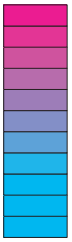


Chapter 4
The Discrete versus the Continuous

There is an interesting boundary between continuous mathematics and digital computing:

- Display monitors are an array of dots. However, the dots are so tiny that the depiction of a continuous function like $\sin(x)$ actually looks continuous on the screen.
- The number of possible display colors is limited. However, the number is so large that for all practical purposes, it looks like we are free to choose from anywhere in the continuous color spectrum.
- Computer arithmetic is inexact. However, the hardware can support so many digits of numerical precision that there is the appearance of perfect computation. We begin to think that one third *is* .333333333333333.

In this chapter we build respect for these scientifically based illusions and an appreciation for what they can hide.

The stakes are high. In many applications, the volume of data that make up "the answer" is too big for the human mind to assimilate. Increasingly, we must rely upon quality graphics to help us spot patterns that would otherwise be hidden. Will the critical discontinuities of a function be exposed? Will the shading of an object from hot pink to deep purple look real or fake? Regarding arithmetic, if we think of the computer as a kind of telescope, then rounding errors affect what we can distinguish in deep "computational" space. Double stars will look like single stars if we are not careful.

Part of being a good computational scientist or engineer is to recognize professionally (not cynically) that seeing is not always believing.

Programming Preview

Concepts

Visualization, plotting functions, granularity, one-dimensional arrays, vectors, vector operations, vector notation, subvectors, color, RGB, interpolation, floating point arithmetic, error.

Language Features

`vector`: A one-dimensional list of values. May be a row or a column.

`length`: Returns the number of components in a vector.

`zeros`, `ones`: Return an array of 0's or an array of 1's.

`linspace`, `logspace`: Return an array of values that are equally spaced on a linear scale or a logarithmic scale.

Colon notation: Specifies a set of values with a fixed increment, e.g., 1:2:9 is the set "1 to 9 in steps of 2".

`for`-loop: The loop index can count up or down with a fixed increment, e.g., 9:−2:1, or take on the values in an arbitrary vector, e.g., [5, −2.5, 1].

`plot`, `fill`: Draw an xy plot or a colored polygon.

`eps`, `inf`, `NaN`, `realmax`, `realmin`: Several MATLAB predefined constants.

MatTV

Video 10. Creating Arrays

How to create arrays of numbers.

Video 11. Array Addressing

How to access subarrays.

Video 12. Basic Mathematical Operations on Arrays

How to write simple expressions that perform a mathematical operation on all elements of a vector. *Vectorized code* performs arithmetic (and relational and logical) operations on multiple elements of an array in one statement.

Video 13. Script

How to write and run MATLAB scripts that create simple graphics.

4.1 Connect the Dots

Problem Statement

Write a script that displays a plot of the function

$$f(x) = \frac{\sin(5x)\exp(-x/2)}{1+x^2}$$

across the interval $[-2,3]$.

Program Development

Let us first consider a much simpler problem: the plotting of the sine function across the interval $[0,2\pi]$. Even more, let us consider how we would approach such a problem "by hand." First, we would produce a table of values, e.g.,

x	0.000	1.571	3.142	4.712	6.283
$\sin(x)$	0.000	1.000	0.000	-1.000	0.000

We would then connect the five points

$$P_1 = (0.000, 0.000)$$
$$P_2 = (1.571, 1.000)$$
$$P_3 = (3.142, 0.000)$$
$$P_4 = (4.712, -1.000)$$
$$P_5 = (6.283, 0.000)$$

obtaining the simple plot that is illustrated in Figure 4.1. It is hard to be happy with such a coarse depiction of such a smooth function. Five evenly distributed sample points means an x-spacing of $\pi/2$ and that is just too crude. If we reduce the spacing from $\pi/2$ to $\pi/4$, then the table of values expands to

x	0.000	0.785	1.571	2.356	3.142	3.927	4.712	5.498	6.283
$\sin(x)$	0.000	0.707	1.000	0.707	0.000	-0.707	-1.000	-0.707	-0.000

(4.1)

and we obtain the plot shown in Figure 4.2. The graph still has "kinks" but there is definitely an improvement.

Obviously, we can repeat this process of refining the graphs by checking out plots that are based on more and more sample points. Eventually, the kinks disappear and our eyes

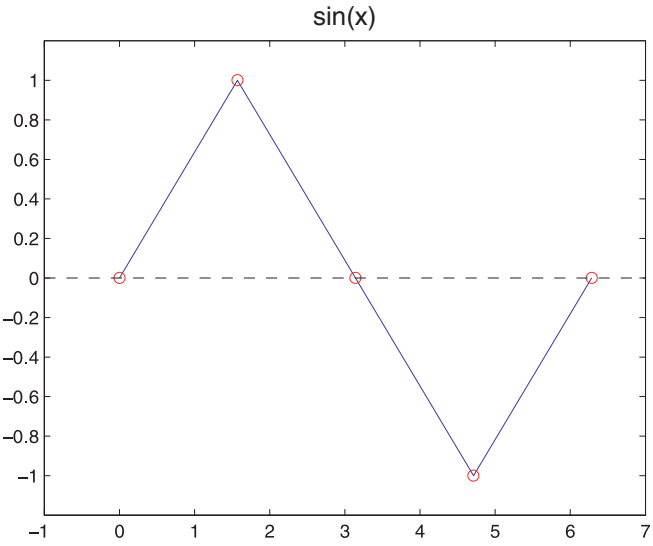


Figure 4.1. *Plot of the Sine Function with* 5 *Sample Points.*

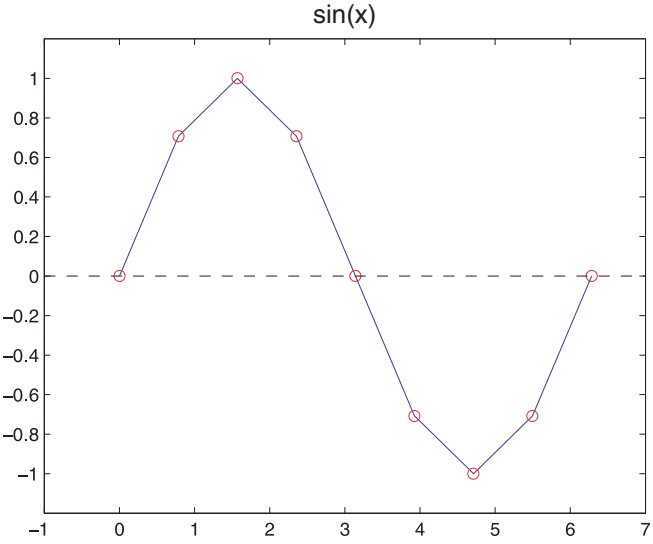


Figure 4.2. *Plot of the Sine Function with* 9 *Sample Points.*

are "fooled" into seeing a smooth function. Just how much sampling is required to produce an acceptable plot depends on human perception factors (How good are your eyes?), screen granularity (Are there 100 pixels per inch or 500 pixels per inch?), and the underlying application (How wild is the underlying function?). Let us write a script that sheds light on the quality of our sine plot as a function of n, the number of sample points.

The graphing of a given function $y = f(x)$ across a given interval $[L, R]$ involves three basic steps:

Step 1. The production of a table of x-values chosen from the interval.

Step 2. The production of a table of y-values that correspond to the evaluation of f at the x-values.

Step 3. A mechanism that connects the dots defined by the xy-pairs and displays the resulting polygonal line.

