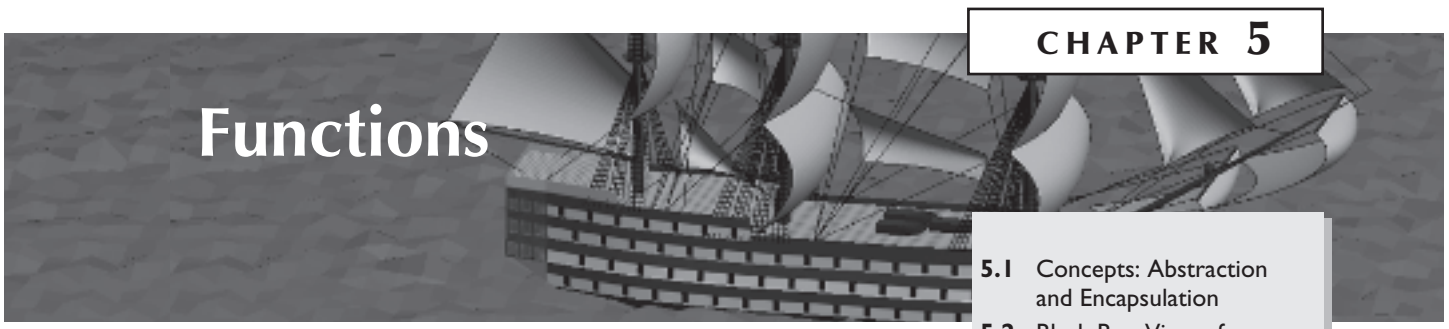
CHAPTER 5

Functions



Chapter Objectives

This chapter discusses the nature, implementation, and behavior of user-defined functions in MATLAB:

- How to define a function
- How data are passed into a function
- How to return data, including multiple results
- How to include other functions not needed except as helpers to your own function

Writing a user-defined function allows you to isolate and package together a code block, so that you can apply that code block to different sets of input data. We have already made use of some built-in functions like `sin(...)` and `plot(...)` by calling them; this chapter will deal with creating and using your own functions.

5.1 Concepts: Abstraction and Encapsulation

A **function** is an implementation of procedural abstraction and encapsulation. **Procedural abstraction** is the concept that permits a code block that solves a particular sub-problem to be packaged and applied to different data inputs. This is exactly analogous to the concept of data abstraction we discussed in Chapter 3 where individual data items are gathered to form a collection. We have already used a number of built-in procedural abstractions in the form of functions. All the mathematical functions that compute—for example, the sine of a collection of angles or the maximum value of a vector—are examples of procedural abstraction. They allow us to apply a code block about which we know nothing to data sets that we provide. To make use of a built-in function, all we have to do is provide data in the form the function expects and interpret the results according to the function's specification.

Encapsulation is the concept of putting a wrapper around a collection that you wish to protect from outside influence. Functions encapsulate the code they contain in two ways: the variables declared within the function are not visible from elsewhere, and the function's ability to change the values of variables (otherwise known as causing side effects) is restricted to its own code body.

5.2 Black Box View of a Function

The most abstract view of a function can be seen in Figure 5.1. It consists of two parts: the definition of the interface by which the user passes data items to and from the function, and the code block that produces the results required by that interface. A function definition consists of the following components:

- A name that follows the same syntactic rules as a variable name
- A set of 0 or more parameters provided to the function
- Zero or more results to be returned to the caller of the function

The basic operation of a function begins before execution of the function actually starts. If the function definition requires *n* parameters, the calling instructions first prepare *n* items of data from its workspace to be provided

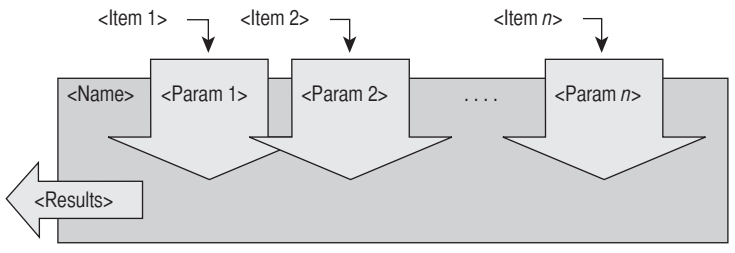


Figure 5.1 *Black box view of a function*

to the function. These data are then passed to the function, the code body is executed, and the results are returned to the caller.

