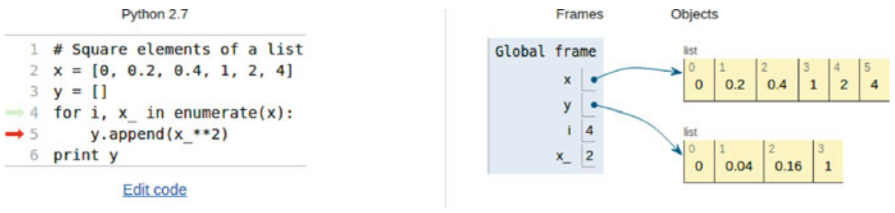
2 Basic Constructions



```
# Square elements of a list
x = [0, 0.2, 0.4, 1, 2, 4]
y = []
for i, x_ in enumerate(x):
    y.append(x_**2)
print y
```

2.1 If Tests

Very often in life, and in computer programs, the next action depends on the outcome of a question starting with "if". This gives the possibility to branch into different types of action depending on some criterion. Let us as usual focus on a specific example, which is the core of so-called random walk algorithms used in a wide range of branches in science and engineering, including materials manufacturing and brain research. The action is to move randomly to the north (N), east (E), south (S), or west (W) with the same probability. How can we implement such an action in life and in a computer program?

We need to randomly draw one out of four numbers to select the direction in which to move. A deck of cards can be used in practice for this purpose. Let the four suits correspond to the four directions: clubs to N, diamonds to E, hearts to S, and spades to W, for instance. We draw a card, perform the corresponding move, and repeat the process a large number of times. The resulting path is a typical realization of the path of a diffusing molecule.

In a computer program, we need to draw a random number, and depending on the number, update the coordinates of the point to be moved. There are many ways to draw random numbers and translate them into (e.g.) four random directions, but the technical details usually depend on the programming language. Our technique here is universal: we draw a random number in the interval $[0, 1)$ and let $[0, 0.25)$ correspond to N, $[0.25, 0.5)$ to E, $[0.5, 0.75)$ to S, and $[0.75, 1)$ to W. Let `x` and `y` hold the coordinates of a point and let `d` be the length of the move. A pseudo code (i.e., not "real" code, just a "sketch of the logic") then goes like

S. Linge, H.P. Langtangen, *Programming for Computations – MATLAB/Octave*,
Texts in Computational Science and Engineering 14, DOI 10.1007/978-3-319-32452-4_2

```
r = random number in [0,1)
if 0 <= r < 0.25
   move north: y = y + d
else if 0.25 <= r < 0.5
   move east:  x = x + d
else if 0.5 <= r < 0.75
   move south:  y = y - d
else if 0.75 <= r < 1
   move west:  x = x - d
```

Note the need for first asking about the value of `r` and then performing an action. If the answer to the “if” question is positive (true), we are done and can skip the next `else if` questions. If the answer is negative (false), we proceed with the next question. The last test if $0.75 \leq r < 1$ could also read just `else`, since we here cover all the remaining possible `r` values.

The exact code in Matlab reads

```
r = rand()          % random number in [0,1)
if 0 <= r < 0.25
   % move north
   y = y + d;
elseif 0.25 <= r < 0.5
   % move east
   x = x + d;
elseif 0.5 <= r < 0.75
   % move south
   y = y - d;
else
   % move west
   x = x - d;
end
```

We use `else` in the last test to cover the different types of syntax that is allowed. Matlab recognizes the reserved words `if`, `elseif`, and `else` and expects the code to be compatible with the rules of if tests:

- The test reads `if condition`, `elseif condition`, or `else`, where `condition` is a *boolean expression* that evaluates to `true` (1) or `false` (0).
- If `condition` is `true`, the following statements up to the next `elseif`, `else`, or `end` are executed, and the remaining `elseif` or `else` branches are skipped.
- If `condition` is `false`, the program flow jumps to the next `elseif` or `else` branch.

The blocks after `if`, `elseif`, or `else` may contain new if tests, if desired.

Working with if tests requires mastering boolean expressions. Here are some basic boolean expressions involving the *logical operators* ==, =, <, <=, >, and >=. Given the assignment to `temp`, you should go through each boolean expression below and determine if it is true or false.

```
temp = 21        % assign value to a variable
temp == 20  % temp equal to 20
temp ~= 20  % temp not equal to 20
temp <  20  % temp less than 20
temp >  20  % temp greater than 20
temp <= 20  % temp less than or equal to 20
temp >= 20  % temp greater than or equal to 20
```

2.2 Functions

Functions are widely used in programming and is a concept that needs to be mastered. In the simplest case, a function in a program is much like a mathematical function: some input number x is transformed to some output number. One example is the $\tanh^{-1}(x)$ function, called `atan` in computer code: it takes one real number as input and returns another number. Functions in Matlab are more general and can take a series of variables as input and return one or more variables, or simply nothing. The purpose of functions is two-fold:

1. to *group statements* into separate units of code lines that naturally belong together (a strategy which may dramatically ease the problem solving process), and/or
2. to *parameterize* a set of statements such that they can be written only once and easily be re-executed with variations.

Examples will be given to illustrate how functions can be written in various contexts.

