Nested Functions

What Are Nested Functions?

A nested function is a function that is completely contained within a parent function. Any function in a program file can include a nested function.

For example, this function named `parent` contains a nested function named `nestedfx`:

```
function parent
disp('This is the parent function')
nestedfx

   function nestedfx
      disp('This is the nested function')
   end

end
```

The primary difference between nested functions and other types of functions is that they can access and modify variables that are defined in their parent functions. As a result:

- Nested functions can use variables that are not explicitly passed as input arguments.
- In a parent function, you can create a handle to a nested function that contains the data necessary to run the nested function.
-

Requirements for Nested Functions

- Typically, functions do not require an `end` statement. However, to nest any function in a program file, *all* functions in that file must use an `end` statement.
- You cannot define a nested function inside any of the MATLAB® program control statements, such as `if/elseif/else`, `switch/case`, `for`, `while`, or `try/catch`.
- You must call a nested function either directly by name (without using `feval`), or using a function handle that you created using the `@` operator (and not `str2func`).
- All of the variables in nested functions or the functions that contain them must be explicitly defined. That is, you cannot call a function or script that assigns values to variables unless those variables already exist in the function workspace. (For more information, see Variables in Nested and Anonymous Functions.)

Sharing Variables Between Parent and Nested Functions

In general, variables in one function workspace are not available to other functions. However, nested functions can access and modify variables in the workspaces of the functions that contain them.

This means that both a nested function and a function that contains it can modify the same variable without passing that variable as an argument. For example, in each of these functions, `main1` and `main2`, both the main function and the nested function can access variable `x`:

```
function main1
x = 5;
nestfun1

   function nestfun1
     x = x + 1;
   end

end
```

```
function main2
nestfun2

   function nestfun2
      x = 5;
   end

x = x + 1;
end
```

When parent functions do not use a given variable, the variable remains local to the nested function. For example, in this function named `main`, the two nested functions have their own versions of `x` that cannot interact with each other:

```
function main
   nestedfun1
   nestedfun2

   function nestedfun1
      x = 1;
   end

   function nestedfun2
      x = 2;
   end
end
```

Functions that return output arguments have variables for the outputs in their workspace. However, parent functions only have variables for the output of nested functions if they explicitly request them. For example, this function `parentfun` does *not* have variable `y` in its workspace:

```
function parentfun
x = 5;
nestfun;

   function y = nestfun
     y = x + 1;
   end

end
```

If you modify the code as follows, variable `z` is in the workspace of `parentfun`:

```
function parentfun
x = 5;
z = nestfun;

   function y = nestfun
     y = x + 1;
   end

end
```

Using Handles to Store Function Parameters

Nested functions can use variables from three sources:

- Input arguments
- Variables defined within the nested function
- Variables defined in a parent function, also called *externally scoped* variables

When you create a function handle for a nested function, that handle stores not only the name of the function, but also the values of externally scoped variables.

For example, create a function in a file named `makeParabola.m`. This function accepts several polynomial coefficients, and returns a handle to a nested function that calculates the value of that polynomial.

```
function p = makeParabola(a,b,c)
p = @parabola;

   function y = parabola(x)
   y = a*x.^2 + b*x + c;
   end

end
```

The `makeParabola` function returns a handle to the `parabola` function that includes values for coefficients `a`, `b`, and `c`.

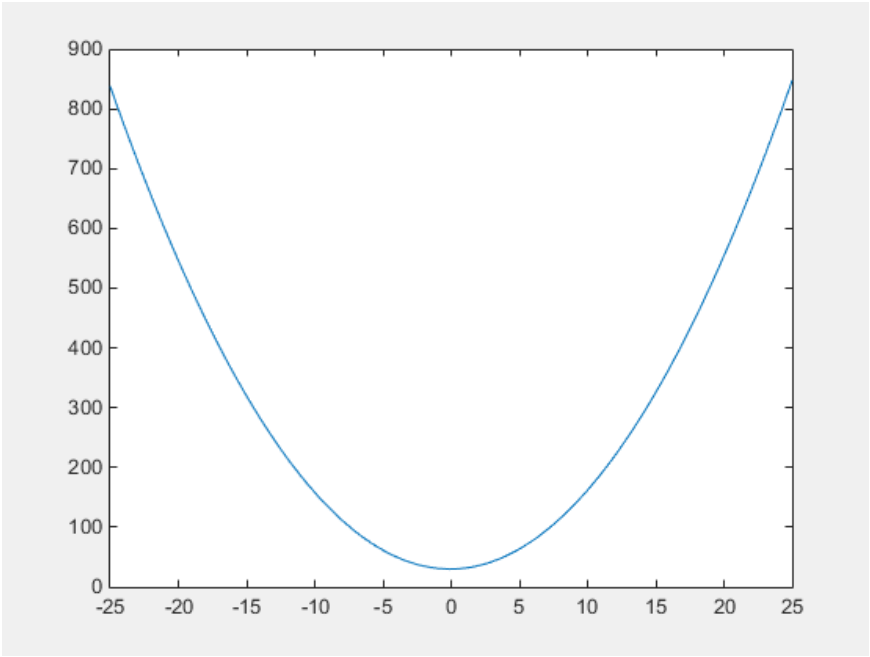
At the command line, call the `makeParabola` function with coefficient values of `1.3`, `.2`, and `30`. Use the returned function handle `p` to evaluate the polynomial at a particular point:

```
p = makeParabola(1.3,.2,30);

X = 25;
Y = p(X)
Y =
  847.5000
```

