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 $\sqrt{2} \approx 1.414$

How computers represent functions

From classes like calculus, we are probably used to representing functions **symbolically**, such as:

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

Advantage. Because this is an exact expression, there is no loss of precision. Moreover, our calculus skills allow us to compute things like derivatives exactly.

Disadvantage. Computing with such formulas quickly results in very complicated expressions. Generally, things can get very slow very quickly. (Keep in mind the explosion in size of the rational numbers when we ran, for instance, Newton's method using exact numbers rather than floats.)

Moreover, in practice, a function of interest often simply doesn't have a symbolic expression. In such cases, we have to work with an approximation. There are many different ways to **numerically** represent and approximate a function.

For instance, we have already approximated functions using Taylor polynomials (these are particularly good at representing a function near a single point), which are truncations of the function's Taylor series. Another representation you have likely seen in other classes are Fourier series (combinations of sine and cosine functions) and their truncations.

A very basic numerical way to describe a function numerically is via a table of function values. In that case, how should be compute the function at an intermediate value not included in the table? This leads us to polynomial interpolation as well as to splines.

Interpolation

Suppose we have $d+1$ points $(x_0, y_0), (x_2, y_2), \ldots, (x_d, y_d)$. A function $f(x)$ is said to **interpolate** these points if $f(x_i) = y_i$ for all $i \in \{0, 1, \ldots, d\}$.

An important case is to interpolate given points using a polynomial.

Why polynomials? Recall that any polynomial in x of degree n can be written as

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

with some coefficients a_j. The reason that these are so ubiquitous is that polynomials are precisely what can be constructed via addition and multiplication starting with numbers and a variable x.

Example 67. (warmup) Suppose we wish to interpolate the points $(0,1), (1,2), (2,5)$ using a polynomial.

Just as two points determine a line, these three points should determine a parabola $a_0 + a_1 x + a_2 x^2$.

Indeed, note that there are three degrees of freedom (namely, a_0, a_1, a_2) and that each point gives rise to an equation. Three equations should determine the values of the three unknowns uniquely.

Let us spell out the three equations:

$(0,1)$: $a_0 = 1$

$(1,2)$: $a_0 + a_1 + a_2 = 2$

$(2,5)$: $a_0 + 2a_1 + 4a_2 = 5$

We can solve these to find $a_0 = 1$, $a_1 = 0$, $a_2 = 1$ so that the corresponding interpolating polynomial is $x^2 + 1$.

However, in general solving a system of equations is a lot of work! In the next sections, we will see that there are better and more efficient ways that avoid a lot of this work.

Also note that, from this point of view, it is not completely clear that the system of equations is always solvable.

Example 68. Interpolate the points $(0,1),(1,2),(2,5)$ using a polynomial.

Solution. Without any computations, we can actually immediately write down such a polynomial:

$$p(x) = 1\frac{(x-1)(x-2)}{(0-1)(0-2)} + 2\frac{(x-0)(x-2)}{(1-0)(1-2)} + 5\frac{(x-0)(x-1)}{(2-0)(2-1)}$$

This is the **Lagrange form** of the interpolating polynomial. Carefully look at this expression to see why it interpolates the three points! For instance, why is $p(1)=2$? Note that only the middle term contributes to $p(1)$ because the other two terms have a factor of $x-1$.

We can then simplify the above $p(x)$ to get $p(x) = \frac{1}{2}(x-1)(x-2) - 2x(x-2) + \frac{5}{2}x(x-1) = x^2+1$.

As in the previous example, we can always write down an interpolating polynomial. In particular, this allows us to conclude the following.

Comment. Note that this is not surprising. It is a generalization of the well-known fact that two points determine a unique interpolating line.

Theorem 69. Given $d+1$ points $(x_0, y_0), (x_2, y_2), \ldots, (x_d, y_d)$ with distinct x_i, there exists a unique interpolating polynomial of degree at most d.

Proof. Existence follows because, as in the previous example, we can explicitly write down such an interpolating polynomial $p(x)$ as

$$p(x) = \sum_{j=0}^{d} y_j \frac{\prod_{i\neq j}(x-x_i)}{\prod_{i\neq j}(x_j-x_i)}.$$

For uniqueness, suppose there are two interpolating polynomials $p(x)$ and $q(x)$ of degree at most d. The difference $p(x)-q(x)$ has degree at most d. On the other hand, $p(x)-q(x)$ has $d+1$ roots because it vanishes for $x = x_0, x_1, \ldots, x_d$. Since a nonzero polynomial of degree at most d cannot have more than d roots, we conclude that $p(x)-q(x)=0$. □

Example 70. Interpolate the points $(0,1),(1,3),(2,5)$ using a polynomial of least degree.

Solution. The interpolating polynomial in Lagrange form is:

$$\begin{aligned} p(x) &= 1\frac{(x-1)(x-2)}{(0-1)(0-2)} + 3\frac{(x-0)(x-2)}{(1-0)(1-2)} + 5\frac{(x-0)(x-1)}{(2-0)(2-1)} \\ &= \frac{1}{2}(x-1)(x-2) - 3x(x-2) + \frac{5}{2}x(x-1) \\ &= 2x+1 \end{aligned}$$

Comment. Note that this is the special case where three points end up being on a single line. However, note that this wasn't obvious from the Lagrange form until we expanded it out to $2x+1$.

We will next see an alternative approach (Newton's divided differences) to obtaining the interpolating polynomial which, as one benefit (among others) would make it apparent that the resulting polynomial is of lower than expected degree.

Polynomial interpolation: the Newton form

We have seen that, given $d+1$ points $(x_0, y_0), (x_2, y_2), \ldots, (x_d, y_d)$ with distinct x_i, there exists a unique interpolating polynomial $p(x)$ of degree at most d.

The **Lagrange form** of this polynomial is $p(x) = \sum_{j=0}^{d} y_j \frac{\prod_{i\neq j}(x-x_i)}{\prod_{i\neq j}(x_j-x_i)}$.

The **Newton form** instead expresses the polynomial in the form

$$p(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + c_3(x-x_0)(x-x_1)(x-x_2) + \ldots$$

Comment. Note that the Newton form of a polynomial can be thought of as a generalization of Taylor expansion. Indeed, we get Taylor expansion around $x=c$ if we choose all x_i to be equal to c (of course, for the purposes of interpolation, the x_i will be different).

Example 71. Determine the minimal polynomial interpolating the points $(-3,-1), (-1,5), (0,8), (2,-1)$.

Solution. (Lagrange) The interpolating polynomial in Lagrange form is:

$$\begin{aligned} p(x) &= -1\frac{(x+1)x(x-2)}{(-3+1)(-3)(-3-2)} + 5\frac{(x+3)x(x-2)}{(-1+3)(-1)(-1-2)} + 8\frac{(x+3)(x+1)(x-2)}{(0+3)(0+1)(0-2)} - 1\frac{(x+3)(x+1)x}{(2+3)(2+1)2} \\ &= -\frac{1}{2}x^3 - 2x^2 + \frac{3}{2}x + 8 \end{aligned}$$

Solution. (Newton, direct approach) The interpolating polynomial in Newton form is

$$p(x) = c_0 + c_1(x+3) + c_2(x+3)(x+1) + c_3(x+3)(x+1)x.$$

We use the four points to solve for the coefficients c_i:

$$\begin{aligned} (-3,-1) &: c_0 = -1 \\ (-1,5) &: \underbrace{c_0}_{-1} + 2c_1 = 5 \quad \Longrightarrow \quad c_1 = 3 \\ (0,8) &: \underbrace{c_0}_{-1} + 3\underbrace{c_1}_{3} + 3c_2 = 8 \quad \Longrightarrow \quad c_2 = 0 \\ (2,-1) &: \underbrace{c_0}_{-1} + 5\underbrace{c_1}_{3} + 15\underbrace{c_2}_{0} + 30c_3 = -1 \quad \Longrightarrow \quad c_2 = -\frac{1}{2} \end{aligned}$$

Hence, $p(x) = -1 + 3(x+3) - \frac{1}{2}(x+3)(x+1)x = -\frac{1}{2}x^3 - 2x^2 + \frac{3}{2}x + 8$.

Comment. Note how, by design of the Newton form, the equation for each point engaged one additional coefficient c_j, allowing us to solve for c_j (without having to combine several equations).

In particular, note how we are building intermediate interpolating polynomials:

- $c_0 + c_1(x+3) = -1 + 3(x+3)$ interpolates $(-3,1), (-1,6)$.
- $c_0 + c_1(x+3) + c_2(x+3)(x+1) = -1 + 3(x+3)$ interpolates $(-3,1), (-1,6), (0,3)$.

Newton's divided differences

Next, we observe an alternative way of computing the coefficients c_j in the Newton form

$$p(x) = c_0 + c_1(x-x_0) + c_2(x-x_0)(x-x_1) + c_3(x-x_0)(x-x_1)(x-x_2) + \dots \tag{1}$$

for interpolating a function f at $x = x_0, x_1, x_2, \dots$.

Example 72. Determine the first few coefficients. Below, we will use the following notation for these coefficients: $c_0 = f[x_0]$, $c_1 = f[x_0, x_1]$, $c_2 = f[x_0, x_1, x_2]$, ...

Solution. For brevity, we write $y_j = f(x_j)$.

- Using (x_0, y_0): $p(x_0) = c_0 \overset{!}{=} y_0$
  $f[x_0] = c_0 = y_0$
- Using (x_1, y_1): $p(x_1) = c_0 + c_1(x_1 - x_0) \overset{!}{=} y_1$
  $f[x_0, x_1] = c_1 = \dfrac{y_1 - y_0}{x_1 - x_0}$
  Note that the coefficient $f[x_0, x_1]$ is a **divided difference** (the slope of the line through the two points).
- Using (x_2, y_2): $p(x_2) = c_0 + c_1(x_2 - x_0) + c_2(x_2 - x_0)(x_2 - x_1) \overset{!}{=} y_2$
  $f[x_0, x_1, x_2] = c_2 = \dfrac{y_2 - y_0 - \frac{y_1 - y_0}{x_1 - x_0}(x_2 - x_0)}{(x_2 - x_0)(x_2 - x_1)} \overset{\text{check!}}{=} \dfrac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0} = \dfrac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$
  The coefficient $f[x_0, x_1, x_2]$ is what we call a **divided difference of order 2**.

Definition 73. Define $f[x_0, x_1, \dots, x_n]$ to be the coefficient of x^n (the highest power of x) in the minimal-degree polynomial interpolating f at $x = x_0, x_1, \dots, x_n$.

Important. In other words, $f[x_0, x_1, \dots, x_n]$ is the coefficient c_n in the Newton form (1).
$f[x_0, x_1, \dots, x_n]$ is called a **divided difference of order n** of the function f because of the recursive relation illustrated in the previous example, which is proven in general in the next theorem.
Note that, by definition, $f[x_0, x_1, \dots, x_n]$ does not depend on the order of the points.

Theorem 74. The divided differences $f[x_0, x_1, \dots, x_n]$ are recursively determined by $f[a] = f(a)$ as well as the relation

$$f[P, a, b] = \frac{f[P, b] - f[P, a]}{b - a},$$

where P is a set of points.

For instance. With $P = x_1, \dots, x_{n-1}$ and $a = x_0$, $b = x_n$, the recursive relation becomes

$$f[x_0, \dots, x_n] = \frac{f[x_1, \dots, x_n] - f[x_0, \dots, x_{n-1}]}{x_n - x_0}.$$

Proof. Suppose that $P=\{x_0,\dots,x_n\}$ and that

$$p_0(x)=c_0+c_1(x-x_0)+c_2(x-x_0)(x-x_1)+\dots+c_n(x-x_0)(x-x_1)\cdots(x-x_{n-1})$$

is the interpolating polynomial for $x_0,\dots,x_n$. Then

$$\begin{aligned} p_a(x) &= p_0(x)+f[P,a](x-x_0)(x-x_1)\cdots(x-x_{n-1})(x-x_n),\\ p_b(x) &= p_0(x)+f[P,b](x-x_0)(x-x_1)\cdots(x-x_{n-1})(x-x_n) \end{aligned}$$

are the interpolating polynomials for $x_0,\dots,x_n,a$ and $x_0,\dots,x_n,b$, respectively. The claim follows if we can show that

$$p_{a,b}(x)=p_a(x)+\cdot\frac{f[P,b]-f[P,a]}{b-a}(x-x_0)(x-x_1)\cdots(x-x_{n-1})(x-x_n)(x-a)$$

is the interpolating polynomial for $x_0,\dots,x_n,a,b$. By construction, it interpolates $x=x_0,\dots,x_n,a$. To see that it also interpolates $x=b$, note that

$$\begin{aligned} p_{a,b}(b) &= p_a(b)+\cdot(f[P,b]-f[P,a])\boxed{(b-x_0)(b-x_1)\cdots(b-x_{n-1})(b-x_n)}\\ &= p_0(b)+f[P,a]\boxed{(b-x_0)\cdots(b-x_{n-1})(b-x_n)}+\cdot(f[P,b]-f[P,a])\boxed{(b-x_0)\cdots(b-x_n)}\\ &= p_0(b)+f[P,b]\boxed{(b-x_0)\cdots(b-x_{n-1})(b-x_n)}\\ &= p_b(b)=f(b). \end{aligned}$$

□

(Newton form using divided differences)

The Newton form of the polynomial $p(x)$ interpolating f at $x=x_0,x_1,\dots$ is

$$p(x)=c_0+c_1(x-x_0)+c_2(x-x_0)(x-x_1)+c_3(x-x_0)(x-x_1)(x-x_2)+\dots,$$

where the coefficients $c_n=f[x_0,x_1,\dots,x_n]$ can be computed using the triangular scheme:

	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$	…
x_0	$f[x_0]$				
		$f[x_0,x_1]=\frac{f[x_1]-f[x_0]}{x_1-x_0}$			
x_1	$f[x_1]$		$f[x_0,x_1,x_2]=\frac{f[x_1,x_2]-f[x_0,x_1]}{x_2-x_0}$		
		$f[x_1,x_2]=\frac{f[x_2]-f[x_1]}{x_2-x_1}$		$f[x_0,x_1,x_2,x_3]=\dots$	
x_2	$f[x_2]$		$f[x_1,x_2,x_3]=\frac{f[x_2,x_3]-f[x_1,x_2]}{x_3-x_1}$		…
		$f[x_2,x_3]=\frac{f[x_3]-f[x_2]}{x_3-x_2}$		…	
x_3	$f[x_3]$	…			…

Note that the coefficients $c_n=f[x_0,x_1,\dots,x_n]$ needed for the Newton form appear at the top edge of the triangle (in the shaded cells).

Example 75. Determine the minimal polynomial interpolating the points $(-3,-1),(-1,5),(0,8),(2,-1)$.

Solution. (Newton, direct approach; again, for comparison) The interpolating polynomial in Newton form is

$$p(x) = c_0 + c_1(x+3) + c_2(x+3)(x+1) + c_3(x+3)(x+1)x.$$

We use the four points to solve for the coefficients c_i:

$$\begin{aligned}
(-3,-1) &: c_0 = -1 \\
(-1,5) &: \underbrace{c_0}_{-1} + 2c_1 = 5 \quad \Longrightarrow \quad c_1 = 3 \\
(0,8) &: \underbrace{c_0}_{-1} + 3\underbrace{c_1}_{3} + 3c_2 = 8 \quad \Longrightarrow \quad c_2 = 0 \\
(2,-1) &: \underbrace{c_0}_{-1} + 5\underbrace{c_1}_{3} + 15\underbrace{c_2}_{0} + 30c_3 = -1 \quad \Longrightarrow \quad c_2 = -\frac{1}{2}
\end{aligned}$$

Hence, $p(x) = -1 + 3(x+3) - \frac{1}{2}(x+3)(x+1)x = -\frac{1}{2}x^3 - 2x^2 + \frac{3}{2}x + 8$.

Solution. (Newton, divided differences)

	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$	$f[\cdot,\cdot,\cdot,\cdot]$
-3	-1			
		$\frac{5-(-1)}{-1-(-3)} = 3$		
-1	5		$\frac{3-3}{0-(-3)} = 0$	
		$\frac{8-5}{0-(-1)} = 3$		$\frac{-\frac{5}{2}-0}{2-(-3)} = -\frac{1}{2}$
0	8		$\frac{-\frac{9}{2}-3}{2-(-1)} = -\frac{5}{2}$	
		$\frac{-1-8}{2-0} = -\frac{9}{2}$		
2	-1			

Accordingly, reading the coefficients from the top edge of the triangle (as shaded above), the Newton form is

$$p(x) = -1 + 3(x+3) - \frac{1}{2}(x+3)(x+1)x = -\frac{1}{2}x^3 - 2x^2 + \frac{3}{2}x + 8,$$

in agreement with what we had computed earlier.

Example 76. (homework) Determine the minimal polynomial interpolating $(0,-1),(2,1),(3,8)$.

Solution. (Lagrange) The interpolating polynomial in Lagrange form is:

$$\begin{aligned} p(x) &= -1\frac{(x-2)(x-3)}{(0-2)(0-3)} + 1\frac{(x-0)(x-3)}{(2-0)(2-3)} + 8\frac{(x-0)(x-2)}{(3-0)(3-2)} \\ &= -\frac{1}{6}(x-2)(x-3) - \frac{1}{2}x(x-3) + \frac{8}{3}x(x-2) \\ &= 2x^2 - 3x - 1 \end{aligned}$$

Solution. (Newton, direct approach) The interpolating polynomial in Newton form is

$$p(x) = c_0 + c_1(x-0) + c_2(x-0)(x-2).$$

We use the three points to solve for the coefficients c_i:

- $(0,-1)$: $c_0 = -1$.
- $(2,1)$: $\underbrace{c_0}_{-1} + 2c_1 = 1$, so that $c_1 = 1$.
- $(3,8)$: $\underbrace{c_0}_{-1} + 3\underbrace{c_1}_{1} + 3c_2 = 8$, so that $c_2 = 2$.

Hence, $p(x) = -1 + 1(x-0) + 2(x-0)(x-2) = 2x^2 - 3x - 1$.

Solution. (Newton, divided differences)

$$\begin{array}{lccc} 0\colon & -1 & & \\ & & \frac{1-(-1)}{2-0} = 1 & \\ 2\colon & 1 & & \frac{7-1}{3-0} = 2 \\ & & \frac{8-1}{3-2} = 7 & \\ 3\colon & 8 & & \end{array}$$

Accordingly, reading the coefficients from the top edge of the triangle, the Newton form is

$$p(x) = -1 + 1(x-0) + 2(x-0)(x-2) = 2x^2 - 3x - 1.$$

Example 77. (homework) Repeat the previous example with the additional point $(1,-2)$.

Solution. (Newton, divided differences) Notice how only the shaded entries are new.

$$\begin{array}{lcccc} 0\colon & -1 & & & \\ & & \frac{1-(-1)}{2-0} = 1 & & \\ 2\colon & 1 & & \frac{7-1}{3-0} = 2 & \\ & & \frac{8-1}{3-2} = 7 & & \frac{2-2}{1-0} = 0 \\ 3\colon & 8 & & \frac{5-7}{1-2} = 2 & \\ & & \frac{-2-8}{1-3} = 5 & & \\ 1\colon & -2 & & & \end{array}$$

Since the point $(1,-2)$ is on the graph of $2x^2 - 3x - 1$, we obtained the same final polynomial. If we had added a point not on the graph, then we would have found a degree 3 polynomial interpolating the total of four points.

Important comment. This is a considerable advantage for many practical purposes since often one does not know a priori how many interpolation points to use.

Example 78. Python numpy and scipy are powerful scientific libraries for Python. While numpy provides core functionality, scipy implements more specialized routines such as interpolation.

```
>>> from numpy import linspace, pi, sin
```

```
>>> from scipy import interpolate
```

Comment. We previously used the sin function available in the math Python standard library. The numpy library offers its own sin function with additional features. For instance, try sin([1,2]). This evaluates $\sin(x)$ at both $x=1$ and $x=2$. On the other hand, this results in an error with the sin function not from numpy.

Let us interpolate $f(x)=\sin(x)$ using 3 points, namely $x_0=0$, $x_1=\frac{\pi}{2}$, $x_2=\pi$. We begin by making lists of the x and y values as follows:

```
>>> xpoints = [0, pi/2, pi]
```

```
>>> ypoints = [sin(x) for x in xpoints]
```

Comment. As pointed out in the previous comment, we can even simply use ypoints = sin(xpoints). (The result would be a numpy array instead of a standard list but, for basic purposes, these behave alike. The numpy library introduces and uses arrays for additional features and performance for scientific computations.)

Let us check that xpoints and ypoints hold the expected values:

```
>>> xpoints
```

```
[0, 1.5707963267948966, 3.141592653589793]
```

```
>>> ypoints
```

```
[0.0, 1.0, 1.2246467991473532e-16]
```

We now ask scipy to create the interpolating polynomial:

```
>>> poly = interpolate.lagrange(xpoints, ypoints)
```

The resulting polynomial can be evaluated at any other point (such as $x=\pi/4$) and we can access its coefficients (which tell us that the polynomial is approximately $-0.41x^2+1.27x$):

```
>>> poly(pi/4)
```

```
0.75
```

```
>>> sin(pi/4)
```

```
0.7071067811865475
```

```
>>> poly.coeffs
```

```
[-0.40528473456935116, 1.2732395447351628, 0.0]
```

Homework. Show that the exact interpolation polynomial is $\frac{4}{\pi}x-\frac{4}{\pi^2}x^2$.

