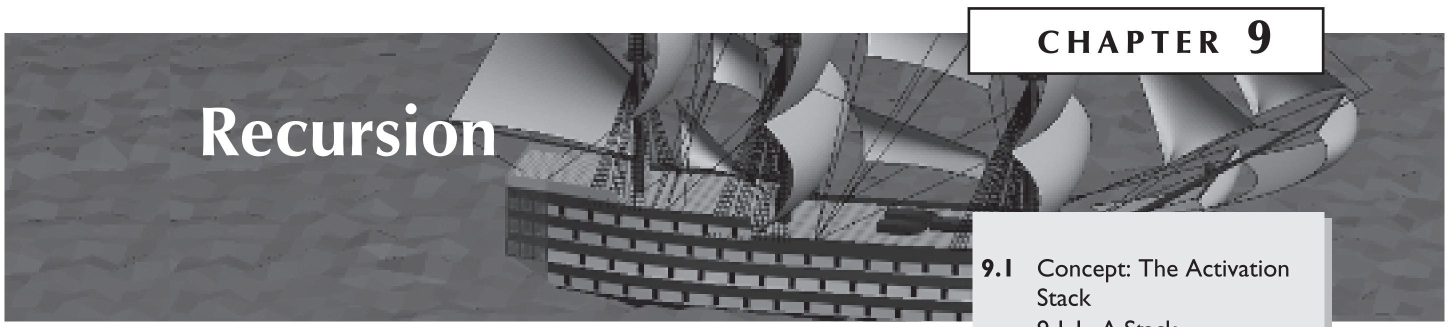
72 DPI M09_SMIT8708_03_SE_C09.indd Page 185 02/02/12 4:57 PM user-f396 F-396 EQA

CHAPTER 9



Recursion

Chapter Objectives

This chapter discusses the following basic ideas of recursive programming:

- Three basic characteristics must be present for a recursive function to work
- Exceptions are a powerful mechanism for detecting and trapping errors
- A wrapper function is used to set up the recursion
- Other forms of recursion occur in special circumstances

Introduction

Recursion is an alternative technique by which a code block can be repeated in a controlled manner. In Chapter 4, we saw repetition achieved by inserting control statements in the code (either `for` or `while`) to determine how many times a code block would be repeated. Recursion uses the basic mechanism for invoking functions to manage the repetition of a block of code.

While some problems are naturally solved by iterative solutions, there are many problems for which a recursive solution is elegant and easily understood. Frequently, a recursive function needs a "wrapper function" to set up the recursion correctly, and to check for erroneous initial conditions that might cause errors. The actual recursive function then becomes a private helper function.

9.1 Concept: The Activation Stack

In order to understand recursive programming, we must look deeper into the mechanism by which function calls are mechanized. Calling any function depends on a special kind of data structure built into the architecture of the central processing unit (CPU). This is called the **activation stack.** It enables the CPU to determine which functions are active or suspended awaiting the completion of other function calls. To understand the activation stack, first we consider the basic concept of a stack.

9.1.1 A Stack

A stack is one of the fundamental data structures of computer science. It is best modeled by considering the trays at the front of the cafeteria line. You cannot see how many trays there are on the stack, and the only access you have to them is to take a tray off the stack or put one on. So a stack is a collection of objects of arbitrary size with a restricted number of operations we are allowed to perform on that collection (see Figure 9.1). Unlike a vector, where it is permissible to read, add, or remove items anywhere in the collection, we are only allowed the following operations with a stack:

- Push an object onto the stack
- Pop an object off the stack
- Peek at the top object without removing it
- Check whether the stack is empty

9.1.2 Activation Stack

The core concept that enables any function (especially a recursive function) to operate is the concept of an activation stack. The activation stack is the means by which the operating system allocates memory to functions for

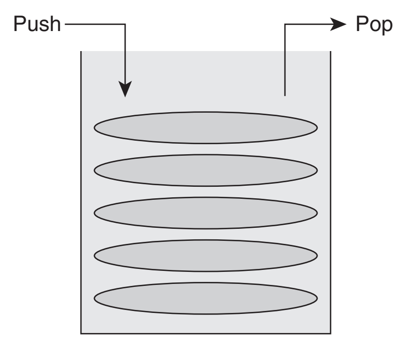


Figure 9.1 *Behavior of a stack*

local storage. Typically, local storage is required by a function for the following reasons:

- Storing the location in memory to which control must be returned when the function execution completes
- Storing copies of the function parameter values
- Providing space for the values of any local variables defined within the function

When MATLAB is initializing, the operating system allocates a block of memory to contain its activation stack and allocates the first item (usually called a "frame") on the activation stack to store variables defined in the Command window and by scripts. An astute reader might recognize this as the initial workspace for the system. When the user starts a script or makes an entry in the Command window, any variables created are stored in that **stack frame**. When that application calls a function, a new stack frame is allocated and "pushed" onto the activation stack. The calling program is then suspended, actual parameters are copied to formal parameters in the new workspace and control is passed to the function. Any new variables created are stored in its stack frame. When that function completes, its frame is popped off the stack and destroyed, and control is returned to the frame beneath, which is now the top of the stack. If an active function calls another function, this process is repeated. The calling function is suspended, a stack frame is pushed onto the activation stack for the new function, and the original function is suspended until the new function completes.

Technical Insight 9.1

In most computer languages, user programs and functions are compiled before they can be run. Part of that compilation process is defining the variable names and data types. This allows the system processes to compute the exact size of each stack frame before the program begins to run. Since the MATLAB language is interpreted and interactive, this information is not available. Consequently, every stack frame must be dynamically sized to allow for the "surprises" inherent in this style of programming.

