1 Numeric Types in Matlab

In this chapter we will explore how Matlab stores various types of numbers. First, we will investigate the various integer types that Matlab offers. After a thorough discussion of integer storage, we will move on to the more difficult topic of floating point numbers, the technique used to store approximations of real numbers in Matlab.

Table of Contents

1.1 Integer Types in Matlab

In this section we will introduce the various datatypes available in Matlab that are used for storing integers. There are two distinct types: one for unsigned integers, and a second for signed integers. An unsigned integer type is only capable of storing positive integers (and zero) in a well defined range. Unsigned integer types are used to store both positive and negative integers (and zero) in a well defined range.

Each type, signed and unsigned, has different classes that are distinguished by the number of bytes used for storage. As we shall see, **uint8**, **uint16**, **uint32**, and **uint64** use 8 bits, 16 bits, 32 bits, and 64 bits to store unsigned integers, respectively. On the other hand, **int8**, **int16**, **int32**, and **int64** use 8 bits, 16 bits, 32 bits, and 64 bits to store signed integers, respectively.

Let's begin with a discussion of the base ten system for representing integers.

Base Ten

Most of us are familiar with base ten arithmetic, simply because that is the number system we have been using for all of our lives. For example, the number 2345, when expanded in powers of ten, is written as follows.

$$\begin{aligned} 2345 &= 2000 + 300 + 40 + 5 \\ &= 2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5 \\ &= 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 \end{aligned}$$

There is an old-fahsioned algorithm which will allows us to expand the number 2345 in powers of ten. It involves repeatedly dividing by 10 and listing the remainders, as shown in **Table 1.1**.

10	2345	
10	234	5
10	23	4
10	2	3
10	0	2

Table 1.1. Determining the coefficients of the powers of ten.

If you read the remainders in the third coloumn in reverse order (bottom to top), you capture the coefficients of the expansion in powers of ten, namely the 2, 3, 4, and 5 in $2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$.

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

In the case of base ten, the algorithm demonstrated in **Table 1.1** is a bit of overkill. Most folks are not going to have trouble writing 8235 as $8 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 5 \cdot 10^0$. However, we will find the algorithm demonstrated in **Table 1.1** quite useful when we want to express base tens numbers in a different base.

The process is easily reversible. That is, it is a simple matter to expand a number that is experessed in powers of ten to capture the original base ten integer.

$$\begin{aligned} 2 \cdot 10^3 + 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0 &= 2 \cdot 1000 + 3 \cdot 100 + 4 \cdot 10 + 5 \\ &= 2000 + 300 + 40 + 5 \\ &= 2345 \end{aligned}$$

Base Ten. An integer can be expressed in base ten as

$$t_n \cdot 10^n + t_{n-1} \cdot 10^{n-1} + \cdots + t_2 \cdot 10^2 + t_1 \cdot 10^1 + t_0 \cdot 10^0,$$

where each of the coefficients t_n, t_{n-1}, $\ldots t_1$, t_1, and t_0 are "digits," i.e., one of the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. Note that the highest possible coefficient is one less than the base.

However, base ten is not the only possibility. Indeed, we are free to use any base that we wish. For example, we could use base seven. If we did, then the number $(2316)_7$ would be interpreted to mean

$$(2316)_7 = 2 \cdot 7^3 + 3 \cdot 7^2 + 1 \cdot 7^1 + 6 \cdot 7^0.$$

This is easily expanded and written in base ten.

$$\begin{aligned} (2316)_7 &= 2 \cdot 343 + 3 \cdot 49 + 1 \cdot 7 + 6 \cdot 1 \\ &= 686 + 147 + 7 + 6 \\ &= 846 \end{aligned}$$

Base Seven. An integer can be expressed in base seven as

$$s_n \cdot 7^n + s_{n-1} \cdot 7^{n-1} + \cdots + s_2 \cdot 7^2 + s_1 \cdot 7^1 + s_0 \cdot 7^0,$$

where each of the coefficients s_n, s_{n-1}, $\ldots s_1$, s_1, and s_0 are one of the numbers 0, 1, 2, 3, 4, 5, or 6. Note that the highest possible coefficient is one less than the base.

Matlab has a useful utility called **base2dec** for converting numbers in different bases to base ten. You can learn more about this utility by typing **help base2dec** at the Matlab prompt.

```
>> help base2dec
 BASE2DEC Convert base B string to decimal integer.
    BASE2DEC(S,B) converts the string number S of base B into
    its decimal (base 10) equivalent.  B must be an integer
    between 2 and 36. S must represent a non-negative integer
    value.
```

Strings in Matlab are delimited with single apostrophes. Therefore, if we wish to use this utility to change the base seven $(2316)_7$ to base ten, we enter the following at the Matlab prompt.

```
>> base2dec('2316',7)
ans =
   846
```

Note that this agrees with our hand calculation above.

Hopefully, readers will now intuit that integers can be expressed in terms of an aribitrary base.

Arbitrary Base. An integer can be expressed in base B as

$$c_n \cdot B^n + c_{n-1} \cdot B^{n-1} + \cdots + c_2 \cdot B^2 + c_1 \cdot B^1 + c_0 \cdot c^0,$$

where each of the coefficients c_n, c_{n-1}, $\ldots c_1$, c_1, and c_0 are one of the numbers 0, 1, 2, $\ldots$, $B-1$. Note that the highest possible coefficient is one less than the base.

It is important to note the restriction on the coefficients. If you expand an integer in powers of 3, the permissible coefficients are 0, 1, and 2. If you expand an integer in powers of 8, the permissible coefficients are 0, 1, 2, 3, 4, 5, 6, and 7.

Binary Integers

At the most basic level, the fundamental storage unit on a computer is called a **bit**. A bit has two states: it is either "on" or it is "off." The states "on" and "off" are coded with the integers 1 and 0, respectively. A *byte* is made up of eight

bits, each of which has one of two states: "on" (1) or "off" (0). Consequently, computers naturally use base two arithmetic.

As an example, suppose that we have a byte of storage and the state of each bit is coded as 10001011. The highest ordered bit is "on," the next three are "off', the next one is "on," followed by an "off," and finally the last two bits are "on." This represents the number $(10001011)_2$, which can be converted to base ten as follows.

$$\begin{aligned}(10001011)_2 &= 1\cdot 2^7 + 0\cdot 2^6 + 0\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 \\ &= 128 + 0 + 0 + 0 + 8 + 0 + 2 + 1 \\ &= 139\end{aligned}$$

This is easily checked with Matlab's **base2dec** utility.

```
>> base2dec('10001011',2)
ans =
   139
```

However, since base two is commonly used when working with computers, Matlab has a special command for changing base two numbers into base ten numbers called **bin2dec**.

```
>> help bin2dec
 BIN2DEC Convert binary string to decimal integer.
    X = BIN2DEC(B) interprets the binary string B and returns
    in X the equivalent decimal number.
```

We can use **bin2dec** to check our conversion of $(10001011)_2$ to a base ten number.

```
>> bin2dec('10001011')
ans =
   139
```

This process is reversible. We can start with the base ten integer 139 and change it to base two by extracting powers of two. To begin, the highest power of two contained in 139 is $2^7 = 128$. Subtract 128 to leave a remainder of 11. The highest power of two contained in 11 is $2^3 = 8$. Subtract 8 from 11 to leave a remainder

of 3. The highest power of two contained in 3 is $2^1 = 2$. Subtract 2 from 3 to leave a remainder of 1. Thus,

$$\begin{aligned} 139 &= 128 + 8 + 2 + 1 \\ &= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0. \end{aligned}$$

Thus, $139 = (10001011)_2$.

However, this process is somewhat tedious, particularly for larger numbers. We can use the tabular method (shown previously for powers of ten in **Table 1.1**), repeatedly dividing by 2 while listing our remainders in a third column, as shown in **Table 1.2**.

2	139	
2	69	1
2	34	1
2	17	0
2	8	1
2	4	0
2	2	0
2	1	0
2	0	1

Table 1.2. Determining the coefficients of the powers of two.

If you read the remainders in the third column of **Table 1.2** in reverse order (bottom to top), you capture the coefficients of the expansion in powers of two, providing $(139)_{10} = (10001011)_2$.

Matlab provides a utility called **dec2bin** for changing base ten integers to base two.

```
>> dec2bin(139)
ans =
10001011
```

Note that this agrees nicely with our tabular result in **Table 1.2**.

