

4.2 Control Structures in Matlab

In this section we will discuss the control structures offered by the Matlab programming language that allow us to add more levels of complexity to the simple programs we have written thus far. Without further ado and fanfare, let's begin.

If

If evaluates a logical expression and executes a block of statements based on whether the logical expressions evaluates to true (logical 1) or false (logical 0). The basic structure is as follows.

```
if logical_expression
    statements
end
```

If the **logical_expression** evaluates as true (logical 1), then the block of statements that follow **if logical_expression** are executed, otherwise the statements are skipped and program control is transferred to the first statement that follows **end**. Let's look at an example of this control structure's use.

Matlab's **rem(a,b)** returns the remainder when a is divided by b . Thus, if a is an even integer, then **rem(a,2)** will equal zero. What follows is a short program to test if an integer is even. Open the Matlab editor, enter the following script, then save the file as **evenodd.m**.

```
n = input('Enter an integer: ');
if (rem(n,2)==0)
    fprintf('The integer %d is even.\n', n)
end
```

Return to the command window and run the script by entering **evenodd** at the Matlab prompt. Matlab responds by asking you to enter an integer. As a response, enter the integer 12 and press **Enter**.

¹ Copyrighted material. See: <http://msenux.redwoods.edu/Math4Textbook/>

```
>> evenodd
Enter an integer: 12
The integer 12 is even.
```

Run the program again. When prompted, enter the nonnegative integer 17 and press **Enter**.

```
>> evenodd
Enter an integer: 17
```

There is no response in this case, because our program provides no alternative if the input is odd.

Besides the new conditional control structure, we have two new commands that warrant attention.

1. Matlab's **input** command, when used in the form **n = input('Enter an integer: ')**, will display the string as a prompt and wait for the user to enter a number and hit **Enter** on the keyboard, whereupon it stores the number input by the user in the variable *n*.
2. The command **fprintf** is used to print formatted data to a file. The default syntax is **fprintf(FID,FORMAT,A,...)**.
 - i. The argument **FID** is a file identifier. If no file identifier is present, then **fprintf** prints to the screen.
 - ii. The argument **FORMAT** is a format string, which may contain certain *conversion specifications* from the C programming language. In the **evenodd.m** script, **%d** is conversion specification which will format the first argument following the format string as a signed decimal number. Note the **\n** at the end of the format string. This is a *newline* character, which creates a new line after printing the format string to the screen.
 - iii. The format string can be followed by zero or more arguments, which will be substituted in sequence for the C-language conversion specifications in the format string.

Else

We can provide an alternative if the **logical_expression** evaluates as false. The basic structure is as follows.

```

if logical_expression
    statements
else
    statements
end

```

In this form, if the **logical_expression** evaluates as true (logical 1), the block of statements between **if** and **else** are executed. If the **logical_expression** evaluates as false (logical 0), then the block of statements between **else** and **end** are executed.

We can provide an alternative to our **evenodd.m** script. Add the following lines to the script and resave as **evenodd.m**.

```

n = input('Enter an integer: ');
if (rem(n,2)==0)
    fprintf('The integer %d is even.\n', n)
else
    fprintf('The integer %d is odd.\n', n)
end

```

Run the program, enter 17, and note that we now have a different response.

```

>> evenodd
Enter an integer: 17
The integer 17 is odd.

```

Because the logical expression **rem(n,2)==0** evaluates as false when $n = 17$, the **fprintf** command that lies between **else** and **end** is executed, informing us that the input is an odd integer.

Elseif

Sometimes we need to add more than one alternative. For this, we have the following structure.

```

if logical_expression1
    statements
elseif logical_expression2
    statements
else
    statements
end

```

Program flow is as follows.

1. If **logical_expression1** evaluates as true, then the block of statements between **if** and **elseif** are executed.
2. If **logical_expression1** evaluates as false, then program control is passed to **elseif**. At that point, if **logical_expression2** evaluates as true, then the block of statements between **elseif** and **else** are executed. If the logical expression **logical_expression2** evaluates as false, then the block of statements between **else** and **end** are executed.

You can have more than one **elseif** in this control structure, depending on need. As an example of use, consider a program that asks a user to make a choice from a menu, then reacts accordingly.

First, ask for input, then set up a menu of choices with the following commands.

```

a=input('Enter a number a: ');
b=input('Enter a number b: ');
fprintf('\n')
fprintf('1) Add a and b.\n')
fprintf('2) Subtract b from a.\n')
fprintf('3) Multiply a and b.\n')
fprintf('4) Divide a by b.\n')
fprintf('\n')
n=input('Enter your choice: ');

```

Now, execute the appropriate choice based on the user's menu selection.

```

if n==1
    fprintf('The sum of %0.2f and %0.2f is %0.2f.\n',a,b,a+b)
elseif n==2
    fprintf('The difference of %0.2f and %0.2f is %0.2f.\n',a,b,a-b)
elseif n==3
    fprintf('The product of %0.2f and %0.2f is %0.2f.\n',a,b,a*b)
elseif n==4
    fprintf('The quotient of %0.2f and %0.2f is %0.2f.\n',a,b,a/b)
else
    fprintf('Not a valid choice.\n')
end

```

Save this script as **mymenu.m**, then execute the command **mymenu** at the command window prompt. You will be prompted to enter two numbers a and b . As shown, we entered 15.637 for a and 28.4 for b . The program presents a menu of choices and prompts us for a choice.

```

Enter a number a: 15.637
Enter a number b: 28.4

1). Add a and b.
2). Subtract b from a.
3). Multiply a and b.
4). Divide a by b.

Enter your choice:

```

We enter 1 as our choice and the program responds by adding the values of a and b .

```

Enter your choice: 1

The sum of 15.64 and 28.40 is 44.04.

```

Some comments are in order.

1. Note that after several **elseif** statements, we still list an **else** command to catch invalid entries for n . The choice for n should match a menu designation, either 1, 2, 3, or 4, but any other choice for n causes the statement following **else** to

be executed. This is a good programming practice, having a sort of “otherwise block” as a catchall for unexpected responses by the user.