Finally, let us plot the sine function together with the polynomial interpolation. In the code below we use matplotlib, a powerful and widely used plotting library.

```
>>> import matplotlib.pyplot as plt
```

```
>>> xplot = linspace(0, pi, 100)
```

```
>>> plt.plot(xpoints, ypoints, 'o', xplot, sin(xplot), '-', xplot, poly(xplot), ':')
```

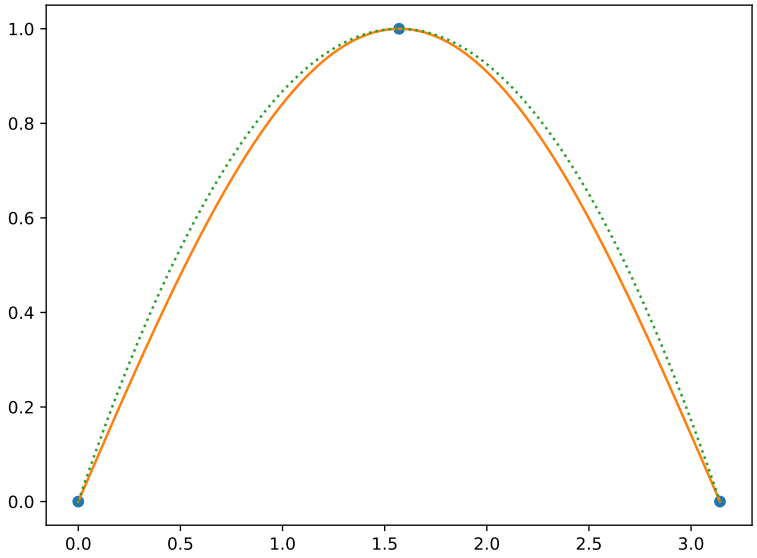
```
>>> plt.show()
```

Comment. Note that we are making three plots in one line here (namely, we plot the three points, we plot sine, and we plot the polynomial interpolation).

To plot just sine, simplify the plot command to plt.plot(xplot, sin(xplot), '-'). The '-' connects the 100 points (with x-coordinates from xplot) by a line. Replace it, for instance, with 'r-.' to get a red dotted line.

https://matplotlib.org/stable/tutorials/introductory/pyplot.html

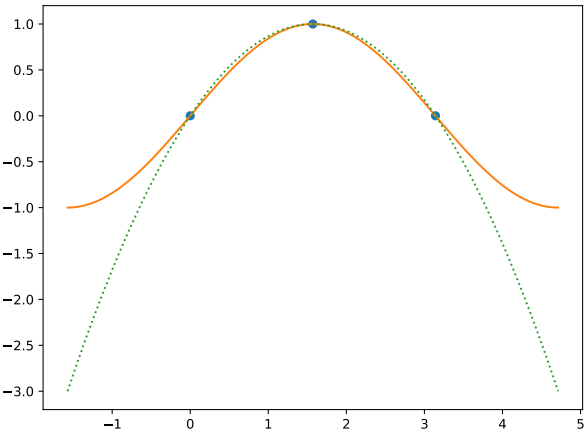
The resulting output should look as follows:

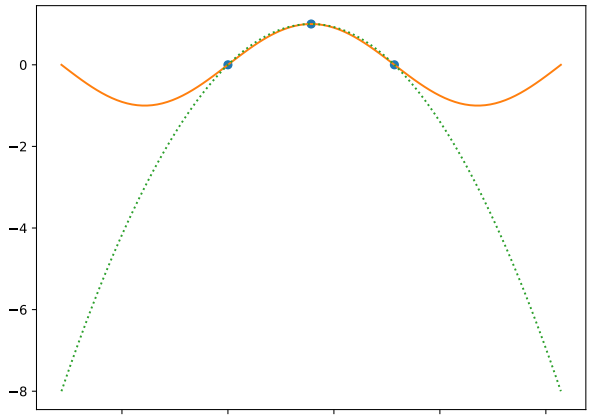


This shows pretty decent interpolation on the interval $[0, \pi]$.

Which function is which?! (You can tell from the fact that we dotted one graph or from the plots below.)

On the other hand, here are the same plots on $\left[-\frac{\pi}{2}, \frac{3\pi}{2}\right]$ and $[-\pi, 2\pi]$:





Homework. Adjust our code above (only the line `linspace(0, pi, 100)` needs to be changed) to produce these two plots.

As we can see (and as we probably expected), the polynomial interpolation does not approximate the sine function outside the interval $[0, \pi]$.

Comment. Given the three interpolation points $0, \pi/2, \pi$, an attempt to approximate the function at values much less than 0 or much larger than π (that is, outside of the range of our data) is typically referred to as **extrapolation**.

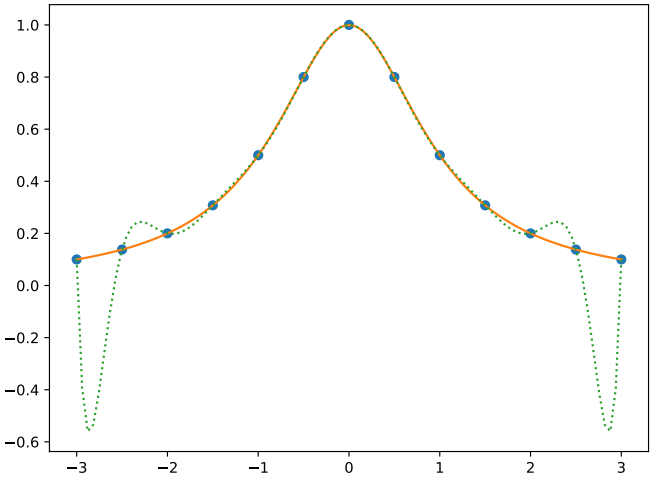
Python assignment #3

The goal of this assignment is to observe **Runge's phenomenon**.

https://en.wikipedia.org/wiki/Runge%27s_phenomenon

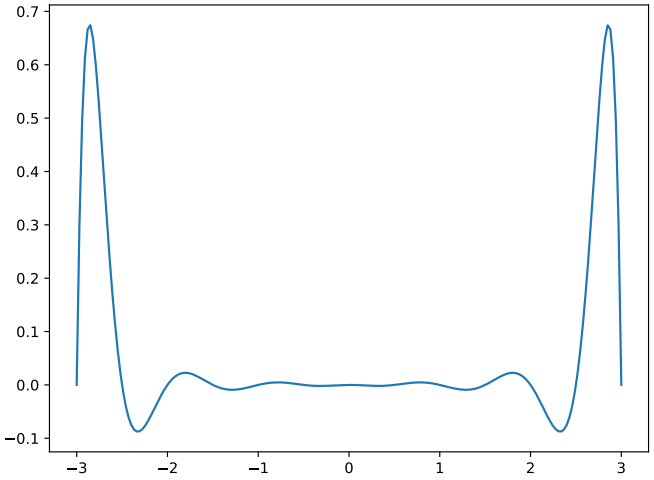
Example 79. We wish to interpolate the function $f(x) = \frac{1}{1+x^2}$ at the equally spaced points $x_0, x_1, \ldots, x_{12} = -3, -2.5, -2, \ldots, 2.5, 3$. Let $p(x)$ be the resulting polynomial interpolation. Proceed as we did in Example 78.

(a) Plot both $p(x)$ and $f(x)$ on the interval $[-3, 3]$. The output should look as follows:



Observe the problematic oscillations towards the boundaries of the interval $[-3, 3]$.

(b) Plot the error $f(x) - p(x)$ on the interval $[-3, 3]$. The output should look as follows:



(c) What is $p\left(\frac{1}{3}\right)$? For comparison, what is $f\left(\frac{1}{3}\right)$?

Solution. $p\left(\frac{1}{3}\right) \approx 0.9019025435658377$ and $f\left(\frac{1}{3}\right) = 0.9$.

(d) Using 100 equally spaced points, approximate the maximum error on the interval $[-3, 3]$ if $f(x)$ is approximated by $p(x)$.

If `poly` is the interpolating polynomial and if you defined a function called `f`, then this error is:

```
>>> max([abs(f(x)-poly(x)) for x in linspace(-3,3,100)])
```

Example 80. Repeat the previous example using instead the points

$$x_j = 3\cos\left(\frac{2j+1}{26}\pi\right), \quad j = 0, \ldots, 12.$$

Note that this is the same number of points as before but that they are no longer equally spaced.

After you have submitted your code on Replit, send me an email with the following:

(a) The approximation of the maximum error when using equally spaced points.

(b) The approximation of the maximum error when using the not equally spaced points.

(c) A sentence or two on what you observe when comparing the plots of the errors $f(x) - p(x)$.

Finally, a pointer and a hint:

- First run the code corresponding to Example 78 to make sure that things are working.

  Initially, the last two lines (responsible for plotting) are commented. That allows you to interact with the code in the Console. For instance, run the code; then enter `poly(pi/4)` into the Console to see the value of the interpolating polynomial at $\pi/4$ (this value should be 0.75).

  On the other hand, to see the plot, uncomment those two lines. If you now run the code, the command `plt.show()` opens a little output window with the plot. We then need to click the **Stop** button after we are done admiring the plot.

- We should not enter the x-coordinates of our interpolation points by hand. Instead, check out the following example:

```
>>> [-2 + 0.5*j for j in range(9)]

   [-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2.0]
```

  Comment. You can also try `linspace(-2, 2, 9)` or `arange(-2, 2.5, 0.5)` (both of these must be imported from `numpy`) for a similar result.

Example 81. Determine the minimal polynomial interpolating $(0,1),(1,2),(2,5)$.

Solution. (Lagrange, review) The interpolating polynomial in Lagrange form is:

$$\begin{aligned} p(x) &= 1\frac{(x-1)(x-2)}{(0-1)(0-2)} + 2\frac{(x-0)(x-2)}{(1-0)(1-2)} + 5\frac{(x-0)(x-1)}{(2-0)(2-1)} \\ &= \frac{1}{2}(x-1)(x-2) - 2x(x-2) + \frac{5}{2}x(x-1) \\ &= x^2+1 \end{aligned}$$

Solution. (Newton, divided differences)

$$\begin{array}{lccc} 0\colon & 1 & & \\ & & \frac{2-1}{1-0}=1 & \\ 1\colon & 2 & & \frac{3-1}{2-0}=1 \\ & & \frac{5-2}{2-1}=3 & \\ 2\colon & 5 & & \end{array}$$

Accordingly, reading the coefficients from the top edge of the triangle:

$$p(x) = 1 + 1(x-0) + 1(x-0)(x-1) = x^2+1$$

A mean value theorem for divided differences

Review. The **mean value theorem** (see Theorem 49; the special case $M=0$ of Taylor's theorem) states that, if $f(x)$ is differentiable, then

$$f[a,b] = \frac{f(b)-f(a)}{b-a} = f'(\xi)$$

for some ξ between a and b.

Recall that the Newton form of the polynomial interpolating $f(x)$ at $x=x_0,x_1,\ldots$ is

$$f[x_0] + f[x_0,x_1](x-x_0) + f[x_0,x_1,x_2](x-x_0)(x-x_1) + f[x_0,x_1,x_2,x_3](x-x_0)(x-x_1)(x-x_2) + \ldots$$

Note that this is somewhat similar to the Taylor expansion of $f(x)$ at $x=x_0$, which is

$$f(x_0) + f'(x_0)(x-x_0) + \tfrac{1}{2!}f''(x_0)(x-x_0)^2 + \tfrac{1}{3!}f'''(x_0)(x-x_0)^3 + \ldots$$

Indeed, if all the x_j are equal to x_0 (this is technically not allowed when interpolating, but you can still think of choosing them all close to x_0), then the Newton form would turn into a Taylor polynomial.
In that case, $f[x_0,x_1,\ldots,x_n]$ would become $\frac{1}{n!}f^{(n)}(x_0)$.

With that (as well as the mean value theorem and Taylor's theorem (see Theorem 48)) in mind, the next result does not come as a surprise.

Theorem 82. (mean value theorem for divided differences) If $f(x)$ is differentiable, then

$$f[x_0, x_1, ..., x_n] = \frac{f^{(n)}(\xi)}{n!}$$

for some ξ between the smallest and the largest of the x_i.

Proof. Without loss of generality, we may assume that $x_0 < x_1 < \cdots < x_n$ (because divided differences do not depend on the ordering of the points x_i).
Let $P(x)$ be the interpolation polynomial for f at $x_0, x_1, ..., x_n$. Then $d(x) = f(x) - P(x)$ has $n+1$ zeros, namely $x_0, x_1, ..., x_n$. The mean value theorem implies that between any two zeros of a function, there must be a zero of its derivative (this is often referred to as Rolle's theorem). It therefore follows that $d'(x)$ has n zeros (between x_0 and x_n). Applying the same argument to $d'(x)$, we then find that $d''(x)$ has $n-1$ zeros. Continuing like this, $d^{(n)}(x)$ must have a zero ξ between x_0 and x_n. As such,

$$0 = d^{(n)}(\xi) = f^{(n)}(\xi) - P^{(n)}(\xi).$$

Recall that $P(x)$ is a polynomial of degree n or less, and that its Newton form is

$$P(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + ... + c_n(x - x_0)(x - x_1)\cdots(x - x_{n-1}),$$

where $c_j = f[x_0, x_1, ..., x_j]$. Note that $P^{(n)}(x) = n!\, c_n = n!\, f[x_0, x_1, ..., x_n]$. We therefore conclude that

$$0 = d^{(n)}(\xi) = f^{(n)}(\xi) - P^{(n)}(\xi) = f^{(n)}(\xi) - n!\, f[x_0, x_1, ..., x_n],$$

which proves the claim. □

Comment. Note that this provides us with a way to numerically approximate an nth derivative $f^{(n)}(x)$. Namely, choose $n+1$ points $x_0, x_1, ..., x_n$ near x. Then $f^{(n)}(x) \approx \underbrace{n!\, f[x_0, x_1, ..., x_n]}_{= f^{(n)}(\xi)}$.

Bounding the interpolation error

Theorem 83. (interpolation error) Suppose that $f(x)$ is $n+1$ times continuously differentiable. Let $P_n(x)$ be the interpolating polynomial for $f(x)$ at $x_0, x_1, ..., x_n$. Then

$$f(x) - P_n(x) = \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n)}_{\text{interpolation error}}$$

for some ξ between the smallest and the largest of the x_i together with x.

Proof. Let $P_{n+1}(x)$ be the interpolating polynomial for $f(x)$ at $x_0, x_1, ..., x_n, x_{n+1}$. We know that

$$\begin{aligned} P_{n+1}(x) &= P_n(x) + f[x_0, x_1, ..., x_{n+1}](x - x_0)(x - x_1)\cdots(x - x_n) \\ &= P_n(x) + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n) \end{aligned}$$

for some ξ between the smallest and the largest of the x together with x.
Given any fixed value t, choose $x_{n+1} = t$ in this formula (so that $P_{n+1}(t) = f(t)$) to conclude that

$$f(t) = P_{n+1}(t) = P_n(t) + \frac{f^{(n+1)}(\xi)}{(n+1)!}(t - x_0)(t - x_1)\cdots(t - x_n),$$

which is the claimed expression for the error term (with x replaced by t). □

Example 84. Suppose we approximate $f(x)=\sin(x)$ by the polynomial $P(x)$ interpolating it at $x=0,\frac{\pi}{2},\pi$. Without computing $P(x)$, give an upper bound for the error when $x=\frac{\pi}{4}$.

[Compare with Example 78 where we computed and plotted $P(x)$.]

Solution. By Theorem 83, the error is

$$\sin(x)-P(x)=\frac{f^{(3)}(\xi)}{3!}(x-0)\Big(x-\frac{\pi}{2}\Big)(x-\pi),$$

where ξ is between 0 and π (provided that x is in $[0,\pi]$). Note that $f^{(3)}(x)=-\cos(x)$ so that $|f^{(3)}(\xi)|\leqslant 1$. Hence, the error is bounded by

$$|\sin(x)-P(x)|\leqslant\frac{1}{6}\Big|x\Big(x-\frac{\pi}{2}\Big)(x-\pi)\Big|.$$

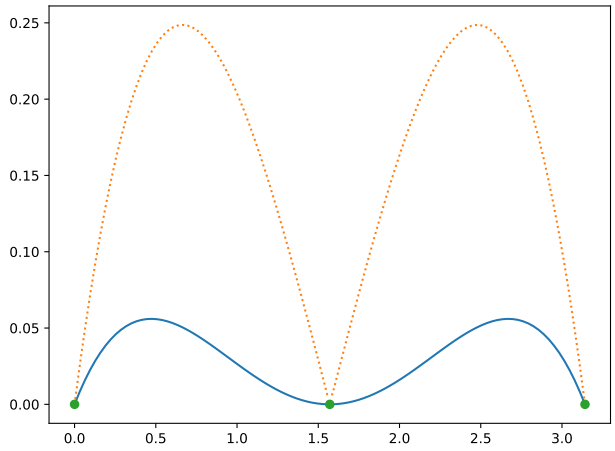
In particular, in the case $x=\frac{\pi}{4}$,

$$\Big|\sin\Big(\frac{\pi}{4}\Big)-P\Big(\frac{\pi}{4}\Big)\Big|\leqslant\frac{1}{6}\Big|\frac{\pi}{4}\Big(-\frac{\pi}{4}\Big)\Big(-\frac{3\pi}{4}\Big)\Big|=\frac{\pi^3}{128}\approx 0.242.$$

For comparison. In this particularly simple case, we can easily calculate the exact error.

Namely, since $\sin\big(\frac{\pi}{4}\big)=\frac{1}{\sqrt{2}}$ and $P\big(\frac{\pi}{4}\big)=\frac{3}{4}$ (see Example 78), the actual error is $\big|\sin\big(\frac{\pi}{4}\big)-P\big(\frac{\pi}{4}\big)\big|\approx 0.0428$.

Below is a plot of the actual error (in blue) together with our bound (dotted).



Homework. Following what we did in Example 78, try to reproduce this plot.

For which x in $[0,\pi]$ is our bound for the error maximal? What is the bound in that case?

Solution. Recall that our bound for the error is $\frac{1}{6}\big|x\big(x-\frac{\pi}{2}\big)(x-\pi)\big|$.

$x\big(x-\frac{\pi}{2}\big)(x-\pi)$ is maximal on $[0,\pi]$ for $x=\Big(1\pm\frac{1}{\sqrt{3}}\Big)\frac{\pi}{2}\approx 0.664, 2.478$. (Fill in the details!)

The corresponding error bound is $\frac{1}{72\sqrt{3}}\pi^3\approx 0.249$.

Comment. Note that this shows that our earlier error bound for $x=\frac{\pi}{4}\approx 0.785$ was close to maximal. That is not too much of a surprise since $\frac{\pi}{4}$ sits right between 0 and $\frac{\pi}{2}$ for which the error is 0 by construction.

For comparison. The actual maximal error occurs when $\cos(x)-\frac{4}{\pi}+\frac{8}{\pi^2}x=0$. (Why?!)

The approximate solutions are $x\approx 0.472, 2.670$ with corresponding (actual) error of 0.0560.

Make sure that you can identify both the x values and the error in the above plot.

Armin Straub
straub@southalabama.edu

Review. The Newton form of the polynomial interpolating $f(x)$ at $x = x_0, x_1, \ldots$ is

$$f[x_0] + f[x_0, x_1](x - x_0) + f[x_0, x_1, x_2](x - x_0)(x - x_1) + f[x_0, x_1, x_2, x_3](x - x_0)(x - x_1)(x - x_2) + \ldots$$

Comparing this to the Taylor expansion of $f(x)$ at $x = x_0$, which is

$$f(x_0) + f'(x_0)(x - x_0) + \tfrac{1}{2!}f''(x_0)(x - x_0)^2 + \tfrac{1}{3!}f'''(x_0)(x - x_0)^3 + \ldots,$$

it is not surprising that, as we showed, $f[x_0, x_1, \ldots, x_n] = \frac{1}{n!}f^{(n)}(\xi)$ for some ξ between the x_i.
Recall that, if $P_n(x)$ is the Taylor polynomial of order n, then the error term is $\frac{1}{(n+1)!}f^{(n+1)}(\xi)(x - x_0)^{n+1}$.
Likewise, if $P_n(x)$ is the interpolating polynomial for $f(x)$ at $x_0, x_1, \ldots, x_n$, then

$$f(x) = P_n(x) + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n)}_{\text{error term}}$$

for some ξ between the x_i and x.

Example 85. Suppose we approximate a function $f(x)$ by the polynomial $P(x)$ interpolating it at $x = -1, -\frac{1}{3}, \frac{1}{3}, 1$. Suppose that we know that $|f^{(n)}(x)| \leqslant n$ for all $x \in [-1, 1]$.