Hexedecimal Integers

As the number of bits used to store integers increases, it becomes painful to deal with all the zeros and ones. If we use 16 bits, most would find it challenging to correctly write a binary number such as

$$(1110001000001111)_2.$$

This number, when expanded in powers of 2, becomes

$$\begin{aligned} &1\cdot 2^{15}+1\cdot 2^{14}+1\cdot 2^{13}+0\cdot 2^{12}\\ &\quad +0\cdot 2^{11}+0\cdot 2^{10}+1\cdot 2^{9}+0\cdot 2^{8}\\ &\quad +0\cdot 2^{7}+0\cdot 2^{6}+0\cdot 2^{5}+0\cdot 2^{4}\\ &\quad +1\cdot 2^{3}+1\cdot 2^{2}+1\cdot 2^{1}+1\cdot 2^{0}. \end{aligned}$$

This can be rewritten as follows.

$$\begin{aligned} &(1\cdot 2^{3}+1\cdot 2^{2}+1\cdot 2^{1}+0\cdot 2^{0})\cdot 2^{12}\\ &\quad +(0\cdot 2^{3}+0\cdot 2^{2}+1\cdot 2^{1}+0\cdot 2^{0})\cdot 2^{8}\\ &\quad +(0\cdot 2^{3}+0\cdot 2^{2}+0\cdot 2^{1}+0\cdot 2^{0})\cdot 2^{4}\\ &\quad +(1\cdot 2^{3}+1\cdot 2^{2}+1\cdot 2^{1}+1\cdot 2^{0}.) \end{aligned} \tag{1.1}$$

This is equivalent to the following expression.

$$14\cdot (2^4)^3+2\cdot (2^4)^2+0\cdot (2^4)^1+15\cdot (2^4)^0$$

Finally, we see that our number can be expanded in powers of 16.

$$14\cdot 16^3+2\cdot 16^2+0\cdot 16^1+15\cdot 16^0. \tag{1.2}$$

We need to make two points:

1. Because we are expanding in base 16, the coefficients must be selected from the integers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15, as they are in the above expansion (the coefficients are 14, 2, 0 and 15).

2. The coeffients 10, 11, 12, 13, 14, and 15 are not single digits.

To take care of the second point, we make the following assignments: $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, and $F = 15$. With these assgignments, we can rewrite the expression in (**1.2**) as

$$E\cdot 16^3+2\cdot 16^2+0\cdot 16^1+F\cdot 16^0. \tag{1.3}$$

In practice, this is written in *hexedecimal format* as $(E20F)_{16}$.

We can check our result using Matlab utilities. First, use **bin2dec** to change $(1110001000001111)_2$ into decimal format.

```
>> bin2dec('1110001000001111')
ans =
       57871
```

Follow this with Matlab's **dec2hex** command to find the hexedecimal representation.

```
>> dec2hex(57871)
ans =
E20F
```

Note that this agrees with our hand-crafted result (**1.3**).

In practice, changing an integer from binary to hexedecimal is not as complicated as it might appear. In (**1.1**), note how we first started by breaking the binary integer $(1110001000001111)_2$ into groups of four. Each group of four eventually led to a coefficient of a power of 16. Thus, to move faster, simply block the binary number $(1110001000001111)_2$ into groups of four, starting from the right end.

$$(1110001000001111)_2 = (1110 - 0010 - 0000 - 1111)_2$$

Now, moving from left to right, $1110 = E$, $0010 = 2$, $0000 = 0$, and $1111 = F$, so

$$(1110 - 0010 - 0000 - 1111)_2 = (E20F)_{16}.$$

Pretty slick!

Unsigned Integers

We will now discuss Matlab's numeric types for storing *unsigned integers*. Unsigned integers are nonnegative. Negative integers are excluded. If you are working on a project that does not require negative integers, then the unsigned integer can save storage space.

Let's start with a storage space of one byte (eight bits), where each bit can attain one of two states: "on" (1) or "off" (0). A useful analogy is to think of the odometer in your car. Old fashioned base ten odometers (before digital) consisted of a sequence of dials, each having the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 imprinted

on them. When your car was brand new, before the wheels even rolled forward an inch, the dial would read 00000000 (we're assuming eight dials here). As the car took to the highway, the dial farthest to the right would rotate through 1 mile to 00000001, then 2 miles to 00000002, etc., until it reached 9 miles and recorded 00000009. Now, as the car moves through the tenth mile, the far right wheel rotates and returns to the digit 0 and the second to the last wheel rotates to the digit 1, providing the number 00000010 on the odometer. This is simply base ten counting in action.

Now imagine a base two odometer with eight dials, each having the digits 0 and 1 imprinted on them. At the start, the odometer reads 00000000. After traveling 1 mile, the odometer reads 00000001. At the end of mile 2, the last wheel must rotate and return to zero, and the second to the last wheel rotates around to 1, providing 00000010. At the end of mile 3, the last wheel spins again to 1 providing 00000011. At the end of mile 4, the last wheel must spin back to zero, the second to last wheel must now spin to zero, and the third to the last wheel spins to 1, providing 00000100. This is base two counting in action.

The base two odoemter with eight wheels represents one byte, or eight bits. The smallest possible integer that can be stored in one byte (8 bits) is the integer $(00000000)_2$, which is equal to the integer zero is base ten. The largest possible integer that can be stored is $(11111111)_2$, which can be converted to base ten with the following calculation.

$$\begin{aligned}(11111111)_2 &= 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 \\ &= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 \\ &= 255\end{aligned}$$

This last calculation is a bit painful. To convert $(11111111)_2$ to base ten, let's be a little more creative and let the odometer roll through 1 additional mile, arriving at $(100000000)_2$ (presuming we magically add one more wheel). This number is easily converted to base ten, as $(100000000)_2 = 2^8 = 256$. Because the number $(11111111)_2$ is one less than $(100000000)_2$,

$$(11111111)_2 = 2^8 - 1 = 256 - 1 = 255,$$

which is identical to the result calculated above. Of coure, we can use Matlab to check our result.

```
>> bin2dec('11111111')
ans =
   255
```

Thus, with 8 bits (1 byte) of memory, we are able to store unsigned integers in the range 0 to 255. Matlab has a special datatype, **uint8**, designed specifically for this purpose. For example, suppose that we want to store 123 as an unsigned 8-bit integer in the variable x.

```
>> x=uint8(123)
x =
  123
```

We can obtain information on the variable x with Matlab's **whos** command.

```
>> whos('x')
  Name      Size                    Bytes  Class

  x         1x1                         1  uint8 array
```

Note that the size is **1x1**, which represents a one-by-one matrix (one row by one column), so x must be a scalar. Secondly, note that the class is **uint8 array** as expected. Finally, note that it takes 1 byte to store the number 123 in the variable x.

You can determine the maximum and minimum integers that can be stored using the `uint8` datatype with the following commands.

```
>> intmin('uint8')
ans =
    0
>> intmax('uint8')
ans =
  255
```

Numbers outside this range "saturate." That is, numbers smaller than 0 are mapped to zero, numbers larger than 255 are mapped to 255.

```
>> uint8(273)
ans =
  255
>> uint8(-13)
ans =
    0
```

If you know in advance that you won't need any integers outside **uint8**'s range $[0, 255]$, then you can save quite a bit of storage space by using the **uint8** datatype (e.g., in the processing of gray-scale images). However, if you have need of larger integers, then you will need to use more storage space. Fortunately, Matlab has three other unsigned integer types, **uint16**, **uint32**, and **uint64**, that can be used to store larger unsigned integers.

As you might imagine, **uint16** uses 16 bits (2 bytes) of memory to store an unsigned integer. Again, the smallest possible integer that can be stored using this type is $(0000000000000000)_2 = 0$. On the top end, the largest unsigned integer that can be stored using this data type is $(1111111111111111)_2$. Again, you can change this to base ten by noting that $(1111111111111111)_2$ is 1 less than the binary integer $(1000000000000000)_2$. That is,

$$(1111111111111111)_2 = (1000000000000000) - 1 = 2^{16} - 1 = 65535.$$

Again, you can check these bounds on the range of **uint16** with the following commands.

```
>> intmin('uint16')
ans =
      0
>> intmax('uint16')
ans =
  65535
```

Thus, any number in the range $[0, 65535]$ can be stored using **uint16**.