5.3 MATLAB Implementation

In this section, first we consider the general template for implementing functions and then the MATLAB implementation of that template.

5.3.1 General Template

The general layout of a function definition is shown in Template 5.1. The `<return info>` section for most functions involves providing the name(s) of the results returned followed by an = sign. If more than one result is to be returned, they are defined in a vector-like container. If nothing is to be returned from this function, both the result list and the = sign are omitted. The `<function name>` is a name with the same syntactic rules as a variable name and will be used to invoke the code body. The `<parameters>` section is a comma-separated list of the names of the data to be provided to the function. The `<documentation>` section is one or more lines of comments that describe what the function does and how to call it. These lines will appear in two situations:

- All the documentation lines up to the first non-document line are printed in the Command window when you type the following:
  ```
  >> help <function name>
  ```
- The first line is listed next to the file name in the Current Directory listing

5.3.2 Function Definition

In the MATLAB language, functions must be stored in a separate file located in a directory accessible to any script or function that calls it. The file containing the definition of a function named `function_name` must be `<function_name>.m`. For the general user, the Current Directory is the normal place to store it. Listing 5.1 illustrates a typical MATLAB function called `cylinder` that consumes two parameters, the `height` and `radius` of a cylinder, and produces the return variable `volume`.

In Listing 5.1:

Line 1: The MATLAB function definition is introduced by the key word `function`, followed by the name of the return variable (if any)

Template 5.1 General template for a function

```
function <return info> <function name> (<parameters>)
<documentation>
<code body> % must return the results
```

Listing 5.1 Cylinder function

```
1. function volume = cylinder(height, radius)
   % function to compute the volume of a cylinder
   % volume = cylinder(height, radius)
2.     base = pi * radius^2
3.     volume = base * height
4. end
```

and the = sign, then the name of the function and the names of the formal parameters in parentheses. All comments written immediately after the function header are available to the MATLAB Command window when you enter:

```
>>help <function_name>
```

The first comment line also appears in the Current Directory window as an indication of the basic purpose of the function. It is a good idea to include in the comments a usage statement showing copy of the function header line, sometimes referred to as the Application Programmer Interface (API), to remind a user exactly how to use this function.

Line 3: Although encapsulation rules forbid access to the caller's variables, the code body still has access to all built-in MATLAB variables and functions (e.g., `pi`, as used here).

Line 5: You must make at least one assignment to the result variable.

Line 6: Regrettably, the `end` statement is not required if there is only one function in the file; without it, the code body terminates at the end of the file. However, it must be present if there are other function definitions in the same file.

Try saving and testing the cylinder function in Exercise 5.1.

Exercise 5.1 Saving and testing the `cylinder` function

Enter the function definition from Listing 5.1 in the Text Editor and save it as `cylinder.m` in your Current Directory. Then enter the following experiments in the Interactions window. Notice that the first help line appears next to this file name in the Current Directory.

```
>> help cylinder
  function to compute the volume of a cylinder
    volume = cylinder(height, radius)
>> cylinder(1, 1)
ans =
    3.1416
>>
```

5.3.3 Storing and Using Functions

All user-defined MATLAB functions must be created like scripts in an m-file. When the file is first created, it must be saved in an m-file with the same file name as the function. For example, the function in Listing 5.1 named `cylinder` must be saved in a file named `cylinder.m`. Once the file has been saved, you may invoke the function by entering its name and parameters of the right type and number in the Command window, in a script, or in other function definitions. If you do not specify an assignment for the result of the function call, it will be assigned to the variable `ans`.

5.3.4 Calling Functions

When a function is defined, the user provides a list of the names of each data item expected to be provided by the caller. These are called the **formal parameters.** When this function is called, the caller must provide the same number of data values expected by the function definition. These are the **actual parameters** and can be generated in the following ways:

- Constants
- Variables that have been defined
- The result of some mathematical operation(s)
- The result returned from other functions

When the actual parameters have been computed, ***copies of*** their values are assigned as the values of the formal parameters the function is expecting. Values are assigned to parameters by position in the calling statement and function definition.

