Notes for Lecture 1

Wed, 8/20/2025

How computers represent numbers

Digital computers deal with all data in the form of plenty of **bits**. Each bit is either a 0 or a 1.

Comment. Quantum computers instead work with **qubits** (short for quantum bit), each of which is a linear combination $\alpha\boxed{0}+\beta\boxed{1}$ of basic bits $\boxed{0}$ and $\boxed{1}$, where α and β are complex numbers with $|\alpha|^2+|\beta|^2=1$. As such a single qubit theoretically contains an infinite amount of classical information. Note that a classical bit is the special case where α and β are both 0 or 1.

For efficiency, the **CPU** (central processing unit) of a computer deals with several bits at once. Current CPUs typically work with 64 bits at a time.

About 20 years ago, CPUs were typically working with 32 bits at a time instead.

Note that 64 bits can store $2^{64}=18446744073709551616$ many different values. That is a large number but may be limited for certain applications.

For instance, modern cryptography often works with integers that are 2048 bits large. Clearly, such an integer cannot be stored in a single fundamental 64 bit block.

Representations of integers in different bases

In everyday life, we typically use the **decimal system** to express numbers. For instance:

$$1234=1\cdot 10^3+2\cdot 10^2+3\cdot 10^1+4\cdot 10^0.$$

10 is called the base, and $1,2,3,4$ are the digits in base 10. To emphasize that we are using base 10, we will write $1234=(1234)_{10}$. Likewise, we write

$$(1234)_b=1\cdot b^3+2\cdot b^2+3\cdot b^1+4\cdot b^0.$$

In this example, $b>4$, because, if b is the base, then the digits have to be in $\{0,1,\ldots,b-1\}$.

Comment. In the above examples, it is somewhat ambiguous to say whether 1 or 4 is the first or last digit. To avoid confusion, one refers to 4 as the **least significant digit** and 1 as the **most significant digit**.

Example 1. $25=16+8+1=\boxed{1}\cdot 2^4+\boxed{1}\cdot 2^3+\boxed{0}\cdot 2^2+\boxed{0}\cdot 2^1+\boxed{1}\cdot 2^0$.
Accordingly, $25=(11001)_2$.

While the approach of the previous example works well for small examples when working by hand (if we are comfortable with powers of 2), the next example illustrates a more algorithmic approach.

Example 2. Express 49 in base 2.

Solution.

- $49=24\cdot 2+\boxed{1}$. Hence, $49=(\ldots 1)_2$ where $\ldots$ are the digits for 24.
- $24=12\cdot 2+\boxed{0}$. Hence, $49=(\ldots 01)_2$ where $\ldots$ are the digits for 12.
- $12=6\cdot 2+\boxed{0}$. Hence, $49=(\ldots 001)_2$ where $\ldots$ are the digits for 6.
- $6=3\cdot 2+\boxed{0}$. Hence, $49=(\ldots 0001)_2$ where $\ldots$ are the digits for 3.
- $3=1\cdot 2+\boxed{1}$. Hence, $49=(\ldots 10001)_2$ where $\ldots$ are the digits for 1.
- $1=0\cdot 2+\boxed{1}$. Hence, $49=(110001)_2$.

Armin Straub
straub@southalabama.edu

Example 3. Express 49 in base 3.

Solution.

- $49 = 16 \cdot 3 + \boxed{1}$
- $16 = 5 \cdot 3 + \boxed{1}$
- $5 = 1 \cdot 3 + \boxed{2}$
- $1 = 0 \cdot 3 + \boxed{1}$

Hence, $49 = (1211)_3$.

Other bases.
What is 49 in base 5? $49 = (144)_5$.
What is 49 in base 7? $49 = (100)_7$.

Fixed-point numbers

Example 4. (warmup)

(a) Which number is represented by $(11001)_2$?

(b) Which number is represented by $(11.001)_2$?

(c) Express 5.25 in base 2.

(d) Express 2.625 in base 2. [Note that $2.625 = 5.25/2$.]

Solution.

(a) $(11001)_2 = 2^4 + 2^3 + 2^0 = 16 + 8 + 1 = 25$

(b) $(11.001)_2 = 2^1 + 2^0 + 2^{-3} = 3.125$
Alternatively, $(11.001)_2$ should be $(11001)_2 = 25$ divided by 2^3 (because we move the "decimal" point by three places). Indeed, $(11.001)_2 = 25/2^3 = 3.125$.
Comment. The professional term for "decimal" point would be radix point or, in base 2, binary point (but I have heard neither of these used much in my personal experience).

(c) Note that $5.25 = 2^2 + 2^0 + 2^{-2}$. Hence $5.25 = (101.01)_2$.

(d) Since multiplication (respectively, division) by 2 shifts the digits to the left (respectively, right), we deduce from $5.25 = (101.01)_2$ that $2.625 = (10.101)_2$.

Armin Straub
straub@southalabama.edu

Example 5. Express 1.3 in base 2.

Solution. Suppose we want to determine 6 binary digits after the "decimal" point. Note that multiplication by $2^6 = 64$ moves these 6 digits before the "decimal" point.
$2^6 \cdot 1.3 = 83.2$ and $83.2 = (1010011.\cdots)_2$ (fill in the details!).
Hence, shifting the "decimal" point, we find $1.3 = (1.010011\cdots)_2$.

Solution. Alternatively, we can compute one digit at a time by multiplying with 2 each time:

- $\boxed{1}.3$ [Hence, the most significant digit is $\boxed{1}$ with 0.3 still to be accounted for.]
- $2 \cdot 0.3 = \boxed{0}.6$ [Hence, the next digit is $\boxed{0}$ with 0.6 still to be accounted for.]
- $2 \cdot 0.6 = \boxed{1}.2$ [Hence, the next digit is $\boxed{1}$ with 0.2 still to be accounted for.]
- $2 \cdot 0.2 = \boxed{0}.4$ [Hence, the next digit is $\boxed{0}$ with 0.4 still to be accounted for.]
- $2 \cdot 0.4 = \boxed{0}.8$ [Hence, the next digit is $\boxed{0}$ with 0.8 still to be accounted for.]
- $2 \cdot 0.8 = \boxed{1}.6$ [Hence, the next digit is $\boxed{1}$ with 0.6 still to be accounted for.]
- And now things repeat because we started with 0.6 before...

Hence, $1.3 = (1.01001\cdots)_2$ and the final digits 1001 will be repeated forever: $1.3 = (1.0100110011001\cdots)_2$

Comment. As we saw here, fractions with a finite decimal expansion (like $13/10 = 1.3$) do not need to have a finite binary expansion (and typically don't).

Example 6. Express 0.1 in base 2.

Solution.

- $2 \cdot \boxed{0}.1 = \boxed{0}.2$
- $2 \cdot 0.2 = \boxed{0}.4$
- $2 \cdot 0.4 = \boxed{0}.8$
- $2 \cdot 0.8 = \boxed{1}.6$
- $2 \cdot 0.6 = \boxed{1}.2$ and now things repeat...

Hence, $0.1 = (0.00011\cdots)_2$ and the final digits 0011 repeat: $0.1 = (0.0001100110011\cdots)_2$

Example 7. Express $35/6$ in base 2.

Solution. Note that $35/6 = 5 + 5/6$ so that $35/6 = (101.\cdots)_2$ with $5/6$ to be accounted for.

- $2 \cdot 5/6 = \boxed{1} + 4/6$
- $2 \cdot 4/6 = \boxed{1} + 2/6$
- $2 \cdot 2/6 = \boxed{0} + 4/6$ and now things repeat...

