4.1 Logical Arrays

We begin this section by demonstraing how Matlab determines the truth or falsehood of a statement. Enter the following array at the Matlab prompt.

```
>> x=[true false]
x =
     1     0
```

Note that that **true** evaluates to 1, while false evaluates to zero. Moreover, the array stored in the variable **x** in an entirely new type of datatype, called a *logical array*.

```
>> whos
  Name      Size                    Bytes  Class

  x         1x2                         2  logical array
```

Note that each entry in the logical vector **x** takes one byte of storage. Note that this logical type is a completely new datatype. Indeed, it is instructive to compare the vector **x** with the following vector **y**

```
>> y=[1 0]
y =
     1     0
```

On the surface, it appears that the variables **x** and **y** contain exactly the same data, that is until one checks the datatypes with Matlab's **whos** command.

```
>> whos
  Name      Size                    Bytes  Class

  x         1x2                         2  logical array
  y         1x2                        16  double array
```

[1] Copyrighted material. See: http://msenux.redwoods.edu/Math4Textbook/

Note that vector **y** is of class double, using 8 bytes to store each entry.

Alternatively, we can check the class with Matlab's **class** command. On the one hand, the command

```
>> class(x)
ans =
logical
```

informs us that vector **x** has class **logical**. On the other hand, the command

```
>> class(y)
ans =
double
```

informs us that the vector **y** has class **double**.

Like some of the other conversion operators we've seen (like **uint8**), Matlab provides an operator that will convert numeric arrays to logical arrays. Any nonzero real number is converted to a logical 1 (true) and zeros are converted to logical 0 (false).

To see this in action, first enter the following matrix A.

```
>> A=[0 1 3;4 0 0; 5 7 -11]
A =
     0     1     3
     4     0     0
     5     7   -11
```

Note that A has class **double**.

```
>> class(A)
ans =
double
```

Now, convert A to a logical array.

```
>> A=logical(A)
Warning: Values other than 0 or 1 converted to logical 1.
A =
     0     1     1
     1     0     0
     1     1     1
```

Note that all nonzero entries in the original matrix A are now logical 1, while all zeros in the original matrix A are now logical zero. You can test the new class of matrix A with the following command.

```
>> class(A)
ans =
logical
```

Relational Operators

Matlab provides a complete list of *relational operators* (see **Table 4.1**).

Operator	Description
$<$	Less than
$\leq$	Less than or equal to
$>$	Greater than
$\geq$	Greater than or equal to
$==$	Equal to
$\sim=$	Not equal to

Table 4.1. Matlab's relational operators.

The Matlab relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. That is, they are "array smart."

For example, consider the vectors **v** and **w**.

```
>> v=[1 3 6]
v =
     1     3     6
>> w=[1 2 4]
w =
     1     2     4
```

The output of the command **v>w** will return a logical vector of the same size as the vectors **v** and **w**. Locations where the specified relation is true receive a 1 and locations where the specified relation is false receive a 0.

```
>> v>w
ans =
     0     1     1
```

Because the first entry of **v** is not larger than the first entry of **w**, the first entry of the answer receives a zero (false). Because the second and third entries of the vector **v** are larger than the second and third entries of the vector **w**, the second and third entries of the answer receive a 1 (true). This is what we mean when we say "Relational operators always operate element-by-element."

Like the other array operators we've seen that act in an element-by-elemnt manner, the arrays being compared must have the same size.

```
>> u=[10 10]
u =
    10    10
>> u>v
??? Error using ==> gt
Matrix dimensions must agree.
```

Comparing with a scalar. One exception to the "same size rule" is when an array is compared to a scalar. In this case, Matlab tests the scalar against every element of the other operand.

```
>> u=rand(1,6)
u =
    0.9501    0.2311    0.6068    0.4860    0.8913    0.7621
>> u>0.5
ans =
     1     0     1     0     1     1
```

Note that the first, third, fifth, and sixth entries of the vector **u** are all greater than 0.5, so the answer receives a 1 (true) in these positions, with the remaining positions receiving a 0 (false).

How Logical Arrays are Used

Matlab uses logical arrays for a type of indexing. A *logical index* designates the elements of a matrix A based on their position in the indexing array, B. In this masking type of operation, every true element in the indexing array is treated as a positional index into the array being accessed

For example, create a matrix A of uniform random numbers.

```
>> A=rand(5)
A =
    0.4565    0.7919    0.9355    0.3529    0.1987
    0.0185    0.9218    0.9169    0.8132    0.6038
    0.8214    0.7382    0.4103    0.0099    0.2722
    0.4447    0.1763    0.8936    0.1389    0.1988
    0.6154    0.4057    0.0579    0.2028    0.0153
```

Next, find the positions of the random numbers that are less than 0.5, and store the result in the matrix B.

```
>> B=A<0.5
B =
     1     0     0     1     1
     1     0     0     0     0
     0     0     1     1     1
     1     1     0     1     1
     0     1     1     1     1
```

Take a moment to check that in each position in B that contains a logical 1 (true), the corresponding position in matrix A contains a uniform random number that is less than 0.5.

There are a number of things that we can do with this information. We could list all the random numbers in A that are less than 0.5 by using matrix B as a logical index into the matrix A.[2]

```
>> A(B)
ans =
    0.4565
    0.0185
    0.4447
    0.1763
    0.4057
    0.4103
    0.0579
    0.3529
    0.0099
    0.1389
    0.2028
    0.1987
    0.2722
    0.1988
    0.0153
```

Or we could set each of these positions to zero.

```
>> A(B)=0
A =
         0    0.7919    0.9355         0         0
         0    0.9218    0.9169    0.8132    0.6038
    0.8214    0.7382         0         0         0
         0         0    0.8936         0         0
    0.6154         0         0         0         0
```

[2] Matlab has roots that began their growth in Fortran, where the entries in arrays were loaded by column. The Matlab command **A(:)** will list all the entries in A, first going down the first column, then the second, etc. With the command **A(B)**, Matlab first lists all the "true" positions in the first column, then the second, etc.

A number of Matlab's functions are designed to execute a test of some sort, then return a logical array. The output of these commands is usually used as a mask or index. For example, create a *magic square* with the following command.

```
>> A=magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

We'll find all the primes in matrix A with Matlab's **ispime** command.

```
>> B=isprime(A)
B =
     1     0     0     0     0
     1     1     1     0     0
     0     0     1     0     0
     0     0     1     0     1
     1     0     0     1     0
```

Each position in matrix B that contains a logical 1 (true) indicates a prime in the corresponding position of matrix A. We can get a listing of the primes in matrix A.

```
 >> A(B)
ans =
    17
    23
    11
     5
     7
    13
    19
     2
     3
```

Or we can replace all of the primes in Matrix A with -1 with this command.

```
>> A(B)=-1
A =
    -1    24     1     8    15
    -1    -1    -1    14    16
     4     6    -1    20    22
    10    12    -1    21    -1
    -1    18    25    -1     9
```

A Practical Example. Although the examples provided thus far provide explanation on how logical arrays are used as indices, they might not seem to have much practical use. Let's look at some examples where logical indexing is helpful in a much more practical way.

▶ **Example 1.** *Plot the graph of* $y = \sqrt{x^2 - 1}$ *over the interval* $[-5, 5]$.

Seems like a simple request.

