1 Representing Numbers in the Computer

When we enter a number in a file or in a spreadsheet, we don't have to give any serious thought to how it will be represented in the computer and that its meaning is clear when we map it from our application to the computer. We are fortunate that this has been worked out for us and it is something we can quickly gloss over and assume will be done correctly. Since we are studying statistical computing and computational statistics, we ought to at least understand when things can possibly go wrong and, like all good software, handle the special cases that might lead to erroneous results. In other words, we should understand the limits of the computer and when these limits are likely to be encountered.

The main lessons to take out of this section are the following:

- Representing real numbers in a computer always involves an approximation and a potential loss of significant digits.
- Testing for the equality of two real numbers is not a realistic way to think when dealing with the numbers in a computer. Two equal values may not have the same representation in the computer because of the approximate representation and so we must ask is the difference less than the discernible amount as measured on the particular computer.
- Performing arithmetic on very small or very large numbers can lead to errors that are not possible in abstract mathematics. We can get underflow and overflow, and the order in which we do arithmetic can be important. This is something to be aware of when writing low-level software to to do computations.
- The more bits we use to represent a number, the greater the precision of the representation and the more memory we consume. Also, if the natural representation of the machine does not use that number of bits and representation, arithmetic, etc. will be slower as it must be externally specified, i.e. off the chip.

In order to understand these points properly, we need to first look at how numbers are represented on a computer. We will start with the basic type, an integer. Once we understand integer representation and its limitations, we can move to the representation of real numbers. To understand integer representation, we review how information is stored in a computer.

1.1 Bits and bytes

Computers are digital devices. The smallest unit of information has two states: memory cells in RAM are either charged or cleared, magnetic storage devices have areas that are either magnetized or not, and optical storage devices use two different levels of light reflectance. These two states can be represented by the binary digits 1 and 0.

The term "bit" is an abbreviation for binary digit. Bits are the building blocks for all information processing in computers. Interestingly, it was a statistician from Bell Labs, John Tukey, who coined the term bit in 1946. While the machine is made up of bits, we typically work at the level of "bytes", (computer) words, and higher level entities. A byte is a unit of information built from bits; one byte equals 8 bits. Bytes are combined into groups of 1 to 8 bytes called words. The term byte was first used in 1956 by computer scientist Werner Buchholz. He described a byte as a group of bits used to encode a character. The eight-bit byte was created that year and was soon adopted by the computer industry as a standard.

Computers are often classified by the number of bits they can process at one time, as well as by the number of bits used to represent addresses in their main memory (RAM). The number of bits used by a computer's CPU for addressing information represents one measure of a computer's speed and power. Computers today often use 32 or 64 bits in groups of 4 and 8 bytes, respectively, in their addressing. A 32-bit processor, for example, has 32-bit registers, 32-bit data busses, and 32-bit address busses. This means that a 32-bit address bus can potentially access up to $2^{32} = 4,294,967,296$ memory cells (or bytes) in RAM. Memory and data capacity are commonly measured in kilobytes ($2^{10} = 1024$ bytes), megabytes ($2^{20} = 1,048,576$ bytes), or gigabytes ($2^{30} = 1,073,741,824$, about 1 billion bytes). So, a 32-bit address bus, means that the CPU has the potential to access about 4 GB of RAM.

2 Integers

When we work with integers or whole numbers, we know that there are infinitely many of them. However, the computer is a finite state machine and can only hold a finite amount of information, albeit a very, very large amount. If we wanted to represent an integer, we could do this by having the computer store each of the digits in the number along with whether the integer is positive or negative.

Now, when we want add two integers, we would have to use an algorithm that performed the standard decimal addition by traversing the digits from right to left, adding them and carrying the extra term to the higher values. Representing each digit would require us to encode 0, 1, ... 9 in some way which we can do in terms of 4 bits (There are $2^4 = 16$ distinct combinations of four 1s and 0s, eg. 0000 maps to 0, 0001 maps to 1, 0010 maps to 2, etc). Many operations in a typical computer typically try to work at the level of words, i.e. bringing a word of memory onto a register or from disk to RAM. With two-byte words, we would be wasting 12 bits on each of the digits since we are only using 4 bits in the word. And each time we add two integers, we have to do complicated arithmetic on an arbitrary number of digits.

Clever people who designed and developed computers realized that it is significantly more efficient if we take advantage of the on-off nature of bits, and represent numbers using base 2 rather than decimal (base 10). And if we use a fixed number of binary digits, then we can do the computations not in software but on the chip itself and be much, much faster.