To illustrate, here is a "rough draft" of the script that produced the plot shown in Figure 4.2:

```
x = linspace(0,2*pi,9);
y = sin(x);
plot(x,y)
```

The built-in function `linspace` is used to generate a table of equally spaced values across the given interval. The syntax for `linspace` is as follows:

`linspace(` Left Endpoint `,` Right Endpoint `,` Number Sample Points `)`

The assignment `x = linspace(0,2*pi,9)` is the assignment of an array to `x`:

x:	0.000	0.785	1.571	2.356	3.1412	3.927	4.712	5.498	6.283

A table of values like this is a one-dimensional *array*. In MATLAB this assembly of data is also known as a *vector*. Higher-dimensional arrays will be discussed later. For now, we use the terms "array," "vector," and "table" interchangeably.

The statement `y = sin(x)` looks familiar enough, only now the `sin` function is handed a table of values instead of just a single number as is more customary. The result is that `y` is assigned the array of sine evaluations that correspond to `x`:

y:	0.000	0.707	1.000	0.707	0.000	−0.707	−1.000	−0.707	−0.000

Notice that `x` and `y` house the top and bottom half, respectively, of the table of values (4.1).

The last line in the above script involves the `plot` function. If `x` and `y` are vectors with the same length, then they define a set of points in the plane and the command `plot(x,y)` simply "connects the dots."

With these preliminaries we can address the question of how many function evaluations are necessary to produce a smooth sine plot. A script of the form

```
for n=25:25:500
   % Show sin(x) with n points
   x = linspace(0,2*pi,n);
   y = sin(x);
   plot(x,y)
   title(sprintf('n = %3d',n))
   pause
end
```

reveals that $n = 100$ is "good enough" given typical screen granularity and average eyesight. See Figure 4.3.

The `title` command is used to display the string `sprintf('n = %3d',n)` across the top of the plot window. The function `sprintf` is just like `fprintf`, except that it returns the specified string message instead of writing it to the command window.

The `pause` command halts the program until the user strikes a key. With this feature, we can study the sequence of plots at our own pace.

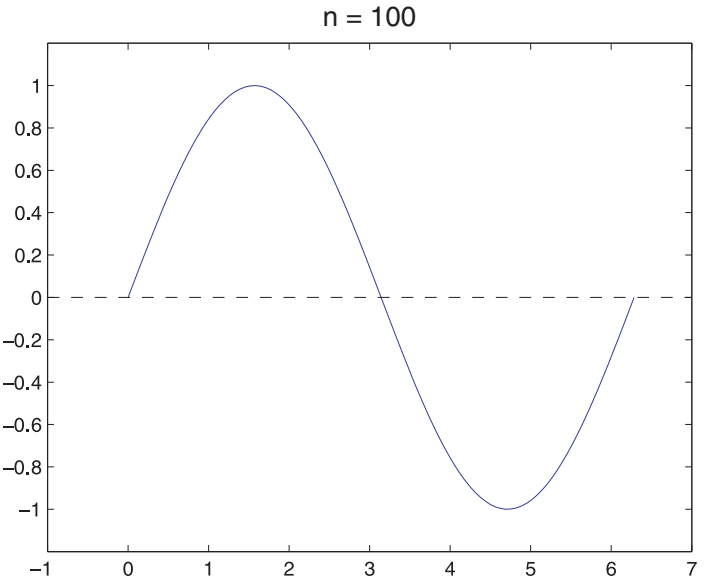
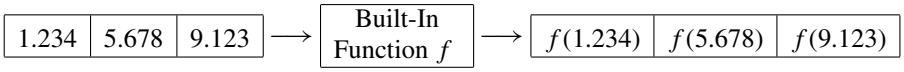
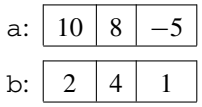


Figure 4.3. *Plot of the Sine Function with* 100 *Sample Points.*

Let us return to the problem posed at the beginning of the section. Two features of MATLAB make the task of function plotting easy. First, built-in functions like `sin`, `cos`, `exp`, and `log` can accept an input argument that is a vector. When this is the case, they return the corresponding array of function evaluations, e.g.,



A second feature of MATLAB that is very handy in plot situations is its support of vector-level operations. To illustrate, suppose variables `a` and `b` are initialized as follows:



These vectors can be

scaled(*)	`c = 5*a`	⇒	c:	50	40	−25
scaled(/)	`c = a/2`	⇒	c:	5	4	−2.5
negated	`c = -a`	⇒	c:	−10	−8	5
reciprocated	`c = 1./a`	⇒	c:	.100	.125	−.200
shifted	`c = 5+a`	⇒	c:	15	13	0
exponentiated	`c = a.^2`	⇒	c:	100	64	25
added	`c = a+b`	⇒	c:	12	12	−4
subtracted	`c = a-b`	⇒	c:	8	4	−6
multiplied	`c = a.*b`	⇒	c:	20	32	−5
divided	`c = a./b`	⇒	c:	5	2	−5

With this repertoire, we can easily build a table of values for more complicated functions. For example, the script

```
x = linspace(-2,3,100);
y1 = 5*x;                % vector scaling
y2 = sin(y1);            % vector of sine evaluations
y3 = -x;                 % vector negation
y4 = y3/2;               % vector scaling
y5 = exp(y4);            % vector of exp evaluations
y6 = y2.*y5;             % vector multiplication
y7 = x.^2;               % vector exponentiation
y8 = 1 + y7;             % vector shifting
y  = y6./y8;             % vector division
plot(x,y)
```

results in the plotting of the function

$$f(x) = \frac{\sin(5x)\exp(-x/2)}{1+x^2}$$

across the interval $[-2,3]$. The code for `y` can be collapsed down to a single assignment statement:

```
y = (sin(5*x).*exp(-x/2))./(1 + x.^2);
```

Notice how this looks just like a typical scalar assignment except for the "dot operations" that designate vector multiplication, vector division, and vector exponentiation. Here is the full solution script and the plot that it produces.

The Script `Eg4_1`

```
% Script Eg4_1
% Plots the function f(x) = sin(5x)*exp(x/2)/(1 + x^2)
% across [-2,3].

L = -2;   % Left endpoint
R =  3;   % Right endpoint
N = 200;  % Number of sample points

% Obtain the vector of x-values and f-values...
x = linspace(L,R,N);
y = sin(5*x) .* exp(-x/2) ./ (1 + x.^2);

% Plot and label...
plot(x,y,[L R],[0 0],':')
title('The function f(x) = sin(5x) * exp(x/2) / (1 + x^2)')
ylabel('y = f(x)')
xlabel('x')
```

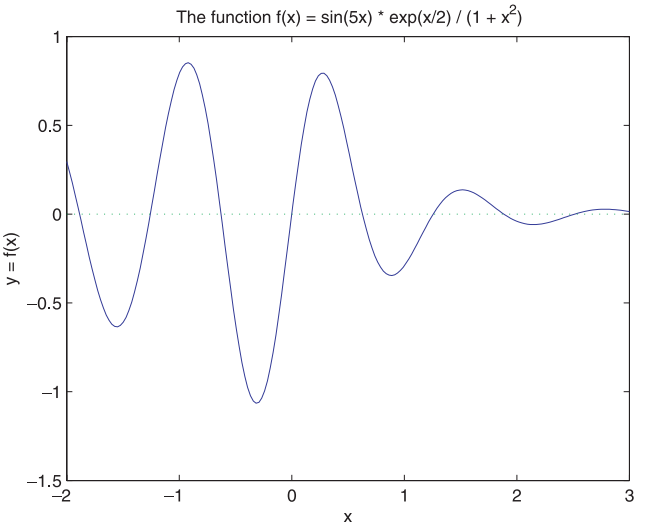


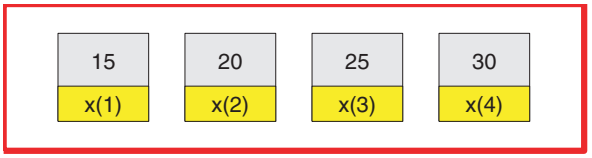
Figure 4.4. *Sample Output from the Script* `Eg4_1`.