2. We use the conversion specification **%0.2f** in this example, which outputs a number in fixed point format with two decimal places. Note that this results in rounding in the output of numbers with more than two decimal places.

Switch, Case, and Otherwise

As the number of **elseif** entries in a conditional control structure increases, the code becomes harder to understand. Matlab provides a much simpler control construct designed for this situation.

```
switch expression
  case value1
    statements
  case value2
    statements
  ...
  otherwise
    statements
end
```

The **expression** can be a scalar or string. **Switch** works by comparing the input **expression** to each **case** value. It executes the statements following the first match it finds, then transfers control to the line following the **end**. If **switch** finds no match, it executes the statement block following the **otherwise** statement.

Let’s create a second script having the same menu building code.

```
a=input('Enter a number a: ');
b=input('Enter a number b: ');
fprintf('\n')
fprintf('1) Add a and b.\n')
fprintf('2) Subtract b from a.\n')
fprintf('3) Multiply a and b.\n')
fprintf('4) Divide a by b.\n')
fprintf('\n')
n=input('Enter your choice: ');
fprintf('\n')
```

However, instead of using an **if ...elseif...else...end** control structure to select a block of statements to execute based on the user's choice of menu item, we will implement a **switch** structure instead.

```
switch n
case 1
    fprintf('The sum of %.2f and %.2f is %.2f.\n',a,b,a+b)
case 2
    fprintf('The difference of %.2f and %.2f is %.2f.\n',a,b,a-b)
case 3
    fprintf('The product of %.2f and %.2f is %.2f.\n',a,b,a*b)
case 4
    fprintf('The quotient of %.2f and %.2f is %.2f.\n',a,b,a/b)
otherwise
    fprintf('Not a valid choice.\n')
end
```

Save the script as **mymenuswitch.m**, change to the command window, and enter and execute the command **mymenuswitch** at the command prompt. Note that the behavior of the program **mymenuswitch** is identical to that of **mymenu**.

```
Enter a number a: 15.637
Enter a number b: 28.4

1) Add a and b.
2) Subtract b from a.
3) Multiply a and b.
4) Divide a by b.

Enter your choice: 1

The sum of 15.64 and 28.40 is 44.04.
```

If the user enters an invalid choice, note how control is passed to the statement following **otherwise**.

```
Enter a number a: 15.637
Enter a number b: 28.4
```

- 1) Add a and b.
- 2) Subtract b from a.
- 3) Multiply a and b.
- 4) Divide a by b.

```
Enter your choice: 5
```

```
Not a valid choice.
```

Loops

A **for** loop is a control structure that is designed to execute a block of statements a predetermined number of times. Here is its general use syntax.

```
for index=start:increment:finish
    statements
end
```

As an example, we display the squares of every other integer from 5 to 13, inclusive.

```
for k=5:2:13
    fprintf('The square of %d is %d.\n', k, k^2)
end
```

The output of this simple loop follows.

```
The square of 5 is 25.
The square of 7 is 49.
The square of 9 is 81.
The square of 11 is 121.
The square of 13 is 169.
```

Another looping construct is Matlab's **while** structure whose basic syntax follows.

```

while expression
    statements
end

```

The **while** loop executes its block of statements as long as the logical controlling **expression** evaluates as true (logical 1). For example, we can duplicate the output of the previous **for** loop with the following **while** loop.

```

k=5;
while k<=13
    fprintf('The square of %d is %d.\n', k, k^2)
    k=k+2;
end

```

When using **while**, it is not difficult to fall into a trap of programming a loop that iterates indefinitely. In the example above, the loop will iterate as long as **k<=13**, so it is imperative that we increment the value of k in the interior of the loop, as we do with the command **k=k+2**. This command adds 2 to the current value of k , then replaces k with this incremented value. Thus, the first time through the loop, $k = 5$, the second time through the loop, $k = 7$, etc. When k is no longer less than or equal to 13, the loop terminates.

The simple use of **for** and **while** loops to generate the squares of every other integer from 5 to 13 can be accomplished just as easily with array operations.

```

k=5:2:13;
A=[k; k.^2];
fprintf('The square of %d is %d.\n', A)

```

The use of **fprintf** in this example warrants some explanation. First, the command **A=[k; k.^2]** generates a matrix with two rows, the first row holding the values of k , the second the squares of k .

```

A =
     5     7     9    11    13
    25    49    81   121   169

```

Again, with roots deeply planted in Fortran, Matlab enters the entries of matrix A in columnwise fashion. The following command will print all entries in the matrix A .

```
>> A(:)
ans =
    5
   25
    7
   49
    9
   81
   11
  121
   13
  169
```

It is important to note the order in which the entries were extracted from the matrix A , that is, the entries from the first column, followed by the entries from the second column, etc, until all of the entries of matrix A are on display. This is precisely the order in which the **fprintf** command picks off the entries of matrix A in our example above. The first time through the loop, **fprintf** replaces the two occurrences of **%d** in its format string with 5 and 25. The second time through the loop, the two occurrences of **%d** are replaced with 7 and 49, and so on, until the loop terminates.

For additional help on **fprintf**, type **doc fprintf** at the Matlab command prompt.

Although a good introductory example of using loops, producing the squares of integers is not a very interesting task. Let's look at a more interesting example of the use of **for** and **while** loops.

► **Example 1.** *For a number of years, a number of leading mathematicians believed that the infinite series*

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots \quad (4.1)$$

*converged to a finite sum, but none of them could determine what that sum was. Until 1795, that is, when Leonhard Euler produced the surprising result that the sum of the series equaled $\pi^2/6$. Use a **for** loop to sum the first twenty terms of the series and compare this partial sum with $\pi^2/6$ (compute the relative*

error). Secondly, write a **while** loop that will determine how many terms must be summed to produce an approximation of $\pi^2/6$ that is correct to 4 significant digits.