(a) Give an upper bound for the error when $x = -\frac{2}{3}$.

(b) Give an upper bound for the error when $x = 0$.

(c) Give an upper bound for the error for all $x \in [-1, 1]$.

Solution. By Theorem 83, the error is

$$f(x) - P(x) = \frac{f^{(4)}(\xi)}{4!}(x+1)\left(x + \frac{1}{3}\right)\left(x - \frac{1}{3}\right)(x-1) = \frac{f^{(4)}(\xi)}{4!}(x^2-1)\left(x^2 - \frac{1}{9}\right),$$

where ξ is between -1 and 1 (provided that $x \in [-1, 1]$). Since $\frac{1}{4!}|f^{(4)}(\xi)| \leqslant \frac{4}{4!} = \frac{1}{6}$, the error is bounded by

$$|f(x) - P(x)| \leqslant \frac{1}{6}\left|(x^2-1)\left(x^2 - \frac{1}{9}\right)\right|.$$

(a) If $x = -\frac{2}{3}$, then this bound becomes $|f(x) - P(x)| \leqslant \frac{1}{6}\left|(x^2-1)\left(x^2 - \frac{1}{9}\right)\right| = \frac{1}{6} \cdot \frac{5}{27} \approx 0.0309$.

(b) If $x = 0$, then this bound becomes $|f(x) - P(x)| \leqslant \frac{1}{6}\left|(x^2-1)\left(x^2 - \frac{1}{9}\right)\right| = \frac{1}{6} \cdot \frac{1}{9} \approx 0.0185$.
Comment. It is not surprising that this error bound is better than the one for $x = -\frac{2}{3}$ since, roughly speaking, there are more interpolation nodes around 0.

(c) Consider $g(x) = (x^2-1)\left(x^2 - \frac{1}{9}\right) = x^4 - \frac{10}{9}x^2 + \frac{1}{9}$. We need to compute $\max\limits_{x \in [-1,1]} |g(x)|$.
Since $g(\pm 1) = 0$, the maximum value of $|g(x)|$ must be attained at a point where $g'(x) = 0$.
We compute $g'(x) = 4x^3 - \frac{20}{9}x$. Hence $g'(x) = 0$ if $x = 0$ or $x = \pm\frac{\sqrt{5}}{3}$.
Since $|g(0)| = \frac{1}{9}$ and $\left|g\left(\pm\frac{\sqrt{5}}{3}\right)\right| = \frac{16}{81} > \frac{1}{9}$, we conclude that $\max\limits_{x \in [-1,1]} |g(x)| = \frac{16}{81}$.
Therefore, our bound for the error is $|f(x) - P(x)| \leqslant \frac{1}{6}\max\limits_{z \in [-1,1]}\left|(z^2-1)\left(z^2 - \frac{1}{9}\right)\right| = \frac{1}{6} \cdot \frac{16}{81} \approx 0.0329$.

Armin Straub
straub@southalabama.edu

Example 86. Python We can approximate $\frac{1}{6}\max_{z\in[-1,1]}\left|(z^2-1)\left(z^2-\frac{1}{9}\right)\right| = \frac{1}{6}\cdot\frac{16}{81} \approx 0.0329$ as follows using 100 points.

```
>>> from numpy import linspace
>>> max([1/6*abs((z**2-1)*(z**2-1/9)) for z in linspace(-1,1,100)])
```

```
0.0328984640831
```

Example 87. Python The following code measures how well a function f is approximated by the polynomial interpolating f at the given points. It returns an approximation of the maximal error on the interval $[a,b]$.

```
>>> from numpy import linspace, pi, cos, sin
>>> from scipy import interpolate
>>> def max_interpolation_error(f, a, b, xpoints, nr_sample_points):
        ypoints = [f(x) for x in xpoints]
        poly = interpolate.lagrange(xpoints, ypoints)
        max_error = max([abs(f(x)-poly(x)) for x in linspace(a,b,nr_sample_points)])
        return max_error
```

Let us verify that this works using an example we have discussed before:

```
>>> max_interpolation_error(sin, 0, pi, [0,pi/2,pi], 100)
```

```
0.0560067197786
```

This agrees with the maximal error that we observed at the end of Example 84. Let us look how the error develops as we add more points:

```
>>> [max_interpolation_error(sin, 0, pi, linspace(0,pi,n), 100) for n in range(2,9)]
```

```
[0.99987412767387496, 0.056006719778558423, 0.043613266903306247,
0.0018097268033398783, 0.0013114413108160916, 3.385907546618605e-05,
2.4246231325325551e-05]
```

It is pleasing to see that the error decreases. However, as we will see in the next example, this does not have to be the case.

Comment. Note that the error seems to really decrease every second step (i.e. after adding two more points). Can you offer an explanation for what might be the cause of this?

Example 88. Python However, this is not the end of the story. It turns out that the interpolation error does not always go down if we add additional points.

```
>>> def f(x):
        return 1/(1+25*x**2)

>>> [max_interpolation_error(f, -1, 1, linspace(-1,1,n), 100) for n in range(2,18)]
```

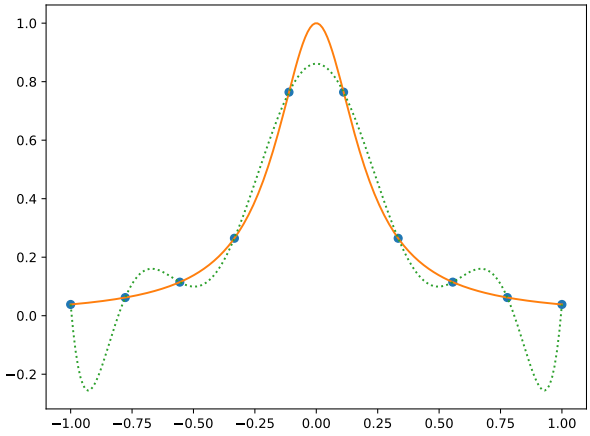
```
[0.9589941912351845, 0.6459699748665507, 0.7044952736346626, 0.4382728746134098,
0.43032461596244886, 0.6164015686420344, 0.24528527039305037, 1.0450782163781276,
0.297971540151836, 1.9154342696798625, 0.5538529081557272, 3.6117015978042333,
1.064460371610917, 7.189298472061041, 2.0967229089912, 14.013534491466531]
```

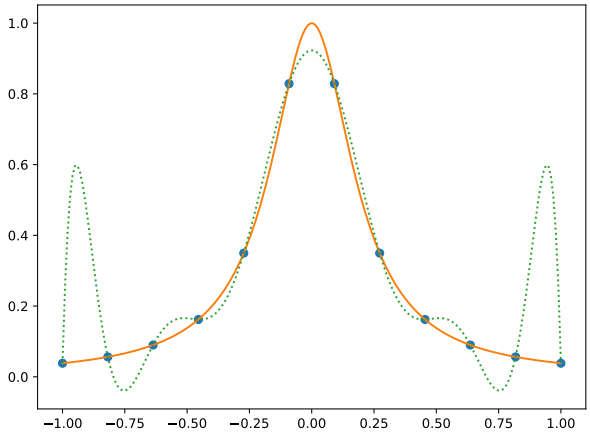
The function $f(x) = \frac{1}{1+25x^2}$ in this example is known as the **Runge function** and one can show that, by adding more points, the error grows without bound.

$$\lim_{n\to\infty} \max_{x\in[-,1,1]} |f(x) - P_n(x)| = \infty$$

https://en.wikipedia.org/wiki/Runge%27s_phenomenon

The following plots show the situation using 10 and 12 interpolation nodes.





While the approximation becomes better towards the center of the interval $[-1,1]$, the oscillations towards the ends of the interval become more violant (resulting in an increasing worst-case error).

Homework. Recreate the above plots following Example 78.

In the next section, we will see that we can avoid this issue if we don't choose equally spaced points but carefully chosen ones called **Chebyshev nodes**.

Chebyshev interpolation

In this section it will be convenient to use $x_1, \dots, x_n$ rather than $x_0, x_1, \dots, x_n$.

Review. If $P_{n-1}(x)$ is the interpolating polynomial for $f(x)$ at $x_1, \dots, x_n$, then

$$f(x) - P_{n-1}(x) = \underbrace{\frac{f^{(n)}(\xi)}{n!}(x-x_1)\cdots(x-x_n)}_{\text{interpolation error}}$$

for some ξ between the x_i and x.

Suppose we wish to minimize the maximal error on some interval $[a,b]$. After shifting and scaling, we can normalize this interval to the interval $[-1,1]$.

It therefore is natural to choose $x_1, \dots, x_n$ such that $\max_{x\in[-1,1]} |(x-x_1)\cdots(x-x_n)|$ is minimized.

Amazingly, in Theorem 91, we will be able to say exactly for which choice of x_i this happens!

Example 89. For small n, choose $x_1, x_2, \dots, x_n$ such that $\max_{x\in[-1,1]} |(x-x_1)\cdots(x-x_n)|$ is minimal.

Solution. In the cases below, we will appeal to symmetry and assume that the optimal nodes must be such that $x_1 = -x_n$, $x_2 = -x_{n-1}$, ... As such, the arguments only prove that our choices are optimal if that assumption is correct. In hindsight, from our general proof in Theorem 91, this will prove to be correct.

- $n=1$: By symmetry, the optimal choice should be $x_1 = 0$.
- $n=2$: By symmetry, $x_1 = -x_2$. Write $c = x_2$ and let $f(x) = (x+c)(x-c) = x^2 - c^2$.
  Since $f'(x) = 2x = 0$ only if $x = 0$, it follows that $\max_{x\in[-1,1]} |f(x)|$ has to occur at $x=0$ or at the endpoints $x = \pm 1$. The corresponding values are $|f(0)| = c^2$, $|f(\pm 1)| = 1 - c^2$.
  From a plot of $m(c) = \max(c^2, 1-c^2)$ it is clear that the minimum of $m(c)$ is achieved when $c^2 = 1-c^2$.
  This latter equation has the unique positive solution $c = \frac{\sqrt{2}}{2} = \cos\left(\frac{\pi}{4}\right)$.
  Note that the x_i are $\underbrace{\cos\left(\frac{\pi}{4}\right)}_{\sqrt{2}/2}, \underbrace{\cos\left(\frac{3\pi}{4}\right)}_{-\sqrt{2}/2}$.
- $n=3$: By symmetry, $x_1 = -x_3$ and $x_2 = 0$. Write $c = x_3$ and let $f(x) = (x+c)x(x-c) = x(x^2 - c^2)$.
  Since $f'(x) = 3x^2 - c^2 = 0$ only if $x = \pm\frac{c}{\sqrt{3}}$, it follows that $\max_{x\in[-1,1]} |f(x)|$ has to occur at $x = \pm\frac{c}{\sqrt{3}}$ or at the endpoints $x = \pm 1$. The corresponding values are $\left|f\left(\frac{c}{\sqrt{3}}\right)\right| = \frac{|c|}{\sqrt{3}}\left(\frac{2c^2}{3}\right) = \frac{2|c|^3}{3\sqrt{3}}$, $|f(\pm 1)| = 1 - c^2$.
  From a plot of $m(c) = \max\left(\frac{2|c|^3}{3\sqrt{3}}, 1 - c^2\right)$ it is clear that the minimum of $m(c)$ is achieved when $\frac{2|c|^3}{3\sqrt{3}} = 1 - c^2$. This latter equation has the unique positive solution $c = \frac{\sqrt{3}}{2} = \cos\left(\frac{\pi}{6}\right)$.
  Note that the x_i are $\underbrace{\cos\left(\frac{\pi}{6}\right)}_{\sqrt{3}/2}, \underbrace{\cos\left(\frac{3\pi}{6}\right)}_{0}, \underbrace{\cos\left(\frac{5\pi}{6}\right)}_{-\sqrt{3}/2}$.
- $n=4$: The pattern continues and the x_i turn out to be $\cos\left(\frac{\pi}{8}\right), \cos\left(\frac{3\pi}{8}\right), \cos\left(\frac{5\pi}{8}\right), \cos\left(\frac{7\pi}{8}\right)$.

  Comment. We have the less familiar trig values $\cos\left(\frac{\pi}{8}\right) = \frac{1}{2}\sqrt{2+\sqrt{2}}$ and $\cos\left(\frac{3\pi}{8}\right) = \frac{1}{2}\sqrt{2-\sqrt{2}}$.

Example 90. (bonus!) Suppose we are doing interpolation on the interval $[-1,1]$ and we want the endpoints to be interpolation nodes; that is, $x_1=-1$ and $x_n=1$. Choose the remaining nodes such that $\max_{x\in[-1,1]} |(x-x_1)\cdots(x-x_n)|$ is minimal.

Do this for $n=1,2,3$ to collect a bonus point.

An extra bonus point if you can figure out what happens for any n? (*Hint*: compare with the Chebyshev case.)

Theorem 91. (Chebyshev's theorem) For the **Chebyshev nodes**

$$x_j=\cos\left(\frac{(2j-1)}{2n}\pi\right), \quad j=1,\ldots,n,$$

we have

$$\max_{x\in[-1,1]} |(x-x_1)\cdots(x-x_n)| = \frac{1}{2^{n-1}}.$$

That value is the minimal value for any choice of roots $x_1,\ldots,x_n$.

The corresponding polynomials $T_n(x)=2^{n-1}(x-x_1)\cdots(x-x_n)$ are known as the **Chebyshev polynomials** of the first kind.

Note that these are scaled by 2^{n-1} so that the maximum is 1.

Proof. We will show below that $T_n(\cos(\theta))=\cos(n\theta)$, which implies that $|T_n(x)|\leqslant 1$ for all $x\in[-1,1]$.
Moreover, at $x=\cos\left(k\frac{\pi}{n}\right)$ for $k=0,1,\ldots,n$ the values of $T_n(x)$ alternate between 1 and -1.
Write $P_n(x)=(x-x_1)\cdots(x-x_n)$. It follows that $\max_{x\in[-1,1]} |P_n(x)|=\frac{1}{2^{n-1}}$ as claimed.
Suppose that there is a polynomial $Q_n(x)=(x-r_1)\cdots(x-r_n)$ for which $\max_{x\in[-1,1]} |Q_n(x)|<\frac{1}{2^{n-1}}$.
Note that $d(x):=P_n(x)-Q_n(x)$ has the following properties:

- $d(x)$ is of degree at most $n-1$ (because the x^n terms cancel).
- At $x=\cos\left(k\frac{\pi}{n}\right)$ for $k=0,1,\ldots,n$ the values of $d(x)$ alternate between $+$ and $-$.
  (Because $P_n(x)=\pm\frac{1}{2^{n-1}}$ while $|Q_n(x)|<\frac{1}{2^{n-1}}$.)
- Hence, between these $n+1$ values, there must be n zeros. That is impossible because $d(x)$ has degree less than n.

This contradiction shows that no such polynomial $Q_n(x)$ can exist. □

The following is Theorem 83 combined with Chebyshev's Theorem 91.

Theorem 92. (interpolation error using Chebyshev nodes) If P_{n-1} is the interpolating polynomial for f at n Chebyshev nodes, then the interpolation error can be bounded as

$$\max_{x\in[-,1,1]} |f(x)-P_{n-1}(x)| \leqslant \frac{1}{2^{n-1}\,n!} \max_{\xi\in[-,1,1]} |f^{(n)}(\xi)|.$$

Fine print. As in Theorem 83, we need that f is n times continuously differentiable.

Comment. Theorem 92 guarantees convergence, as $n\to\infty$, of the interpolating polynomials P_n to f provided that the derivatives of f don't grow too fast. On the other hand, one can show that, for certain functions f, no sequence of interpolating polynomials will converge to f.

Advanced comment. Theorem 92 can be interpreted as showing that, for a given function f, the Chebyshev interpolant P_n is a good approximation of f on the interval $[-1,1]$. However, that does not mean that it is the best polynomial approximation of degree n (in the sense of minimizing the maximal error). One can show that there exists a unique such best polynomial B_n. However, B_n is difficult to compute. On the other hand, the Chebyshev interpolant Q_n is close to best in the sense that

$$\max_{x\in[-,1,1]} |f(x)-Q_n(x)| \leqslant \left(4+\frac{4}{\pi^2}\log(n)\right) \max_{x\in[-,1,1]} |f(x)-B_n(x)|.$$

Example 93. Suppose we approximate a function $f(x)$ on the interval $[-1,1]$ by a polynomial interpolation $P(x)$. Suppose that we know that $|f^{(n)}(x)| \leqslant n$ for all $x\in[-1,1]$.

(a) Give an upper bound for the maximal error if we use the interpolation nodes $-1, -\frac{1}{3}, \frac{1}{3}, 1$.

(b) Give an upper bound for the maximal error if we use 4 Chebyshev nodes instead.

(c) How many Chebyshev nodes do we need to use in order to guarantee that the maximal error is at most 10^{-3}?

Solution.

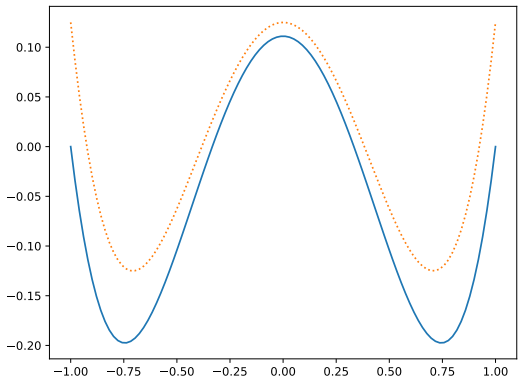
(a) This is the same problem as in the last part of Example 85.

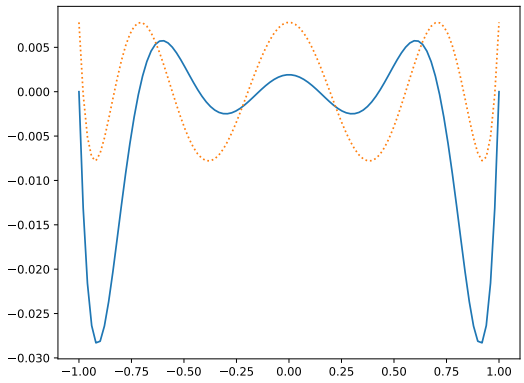
Our bound for the error was $|f(x)-P(x)| \leqslant \frac{1}{6} \max_{z\in[-1,1]} \left|(z^2-1)\left(z^2-\frac{1}{9}\right)\right| = \frac{1}{6}\cdot\frac{16}{81} \approx 0.0329$.

(b) By Theorem 91, $\max_{x\in[-1,1]} |(x-x_1)\cdots(x-x_n)| = \frac{1}{2^{n-1}}$ for Chebyshev nodes. In our case, $n=4$.

Therefore, our bound for the error is $|f(x)-P(x)| \leqslant \frac{1}{6}\frac{1}{2^{4-1}} = \frac{1}{48} \approx 0.0208$.

Comment. This bound is better than the one for the same number of equally spaced points. Indeed, for the Chebyshev nodes, this error estimate is best possible. In the plots below, we can see the difference between $(x-x_1)\cdots(x-x_n)$ in the case of equally spaced x_i and Chebyshev nodes x_i (in dotted). The first plot shows the case $n=4$ and the difference is moderate. The difference becomes very visible in the second plot which shows the case $n=8$. We can see how, for the equally spaced nodes, we get large (negative) values towards the endpoints of $[-1,1]$ while, for the Chebyshev nodes, there are no such wild swings.





(c) By Theorem 92, using n Chebyshev nodes, the error is bounded as

$$\max_{x\in[-,1,1]} |f(x)-P_{n-1}(x)| \leqslant \frac{1}{2^{n-1}\,n!} \max_{\xi\in[-,1,1]} |f^{(n)}(\xi)| \leqslant \frac{1}{2^{n-1}\,(n-1)!}.$$

We thus need to choose n so that $2^{n-1}\,(n-1)! \geqslant 10^3$.

Computing $2^{n-1}\,(n-1)!$ for $n=1,2,\ldots$, we obtain $1, 2, 8, 48, 384, 3840$. Thus, for $n=6$ Chebyshev nodes the maximal error is guaranteed to be less than 10^{-3}.

Example 94. Suppose we approximate $\sin(x)$ on the interval $[-1,1]$ by a polynomial interpolation $P(x)$. How many Chebyshev nodes do we need to use in order to guarantee that the maximal error is at most 10^{-16}?

Solution. In the case $f(x)=\sin(x)$ we know that $|f^{(n)}(x)|\leqslant 1$ for all $x\in[-1,1]$ and all n.

Therefore, by Theorem 92, using n Chebyshev nodes, the error is bounded as

$$\max_{x\in[-,1,1]}|f(x)-P_{n-1}(x)|\leqslant\frac{1}{2^{n-1}\,n!}\max_{\xi\in[-,1,1]}|f^{(n)}(\xi)|\leqslant\frac{1}{2^{n-1}\,n!}.$$

We need to choose n so that $2^{n-1}\,n!\geqslant 10^{16}$. We compute $2^{n-1}\,n!$ for $n=1,\,2,\,\ldots$ and find that this first happens when $n=15$ (see the next Python example).

Thus, for $n=15$ Chebyshev nodes the maximal error is guaranteed to be less than 10^{-16}.