The support of vector-level operations is a rich feature of the MATLAB language that enables us to write more readable and efficient code. To appreciate this point we need to look more carefully at the vector structure and how particular components can be accessed through the use of *subscripts.*

Consider the assignment `x = linspace(15,30,4)` which establishes `x` as a length-4 row vector:

x:	15	20	25	30

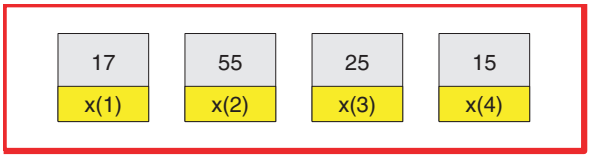
Here is a better way to visualize x:



This schematic reinforces the idea that x is an array variable having four components that are (traditional) simple variables. The names of the component variables are `x(1)`, `x(2)`, `x(3)`, and `x(4)`. The numbers within the parentheses are *subscripts*. Subscripted variables can be involved in assignment statements. For example, the fragment

```
x(1) = x(1) + 2;
x(2) = x(3) + x(4);
x(4) = x(4)/2;
```

transforms the vector x above into



The value of a subscript can be specified by a variable or an arithmetic expression. Thus,

```
x(1) = x(1) + 2;
```

is equivalent to

```
k = 1;
x(k) = x(k) + 2;
```

while

```
x(2) = x(3) + x(4);
```

is equivalent to

```
i = 1;
x(i) = x(i+2) + x(i+3);
```

When a subscript is specified by an arithmetic expression, the expression must evaluate to a "legal" subscript. Thus

```
i = 3;
x(i) = x(i+1) + x(i+2);
```

will result in an error because `i+2` evaluates to 5 and there is no `x(5)`.

To further our ability to reason about subscripts, let us use a `for`-loop to set up the vector `x = linspace(15,30,4)`. Of course, for such a short vector we can do this "by hand":

```
x(1) = 15;
x(2) = 20;
x(3) = 25;
x(4) = 35;
```

But the recipes for the component values are simple functions of the subscript: the value of `x(k)` is `10+5*k` for `k = 1`, 2, 3, and 4. Thus,

```
for k=1:4
   x(k) = 10 + 5*k;
end
```
(4.1)

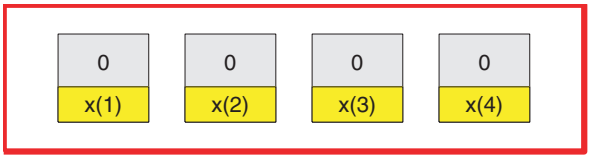
does the job. Knowing that the loop counter `k` steps from 1 to 2 to 3 to 4, this is equivalent to

```
k = 1;
x(k) = 10 + 5*k;
k = 2;
x(k) = 10 + 5*k;
k = 3;
x(k) = 10 + 5*k;
k = 4;
x(k) = 10 + 5*k;
```

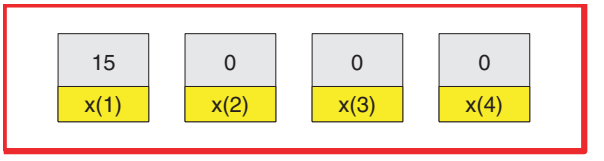
When using a loop to set up a vector, it is a good habit to establish the size and orientation of the vector using the `zeros` function before beginning the iteration. The command

```
x = zeros(1,4);
```

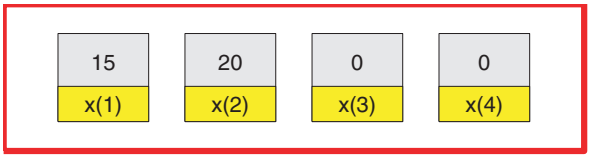
establishes `x` as a length-4 row vector with components initialized to zero:



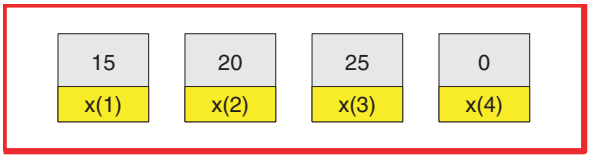
Let us trace the execution of (4.1) now that `x` has this form. The first time through the loop, `k` has the value of 1. The dynamics of the assignment `x(k) = 10 + 5*k` is very similar to what we learned for simple variable assignments. The right-hand side is a recipe for a value, in this case 15. The left-hand side names the target variable, but now the name of the target variable is computed: `x(1)`. After the first pass the `x` array looks like this



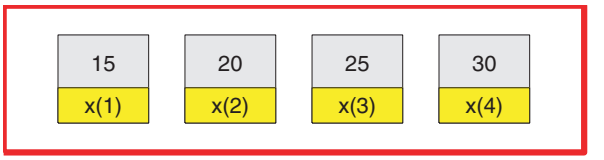
The processing is similar during the second pass when k is 2:



And during the third pass when k is 3:



And during the fourth pass when k is 4:



As a more general illustration of vector setup using a `for`-loop, here is a fragment that is equivalent to `x = linspace(a,b,n)` assuming that the variables `a`, `b`, and `n` (positive integer greater than one) are initialized:

```
% The spacing factor...
h = (b-a)/(n-1);
% A length-n row vector...
x = zeros(1,n);
for k=1:n
   x(k) = a + (k-1)*h;
end
```
(4.2)

The recipe for the spacing factor `h` is derived from the fact that `linspace` generates n equally spaced sample points across the interval $[a,b]$ *including* both endpoints a and b. Thus, the value of `x(1)` is `a` and the value of `x(n)` is `b`.

The above loop implementation requires the derivation of an explicit recipe for the value of `x(k)`. Here is an equivalent solution that avoids this:

```
% The spacing factor...
h = (b-a)/(n-1);
% A length-n row vector...
x = zeros(1,n);
x(1) = a;
for k=2:n
   x(k) = x(k-1) + h;
end
```
(4.3)

The idea behind this approach is that the kth sample point is h plus the $(k-1)$st sample point.

Talking Point: Granularity and the Array

With a sufficient number of function evaluations, the plot of a continuous function appears smooth. With enough pixel support, a digital camera can produce a high-resolution wall poster. With a sufficient number of frames per second a video can create the illusion of motion. The message is clear: we are usually happy with a discretization if it is sufficiently "fine grained." To say that a plot or an image or a movie looks real is to say that the underlying samplings are close to one another in space and/or time.

Underlying all this technology are arrays, squarely on the border between the discrete and the continuous. Arrays are used to house digitized information. As we will see, a music CD is a (very) big one-dimensional array, a digitized picture is a (very, very) big two-dimensional array, and a DVD, being a sequence of still images, is a (very, very, very) big three-dimensional array. Problem solving in these venues requires having a facility with subscripts and an ability to reason at the vector level. It will take practice. Lots of practice!

MATLAB Review

Vector Orientation and Subscripts

A vector is a one-dimensional array. Vectors have a length and can be row oriented or column oriented. If `x` is a vector, then `y = x'` has the opposite orientation and we say that `x` has been *transposed.* Subscripts start at one and must be integers. An error results if the value of a subscript is less than one or is not an integer. A *scalar* variable is a length-1 vector.

