-1,0,1,2,3,4.
s1 = dsolve('Dy = x+y', 'y(0)=-2', 'x')
s2 = dsolve('Dy = x+y', 'y(0)=-1', 'x') ... etc .. **s7 = dsolve('Dy = x+y', 'y(0)=4', 'x')**
Then plot them on the same graph. **hold on ezplot(s1) ezplot(s2)** ...etc... **ezplot(s7) hold off**

3. **>> dsolve('D2y-4*Dy+4*y=exp(x)+x^2', 'x')**
ans = exp(x)+1/4*x^2+1/2*x+3/8+C1*exp(2*x)+C2*exp(2*x)*x
>> dsolve('D2y-4*Dy+4*y=exp(x)+x^2','y(0)=8', 'Dy(0)=3', 'x')
ans = exp(x)+1/4*x^2+1/2*x+3/8+53/8*exp(2*x)-47/4*exp(2*x)*x

4. M-files

4.1 Basics on M-files

Suppose that you want to perform the same operation many times for different input values. Matlab allows you to create a function or a script that you can execute repeatedly with different input values (i.e. to do programming). Such function or a script is called an M-file. An M-file is an ordinary text file containing Matlab commands. You can create them using any text editor that can save files as plain (ASCII) text. You can also create an M-file by using “File” menu and choosing “New M-file” option from your Matlab command window. An important fact to keep in mind when using M-files is that an M-file can be executed from the command window just if the "Current Directory" on the top of the command window is set to be **the same directory** where the M-file is saved.

Example 1. Suppose that you want to solve the equation $ax^2 + bx + c = 0$ using the quadratic formula. We can create a function which will have as input values of *a, b* and *c* and which will give the solution(s) of the quadratic equation. We can enter the following in an M-file.

```
function [x1 x2] = quadratic1(a, b, c)
x1 = (-b+sqrt(b^2-4*a*c))/(2*a);
x2 = (-b-sqrt(b^2-4*a*c))/(2*a);
```

To execute the M-file and find solutions of $x^2 -3x + 2 = 0$, you can use
>> [x1, x2] = quadratic(1, -3, 2)
The output is **x1 = 2** **x2 = 1**
Note that the semi-column symbol after the commands suppresses display of the results of these commands. The values of x1 and x2 will still be displayed since we listed these variables in the brackets of the function.
Alternatively, the following file can be used.

```
function [ ] = quadratic2(a, b, c)
x1 = (-b+sqrt(b^2-4*a*c))/(2*a)
x2 = (-b-sqrt(b^2-4*a*c))/(2*a)
```

Here we do not need to list the variables in the first line since we did not use the semi-column symbols when computing the values of x1 and x2. Also, you do not need to list **x1** and **x2** in the first line since they will be displayed when 2nd and 3rd lines are executed. Also, you can execute this M-file to solve $x^2 -3x + 2 = 0$ simply as **quadratic(1, -3, 2)** instead of **[x1, x2] = quadratic(1, -3, 2)** The output is the same as for the first version.
>> **quadratic(1, -3, 2)**
This means that you want to solve the equation. Matlab gives you the answers:

```
x1 =     2                              x2 =     1
```

These M-files produce the complex values if the solutions of a quadratic equation are complex numbers. For example, to solve $x^2 + 4 = 0$, we can use

```
>> quadratic(1, 0, 4)
x1 =       0 + 2.0000i            x2 =       0 - 2.0000i
```

Example 2. Suppose that we need to write a program that calculates the Cartesian coordinates *(x,y)* of a point given by polar coordinates *(r,θ)*.
One possible solution is the following M-file.

```
function [x, y] = polar(r, theta)
x = r*cos(theta);
y = r*sin(theta);
```

For example, to calculate *(x,y)* coordinates of for *r*=3 and $\theta = \pi$, we can use the command **[x,y]=polar(3, pi)** and get the answer

```
x = -3                    y =  3.6739e-016
```

Note that *y* is very close to 0. The inaccuracy comes from the fact that using **pi** for *π* in Matlab gives an approximation of π not the exact value. The exact representation of π is obtained by **sym('pi')**. The command **[x,y]=polar(3, sym('pi'))** gives you the expected answer

```
x = -3                    y = 0.
```

4.2 Branching and Loops

In most cases, the programs cannot be reduced to a simple application of a single formula. In those cases, we need to use branching or loops.