If we modify the program `ball.m` from Sect. 1.2 slightly, and include a function, we could let this be a new program `ball_function.m` as

```
function ball_function()
    % This is the main program
    time = 0.6;                     % Just pick some time
    vertical_position = y(time);
    fprintf('%f \n',vertical_position)
    time = 0.9;                     % Pick another time
    vertical_position = y(time);
    fprintf('%f \n',vertical_position)
end

% The function 'y' is a _local_ function in this file
function result = y(t)
    g = 9.81;       % Acceleration of gravity
    v0 = 5;         % Initial velocity
    result = v0*t - 0.5*g*t^2;
end
```

Here, Matlab interprets this as the definition of two functions, recognized by the reserved word `function` that appears two places. The first function `ball_function`, is defined by the statements between (and including) `function` ball_function() and the first end. Note that the first function in a file should have the same name as the name of the file (apart from the extension .m). The second function, i.e. y, is similarly defined between function result = y(t) and the second end.

Opposed to the function y, the function ball_function does *not* return a value. This is stated in the first line of each function definition. Comparing, you notice that y has an assignment there, whereas ball_function has not. The final statement of the function y, i.e.

```
result = v0*t - 0.5*g*t^2;
```

will be understood by Matlab as "first compute the expression, then place the result in result and send it back (i.e. return) to where the function was called from". The function depends on one variable (or we say that it takes one *argument* or *input parameter*), the value of which must be provided when the function is called.

What do these things mean? Well, the function definition itself, e.g. of y, just tells Matlab that there is a function y, taking the specified arguments as input, and returning a specified output result. Matlab keeps this information ready for use in case a call to y is performed elsewhere in the code. In our case, a call to y happens twice by the line vertical_position = y(time). By this instruction, Matlab takes y(time) as a call to the function y, assigning the value of time to the variable t. So in the first call, t becomes 0.6, while in the second call t becomes 0.9. In both cases the code lines of y are executed and the returned result (the *output parameter*) is stored in vertical_position, before it is next printed by Matlab.

Note that the reserved word return may be used to enforce a return from a function before it reaches the end. For example, if a function contains if-elseif-else constructions, return may be done from within any of the branches. This may be illustrated by the following function containing three return statements:

```
function result = check_sign(x)
    if x > 0
        result = 'x is positive';
        return;
    elseif x < 0
        result = 'x is negative';
        return;
    else
        result = 'x is zero';
        return;
    end
end
```

Remember that only one of the branches is executed for a single call on check_sign, so depending on the number x, the return may take place from any of the three return alternatives.

One phrase you will meet often when dealing with programming, is *main program* or *main function*, or that some code is *in main*. This is nothing particular to Matlab, and simply means the first function that is defined in a file, e.g.

`ball_function` above. You may define as many functions as you like in a file after the main function. These then become *local functions*, i.e. they are only known inside that file. In particular, only the main function may be called from the command window, whereas local functions may not.

A function may take no arguments, or many, in which case they are just listed within the parentheses (following the function name) and separated by a comma. Let us illustrate. Take a slight variation of the ball example and assume that the ball is not thrown straight up, but at an angle, so that two coordinates are needed to specify its position at any time. According to Newton's laws (when air resistance is negligible), the vertical position is given by $y(t) = v_{0y}t - 0.5gt^2$ and the horizontal position by $x(t) = v_{0x}t$. We can include both these expressions in a new version of our program that prints the position of the ball for chosen times. Assume we want to evaluate these expressions at two points in time, $t = 0.6\,\mathrm{s}$ and $t = 0.9\,\mathrm{s}$. We can pick some numbers for the initial velocity components `v0y` and `v0x`, name the program `ball_position_xy.m`, and write it for example as

```
function ball_position_xy()
    initial_velocity_x = 2.0;
    initial_velocity_y = 5.0;

    time = 0.6;  % Just pick one point in time
    x_pos = x(initial_velocity_x, time);
    y_pos = y(initial_velocity_y, time);
    fprintf('%f %f \n', x_pos, y_pos)

    time = 0.9;  % Pick another point in time
    x_pos = x(initial_velocity_x, time);
    y_pos = y(initial_velocity_y, time);
    fprintf('%f %f \n', x_pos, y_pos)
end

function result = y(v0y, t)
    g = 9.81;      % Acceleration of gravity
    result = v0y*t - 0.5*g*t^2;
end

function result = x(v0x, t)
    result = v0x*t;
end
```

Now we compute and print the two components for the position, for each of the two chosen points in time. Notice how each of the two functions now takes *two* arguments. Running the program gives the output

```
1.2  1.2342
1.8  0.52695
```

A function may also return more than one value. For example, the two functions we just defined could alternatively have been defined into one as

```
function [result1, result2] = xy(v0x, v0y, t)
    g = 9.81;        % acceleration of gravity
    result1 = v0x*t;
    result2 = v0y*t - 0.5*g*t^2;
end
```

Notice the two return values `result1` and `result2` that are listed in the function header, i.e., the first line of the function definition. When calling the function, arguments must appear in the same order as in the function definition. We would then write

```
[x_pos,y_pos] = xy(initial_x_velocity, initial_y_velocity, time);
```

The variables `x_pos` and `y_pos` could then have been printed or used in other ways in the code.

There are possibilities for having a variable number of function input and output parameters (using `nargin` and `nargout`). However, we do not go further into that topic here.

Variables that are defined inside a function, e.g., `g` in the last `xy` function, are *local variables*. This means they are only known inside the function. Therefore, if you had accidentally used `g` in some calculation outside the function, you would have got an error message. By use of the reserved word `global`, a variable may be known also outside the function in which it is defined (without transferring it as a parameter). For example, if, in some function `A`, we write

```
global some_variable;
some_variable = 2;
```

then, in another function `B`, we could use `some_variable` directly if we just specify it first as being global, e.g.

```
global some_variable;
some_other_variable = some_variable*2;
```

We could even change the value of `some_variable` itself inside `B` if we like, so that upon return to the function `A`, `some_variable` would have a new value. If you define one global and one local variable, both with the same name, the function only sees the local one, so the global variable is not affected by what happens with its local companion of the same name. The arguments named in the header of a function definition are by rule local variables inside the function. One should strive to define variables mostly where they are needed and not everywhere.