9.1.3 Function Instances

In Chapter 2 we discussed the difference between the type of data defined by its class and an object—an instance of that class assigned to a variable. In the same way, we draw the distinction between the `.m` file that defines the behavior of a function and the instance(s) of that function that results when the function is called. Each new instance of a function has its own workspace that occupies a temporary stack frame allocated from the activation stack.

9.2 Recursion Defined

Following the previous line of reasoning, in principle there is no reason why a function could not in fact "call itself," and this is the logical basis for recursive programming. Of course, as with iterative programming, if there is no mechanism to stop the recursion, the process would repeat endlessly. In the case of endless recursion, since space is being consumed on the activation stack, the system will eventually terminate the process when the memory originally allocated for the activation stack is exhausted.

The canonical illustration of recursion is the computation of `n` factorial. We could view the calculation of `5!` in the following ways:

```
5! = 5 × 4 × 3 × 2 × 1
5! = 5 × 4!
```

The second representation is the recursive view, which warrants a closer examination as follows:

```
n! = n × (n-1)!
```

This definition would not be complete, however, without realizing that it must stop somewhere. In the original definition above, we did not continue the chain of multiplication with `" * 0 * (-1) * (-2) ..."` for obvious reasons—multiplying by 0 makes all factorial values 0! Mathematically, we "artificially" define the terminating condition for the factorial calculation as the state where `0! = 1`.

We can derive from this example the three necessary characteristics of a recursive function:

1. There must be a terminating condition to stop the process
2. The function must call a ***clone*** of itself
3. The parameters to that call must move the function toward the terminating condition

The word *clone* is important here—a recursive function really does not "call itself," because it requests a new stack frame and passes different parameters to the instance of the function that occupies the new frame.

9.3 Implementing a Recursive Function

Template 9.1 shows the general template for recursive functions. The following general guidelines indicate how the recursive template is implemented:

- The `<function_name>`, like the name of any other function, may be any legal variable name
- The variable `<result>` may be any legal variable name or a vector of variable names
- As usual with functions, you should supply at least one line of `<documentation>` to define its purpose and implementation

Template 9.1 General template for a recursive function

```
function <result> = <function_name> (<formal_params>)
<documentation>
  if <terminating condition 1>
     <result> = <initial value 1>
  elseif <terminating condition 2>
     <result> = <initial value 2>
      ...
  else
     <result> = <operation> ...
         (<formal_params>, ...
          <function_name> (<new_params>))
  end
```

- Each exit from the function must assign values to all the result variables
- The first design decision is to determine the condition(s) under which the recursive process should stop, and how to express this as the `<terminating condition N>` tests
- The `<initial value N>` entries are the value(s) of the result(s) at the terminating condition(s)
- The second design decision is to determine the `<operation>`—the specific mathematical or logical operation that must be performed to combine the current formal parameters with the result of the recursive call to create a new value of the `<result>`
- The last design decision is to determine how to compute the `<actual_params>` of the recursive call to ensure that the process moves toward at least one of the `<terminating condition N>` states

The implementation of the factorial function is shown in Listing 9.1.

In Listing 9.1:

Before Line 2, we show a diagnostic print call that, if not commented, enables you to observe the sequence of events.

Line 2: The terminating condition.

Listing 9.1 Function to compute N factorial

```
1. function result = fact(N)
   % recursive computation of N!
   % fprintf('fact( %d )\n', N); % testing only
2.     if N == 0
3.         result = 1;
4.     else
5.         result = N * fact(N - 1);
6. end
7.     end
```

Exercise 9.1 Analyzing recursive behavior

1. Create the `fact(...)` function from Listing 9.1, remove the first '%' from Line 3 to enable the printout, and run it from the Command window:

```
>> fact(4)
fact( 4 )
fact( 3 )
fact( 2 )
fact( 1 )
fact( 0 )
ans =
    24
```

2. Put a break point at Line 4 and run `fact(2)`. The function should pause in the first stack frame. Notice that the only variable in the workspace is `N` with a value `2`.
3. Find the "step into" button and click it. Since `N` is not `0`, the arrow should move to Line 7.
4. Click again, and the workspace should change to a new workspace with the value `N = 1`— you just called a clone of the original function with its own stack frame. There should be a second, transparent arrow at Line 7 to indicate that some clone of this function is waiting at that point for a result.
5. Continue stepping into functions until you return from the copy where `N = 0`. When this return happens, you return to the frame with `N = 1`, the frame "underneath,"at Line 7, and are then able to compute the first result.
6. Further stepping will return from each stack frame until you finally return to your script's workspace with the final answer.

Line 5: The result at termination.

Line 7: Calls a clone of the function, which moves closer to termination by reducing `N` and computing the result.

Exercise 9.1 provides an analysis of recursive behavior. In particular, notice that all the mathematical operations are performed as the activation stack "unwinds."

9.4 Exceptions

We digress here to discuss how programs deal with unexpected circumstances. Exceptions are a powerful tool for gracefully managing run-time errors caused by programming errors or bad data. The general need for an exception mechanism might best be established by way of an example.

Suppose you write a program that requests some data from a user and then launches a significant number of nested function calls—perhaps even a recursive function—to perform an analysis on the data received. Somewhere in the depths of these function calls, the program divides something by a value, but in this instance that value is zero. The cause of this problem is probably bad data entered by the user in the top-level script. However, the effect is discovered deep in the activation stack in the middle of some obscure numerical computation.