For example, we can again store the number 123 in x, but this time as an unsigned 16 bit integer.

```
>> x=uint16(123)
x =
    123
```

On the surface, there doesn't appear to be any difference. However, the **whos** command reveals the difference.

```
>> whos('x')
  Name      Size                    Bytes  Class

  x         1x1                         2  uint16 array
```

Note that this time the class is `uint16 array`, but more importantly, note that it now takes two bytes (16 bits) of memory to store the number 123 in the variable x.

Integers outside the range $[0, 65535]$ again saturate. Integers smaller than zero are mapped to zero; integers larger than 65535 are mapped to 65535.

```
>> uint16(-123)
ans =
      0
>> uint16(123456)
ans =
  65535
```

Two further datatypes for unsigned integers exist, **uint32** and **uint64**, which use 32 bits (4 bytes) and 64 bits (8 bytes) to store unsigned integers, respectively.

Signed Integers

You probably noticed the conspicuous absence of negative numbers in our discussion of unsigned integers and the Matlab datatypes **uint8**, **uint16**, **uint32**, and **uint64**. We will rectify that situation in this section.

First, let's discuss Matlab's signed integer datatype **int8**, which uses 8 bits (one byte) to store *signed* integers. The leftmost bit (most significant bit) is used to denote the sign of the integer. If this first bit is "on" (1), then the integer is negative, and if this first bit is "off" (0), then the integer is positive. Consequently, the largest possible positive integer that can be stored with this strategy is the binary number $(01111111)_2$. Note that this number is one less than the number $(10000000)_2$ (recall the analogy of the base two odometer). Thus,

$$(01111111)_2 = (10000000)_2 - 1 = 2^7 - 1 = 127.$$

This is easily verified with Matlab's `intmax` command.

```
>> intmax('int8')
ans =
  127
```

To represent negative numbers with this storage strategy, computers (and Matlab) use a technique called *twos complement* to determine a negative integer. For example, consider the number 7, written in binary.

$$7 = (00000111)_2$$

To determine how -7 is stored using signed 8 bit arithmetic, we "complement" each bit, then add 1. When we say that we will "complement each bit," we mean that we will replace all zeros with ones and all ones with zeros. Again, complement each bit of $7 = (00000111)_2$ then add 1.

$$-7 = (11111000)_2 + (00000001)_2 = (11111001)_2.$$

We can verify this result by adding the binary representations of 7 and -7.

	^{1}0	^{1}0	^{1}0	^{1}0	^{1}0	^{1}1	^{1}1	1
+	1	1	1	1	1	0	0	1
	0	0	0	0	0	0	0	0

This "binary addition" warrants some explanation. First one and one is two, correct? In binary, $(1)_2$ and $(1)_2$ is $(10)_2$. In the addition above, we will add as we did in elementary school. We start at the right end, add $(1)_2$ and $(1)_2$, which is $(10)_2$. We write the zero in the result and "carry" the 1 to the next column. Of course, this means that in the second to last column we are now adding $(0)_2$, $(1)_2$, and the "carried" $(1)_2$, which is again $(10)_2$. So, we write the zero in the second to last column of the answer, then "carry" the 1 to the next column. Proceeding in this manner, one can see that all the columns will "zero out." When we finally get to the first column, the "carried" $(1)_2$ and the $(1)_2$ and $(0)_2$ sum again to $(10)_2$. We write the zero in the result, but when we try to "carry" the 1, it gets "pushed off" the left end where there is no further storage space and (poof!) disappears.

Hence,

$$(00000111)_2 + (11111001)_2 = (00000000)_2.$$

This makes $(11111001)_2$ the negative of $(00000111)_2$. Hence, $(11111001)_2 = -7$. If we were using unsigned storage, $(11111001)_2$ would equal 249 in base ten, but with signed storage, this spot is now reserved for -7.

Twos Complement (8 bit). To determine the negative of a integer stored as a signed 8 bit (one byte) integer, use "twos complement." That is, complement each bit (change zeros to ones and ones to zeros), then add one.

As a second example, consider $127 = (01111111)_2$. To determine how -127 is stored as an 8 bit signed integer, complement each bit of $127 = (01111111)_2$ and add one. That is,

$$-127 = (10000000)_2 + (00000001)_2 = (10000001)_2.$$

Note that if we were using *unsigned* 8 bit integer storage, $(10000001)_2$ would represent the next number after 128, namely 129. But in *signed* 8 bit integer storage, $(10000001)_2$ represents -127. As a check, note that adding the binary representations of 127 and -127 produces zero.

$$\begin{array}{cccccccccc} & {\scriptstyle 1}0 & {\scriptstyle 1}1 & {\scriptstyle 1}1 & {\scriptstyle 1}1 & {\scriptstyle 1}1 & {\scriptstyle 1}1 & {\scriptstyle 1}1 & 1 \\ + & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Using 8 bit unsigned storage, we can represent the numbers 0 through 255, as labeled on the top of the number line in **Figure 1.1**.

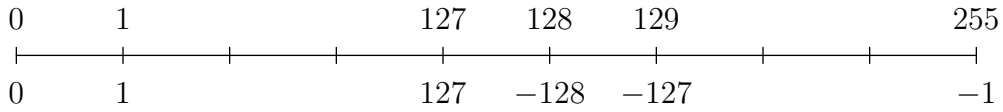

Figure 1.1. Unsigned integers on top are mapped to signed integers on the bottom.

With signed storage, when the binary odometer rolls over from $127 = (01111111)_2$ to $(10000000)_2$, the leading bit is now "on," so all numbers from this point will represent negative integers, as shown below the number line in **Figure 1.1**.

The mapping of the unsigned integers from 129 to 255 to the signed integers from -127 to -1 on the number line in **Figure 1.1** is best portrayed if we place the signed and unsigned integers on a circle, as shown in **Figure 1.2**.

Thus, using signed 8 bit signed integer storage, we can represent any integer between -128 and 127, including -128 and 127.

Perhaps a simpler argument for the range of signed 8 bit integers can be made by noting that 8 bits affords space for $2^8 = 256$ integers. If we count the integers from 0 through 127 inclusive, we find there are 128 integers. Subtracting from 256 tells us that we have room for an additional 128 integers. Note that the negative

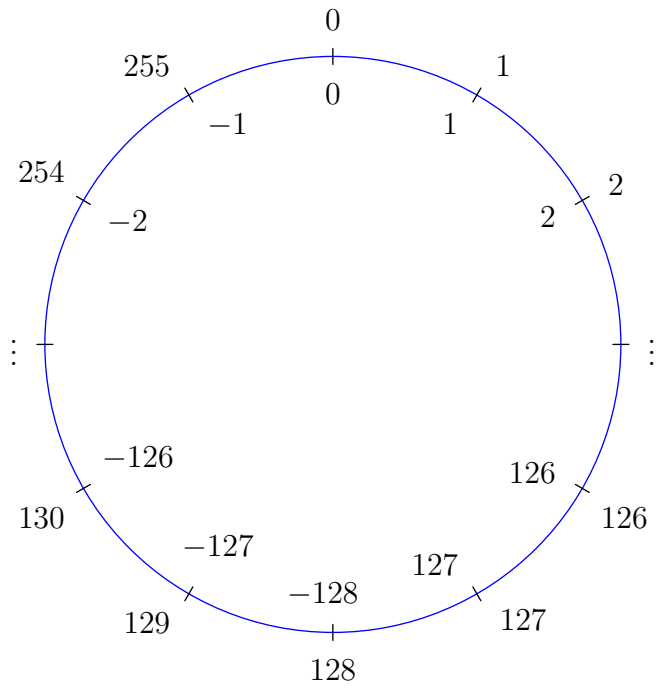

Figure 1.2. Unsigned integers on the outer rim are mapped to signed integers on the inner rim.

integers, starting at -1 and ending at -128, are 128 in number. These maximum and minimum values are easily verified using Matlab.

```
>> intmin('int8')
ans =
 -128
>> intmax('int8')
ans =
  127
```

As an example, we can store -123 in the variable x using signed 8 bit integer storage.

```
>> x=int8(-123)
x =
 -123
```

The whos command reveals the class of the variable x and the amount of memory required to store -123 in the variable x.

```
>> whos('x')
  Name      Size                    Bytes  Class

  x         1x1                         1  int8 array
```

Note that the class is `int8 array` and one byte is required to store the integer -123 in x.