Hence, $35/6 = (101.110\cdots)_2$ and the final two digits 10 repeat: $35/6 = (101.110101010\cdots)_2$

Armin Straub
straub@southalabama.edu

Floating-point numbers (and IEEE 754)

Possibilities for binary representations of real numbers:

- fixed-point number: $\pm x.y$ with x and y of a certain number of bits

$\pm x$ is called the integer part, and y the fractional part.

- floating-point number: $\pm 1.x \cdot 2^y$ with x and y of a certain number of bits

$\pm 1.x$ is called the significand (or mantissa), and y the exponent.

In other words, the floating-point representation is "scientific notation in base 2".

Important comment. In order to represent as many numbers as possible using a fixed number of bits, it is crucial that we avoid unnecessarily having different representations for the same number. That is why the exponent y above is chosen so that the significand starts with 1 followed by the "decimal" point. This has the added benefit of not needing to actually store that 1 (rather it is "implied" or "hidden").

IEEE 754 is the most widely used standard for floating-point arithmetic and specifies, most importantly, how many bits to use for significand and exponent.

1985: first version of the standard

IEEE: Institute of Electrical and Electronics Engineers

Used by many hardware FPUs (floating point units) which are part of modern CPUs.

For more details: https://en.wikipedia.org/wiki/IEEE_754

Example 8. $4.5 = (100.1)_2 = (1.001)_2 \cdot 2^2 = +\underbrace{1.001}_{\text{binary}} \cdot 2^2$

Next, we will see exactly how IEEE 754 would store this as bits (32 bits in the case of "single precision").

IEEE 754 offers several choices but the two most common are:

- **single precision**: 32 bit (1 bit for sign, 23 bit for significand, 8 bit for exponent)
- **double precision**: 64 bit (1 bit for sign, 52 bit for significand, 11 bit for exponent)

A floating-point number: $\pm 1.x \cdot 2^y$ is then stored as follows:

- **Sign:** 1 bit is used for the sign: 0 for $+$ and 1 for $-$.
- **Significand:** Recall that the significand is preceded by an implied bit equal to 1.

In other words, if the significand is $\pm 1.x$ then only the bits for x are stored.

- **Exponent:** In IEEE 754, a constant, called a **bias**, is added to the exponent so that all exponents are positive (this avoids using a sign bit for the exponent).

If the exponent is y, then one stores $y + \text{bias}$ where $\text{bias} = 2^7 - 1 = 127$ for single precision ($\text{bias} = 2^{10} - 1$ for double precision).

Comment. IEEE 754 also offers half precision as well as higher precisions but single and double are the most commonly used because this is what older and current CPUs use. Moreover, the base (also called radix) can also be 10 instead of 2.

Example 9. Represent 4.5 as a single precision floating-point number according to IEEE 754.

Solution. $4.5 = (100.1)_2 = (1.001)_2 \cdot 2^2 = \boxed{+}\underbrace{1.\boxed{001}}_{\text{binary}} \cdot 2^2$

The exponent 2 gets stored as $2 + 127 = \boxed{1000,0001}$.

Overall, 4.5 is stored as $\boxed{0}\,\boxed{1000,0001}\,\boxed{0010,0000,0000,0000,0000,000}$.

Example 10. Represent -0.1 as a single precision floating-point number according to IEEE 754.

Solution. In Example 6, we computed that $0.1 = (0.0001100110011\cdots)_2$.

Hence: $-0.1 = \boxed{-}\underbrace{1.\boxed{1001,1001...}}_{\text{binary}} \cdot 2^{-4}$

The exponent -4 gets stored as $-4 + 127 = \boxed{0111,1011}$.

Overall, -0.1 is stored as $\boxed{1}\,\boxed{0111,1011}\,\boxed{1001,1001,1001,...}$.

Caution. Note that we are not able to store -0.1 exactly. Therefore we need to be careful about how to choose the final bit to best approximate -0.1. According to IEEE 754, the final bit should be 1 (rather than 0 which we would get if we simply truncated) so that -0.1 gets stored as $\boxed{1}\,\boxed{0111,1011}\,\boxed{1001,1001,1001,1001,1001,101}$.

Note that it certainly makes sense to round up the final 0 to 1 because it is followed by $1...$ (this is similar to us rounding up a final 0 in decimal to 1 if it is followed by $5...$).

Example 11. Give an example where one should not use floats.

Solution. One should not use floats when dealing with money. That is because, as we just saw, an amount such as 0.10 dollars cannot be represented exactly using a float (when using base 2, as is the default in most programming languages such as Python) and thus will get rounded. This is problematic when working with money.

Comment. For most purposes, the easiest way to avoid these issues is to store dollar amounts as cents. For the latter we can then simply use integers and work with exact numbers (no rounding).

Armin Straub
straub@southalabama.edu

Example 12. (reasons for floats) Almost universally, major programming languages use floating-point numbers for representing real numbers. Why not fixed-point numbers?

Solution. Fixed-point numbers have some serious issues for scientific computation. Most notably:

- Scaling a number typically results in a loss of precision.
  For instance, dividing a number by 2^r and then multiplying it with 2^r loses r digits of precision (in particular, this means that it is computationally dangerous to change units). Make sure that you see that this does not happen for floating-point numbers.
- The range of numbers is limited.
  For instance, the largest number is on the order of 2^N where N is the number of bits used for the integer part. On the other hand, a floating-point number can be of the order of $2^{2^{M-1}}$ where M is the number of bits used for the exponent. (Make sure you see how enormous of a difference this is! See the previous example for the case of double precision.)

Moreover, as noted in the box below, fixed-point numbers do not really offer anything that isn't already provided by integers. This is the reason why most programming languages don't even offer built-in fixed-point numbers.

Fixed-point numbers are essentially like integers.

For instance, instead of 21.013 (say, seconds) we just work with 21013 (which now is in milliseconds).

Using Python as a basic calculator

Example 13. `Python` We can use Python as a basic calculator. Addition, subtraction, multiplication and division work as we would probably expect:

```
>>> 16*3+1

    49

>>> 3/2

    1.5
```

To compute powers like 2^{64}, we can use ** (two asterisks).

```
>>> 2**64

    18446744073709551616
```

Division with remainder of, say, 49 by 3 results in $49 = 16 \cdot 3 + 1$. In Python, we can use the operators // and % to compute the result of the division as well as the remainder:

```
>>> 49 // 3

    16

>>> 49 % 3

    1
```

% is called the **modulo** operator. For instance, we say that 49 modulo 3 equals 1 (and this is often written as $49 \equiv 1 \pmod 3$).

IMPORTANT. The commands here are entered into an interactive Python interpreter (this is indicated by the >>>). When running a Python script, we need to use `print(16*3+1)` or `print(3/2)` to receive the first two outputs above.

Example 14. `Python` Python automatically uses (double precision) floats when we enter numbers with a decimal point or as the result of divisions.

```
>>> 1.1

    1.1

>>> 1/3

    0.3333333333333333
```

Comment. Note how we can see (roughly) the 52 bit precision of the double precision floats (there are 16 decimal digits after the decimal point, which translates to about $16 \cdot \log_2 10 \approx 53.15$ binary digits; this corresponds to about 52 bit after the first implicit 1).

IMPORTANT. As noted earlier, the commands here are entered into an interactive Python interpreter (this is indicated by the >>>). When running a Python script, we need to use `print(1.1)` or `print(1/3)` to receive the above outputs.