In any programming language, it is a good habit to include a little explanation of what the function is doing, unless what is done by the function is obvious, e.g., when having only a few simple code lines. This explanation (sometimes known as a *doc string*) should be placed just at the top of the function. This explanation is meant for a human who wants to understand the code, so it should say something about the purpose of the code and possibly explain the arguments and return values if needed. If we do that with our `xy` function from above, we may write the first lines of the function as

```
function xy(v0x, v0y, t)
    % Compute the x and y position of the ball at time t
```

Note that a function you have written may call another function you have written, even if they are not defined within the same file. Such a call requires the called function to be located in a file with the same name as the function (apart from the extension `.m`). This file must also be located in a folder where Matlab can find it, e.g. in the same folder as the calling function.

Functions are straightforwardly passed as arguments to other functions, as illustrated by the following script `function_as_argument.m`:

```
function function_as_argument()
    x = 2;
    y = 3;

    % Create handles to the functions defined below
    sum_xy_handle = @sum_xy;
    prod_xy_handle = @prod_xy;

    sum = treat_xy(sum_xy_handle, x, y);
    fprintf('%f \n', sum);
    prod = treat_xy(prod_xy_handle, x, y);
    fprintf('%f \n', prod);
end

function result = treat_xy(f, x, y)
    result = f(x, y);
end

function result = sum_xy(x, y)
    result = x + y;
end

function result = prod_xy(x, y)
    result = x*y;
end
```

When run, this program first prints the sum of `x` and `y` (i.e., 5), and then it prints the product (i.e., 6). We see that `treat_xy` takes a function name as its first parameter. Inside `treat_xy`, that function is used to actually *call* the function that was given as input parameter. Therefore, as shown, we may call `treat_xy` with either `sum_xy` or `prod_xy`, depending on whether we want the sum or product of `x` and `y` to be calculated.

To transfer a function to the function `treat_xy`, we must use *function handles*, one for each function we want to transfer. This is done by the sign `@` combined with the function name, as illustrated by the lines

```
sum_xy_handle = @sum_xy;
prod_xy_handle = @prod_xy;
```

Note that it is the *handle* that is used in the function call, as, e.g., in

```
sum = treat_xy(sum_xy_handle,x,y);
```

Functions may also be defined *within* other functions. It that case, they become *local functions*, or *nested functions*, known only to the function inside which they are defined. Functions defined in main are referred to as *global functions*. A nested function has full access to all variables in the *parent function*, i.e. the function within which it is defined.

One convenient way of defining one-line functions (they can *not* be more than one line!), is by use of *anonymous functions*. You may then define, e.g., a square function by the name `my_square`, as

```
my_square = @(x) x^2;
```

and then use it simply as

```
y = my_sqare(2);
```

which would have assigned the value 4 to `y`. Note that `my_square` here becomes a handle that may be used directly as a function parameter for example.

Overhead of function calls

Function calls have the downside of slowing down program execution. Usually, it is a good thing to split a program into functions, but in very computing intensive parts, e.g., inside long loops, one must balance the convenience of calling a function and the computational efficiency of avoiding function calls. It is a good rule to develop a program using plenty of functions and then in a later optimization stage, when everything computes correctly, remove function calls that are quantified to slow down the code.

2.3 For Loops

Many computations are repetitive by nature and programming languages have certain *loop structures* to deal with this. Here we will present what is referred to as a *for loop* (another kind of loop is a *while* loop, to be presented afterwards). Assume you want to calculate the square of each integer from 3 to 7. This could be done with the following program.

```
for i = 3:7
    i^2
end
```

What happens when Matlab interprets your code here? First of all, the word `for` is a reserved word signalling to Matlab that a `for` loop is wanted. Matlab then sticks to the rules covering such constructions and understands that, in the present example, the loop should run 5 successive times (i.e., 5 *iterations* should be done), letting the variable i take on the numbers 3, 4, 5, 6, 7 in turn. During each iteration, the statement inside the loop (i.e. i^2) is carried out. After each iteration, i is automatically (behind the scene) updated. When the last number is reached, the last iteration is performed and the loop is finished. When executed, the program will therefore print out 9, 16, 25, 36 and 49. The variable i is often referred to as a *loop index*, and its name (here i) is a choice of the programmer.

Note that, had there been several statements within the loop, they would all be executed with the same value of i (before i changed in the next iteration). Make sure you understand how program execution flows here, it is important.

The specification of the values desired for the loop variable (here 3:7) is more generally given as start:step:stop, meaning that the loop variable should take on the integers from start to stop, inclusive at both ends, in steps of step. If step is skipped, the default value is 1, as in the example above. Note that decreasing integers may be produced by letting start > stop combined with a negative step. This makes it easy to, e.g., traverse arrays in either direction.

Let us modify ball_plot.m from Sect. 1.4 to illustrate how useful for loops are if you need to traverse arrays. In that example we computed the height of the ball at every milli-second during the first second of its (vertical) flight and plotted the height versus time.

