MATH50003 Numerical Analysis Lecture Notes

Original Author: Dr. Sheehan Olver Compiled by: Isaac Lee | **Date: 02/03/2022** |  Github Source

Week 0: Asymptotics and Computational Cost

> **YouTube Lectures:**
> Asymptotics and Computational Cost

We introduce Big-O, little-o and asymptotic notation and see
how they can be used to describe computational cost.

1. Asymptotics as $n \to \infty$
2. Asymptotics as $x \to x_0$
3. Computational cost

1. Asymptotics as $n \to \infty$

Big-O, little-o, and "asymptotic to" are used to describe behaviour of functions
at infinity.

Definition (Big-O)

$$f(n) = O(\phi(n)) \qquad (\text{as } n \to \infty)$$

means

$$\left| \frac{f(n)}{\phi(n)} \right|$$

is bounded for sufficiently large n. That is,
there exist constants C and N_0 such
that, for all $n \geq N_0$, $|\frac{f(n)}{\phi(n)}| \leq C$.

Definition (little-O)

$$f(n) = o(\phi(n)) \qquad (\text{as } n \to \infty)$$

means

$$\lim_{n\to\infty} \frac{f(n)}{\phi(n)} = 0.$$

Definition (asymptotic to)

$$f(n) \sim \phi(n) \qquad (\text{as } n \to \infty)$$

means

$$\lim_{n\to\infty} \frac{f(n)}{\phi(n)} = 1.$$

Examples

$$\frac{\cos n}{n^2 - 1} = O(n^{-2})$$

as

$$\left| \frac{\frac{\cos n}{n^2-1}}{n^{-2}} \right| \leq \left| \frac{n^2}{n^2 - 1} \right| \leq 2$$

for $n \geq N_0 = 2$.

$$\log n = o(n)$$

as

$$lim_{n\to\infty} \frac{\log n}{n} = 0.$$

$$n^2 + 1 \sim n^2$$

as

$$\frac{n^2+1}{n^2} \to 1.$$

Note we sometimes write $f(O(\phi(n)))$ for a function of the form $f(g(n))$ such that $g(n) = O(\phi(n))$.

Rules

We have some simple algebraic rules:

Proposition (Big-O rules)

$$\begin{aligned} O(\phi(n))O(\psi(n)) &= O(\phi(n)\psi(n)) \qquad (\text{as } n \to \infty) \\ O(\phi(n)) + O(\psi(n)) &= O(|\phi(n)| + |\psi(n)|) \qquad (\text{as } n \to \infty). \end{aligned}$$

2. Asymptotics as $x \longrightarrow x_0$

We also have Big-O, little-o and "asymptotic to" at a point:

Definition (Big-O)

$$f(x) = O(\phi(x)) \qquad (\text{as } x \to x_0)$$

means

$$\frac{|f(x)}{\phi(x)|}$$

is bounded in a neighbourhood of x_0. That is, there exist constants C and r such that, for all $0 \leq |x - x_0| \leq r$, $|\frac{f(x)}{\phi(x)}| \leq C$.

Definition (little-O)

$$f(x) = o(\phi(x)) \qquad (\text{as } x \to x_0)$$

means

$$\lim_{x \to x_0} \frac{f(x)}{\phi(x)} = 0.$$

Definition (asymptotic to)

$$f(x) \sim \phi(x) \qquad (\text{as } x \to x_0)$$

means

$$\lim_{x \to x_0} \frac{f(x)}{\phi(x)} = 1.$$

Example

$$\exp x = 1 + x + O(x^2) \qquad \text{as } x \to 0$$

Since

$$\exp x = 1 + x + \frac{\exp t}{2} x^2$$

for some $t \in [0, x]$ and

$$\left| \frac{\frac{\exp t}{2} x^2}{x^2} \right| \leq \frac{3}{2}$$

provided $x \leq 1$.

