2 Representation of real numbers

In order to simulate systems, we must be able to store and manipulate real numbers; however, most real numbers have non-terminated and non-repeating fractional component, and even calculations with integral values will quickly result in infinite repeating decimal number; for example,

$$\frac{355}{113} = 3.14159292035398230088495575221238938053097345132743362831858407079646017699115044247787610619469026548672566371 68\ldots.$$

We cannot use rational numbers as approximations, as these quickly explode, as well:

$$\frac{355}{113} + \frac{723}{991} = \frac{433504}{111983}$$
$$\frac{433504}{111983} + \frac{502}{997} = \frac{488418954}{111647051}$$
$$\frac{488418954}{111647051} + \frac{302}{977} = \frac{510902727460}{109079168827}$$

Each calculation approximately doubles the number of digits that must be stored, and thus on a computer, would approximately double the number of bits required.

Instead, we should consider the scientific notation that was introduced in secondary school science classes:

The electric charge e is approximately $1.6021766208 \times 10^{-19}$ C.

This number stores an eleven-decimal-digit approximation of the electric charge e. The components of this number include a *significand*[1] of 1.6021766208 and an exponent of –19. The significand is a real number on the interval [1, 10) and if the significand contains n digits, we say that the representation contains n digits of precision. For example, the above approximation of e contains 11 significant digits.

To store such a number on a computer, however, requires us to use a fixed number of digits. We will look at two examples:

1. a human-readable six-decimal-digit floating-point representation, and
2. the double-precision floating-point format.

However, prior to the first, we will review scientific notation and the benefits thereof, and prior to the second, we will review binary numbers and binary arithmetic.

[1] This is often called the *mantissa*; however, as this number contains the most significant digits of the number, significand is more appropriate.

2.1 Fixed-precision representations of real numbers

Suppose you are required to store any number using only six decimal digits:

NNNNNN

As an integer, this could store numbers between 0 and 999999, or one million different integer values.

Another possibility would be to insert a decimal place; for example, we could insert a fixed decimal in the representation; for example, NNNNNN represents the real number *nnn.nnn*. In this case, the smallest non-zero and largest numbers we can represent are:

1. `000001` which equals 0.001, and
2. `999999` which equals 999.999.

The range of non-zero numbers stored in this manner is 0.001 up 999.999, which generally isn't very large. Such a representation is referred to as a *fixed-point representation* of real numbers.

Such fixed-point representations do have a few applications; for example, they are often used in embedded systems, where integer data types are used to approximate real numbers (integer calculations are faster, and some microcontrollers may not even have a floating-point unit (FPU). For example, *rate-monotonic scheduling* can schedule n periodic tasks, each with possibly different periods p_k and requiring a_k units of time per period, if $\prod_{k=1}^{n}\left(1+\frac{a_k}{p_k}\right)\le 2$. In this case, it is easier to store the utilization $u_k=\frac{a_k}{p_k}$ as a binary integer *bbbbbb*$\cdots$*b* where this represents the utilization 0. *bbbbbb*…*b* and to then perform calculations with these integers. For example, if we were using `short` (16 bits) to store these integers, and a task required 137 ms each minute, the utilization would be 0.0000000010010101010001111111001111111000$\cdots$ which could be stored (rounding up) as the 16-bit number `0000000010010110` which represents 0.0000000010010110.

There are two serious drawbacks with a fixed-point representation:

1. The range is not significant; in this example, there are only six orders of magnitude between the smallest and largest non-zero numbers that can be represented.
2. The real number 0.0015 must be represented as either 0.001 or 0.002 (`000001` or `000002`), depending on how you chose to round the value. In either case, the relative error of the approximation is $\frac{0.0005}{0.0015}\cdot 100\%\approx 33\%$, so there are real numbers in the range [0.001, 999.999] that can only be represented with a very large relative error. Similarly, the value 0.0005 could be represented either by `000000` or `000001`. Either representation has a relative error of $\frac{0.0005}{0.0005}\cdot 100\%=100\%$. On the other hand, 999.9985 could be represented by either `999998` or `999999`, and in each case, the relative error is $\frac{0.0005}{999.9985}\cdot 100\%\approx 0.00005\%$.

Thus, there is small range, and the maximum relative error depends inversely on the magnitude of the number being stored.

2.2 Floating-point representations

Let's consider a different representation where we will harken back to scientific notation:

$$\pm n.nnnnn\cdots n\times 10^{\pm eee\cdots e}$$

where whether the number is positive or negative is referred to as the *sign*; the digits $n.nnnnn\cdots n$ are referred to as the *significand*, and $\pm eee\cdots e$ is referred to as the *exponent*. Any such representation where the magnitude is affected by a base raised to a power is referred to as a *floating-point* representation, as the decimal point is allowed to "float" relative to the significand. We will look at a six-decimal-digit floating-point representation, followed by rounding, and then looking at issues with floating-point representaitons.

2.2.1 A six-decimal-digit floating-point representation

Restricting ourselves to just six decimal digits, we could have `±EENNNN` represent the number $\pm n.nnn\times 10^{ee}$, although this would only allow us to represent positive powers of 10 so we would be restricted to storing numbers from 0.001 to 9.999 × 10^{99}, so we will introduce a *bias* in the exponent so `±EENNNN` represent the number $\pm n.nnn\times 10^{ee-49}$. The choice of 49 is arbitrary, but gives us a useful range. Thus, we can now represent positive numbers from 0.001 × 10^{-49} to 9.999 × 10^{50}, or over 100 orders of magnitude. Additionally, every single number on the range [0.001 × 10^{-49}, 9.999 × 10^{50}] can be written with a relative error no larger than 0.0005 or 0.05 %.