```
>> x=linspace(-5,5,100);
>> y=sqrt(x.^2-1);
>> plot(x,y)
```

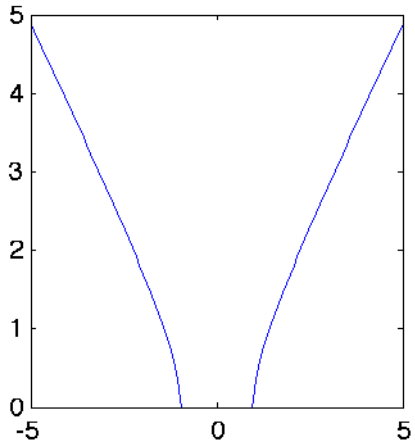


Figure 4.1. Plotting of the graph of $y = \sqrt{x^2 - 1}$.

Seems to have worked (see **Figure 4.1**(a)), but we do receive a warning at the command line when we run this script.

```
Warning: Imaginary parts of complex X and/or Y arguments ignored
```

Complex numbers have seemingly crept into our calculations. They certainly don't exist in the vector **x**, because the command **x=linspace(-5,5,200)** generates **real** numbers between -5 and 5. It must be the vector y that contains the complex numbers, but how to find them?

Real and imaginary parts of complex numbers. A complex number has the form $a + bi$, where $i = \sqrt{-1}$. The *real part* of the complex number $a + bi$ is the number a. The *imaginary part* of the complex number $a + bi$ is the number b. Matlab know all about the real and imaginary parts of complex numbers.

First, check that the variable i contains the complex number i. We do this because i is a popular variable to use in scripts and **for** loops and may have been set to something other than the complex number i.

```
>> i
ans =
        0 + 1.0000i
```

If you don't get this result, type **clear i** and try again.

Enter $z = 3 + 4i$.

```
>> z=3+4i
z =
   3.0000 + 4.0000i
```

The real part of $z = 3 + 4i$ is 3.

```
>> real(z)
ans =
     3
```

The imaginary part of $z = 3 + 4i$ is 4.

```
>> imag(z)
ans =
     4
```

Now, a number r is **real** if and only if the real part of r is identical to the number r. For example, set $r = 4$.

```
>> r=4
r =
     4
```

Now test if its real part is identical to itself.

```
>> real(r)==r
ans =
     1
```

Note that the logical 1 indicates a "true" response. The real part of r equals r. Thus, r is a real number.

On the other hand, set $z = 2 - 3i$.

```
>> z=2-3i
z =
   2.0000 - 3.0000i
```

Test if the real part of z equals z.

```
>> real(z)==z
ans =
     0
```

Note that the logical 0 indicates a "false" response. The real part of z doesn't equal z. Thus, z is not a real number, z is complex.

Let's use this idea to find the positions in the vector **y** that hold complex numbers.

```
>> k=real(y)~=y
k =
  Columns 1 through 10
     0     0     0     0     0     0     0     0     0     0
  Columns 11 through 20
     0     0     0     0     0     0     0     0     0     0
  Columns 21 through 30
     0     0     0     0     0     0     0     0     0     0
  Columns 31 through 40
     0     0     0     0     0     0     0     0     0     0
  Columns 41 through 50
     1     1     1     1     1     1     1     1     1     1
  Columns 51 through 60
     1     1     1     1     1     1     1     1     1     1
  Columns 61 through 70
     0     0     0     0     0     0     0     0     0     0
  Columns 71 through 80
     0     0     0     0     0     0     0     0     0     0
  Columns 81 through 90
     0     0     0     0     0     0     0     0     0     0
  Columns 91 through 100
     0     0     0     0     0     0     0     0     0     0
```

The vector **k** is a logical vector, having logical 1 (true) in each position that y has a complex number. We can use this as an index into the vector **x** to see what values of x are causing complex numbers to appear in the vector **y**.

```
>> x(k)
ans =
  Columns 1 through 6
   -0.9596   -0.8586   -0.7576   -0.6566   -0.5556   -0.4545
  Columns 7 through 12
   -0.3535   -0.2525   -0.1515   -0.0505    0.0505    0.1515
  Columns 13 through 18
    0.2525    0.3535    0.4545    0.5556    0.6566    0.7576
  Columns 19 through 20
    0.8586    0.9596
```

It would appear that values of x between -1 and 1 are creating complex values in the vector **y**. This makes sense because the domain of the function $y = \sqrt{x^2 - 1}$

requires that $x^2 - 1 \geq 0$. Solving for x, $|x| \geq 1$, or equivalently, $x \leq -1$ or $x \geq 1$. For values of x between -1 and 1, the expression $x^2 - 1$ is negative, so the square root produces complex numbers. We can see the complex numbers in **y** with logical indexing.

```
>> y(k)
ans =
  Columns 1 through 3
        0 + 0.2814i        0 + 0.5127i        0 + 0.6527i
  Columns 4 through 6
        0 + 0.7543i        0 + 0.8315i        0 + 0.8907i
  Columns 7 through 9
        0 + 0.9354i        0 + 0.9676i        0 + 0.9885i
  Columns 10 through 12
        0 + 0.9987i        0 + 0.9987i        0 + 0.9885i
  Columns 13 through 15
        0 + 0.9676i        0 + 0.9354i        0 + 0.8907i
  Columns 16 through 18
        0 + 0.8315i        0 + 0.7543i        0 + 0.6527i
  Columns 19 through 20
        0 + 0.5127i        0 + 0.2814i
```

Note that the real part of each of these complex numbers is zero.

So, how does Matlab's **plot** command handle complex numbers. The answer is found by typing **help plot**. The relevant part of the helpfile follows.

```
PLOT(Y) plots the columns of Y versus their index.
    If Y is complex, PLOT(Y) is equivalent to
    PLOT(real(Y),imag(Y)). In all other uses of PLOT,
    the imaginary part is ignored.
```

In the case at hand, we used **plot(x,y)**, so this paragraph of the help file for the **plot** command would indicate that Matlab simply ignores the imaginary parts of the vector y when the entries are complex. For the output of **y(k)** above, when Matlab ignores the imaginary part, that is equivalent to saying that Matlab considers all of these complex numbers to equal their real part, or zero. This can be seen by adding the command **axis([-5,5,-5,5])** to produce the image in **Figure 4.2**.