Three further datatypes for signed integers exist, **int16**, **int32**, and **int64**. They allot 16, 32, and 64 bits for signed integer storage, respectively.

1.1 Exercises

In **Exercises 1-4**, use hand calculations (and a calculator) to change each of the numbers in the given base to base ten. Use Matlab's **base2dec** command to check your answer.

1. $(3565)_7$

2. $(2102)_3$

3. $(11111111)_2$

4. $(111011011)_2$

In **Exercises 5-9**, use the tabular technique to change each of the given base ten integers to base two. Check your results with Matlab's **dec2bin** command.

5. 127

6. 67

7. 255

8. 256

In **Exercises 9-12**, use hand calculations to place each of the given binary numbers in hexedecimal format. Use Matlab's **bin2dec** and **dec2hex** commands to check your work.

9. $(11110101)_2$

10. $(1110011101011001)_2$

11. $(11111101101010010101111110101100)_2$

12. $(11101101101010010101111110111100)_2$

In **Exercises 13-16**, Use the tabular method demonstrated in **Table 1.2** to place each of the given base ten integers into binary format. Then place your result in hexidecimal format. Check your results with Matlab's **dec2bin** and **dec2hex** commands.

13. 143

14. 509

15. 1007

16. 12315

17. Using hand calculations, determine the range of the unsigned integer datatypes **uint32** and `uint64`. Use Matlab's **intmin** and **intmax** commands to verify your solutions. Store the number 123 in x, using each datatype, then use the **whos** command to determine the class and storage requirements for the variable x.

18. Using hand calculations, determine the range of the signed integer types **int16**, **int32**, and **int64**. Use Matlab's **intmin** and **intmax** commands to check your results. Store the number -123 in x, using each datatype, then use the **whos** command to determine the class and storage requirements for the variable x.

1.1 Answers

---

1. 1321

3. 255

5. $(1111111)_2$

7. $(11111111)_2$

9. $(F5)_{16}$

11. $(FDA95FAC)_{16}$

13. $(10001111)_2$

15. $(1111101111)_2$

17. Range for **uin32** is $[0, 4294967295]$.
Range for **uint64** is $[0, 18446744073709551615]$.

1.2 Decimals and Fractions

In this section we will continue our exploration of numbers in binary format. The specific task in this section is to determine how to represents decimals and fractions in binary and hexadecimal format, a skill we need to master before learning how to store real numbers in floating point format on computers.

Decimals in Binary

Recall that the decimal 0.1485 can be expanded in powers of 10 as follows.

$$0.1485 = 1 \cdot 10^{-1} + 4 \cdot 10^{-2} + 8 \cdot 10^{-3} + 5 \cdot 10^{-4}$$

In similar fashion, we can expand a binary decimal in powers of 2.

$$\begin{aligned}(0.1101)_2 &= 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} \\ &= \frac{1}{2} + \frac{1}{4} + \frac{0}{8} + \frac{1}{16} \\ &= \frac{13}{16}\end{aligned}$$

In base ten, multiplying by 10 will move the decimal one place to the right.

$$\begin{aligned}0.1485 \cdot 10 &= (1 \cdot 10^{-1} + 4 \cdot 10^{-2} + 8 \cdot 10^{-3} + 5 \cdot 10^{-4}) \cdot 10 \\ &= 1 \cdot 10^{0} + 4 \cdot 10^{-1} + 8 \cdot 10^{-2} + 5 \cdot 10^{-3} \\ &= 1.485\end{aligned}$$

This fact gives us an algorithm for determining the coefficients of a decimal expansion in powers of 10.

In **Table 1.3**, begin by placing 0.1485 in the third column of the first row. Multiply by 10 and place the result in the second column of the second row. Strip the digit to the left of the decimal point in the result and place it in the fourth column of the second row. Place the remainder in the third column of the second row. Iterate until no more digits exist to the right of the decimal point.

10		0.1485	
10	1.485	0.485	1
10	4.85	0.85	4
10	8.5	0.5	8
10	5.	0.	5

Table 1.3. Repeated multiplication by 10 shifts the coefficients to the left of the decimal point, one-by-one.

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

Finally, the digits in the fourth column, read from top to bottom, will be the coefficients in the expansion in powers of 10.

This process might not terminate. For example, we follow the same procedure to find the coefficients for the decimal expansion of 1/7 in powers of ten in **Table 1.4**.

10		1/7	
10	$10/7 = 1 + 3/7$	3/7	1
10	$30/7 = 4 + 2/7$	2/7	4
10	$20/7 = 2 + 6/7$	6/7	2
10	$60/7 = 8 + 4/7$	4/7	8
10	$40/7 = 5 + 5/7$	5/7	5
10	$50/7 = 7 + 1/7$	1/7	7
10	$10/7 = 1 + 3/7$	3/7	1

Table 1.4. Sometimes the process produces a repeating decimal.

Note that in the last row of **Table 1.4**, we have entries identical to the second row. Hence, the pattern $1/7 = 0.142857142857\ldots$ repeats indefinitely.

In base two, multiplying by 2 will move the decimal one place to the right.

$$\begin{aligned}(0.1101)_2 \cdot 2 &= (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}) \cdot 2 \\ &= 1 \cdot 2^{0} + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= (1.101)_2\end{aligned}$$

This means that the algorithm demonstrated in **Tables 1.3** and **1.4** will work equally well for base two. We simply multiply repeatedly by two instead of ten.

▶ **Example 1.** *Find the binary representation of the base ten decimal* 0.875.

Place 0.875 in the third column of row one in **Table 1.5**. Multiply by 2 and place the result in the second column of row two. Strip off the digit to the left of the decimal point of this result and place it in the fourth column of row two. Place the remainder in the third column of row two. Iterate.

Note that further multiplication by 2 will only produce zeros, which we can ignore. Hence, $0.875 = (0.111)_2$.

The result is easily checked by expanding in powers of two.

2		0.875	
2	1.75	0.75	1
2	1.5	0.5	1
2	1.	0.	1

Table 1.5. Repeated multiplication by 2 shifts the coefficients to the left of the decimal point, one-by-one.

$$\begin{aligned}(0.111)_2 &= 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \\ &= \frac{7}{8}\end{aligned}$$

If you divide 7 by 8 (or repeat the base ten algorithm of **Table 1.3**), you will see that $7/8 = 0.875$. Of course, you could also check in Matlab.

```
>> 7/8
ans =
    0.8750
```

As with base ten, some decimals have binary expansions that will repeat.

▶ **Example 2.** *Find a binary expansion for the decimal* 0.1.

In **Table 1.6**. After repeatedly multiplying by 2, a pattern emerges.

2		0.1	
2	0.2	0.2	0
2	0.4	0.4	0
2	0.8	0.8	0
2	1.6	0.6	1
2	1.2	0.2	1
2	0.4	0.4	0
2	0.8	0.8	0
2	1.6	0.6	1

Table 1.6. Sometimes the process produces a repeating decimal.

Thus,

$$0.1 = (0.0001100110011001100\ldots)_2.$$

Note further that because of its infinite repeating nature, it is not possible to store the binary number $(0.0001100110011001\ldots)_2$ **exactly** in a computer that uses finite base two storage.

Decimals in Hex

We can expand a hexadecimal in powers of 16. Recall that working with base 16, $A = 10$, $B = 11$, $C = 12$, $D = 13$, $E = 14$, and $F = 15$. So, for example,

$$\begin{aligned}(0.2A8F)_{16} &= 2 \cdot 16^{-1} + A \cdot 16^{-2} + 8 \cdot 16^{-3} + F \cdot 16^{-4} \\ &= \frac{2}{16} + \frac{10}{16^2} + \frac{8}{16^3} + \frac{15}{16^4} \\ &= \frac{2 \cdot 16^3 + 10 \cdot 16^2 + 8 \cdot 16 + 15}{16^4} \\ &= \frac{10895}{65536}.\end{aligned}$$

In the next few examples, we convert binary decimals into hexadecimals.