We will introduce a few special numbers:

Normally, the most significant digit must be non-zero. This is done to ensure that the representation is unique. But what happens if the exponent is already the smallest possible:

`+00MNNN`

Here, the exponent is 10^{-49}, and you cannot get a smaller exponent. Thus, we could let `M` equal zero in this case:

Representation	Value	Comment
`+001000`	1.000 × 10^{-49} = 10^{-49}	Smallest *normalized* positive number
`+000100`	0.100 × 10^{-49} = 10^{-50}	
`+000010`	0.010 × 10^{-49} = 10^{-51}	
`+000001`	0.001 × 10^{-49} = 10^{-52}	Smallest positive number

The last number represents all numbers on the interval (0.5 × 10^{-52}, 1.5 × 10^{-52}). Because we allow the leading digit to be zero, we call all these numbers *denormalized*.

Next, `+000000` appears to be "0", but in reality it represents all real numbers on the interval [0, 0.5 × 10^{-52}] and now `-000000` represents all real numbers on the interval [–0.5 × 10^{-52}, 0]. Having a *signed zero* is useful in many cases, as $1/10^{100}$, while small, is never-the-less positive, and $-1/10^{100}$ is small and negative. The logarithm of a small positive number is large and negative, but the logarithm of a small negative number is undefined.

Thus, we have our next special number: like now where we use 00 to identify denormalized numbers, we will use the exponent 99 to represent others:

Representation	Value	Comment
`+989999`	$9.999 \times 10^{49} \approx 10^{50}$	Largest real number
`+990000`	∞	Positive infinity
`-990000`	$-\infty$	Negative infinity
`+991000`	NAN	*Not-a-Number*

Because 9.9995×10^{49} rounds up to 1.000×10^{50}, the floating-point infinity represents not just infinity, but all real numbers on the semi-infinite interval $[9.9995 \times 10^{49}, \infty)$, while negative infinity represents all numbers on the interval $(-\infty, -9.9995 \times 10^{49}]$. Thus, a calculation like `+491000 ÷ +000000` will equal `+990000` or infinity, while `+491000 ÷ -000000` will equal `-990000` or negative infinity. Similarly, `log(+000000)` will equal `+990000`.

You can also get positive infinity as a result of an arithmetic computation: `+989999 + +945000`, as this results in 9.9995×10^{49}, which rounds up and not down.

The last number is used when the result of a computation is either not a real number (the logarithm of a negative number), so `log(-000000)` will equal `+991000`, indicating the result is not a number. It is also possible to get a result that is not a number if you have:

a. Either zero divided by either zero.
b. Either zero multiplied by either infinity.
c. Either infinity divided by either zero.
d. Either infinity divided by either infinity.
e. Positive infinity plus negative infinity.
f. Any arithmetic operation involving NAN.

For example,

1. 1.0005 is represented by either 1.000 or 1.001, each having a relative error of approximately 0.00049975 or 0.049975 %; and
2. 9.9995 is represented by either 9.999 or 10.00, each of which has a relative error of approximately 0.0000500025 or 0.00500025 %.

Now, there is one issue with our representation: all four representations `+491000` ($1.000\times10^{49-49}$), `+500100` ($0.100\times10^{50-49}$), `+510010` ($0.010\times10^{51-49}$) and `+520001` ($0.001\times10^{52-49}$) represent the value $1.000\times10^{0}=1$; the representation is not necessarily unique. We can ensure the representation is unique if we require that the leading digit in the significand is non-zero: `±EEMNNN` where `M ≠ 0`.

Thus, simply by looking at two such representations, `123456` and `598235`, we see that the numbers they represent must be different because the digits are different. It is not necessary to convert the first to 3.456×10^{-37} and the second to 8.235×10^{10} to see that they are different.

There is even one greater benefit: we can also compare the relative magnitudes simply by comparing the relative magnitudes of the integer representations: the number represented by `+123456` must be less than the number represented by `+598235`, because 123456 < 598235. Again, it is not necessary to first determine which numbers these represent.

Problems

1. Represent the following real numbers in our six-decimal-digit representation:

a. 3.957 (`+493957`)
b. –19310 (`-531931`)
c. 22420000 (`+562242`)
d. –8001000000000 (`-618001`)
e. 0.004275 (`+464275`)
f. –0.0000000008426 (`-398426`)

2. What numbers are represented by the following?

a. `+495840` (5.840×10^{0})
b. `-514180` (-4.180×10^{2})
c. `+477450` (7.450×10^{-2})
d. `-189265` (-9.265×10^{-31})
e. `+981023` (1.023×10^{49})
f. `-292946` (-2.946×10^{-31})

3. Up to this point, what are the smallest and largest positive numbers that can be stored exactly in this representation, and what are those representations?