```
>> axis([-5,5,-5,5])
```

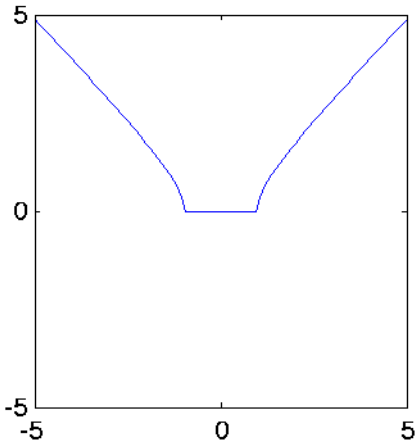


Figure 4.2. The **axis** command reveals that the **plot** command believes that all y-values equal zero for values of x between -1 and 1.

Note the horizontal line connecting the right- and left-hand halves of the hyperbola in **Figure 4.2**. This line simply does not belong in this image, as x-values between -1 and 1 are not in the domain of of $y = \sqrt{x^2 - 1}$. However, Matalb simply ignores the imaginary part of the complex numbers in the vector **y**, considers them to be zero, and plots this "false" horizontal line.

Let's get rid of this "false" horizontal line by setting all the offending complex number entries in the vector **y** to **NaN** (Matlab's "Not a Number.")

```
>> y(k)=NaN;
```

If you remove the suppressing semicolon at the end of this command, you will be able to see that all the complex numbers in the vector **y** have been replaced by **NaN**.

The good news is the fact that Matlab's **plot** command ignores these **NaN**'s and does not plot them. The following command will produce the image in **Figure 4.3**.

```
>> plot(x,y)
>> axis([-5,5,-5,5])
```

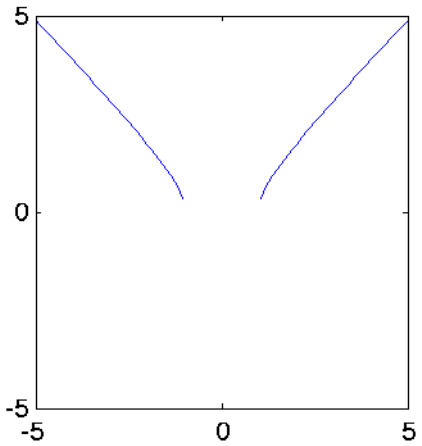


Figure 4.3. Removing the complex numbers from the plot.

Preplanning. All of these contortions with complex numbers can be avoided with proper preplanning. As we've said, the domain of $y = \sqrt{x^2 - 1}$ is $\{x : x \leq -1 \text{ or } x \geq 1\}$.

We'll start by defining an anonymous function, making it "array smart."

```
>> f=@(x) sqrt(x.^2-1)
f =
    @(x) sqrt(x.^2-1)
```

Test the anonymous function at $x = 0$, where $f(0) = \sqrt{0^2 - 1} = \sqrt{-1} = i$.

```
>> f(0)
ans =
        0 + 1.0000i
```

Note that $f(0) = i$ is a correct response.

We'll first plot the left-hand branch of the hyperbola on $\{x : x \leq -1\}$.

```
>> x=linspace(-5,-1);
>> y=f(x);
>> plot(x,y)
```

Because each of the entries in the vector **x** lie in the domain of the function $y = \sqrt{x^2 - 1}$, no complex numbers are produced when evaluating the function on the entries in **x**.

Next, we'll plot the right-hand branch of the hyperbola on $\{x : x \geq 1\}$. Again, no complex numbers are produced because we are using x-values that lie in the domain of the function.

```
>> x=linspace(1,5);
>> y=f(x);
>> line(x,y)
```

Remember that the **line** command is used to append to an existing plot. We could also have used a **hold on**, followed by a **plot(x,y)**, but the **line** command does not require that we "hold" the plot.

Finally, adjust the window boundaries with the **axis** command.

```
>> axis([-5,5,-5,5])
```

The result is an image indentical to that in **Figure 4.3**.

Complex numbers are not so easily avoided in other situations. Let's look at an example that uses Matlab's **mesh** command.

▶ **Example 2.** *Sketch the surface* $z = \sqrt{x^2 - y^2}$ *on the rectangular domain* $D = \{(x, y) : -2 \leq x, y \leq 2\}$.

Let's again use an anonymous function, making it "array smart."

```
>> f=@(x,y) sqrt(x.^2-y.^2)
f =
    @(x,y) sqrt(x.^2-y.^2)
```

Test the anonymous function. Note that $f(5,3) = \sqrt{5^2 - 3^2} = 4$.

```
>> f(5,3)
ans =
     4
```

That looks correct.

Now, the following commands produce the image in **Figure 4.4**(a). At first glance, it appears that all is well.

```
>> [x,y]=meshgrid(linspace(-2,2,50));
>> z=f(x,y);
>> mesh(x,y,z)
```

However, this time Matlab ignores the imaginary part of the complex numbers in **z** without even providing a warning. This is evident in the graph of $z = \sqrt{x^2 - y^2}$ where you see that a large number of points in the mesh have z-value equal to zero. These points in the xy-plane are where Matlab is ignoring the imaginary part of the complex entries in z and plotting only the real part (which is zero).

We can check to see if the matrix z contains any complex numbers.

```
>> isreal(z)
ans =
     0
```

The false response indicates the presence of complex numbers in matrix z. We can replace all the complex entries in z with these commands.

```
>> k=real(z)~=z;
>> z(k)=NaN;
```

You may want to remove the suppressing semicolons and view the output of the last two commands.

Matlab's **mesh** command refuses to plot the **Nan**'s and produces the image in **Figure 4.4**(b).

```
>> mesh(x,yz)
```

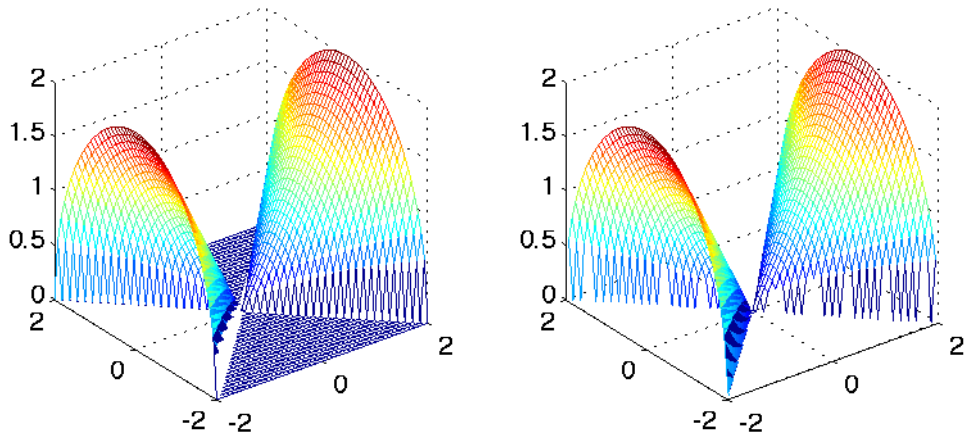


Figure 4.4. Removing the complex entries removes the false zero plane.

Logical Operators

The Matlab symbols **&**, **|**, and **~** are the logical operators **AND**, **OR**, and **NOT**, respectively. Truth tables for each logical operator are gathered in **Table 4.2**. Remember that logical 1 is true and logical 0 is false.

- The statment "A and B" is true if and only if both statements A and B are true. This is reflected in the truth table for **&** (**AND**) in **Table 4.2**(a).
- The statement "A or B" is true if and only if one or the other of statements A and B are true. This is reflected in te truth table for **|** (**OR**) in **Table 4.2**(b).
- The statement "not A" simply reverses the truth or falsehood of statement A. This is reflected in the truth table for **~** (**NOT**) in **Table 4.2**(c).

Each of the logical operators **&**, **|**, and **~** acts *element-wise*. Moreover, the output is of type logical. For example, create two logical vectors **u** and **v**.

