Chapter 4

More on Functions

The previous chapter covered the bare minimum on MATLAB functions. Here are some more advanced concepts and tools that at some point you will probably find very useful.

4.1 Function handles and anonymous functions

One way to look at a function is as an algorithm or process. Sometimes, though, we might prefer to look at a function as data. For instance, the mathematical problem of definite integration could be seen as a map from inputs in the form of an interval $[a,b]$ and a function $f(x)$ to the value of $\int_a^b f(x)\,dx$. As you might expect, this is a classical problem in computing, and there are many algorithms available to approximate this map. Thus we can find ourselves in the situation of writing or using a function that accepts another function as data. MATLAB likes to call these **function functions**, and you can get a list of them by typing `help funfun`. Other common problems that involve operation on functions as data are finding roots, optimization, and solving differential equations.

MATLAB requires you to use a special construct called a **function handle** to distinguish between the invocation of a function and the abstraction of that function as data. For example, the built-in `fzero` attempts to find a root of a function supplied as its first input argument, starting from a guess in the second argument. We cannot call it to find a root of $\sin(x)$ by typing

```
>> fzero(sin,3)  % error!
```

because the MATLAB interpreter would first try to call the built-in `sin` function with no inputs, which is an error. Instead, you must create a function handle by using the `@` prefix, as in

```
>> fzero(@sin,3)
ans =
    3.1416
```

Any callable function in scope can be converted into a handle, including built-ins and your own M-file functions.

Often, you need a function definition that combines built-in operations in an elementary way. The best tool for this situation is an **anonymous function**. An anonymous function lets you simultaneously define a function and create a handle to it.[7] For instance, to create a representation of the mathematical function $\sin(x)+\cos(x)$, you would enter

```
>> sincos = @(x) sin(x) + cos(x);
```

Now `sincos` is a function of one argument, and the syntaxes `sincos(pi/4)` and `sincos(rand(2))` are perfectly acceptable. It is also straightforward to create an anonymous function of multiple arguments, as in

```
>> w = @(x,t,c) cos(x-c*t);
```

There is no need to assign the anonymous function to a named variable, however, as in this syntax:

```
>> fzero( @(x) sin(x)+cos(x), 0 )
ans =
   -0.7854
```

When you define an anonymous function, the variable names following the `@` symbol are inputs to the function. If the same names are defined in the current workspace, those values are unavailable within the anonymous definition. However, other values defined in the current workspace at the time of definition *are* available to the anonymous function definition. Furthermore, any such values used within the definition are "baked in" to the anonymous function: they remain fixed, even if the values are later changed or the defining workspace goes out of scope.

This feature can be amazingly convenient. A prototypical situation is that you have an M-file function `bigfun(a,x,b)` that accepts parameters a and b in addition to a variable x of interest. Syntactically, many function functions accept only a function of one variable as data. Hence you define an anonymous function wrapper such as

```
>> f = @(x) bigfun(0.5,x,b_value);
```

that locks in particular parameter values and accepts only one input variable. For example, consider again `fzero`, which finds a root of a function of one variable. Suppose you wish to find a solution of $e^{-ax} = x$ for multiple values of the parameter a. An anonymous function makes this a snap:

[7] Before anonymous functions became available, **inline functions** offered some of the same capability in older versions of MATLAB. Inline functions are less useful in several ways, however, and offer no real advantages.

```
>> for a = 1:0.25:2, fzero( @(x) exp(-a*x)-x, 0 ), end
ans =
    0.5671
ans =
    0.5212
ans =
    0.4839
ans =
    0.4528
ans =
    0.4263
```

In the body of the loop, the variable `a` is assigned each of the values 1, 1.25, 1.5, 1.75, 2 in turn. Each time, a new anonymous function is created to lock in that value of `a` and create a function of `x` alone, which is fed to `fzero`.

4.2 Subfunctions and nested functions

A single M-file may hold more than one function definition. The function header line at the top of a file defines the **primary function** of the file. Two other types of functions can be in the same file: subfunctions and nested functions.