3. Computational cost

We will use Big-O notation to describe the computational cost of algorithms.
Consider the following simple sum

$$\sum_{k=1}^{n} x_k^2$$

which we might implement as:

```
function sumsq(x)
    n = length(x)
    ret = 0.0
    for k = 1:n
        ret = ret + x[k]^2
    end
    ret
end

n = 100
x = randn(n)
sumsq(x)
```

Each step of this algorithm consists of one memory look-up (`z = x[k]`),
one multiplication (`w = z*z`) and one addition (`ret = ret + w`).
We will ignore the memory look-up in the following discussion.
The number of CPU operations per step is therefore 2 (the addition and multiplication).
Thus the total number of CPU operations is $2n$. But the constant 2 here is
misleading: we didn't count the memory look-up, thus it is more sensible to
just talk about the asymptotic complexity, that is, the *computational cost* is $O(n)$.

Now consider a double sum like:

$$\sum_{k=1}^{n} \sum_{j=1}^{k} x_j^2$$

which we might implement as:

```
function sumsq2(x)
    n = length(x)
    ret = 0.0
    for k = 1:n
        for j = 1:k
            ret = ret + x[j]^2
        end
    end
    ret
end

n = 100
x = randn(n)
sumsq2(x)
```

Now the inner loop is $O(1)$ operations (we don't try to count the precise number),
which we do k times for $O(k)$ operations as $k \to \infty$. The outer loop therefore takes

$$\sum_{k=1}^n O(k) = O\left(\sum_{k=1}^n k\right) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

operations.

Week 1: Numbers

> **YouTube Lectures:**
> Integers
> Floating Point Numbers
> Rounding
> Bouding Rounding Errors

Reference: Overton

In this chapter, we introduce the Two's-complement
storage for integers and the
IEEE Standard for Floating-Point Arithmetic.
There are many possible ways of representing real numbers on a computer, as well as
the precise behaviour of operations such as addition, multiplication, etc.
Before the 1980s each processor had potentially a different representation for
real numbers, as well as different behaviour for operations.
IEEE introduced in 1985 was a means to standardise this across
processors so that algorithms would produce consistent and reliable results.

This chapter may seem very low level for a mathematics course but there are
two important reasons to understand the behaviour of integers and floating-point numbers:

1. Integer arithmetic can suddenly start giving wrong negative answers when numbers
   become large.
2. Floating-point arithmetic is very precisely defined, and can even be used
   in rigorous computations as we shall see in the problem sheets. But it is not exact
   and its important to understand how errors in computations can accumulate.
3. Failure to understand floating-point arithmetic can cause catastrophic issues
   in practice, with the extreme example being the
   explosion of the Ariane 5 rocket.

In this chapter we discuss the following:

1. Binary representation: Any real number can be represented in binary, that is,
   by an infinite sequence of 0s and 1s (bits). We review binary representation.
2. Integers: There are multiple ways of representing integers on a computer. We discuss the
   the different types of integers and their representation as bits, and how arithmetic operations behave
   like modular arithmetic. As an advanced topic we discuss `BigInt`, which uses variable bit length storage.
3. Floating-point numbers: Real numbers are stored on a computer with a finite number of bits.
   There are three types of floating-point numbers: *normal numbers*, *subnormal numbers*, and *special numbers*.
4. Arithmetic: Arithmetic operations in floating-point are exact up to rounding, and how the
   rounding mode can be set. This allows us to bound errors computations.
5. High-precision floating-point numbers: As an advanced topic, we discuss how the precision of floating-point arithmetic
   can be increased arbitrary
   using `BigFloat`.

Before we begin, we load two external packages. SetRounding.jl allows us
to set the rounding mode of floating-point arithmetic. ColorBitstring.jl
implements functions `printbits` (and `printlnbits`)
which print the bits (and with a newline) of floating-point numbers in colour.

```
using SetRounding, ColorBitstring
```

1. Binary representation

Any integer can be presented in binary format, that is, a sequence of `0`s and `1`s.