Comment. Recall that double precision floats have a precision of slightly less than 16 decimal digits. Thus we could, in theory, implement an accurate $\sin(x)$ function by using a degree 14 polynomial.

Advanced comment. Note that some care is required to translate this result into practice. For instance, if we proceed as in Example 87 then the resulting maximal error using 15 Chebyshev nodes is about $4.7\cdot 10^{-11}$, which is considerably larger than 10^{-16} (even if we take into account that we are limited by using double precision floats). The cause for this is that the computed interpolating polynomial is not accurate to full precision. Indeed, in the documentation for the function `interpolate.lagrange`, we find the following statement: *Warning: This implementation is numerically unstable. Do not expect to be able to use more than about 20 points even if they are chosen optimally.*

Example 95. Python A function for computing $n!$ is available in the Python `math` library (as well as in `numpy` and `scipy`). However, here is a possible quick implementation from scratch:

```
>>> def factorial(n):
        f = 1
        for k in range(1,n+1):
            f = f*k
        return f

>>> factorial(3)

   6
```

For the computation in the previous example, we now increase n until $2^{n-1}\,n!\geqslant 10^{16}$.

```
>>> n = 1

>>> while 2**(n-1) * factorial(n) < 10**16:
        n = n+1

>>> n

   15

>>> 2**(n-1) * factorial(n)

   21424936845312000

>>> 2.**(n-1) * factorial(n)

   2.1424936845312e+16
```

Example 96. Python Let us redo Example 88 but with Chebyshev nodes instead of equally spaced interpolation nodes.

```
>>> def chebyshev_nodes(n):
        return [cos((2*j+1)*pi/(2*n)) for j in range(n)]

>>> chebyshev_nodes(3)
```

```
[0.8660254037844387, 6.123233995736766e-17, -0.8660254037844387]
```

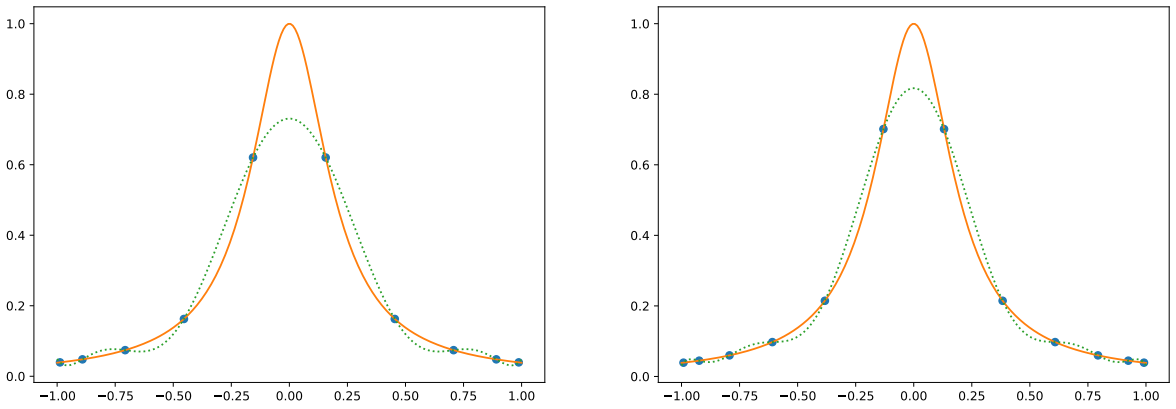
We observed in Example 88 that the maximal interpolation error for the Runge function $f(x) = 1/(1+25x^2)$ did not go down as we increased the number of (equally spaced) interpolation nodes.

```
>>> def f(x):
        return 1/(1+25*x**2)

>>> [max_interpolation_error(f, -1, 1, chebyshev_nodes(n), 100) for n in range(2,18)]
```

```
[0.9233816556226488, 0.60057189596611948, 0.74778034684079508, 0.40195613012685899,
0.5534788672877784, 0.26410513077643449, 0.38946847488552683, 0.17006563147899745,
0.26712486571968486, 0.10902564197574982, 0.1809557278548104, 0.06902642915187851,
0.12185501126173093, 0.0460893689663045, 0.081815311151541836, 0.032580232210393967]
```

Unlike in Example 88, these values suggest that, by increasing the number of Chebyshev nodes, the maximal interpolation error will go to zero.

For comparison with Example 88, the following plots are for 10 and 12 interpolation nodes.



Comment. Note how we no longer have oscillations towards the endpoints of the interval. These plots also reveal why (as we can see from the above list of maximal errors) an even number of Chebyshev nodes leads to a relatively worse interpolation error compared to an odd number. (Namely, for an odd number of nodes, we have a node at $x = 0$, the peak of our function; while, for an even number of nodes, that peak is underestimated by the interpolation.)

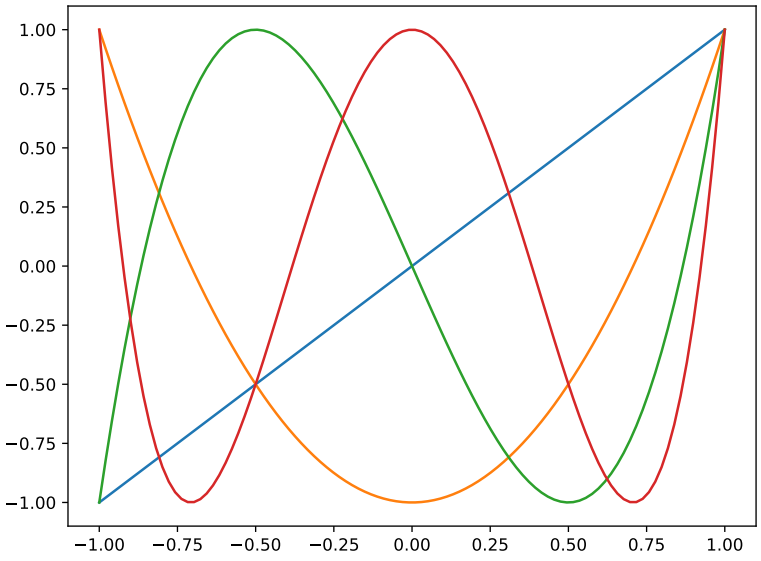
Chebyshev polynomials

As introduced after Chebyshev's Theorem 91, the **Chebyshev polynomials** of the first kind are

$$T_n(x) = 2^{n-1}(x-x_1)\cdots(x-x_n), \quad x_j = \cos\left(\frac{(2j-1)}{2n}\pi\right).$$

These are scaled by 2^{n-1} so that the maximum is 1. Indeed, $T_n(1) = 1$.

We can see in the following plot of $T_n(x)$ for $n \in \{1, 2, 3, 4\}$ that the Chebyshev polynomials alternate between the values ± 1. The goal of this section is to prove this and other properties.



Review. (trig identities through Euler) By **Euler's identity**, $e^{i\theta} = \cos(\theta) + i\sin(\theta)$. In other words, $\cos(\theta) = \mathrm{Re}(e^{i\theta})$ and $\sin(\theta) = \mathrm{Im}(e^{i\theta})$ are "parts" of the exponential function.

All of the trig function identities can then be obtained from simpler identities of the exponential function. For instance, the exponential function satisfies $e^{A+B} = e^A e^B$. For the cosine, this relation translates into

$$\cos(\alpha+\beta) = \mathrm{Re}(e^{i\alpha}e^{i\beta}) = \mathrm{Re}(\cos(\alpha)+i\sin(\alpha)(\cos(\beta)+i\sin(\beta))) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta).$$

Theorem 97. The **Chebyshev polynomials** $T_n(x)$ of the first kind satisfy:

(a) $T_n(\cos(\theta)) = \cos(n\theta)$

Equivalently, $T_n(x) = \cos(n\arccos(x))$.

(b) $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$

This Fibonacci-like recursive relation together with $T_0(x) = 1$ and $T_1(x) = x$ characterizes $T_n(x)$.

Proof.

(a) It follows from trig identities (see the first part of the next example) that $\cos(n\theta)$ can be written as a polynomial in $\cos(\theta)$. In other words, there is a (unique) polynomial $p_n(x)$ such that $\cos(n\theta) = p_n(\cos(\theta))$. We need to show that $T_n(x) = p_n(x)$. Since both are polynomials of degree n, this follows if we can show that they agree at $n+1$ points.

By definition, for $j \in \{1, 2, \ldots, n\}$, $T_n(x)$ has a root at $x_j = \cos(\theta_j)$ where $\theta_j = \frac{(2j-1)}{2n}\pi$.

On the other hand, $p_n(x_j) = p(\cos(\theta_j)) = \cos(n\theta_j) = \cos\Big(\big(j - \frac{1}{2}\big)\pi\Big) = 0$.

$T_n(x)$ and $p_n(x)$ therefore have the same n roots. It follows that they are the same if they have the same leading coefficient. For $T_n(x)$ it is clear from the definition $T_n(x) = 2^{n-1}(x-x_1)\cdots(x-x_n)$ that the leading coefficient is 2^{n-1}. That the same is true for $p_n(x)$ follows from the recursive relation for $p_n(x)$ that we show in the second part.

(b) It follows from the trig identity $\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$ (which we derived above) that

$$\begin{aligned} \cos((n+1)\theta) &= \cos(n\theta+\theta) = \cos(n\theta)\cos(\theta) - \sin(n\theta)\sin(\theta), \\ \cos((n-1)\theta) &= \cos(n\theta-\theta) = \cos(n\theta)\cos(\theta) + \sin(n\theta)\sin(\theta), \end{aligned}$$

where we used that $\sin(-\theta) = -\sin(\theta)$ for the last term. Adding these two, and then writing $T_n(x) = \cos(n\theta)$ with $\theta = \arccos(x)$, we obtain

$$\underbrace{\cos((n+1)\theta)}_{T_{n+1}(x)} + \underbrace{\cos((n-1)\theta)}_{T_{n-1}(x)} = 2\underbrace{\cos(n\theta)}_{T_n(x)}\underbrace{\cos(\theta)}_{x},$$

which is the claimed recursive relation. □

Example 98. Determine the first few Chebyshev polynomials $T_n(x)$.

Solution. (using cosines) We use $T_n(x) = \cos(n\theta)$ with $x = \cos(\theta)$ combined with Euler's identity $e^{i\theta} = \cos(\theta) + i\sin(\theta)$ as well as the trig identity $\cos(\theta)^2 + \sin(\theta)^2 = 1$.

- $e^{2i\theta} = (e^{i\theta})^2 = (\cos(\theta) + i\sin(\theta))^2$ has real part $\cos(2\theta) = \cos(\theta)^2 - \sin(\theta)^2 = 2\cos(\theta)^2 - 1$.
  Hence $T_2(x) = 2x^2 - 1$.
- $e^{3i\theta} = (\cos(\theta) + i\sin(\theta))^3$ has real part $\cos(3\theta) = \cos(\theta)^3 - 3\cos(\theta)\sin(\theta)^2 = 4\cos(\theta)^3 - 3\cos(\theta)$.
  Hence $T_3(x) = 4x^3 - 3x$.

Solution. (using recursion) Starting with $T_0(x) = 1$ and $T_1(x) = x$, we apply $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$ to compute $T_2(x), T_3(x), \ldots$

- $T_2(x) = 2xT_1(x) - T_0(x) = 2x^2 - 1$
- $T_3(x) = 2xT_2(x) - T_1(x) = 2x(2x^2-1) - x = 4x^3 - 3x$
- $T_4(x) = 2xT_3(x) - T_2(x) = 2x(4x^3 - 3x) - (2x^2 - 1) = 8x^4 - 8x^2 + 1$
- ...

(Cubic) splines: piecewise polynomial interpolation

If, given data points $(x_0, y_0), \ldots, (x_n, y_n)$ (also called **knots**) with $x_0 < x_1 < \ldots < x_n$, we connect them via straight lines, then we obtain what is called a **linear spline**. This linear spline interpolates the given points but it is not a polynomial; instead it is a piecewise polynomial (namely, in this case, it is a line on each segment $[x_i, x_{i+1}]$).

A problematic feature of linear splines is their lack of smoothness. Between each linear piece, we usually have a sharp corner at which the spline is not differentiable.

C^n smooth. We say that a function is C^n smooth if its nth derivative exists and is continuous.

For instance, linear splines are C^0 (continuous) but not C^1 (unless the spline is a single line).

Of particular importance are **cubic splines** which are piecewise polynomials where each piece is a polynomial of degree 3 (or less).

On each segment $[x_i, x_{i+1}]$, we therefore have 4 degrees of freedom (coming from the 4 coefficients of a cubic polynomial). We need 2 of these to interpolate at the two endpoints. This leaves us with $4 - 2 = 2$ degrees of freedom which we can use to achieve smoothness. This allows us to demand that the first and the second derivative agree with the neighboring pieces. Thus the resulting spline will be C^2 smooth.

A **cubic spline** $S(x)$ through $(x_0, y_0), \ldots, (x_n, y_n)$ with $x_0 < x_1 < \ldots < x_n$ is piecewise defined by n cubic polynomials $S_1(x), \ldots, S_n(x)$ such that $S(x) = S_i(x)$ for $x \in [x_{i-1}, x_i]$. Moreover:

- $S(x)$ interpolates the given points.

This means that $S_i(x_{i-1}) = y_{i-1}$ and $S_i(x_i) = y_i$ for $i \in \{1, \ldots, n\}$. ($2n$ equations)

- $S(x)$ is C^2 smooth (i.e. it has a continuous second derivative).

This means that $S_i'(x_i) = S_{i+1}'(x_i)$ and $S_i''(x_i) = S_{i+1}''(x_i)$ for $i \in \{1, \ldots, n-1\}$. ($2n - 2$ equations)

Note that there are $4n$ degrees of freedom for such a spline $S(x)$, while we only have $2n + (2n - 2) = 4n - 2$ equations. To define a unique spline, we therefore need to impose 2 more constraints. These are usually chosen as **boundary conditions**.

The following are common choices for the **boundary conditions of cubic splines**:

- **natural**: $S_1''(x_0) = S_n''(x_n) = 0$

The resulting splines are simply called **natural cubic splines**.

- **not-a-knot**: $S_1'''(x_1) = S_2'''(x_1)$ and $S_n'''(x_{n-1}) = S_{n-1}'''(x_{n-1})$
- **periodic**: $S_1'(x_0) = S_n'(x_n)$ and $S_1''(x_0) = S_n''(x_n)$ (only makes sense if $y_0 = y_n$)

There are other common choices such **clamped cubic splines** for which the first derivatives at the endpoints are being set ("clamped") to user-specified values.

Comment. The name "natural" comes from the fact that the resulting spline is what one gets if one pins thin (idealized) elastic strips in the position of the given knots.

Comment. The "not-a-knot" condition has the consequence that $S_1 = S_2$ and $S_n = S_{n-1}$ (because cubic polynomials must be equal if they agree up to the third derivative at a point). Hence x_1 and x_{n-1} are no longer "knots" between separate polynomials.

As illustrated in the next example, we can compute splines (with 2 boundary conditions) by spelling out the $4n$ defining equations. We can then solve the resulting linear equations using linear algebra. There are, of course, various clever observations that make this approach more efficient. However, since we have other sights to see on our journey, we will not get into these aspects.

Review. By Taylor's theorem (Theorem 48), applied to a function $f(x)$ around $x=x_0$, we have

$$f(x) = \underbrace{f(x_0)+f'(x_0)(x-x_0)+\cdots+\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n}_{n\text{th Taylor polynomial}} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}}_{\text{error}}.$$

If $f(x)$ is a polynomial of degree n, then $f^{(n+1)}(x)=0$ so that $f(x)$ is equal to its nth Taylor polynomial. In particular, if $f(x)$ is a cubic polynomial then, for any x_0,

$$f(x) = a(x-x_0)^3+b(x-x_0)^2+c(x-x_0)+d,$$

with $a=\frac{1}{6}f^{(3)}(x_0)$, $b=\frac{1}{2}f^{(2)}(x_0)$, $c=f'(x_0)$ and $d=f(x_0)$.

Example 99. Determine the natural cubic spline through $(-1,2)$, $(1,0)$, $(2,5)$.

Solution. Let us write the spline as $S(x)=\begin{cases} S_1(x), & \text{if } x\in[-1,1], \\ S_2(x), & \text{if } x\in[1,2]. \end{cases}$

To simplify our life, we expand both S_i around $x=1$ (the middle knot).

$$S_i(x) = a_i(x-1)^3+b_i(x-1)^2+c_i(x-1)+d_i.$$

- As noted in the review above, $d_i=S_i(1)$, $c_i=S_i'(1)$ and $b_i=\frac{1}{2}S_i''(1)$. Because $S(x)$ is C^2 smooth, we have $b_1=b_2$, $c_1=c_2$ and $d_1=d_2$. We simply write b, c and d for these values in the sequel.
- $d=0$ because $S_1(1)=S_2(1)=0$.
- $S(x)$ further interpolates the other two points, $(-1,2)$ and $(2,5)$, resulting in the following two equations:

$$\begin{aligned} S_1(-1) &= -8a_1+4b-2c = 2 \\ S_2(2) &= a_2+b+c = 5 \end{aligned}$$

- The natural boundary conditions provide two more equations: (Note that $S_i''(x)=6a_i(x-1)+2b_i$.)

$$\begin{aligned} S_1''(-1) &= -12a_1+2b=0 \\ S_2''(2) &= 6a_2+2b=0 \end{aligned}$$

We use these last two equations to replace $a_1=\frac{1}{6}b$ and $a_2=-\frac{1}{3}b$ in the other two equations in terms of b:

$$\begin{aligned} -8\cdot\frac{1}{6}b+4b-2c &= \frac{8}{3}b-2c=2 \\ -\frac{1}{3}b+b+c &= \frac{2}{3}b+c=5 \end{aligned}$$

Solving these two equations in two unknowns, we find $b=3$ and $c=3$.

Consequently, $a_1=\frac{1}{6}b=\frac{1}{2}$ and $a_2=-\frac{1}{3}b=-1$.

Hence, the desired natural cubic spline is

$$S(x)=3(x-1)+3(x-1)^2+(x-1)^3\begin{cases} \frac{1}{2}, & \text{if } x\in[-1,1], \\ -1, & \text{if } x\in[1,2]. \end{cases}$$

Example 100. Under which conditions is

$$S(x) = \begin{cases} S_1(x), & \text{if } x \in [0,a], \\ S_2(x), & \text{if } x \in [a,b], \end{cases}$$

is a cubic spline? A natural cubic spline?

Solution. $S(x)$ is a cubic spline if $S_1(x)$ and $S_2(x)$ are cubic polynomials such that

$$S_1(a) = S_2(a), \quad S_1'(a) = S_2'(a), \quad S_1''(a) = S_2''(a).$$

$S(x)$ is a natural cubic spline if, in addition, $S_1''(0) = 0$ and $S_2''(b) = 0$.

Comment. Together with the three conditions coming from prescribing the values $S(0)$, $S(a)$ and $S(b)$, these are 8 conditions in order for $S(x)$ to be a natural cubic spline. 8 equations are just the right number to uniquely determine the underlying $2 \cdot 4 = 8$ unknowns.

Example 101. The following function $S(x)$ is a cubic spline.

$$S(x) = -1 - \frac{2}{9}(a-5)x - \frac{1}{3}(2a-1)x^2 + \frac{1}{36}x^3 \begin{cases} (-10a-13), & \text{if } x \in [-2,0], \\ 8(4a+7), & \text{if } x \in [0,1]. \end{cases}$$

(a) Spell out the conditions we need to check to see that this is a cubic spline.

(b) What are the underlying data points?

(c) Is there a choice of a such that $S(x)$ is a natural cubic spline?

Solution.

(a) Write $S_1(x)$ for $S(x)$ on $[-2, 0]$ and $S_2(x)$ for $S(x)$ on $[0, 1]$. Then, as in the previous example, the conditions for $S(x)$ to be a cubic spline are

$$S_1(0) = S_2(0), \quad S_1'(0) = S_2'(0), \quad S_1''(0) = S_2''(0).$$

These conditions are visibly satisfied since the formulas for $S_1(x)$ and $S_2(x)$ agree up to a multiple of x^3.

(b) The knots of the spline are $-2, 0, 1$. We compute $S(-2) = 1$, $S(0) = -1$, $S(1) = 2$.
Hence the data points are $(-2, 1)$, $(0, -1)$, $(1, 2)$.

(c) In order for $S(x)$ to be a natural spline, we need $S''(-2) = 0$ as well as $S''(1) = 0$. Using

$$S''(x) = -\frac{2}{3}(2a-1) + \frac{1}{6}x \begin{cases} (-10a-13), & \text{if } x \in [-2,0], \\ 8(4a+7), & \text{if } x \in [0,1], \end{cases}$$

we have $S''(-2) = -\frac{2}{3}(2a-1) - \frac{1}{3}(-10a-13) = 2a+5$ and $S''(1) = -\frac{2}{3}(2a-1) + \frac{4}{3}(4a+7) = 4a+10$.
Both of these are 0 if and only if $a = -\frac{5}{2}$. Therefore, $S(x)$ is a natural cubic spline if $a = -\frac{5}{2}$.