We wish to sum the first 20 terms, so we begin by setting $n = 20$. We will store the running sum in the variable s , which we initialize to zero. A **for** loop is used to track the running sum. On each pass of the **for** loop, the sum is updated with the command $s=s+1/k^2$, which takes the previous value of the running sum stored in s , adds the value of the next term in the series, and stores the result back in s .

```
n=20;
s=0;
for k=1:n
    s=s+1/k^2;
end
```

The true sum of the series is $\pi^2/6$. Hence, we compute the relative error with the following statement.

```
rel=abs(s-pi^2/6)/abs(pi^2/6);
```

We can then use **fprintf** to report the results.

```
fprintf('The sum of the first %d terms is %f.\n', n, s)
fprintf('The relative error is %e.\n', rel)
```

You should obtain the following results.

```
The sum of the first 20 terms is 1.596163.
The relative error is 2.964911e-02.
```

Note that the relative error is less than 5×10^{-2} , so our approximation of $\pi^2/6$ (using the first 20 terms of the series) contains only 2 significant digits.

The next question asks how many terms must be summed to produce an approximation of $\pi^2/6$ correct to 4 significant digits. Thus, the relative error in the approximation must be less than 5×10^{-4} .

We start by setting **tol=5e-4**. The intent is to loop until the relative error is less than this tolerance, which signifies that we have 4 significant digits in our approximation of $\pi^2/6$ using the infinite series. The variable s will again contain the running sum and we again initialize the running sum to zero. The variable k contains the term number, so we initialize k to 1. We next initialize **rel_error** to 1 so that we enter the **while** loop at least once.

```
tol=5e-4;
s=0;
k=1;
rel_error=1;
```

Each pass through the loop, we do three things:

1. We update the running sum by adding the next term in the series to the previous running sum. This is accomplished with the statement **s=s+1/k²**.
2. We increment the term number by 1, using the statement **k=k+1** for this purpose.
3. We calculate the current relative error.

We continue to loop while the relative error is greater than or equal to the tolerance.

```
while rel_error>=tol
    s=s+1/k^2;
    k=k+1;
    rel_error=abs(s-pi^2/6)/abs(pi^2/6);
end
```

When the loop is completed, we use **fprintf** to output results.

```
fprintf('The actual value of pi^2/6 is %f.\n', pi^2/6)
fprintf('The sum of the first %d terms is %f.\n', k, s)
fprintf('The relative error is %f.\n', rel_error)
```

The results follow.

```
The actual value of pi^2/6 is 1.644934.
The sum of the first 1217 terms is 1.644112.
The relative error is 0.000500.
```

Note that we have agreement in approximately 4 significant digits.



This example demonstrates when we should use a **for** loop and when a **while** loop is more appropriate. We offer the following advice.

What type of loop should I use? When you know in advance the precise number of times the loop should iterate, use a **for** loop. On the other hand, if you have no predetermined knowledge of how many times you will need the loop to execute, use a **while** loop.

For k = A

Matlab also supports a type of **for** loop whose block of statements are executed for each column in a matrix A . The syntax is as follows.

```
for k = A
    statements
end
```

In this construct, the columns of matrix A are stored one at a time in the variable k while the following statements, up to the **end**, are executed. For example, enter the matrix A .

```
>> A=[1 2;3 4]
A =
     1     2
     3     4
```

Now, enter and execute the following loop.

```
>> for k=A, k, end
k =
    1
    3
k =
    2
    4
```

Note that k is assigned a *column* of matrix A at each iteration of the loop.

This column assignment does not contradict the use we have already investigated, i.e., **for k=start:increment:finish**. In this case, the Matlab construct **start:increment:finish** creates a *row vector* and k is assigned a new column of the row vector at each iteration. Of course, this means that k is assigned a new element of the row vector at each iteration.

As an example of use, let's plot the *family* of functions defined by the equation

$$y = 2Cte^{-t^2}, \quad (4.2)$$

where C is one of a number of specific constants. The following code produces the family of curves shown in **Figure 4.1**. At each iteration of the loop, C is assigned the next value in the row vector **[-5,-3,-1,0,1,3,5]**.

```
t=linspace(-4,4,200);
for C=[-5,-3,-1,0,1,3,5]
    y=-2*C*t.*exp(-t.^2);
    line(t,y)
end
```

Break and Continue

There are two situations that frequently occur when writing loops.

1. If a certain state is achieved, the programmer wishes to terminate the loop and pass control to the code that follows the end of the loop.
2. The programmer doesn't want to exit the loop, but does want to pass control to the next iteration of the loop, thereby skipping any remaining code that remains in the loop.

As an example of the first situation, consider the following snippet of code. Matlab's **break** command will cause the loop to terminate if a prime is found in the loop index k .

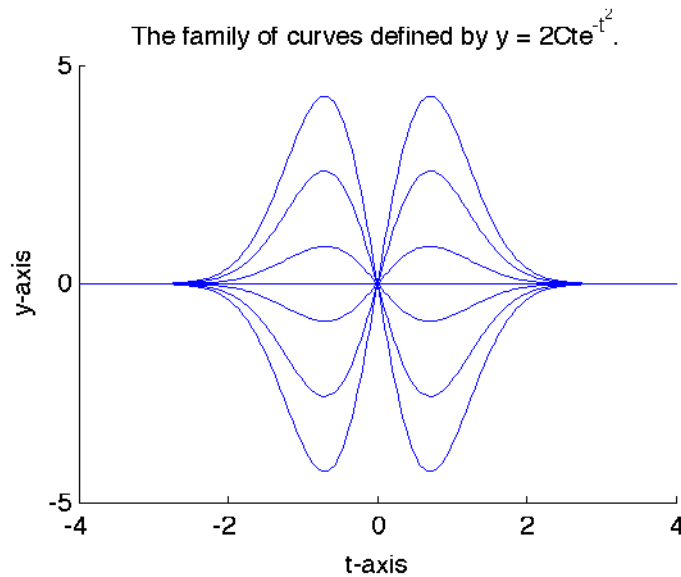


Figure 4.1. A family of curves for $C = -5, -3, -1, 0, 1, 3,$ and 5 .

```
for k=[4,8,12,15,17,19,21,24]
    if isprime(k)
        fprintf('Prime found, %d, exiting loop.\n',k)
        break
    end
    fprintf('Current value of index k is: %d\n',k)
end
```