2.1 Binary Representation

The main system of mathematical notation today is the decimal system, which is a base-10 system. The decimal system uses 10 digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and numbers are represented by the placement of these digits. For example, the numbers 200, 70, and 8 represent 2 hundreds, 7 tens, and 8 units, respectively. Together, they make the number, 278. Another way to express this representation is with exponents:

$$278 = (2 \times 10^2) + (7 \times 10^1) + (8 \times 10^0). \tag{1}$$

The binary system, which is the base-2 system, uses two digits, 0 and 1, and numbers are represented through exponents of 2. For example,

$$\begin{aligned} 1101_2 &= (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \\ &= (8) + (4) + (0) + (1) = 13 \end{aligned}$$

Note that the subscript 2 on the number 1101 lets us know that the number is being represented in base-2 rather than base-10. This representation shows how to convert numbers from base-2 into base-10. A picture might also be helpful in providing a different perspective on this.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
0	0	1	1	0	0	0	1	
0	0	32	16	0	0	0	1	49

Figure 1: Think of putting 1s and 0s in the different buckets here. To represent a number, we put a 1 in the bucket when we want to include that term. The number is then the sum of the values for the buckets with a 1 in them.

This technique works for fractions as well. For example,

$$\begin{aligned} 1.101_2 &= (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) \\ &= (1) + (1/2) + (0) + (1/8) = 1.625 \end{aligned}$$

To convert a decimal number into base-2, we find the highest power of two that does not exceed the given number, and place a 1 in the corresponding position in the binary number. Then we repeat the process with the remainder until the remainder is 0. For example, consider the number 614. Since $2^9 = 512$ is the largest power of 2 that does not exceed 614, we place a 1 in the tenth position of the binary number, 1000000000. Continue with the remainder, namely 102. We see that the highest power of 2 is now 6, as $2^6 = 64$. Next we place a 1 in the seventh position: 1001000000. Continuing in this fashion we see that

$$614 = 512 + 64 + 32 + 4 + 2 = 1001100110_2. \tag{2}$$

Converting fractions in base 10 to base 2 proceeds in a similar fashion. Take the fraction, 0.1640625. Using the same method we see that since $2^{-3} = 0.125$ and $2^{-2} = 0.25$, we begin by placing a 1 in the third position to the right of the decimal place: 0.001. The remainder is now $0.1640625 - 0.125 = 0.0390625$, which holds the power $2^{-5} = 0.03125$ with a remainder of 0.0078125. This remainder is 2^{-7}, and we have 0.0010101_2 represents the decimal fraction 0.1640625. In some cases, the binary representation of a fraction may not have a finite number of digits. Take the simple fraction, 0.1. Using the same method we see that since $2^{-4} = 0.0625$, we begin by placing a 1 in the fourth position to the right of the decimal place. We see that $2^{-4} + 2^{-5} = 0.00625$. Continuing in this way we find that we have a repeating decimal representation: $0.1 = 0.00011001100110011...$, where 0011 repeats forever. To convert a decimal number with a fractional part into binary, we separately convert to binary the integer part and fractional part.

hexadecimal	1	A (10)	D (13)
binary	0001	1010	1101

Figure 2: To convert a hexadecimal number to binary, simple go digit by digit converting each digit into its four-bit binary representation.

2.2 Hexadecimal representation

Hexadecimal notation is also used when working with computers. The representation uses base 16 rather than base 2. The logic is the same. We need 16 ‘digits’, so we use 0, 1, 2, ..., 9, A, B, C, D, E, F. The letters A, B, C, D, E, and F represent the values 10, 11, 12, 13, 14, and 15 respectively. They are also called, ”Able”, ”Baker”, ”Charlie”, ”Dog”, ”Easy”, and ”Fox”. To represent the value 19, we observe that $19 = 16 + 3$, so we use 13_{16} because $19 = 1 \times 16^1 + 3 \times 16^0$. That is, 13 in base 16 is equivalent to 19 in base 10.

As another example, to represent a decimal value such as 429 in hexadecimal, we use the simple iterative technique described for converting to binary. Since $16^2 = 256$ and $16^3 = 4096$, we start with 256, and notice that 256 divides 429 once, with a remainder of 173. This means that we place a 1 in the third position of the hexadecimal number. Now, 173 is divisible by the the next lower power, namely 16, 10 times with 13 as a remainder. So the next term in our hexadecimal representation is A and it is placed in the second position. Finally, the remainder 13 is the digit D in base 16. Our number in base 16 is $1AD_{16}$. We now see why we need to use A, B, C, D, E, and F as digits because otherwise we would not be able to keep the places straight.