```
>> u=logical([1 1 1 0 0]);
>> v=logical([1 0 1 0 1]);
```

A	B	$A \,\&\, B$
1	1	1
1	0	0
0	1	0
0	0	0

(a) **AND**

A	B	$A \mid B$
1	1	1
1	0	1
0	1	1
0	0	0

(b) **OR**

A	$\sim A$
1	0
0	1

(c) **NOT**

Table 4.2. Truth tables for Matlab's logical operators.

The first and third entries of vectors **u** and **v** are logical 1's (true), hence the first and third entries of **u&v** will be logical 1's (true). According to **Table 4.2**(a), all other entries of **u&v** will be logical 0's (false).

```
>> u, v, u&v
u =
     1     1     1     0     0
v =
     1     0     1     0     1
ans =
     1     0     1     0     0
```

On the other hand, **Table 4.2**(b) indicates that **u|v** (**OR**) will be false if and only if both entries are false. In the case of the vectors **u** and **v**, the fourth entry of each vector is logical 0 (false), so the fourth entry of **u|v** will be logical 0 (false). All other entries of **u|v** will be logical 1's (true).

```
>> u, v, u|v
u =
     1     1     1     0     0
v =
     1     0     1     0     1
ans =
     1     1     1     0     1
```

Finally, **~u** simply reverses the truth or falsehood of each entry of the vector **u**.

```
>> u, ~u
u =
     1     1     1     0     0
ans =
     0     0     0     1     1
```

If the vectors and/or matrices involved are not logical arrays, the logical operators first convert the operands to logical arrays. First, create a "magic" matrix A.

```
>> A=magic(3)
A =
     8     1     6
     3     5     7
     4     9     2
```

Create a *diagonal matrix* D with the vector $[5, 6, 7]$ on its main diagonal.

```
>> D=diag([5,6,7])
D =
     5     0     0
     0     6     0
     0     0     7
```

We could first convert these to logical matrices and then "and" them.

```
>> LA=logical(A), LD=logical(D), LA&LD
Warning: Values other than 0 or 1 converted to logical 1.
LA =
     1     1     1
     1     1     1
     1     1     1
Warning: Values other than 0 or 1 converted to logical 1.
LD =
     1     0     0
     0     1     0
     0     0     1
ans =
     1     0     0
     0     1     0
     0     0     1
```

However, this is not necessary, because the operator **&** will convert each of its operands to a logical type before performing the **AND** operation.

```
>> A, D, A&D
A =
     8     1     6
     3     5     7
     4     9     2
D =
     5     0     0
     0     6     0
     0     0     7
ans =
     1     0     0
     0     1     0
     0     0     1
```

Finally, because the logical operators act element-wise, the arrays used as operands must have the same dimensions.

```
>> u=[2,3], v=[4 5 6], u|v
u =
     2     3
v =
     4     5     6
??? Error using ==> or
Inputs must have the same size.
```

Let's look at some powerful applications of these operators.

▶ **Example 3.** *Find all entries of matrix* **magic(5)** *that are* **not** *prime.*

Start by entering the matrix and using Matlab's **isprime** function to find the prime entries.

```
>> M=magic(5); B=isprime(M);
```

We saw in an earlier example that **M(B)** will list all of the primes in the matrix M. So, let's use the logical operator ~ to list all the nonprimes. For convenience and purposes of comparison, we list the matrix M.

```
>> M
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Now we list all the nonprimes in the matrix M.

```
>> M(~B)
ans =
     4
    10
    24
     6
    12
    18
     1
    25
     8
    14
    20
    21
    15
    16
    22
     9
```

Even more succinctly, try the command **M(~isprime(M))** to achieve the same result.

Let's look at another example.

▶ **Example 4.** *Sketch the surface* $z = x^2 + y^2$ *on the rectangular domain* $d = \{(x, y) : -1 \leq x, y \leq 1\}$, *but only for those pairs* (x, y) *that satisfy* $y < x$ *and* $y > -x$.

Let's again use an anonymous function, making it "array smart."

```
>> f=@(x,y) x.^2+y.^2
f =
    @(x,y) x.^2+y.^2
```

Test the function. Note that $f(3, 4) = 3^2 + 4^2 = 25$.

```
>> f(3,4)
ans =
    25
```

That checks.

Next, create the rectangular grid and evaluate z at each point (x, y) in the grid.

```
[x,y]=meshgrid(linspace(-1,1,50));
z=f(x,y);
```

Now, find those values of the grid that satisfy $y < x$ and $y > -x$.

```
k=(y<x) & (y>-x);
```

We want to eliminate from the mesh all points that do **not** satisfy this requirement.

```
z(~k)=NaN;
```

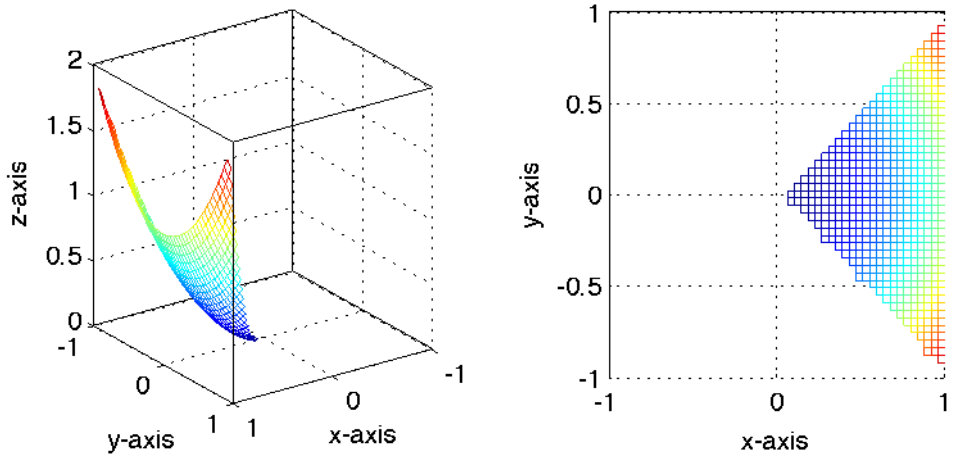
We can now draw the resulting mesh, label the axes, and provide an orientation.

```
mesh(x,y,z)
xlabel('x-axis')
ylabel('y-axis')
zlabel('z-axis')
box on
view(145,20)
```

The result is shown in **Figure 4.5**(a).

An interesting view is provided by the following command, the result of which is shown in **Figure 4.5**(b). In this view, we've declined to rotate the xy-axes, but elevated our eye 90°, so that we are gazing directly down at the xy-plane. This view clearly shows that we have sketched the surface of a region in the rectangular domain restricted to (x, y) pairs satisfying $y < x$ and $y > -x$.

```
view(0,90)
```



(a) (b)

Figure 4.5. The surface $z = x^2 + y^2$ restricted to points where $y < x$ and $y > -x$.