Creating Short Vectors

To specify explicitly a short row vector, enclose the component values with square brackets and separate the component values with spaces or with commas, e.g., `v = [19 2 -3]` or `v = [19, 2, -3]`. The same rules apply for column vectors, except that the component values are separated by semicolons, e.g., `v = [19; 2; -3]`.

Concatenating Vectors

It is possible to make longer vectors by "gluing together" shorter vectors. If `x` is a length-n row vector and `y` is a length-m row vector, then `z = [x y]` is a length $n+m$ row vector obtained by augmenting `x` with values from `y`. Thus,

```
x = [10 20];  y = [30 40 50];  z = [x y];
```

is equivalent to `z = [10 20 30 40 50]`. Column vectors may also be concatenated, but the column vectors that make up the concatenation must be separated by semicolons, e.g.,

```
x = [10; 20];  y = [30; 40; 50];  z = [x; y];
```

length

If `x` is a vector, then `n = length(x)` assigns its length to `n`.

Addressing an Array Component

If `x` is a vector, then `x(k)` is the kth component of `x`. To *access* an existing component, `k` is an integer satisfying $1 \le$ `k` $\le$ `length(x)`, e.g.,

```
x = [10 25 20];  y = x(2);
```

To *create* a vector component, `k` can be any positive integer, e.g.,

```
x = [10 25 20];  y(4) = x(2);
```

creates a vector `y` with the values `[0 0 0 25]`.

linspace

Use `linspace` to construct row vectors with equally spaced values. For example, the assignment `x = linspace(0,3,7)` is equivalent to

```
x = [0.0  0.5  1.0  1.5   2.0   2.5   3.0]
```

In general, if `a` and `b` are real-valued scalars and `n` is an integer with $n \ge 2$, then the assignment `x = linspace(a,b,n)` is equivalent to

```
h = (b-a)/(n-1);
for k=1:n
   x(k) = a + (k-1)*h;
end
```

Note that the spacing between components is $(b-a)/(n-1)$ and not $(b-a)/n$.

logspace

Use `logspace` to construct vectors with values that are equally spaced logarithmically. For example, `x = logspace(-1,-6,6)` is equivalent to

```
x = [10^-1  10^-2  10^-3  10^-4  10^-5  10^-6]
```

More generally, `x = logspace(a,b,n)` is equivalent to

```
e = linspace(a,b,n);
for k=1:n
   x(k) = 10^e(k);
end
```

Colon Notation

The colon notation can be used to generate a row vector of equally spaced values with a prescribed spacing. For example, `x = 1:.3:2` is equivalent to `x = [1.0 1.3 1.6 1.9]`, i.e., read `x = 1:.3:2` as "x goes from 1 up to 2 in steps of 0.3." Note that the last component has the value 1.9, not 2. In general, if $a < b$ and $s > 0$ or if $b < a$ and $s < 0$, then `x = a:s:b` is equivalent to

```
n = floor((b-a)/s) + 1;  % Number of components
for k = 1:n
    x(k) = a + (k-1)*s;
end
```

The value of `s` is referred to as the *stride*. Use `linspace(a,b,n)` if it is handier to reason about the number of sample points rather than their spacing or if it is critical to "land" on the "target" *b*. If the stride in the colon expression is "missing," then it is assumed to be one. Thus, `2:5` is the same as `2:1:5` and the vector of integers `[2 3 4 5]`.

zeros and ones

If `n` is an initialized positive integer, then `x = zeros(1,n)` assigns to `x` a length-*n* row vector of zeros. Similarly, `x = zeros(n,1)` assigns to `x` a length-*n* column vector of zeros. The `ones` function behaves the same way except that the components are assigned the value one instead of zero.

Vector Operations

If `x` is a vector and `s` is a scalar, then `x*s` multiplies each component by `s`, `x+s` adds `s` to each component, `x/s` divides each component by `s`, `s./x` reciprocates each component and then multiplies by `s`, and `x.^s` raises each component to the power of `s`. If `y` has the same length and orientation as `x`, then `x+y`, `x-y`, `x.*y`, `x./y`, `x.^y` produce vectors by combining components in the indicated fashion.

`for`-Loops (Again)

`for`-loops have the form

```
for  loop variable =  vector of loop values
     code fragment
end
```

where the loop variable successively takes on the values in the vector of loop values. The number of loop body repetitions is therefore the length of this vector. In the simplest case, the vector of loop values has the form `1:n`. Other possibilities include `0:.1:10`, `[1 -2 3 -4]`, `n:-1:1`, ..., etc.

pause

`pause` stops the program and waits for the user to press any key before continuing. `pause(t)` stops the program for `t` seconds before continuing; `t` can be a fraction.

plot

Use `plot` to display the data points (x_k, y_k) defined by a pair of vectors that have equal length and orientation. The command `plot(x,y)` "connects the dots," thereby displaying the graph of y versus x in a figure window. It is possible to display more than one graph with a single `plot` command, e.g., `plot(x,y1,x,y2,x,y3,...)` plots `y1` versus `x`, `y2` versus `x`, `y3` versus `x`, etc.

Line and Marker Formats

It is possible to specify how the dots are connected in a `plot` command, e.g.,

```
plot(x,y1,'-',x,y2,':',x,y3,'-.',x,y4,'--')
```

Use `'-'` for solid lines, use `':'` for dotted lines, use `'-.'` for dash-dot lines, and use `'--'` for dashed lines. Instead of connecting the dots, it is possible just to put a specified marker at the dots, e.g.,

```
plot(x,y1,'.',x,y2,'o',x,y3,'+',x,y4,'x',x,y5,'*')
```

Use `'.'` for point marks, use `'o'` for circle marks, use `'+'` for plus marks, use `'x'` for x-marks, and `'*'` for star marks.

Line and Marker Colors

Lines and markers can be colored using these mnemonics:

`w`	white	`c`	cyan	`b`	blue	`m`	magenta
`r`	red	`y`	yellow	`g`	green	`k`	black

For example,

```
plot(x,y1,'r',x,y2,'*b',x,y3)
```

colors the first plot red (line) and the second plot blue (star marks). The color of the third plot (line) is automatically selected since no color is specified.

title, xlabel, ylabel

It is possible to place a title over a plot and to label the axes:

```
title( string )
xlabel( string )
ylabel( string )
```

sprintf

The `sprintf` function, like `fprintf`, is used to produce formatted strings that incorporate values of specified variables. It has the form

```
sprintf( string with format controls , list of variables )
```

Example:

```
title(sprintf('Temperature = %4d degrees',T))
```

`sprintf` returns a string that can be stored in a variable or used as the argument to a function, as shown in the example above with function `title`.

semilogx, semilogy, loglog

A plot with logarithmic scaling along the x-axis, y-axis, or both can be achieved by using `semilogx(a,b)`, `semilogy(a,b)`, or `loglog(a,b)`, where `a` and `b` are vectors of the same length.

More Refined Graphics

Appendix A covers line width, font size, special characters, legends, axis labelling, coloring, and other features that can be used to produce professional-looking graphics.

Exercises

M4.1.1 Experiment with `Eg4_1` by changing the number of sample points in the interval $[-2,3]$. If you double `N`, is the graph visibly smoother? If you halve `N`, is the shape of the function still clear? For this function in the given range, roughly how small can `N` be and still give a reasonable graph?

M4.1.2 Modify `Eg4_1` so that it places a green x marker at the point (z,o) if z is a zero, a blue circle marker at $(z, f(z))$ if z is a local minimum, and a red circle marker at $(z, f(z))$ if z is a local maximum.