(`+001000` equal to 1.000×10^{-49} and `+999999` equal to 9.999×10^{-50})

2.2.2 The best representation and rounding

When a real number is written in scientific notation, such as

$$n_1.n_2n_3n_4n_5n_6\cdots\times10^e$$

with $n_1 \neq 0$, we will describe the decimal digit n_k as the k^{th} *significant digit*. The digit n_1 is also sometimes referred to as the *most significant digit*. For example, the first 500 significant digits of π are

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194 91

and the 2nd significant digit is '1' as is the 500th significant digit. When using or storing numbers, we are often forced to store a fixed number of digits. The number

$$n_1.n_2n_3n_4n_5n_6\cdots n_m\times10^e$$

is said to hold *m* significant digits of precision. Thus, the numbers

$$3.92\times10^5,\ 5.928\times10^{16}\ \text{and}\ 4.982017346\times10^{-12}$$

hold 3, 4 and 10 significant digits of precision (the number of significant digits being independent of the exponent).

Suppose we want to store π to *m* significant digits. In this case, the goal is to find that number holding *m* significant digits that has the smallest relative error as an approximation to π; for example, approximating π to 10 significant digits gives us one of two choices 3.141592653 or 3.141592654. The relative error of the first is 1.88×10^{-10} while the relative error of the second is 1.31×10^{-10}; thus the second approximation is better.

Similarly, for example 5.3729 in our four-digit decimal representation, we could use either `495372` (5.372) or `495373` (5.373). In this case, however, the percent relative error of these two is 0.017 % and 0.0019 %, respectively, and therefore clearly `495373` is the optimal representation.

The easiest way to find the best representation is to simply round to *n* significant figures:

1. If the $(n+1)^{\text{st}}$ digit is 0, 1, 2, 3 or 4, truncate all digits beyond the n^{th} digit (rounding down).
2. If either the $(n+1)^{\text{st}}$ digit is 6, 7, 8 or 9 or the fifth digit is 5 and subsequent digits are not all 0, truncate all digits beyond the n^{th} digit and add one to the n^{th} digit, which may cause a carry that must then be dealt with (rounding up).
3. Otherwise, the only other situation is that the $(n+1)^{\text{st}}$ digit is 5 and all subsequent digits are 0; in which case, both possible representations have equal relative error. In this case, we examine the n^{th} digit and:
    a. If the n^{th} digit is even, we leave it; otherwise
    b. the n^{th} digit is odd, so increment it, which may cause a carry that must then be dealt with.

This last rule is described as *rounding towards even*; meaning, if there is an equal choice, we choose that representation that has the n^{th} digit being even. This 3rd rule is to prevent a bias being introduced into our calculations: if every time we either only rounded down or only rounded down, then over time, after many calculations, our final result will be biased in the direction we round. By occasionally rounding up and occasionally rounding down, this bias is averaged out over a large number of computations.

As for some examples:

1. The value 39571.8860534 rounded to 3 digits is 3.96×10^4.
2. The value 19.31398 rounded to 3 digits is 1.93×10^1.
3. The value 22425170.465 rounded to 4 digits is 2.243×10^7.
4. The value 80018.74496869 rounded to 7 digits is 8.001874×10^4.
5. The value 0.04275001072 rounded to 3 digits is 4.28×10^{-2}.
6. The value 842.62265 rounded to 7 digits is 8.426226×10^2.
7. The value 0.0415 rounded to 2 significant digits is 4.2×10^{-2}.
8. The value 99.9995 rounded to 5 significant digits is 1.0000×10^3

Problems

1. Round the following numbers to six significant digits:

a. 1532.475 (1.53248×10^3)
b. 72300.45 (7.23004×10^4)
c. 350099.5 (3.50100×10^5)
d. 4289225 (4.28922×10^6)
e. 99999950 (1.00000×10^8)

2.2.3 Issues with floating-point representations

There are some serious issues with floating-point representations; for example, we see that if we add `499738` and `453456`, we are adding 9.738×10^0 and 3.456×10^{-4}. Now, if you were to add these on paper, the result would be 9.7383456×10^0; however, if we were to store this in our representation, we can only store four digits of the significand, so the result is 9.738×10^0 or `499738`. That is, we now have the situation that

$$x + y = x$$

even though $y \neq 0$. Another property of real numbers that we learned in secondary school is that it doesn't matter what order we add a series of numbers: $x + y + z = (x + y) + z = x + (y + z)$. However, if you try to add the following three numbers: `499738`, `-499737` and `453456`. If we add the first two together, we get `461000` and adding that to the third number yields `461346` as we would round the mantissa 1.3456 to 1.346. If, however, we were to add the second and third numbers first, like above, the result would be `-499737`, so if you add to this `499738`, the result would now be `461000`. Now, the correct answer is 1.3456×10^{-3}, so both 1.346×10^{-3} and 1.000×10^{-3} are approximations, but the relative error of the first is 0.0297 %, the relative error of the second is 25.7 %.

Consequently, $(x + y) + z \neq x + (y + z)$ and thus we must be careful with how we add numbers.