▶ **Example 3.** *Convert the binary decimal* $(0.111011110011)_2$ *into hexadecimal format.*

We can expand the binary decimal $(0.111011110011)_2$ in powers of 2.

$$\begin{aligned}&1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} \\ &\quad + 1 \cdot 2^{-5} + 1 \cdot 2^{-6} + 1 \cdot 2^{-7} + 1 \cdot 2^{-8} \\ &\quad + 0 \cdot 2^{-9} + 0 \cdot 2^{-10} + 1 \cdot 2^{-11} + 1 \cdot 2^{-12}\end{aligned}$$

This can be rewritten as follows.

$$\begin{aligned}&(1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) \cdot 2^{-4} \\ &\quad + (1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \cdot 2^{-8} \\ &\quad + (0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \cdot 2^{-12}\end{aligned}$$

This is equivalent to the following expression.

$$13 \cdot (2^{-4})^1 + 15 \cdot (2^{-4})^2 + 3 \cdot (2^{-4})^3$$

However, $2^{-4} = (2^4)^{-1} = 16^{-1}$, so this last expression can be written as

$$13 \cdot (16^{-1})^1 + 15 \cdot (16^{-1})^2 + 3 \cdot (16^{-1})^3,$$

which is equivalent to

$$13 \cdot 16^{-1} + 15 \cdot 16^{-2} + 3 \cdot 16^{-3}.$$

Finally, $D = 13$ and $F = 15$ in hex, so

$$(0.111011110011)_2 = (DF3)_{16}.$$

In pracice, we don't have to work so hard. We can simply group the binary digits in groups of four, then use the mappings in **Table 1.7**.

Binary	Hex	Binary	Hex
0	0	1000	8
01	1	1001	9
10	2	1010	A
11	3	1011	B
100	4	1100	C
101	5	1101	D
110	6	1110	E
111	7	1111	F

Table 1.7. Binary-Hex conversions.

Thus,

$$(0.111011110011)_2 = (0.1110 - 1111 - 0011)_2 = (DF3)_{16}.$$

Let's look at another example.

▶ **Example 4.** *In* **Example 2**, *we found a binary decimal for the base ten decimal 0.1. Convert that expansion to hexadecimal.*

In **Example 2**, we found

$$0.1 = (0.0001100110011001\ldots)_2.$$

We can group the binary digits in groups of four,

$$(0.0001 - 1001 - 1001 - 1001 - 1001 - \ldots)_2,$$

then use the conversions from **Table 1.7** to make the change to hexadecimal.

$$(0.19999\cdots)_{16}$$

1.2 Exercises

For each of the base ten decimals in **Exercises 1-4**, use hand calculations to find an equivalent binary decimal. Check your result by expanding in powers of two and simplifying, then use Matlab to check that your fractional result is equivalent to the original decimal.

1. 0.3125

2. 0.53125

3. 0.6171875

4. 0.232421875

In **Exercises 5-8**, use hand calculations find a repeating binary expansion for the given base ten decimal.

5. 0.3

6. 0.9

7. 0.05

8. 0.15

In **Exercises 9-14**, use hand calculations to place the given binary decimal in hexadecimal format.

9. $(0.10110111)_2$

10. $(0.111110110011)_2$

11. $(0.0111011111101101)_2$

12. $(0.0111011111001111)_2$

13. $(0.0111011101110111\ldots)_2$

14. $(0.001101100000011000000110\ldots)_2$

In **Exercises 15-18**, use hand calculations to place the given base ten decimal in hexadecimal format.

15. 0.259765625

16. 0.88671875

17. 0.8

18. 1/3

1.2 Answers

1. $(0.0101)_2$

3. $(0.1001111)_2$

5. $(0.0100110011001\ldots)_2$

7. $(0.0000110011001100\ldots)_2$

9. $(0.B7)_{16}$

11. $(0.77ED)_{16}$

12. $(0.7777\ldots)_{16}$

14. $(0.428)_{16}$

16. $(0.110011001100\ldots)_{16}$

1.3 Floating Point Form

Floating point numbers are used by computers to approximate real numbers. On the surface, the question is a simple one. There are an infinite number of real numbers, but a computer is a finite machine so it can only represent a finite number of real numbers. That is, not all real numbers can be stored exactly. Therein lies the problem.

If the computer can only store an approximation of a real number, then it is essential that there is a discussion of the error involved.

In this section we will address each of these issues.

Floating Point Numbers

Each of the following numbers is equal to 123.4567 in base ten:

$$12345.67 \cdot 10^{-2}, \qquad 1.234567 \cdot 10^{2}, \qquad \text{and} \qquad 0.01234567 \cdot 10^{4}.$$

- In the first case, multiplying by 10^{-2} moves the decimal point two places to the left, so $12345.67 \cdot 10^{-2} = 123.4567$.
- In the second case, multplying by 10^{2} moves the decimal point two places to the right, so $1.234567 \cdot 10^{2} = 123.4567$.
- In the third case, multiplying by 10^{4} moves the decimal point four places to the right, so $0.01234567 \cdot 10^{4} = 123.4567$.

Computers use a form of scientific notation to store approximations of real numbers in n-digit floating point form.

n-digit Floating Point Form. An n-digit floating point has the form

$$\pm d_1.d_2d_3 \ldots d_n \cdot b^m.$$

Note that the sign occurs first (plus or minus), followed by an n-digit number $d_1.d_2d_3 \ldots d_n$ called the **matissa**. The number b is called the **base** and m is called its **exponent**. Each digit d_i of the mantissa is an integer such that $0 \leq d_i < b$, for $i = 2, 3, \ldots, n$. The first digit must satisfy $0 < d_1 < b$ unless the floating point number is zero.

For example, in base ten, the number $2.3854 \cdot 10^{-13}$ is in 5-digit floating point format, but the numbers $238.54 \cdot 10^{-12}$ and $0.0023854 \cdot 10^{12}$ are not.

- In the first case, $238.54 \cdot 10^{-12}$ has more than one digit to the left of the decimal point. We can place this number in 5-digit floating point form by repositioning the decimal point and adjusting the exponent. That is,

$$238.54 \cdot 10^{-12} = 2.3854 \cdot 10^{-10}.$$

- In the second case, the first digit to the left of the decimal point in the number $0.0023854 \cdot 10^{12}$ is zero, but $0.0023854 \cdot 10^{12}$ is not zero. Again, we can place this number in floating point form by repositioning the decimal point and adjusting the exponent. That is,

$$0.0023854 \cdot 10^{12} = 2.3854 \cdot 10^{9}.$$

In the examples that follow, let's assume that we are working on a base ten machine that stores numbers in 5-digit floating point format.

▶ **Example 1.** *Change the number* 888.341983765 *into 5-digit base ten floating point format.*

First, reposition the decimal point so that there is exactly one nonzero digit to the left of the decimal point.

$$888.341983765 = 8.88341983765 \cdot 10^{2}$$

The machine we are working on can only handle 5-digit floating point form. It can't store all the digits of the mantissa above. Therefore, we must determine the closest 5-digit floating point number available and use that as an approximation for our number. Note that our number lies between the two 5-digit floating point numbers

$$8.8834 \cdot 10^{2} < 8.88341983765 \cdot 10^{2} < 8.8835 \cdot 10^{2},$$

but it is closer to $8.8834 \cdot 10^{2}$. Hence, in 5-digit floating point form,

$$888.341983765 \approx 8.8834 \cdot 10^{2}.$$

Note that we *rounded* towards zero in this example. Because the digit following the 4 in $8.88341983765 \cdot 10^{2}$ is a 1, which is less than 5, we truncate the number at $8.8834 \cdot 10^{2}$.

Let's look at another example.

▶ **Example 2.** *Change the number* 0.00075493671278 *into 5-digit base ten floating point form.*

First, reposition the decimal point so that there is exactly one nonzero digit to the left of the decimal point.

$$0.00075493671278 = 7.5493671278 \cdot 10^{-4}$$

Again, our machine can only handle 5-digit mantissas. Our number lies between the following two 5-digit floating point numbers

$$7.5493 \cdot 10^{-4} < 7.5493671278 \cdot 10^{-4} < 7.5494 \cdot 10^{-4},$$

but it is closer to the number $7.5494 \cdot 10^{-4}$. Hence, in 5-digit floating point form,

$$0.00075493671278 = 7.5494 \cdot 10^{-4}.$$

Note that we *rounded away* from zero in this example. Because the digit following the 3 in $7.5493671278 \cdot 10^{-4}$ is a 6, which is 5 or greater, we add 1 to the previous place before truncating to get $7.5494 \cdot 10^{-4}$.

Let's look at another example.