Comment. Can you explain why do the two segments of the spline only differ in the cubic term?
[*Hint*: Note that 0 is a knot and look again at the first part.]

Example 102. Python Let us construct cubic splines using Python with `scipy`.

```
>>> from scipy import linspace, interpolate
```

Comment. Many basic functions like `linspace` are provided by both `numpy` and `scipy`.

We start by defining the data points that we wish to interpolate.

```
>>> xpoints = [1, 2, 4, 5, 7]
```

```
>>> ypoints = [2, 1, 4, 3, 2]
```

We can then construct the cubic spline with natural boundary conditions as follows.

```
>>> spline = interpolate.CubicSpline(xpoints, ypoints, bc_type='natural')
```

Comment. Other standard choices for the boundary conditions include `'not-a-knot'` (the default) as well as `'clamped'` and `'periodic'` (this one requires the first and last point to have the same y-coordinates).
https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.CubicSpline.html

The resulting natural cubic spline is piecewise defined by a collection of cubic polynomials. We can plot it as we did in Example 78 (this time we also include a legend).

```
>>> import matplotlib.pyplot as plt
```

```
>>> xplot = linspace(1, 7, 100)
```

```
>>> plt.plot(xplot, spline(xplot), '-', label='spline (natural)')
```

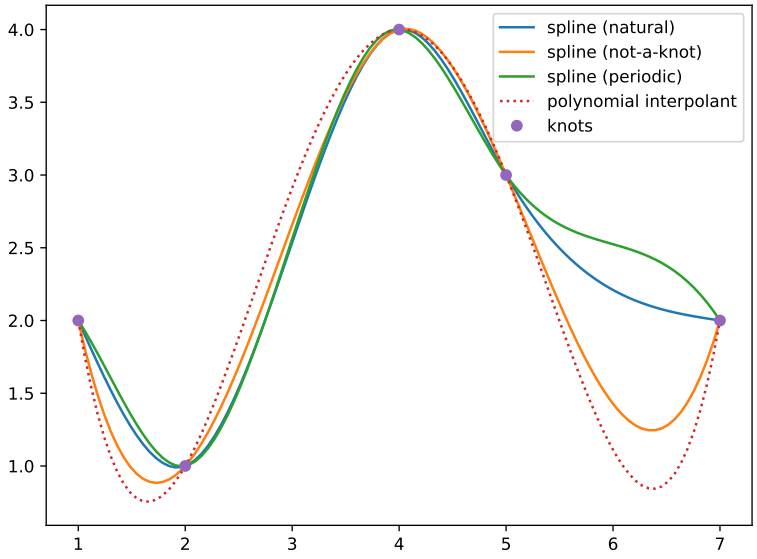
```
>>> plt.plot(xpoints, ypoints, 'o', label='knots')
```

```
>>> plt.legend()
```

```
>>> plt.show()
```

The resulting plot is a simpler version of the following one where we also included two other cubic splines as well as the polynomial interpolant:

Homework. Can you reproduce this plot?



Can you identify (some of) the splines without the labels? Try other knots and plot the splines!

For instance. The periodic spline is easily identified here because of the matching derivatives at the endpoints.
The natural spline is the one that is most like a clothesline pinned to the knots.
The not-a-knot spline is closer to polynomial interpolation.

If desired, we can access the piecewise polynomials as follows:

```
>>> spline.c

   [[ 6.37096774e-01 -6.49193548e-01  8.54838710e-01 -9.67741935e-02]
    [ 2.22044605e-16  1.91129032e+00 -1.98387097e+00  5.80645161e-01]
    [-1.63709677e+00  2.74193548e-01  1.29032258e-01 -1.27419355e+00]
    [ 2.00000000e+00  1.00000000e+00  4.00000000e+00  3.00000000e+00]]

>>>
```

For instance, the first column refers to $2 - 1.637(x-1) + 0.637(x-1)^3$ (the cubic used on $[1,2]$, the first interval) while the fourth column encodes $3 - 1.274(x-5) + 0.581(x-5)^2 - 0.097(x-5)^3$ (the cubic used on $[5,7]$, the last interval).

Comment. The exact cubics are $2 - \frac{203}{124}(x-1) + \frac{79}{124}(x-1)^3$ and $3 - \frac{79}{62}(x-5) + \frac{18}{31}(x-5)^2 - \frac{3}{31}(x-5)^3$.

Note how, for the first one, $S_1(x)$, we can immediately see that $S_1''(1) = 0$. Because we created a natural cubic spline, we also have $S_4''(7) = 0$. (Check it from the above exact formula!)

Example 103. In the case of four nodes/knots, how is the polynomial interpolant related to the cubic splines?

Solution. Note that the polynomial interpolant for four nodes is a cubic polynomial.

On the other hand, each cubic spline consists of three cubic polynomials S_1, S_2, S_3. In the case of the not-a-knot cubic spline, we have $S_1 = S_2$ as well as $S_3 = S_2$, which implies that all three are equal so that the not-a-knot cubic spline is a single cubic polynomial (interpolating the four given points).

Therefore, the polynomial interpolant must equal the not-a-knot cubic spline in this case.

Working with functions: differentiation + integration

Numerical differentiation

We know from Calculus that $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$.

To numerically approximate $f'(x)$ we could use $f'(x) \approx \frac{1}{h}[f(x+h) - f(x)]$ for small h.

In this section, we analyze this and other ways of numerically differentiating a function.

Application. These approximations are crucial for developing tools to numerically solve (partial) differential equations by discretizing them.

Review. We can express **Taylor's theorem** (Theorem 48) in the following manner:

$$f(x+h) = \underbrace{f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \ldots + \frac{1}{n!}f^{(n)}(x)h^n}_{\text{Taylor polynomial}} + \underbrace{\frac{1}{(n+1)!}f^{(n+1)}(\xi)h^{n+1}}_{\text{error}}$$

This form is particularly convenient for the kind of error analysis that we are doing here.

Important notation. When the exact form of the error is not so important, we simply write $O(h^{n+1})$.

Definition 104. We write $e(h) = O(h^n)$ if there is a constant C such that $|e(h)| \leqslant Ch^n$ for all small enough h.

For our purposes, $e(h)$ is usually an error term and this notation allows us to talk about that error without being more precise than necessary.

If $e(h)$ is the error, then we often say that an approximation is of order n if $e(h) = O(h^n)$.

Caution. This notion of order is different from the order of convergence that we discussed in the context of fixed-point iteration and Newton's method.

Example 105. Determine the order of the approximation $f'(x) \approx \frac{1}{h}[f(x+h) - f(x)]$.

Comment. This approximation of the derivative is called a **(first) forward difference** for $f'(x)$.

Likewise, $f'(x) \approx \frac{1}{h}[f(x) - f(x-h)]$ is a **(first) backward difference** for $f'(x)$.

Solution. By Taylor's theorem, $f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)$. It follows that

$$\frac{1}{h}[f(x+h) - f(x)] = f'(x) + \boxed{\frac{h}{2}f''(x) + O(h^2)} = f'(x) + \boxed{O(h)}.$$

Hence, the $\boxed{\text{error}}$ is of order 1.

Comment. The presence of the term $\frac{h}{2}f''(x)$ tells us that the order is exactly 1 unless $f''(x) = 0$ (that is, the order cannot generally be improved to δ for some $\delta < 1$).

Example 106. Determine the order of the approximation $f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$.

Comment. This approximation of the derivative is called a **(first) central difference** for $f'(x)$.

Solution. By Taylor's theorem (Theorem 48),

$$\begin{aligned} f(x+h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5), \\ f(x-h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \frac{h^4}{24}f^{(4)}(x) + O(h^5). \end{aligned} \tag{2}$$

(Note that the second formula just has h replaced with $-h$.) Subtracting the second from the first, we obtain

$$\frac{1}{2h}[f(x+h) - f(x-h)] = f'(x) + \boxed{\frac{h^2}{6}f'''(x) + O(h^3)} = f'(x) + \boxed{O(h^2)}.$$

Hence, the $\boxed{\text{error}}$ is of order 2.

Example 107. Use both forward and central differences to approximate $f'(x)$ for $f(x) = x^2$.

Solution. We get $\frac{1}{h}[f(x+h) - f(x)] = 2x + h$ and $\frac{1}{2h}[f(x+h) - f(x-h)] = 2x$.

Comment. In the forward difference case, the error is of order 1 (also note that $\frac{h}{2}f''(x) = h$). In the central difference case, we find that we get $f'(x)$ without error. In hindsight, with the error formulas in mind, this is not a surprise and reflects the fact that $f'''(x) = 0$.

Example 108. Use both forward and central differences to approximate $f'(2)$ for $f(x) = 1/x$.

Solution. In each case, we use $h = \frac{1}{10}$ and $h = \frac{1}{20}$.

- $h = \frac{1}{10}$: $\frac{1}{h}[f(x+h) - f(x)] = -\frac{5}{21} \approx -0.2381$, error 0.0119
  $h = \frac{1}{20}$: $\frac{1}{h}[f(x+h) - f(x)] = -\frac{10}{41} \approx -0.2439$, error 0.0061 (reduced by about $\frac{1}{2}$)
- $h = \frac{1}{10}$: $\frac{1}{2h}[f(x+h) - f(x-h)] = -\frac{100}{399} \approx -0.25063$, error -0.00063
  $h = \frac{1}{20}$: $\frac{1}{2h}[f(x+h) - f(x-h)] = -\frac{400}{1599} \approx -0.25016$, error -0.00016 (reduced by about $\frac{1}{4}$)

Important comment. The forward difference has an error of order 1. In other words, for small h, it should behave like Ch. In particular, if we replace h by $h/2$, then the error should be about $1/2$ (as we saw above).
On the other hand, the central difference has an error of order 2 and so should behave like Ch^2. In particular, if we replace h by $h/2$, then the error should be about $1/2^2 = 1/4$ (and, again, this is what we saw above).

Example 109. Find a central difference for $f''(x)$ and determine the order of the error.

Solution. Adding the two expansions in (2) to kill the $f'(x)$ terms, and subtracting $2f(x)$, we find that

$$\frac{1}{h^2}[f(x+h) - 2f(x) + f(x-h)] = f''(x) + \boxed{\frac{h^2}{12}f^{(4)}(x) + O(h^3)} = f''(x) + \boxed{O(h^2)}.$$

The $\boxed{\text{error}}$ is of order 2.

Alternatively. If we iterate the approximation $f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$ (in the second step, we apply it with x replaced by $x \pm h$), we obtain

$$f''(x) \approx \frac{1}{2h}[f'(x+h) - f'(x-h)] \approx \frac{1}{4h^2}[f(x+2h) - 2f(x) + f(x-2h)],$$

which is the same as what we found above, just with h replaced by $2h$.

Armin Straub
straub@southalabama.edu

Example 110. Obtain approximations for $f'(x)$ and $f''(x)$ using the values $f(x)$, $f(x+h)$, $f(x+2h)$ as follows: determine the polynomial interpolation corresponding to these values and then use its derivatives to approximate those of f. In each case, determine the order of the approximation and the leading term of the error.

Solution. We first compute the polynomial $p(t)$ that interpolates the three points $(x, f(x))$, $(x+h, f(x+h))$, $(x+2h, f(x+2h))$ using Newton's divided differences:

	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$
x	$f(x)$		
		$\dfrac{f(x+h)-f(x)}{h} =: c_1$	
$x+h$	$f(x+h)$		$\dfrac{f(x+2h)-2f(x+h)+f(x)}{2h^2} =: c_2$
		$\dfrac{f(x+2h)-f(x+h)}{h}$	
$x+2h$	$f(x+2h)$		

Hence, reading the coefficients from the top edge of the triangle, the interpolating polynomial is

$$p(t) = f(x) + c_1(t-x) + c_2(t-x)(t-x-h).$$

- **(approximating $f'(x)$)** Since $p'(t) = c_1 + c_2(2t-2x-h)$, we have

$$\begin{aligned} p'(x) &= c_1 - hc_2 = \frac{f(x+h)-f(x)}{h} - \frac{f(x+2h)-2f(x+h)+f(x)}{2h} \\ &= \frac{-f(x+2h)+4f(x+h)-3f(x)}{2h}. \end{aligned}$$

  This is our approximation for $f'(x)$. To determine the order and the error, we combine

$$\begin{aligned} f(x+h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{f'''(x)}{6}h^3 + O(h^4), \\ f(x+2h) &= f(x) + 2f'(x)h + 2f''(x)h^2 + \frac{4f'''(x)}{3}h^3 + O(h^4) \end{aligned}$$

  (note that the latter is just the former with h replaced by $2h$) to find

$$-f(x+2h)+4f(x+h)-3f(x) = 2f'(x)h - \frac{2f'''(x)}{3}h^3 + O(h^4).$$

  Hence, dividing by $2h$, we conclude that

$$\frac{-f(x+2h)+4f(x+h)-3f(x)}{2h} = f'(x) - \frac{f'''(x)}{3}h^2 + O(h^3).$$

  Consequently, the approximation is of order 2.

- **(approximating $f''(x)$)** Since $p''(t) = 2c_2$, we have $p''(x) = 2c_2 = \dfrac{f(x+2h)-2f(x+h)+f(x)}{h^2}$.
  This is our approximation for $f''(x)$. To determine the order and the error, we proceed as before to find

$$f(x+2h) - 2f(x+h) + f(x) = f''(x)h^2 + f'''(x)h^3 + O(h^4).$$

  Hence, dividing by h^2, we conclude that

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

  Consequently, the approximation is of order 1.

Example 111. (homework) Obtain approximations for $f'(x)$ and $f''(x)$ using the values $f(x-2h)$, $f(x)$, $f(x+3h)$ as follows: determine the polynomial interpolation corresponding to these values and then use its derivatives to approximate those of f. In each case, determine the order of the approximation and the leading term of the error.

Solution. We first compute the polynomial $p(t)$ that interpolates the three points $(x-2h, f(x-2h))$, $(x, f(x))$, $(x+3h, f(x+3h))$ using Newton's divided differences:

	$f[\cdot]$	$f[\cdot,\cdot]$	$f[\cdot,\cdot,\cdot]$
$x-2h$	$f(x-2h)$		
		$\frac{f(x)-f(x-2h)}{2h} =: c_1$	
x	$f(x)$		$\frac{2f(x+3h)-5f(x)+3f(x-2h)}{30h^2} =: c_2$
		$\frac{f(x+3h)-f(x)}{3h}$	
$x+3h$	$f(x+3h)$		

Hence, reading the coefficients from the top edge of the triangle, the interpolating polynomial is

$$p(t) = f(x) + c_1(t-x+2h) + c_2(t-x+2h)(t-x).$$

- **(approximating $f'(x)$)** Since $p'(t) = c_1 + c_2(2t-2x+2h)$, we have

$$\begin{aligned} p'(x) &= c_1 + 2h\,c_2 = \frac{f(x)-f(x-2h)}{2h} + \frac{2f(x+3h)-5f(x)+3f(x-2h)}{15h} \\ &= \frac{4f(x+3h)+5f(x)-9f(x-2h)}{30h}. \end{aligned}$$

  This is our approximation for $f'(x)$. To determine the order and the error, we combine

$$\begin{aligned} f(x+h) &= f(x) + f'(x)h + \frac{1}{2}f''(x)h^2 + \frac{f'''(x)}{6}h^3 + O(h^4), \\ f(x-2h) &= f(x) - 2f'(x)h + 2f''(x)h^2 - \frac{4f'''(x)}{3}h^3 + O(h^4), \\ f(x+3h) &= f(x) + 3f'(x)h + \frac{9}{2}f''(x)h^2 + \frac{9f'''(x)}{2}h^3 + O(h^4) \end{aligned}$$

  to find

$$4f(x+3h) + 5f(x) - 9f(x-2h) = 30f'(x)h + 30f'''(x)h^3 + O(h^4).$$

  Hence, dividing by $30h$, we conclude that

$$\frac{4f(x+3h)+5f(x)-9f(x-2h)}{30h} = f'(x) + f'''(x)h^2 + O(h^3).$$

  Consequently, the approximation is of order 2.

- **(approximating $f''(x)$)** Since $p''(t) = 2c_2$, we have $p''(x) = 2c_2 = \frac{2f(x+3h)-5f(x)+3f(x-2h)}{15h^2}$.
  This is our approximation for $f''(x)$. To determine the order and the error, we proceed as before to find

$$2f(x+3h) - 5f(x) + 3f(x-2h) = 15f''(x)h^2 + 5f'''(x)h^3 + O(h^4).$$

  Hence, dividing by $15h^2$, we conclude that

$$\frac{2f(x+3h)-5f(x)+3f(x-2h)}{15h^2} = f''(x) + \frac{1}{3}f'''(x)h + O(h^2).$$

  Consequently, the approximation is of order 1.

Python assignment #4

The goal of this assignment is to gain some first acquaintance with recursion as a coding technique. First, in the case of the factorial function, we observe that recursion provides a simple and elegant way to implement certain kinds of functions. In a second example, we then encounter a potential issue of recursion that one needs to be aware of.

Example 112. Python We can quickly implement a function for computing $n!$ in an iterative fashion as follows (such a function is also available in the Python `math` library as well as in `numpy` and `scipy`).

```
>>> def factorial_iterative(n):
        f = 1
        for k in range(1,n+1):
            f = f*k
        return f

>>> factorial_iterative(3)

    6
```

On the other hand, a recursive implementation would take the following form:

```
>>> def factorial_recursive(n):
        if n == 0: return 1
        return n * factorial_recursive(n-1)

>>> factorial_recursive(4)

    24
```

Sometimes it is useful to measure how fast code is running. One tool for doing this in Python is the `timeit` module. The following measures how many seconds it takes to compute $100!$ using our two functions 10^4 many times.

```
>>> from timeit import timeit

>>> timeit('factorial_iterative(100)', number=10**4, globals=globals())

    0.051031902898103

>>> timeit('factorial_recursive(100)', number=10**4, globals=globals())

    0.11693716002628207
```

As we can see, the recursive implementation is a bit slower (by about a factor of two in this case of computing $100!$) than the iterative implementation. However, unless performance was critical, such a difference should be considered relatively minor and not a reason for us to go out of our way to avoid one or the other (the time spent coding can be much more valuable than the milliseconds saved by optimized code).

Comment. Finally, just out of curiousity, here is a comparison with the factorial function implemented in the standard `math` library that comes with Python. Not surprisingly, that is the fastest (at about 8 times faster than our iterative implementation).

```
>>> from math import factorial

>>> timeit('factorial(100)', number=10**4, globals=globals())

    0.0072873481549322605
```

Advanced comment. If you try our recursive function on a large input, you will run into an error about exceeding the recursion limit. If necessary, that limit can be increased using the function `setrecursionlimit` from the `sys` module.

Example 113. Python The Fibonacci numbers F_n are defined by the formula $F_{n+1} = F_n + F_{n-1}$ together with the initial conditions $F_0 = F_1 = 1$. Below is a recursive implementation that directly mirrors the mathematical description.

```
>>> def fibonacci_recursive(n):
        if n == 0: return 0
        if n == 1: return 1
        return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

>>> [fibonacci_recursive(n) for n in range(10)]

   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

However, there is trouble!