Note that the output lists each value of k until a prime is found. The **fprintf** command issues a warning message and the **break** command terminates the loop.

```
Current value of index k is: 4
Current value of index k is: 8
Current value of index k is: 12
Current value of index k is: 15
Prime found, 17, exiting loop.
```

In this next code snippet, we want to print the multiples of three that are less than or equal to 20. At each iteration of the **while** loop, Matlab checks to see if k is divisible by 3. If not, it increments the counter k , then the **continue** statement

that follows skips the remaining statements in the loop, and returns control to the next iteration of the loop.

```
N=20;
k=1;
while k<=N
    if mod(k,3)~=0
        k=k+1;
        continue
    end
    fprintf('Multiple of 3: %d\n',k)
    k=k+1;
end
```

If a multiple of 3 is found, **fprintf** is used to output the result in a nicely formatted statement.

```
Multiple of 3: 3
Multiple of 3: 6
Multiple of 3: 9
Multiple of 3: 12
Multiple of 3: 15
Multiple of 3: 18
```

Any and All

The command **any** returns **true** if any element of a vector is a nonzero number or is logical 1 (true).

```
>> v=[0 0 1 0 1]; any(v)
ans =
     1
```

On the other hand, **any** returns **false** if non of the elements of a vector are nonzero numbers or logical 1's.

```
>> v=[0 0 0 0]; any(v)
ans =
     0
```

The command **all** returns **true** if all of the elements of a vector are nonzero numbers or are logical 1's (true).

```
>> v=[1 1 1 1]; all(v)
ans =
     1
```

On the other hand, **all** returns **false** if the vector contains any zeros or logical 0's.

```
>> v=[1 0 1 1 1]; all(v)
ans =
     0
```

The **any** and **all** command can be quite useful in programs. Suppose for example, that you want to pick out all the numbers from 1 to 2000 that are multiples of 8, 12, and 15. Here's a code snippet that will perform this task. Each time through the loop, we use the **mod** function to determine the remainder when the current value of k is divided by the numbers in **divisors**. If "all" remainders are equal to zero, then k is divisible by each of the numbers in **divisors** and we append k to a list of multiples of 8, 12, and 15.

```
N=2000;
divisors=[8,12,15];
multiples=[];
for k=1:N
    if all(mod(k,divisors)==0)
        multiples=[multiples,k];
    end
end
fmt='%5d %5d %5d %5d %5d\n';
fprintf(fmt,multiples)
```

The resulting output follows.

```

120   240   360   480   600
720   840   960  1080  1200
1320  1440  1560  1680  1800
1920

```

Vectorizing the Code. In this case we can avoid loops altogether by using logical indexing.

```

k=1:2000;
b=(mod(k,8)==0) & (mod(k,12)==0) & (mod(k,15)==0);
fmt='%5d %5d %5d %5d %5d\n';
fprintf(fmt,k(b))

```

The reader should check that this code snippet produces the same result, i.e., all multiples of 8, 12, and 15 from 1 to 2000.

Nested Loops

It's possible to nest one loop inside another. You can nest a second **for** loop inside a primary **for** loop, or you can nest a **for** loop inside a **while** loop. Other combinations are also possible. Let's look at some situations where this is useful.

► **Example 2.** A Hilbert Matrix H is an $n \times n$ matrix defined by

$$H(i, j) = \frac{1}{i + j - 1}, \quad (4.3)$$

where $1 \leq i, j \leq n$. Set up nested **for** loops to construct a Hilbert Matrix of order $n = 5$.

We could create this Hilbert Matrix using hand calculations. For example, the entry in row 3 column 2 is

$$H(3, 2) = \frac{1}{3 + 2 - 1} = \frac{1}{4}.$$

These calculations are not difficult, but for large order matrices, they would be tedious. Let's let Matlab do the work for us.

```
n=5;
H=zeros(n);
```

Now, doubly nested **for** loops achieve the desired result.

```
for i=1:n
    for j=1:n
        H(i,j)=1/(i+j-1);
    end
end
```

Set rational display format, display the matrix H , then set the format back to default.

```
format rat
H
format
```

The result is shown in the following *Hilbert Matrix* of order 5.

```
H =
    1          1/2          1/3          1/4          1/5
    1/2         1/3         1/4         1/5         1/6
    1/3         1/4         1/5         1/6         1/7
    1/4         1/5         1/6         1/7         1/8
    1/5         1/6         1/7         1/8         1/9
```

Note that the entry in row 3 column 2 is $1/4$, as predicted by our hand calculations above. Similar hand calculations can be used to verify other entries in this result.

How does it work? Because $n = 5$, the primary loop becomes **for i=1:5**. Similarly, the inner loop becomes **for j=1:5**. Now, here is the way the nested structure proceeds. First, set $i = 1$, then execute the statement **H(i,j)=1/(i+j-1)** for $j = 1, 2, 3, 4$, and 5 . With this first pass, we set the entries $H(1, 1)$, $H(1, 2)$, $H(1, 3)$, $H(1, 4)$, and $H(1, 5)$. The inner loop terminates and control returns to the primary loop, where the program next sets $i = 2$ and again executes the statement **H(i,j)=1/(i+j-1)** for $j = 1, 2, 3, 4$, and 5 . With this second pass, we set the entries $H(2, 1)$, $H(2, 2)$, $H(2, 3)$, $H(2, 4)$, and $H(2, 5)$. A third and fourth

pass of the primary loop occur next, with $i = 3$ and 4, each time iterating the inner loop for $j = 1, 2, 3, 4$, and 5. Finally, on the last pass through the primary, the program sets $i = 5$, then executes the statement $\mathbf{H(i,j)=1/(i+j-1)}$ for $j = 1, 2, 3, 4$, and 5. On this last pass, the program sets the entries $H(5, 1)$, $H(5, 2)$, $H(5, 3)$, $H(5, 4)$, and $H(5, 5)$ and terminates.