Definition
For $B_0, \ldots, B_p \in \{0,1\}$ denote a non-negative integer in *binary format* by:

$$(B_p \ldots B_1 B_0)_2 := 2^p B_p + \cdots + 2B_1 + B_0$$

For $b_1, b_2, \ldots \in \{0,1\}$, Denote a non-negative real number in *binary format* by:

$$(B_p \ldots B_0 . b_1 b_2 b_3 \ldots)_2 = (B_p \ldots B_0)_2 + \frac{b_1}{2} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \cdots$$

First we show some examples of verifying a numbers binary representation:

Example (integer in binary)
A simple integer example is $5 = 2^2 + 2^0 = (101)_2$.

Example (rational in binary)
Consider the number `1/3`. In decimal recall that:

$$1/3 = 0.3333\ldots = \sum_{k=1}^{\infty} \frac{3}{10^k}$$

We will see that in binary

$$1/3 = (0.010101\ldots)_2 = \sum_{k=1}^{\infty} \frac{1}{2^{2k}}$$

Both results can be proven using the geometric series:

$$\sum_{k=0}^{\infty} z^k = \frac{1}{1-z}$$

provided $|z| < 1$. That is, with $z = \frac{1}{4}$ we verify the binary expansion:

$$\sum_{k=1}^{\infty} \frac{1}{4^k} = \frac{1}{1-1/4} - 1 = \frac{1}{3}$$

A similar argument with $z = 1/10$ shows the decimal case.

2. Integers

On a computer one typically represents integers by a finite number of p bits,
with 2^p possible combinations of 0s and 1s. For *unsigned integers* (non-negative integers)
these bits are just the first p binary digits: $(B_{p-1} \ldots B_1 B_0)_2$.

Integers on a computer follow modular arithmetic:

Definition (ring of integers modulo m) Denote the ring

$$\mathbb{Z}_m := \{0 \pmod m, 1 \pmod m, \dots, m-1 \pmod m\}$$

Integers represented with p-bits on a computer actually represent elements of $\mathbb{Z}_{2^p}$ and integer arithmetic on a computer is equivalent to arithmetic modulo 2^p.

Example (addition of 8-bit unsigned integers) Consider the addition of two 8-bit numbers:

$$255 + 1 = (11111111)_2 + (00000001)_2 = (100000000)_2 = 256$$

The result is impossible to store in just 8-bits! It is way too slow for a computer to increase the number of bits, or to throw an error (checks are slow). So instead it treats the integers as elements of $\mathbb{Z}_{256}$:

$$255 + 1 \pmod{256} = (00000000)_2 \pmod{256} = 0 \pmod{256}$$

We can see this in Julia:

```
x = UInt8(255)
y = UInt8(1)
printbits(x); println(" + "); printbits(y); println(" = ")
printbits(x + y)
```

Example (multiplication of 8-bit unsigned integers) Multiplication works similarly: for example,

$$254 * 2 \pmod{256} = 252 \pmod{256} = (11111100)_2 \pmod{256}$$

We can see this behaviour in code by printing the bits:

```
x = UInt8(254) # 254 represented in 8-bits as an unsigned integer
y = UInt8(2)   #   2 represented in 8-bits as an unsigned integer
printbits(x); println(" * "); printbits(y); println(" = ")
printbits(x * y)
```

Signed integer

Signed integers use the Two's complemement convention. The convention is if the first bit is 1 then the number is negative: the number $2^p - y$ is interpreted as $-y$.
Thus for $p = 8$ we are interpreting 2^7 through $2^8 - 1$ as negative numbers.