Binary and hexadecimal representations go hand in hand because $2^4 = 16$. This means that we can think of each hexadecimal digit as four bits. From this viewpoint, we can easily convert hexadecimal represenations to binary and vice versa. Figure2.2 gives an example of how to do this conversion.

We see that numbers have equivalent representations in base 2, base 10, or any other base. It is easy to switch between the different bases and there is no loss of information.

3 Approximate integer representations

Restricting the number of (binary) digits we use to store a number makes the quantity of different possible numbers that we can represent in a computer finite. So we must now go from our symbolic, mathematical world with an infinite number of integers to a very finite one with only a fixed number of possible integers. This restriction means that we have to be careful when we encounter an integer that doesn't fit into this fixed collection of possible integers that we can represent in the computer.

With this quite major change in the way we need to think about doing basic arithmetic on integers, we formalize how to represent an integer on a computer. In many computers, integers are represented using 4 bytes, or 32 bits. In the 32-bit binary representation, each integer appears as,

$$\sum_{i=1}^{32} x_i \times 2^{i-1}, \tag{3}$$

where each x_i is either 1 or 0 to indicate whether we include that term in the sum or not.

That is a good way to represent an integer in a computer because we are exploiting the on-off, or binary, nature of the machine and its gates. We see that the smallest integer is 0, when all of the $x_i = 0$ and the largest integer is $2^{32} - 1 = 4294967295$. But of course, as we have written this, we only have non-negative integers. One simple thing we can do to include negative integers is to subtract from this value 2^{31}. Now we have values between $-2^{31} = -2147483648$ and $2^{32} - 1 - 2^{31} = 2^{31} - 1 = 2147483647$. You might want to verify that every integer between these values has a unique representation, i.e. a set of coefficients $\{x_i : 1 \leq i \leq 32\}$ in this scheme. This is sometimes called a *biased* representation or an *offset* representation.

Notice that to add two numbers, we can add the shifted numbers. That is, the binary integers x and y, are represented as $2^{31} + x$ and $2^{31} + y$, respectively. When these two representations are added, we get $2^{32} + x + y$ due to overflow. Which, when shifted by 2^{31} is the representation, $2^{31} + (x + y)$, that we want.

Another popular representation is the *two's complement*. In 32-bit binary, the two's complement representation of a negative integer is the binary number that when added to its positive counterpart produces 2^{32}, i.e. a 1 followed by 32 zeroes. That is, since the 32-bit binary representation of $+1$ has 31 leading zeroes followed by 1, i.e. $0 \cdots 01$, we would represent -1 by 32 ones, i.e. $1 \cdots 11$. The sum of these two binary numbers will be 2^{32}. We will see in section 3.1 that because we only have 32 binary digits, the leftmost digit is dropped, and the number *overflows* to 0. A fast way to find the twos complement of a negative integer is to take the binary representation of the positive integer, flip all of the bits (0 to 1, 1 to 0), and add 1. One of the nice properties of two's complement is that addition and subtraction is made very simple. That is, the circuitry on the microprocessor for addition and subtraction can be unified, and treated as the same operation. Can you figure out how to easily convert a two's complement representation to an offset representation?

Two other representations are the *sign-magnitude*, which uses the leftmost bit to denote the sign, and the *one's complement*, which flips the bits to represent a negative number. In both of these representation there are two ways to represent 0. What are they? Can you use binary addition to add numbers in these representations?

3.1 Operations and Errors

The computer takes care of arithmetic operations on integers for us. We can think of addition, subtraction, multiplication and division as being essentially built-in to the computer. When we add integers, there is a "carry" step with which we are familiar. Now consider what happens when we add a value to a very large positive integer such that it exceeds the range of the integer representation, 2147483647. Without loss of generality (because of the carry operation), let's just look at adding 1 to the maximum value 2147483647. The maximum value is represented by 32 ones, and adding a one to it gives 1 followed by 32 zeroes. This is a 33-bit representation! When we try to map the result into 32-bits, what happens? In our representation, all the coefficients are 0 except the one in the position for 2^{32} which is not in our representation. This means trouble in terms of representing the result as an integer. If the leading digit is dropped then we are left with 32 zeroes, which represents -2147483648.

