Notes for Lecture 1

Tue, 8/16/2022

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:

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

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

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

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? $49 = (100)_7$.

Example 4. `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 need to 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 $49 \equiv 1 \pmod 3$).

Fixed-point numbers

Example 5. (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) $(11001)_2 = 1 + 8 + 16 = 25$

(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 $2.625 = (10.101)_2$

Example 6. 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).

Armin Straub
straub@southalabama.edu

Example 7. 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 8. (extra) 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: $35/6 = (101.110101010\cdots)_2$

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

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)

In each case, 1 bit is used for the sign. Also, recall that the significand is preceded by an implied bit equal to 1.

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. $4.5 = 1.125 \cdot 2^2 = +\underbrace{1.001}_{\text{binary}} \cdot 2^2$

Next time, we will see exactly how IEEE 754 (single precision) would store this as 32 bits.

Example 10. `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 15 decimal digits after the 0.3, which translates to about $15 \cdot \log_2 10 \approx 50$ binary digits).

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(1.1)` or `print(1/3)` to receive the above output. When using Replit, you can either run a script (with code written on the left side) or you can enter single lines of code in an interactive Python interpreter (on the right side; this is also where you see the output from running a script).

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).

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, 8 bit for exponent)
  IEEE 754, double precision: 64 bit (1 bit for sign, 52 bit for significand, 11 bit for exponent)

Example 11. (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} where M is the number of bits used for the exponent. (Make sure you see how enormous of a difference this is!)

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).

Example 12. Give an example where one should not use floats.

Solution. Most notably, one should not use floats when dealing with money. That is because, as we saw earlier, 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 very 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).

Recall that a floating-point number (with base 2) is of the form $\pm 1.x \cdot 2^y$ where $1.x$ is the significand and y the exponent. IEEE 754 offers the following choices:

- **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)

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). Namely, one stores $x + \text{bias}$ where $\text{bias} = 2^7 - 1 = 127$ for single precision ($\text{bias} = 2^{10} - 1$ for double precision).

Armin Straub
straub@southalabama.edu

Example 13. Represent 4.5 as a single precision floating-point number according to IEEE 754.

Solution. $4.5 = 1.125 \cdot 2^2 = \boxed{+}1.\underbrace{\boxed{001}}_{\text{binary}} \cdot 2^2$

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

Solution. In a previous example, we computed that $0.1 = (0.0001100110011\cdots)_2$.

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

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

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

   False

>>> 0.1 + 0.1 + 0.1

   0.30000000000000004
```

Solution. As we saw in the previous example, 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
```

Example 16. `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 next time.

```
>>> 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 next time, as well as discuss several other important improvements.

Interlude: Representing negative integers

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

- **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.
  Typically, if we use r bits, then 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:** If using r bits, the most significant bit is counted as -2^{r-1} instead of 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, while -3 is stored as 1101. 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).

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 17. 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 18. 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.

Another floating-point issue

Example 19. 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 $53/\log_2 10 \approx 16$ decimal digits. However, storing $10^9 + 10^{-9}$ exactly requires 20 decimal digits.

[Recall that, if $2^{53} = 10^r$, then $r = \log_{10}(2^{53}) = 53\log_{10}(2) = 53/\log_2 10$.]

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 20. 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 21. (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 $\left|\frac{93{,}648 - \pi^{10}}{\pi^{10}}\right| \approx 5.07 \cdot 10^{-7}$.

Coding in Python: binary digits of 0.1

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

Example 22. 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 23. 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 24. 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.

Example 25. 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 discovered that

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

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 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

Example 26. 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 $\left|\frac{20-(e^\pi-\pi)}{e^\pi-\pi}\right|\approx 4.5\cdot 10^{-5}$.

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 27. 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.

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 28. (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.

Example 29. 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:

```
>>> # starting with the interval [a, b]
    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.

Armin Straub
straub@southalabama.edu

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 30. 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 27, 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 27.

Example 31. 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 27.

Advanced comment. Unlike in 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 27 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 32. Python Note that our bisection method code in Example 30 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 27; 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 33. Derive a formula for the root of the line through $(a,f(a))$ and $(b,f(b))$.

Solution. The line has slope $m=\frac{f(b)-f(a)}{b-a}$. (We could also pick the other point.)

Since it passes through $(a,f(a))$, the line has the equation $y-f(a)=m(x-a)$.

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 $f(x)=Ax+B=0$ 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 34. 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 35. Python We can easily adjust our code from Example 30 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 34. 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(1523674852078629 6242, 1212831548221738 2469), 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 36. In Example 34, 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 length intervals produced by regula falsi usually don't drop below a certain length.

  For instance. This is illustrated by Example 34. 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 37. 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 $a_n = a_{n-1}$.

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 38. 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.

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

After 3 steps of the secant method, our approximation for $\sqrt[3]{2}$ is $\frac{989{,}312}{782{,}041} \approx 1.265$.

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

Compare Example 38 to Example 34 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 39. Python The following is code for performing a fixed number of iterations of the secant method. Note that the code is a simplified version of our code in Example 35 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 38.

```
>>> 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)]
```

Newton's method

The **Newton method** proceeds as the secant method, except that it uses tangents instead of secants. In particular, instead of two previous points x_{n-1}, x_n (so that we construct a secant line) we only require a single point x_n to compute the next point.

Example 40. Derive a formula for the root of the tangent line through $(a, f(a))$.

Solution. The line has slope $m = f'(a)$. (We could also pick the other point.)
Since it passes through $(a, f(a))$, the line has the equation $y - f(a) = m(x - a)$.
To find its root (or x-intercept), we set $y = 0$ and solve for x.
We find that the root is $x = a - \frac{f(a)}{m} = a - \frac{f(a)}{f'(a)}$.

Comment. Compare the above derivation with what we did for the regula falsi method.

Thus, given an initial approximation x_0, the Newton method constructs $x_1, x_2, \ldots$ by the rule

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

Comment. In contrast to the secant method, the Newton method requires us to be able to compute $f'(x)$. Also, per iteration we need two function evaluations (one for f and one for f') whereas the secant method only requires a single function evaluation.

Comment. If we use the approximation $f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$ (which is a good approximation if x_{n-1} and x_n are sufficiently close) in the Newton method, then we actually obtain the secant method.