4.1 Exercises

In **Exercises 1-6**, first enter the following vectors.

```
>> v=[1,3,5,7], w=[5,2,5,4]
v =
     1     3     5     7
w =
     5     2     5     4
```

In each exercise, first predict the output of the given command, then validate your response with the appropriate Matlab command. *Note: The idea here is not to simply enter the command. Rather, spend some time thinking, then predict the output before you enter the command to verify your conclusion.*

1. `v>w`

2. `v>=w`

3. `v<=w`

4. `v<w`

5. `v==w`

6. `v~=w`

Store the following "magic" matrix in the variable A.

```
>> A=magic(5)
A =
  17   24    1    8   15
  23    5    7   14   16
   4    6   13   20   22
  10   12   19   21    3
  11   18   25    2    9
```

In **Exercises 7-12**, first predict the output of the given command, then validate your response with the appropriate Matlab command. *Note: The idea here is not to simply enter the command. Rather, spend some time thinking, then predict the output before you enter the command to verify your conclusion.*

7.

```
>> A>14
```

8.

```
>> A<=12
```

9.

```
>> (A>3) & (A<=20)
```

10.

```
>> (A<5) | (A>=21)
```

11.

```
>> ~(A<=6)
```

12.

```
>> (A>6) & ~(A>8)
```

---

13. Set **P=pascal(5)**. Write a Matlab command that will list all entries of matrix P that are less than 20 but not equal to 1.

14. Set **P=pascal(5)**. Write a Matlab command that will list all entries of matrix P that are not equal to 1.

15. Set **P=pascal(5)**. Write a Matlab command that will list all entries of matrix P that are less than or equal to 4 or greater than 20.

16. Set **P=pascal(5)**. Write a Matlab command that will list all entries of matrix P that are greater than 3 but not greater than 35.

When a and b are positive integers, the Matlab command **mod(a,b)** returns the remainder when a is divided by b. Use this command to produce the requested entries of the given matrix in **Exercises 17-20**.

17. Set **A=magic(4)**. Write a Matlab command that will return all entries of matrix A that are divisible by 2. *Hint: An integer k is divisible by 2 if the remainder is zero when k is divided by 2.*

18. Set **A=magic(4)**. Write a Matlab command that will return all entries of matrix A that are odd integers. *Hint: A integer k is odd if its remainder is 1 when k is divided by 2.*

19. Set **A=magic(4)**. Write a Matlab command that will return all entries of matrix A that are divisible by 3.

20. Set **A=magic(4)**. Write a Matlab command that will return all entries of matrix A that are divisible by 5.

---

21. Execute the following command to list the primes less than or equal to 100.

```
>> primes(100)
```

Secondly, create a vector **u** holding the integers from 1 to 100, inclusive. Use Matlab's **isprime** command and logical indexing to pick out and list all the primes in vector **u**. Compare the results.

22. Create a vector **u** holding the integers from 100 to 1000, inclusive. Use Matlab's **isprime** command and logical indexing to pick out all the primes in vector **u**. Use Matlab's **max** command on the result to find the largest

prime between 100 and 1000.

In **Exercises 23-26**, perform each of the following tasks for the given function.

i. Write an "array smart" anonymous function f for the given function. Test your anonymous function before proceeding.
ii. Set **x=linspace(-10,10,200)** and evaluate the function with **y=f(x)**.
iii. Use the **plot(x,y)** command to plot the function.
iv. Use **axis([-10,10,-10,10])** to set the window boundaries.
v. Use logical indexing to set all of the complex entries in the vector **y** to **NaN**. Open a second figure window with the command **figure**. Replot the result and reset the window boundaries as above, if necessary. Add axis labels and a title and turn the grid on.

23. $f(x) = 2 + \sqrt{x+5}$

24. $f(x) = 3 - \sqrt{x-3}$

25. $f(x) = \sqrt{9-x^2}$

26. $f(x) = \sqrt{x^2-25}$

In **Exercises 27-30**, use "advanced planning" to plot the given function on a subset of the domain $[-10, 10]$ to avoid complex entries when evaluating the given function. In each case, set the window boundaries with the command **axis([-10,10,-10,10])**, turn on the grid, and add axes labels and a title.

27. The function in **Exercise 23**.

28. The function in **Exercise 24**.

29. The function in **Exercise 25**.

30. The function in **Exercise 26**.

In **Exercises 31-34**, perform each of the following tasks for the given function.

i. Write an "array smart" anonymous function f for the given function. Test your anonymous function before proceeding.
ii. Set:

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function with **z=f(x,y)**.
iii. Use **mesh(x,y,z)** to plot the surface defined by the function.
iv. Use logical indexing to set all of the complex entries in the vector **z** to **NaN**. Open a second figure window with the command **figure**. Replot the surface. Add axis labels and a title.

31. $f(x, y) = \sqrt{1+x}$

32. $f(x, y) = \sqrt{1-y}$

33. $f(x, y) = \sqrt{9-x^2-y^2}$

34. $f(x, y) = \sqrt{x^2+y^2-1}$

In **Exercises 35-40**, perform each of the following tasks for the given func-

tion.

i. Write an "array smart" anonymous function f for the given function. Test your anonymous function before proceeding.
ii. Set:

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function with **z=f(x,y)**.
iii. Use logical indexing to replace all entries in **z** with **NaN** that do not satisfy the given constraint.
iv. Use **mesh(x,y,z)** to plot the surface defined by the function and constraint. Add axis labels and a title.
v. Open a second figure window with the command **figure**. Replot the surface and orient the view with **view(0,90)**. Add axis labels and a title. Does the pictured region satisfy the given constraint?

35. $f(x,y) = 12 - x - y$ where

$$x > -1.$$

36. $f(x,y) = 10 - 2x + y$ where

$$y \leq 3.$$

37. $f(x,y) = 14 + x - 2y$ where

$$x + y < 0.$$

38. $f(x,y) = 14 + x - 2y$ where

$$x - y \geq 0.$$

39. $f(x,y) = 11 - 2x + 2y$ where

$$x + y < 0 \quad \text{and} \quad x \geq -2.$$

40. $f(x,y) = 10 + 2x - 3y$ where

$$x - 2y \leq 0 \quad \text{or} \quad y \leq 0.$$

4.1 Answers

1.

```
>> v>w
ans =
     0     1     0     1
```

3.

```
>> v<=w
ans =
     1     0     1     0
```

5.

```
>> v==w
ans =
     0     0     1     0
```

7.

```
>> A>14
ans =
   1     1     0     0     1
   1     0     0     0     1
   0     0     0     1     1
   0     0     1     1     0
   0     1     1     0     0
```

9.

```
>> (A>3) & (A<=20)
ans =
   1     0     0     1     1
   0     1     1     1     1
   1     1     1     1     0
   1     1     1     0     0
   1     1     0     0     1
```

11.

```
>> ~(A<=6)
ans =
   1     1     0     1     1
   1     0     1     1     1
   0     0     1     1     1
   1     1     1     1     0
   1     1     1     0     1
```