Example 3. Consider the following program that calculates the absolute value of a given number.

```
function y = absval(x)
if x >= 0
        y=x;
else
        y=-x;
end
```

For example, the command **y=absval(-5)** gives us the answer **y = 5**.

The previous example exhibits a program where a decision making is involved: in certain case one sequence of commands is performed, in certain other cases the other sequence of commands is performed. This is called **branching.** The simplest type of branching is with commands **if .. else** (followed by **end**) just like in the above example.

Another situation that frequently appears in programming is when a sequence of commands should be repeated several times.

Example 4. Find a sum of the series $\sum_{n=1}^{100} \frac{1}{n^2}$

In order to find this sum, it would be impractical to add 100 numbers by hand. The basic idea that could be used in order to create a more practical solution involves recursive reasoning: keep adding 1/n^2 term at each step to the sum computed at previous step. This give us the following informal idea for a program:

1. At the zero-th step, the sum is 0.

2. **For** all *n*'s between 1 and 100, compute the sum at n-th step by adding the term 1/n^2 to the sum of previous terms. If we denote the sum of n-1 terms by **s**, the sum of *n* terms will be **s+1/n^2**.

In Matlab code, this can be executed by using the command **for n = 1:100** meaning that the variable **n** is taking integer values between 1 and 100.

The Matlab code for this is: **s=0; for n=1:100 s = s+1/n^2; end**
To display the final value of **s**, simply type "**s**".

Example 5. Generalize the idea from the previous example to write a program that computes the sum of the first *n* terms of the sum f $\sum_{i=1}^{n} \frac{1}{i^2}$ or a given *n*.

The main difference is that the value of n has to be specified as input. The idea is the same:
1. At the zero-th step, the sum is 0.
2. **For** all *i*'s between 1 and *n*, if the sum of previous terms is **s**, the sum of *i* terms will be **s+1/i^2**.

Matlab code for this is:

```
function s = series(n)
s=0;
for i=1:n
        s = s+1/i^2;
end
```

For example, the command **s = series(20000)** produces **s =1.6449**.

Example 6. Calculate the (convergent) sum $\sum_{i=1}^{\infty} \frac{1}{i^2}$ to first few decimals.

In this case, the **for** loop is not convenient to use since we do not know how many terms we need to add in order to obtain the required accuracy. The informal idea is the following

1. The input of the program is a small number e representing the required accuracy.
2. **While** the difference *D* of the sum of *n+1* and the sum of *n* terms is greater than e, keep increasing the number of terms *n.*
3. D becomes smaller than the given small number, approximate the infinite sum with the sum of *n* terms.

Matlab code for this idea is:

```
function [s, n] = conv_series(e)
n=1;
p=0;           (zero-th term of sum)
s=p+1;         (sum of the zero-th and the first terms)
while abs(s-p)>e
   n=n+1;      (number of steps is increased by one)
   p=s;        (n-th term of the sum becomes what n+1st term was in previous step)
   s=p+1/n^2;          (n+1th term is calculated)
end
```

The following example is also a good practice for mastering the idea of recursion.

Example 7. Write a program computing the factoriel of a given positive integer.
In order to obtain the algorithm, it is helpful to think of factoriel function

n! = 1 times 2 times 3 times ... times n

as of a recursive sequence:

1. Factoriel(1) = 1.
2. Factoriel(n) = Factoriel(n-1) times n

This idea transfers to the following Matlab function that calculates the factoriel of a given positive integer.

```
function f = factoriel(n)
f=1;
for i=1:n
   f=f*i;
end
```

For example, the command **f = factoriel(6)** produces the output **f = 720**.