Example 41. Determine an approximation for $\sqrt[3]{2}$ by applying Newton's method to the function $f(x)=x^3-2$ with initial approximation $x_0=1$. Perform 3 steps.

Solution. We compute that $f'(x)=3x^2$.

- $x_1=x_0-\dfrac{f(x_0)}{f'(x_0)}=\dfrac{4}{3}$
- $x_2=x_1-\dfrac{f(x_1)}{f'(x_1)}=\dfrac{91}{72}$
- $x_3=x_2-\dfrac{f(x_2)}{f'(x_2)}=\dfrac{1{,}126{,}819}{894{,}348}$

After 3 steps of Newton's method, our approximation for $\sqrt[3]{2}$ is $\frac{1{,}126{,}819}{894{,}348}\approx 1.259933$.

Comment. For comparison, $\sqrt[3]{2}\approx 1.259921$. The error is only 0.000012!

After one more iteration, the error drops to an astonishing $1.2\cdot 10^{-10}$.

After one more iteration, the error drops to an astonishing $1.2\cdot 10^{-20}$.

It looks like the number of correct digits is doubling at each step!!

We will soon prove that this is indeed the case.

Example 42. Python The following code implements the Newton method specifically for computing a root of $f(x)=x^3-2$ as in Example 41 (cr2 is meant to be short for cube root of 2).

```
>>> def newton_cr2(x0, nr_steps):
        for i in range(nr_steps):
            x0 = x0 - (x0**3-2)/(3*x0**2)
        return x0
```

Let us use this code to automatically perform the computations from Example 41.

```
>>> from fractions import Fraction
>>> [newton_cr2(Fraction(1), n) for n in range(1,4)]
   [Fraction(4, 3), Fraction(91, 72), Fraction(1126819, 894348)]
```

Next, let us compare these values to $\sqrt[3]{2}$, the actual root of $f(x)$. Note that, if x is a (close) approximation of $\sqrt[3]{2}$, then the number of correct binary digits of x is $\lfloor -\log_2|x-\sqrt[3]{2}|\rfloor$. The following function uses this to compute the correct number of bits after a certain number of steps of the Newton method:

```
>>> from math import log2
>>> def newton_cr2_correctbits(x0, nr_steps):
        return -log2(abs(2**(1/3) - newton_cr2(x0, nr_steps)))
>>> [newton_cr2_correctbits(Fraction(1), n) for n in range(1,5)]
   [3.7678347129038254, 7.977430799070329, 16.294241754402005, 32.92183615918938]
```

Comment. What about the number of correct bits after 5 steps (one more step)? If you run our code above, you will receive an error (`ValueError: math domain error`). Can you explain why?

Well, we expect about 64 correct digits. That is more than the number of significant digits that can be stored in a double-precision float. Accordingly, the error is going to be rounded down to 0. We then run into trouble because we ask for the logarithm of 0 (which is a singularity).

Example 43. Apply Newton's method to $g(x)=x^3-2x+2$ and initial value $x_0=0$.

Solution. Using $g'(x)=3x^2-2$, we compute that $x_1=x_0-\frac{g(x_0)}{g'(x_0)}=1$, $x_2=x_1-\frac{g(x_1)}{g'(x_1)}=1-\frac{1}{1}=0$.

Since $x_2=x_0$, the Newton method will now repeat and we are stuck in a 2-cycle.

In particular, the Newton method does not converge in this case.

Comment. It is possible to run into n-cycles for larger n as well when doing Newton iterations (for instance, try $f(x)=x^5-x-1$ and initial value $x_0=0$). When computing numerically, it is not particularly likely that we will run into a perfect cycle. However, such cycles can be **attractive**. Meaning that we get closer and closer to the cycle if we start with a nearby point. This is illustrated by the Python code experiment below.

Example 44. Python The following code implements the Newton method specifically for computing a root of $g(x)=x^3-2x+2$ as in the previous example.

```
>>> def newton_g(x0, nr_steps):
        for i in range(nr_steps):
            x0 = x0 - (x0**3-2*x0+2)/(3*x0**2-2)
        return x0
```

The following confirms that we have a 2-cycle starting with 0:

```
>>> [newton_g(0, n) for n in range(8)]

   [0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0]
```

On the other hand, this is what happens if we start with a point close to 0:

```
>>> [newton_g(0.1, n) for n in range(8)]

   [0.1, 1.0142131979695432, 0.07965576631987636, 1.0090987403727651, 0.05222652653371296,
   1.0039651847274838, 0.02332943565497303, 1.0008043531824031]
```

Notice how we are being attracted by the 2-cycle.

Review: Taylor series

Recall from Calculus that, if $f(x)$ is **analytic** at $x=c$, then

$$f(x) \quad = \quad \sum_{n=0}^{\infty} \frac{f^{(n)}(c)}{n!}(x-c)^n = f(c)+f'(c)(x-c)+\frac{1}{2}f''(c)(x-c)^2+\ldots$$

The series on the right-hand side is called the **Taylor series** of $f(x)$ at $x=c$.

Advanced comment. We only get the equality between $f(x)$ and its Taylor series for functions that are **analytic** at $x=c$ (by definition, these are functions than can be expanded as a power series; the above makes it explicit what the coefficients of that power series have to be). Fortunately, all elementary functions (the ones we can express as algebraic expressions with exponentials, logarithms and trig functions) are analytic at almost all points.

On the other hand, for instance, the Taylor series for the function $f(x)=e^{-1/x^2}$ at $x=0$ is zero (because all derivatives of $f(x)$ are zero for $x=0$) while $f(x)$ is not zero (however, note that $x=0$ is clearly a problematic point of the formula for $f(x)$; that function is analytic at all other points). Since $f(x)$ is infinitely differentiable, this illustrates that being analytic is a stronger property than being infinitely differentiable. For other functions, it is possible that the Taylor series might not converge at all.

Comment. The Taylor series of $f(x)$ at $x=0$ takes the form $f(0)+f'(0)x+\frac{1}{2}f''(0)x^2+\ldots$. In practice, we can often shift things so that the Taylor series of interest are around $x=0$.

Example 45. Determine the Taylor series of e^x at 0.

Solution. If $f(x)=e^x$, then $f^{(n)}(x)=e^x$. In particular, we have $f^{(n)}(0)=1$ for all n.