Fourier Series — An Application of a For Loop

Matlab's **mod(a,b)** works equally well with real numbers, providing the remainder when a is divided by b . For example, if you divide 5.7 by 2, the quotient is 2 and the remainder is 1.7. The command **mod(5.7,2)** should return the remainder, namely 1.7.

```
>> mod(5.7,2)
ans =
    1.7000
```

The following commands produce what engineers call a *sawtooth curve*, shown in **Figure 4.2(a)**.

```
t=linspace(0,6,500);
y=mod(t,2);
plot(t,y)
```

In **Figure 4.2(a)**, we plot the remainders when the time is divided by 2. Hence, we get the a periodic function of period 2, where the remainders grow from 0 to just less than 2, then repeat every 2 units.

With the following adjustment, we produce what engineers call a *square wave* (shown in **Figure 4.2(b)**). Recall that **y<1** returns true (logical 1) when y is less than 1, and false (logical 0) when y is not less than 1.

```
y=(y<1);
plot(t,y,'*')
```

This type of curve can emulate a switch that is “on” for one second (y -value 1), then off for the next second (y -value 0), and then periodically repeats this “on-off” cycle over its domain.

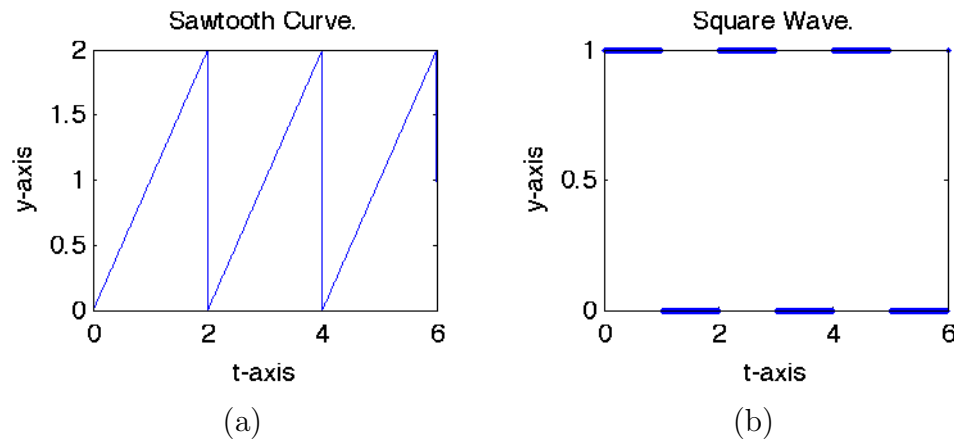


Figure 4.2. Generating a square wave with `mod`.

Let's make one last adjustment, increasing the amplitude by 2, then shifting the graph downward 1 unit. This produces the *square wave* shown in **Figure 4.3**.

```
y=2*y-1;
plot(t,y,'*')
```

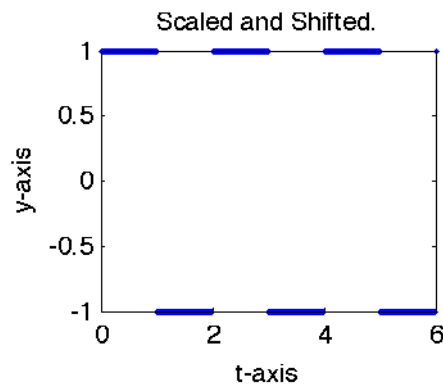


Figure 4.3. A square wave with period 2.

Note that this square wave alternates between the values 1 and -1 . The curve equals 1 on the first half of its period, then -1 on its second half. This pattern then repeats with period 2 over the remainder of its domain.

Fourier Series. Using advanced mathematics, it can be shown that the following *Fourier Series* “converges” to the square wave pictured in **Figure 4.3**

$$\sum_{n=0}^{\infty} \frac{4}{(2n+1)\pi} \sin(2n+1)\pi t \quad (4.4)$$

It is certain remarkable, if not somewhat implausible, that one can sum a series of sinusoidal function and get the resulting square wave picture in **Figure 4.3**.

We will generate and plot the first five terms of the series (4.4), saving each term in a row of matrix A for later use. First, we initialize the time and the number of terms of the series that we will use, then we allocate appropriate space for matrix A . Note how we wisely save the number of points and the number of terms in variables, so that if we later decide to change these values, we won't have to scan every line of our program making changes.

```
N=500;
numterms=5;
t=linspace(0,6,N);
A=zeros(numterms,N);
```

Next, we use a **for** loop to compute each of the first 5 terms (**numterms**) of series (4.4), storing each term in a row of matrix A , then plotting the sinusoid in three space, where we use the angular frequency as the third dimension.

```
for n=0:numterms-1
    y=4/((2*n+1)*pi)*sin((2*n+1)*pi*t);
    A(n+1,:)=y;
    line((2*n+1)*pi*ones(size(t)),t,y)
end
```

We orient the 3D-view, add a box for depth, turn on the grid, and annotate each axis to produce the image shown in **Figure 4.4(a)**.

```
view(20,20)
box on
grid on
xlabel('Angular Frequency')
ylabel('t-axis')
zlabel('y-axis')
```

As one would expect, after examining the terms of series (4.4), each each consecutive term of the series is a sinusoid with increasing angular frequency and decreasing amplitude. This is evident with the sequence of terms shown in **Figure 4.4(a)**.