```
>>> fibonacci_recursive(34)

   5702887
```

This computation of F_{34} took more than a second (at least on my little laptop). And the computation of, say, F_{40} will take ages. Try it and see your machine sweat!

(a) Can you explain why the recursive implementation above is becoming so slow?

(b) Provide an iterative implementation of the Fibonacci numbers that avoids the above issue.

Advanced comment. An alternative approach around our issue is to keep the recursive logic but to store previously computed values. Typically, one uses a dictionary of previously computed values and, at the beginning of the function, checks whether the current function input has occurred before. In Python one can also add this behaviour to a function by using a suitable "decorator".

https://docs.python.org/3/library/functools.html#functools.cache

After you have submitted your code on Replit, send me an email with the following:

(a) The exact value of $100!$ (the value we used for the timings in the first example).

(b) The exact value of F_{100}.

(c) A sentence or two explaining why the recursive implementation of the Fibonacci numbers becomes unusably slow so quickly.

Richardson extrapolation

Suppose that $A(h)$ is an approximation of a quantity A^* of order n. Our goal is to construct an approximation of A^* of higher order (by combining approximations $A(h)$ for different h).

If $A(h)$ approximates A^* to order n, then we often have $A(h) = A^* + Ch^n + O(h^{n+1})$ for some constant C. This is true, for instance, for all of our numerical differentiation examples.

In that case, we have $A(2h) = A^* + C(2h)^n + O(h^{n+1})$ as well as $2^n A(h) = 2^n A^* + C(2h)^n + O(h^{n+1})$.

Thus the difference is $2^n A(h) - A(2h) = (2^n - 1)A^* + O(h^{n+1})$.

Dividing both sides by $2^n - 1$, we therefore get an approximation of order higher than n.

(Richardson extrapolation) Starting with an approximation $A(h)$ of a quantity A^* of order n, its **Richardson extrapolation** is the approximation

$$R(h) := \frac{2^n A(h) - A(2h)}{2^n - 1}.$$

As we showed above, it typically is an approximation of higher order.

More generally, we get a Richardson extrapolation $R_\lambda(h) = \frac{\lambda^n A(h) - A(\lambda h)}{\lambda^n - 1}$ for any choice of $\lambda > 0$ (the choice $\lambda = 2$ is common but not mathematically special).

Comment. Note that, based on the values at h and $2h$, we are trying to get our hands on A^* which is the value at 0. Because 0 is outside of the interval $[h, 2h]$, this is an extrapolation.

Example 114. Suppose that $A\left(\frac{1}{2}\right) = \frac{3}{8}$ and $A\left(\frac{1}{3}\right) = \frac{5}{12}$ are approximations of order 3 of some quantity A^*. What is the approximation we obtain from using Richardson extrapolation?

Solution. Since $A(h)$ is an approximation of order 3, we expect $A(h) \approx A^* + Ch^3$ for some constant C.

Correspondingly, $A\left(\frac{1}{2}\right) \approx A^* + \frac{1}{8}C$ and $A\left(\frac{1}{3}\right) \approx A^* + \frac{1}{27}C$.

Hence, $27A\left(\frac{1}{3}\right) - 8A\left(\frac{1}{2}\right) \approx (27-8)A^* = 19A^*$. To get an approximation of A^*, we need to divide by 19.

The Richardson extrapolation is $\frac{27}{19}A\left(\frac{1}{3}\right) - \frac{8}{19}A\left(\frac{1}{2}\right) = \frac{27}{19}\frac{5}{12} - \frac{8}{19}\frac{3}{8} = \frac{33}{76}$.

Comment. Note that we did is equivalent to using the formula $R_\lambda(h) = \frac{\lambda^n A(h) - A(\lambda h)}{\lambda^n - 1}$ with $h = \frac{1}{3}$ and $\lambda = \frac{3}{2}$.

Comment. The numbers above are not random. Instead, $A(h) = \frac{1}{2} - \frac{h}{4}$ which is an order 3 approximation of $f(0) = \frac{1}{2}$ for $f(x) = \frac{1}{1+e^x} = \frac{1}{2} - \frac{x}{4} + \frac{x^3}{48} + O(x^5)$. Indeed, $\frac{33}{76} \approx 0.434$ is slightly better than $A\left(\frac{1}{2}\right)$ and $A\left(\frac{1}{3}\right)$.

Example 115. Apply Richardson extrapolation to $f'(x) \approx \frac{1}{h}[f(x+h) - f(x)]$.

Solution. We have seen in Example 105 that the approximation $A(h) = \frac{1}{h}[f(x+h) - f(x)]$ is of order 1. Hence,

$$\begin{aligned} \frac{2A(h) - A(2h)}{2-1} &= \frac{2}{h}[f(x+h) - f(x)] - \frac{1}{2h}[f(x+2h) - f(x)] \\ &= \frac{1}{2h}[-f(x+2h) + 4f(x+h) - 3f(x)] \end{aligned}$$

is an approximation of higher order (we expect it to be of order 2).

Indeed, this approximation of $f'(x)$ is the same as what we obtained in Example 110 when applying polynomial interpolation to $f(x)$, $f(x+h)$, $f(x+2h)$. As observed there, the error is $-\frac{1}{3}f'''(x)h^2 + O(h^3)$ showing that this is indeed an approximation of order 2.

Example 116. Apply Richardson extrapolation to $f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$.

Solution. We have seen in Example 106 that the approximation $A(h) = \frac{1}{2h}[f(x+h) - f(x-h)]$ is of order 2. Hence,

$$\begin{aligned} \frac{2^2 A(h) - A(2h)}{2^2 - 1} &= \frac{2}{3h}[f(x+h) - f(x-h)] - \frac{1}{12h}[f(x+2h) - f(x-2h)] \\ &= \frac{1}{12h}[-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)] \end{aligned}$$

is an approximation of $f'(x)$ of higher order. With some more work (do it!), we find that the error is $-\frac{1}{30}f^{(5)}(x)h^4 + O(h^6)$ so that this is an approximation of order 4.

Comment. Note that the above approximations don't change when h is replaced by $-h$ (note the factor of $1/h$). In other words, the approximations are even functions in h. Consequently, their Taylor expansion in h will only have even powers of h. That's the reason why the order of the Richardson extrapolation is 4 rather than order 3 which is what one would otherwise expect when extrapolating an order 2 formula.

Example 117. Apply Richardson extrapolation to $f''(x) \approx \frac{1}{h^2}[f(x+h) - 2f(x) + f(x-h)]$.

Solution. We have seen in Example 109 that the approximation $A(h) = \frac{1}{h^2}[f(x+h) - 2f(x) + f(x-h)]$ is of order 2. Hence,

$$\frac{2^2 A(h) - A(2h)}{2^2 - 1} = \frac{1}{12h^2}[-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)]$$

is an approximation of $f''(x)$ of higher order. With some more work, we find that the error is $-\frac{1}{90}f^{(6)}(x)h^4 + O(h^6)$ so that this is an approximation of order 4.

In the previous example, we combined $A(h)$ and $A(2h)$ to obtain an approximation of higher order. There is nothing special about $2h$. We can likewise combine $A(h_1)$ and $A(h_2)$ for any h_1, h_2.

Example 118. (homework) $A(h) = \frac{1}{h^2}[f(x+h) - 2f(x) + f(x-h)]$ is an order 2 approximation of $f''(x)$. Apply Richardson extrapolation to $A(h)$ and $A\left(\frac{3}{2}h\right)$ to obtain an approximation of $f''(x)$ of higher order.

Solution. Since $A(h)$ is an approximation of order 2, we expect $A(h) \approx A^* + Ch^2$ for some constant C. Correspondingly, $A\left(\frac{3}{2}h\right) \approx A^* + \frac{9}{4}Ch^2$. Hence, $\frac{9}{4}A(h) - A\left(\frac{3}{2}h\right) \approx \left(\frac{9}{4} - 1\right)A^* = \frac{5}{4}A^*$.

The Richardson extrapolation of $A(h)$ and $A\left(\frac{3}{2}h\right)$ therefore is:

$$\frac{\frac{9}{4}A(h) - A\left(\frac{3}{2}h\right)}{\frac{5}{4}} = \frac{1}{45h^2}\left[-16f\left(x + \tfrac{3}{2}h\right) + 81f(x+h) - 130f(x) + 81f(x-h) - 16f\left(x - \tfrac{3}{2}h\right)\right]$$

This is an approximation of $f''(x)$ of higher order. With some more work, we find that the error is $-\frac{1}{160}f^{(6)}(x)h^4 + O(h^6)$ so that this is an approximation of order 4.

Example 119. Python Let us see how the forward and central difference compare in practice.

```
>>> def forward_difference(f, x, h):
        return (f(x+h)-f(x))/h

>>> def central_difference(f, x, h):
        return (f(x+h)-f(x-h))/(2*h)
```

We apply these to $f(x)=2^x$ at $x=1$. In that case, the exact derivative is $f'(1)=2\ln(2)\approx 1.386$.

```
>>> def f(x):
        return 2**x

>>> [forward_difference(f, 1, 10**-n) for n in range(5)]

   [2.0, 1.4354692507258626, 1.3911100113437769, 1.3867749251610384, 1.3863424075299946]

>>> [central_difference(f, 1, 10**-n) for n in range(5)]

   [1.5, 1.3874047099948572, 1.3863054619682957, 1.3862944721280135, 1.3862943622289237]
```

It is probably easier to see what happens to the error if we subtract the true value from these approximations:

```
>>> from math import log

>>> [forward_difference(f, 1, 10**-n) - 2*log(2) for n in range(12)]

   [0.6137056388801094, 0.04917488960597205, 0.004815650223886303, 0.0004805640411478 2403,
   4.80464101040301e-05, 4.804564444071957e-06, 4.80467807983942e-07, 4.703673917028084e-
   08, 7.068710283775204e-09, 2.7352223619381277e-07, 1.161700655893938e-06,
   1.8925269049896443e-05]

>>> [central_difference(f, 1, 10**-n) - 2*log(2) for n in range(12)]

   [0.11370563888010943, 0.001110348874966638, 1.1100848405165564e-05,
   1.1100812291608975e-07, 1.1090330875873633e-09, 1.879385536085465e-11,
   7.43052286367174e-11, -7.028508886008922e-10, 7.068710283775204e-09,
   1.6249993373129712e-07, 1.161700655893938e-06, 7.823038803644877e-06]
```

For the forward difference, we can see how the error decreases roughly by $1/10$ initially, as expected. Likewise, for the central difference, the error decreases roughly by $1/10^2$ (order 2) initially. However, in both cases, the errors end up increasing after a while before getting close to machine precision. We discuss this in the next section.

Note how, for the forward difference, our best approximation has error $7.07\cdot 10^{-9}$ while, for the central difference, our best approximation has error $1.88\cdot 10^{-11}$. While the latter is an improvement, either is worryingly large.

Example 120. Python Let us now repeat Example 119 with the formula

$$f'(x) \approx \frac{1}{12h}[-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)]$$

that we obtained in Example 116.

```
>>> def central_difference_richardson(f, x, h):
        return (-f(x+2*h)+8*f(x+h)-8*f(x-h)+f(x-2*h))/(12*h)
```

Let us again approximate $f'(1) = 2\ln(2) \approx 1.386$ for $f(x) = 2^x$ at $x = 1$.

```
>>> [central_difference_richardson(f, 1, 10**-n) for n in range(5)]
```

```
[1.375, 1.3862932938249581, 1.3862943610132332, 1.3862943611198109, 1.3862943611187004]
```

Does the error behave as expected?

```
>>> [central_difference_richardson(f, 1, 10**-n) - 2*log(2) for n in range(6)]
```

```
[-0.011294361119890572, -1.0672949324330716e-06, -1.0665734961889939e-10, -
7.971401316808624e-14, -1.1901590823981678e-12, 1.5093037930569153e-11]
```

We noted in Example 116 that the approximation is of order 4. Indeed, we can see how the error decreases roughly by $1/10^4$ initially, as expected.

Moreover, we are able to obtain a much better numerical estimate compared to Example 119: this time, our best approximation has error $7.97 \cdot 10^{-14}$, which is decently close to the machine precision of $\varepsilon \approx 2^{-52} \approx 2.2 \cdot 10^{-16}$. This is because the effect of rounding errors becomes devastating as h becomes very small. Using a high-order approximation, we are often able to avoid having to work with very small h.

Indeed, note how we got the best approximation with $h = 10^{-3}$ (whereas we previously needed to a much smaller h for the best approximations).

The errors in numerical differentiation

In practice, we always get two kinds of errors: the **theoretical error** as well as a **rounding error** (due to the fact that we have to round all involved quantities to, say, double precision). In the past, we have been able to mostly ignore the rounding error. However, we cannot afford to do so in the case of numerical differentiation. The reason for the trouble is that our finite difference approximations always subtract nearly equal quantities (that's in the nature of differentiation!) which can lead to a devastating loss of precision.

Comment. The theoretical error is often also called **truncation error**. Here, truncation is meant in the sense of, for instance, having a function $f(x)$ and truncating its Taylor series to get an approximation of $f(x)$.

Example 121. Analyze the overall error in using the approximation $f'(x) \approx \frac{1}{h}[f(x+h) - f(x)]$.

Solution. As we worked out in Example 105, the theoretical error is $\frac{1}{h}[f(x+h) - f(x)] = f'(x) + \frac{h}{2}f''(\xi)$.

In practice, all quantities including $f(x+h)$ and $f(x)$ are slightly rounded and only accurate to within some ε (the machine precision). In the sequel, we assume double precision, in which case $\varepsilon \approx 2^{-52} \approx 2.2 \cdot 10^{-16}$. Our approximation therefore is (ε_1 and ε_2 are the rounding errors for $f(x+h)$ and $f(x)$ respectively) roughly:

$$\begin{aligned} \frac{1}{h}[(f(x+h)+\varepsilon_1) - (f(x)+\varepsilon_2)] &= \frac{1}{h}[f(x) - f(x-h)] + \frac{\varepsilon_1 - \varepsilon_2}{h} \\ &= f'(x) + \underbrace{\frac{1}{2}f''(\xi)h}_{\text{theoretical error}} + \underbrace{\frac{\varepsilon_1 - \varepsilon_2}{h}}_{\text{rounding error}} \end{aligned}$$

Note that $\left|\frac{\varepsilon_1 - \varepsilon_2}{h}\right| \leqslant \frac{2\varepsilon}{h}$. Writing $M = \frac{1}{2}|f''(\xi)|$, the overall error is bounded by

$$E(h) = Mh + \frac{2\varepsilon}{h}.$$

Plot the function $E(h)$ to see that this bound becomes bad as h gets too small (that's because of the rounding error $2\varepsilon/h$). In particular, we can see that there must be a "best" choice for h which minimizes this bound for the error. To find this best value $h^\star$, we compute $E'(h) = M - \frac{2\varepsilon}{h^2} = 0$ and solve for h. We find

$$h^* = \sqrt{\frac{2\varepsilon}{M}} \approx M^{-1/2} \cdot 2.1 \cdot 10^{-8}.$$

The corresponding bound for the error is $E(h^*) \approx Mh^* + \frac{2\varepsilon}{h^*} \approx M^{1/2} \cdot 4.2 \cdot 10^{-8}$.

Comment. Note that our estimates match the values we observed in Example 119 fairly well.

Example 122. Analyze the overall error in approximating $f'(x) \approx \frac{1}{2h}[f(x+h) - f(x-h)]$.

Solution. The theoretical error is $\frac{1}{2h}[f(x+h) - f(x-h)] = f'(x) + \frac{h^2}{6}f'''(x) + O(h^3)$ (see Example 106).

In practice, proceeding as in the previous example, our approximation is roughly:

$$\begin{aligned} \frac{1}{2h}[(f(x+h)+\varepsilon_1) - (f(x-h)+\varepsilon_2)] &= \frac{1}{2h}[f(x+h) - f(x-h)] + \frac{\varepsilon_1 - \varepsilon_2}{2h} \\ &= f'(x) + \underbrace{\frac{1}{6}f'''(\xi)h^2}_{\text{theoretical error}} + \underbrace{\frac{\varepsilon_1 - \varepsilon_2}{2h}}_{\text{rounding error}} \end{aligned}$$

Note that $\left|\frac{\varepsilon_1 - \varepsilon_2}{2h}\right| \leqslant \frac{\varepsilon}{h}$. Writing $M = \frac{1}{6}|f'''(\xi)|$, the overall error is bounded by

$$E(h) = Mh^2 + \frac{\varepsilon}{h}.$$

Again, plotting the function $E(h)$ we see that this bound becomes bad as h gets too small and that there is a "best" value h^* which minimizes this bound for the error. We compute $E'(h) = 2Mh - \frac{\varepsilon}{h^2} = 0$ and solve for h. We find

$$h^* = \sqrt[3]{\frac{\varepsilon}{2M}} \approx M^{-1/3} \cdot 4.8 \cdot 10^{-6}.$$

The corresponding bound for the error is $E(h^*) \approx M\,(h^*)^2 + \frac{\varepsilon}{h^*} \approx M^{1/3} \cdot 6.9 \cdot 10^{-11}$.

Comment. Again, our estimates match the values we observed in Example 119 rather well.

Numerical integration (also known as quadrature)

To numerically integrate a function $f(x)$ on an interval $[a,b]$, one usually uses approximations of the form

$$\int_a^b f(x)\mathrm{d}x \quad \approx \quad w_0 f(x_0) + \cdots + w_n f(x_n) = \sum_{i=0}^{n} w_i f(x_i),$$

where the points x_i and the weights w_i are chosen appropriately.

Comment. Such **quadrature rules** are typically judged by the maximal degree d of polynomials that they can integrate without error. For instance, to correctly integrate constant functions (degree 0 polynomials), the weights need to be such that they add up to $b-a$. (Why?!)

Common quadrature rules include:

- Newton–Cotes rules: equally spaced points x_i

  These are most useful if $f(x)$ already computed at equally spaced points, or if evaluation is fast.

  There are closed Newton–Cotes rules and open ones. Open means that a and b are not part of the x_i.

- Gaussian quadrature: the x_i are not equally spaced but chosen carefully

  Choosing the x_i is similar to our discussion of Chebyshev nodes in polynomial interpolation.

  Gaussian quadrature is particularly useful if $f(x)$ is expensive to compute.

Comment. In the case of integrable singularities, such as in $\int_0^1 \frac{1}{\sqrt{x}}\mathrm{d}x$, we cannot use closed Newton–Cotes.

Because of time constraints, we will focus on the simplest example of a closed Newton–Cotes rule, namely the trapezoidal rule.

We will then see that combining this with Richardson extrapolation, we can obtain higher order Newton–Cotes rules such as Simpson's rule.

The (composite) trapezoidal rule

Given equally spaced nodes $x_0, x_1, \ldots, x_n$ with $x_0 = a$ and $x_n = b$, we interpolate $f(x)$ on each segment $[x_{i-1}, x_i]$ by a linear function. Writing $h = (b-a)/n$ for the distance between nodes, the resulting integration rule is the following:

(trapezoidal rule) The following is an approximation of order 2:

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)]$$

Why? On each segment $[x_{i-1}, x_i]$, we approximate $f(x)$ by a linear function so that the integral on that segment becomes the area of a trapezoid and we get

$$\int_{x_{i-1}}^{x_i} f(x)\mathrm{d}x \approx \underbrace{\text{width} \cdot \text{average height}}_{\text{area}} = h \cdot \frac{f(x_{i-1}) + f(x_i)}{2} = \frac{h}{2} f(x_{i-1}) + \frac{h}{2} f(x_i).$$

Make a sketch! Adding together the integrals over all segments, each node (except x_0 and x_n) will show up twice (hence the factor of 2 in front of $f(x_1), \ldots, f(x_{n-1})$) and we get the claimed integration rule. The fact that the trapezoidal rule provides an approximation of order 2 is proved in Theorem 123 below.

Sanity check. Note that the weights are $\frac{h}{2}$ for the first and last node, and h for the others. The sum of the weights is $2 \cdot \frac{h}{2} + (n-1) \cdot h = nh = b - a$. That is what we need to integrate constant functions without error. Indeed, from the construction it is clear that the composite trapezoidal rule integrates linear functions exactly.

Armin Straub
straub@southalabama.edu

Theorem 123. (trapezoidal rule with error term) If f is C^2 smooth, then

$$\int_a^b f(x)\mathrm{d}x = \frac{h}{2}[f(x_0)+2f(x_1)+\cdots+2f(x_{n-1})+f(x_n)] - \frac{(b-a)}{12}f''(\xi)h^2$$

for some $\xi \in [a,b]$. In particular, the trapezoidal rule is of order 2.

Proof. On each segment $[x_{i-1}, x_i]$, the error of the interpolation is

$$f(x) = \text{linear approximation} + \frac{1}{2}f''(\xi)(x-x_{i-1})(x-x_i).$$

Hence, when integrating

$$\int_{x_{i-1}}^{x_i} f(x)\mathrm{d}x = \underbrace{\frac{h}{2}f(x_{i-1})+\frac{h}{2}f(x_i)}_{\text{integral of linear approx.}} + \underbrace{\int_{x_{i-1}}^{x_i}\frac{1}{2}f''(\xi)(x-x_{i-1})(x-x_i)\mathrm{d}x}_{\text{error}_i},$$

so that the error when integrating is

$$\text{error}_i = \frac{1}{2}f''(\psi)\underbrace{\int_{x_{i-1}}^{x_i}(x-x_{i-1})(x-x_i)\mathrm{d}x}_{=\int_0^h x(x-h)\mathrm{d}x=\left[\frac{1}{3}x^3-\frac{h}{2}x^2\right]_0^h=-\frac{1}{6}h^3} = -\frac{1}{12}f''(\psi)h^3$$

where ψ is some value between x_{i-1} and x_i. Let us briefly justify the "pulling out" of $f''(\xi)$ even though ξ depends on x. Note that $(x-x_{i-1})(x-x_i)$ is always $\leqslant 0$ in the integral and, therefore, does not change sign. This means that the error integral lies between the corresponding integrals where we replace $f''(\xi)$ with its maximum value M and minimum value L; the values M and L no longer depend on x and therefore can be pulled out of the integral. The above computation then shows that error_i is between $-\frac{1}{12}h^3M$ and $-\frac{1}{12}h^3L$, hence must be equal to $-\frac{1}{12}h^3m$ for some $m \in [L,M]$. Since L and M are the minimum and maximum value of f'' on $[x_{i-1},x_i]$, and since f'' is continuous, it follows that $m = f''(\psi)$ for some ψ.

To get the overall error, we need to add the errors $-\frac{1}{12}f''(\psi_i)h^3$ from each segment $[x_{i-1},x_i]$, where $i=1,2,\ldots,n$ and where $\psi_i \in [x_{i-1},x_i]$. The result is

$$-\frac{1}{12}f''(\psi_1)h^3+\cdots+-\frac{1}{12}f''(\psi_n)h^3 = -\frac{nh}{12}\underbrace{\frac{f''(\psi_1)+\cdots+f''(\psi_n)}{n}}_{=\text{average}=f''(\xi)}h^2 = -\frac{b-a}{12}f''(\xi)h^2,$$

where ξ is some value between a and b. □

Comment. A closer inspection of our proof shows that the $f''(\xi)$ in the error formula converges, as $h \to 0$, to the average value of f'' on $[a,b]$. This means that we have a way to obtain an **error estimate** (rather than only an error bound). This observation is also useful because it shows that the error is of a form that allows us to perform Richardson extrapolation.

Advanced comment. Indeed, using the Euler–Maclaurin one can show that the error is

$$-\frac{f'(b)-f'(a)}{12}h^2+\frac{f'''(b)-f'''(a)}{720}h^4+\ldots-B_{2m}\frac{f^{(2m-1)}(b)-f^{(2m-1)}(a)}{(2m)!}h^{2m}+O(h^6),$$

where the B_{2m} are rational numbers known as Bernoulli numbers (provided, of course, that f is C^{2m-1} smooth). The fact that only even powers of h show up reflects the fact that the trapezoidal rule is symmetric (and therefore correctly integrates $(x-c)^n$ where $c=(a+b)/2$ and n is odd).

Note that this more precise form of the error tells us that the Richardson extrapolation of the trapezoidal rule will be of order 4 (rather than order 3).

Example 124. Use the trapezoidal rule to approximate $\int_1^3 \frac{1}{x}\,\mathrm{d}x = \log(3) \approx 1.09861$.

(a) Use $h=1$ and $h=1/2$.

(b) Using Richardson extrapolation, combine the previous two approximations to obtain an approximation of higher order. What are absolute and relative error?

Comment. We will see in the next section that this is equivalent to using Simpson's rule!

Solution. Let us write $f(x)=\frac{1}{x}$.

(a) $\int_1^3 f(x)\,\mathrm{d}x \approx \frac{h}{2}[f(1)+2f(2)+f(3)] = \frac{1}{2}\left[1+2\cdot\frac{1}{2}+\frac{1}{3}\right] = \frac{7}{6} \approx 1.1667$

Comment. Make a sketch! Can you explain why our approximation (for any h) will be an underestimate of the true value of the integral?

(b) $\int_1^3 f(x)\,\mathrm{d}x \approx \frac{h}{2}\left[f(1)+2f\left(\frac{3}{2}\right)+2f(2)+2f\left(\frac{5}{2}\right)+f(3)\right] = \frac{1}{4}\left[1+2\cdot\frac{2}{3}+2\cdot\frac{1}{2}+2\cdot\frac{2}{5}+\frac{1}{3}\right]$
$= \frac{67}{60} \approx 1.1167$

(c) Let us write $A(h)$ and $A\left(\frac{h}{2}\right)$ for our two approximations, and A^* for the true value of the integral.
Since $A(h)$ is an approximation of order 2, we expect $A(h) \approx A^* + Ch^2$ for some constant C.
Correspondingly, $A\left(\frac{h}{2}\right) \approx A^* + \frac{1}{4}Ch^2$. Hence, $4A\left(\frac{h}{2}\right) - A(h) \approx (4-1)A^* = 3A^*$.
Therefore, the Richardson extrapolation is $\frac{1}{3}\left[4A\left(\frac{h}{2}\right) - A(h)\right] = \frac{1}{3}\left[4\cdot\frac{67}{60} - \frac{7}{6}\right] = \frac{11}{10} = 1.1$.
The absolute error is $|1.1 - \log(3)| \approx 0.00139$ and the relative error is $\left|\frac{1.1-\log(3)}{\log(3)}\right| \approx 0.00126$.

Example 125. `Python` Let us redo Example 124 by implementing the trapezoidal rule.