9.4.1 Historical Approaches

Early programming languages attempted to deal with this problem in one of two equally unpleasant ways:

- Some languages require any mathematical function that might produce an error to return the status of that calculation to the calling function. They allow errors to be reported and processed, but they have two unpleasant consequences: using up the ability of a function to return a value and calling this function, which means choosing between testing for errors and solving the problem locally and passing the error condition back to its calling function in the hope that somewhere the error will be dealt with.
- Perhaps worse than this are the languages that use a globally accessible variable, such as `ierror`, to report status. For example, if `ierror` were normally set to `0`, an error could be announced by setting its value to something other than `0` to indicate the nature of the failure. This frees the function from needing to return status, but it does not relieve the calling function of the need to check whether the `ierror` value is bad, or solving the problem, or elevating it. Furthermore, if an error does occur within a function, since it is now still returning a value, what value should it return if it is unable to complete its assigned calculation?

9.4.2 Generic Exception Implementation

By contrast, most modern programming languages provide an exception mechanism whereby if an error occurs, regardless of how deep in the activation stack, program implementation is immediately suspended in the current stack frame. The activation stack below this frame is then searched for the frame of a program that has "volunteered" to process this type of exception. When it is found, all the stack frames above this frame are removed from the stack and the code in the exception handling mechanism is activated. If no such frame is discovered, the overall program aborts with an error code.

The following mechanisms are necessary to implement the exception mechanism effectively:

- *Throwing an exception.* Whenever a problem occurs, the operating system must suspend operations at that point in the activation stack and go looking for a function equipped to handle the specific exception. If no such function is found, the program is terminated and an exception is shown to the user (in MATLAB, it is written in red in the Command window).
- *Catching an exception.* A function that is able to deal with a specific exception uses a `try ... catch` construct to identify the suspect code and resolve the problem. Between `try` and `catch`, the programmer puts a code block that contains activities that could throw exceptions. After the `catch` statement, there is a code block that should fix the problem.

Depending on the specific language implementation, the exception-catching mechanism usually offers facilities both for determining exactly where the exception occurred and for reconstructing the activation stack with all the variable values as they were at the time of the exception.

In the previous example, the general template for successfully interacting with the user is shown in Template 9.2. The successful Boolean flag will be set only if the data are processed without error. It does not matter how deep in the data processing code the error occurs—the user interface catches the error, reports it to the user, and prompts the user for better data.

For example, you might have noticed earlier that the input(...) function has a built-in `try ... catch` mechanism to deal with erroneous user input. If something is entered that cannot be parsed, rather than throw red ink in the Command window, the exception is caught and the prompt repeated for the user.

Template 9.2 General template for processing exceptions

```
successful = false
while <not successful>
   try
       <request data from the user>
       <process the data>
       successful = true
   catch
       <announce the error to the user>
   end
end
```

9.4.3 MATLAB Implementation

MATLAB implements a simplified version of the most general form of exception processing. The `try ... catch ... end` construct is fully supported. However, unlike some languages, the MATLAB language does not distinguish between the kinds of exception that can be thrown.

- All built-in functions throw exceptions when they discover error conditions—attempting to open a nonexistent file for reading, for example—and expect the programmer to catch these exceptions if they are recoverable.
- To throw an exception manually, the program calls the `error(...)` function that takes one parameter, a string defining the error. If the exception is not caught, the string provided is displayed in red to the user. If the exception is caught, that string is ignored.
- To handle an exception, a code block we suspect might throw an exception is placed between `try` and `catch` statements. If no error occurs in the code block, the `catch` statement is ignored. If an exception is thrown from that code block, however, execution is suspended at that point. No further processing is performed, no data are returned from functions, and the code in the closest `catch` block is executed up to the associated `end` statement. To determine the cause of the exception, you can use the `lasterror` function. It returns the textual information provided at the exception and a structure array describing the activation stack.
- In more complex situations where this function may not be able to actually handle the error, a further exception can be thrown from the `catch` block. This exception will escape from this `try ... catch` block and must be caught (if at all) by another function or script deeper in the activation stack.

Listing 9.2 illustrates a simple example. The objective is to have the user define a triangle by entering a vector of three sides and to calculate the angle between the first two sides. The `acosd(...)` function computes the inverse cosine of a ratio. If that ratio is greater than one, there is something seriously wrong with the triangle, and `acosd` returns a complex number. This script detects that the answer is complex and throws an exception.

In Listing 9.2:

Lines 1 and 2: We will repeat the attempts to compute the angle of a triangle until successful.

Line 3: Begins the suspect code.

Line 8: Detects the problem with the data.

Listing 9.2 MATLAB script using exception processing

```
 1. OK = false;
 2. while ~OK
 3.     try
 4.         side = input('enter a triangle: ');
 5.         a = side(1); b = side(2); c = side(3);
 6.         cosC = (c^2 - a^2 - b^2)/(2 * a * b);
 7.         angle = acosd(cosC);
 8.         if imag(angle) ~= 0
 9.             error('bad triangle')
10.         end
11.     catch
12.         disp('bad triangle - try again')
13.     end
14.     OK = true;
15. end
16. fprintf('the angle is %f\n', angle)
```

Line 9: Throws the exception. In this case, the exception occurs visibly in this script. However, the `try ... catch` behavior is the same if the exception occurs deep in a set of nested function calls.

Line 11: The end of the suspect code block and the beginning of the exception handler—in this case, it's a warning to the user that the data are bad.

Line 14: This line is reached only if the suspect code block executed correctly, in which case we can exit the `while` loop.