M4.1.3 Modify `Eg4_1` to solicit user input values of `L` and `R`. The plot should be a red dashed line across $[L,c]$, a black solid line across $[c,d]$, and a red dashed line across $[d,R]$, where $c = (2L+R)/3$ and $d = (L+2R)/3$.

P4.1.4 Modify `Eg4_1` to use the colon expression instead of the built-in function `linspace` to create the vector of `x`-values. `linspace` allows you to specify the number of points, while the colon expression allows you to specify the increment between points. Write a colon expression that matches the `linspace` function call in `Eg4_1` exactly.

P4.1.5 Write a script that inputs three values a, b, c and then prints the length of `a:c:b` and the distance from the last vector component to b.

P4.1.6 How long is the vector `x` after the following script is executed?

```
x = 1:10:100;
while length(x) < 1000
   x = [x x(1) x];
end
```

P4.1.7 Write a script that uses `semilogy` to display the function $f(x) = 3^x + (2+\sin(x))2^x$ across [0,10].

P4.1.8 Recall that the cosine function has period 2π. In the following fragment, complete the `plot` statement so that the cosine function is displayed across the interval $[-2\pi, 6\pi]$:

```
x = linspace(0,2*pi);
y = cos(x);
plot( ??? )
```

Your solution should not involve any additional cosine evaluations.

P4.1.9 Write a script that displays in a single figure window a plot of the functions x, x^2, x^3, x^4, and x^5 across the interval $[0,1]$.

P4.1.10 Plot the function

$$f(x) = 1 + \cfrac{x}{1 - \cfrac{x/2}{1 + \cfrac{x/6}{1 - \cfrac{x/6}{1 + \cfrac{x/10}{1 - x/10}}}}}$$

across the interval $[-2,5]$.

P4.1.11 Write a script that inputs a positive integer n and then generates a length-10 row vector `f` according to the following formula:

$$f_k = \begin{cases} n & \text{if } k = 1 \\ 3f_{k-1}+1 & \text{if } 10 \geq k > 1 \text{ and } f_{k-1} \text{ is odd} \\ f_{k-1}/2 & \text{if } 10 \geq k > 1 \text{ and } f_{k-1} \text{ is even.} \end{cases}$$

Your script should plot the points $(1, f_1), \ldots, (10, f_{10})$ using the star marker.

4.2 From Cyan to Magenta

Problem Statement

The idea of interpolating values in a table is familiar. Consider this excerpt from a sine table:

x°	$\sin(x^\circ)$
⋮	⋮
44	0.6947
45	0.7071
46	0.7193
47	0.7314
⋮	⋮

To estimate sin(45.2), sin(45.4), sin(45.6), and sin(45.8), we take appropriate linear combinations of sin(45) and sin(46):

$$\sin(45.2) = \sin(45) + \frac{1}{5}(\sin(46)-\sin(45)) = \frac{4}{5}\sin(45)+\frac{1}{5}\sin(46) = 0.7096$$

$$\sin(45.4) = \sin(45) + \frac{2}{5}(\sin(46)-\sin(45)) = \frac{3}{5}\sin(45)+\frac{2}{5}\sin(46) = 0.7120$$

$$\sin(45.6) = \sin(45) + \frac{3}{5}(\sin(46)-\sin(45)) = \frac{2}{5}\sin(45)+\frac{3}{5}\sin(46) = 0.7144$$

$$\sin(45.8) = \sin(45) + \frac{4}{5}(\sin(46)-\sin(45)) = \frac{1}{5}\sin(45)+\frac{4}{5}\sin(46) = 0.7169.$$

The idea is that if we "walk" from $x = 45$ to $x = 46$ and have completed fraction f of the journey, then we should see the same fractional change in the sine value, e.g.,

$$\frac{2}{5} = \frac{45.4-45.0}{46.0-45.0} = \frac{\sin(45.4)-\sin(45)}{\sin(46)-\sin(45)}.$$

This is *linear interpolation.*

Linear interpolation can be used to interpolate between "known" colors just as it can be used to interpolate between known numerical values. Colors in the MATLAB graphics environment are represented by length-3 vectors whose first, second, and third components specify the amounts of red, green, and blue that makes up the color, e.g.,

We refer to vectors that represent colors as *rgb* vectors. Each component value is between 0 and 1. Thus, cyan is an equal mix of green and blue while magenta is an equal mix of red and blue. We can generate an interpolation of these two colors by applying linear interpolation to each component. Thus, we can compute the rgb vector for a color that is 3/5 cyan and 2/5 magenta as follows:

$$\frac{3}{5}\begin{bmatrix} 0.0 & 1.0 & 1.0 \end{bmatrix} + \frac{2}{5}\begin{bmatrix} 1.0 & 0.0 & 1.0 \end{bmatrix} = \begin{bmatrix} 0.4 & 0.6 & 1.0 \end{bmatrix}.$$

Here is what it looks like:

Write a script that displays eleven "paint chips" that range from cyan to magenta:

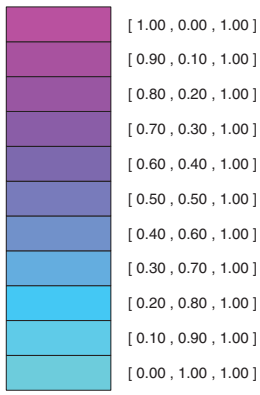


Figure 4.5. *Cyan to Magenta.*

The nine "in between" colors should be equally spaced. Use linear interpolation.

Program Development

The `fill` command can be used to display a "tile" of a particular color. For example,

```
x = [0 3 3 0];    y = [0 0 1 1];    v = [0.0 1.0 1.0];
fill(x,y,v)
```

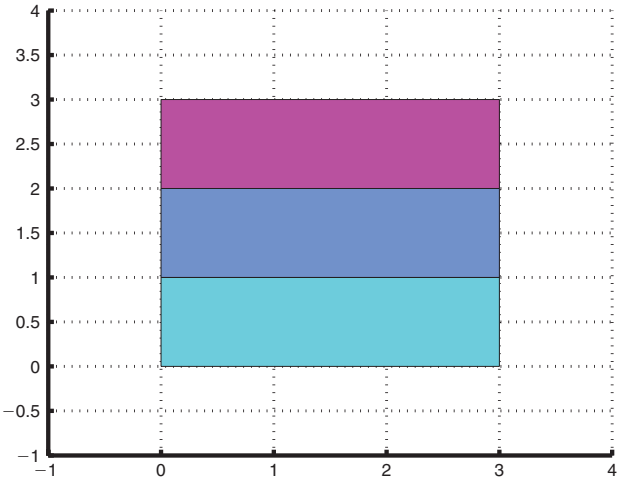


Figure 4.6. *Three Colored Rectangles.*

produces a rectangle with vertices (0,0), (3,0), (3,1), and (0,1) and fills it with the color specified by the rgb vector `v` (cyan).

The following fragment displays in a single window a cyan rectangle, a magenta rectangle, and a rectangle that has the interpolated color that we derived above:

```
x = [0 3 3 0];
y = [0 0 1 1];
hold on
fill(x,y,[0.0 1.0 1.0])
fill(x,y+1[0.4 0.6 1.0])
fill(x,y+2,[1.0 0.0 1.0])
```

See Figure 4.6. The `hold` command ensures that all subsequent plots are added to the current figure window.