Consequently, we have $e^x=\sum_{n=0}^{\infty}\frac{x^n}{n!}=1+x+\frac{1}{2}x^2+\frac{1}{6}x^3+\ldots$

Comment. e^x is analytic everywhere and so it equals its Taylor series.

Python assignment #2

Example 46. `Python` In Example 35 we implemented the regula falsi method. As we have observed, a weakness of this method is that we typically end up only updating one endpoint of the interval. The **Illinois algorithm** is an extension of the regula falsi method that works to remedy this issue.

Recall that the regula falsi method uses $c = \frac{a f_b - b f_a}{f_b - f_a}$ with $f_a = f(a)$ and $f_b = f(b)$ to cut each interval $[a, b]$ into the two parts $[a, c]$ and $[c, b]$.

The Illinois algorithm proceeds likewise but, after an endpoint has been retained for a second time, the corresponding value f_a or f_b is replaced with half its value. In other words, if a was not updated in this or the previous step, then f_a (to be used in the next iteration) is replaced with $f_a/2$; likewise, if b was not updated in this or the previous step, then f_b is replaced with $f_b/2$.

```
# start with an interval [a,b]
fa = f(a)
fb = f(b)
repeat
  # compute the regula falsi point
  c = (a*fb - b*fa) / (fb - fa)
  fc = f(c)
  # set new interval [a,b] according to signs of f
  ...
  if left endpoint was also updated the previous time
    fb = fb/2
  if right endpoint was also updated the previous time
    fa = fa/2
```

Here is one approach that we can take:

- Start with the code that we wrote in class for the regula falsi method.
- Adjust that code (like we did for the bisection method) to only use one function evaluation per iteration. Do that by introducing variables `fa`, `fb`, `fc` for the values of $f(x)$ at $x = a$, b, c.
- Add a new variable to your code that keeps track of whether we most recently changed the left or the right endpoint of the interval. You can, for instance, define a variable `updated_endpoint` that is initially set to 0, and which is set to 1 after the right endpoint is updated and to -1 after the left endpoint is updated.

  That way, if we are about to update, say, the left endpoint, then we can test whether `updated_endpoint` is -1 as that would tell us that we are now updating the left endpoint for a second time in a row. In that case, we set `fb = fb/2`.

Advanced comment. There are other, more clever, approaches to implementing the Illinois method. For instance, one could stop making a and b the left and right endpoints of the interval and, instead, always make b the newly added endpoint; then one can test whether we repeatedly change the same endpoint by looking at the signs of the corresponding values of f. This is done by M. Dowell and P. Jarratt in [*A Modified Regula Falsi Method for Computing the Root of an Equation*, 1971], where they describe and analyze the Illinois method. As a very minor point, their implementation might proceed slightly different from ours because we start with an interval $[a, b]$ whereas their implementation thinks of a and b as two approximations, with b being the more "recent" one (accordingly, their implementation might divide f_a by 2 already at the end of the first iteration).

Let us revisit the computations we did in Example 34 but with the regula falsi method updated to the Illinois algorithm. The first two iterations should result in the same intervals:

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

>>> from fractions import Fraction

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

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

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

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

However, at the end of the second iteration (since the right endpoint has not changed in this or the previous iteration), $f_b = 6$ (since $f(2) = 6$) is replaced with $f_b = 3$. As a result, in the third iteration, we end up replacing the right endpoint:

```
>>> illinois(my_f, Fraction(1), Fraction(2), 3)

   [Fraction(75, 62), Fraction(974462, 769765)]
```

For further testing, in the next two iterations we replace the left endpoints (since the fractions are becoming large, we are using floats below; note that the first command just repeats the above computation with floats):

```
>>> illinois(my_f, 1, 2, 3)

   [1.2096774193548387, 1.2659214175754938]

>>> illinois(my_f, 1, 2, 4)

   [1.2596760796087871, 1.2659214175754938]

>>> illinois(my_f, 1, 2, 5)

   [1.2599198867703156, 1.2659214175754938]
```

Consequently, at the end of the fifth iteration, the value of f_b (which is $f(1.2659\ldots)$) is again replaced by half its value. Once more, this results in b being updated in the next iteration:

```
>>> illinois(my_f, 1, 2, 6)

   [1.2599198867703156, 1.2599222015292841]
```

Review: Taylor series, continued

Review. If $f(x)$ is **analytic** around $x=c$, then it equals its **Taylor series** of $f(x)$ at $x=c$:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(c)}{n!}(x-c)^n = f(c) + f'(c)(x-c) + \frac{1}{2}f''(c)(x-c)^2 + \dots$$

Example 47. Determine the Taylor series of $\ln x$ at $x=1$.

Solution. If $f(x)=\ln(x)$, then $f'(x)=\frac{1}{x}$, $f''(x)=-\frac{1}{x^2}$, $f'''(x)=2\frac{1}{x^3}$, $f^{(4)}(x)=2\cdot 3\frac{1}{x^4}$, ...

For $n \geqslant 1$, we thus have $f^{(n)}(x)=(-1)^{n+1}\frac{(n-1)!}{x^n}$ and, in particular, $\frac{f^{(n)}(1)}{n!}=(-1)^{n+1}\frac{(n-1)!}{n!}=\frac{(-1)^{n+1}}{n}$.

Consequently, we have $\ln x = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}(x-1)^n$.

Comment. By replacing x with $1-x$, we obtain $\sum_{n=1}^{\infty} \frac{x^n}{n} = -\ln(1-x) = \ln\left(\frac{1}{1-x}\right)$.

If we take the derivative of both sides, we further find $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$. This is the famous **geometric series**.

The truncation $\sum_{n=0}^{M} \frac{f^{(n)}(c)}{n!}(x-c)^n$ is called the **Mth Taylor polynomial** of $f(x)$ at $x=c$.

Comment. The Mth Taylor polynomial is a polynomial of degree at most M (note that the degree can be smaller if $f^{(M)}(c)=0$).

Important comment. The first Taylor polynomial of $f(x)$ at $x=c$ is the tangent line of $f(x)$ at $x=c$. In other words, it is the best linear approximation of $f(x)$ at $x=c$.

Likewise, the Mth Taylor polynomial is the best polynomial approximation at $x=c$ of degree up to M.