A **subfunction** is mostly a convenient way to avoid directory and name clutter. The subfunction begins after the end of the primary function, with a new function header line. Every subfunction in the file is available to be called by the primary function and the other subfunctions. In all respects, it behaves like a primary function in a separate file, including the use of a private variable workspace. As a silly example of a function using a subfunction, consider a new version of `quadform.m`:

```
function [x1,x2] = quadform(a,b,c)
  d = discrim(a,b,c);
  x1 = (-b + d) / (2*a);
  x2 = (-b - d) / (2*a);
end  % quadform()

function D = discrim(A,B,C)
  D = sqrt(B^2 - 4*A*C);
end  % discrim()
```

The `end` line is optional for single-function files, but it is a good idea when subfunctions are involved, and it is mandatory when using nested functions. Changes made to a, b, or c inside `discrim` would not propagate into the rest of `quadform`, nor would any other variables in the primary function be available to `discrim`.

Outside of the defining file, subfunctions are not visible—only the primary function can be called. However, it is possible to exploit a major and useful

exception to this rule. Suppose you want to investigate the eigenvalues of a certain matrix $A(x)$—specifically, you wish to find the smallest x such that the maximum real part of the eigenvalues equals 1. Here is a simple approach using `fzero` to reach the target value:

```
function x0 = findx
  x0 = fzero(@objective,[0 10]);
end

function r = objective(x)
  B = diag(ones(499,1),1);  A = B-B';
  A(1,1) = x;
  e = eig(A);  r = max(real(e)) - 1;
end
```

The subfunction `objective` is passed as a handle to `fzero`, which is able to use the handle to access and call the subfunction, even though it would be out of scope if `fzero` were to try to invoke it directly by name.

A **nested function** is similar to a subfunction, but it is defined *within* the scope of a parent function. It behaves differently from a subfunction in one very important respect: its variable workspace can overlap that of its parent. Any variable used in both the nested function and its parent is shared.

One reason for using a nested function is to create lasting side effects. Consider again the eigenvalue example just above. Because `fzero` is a "black box" with a standardized syntax, it can return only the value of x we seek. It cannot tell us, for example, the imaginary part of the eigenvalue whose real part equals 1, nor can it tell us any other information about the eigenvalues. To get that information, we would have to repeat much of the `objective` function once again, which makes for messy code and extra computation time. Reorganizing the computation using a nested function, however, allows us to pass extra information out of the objective calculation:

```
function x0 = findx
  function r = objective(x)
    A(1,1) = x;
    e = eig(A);  r = max(real(e)) - 1;
  end

  B = diag(ones(499,1),1);  A = B-B';
  e = [];  % create a shared variable for side effect
  x0 = fzero(@objective,[0 10]);
  plot(e,'x')
end
```

The line `e = [];` is critical. Giving the variable `e` a definition in the parent workspace allows it to be shared between the nested `objective` and its parent, so that at the termination of `fzero`, the value of `e` is the same as what it was at the end of the most recent eigenvalue computation. Notice that we also passed static information about the variable `A` *into* the nested function, potentially saving otherwise wasted time over the previous version.

Variable sharing with nested functions is a technique best used cautiously. Experience shows that reliance on common workspaces can lead to code that is easily broken and confusing to read. There are situations like this one, however, in which the technique offers possibilities that are hard to duplicate otherwise.

4.3 Errors and warnings

MATLAB functions may encounter statements that are impossible to execute, in a way that could not have been predicted until the execution is to take place. For example, the statement `A*B` is syntactically valid but meaningless in context when `A` and `B` are defined as matrices of incompatible sizes. In such a situation, an **error** is thrown: execution halts, a message is displayed, and control is returned to the prompt, with the output arguments of the function ignored. You can throw errors in your own functions with the `error` statement, called with a string that is displayed as the message. Similar to an error is a **warning**, which displays a message but allows execution to continue. You create such messages using `warning`.

Sometimes you would like the ability to recover from an error in a subroutine and continue with a contingency plan. This can be done using the `try/catch` construct. For example, the following will continue asking for a statement until you give it one that executes successfully:

```
done = false;
while ~done
  state = input('Enter a valid statement: ','s');
  try
    eval(state);
    done = true;
  catch me
    disp('That was not a valid statement! Look:')
    disp(me.message)
  end
end
```

Within the `catch` block you can find the most recent error message using `lasterr`, or (as is preferred in recent MATLAB versions) by inspecting the exception object `me`, as shown above and explained in the help pages for `MException`.