The problem we are to solve requires a similar display, only there are to be eleven rectangles. Using a loop to oversee this, we obtain the following pseudocode solution:

```
n = 10;
```

Other Initializations

```
for j=0:n
   % Display rectangle j
```

Compute the rgb vector `v` for the jth tile's color. (4.4)

Compute the `x` and `y` vectors that locate the position of the jth tile.

```
   fill(x,y,v)
end
```

Let us start with the color computation. Our plan is to display the jth tile with a color that has cyan fraction $(1 - j/10)$ and magenta fraction $j/10$:

```
cyan = [0 1 1];
magenta = [1 0 1];                                    (4.5)
f = j/n;
v = (1-f)*cyan + f*magenta;
```

Note that tile 0 is cyan and tile 10 is magenta. If the value of `j` is 4, then `v` defines the mixed color displayed above.

Regarding the positioning of the tiles, we build on the idea behind Figure 4.6 where each tile has width 3 and height 1. We position tile 0 at the bottom of the stack, e.g.,

```
fill([0 3 3 0],[0 0 1 1],v).
```

The `fill` command for subsequent tiles is the same except that the values in the `y` vector increase by one each step. Thus, the recipes for the `x` and `y` vectors are

```
x = [0 3 3 0];                                        (4.6)
y = [0 0 1 1] + j;
```

Substituting (4.5) and (4.6) into (4.4) and doing a little rearranging, we obtain

```
cyan = [0 1 1];      % rgb of the "bottom" color
magenta = [1 0 1];   % rgb of the "top" color
n = 10;              % the number of "in between" colors is n-1
x = [0 3 3 0];       % locates the x-values in the tiles
y = [0 0 1 1];       % locates the y-values in the tiles

for j=0:n
   % Display the jth tile ...
   f = j/n;
   v = (1-f)*cyan + f*magenta
   fill(x,y+j,v)
end
```

This essentially completes the solution. However, we add a few features so that the overall graphic looks better and is more informative.

For starters, next to each tile we display its rgb value using the `text` command. This command expects an xy location and a string that is to be displayed at that location. Inserting the statement

```
text(3.5,j+.5,sprintf('[ %4.2f , %4.2f , %4.2f ]',v(1),v(2),v(3)))
```

just after the `fill` statement results in the display of the rgb values. Notice how individual components of the rgb vector `v` are referenced by the `sprintf` command. The xy position is just to the right of the tiles. It usually takes a bit of trial and error to get the location of a "text message" exactly right.

At the beginning of the script we prepare the figure window with

```
close all
figure
axis equal off
hold on
```

These commands (a) close any open figure windows, (b) create a new figure window, (c) hide the axes and force the scaling in the x and y directions to be the same, and (d) hold the current figure window in place so that the results of the subsequent `fill` commands are added to the current window. After the loop we add the statements

```
hold off
shg
```

which turns off the `hold` toggle (good programming) and ensures that the figure window is displayed on the screen (on top of the command window instead of being hidden by it). Here is the finished script:

The Script `Eg4_2`

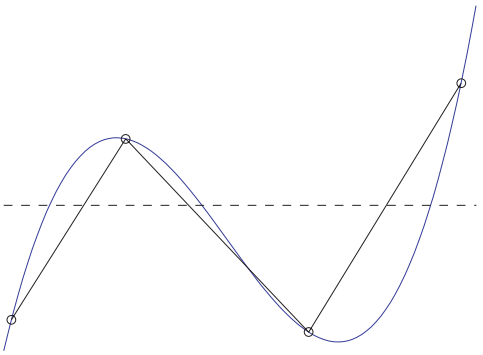
```
% Script Eg4_2
% Displays interpolants of the colors cyan and magenta

% Prepare the figure window...
close all
figure
axis equal off
hold on

% Initializations...
cyan    = [0 1 1];  % rgb of the "bottom" color
magenta = [1 0 1];  % rgb of the "top" color
n = 10;             % the number of "in between" colors is n-1
x = [0 3 3 0];      % locates the x-values in the tiles
y = [0 0 1 1];      % locates the y-values in the tiles

% Add colored tiles to the figure window...
for j=0:n
    % Display the jth tile and its rgb value...
    f = j/n;
    v = (1-f)*cyan + f*magenta;
    fill(x,y+j,v)
    text(3.5,j+.5,sprintf('[ %4.2f , %4.2f , %4.2f ]',...
                                              v(1),v(2),v(3)))
end
hold off
shg
```

It produces the "paint chip" graphic that is displayed in Figure 4.5.

Talking Point: Interpolation

Interpolation is a way of inferring the value of a function from known, surrounding values of the function. We have examined the notion in the context of color.

An interesting step up from linear interpolation is *cubic* interpolation where the interpolant is a cubic polynomial rather than a linear polynomial. Whereas a linear interpolant is based upon a pair of data points, a cubic interpolant is based on four.



Cubic interpolants do a better job of capturing nonlinearity. Moreover, they have a "smoothness" about them that can translate into graphical renditions that are more pleasing to the human eye.

Because it serves as a bridge between the discrete and the continuous, interpolation has a central role to play in computational science and engineering. There is a never-ending search for clever new ways to estimate what is going on in between the dots (or colors).

MATLAB Review

fill

Use `fill` to display colored polygons. It works like `plot` except that in `fill(x,y,c)` the last point specified by vectors `x` and `y` is connected to the first point and the enclosed area is colored according to `c`. Examples:

```
fill(x,y,'r')
fill(x,y,[.3 .1 .9])
fill(x1,y1,'k',x2,y2,'w',x3,y3,[.5 .5 .5])
```

The perimeter of `fill(x,y,c)` is displayed by `plot([x x(1)],[y y(1)],'k')`.

rgb Vectors

A 3-vector `c` with component values chosen from [0,1] can be used to represent a color. The red intensity value is `c(1)`, the green intensity value is `c(2)`, and the blue intensity value is `c(3)`. Here are the rgb values for the eight basic colors that have MATLAB mnemonics:

Color	Mnemonic	rgb
black	k	[0 0 0]
blue	b	[0 0 1]
green	g	[0 1 0]
cyan	c	[0 1 1]
red	r	[1 0 0]
magenta	m	[1 0 1]
yellow	y	[1 1 0]
white	w	[1 1 1]

text

Use `text` to display strings in the figure window. A `text` command has the form

`text(x-coordinate, y-coordinate, string).`

figure

This opens up a new figure window and makes it the "current" figure window. Figure windows are indexed. Thus, `figure(3)` makes the third figure window the current figure window, if it exists, or creates a figure window that is indexed 3 if it does not already exist. The fragment

```
x = linspace(0,2*pi,100);
figure
plot(x,sin(x))
figure
plot(x,cos(x))
```

creates two figure windows; one displays the sine function and the other the cosine function.

close all

This closes (deletes) all open figure windows.

hold on, hold off

Following a `hold on` command, subsequent executions of `plot` and `fill` are placed in the current figure window. To turn off this feature use `hold off`. It is good to set the hold toggle to `off` at the end of a script. Otherwise, the next script that you run might exhibit strange behavior if it involves graphics.

shg

This brings the current figure to the front, eclipsing the command window and all other open figure windows. It is often handy to place an `shg` at the end of an important graphical computation that requires an immediate evaluation.

axis off, axis equal, axis square

These commands adjust the default axis properties. Use `axis off` to hide the axes. (In this case `title` still works but `xlabel` and `ylabel` do not.) Use `axis equal` to force equal scaling along the x- and y-axes. (Otherwise, a circle will appear as an ellipse.) Use `axis square` if you want the plot window to be square instead of the default rectangular shape. Note that

```
axis off equal
```

is equivalent to

```
axis equal
axis off
```

Exercises

M4.2.1 Modify `Eg4_2` so that instead of cyan and magenta the "endpoint" colors are `[x x x]` and `[1 1 1]` (white) where the value of `x` satisfies $0 \leq x \leq 1$ and is obtained via `input`. How small must x be before you see a distinct levels of grayness across the 11 tiles?