We have the following fundamental result for what happens when we truncate a Taylor series.

Theorem 48. (Taylor's theorem with error term) Suppose that $f(x)$ is $M+1$ times continuously differentiable on the interval between x and c. Then we have

$$f(x) = \underbrace{\sum_{n=0}^{M} \frac{f^{(n)}(c)}{n!}(x-c)^n}_{M\text{th Taylor polynomial}} + \underbrace{\frac{f^{(M+1)}(\xi)}{(M+1)!}(x-c)^{M+1}}_{\text{error term}}$$

for some ξ between x and c.

Advanced comment. We only need that $f(x)$ is $M+1$ times differentiable and that $f^{(M)}(x)$ is continuous.

The special case $M=0$ of Taylor's theorem is equivalent to the mean value theorem:

Theorem 49. (mean value theorem) Suppose that $f(x)$ is differentiable on $[a,b]$. Then there exists $\xi \in (a,b)$ such that

$$f'(\xi) = \frac{f(b)-f(a)}{b-a}.$$

Proof. Make a picture! □

Armin Straub
straub@southalabama.edu

Note that Taylor's theorem provides us with a representation for the error when we approximate $f(x)$ with a Taylor polynomial. This is illustrated in the next example.

Example 50. Suppose we use the approximation $e^x \approx 1 + x + \frac{x^2}{2}$.

(a) Using Taylor's theorem, provide an upper bound for the error on the interval $[0, 1]$.

(b) Using Taylor's theorem, provide an upper bound for the error on the interval $[0, 0.1]$.

(c) Using Taylor's theorem, how many terms of the Taylor series do we need so that the error on $[0, 0.1]$ is less than 10^{-16}?

Solution. Note that $1 + x + \frac{x^2}{2}$ is the 2nd Taylor polynomial of e^x at $x = 0$.

(a) Taylor's theorem implies that

$$e^x - \left(1 + x + \frac{x^2}{2}\right) = \frac{e^\xi}{3!}x^3$$

for some ξ between 0 and x.
Note that $|e^\xi| \leqslant e$ for all $\xi \in [0, 1]$.
On the other hand, $|x^3| \leqslant 1$ for all $x \in [0, 1]$.
We therefore conclude that the error on the interval $[0, 1]$ is bounded by

$$\left|e^x - \left(1 + x + \frac{x^2}{2}\right)\right| = \left|\frac{e^\xi}{3!}x^3\right| \leqslant \frac{e}{3!} \approx 0.453.$$

Comment. In this simple case, we can determine the maximal error exactly (without using Taylor's theorem). Since the function $e^x - \left(1 + x + \frac{x^2}{2}\right)$ is increasing on the interval $[0, 1]$, starting with the value 0, the maximal error must occur at $x = 1$ and is $e - \frac{5}{2} \approx 0.218$. We thus find that our earlier error bound was a bit conservative but not a bad upper bound.

(b) As above, we conclude that the error on the interval $[0, 0.1]$ is bounded by

$$\left|e^x - \left(1 + x + \frac{x^2}{2}\right)\right| = \left|\frac{e^\xi}{3!}x^3\right| \leqslant \frac{e^{0.1}}{3!}0.1^3 \approx 0.000184 = 1.84 \cdot 10^{-4}.$$

Comment. For comparison, as above, the maximal actual error is $1.71 \cdot 10^{-4}$.

(c) By Taylor's theorem,

$$|e^x - p_M(x)| = \left|\frac{e^\xi}{(M+1)!}x^{M+1}\right| \leqslant \frac{e^{0.1}}{(M+1)!}0.1^{M+1}.$$

We wish to choose M so that the right-hand side is less than 10^{-16}. Since the right-hand side decreases very rapidly, we simply increase M until that happens:

$$\frac{e^{0.1}}{3!}0.1^3 \approx 1.8 \cdot 10^{-4}, \quad \ldots, \quad \frac{e^{0.1}}{9!}0.1^9 \approx 3.0 \cdot 10^{-15}, \quad \frac{e^{0.1}}{10!}0.1^{10} \approx 3.0 \cdot 10^{-17}.$$

We conclude that the 9th Taylor polynomial will approximate e^x in such a way that the error on $[0, 0.1]$ is less than 10^{-16}.

Fixed-point iteration

Definition 51. x^* is a **fixed point** of a function $f(x)$ if $f(x^*)=x^*$.

Example 52. Determine all fixed points of the function $f(x)=x^3$.

Solution. $x^3=x$ has the three solutions $x^*=0,\pm 1$ (and a cubic equation cannot have more than 3 solutions). These are the fixed points.

Idea. Suppose x^* is a fixed point of a continuous function f. If $x_n\approx x^*$, then $f(x_n)\approx f(x^*)=x^*\approx x_n$. If we can guarantee that $f(x_n)$ is closer to x^* than x_n, then we can set

$$x_{n+1}=f(x_n),$$

with the expectation that iterating this process will bring us closer and closer to x^*.

When does this converge? This process converges if $|f(x_n)-x^*|<|x_n-x^*|$ for all x_n close to x^*.

This condition is equivalent to $\left|\frac{f(x_n)-x^*}{x_n-x^*}\right|<1$.

Since $x^*=f(x^*)$, we have $\frac{f(x_n)-x^*}{x_n-x^*}=\frac{f(x_n)-f(x^*)}{x_n-x^*}\approx f'(x^*)$ provided that x_n is sufficiently close to x^*.

This essentially proves the following result. (See below for a full proof using the mean value theorem.)

Theorem 53. Suppose that x^* is a fixed point of a continuously differentiable function f. If $|f'(x^*)|<1$, then **fixed-point iteration**

$$x_{n+1}=f(x_n),\quad x_0=\text{initial approximation},$$

converges to x^* locally.

In that case, we say that x^* is an **attracting fixed point**.

Divergence. If $|f'(x^*)|>1$, then x^* is a **repelling fixed point**. Our argument shows that fixed-point iteration will not converge to x^* except in the "freak" case where $x_n\not\approx x^*$ but $f(x_n)=x^*$.

Comment. Local convergence means that we have convergence for all initial values x_0 close enough to x^*.

