Chapter 1

Functions, scripts and good programming practice

The art of programming is the ability to translate from human notation to one which a computer understands. In the spirit of gradual increase of complexity, we always start with mathematical notation, which serves as the bridge between human language and computer (programming) language. Essentially, mathematical notation is the universal language from which we can always go to an arbitrary programming language, including MATLAB.

1.1 Motivational examples

Before we jump to functions and scripts, we will cover two examples: the first from personal finances and the second from physics.

Bank interest rate problem

Suppose someone desires to buy a car with price Mc two years from now and this person currently has a starting amount of money Ms. What interest rate is required to grow the starting sum to the required one?

As usual, our first job is to translate the problem to the language of mathematics and then convert it to a programming problem. Usually, the interest rate is given as a percentage (p) by which the account grows every year. Well, percents are really for accountants and everyone else uses fractions of one hundred, i.e. $r = p/100$, so we say that our initial investment grows by $1 + r$ every year. Thus, in two years it will grow as $(1 + r)^2$ and we finally can form the following equation

$$Ms \times (1 + r)^2 = Mc$$

Let's go back to using fractions

$$Ms \times (1 + p/100)^2 = Mc$$

Now we expand the equation

$$1 + 2\frac{p}{100} + \frac{p^2}{100^2} = \frac{Mc}{Ms}$$

With the following relabeling $p \to x$, $1/100^2 \to a$, $1/50 \to b$, and $(1 - Mc/Ms) \to c$, the canonical quadratic equation is easily seen

$$ax^2 + bx + c = 0 \tag{1.1}$$

For now we will postpone the solution of this equation and see one more problem where we need to solve such an equation as well.

Time of flight problem

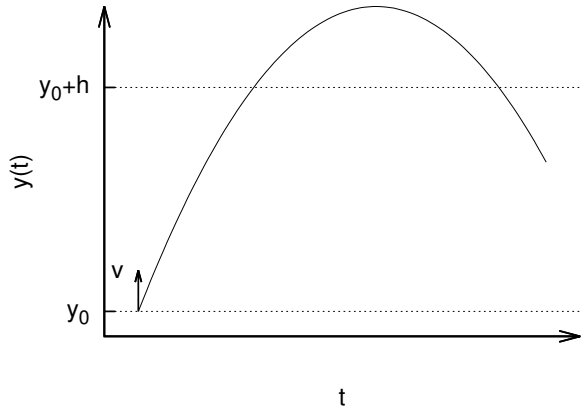

Figure 1.1: The firework rocket's height dependence on time.

A fireworks pyrotechnician would like to ignite a charge at the height (h) to maximize the visibility of the flare and to synchronize its position with other flares. The firework's shell leaves a gun with vertical velocity (v). We need to determine the delay time to which we must set the ignition timer, i.e. find how long it takes for the shell to reach the desired height (h) with respect to the firing position (see fig. 1.1). Again, first we need to translate the problem from the language of physics to mathematics. Assuming that the gun is not too powerful, we can treat the acceleration due to gravity (g) as a constant at all shell heights. We will also neglect air resistance. The shell's vertical position (y) versus flight time (t) is governed by the equation of motion with constant acceleration and expressed as $y(t) = y_0 + vt - gt^2/2$, where y_0 is the height of the shell at the firing time, i.e. $y(0) = y_0$. So we need to solve the following equation:

$$h = y(t) - y_0$$

Substituting known parameters we obtain

$$h = vt - gt^2/2 \tag{1.2}$$

Finally, we convert the above equation to the canonical quadratic eq. (1.1) with the following substitutions $t \to x$, $-g/2 \to a$, $v \to b$, and $-h \to c$.

1.2 Scripts

So far when we interacted with MATLAB, we typed our commands in the command window in sequence. However, if we need to execute repetitive sequences of commands, it is much more convenient and, more importantly, less error prone to create a separate file with such commands, i.e. the script file.

The name of the script file is arbitrary[1] but it should end with `.m` to mark this file as executable by MATLAB. The script file by itself is nothing but a simple text file which can be modified not only by MATLAB's built-in editor, but by any text editor as well.