M4.2.2 Modify `Eg4_2` so that it produces three figures. The first should display the tiles with endpoint colors red and green, the second with endpoint colors red and blue, and the third with endpoint colors green and blue.

P4.2.3 Consider the regular n-gon with vertices

$$\left(\cos\left(\frac{2\pi k}{n}\right), \sin\left(\frac{2\pi k}{n}\right) \right) \qquad k = 1{:}n.$$

Write a script that inputs n and draws a regular n-gon that is colored yellow and has a red perimeter.

P4.2.4 Write a script that draws an 8-by-8 checkerboard with red and black tiles.

P4.2.5 Write a script that draws an equilateral triangle that is partitioned into four smaller equilateral triangles by connecting the midpoints of its sides. The four little triangles should be colored differently.

P4.2.6 Define the ellipse $E(\theta, a, b)$ by

$$x(t) = a\cos(t)\cos(\theta) - b\sin(t)\sin(\theta)$$
$$y(t) = b\sin(t)\cos(\theta) + a\cos(t)\sin(\theta),$$

where $0 \leq t \leq 2\pi$ and θ is the "tilt angle." Write a script that displays $E(0,3,1)$, $E(\pi/6,3,1)$, $E(\pi/3,3,1)$, and $E(\pi/2,3,1)$ in four separate figures. Paint the ellipses magenta.

P4.2.7 Write a script that draws a bullseye with n concentric rings. The kth ring should have inner radius $k-1$ and outer radius k. (Thus, the innermost ring is just a radius-1 disk.) The kth ring should be colored white if k is odd and red if k is even.

P4.2.8 Write a script that draws the 5-ring Olympic symbol. Get the colors and proportions right!

P4.2.9 Look up the design parameters for the flags of Japan, Switzerland, France, and Sweden and write a script that produces accurate renditions in four separate figures.

P4.2.10 Write a script that draws an 11-by-11 checkerboard with the property that the tile in row i and column j has color $[\ (i-1)/11 \ \ 0 \ \ (j-1)/11\ ]$. Assume that row 1 is the bottom row and column 1 is the leftmost column.

4.3 One Third Plus One Third Is Not Two Thirds

Problem Statement

Consider the following command window interaction:

```
>> format long
>> x = 1/3
x =
   0.333333333333333
>> y = x+x
y =
   0.666666666666667
```

The computer is doing what we do with never-ending decimals: it rounds. Rounding is necessary because the hardware that is used to store numbers is finite.

The finiteness of computer arithmetic has other ramifications. For sufficiently large values of k, $1 + 1/2^k$ will equal 1 and $1/2^k$ will equal zero. Moreover, there is a limit to the size of 2^k. These features distinguish computer arithmetic, which is discrete, from real arithmetic, which is continuous.

Write a script that showcases the finiteness of computer arithmetic and sheds light on how much memory the computer allocates for the storage of a real value.

Program Development

Let us first confirm that there are indeed issues with computer arithmetic. At first glance, it sure looks like the following script would never terminate:

```
k = 0;
while (1 + 1/2^k) > 1                                    (4.7)
   k = k+1
end
```

However, it does terminate and the last value of k that it reports is 53. This is because the computer does *floating point arithmetic*, a system of calculation that basically represents numbers in a "constrained" scientific notation. The constraints are necessary because computer memory is finite—there is not enough room to store never-ending decimals like π or $\sqrt{2}$ or 1/3.

Recall that any nonzero number x can be expressed in the form

$$x = \pm m \times 10^e$$

where m satisfies $1 \le m < 10$ and e is an integer.[2] Here are some examples:

$$1230 = +1.23 \times 10^{+3} \qquad -.000083615 = -8.3615 \times 10^{-5}.$$

[2] "Classical" scientific notation restricts m to the range $1/10 \le m < 1$. The style we are adopting is sometimes referred to as "engineering notation."

The representation is unique. (A special convention is required for the representation of zero, e.g., 0.0×10^0.)

Floating point arithmetic systems represent real numbers in this style with limits placed on the precision of m and the size of e. To illustrate, let us consider a "toy" system in which three digits are allocated for m and one digit is allocated for e. Here are some numbers and their representations in this environment:

a	=	12.3	a:	$+$ 1 2 3	$+$ 1
b	=	.000000123	b:	$+$ 1 2 3	$-$ 7
c	=	-12.3	c:	$-$ 1 2 3	$+$ 1

Note that with 3-digit precision, some numbers can only be stored approximately:

x	=	12.34	x:	$+$ 1 2 3 $+$ 1
y	=	12.37	y:	$+$ 1 2 4 $+$ 1
z	=	π	z:	$+$ 3 1 4 $+$ 0

A reasonable thing to do if there is not enough room to store a value is to *round*. Since 12.37 is closer to 1.24×10^1 than 1.23×10^1, the former value is stored. A tie-breaking rule is required in the event that the value to be represented is midway between two floating point numbers.

Rounding is a necessary feature of just about every floating point calculation because the mantissa, m, of the answer is almost always bigger than the mantissas of the operands, e.g.,

$$[+\,1\,2\,3\,\|\,+\,1] \times [+\,4\,5\,6\,\|\,+\,1]$$

evaluates to 5.6088×10^2. The mantissa must be rounded. The official floating point product becomes

$$[+\,5\,6\,1\,\|\,+\,2]$$

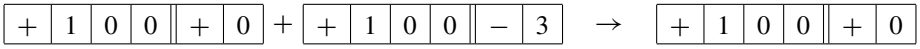
In general, when two floating point numbers are combined through addition, subtraction, multiplication, or division, the floating point result is the rounded version of the correct answer.

A consequence of having limited precision is that a small number can have zero impact when it shows up in an arithmetic operation. For example, in our toy system, $1 + 10^{-1}$ and $1 + 10^{-2}$ can be computed exactly:

$$[+\,1\,0\,0\,\|\,+\,0] + [+\,1\,0\,0\,\|\,-\,1] \rightarrow [+\,1\,1\,0\,\|\,+\,0]$$

$$[+\,1\,0\,0\,\|\,+\,0] + [+\,1\,0\,0\,\|\,-\,2] \rightarrow [+\,1\,0\,1\,\|\,+\,0]$$

while the floating point addition of 1 and 10^{-3} is 1:



This is because the floating point representation of 1.001 requires four digits which is beyond the capability of our toy system. Notice that if we were able to execute the script

```
k = 0;
while 1 + 1/10^k > 1
   k = k+1
end
```

on a computer with our toy floating point number system, then the last value of k displayed would be three—precisely the number of digits allocated to m.

We are now set to explain why on a "real computer" 53 is the smallest integer value of k for which the floating point addition of 1 and $1/2^k$ is 1. Computers encode information with 0's and 1's. This is why the base-2 system is used to represent numerical information. With a base-2 place value system, 1.0101 represents $1+5/16$ because

$$1\cdot 2^0+0\cdot 2^{-1}+1\cdot 2^{-2}+0\cdot 2^{-3}+1\cdot 2^{-4}=1\frac{5}{16}.$$

Just about every computer manufacturer implements the *IEEE floating point standard*. In this standard, 52 base-2 bits are allocated for the fraction part of m. Thus, the number 1 has the representation

$$1.\underbrace{000\cdots000}_{52\text{ bits}}$$

On the other hand, because the sum of 1 and $1/2^{53}$ has the form

$$1.\underbrace{000\cdots0001}_{53\text{ bits}}$$

it evaluates to 1 because there is not enough room to store the 53-bit fraction part.[3] This explains the output of the script (4.7).

