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

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

This can be rewritten as follows.

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

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 $(10000000000000000)_2$. That is,

$$(1111111111111111)_2 = (10000000000000000) - 1 = 2^{16} - 1 = 65535.$$

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.

$$\begin{array}{c|c|c|c|c|c|c|c|c} & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}0 & {}^{1}1 & {}^{1}1 & 1 \\ + & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ \hline & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

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.

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

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**.

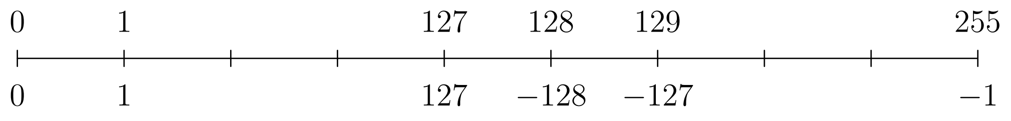

Figure 1.1. Unsigned integers on top are mapped to signed integers on the bottom.

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

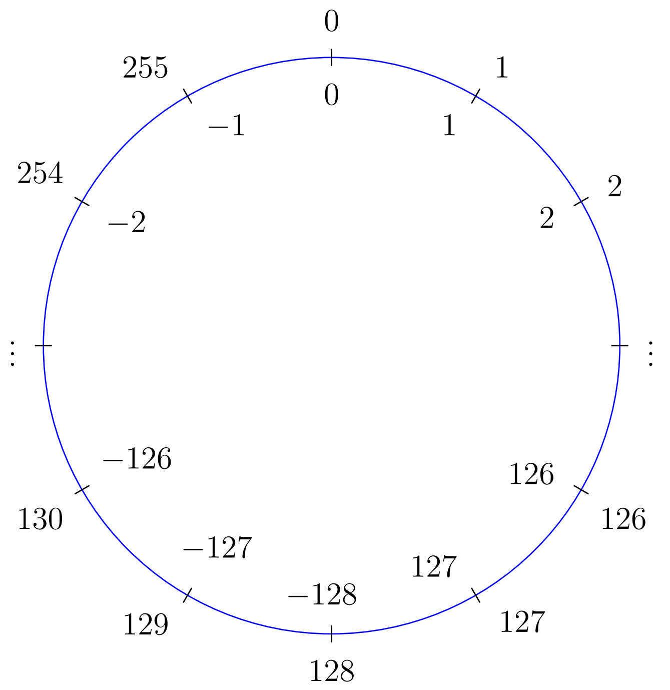

Figure 1.2. Unsigned integers on the outer rim are mapped to signed integers on the inner rim.

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 $[0, 18446744073709551615]$.