Contents

ANONYMOUS FUNCTIONS

Anonymous functions are a simple and concise way to define a function that contains only a single executable statement, such as a polynomial expression or any basic formula that can be defined in one line of code. Anonymous functions are stored as a type of variable (`function_handle`) and thus do not need any special construct nor do they need to be defined in their own M-File (as we learned previously with user-defined functions). Because anonymous functions are defined as a type of variable, they will show up in the workspace and follow the same rules are other variables. Perhaps more importantly, they will be erased with the `clear` command, and are defined only locally in the workspace. Anonymous functions are created using only a single line of code and can be defined anywhere inside of an M-File, even if the M-File is a function file. They accept input arguments and return outputs similar to how built-in functions do.

Anonymous functions are most useful if you have a formula or calculation that needs to be carried out many times throughout your code with various different values for input arguments. By defining an anonymous function, you avoid any potential errors that could result from re-typing or copy/pasting the formula. If you use an anonymous function, you would only need to type the formula once when the anonymous function is created, and the formula would be stored in the `function_handle` for later use. Calling the function is concise and easy, and the output of the function can be saved to a new variable if desired.

First, let's look at the syntax used to create anonymous functions:

```
name_of_func = @(inputs,listed,here) formula/expression/calculation
```

- When creating an anonymous functions, you always start with the name of the function to the left of the equals sign. Remember that an anonymous function will be defined as a variable, so be sure to follow the rules for valid variable names, do not use any keywords or built-in function names, and chose a name that is intuitive for what the function does. Here the variable `name_of_func` will be created as a `function_handle` in the workspace.
- After the equals sign, you use the "at symbol", `@`, immediately followed by the input arguments enclosed in parenthesis.
    - An anonymous function can have any number of inputs, including none.

        - Be careful! If your anonymous function has no inputs, you need to use empty parenthesis "`()`" when you define and call the anonymous function.
- After listing the inputs enclosed in parenthesis, you provide the expression (formula/calculations/command) that the function needs to perform. Be sure that any variables used in the expression have already been defined and are stored in memory.
    - You do not need to pass all variables used in the expression in as input arguments. For example, if there are constants in your equation such as in `y = a*x^2 + b*x + c` we only need to pass in `x` as an input argument if `a`, `b`, and `c` do not change. In this case, the values of `a`, `b`, and `c` that already exist in memory when we define the anonymous function will be stored in the `function_handle`, and when we call the function, those values will be used with whatever value for `x` we send as an input argument. See the example below to see how we can create an anonymous function that solves this polynomial for `y` (the output) as a function of `x` (the input) with `a`, `b`, and `c` as constants.
- You do not list any output variables when creating anonymous functions as you do with user-defined functions. The output variable is defined when you call the anonymous function.
    - In MAE10, we will only work with anonymous functions that have one output.
- Adding a semicolon to the end of an anonymous function definition will prevent MATLAB from displaying the `function_handle` that you just created, but it will of course still be stored in memory to be called/used later in the code.

Once we have created an anonymous function, we need to call the function in order for it to be executed (just like when we use built-in or user-defined functions). The syntax we use to call the function is as follows:

```
output_name = name_of_func(inputs,listed,here)
```

As was the case with user-defined functions, the names of the input variables listed when we define the function do not need to match the names of the input variables on the call to the function (i.e., **dummy variables**).

Let's look at a basic example that simply adds `2` to any variable we send as input.

In `myfile.m`:

```
y = 5;
add2 = @(x) x+2;
mynum = add2(y)
```

Sample Output:

```
y =

     5


add2 =

  function_handle with value:

    @(x)x+2


mynum =

     7
```

In this first example we left everything unsuppressed so we can see the creation of the `function_handle` and subsequent calculations. However, in most cases, this is unnecessary output so we can include the semicolon. Notice that when we call the function, we use the value from the variable `y` even though inside the function the input is listed as `x`. Again this is not a problem, since `x` is just a dummy variable (essentially a placeholder). As we learned with user-defined functions, it is the ***values*** that are passed into the function and the names of the variables do not need to match. The output of the function is saved to a variable called `mynum`, which is simply the value of the input plus `2`.

Let's now create an anonymous function to evaluate the polynomial expression we mentioned previously:

In `myfile.m`:

```
a = 3;
b = 5;
c = 2;
f = @(x) a*x^2 + b*x + c;
y = f(5)
```

Sample Output:

```
y =

   102
```

Notice that we do not need to include `a`, `b`, and `c` as inputs. Since these variables are defined before the anonymous function is created, their values are automatically stored in the `function_handle` and used in the anonymous function when it is called. In other words, the `function_handle` stores not only the expression, but also variables required to evaluate the expression that are not listed as inputs (the constants `a`, `b`, and `c` in this example). Note that if you want to use different values for the constants, you will need to create a new `function_handle` (and define new values for `a`, `b`, and `c` beforehand). The value for `x` needed in the function is passed in when we call the function on the fifth line. Using the value of `5` for `x`, the expression is evaluated and the resulting value is saved to the variable `y`. Once the anonymous function is created and the `function_handle` is stored in memory, we call it as many times as we like and evaluate the expression at different values of `x`. We can also call anonymous functions from the command line or any other M-File once the `function_handle` is stored in memory (i.e., they can be used outside of the original M-File they are created in until you use `clear` to erase variables in memory or exit MATLAB).

Using Anonymous Functions with Arrays

In the previous examples we sent only scalars as input arguments, but we can send arrays as input in the same way. However, we need to be careful when we create the anonymous function to ensure that the operations included in the expression are valid for arrays as well as scalars. For example, we should include the dot operator when doing multiplication of exponentiation to ensure that the expression will work for both scalars and arrays. To illustrate this point, let's do one more example where we have more than one input argument, and at least one of the inputs is an array.

In `myfile.m`:

```
y0 = 0; % initial position [m]
v0 = 25; % initial vel [m/s]
g = -9.8; % vertical accel [m/s^2]
theta = 45; % angle [deg]
height = @(t,v0) y0 + v0*sin(theta*(pi/180))*t + 0.5*g*t.^2;
y = height(1,10)
time = 0:0.5:3;
y2 = height(time,v0)
```

Sample Output:

```
y =

    2.1711
```

```
y2 =

         0    7.6138   12.7777   15.4915   15.7553   13.5692    8.9330
```

In this example we created an anonymous function to calculate the height of a projectile at any given time (if we send a scalar input) or at many different times (if we send an array as input). Note that we had to include the dot operator in the exponentiation (`t.^2`) to ensure the function would work if we send an array as input for the time. In this example we have two input arguments, `t` and `v0`, so that we can easily calculate the height of the project at different times or with different initial velocities. The constants `theta` and `g` are not inputs – they are stored in the `function_handle`. Notice that we also used `pi` in the expression. Because `pi` is a built-in, globally defined variable, we can use it anywhere in MATLAB, including anonymous functions.

Anonymous functions are a small but important topic as they can be easily integrated into your MATLAB codes to perform key tasks or calculations. It is possible to use anonymous functions in even more sophisticated ways, but this is beyond the scope of what is covered in MAE10. For additional information and examples, please see the MathWorks help page on anonymous functions and/or consult your textbook.