▶ **Example 3.** *Suppose that the 5-digit base ten floating point representation of a real number x is $x^* = 2.3086 \cdot 10^{-4}$. Find the range of possible values for the real number x.*

In **Examples 1** and **2**, we saw that the computer will sometime rounds towards zero and other times round away from zero, depending on the value of the sixth digit in 5-digit floating point format. In this example, we're given the 5-digit base ten floating point form of the number, namely

$$x^* = 2.3086 \cdot 10^{-4}.$$

- The very smallest that x could be is $x = 2.30855 \cdot 10^{-4}$. Any smaller, such as $x = 2.30854999\ldots \cdot 10^{-4}$, and x would have been rounded towards zero to $x^* = 2.3085 \cdot 10^{-4}$.

- The very largest that x could be is $x = 2.30864999\ldots \cdot 10^{-4}$. Any larger, such as $x = 2.30865 \cdot 10^{-4}$, and x would have been rounded away from zero to $x^* = 2.3087 \cdot 10^{-4}$.

  Therefore, x could be any number in the range

$$2.30855 \cdot 10^{-4} < x < 2.30864999\ldots \cdot 10^{-4}.$$

Binary Floating Point Form

In binary (base two), things work pretty much the same. Note that the number $1.0011 \cdot 2^{-3}$ is in 5-digit base two floating point form, but the numbers $1101.1 \cdot 2^{-4}$ and $0.00011001 \cdot 2^5$ are not.

- In the first case, $1101.1 \cdot 2^{-4}$ has more than one digit to the left of the decimal point. We can place this number in floating point form by repositioning the decimal point and adjusting the exponent.

$$1101.1 \cdot 2^{-4} = 1.1011 \cdot 2^{-1}$$

- In the second case, the first digit to the left of the decimal point in the number $0.00011001 \cdot 2^5$ is not zero, but we can again reposition the decimal point and adjust the exponent.

$$0.00011001 \cdot 2^5 = 1.1001 \cdot 2^1$$

Error

In this section we discuss the error made when storing a real number in n-digit floating point form on a computer.

We will discuss two important types of error: (1) *absolute* error, and (2) *relative* error.

In the discussion that follows, we will let x represent the real number and x^* represent the n-digit floating point approximation of x.

Absolute and Relative Error. Let x^* be the n-digit floating point representation of the real number x. Then the absolute and relative error in approximating x with x^* is given by the formulae

$$\text{Absolute Error} = |x^* - x|$$

and

$$\text{Relative Error} = \frac{|x^* - x|}{|x|}.$$

Let's look at an example.

▶ **Example 4.** *Calculate both the absolute and relative error when the real number $x = 938\,756$ is stored in 3-digit base ten floating point form.*

First, reposition the decimal point so that there is one nonzero digit to the left of the decimal point.

$$x = 938\,756 = 9.38756 \cdot 10^5$$

We can only use 3 digits in the mantissa. The next digit to the right of 8 is a 7, which is greater than 5, so we round up (away from zero) to

$$9.38756 \cdot 10^5 = 9.39 \cdot 10^5.$$

The result $x^* = 9.39 \cdot 10^5$ is in 3-digit base ten floating point form. We calculate the absolute error with the following computation.

$$|x^* - x| = \left|9.39 \cdot 10^5 - 938\,756\right| = 244$$

That seems to be an very large error! But on second glance, note what the relative error reveals.

$$\frac{|x^* - x|}{|x|} = \frac{\left|9.39 \cdot 10^5 - 938\,756\right|}{|938\,756|} \approx 2.6 \cdot 10^{-4}$$

A calculator was used to determine the approximation. Note that the number $x = 9.38756 \cdot 10^5$ and its approximation $x^* = 9.39 \cdot 10^5$ agree in about 3 places and the exponent in the relative error $2.6 \cdot 10^{-4}$ is -4.

We'll see that the relative error is more useful. Let's look at another example.

▶ **Example 5.** *Calculate both the absolute and relative error when the real number 0.000005823417658 is stored in 5-digit base ten floating point form.*

Reposition the decimal point so that there is one nonzero digit to the left of the decimal point.

$$x = 0.000005823417658 = 5.823417658 \cdot 10^{-6}$$

The mantissa is allowed 5 digits. Note that the next digit after the 4 is a 1, which is less than 5, so we round down (towards zero) by truncating.

$$5.823417658 \cdot 10^{-6} = 5.8234 \cdot 10^{-6}$$

The result $x^* = 5.8234 \cdot 10^{-6}$ is in 5-digit base ten floating point form. The absolute error is

$$|x^* - x| = \left|5.8234 \cdot 10^{-6} - 0.000005823417658\right| = 1.7658 \cdot 10^{-11},$$

which at first glance, appears very small indeed. But again, how small is the error relative to the numbers involved? The relative error reveals the answer.

$$\frac{|x^* - x|}{|x|} = \frac{\left|5.8234 \cdot 10^{-6} - 0.000005823417658\right|}{|0.000005823417658|} \approx 3.0 \cdot 10^{-6} \tag{1.4}$$

A calculator was used to find an approximation for the relative error. Note that this error is much larger than the absolute error.

Also, note that $x = 5.823417658 \cdot 10^{-6}$ and $x^* = 5.8234 \cdot 10^{-6}$ agree in approximately 5 digits and the exponent on the relative error $3.0 \cdot 10^{-6}$ is -6.

Indeed, there is a technical definition for the number of *significant digits*.

Significant Digits. The number x^* is said to approximate x to n significant digits if n is the largest nonnegative integer for which

$$\frac{|x^* - x|}{|x|} < 5 \cdot 10^{-n}.$$

Thus, for example, in **Example 4**, we approximated $x = 938\,756$ with $x^* = 9.39 \cdot 10^5$ and found that the relative error was

$$\frac{|x^* - x|}{|x|} \approx 2.6 \cdot 10^{-4},$$

so the relative error is less than $5 \cdot 10^{-4}$. Thus, by the definition, we say that $x^* = 9.39 \cdot 10^5$ approximates $x = 938\,756$ to 4 significant digits. However, note that only the first two leading digits are the same.

In **Example 5**, we approximated $x = 5.823417658 \cdot 10^{-6}$ with $x^* = 5.8234 \cdot 10^{-6}$ and found that the relative error was

$$\frac{|x^* - x|}{|x|} \approx 3.0 \cdot 10^{-6},$$

so the relative error is less than $5 \cdot 10^{-6}$. Thus, by the definition, we say that $x^* = 5.8234 \cdot 10^{-6}$ approximates $x = 5.823417658 \cdot 10^{-6}$ to 6 significant digits. Note, however, that only the first 5 leading digits are the same.

It is important to realize that the notion of significant digits and the the number of digits of agreement between a number and its floating point form are related, but not exactly the same. For example, in 5-digit floating point form, approximating $x = 7.899966666 \cdot 10^3$ with with its 5-digit floating pont form $x^* = 7.9000 \cdot 10^3$ provides a relative error

$$\frac{|x^* - x|}{|x|} \approx 4.2 \cdot 10^{-6},$$

which is less than $5 \cdot 10^{-6}$. Thus, x^* approximates x to 6 significant digits. However, the numbers $x = 7.89996666 \cdot 10^3$ and $x^* = 7.9000 \cdot 10^3$ have only the first leading digit in common. Still, in the sense of the relative error, it's not difficult to imagine the closeness of the digits in $x^* = 7.9000 \cdot 10^{-6}$ to the first 6 digits of $7.89996666 \cdot 10^{-6}$.

n-Digit Floating Point Form and Significant Digits. What is most important to understand is the fact that there is a definite relationship between the the number of digits used to store the mantissa, the relative error, and the number of significant digits.

Propogation of Error

Whenever we store an n-digit floating point form of a real number, we are making an error. This error has a special name.

Roundoff Error. The error incurred when we store a real number in n-digit flooting point form is called **roundoff error**.

With today's modern computers, we can store numbers so that the initial roundoff error is fairly insignificant. The difficulty lies in the fact that computers can literally do billions of computations very quickly, so it is not uncommon to see the original roundoff error propogate through a series of calculations, diverging quickly so as to make the final outcome meaningless.

In the next section we will study some ways to keep this propogation of error under control.

1.3 Exercises

In **Exercises 1-8**, place the given number in 4-digit base ten floating point form. In each case, calculate the absolute and relative error made.

1. 1 885 934

2. 12 345 612

3. 0.0001234567

4. 0.0085188342

5. 888.456123

6. 1 765.33458

7. 0.0002312316

8. 0.00000556781245

In **Exercises 9-12**, a 4-digit base ten floating point approximation x^* of a real number x is given. Determine a range of possible values for x.