You have an opportunity to work with exception processing in Exercise 9.2.

Style Points 9.1

1. You should allow the exception-processing mechanism to simplify the structure of your code. Rather than attempting to detect every possible data error and return error condition, perhaps from deeply nested function calls, allow the exception mechanism to return control directly to the code that can deal with the problem.

2. Exception processing is for processing events that occur outside the normal thread of execution. It may be tempting at times to use the exception mechanism as a clever means of changing the normal flow of program control, but resist that temptation. It produces ugly, untraceable code and should be avoided.

Exercise 9.2 Processing exceptions

Put the code from Listing 9.2 in a script and execute it, using the following data:

```
enter a triangle: [3 4 8]
bad triangle - try again
enter a triangle: [3 4 6]
the angle is 62.720387
```

Then, edit the script to remove the `try` statement and the `catch` block and repeat the test.

9.5 Wrapper Functions

Consider the factorial function again for a moment—specifically, ask how you would deal with a user who accidentally called for the factorial of a negative number or of a number containing a fractional part. Our original recursive `fact(...)` function is not protected from these programmer errors. There are three possible strategies for dealing with this situation:

1. *The legalist approach* ignores the bad values, lets the user's program die, and then responds to user complaints by pointing out that the documentation clearly indicates that you should not call for the factorial of a negative number. Usually this is not the best approach from the customer relations viewpoint or from the technical support effort viewpoint, especially since recursive code that hangs up typically crashes with a stack overflow—not the easiest symptom to diagnose!
2. *In-line coding* builds into the code a test for N less than zero (or fractional) and throws an exception with a meaningful error message. Although this is an improvement over the first choice because it exits gracefully, the test is in a bad place. The function is recursive; therefore, the code for that test is repeated as many times as the function is called. While modern computers are fast enough that one would probably not notice the difference, in general this is a poor implementation that punishes those who are using the function correctly with the same test each time the recursive function is called.
3. *A wrapper function* is the best solution. A wrapper function is called once to perform any tests or setup that the recursion requires and then to call the recursive function as a helper to the main function call. While there is a small computational cost to using a wrapper, it is only executed once rather than each time the recursive function is called. Template 9.3 illustrates this idea.

The first function named `<function_name>` is actually the wrapper function with the return result, parameters, and documentation expected by the caller. It makes whatever tests are necessary to validate the input data, cleans it up if necessary, and calls the helper function named `<private_name>`.

Listing 9.3 is the implementation of the factorial function with protection from bad data.

In Listing 9.3:

Line 1: To the outside world, this is the function actually called. (Ugly secret: even if the name is not the same name as the file, the first function in the file is always executed first.)

Line 2: Checks for negative and fractional inputs.

Template 9.3 General template for a wrapper function

```
function <result> = <function_name> (<formal_params>)
<documentation>
      if <bad_condition>
         <throw exception>
      else
         <result> = <private_name> (<actual_params>)
      end

function <result> = <private_name> (<formal_params>)
<documentation>
  if <terminating condition 1>
     <result> = <initial value 1>
  elseif <terminating condition 2>
     <result> = <initial value 2>
  else
     <result> = <operation>
         (<formal_params>, ...
          <private_name> (<new_params>) )
  end
```

Listing 9.3 Wrapper implementation for the factorial function

```
 1. function result = fact(N)
    % computation of N!
 2.     if (N < 0) || ((N - floor(N)) > 0)
 3.         error('bad parameter for fact');
 4.     else
 5.         result = local_fact(N);
 6.     end
 7. end
 8. function result = local_fact(N)
    % recursive computation of N!
 9.     fprintf('fact( %d )\n', N);
10.     if N == 0
11.         result = 1;
12.     else
13.         result = N * local_fact(N - 1);
14. end
15.     end
```

Line 4: Throws an exception if the data are bad.

Line 6: Calls the recursive version if the data are valid.

Line 9: Definition of the recursive function. By convention, some MATLAB users tend to give the prefix `local_` to private functions like this, but this has no significance to the system.

Line 15: Calls the local recursive function—it is not necessary to go back to the wrapper by calling `fact(..)` here.

Exercise 9.3 gives you an opportunity to work with the protected factorial.

Exercise 9.3 Writing the protected factorial

Write the `fact(...)` function as shown in Listing 9.3, and test it in the Command window:

```
>> fact(-1)
??? Error using ==> fact
bad parameter for fact
>> fact(.5)
??? Error using ==> fact
bad parameter for fact
>> fact(4)
ans =
    24
```

9.6 Examples of Recursion

We conclude this chapter with three examples of recursive programming: detecting palindromes, computing the Fibonacci series of numbers, and finding zeros of a function. The examples are followed by a practical engineering example of the use of zero finding.

9.6.1 Detecting Palindromes

We might want to determine whether a word or phrase received as a string is a palindrome, that is, whether it is spelled the same forward and backward. Of course, you could accomplish this in one line with vector operations (think about it!) but that would not be a good recursive exercise. One could design a recursive function named `isPal(<string>)` as follows:

- The function `isPal(<string>)` terminates if the `<string>` has zero or one character, returning `true`.
- It also terminates if the first and last characters are not equal, returning `false`.
- Otherwise (first and last are equal), the function returns `isPal(<shorter string>)`, where the shorter string is obtained by removing the first and last characters of the original string.
- Clearly, since the string is always being shortened, the recursive solution is approaching the terminating condition.

The MATLAB implementation of the palindrome detector is shown in Listing 9.4.