Problems

1. For which of these calculations does $x + y = x$? Remember, you should compute the full answer first to all significant digits and then round the result to four significant figures.

a. `-522344 + +491802` (`-522342`, so yes)
b. `-069248 + +753241` (`+753241`, so no)
c. `+565388 + -525000` (`+765388`, so yes)
d. `+545000 + -582935` (`-582934`, so no)
e. `+513534 + +475000` (`+413534`, so yes)

2.2.4 Arithmetic with our 6-digit decimal floating-point representation

When we add two numbers, there is an intelligent approach and a sub-optimal approach. Suppose we are adding

`573532 + 591840`

The wrong approach would be to add determine the first is 3.532×10^8 and the second is 1.840×10^{10}, and to then add these two:

$$\begin{array}{r} 353200000 \\ +\underline{18400000000} \\ 18753200000 \end{array}$$

and to then store this as `591853` (we must round to four significant digits). Instead, we could instead just consider the difference in the exponents and only offset the smaller number:

$$\begin{array}{r} 0.03532\times10^{59-49} \\ +\underline{1.84000\times10^{59-49}} \\ 1.87532\times10^{59-49} \end{array}$$

Thus, the result is `591875`. If there is an additional carry, it isn't that difficult:

`288532 + 299746`

may be calculated as

$$\begin{array}{r} 0.8532\times10^{29-49} \\ +\underline{9.7460\times10^{29-49}} \\ 10.5992\times10^{29-49} \end{array}$$

and so the exponent must be incremented by one to get that this is stored as `301060`.

With multiplication, it is necessary to determine the relative magnitudes of the exponents, but the calculation of the exponent is independent of the product of the significands. For example, in multiplying

`473532 × 531840`

we have the significands to multiply:

$$\begin{array}{r} 3.532 \\ \times\underline{1.840} \\ 6.498880 \end{array}$$

and then the exponent of the result is –2 + 4 = 2, so this would be stored as `516499`. It may be necessary to have to increment the exponent if the product of the significands produces a result greater of 10 or greater:

`508197 × 566423`

which has us calculate

$$\begin{array}{r} 8.197 \\ \times\ \underline{6.423} \\ 52.649331 \end{array}$$

and looking at the exponents, 1 + 7 + 1 = 9, so the result is `585265`.

Problems

1. Calculate each of the following, and see if you get the same answer.

a. `+504527` + `+502953` = `+507480`
b. `+526922` + `+522199` = `+529121`
c. `+568002` + `+569295` = `+571730`
d. `-493285` + `-509825` = `-511015`
e. `+493572` × `+492671` = `+499541`
f. `+494271` × `+498234` = `+503517`
g. `+494271` × `-498234` = `-503517`
h. `-463524` × `-515342` = `+491883`
i. `-507883` × `+458413` = `-476632`
j. `+210992` × `+138390` = `+000000`
k. `-707883` × `+858413` = no representation (why?)

2. Suppose you have `513153` and `519354`. For each of these, find the smallest floating-point number *x* such that *x* plus this number leaves *x* unchanged. Solutions: `551000` and `561000`, respectively.

3. Suppose we have `513153` and `519354`. For each of these, find the largest floating-point number *x* such that *x* plus these numbers leaves these numbers unchanged. Solution: `474999` and `475000`, respectively.

2.3 A review of the binary representation

When you see the number 1024, you understand this to be

$$1024 = 1\cdot 10^3 + 0\cdot 10^2 + 2\cdot 10^1 + 4\cdot 10^0 .$$

The only reason we choose 10 is likely due to the number of digits on our hands; there really is no real benefit otherwise. Because 10 is the base of each of the powers, we call this representation *base 10*. Note, however, that every number has a unique representation as a sum of powers of 2, 3, 4, 5, 6, etc.:

$$\begin{aligned}
1000 &= 1\cdot 2^9 + 1\cdot 2^8 + 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^3 \\
1000 &= 1\cdot 3^6 + 1\cdot 3^5 + 1\cdot 3^3 + 1\cdot 3^0 \\
1000 &= 3\cdot 4^4 + 3\cdot 4^3 + 2\cdot 4^2 + 2\cdot 4^1 \\
1000 &= 1\cdot 5^4 + 3\cdot 5^3 \\
1000 &= 4\cdot 6^3 + 4\cdot 6^2 + 3\cdot 6^1 + 4\cdot 6^0 \\
1000 &= 2\cdot 7^3 + 6\cdot 7^2 + 2\cdot 7^1 + 6\cdot 7^0
\end{aligned}$$

You can see this, from the geometric series, so taking $\sum_{k=0}^{n-1} b^k = \frac{b^n - 1}{b-1}$ and rearranging this, we get

$$b^n = (b-1)\sum_{k=0}^{n-1} b^k + 1 .$$

Thus, for example, $6\cdot 7^3 + 6\cdot 7^2 + 6\cdot 7^1 + 6\cdot 7^0 + 1 = 7^4$. If you are working in base b, then you require b different digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 works well for base 10 and 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, χ, ε works well for base 12. In a computer, however, digits must be stored as voltages, and having 10 or 12 different voltage levels is actually a very difficult, complicated and expensive design. Instead, it is easier to have two voltage levels: normally chosen to be 0 V and 5 V. This would allow for two digits, so base 2 is the appropriate choice.

We will now

1. describe the binary representation of both integers and real numbers, and
2. describe arithmetic operations with binary numbers.

2.3.1 Binary representations

In base 2, the digits are represented usually as 0 and 1. Like base 10, when you run out of digits for one power of 2, you increment the next power of 2; thus, the representation of the integers 0 through 17 are:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111	10000	10001

To convert a number in base 2 to base 10, you simply multiply out the powers:

$$\begin{aligned} 10010101 &= 1\times 2^7 + 1\times 2^4 + 1\times 2^2 + 1\times 2^0 \\ &= 128 + 16 + 4 + 1 \\ &= 149 \end{aligned}$$

Now, when it isn't clear, to differentiate between 10 being "2" in binary or "ten" in decimal, it is customary to put the base as a subscript, so 101_2 is 5_{10} and 101_{10} is 1100101_2.