However, a truly remarkable result occurs when we add the sinusoids in **Figure 4.4(a)**. This is easy to do because we saved each individual term in a row of matrix A . Matlab's **sum** command will sum the columns of matrix A , effectively summing the first five terms of series (4.4) at each instant of time t .

```
figure
plot(t,sum(A))
```

We “tighten” the axes, add a grid, then annotate each axis. This produces the image in **Figure 4.4(b)**. Note the striking similarity to the square wave in **Figure 4.3**.

```
axis tight
grid on
xlabel('t-axis')
ylabel('y-axis')
```

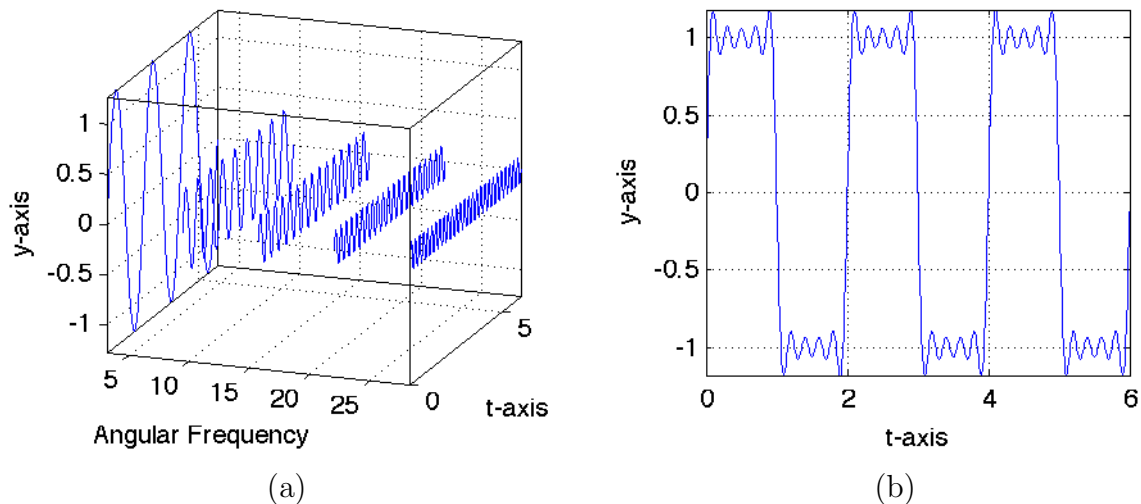


Figure 4.4. Approximating a square wave with a Fourier series (5 terms).

Because we smartly stored the number of terms in the variable **numterms**, to see the effect of adding the first 10 terms of the fourier series (4.4) is a simple matter of changing **numterms=5** to **numterms=10** and running the program again. The output is shown in **Figures 4.5(a)** and (b). In **Figure 4.5(b)**, note the even closer resemblance to the square wave in **Figure 4.3**, except at the ends, where the “ringing” exhibited there is known as *Gibb’s Phenomenon*. If you one

day get a chance to take a good upper-division differential equations course, you will study Fourier series in more depth.

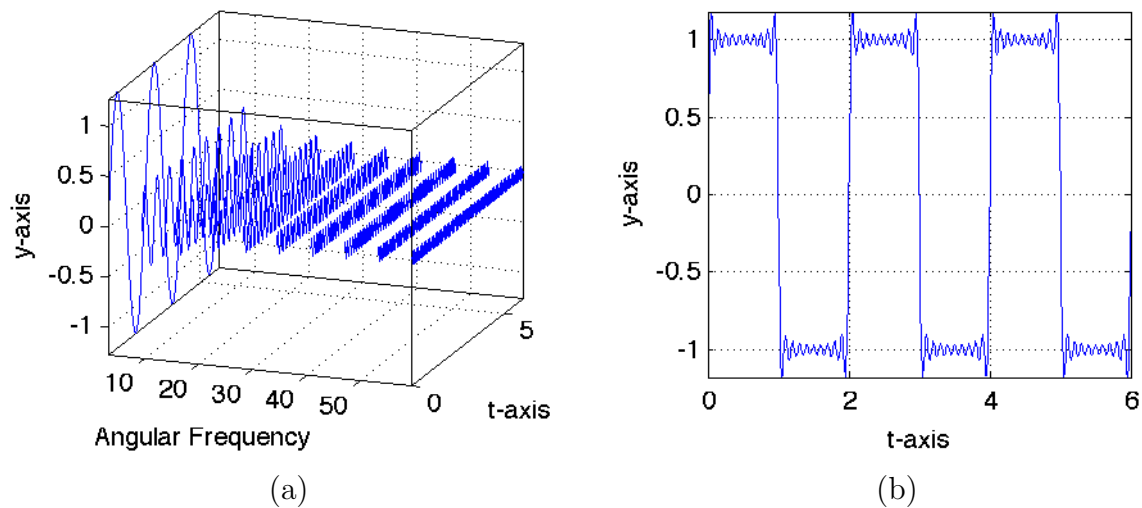


Figure 4.5. Adding additional terms of the Fourier series (4.4).

4.2 Exercises

1. Write a **for** loop that will output the cubes of the first 10 positive integers. Use **fprintf** to output the results, which should include the integer and its cube. Write a second program that uses a **while** loop to produce an identical result.

2. Without appealing to the Matlab command **factorial**, write a **for** loop to output the factorial of the numbers 10 through 15. Use **fprintf** to format the output. Write a second program that uses a **while** loop to produce an identical result. *Hint: Consider the **prod** command.*

3. Write a single program that will count the number of divisors of each of the following integers: 20, 36, 84, and 96. Use **fprintf** to output each result in a form similar to “The number of divisors of 12 is 6.”

4. Set **A=magic(5)**. Write a program that uses nested **for** loops to find the sum of the squares of all entries of matrix *A*. Use **fprintf** to format the output. Write a second program that uses array operations and Matlab’s **sum** function to obtain the same result.

5. Write a program that uses nested **for** loops to produce *Pythagorean Triples*, positive integers *a*, *b* and *c* that satisfy $a^2 + b^2 = c^2$. Find all such triples such that $1 \leq a, b, c \leq 20$ and use **fprintf** to produce nicely formatted results.

6. Another result proved by Leon-

hard Euler shows that

$$\frac{\pi^4}{90} = 1 + \frac{1}{2^4} + \frac{1}{3^4} + \frac{1}{4^4} + \dots$$

Write a program that uses a **for** loop to sum the first 20 terms of this series. Compute the relative error when this sum is used as an approximation of $\pi^4/90$. Write a second program that uses a **while** loop to determine the number of terms required so that the sum approximates $\pi^4/90$ to four significant digits. In both programs, use **fprintf** to format your output.