Style Point 5.1 Parameter Passing

Some languages provide an alternative technique—"passing by reference"—whereby the memory location for the parameters is passed to the function while the values remain in the caller's workspace. Syntactically, this is usually a bad thing, allowing deliberate or accidental assignments to "reach back" into the scope of the calling code and thereby perhaps causing undesirable side effects. However, restricting parameter access to passing by value can result in poor program performance. When a function needs access to large sets of data, consider improving the efficiency by using global variables.

The process of copying the actual parameters into the formal parameters is referred to as "passing by value"—the only technique defined in the MATLAB language for passing data into a function.

Once the parameter names have been defined in the function's workspace, the function's code body is executed, beginning with the first instruction. If return variables have been defined for the function, every exit from the code body must assign valid values for the results.

5.3.5 Variable Numbers of Parameters

Although the number of parameters is usually fixed, most languages, including MATLAB, provide the ability to deal with a variable number of

parameters, both incoming and returning. The built-in function `nargin` computes the actual number of parameters provided by the user in the current function call. If the function is designed to make use of `nargin`, the user calling this function can provide any values he deems important and allow the function to set default values for the unnecessary parameters.

Similarly, the function `nargout` computes the number of storage variables actually provided by the user. So if one or more of the results requires extensive computation or user interaction and the caller has not asked for that data, that computation can be omitted.

5.3.6 Returning Multiple Results

The MATLAB language is unique among programming languages in providing the ability to return more than one result from a function by name. The multiple results are specified as a "vector" of variable names, for example, `[area, volume]`, as shown in Listing 5.2. Assignments must be made to each of the result variables. However, the calling program is not required to make use of all the return values.

In Listing 5.2:

Line 1: Multiple results to be returned are specified as a "vector" of variable names, each of which must be assigned from the code body.

Lines 2–3: Same as Listing 5.1

Line 4: Added to set the value of the second result.

Exercise 5.2 shows how to invoke a function that can return multiple results. Notice that the normal method to access the multiple answers is to put the names of the variable to receive the results in a vector. The names may be any legal variable name, and the values are returned in the order of the results defined. If you choose less than the full number of results (or none at all), the answers that are specified are allocated from left to right from the available results. As with parameter assignment, the results are allocated by position in these vectors. Although we called the variable `v` in the last test, it still receives the value of the first result, `area`. If you really only want

Listing 5.2 `cylinder` function with multiple results

```
1. function [area, volume] = cylinder(height, radius)
   % function to compute the area and volume of a cylinder
   % usage: [area, volume]=cylinder(height, radius)
2.     base = pi .* radius.^2;
3.     volume = base .* height;
4.     area = 2 * pi * radius .* height + 2 * base;
5. end
```

Exercise 5.2 Testing multiple returns

Adapt the original `cylinder` function as shown in Listing 5.2 and perform the following tests in the Command window:

```
>> [a, v] = cylinder(1, 1)
a =
    6.2832
v =
    3.1416
>> cylinder(1, 1)
ans =
    6.2832
>> a = cylinder(1, 1)
a =
    6.2832
>> v = cylinder(1, 1)
v =
    6.2832
>>
```

the second result value, you must put either a `'~'` marker or a dummy variable name like `'junk'` in the place of any variable you wish to ignore. So this call:

```
[~, v] = cylinder(1, 1);
```

will put the volume in the variable v.

5.3.7 Auxiliary Local Functions

Since the MATLAB language uses the name of the file to identify a function, every function should normally be saved in its own m-file. However, there are times when auxiliary functions (sometimes called "helper functions") are needed to implement the algorithm contained in the main function in a file. If this auxiliary function is only used in the main function or its helpers, it can be written in the same file as its calling function after the definition of the main function. By convention, some people append the word `local_` to the name of local functions.

Scripts or functions that use the code in an m-file can reach only the first function. Other functions in the m-file, the auxiliary functions, can only be called from the first function or other auxiliary functions in the same file.

5.3.8 Encapsulation in MATLAB Functions

Encapsulation is accomplished in most modern languages, including MATLAB, by implementing the concept of variable scoping. In practice, this is achieved by allocating a separate workspace to each function. When

MATLAB is first started, a default workspace is created in which variables created in the Command window or by running scripts are stored. When a function is called, a fresh workspace is created (see Section 9.1.2 for details), and the actual parameter values are copied into the formal parameter names in that new workspace. When the function finishes, this operation is reversed. The returning parameters are copied into the variables provided by the caller in the previous workspace, and the function's workspace is released. The Variables window always shows you the contents of the current workspace.