13.

```
>> P((P<20) & ~(P==1))
ans =
     2
     3
     4
     5
     3
     6
    10
    15
     4
    10
     5
    15
```

15.

```
>> P((P<=4) | (P>20))
ans =
     1
     1
     1
     1
     1
     1
     2
     3
     4
     1
     3
     1
     4
    35
     1
    35
    70
```

17.

```
>> A(mod(A,2)==0)
ans =
    16
     4
     2
    14
    10
     6
     8
    12
```

19.

```
>> A(mod(A,3)==0)
ans =
     9
     3
     6
    15
    12
```

21.

```
>> u=1:100;
>> k=isprime(u);
>> u(k)
ans =
  Columns 1 through 4
     2     3     5     7
  Columns 5 through 8
    11    13    17    19
  Columns 9 through 12
    23    29    31    37
  Columns 13 through 16
    41    43    47    53
  Columns 17 through 20
    59    61    67    71
  Columns 21 through 24
    73    79    83    89
  Column 25
    97
```

23. Define the anonymous function.

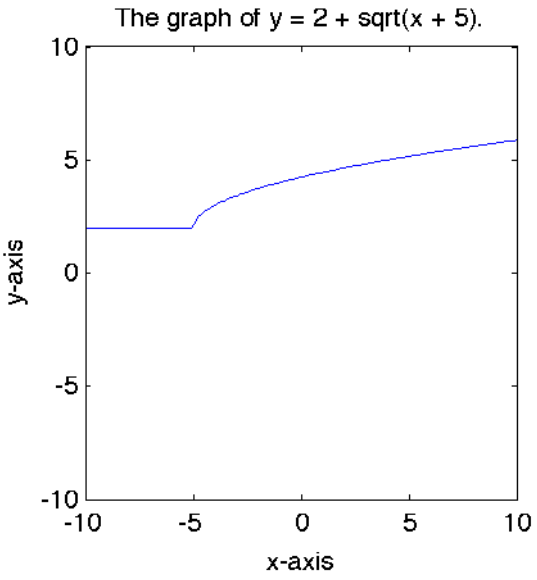
```
f=@(x) 2+sqrt(x+5);
```

Evaluate the function on $[-10, 10]$ and plot the result.

```
x=linspace(-10,10,200);
y=f(x);
plot(x,y)
```

Adjust the window boundaries.

```
axis([-10,10,-10,10])
```

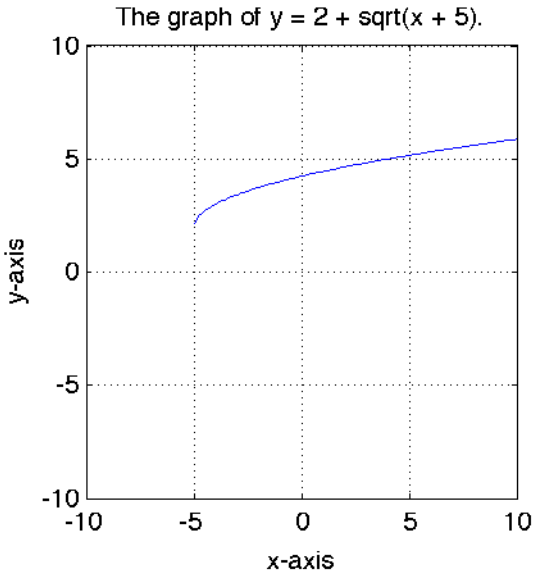


Eliminate complex numbers.

```
k=real(y)~=y;
y(k)=NaN;
```

Open a new figure window and replot. Turn on the grid.

```
figure
plot(x,y)
axis([-10,10,-10,10])
grid on
```



25. Define the anonymous function.

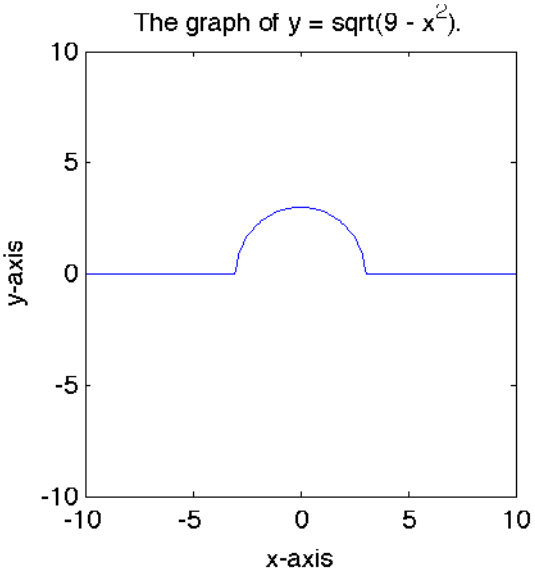
```
f=@(x) sqrt(9-x.^2);
```

Evaluate the function on $[-10, 10]$ and plot the result.

```
x=linspace(-10,10,200);
y=f(x);
plot(x,y)
```

Adjust the window boundaries.

```
axis([-10,10,-10,10])
```

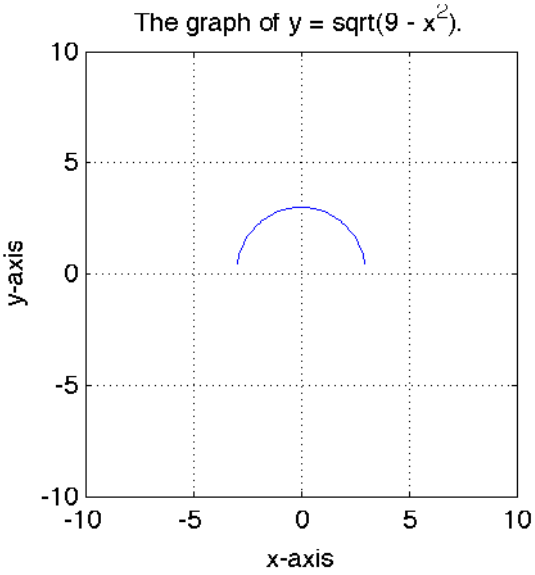


Eliminate complex numbers.

```
k=real(y)~=y;
y(k)=NaN;
```

Open a new figure window and replot. Turn on the grid.

```
figure
plot(x,y)
axis([-10,10,-10,10])
grid on
```



27. Define the anonymous function.

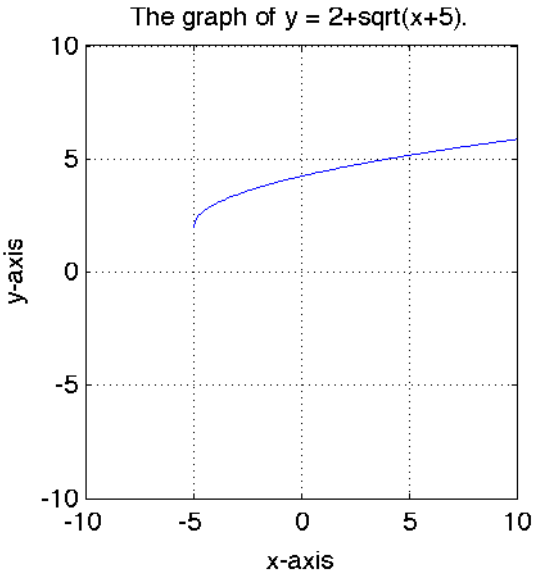
```
f=@(x) 2+sqrt(x+5);
```

The domain of $y = 2 + \sqrt{x+5}$ is the set of all real numbers greater than or equal to -5. Evaluate the function on $[-5, 10]$ and plot the result.

```
x=linspace(-5,10,200);
y=f(x);
plot(x,y)
```