Even when we promote the two integers being added to real values and do the computations with only the error associated with that class of numbers in the computer, and this is basically 0, we run into trouble when we map the result into a 32-bit representation. In *R*, we promote the result from an integer to a real value and we get no loss of information. If we try bring it to an integer, we get a warning and the result is a undefined value, NA. In sum, when working with these types of numbers, we have to be careful. *R*and other packages will take care of many details. You should be aware of them as not all systems will do this. Try this in Excel, Matlab. If the system only provides the resolution of number or text or whatever (e.g. date), it is not an issue but we have lost information as we no longer validate that the values are integer-valued.

4 Floating point representation

Let's think about how we might represent a real number in the computer. We can represent the number 2.3 in scientific notation as 2.3×10^0, as 0.23×10^1, as 23×10^{-1}, or as 0.023^2, etc. There is not a unique representation, unless we settle on a convention that the mantissa is less than 10 and its first digit is non-zero. In our example, that means we settle on 2.3×10^0. This unique representation of a real number has four components: the sign (+1) in this example, the mantissa (2.3), the base (10), and the exponent (0).

When we work in base 2, how do these components look? Consider the number 110.0101_2. Following the above convention, it would be uniquely represented in scientific notation as

$$(+1)\times(1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+0\times 2^{-3}+1\times 2^{-4}+0\times 2^{-5}+1\times 2^{-6})\times 2^2 \quad (4)$$

We see that the sign is $+1$, the mantissa is 1.100101, the base is 2, and the exponent is 2. For consistency, when we convert all four components to base 2, we would have

$$(+1_2) \times (1.100101_2) \times 10_2^{10_2}. \quad (5)$$

That is, the sign is $+1$, the mantissa is 1.100101, the base is 10 and the exponent is 10, all in base 2! We typically use the former representation in our mathematical notation, as it is easier for us to think in terms of 2^2 than $10_2^{10_2}$.

We can represent any real number in base-2 floating point.

$$(-1)^{x_0}\{1 + \sum_{i=1}^{t} x_i 2^{-i}\} \times 2^k, \quad (6)$$

where x_0 is the sign bit, the $\{x_i\}$ represent the mantissa, and k the exponent. This representation is ideal for storing real numbers in the computer.

5 Approximate floating point representations

We saw with integers that we could not store arbitrarily large (positive or negative) integers in the computer. Similarly, with floating point numbers, we can't store a number to arbitrary precision in the computer because the computer is a finite-state machine. Instead, we approximate the number and then store that in the computer. The IEEE single-precision floating point representation uses 32 bits, with one bit for the sign (1 for negative and 0 for positive), 23 bits for the mantissa, and eight bits for the exponent. Since the leading digit of the mantissa will always be 1 in our base-two representation, we need not store it. The leading 1 is a hidden digit. This gives us an extra digit of precision in our mantissa. (We will see later that for very small numbers, we may need to make the first digit 0.) We also don't need to store the base, as it is always 2. Then most numbers are represented as

$$(-1)^{x_0}\{1+\sum_{i=1}^{23} x_i 2^{-i}\} \times 2^k, \qquad (7)$$

where the mantissa contains 23 terms or digits in the 32-bit representation.

How large a number can the exponent get? Well, it can have $2^8 = 256$ possible values. Since we need to represent both large and small values, we need both negative and positive exponents. In order to do this, we can use an integer representation for the exponent, and apply a shift or bias, as described in Section 2.1. Typically, the exponent ranges from $-2^7 + 2 = -126$ to $2^7 - 1 = 127$, where we usually reserve two values, -128 and -127, as flags for special values. We hold one for representing very, very small numbers that we cannot represent properly. And we hold the other to indicate that the number really doesn't make sense, i.e. is the result of computing a value such as $1/0$, or to indicate a special case such as underflow, overflow or "not a number" (NaN).

In double-precision floating point, we typically use 11 bits for the exponent and 52 bits for the mantissa. So the exponent has $2^{11} = 2048$ possible values. For 11 bits, the shift is typically 1024, and the possible (integer) values stored in the exponent are -1022 to 1023. Again, two values for the exponent are reserved for special cases.

6 Machine Constants: Smallest Values

The restrictions on the size of the mantissa and exponent translate into restrictions on the accuracy of the representation of the number in the computer.