9. $2.446 \cdot 10^{-12}$

10. $4.453 \cdot 10^{8}$

11. $5.684 \cdot 10^{5}$

12. $1.104 \cdot 10^{-6}$

In **Exercises 13-16**, Place the given number into n digit floating point format for the given value of n, calculate the relativer error, then use the result to determine the the number of significant digits in the approximation.

13. 1 789.23456, $n = 5$

14. $0.008456174 \cdot 10^{-6}$, $n = 3$

15. $0.0000456712345 \cdot 10^{-11}$, $n = 6$

16. $18.9123456 \cdot 10^{6}$, $n = 4$

1.3 Answers

1. $1.886 \cdot 10^{6}$

3. $1.235 \cdot 10^{-4}$

5. $8.885 \cdot 10^{2}$

7. $2.312 \cdot 10^{-4}$

9. Range from $2.4455 \cdot 10^{-12}$ to $2.4464999\ldots \cdot 10^{-12}$.

11. Range from $5.6835 \cdot 10^{5}$ to $5.6844999\ldots \cdot 10^{5}$.

13. $1.7892 \cdot 10^{3}$, relative error is approximately $1.9 \cdot 10^{-5}$, 5 significant digits.

15. $4.56712 \cdot 10^{-5}$, relative error approximately $7.5 \cdot 10^{-6}$, 5 significant digits.

1.4 Floating Point Arithmetic

Today's modern computers, even home personal computers, can usually store enough digits so that roundoff error is not too much of a problem, at least at the storage level. However, computers can can literally make millions or billions of calculations in a numerical process, so the original roundoff error can propogate throughout the calculation, rendering the final answer meaningless.

In this section we look at floating point arithmetic and discuss some numerical effects that can have disastrous results on caclulations.

Let's begin with an example that demonstrates catastrophic cancellation of digits and loss of precision.

▶ **Example 1.** *Compute* $\sqrt{9876}-\sqrt{9875}$ *using 5-digit floating point arithmetic.*

First, let's examine the relative error in storing $\sqrt{9876}$ in 5-digit floating point form. We're going to use Matlab's *Symbolic Toolbox*, an interface to the computer algebra system (CAS) Maple. Computations in Maple are done symbolically and are exact. Thus, instead of turning a numeric result, an exact expression is returned.

```
  >> sym('sqrt(9876)')
ans =
sqrt(9876)
```

The *Symbolic Toolbox* has a command that will return a numerical approximation of a symbolic object, correct to as many places of accuracy that the user needs. The command **vpa** does the work.

```
>> help vpa
VPA    Variable precision arithmetic.
   R = VPA(S) numerically evaluates each element of the
   double matrix S using variable precision floating point
   arithmetic with D decimal digit accuracy, where D is
   the current setting of DIGITS. The resulting R is a SYM.

   VPA(S,D) uses D digits, instead of the current setting
   of DIGITS. D is an integer or the SYM representation of
   a number.
```

We will use the last paragraph of the help message to approximate $\sqrt{9876}$, correct to 20 digits. Note that the symbolic object must be entered as a string (delimited by single apostrophes (ticks)).

```
>> vpa('sqrt(9876)',20)
ans =
99.378065990438755368
```

Use this result to store $\sqrt{9876}$ in 5-digit floating point format.

$$\sqrt{9876} = 9.9378 \cdot 10^1.$$

We calculate the relative error with

$$\frac{|x^* - x|}{|x|} = \frac{|9.9378 \cdot 10^1 - \sqrt{9876}|}{|\sqrt{9876}|}. \tag{1.5}$$

To find an approximate value of the relative error, we will again turn to the *Symbolic Toolbox*. We'll first store the needed computation as a string in the variable **rel** (use any variable name you like).

```
>> rel='abs(9.9378e1-sqrt(9876))/abs(sqrt(9876))'
rel =
abs(9.9378e1-sqrt(9876))/abs(sqrt(9876))
```

Now we'll use the **vpa** command to approximate to 20 digits of accuracy.

```
>> vpa(rel,20)
ans =
.66403424234193683981e-6
```

Thus, the relative error in storing $x = \sqrt{9876}$ in 5-digit floating point form $x^* = 9.9378 \cdot 10^1$ is approximately $6.6 \cdot 10^{-7}$.

Recall the definition of *significant digits*.

Significant Digits. The number x^* is said to approximate x to n significant digits if n is the largest nonnegative integer for which

$$\frac{|x^* - x|}{|x|} < 5 \cdot 10^{-n}.$$

Because the relative error is less than $5 \cdot 10^{-6}$, we know that $x^* = 9.9378 \cdot 10^1$ approximates $x - \sqrt{9876}$ to 6 significant digits.

In similar fashion, we can use the *Symbolic Toolbox* to approximate $\sqrt{9875}$ and find the relative error in storing this number in 5-digit floating point form. First, an approximation of $\sqrt{9875}$.

```
>> vpa('sqrt(9875)',20)
ans =
99.373034571758952600
```

Thus, the 5-digit floating point storage of $x = \sqrt{9875}$ is $x^* = 9.9373 \cdot 10^1$. The relative error in storing $\sqrt{9875}$ is next.

```
>> rel='abs(9.9373e1-sqrt(9875))/abs(sqrt(9875))'
rel =
abs(9.9373e1-sqrt(9875))/abs(sqrt(9875))
>> vpa(rel,20)
ans =
.34789879469399867106e-6
```

Thus, the relative error is approximately $3.4 \cdot 10^{-7}$, which is about the same as the relative error in storing $\sqrt{9876}$ in 5-digit floating point form. Because the relative error is less than $5 \cdot 10^{-7}$, we know that $x^* = 9.9373 \cdot 10^1$ approximates $x - \sqrt{9875}$ to 7 significant digits.

Now, let's subtract the stored 5-digit floating point form numbers and see what happens.

$$9.9378 \cdot 10^1 - 9.9373 \cdot 10^1 = 0.0005 \cdot 10^1$$

Note that all of the digits of the mantissa are gone save one. Further, note that there are no more digits to the right of the 5 in $0.00005 \cdot 10^1$, so when we adjust the exponent to place this result in 5-digit floating point form, as in

$$5.0000 \cdot 10^{-3},$$

the digits to the right of the decimal point in $5.0000 \cdot 10^{-3}$ are meaningless. Indeed, the computer could quite possibly shove in some digits from memory that are not zeros, just to pad the number.

Now, let's look at the relative error in approximating $x = \sqrt{9876} - \sqrt{9875}$ with this result.

```
>> rel='abs(5.0e-3-(sqrt(9876)-sqrt(9875)))/abs(sqrt(9876)-sqrt(9875))'
rel =
abs(5.0e-3-(sqrt(9876)-sqrt(9875)))/abs(sqrt(9876)-sqrt(9875))
>> vpa(rel,20)
ans =
.62444971890114329880e-2
```

The relative error is approximately $6.2 \cdot 10^{-3}$. Recall that the relative error in approximating $\sqrt{9876}$ and $\sqrt{9875}$ with 5-digit floating point numbers was of the order of 10^{-7}, so roughly speaking, the relative error made by subtracting, which is of the order 10^{-3}, is roughly $10^{-4}/10^{-7} = 10^4$, or 10 000 times larger! Further, because the relative error of the subtraction is less than $5 \cdot 10{-}2$, only 2 significant digits remain!