Example (converting bits to signed integers) What 8-bit integer has the bits `01001001`? Adding the corresponding decimal places we get:

```
2^0 + 2^3 + 2^6
```

What 8-bit (signed) integer has the bits `11001001`? Because the first bit is `1` we know it's a negative number, hence we need to sum the bits but then subtract `2^p`:

```
2^0 + 2^3 + 2^6 + 2^7 - 2^8
```

We can check the results using `printbits`:

```
printlnbits(Int8(73))
printbits(-Int8(55))
```

Arithmetic works precisely
the same for signed and unsigned integers.

Example (addition of 8-bit integers)
Consider `(-1) + 1` in 8-bit arithmetic. The number -1 has the same bits as $2^8 - 1 = 255$. Thus this is equivalent to the previous question and we get the correct result of `0`. In other words:

$$-1 + 1 \pmod{2^p} = 2^p - 1 + 1 \pmod{2^p} = 2^p \pmod{2^p} = 0 \pmod{2^p}$$

Example (multiplication of 8-bit integers)
Consider `(-2) * 2`. -2 has the same bits as $2^{256} - 2 = 254$ and -4 has the same bits as $2^{256} - 4 = 252$, and hence from the previous example we get the correct result of `-4`. In other words:

$$(-2) * 2 \pmod{2^p} = (2^p - 2) * 2 \pmod{2^p} = 2^{p+1} - 4 \pmod{2^p} = -4 \pmod{2^p}$$

Example (overflow) We can find the largest and smallest instances of a type using `typemax` and `typemin`:

```
printlnbits(typemax(Int8)) # 2^7-1 = 127
printbits(typemin(Int8)) # -2^7 = -128
```

As explained, due to modular arithmetic, when we add `1` to the largest 8-bit integer we get the smallest:

```
typemax(Int8) + Int8(1) # returns typemin(Int8)
```

This behaviour is often not desired and is known as *overflow*, and one must be wary of using integers close to their largest value.

Variable bit representation (advanced)

An alternative representation for integers uses a variable number of bits,
with the advantage of avoiding overflow but with the disadvantage of a substantial
speed penalty. In Julia these are `BigInt`s, which we can create by calling `big` on an
integer:

```
x = typemax(Int64) + big(1) # Too big to be an `Int64`
```

Note in this case addition automatically promotes an `Int64` to a `BigInt`.
We can create very large numbers using `BigInt`:

```
x^100
```

Note the number of bits is not fixed, the larger the number, the more bits required
to represent it, so while overflow is impossible, it is possible to run out of memory if a number is
astronomically large: go ahead and try `x^x` (at your own risk).

Division

In addition to `+`, `-`, and `*` we have integer division `÷`, which rounds down:

```
5 ÷ 2 # equivalent to div(5,2)
```

Standard division `/` (or `\` for division on the right) creates a floating-point number,
which will be discussed shortly:

```
5 / 2 # alternatively 2 \ 5
```

We can also create rational numbers using `//`:

```
(1//2) + (3//4)
```

Rational arithmetic often leads to overflow so it
is often best to combine `big` with rationals:

```
big(102324)//132413023 + 23434545//4243061 + 23434545//42430534435
```

3. Floating-point numbers

Floating-point numbers are a subset of real numbers that are representable using
a fixed number of bits.

> **Definition (floating-point numbers)**
> Given integers σ (the "exponential shift") Q (the number of exponent bits) and
> S (the precision), define the set of
> *Floating-point numbers* by dividing into *normal*, *sub-normal*, and *special number* subsets:

$$F_{\sigma,Q,S} := F^{\text{normal}}_{\sigma,Q,S} \cup F^{\text{sub}}_{\sigma,Q,S} \cup F^{\text{special}}.$$

The *normal numbers* $F^{\text{normal}}_{\sigma,Q,S} \subset \mathbb{R}$ are defined by

$$F^{\text{normal}}_{\sigma,Q,S} = \{\pm 2^{q-\sigma} \times (1.b_1 b_2 b_3 \dots b_S)_2 : 1 \le q < 2^Q - 1\}.$$

The *sub-normal numbers* $F^{\text{sub}}_{\sigma,Q,S} \subset \mathbb{R}$ are defined as

$$F^{\text{sub}}_{\sigma,Q,S} = \{\pm 2^{1-\sigma} \times (0.b_1 b_2 b_3 \dots b_S)_2\}.$$

The *special numbers* $F^{\text{special}} \not\subset \mathbb{R}$ are defined later.