Let's say we have a script file labeled `'script1.m'`. To execute its content we need to type its name without `.m` in the command prompt, i.e. `script1`. MATLAB will go over all commands listed in the script and execute it as if we were typing them one by one. An important consequence of this is that it will modify our workspace variables, as we will soon see.

Quadratic equation solver script

After above formal definitions, we address our problem of solving the quadratic equation

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

with the script (i.e. program).

Before we start programming, we still need to spend time in the realm of mathematics. The quadratic equation has in general two roots x_1 and x_2 which are given by the following equation:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The MATLAB compatible representation of the above is shown in the following listing, which we will save into the file `'quadSolvScrpt.m'`.

[1] Though a script filename is arbitrary, it is good idea to make the filename so it reflects the purpose of the script.

Listing 1.1: quadSolvScrpt.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/quadSolvScrpt.m)

```
% solve quadratic equation  a*x^2 + b*x + c = 0
x1 = ( - b - sqrt( b^2 - 4*a*c ) ) / (2*a)
x2 = ( - b + sqrt( b^2 - 4*a*c ) ) / (2*a)
```

Words of wisdom

Put many comments in your code, even if you are the only intended reader. Trust the author; in just about two weeks after completion of your script, you likely will not be able to recall the exact purpose of the code or why a particular programming decision was made.

Now let's define coefficients a, b, and c and run our script to find roots of the quadratic equation. We need to execute the following sequence

```
>> a = 2; b =-8; c = 6;
>> quadSolvScrpt
x1 =
    1
x2 =
    3
```

Notice that MATLAB creates variables `x1` and `x2` in the workspace and also displays their values as the output of the script. As usual, if we want to suppress the result of any MATLAB statement execution, we need to screen the statement by terminating it with `;`.

If we would like to find the solution of the quadratic equation with different coefficients, we just need to redefine them and run our script again

```
>> a = 1; b =3; c = 2;
>> quadSolvScrpt
x1 =
   -2
x2 =
   -1
```

Note that our old values of `x1` and `x2` are overwritten with new ones.

The ability of scripts to overwrite the workspace variables can be quite handy. For example, one might want to create a set of coefficients or universal constants for further use during a MATLAB session. See the sample script, which sets some universal constants, below

Listing 1.2: universal_constants.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/universal_constants.m)

```
% the following are in SI units
g = 9.8;  % acceleration due to gravity
c = 299792458; % speed of light
G = 6.67384e-11; % gravitational constant
```

We can use the above script to easily find, for example, what height the firework's shell (which we discussed in section 1.1) will reach at a given time $t = 10$ seconds, assuming that the initial vertical velocity is $v = 142$ m/s. Since `g` is defined in the above script, to calculate the height according to eq. (1.2), all we need to do is to execute the following:

```
>> universal_constants
>> t=10; v= 142;
>> h = v*t - (g*t^2)/2
h =
  930
```

1.3 Functions

In general, it is considered bad practice to use scripts, as their ability to modify workspace variables is actually a downside when working on a complex program. It would be nice to use code such that after execution, only the results are provided without affecting the workspace.

In MATLAB, this is done via *function*. A function is a file which contains the following structure

```
function [out1, out2, ..., outN] = function_name (arg1, arg2, ..., argN)
        % optional but strongly recommended function description
        set of expressions of the function body
end
```

Note that the file name must end with `'.m'` and the leading part of it must be the same as function name, i.e. `'function_name.m'`.

Words of wisdom

White space, consisting of tabulations and spaces in front of any line, is purely cosmetic but greatly improves the readability of a program.

A function might accept several input arguments or parameters. Their names can be arbitrary but it is a good idea to give them more meaningful names, i.e. not just `arg1`, `arg2`, ..., `argN`. For example, our quadratic solver could have much better names for input parameters than `a`, `b`, and `c`. Similarly, a function can return several parameters, but they must be listed in the square brackets `[]` after the `function` keyword. Their order is completely arbitrary,

like assignment names. Again, it is much better to use something like `x1` and `x2` instead of `out1` and `out2`. The only requirements is that the return parameters need to be assigned somewhere in the body of the function, but do not worry — MATLAB will prompt you if you forget about this.