Problems

Determine the values of the following numbers in binary:

a. $111_2 = 7_{10}$
b. $10001_2 = 17_{10}$
c. $10000_2 = 16_{10}$
d. $11011_2 = 27_{10}$
e. $100110_2 = 38_{10}$
f. $101010_2 = 42_{10}$
g. $101100_2 = 44_{10}$
h. $1011100101_2 = 741_{10}$
i. $1011111011_2 = 763_{10}$
j. $10100010010_2 = 1298_{10}$

2.3.2 The best representation and rounding

When a real number is written in scientific notation using binary, such as

$$b_1.b_2b_3b_4b_5b_6\cdots\times 2^e$$

with $b_1 \neq 0$, we will describe the decimal binary digit (*bit*) b_k as the k^{th} *significant bit*. The bit b_1 is also sometimes referred to as the *most significant bit*. For example, the first 500 significant bits of π are

11.00100100001111110110101010001000100001011010001100001000110100110001001100011001100010100010111000000011011100000111001101000100101001000000100100111000001000100010100110011111001100011101000000001000001011101111101010011000111011000100100111001101100100010010100001010010100000100001111001100011100011010000000100110111011110111110010101000110011011001111100110101001110100100001100011011001100000010101100001010011011011111001001011111000101000011011101001111111000010011010101101101101101101010100011100

and the 2nd significant bit is '1' as is the 498th significant bit. When using or storing numbers, we are often forced to store a fixed number of bits. The number

$$b_1.b_2b_3b_4b_5b_6\cdots b_m\times 2^e$$

is said to hold *m* significant bits of precision. Thus, the numbers

$$1.10\times 2^5,\ 11.01\times 2^{16} \text{ and } 1.001011001\times 2^{-12}$$

hold 3, 4 and 10 significant bits of precision, respectively.

Suppose we want to store π to *m* significant bits. In this case, the goal is to find that number holding *m* significant bits that has the smallest relative error as an approximation to π; for example, approximating π to 10 significant bits gives us one of two choices 11.00100100 or 11.00100101. The relative error of the first is 3.08×10^{-4} while the relative error of the second is 9.35×10^{-4}; thus the first approximation is better.

The easiest way to find the best representation is to simply round to *n* significant bits:

1. If the $(n + 1)^{\text{st}}$ bit is 0, truncate all digits beyond the n^{th} bit (rounding down).
2. If the $(n + 1)^{\text{st}}$ bit is 1and at least one subsequent bit is not all 0,
   truncate all bits beyond the n^{th} bit and add one to the n^{th} bit, which may cause a carry that must then be dealt with (rounding up).
3. Otherwise, the only other situation is that the $(n + 1)^{\text{st}}$ bit is 1 and all subsequent bit are 0; in which case, both possible representations have equal relative error. In this case, we examine the n^{th} digit and:
   a. If the n^{th} bit is 0, we leave it;
   b. otherwise, the n^{th} bit is 1 so we add 1 which will cause a carry that must then be dealt with.

This last rule is described as *rounding towards even*; meaning, if there is an equal choice, we choose that representation that has the n^{th} bit being 0. Like with decimal digits, this 3rd rule is to prevent a bias being introduced into our calculations: if every time we either only rounded down or only rounded down, then over time, after many calculations, our final result will be biased in the direction we round. By occasionally rounding up and occasionally rounding down, this bias is averaged out over a large number of computations.

As for some examples:

a. The value 10101.100001 rounded to 3 bits is 1.01×2^4.
b. The value 11.00111 rounded to 3 bits is 1.10×2^1.
c. The value 10111001.101 rounded to 4 bits is 1.100×2^7.
d. The value 10010.000111101 rounded to 7 bits is 1.001000×2^4.
e. The value 0.011010000101 rounded to 3 bits is 1.11×2^{-2}.
f. The value 101.00111 rounded to 7 bits is 1.010011×2^2.
g. The value 0.0101 rounded to 2 significant bits is 1.1×2^{-2}.

Problems

1. Round the following numbers to six significant digits:

a. 1001.111 (1.01000×2^3)
b. 11011.01 (1.10110×2^4)
c. 111110.1 (1.11110×2^5)
d. 1011011 (1.01110×2^6)
e. 11111110 (1.00000×2^8)

2.3.3 Arithmetic with binary numbers

When adding two numbers, it is like addition in base 10:

$$\begin{array}{rr} & {\scriptstyle 1\,1\,1\,1\ \ 1\qquad 1\,1} \\ & 10001101010011 \\ + & \quad 111101001001 \\ \hline & 11001010011100 \end{array}$$

As an exercise, add the following binary numbers and see if you get the given result:

1. 1000010110 + 110011111 = 1110110101
2. 111010001 + 111001011 = 1110011100
3. 1101100101 + 110111010 = 10100011111
4. 1101001000 + 10110100 = 1111111100
5. 111000010 + 100001001 = 1011001011
6. 10111 + 101110000010 = 101110011001
7. 111001001001 + 101110111011 = 1101000000100
8. 10001011011010 + 11101 = 10001011110111