Note this set of real numbers has no nice algebraic structure: it is not closed under addition, subtraction, etc. We will therefore need to define approximate versions of algebraic operations later.

Floating-point numbers are stored in $1 + Q + S$ total number of bits, in the format

$$s q_{Q-1} \dots q_0 b_1 \dots b_S$$

The first bit (s) is the sign bit: 0 means positive and 1 means negative. The bits $q_{Q-1} \dots q_0$ are the exponent bits: they are the binary digits of the unsigned integer q:

$$q = (q_{Q-1} \dots q_0)_2.$$

Finally, the bits $b_1 \dots b_S$ are the significand bits.
If $1 \le q < 2^Q - 1$ then the bits represent the normal number

$$x = \pm 2^{q-\sigma} \times (1.b_1 b_2 b_3 \dots b_S)_2.$$

If $q = 0$ (i.e. all bits are 0) then the bits represent the sub-normal number

$$x = \pm 2^{1-\sigma} \times (0.b_1 b_2 b_3 \dots b_S)_2.$$

If $q = 2^Q - 1$ (i.e. all bits are 1) then the bits represent a special number, discussed later.

IEEE floating-point numbers

Definition (IEEE floating-point numbers)
IEEE has 3 standard floating-point formats: 16-bit (half precision), 32-bit (single precision) and 64-bit (double precision) defined by:

$$F_{16} := F_{15,5,10}$$
$$F_{32} := F_{127,8,23}$$
$$F_{64} := F_{1023,11,52}$$

In Julia these correspond to 3 different floating-point types:

1. `Float64` is a type representing double precision (F_{64}).
   We can create a `Float64` by including a
   decimal point when writing the number:
   `1.0` is a `Float64`. `Float64` is the default format for
   scientific computing (on the *Floating-Point Unit*, FPU).
2. `Float32` is a type representing single precision (F_{32}). We can create a `Float32` by including a `f0` when writing the number:
   `1f0` is a `Float32`. `Float32` is generally the default format for graphics (on the *Graphics Processing Unit*, GPU), as the difference between 32 bits and 64 bits is indistinguishable to the eye in visualisation, and more data can be fit into a GPU's limitted memory.
3. `Float16` is a type representing half-precision (F_{16}).
   It is important in machine learning where one wants to maximise the amount of data and high accuracy is not necessarily helpful.

Example (rational in `Float32`) How is the number $1/3$ stored in `Float32`?
Recall that

$$1/3 = (0.010101\ldots)_2 = 2^{-2}(1.0101\ldots)_2 = 2^{125-127}(1.0101\ldots)_2$$

and since $125 = (1111101)_2$ the exponent bits are `01111101`.
.
For the significand we round the last bit to the nearest element of F_{32}, (this is explained in detail in the section on rounding), so we have

$$1.01010101010101010101010\ldots \approx 1.01010101010101010101011 \in F_{32}$$

and the significand bits are `01010101010101010101011`.
Thus the `Float32` bits for $1/3$ are:

```
printbits(1f0/3)
```

For sub-normal numbers, the simplest example is zero, which has $q = 0$ and all significand bits zero:

```
printbits(0.0)
```

Unlike integers, we also have a negative zero:

```
printbits(-0.0)
```

This is treated as identical to `0.0` (except for degenerate operations as explained in special numbers).

Special normal numbers

When dealing with normal numbers there are some important constants that we will use to bound errors.

