Chapter 17 - Function M-Files Part 3

Table of Contents

I'm Lost - Getting Started

Suppose we want to write a function m-file called `myfunstuff`. Don't worry about anything else just yet. The first thing we should do, in Matlab is:

```
edit myfunstuff.m
```

Note that: The filename has to match the function name.

Create the basic framework. We're going to assume that the function returns a single return variable and we'll denote it `r`. If the function m-file is going to take one parameter called `a`, then write the following just to get thing testing:

```
function r = myfunstuff(a)
  r = a+1;
end
```

Save this file. Now, back in the Matlab console, try typing the following and we'll see:

```
myfunstuff(17)
ans =
      18
```

Take a minute to understand what's happening. In the console we type `funstuff(17)`. Matlab calls the `funstuff` function and sends it the value `17`. The `funstuff` function gets the value `17` and puts it in the variable `a`. It then sets `r=a+1` so `r=18`. Since we used `r` as the return variable that gets sent back and shows in the Matlab console.

We can try it with other values too:

```
myfunstuff(-6)
ans =
      -5
myfunstuff(0)
ans =
      1
```

Now then, adding 1 is pretty boring. The beauty of a function m-file is that we can add all sort of logic inside the file. Suppose we wanted the function to look at `a` and if it's positive, add one, if it's negative, subtract 1 and if it's zero, leave it alone. We would do:

```
function r = myfunstuff(a)
  if (a > 0)
```

```
    r = a + 1;
  elseif (a < 0)
    r = a - 1;
  else
    r = a;
  end
end
```

Again, take minute to think about how this works before trying it. When we try it we'll see:

```
myfunstuff(-6)
ans =
      -7
myfunstuff(0)
ans =
      0
myfunstuff(7.5)
ans =
      8.5
```

A Fancier Example

The real power in all of this is that we can pass many parameters to the function m-file and the function m-file can do almost anything with those parameters and pass something back. Moreover we don't have to pass just numbers. We can pass functions, for example, but they have to be function handles. Inside the function m-file we can have `if` statements, `for` loops and `while` loops, just to name a few things.

Suppose we want our function m-file to accept a function and a value. First our very basic structure would look something like this;

```
function r = myfunstuff(f,a)
  r = 12345;
end
```

Notice that it doesn't do anything at all, nothing, yet, except return the value `12345` and that's just for testing.

We can try it, keeping in mind that the function must be sent as a function handle:

```
myfunstuff(@(x) x^2,23)
ans =
       12345
myfunstuff(@(x) sin(x),17)
ans =
       12345
```

It returns `12345` each time, because that's what `r` is. Everything else is totally ignored.

Okay, just for testing, suppose we want to plug the value into the function and return that. Easy!

```
function r = myfunstuff(f,a)
  r = f(a);
end
```

And test it:

```
myfunstuff(@(x) sin(x),17)
  ans =
     -0.9614
```

That's because `sin(17)=-0.9614` as Matlab can tell us:

```
sin(17)
```

```
ans =
   -0.9614
```

Okay now, let's get fancy. Suppose we know that `a` will be positive and we want to plug `a` into `f`, then `a/2`, then `a/3`, and so on, until we reach some `n` where `a/n` is less than 1, and we want to add all the resulting values up, excluding that final value. How would we do this?

First we know it'll be a `while` loop because we don't know how many steps it'll take.

Moreover we want it to do `f(a/n)` for `n=1,...` until `a/n<1`. In other words we want it to continue `while (a/n >= 1)`. Okay, so far consider this:

```
function r = myfunstuff(f,a)
  n = 1;
  while (a/n >= 1)
    n = n + 1;
  end
  r = f(a);
end
```

Really, really take a minute to understand what this does and doesn't do. It accepts parameters `f` and `a`. It sets `n=1` and then, `while (a/n >= 1)` it adds `1` to `n`. This means when `a/n < 1` it'll stop.

Notice that other than increasing `n` nothing else happens inside the loop. Otherwise the function m-file still returns `f(a)`.

If we test it from Matlab we'll see:

```
myfunstuff(@(x) sin(x),17)
  ans =
     -0.9614
```

Just like before. The function m-file is going through the `while` loop but we don't see anything related to that.

Okay, so next we need to take the `a/n` values and plug them into `f`, and we need to add them. To add them we'll declare a new variable, called `runningtotal`, we'll start it at `0` and increase it inside the loop:

```
function r = myfunstuff(f,a)
  n = 1;
  runningtotal = 0;
  while (a/n >= 1)
    runningtotal = runningtotal + f(a/n);
    n = n + 1;
  end
  r = f(a);
end
```

If we test this:

```
myfunstuff(@(x) sin(x),17)
  ans =
     -0.9614
```

Oh drat, same result. Why? Oh, we're still doing `r=f(a)` we need to return our sum, our `runningtotal`. So finally:

```
function r = myfunstuff(f,a)
  n = 1;
  runningtotal = 0;
  while (a/n >= 1)
    runningtotal = runningtotal + f(a/n);
    n = n + 1;
  end
  r = runningtotal;
end
```

And now we see:

```
myfunstuff(@(x) sin(x),17)
  ans =
      8.3691
```

More Notes and the Midpoint Rule

Don't be afraid to play inside the m-file while we work. we can display things to check on them as we go.