Like decimal numbers, we can also have a radix point (not a decimal point) to indicate the start of negative powers of 2, so

$$57.496 = 5\cdot 10^{1} + 7\cdot 10^{0} + 4\cdot 10^{-1} + 9\cdot 10^{-2} + 6\cdot 10^{-3}.$$

A binary number with a radix point has a similar interpretation:

$$\begin{aligned} 11.1011 &= 1\cdot 2^{1} + 1\cdot 2^{0} + 1\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 1\cdot 2^{-4} \\ &= 2 + 1 + 0.5 + 0.125 + 0.0625 \\ &= 3.6875 \end{aligned}$$

When adding two binary numbers with radix points, we do the same as we do with decimal numbers: we line up the points and proceed to add. Thus, if we are adding 10001101.010011 + 111101.001, we line up the radix points and add the corresponding columns:

$$\begin{array}{rr} & {\scriptstyle 1\,1\,1\,1\ \ 1} \\ & 10001101.010011 \\ + & \quad 111101.001000 \\ \hline & 11001010.011011 \end{array}$$

Thus, we have that 141.296875 + 61.125 = 202.421875.

As an exercise, add the following binary numbers and see if you get the given result:

1. 100001.011 + 11001.1111 = 111011.0101
2. 111.010001 + 111.001011 = 1110.0111
3. 11011001.01 + 1101110.1 = 101000111.11
4. 110.1001000 + 1.01101 = 111.11111
5. 11100.001 + 10000.1001 = 101100.1011
6. 0.010111 + 101110.00001 = 101110.011001

7. 11100.1001001 + 10111.0111011 = 110100.00001
8. 100010.11011010 + 0.00011101 = 100010.11110111

Binary multiplication is similar to decimal multiplication: the multiplicand is multiplied by each digit of the multiplier, and these products are added:

81476	101011
× 235	× 101
407380	101011
2444280	0000000
+ 16295200	+ 10101100
19146860	11010111

When summing, you must remember that 1 + 1 + 1 = 11, 1 + 1 + 1 + 1 = 100, etc. Here is a slightly more difficult multiplication; however, while working through this on your own, you should realize that this is something that should be quite easy to implement in hardware:

```
  10010011011
× 11010011
  10010011011
 100100110110
1001001101100000
1001001101100000
+1001001101100000000
 111100101111000001
```

Recall that when multiplying real numbers, you counted the number of decimal places in both the multiplicand and the multiplier, and you added that many decimal places in the product:

```
   7.924
× 185.6
  4.7544
 39.6200
633.9200
+ 792.4000
1470.6944
```

The same holds for multiplying real numbers represented in binary:

```
  1.101
× 100.1
  .1101
+ 110.1000
 111.0101
```

and we see that $1.625 \times 4.5 = 7.3125$ and the binary representation of 7.3125 is 111.0101_2.

Problems

1. Calculate each of the following, and see if you get the same answer.

a. $11 \times 10 = 110$
b. $101 \times 11 = 1111$
c. $100 \times 10 = 1000$
d. $101 \times 111 = 100011$
e. $1000 \times 111 = 111000$
f. $1111 \times 1000 = 1111000$
g. $111100 \times 101 = 100101100$
h. $10101 \times 10111 = 111100011$
i. $11101 \times 11010 = 1011110010$
j. $110110 \times 11001 = 10101000110$
k. $101.110 \times 10000.00 = 1011100.00000$
l. $10.010 \times 1011011.1 = 11001101.1110$
m. $1.11100 \times 1.011011 = 10.10101010100$
n. $1.0010000 \times 1.11011 = 10.000100110000$

As an exercise, multiply the following binary numbers and see if you get the given result:

1. $10 \times 1.1 = 11$
2. $1.01 \times 1.1 = 1.111$
3. $110 \times 1.11 = 1010.1$
4. $10.1 \times 1.01 = 11.001$
5. $0.111 \times 1.11 = 1.10001$
6. $11.11 \times 0.01011 = 1.0100101$
7. $0.110 \times 0.011101 = 0.01010111$
8. $1.1 \times 111.11 = 1011.101$
9. $101.1 \times 10.1101 = 1111.01111$
10. $101100 \times 1.0111 = 111111.01$
11. $1.010010 \times 1110.111 = 10011.00001111$
12. $1.1 \times 0.110111 = 1.0100101$
13. $0.0011110011 \times 0.00100101 = 0.000010001100011111$
14. $1100.0011 \times 11000110 = 100101101101.001$

2.4 The double-precision floating-point representation

In the computer, we do not use decimal digits (although some of the first computers did). Instead, the computer uses binary. The double-precision floating-point representation is analogous to the above 6-decimal-digit floating-point representation, only it uses 64 bits instead of six decimal digits. Those bits represent the following:

1. The first bit represents the sign: `0` for positive numbers, and `1` for negative numbers.
2. The next 11 bits represents the exponent, and the bias is `01111111111` which equals 1023. Eleven bits can represent values between 0 and 2047, so by default, the exponent that is stored could be as large as 2047 – 1023 = 1024, and as small as 0 – 1023 = –1023. In reality, the exponent 0 and the exponent 2047 are reserved for other purposes, so the possibly multipliers go from $2^{-1022} = 2.225 \times 10^{-308}$ up to $2^{1023} = 8.988 \times 10^{308}$. There are two exponents that are considered special: all zeros and all ones, or `00000000000` and `11111111111`. If the exponent is either of these, do not calculate with them as if they were normal exponents, instead, see the following sections on zero, infinity and not-a-number.
3. The remaining 52 bits store the significand. Recall that we required the leading digit in the significand to be non-zero; however, the only non-zero binary number is 1, so we won't even store this leading one: we will assume it is there in our representation.