Variable scoping defines the places within your Command window, MATLAB system, and m-files to which instructions have access. It is related to the Variables window, which shows you your current workspace. When using the Command window or running a script and you access the value of a variable, the system will reach into your current workspace and then into the MATLAB system libraries to find its current value. This is referred to as **Global Scope.** When you run a function, its local variables, including the internal names of its parameters, are not included in your current workspace, and it does not look into your current workspace for values of variables it needs. This is referred to as **Local Scope,** wherein the variables within a function are not visible from outside and the function is unable to cause side effects by making assignments to variables in other workspaces except by returning results.

To illustrate variable scoping, do Exercise 5.3.

5.3.9 Global Variables

Because MATLAB always copies the input data into the function's workspace, there are occasions when it is very inefficient to pass large data sets into and out of a function. To avoid passing large amounts of data, we can use global variables. Global variables must be defined in both the calling script and the function using the key word `global`. For example, suppose we collect a large volume of data in a variable `buffer` and do not want to copy the whole buffer into and out of a function that processes that data. In this case, we declare the variable to be global in both the calling space and the called function by placing the following line of code before the variable is first used in both places:

```
global buffer
```

The function will then be able to access and modify the values in `buffer` without having to pass it in and out as a parameter. This feature must be used with caution, however, because

Style Points 5.2

1. Before you include a function in a complex algorithm, you should always test its behavior in isolation in a script. This test script should validate not only the normal operation of the function, but also its response to erroneous input data it might receive.

2. Although any legal MATLAB instruction is permitted within the code body of a function, it is considered bad form (except temporarily for debugging purposes) to display values in the Interactions window.

3. We also actively discourage the use of the `input(...)` function within the code body. If you need to input some values to test a function, do so from the Interactions window or a test script.

Exercise 5.3 Observing variable scoping

Put a break point at Line 6 of your version of the code in Listing 5.2, and then rerun the function by entering:

```
>> [a, v] = cylinder(1, 1)
```

Notice that the logic stops at that break point and the Text Editor displays an arrow. The Workspace window shows you the values of `height`, `radius`, and `base` but none of the variables you left in the workspace for the Interactions window. The function has no access to other workspaces.

Observe that as you step through the function, the variables appear in the Variables window and are updated. When you return from the cylinder function to display the results, the workspace for the function disappears. The calling environment has no access to the variables within the function.

any function with global access to data is empowered to change that data. In other words, the use of global data circumvents the natural MATLAB language's encapsulation mechanisms.

5.4 Engineering Example—Measuring a Solid Object

Problem:

Consider the disk shown in Figure 5.2. It has a radius *R*, height *h*, and eight cylindrical holes each of radius *r* bored in it. This might be a component of a machine that must be painted and then assembled with other components. During the process of designing this machine, we may need to know the weight of this disk and the amount of paint required to finish it. The weight and the amount of paint for the machine is the sum of the values for each component. Since the weight of our disk is proportional to its volume and the amount of paint is proportional to its "wetted area," we need the volume and area of this disk.

Write a script to compute the volume of the disk and its wetted area.

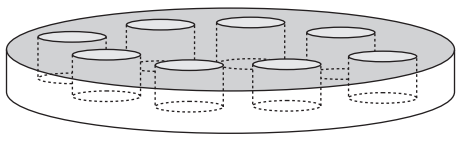


Figure 5.2 *Disk with holes*

Solution:

Listing 5.3 shows the code that solves this problem.

In Listing 5.3:

Lines 1–3: Set up the disk sizes. Notice that the script works fine with a vector of disk thicknesses to check the behavior as thickness varies.

Listing 5.3 Volume and area of a disk

```
1. h = 1:5;     % set a range of disk thicknesses
2. R = 25;
3. r = 3;

4. [Area Vol] = cylinder(h, R) % dimensions of large disk
5. [area vol] = cylinder(h, r) % dimensions of the hole

   % compute remaining volume
6. Vol = Vol - 8*vol
   % the wetted area is a little messier. If we total the
   % large disk area and the areas of the holes, we get the
   % wetted area of the curved edges inside and out.
   % However, for each hole, the top and bottom areas have
   % been included not only in the top and bottom of the big
   % disk, but also as the contributions of each hole.
   % From the sum of the top areas, we therefore have to
   % remove 32 times the hole top area

7. Area = Area + 8*(area - 2*2*pi*r.^2)
```

Hint 5.1

If you experiment with this script a little, you will discover the power of vector processing for rapidly determining the sensitivity of results to different parameters. The mathematics may not work if you provide vectors for more than one of the given data items. However, vectors supplied for each of them in turn provide insight into the sensitivity of the results to each parameter.