In Listing 9.4:

Line 3: The successful terminating condition is when the length of the string is under 2.

Listing 9.4 Recursive palindrome detector

```
1. function ans = isPal(str)
   % recursive palindrome detector
2.     if length(str) < 2
3.         ans = true;
4.     elseif str(1) ~= str(end)
5.         ans = false;
6.     else
7.         ans = isPal(str(2:end-1));
8.     end
9. end
```

Line 5: The failure condition is when the first and last characters do not match.

Line 8: To move toward termination, remove the first and last characters that have already been checked.

We should observe further that a serious student of palindromes might know that real palindromes contain spaces, punctuation marks, and uppercase and lowercase characters. We leave it as an exercise for you to write a wrapper function that cleans up strings containing these issues before passing the string to the recursive palindrome detector.

9.6.2 Fibonacci Series

The Fibonacci series was originally named for the Italian mathematician Leonardo Pisano Fibonacci, who was studying the growth of rabbit populations in the eleventh century. He hypothesized that rabbits mature one month after birth, after which time each pair would produce a new pair of rabbits each month. Starting with a pair of newborn rabbits free in a field, he wanted to calculate the rabbit population after a year. Figure 9.2 illustrates the calculation for the first six months, counting rabbit pairs. It

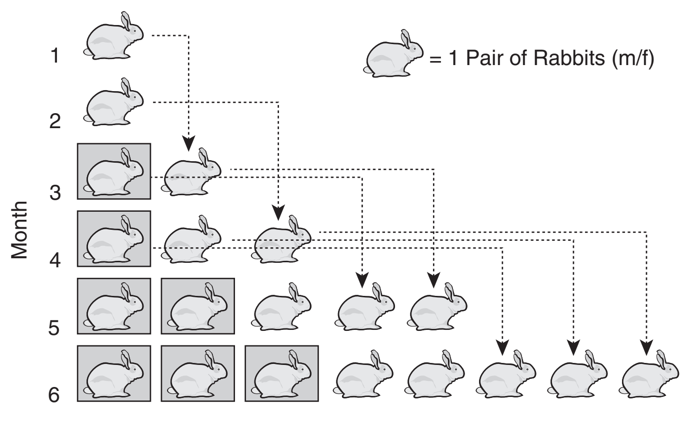


Figure 9.2 *Computing rabbit populations*

Technical Insight 9.2

Truthfulness requires pointing out that while computing the Fibonacci series recursively is a very nice, conceptually simple approach, it is a nightmare as far as the computational load on your processor. Do not try to compute beyond about 27 numbers in the series. An iterative solution, while less elegant, runs in linear time rather than exponential.

soon becomes clear that the number of rabbits in month `N` comprises the number in month `N-1` (since in this ideal example, none of them die) plus the new rabbits born to the mature pairs (shown in boxes in the figure). Since the rabbits mature after a month, the number of mature pairs that produce a new pair is the number of rabbits in the month before, `N-2`. So the algorithm for computing the population of pairs after `N` months, `fib(N)`, is recursive:

- There is a terminating condition: when `N = 1` or `N = 2`, the answer is `1`
- The recursive condition is: `fib(N) = fib(N-1) + fib(N-2)`
- The solution is moving toward the terminating condition, since as long as `N` is a positive integer, computing `N-1` and `N-2` will move toward `1` or `2`.

The implementation of the Fibonacci function is shown in Listing 9.5.

The algorithm produces the Fibonacci series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, . . ., giving a population after a year of 144.

A closely related phenomenon is the golden ratio or golden number computed as the limit of the ratio of successive Fibonacci series values—approximately 1.618034—that has been found to occur in nature. To the surprise of naturalists, this series of numbers occurs in nature in a remarkable number of circumstances. Consider Figure 9.3 for example, where a set of squares placed side by side in a rotating sequence is drawn using the Fibonacci series for the size of each square. The resulting geometric figure is a close approximation to the logarithmic spiral so frequently found in nature, such as the nautilus shell pictured in the figure.

9.6.3 Zeros of a Function

Frequently we need to solve nonlinear equations by seeking the values of the independent variable that produced a zero result. There are a number of well-known numerical techniques for achieving this goal. We will examine

Listing 9.5 The Fibonacci function

```
function result = fib(N)
% recursive computation the Nth Fibonacci number
    if N == 1 || N == 2
        result = 1;
    else
        result = fib(N-1) + fib(N-2);
    end
end
```

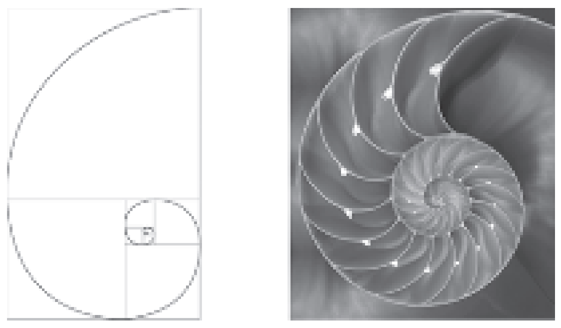


Figure 9.3 *Fibonacci in nature*

a recursive approach to determining the zeros of functions. However, especially when there are multiple zero crossings, it is very helpful to have a good initial estimate of the location(s) of the crossing(s). As an example, consider a function *f* (*x*). We will use the function given by:

$$f(x) = \mathbf{0.0333}x^6 - \mathbf{0.3}x^5 - \mathbf{1.3333}x^4 + \mathbf{16}x^3 - \mathbf{187.2}x + \mathbf{172.9}$$

as plotted in Figure 9.4. However, this algorithm will work for any function of *x*. We assume that the continuous line describes the exact function, and the plus marks indicate locations for which we have measurements. Clearly, there are a number of zero crossings of this function, including a very messy looking crossing at around = 6.