Assume we want to find the maximum height during that time, how can we do it with a computer program? One alternative may be to compute all the thousand heights, store them in an array, and then run through the array to pick out the maximum. The program, named ball_max_height.m, may look as follows.

```
g = 9.81;
v0 = 5;
t = linspace(0, 1, 1000);
y = v0*t - 0.5*g*t.^2;

% At this point, the array y with all the heights is ready.
% Now we need to find the largest value within y.

largest_height = y(1);    % Preliminary value
for i = 2:1000
   if y(i) > largest_height
       largest_height = y(i);
   end
end

fprintf('The largest height achieved was %f m \n',largest_height);

% We might also like to plot the path again just to compare
plot(t,y);
xlabel('Time (s)');
ylabel('Height (m)')
```

There is nothing new here, except the for loop construction, so let us look at it in more detail. As explained above, Matlab understands that a for loop is desired when it sees the word for. The value in y(1) is used as the *preliminary* largest height, so that, e.g., the very first check that is made is testing whether `y(2)` is larger than this height. If so, `y(2)` is stored as the largest height. The `for` loop then updates `i` to 2, and continues to check `y(3)`, and so on. Each time we find a larger number, we store it. When finished, `largest_height` will contain the largest number from the array `y`. When you run the program, you get

```
The largest height achieved was 1.274210 m
```

which compares favorably to the plot that pops up.

To implement the traversing of arrays with loops and indices, is sometimes challenging to get right. You need to understand the start, stop and step length choices for an index, and also how the index should enter expressions inside the loop. At the same time, however, it is something that programmers do often, so it is important to develop the right skills on these matters.

Having one loop inside another, referred to as a *double loop*, is sometimes useful, e.g., when doing linear algebra. Say we want to find the maximum among the numbers stored in a 4×4 matrix `A`. The code fragment could look like

```
largest_number = A(1,1);

for i = 1:length(A)
    for j = 1:length(A)
        if A(i,j) > largest_number
            largest_number = A(i,j);
        end
    end
end
```

Here, all the `j` indices (`1 - 4`) will be covered for *each* value of index `i`. First, `i` stays fixed at `i = 1`, while `j` runs over all its indices. Then, `i` stays fixed at `i = 2` while `j` runs over all its indices again, and so on. Sketch `A` on a piece of paper and follow the first few loop iterations by hand, then you will realize how the double loop construction works. Using two loops is just a special case of using *multiple* or *nested loops*, and utilizing more than two loops is just a straightforward extension of what was shown here. Note, however, that the loop index *name* in multiple loops must be unique to each of the nested loops. Note also that each nested loop may have as many code lines as desired, both before and after the next inner loop.

The vectorized computation of heights that we did in `ball_plot.m` (Sect. 1.4) could alternatively have been done by traversing the time array (`t`) and, for each `t` element, computing the height according to the formula $y = v_0 t - \frac{1}{2}gt^2$. However, it is important to know that vectorization goes much quicker. So when speed is important, vectorization is valuable.

Use loops to compute sums

One important use of loops, is to calculate sums. As a simple example, assume some variable x given by the mathematical expression

$$x = \sum_{i=1}^{N} 2 \cdot i,$$

i.e., summing up the N first even numbers. For some given N, say $N = 5$, x would typically be computed in a computer program as:

```
N = 5;
x = 0;
for i = 1:N
    x = x + 2*i;
end
x
```

Executing this code will print the number 30 to the screen. Note in particular how the *accumulation variable* `x` is initialized to zero. The value of `x` then gets updated with each iteration of the loop, and not until the loop is finished will `x` have the correct value. This way of building up the value is very common in programming, so make sure you understand it by simulating the code segment above by hand. It is a technique used with loops in any programming language.

2.4 While Loops

Matlab also has another standard loop construction, the *while loop*, doing iterations with a loop index very much like the `for` loop. To illustrate what such a loop may look like, we consider another modification of `ball_plot.m` in Sect. 1.4. We will now change it so that it finds the time of flight for the ball. Assume the ball is thrown with a slightly lower initial velocity, say 4.5 m s^{-1}, while everything else is kept unchanged. Since we still look at the first second of the flight, the heights at the end of the flight become negative. However, this only means that the ball has fallen below its initial starting position, i.e., the height where it left the hand, so there is no problem with that. In our array `y` we will then have a series of heights which towards the end of `y` become negative. Let us, in a program named `ball_time.m` find the time when heights start to get negative, i.e., when the ball crosses $y = 0$. The program could look like this

```
g = 9.81;
v0 = 4.5;                    % Initial velocity
t = linspace(0, 1, 1000);    % Acceleration of gravity
y = v0*t - 0.5*g*t.^2;       % Generate all heights

% At this point, the array y with all heights is ready

i = 1;
while y(i) >= 0
   i = i + 1;
end

% Having the index, we may look up the time in the array t
fprintf('The time (switch from positive to negative): %f\n', t(i));
```

```
% We plot the path again just for comparison
plot(t, y);
xlabel('Time (s)');
ylabel('Height (m)');
```

If you type and run this program you should get

```
y=0 at 0.917417417417
```

The new thing here is the `while` loop only. The loop will run as long as the boolean expression `y(i) >= 0` evaluates to `true`. Note that the programmer introduced a variable (the loop index) by the name `i`, initialized it (`i = 1`) before the loop, and updated it (`i = i + 1`) in the loop. So for each iteration, `i` is *explicitly* increased by 1, allowing a check of successive elements in the array `y`.