Adjust the window boundaries and add a grid.

```
axis([-10,10,-10,10])
grid on
```



29. Define the anonymous function.

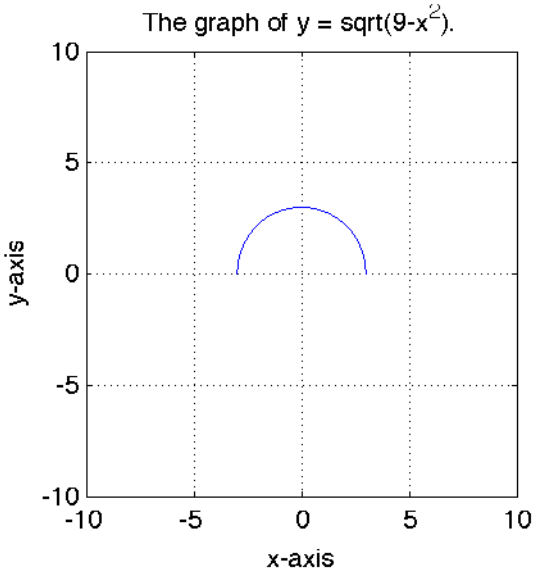
```
f=@(x) sqrt(9-x.^2);
```

The domain of $y = \sqrt{9 - x^2}$ is the set of all real numbers greater than or equal to -3 and less than or equal to 3. Evaluate the function on $[-3, 3]$ and plot the result.

```
x=linspace(-5,10,200);
y=f(x);
plot(x,y)
```

Adjust the window boundaries and add a grid.

```
axis([-10,10,-10,10])
grid on
```



31. Define the anonymous function.

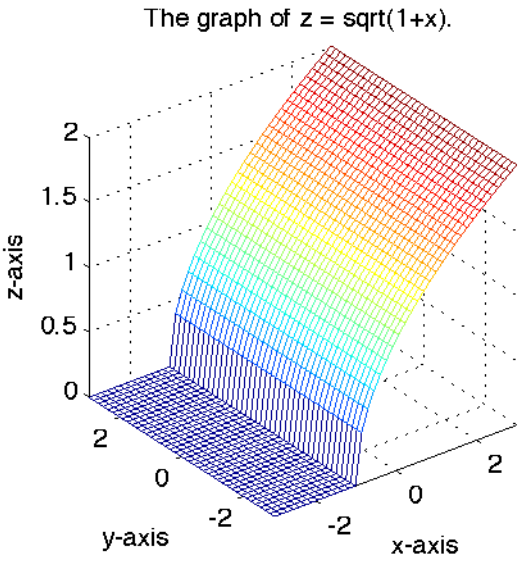
```
f=@(x,y) sqrt(1+x);
```

Set up the grid.

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function at each (x, y) pair in the grid and plot the resulting surface.

```
z=f(x,y);
mesh(x,y,z)
```

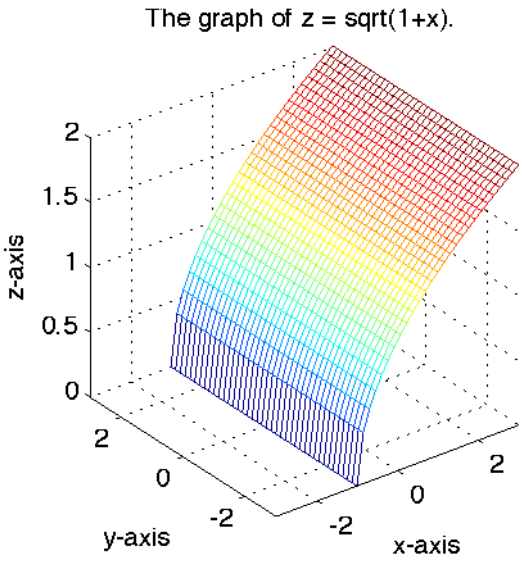


Eliminate complex numbers from the matrix z.

```
k=real(z)~=z;
z(k)=NaN;
```

Open a new figure window and replot the surface.

```
figure
mesh(x,y,z)
```



33. Define the anonymous function.

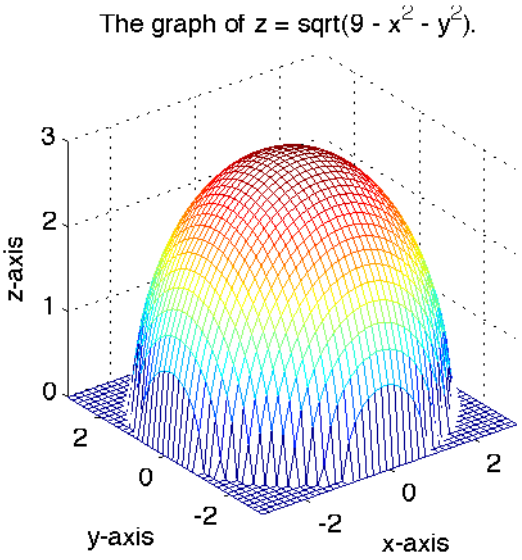
```
f=@(x,y) sqrt(9-x.^2-y.^2);
```

Set up the grid.

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function at each (x, y) pair in the grid and plot the resulting surface.

```
z=f(x,y);
mesh(x,y,z)
```

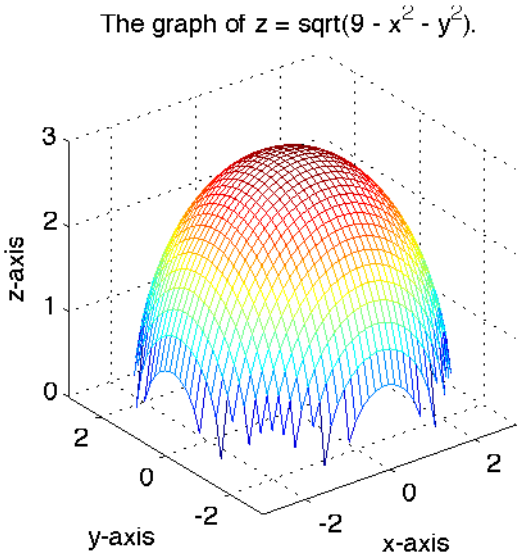


Eliminate complex numbers from the matrix z.

```
k=real(z)~=z;
z(k)=NaN;
```

Open a new figure window and replot the surface.

```
figure
mesh(x,y,z)
```



35. Define the anonymous function.

```
f=@(x,y) 12-x-y;
```

Set up the grid.