We will find the exact value of one of the zeros of this function by first estimating the zero crossings and then by using a recursive technique for refining a better estimate to arbitrary levels of accuracy.

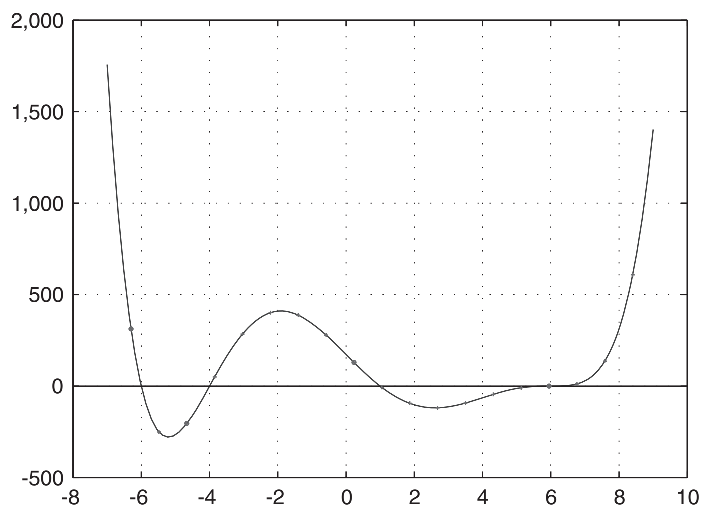


Figure 9.4 *A function f(x)*

Estimating Critical Points of a Function First, we need to compute an approximation to the roots of this equation. These approximations will be found by finding the *x* values at which adjacent values of the function change sign. The technique for determining where adjacent points change sign is simply to multiply adjacent values of *f*(*x*) and find where that product is not positive, as shown in Listing 9.6.

In Listing 9.6:

Line 1: Establishes samples of *x*.

Line 2: Computes $y = f(x)$.

Line 3: Detects the indices where the zero crossings occur by shifting *y* to the right by one and by shortening the original by one to keep the vector size equal.

Lines 4 and 5: Display the zero crossing estimates.

Line 6: Calls the recursive function to find the third zero crossing that estimates one root of this equation. Clearly, one could iterate here to find all of the roots.

Listing 9.6 produces the following results, which can be verified by observing the circled data points shown in Figure 9.4:

```
zeros occur just after
ans =
   -6.3000    -4.6667    0.2333    5.9500
```

Having observed these results, we decide to compute the exact value of the first positive root, occurring at the third crossing.

Recursive Refinement of the Estimate The recursive function to find the third root of *f*(*x*) works on the principle of binary division. It consumes a vector of adjacent values of *x* that are guaranteed to have values of `f(x)` of opposite sign. The fundamental features of the recursive solution are as follows:

- The terminating condition is when the two `x` values are within acceptable error—in this case, 0.001
- Otherwise, we find the middle of this `x` range, `mx`, find its `f(mx)`, and then make the recursive call either with `[x(1) mx]` or `[mx x(2)]`, depending on the sign of `f(mx) × f(x(1))`
- This will always converge because each recursive call halves the distance between the *x* limits.

Listing 9.6 Initial zero crossings

```
1. px = linspace(-6.3, 8.4, 19);
2. py = f(px);
3. zeros = find(py(1:end-1) .* py(2:end) <= 0)
4. disp('zeros occur just after')
5. px(zeros)
6. root = findZero([px(zeros(3)) px(zeros(3)+1)])
```

Listing 9.7 Recursive root finding

```
 1. function pt = findZero(x)
    % x is a lower-upper pair guaranteed to have
    % y values of opposite sign
    % return the x coordinate of the root
 2.     if abs(x(1)-x(2)) < .001
 3.         pt = x(1);
 4.     else
 5.         mx = sum(x)/2;
 6.         my = f(mx);
 7.         if my*f(x(1)) <= 0
 8.             pt = findZero([x(1) mx]);
 9.         else
10.             pt = findZero([mx x(2)]);
11.         end
12.     end
13. end
```

In general, this method is a little slower than Newton's method, which uses the slope of $f(x)$ to compute the next estimate. However, it is very strong and somewhat immune from the instability suffered by Newton's method on undulating data. The function that solves this problem is shown in Listing 9.7.

In Listing 9.7:

Line 1: The function consumes a pair of x limits and produces the x root.

Lines 2–4: Check for the terminating condition and return the x root.

Lines 5 and 6: Calculate the x and y values of the midpoint of the x range.

Line 7: Checks the sign of the y value of the midpoint.

Line 8: If different from the sign at first limit, it makes the recursive call with the first limit and the midpoint.

Lines 9 and 10: Otherwise, it uses the range from the midpoint to the second limit.

This function computes the correct crossing at $x = 1.00$.

9.7 Engineering Example—Robot Arm Motion

Here we consider the problem of programming the arm of a robot to move in a straight line. Consider the arm shown in Figure 9.5. It consists of two jointed limbs of length r_1 and r_2 at angles α and β, respectively, to the horizontal.