7. Some attribute the following series to Leibniz.

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

Write a program that uses a **for** loop to sum the first 20 terms of this series. Compute the relative error when this sum is used as an approximation of $\pi/4$. Write a second program that uses a **while** loop to determine the number of terms required so that the sum approximates $\pi/4$ to four significant digits. In both programs, use **fprintf** to format your output.

8. *Goldbach’s Conjecture* is one of the most famous unproved conjectures in all of mathematics, which is remarkable in light of its simplistic statement:

All even integers greater than 2 can be expressed as a sum of two prime integers. Write a program that expresses 1202 as the sum of two primes in ten

different ways. Use **fprintf** to format your output.

9. Here is a simple idea for generating a list of prime integers. Create a vector **primes** with a single entry, the prime integer 2. Write a program to test each integer less than 100 to see if it is a prime using the following procedure.

- i. If an integer is divisible by any of the integers in **primes**, skip it and go to the next integer.
- ii. If an integer is **not** divisible by any of the integers in **primes**, append the integer to the vector **primes** and go to the next integer.

Use **fprintf** to output the vector **primes** in five columns, right justified.

10. There is a famous story about Sir Thomas Hardy and Srinivasa Ramanujan, which Hardy relates in his famous work “A Mathematician’s Apology,” a copy of which resides in the CR library along with the work “The Man Who Knew Infinity: A Life of the Genius Ramanujan.”

I remember once going to see him when he was ill at Putney. I had ridden in taxi cab number 1729 and remarked that the number seemed to me rather a dull one, and that I hoped it was not an unfavorable omen. “No,” he replied, “it is a very interesting number; it is the smallest number expressible as the sum of two cubes in two different ways.”

Write a program with nested loops to

find integers a and b (in two ways) so that $a^3 + b^3 = 1729$. Use **fprintf** to format the output of your program.

11. Write a program to perform each of the following tasks.

- i. Use Matlab to draw a circle of radius 1 centered at the origin and inscribed in a square having vertices $(1, 0)$, $(-1, 1)$, $(-1, -1)$, and $(1, -1)$. The ratio of the area of the circle to the area of the square is $\pi : 4$ or $\pi/4$. Hence, if we were to throw darts at the square in a random fashion, the ratio of darts inside the circle to the number of darts thrown should be approximately equal to $\pi/4$.
- ii. Write a **for** loop that will plot 1000 randomly generated points inside the square. Use Matlab’s **rand** command for this task. Each time random point lands within the unit circle, increment a counter **hits**. When the **for** loop terminates, use **fprintf** to output the ratio darts that land inside the circle to the number of darts thrown. Calculate the relative error in approximating $\pi/4$ with this ratio.

12. Write a program to perform each of the following tasks.

- i. Prompt the user to enter a 3×4 matrix. Store the result in the matrix A .
- ii. Use **if..elseif..else** to provide a menu with three choices: (1) Switch two rows of the matrix; (2) Multiply a row of the matrix by a scalar; and (3) Subtract a scalar multiple of a

row from another row of the matrix.

- iii. Perform the task in the requested menu item, then return the resulting matrix to the user.
 - a. In the case of (1), your program should prompt the user for the rows to switch.
 - b. In the case of (2), your program should prompt the user for a scalar and a row that will be multiplied by the scalar.
 - c. In the case of (3), your program should prompt the user for a scalar and two row numbers, the first of which is to be multiplied by the scalar and subtracted from the second.

13. The solutions of the quadratic equation $ax^2 + bx + c = 0$ are given by the quadratic formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

The discriminant $D = b^2 - 4ac$ is used to predict the number of roots. There are three cases.

1. If $D > 0$, there are two real solutions.
2. If $D = 0$, there is one real solution.
3. If $D < 0$, there are no real solutions.

Write a program that performs each of the following tasks.

- i. The program prompts the user to input a , b , and c .
- ii. The program computes the discriminant D which it uses with the conditional **if..elseif..else** to de-

termine and output the number of solutions.

- iii. The program should also output the solutions, if any.

Use well crafted format strings with **fprintf** to output all results.

14. Plot the modified *sawtooth* curve.

```
N=500;
t=linspace(0,6*pi,N);
y=mod(t+pi,2*pi)-pi;
line(t,y)
```

It can be shown that the Fourier series

$$\sum_{n=1}^{\infty} \frac{2(-1)^{n+1}}{n} \sin nx$$

“converges” to the sawtooth curve. Perform each of the following tasks.

- i. Sketch each of the individual terms in 3-space, as shown in the narrative.
- ii. Sketch the sawtooth, hold the graph, then sum the first five terms of the series and superimpose the plot of the result.

15. Logical arrays are helpful when it comes to drawing piecewise functions. For example, consider the piecewise definition

$$f(x) = \begin{cases} 0, & \text{if } -\pi \leq x < 0 \\ \pi - x, & \text{if } 0 \leq x \leq \pi. \end{cases}$$

Enter the following to plot the piecewise function f .

```
N=500;
x=linspace(-pi,pi,N);
y=0*(x<0)+(pi-x).*(x>=0);
plot(x,y,'.')
```

A Fourier series representation of the function f is given by

$$\frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos nx + b_n \sin nx],$$

where $a_0 = \pi/2$, and

$$a_n = \frac{1 - \cos n\pi}{n^2\pi} \quad \text{and} \quad b_n = \frac{1}{n}.$$

Hold the plot of f , then superimpose the Fourier series sum for $n = 1$ to $n = 5$. *Hint: This is a nice place for anonymous functions, for example, set:*

```
b = @(n) 1/n;
```

4.2 Answers

1. This **for** loop will print the cubes of the first 10 positive integers.

```
for k=1:10
    fprintf('The cube of %d is %d.\n',k,k^3)
end
```