For very large (or very small) numbers, scientific notation is often used:

```
>>> 2.0 * 10**80

    2e+80

>>> (1/2)**100

    7.888609052210118e-31
```

The following are two things that are (somewhat) special to Python and are often handled differently in other programming languages. First, integers are not limited in size (often, integers are limited to 64 bits, which can cause issues like overflow when one exceeds the 2^{64} possibilities). This is illustrated by the following (this explains why we wrote 2.0 above):

```
>>> 2 * 10**80

    200000000000000000000000000000000000000000000000000000000000000000000000000000
```

Second, Python likes to throw errors when a computation runs into an issue (there are nice ways to "catch" these errors in a program and to react accordingly, but that is probably beyond what we will use Python for).

```
>>> 1 / 0

    ZeroDivisionError: division by zero
```

Some other programming languages would instead (silently, without error messages) return special floats representing $+\infty$, $-\infty$ or `NaN` (not-a-number).

Example 15. `Python` Explain the following floating-point rounding issue:

```
>>> 1 + 1 + 1 == 3
```

```
True
```

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

```
>>> 0.1 + 0.1 + 0.1
```

```
0.30000000000000004
```

Solution. As we saw in Example 6, 0.1 cannot be stored exactly as a floating-point number (when using base 2). Instead, it gets rounded up slightly. After adding three copies of this number, the error has increased to the point that it becomes visible as in the above output.

IMPORTANT. In the Python code above, we used the operator == (two equal signs) to compare two quantities. Note that we cannot use = (single equal sign) because that operator is used for assignment (`x = y` assigns the value of `y` to `x`, whereas `x == y` checks whether `x` and `y` are equal).

Comment. As the above issue shows, we should never test two floats x and y for equality. Instead, one typically tests whether the difference $|x - y|$ is less than a certain appropriate threshold. An alternative practical way is to round the floats before comparison (below, we round to 8 decimal digits):

```
>>> round(0.1 + 0.1 + 0.1, 8) == round(0.3, 8)
```

```
True
```

Review. Possibilities for binary representations of real numbers:

- fixed-point number: $\pm x.y$ with x and y of a certain number of bits
- floating-point number: $\pm 1.x \cdot 2^y$ with x and y of a certain number of bits
  IEEE 754, single precision: 32 bit (1 bit for sign, 23 bit for significand x, 8 bit for exponent y)
  IEEE 754, double precision: 64 bit (1 bit for sign, 52 bit for significand x, 11 bit for exponent y)

Example 16. Represent -0.375 as a single precision floating-point number according to IEEE 754.

Solution. $-0.375 = -\frac{3}{8} = -3 \cdot 2^{-3} = \boxed{-}\underbrace{1.\boxed{1}}_{\text{binary}} \cdot 2^{-2}$

The exponent -2 gets stored as $-2+127 = \boxed{0111,1101}$. (Recall that the bias $2^7-1=127$ is being added to the exponents. Also, it helps to keep in mind that $127 = (0111,1111)_2$.)

Overall, -0.375 is stored as $\boxed{1}\boxed{0111,1101}\boxed{1000,0000,\ldots}$.

Example 17. Represent $-1/7$ as a single precision floating-point number according to IEEE 754.

Solution. To write $1/7$ in binary, we repeatedly multiply with 2:

- $2 \cdot 1/7 = \boxed{0} + 2/7$
- $2 \cdot 2/7 = \boxed{0} + 4/7$
- $2 \cdot 4/7 = \boxed{1} + 1/7$ and now things repeat...

Hence, $1/7 = (0.001\cdots)_2$ and the final three digits 001 repeat: $1/7 = (0.001001001\cdots)_2$. Hence:
$-\frac{1}{7} = \boxed{-}\underbrace{1.\boxed{001001\ldots}}_{\text{binary}} \cdot 2^{-3}$

The exponent -3 gets stored as $-3+127 = \boxed{0111,1100}$.

Overall, $-1/7$ is stored as $\boxed{1}\boxed{0111,1100}\boxed{0010,0100,\ldots}$.

Example 18. What is the largest single precision floating-point number according to IEEE 754?

Technical detail. The exponent $(1111,1111)_2 = 2^8-1 = 255$ is reserved for special numbers (such as infinities or "NaN"). Hence, the largest exponent corresponds to $(1111,1110)_2 = 254$.

Solution. The largest single precision floating-point number is

$$\boxed{1}\boxed{1111,1110}\boxed{1111,1111,\ldots} = +1.111\ldots \cdot 2^{127} \approx 2^{128} = 2^{2^7} \approx 3.4 \cdot 10^{38}.$$

Here, we used that $(1111,1110)_2 = 2^8-2 = 254$ so that the actual exponent is $254-127=127$.

For comparison. The largest 32 bit (signed) integer is $2^{31}-1 \approx 2.1 \cdot 10^9$ (the exponent is $31 = 32-1$ to account for using 1 bit to store the sign). You might find this surprisingly small. And, indeed, 32 bit is not enough to address all memory locations in modern systems which is why the step to 64 bit was necessary.

Double precision. Likewise, the largest double precision floating-point number is

$$\boxed{1}\boxed{111,1111,1110}\boxed{1111,1111,\ldots} = +1.111\ldots \cdot 2^{1023} \approx 2^{1024} = 2^{2^{10}} \approx 1.8 \cdot 10^{308}.$$

On the other hand, the largest 64 bit (signed) integer is $2^{63}-1 \approx 9.2 \cdot 10^{18}$.

Armin Straub
straub@southalabama.edu

Interlude: Representing negative integers

In our discussion of IEEE 754, we have seen two ways of storing signed numbers using r bits:

- **sign-magnitude:** One bit is used for the sign, the other bits for the absolute value.
  Typically, the most significant bit is set to 0 for positive numbers and 1 for negative numbers.
  This is what happens in IEEE 754 for the most significant bit.
- **offset binary (or biased representation):** Instead of storing the signed number n, we store $n+b$ with b called the bias.
  Often the bias is chosen to be $b=2^{r-1}$.
  This is what happens in IEEE 754 for the exponent (however, the bias is chosen as $b=2^{r-1}-1$).

(Perhaps) surprisingly, neither of these is how signed integers are most commonly stored:

- **two's complement:** The most significant bit is counted as -2^{r-1} instead of as 2^{r-1}.
  Two's complement is used by nearly all modern CPUs.
  For instance, using 4 bits, the number 3 is stored as 0011 (2^1+2^0), while -3 is stored as 1101 ($-2^3+2^2+2^0$). Similarly, 5 is stored as 0101, while -5 is stored as 1011.
  To negate a number n in this representation, we invert all its bits and then add 1.
  As a result, adding a number to its negative produces all ones plus 1 (using r bits in the usual way, all ones corresponds to 2^r-1 so that adding 1 results in 2^r; which is where the name comes from).
  Important. At the level of bits, addition in the two's complement representation works exactly as addition of unsigned integers. For instance, consider the addition $0010+1011=1101$: interpreted as unsigned integers, this is $2+11=13$; alternatively, interpreted as signed integers (using two's complement), this is $2+(-5)=-3$. Likewise, the same is true for multiplication.
  Advanced comment. Two's complement makes particular sense when we interpret it in terms of modular arithmetic (ignore this comment if you are not familiar with this). Namely, if using r bits, all numbers are interpreted as residues modulo 2^r (recall that -1 and 2^r-1 represent the same residue; similarly, 2^{r-1} and -2^{r-1} are the same modulo 2^r). Instead of representing the residues by $0,1,2,\ldots,2^r-1$, we then make the choice to represent them by $0,1,2,\ldots,-3,-2,-1$. Since we can compute with residues as with ordinary numbers, this explains why, using two's complement, addition and multiplication work the same as if the numbers are interpreted as unsigned (assuming that no overflow occurs).
  Comment. Two's complement can also be interpreted as follows: instead of storing the signed number n, we store $n+2^r$, where we only use the r least significant bits (thus throwing away the bit with value 2^r). Apart from this last bit (pun intended!), this is like offset binary.