Words of wisdom

By the way, many built-in MATLAB functions are implemented as `'.m'` files following the above conventions. You can learn good techniques and trickery of MATLAB by reading these files.

Quadratic equation solver function

Usually, when I develop a function, I start with a script, debug it and then wrap it with the `function` related statements. We will modify our quadratic equation solver script to become a valid MATLAB function.

Listing 1.3: `quadSolverSimple.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/quadSolverSimple.m)

```
function [x1, x2] =  quadSolverSimple( a, b, c )
% solve quadratic equation  a*x^2 + b*x + c = 0
x1 = ( - b - sqrt( b^2 - 4*a*c ) ) / (2*a);
x2 = ( - b + sqrt( b^2 - 4*a*c ) ) / (2*a);
end
```

Note that we need to save it to `'quadSolverSimple.m'`. Now let's see how to use our function.

```
>> a = 1; b =3; c = 2;
>> [x1, x2] = quadSolverSimple(a,b,c)
x1 =
    -2
x2 =
    -1
```

Now we highlight some very important features of MATLAB functions.

```
>> x1=56;
>> clear x2
>> a = 1; b =3; cc = 2; c = 200;
>> [xx1, xx2] = quadSolverSimple(a,b,cc)
xx1 =
    -2
```

```
xx2 =
    -1
>> x1
x1 =
    56
>> x2
Undefined function or variable 'x2'.
```

Note that at the very first line we assign `x1` to be 56 and then at the very end we check the value of the `x1` variable which is still 56. As our function listing shows, MATLAB assigns the `x1` variable as one of the roots of the quadratic equation inside of the function body. However, this assignment does not affect the `x1` variable in the workspace. The second thing to note is that the function assigned `xx1` and `xx2` variables in our workspace; this is because we asked to assign these variables. This is another feature of MATLAB functions — they assign variables in the workspace in the same order and to variables which the user asked during the function call. Notice also that since we clear `x2` before the function call, there is still no assigned `x2` variable after the execution of the function, though the `quadSolverSimple` function uses it internally for its own needs. Finally, notice that the function clearly did not use assigned value of `c = 200`. Instead it used the value of `cc = 2` which we provided as the third argument to the executed function. So we note that the names of parameters during the function call are irrelevant and only their position is important.

We can simply remember this: *what happens in the function stays in the function*, what comes out of the function goes into the return variable placeholders.

1.4 Good programming practice

If this is your first time studying programming, then you can skip this section. Come back to it later, once you are fluent with the basics of functions and scripts.

Here we discuss how to write programs and functions in a robust and manageable way based on our simple quadratic equation solver function.

Simplify the code

Let's have a look at the original `quadSolverSimple` function listing 1.3. At first glance, everything looks just fine, but there is a part of the code which repeats twice, i.e. calculation of `b^2 - 4*a*c`. MATLAB wastes cycles to calculate it second time, but, more importantly, the expression is the definition of the discriminant of our equation which can be very useful (we will see it very soon). Additionally, if we find a typo in our code it is very likely that

we will forget to fix it at the second occurrence of such code, especially if it is separated by more than a few lines. So we transform our function to the following

Listing 1.4: quadSolverImproved.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/quadSolverImproved.m)

```
function [x1, x2] =  quadSolverImproved( a, b, c )
% solve quadratic equation  a*x^2 + b*x + c = 0
D = b^2 - 4*a*c;
x1 = ( - b - sqrt( D ) ) / (2*a);
x2 = ( - b + sqrt( D ) ) / (2*a);
end
```

Try to foresee unexpected behavior

This looks much better, but what if our discriminant is negative? Then we cannot extract the square root and the function will fail (technically we can do it, but this involves manipulation with complex numbers and we pretend that this is illegal). Therefore, we need to check if the discriminant is positive and produce a meaningful error message otherwise. For the latter, we will use MATLAB's `error` function, which stops program execution and produces a user defined message in the command window.