Proof. Note that

$$\begin{aligned} x_{n+1}-x^* &= g(x_n)-g(x^*) \\ &= g'(\xi_n)(x_n-x^*) \end{aligned}$$

where we applied the mean value theorem for the second equation and where ξ_n is between x_n and x^*. Thus

$$|x_{n+1}-x^*|=|g'(\xi_n)|\cdot|x_n-x^*|$$

Since g' is continuous and $|g'(x^*)|<1$, we have $|g'(x)|<\delta$ for some $\delta<1$ for all x sufficiently close to x^*. If x_0 is sufficiently to x^* in that sense, then it follows that $|x_1-x^*|<\delta\cdot|x_0-x^*|$. In particular, x_1 is even closer to x^* and we can repeat this argument to conclude that $|x_{n+1}-x^*|<\delta\cdot|x_n-x^*|$ for all n. This implies that $|x_n-x^*|<\delta^n\cdot|x_0-x^*|$. Since $\delta<1$, this further implies that x_n converges to x^*. □

Example 54. From a plot of $\cos(x)$, we can see that it has a unique fixed point in the interval $[0,1]$.

Solution. If $f(x)=\cos(x)$, then $f'(x)=-\sin(x)$. Since $|\sin(x)|<1$ for all $x\in[0,1]$, we conclude that $|f'(x^*)|<1$. By Theorem 53, fixed-point iteration will therefore converge to x^* locally.

Example 55. Python Let us implement the fixed-point iteration of $\cos(x)$ from the previous example in Python.

```
>>> from math import cos

>>> def cos_iterate(x, n):
        for i in range(n):
            x = cos(x)
        return x

>>> [cos_iterate(1, n) for n in range(20)]
```

```
[1, 0.5403023058681398, 0.8575532158463934, 0.6542897904977791, 0.7934803587425656,
0.7013687736227565, 0.7639596829006542, 0.7221024250267077, 0.7504177617637605,
0.7314040424225098, 0.7442373549005569, 0.7356047404363474, 0.7414250866101092,
0.7375068905132428, 0.7401473355678757, 0.7383692041223232, 0.7395672022122561,
0.7387603198742113, 0.7393038923969059, 0.7389377567153445]
```

Comment. Instead of using a loop, we could also implement the above fixed-point iteration **recursively** in the following way (the recursive part is that the function is calling itself).

```
>>> def cos_iterate_recursively(x, n):
        if n > 0:
            return cos_iterate_recursively(cos(x), n-1)
        return x

>>> [cos_iterate_recursively(1, n) for n in range(20)]
```

```
[1, 0.5403023058681398, 0.8575532158463934, 0.6542897904977791, 0.7934803587425656,
0.7013687736227565, 0.7639596829006542, 0.7221024250267077, 0.7504177617637605,
0.7314040424225098, 0.7442373549005569, 0.7356047404363474, 0.7414250866101092,
0.7375068905132428, 0.7401473355678757, 0.7383692041223232, 0.7395672022122561,
0.7387603198742113, 0.7393038923969059, 0.7389377567153445]
```

Sometimes recursion results in cleaner code. However the use of loops is usually more efficient.

Newton's method as a fixed-point iteration

Recall that Newton's method for finding a root of $f(x)$ proceeds from an initial approximation x_0 and iteratively computes

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

Note that this is equivalent to fixed-point iteration of the function $g(x)=x-\frac{f(x)}{f'(x)}$.

Comment. Note that x^* is a fixed point of $g(x)=x-\frac{f(x)}{f'(x)}$ if and only if $\frac{f(x^*)}{f'(x^*)}=0$.

We have already proven a criterion for convergence of fixed-point iterations (Theorem 53). Our next goal is to develop the tools to analyze the speed of that convergence.

Example 56.

(a) Newton's method applied to finding a root of $f(x)=x^3-2$ is equivalent to fixed-point iteration of which function $g(x)$?

(b) Determine whether Newton's method converges locally to $\sqrt[3]{2}$.

Solution.

(a) Newton's method applied to $f(x)$ is equivalent to fixed-point iteration of

$$g(x)=x-\frac{f(x)}{f'(x)}=x-\frac{x^3-2}{3x^2}=\frac{2}{3}\left(x+\frac{1}{x^2}\right).$$

(b) By Theorem 53, Newton's method converges locally to $x^*=\sqrt[3]{2}$ if $|g'(x^*)|<1$.

We compute that $g'(x)=\frac{2}{3}-\frac{4}{3x^3}$ so that $g'(x^*)=\frac{2}{3}-\frac{4}{3\cdot 2}=0$.

Hence Newton's method converges locally to $\sqrt[3]{2}$.

Important comment. Notice that $g'(x^*)=0$ is, in a way, the strongest sense in which $|g'(x^*)|<1$. We will see shortly that $g'(x^*)=0$ implies especially fast convergence of the type we observed in Example 41.

Example 57.

(a) What are the fixed points of $g(x)=\frac{x}{2}+\frac{1}{x}$?

(b) Does fixed-point iteration of $g(x)$ converge?

(c) Find a function $f(x)$ such that the fixed-point iteration of $g(x)$ is equivalent to Newton's method applied to $f(x)$.

(d) Inspired by the previous parts, suggest a fixed-point iteration to compute square roots.

Solution.

(a) Solving $\frac{x}{2}+\frac{1}{x}=x$, we find $x^2=2$ and thus $x=\pm\sqrt{2}$.

Comment. Note that $g(x)=\frac{1}{2}\left(x+\frac{2}{x}\right)$. Suppose that $x<\sqrt{2}$. Then $2/x>\sqrt{2}$.

When iterating $g(x)$, we are averaging the underestimate and the overestimate, and it is reasonable to expect that the result is a better approximation.

(b) Since $g'(x)=\frac{1}{2}-\frac{1}{x^2}$, we have $g'(\pm\sqrt{2})=\frac{1}{2}-\frac{1}{2}=0$. Hence, both fixed points are attracting fixed points.

By Theorem 53, fixed-point iteration of $g(x)$ converges locally to both fixed points.

(c) We are looking for a function $f(x)$ such that $x-\frac{f(x)}{f'(x)}=g(x)$. Equivalently, $\frac{f'(x)}{f(x)}=\frac{1}{x-g(x)}=\frac{2x}{x^2-2}$.