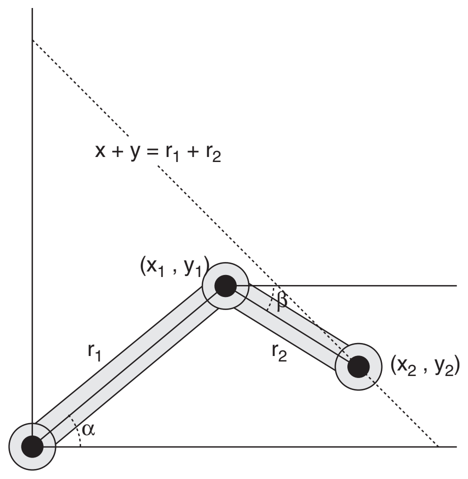


Figure 9.5 *The robot arm problem*

Overall Objective
The ultimate challenge of this situation is to calculate the sequence of values of α and β that will guide the end of the arm along the straight line:

$$x + y = r_1 + r_2 \tag{1}$$

However, this complete problem is more complex than necessary for this point in the text. First, we will address a necessary component of the problem.

Immediate Objective
The sub-problem we address here is to determine for a given value of α, the value β that will place the end of the arm at some place on the line. The algebra and trigonometry of this problem are quite simple. The position of the end of the arm, $[x_2\ y_2]$, is expressed as:

$$x_2 = r_1 \cos \alpha + r_2 \cos \beta \tag{2}$$

$$y_2 = r_1 \sin \alpha + r_2 \sin \beta \tag{3}$$

Combining these two relationships with Equation (1) gives the equation for $F(\beta)$, the difference between the end point derived from β and the straight line. We need to solve this for $F(\beta) = 0$:

$$F(\beta) = r_1 \cos \alpha + r_2 \cos \beta + r_1 \sin \alpha + r_2 \sin \beta - (r_1 + r_2) \tag{4}$$

If we are given values for r_1, r_2, and the angle α, we will use the method of Section 9.6.3 to find the value(s) of α that satisfies this equation. By inspecting Figure 9.5, we might expect two answers—one with a small negative value and one "bending backward" at an angle greater than 90°.

Figure 9.6 shows a plot of this function for $r_1 = 4$, $r_2 = 3$, and $\alpha = 30°$. The zero crossings of this function confirm our intuition that there are two values

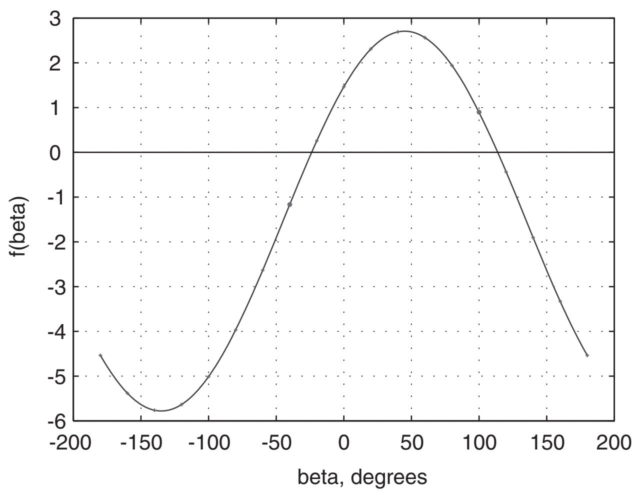


Figure 9.6 *The relationship between* β *and the value of F(*β*)*

of α that satisfy the equation for small, positive values of β: one around –30° and one around 110°.

The Solution to the Sub-problem

As before, since there is no analytical solution to this function, we will find the approximate location of the zero crossings and then use a recursive function to find the exact roots. The script that accomplishes this is shown in Listing 9.8.

In Listing 9.8:

> Lines 1–6: Establish the parameters of the problem as `global` variables to avoid the overhead cost of passing them into recursive functions.

Listing 9.8 Finding arm position

```
 1. global r1
 2. r1 = 4
 3. global r2
 4. r2 = 3
 5. global alpha
 6. alpha = pi/6 % 30 deg
 7. beta = linspace(-pi, pi, 19);
 8. pf = fab(beta);
 9. zeros = find(pf(1:end-1) .* pf(2:end) <= 0)
10. disp('zeros occur just after')
11. beta(zeros)
    %
12. zero = findZeroAB([beta(zeros(1)) ...
13.         beta(zeros(1)+1)])
```

Lines 7 and 8: Sample the possible range of β values with enough values to identify the zero crossings, and compute the corresponding values of F(β).

Lines 9–11: Estimate the zero locations by multiplying adjacent function values and display the results.

Line 12: Calls the recursive function to find the zero crossing.

Running this script produces the following:

```
r1 =
     4
r2 =
     3
alpha =
    0.5236
zeros =
     8 15
zeros occur just after
ans =
   -0.6981 1.7453
zero =
   -0.4152 -0.0009
```

The function for which we are seeking the zero is shown in Listing 9.9.

In Listing 9.9:

Lines 2–4: Gain access to the global parameters.

Lines 5–6: Compute the left-hand side of Equation (4).

The function that finds the zero crossings of `fab(beta)` is shown in Listing 9.10.

In Listing 9.10:

Line 2: Computes the *y* values corresponding to the *x* limits.

Lines 3 and 4: Check the terminating condition and return the `[x y]` coordinates of the result.

Lines 6 and 7: Find the *x* and *y* values of the midpoint.

Lines 8 and 9: If the midpoint is on the opposite side of the *x*-axis from the lower limit, make a recursive call using these limits.