Listing 1.5: quadSolverImproved2nd.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/quadSolverImproved2nd.m)

```
function [x1, x2] =  quadSolverImproved2nd( a, b, c )
% solve quadratic equation  a*x^2 + b*x + c = 0
D = b^2 - 4*a*c;
if ( D < 0 )
    error('Discriminant is negative: cannot find real roots');
end
x1 = ( - b - sqrt( D ) ) / (2*a);
x2 = ( - b + sqrt( D ) ) / (2*a);
end
```

Run test cases

At this point our quadratic solver looks quite polished and nothing can possibly go wrong with it. Right? As soon as you come to this conclusion a loud bell should ring inside of your

mind: I am getting too comfortable; thus, it is time to check. *Never release or use a code which you did not check*[2].

Equipped with this motto, we will start with testing our own code; this is commonly called "running test cases." Ideally, testing should cover all possible input parameters, although this is clearly impossible. We should check that our program produces the correct answer in at least one case, and we need to verify this answer in a somewhat independent way. Yes, often it means the use of paper and pencil. We also need to poke our program with somewhat random input parameters to see how robust it is.

First, we double check the correctness of the function. We will use simple enough numbers so we can do it in our head. Actually, we already did it in previous test runs during this chapter, but you can never have enough tests. So we will do it one more time.

```
>> a = 1; b =−3; c = −4;
>> [x1,x2] = quadSolverImproved2nd(a, b, c)
x1 =
    −1
x2 =
     4
```

It is easy to see that $(x-4)*(x+1) = x^2 - 3x - 4 = 0$, i.e. produced roots, indeed, satisfy the equation with the same $a = 1$, $b = -3$, and $c = -4$ coefficients. By the way, *do not use the same code for correctness verification, use some independent method.*

Now, we check the case when the discriminant is negative:

```
>> a = 1; b = 3; c = 4;
>> [x1,x2] = quadSolverImproved2nd(a, b, c)
Error using quadSolverImproved2nd (line 5)
Discriminant is negative: cannot find real roots
```

[2] It is true for your programs and especially for the code which you received from others, even if this source is very trustworthy and reputable. You might think that big software companies have tons of experience and produce "bulletproof" quality code. Their experience does not guarantee error-free code. *Absolutely all* software packages which this author has seen in his life come with a clause similar to this "THERE IS NO WARRANTY FOR THE PROGRAM. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION" (excerpt from GPL license). It comes in capital letters so we should take it seriously. Also, recall the MATLAB license agreement which you agreed to at the beginning of MATLAB usage. MATLAB expresses it in a bit less straight forward way, but the meaning is the same "any and all Programs, Documentation, and Software Maintenance Services are delivered "as is" and MathWorks makes and the Licensee receives no additional express or implied warranties."

This seems like a very long paragraph with more emphasized phrases than in the rest of this book; it also seems to be irrelevant to the art of programming. But this author has spent a lot of his time pulling hair from his head trying to see mistakes in his code and finding the problem rooted in someone else's code (I do not imply that I produce error free code). So *trust but verify* everyone.

Excellent! As expected, the program terminates with the proper error message.

Check and sanitize input arguments

One more test

```
>> a = 0; b = 4; c = 4;
>> [x1,x2] = quadSolverImproved2nd(a, b, c)
x1 =
  -Inf
x2 =
   NaN
```

Wow! There is no way that equation with $a = 0$ (simplified to $bx + c = 4x + 4 = 0$) has one root equal to infinity and the other root being NaN which stands for "not a number." We do not need a calculator to see that the root of this equation is -1. What is going on?

Let's closely examine our code in listing 1.5; the problem part is division by $2a$, which is actually 0 in this case. This operation is undefined but, unfortunately, MATLAB is trying to be smart and not produce an error message. Sometimes it is welcome behavior, but now it is not. So we need to be in control: intercept the case of $a = 0$ and handle it separately producing solutions $x1 = x2 = -c/b$. It is easy to see that we need to handle the case $a = b = 0$ as well. So our final quadratic solver function will be

Listing 1.6: quadSolverFinal.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/quadSolverFinal.m)

```
function [x1, x2] =  quadSolverFinal( a, b, c )
% solve quadratic equation  a*x^2 + b*x + c = 0

% ALWAYS check and sanitize input parameters
if ( (a == 0) & (b == 0) )
    error('a==0 and b==0: impossible to find roots');
end

if ( (a == 0) & (b ~= 0) )
    % special case: we essentially solve b*x = -c
    x1 = -c/b;
    x2=x1;
else
    D = b^2 - 4*a*c; % Discriminant of the equation
    if ( D < 0 )
        error('Discriminant is negative: no real roots');
    end
```

```
    x1 = ( - b - sqrt( D ) ) / (2*a);
    x2 = ( - b + sqrt( D ) ) / (2*a);
end
end
```

Is the solution realistic?