For another example we're going to step through writing a midpoint rule calculator, including baby steps, testing, etc., as if we'd never done this before.

First we edit the file:

```
edit mymidpointrule.m
```

We create the basic structure. We want to pass it a function, an interval and a number of subintervals, so something like:

```
function r = mymidpointrule(f,a,b,n)
  r = 17;
end
```

The `r=17` is there for testing and to remind myself that I'd better make `r` equal to the thing we want at the end!

Next, the midpoint rule involves dividing `[a,b]` into `n` subintervals, finding the midpoint of each, plugging each into `f`, multiplying by the subinterval width, and adding. Whew!

First an easy bit, let's find the subinterval width, which would be `(b-a)/n`:

```
function r = mymidpointrule(f,a,b,n)
  sw = (b-a)/n
  r = 17;
end
```

If we test this in Matlab:

```
mymidpointrule(@(x) x^2,0,2,4)
```

```
sw =
    0.5000
ans =
  17
```

Whoa, why are two things showing up? Well, the `sw` is showing up (and it's right, because the interval length is 2 and we're using 4 subintervals) because that line in the m-file has no semicolon at the end. The `ans` is the return value. For now we can leave the semicolon out to see the value.

Next we need a `for` loop. It needs to start at the first midpoint and go to the last, and it needs to do it in steps of `sw`.

The first midpoint is `a+sw/2` and the last is `b-sw/2`. So how about:

```
function r = mymidpointrule(f,a,b,n)
  sw = (b-a)/n
  for x = [a+sw/2:sw:b-sw/2]
    x
  end
  r = 17;
end
```

That single line `x` in there just spits the value back, so if we test this:

```
mymidpointrule(@(x) x^2,0,2,4)

  sw =
       0.5000
  x =
      0.2500
  x =
      0.7500
  x =
      1.2500
  x =
      1.7500
  ans =
      17
```

Now it shows we `sw` and then it goes through the midpoints, one by one, from `0.2500` (the midpoint of the first subinterval) in steps of `0.5000`, to `1.7500` (the midpoing of the second subinterval).

Next, those `x` values should be plugged into `f`, so let's do that. Also let's suppress the `sw` with a semicolon since we know it's behaving properly:

```
function r = mymidpointrule(f,a,b,n)
  sw = (b-a)/n;
  for x = [a+sw/2:sw:b-sw/2]
    f(x)
  end
  r = 17;
end
```

Let's test it:

```
mymidpointrule(@(x) x^2,0,2,4)
```

```
  ans =
     0.0625
  ans =
      0.5625
  ans =
      1.5625
  ans =
      3.0625
  ans =
      17
```

Now we no longer see `sw` and we see `f` at each midpoint.

Next, we'd like to multiply each of those `f` values by `sw`:

```
function r = mymidpointrule(f,a,b,n)
  sw = (b-a)/n;
  for x = [a+sw/2:sw:b-sw/2]
    f(x)*sw
  end
  r = 17;
end
```

And test it:

```
mymidpointrule(@(x) x^2,0,2,4)
  ans =
      0.0312
  ans =
      0.2812
  ans =
      0.7812
  ans =
      1.5312
  ans =
      17
```

Looking better, these are the areas of the small rectangles! Almost finished. Next, let's add them up, with a `runningtotal`:

```
function r = mymidpointrule(f,a,b,n)
  sw = (b-a)/n;
  runningtotal = 0;
  for x = [a+sw/2:sw:b-sw/2]
    runningtotal = runningtotal + f(x)*sw
  end
  r = 17;
end
```

And test it:

```
mymidpointrule(@(x) x^2,0,2,4)
  runningtotal =
      0.0312
  runningtotal =
      0.3125
  runningtotal =
      1.0938
  runningtotal =
      2.6250
  ans =
      17
```

Almost done! Lastly, that final `runningtotal` should be the value returned and we should add a semi-colon to suppress the output from each step:

```
function r = mymidpointrule(f,a,b,n)
  sw = (b-a)/n;
  runningtotal = 0;
  for x = [a+sw/2:sw:b-sw/2]
    runningtotal = runningtotal + f(x)*sw;
  end
  r = runningtotal;
end
```

And now, ta-da!

```
mymidpointrule(@(x) x^2,0,2,4)
  ans =
    2.6250
```

Moreover since the midpoint rule approximate the integral we can test it using a really big `n` value and comparing to the integral:

```
mymidpointrule(@(x) x^2,0,2,1000)
  ans =
      2.6667
int(x^2,0,2)
  ans =
      8/3
```

It works!

Published with MATLAB® R2017a