In MATLAB you can reference one program within another.

Example 8. Approximate the number *e* by the first *n* terms of the following series $e=\sum_{n=0}^{\infty}\frac{1}{n!}$

Since the formula for the n-th term of the series has factoriel in it, this program requires the use of the function **factoriel** from the previous example. This is one possible realization of this idea.

```
function e = natural(n)
e=1;
for i=1:n
        e=e+1/factoriel(i);
end
```

The command **e = natural(40)** produces the output **e =2.7183**.

Practice problems 2

1. The number π can be calculated using the following formula: $\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}$

   Write programs that:

   a) Approximates π with the first 1000 terms of the above sum.
   b) Approximates π with the first *n* terms of the above sum, for given *n*.
   c) Approximates π using the above sum to three decimals.

2. Consider the recursive sequence $a(n+1) = \sqrt{2 + a(n)}$ with an initial condition *a(0)=0*.

   Write a program that computer the n-th term for a given value of n. Executing the program for n=100, 1000 and 10,000 compute the limit of this sequence to first few decimals.
3. For a given n, calculate the n-th term of the sequence *a(n+1)= 1/(1+a(n))* with initial condition *a(1)=1*. Use that program to determine the limit of this sequence.
4. To approximate the integral $\int_a^b f(x)\,dx$, we can use the left sum with n subintervals. In this case, we are adding the areas of rectangles of height *f(x_i)* for *i=0,..., n-1* and of the width *h=(b-a)/n*.

   L(n) = *h* times *f(x_0)* + *h* times *f(x_1)* + ... + *h* times *f(x_n-1)*.

   To formalize this idea, it is again useful to think of the above formula for *L* in a recursive form

   1. *L(x_0)* = *h* times *f(x_0),*

2. *L(i+1) = L(i) + h* times *f(x_i)*.

Write a Matlab function that approximates a definite integral using left sum. When executing the program, you will need to inline the function f first. Test your program by evaluating the integral from 0 to 1 of x^2 using 1000 subintervals. You should obtain 0.33. Adapt the program to calculate the Right Sum.

5. Using the **if... else** command, write a function that calculates the solution of quadratic equation if real solutions exist and has the text "No real solution" for output otherwise. To display text, you can use the command **disp('...')** with text that you want displayed between the quotes.

Solutions

1. a) **p=1;   for n=1:1000   p=p+4*(-1)^n/(2*n+1);   end   p**

   b) **function p = series_sum(n)   p=1   for i=1:n   p= p+4*(-1)^i/(2*i+1); end**

   c) **function p=series_sum2(e)**

```
    n=1;    p=1;   (zero-th term of sum)   s=p-4/3;   (sum of the zero-th and the first terms)
    while abs(s-p)>e
        n=n+1;          (number of steps is increased by one)
        p=s;   (n-th term of the sum becomes what n+1st term was in previous step)
        s=p+4*(-1)^n/(2*n+1);        (n+1th term is calculated)
    end
```

2. **function a = recursive_sequence(n)   a=0   for i=1:n   a= sqrt(2+a); end**

   The limit of this sequence is 2.

3. **function a = recursive_sequence(n)   a=1   for i=1:n   a= 1/(1+a);   end**

   The limit of this sequence is 0.62.

4. **function L=left_sum(f,a,b,n)**

```
h=(b-a)/(n);
L=h*f(a);
for i=1:n-1
        L=L+h*f(a+h*i);
end
```

   To test the program, use **f=inline('x^2', 'x')** and **L=left_sum(f, 0, 1, 1000)**. Get L=0.333.

   For the right sum, you can use

```
function R=right_sum(f,a,b,n)
h=(b-a)/(n);
R=h*f(a+h);
for i=2:n
        R=R+h*f(a+h*i);
end
```

5. **function [ ] = quadratic(a, b, c)**

```
D=b^2-4*a*c;
if D<0  disp('No real solutions')
else    x1 = (-b+sqrt(D))/(2*a)        x2 = (-b-sqrt(D))/(2*a)
end
```