Consequently,

`0`10000000000`1001001000011111101101010100010001000010110100011000`

represents the number

$$1.1001001000011111101101010100010001000010110100011000 \times 2^{1024-1023}$$

and this equals 1.5707963267948965 × 2 = 3.141592653589793, or the best possible approximation of π using the double-precision floating-point format.

The number 1, 2, 3, 0.5, and 0.625 are represented as

$$1.0000000000000000000000000000000000000000000000000000 \times 2^{1023-1023}$$
$$1.0000000000000000000000000000000000000000000000000000 \times 2^{1024-1023}$$
$$1.1000000000000000000000000000000000000000000000000000 \times 2^{1024-1023}$$
$$1.0000000000000000000000000000000000000000000000000000 \times 2^{1022-1023}$$
$$1.0100000000000000000000000000000000000000000000000000 \times 2^{1022-1023}$$

so these would be stored as

```
0011111111110000000000000000000000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000000
0100000000001000000000000000000000000000000000000000000000000000
0011111111100000000000000000000000000000000000000000000000000000
0011111111100100000000000000000000000000000000000000000000000000
```

Now, how would you store 42.3125?

Very often we will write floating-point numbers in hexadecimal, where we group the bits into groups of four, and then substitute each grouping of four bits with the corresponding hexadecimal number.

1.0	3ff0000000000000	–1.0	bff0000000000000
2.0	4000000000000000	–2.0	c000000000000000
3.0	4008000000000000	–3.0	c008000000000000
0.5	3fe0000000000000	–0.5	bfe0000000000000
0.625	3fe4000000000000	–0.625	bfe4000000000000

You may wish to either memorize, or be able to quickly write out this table:

```
0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
```

One issue this author runs into is remembering that while ‘0’, ‘2’, ‘4’, ‘6’, ‘8’ are even, it is ‘a’, ‘c’ and ‘e’ that are even as hexadecimal numbers even if they are the 1st, 3rd, and 5th letters of the alphabet. This author uses the phrase “ace even” as an *aide-mémoire* to recall quickly that these are even digits.

2.4.1 Zero

There are two exponents that are not used to represent special floating-point numbers. The first is when all the bits of the exponent are zero. If all the bits in the significand are zero, too, then this represents +0:

0	0000000000000000

If the sign bit is 1, this represents a –0:

–0	8000000000000000

In elementary school, you learned that –0 = 0, so why are there two different zeros for floating-point numbers? The justification is that a floating-point zero does not represent the mathematical zero, but rather, it represents all real numbers that are too small to be represented by any other floating-point number. Thus, +0 represents all real

numbers on the range [0, ε) where ε is the smallest real number representable by a non-zero floating-point number, and –0 represents all numbers on the range ($-\varepsilon$, 0].

Not required for this course, but if any of the bits of the significand are non-zero, this represents *denormalized numbers*. You will recall that previously we required the most significant bit of the significand is "1"; so

`0000000000001bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb`

represents the number

$$1.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb \times 2^{-1022}$$

as 1 – 1023 = –1022.

2.4.2 Denormalized numbers

Recall that the smallest non-zero positive number is

`0010000000000000` $1.0000000000000000000000000000000000000000000000000000 \times 2^{1-1023} = 2^{-1022}$

If the exponent is zero, like before, this represents a denormalized number with a leading 0, but with the same exponent as the smallest normalized number:

`000xxxxxxxxxxxx` $0.bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb \times 2^{-1022}$

For example,

`00000306a00510e2` $0.0000\ 0000\ 0011\ 0000\ 0110\ 1010\ 0000\ 0000\ 0101\ 0001\ 0000\ 1110\ 0010 \times 2^{-1022}$

$= 1.10000011010100000000001010001000011100010 \times 2^{-1033}$

Thus, the smallest non-zero and positive double-precision floating point number is

`0000000000000001` $0.0000000000000000000000000000000000000000000000000001 \times 2^{-1022}$

$= 1.10000011010100000000001010001000011100010 \times 2^{-1074}$

where –1022 – 52 = –1074.

Therefore, this smallest number represents all real numbers on the range (0.5×2^{-1074}, 1.5×2^{-1074}), and therefore the positive floating-point zero `0000000000000000` represents all real numbers on the range [0, 0.5×2^{-1074}].

2.4.3 Infinity and not-a-number

When you perform integer division in C or C++ and you perform a division-by-zero exception, this will by default terminate the program. You can try this with

```
#include <iostream>

int main();

int main() {
    int a{0};
    int b{0};
    int c{a/b};

    return 0;
}
```

On Linux, the run-time error message when executing this program is:

```
Floating point exception (core dumped)
```

For floating-point numbers, it is much more likely that divisions by zero may occur, and thus the representation itself contains both positive and negative infinity:

```
 ∞    0111111111110000000000000000000000000000000000000000000000000000  7ff0000000000000
–∞    1111111111110000000000000000000000000000000000000000000000000000  fff0000000000000
```

Now, if you were to calculate 1/0, you would get ∞, while 1/∞ equals zero. Similarly, 1/–0 = –∞ and vice versa. The reason for a –0 is that 0 does not represent just 0, but rather, it represents every real number that cannot be represented as any other double-precision floating-point number. Thus, it represents an interval around zero. +0 represents small numbers that are positive, while –0 represents small numbers that are negative.