There are yet further possibilities that are used in practice, most notably:

- **ones' complement:** A positive number n is stored as usual with the most significant bit set to 0, while its negative $-n$ is stored by inverting all bits.
  For instance, using 4 bits, the number 5 is stored as 0101, while -5 is stored as 1010.
  As a result, adding a number to its negative produces all ones (hence the name).

https://en.wikipedia.org/wiki/Signed_number_representations

Example 19. Which of the above four representations has more than one representation of 0?

Solution. In the sign-magnitude as well as in the ones' complement representation, we have a $+0$ and a -0.

Example 20. Express -25 in binary using the two's complement representation with 6 bits.

Solution. Since $25=(011001)_2$, -25 is represented by 100111 (invert all bits, then add 1).
Alternatively, note that $-25=-2^5+7$ and $7=(111)_2$ to arrive at the same representation.
As another alternative, we store $-25+2^6=7$, resulting in the same representation.

Example 21. (warmup) Using 64 bits, how many decimal digits can we store? With 53 bits?

Solution. Using 64 bits we can store 2^{64} different numbers. Since $\log_{10}(2^{64}) = 64\log_{10}(2) \approx 19.27$, we can store 19 decimal digits using 64 bits.

Likewise, since $\log_{10}(2^{53}) = 53\log_{10}(2) \approx 15.95$, we can store 15 decimal digits using 53 bits.

Example 22. To store 19 decimal digits, how many bits are needed?

Solution. There are 10^{19} many possibilities with 19 decimal digits. Since $\log_2(10^{19}) = 19\log_2(10) \approx 63.12$, we need 64 bits.

Comment. Note how this matches the conclusion of our computation in the previous example.

Example 23. `Python` Let us perform the previous calculation of $13\log_2(10)$ using Python. First of all, we need to get access to the $\log$ function because it is not available by default. Instead it resides in a module called `math`:

```
>>> from math import log
>>> 13*log(10, 2)
   43.18506523353572
```

It might be unexpected that the 2 is the second argument of $\log$. If you want to learn more about how to use a function, you can enter the function name followed by a question mark:

```
>>> log?
```

Another floating-point issue

Example 24. Explain the following floating-point issue about mixing large and small numbers:

```
>>> 10.**9 + 10.**-9
   1000000000.0
>>> 10.**9 + 10.**-9 == 10.**9
   True
>>> 10.**9
   1000000000.0
>>> 10.**-9
   1e-09
```

Solution. Recall that double precision floats (which is what Python currently uses) use 52 bits for the significand which, together with the initial 1 (which is not stored), means that we are able to store numbers with 53 binary digits of precision. This translates to about $\log_{10}(2^{53}) = 53\log_{10}(2) \approx 15.95$ decimal digits. However, storing $10^9 + 10^{-9}$ exactly requires 19 decimal digits.

Coding in Python: binary digits of 0.1

In the next several examples, we will gradually get more professional in using Python for writing our first serious code.

Example 25. `Python` Recall that, to express, say, 0.1 in binary, we compute:

- $2 \cdot 0.1 = \boxed{0}.2$
- $2 \cdot 0.2 = \boxed{0}.4$
- $2 \cdot 0.4 = \boxed{0}.8$
- $2 \cdot 0.8 = \boxed{1}.6$
- $2 \cdot 0.6 = \boxed{1}.2$
- and so on...

The above 5 multiplications with 2 reveal 5 digits after the "decimal" point: $0.1 = (0.00011\cdots)_2$. (We can further see that the last four digits repeat; but we will ignore that fact here.)

Let us use Python to do this computation for us. We will start with very basic and naive code, and then upgrade it later.

```
>>> x = 0.1  # or any value < 1
```

Comment. Everything after the # symbol is considered a comment. This is useful for reminding ourselves of things related to the surrounding code. Comments are usually on a separate line but can be used as above (here, we remind ourselves that the code that follows is not going to handle a number like 2.1 correctly).

To have Python do the above computation for us, we plan to multiply x by 2 (call the result x again), collect a digit (we get that digit as the integer part of x), then subtract that digit from x and repeat. Python has a function called `trunc` which "truncates" a float to its integer part but we need to import it from a package called `math` to make it available.

```
>>> from math import trunc
```

Advanced comment. We can also use `*` in place of `trunc` to import all the functions from the `math` package. However, it is good practice to be explicit about what we need from a package. Note that the function `trunc` is very close to the function `floor` (which computes $\lfloor x \rfloor$, the floor of x, which is the closest integer when rounding down) which also seems appropriate here; however, `floor` returns a float rather than an integer, and we prefer the latter. Also note that we could use `int` (this is a general function that converts an input to an integer) instead of `trunc`. We chose `trunc` because it is more explicitly what we want, and because it gives us a chance to see how to import functions from a package.

We are now ready to compute the first digit:

```
>>> x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit

0
```

By copying-and-pasting these four lines four more times, we can produce the next four digits:

```
>>> x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
    x = 2*x
    digit = trunc(x)
    print(digit)
    x = x-digit
```

```
0
0
1
1
```

Clearly, we should not have to copy-and-paste repeated code like this. We will fix this issue in the next example, as well as discuss several other important improvements.

Example 26. Python Let us return to the task of using Python to express, say, 0.1 in binary. Last time, we copied four lines of code 5 times to produce 5 digits. Instead, to repeat something a certain number of times, we should use a **for loop**. For instance:

```
>>> for i in range(3):
        print('Hello')
```

```
Hello
Hello
Hello
```

Homework. Replace `print('Hello')` with `print(i)`.

Important comment. The indentation in the second line serves an important purpose in Python. All lines (after the first) that are indented by the same amount will be repeated. Test this by adding a non-indented line containing `print('Bye!')` at the end.

With this in mind, we can upgrade our previous code as follows:

```
>>> x = 0.1  # or any value < 1
    nr_digits = 5  # we want this many digits of x
    from math import trunc
    for i in range(nr_digits):
        x = 2*x
        digit = trunc(x)
        print(digit)
        x = x-digit
```

```
0
0
0
1
1
```

Example 27. Python As our next upgrade, let us collect the digits in a list instead of printing them to the screen. Here is how we can create a list in Python and add an element to it:

```
>>> L = [1, 2, 3]

>>> L.append(4)

>>> print(L)

   [1, 2, 3, 4]
```

Here is our code adjusted for using a list (and now it is more pleasant to ask for more digits):

```
>>> x = 0.1  # or any value < 1
    nr_digits = 10  # we want this many digits of x
    digits = []  # this list will store the digits of x
    from math import trunc
    for i in range(nr_digits):
        x = 2*x
        digit = trunc(x)
        digits.append(digit)
        x = x-digit
    print(digits)

   [0, 0, 0, 1, 1, 0, 0, 1, 1, 0]
```