This was a lot of work to make a quite simple function perform to specifications. Now let's use our code to solve our motivational examples.

We start with the interest rate problem described in section 1.1. Suppose that we initially had $\$10,000$ (`Ms = 10000`) and the desired final account value is $\$20,000$ (`Mc = 20000`), thus

```
>> a = 1/100^2; b = 1/50; c = 1 - 20000/10000;
>> [p1,p2] = quadSolverFinal(a, b, c)
p1 =
 -241.4214
p2 =
   41.4214
```

At first glance everything is fine, since we obtain two solutions for required interest rate -241.4% and 41.4%. But if we look more closely, we realize that negative percentage means that we owe to the bank after each year, so the account value will decrease every year — which is opposite to our desire to grow money.

What is the reason for such an "unphysical" solution? The real meaning of the problem was lost in translation to mathematical form. Once we have the $(1+r)^2$ term, the computer does not care if the squared number is negative or positive, since it produces the valid equation root. But we humans do care!

We have just learned one more important lesson: *it is up to the human to decide if the solution is valid*. Never blindly trust a solution produced by a computer! They do not care about the reality or the "physics" of the problem.

Summary of good programming practice

- Foresee problem spots
- Sanitize and check input arguments
- Put a lot of comments in your code
- **Run test cases**

- Check the meaning of the solutions and exclude unrealistic ones
- Fix problem spots and repeat from the top

1.5 Recursive and anonymous functions

Before we move on, we need to consider a couple of special function use cases.

Recursive functions

Functions can call other functions (this is not a big surprise, otherwise they would be quite useless) and they can call themselves, which is called "recursion." If we go into detail, there is a limit of how many times a function can call itself. This is due to the limited memory size of a computer, since every function call requires the computer to reserve some amount of memory space to recall it later.

Let's revisit the account growth problem which we discussed in section 1.1. Now we would like to find the account value including interest after a certain number of years. The account value (Av) after N years is equal to the account value in the previous year ($N-1$) multiplied by the growth coefficient ($1+p/100$). Assuming that we initially invested an amount of money equal to Ms, we can calculate the final account value according to

$$Av(N) = \begin{cases} Ms & \text{if } N = 0 \\ (1 + p/100) \times Av(N-1) & \text{if } N > 0 \end{cases} \tag{1.3}$$

The above equation resembles a typical recursive function where the function calls itself in order to calculate the final value. The MATLAB implementation of such function is shown below

Listing 1.7: accountValue.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/code/accountValue.m)

```
function Av = accountValue( Ms, p, N)
% calculates grows of the initial account value (Ms)
% in the given amount of years (N)
% for the bank interest percentage (p)

% We sanitize input to ensure that stop condition is possible
    if ( N < 0 )
        error('Provide positive and integer N value');
    end
    if ( N ~= floor ( N ) )
        error ('N is not an integer number');
```

```
    end

% Do we meet stop condition?
    if ( N == 0 )
        Av = Ms;
        return;
    end

    Av = (1+p/100)*accountValue( Ms, p, N-1 );
end
```

Let's see how the initial sum $Ms = \$535$ grows in 10 years, if the account growth percentage is 5.

```
>> Ms=535; p=5; N=10; accountValue(Ms, p, N)
ans =
  871.4586
```

1.5.1 Anonymous functions

Anonymous functions look very confusing at first, but they are useful in cases when one function should call another or if you need to have a short-term-use function, which is simple enough to fit in one line.