This phenomenon is called *catastrophic cancelllation* of digits and occurs whenever you attempt to subtract two numbers that are very close to one another. The programmer needs to be aware of this phenomenon and avoid subtracting two nearly equal numbers. In this case, we can change the subtraction into addition by rationalizing the numerator with this calculation.

$$\sqrt{9876} - \sqrt{9875} = \frac{9876 - 9875}{\sqrt{9876} + \sqrt{9875}} = \frac{1}{\sqrt{9876} + \sqrt{9875}}$$

If we add the floating point representations of $\sqrt{9876}$ and $\sqrt{9875}$, we get

$$9.9378 \cdot 10^1 + 9.9373 \cdot 10^1 = 19.8751 \cdot 10^2,$$

which when stored in 5-digit floating point form is $1.9875 \cdot 10^2$. Next, we perform the division,

$$\frac{1.0000 \cdot 10^0}{1.9875 \cdot 10^2} = 0.50314465408805 \cdot 10^{-2},$$

which when stored in 5-digit floating form is $x^* = 5.0314 \cdot 10^{-3}$. The relative error when approximating $x = \sqrt{9876} - \sqrt{9875}$ with $x^* = 5.0314 \cdot 10^{-3}$ is

$$\frac{|x^* - x|}{|x|} = \frac{\left|5.0314 \cdot 10^{-3} - (\sqrt{9876} - \sqrt{9875})\right|}{\left|\sqrt{9876} - \sqrt{9875}\right|}.$$

We can use the *Symbolic Toolbox* to help with this calculation.

```
>> rel='abs(5.0313e-3-(sqrt(9876)-sqrt(9875)))/abs(sqrt(9876)-sqrt(9875))'
rel =
abs(5.0313e-3-(sqrt(9876)-sqrt(9875)))/abs(sqrt(9876)-sqrt(9875))
>> vpa(rel,20)
ans =
.23587741414644558543e-4
```

Thus, the relative error when approximating $x = \sqrt{9876} - \sqrt{9875}$ with $x^* = 5.0314 \cdot 10^{-3}$ is approximately $2.3 \cdot 10^{-5}$, which is less than $5 \cdot 10^{-5}$, so this time we've kept 5 significant digits, which is much better than the 2 significant digits of the previous computation.

Let's look at another example.

▶ **Example 2.** *Solve the quadratic equation* $x^2 - 1634x + 2 = 0$ *using 10-digit floating point arithmetic.*

Using the quadratic formula to solve $x^2 - 1634x + 2 = 0$, we get

$$x = \frac{1634 \pm \sqrt{1634^2 - 4(1)(2)}}{2(1)} = \frac{817 \pm \sqrt{667\,487}}{.}$$

We can use the *Symbolic Toolbox* to approximate $\sqrt{667\,487}$.

```
>> vpa('sqrt(667487)',20)
ans =
816.99877600887505858
```

Thus, in 10-digit floating point form, $\sqrt{667\,487} \approx 8.169987760 \cdot 10^2$. Thus, one solution of the quadratic is

$$x_1^* = 8.170000000 \cdot 10^2 + 8.169987760 \cdot 10^2 = 16.339987760 \cdot 10^2,$$

or $x_1^* = 1.633998776 \cdot 10^3$ in 10-digit floating point form. We can use the *Symbolic Toolbox* to calculate the relative error in approximating $x_1 = 817 + \sqrt{667\,487}$ with this 10-digit floating point number.

```
>> rel='abs(1.633998776e3-(817+\sqrt(667487)))/abs(817+sqrt(667487))'
rel =
abs(1.633998776e3-(817+\sqrt(667487)))/abs(817+sqrt(667487))
>> vpa(rel,20)
ans =
.54314964676275835537e-11
```

Thus, the relative error is approximately $5.4 \cdot 10^{-12}$, which is less than $5 \cdot 10^{-11}$, so we are approximating $817 + \sqrt{667\,487}$ to 11 significant digits.

The second root is

$$x_2^* = 8.170000000 \cdot 10^2 - 8.169987760 \cdot 10^2 = 0.000012240 \cdot 10^2,$$

or $x_2 = 1.224000000 \cdot 10^{-3}$, in 10-digit floating point form. Note the catastrophic cancellation of digits. The last 5 zeros of 1.224000000 are completely meaningless and in some cases the computer will pad these places with nonzero digits that happen to be lying around in memory.

We can use the *Symbolic Toolbox* to calculate the relative error made when approximating $x_2 = 817 - \sqrt{667\,487}$ with the 10-digit floating point number $x_2^* = 1.224000000 \cdot 10^{-3}$.

```
>> rel='abs(1.224000000e-3-(817-\sqrt(667487)))/abs(817-sqrt(667487))'
rel =
abs(1.224000000e-3-(817-\sqrt(667487)))/abs(817-sqrt(667487))
>> vpa(rel,20)
ans =
.72509174283635093702e-5
```

Thus, the relative error is approximately $7.3 \cdot 10^{-6}$, which is less than $5 \cdot 10^{-5}$, so $x_2^* = 1.224000000 \cdot 10^{-3}$ is approximating $x_2 = 817 - \sqrt{667\,487}$ to only 5 significant digits!

Programmers have to be aware of catastrophic cancellation of digits any time to numbers are subtracted that are very nearly equal in value. The programmer has to look for an algorithm that avoids subtraction.

In this case, consider the idea that if x_1 and x_2 are roots of the quadratic equation $x^2 - 1634x + 2 = 0$, then the quadratic quadratic $x^2 - 1634x + 2$ can be factored as $(x - x_1)(x - x_2)$. If we expand and compare to the original quadratic, then

$$\begin{aligned} x^2 - 1634x + 2 &= (x - x_1)(x - x_2) \\ &= x^2 - (x_1 + x_2)x + x_1x_2. \end{aligned}$$

Comparing the constant terms, $x_1x_2 = 2$, or equivalently,

$$x_2 = \frac{1}{x_1}.$$

This last result will allow avoid subtraction in calculation the second root x_2.

$$\begin{aligned} x_2^* &= \frac{2}{x_1^*} \\ &= \frac{2.000000000}{1.633998776} \cdot \frac{10^0}{10^3} \\ &= 1.2239911369332985296 \cdot 10^{-3} \\ &= 1.223991137 \cdot 10^{-3} \end{aligned}$$

We can use Matlab to calculate the relative error in approximating $x_2 = 817 - \sqrt{667\,487}$ with the 10-digit floating point number $x_2* = 1.223991137 \cdot 10^{-3}$.

```
>> rel='abs(1.223991137e-3-(817-\sqrt(667487)))/abs(817-sqrt(667487))'
rel =
abs(1.223991137e-3-(817-\sqrt(667487)))/abs(817-sqrt(667487))
>> vpa(rel,20)
ans =
.98518524802025190487e-8
```

Thus, the relative error is approximately $9.9 \cdot 10^{-9}$, which is less than $5 \cdot 10^{-8}$, so this time we've managed to hang on to 8 significant digits.

1.4 Exercises

In **Exercises 1-4**, perform each of the following tasks for the given number.

i. Use Matlab's **vpa** command to approximate the given number to 20 digits.
ii. Place the result of the **vpa** command into n-digit floating point format for the given value of n.
iii. Use the **vpa** command to determine the relative error.
iv. Use the definition of significant digits to determine the number of significant digits in your n-digit floating point approximation.

1. $\sqrt{14\,385}$, $n = 5$

2. $\sqrt{23\,888}$, $n = 5$

3. $ln888\,475$, $n = 10$

4. $\ln 1\,234\,555$, $n = 10$

In **Exercises 5-8**, perform each of the following tasks for the given expression.

i. Use the **vpa** command of the *Symbolic Toolbox* to assist in finding n-digit floating point approximations of each root for the given value of n. Calculate the relative error for each and state the number of significant digits for each.
ii. Use the results of the previous part to hand calculate an n-digit floating point approximation for the given expression.
iii. Use Matlab's **vpa** command to find the relative error in approximating the given expression with the n-digit floating point number of part (ii).
iv. Use the definition of significant digits to determine the number of significant digits in the approximation of part (ii).

5. $\sqrt{8355} - \sqrt{8354}$, $n = 5$

6. $\sqrt{7565} - \sqrt{7564}$, $n = 5$

7. $\sqrt{45619} - \sqrt{45617}$, $n = 10$

8. $\sqrt{387159} - \sqrt{387156}$, $n = 10$

In **Exercises 9-12**, perform each of the following tasks for the given quadratic equation.

i. Solve the given quadratic by hand. Place your solution in simple radical form.
ii. Use Matlab's **vpa** command to approximate the radical in your solution to 20 digits, then place the result in 10-digit floating point format.
iii. Use hand calculations to determine the solution that doesn't cause catastrophic cancellation of digits in 10-digit floating point form. Use the **vpa** command to determine the relative error and the number of significant digits of this result.
iv. Follow the lead of **Example 2** in the narrative to obtain a 10-digit floating point approximation of the second solution. Then use the **vpa** command to determine the rela-

tive error and the number of significant digits.

9. $x^2 - 4744x + 2 = 0$

10. $x^2 - 5666x + 4 = 0$

11. $x^2 - 388x + 2 = 0$

12. $x^2 - 644x + 1 = 0$