This is a first-order differential equation which we can solve for $f(x)$ using separation of variables or by realizing that it is a linear DE. (Our approach below is equivalent to separation of variables.)

Note that $\frac{f'(x)}{f(x)}=\frac{\mathrm{d}}{\mathrm{d}x}\ln(f(x))$. Thus, integrating both sides of the DE,

$$\ln(f(x)) \;=\; \int\frac{1}{x-g(x)}\mathrm{d}x=\int\frac{2x}{x^2-2}\mathrm{d}x=\ln|x^2-2|+C.$$

We conclude that fixed-point iteration of $g(x)$ is equivalent to Newton's method applied to $f(x)=x^2-2$.

Comment. The general solution of the DE has one degree of freedom (the C above, which we chose as 0). On the other hand, we know from the beginning that Newton's method applied to $f(x)$ and $Df(x)$ results in the same fixed-point iteration.

(d) Newton's method applied to $f(x)=x^2-a$ is equivalent to fixed-point iteration of $g(x)=\frac{1}{2}\left(x+\frac{a}{x}\right)$.

Comment. The resulting method for computing square roots $\sqrt{a}$ is known as the **Babylonian method**. It consists of starting with an approximation $x_0\approx\sqrt{a}$ and then iteratively computing $x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right)$.

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

Order of convergence

Example 58. Suppose that x_n converges to x^* in such a way that the number of correct digits doubles from one term to the next. What does that mean in terms of the error $e_n = |x_n - x^*|$?

Comment. This is roughly what we observed numerically for the Newton method in Example 41.

Comment. It doesn't matter which base we are using because the number of digits in one base is a fixed constant multiple of the number of digits in another base. Make sure that this clear! (If unsure, how does the number of digits of an integer x in base 2 relate to the number of digits of x in base 10?)

Solution. Recall that the number of correct digits in base b is about $-\log_b(e_n)$.
Doubling these from one term to the next means that $-\log_b(e_{n+1}) \approx -2\log_b(e_n)$.
Equivalently, $\log_b(e_{n+1}) - 2\log_b(e_n) = \log_b\left(\frac{e_{n+1}}{e_n^2}\right) \approx 0$.
This in turn is equivalent to $\frac{e_{n+1}}{e_n^2} \approx 1$.

What if the number of correct digits triples? By the above arguments, we would have $\frac{e_{n+1}}{e_n^3} \approx 1$.
Of course, there is nothing special about 2 or 3.

Example 59. Suppose that x_n converges to x^*. Let $e_n = |x_n - x^*|$ be the error and $d_n = -\log_b(e_n)$ be the number of correct digits (in base b). If $d_{n+1} = A d_n + B$, what does that mean in terms of the error e_n?

Solution. $-\log_b(e_{n+1}) = -A\log_b(e_n) + B$ is equivalent to $\log_b(e_{n+1}) - A\log_b(e_n) = \log_b\left(\frac{e_{n+1}}{e_n^A}\right) = -B$.
This in turn is equivalent to $\frac{e_{n+1}}{e_n^A} = b^{-B}$.

This motivates the following definition.

Definition 60. Suppose that x_n converges to x^*. Let $e_n = |x_n - x^*|$. We say that x_n **converges to x of order q and rate r** if

$$\lim_{n\to\infty} \frac{e_{n+1}}{e_n^q} = r.$$

Order 1. Convergence of order 1 is called **linear convergence**. As in the previous example, the rate r provides information on the number of additional correct digits per term.

Order 2. Convergence of order 2 is also called **quadratic convergence**. As we saw above, it means that number of correct binary digits d_n roughly doubles from one term to the next. More precisely, $d_{n+1} \approx 2d_n + B$ where the rate $r = 2^{-B}$ tells us that $B = -\log_2(r)$. [Note that r has the advantage of being independent of the base in which we measure the number of correct digits.]

Order of convergence of fixed-point iteration

Theorem 61. Suppose that x^* is a fixed point of a sufficiently differentiable function f. Suppose that $|f'(x^*)|<1$ so that, by Theorem 53, fixed-point iteration of $f(x)$ converges to x^* locally. Then the convergence is of order M with rate $\frac{1}{M!}|f^{(M)}(x^*)|$ where $M\geqslant 1$ is the smallest integer so that $f^{(M)}(x^*)\neq 0$.

In particular.

- If $f'(x^*)\neq 0$, then the convergence is linear with rate $|f'(x^*)|$.
- If $f'(x^*)=0$ and $f''(x^*)\neq 0$, then the convergence is quadratic with rate $\frac{1}{2}|f''(x^*)|$.

Comment. Here, sufficiently differentiable means that f needs to be M times continuously differentiable so that we can apply Taylor's theorem.

Proof. By Taylor's theorem (Theorem 48), if $f'(x^*)=f''(x^*)=\cdots=f^{(M-1)}(x^*)=0$ for some $M\geqslant 1$, then

$$f(x)=f(x^*)+\frac{1}{M!}f^{(M)}(\xi)(x-x^*)^M$$

for some ξ between x and x^*. It follows that

$$\begin{aligned} x_{n+1}-x^* &= f(x_n)-f(x^*) \\ &= \frac{1}{M!}f^{(M)}(\xi_n)(x_n-x^*)^M \end{aligned}$$

for some ξ_n between x_n and x^*.

Thus

$$\frac{x_{n+1}-x^*}{(x_n-x^*)^M}=\frac{1}{M!}f^{(M)}(\xi_n) \quad \xrightarrow[n\to\infty]{} \quad \frac{1}{M!}f^{(M)}(x^*),$$

where the limit follows from the continuity of $f^{(M)}(x)$ (and convergence of $x_n\to x^*$). □

Applying fixed-point iteration directly

Note that any equation $f(x)=0$ can be rewritten in many ways as a fixed-point equation $g(x)=x$.

For instance. We can always rewrite $f(x)=0$ as $f(x)+x=x$ (i.e. choose $g(x)=f(x)+x$).

We can then attempt to find a root x^* of $f(x)$ by fixed-point iteration on $g(x)$.

In other words, we start with a value x_0 (an initial approximation) and then compute $x_1,x_2,\ldots$ via $x_{n+1}=g(x_n)$.

Theorem 61 tells us whether that such a fixed-point iteration on $g(x)$ will locally converge to x^*. Moreover, it tells us the order of convergence.