Example 28. Python For our final upgrade, we collect the code into a function that we call `fracpart_digits`. This is crucial for making it possible to use the code on different numbers.

```
>>> def fracpart_digits(x, nr_digits):
        digits = []
        from math import trunc
        for i in range(nr_digits):
            x = 2*x
            digit = trunc(x)
            digits.append(digit)
            x = x-digit
        return digits
```

We are now able to compute the digits of numbers by simply calling our function:

```
>>> fracpart_digits(0.1, 10)

   [0, 0, 0, 1, 1, 0, 0, 1, 1, 0]

>>> fracpart_digits(0.2, 10)

   [0, 0, 1, 1, 0, 0, 1, 1, 0, 0]

>>> from math import pi

>>> fracpart_digits(pi/4, 10)

   [1, 1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Comment. Recall that, if you are not in a Python console, you need to add `print(..)` to see any output.

As an advanced use of lists, here is how we could compute 5 digits of $1/n$ for $n \in \{2,3,4,5\}$:

```
>>> [fracpart_digits(1./n, 5) for n in range(2,6)]

   [[1, 0, 0, 0, 0], [0, 1, 0, 1, 0], [0, 1, 0, 0, 0], [0, 0, 1, 1, 0]]
```

Comment. Note how the digits of $1/2 = (0.1)_2$ and $1/4 = (0.01)_2$ are particularly easy to verify.

Errors: absolute and relative

Suppose that the true value is x and that we approximate it with y.

- The **absolute error** is $|y-x|$.
- The **relative error** is $\left|\frac{y-x}{x}\right|$.

For many applications, the relative error is much more important. Note, for instance, that it does not change if we scale both x and y (in other words, it doesn't change if we change units from, say, meters to millimeters). Speaking of units, note that the relative error is dimensionless (it has no units even if x and y do).

Example 29. There are lots of interesting approximations of π. In each of the following cases, determine both the absolute and the relative error.

(a) $\pi \approx \frac{22}{7}$ $(22/7 \approx 3.14286)$

(b) $\pi \approx \sqrt[4]{9^2+19^2/22}$ (This approximation is featured in https://xkcd.com/217/.)

Solution.

(a) The absolute error is $\left|\frac{22}{7}-\pi\right| \approx 0.0013 = 1.3\cdot 10^{-3}$.

The relative error is $\left|\frac{\frac{22}{7}-\pi}{\pi}\right| \approx 0.00040 = 4.0\cdot 10^{-4}$.

Comment. Sometimes the relative error is quoted as a "percentage error". Here, this is 0.04%.

(b) The absolute error is $\left|\sqrt[4]{9^2+19^2/22}-\pi\right| \approx 1.0\cdot 10^{-9}$.

The relative error is $\left|\frac{\sqrt[4]{9^2+19^2/22}}{\pi}\right| \approx 3.2\cdot 10^{-10}$.

Example 30. (homework) π^{10} is rounded to the closest integer. Determine both the absolute and the relative error (to three significant digits).

Solution. $\pi^{10} \approx 93{,}648.0475$

The absolute error is $|93{,}648-\pi^{10}| \approx 0.0475$.

The relative error is $\left|\frac{93{,}648-\pi^{10}}{\pi^{10}}\right| \approx 5.07\cdot 10^{-7}$.

Example 31. Strangely, $e^{\pi}-\pi = 19.999099979\ldots$. Determine both the absolute and the relative error when approximating this number by 20.

https://xkcd.com/217/

Solution. The absolute error is $|20-(e^{\pi}-\pi)| \approx 9.0\cdot 10^{-4}$.

The relative error is $\left|\frac{20-(e^{\pi}-\pi)}{e^{\pi}-\pi}\right| \approx 4.5\cdot 10^{-5}$.

Example 32. One of the most famous/notorious mathematical results is **Fermat's last theorem**. It states that, for $n > 2$, the equation $x^n + y^n = z^n$ has no positive integer solutions!

Pierre de Fermat (1637) claimed in a margin of Diophantus' book *Arithmetica* that he had a proof ("I have discovered a truly marvellous proof of this, which this margin is too narrow to contain.").

It was finally proved by Andrew Wiles in 1995 (using a connection to modular forms and elliptic curves).

This problem is often reported as the one with the largest number of unsuccessful proofs.

On the other hand, in a Simpson's episode, Homer (in 3D!) encounters the formula

$$1782^{12} + 1841^{12} \text{ “}=\text{” } 1922^{12}.$$

If you check this on an old calculator it might confirm the equation. However, the equation is not correct, though it is "nearly": $1782^{12} + 1841^{12} - 1922^{12} \approx -7.002 \cdot 10^{29}$.

Why would that count as "nearly"? Well, the smallest of the three numbers, $1782^{12} \approx 1.025 \cdot 10^{39}$, is bigger by a factor of more than 10^9. So the difference is extremely small in comparison.

More precisely, if $1782^{12} + 1841^{12}$ is the true value, then approximating it with 1922^{12} produces

- an absolute error of $|1782^{12} + 1841^{12} - 1922^{12}| \approx 7.00 \cdot 10^{29}$ (rather large), and
- a relative error of $\left|\frac{1782^{12} + 1841^{12} - 1922^{12}}{1782^{12} + 1841^{12}}\right| \approx 2.76 \cdot 10^{-10}$ (very small).

Comment. We can immediately see that Homer's formula is not quite correct by looking at whether each term is even or odd. Do you see it?

http://www.bbc.com/news/magazine-24724635

Solving equations

Now that we have discussed how computers deal with numbers, it is natural to think about how to compute numbers of interest. Often, these arise as solutions of equations.

For instance. As simple but instructive instances, how do we compute numbers like $\sqrt{2}$, $\log(3)$ or π?

Note that any equation can be put into the form $f(x)=0$ where $f(x)$ is some function. Solving that equation is equivalent to finding roots of that function.

There are many approaches to root finding see, for instance:

https://en.wikipedia.org/wiki/Root-finding_algorithms

Comment. The solve routines implemented in professional libraries often use hybrid versions of the methods we discuss below (as well as others). For instance, Brent's method (used, for instance, in MATLAB, PARI/GP, R or SciPy) is a hybrid of three: the bisection and secant methods as well as inverse quadratic interpolation.

Comment. It depends very much on $f(x)$ which approach to root finding is best. For instance, is $f(x)$ a nice (i.e. differentiable) function? Is it costly to evaluate $f(x)$? This is the reason for why there are many different approaches to finding roots and why it is important to understand their strenghts and weaknesses.

The bisection method

Suppose that we wish to find a root of the continuous function $f(x)$ in the interval $[a,b]$.

If $f(a)<0$ and $f(b)>0$, then the **intermediate value theorem** tells us that there must be an $r\in[a,b]$ such that $f(r)=0$. (Likewise if $f(a)>0$ and $f(b)<0$.)

Comment. Recall that the intermediate value theorem requires $f(x)$ to be continuous so that there are no jumps or singularities.

The **bisection method** now cuts the interval $[a,b]$ into two halves by computing the **midpoint** $c=\frac{a+b}{2}$. Depending on whether $f(c)\leqslant 0$ or $f(c)\geqslant 0$, we conclude that there must be a root in $[a,c]$ or in $[c,a]$. In either case, we have cut the length of the interval of uncertainty in half.

This process is then repeated until we have a sufficiently small interval that is guaranteed to contain a root of $f(x)$.

Example 33. Determine an approximation for $\sqrt[3]{2}$ by applying the bisection method to the function $f(x)=x^3-2$ on the interval $[1,2]$. Perform 4 steps of the bisection method.

Comment. Note that it is obvious that $1<\sqrt[3]{2}<2$ so that the interval $[1,2]$ is a natural choice.

Armin Straub
straub@southalabama.edu

Solution. Note that $f(1) = -1 < 0$ while $f(2) = 6 > 0$. Hence, $f(x)$ must indeed have a root in the interval $[1, 2]$.

- $[a, b] = [1, 2]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).
  The midpoint is $c = \frac{a+b}{2} = \frac{1+2}{2} = \frac{3}{2}$. We compute $f(c) = c^3 - 2 = \frac{27}{8} - 2 = \frac{11}{8} > 0$.
  Hence, $[a, c] = \left[1, \frac{3}{2}\right]$ must contain a root of $f(x)$.

- $[a, b] = \left[1, \frac{3}{2}\right]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).
  The midpoint is $c = \frac{a+b}{2} = \frac{1+3/2}{2} = \frac{5}{4}$. We compute $f(c) = -\frac{3}{64} < 0$.
  Hence, $[c, b] = \left[\frac{5}{4}, \frac{3}{2}\right]$ must contain a root of $f(x)$.

- $[a, b] = \left[\frac{5}{4}, \frac{3}{2}\right]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).
  The midpoint is $c = \frac{a+b}{2} = \frac{11}{8}$. We compute $f(c) = \frac{307}{512} > 0$.
  Hence, $[a, c] = \left[\frac{5}{4}, \frac{11}{8}\right]$ must contain a root of $f(x)$.

- $[a, b] = \left[\frac{5}{4}, \frac{11}{8}\right]$ contains a root of $f(x)$ (since $f(a) < 0$ and $f(b) > 0$).
  The midpoint is $c = \frac{a+b}{2} = \frac{21}{16}$. We compute $f(c) = \frac{1069}{4096} > 0$.
  Hence, $[a, c] = \left[\frac{5}{4}, \frac{21}{16}\right]$ must contain a root of $f(x)$.

After 4 steps of the bisection method, we know that $\sqrt[3]{2}$ must lie in the interval $\left[\frac{5}{4}, \frac{21}{16}\right] = [1.25, 1.3125]$.

Comment. For comparison, $\sqrt[3]{2} \approx 1.2599$. Note that our approximations are a bit more impressive if we think in terms of binary digits. Then each step provides one additional digit of accuracy (because the length of the interval of uncertainty is cut by half).

Comment. The above steps are on purpose written in a repetitive manner (reusing the same variable names a, b, c with new values) to make it easier to translate the process into Python code.

Example 34. (continued) We wish to determine an approximation for $\sqrt[3]{2}$ by applying the bisection method to the function $f(x) = x^3 - 2$ on the interval $[1, 2]$.

(a) After how many iterations of bisection will the final interval have size less than 10^{-6}?

(b) If we approximate $\sqrt[3]{2}$ using the midpoint of the final interval, how many iterations of bisection do we need to guarantee that the (absolute) error is less than 10^{-6}?

Solution.

(a) At each iteration, the width of the interval is divided by 2. Hence, the width of the interval after n steps will be exactly $\frac{2-1}{2^n} = \frac{1}{2^n}$. Solving $\frac{1}{2^n} < 10^{-6}$ for n, we find $n > -\log_2(10^{-6}) = 6\log_2(10) \approx 19.93$. Hence, we need 20 iterations.

(b) Again, the width of the interval after n steps will be exactly $\ell = \frac{2-1}{2^n} = \frac{1}{2^n}$. Since $\sqrt[3]{2}$ is contained in this interval, the absolute error of approximating it with the midpoint is at most $\ell/2 = \frac{1}{2^{n+1}}$. Solving $\frac{1}{2^{n+1}} < 10^{-6}$ for n, we find $n > -\log_2(10^{-6}) - 1 = 6\log_2(10) - 1 \approx 18.93$. Hence, we need 19 iterations.

Comment. We didn't refer to the first part on purpose. Given the answer 20 from the first part, can you see that the answer must be $20 - 1 = 19$?

If we use bisection to compute a root of a (continuous) function $f(x)$ on $[a, b]$, then:

- After n iterations, the (absolute) error is less than $\frac{b-a}{2^{n+1}}$.

This assumes that we approximate the root using the midpoint of the final interval.

Important comment. This error bound is independent of $f(x)$.

- Each additional iteration requires 1 function evaluation.

Python: Implementing the bisection method

Example 35. Recall that the bisection method cuts an interval $[a,b]$ into two halves by computing the midpoint $c=\frac{a+b}{2}$. It then chooses either $[a,c]$ or $[c,a]$ as the improved new interval.

Give a simple condition for when the interval $[a,c]$ is chosen.

Solution. We choose $[a,c]$ if $f(a)$ and $f(c)$ have opposite signs.

This is equivalent to $f(a)f(c)<0$.

Comment. Here, and in the sequel, we will be very nonchalant about the possibility that $f(c)=0$. This would mean that we have accidentally found the actual root. This is not something that we expect to happen in most applications (though serious code would have to consider the case where $f(c)$ is 0 to within the precision we are working with). Note that if $f(c)=0$ then our above rule would always choose the right half of the interval (and that would be fine).

Comment. In Python, we can therefore implement an iteration of bisection as follows:

```
>>> # we start with the interval [a, b]
    c = (a + b) / 2
    if f(a)*f(c) < 0:
        b = c  # pick [a, c] as the next interval
    else:
        a = c  # pick [c, b] as the next interval
```

Even if you have never used/seen such an if-then-else statement before, the above code can be read almost as an English sentence. Note how we indent things after `if` and after `else` (the `else` part can be omitted if we only want to do something if a condition is true) to group the code that we want to be executed in each case.

Advanced comment. One potential issue with using the product $f(a)f(c)$ rather than directly comparing the signs is that, depending on our available precision, $f(a)f(c)$ might run into an underflow (note that $f(a)$ and $f(c)$ are each getting smaller by construction) and be treated as zero. Here is a simple artificial example illustrating the issue:

```
>>> def f(x):
        return (x-1)**99

>>> f(0.99) < 0

   True

>>> f(1.01) > 0

   True

>>> f(0.99) * f(1.01) < 0

   False
```

With the representation of floats and their precision in mind, explain the above output!

When writing serious code, we therefore have to account for this and should instead compare the sign of $f(a)$ to the sign of $f(b)$ (and not compute the product). We will ignore this issue in the sequel.

Example 36. Python Let us implement the bisection method in Python (using the observation from the previous example). As input, we use a function f, the end points a and b of an interval guaranteed to contain a root of f, and the number of iterations that we wish to perform. The output is the final interval.

```
>>> def bisection(f, a, b, nr_steps):
        for i in range(nr_steps):
            c = (a + b) / 2
            if f(a)*f(c) < 0:
                b = c
            else:
                a = c
        return [a, b]
```

In order to use this function, we need to define a function $f(x)$. For comparison with our computation in Example 33, let us define $f(x) = x^3 - 2$. We can call it anything.

```
>>> def my_f(x):
        return x**3 - 2
```

Let us verify (always check simple examples when writing code!) the definition of $f(x)$ by computing the values for $x = 1$ and $x = 2$.

```
>>> my_f(1)

   -1

>>> my_f(2)

   6
```

We are now ready to perform bisection with this function on the interval $[a, b]$ with $a = 1$ and $b = 2$.