Line 4: Area and volume of the large disk.

Line 5: Area and volume of one hole.

Line 6: Volume computation.

Line 7: The area computation.

Table 5.1 shows the results when this code is run. Notice that for thin disks, the area is smaller with the holes. However, as the thickness increases, the area with the holes is larger than without, as one would expect.

Table 5.1 Volume and area results

Area = 4,084 4,241 4,398 4,555 4,712
Vol = 1,963 3,927 5,890 7,854 9,817
area = 75 94 113 132 151
vol = 28 57 85 113 141
Vol = 1,737 3,474 5,212 6,949 8,687
Area = 3,782 4,090 4,398 4,706 5,014

Chapter Summary

This chapter showed you how to encapsulate a code block to allow it to be reused:

- Functions are defined in a file of the same name using the key word `function` to distinguish them from scripts
- Parameters are copied in sequence into the function and given the names of the formal parameters
- Results are returned to the caller by assigning value(s) to the return variable(s)
- Variables within the function can be accessed only in the function's code block unless they are declared global
- Helper functions accessible only to functions within the same file may be added below the main function and otherwise obey the same rules as the main function

Special Characters, Reserved Words, and Functions

Special Characters, Reserved Words, and Functions	Description	Discussed in This Section
`( )`	Used to identify the formal and actual parameters of a function	5.3.2, 5.3.4
`help`	Invokes help utility	5.3.1
`function`	Identifies an m-file as a function	5.3.2
`nargin`	Determines the number of input parameters actually supplied by a function's caller	5.3.4
`nargout`	Determines the number of output parameters actually requested by a function's caller	5.3.4
`global <var>`	Defines the scope of the variable `<var>` as globally accessible	5.3.8

Self Test

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

True or False

1. All data used by a function must be passed in as parameters to the function.
2. The name of the first function in an m-file must match the name of the file containing its definition.
3. The first documentation line appears in the Current Directory listing.

4. Functions must consume at least one parameter.
5. The calling code must provide assignments for every result returned from a function.
6. The names of auxiliary functions must begin with `local_`.

Fill in the Blanks

1. _____ permits a code block to be packaged and referred to collectively rather than individually.
2. Values of the _____ parameters are copied to define the _____ parameters inside the function.
3. If more than one result is to be returned from a function, they are defined in a(n) _____.
4. _____ describes the situation where the variables within a function are not visible from outside, and the function is unable to cause side effects by making assignments to outside variables.
5. Calling code can only reach the _____ function in an m-file. Other functions in the m-file can only be called from the _____ or _____.

Programming Projects

1. Write a function called `checkFactor` that takes in two numbers and checks if they are divisible, that is, if the first is divisible by the second. You may assume that both numbers are positive. Your function should return a logical value, `true` or `false`.

Hint:

mod(`x, y`) gives the remainder when `x` is divided by `y`.

For example:

```
checkFactor(25,6) should return false.
checkFactor (9,3) should return true.
checkFactor (3,9) should return false.
```

2. Write and test the code for the function `mysteryFunction` that consumes a vector, `v`, and produces a new vector, `w`, of the same length where each element of `w` is the sum of the corresponding element in `v` and the previous element of `v`. Consider the previous element of `v(1)` to be 0.

For example:

```
mysteryFunction( 1:8 ) should return
      [1  3  5  7  9 11 13 15]
mysteryFunction([1:6].^2) should return
      [1  5 13 25 41 61]
```

3. Coming off a respectable 7–6 record last year, your football team is looking to improve on that this season. They have contacted you and asked for your help projecting some of the scenarios for their

win–loss record. They want you to write a function called `teamRecord` that takes in two parameters—`wins`, and `losses`, and returns two values—`season` and `wPercentage`. `Season` should be a logical result that is `true` for a winning season. `wPercentage` is the percentage of games won (ranging from 0 to 100).