Armin Straub
straub@southalabama.edu

Example 62. Suppose we are interested in computing the roots of $x^2 - x - 1 = 0$.

The roots are the golden ratio $\phi = \frac{1}{2}(1+\sqrt{5}) \approx 1.618$ and $\psi = \frac{1}{2}(1-\sqrt{5}) \approx -0.618$.

There are many ways to rewrite this equation as a fixed-point equation $g(x) = x$. The following are three possibilities:

(a) Rewrite as $x = x^2 - 1$, so that $g(x) = x^2 - 1$.

(b) Rewrite first as $x^2 = x + 1$ and then as $x = 1 + \frac{1}{x}$, so that $g(x) = 1 + \frac{1}{x}$.

(c) Rewrite first as $\underbrace{x^2 - x}_{=x(x-1)} = 1$ and then as $x = \frac{1}{x-1}$, so that $g(x) = \frac{1}{x-1}$.

In each of these three cases and for each root, decide whether fixed-point iteration converges. If it does, determine the order and rate of convergence.

Solution.

(a) In this case, we have $g(x) = x^2 - 1$ and $g'(x) = 2x$.
Since $|g'(\phi)| \approx 3.236 > 1$ as well as $|g'(\psi)| \approx 1.236 > 1$, fixed-point iteration does not converge locally to either root.

(b) In this case, we have $g(x) = 1 + \frac{1}{x}$ and $g'(x) = -\frac{1}{x^2}$.
Since $|g'(\phi)| = \frac{1}{\phi+1} \approx 0.382 < 1$ and $|g'(\psi)| = \phi + 1 \approx 2.618 > 1$, fixed-point iteration converges locally to ϕ but does not converge locally to ψ. Moreover, the convergence to ϕ is linear with rate 0.382.

(c) In this case, we have $g(x) = \frac{1}{x-1}$ and $g'(x) = -\frac{1}{(x-1)^2}$.
Since $|g'(\phi)| = \phi + 1 \approx 2.618 > 1$ and $|g'(\psi)| = \frac{1}{\phi+1} \approx 0.382 < 1$, fixed-point iteration converges locally to ψ but does not converge locally to ϕ. Moreover, the convergence to ψ is linear with rate 0.382.

Order of convergence of Newton's method

Recall that computing a root x^* of $f(x)$ using Newton's method is equivalent to fixed-point iteration of $g(x) = x - \frac{f(x)}{f'(x)}$.

Comment. In each case, we start with x_0 and iteratively compute $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Theorem 63. Suppose that f is twice continuously differentiable and $f(x^*) = 0$.

- **(typical case)** $f'(x^*) \neq 0$, then Newton's method (locally) converges to x^* quadratically with rate $\frac{1}{2}|f''(x^*)/f'(x^*)|$.
- **(troubled case)** If $f'(x^*) = 0$, then Newton's method either does not converge at all or it converges linearly.

Note that, if $f(x^*) = 0$ and $f'(x^*) = 0$, then x^* is a repeated root of $f(x)$. We thus conclude that Newton's method is troubled if we are trying to compute a repeated root.

- **(exceptionally good case)** If $f'(x^*) \neq 0$ and $f''(x^*) = 0$, then Newton's method even converges with order at least 3.

Important comment. In short, Newton's method typically converges quadratically (though in very special cases it can converge even faster) except in the case of repeated roots.

Proof. We apply Theorem 61 to analyze the fixed-point iteration of $g(x) = x - \dfrac{f(x)}{f'(x)}$.

Using the quotient rule we compute that

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{f'(x)^2} = \frac{f(x)f''(x)}{f'(x)^2}.$$

If $f(x^*) = 0$ and $f'(x^*) \neq 0$, then we have $g'(x^*) = 0$. By Theorem 61 this implies that fixed-point iteration converges at least quadratically.

To determine the rate of convergence, we further compute (again using the quotient and product rule) that

$$g''(x) = \frac{(f'(x)f''(x) + f(x)f'''(x))f'(x)^2 - 2f(x)f''(x)f'(x)f''(x)}{f'(x)^4}.$$

From this (unsimplified) expression and $f(x^*) = 0$ we conclude that $g''(x^*) = \dfrac{f''(x^*)}{f'(x^*)}$.

By Theorem 61 this implies that the convergence is quadratic with rate $\dfrac{1}{2}\left|\dfrac{f''(x^*)}{f'(x^*)}\right|$.

Moreover, if $f''(x^*) = 0$ then $g''(x^*) = 0$ so that the convergence is even cubic (or higher). □

Example 64. $f(x) = e^{-x} - x$ has the unique root $x^* \approx 0.567$. Determine whether Newton's method converges locally to x^*. If it does, what is the order and rate of convergence?

Solution. We compute that $f'(x) = -e^{-x} - 1$ and $f''(x) = e^{-x}$.

Since $x^* = e^{-x^*}$, we have $f'(x^*) = -x^* - 1 \neq 0$.

Hence, by Theorem 63, Newton's method converges to x^* quadratically.

Moreover, the rate is $\frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right| = \frac{1}{2}\left|\frac{e^{-x^*}}{-e^{-x^*} - 1}\right| = \frac{1}{2}\left|\frac{x^*}{-x^* - 1}\right| \approx 0.181$.

Review. If $f(x^*)=0$ and $f'(x^*)\neq 0$, then Newton's method (locally) converges to x^* quadratically with rate $\frac{1}{2}|f''(x^*)/f'(x^*)|$.

Note that we can see from here that $f'(x^*)=0$ is problematic; indeed, in that case, we don't get quadratic convergence (but rather divergence or linear convergence).

We can also see that, if $f''(x^*)=0$, then we should get even better convergence; indeed, in that case, we get cubic convergence or better.

Example 65. Consider $f(x)=(x-r)(x-1)(x+2)$ where r is some constant. Suppose we want to use Newton's method to calculate the root $x^*=1$.

(a) For which values of r is Newton's method guaranteed to converge (at least) quadratically to $x^*=1$?

(b) Analyze the cases in which Newton's method does not converge quadratically to $x^*=1$. Does it still converge? If so, what can we say about the order and rate of convergence?

(c) For which values of r does Newton's method converge to $x^*=1$ faster than quadratically?

Solution.