For example, consider the single precision representation of the decimal value 0.1. Earlier, in Section 2.1, we saw that we need an infinite number of digits to represent 0.1 in base 2, $0.000110011001100110011001100110011\ldots$. We only have 23 digits available to us (24 with the first digit being hidden), which means that this number is represented as $0.000110011001100110011001100$ which has a decimal value of $0.0999999940395522460 94$, and so is accurate only to seven decimal places. We call the difference between the true value and its approximation the rounding error. The relative size of the error is about $0.000000006/0.1 = 6 \times 10^{-8}$.

What happens when we add the values 100,000 and 0.1? Well, the binary representation of 100,000 is 11000011010100000. And the sum in 32-bit floating point representation is,

$$11000011010100000.0001100$$

(notice there are 24 digits here) which has a decimal value of 100000.09375. The size of the error has grown, but the relative error $0.006/100000.1$ is still of the same order, as the error in representing 0.1.

To help us make sense of the size of these approximation errors, we might ask, what is the smallest number that can be added to 1 to get get a result different than 1? The 32-bit floating point representation of 1 is,

$$(-1)^{x_0}\{1 + \sum_{i=1}^{23} x_i 2^{-i}\} \times 2^0, \tag{8}$$

where, $x_0 = 0$, and $x_i = 0$, $i = 1, \ldots, 23$. We see that if we add 2^{-23} we get

$$1 + 2^{-23} = (-1)^0\{1 + (\sum_{i=1}^{22} 0/2^i) + 1/2^{23}\} \times 2^0, \tag{9}$$

which is different from 1. Any number smaller than 2^{-23} will result in no change to the sum. This machine constant, is often called the floating point precision, and is denoted by ϵ_m.

6.1 Relative error

The constant ϵ_m provides an upper bound on the relative size of the error in approximating a real number x by its finite floating point representation, which we call $f(x)$. In our example with 0.1 we found that $|x - f(x)| = 6 \times 10^{-9}$ and the relative error in the approximation is $|x - f(x)|/|x| = 6 \times 10^{-8}$, which is less than $2^{-23} = 1.2 \times 10^{-7}$. Note that for our single precision case,

$$\begin{aligned} \frac{|x - f(x)|}{|x|} &\leq 2^{-23} 2^k / |x| \\ &\leq 2^{-23} 2^k 2^{-k} \\ &= 2^{-23}. \end{aligned}$$

which establishes that the relative error is at most ϵ_m.

The IEEE standard for floating point arithmetic calls for basic arithmetic operations to be performed to higher precision and then rounded to the nearest representable floating point number.

6.2 Error accumulation

How do errors change over arithmetic operations? Consider the simple case of addition of two numbers. For any real number x, the error in a single-precision floating-point approximation can be as large as 2^{k-23}, where k is the exponent in the representation of x. (Note that $x \geq 2^k$ since by uniqueness of the representation, the first digit in the mantissa must be a 1 for large numbers).

When two nearly equal numbers are subtracted from each other then the resulting relative error can be very large. To see this, take 0.100002288818358375 and subtract 0.1 from it. This first number was chosen because it has a finite binary representation with a mantissa of fewer than 24 digits, 0.000110011001100111. In other words, in single precision, this real number is exactly represented by a mantissa of $1.100110011001110 \cdots 0$ and an exponent of $k = -4$. We have seen already that 0.1 is approximated by a mantissa of $1.10011001100110011001100$ and an exponent of -4. (Don't forget about the hidden digit!) The error in this approximation is about 2^{-28}, and the relative error is 2^{-27}. When this representation of 0.1 is subtracted from the other number, the difference is,

$$0.00000000000000000100110100$$

, which is about 0.0000022947. The true difference is about 0.0000022888. The error is about 6×10^{-9}, and the relative error is about 3×10^{-3} or about 2^{-8}, which is quite large in comparison to the original relative errors of 0 and 2^{-27}, respectively.

This type of cancellation error doesn't occur when we add numbers of the same sign. (Subtraction is simply adding numbers of different signs). For example, take two positive numbers, x and y, with single precision floating point representation $f(x)$ and $f(y)$, and denote the relative error by $\delta_x = (f(x) - x)/x$ and δ_y. We have shown already that the relative error $|\delta_x|$ is bounded above by $2^{-23} = \epsilon_m$. Then

$$\begin{aligned} f(x) + f(y) &= x + y + x\delta_x + y\delta_y \\ &\leq (x + y)(1 + \epsilon_m) \end{aligned}$$