Definition (machine epsilon/smallest positive normal number/largest normal number) *Machine epsilon* is denoted

$$\epsilon_{\mathrm{m},S} := 2^{-S}.$$

When S is implied by context we use the notation ϵ_{m}.
The *smallest positive normal number* is $q = 1$ and b_k all zero:

$$\min |F^{\mathrm{normal}}_{\sigma,Q,S}| = 2^{1-\sigma}$$

where $|A| := \{|x| : x \in A\}$.
The *largest (positive) normal number* is

$$\max F^{\mathrm{normal}}_{\sigma,Q,S} = 2^{2^Q-2-\sigma}(1.11\ldots1)_2 = 2^{2^Q-2-\sigma}(2-\epsilon_{\mathrm{m}})$$

We confirm the simple bit representations:

```
σ,Q,S = 127,23,8 # Float32
εₘ = 2.0^(-S)
printlnbits(Float32(2.0^(1-σ))) # smallest positive Float32
printlnbits(Float32(2.0^(2^Q-2-σ) * (2-εₘ))) # largest Float32
```

For a given floating-point type, we can find these constants using the following functions:

```
eps(Float32),floatmin(Float32),floatmax(Float32)
```

Example (creating a sub-normal number) If we divide the smallest normal number by two, we get a subnormal number:

```
mn = floatmin(Float32) # smallest normal Float32
printlnbits(mn)
printbits(mn/2)
```

Can you explain the bits?

Special numbers

The special numbers extend the real line by adding $\pm\infty$ but also a notion of "not-a-number".

> **Definition (not a number)**
> Let NaN represent "not a number" and define
>
> $$F^{\text{special}} := \{\infty, -\infty, \mathrm{NaN}\}$$

Whenever the bits of q of a floating-point number are all 1 then they represent an element of F^{special}.
If all $b_k = 0$, then the number represents either $\pm\infty$, called `Inf` and `-Inf` for 64-bit floating-point numbers (or `Inf16`, `Inf32`
for 16-bit and 32-bit, respectively):

```
printlnbits(Inf16)
printbits(-Inf16)
```

All other special floating-point numbers represent NaN. One particular representation of NaN
is denoted by `NaN` for 64-bit floating-point numbers (or `NaN16`, `NaN32` for 16-bit and 32-bit, respectively):

```
printbits(NaN16)
```

These are needed for undefined algebraic operations such as:

```
0/0
```

Example (many `NaN`s) What happens if we change some other b_k to be nonzero?
We can create bits as a string and see:

```
i = parse(UInt16, "0111110000010001"; base=2)
reinterpret(Float16, i)
```

Thus, there are more than one `NaN`s on a computer.

4. Arithmetic

Arithmetic operations on floating-point numbers are *exact up to rounding*.
There are three basic rounding strategies: round up/down/nearest.
Mathematically we introduce a function to capture the notion of rounding:

Definition (rounding) $\mathrm{fl}^{\mathrm{up}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes
the function that rounds a real number up to the nearest floating-point number that is greater or equal.
$\mathrm{fl}^{\mathrm{down}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes
the function that rounds a real number down to the nearest floating-point number that is greater or equal.
$\mathrm{fl}^{\mathrm{nearest}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes
the function that rounds a real number to the nearest floating-point number. In case of a tie,
it returns the floating-point number whose least significant bit is equal to zero.
We use the notation fl when σ, Q, S and the rounding mode are implied by context,
with $\mathrm{fl}^{\mathrm{nearest}}$ being the default rounding mode.

In Julia, the rounding mode is specified by tags `RoundUp`, `RoundDown`, and `RoundNearest`. (There are also more exotic rounding strategies `RoundToZero`, `RoundNearestTiesAway` and `RoundNearestTiesUp` that we won't use.)

WARNING (rounding performance, advanced) These rounding modes are part
of the FPU instruction set so will be (roughly) equally fast as the default, `RoundNearest`.
Unfortunately, changing the rounding mode is expensive, and is not thread-safe.