For example:

```
[season wPercentage] = teamRecord(3, 9)
should return season = false, wPercentage = 25
[season wPercentage] = teamRecord(10, 2)
should return season = true, wPercentage = 83.3
```

4. Write a function called `classAverage` that takes in an array of numbers and, after normalizing the grades in such a way that the highest corresponds to 100 (see Chapter 3, Problem 5), returns the letter grade of the class average. The grade ranges are as follows:

```
average>90 =>     A
80<=average<90 => B
70<=average<80 => C
60<=average<70 => D
average<60     => F
```

For example:

```
classAverage( [70 87 95 80 80 78 85 90 66
               89 89 100] ) should return B
classAverage( [50 90 61 82 75 92 81 76 87 41
                31 98] ) should return C
classAverage( [10 10 11 32 53 12 34 74 31 30
                26 22] ) should return F
```

5. Write a function called `myMin4` that will take in four numbers and returns the minimum value and an index showing which parameter it was. You may not use the built-in `min()` function.

For example:

```
myMin4(1,3,5,7) should return 1 and 1
myMin4(8,9,2,4) should return 2 and 3
```

6. Write the function `meansAndMedian` that takes in a vector of numbers and returns the arithmetic and geometric means, as well as the median. You may not use the built-in functions `mean()`, `median()`, or `geomean()`. However, you could type `"help geomean"` to familiarize yourself with computing the geometric mean of a group of numbers.

Hint:
The built-in function `sort()` might help to compute the median of the vector.

7. Given an array of numbers that could be negative, write a function `posavg(a)` to calculate and return the average (`mean`) of the non-negative numbers in the single dimensional array, a. One such solution is `mean(a(find(a>0)))`. In order to test your understanding

of class concepts, re-implement the `posavg(a)` function using iteration. You may not use the built-in functions `sum(...)`, `find(...)`, or `mean(...)` in your solution.

8. Write a function called `sumAndAverage`. It should take in an array of numbers and return the sum and average of the array in that order.

   For example:

   ```
   sumAndAverage([3 2 3 2]) should return 10 and 2.5
   sumAndAverage([5 -5 2 8 0]) should return 10 and 2
   sumAndAverage([]) should return 0 and 0
   ```

9. You are already familiar with the logical operators `&&` (and) and `||` (or), as well as the unary negation operator ~(not). In a weakly typed language such as MATLAB, the binary states `true` and `false` could be equivalently expressed as a `1` or a `0`, respectively. Let us now consider a ternary number system, consisting of the states `true(1)`, `maybe(2)`, and `false(0)`. The truth table for such a system is shown below. Implement the truth table by writing the functions `f=tnot(x)`, `f=tand(x,y)`, and `f=tor(x,y)`. You may not assume that only valid input numbers will be entered.

x	y	tnot(x)	tand(x,y)	tor(x,y)
1	1	0	1	1
1	0	0	0	1
1	2	0	2	1
0	1	1	0	1
0	0	1	0	0
0	2	1	2	0
2	1	2	2	1
2	0	2	2	0
2	2	2	2	2

10. Write a function called `multiSum(A)`. This particular function should take in a `N` × `M` array, `A`, and return four results:

    ```
    A 1 × M vector with the sum of the columns,
    A N × 1 vector with the sum of the rows, and
    Two numbers containing the sums of the two diagonals, the
    major diagonal first.
    ```

    For example:

    ```
    columnSum([1 2 3; 4 5 6; 7 8 9]) should return
          [12 15 18], [6 15 24]', 15 and 15
    columnSum([0 2 3; 4 0 6; 7 8 0]) should return
          [11 10 9], [5 10 15]', 0 and 10
    columnSum(eye[5,5]) should return
    [1 1 1 1 1], [1 1 1 1 1]', 5 and 1
          columnSum([]) should return [], [], 0 and 0
    ```

11. You are playing a game where you roll a die 10 times. If you roll a 5 or 6 seven or more times, you win 2 dollars; four or more times, you win 1 dollar; and if you roll a 5 or 6 three or less times, you win no money. Write a function called `diceGame` that takes in a vector representing the die values and returns the amount of money won.

For example:

```
diceGame([5 1 4 6 5 5 6 6 5 2]) should return 2
diceGame([2 4 1 3 6 6 6 4 5 3]) should return 1
diceGame([1 4 3 2 5 3 4 2 6 5]) should return 0
```

Note: This function should work for any length vector.