A second possibility is that a calculation is *indeterminant*. For example, what is 0/0, ∞ – ∞ or ∞/∞? These are represented by a special floating-point number

```
nan   0111111111111000000000000000000000000000000000000000000000000000  7ff8000000000000
```

There are special properties of nan: nan is not equal to nan, but it is also not not-equal no nan. Almost all arithmetic operations with nan result in a nan.

Reminder: If the sign and exponent bits are `000`, `800`, `7ff` or `fff`, the `double` must not be interpreted in the usual way. Instead, it is either a zero, a denormalized number, infinity, or not-a-number.

2.4.4 float versus double

There is a second floating-point representation, namely `float`, which comes from the *single*-precision floating-point representation. It only uses four bytes, of which 8 bits are for the exponent and 23 are for the significand:

`seeeeeeeebbbbbbbbbbbbbbbbbbbbbbb`

This represents the number

$$1.bbbbbbbbbbbbbbbbbbbbbbb \times 10^{eeeeeeee - 01111111}$$

The bias is 127, so using the same design of the `double`, this allows us to represent numbers as large as 1.7×10^{38} with the smallest denormalized float being 2^{-150}, which is approximately 7.0×10^{-46}. Because the exponent crosses a boundary of a hexadecimal character, it isn't even possible to easily read off the exponent and the significand from the hexadecimal representation. The precision of this format is so poor that it should never be used for any scientific or engineering computations. It is best used for graphics, where even gross errors are often overlooked by the observing human eye.

You do not have to understand the representation of `float` for this course. This section is for informational purposes only.

2.4.5 Arithmetic with double-precision floating-point numbers

When performing arithmetic with binary numbers, we will consider addition and multiplication; however, the operations are very similar to that of our 6-digit decimal representation:

When adding two floating-point numbers, we will assume that neither number nor the result is either infinity or not-a-number, as these may be dealt with as special cases. Suppose, for example, we are adding these two doubles:

```
424921c000000000
4245e10000000000
```

In binary, these are

```
0100001001001001001000011100000000000000000000000000000000000000
0100001001001010111100001000000000000000000000000000000000000000
```

Both exponents are the same, so we need only consider the addition of the significands:

$$\begin{array}{r} 1.10010010000111 \\ +\,\underline{1.01011110000100} \\ 10.10110000001011 \end{array}$$

Thus, the result must have one higher exponent, and is stored as

```
0100001001010101100000010110000000000000000000000000000000000000
```

Note that we incremented the exponent by 1. In hexadecimal, this would be

```
4255816000000000
```

and 425 is one greater in as a hexadecimal number than 424.

Note that `3ff00…0` equals 1, so `3ff00…0` × `3ff00…0` = `3ff00…0`.

Suppose, for example, we are multiplying these two doubles:

```
3fd9000000000000
4045000000000000
```

In binary, these are

```
0011111111011001000000000000000000000000000000000000000000000000
0100000001000101000000000000000000000000000000000000000000000000
```

We determine that the exponent of the first is –2, while the exponent of the second is 5:

$$\begin{aligned} & 1.1001\times 2^{-2} \\ \times\ & \underline{1.0101}\times 2^{5} \\ & 10.00001101\times 2^{-2+5} \end{aligned}$$

and $10.00001101\times 2^{3} = 1.000001101000\times 2^{4}$ and `3ff` + `4` = `403`. Thus, this is stored as

```
0100000000110000011010000000000000000000000000000000000000000000
```

In hexadecimal, this would be

```
4030680000000000
```

Here are some exercises in addition of `doubles`:

1. `404f000000000000` + `404f000000000000` = `405f000000000000`
2. `3f67000000000000` + `3f5c000000000000` = `3f72800000000000`
3. `6442000000000000` + `6466000000000000` = `646a800000000000`
4. `34ae000000000000` + `342c000000000000` = `34ae1c0000000000`
5. `401f500000000000` + `3fda700000000000` = `40207b8000000000`
6. `43e0900000000000` + `444d900000000000` = `444dd24000000000`
7. `40be000000000000` + `c0ba000000000000` = `4090000000000000`
8. `bed0000000000000` + `3ec4000000000000` = `beb8000000000000`

Here are some exercises in multiplication of `doubles`:

1. `3ffd000000000000` × `3ffb000000000000` = `4008780000000000`
2. `4015000000000000` × `400f000000000000` = `4034580000000000`
3. `4030000000000000` × `3fb0000000000000` = `3ff0000000000000`
4. `4010000000000000` × `3fb7000000000000` = `3fd7000000000000`
5. `4008000000000000` × `3fd5555555555555` = `3ff0000000000000`
6. `4026000000000000` × `c01c000000000000` = `c053400000000000`
7. `bfbc000000000000` × `4032000000000000` = `bfff800000000000`
8. `bfe7000000000000` × `bfd7000000000000` = `3fd0880000000000`

2.5 Summary of representations of real numbers

In this topic,

Acknowledgements

Maryam Al-Ajeel
Aditya Sharma
Haotong Liu
Hariharan Karthikeyan
Sanje Divakaran