```
>>> bisection(my_f, 1, 2, 1)

   [1, 1.5]

>>> bisection(my_f, 1, 2, 2)

   [1.25, 1.5]
```

This matches our computation in Example 33. We previously computed that after 20 iterations the interval will have size less than 10^{-6}. Indeed, the following illustrates that we

```
>>> bisection(my_f, 1, 2, 20)

   [1.2599201202392578, 1.2599210739135742]
```

Accordingly, we obtain the approximation $\sqrt[3]{2} = 1.259920\ldots$.

Example 37. Python Our computation in the previous example automatically ended up using floats (we started with integer values for a and b but we end up with floats after dividing by 2 for the midpoint). To work with fractions (no rounding!) in Python, we can use the `fractions` module as follows.

```
>>> from fractions import Fraction
>>> Fraction(1, 2) + Fraction(1, 3)
   5/6
```

[You will probably see `Fraction(5, 6)` as the output rather than the above prettified version.]

Remarkably, our code can be used with fractions without changing it in any way:

```
>>> bisection(my_f, Fraction(1), Fraction(2), 1)
   [Fraction(1, 1), Fraction(3, 2)]
>>> bisection(my_f, Fraction(1), Fraction(2), 2)
   [Fraction(5, 4), Fraction(3, 2)]
```

Now, this perfectly matches our computation in Example 33.

Advanced comment. Unlike some other programming languages, Python does not make us specify the type of variables. For instance, we never had to tell Python whether the variables `a` and `b` should be an integer or a float or something else. Instead, Python is flexible (and we just used that to our advantage). This is called **duck typing** (true to the idiom that *if it walks like a duck and it quacks like a duck, then it must be a duck*). As such, Python allows the variables `a` and `b` to be any type for which our code (for instance, `(a + b) / 2`) makes sense.

[As always, these features have advantages and disadvantages. For instance, disadvantages of duck typing are speed and safety (as in making problematic kinds of bugs more likely).]

Finally, let us compute all 4 iterations of Example 33 at once:

```
>>> [bisection(my_f, Fraction(1), Fraction(2), n) for n in range(1,5)]
   [[Fraction(1, 1), Fraction(3, 2)], [Fraction(5, 4), Fraction(3, 2)], [Fraction(5, 4),
   Fraction(11, 8)], [Fraction(5, 4), Fraction(21, 16)]]
```

Example 38. Python Note that our bisection method code in Example 36 evaluates the function f twice per iteration (because we compute $f(a)$ and $f(c)$). Rewrite our code to only require one evaluation of f per iteration.

Solution.

```
>>> def bisection(f, a, b, nr_steps):
        fa = f(a)
        for i in range(nr_steps):
            c = (a + b) / 2
            fc = f(c)
            if fa*fc < 0:
                b = c
            else:
                a = c
                fa = fc
        return [a, b]
```

Try it! In terms of output, it should behave exactly as our previous code.

So why is this an improvement? (At first glance, it just looks more complicated...) Well, we need to keep in mind that the function f could potentially be very costly to evaluate (f doesn't have to be a simple function as it was in Example 33; instead, the definition of f might be a long computer program in itself and might require things like querying databases in a complicated way). In such a case, this small detail might make a huge difference.

The regula falsi method

As before, we wish to find a root of the continuous function $f(x)$ in the interval $[a,b]$.

The **regula falsi method** proceeds like the bisection method.

However, as an attempt to improve our approximations, instead of using the midpoint $\frac{a+b}{2}$ of the interval $[a,b]$, it uses the root of the secant line of $f(x)$ through $(a,f(a))$ and $(b,f(b))$.

Comment. Note that the root of the secant line will be a good approximation of the root of $f(x)$ if $f(x)$ is nearly linear on the interval.

Example 39. Derive a formula for the root of the line through $(a,f(a))$ and $(b,f(b))$.

Solution. The line has slope $m=\frac{f(b)-f(a)}{b-a}$.

Since it passes through $(a,f(a))$, the line has the equation $y-f(a)=m(x-a)$. (We could, of course, also pick the other point and the final result will be the same.)

To find its root (or x-intercept), we set $y=0$ and solve for x.

We find that the root is $x=a-\frac{f(a)}{m}=a-\frac{(b-a)f(a)}{f(b)-f(a)}=\frac{a(f(b)-f(a))}{f(b)-f(a)}-\frac{(b-a)f(a)}{f(b)-f(a)}=\frac{af(b)-bf(a)}{f(b)-f(a)}$.

Comment. Note that the final formula $\frac{af(b)-bf(a)}{f(b)-f(a)}$ for the root is symmetric in a and b. (As it must be!)

Historical comment. Regula falsi (also called *false position method*) derives its somewhat strange name from its long history where the above formula (in a time where formulas in the modern sense were not yet a thing) was used to solve linear equations $f(x)=Ax+B=0$ (where A and B typically wouldn't be known explicitly) by choosing two convenient values $x=a$ and $x=b$ (these would be the *false positions* since they are not the root itself).

https://en.wikipedia.org/wiki/Regula_falsi

To cut each interval $[a,b]$ into the two parts $[a,c]$ and $[c,b]$:

- Bisection uses the midpoint $c=\frac{a+b}{2}$.
- Regula falsi uses $c=\frac{af(b)-bf(a)}{f(b)-f(a)}$.

Example 40. Determine an approximation for $\sqrt[3]{2}$ by applying the regula falsi method to the function $f(x)=x^3-2$ on the interval $[1,2]$. Perform 3 steps.

Solution. Note that $f(1)=-1<0$ while $f(2)=6>0$. Hence, $f(x)$ must indeed have a root in the interval $[1,2]$.

- $[a,b]=[1,2]$ contains a root of $f(x)$ (since $f(a)<0$ and $f(b)>0$).
  The regula falsi point is $c=\frac{af(b)-bf(a)}{f(b)-f(a)}=\frac{8}{7}$. We compute $f(c)=-\frac{174}{343}<0$.
  Hence, $[c,b]=\left[\frac{8}{7},2\right]$ must contain a root of $f(x)$.
- $[a,b]=\left[\frac{8}{7},2\right]$ contains a root of $f(x)$ (since $f(a)<0$ and $f(b)>0$).
  The regula falsi point is $c=\frac{af(b)-bf(a)}{f(b)-f(a)}=\frac{75}{62}$. We compute $f(c)=-\frac{54781}{238,328}<0$.
  Hence, $[c,b]=\left[\frac{75}{62},2\right]$ must contain a root of $f(x)$.
- $[a,b]=\left[\frac{75}{62},2\right]$ contains a root of $f(x)$ (since $f(a)<0$ and $f(b)>0$).
  The regula falsi point is $c=\frac{af(b)-bf(a)}{f(b)-f(a)}=\frac{37,538}{30,301}$. We compute $f(c)<0$.
  Hence, $[c,b]=\left[\frac{37,538}{30,301},2\right]$ must contain a root of $f(x)$.

After 3 steps of the regula falsi method, we know that $\sqrt[3]{2}$ must lie in $\left[\frac{37,538}{30,301},2\right]\approx[1.2388,2]$.

Comment. For comparison, $\sqrt[3]{2}\approx 1.2599$.

Example 41. Python We can easily adjust our code from Example 36 for the bisection method to handle the regula falsi method. We only need to change the line that previously computed the midpoint $c = \frac{a+b}{2}$ and replace it with $c = \frac{a\,f(b) - b\,f(a)}{f(b) - f(a)}$ instead:

```
>>> def regulafalsi(f, a, b, nr_steps):
        for i in range(nr_steps):
            c = (a*f(b) - b*f(a)) / (f(b) - f(a))
            if f(a)*f(c) < 0:
                b = c
            else:
                a = c
        return [a, b]
```

Comment. This code is using 6 function evaluations per iteration. As we did in the case of the bisection method, rewrite the code to only use a single function evaluation per iteration.

Let us use this code to automatically perform the computations we did in Example 40. Again, we use fractions to get exact values for easier comparison.