Many MATLAB functions accept function handle inputs to evaluate functions over a range of values. For example, plot the parabolic equation from `-25` to `+25`:

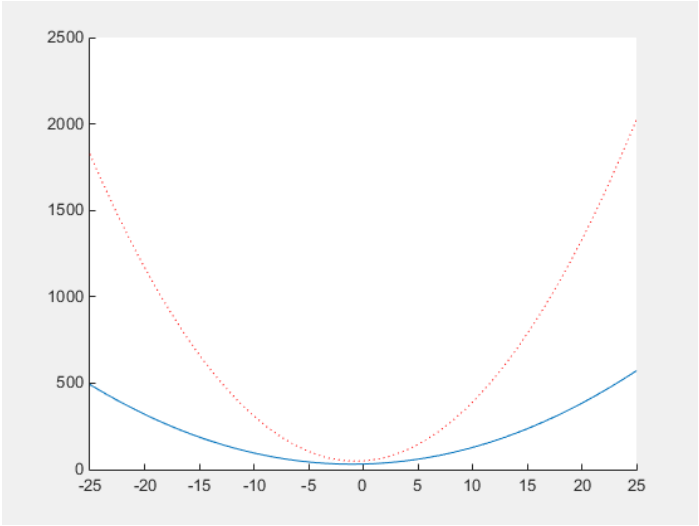
```
fplot(p,[-25,25])
```



You can create multiple handles to the `parabola` function that each use different polynomial coefficients:

```
firstp = makeParabola(0.8,1.6,32);
secondp = makeParabola(3,4,50);
range = [-25,25];

figure
hold on
fplot(firstp,range)
fplot(secondp,range,'r:')
hold off
```



Visibility of Nested Functions

Parameterizing Functions

Overview

This topic explains how to store or access extra parameters for mathematical functions that you pass to MATLAB® *function functions*, such as `fzero` or `integral`.

MATLAB function functions evaluate mathematical expressions over a range of values. They are called function functions because they are functions that accept a function handle (a pointer to a function) as an input. Each of these functions expects that your objective function has a specific number of input variables. For example, `fzero` and `integral` accept handles to functions that have exactly one input variable.

Suppose you want to find the zero of the cubic polynomial $x^3 + bx + c$ for different values of the coefficients b and c. Although you could create a function that accepts three input variables (x, b, and c), you cannot pass a function handle that requires all three of those inputs to `fzero`. However, you can take advantage of properties of anonymous or nested functions to define values for additional inputs.

Parameterizing Using Nested Functions

One approach for defining parameters is to use a *nested function*—a function completely contained within another function in a program file. For this example, create a file named`findzero.m` that contains a parent function `findzero` and a nested function `poly`:

```
function y = findzero(b,c,x0)

y = fzero(@poly,x0);

   function y = poly(x)
   y = x^3 + b*x + c;
   end
end
```

The nested function defines the cubic polynomial with one input variable, `x`. The parent function accepts the parameters `b` and `c` as input values. The reason to nest `poly` within`findzero` is that nested functions share the workspace of their parent functions. Therefore, the `poly` function can access the values of `b` and `c` that you pass to `findzero`.

To find a zero of the polynomial with `b = 2` and `c = 3.5`, using the starting point `x0 = 0`, you can call `findzero` from the command line:

```
x = findzero(2,3.5,0)

x =
   -1.0945
```

Parameterizing Using Anonymous Functions

Another approach for accessing extra parameters is to use an *anonymous function*. Anonymous functions are functions that you can define in a single command, without creating a separate program file. They can use any variables that are available in the current workspace.

For example, create a handle to an anonymous function that describes the cubic polynomial, and find the zero:

```
b = 2;
c = 3.5;
cubicpoly = @(x) x^3 + b*x + c;
x = fzero(cubicpoly,0)
```

```
x =
   -1.0945
```

Variable `cubicpoly` is a function handle for an anonymous function that has one input, `x`. Inputs for anonymous functions appear in parentheses immediately following the `@` symbol that creates the function handle. Because `b` and `c` are in the workspace when you create `cubicpoly`, the anonymous function does not require inputs for those coefficients.

You do not need to create an intermediate variable, `cubicpoly`, for the anonymous function. Instead, you can include the entire definition of the function handle within the call to`fzero`:

```
b = 2;
c = 3.5;
x = fzero(@(x) x^3 + b*x + c,0)
```

```
x =
   -1.0945
```

You also can use anonymous functions to call more complicated objective functions that you define in a function file. For example, suppose you have a file named `cubicpoly.m`with this function definition:

```
function y = cubicpoly(x,b,c)
y = x^3 + b*x + c;
end
```

At the command line, define `b` and `c`, and then call `fzero` with an anonymous function that invokes `cubicpoly`:

```
b = 2;
c = 3.5;
x = fzero(@(x) cubicpoly(x,b,c),0)
```

```
x =
   -1.0945
```