This **while** loop will print the cubes of the first 10 positive integers.

```
k=1;
while k<=10
    fprintf('The cube of %d is %d.\n',k,k^3)
    k=k+1;
end
```

3. The following program will print the number of divisors of 20, 36, 84, and 96.

```
for k=[20, 36, 84, 96]
    count=0;
    divisor=1;
    while (divisor<=k)
        if mod(k,divisor)==0
            count=count+1;
        end
        divisor=divisor+1;
    end
    fprintf('The number of divisors of %d is %d.\n',k,count)
end
```

5. This program will print Pythagorean Triples so that $1 \leq a, b, c \leq 20$.

```

N=20;
for a=1:N
    for b=1:N
        for c=1:N
            if (c^2==a^2+b^2)
                fprintf('Pythagorean Triple: %d, %d, %d\n', a, b, c)
            end
        end
    end
end
end

```

7. The following loop will sum the first 20 terms.

```

N=20;
s=0;
for k=1:N;
    s=s+(-1)^(k+1)/(2*k-1);
end

```

Compute the relative error in approximating $\pi/4$ with the sum of the first 20 terms.

```

rel=abs(s-pi/4)/abs(pi/4);

```

Output the results.

```

fprintf('The actual value of pi/4 is %.6f.\n',pi/4)
fprintf('The sum of the first %d terms is %f.\n',N,s)
fprintf('The relative error is %.2e.\n',rel)

```

9. The `mod(m,primes)==0` comparison will produce a logical vector which contains a 1 in each position that indicates that the current number m is divisible by the number in the `prime` vector in the corresponding position. If “any” of these are 1’s, then the number m is divisible by at least one of the primes in the current list `primes`. In that case, we continue to the next value of m . Otherwise, we append m to the list of primes.

```
primes=2;
for m=3:100
    if any(mod(m,primes)==0)
        continue
    else
        primes=[primes,m];
    end
end
end
```

We create a format of 5 fields, each of width 5, then **fprintf** the vector **primes** using this format.

```
fmt='%5d %5d %5d %5d %5d\n';
```

The resulting list of primes is nicely formatted in 5 columns.

```
 2    3    5    7   11
13   17   19   23   29
31   37   41   43   47
53   59   61   67   71
73   79   83   89   97
```

11. We open a figure window and set a lighter background color of gray.

```
fig=figure;
set(fig,'Color',[0.95,0.95,0.95])
```

We draw a circle of radius one, increase its line width, change its color to dark red, set the axis equal, then turn the axes off.

```
t=linspace(0,2*pi,200);
x=cos(t);
y=sin(t);
line(x,y,'LineWidth',2,'Color',[0.625,0,0])
axis equal
axis off
```

We draw the square in counterclockwise order of vertices, starting at the point (1, 1). In that order, we set the x - and y -values of the vertices in the vectors \mathbf{x} and \mathbf{y} . We then draw the square with a thickened line width and color it dark blue.

```
x=[1,-1,-1,1,1]; y=[1,1,-1,-1,1];
line(x,y,'LineWidth',2,'Color',[0,0,0.625])
```

Next, we set the number of darts thrown at 1000. Then we plot one dart at the origin, change its linestyle to 'none' (points will not be connected with line segments), choose a marker style, and color it dark blue.

```
N=1000;
dart=line(0,0,...
    'LineStyle','none',...
    'Marker','.',...
    'Color',[0,0,0.625]);
```

In the loop that follows, we use the handle to this initial dart to add x - and y -data (additional darts). First, we set the number of **hits** (darts that land inside the circle) to zero (we don't count the initial dart at (0, 0)).

```
hits=0;
for k=1:N
    x=2*rand-1;
    y=2*rand-1;
    set(dart,...
        'XData',[get(dart,'XData'),x],...
        'YData',[get(dart,'YData'),y])
    %drawnow
    if (x^2+y^2<1)
        hits=hits+1;
    end
end
```

In the body of the loop, we choose uniform random numbers between -1 and 1 for both x and y . We then add these new x - and y -values to the **XData** and **YData** of the existing dart by using its handle **dart**. We do this for x by

first “getting” the **XData** for the dart with `get(dart,'XData')`, then we append the current value of x with `[get(dart,'XData'),x]`. The `set` command is used to set the updated list of x -values. A similar task updates the dart's y -values. At each iteration of the loop, we also increment the **hits** counter if the dart lies within the border of the circle, that is, if $x^2 + y^2 < 1$. You can uncomment the `drawnow` command to animate the dart throwing at the expense of waiting for 1000 darts to be thrown to the screen.

Finally, we output the requested data to the command window.

```
fprintf('%d of %d darts fell inside the circle.\n',hits,N)
fprintf('The ratio of hits to throws is %f\n\n',hits/N)

fprintf('The ratio of the area of the circle to the area\n')
fprintf('the area of the square is pi/4, or approximately %f.\n\n',pi/4)

rel_err=abs(hits/N-pi/4)/abs(pi/4);
fprintf('The relative error in approximating pi/4 wiht the ratio\n')
fprintf('of hits to total darts thrown is %.2e.\n',rel_err)
```

15. We begin by using logical relations to craft the piecewise function.

```
N=500;
x=linspace(-pi,pi,N);
y=0*(x<0)+(pi-x).*(x>=0);
line(x,y,...
      'LineStyle','none',...
      'Marker','.')
```

Next, we compose two anonymous functions for the coefficients a_n and b_n .

```
a=@(n) (1-cos(n*pi))/(n^2*pi);
b=@(n) 1/n;
```

Note that $a_0 = \pi/2$, so $a_0/2 = \pi/4$, which must be added to the sum of the first five terms of the series. Thus, we'll start our running sum at $s = \pi/4$. The `for` loop then sums the first five terms. We use another `line` command to plot the result.

```
s=pi/4;  
N=5;  
for k=1:N  
    s=s+(a(k)*cos(k*x)+b(k)*sin(k*x));  
end  
line(x,s,'LineWidth',2,'Color',[0.625,0,0])
```