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function at each (x, y) pair in the grid.

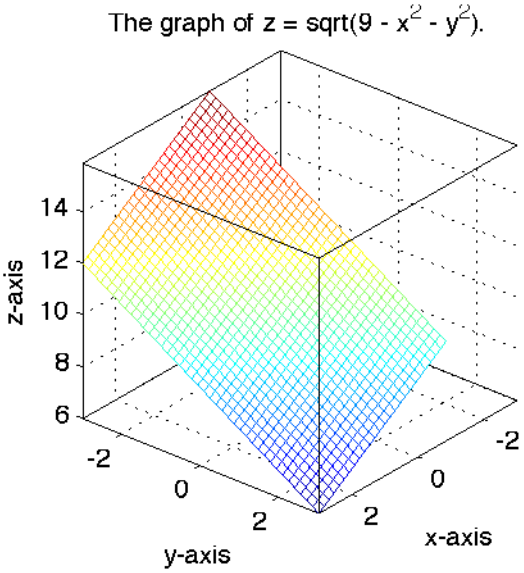
```
z=f(x,y);
```

Determine where $x > -1$ and set all entries in z equal to **NaN** where this contraint is not satisfied.

```
k=x>-1;
z(~k)=NaN;
```

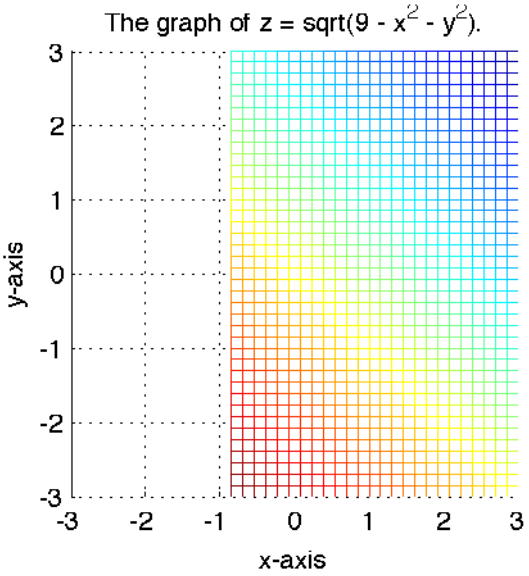
Draw the surface, adjust the orientation, and turn the box on for depth.

```
mesh(x,y,z)
view(130,30)
box on
```



Open a new figure window, replot, and adjust the view so that the eye stares down the z-axis directly at the xy-plane.

```
figure
mesh(x,y,z)
view(0,90)
```



37. Define the anonymous function.

```
f=@(x,y) 14+x-2*y;
```

Set up the grid.

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function at each (x, y) pair in the grid.

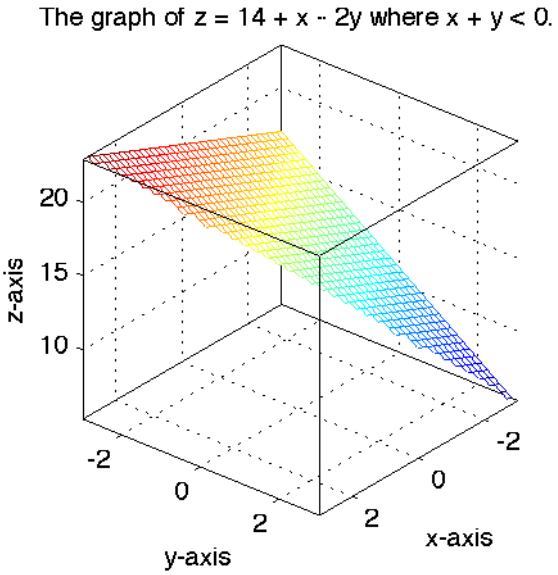
```
z=f(x,y);
```

Determine where $x > -1$ and set all entries in z equal to **NaN** where this contraint is not satisfied.

```
k=x+y<0;
z(~k)=NaN;
```

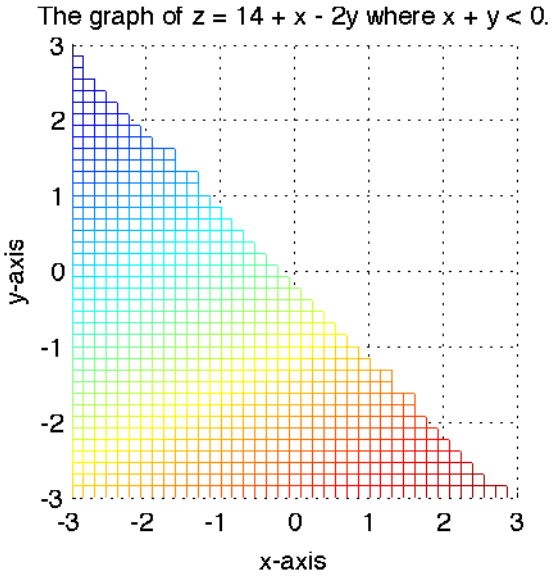
Draw the surface, adjust the orientation, and turn the box on for depth.

```
mesh(x,y,z)
view(130,30)
box on
```



Open a new figure window, replot, and adjust the view so that the eye stares down the z-axis directly at the xy-plane.

```
figure
mesh(x,y,z)
view(0,90)
```



39. Define the anonymous function.

```
f=@(x,y) 11-2*x+2*y;
```

Set up the grid.

```
x=linspace(-3,3,40);
y=x;
[x,y]=meshgrid(x,y);
```

Evaluate the function at each (x, y) pair in the grid.

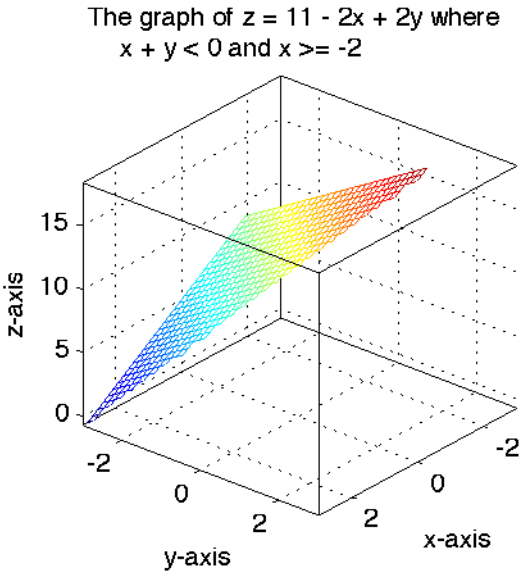
```
z=f(x,y);
```

Determine where $x > -1$ and set all entries in z equal to **NaN** where this contraint is not satisfied.

```
z=f(x,y);
k=(x+y<0) & (x>=-2);
z(~k)=NaN;
```

Draw the surface, adjust the orientation, and turn the box on for depth.

```
mesh(x,y,z)
view(130,30)
box on
```



Open a new figure window, replot, and adjust the view so that the eye stares down the z-axis directly at the xy-plane.

```
figure
mesh(x,y,z)
view(0,90)
```