Let's try rounding a `Float64` to a `Float32`.

```
printlnbits(1/3)  # 64 bits
printbits(Float32(1/3))  # round to nearest 32-bit
```

The default rounding mode can be changed:

```
printbits(Float32(1/3,RoundDown) )
```

Or alternatively we can change the rounding mode for a chunk of code
using `setrounding`. The following computes upper and lower bounds for `/`:

```
x = 1f0
setrounding(Float32, RoundDown) do
    x/3
end,
setrounding(Float32, RoundUp) do
    x/3
end
```

WARNING (compiled constants, advanced): Why did we first create a variable `x` instead of typing `1f0/3`?
This is due to a very subtle issue where the compiler is *too clever for it's own good*:
it recognises `1f0/3` can be computed at compile time, but failed to recognise the rounding mode
was changed.

In IEEE arithmetic, the arithmetic operations `+`, `-`, `*`, `/` are defined by the property
that they are exact up to rounding. Mathematically we denote these operations as follows:

$$
\begin{aligned}
x \oplus y &:= \mathrm{fl}(x + y) \\
x \ominus y &:= \mathrm{fl}(x - y) \\
x \otimes y &:= \mathrm{fl}(x * y) \\
x \oslash y &:= \mathrm{fl}(x / y)
\end{aligned}
$$

Note also that `^` and `sqrt` are similarly exact up to rounding.

Example (decimal is not exact) `1.1+0.1` gives a different result than `1.2`:

```
x = 1.1
y = 0.1
x + y - 1.2 # Not Zero?!?
```

This is because $\mathrm{fl}(1.1) \neq 1 + 1/10$, but rather:

$$
\mathrm{fl}(1.1) = 1 + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \cdots + 2^{-48} + 2^{-49} + 2^{-51}
$$

WARNING (non-associative) These operations are not associative! E.g. $(x \oplus y) \oplus z$ is not necessarily equal to $x \oplus (y \oplus z)$.
Commutativity is preserved, at least.
Here is a surprising example of non-associativity:

```
(1.1 + 1.2) + 1.3, 1.1 + (1.2 + 1.3)
```

Can you explain this in terms of bits?

Bounding errors in floating point arithmetic

Before we dicuss bounds on errors, we need to talk about the two notions of errors:

> **Definition (absolute/relative error)** If $\tilde{x} = x + \delta_{rma} = x(1 + \delta_{\mathrm{r}})$ then
> $|\delta_{\mathrm{a}}|$ is called the *absolute error* and $|\delta_{\mathrm{r}}|$ is called the *relative error* in approximating x by $\tilde{x}$.

We can bound the error of basic arithmetic operations in terms of machine epsilon, provided
a real number is close to a normal number:

> **Definition (normalised range)** The *normalised range* $\mathcal{N}_{\sigma,Q,S} \subset \mathbb{R}$
> is the subset of real numbers that lies
> between the smallest and largest normal floating-point number:
>
> $$
> \mathcal{N}_{\sigma,Q,S} := \{x : \min |F_{\sigma,Q,S}| \leq |x| \leq \max F_{\sigma,Q,S}\}
> $$
>
> When σ, Q, S are implied by context we use the notation $\mathcal{N}$.

We can use machine epsilon to determine bounds on rounding:

Proposition (rounding arithmetic)
If $x \in \mathcal{N}$ then

$$\mathrm{fl}^{\mathrm{mode}}(x) = x(1+\delta_x^{\mathrm{mode}})$$

where the *relative error* is

$$|\delta_x^{\mathrm{nearest}}| \leq \frac{\epsilon_{\mathrm{m}}}{2}$$
$$|\delta_x^{\mathrm{up/down}}| < \epsilon_{\mathrm{m}}.$$

This immediately implies relative error bounds on all IEEE arithmetic operations, e.g.,
if $x + y \in \mathcal{N}$ then
we have

$$x \oplus y = (x+y)(1+\delta_1)$$

where (assuming the default nearest rounding)
$|\delta_1| \leq \frac{\epsilon_{\mathrm{m}}}{2}$.