Compared to a `for` loop, the programmer does not have to specify the number of iterations when coding a `while` loop. It simply runs until the boolean expression becomes `false`. Thus, a loop index (as we have in a `for` loop) is not required. Furthermore, if a loop index is used in a `while` loop, it is not increased automatically; it must be done explicitly by the programmer. Of course, just as in `for` loops and `if` blocks, there might be (arbitrarily) many code lines in a `while` loop. Any `for` loop may also be implemented as a `while` loop, but `while` loops are more general so not all of them can be expressed as a `for` loop.

A problem to be aware of, is what is usually referred to as an *infinite loop*. In those unintentional (erroneous) cases, the boolean expression of the `while` test never evaluates to `false`, and the program can not escape the loop. This is one of the most frequent errors you will experience as a beginning programmer. If you accidentally enter an infinite loop and the program just hangs forever, press `Ctrl+c` to stop the program.

2.5 Reading from and Writing to Files

Input data for a program often come from files and the results of the computations are often written to file. To illustrate basic file handling, we consider an example where we read x and y coordinates from two columns in a file, apply a function f to the y coordinates, and write the results to a new two-column data file. The first line of the input file is a heading that we can just skip:

```
% x and y coordinates
1.0  3.44
2.0  4.8
3.5  6.61
4.0  5.0
```

The relevant Matlab lines for reading the numbers and writing out a similar file are given in the file `file_handling.m`

```
filename = 'tmp.dat';
infileID = fopen(filename, 'r');   % Open file for reading
fgetl(infileID);                   % Read and skip first line

% First read file to count number of lines with data
no_of_lines = 0;
while ~feof(infileID)
    no_of_lines = no_of_lines + 1;
    fgetl(infileID);
end
fclose(infileID);

% Can now define arrays x and y of known length
x = zeros(no_of_lines, 1);
y = zeros(no_of_lines, 1);

% Re-open the file for reading
infileID = fopen(filename, 'r');   % Open file for reading
fgetl(infileID);                   % Read and skip first line

% Read x and y coordinates from the file and store in arrays
i = 1;
while i <= no_of_lines
    x(i) = fscanf(infileID, '%f', 1);
    y(i) = fscanf(infileID, '%f', 1);
    i = i + 1;
end
fclose(infileID);

% Next, we treat the y-coordinates and write to file

F = @(y) log(y);
y = F(y);    % Overwrite y with new values

filename = 'tmp_out.dat';
outfileID = fopen(filename, 'w');   % Open file for writing
i = 1;
while i <= no_of_lines
    fprintf(outfileID, '%10.5f %10.5f', x(i), y(i));
    i = i + 1;
end
fclose(outfileID);
```

Such a file with a comment line and numbers in tabular format is very common so Matlab has functionality to ease reading and writing. Here is the same example (file `file_handling_easy.m`):

```
filename = 'tmp.dat';
data = load(filename);
x = data(:,1);
y = data(:,2);
data(:,2) = log(y); % insert transformed y back in array
filename = 'tmp_out.dat';
outfile = fopen(filename, 'w'); % open file for writing
fprintf(outfile, '%% x and y coordinates\n');
fprintf(outfile, '%10.5f %10.5f\n', data);
fclose(outfile);
```

2.6 Exercises

Exercise 2.1: Introducing errors
Write the program `ball_function.m` as given in the text and confirm that the program runs correctly. Then save a copy of the program and use that program during the following error testing.

You are supposed to introduce errors in the code, one by one. For each error introduced, save and run the program, and comment how well Matlab's response corresponds to the actual error. When you are finished with one error, re-set the program to correct behavior (and check that it works!) before moving on to the next error.

a) Change the first line from `function ball_function()` to `ball_function()`, i.e. remove the word `function`.
b) Change the first line from `function ball_function()` to `function ball_func()`, i.e., change the name of the function.
c) Change the line `function result = y(t)` to `function y(t)`.
d) Change the line `function result = y(t)` to `function result = y()`, i.e., remove the parameter `t`.
e) Change the first statement that calls `y` from `vertical_position = y(time);` to `vertical_position = y();`.

Filename: `introducing_errors.m`.

Exercise 2.2: Compare integers a and b
Explain briefly, in your own words, what the following program does.

```
a = input('Give an integer a: ');
b = input('Give an integer b: ');

if a < b
    fprintf('a is the smallest of the two numbers\n');
elseif a == b
    fprintf('a and b are equal\n');
else
    fprintf('a is the largest of the two numbers\n');
end
```

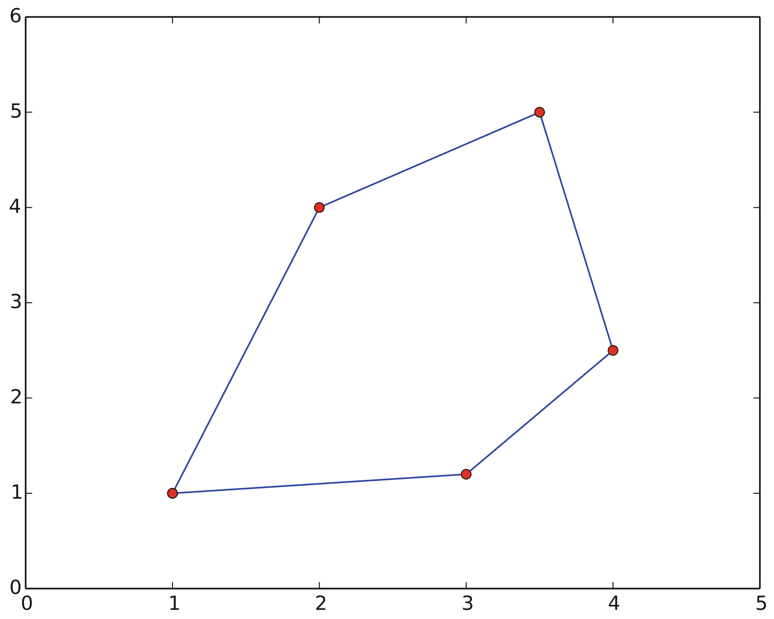
Proceed by writing the program, and then run it a few times with different values for `a` and `b` to confirm that it works as intended. In particular, choose combinations for `a` and `b` so that all three branches of the `if` construction get tested.
Filename: `compare_a_and_b.m`.