It is easier to start with an example. Suppose for some calculations you need the following function $g(x) = x^2 + 45$. It is clearly very simple and would probably be used during only one session, and thus there is no point to create a full-blown `.m` file for this function. So we define an anonymous function

```
>> g = @(x) x^2 + 45;

>> g(0)
ans =
    45
>> g(2)
ans =
    49
```

The definition of the anonymous function g happens in the first line of the above listing; the rest is just a few examples to prove that it is working properly. The `@` symbol indicates that we are defining a *handle*[3] to the function of one variable `x` as indicated by `@(x)` and the rest

[3]Handle is a special variable type. It gives MATLAB a way to store and address a function.

is just this function body which must be just one MATLAB statement resulting in a *single* output.

An anonymous function can be a function of many variables, as shown below

```
>> h = @(x,y) x^2 − y + cos(y);
>> h(1, pi/2)
ans =
   −0.5708
```

Anonymous functions are probably the most useful when you want to define a function which uses some other function with some of the input parameters as a constant. For example, we can "slice" the above $h(x, y)$ along the 'x' dimension for fixed $y = 0$, i.e. define $h1(x) = h(x, 0)$. This is demonstrated below

```
>> h1 = @(x) h(x,0);
>> h1(1)
ans =
     2
```

Another useful property of anonymous functions is their ability to use variables defined in the workspace at the time of definition

```
>> offset = 10; s = @(x) x + offset;
>> clear offset
>> s(1)
ans =
    11
```

Note that in the above transcript the `offset` variable was cleared at the time of the `s` function's execution, yet it still works since MATLAB already used the value of the variable when we defined the function.

We can also evaluate

$$\int_0^{10} s(x)dx$$

with the help of MATLAB's built-in function `integral`

```
>> integral(s,0,10)
ans =
   150
```

Words of wisdom

Avoid using scripts. Instead convert them into functions. This is much safer in the long run since you can execute a function without worrying that it would affect or change something in your workspace in an unpredictable way.

1.6 Self-Study

Problem 1: Write a script which calculates

$$1 + \sum_{i=1}^{N} \frac{1}{x^i}$$

for $N = 10$ and $x = 0.1$

Use loops as much as you wish from now.

Problem 2: Write a script which calculates for $N = 100$.

$$S_N = \sum_{k=1}^{N} a_k$$

where $a_k = 1/k^{2k}$ for odd k and $a_k = 1/k^{3k}$ for even k.

Hint: you may find `mod` function useful to check for even and odd numbers.

Problem 3: Write a function `mycos` which calculates a value of a $\cos(x)$ at the given point x via the Taylor series up to N members. Define your function as
`function cosValue = mycos(x, N)`

Does it handle well the situation with large x values? Take $x = 10\pi$ for example. How far do you need to expand Taylor series to get absolute precision of 10^{-4}, what value of N do you find reasonable (no need to state it beyond one significant digit), and why so?

Problem 4: Download the data file `'hwOhmLaw.dat'`[4]. It represents result of someone's attempt to find the resistance of a sample via measuring voltage drop (V), which is the data in the 1st column, and current (I), which is the data in the 2nd column, passing through the resistor in the same conditions. Judging by the number of samples it was an automated measurement.

- Using Ohms law $R = V/I$ find the resistance (R) of this sample (no need to print it out for each point at this step)

[4]The file is available at http://physics.wm.edu/programming_with_MATLAB_book/ch_functions_and_scripts/data/hwOhmLaw.dat

- Estimate the resistance of the sample (i.e. find the average resistance) and estimate the error bars of this estimate (i.e. find the standard deviation).

For standard deviation use the following definition

$$\sigma(x) = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where x is the set (vector) of data points, $\bar{x}$ its average (mean), and N is the number of the points in the set.

Do not use standard built-in `mean` and `std` functions in your solution. You need to make your own code to do it. But feel free to test against these MATLAB functions.

Note: for help, read MATLAB's `std`; you might want to know about it.

Problem 5: Imagine you are working with an old computer which does not have the built-in multiplication operation. Program the `mult(x,y)` function which returns equivalent of `x*y` for two integer numbers `x` and `y` (either one can be negative, positive, or zero). Do not use the $*$ operator of MATLAB. You can use loops, conditions, `+`, or $-$ operators. Define your function as

```
function product=mult(x,y)
```