4.4 Input and output arguments, revisited

Good programming practice dictates that you check the input arguments to a function for validity. This is an especially good idea in a weakly typed language such

as MATLAB, where valid syntax can lead to meaningless results. There are a number of helper functions for checking inputs, as demonstrated in the following fragment:

```
function [x,y] = myfun(a,b,c)

assert(isnumeric(a), 'First input must be a number.')
assert(numel(a)==1, 'First input must be a scalar.')
assert(~any(isinf(b)), 'Second input must be finite.')
assert(~any(isnan(b)), 'No NaNs allowed in second input.')
assert(ischar(c), 'Third input must be a string.')
```

If the first expression given to `assert` is not true, an error is thrown with the message and is given as the second argument.

Although a function's header spells out the number and names of the input and output arguments, these are not enforced at the time of function invocation. Instead, execution proceeds as long as possible until something undefined or otherwise illegal happens. This fact allows you to accept different numbers of input arguments in different situations. One common use of variable-length argument lists is to give default values to missing inputs, as in the following fragment:

```
function shirt(neck,sleeve,color,cuff)

if nargin < 4
  cuff = 'button';
  if nargin < 3
    color = 'blue';
  end
end
```

The value of `nargin` is always set at run-time to the number of input arguments with which the function was actually called. In the syntax above, default values are assigned to the third and fourth arguments if they are missing. (The inputs are place-sensitive, however, so there is no way to specify a `cuff` without also specifying a `color` in this setup.) If the function is called with just zero or one inputs, execution will continue unless and until a code reference is made to `neck` or `sleeve`, which are not given defaults.

For a different take on allowing flexibility in the number of input arguments, see section 6.6.

Exercises

4.1. Write a function `plusone(f,x)` that, given a function f and value x, returns $f(x)+1$.

4.2. Write a function `trap(f,a,b,n)` that computes the trapezoidal rule approximation to $\int_a^b f(x)\,dx$,

$$\frac{h}{2}\big[f(x_0)+2f(x_1)+2f(x_2)+\cdots+2f(x_{n-1})+f(x_n)\big],$$

where $h=(b-a)/n$ and $x_i=a+ih$. Test your function on $\sin(x)+\cos(x)$ for $0\le x\le \pi/3$. For a greater challenge, write a function `simp` for Simpson's rule,

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\big[f(x_0)+4f(x_1)+2f(x_2)+\cdots+4f(x_{n-1})+f(x_n)\big].$$

This formula requires n to be even. You may choose to check the input for this.

4.3. Write a function `bisect(f,a,b,tol)` that employs the *bisection* method for finding a value of x such that $f(x)=0$. The first input is a handle to a function computing f. Assuming that f is continuous and that $f(a)f(b)<0$, the function has at least one root in the interval (a,b). Define $m=(a+b)/2$, and if $f(m)\neq 0$, then either $f(a)f(m)<0$ or $f(b)f(m)<0$, so the root is either in (a,m) or (m,b). Continue the process until the root is contained in an interval of length less than `2*tol`, and stop. The best code takes care not to evaluate the given `f` more often than necessary.

4.4. Write a function `newton(f,fprime,x0,tol)` that implements Newton's iteration for rootfinding on a scalar function:

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}.$$

The first two inputs are handles to functions computing f and f', and the third input is an initial root estimate. Continue the iteration until either $|f(x_{n+1})|$ or $|x_{n+1}-x_n|$ is less than `tol`.

4.5. Modify `newton` from the previous exercise so that it works on a system of equations $\mathbf{F}(\mathbf{x})$. The function `fprime` now returns the Jacobian matrix `J`, and the Newton update is written mathematically as

$$\mathbf{x}_{n+1}=\mathbf{x}_n-J^{-1}\mathbf{F}(\mathbf{x}_n),$$

although in numerical practice one does not compute the inverse of the Jacobian but solves a linear system of equations in which $\mathbf{F}(\mathbf{x}_n)$ is the right-hand side.

4.6. Many simple financial instruments that have regular equal payments, such as car loans or investment annuities, can be modeled by the equation

$$F=P\left(\frac{(1+r)^t-1}{r}\right),$$

where P is the regular payment, r is a fixed interest rate (say, $r = 0.05$ for 5% interest), t is the number of payment intervals elapsed, and $F(t)$ is the accumulated value of the instrument at time t. This equation is not easily solved for r.

Write a script or function that finds r when $P = 200$, $t = 30$, and F takes the values $10000, 15000, \ldots, 40000$.