Exercise 2.3: Functions for circumference and area of a circle
Write a program that takes a circle radius `r` as input from the user and then computes the circumference `C` and area `A` of the circle. Implement the computations of `C` and

`A` as two separate functions that each takes `r` as input parameter. Print `C` and `A` to the screen along with an appropriate text. Run the program with $r = 1$ and confirm that you get the right answer.
Filename: `functions_circumference_area.m`.

Exercise 2.4: Function for area of a rectangle
Write a program that computes the area $A = bc$ of a rectangle. The values of b and c should be user input to the program. Also, write the area computation as a function that takes b and c as input parameters and returns the computed area. Let the program print the result to screen along with an appropriate text. Run the program with $b = 2$ and $c = 3$ to confirm correct program behavior.
Filename: `function_area_rectangle.m`.

Exercise 2.5: Area of a polygon
One of the most important mathematical problems through all times has been to find the area of a polygon, especially because real estate areas often had the shape of polygons, and it was necessary to pay tax for the area. We have a polygon as depicted below.



The vertices ("corners") of the polygon have coordinates (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), numbered either in a clockwise or counter clockwise fashion. The area A of the polygon can amazingly be computed by just knowing the boundary coordinates:

$$A = \frac{1}{2}\Big|(x_1y_2 + x_2y_3 + \ldots + x_{n-1}y_n + x_ny_1) \\ - (y_1x_2 + y_2x_3 + \ldots + y_{n-1}x_n + y_nx_1)\Big| .$$

Write a function `polyarea(x, y)` that takes two coordinate arrays with the vertices as arguments and returns the area. Assume that `x` and `y` are either lists or arrays.

Test the function on a triangle, a quadrilateral, and a pentagon where you can calculate the area by alternative methods for comparison.
Filename: `polyarea.m`.

Exercise 2.6: Average of integers
Write a program that gets an integer $N > 1$ from the user and computes the average of all integers $i = 1, \ldots, N$. The computation should be done in a function that takes N as input parameter. Print the result to the screen with an appropriate text. Run the program with $N = 5$ and confirm that you get the correct answer.
Filename: `average_1_to_N.m`.

Exercise 2.7: While loop with errors
Assume some program has been written for the task of adding all integers $i = 1, 2, \ldots, 10$:

```
some_number = 0;
i = 1;
while j < 11;
    some_number += 1
print some_number
```

a) Identify the errors in the program by just reading the code and simulating the program by hand.
b) Write a new version of the program with errors corrected. Run this program and confirm that it gives the correct output.

Filename: `while_loop_errors.m`.

Exercise 2.8: Area of rectangle versus circle
Consider one circle and one rectangle. The circle has a radius $r = 10.6$. The rectangle has sides a and b, but only a is known from the outset. Let $a = 1.3$ and write a program that uses a `while` loop to find the largest possible integer b that gives a rectangle area smaller than, but as close as possible to, the area of the circle. Run the program and confirm that it gives the right answer (which is $b = 271$).
Filename: `area_rectangle_vs_circle.m`.

Exercise 2.9: Find crossing points of two graphs
Consider two functions $f(x) = x$ and $g(x) = x^2$ on the interval $[-4, 4]$.

Write a program that, by trial and error, finds approximately for which values of x the two graphs cross, i.e., $f(x) = g(x)$. Do this by considering N equally distributed points on the interval, at each point checking whether $|f(x)-g(x)| < \epsilon$, where ϵ is some small number. Let N and ϵ be user input to the program and let the result be printed to screen. Run your program with $N = 400$ and $\epsilon = 0.01$. Explain the output from the program. Finally, try also other values of N, keeping the value of ϵ fixed. Explain your observations.
Filename: `crossing_2_graphs.m`.

Exercise 2.10: Sort array with numbers
The built-in function `rand` may be used to draw pseudo-random numbers for the standard uniform distribution between 0 and 1 (exclusive at both ends). See `help rand`.

Write a script that generates an array of 6 random numbers between 0 and 10. The program should then sort the array so that numbers appear in increasing order. Let the program make a formatted print of the array to the screen both before and after sorting. The printouts should appear on the screen so that comparison is made easy. Confirm that the array has been sorted correctly.
Filename: `sort_numbers.m`.

Exercise 2.11: Compute π
Up through history, great minds have developed different computational schemes for the number π. We will here consider two such schemes, one by Leibniz (1646–1716), and one by Euler (1707–1783).

The scheme by Leibniz may be written

$$\pi = 8\sum_{k=0}^{\infty}\frac{1}{(4k+1)(4k+3)},$$

while one form of the Euler scheme may appear as

$$\pi = \sqrt{6\sum_{k=1}^{\infty}\frac{1}{k^2}}.$$

If only the first N terms of each sum are used as an approximation to π, each modified scheme will have computed π with some error.