When we add $f(x)$ and $f(y)$, the result must also be represented in single precision, i.e. $f(f(x)+f(y))$, and we are interested in the approximation error relative to the true sum $x+y$. To find this, note that

$$\begin{aligned}
f(f(x)+f(y))-(x+y) &= f(f(x)+f(y))-(f(x)+f(y)) \\
&\quad + f(x)-x+f(y)-y \\
&\leq f(f(x)+f(y))-(f(x)+f(y))+x\epsilon_m+y\epsilon_m \\
&\leq (f(x)+f(y))\epsilon_m+x\epsilon_m+y\epsilon_m \\
&\leq (x+x\epsilon_m+y+y\epsilon_m)\epsilon_m+x\epsilon_m+y\epsilon_m \\
&\leq (x+y)(2\epsilon_m+\epsilon_m^2)
\end{aligned}$$

This argument generalizes to sums of n terms, where it is seen that the errors may accumulate to $n\epsilon_m$ (ignoring terms of smaller order). These bounds are just that, bounds. Consider what might happen when you add one very large number with many small numbers. The order in which these numbers are summed may effect the result. That is, adding the small numbers one at a time to the large number could yield a very different result than when we add all of the small numbers first and then add the sum to the large number. Why?

7 Higher-level Languages

In reality, we don't have to worry about all of these things everyday. The high-level systems we use for working with data insulate us from the representations of different types of numbers and, with varying degrees of utility, from underflow and overflow errors. And this is where we want to be - using commands that allow us to think in terms of the data and the statistical problem on which we are working, and leaving the mapping of this to the low-level, basic computer facilities. And this is the general premise motivating programming languages such as Fortran, C, Java, and much higher-level languages such as S (*R*and S-Plus), Matlab, SAS, and other applications such as Excel.

"One" step above assembler are the C and Fortran. Fortran provides a higher-level mechanism to express computations involving formulae and the language translates these formulae to machine instructions programming languages. This is where the name comes from: formula translator. The Fortran language has evolved from Fortran 66 to Fortran 2000 now with increasingly modern programming facilities. Because of the simplicity of the language, it is quite fast and some argue that it is the most efficient language for scientific computing. This requires much greater specificity about what efficiency means and also how to measure performance appropriately. It is useful, but it is clumsy and is less well suited to good software engineering than most other common languages. It is very useful to be able to read and interact with Fortran code because so much of the existing numerical software was developed in Fortran.

The C language has been a much richer language than Fortran and is used to implement operating systems such as Unix, Linux, and others. It has been the language of choice for many, many years and is used widely in numerical computing. More recently C++ has become fashionable, and offers many benefits over its close ancestor, C. However, these benefits are not without complexity.

Both Fortran and C are languages that are compiled. In other words, programmers author code and then pass it to another application - the compiler - that turns this into very low-level machine code instructions. The result is an executable that can be run to execute the instructions. Such executables are invariably machine-dependent, containing machine-level operations that are specific to that machine targeted by the compiler. The same source code can recompiled on other machines, but this compilation step is necessary. And in many cases, one must either modify the code for the new machine or be conscious of the portability constraints when authoring the code.

Java and C# are very similar languages that one can categorize as being simplified variations of C++ that promote the object-oriented style of programming but without the complexity of C++. They are also compiled languages, requiring a separate compilation step. Unlike C and Fortran compilers, the Java and C# compilers transforms the programmer's code into higher-level instructions targeted at what is termed a Virtual Machine (VM). The VM acts much like a computer, processing the instructions, providing registers, etc. and generally mimicking the actions of the low-level computer. This is done by creating a version of the virtual machine for each computer architecture (e.g. Powerbook, Intel *86, Sparc), but then allowing all code for that virtual machine to run independently of what machine it is running on. The virtual machine is a layer between the compiled code and the low-level computer. And this makes the compiled code (byte code) independent of the particular architecture and so makes it portable without the need for recompiling.

Compiled languages, i.e. those requiring a compiler, are very useful. Typically, the compilation step performs optimization procedures on the programmers code, making it faster by looking at the collection of computations and removing redundancies, etc. Additionally, because compiled languages typically require variable declarations in terms of scope and type information, they can often catch programming errors before the code is run. There are disadvantages, however. Using compiled languages, one typically has to write an entire program before compiling it. This means that

- exploration and rapid prototyping of individual chunks is more complex or impossible. Specifically, it is much more difficult to incrementally create software by running individual commands.
- one cannot typically alter the programming as it is running to correct errors or add functionality.

(See also http://www.math.grin.edu/ stone/courses/fundamentals/IEEE-reals.html and http://icommons.harvard.edu/ hsph-bio248-01/Lecture_Notes/).