```
>>> def trapezoidal_rule(f, a, b, n):
        h = (b - a) / n
        integral = (f(a) + f(b)) / 2
        for i in range(1,n):
            integral += f(a + i*h)
        return h*integral

>>> def f(x):
        return 1/x
```

Comment. Writing `x += y` is a useful and common short alternative to `x = x + y`.

Choosing `n` to be 2 and 4 is equivalent to $h = \frac{3-1}{2} = 1$ and $h = \frac{3-1}{4} = \frac{1}{2}$ and so we get the same values as in Example 124:

```
>>> trapezoidal_rule(f, 1, 3, 2)

   1.1666666666666665

>>> trapezoidal_rule(f, 1, 3, 4)

   1.1166666666666667
```

As expected, further increasing n produces better approximations:

```
>>> [trapezoidal_rule(f, 1, 3, 10**n) for n in range(1,4)]

    [1.1015623265623264, 1.0986419169811203, 1.0986125849642736]
```

Indeed, the following convincingly illustrates that the error in the trapezoidal rule is $O(h^2)$.

```
>>> from math import log

>>> [trapezoidal_rule(f, 1, 3, 10**n) - log(3) for n in range(1,6)]

    [0.0029500378942166616, 2.9628313010565677e-05, 2.962961638264261e-07,
    2.9629636522088276e-09, 2.962430301067798e-11]
```

However, note that our computer had to work pretty hard to get the final approximation, because that entailed computing about 10^5 values. We clearly should use a higher order method in order to compute to higher accuracy. One option is to do what we did in the last part of Example 124.

Simpson's rule

Let us spell out what happens in general when we proceed as in the last part of Example 124. We start with $T(h)$, the trapezoidal rule applied with step size h, which is given by

$$T(h) = \frac{h}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)].$$

Then, since $T(h)$ is an approximation of order $N = 2$, the Richardson extrapolation

$$S(h) := \frac{2^N T(h) - T(2h)}{2^N - 1} = \frac{4}{3}T(h) - \frac{1}{3}T(2h)$$

is an approximation of higher order, known as **Simpson's rule**. It takes the following form:

(Simpson's rule) Suppose n is even. The following is an approximation of order 4:

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

Proof. To see this, note that the trapezoidal approximations $T(h)$ and $T(2h)$ are given by

$$\begin{aligned} T(h) &= \frac{h}{2}[f(x_0) + 2f(x_1) + \cdots + 2f(x_{n-1}) + f(x_n)], \\ T(2h) &= h[f(x_0) + 2f(x_2) + \cdots + 2f(x_{n-2}) + f(x_n)]. \end{aligned}$$

Here, we need n to be even so that $T(2h)$ uses the points x_0, x_2, ..., x_{n-2}, x_n. Therefore, the Richardson extrapolation is

$$\begin{aligned} S(h) = \frac{4}{3}T(h) - \frac{1}{3}T(2h) &= \frac{h}{3}[2f(x_0) + 4f(x_1) + \cdots + 4f(x_{n-1}) + 2f(x_n)] \\ &\quad -\frac{h}{3}[f(x_0) + 2f(x_2) + \cdots + 2f(x_{n-2}) + f(x_n)] \\ &= \frac{h}{3}[2f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-2}) + 4f(x_{n-1}) + 2f(x_n)], \end{aligned}$$

where the right-hand side is Simpson's rule. That this is an approximation of order 4 follows because the error of $T(h)$ is an even function in h, so that the order of $S(h)$ increases to 4 (rather than the otherwise expected 3). □

Alternative approach. We can also proceed like for the trapezoidal rule, except that we use quadratic instead of linear interpolations on each segment. More precisely, starting with equally spaced nodes $x_0, x_1, \ldots, x_n$ with n even, we can interpolate $f(x)$ on each segment $[x_{i-1}, x_{i+1}]$, with i odd, by a quadratic function $p(x)$. The integral on that segment works out to be

$$\int_{x_{i-1}}^{x_{i+1}} f(x)\mathrm{d}x \approx \int_{x_{i-1}}^{x_{i+1}} p(x)\mathrm{d}x \overset{\text{work}}{=} \frac{h}{3}[f(x_{i-1}) + 4f(x_i) + f(x_{i+1})],$$

where we wrote $h = (b-a)/n$ for the distance between nodes. Adding together the integrals over all segments, each node x_i, with i odd, will show up once with the above factor of $4h/3$ while x_i, with i even, (except x_0 and x_n) will show up twice with a factor of $h/3$. We thus again get Simpson's integration rule.

Comment. The above rule is often called **Simpson's 1/3 rule**. There is also **Simpson's 3/8 rule** which is derived similarly but is based on a cubic (instead of a quadratic) interpolation; it thus requires an additional node (the resulting error term is of the same order but about half).

Similar to Theorem 123, one can show that the error in Simpson's rule is as follows:

Theorem 126. (Simpson's rule with error term) If f is C^4 smooth, then

$$\int_a^b f(x)\mathrm{d}x = \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 4f(x_{n-1}) + f(x_n)] - \frac{(b-a)}{180} f^{(4)}(\xi) h^4$$

for some $\xi \in [a, b]$.

Comment. In the alternative approach above, we constructed Simpson's rule so that quadratic polynomials would be integrated without error; the error term tells us that, in fact, cubic polynomials are also integrated without error. In other words, the error term is a pleasant surprise!

Example 127. Use Simpson's rule with $h = \frac{1}{2}$ to approximate $\displaystyle\int_1^3 \frac{1}{x}\,\mathrm{d}x = \log(3) \approx 1.09861$.

Solution.

$$\int_1^3 f(x)\,\mathrm{d}x \approx \frac{h}{3}\left[f(1) + 4f\left(\frac{3}{2}\right) + 2f(2) + 4f\left(\frac{5}{2}\right) + f(3)\right] = \frac{1}{6}\left[1 + 4\cdot\frac{2}{3} + 2\cdot\frac{1}{2} + 4\cdot\frac{2}{5} + \frac{1}{3}\right] = \frac{11}{10} = 1.1$$

Comment. Note that $\frac{11}{10}$ is exactly what we obtained in the last part of Example 124.

Indeed, recall that Simpson's rule with $h = \frac{1}{2}$ ($n = 4$) is equivalent to applying Richardson extrapolation to the trapezoidal approximations with $h = \frac{1}{2}$ ($n = 4$) and $h = 1$ ($n = 2$).

Example 128. Python Let us compute $\int_1^3 \frac{1}{x}\,dx = \log(3) \approx 1.09861$ by implementing Simpson's rule. The following code assumes that n is even.

```
>>> def simpson_rule(f, a, b, n):
        h = (b - a) / n
        integral = f(a) + f(b)
        for i in range(1,n):
            if i%2 == 1:
                integral += 4*f(a + i*h)
            else:
                integral += 2*f(a + i*h)
        return h/3*integral

>>> def f(x):
        return 1/x
```

With n set to 4, we obtain $\frac{11}{10}$ as in Examples 124 and 127:

```
>>> simpson_rule(f, 1, 3, 4)

    1.0999999999999999
```

Our approximations (here, $n = 10$ and 100) quickly approach the true value:

```
>>> [simpson_rule(f, 1, 3, 10**n) for n in range(1,3)]

    [1.0986605986605984, 1.0986122939305363]
```

Indeed, the following convincingly illustrates that the error in Simpson's rule is $O(h^4)$.

```
>>> from math import log

>>> [simpson_rule(f, 1, 3, 10**n) - log(3) for n in range(1,6)]

    [4.830999248861545e-05, 5.262426494567762e-09, 5.282441151166495e-13, -
    1.5543122344752192e-15, -7.105427357601002e-15]
```

Example 129. Python Various integration methods are already implemented in scipy.

```
>>> from scipy import integrate
```

For instance, the following is a way to use Simpson's rule with $n = 4$ (so that 5 points are used). The result matches the $\frac{11}{10}$ that we computed ourselves.

```
>>> def f(x):
        return 1/x

>>> xvalues = [1+1/2*i for i in range(5)]

>>> yvalues = [f(x) for x in xvalues]

>>> integrate.simps(yvalues, xvalues)

    1.0999999999999999
```

On the other hand, the following is a convenient way of "general purpose integration", where we only need to specify the end points:

```
>>> integrate.quad(f, 1, 3)

    (1.0986122886681096, 7.555511459798467e-14)
```

The second part of the result is an estimate for the absolute error.

Numerical methods for solving differential equations

The general form of a first-order differential equation (DE) is $y' = f(x, y)$,

Comment. Recall that higher-order differential equations can be written as systems of first-order differential equations: $\boldsymbol{y}' = f(x, \boldsymbol{y})$ in terms of $\boldsymbol{y} = (y_1, y_2, y_3, \ldots)$ where we set $y_1 = y$, $y_2 = y'$, $y_3 = y''$, ...

It therefore is no loss of generality to develop methods for first-order differential equation. While we will focus on the case of a single function $y(x)$, the methods we discuss extend naturally to the case of several functions $\boldsymbol{y}(x) = (y_1(x), y_2(x), \ldots)$.

In order to have a unique solution $y(x)$ that we can numerically approximate, we will add an initial condition. As such, we discuss methods for solving first-order initial value problems (IVPs)

$$y' = f(x, y), \quad y(x_0) = y_0.$$

Comment. Recall from Differential Equations class that such an IVP is guaranteed to have a unique solution under mild assumptions on $f(x, y)$ (for instance, that $f(x, y)$ is smooth around (x_0, y_0)).

Comment. There would be no loss of generality in only considering only initial conditions of the form $y(0) = y_0$. Indeed, suppose the initial condition is $y(x_0) = y_0$. Then, by replacing x by $x + x_0$ in the DE and rewriting the DE in terms of $\tilde{y}(x) = y(x + x_0)$, we obtain an IVP with initial condition $\tilde{y}(0) = y_0$.

Review of the simplest differential equations

Let's start with one of the simplest (and most fundamental) differential equation (DE). It is **first-order** (only a first derivative) and **linear** (with constant coefficients).

Example 130. Solve $y' = 3y$.

Solution. $y(x) = Ce^{3x}$

Check. Indeed, if $y(x) = Ce^{3x}$, then $y'(x) = 3Ce^{3x} = 3y(x)$.

Comment. Recall we can always easily check whether a function solves a differential equation. This means that (although you might be unfamiliar with certain techniques for solving) you can use computer algebra systems to solve differential equations without trust issues.

To describe a unique solution, additional constraints need to be imposed.

Example 131. Solve the **initial value problem** (IVP) $y' = 3y$, $y(0) = 5$.

Solution. This has the unique solution $y(x) = 5e^{3x}$.

The following is a **non-linear** differential equation. In general, such equations are much more complicated than linear ones. We can solve this particular one because it is **separable**.

Example 132. Solve $y' = xy^2$.

Solution. This DE is separable: $\frac{1}{y^2}\mathrm{d}y = x\,\mathrm{d}x$. Integrating, we find $-\frac{1}{y} = \frac{1}{2}x^2 + C$.

Hence, $y = -\frac{1}{\frac{1}{2}x^2 + C} = \frac{2}{D - x^2}$. [Here, $D = -2C$ but that relationship doesn't matter.]

Comment. Note that we did not find the singular solution $y = 0$ (lost when dividing by y^2). We can obtain it from the general solution by letting $D \to \infty$.

Euler's method

Euler's method is a numerical way of approximating the (unique) solution $y(x)$ to the IVP

$$y' = f(x,y), \quad y(x_0) = y_0.$$

It follows from Taylor's theorem (Theorem 48) that

$$y(x+h) = y(x) + y'(x)h + \frac{1}{2}y''(\xi)h^2.$$

Choose a step size $h>0$. Write $x_n = x_0 + nh$. Our goal is to provide approximations y_n of $y(x_n)$ for $n = 1, 2, \ldots$.

Since we know $y(x_0) = y_0$, we approximate

$$\begin{aligned}
y(x_0+h) &\approx y(x_0) + y'(x_0)h \overset{\text{DE}}{=} y(x_0) + f(x_0, y(x_0))h \\
y(x_0+2h) &\approx y(x_0+h) + y'(x_0+h)h \overset{\text{DE}}{=} y(x_0+h) + f(x_0+h, y(x_0+h))h \\
y(x_0+3h) &\approx y(x_0+2h) + y'(x_0+2h)h \overset{\text{DE}}{=} y(x_0+2h) + f(x_0+2h, y(x_0+2h))h \\
&\vdots
\end{aligned}$$

Comments.

- Here we use $y(x+h) \approx y(x) + y'(x)h$ first with $x = x_0$, then with $x = x_0 + h$ and so on.
- Note how, when approximating $y(x_0+mh)$, we use the previous approximation $y(x_0+(m-1)h)$. All other quantities on the right-hand side are known to us.
- Clearly, the error in these approximations will accumulate and the approximations are likely worse as we continue (in other words, our approximations of $y(x)$ will be worse as x gets further away from x_0).

Write $x_n = x_0 + nh$. Our goal is to provide approximations y_n of $y(x_n)$ for $n = 1, 2, \ldots$.

Note that we start with x_0 and y_0 from the initial condition.

In terms of x_n and y_n our above approximations become:

$$y(x_n+h) \approx y(x_n) + \underbrace{y'(x_n)}_{f(x_n, y(x_n))} h \approx y_n + f(x_n, y_n)h =: y_n$$

Two kinds of errors. There are two different errors involved here: in the first approximation, the error is from truncating the Taylor expansion and we know that this **local truncation error** is $O(h^2)$. On the other hand, in the second approximation, we introduce an error because we use the previous approximation y_n instead of $y(x_n)$. Suppose that we approximate $y(x)$ on some interval $[x_0, x_{\max}]$ using n steps (so that $x_n = x_{\max}$). Then the step size is $h = \frac{x_{\max} - x_0}{n}$. We therefore have $n = \frac{x_{\max} - x_0}{h}$ many local truncation errors of size $O(h^2)$. It is therefore natural to expect that the **global error** is $O(nh^2) = O(h)$.

(Euler's method) The following is an order 1 method for solving IVPs:

$$y_{n+1} = y_n + f(x_n, y_n)h$$

Comment. As explained above, being an order 1 method means that Euler's method has a global error that is $O(h)$ (while the local truncation error is $O(h^2)$).

Comment. While Euler's method is rarely used in practice, it serves as the foundation for more powerful extensions such as the Runge–Kutta methods.

Example 133. Consider the IVP $y' = y$, $y(0) = 1$. Approximate the solution $y(x)$ for $x \in [0,1]$ using Euler's method with 4 steps. In particular, what is the approximation for $y(1)$?

Comment. Of course, the real solution is $y(x) = e^x$. In particular, $y(1) = e \approx 2.71828$.

Solution. The step size is $h = \frac{1-0}{4} = \frac{1}{4}$. We apply Euler's method with $f(x,y) = y$:

$$\begin{aligned}
x_0 &= 0 & y_0 &= 1 \\
x_1 &= \frac{1}{4} & y_1 &= y_0 + f(x_0, y_0)h = 1 + \frac{1}{4} = \frac{5}{4} = 1.25 \\
x_2 &= \frac{1}{2} & y_2 &= y_1 + f(x_1, y_1)h = \frac{5}{4} + \frac{5}{4}\cdot\frac{1}{4} = \frac{5^2}{4^2} = 1.5625 \\
x_3 &= \frac{3}{4} & y_3 &= y_2 + f(x_2, y_2)h = \frac{5^2}{4^2} + \frac{5^2}{4^2}\cdot\frac{1}{4} = \frac{5^3}{4^3} \approx 1.9531 \\
x_4 &= 1 & y_4 &= y_3 + f(x_3, y_3)h = \frac{5^3}{4^3} + \frac{5^3}{4^3}\cdot\frac{1}{4} = \frac{5^4}{4^4} \approx 2.4414
\end{aligned}$$

In particular, the approximation for $y(1)$ is $y_4 \approx 2.4414$.

Comment. Can you see that, if instead we start with $h = \frac{1}{n}$, then we similarly get $x_i = \frac{(n+1)^i}{n^i}$ for $i = 0, 1, \ldots, n$. In particular, $y(1) \approx y_n = \frac{(n+1)^n}{n^n} = \left(1 + \frac{1}{n}\right)^n \to e$ as $n \to \infty$. Do you recall how to derive this final limit?

Example 134. Python Let us implement Euler's method to redo and extend Example 133.

```
>>> def euler(f, x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            y0 = y0 + f(x0,y0)*h
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints

>>> def f_y(x, y):
        return y
```

If we choose the number of steps `n` to be 4 and `xmax` to be 1 (because we want $x_n = 1$), then the following matches exactly our computation in Example 133:

```
>>> euler(f_y, 0, 1, 1, 4)

   [1, 1.25, 1.5625, 1.953125, 2.44140625]
```

As expected, increasing the number of steps provides better approximations to the exact solution $y(x) = e^x$ with $y(1) = e \approx 2.718$.

```
>>> euler(f_y, 0, 1, 1, 10)

   [1, 1.1, 1.2100000000000002, 1.3310000000000002, 1.4641000000000002, 1.61051,
   1.7715610000000002, 1.9487171, 2.1435888100000002, 2.357947691, 2.5937424601]

>>> euler(f_y, 0, 1, 1, 100)[-1]

   2.704813829421526
```

If `ypoints` is a list, then its elements can be accessed as `ypoints[0]`, `ypoints[1]`, ... Moreover, we can access the last element as `ypoints[-1]`. For instance, above, we used `euler_e(f, 0, 1, 1, 100)[-1]` to get the last element of the 101 approximations $y_0, y_1, \ldots, y_{100}$. That last element is the approximation of $y(1) = e$.

The following convincingly illustrates that the error in Euler's method is $O(h)$.