Write a program that takes N as input from the user, and plots the error development with both schemes as the number of iterations approaches N. Your program should also print out the final error achieved with both schemes, i.e. when the number of terms is `N`. Run the program with $N = 100$ and explain briefly what the graphs show.
Filename: `compute_pi.m`.

Exercise 2.12: Compute combinations of sets
Consider an ID number consisting of two letters and three digits, e.g., RE198. How many different numbers can we have, and how can a program generate all these combinations?

If a collection of n things can have m_1 variations of the first thing, m_2 of the second and so on, the total number of variations of the collection equals $m_1m_2\cdots m_n$. In particular, the ID number exemplified above can have $26\cdot26\cdot10\cdot10\cdot10 = 676{,}000$ variations. To generate all the combinations, we must have five nested for loops. The first two run over all letters A, B, and so on to Z, while the next three run over all digits $0, 1, \ldots, 9$.

To convince yourself about this result, start out with an ID number on the form A3 where the first part can vary among A, B, and C, and the digit can be among 1, 2, or 3. We must start with A and combine it with 1, 2, and 3, then continue with

B, combined with 1, 2, and 3, and finally combine C with 1, 2, and 3. A double for loop does the work.

a) In a deck of cards, each card is a combination of a rank and a suit. There are 13 ranks: ace (A), 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (J), queen (Q), king (K), and four suits: clubs (C), diamonds (D), hearts (H), and spades (S). A typical card may be D3. Write statements that generate a deck of cards, i.e., all the combinations `CA`, `C2`, `C3`, and so on to `SK`.
b) A vehicle registration number is on the form `DE562`, where the letters vary from A to Z and the digits from 0 to 9. Write statements that compute all the possible registration numbers and stores them in a list.
c) Generate all the combinations of throwing two dice (the number of eyes can vary from 1 to 6). Count how many combinations where the sum of the eyes equals 7.

Filename: `combine_sets.m`.

Exercise 2.13: Frequency of random numbers
Write a program that takes a positive integer N as input and then draws N random integers in the interval $[1, 6]$ (both ends inclusive). In the program, count how many of the numbers, M, that equal 6 and write out the fraction M/N. Also, print all the random numbers to the screen so that you can check for yourself that the counting is correct. Run the program with a small value for `N` (e.g., `N = 10`) to confirm that it works as intended.

Hint Use `1+floor(6*rand())` to draw a random integer between 1 and 6.
Filename: `count_random_numbers.m`.

Remarks For large N, this program computes the probability M/N of getting six eyes when throwing a dice.

Exercise 2.14: Game 21
Consider some game where each participant draws a series of random integers evenly distributed from 0 and 10, with the aim of getting the sum as close as possible to 21, but *not larger* than 21. You are out of the game if the sum passes 21. After each draw, you are told the number and your total sum, and are asked whether you want another draw or not. The one coming closest to 21 is the winner.

Implement this game in a program.

Hint Use `floor(11*rand())` to draw random integers in [0, 10].
Filename: `game_21.m`.

Exercise 2.15: Linear interpolation
Some measurements y_i, $i = 0, 1, \ldots, N$ (given below), of a quantity y have been collected regularly, once every minute, at times $t_i = i$, $i = 0, 1, \ldots, N$. We want to find the value y *in between* the measurements, e.g., at $t = 3.2$ min. Computing such y values is called *interpolation*.

Let your program use *linear interpolation* to compute y between two consecutive measurements:

1. Find i such that $t_i \leq t \leq t_{i+1}$.
2. Find a mathematical expression for the straight line that goes through the points (i, y_i) and $(i+1, y_{i+1})$.
3. Compute the y value by inserting the user's time value in the expression for the straight line.

a) Implement the linear interpolation technique in a function that takes an array with the y_i measurements as input, together with some time t, and returns the interpolated y value at time t.
b) Write another function with a loop where the user is asked for a time on the interval $[0, N]$ and the corresponding (interpolated) y value is written to the screen. The loop is terminated when the user gives a negative time.
c) Use the following measurements: $4.4, 2.0, 11.0, 21.5, 7.5$, corresponding to times $0, 1, \ldots, 4$ (min), and compute interpolated values at $t = 2.5$ and $t = 3.1$ min. Perform separate hand calculations to check that the output from the program is correct.

Filename: `linear_interpolation.m`.

Exercise 2.16: Test straight line requirement
Assume the straight line function $f(x) = 4x + 1$. Write a script that tests the "point-slope" form for this line as follows. Within a chosen interval on the x-axis (for example, for x between 0 and 10), randomly pick 100 points on the line and check if the following requirement is fulfilled for each point:

$$\frac{f(x_i) - f(c)}{x_i - c} = a, \qquad i = 1, 2, \ldots, 100,$$

where a is the slope of the line and c defines a fixed point $(c, f(c))$ on the line. Let $c = 2$ here.
Filename: `test_straight_line.m`.

Exercise 2.17: Fit straight line to data
Assume some measurements $y_i, i = 1, 2, \ldots, 5$ have been collected, once every second. Your task is to write a program that fits a straight line to those data.

a) Make a function that computes the error between the straight line $f(x) = ax+b$ and the measurements:

$$e = \sum_{i=1}^{5} (ax_i + b - y_i)^2 .$$

b) Make a function with a loop where you give a and b, the corresponding value of e is written to the screen, and a plot of the straight line $f(x) = ax + b$ together with the discrete measurements is shown.

c) Given the measurements 0.5, 2.0, 1.0, 1.5, 7.5, at times 0, 1, 2, 3, 4, use the function in b) to interactively search for a and b such that e is minimized.