```
>>> def my_f(x):
        return x**3 - 2

>>> from fractions import Fraction

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 1)

   [Fraction(8, 7), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 2)

   [Fraction(75, 62), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 3)

   [Fraction(37538, 30301), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 4)

   [Fraction(1534043307, 1226096954), Fraction(2, 1)]

>>> regulafalsi(my_f, Fraction(1), Fraction(2), 5)

   [Fraction(1523674852078629624 2, 12128315482217382469), Fraction(2, 1)]
```

No wonder that we stopped after 3 iterations by hand...

Important comment. The final two outputs give us an indication why performance-critical scientific computations are usually done using floats even if all involved quantities could be exactly represented as rational numbers. Compute the next iteration! The numerator and denominator integers can no longer be stored as 64 bit integers. And this after a measly 6 iterations of a simple algorithm!

This is a typical problem with any exact expressions. In practice, their complexity (the number of bits required to store them) often grows too fast.

Example 42. In Example 40, which we continued in the previous example, only the left point ever got updated. Will that always be the case? Explain!

Solution. For $f(x)=x^3-2$, we have $f'(x)=3x^2$ and $f''(x)=6x$.

Therefore we have $f'(x)>0$ as well as $f''(x)>0$ for all $x\in[1,2]$. This means that our function is increasing as well as concave up (on the interval $[1,2]$).

Because it is concave up, its graph will always lie below the secant lines we construct (see below).

Combined with the function being increasing, the regula falsi points (roots of the secant lines) will always be to the left of the true root (here $\sqrt[3]{2}$).

Accordingly, the right endpoint will never get updated.

Review of concavity. Recall that a function $f(x)$ is **concave up** (like any part of a parabola opening upward) at $x=c$ if $f''(c)>0$. At such a point, the graph of the function lies above the tangent line (at least locally). Make a sketch! On the other hand, this means that for sufficiently small intervals $[a,b]$ around c, the graph will lie below the secant line through $(a,f(a))$ and $(b,f(b))$.

Let us note the following differences between bisection and regula falsi:

- The intervals produced by bisection shrink by a factor of $1/2$ per iteration. On the other hand, the intervals produced by regula falsi usually don't drop below a certain length.

  For instance. This is illustrated by Example 40. In that case, the generated intervals are all of the form $[a,2]$ where the left side a approaches $\sqrt[3]{2}$ from below. In particular, these intervals will always have length larger than $2-\sqrt[3]{2}\approx 0.74$.

- Despite this, the sequence c_n of "new" interval endpoints produced by regula falsi can be shown to always converge to a root. Often the approximations c_n converge faster than the approximations obtained through bisection, but it can also be the other way around.

  Comment. There are, however, variations of regula falsi that are more reliably faster than bisection.

- Bisection is guaranteed to converge to a root at a certain rate (namely, one bit per iteration). Regula falsi frequently but not always converges faster, but we cannot guarantee a certain rate (this depends on the involved function $f(x)$).

Example 43. Suppose we use bisection or regula falsi to compute a root of some function. Several iterations result in the intervals $[a_1,b_1],[a_2,b_2],\ldots,[a_n,b_n]$. Based on these intervals, what is our approximation of the root?

(a) In the case of bisection.

(b) In the case of regula falsi.

Solution.

(a) In the case of bisection, the generically best choice for our approximation is the midpoint of the final interval $(a_n+b_n)/2$.

(b) In the case of regula falsi, our approximation is the "new" endpoint of the final interval. More precisely, the approximation is a_n if $b_n=b_{n-1}$ and it is b_n if $a_n=a_{n-1}$.

Armin Straub
straub@southalabama.edu

Secant method

The **secant method** for computing a root of a function $f(x)$ is a modification of regula falsi where we do not try to bracket the root (in other words, we do not produce intervals that are guaranteed to contain the root).

Instead, starting with two initial approximations x_0 and x_1, we construct $x_2, x_3, \ldots$ by the rule

$$x_{n+1} = \frac{x_{n-1}f(x_n) - x_n f(x_{n-1})}{f(x_n) - f(x_{n-1})}.$$

Comment. In other words, if $a = x_{n-1}$ and $b = x_n$ are the two most recent approximations, then the next approximation is

$$x_{n+1} = c = \frac{a f(b) - b f(a)}{f(b) - f(a)},$$

and, as in regula falsi, this is the root of the secant line through $(a, f(a))$ and $(b, f(b))$. While regula falsi next determines whether to continue with the interval $[a, c]$ or with $[c, b]$, the secant method always continues with b and c as the next pair of approximations (in particular, the root does not need to lie between b and c).

Advanced comment. The formula for x_{n+1} is somewhat problematic because it is prone to round-off errors. Namely, if x_n converges to a root of $f(x)$, then in both the numerator and denominator of that formula we are subtracting numbers of almost equal value. This can result in damaging loss of precision.

Why is this not an issue for regula falsi? (Hint: What do we know about the signs of $f(a)$ and $f(b)$?)

Example 44. Determine an approximation for $\sqrt[3]{2}$ by applying the secant method to the function $f(x) = x^3 - 2$ with initial approximations $x_0 = 1$ and $x_1 = 2$. Perform 3 steps.

Solution.

- $x_2 = \frac{x_0 f(x_1) - x_1 f(x_0)}{f(x_1) - f(x_0)} = \frac{8}{7} \approx 1.1429$
- $x_3 = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)} = \frac{75}{62} \approx 1.2097$
- $x_4 = \frac{x_2 f(x_3) - x_3 f(x_2)}{f(x_3) - f(x_2)} = \frac{989{,}312}{782{,}041} \approx 1.2650$

After 3 steps of the secant method, our approximation for $\sqrt[3]{2}$ is $\frac{989{,}312}{782{,}041} \approx 1.2650$.

Comment. For comparison, $\sqrt[3]{2} \approx 1.2599$.

Comment. Note that the interval $[x_2, x_3]$ does not contain the root $\sqrt[3]{2}$.

Compare Example 44 to Example 40 where we used regula falsi instead (note how the first two iterations resulted in the same approximations). In operational terms, the secant method is a simpler version of regula falsi since we are not trying to determine an interval that is guaranteed to contain a root.

It may therefore come as a surprise that the secant method typically converges considerably faster than regula falsi. However, we no longer have a guarantee of convergence (and the situation in general depends on the initial approximations as well as the function $f(x)$).

Example 45. Python The following is code for performing a fixed number of iterations of the secant method. Note that the code is a (simpler) variation of our code in Example 41 for the regula falsi method.

```
>>> def secant_method(f, a, b, nr_steps):
        for i in range(nr_steps):
            c = (a*f(b) - b*f(a)) / (f(b) - f(a))
            a = b
            b = c
        return b
```

Comment. This time, we return only a single value which is an approximation to the desired root. (Recall that the secant method does not provide intervals containing the true root.)

As before, let us use this code to automatically perform the computations from Example 44.

```
>>> def my_f(x):
        return x**3 - 2

>>> from fractions import Fraction

>>> [secant_method(my_f, Fraction(1), Fraction(2), n) for n in range(1,4)]

   [Fraction(8, 7), Fraction(75, 62), Fraction(989312, 782041)]
```