```
>>> from math import e
>>> [euler(f_y, 0, 1, 1, 10**n)[-1] - e for n in range(6)]
   [-0.7182818284590451, -0.124539368359045, -0.013467999037519274, -
   0.0013578962231490799, -0.00013590163381849152, -1.3591284549807625e-05]
```

However, note that our computer had to work pretty hard to get the final approximation, because that entailed computing 10^5 values. We clearly need a higher order method in order to compute to higher accuracy.

Taylor methods

(Taylor method of order k) The following is an order k method for solving IVPs:

$$y_{n+1} = y_n + f(x_n, y_n)h + \frac{1}{2}f'(x_n, y_n)h^2 + \cdots + \frac{1}{k!}f^{(k-1)}(x_n, y_n)h^k$$

where $f^{(n)}(x, y)$ is short for $\frac{\mathrm{d}^n}{\mathrm{d}x^n}f(x, y(x))$ (expressed in terms of f and its partial derivatives).

For instance. $f'(x,y) = \frac{\mathrm{d}}{\mathrm{d}x}f(x,y(x)) = f_x(x,y) + f_y(x,y)y'(x) = f_x(x,y) + f_y(x,y)f(x,y)$

Especially for higher derivatives, it is easier to compute these for specific f. See next example.

Comment. As for Euler's method, being an order k method means that the method has a global error that is $O(h^k)$ (while the local truncation error is $O(h^{k+1})$; note that we can see this because we truncate the Taylor expansion of $y(x)$ after h^k so that the next term is $O(h^{k+1})$).

Example 135. Spell out the Taylor method of order 2 for numerically solving the IVP

$$y' = \cos(x)y, \quad y(0) = 1.$$

Solution. The Taylor method of order 2 is based on the Taylor expansion

$$y(x+h) = y(x) + y'(x)h + \frac{1}{2}y''(x)h^2 + O(h^3),$$

where we have a local truncation error of $O(h^3)$ so that the global error will be $O(h^2)$.
From the DE we know that $y'(x) = \cos(x)y$, which is $f(x,y)$. We differentiate this to obtain

$$\begin{aligned} y''(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\cos(x)y = -\sin(x)y + \cos(x)y' = -\sin(x)y + \cos^2(x)\,y \\ &= (-\sin(x) + \cos^2(x))y, \end{aligned}$$

which is $f'(x,y)$. Hence, the Taylor method of order 2 takes the form:

$$\begin{aligned} y_{n+1} &= y_n + f(x_n, y_n)h + \frac{1}{2}f'(x_n, y_n)h^2 \\ &= y_n + \cos(x_n)y_n\,h + \frac{1}{2}((-\sin(x_n) + \cos^2(x_n))y_n)h^2 \end{aligned}$$

For any choice of h, we can therefore compute $(x_1, y_1), (x_2, y_2), \ldots$ starting with (x_0, y_0) by the above recursive formula combined with $x_{n+1} = x_n + h$.

Example 136. Spell out the Taylor method of order 3 for numerically solving the IVP

$$y' = \cos(x)y, \quad y(0) = 1.$$

Solution. The Taylor method of order 3 is based on the Taylor expansion

$$y(x+h) = y(x) + y'(x)h + \tfrac{1}{2}y''(x)h^2 + \tfrac{1}{6}y'''(x)h^3 + O(h^4),$$

where we have a local truncation error of $O(h^4)$ so that the global error will be $O(h^3)$.
From the DE we know that $y'(x) = \cos(x)y$, which is $f(x,y)$. As in the previous example, we differentiate this to obtain

$$\begin{aligned} y''(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\cos(x)y = -\sin(x)y + \cos(x)y' = -\sin(x)y + \cos^2(x)\, y \\ &= (-\sin(x) + \cos^2(x))y, \end{aligned}$$

which is $f'(x,y)$. Once more differentiating this, we obtain

$$\begin{aligned} y'''(x) &= \frac{\mathrm{d}}{\mathrm{d}x}(-\sin(x) + \cos^2(x))y \\ &= (-\cos(x) - 2\cos(x)\sin(x))y + (-\sin(x) + \cos^2(x))y' \\ &= (\cos^2(x) - 3\sin(x) - 1)\cos(x)y, \end{aligned}$$

which is $f''(x,y)$. Hence, the Taylor method of order 3 takes the form:

$$\begin{aligned} y_{n+1} &= y_n + f(x_n,y_n)h + \tfrac{1}{2}f'(x_n,y_n)h^2 + \tfrac{1}{6}f''(x_n,y_n)h^3 \\ &= y_n + \cos(x_n)y_n\, h + \tfrac{1}{2}((-\sin(x_n) + \cos^2(x_n))y_n)h^2 + \tfrac{1}{6}((\cos^2(x_n) - 3\sin(x_n) - 1)\cos(x_n)y_n)h^3 \end{aligned}$$

Comment. The exact solution of this IVP is $y(x) = e^{\sin(x)}$. We use this in the next example for comparison.

Example 137. Python Let us use Python to approximate the solution $y(x)$ of the IVP from the previous example for $x \in [0,2]$.

```
>>> from math import e, cos, sin

>>> def taylor_3_cosy(x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            y0 = y0 + cos(x0)*y0*h + 1/2*(cos(x0)**2-sin(x0))*y0*h**2 + \
                 1/6*(cos(x0)**2-3*sin(x0)-1)*cos(x0)*y0*h**3
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints
```

Since the exact solution is $y(x) = e^{\sin(x)}$, we have $y(2) = e^{\sin(2)} \approx 2.48258$.

```
>>> taylor_3_cosy(0, 1, 2, 4)

   [1, 1.625, 2.3475297541746047, 2.7350418255304874, 2.476391322837691]
```

For comparison, the exact values of the solution at these four steps are:

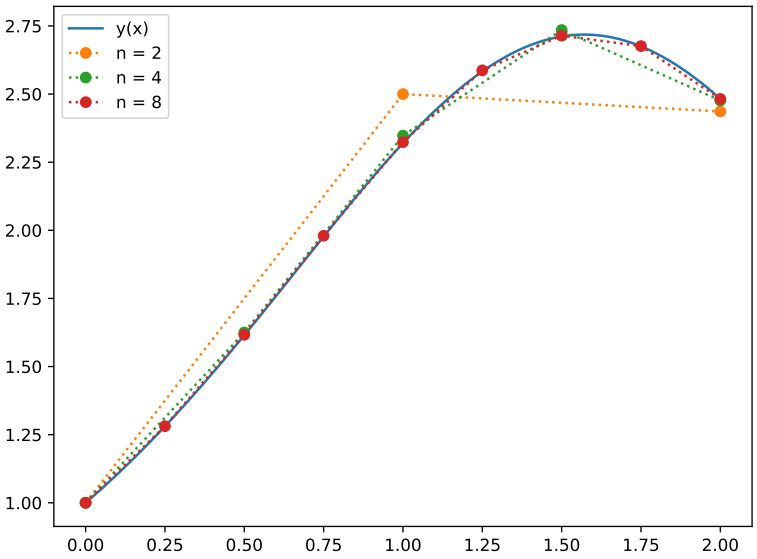
```
>>> [e**sin(i/2) for i in range(5)]

   [1.0, 1.6151462964420837, 2.319776824715853, 2.7114810176821584, 2.4825777280150003]
```

The following convincingly illustrates that the error is indeed $O(h^3)$.

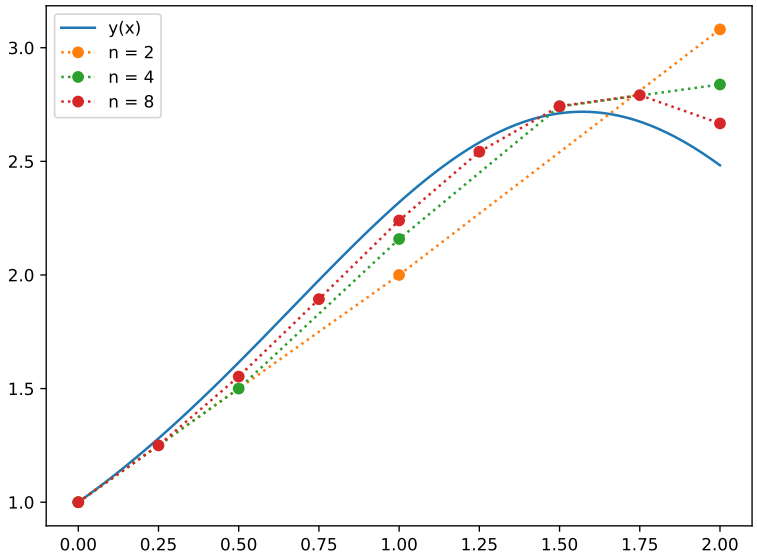
```
>>> [taylor_3_cosy(0, 1, 2, 10**n)[-1] - e**sin(2) for n in range(5)]
```

```
[2.5174222719849997, -0.0002461575553160955, -1.6375769584797695e-07, -
1.5647971807197791e-10, 2.0339285811132868e-13]
```

The following is a plot of the exact solution together with our approximations when using 2, 4 and 8 steps. Already for 4 steps, we obtain an approximation that, at least visually, is remarkably good.



On the other hand, for comparison, the following plot shows the corresponding approximations when using Euler's method instead.



Midpoint method (also known as the modified Euler method)

Recall that, in Euler's method (which is the same as the Taylor method of order 1) with step size h, we use $y_{n+1} = y_n + f(x_n, y_n)h$ because $f(x_n, y_n)$ is an approximation of $y'(x_n)$.

To get a better approximation $\tilde{y}_{n+1}$, we could do two small Euler steps (of size $h/2$ each) instead. We can then use the idea of Richardson extrapolation to combine Euler's y_{n+1} and the twice-Euler $\tilde{y}_{n+1}$ to get an approximation of higher order. This results in the following method:

(midpoint method) The following is an order 2 method for solving IVPs:

$$y_{n+1} = y_n + f\left(x_n + \frac{h}{2}, y_n + f(x_n, y_n)\frac{h}{2}\right)h$$

Advantage over Taylor method. When compared with the Taylor method of order 2, this has the advantage of not requiring us to determine partial derivatives of $f(x, y)$.

Why "midpoint"? In Euler's method, we use $y'(x_n) \approx f(x_n,\ y_n)$ as the slope for stepping from x_n to $x_{n+1} = x_n + h$. That is a bit unbalanced since we use the slope at the left endpoint for the entire interval (that's why Euler's method in the form we have seen is often called the **forward Euler method**). Note that the midpoint method is more balanced because it uses (an approximation of) the derivative at the midpoint $x_n + h/2$ instead.

How to derive the midpoint method. As mentioned above, we can derive the midpoint method by applying the idea of Richardson extrapolation to the Euler method.

- If we do one Euler step, then our approximation of $y(x_{n+1})$ is $y^{(1)}_{n+1} = y_n + f(x_n, y_n)h$.
  We know that the error in this approximation is roughly Ch^2 (in fact, $C \approx \frac{1}{2}y''(x_n)$ if h is small).
- If we do two Euler steps, then our approximation is $y^{(2)}_{n+1} = y_{\text{half}} + f(x_{\text{half}}, y_{\text{half}})\frac{h}{2}$ where $x_{\text{half}} = x_n + \frac{h}{2}$ and $y_{\text{half}} = y_n + f(x_n, y_n)\frac{h}{2}$ are the result of the first Euler step with step size $\frac{h}{2}$.
  Combined, we get $y^{(2)}_{n+1} = y_n + f(x_n, y_n)\frac{h}{2} + f\left(x_n + \frac{h}{2}, y_n + f(x_n, y_n)\frac{h}{2}\right)\frac{h}{2}$.
  The error should be roughly $2C\left(\frac{h}{2}\right)^2 = \frac{1}{2}Ch^2$ because we are doing 2 steps with step size $\frac{h}{2}$.
  [For small h, the constant C should still be as before, at least to first order.]
- As for Richardson extrapolation, $2y^{(2)}_{n+1} - y^{(1)}_{n+1}$ should therefore be an approximation of $y(x_{n+1})$ of order 3 (for the local truncation error).
  Indeed, $2y^{(2)}_{n+1} - y^{(1)}_{n+1} = y_n + f\left(x_n + \frac{h}{2}, y_n + f(x_n, y_n)\frac{h}{2}\right)h$ is precisely the midpoint method, which therefore should have global error of order 2.

An alternative way to show that the order is 2. We can also show that the midpoint method is of order 2 by computing and comparing Taylor expansions. The true solution has the expansion

$$\begin{aligned} y(x+h) &= y(x) + y'(x)h + \tfrac{1}{2}y''(x)h^2 + O(h^3) \\ &= y(x) + f(x, y(x))h + \tfrac{1}{2}\underbrace{\left[\tfrac{\mathrm{d}}{\mathrm{d}x}f(x, y(x))\right]}_{= f_x(x,y) + f_y(x,y)\cdot y'(x)}h^2 + O(h^3) \\ &= y + f(x, y)h + \tfrac{1}{2}(f_x(x, y) + f_y(x, y)f(x, y))h^2 + O(h^3). \end{aligned}$$

On the other hand, expanding the $f(...)$ term on the right-hand side of the midpoint method (with x and y in place of x_n and y_n) using the multivariate chain rule, we find

$$y + f\left(x + \frac{h}{2}, y + f(x, y)\frac{h}{2}\right)h = y + \left(f(x, y) + \left(f_x(x, y)\frac{1}{2} + f_y(x, y)\frac{f(x, y)}{2}\right)h + O(h^2)\right)h,$$

which agrees with the true expansion up to $O(h^3)$.

Example 138. Consider the IVP $y'=y$, $y(0)=1$. Approximate the solution $y(x)$ for $x\in[0,1]$ using the midpoint method with 4 steps. In particular, what is the approximation for $y(1)$?

Comment. Compare with Example 133. The real solution is $y(x)=e^x$ so that $y(1)=e\approx 2.71828$.

Solution. The step size is $h=\frac{1-0}{4}=\frac{1}{4}$. We apply the midpoint method with $f(x,y)=y$:

$$\begin{aligned}
x_0&=0 & y_0&=1\\
x_1&=\frac{1}{4} & y_1&=y_0+f\Big(x_0+\frac{h}{2},y_0+f(x_0,y_0)\frac{h}{2}\Big)h=\frac{41}{32}\approx 1.2813\\
x_2&=\frac{1}{2} & y_2&=y_1+f\Big(x_1+\frac{h}{2},y_1+f(x_1,y_1)\frac{h}{2}\Big)h=\frac{41^2}{32^2}\approx 1.6416\\
x_3&=\frac{3}{4} & y_3&=y_2+f\Big(x_2+\frac{h}{2},y_2+f(x_2,y_2)\frac{h}{2}\Big)h=\frac{41^3}{32^3}\approx 2.1033\\
x_4&=1 & y_4&=y_3+f\Big(x_3+\frac{h}{2},y_3+f(x_3,y_3)\frac{h}{2}\Big)h=\frac{41^4}{32^4}\approx 2.6949
\end{aligned}$$

In particular, the approximation for $y(1)$ is $y_4\approx 2.6949$, which is a noticeable improvement over Example 133, where we used the Euler method instead.

Comment. The above computations become particularly transparent if we realize that, for $f(x,y)=y$, each Euler step takes the simple form $y_{n+1}=y_n+f\Big(x_n+\frac{h}{2},y_n+f(x_n,y_n)\frac{h}{2}\Big)h=y_n\Big(1+h+\frac{h^2}{2}\Big)$.

It follows that $y_n=\Big(1+h+\frac{h^2}{2}\Big)^n$. Can you see how, as $h\to 0$, this allows us to recover the exact solution?

Example 139. Python Let us apply the midpoint method to $y'=y$, $y(0)=1$.

```
>>> def midpoint(f, x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            y0 = y0 + f(x0+h/2,y0+f(x0,y0)*h/2)*h
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints

>>> def f_y(x, y):
        return y
```

The exact solution is $y(x)=e^x$ with $y(1)=e\approx 2.718$.

```
>>> midpoint(f_y, 0, 1, 1, 4)

   [1, 1.28125, 1.6416015625, 2.103302001953125, 2.6948556900024414]
```

The following numerically confirms that the error in the midpoint method is $O(h^2)$.

```
>>> from math import e

>>> [midpoint(f_y, 0, 1, 1, 10**n)[-1] - e for n in range(6)]

   [-0.2182818284590451, -0.004200981850821073, -4.49658990882007e-05, -
   4.5270728232793545e-07, -4.530157138304958e-09, -4.530020802917534e-11]
```

The midpoint method can be written as:

$$\begin{aligned} y_{n+1} &= y_n + K_1 h \\ K_0 &= f(x_n, y_n) \\ K_1 &= f\left(x_n + \frac{h}{2}, y_n + K_0 \frac{h}{2}\right) \end{aligned}$$

Note that replacing the rule by $y_{n+1} = y_n + K_0 h$ results in Euler's method. Indeed, both K_0 and K_1 are approximations of the slope y' that we need for stepping from x_n to $x_{n+1} = x_n + h$.

Adding further such approximations K_i to the mix, one can eliminate further terms in the error expansion and obtain higher order methods known as **Runge–Kutta methods**.

The midpoint method is an example of a Runge–Kutta method of order 2 (but there are others as well).

https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods

Of particular practical importance is the following instance:

(Runge–Kutta method of order 4)

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}(K_0 + 2K_1 + 2K_2 + K_3)h \\ K_0 &= f(x_n, y_n) \\ K_1 &= f\left(x_n + \frac{h}{2}, y_n + K_0 \frac{h}{2}\right) \\ K_2 &= f\left(x_n + \frac{h}{2}, y_n + K_1 \frac{h}{2}\right) \\ K_3 &= f(x_n + h, y_n + K_2 h) \end{aligned}$$

Comment. Note how each of K_0, K_1, K_2, K_3 is an approximation of y' on the interval $[x_n, x_{n+1}]$ (with K_0 closer to $y'(x_n)$ and K_3 closer to $y'(x_{n+1})$). By taking the appropriate weighted average, we are able to get an approximation with a higher order.

Advanced comment. Note that the weights (with K_1 and K_2 combined because they both correspond to the midpoint $x_n + h/2$) are the same as in Simpson's rule for numerical integration. That is more than a coincidence. Indeed, if $f(x, y) = f(x)$ does not depend on y, then solving the DE is equivalent to integrating $f(x)$ and the Runge–Kutta method of order 4 turns into Simpson's rule.

Example 140. Python Let us implement the Runge–Kutta method of order 4.

```
>>> def runge_kutta4(f, x0, y0, xmax, n):
        h = (xmax - x0) / n
        ypoints = [y0]
        for i in range(n):
            K0 = f(x0,y0)
            K1 = f(x0+h/2, y0+K0*h/2)
            K2 = f(x0+h/2, y0+K1*h/2)
            K3 = f(x0+h, y0+K2*h)
            y0 = y0 + (K0 + 2*K1 + 2*K2 + K3)*h/6
            x0 = x0 + h
            ypoints.append(y0)
        return ypoints
```

First, for comparison with earlier methods, let us apply the method to the IVP $y'=y$, $y(0)=1$, which has the exact solution $y(x)=e^x$ with $y(1)=e\approx 2.718$.

```
>>> def f_y(x, y):
        return y

>>> runge_kutta4(f_y, 0, 1, 1, 4)

   [1, 1.2840169270833333, 1.648699469036526, 2.1169580259162033, 2.718209939201323]
```

The following convincingly illustrates that the error is indeed $O(h^4)$.

```
>>> from math import e

>>> [runge_kutta4(f_y, 0, 1, 1, 10**n)[-1] - e for n in range(6)]

   [-0.009948495125712054, -2.0843238792700447e-06, -2.2464119453502462e-10, -
   2.042810365310288e-14, 1.1546319456101628e-14, 6.217248937900877e-15]
```

Pause for a moment to really appreciate how much better these errors are in comparison with Euler's method! Whereas computing 10^5 values with Euler's method resulted in an error of $1.36\cdot 10^{-5}$, we are now able to obtain an error of $2.04\cdot 10^{-14}$ with only 10^3 values.

As a second example, let us consider as in Example 137 the IVP $y'=\cos(x)y$ with $y(0)=1$, which has the exact solution $y(x)=e^{\sin(x)}$ with $y(2)=e^{\sin(2)}\approx 2.48258$.

```
>>> def f_cosx_y(x, y):
        return cos(x)*y

>>> runge_kutta4(f_cosx_y, 0, 1, 2, 4)

   [1, 1.614859377441316, 2.3191895982789603, 2.7107641474177457, 2.481902218021582]
```

The following again convincingly illustrates that the error is indeed $O(h^4)$.

```
>>> from math import e

>>> [runge_kutta4(f_cosx_y, 0, 1, 2, 10**n)[-1] - e**sin(2) for n in range(5)]

   [-0.12999578105593113, -1.726387102785054e-05, -1.6494263732624859e-09, -
   1.6431300764452317e-13, 3.419486915845482e-13]
```

Important comment. Note that, in contrast to Example 137, we did not have to compute partial derivatives of $f(x,y)=\cos(x)y$ by hand. Instead, we were able to simply use $\cos(x)y$ in our `runge_kutta4` function.