Filename: `fit_straight_line.m`.

Remarks Fitting a straight line to measured data points is a very common task. The manual search procedure in c) can be automated by using a mathematical method called the *method of least squares*.

Exercise 2.18: Fit sines to straight line

A lot of technology, especially most types of digital audio devices for processing sound, is based on representing a signal of time as a sum of sine functions. Say the signal is some function $f(t)$ on the interval $[-\pi, \pi]$ (a more general interval $[a, b]$ can easily be treated, but leads to slightly more complicated formulas). Instead of working with $f(t)$ directly, we approximate f by the sum

$$S_N(t) = \sum_{n=1}^{N} b_n \sin(nt), \tag{2.1}$$

where the coefficients b_n must be adjusted such that $S_N(t)$ is a good approximation to $f(t)$. We shall in this exercise adjust b_n by a trial-and-error process.

a) Make a function `sinesum(t, b)` that returns $S_N(t)$, given the coefficients b_n in an array `b` and time coordinates in an array `t`. Note that if `t` is an array, the return value is also an array.
b) Write a function `test_sinesum()` that calls `sinesum(t, b)` in a) and determines if the function computes a test case correctly. As test case, let `t` be an array with values $-\pi/2$ and $\pi/4$, choose $N = 2$, and $b_1 = 4$ and $b_2 = -3$. Compute $S_N(t)$ by hand to get reference values.
c) Make a function `plot_compare(f, N, M)` that plots the original function $f(t)$ together with the sum of sines $S_N(t)$, so that the quality of the approximation $S_N(t)$ can be examined visually. The argument `f` is a Matlab function implementing $f(t)$, `N` is the number of terms in the sum $S_N(t)$, and `M` is the number of uniformly distributed t coordinates used to plot f and S_N.
d) Write a function `error(b, f, M)` that returns a mathematical measure of the error in $S_N(t)$ as an approximation to $f(t)$:

$$E = \sqrt{\sum_i (f(t_i) - S_N(t_i))^2},$$

where the t_i values are M uniformly distributed coordinates on $[-\pi, \pi]$. The array `b` holds the coefficients in S_N and `f` is a Matlab function implementing the mathematical function $f(t)$.
e) Make a function `trial(f, N)` for interactively giving b_n values and getting a plot on the screen where the resulting $S_N(t)$ is plotted together with $f(t)$. The error in the approximation should also be computed as indicated in d). The argument `f` is a Matlab function for $f(t)$ and `N` is the number of terms N in the

sum $S_N(t)$. The `trial` function can run a loop where the user is asked for the b_n values in each pass of the loop and the corresponding plot is shown. You must find a way to terminate the loop when the experiments are over. Use `M=500` in the calls to `plot_compare` and `error`.

f) Choose $f(t)$ to be a straight line $f(t) = \frac{1}{\pi}t$ on $[-\pi, \pi]$. Call `trial(f, 3)` and try to find through experimentation some values b_1, b_2, and b_3 such that the sum of sines $S_N(t)$ is a good approximation to the straight line.

g) Now we shall try to automate the procedure in f). Write a function that has three nested loops over values of b_1, b_2, and b_3. Let each loop cover the interval $[-1, 1]$ in steps of 0.1. For each combination of b_1, b_2, and b_3, the error in the approximation S_N should be computed. Use this to find, and print, the smallest error and the corresponding values of b_1, b_2, and b_3. Let the program also plot f and the approximation S_N corresponding to the smallest error.

Filename: `fit_sines.m`.

Remarks

1. The function $S_N(x)$ is a special case of what is called a *Fourier series*. At the beginning of the 19th century, Joseph Fourier (1768–1830) showed that any function can be approximated analytically by a sum of cosines and sines. The approximation improves as the number of terms (N) is increased. Fourier series are very important throughout science and engineering today.
   (a) Finding the coefficients b_n is solved much more accurately in Exercise 3.12, by a procedure that also requires much less human and computer work!
   (b) In real applications, $f(t)$ is not known as a continuous function, but function values of $f(t)$ are provided. For example, in digital sound applications, music in a CD-quality WAV file is a signal with 44100 samples of the corresponding analog signal $f(t)$ per second.

Exercise 2.19: Count occurrences of a string in a string

In the analysis of genes one encounters many problem settings involving searching for certain combinations of letters in a long string. For example, we may have a string like

```
gene = 'AGTCAATGGAATAGGCCAAGCGAATATTTGGGCTACCA'
```

We may traverse this string letter by letter. The length of the string is given by `length(gene)`, so with a loop index `i`, `for i = 1:length(gene)` will produce the required index values. Letter number `i` is then reached through `gene(i)`, and a substring from index `i` up to and including `j`, is created by `gene(i:j)`.

a) Write a function `freq(letter, text)` that returns the frequency of the letter `letter` in the string `text`, i.e., the number of occurrences of `letter` divided by the length of `text`. Call the function to determine the frequency of `C` and `G` in the `gene` string above. Compute the frequency by hand too.

b) Write a function `pairs(letter, text)` that counts how many times a pair of the letter `letter` (e.g., `GG`) occurs within the string `text`. Use the function to determine how many times the pair `AA` appears in the string `gene` above. Perform a manual counting too to check the answer.

c) Write a function `mystruct(text)` that counts the number of a certain structure in the string `text`. The structure is defined as `G` followed by `A` or `T` until a double `GG`. Perform a manual search for the structure too to control the computations by `mystruct`.

Filename: `count_substrings.m`.