Let us turn our attention to the floating point behavior of $1/2^k$ and 2^k, again using our toy system for motivation. It is natural to think that 10^{-9} is the smallest positive number that can be represented:

$$x=10^{-9}\qquad \texttt{x}:\ \boxed{+}\boxed{1}\boxed{0}\boxed{0}\,\|\,\boxed{-}\boxed{9}$$

However, if the system permits *unnormalized* representations, then we can encode even smaller values, e.g.,

$$x=10^{-11}\qquad \texttt{x}:\ \boxed{+}\boxed{0}\boxed{0}\boxed{1}\,\|\,\boxed{-}\boxed{9}$$

Thus, in our toy floating point environment it makes sense to regard any number smaller than 10^{-11} as zero.

[3] A "round-to-even" tie-breaking rule is used.

At the other extreme, the largest number our miniature system can represent is

$$x = 9.99 \times 10^9 \qquad \texttt{x:} \boxed{+}\boxed{9}\boxed{9}\boxed{9}\boxed{+}\boxed{9}$$

It is natural to regard any number larger than this as infinite.

How does the IEEE floating point standard treat these extreme situations? To answer this question we need to understand the overall IEEE representation. Altogether, 64 bits are used to represent a floating point number: one bit for the sign, 52 bits for the fraction part of m, and 11 bits for e including its sign. Analogous to why the smallest positive floating point number in the toy system is $10^{-2} \cdot 10^{-9} = 10^{-11}$, we find that the smallest positive number in the IEEE system is about $2^{-52} \cdot 2^{-2^{10}} \approx 2^{-1076}$. That is why 1075 is the last value of k displayed by the script

```
k = 0;
while 1/2^k > 0
   k = k+1
end
```

For very large numbers, the IEEE standard assigns the special value of `inf` if an expression evaluates to a quantity that is larger than the largest representable floating point number. The threshold is approximately $2^{2^{10}}$ and that explains why the last value of `k` reported by the script

```
k = 0;
while 2^k < inf
   k = k+1
end
```

is 1024.

The script `Eg4_3` illustrates all these features of the IEEE floating point standard. It is listed below together with its output.

Talking Point: Xeno Revisited

The ancient Greek Xeno of Elea posed a number of paradoxes that have bothered philosophers for centuries. The most famous can be framed in the context of trying to reach a wall through a succession of steps, each of which halves the remaining distance. If $d = 1$ at the start, then $d = 1/2$ after one step, $d = 1/2^2$ after two steps, $d = 1/2^3$ after three steps, etc. The paradox is that you apparently will never reach the wall. However, in the floating point context, you do arrive at your destination even though it may require 1075 steps! Apologies to Xeno.

In computational science and engineering we have to appreciate the finiteness of computer arithmetic. Rounding errors and exponent limits force a departure from business-as-usual mathematics. *Full machine precision* in the IEEE setting means approximately 16 significant (base-10) digits, and it is almost always a challenge to attain such small relative error in a calculation. The floating point scene is rife with less-is-more paradoxes, situations where an approximate algorithm can yield more accurate results than an allegedly

The Script `Eg4_3`

```
% Script Eg4_3
% Floating Point Number Facts

clc
% p = largest positive integer so 1+1/2^p > 1.
x=1; p=0; y=1; z=x+y/2;
while x~=z
    y = y/2;
    p = p+1;
    z = x+y/2;
end
fprintf(...
'p = %2.0f   is the largest positive integer so 1+1/2^p > 1.\n',p)

% q = smallest positive integer so 1/2^q = 0.
x = 1; q = 0;
while x>0
    x = x/2;
    q = q+1;
end;
fprintf(...
  'q = %2.0f is the smallest positive integer so 1/2^q == 0.\n',q)

% r = smallest positive integer so 2^r = inf.
x = 1; r = 0;
while x~=inf
    x = 2*x;
    r = r+1;
end
fprintf(...
  'r = %2.0f is the smallest positive integer so 2^r == inf.\n',r)
```

Sample Output from the Script `Eg4_3`

```
p = 52   is the largest positive integer so 1+1/2^p > 1.
q = 1075 is the smallest positive integer so 1/2^q = 0.
r = 1024 is the smallest positive integer so 2^r = inf.
```

exact algorithm. Xeno with a digital computer would have been the author of numerous paradoxes!

MATLAB Review

eps

This built-in constant is the machine precision, i.e., the smallest number ϵ such that $1+\epsilon > 1$ in floating point arithmetic. For the IEEE standard, it has the value $2^{-52} \approx 10^{-16}$.

inf

This is a special floating point number that behaves like infinity. The following expressions have value `inf`: `1/0`, `tan(pi/2)`, `abs(log(0))`, `sqrt(inf)`, `inf/10000`. If `x = inf`, then `a = 1/x` is assigned the value of zero and `a = -x` is assigned the value of `-inf`. The boolean expression `x==inf` is true if the value of `x` is `inf`.

realmax, realmin

This is the largest positive floating point number and smallest positive floating point number, respectively.

NaN

This is a special floating point number that is referred to as *not-a-number*. If `x = 0/0`, then `x` has the value `NaN`. If a variable has value `NaN`, then any expression involving the variable has value `NaN`.

Exercises

M4.3.1 Consider the following script:

```
x = input('Enter a positive number:');
z = x;
while x+z>x
   z = z/2;
end
```

What is the connection between the last value of `z`, the value of `x`, and `eps`?

M4.3.2 What is the value of `1/0 - 1/0`? What is the value of `2/0 - 1/0`? What is the value of `(1/0)/(1/0)`?

P4.3.3 Calculus tells us that for very small positive values of h,

$$e_h(x) = \left| \frac{\sin(x+h) - \sin(x)}{h} - \cos(x) \right| = O(h).$$

Write a script that inputs x in the range $[0, 2\pi]$ and prints out the value of $e_h(x)$ for $h = 1/10, 1/100, \ldots, 1/10^{16}$. What value of h minimizes the error? Note that in the evaluation of the divided difference, any errors in the evaluation of $\sin(x+h) - \sin(x)$ are magnified by $1/h$. Thus, as h goes to zero the "calculus" error goes to zero but the roundoff error goes to infinity. Thus, the "optimum" choice of h reflects the need to compromise these two tendencies.

P4.3.4 Plot the functions $f(x) = (1-x)^6$ and

$$g(x) = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$$

across the interval [.995,1.005]. Even though $f = g$, the plots will look very different. This is because the g-evaluation attempts to compute very small values through the "lucky cancellation" of large values while the f-evaluation computes very small values through repeated multiplication of modestly small values.

P4.3.5 Horner's scheme rearranges a polynomial as follows:

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 = ((a_n x + a_{n-1})x + \cdots)x + a_0.$$

Evaluating the polynomial in this order helps reduce the cancellation error discussed in P4.3.4. Plot the function $g(x) = x^6 - 6x^5 + 15x^4 - 20x^3 + 15x^2 - 6x + 1$ across the interval [.995,1.005] using Horner's scheme and using the "typical" evaluation order ($g(x)$ exactly as shown above). You will see that the cancellation error using Horner's scheme is smaller.

P4.3.6 What is the smallest value of `n` such that the value of `factorial(n)` is `inf`?

P4.3.7 How many base-10 digits are there in 1000!?