(a) We have $f(x)=x^3-(r-1)x^2-(r+2)x+2r$ and, hence, $f'(x)=3x^2-2(r-1)x-(r+2)$.
Note that $f'(1)=3-3r=0$ if and only if $r=1$.
Theorem 63 implies that Newton's method converges (at least) quadratically to $x^*=1$ if $r\neq 1$.
Comment. Note that $r=1$ is precisely the case where 1 becomes a double root of $f(x)$.

(b) We need to analyze the case $r=1$.
In that case $f(x)=(x-1)^2(x+2)$ and $f'(x)=3x^2-3=3(x-1)(x+1)$.
Newton's method applied to $f(x)$ is equivalent to fixed-point iteration of

$$g(x)=x-\frac{f(x)}{f'(x)}=x-\frac{(x-1)^2(x+2)}{3(x-1)(x+1)}=x-\frac{x^2+x-2}{3(x+1)}=\frac{2}{3}x+\frac{2}{3}\frac{1}{x+1}.$$

We compute that $g'(x)=\frac{2}{3}-\frac{2}{3}\frac{1}{(x+1)^2}$ so that, in particular, $g'(1)=\frac{2}{3}-\frac{2}{3}\frac{1}{4}=\frac{1}{2}$.
Since $0\neq|g'(1)|<1$ we conclude, by Theorem 61, that Newton's method (locally) converges to $x^*=1$. Moreover, the convergence is linear with rate $\frac{1}{2}$.
Comment. Since $\frac{1}{2}=2^{-1}$, this means that we gain roughly one correct binary digit per iteration.

(c) We continue the calculation from the first part. According to Theorem 63, Newton's method converges to 1 faster than quadratic if $f'(1)\neq 0$ and $f''(1)=0$.
We calculate $f''(x)=6x-2(r-1)$. Thus $f''(1)=8-2r=0$ if and only if $r=4$.
Hence, Newton's method converges to 1 faster than quadratic if $r=4$.

Important comment. Note that what we are observing is exactly as what we should expect: Newton's method typically converges quadratically (though in very special cases it can converge even faster; here, $r=4$) except in the case of repeated roots (here, $r=1$).

Armin Straub
straub@southalabama.edu

Example 66. Python The following code implements the Newton method specifically for computing a root of $f(x)=(x-r)(x-1)(x+2)$ as in the previous example.

```
>>> def newton_f(r, x, nr_steps):
        for i in range(nr_steps):
            x = x - ((x-r)*(x-1)*(x+2))/(3*x**2-2*(r-1)*x-r-2)
        return x
```

We then write a function to tell us the how close the result of Newton's method is to $x^*=1$ (the root that we are trying to compute). Namely, `newton_f_cb_1` will return the number of correct digits in base 2.

```
>>> from math import log2

>>> def newton_f_cb_1(r, x, nr_steps):
        return -log2(abs(1 - newton_f(r, x, nr_steps)))
```

Here is the typical behaviour which we get if $r\neq 1$ and $r\neq 4$. We chose $r=2$ and for the initial approximation we chose $x_0=0.4$. First, we list the result of Newton's method and observe that the approximations are indeed approaching 1 (recall that we are only guaranteed convergence if x_0 is close enough to 1). We then list the number of correct bits for those approximations:

```
>>> [newton_f(2, 0.4, n) for n in range(1,5)]

   [0.9333333333333332, 0.9974499089253187, 0.9999956903710115, 0.9999999999876182]

>>> [newton_f_cb_1(2, 0.4, n) for n in range(1,5)]

   [3.9068905956085165, 8.615235511834927, 17.824004894803025, 36.2329923774517]
```

Observe how the number of correct digits indeed roughly doubles.

Next, we likewise consider the problematic case $r=1$:

```
>>> [newton_f(1, 0.4, n) for n in range(1,5)]

   [0.7428571428571429, 0.877751756440281, 0.9402023433223725, 0.9704083354780979]

>>> [newton_f_cb_1(1, 0.4, n) for n in range(1,5)]

   [1.9593580155026542, 3.032114357937968, 4.063767239896592, 5.078665339814252]
```

Observe how the number of correct digits no longer doubles. Instead it roughly increases by 1 per iteration, exactly as we had predicted.

Finally, we consider the exceptionally good case $r=4$:

```
>>> [newton_f(4, 0.4, n) for n in range(1,5)]

   [1.0545454545454547, 0.9999639010889838, 1.0000000000000104, 1.0]

>>> [newton_f_cb_1(4, 0.4, n) for n in range(1,4)]

   [4.1963972128035, 14.757685157968053, 46.445411148322364]
```

Observe how the number of correct digits now roughly triples, in accordance with our prediction.

Comparison of root finding algorithms

Now that we have seen several root finding algorithms, which one is the best?

Well, it really depends on the situation. Below are some of the differences between the methods.

In practice, one often uses hybrid algorithms that combine several methods.

All methods require a continuous function.

- Bisection
  - each iteration is guaranteed to provide a correct binary digit; no other method can guarantee this for all functions
  - requires an initial interval containing a root such that the function values at the endpoints have opposite signs (in particular, does not work for double roots (or any even order roots)); on the other hand, it provides a guaranteed interval containing the root
  - no requirement on $f(x)$ besides continuity; for the other methods, the performance depends on $f(x)$
  - essentially linear convergence with rate $\frac{1}{2}$
- Regula falsi
  - also requires an initial interval containing a root like bisection
  - one endpoint of the interval typically gets stuck
  - rarely used directly, but rather in its improved forms, such as the Illinois method
  - always converges, typically linearly with variable rate
- Illinois method
  - improved version of regula falsi
  - the interval now shrinks to root
  - always converges, typically with order $\sqrt[3]{3} \approx 1.442$
- Secant method
  - only requires an initial approximation
  - only converges if initial approximation is good enough
  - potential numerical issues due to loss of precision in near zero denominator
  - typical order of convergence $\phi = (1+\sqrt{5})/2 \approx 1.618$
- Newton's method
  - similar to secant method
  - requires derivative
  - extends well to other contexts such as approximating functions or power series rather than numbers
  - typical order of convergence 2
  - however, adjusted for two function evaluations ($f(x)$ and $f'(x)$), order of convergence $\sqrt{2} \approx 1.414$