Listing 9.9 Function for zeros

```
1. function res = fab(beta)
   % f(beta) = r1 (cos(alpha) + sin(alpha) - 1)
   %         + r2 (cos(beta) + sin(beta) - 1)
2. global r1
3. global r2
4. global alpha
5. res = r1 * (cos(alpha) + sin(alpha) - 1) ...
6.     + r2 * (cos(beta) + sin(beta) - 1);
```

Listing 9.10 Recursive zero finder

```
 1. function pt = findZeroAB(x)
    % x is a lower-upper pair guaranteed to have
    % y values of opposite sign
 2.     y = fab(x);
 3.     if abs(x(1)-x(2)) < .001
 4.         pt = [x(1) y(1)];
 5.     else
 6.         mx = sum(x)/2;
 7.         my = fab(mx);
 8.         if my*y(1) < 0
 9.             pt = findZeroAB([x(1) mx]);
10.         else
11.             pt = findZeroAB([mx x(2)]);
12.         end
13.     end
14. end
```

Line 11: Otherwise, makes the recursive call using the midpoint and the upper limit.

Reflection
A modest amount of code is all that is required to create an elegant solution to a nontrivial problem. The structure of the recursive function shown in Listing 9.8 clearly reflects the standard recursive template, and that function can be used to find zeros of any continuous function defined in `fab(x)`.

Chapter Summary

This chapter discussed the three basic principles of recursive programming that must be present for a recursive program to succeed:

- There must be a terminating condition
- The function must call a clone of itself
- The parameters of that clone must move the function toward the terminating condition

We have also seen some other important capabilities as follows:

- Exceptions are declared either within system functions or by the user using the `error(...)` function; they are trapped and perhaps remedied using `try ... catch` code blocks
- A wrapper function is used to set up a recursive solution by validating the incoming data

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`catch`	End of a suspect code block where the exception is trapped	9.4.3
`error(str)`	Announces an error with the string provided	9.4.3
`global <var>`	Defines the variable `<var>` as globally accessible	9.1
`lasterror`	Provides a structure describing the environment from which the last exception was thrown	9.4.3
`try`	Begins a block of suspect code from which an exception might be thrown	9.4.3

Self Test

Use the following questions to check your understanding of the material in this chapter:

True or False

1. We limit the functionality of a stack in order to protect the data from corruption.
2. The only way to remove a stack frame from the activation stack is to exit from the function instance hosted by that frame.
3. All the math operations in a recursive function are performed as the activation stack unwinds.
4. Exception processing can be used as a clever means of changing the normal flow of program control.
5. The name of the first function in a function definition m-file must match the name of the file.

Fill in the Blanks

1. Recursion is ____________ by which a code block can be repeated in a controlled manner.
2. Very frequently, a recursive function needs a(n) ________________ to set up the recursion correctly and to _________________ .
3. Exceptions are a powerful tool for managing ______________ caused by either ____________ or _____________ .

4. A wrapper function is called once to perform ________________ that the recursion requires, and then to call the recursive function ________________.

5. You can ________ one of the zeros of a function by first _______________, and then using a(n) ____________ for refining a better estimate to arbitrary levels of accuracy.

Programming Projects

1. For this problem, you will be required to write three functions: `recurSum`, `recurProd`, and `fibVector`. The first one will take in a vector and compute the sum of the elements of the vector. The second one will take in a vector and compute the product of the elements of the vector. The third one will take in a number, `N`, and return a vector containing the first `N` terms of the Fibonacci sequence. You must use recursion to complete these functions. You may not use `for` loops, `while` loops or the functions `sum`, `prod`, or `factorial`. Your function headers should be:

```
function ans = recurSum(arr)
function ans = recurProd(arr)
function vec = fibVector (num)
```

2. Write a recursive function called oddfact(n) that takes in a number and returns the factorial of the odd numbers between the given number and 1.

   For example:

```
oddfact(4) returns 3
oddfact(9) returns 945 = 9*7*5*3*1
```

3. Consider the problem of structures with nested fields.
   a. Write a function called `tracker` that takes in a structure and returns the number of levels at which it has a field called `'Inner'`. Each of these fields can also be structures having a field called `'Inner'`, but at each level there can be only one field called `'Inner'`. The innermost structure will not contain a field called `'Inner'`. You must use recursion. Hint: use the `isfield(...)` function. Your function header should be:

```
function num = tracker(astruct)
```

   b. Create a structure with at least three levels of recurring fields, and use it to test your tracker function.

4. Create a recursive function with a wrapper to protect it from illegal values. The function name should be `recursiveFib`. It should take in

a number `n` and return the `n`th Fibonacci number. You should ensure that `n` is a non-negative integer, and announce an error if that is not the case.

Fibonacci numbers are defined as:

```
F(n) = 0                    if n = 0
F(n) = 1                    if n = 1
F(n) = F(n-1) + F(n-2)      otherwise.
```

This produces the following sequence of numbers:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...
```

For example:

```
a = recursiveFib(0) should return 0
b = recursiveFib(1) should return 1
c = recursiveFib(-1) should cause an error
d = recursiveFib(8) should return 21
```

5. Create and test a function called `recursiveMin` that takes in a vector and returns the element with the minimum value and the index of that element as separate returned values, much as the standard `min(...)` function. If the input vector is of length zero, your function should return two empty vectors. If the input vector contains two minimum elements of equal value, your function should return the index of the first element. Create suitable test cases and use the built-in function `min(...)` only to test your answers.

For example:

```
[m n] = recursiveMin([]) should return [] and []
[m n] = recursiveMin([5]) should return 5 and 1
[m n] = recursiveMin([5 2]) should return 2 and 2
[m n] = recursiveMin([2 5 2]) should return 2 and 1
[m n] = recursiveMin([2 5 2 1 6 7]) should return 1 and 4
```