Example (bounding a simple computation) We show how to bound the error in computing

$$(1.1 + 1.2) + 1.3$$

using floating-point arithmetic. First note that `1.1` on a computer is in
fact $\mathrm{fl}(1.1)$. Thus this computation becomes

$$(\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.2)) \oplus \mathrm{fl}(1.3)$$

First we find

$$(\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.2)) = (1.1(1+\delta_1) + 1.2(1+\delta_2))(1+\delta_3) = 2.3 + 1.1\delta_1 + 1.2\delta_2 + 2.3\delta_3 + 1.1\delta_1\delta_3 + 1.2\delta_2\delta_3 = 2.3 + \delta_4$$

where (note $\delta_1\delta_3$ and $\delta_2\delta_3$ are tiny so we just round up our bound to the nearest decimal)

$$|\delta_4| \leq 2.3\epsilon_{\mathrm{m}}$$

Thus the computation becomes

$$((2.3 + \delta_4) + 1.3(1+\delta_5))(1+\delta_6) = 3.6 + \delta_4 + 1.3\delta_5 + 3.6\delta_6 + \delta_4\delta_6 + 1.3\delta_5\delta_6 = 3.6 + \delta_7$$

where the *absolute error* is

$$|\delta_7| \leq 4.8\epsilon_{\mathrm{m}}$$

Indeed, this bound is bigger than the observed error:

```
abs(3.6 - (1.1+1.2+1.3)), 4.8eps()
```

Arithmetic and special numbers

Arithmetic works differently on `Inf` and `NaN` and for undefined operations.
In particular we have:

```
1/0.0        #  Inf
1/(-0.0)     # -Inf
0.0/0.0      #  NaN

Inf*0        #  NaN
Inf+5        #  Inf
(-1)*Inf     # -Inf
1/Inf        #  0.0
1/(-Inf)     # -0.0
Inf - Inf    #  NaN
Inf ==  Inf  #  true
Inf == -Inf  #  false

NaN*0        #  NaN
NaN+5        #  NaN
1/NaN        #  NaN
NaN == NaN   #  false
NaN != NaN   #  true
```

Special functions (advanced)

Other special functions like `cos`, `sin`, `exp`, etc. are *not* part of the IEEE standard.
Instead, they are implemented by composing the basic arithmetic operations, which accumulate errors. Fortunately many are designed to have *relative accuracy*, that is, `s = sin(x)` (that is, the Julia implementation of $\sin x$) satisfies

$$\mathtt{s} = (\sin x)(1+\delta)$$

where $|\delta| < c\epsilon_{\mathrm{m}}$ for a reasonably small $c > 0$,
provided that $x \in \mathrm{F}^{\mathrm{normal}}$.
Note these special functions are written in (advanced) Julia code, for example, sin.

WARNING (sin(fl(x)) is not always close to sin(x)) This is possibly a misleading statement when one thinks of x as a real number. Consider $x = \pi$ so that $\sin x = 0$.
However, as $\mathrm{fl}(\pi) \neq \pi$. Thus we only have relative accuracy compared to the floating point approximation:

```
π₆₄ = Float64(π)
πᵦ = big(π₆₄) # Convert 64-bit approximation of π to higher precision. Note its the same number.
abs(sin(π₆₄)), abs(sin(π₆₄) - sin(πᵦ)) # only has relative accuracy compared to sin(πᵦ), not sin(π)
```

Another issue is when x is very large:

```
ε = eps() # machine epsilon, 2^(-52)
x = 2*10.0^100
abs(sin(x) - sin(big(x)))  ≤  abs(sin(big(x))) * ε
```

But if we instead compute `10^100` using `BigFloat` we get a completely different answer that even has the wrong sign!

```
x̃ = 2*big(10.0)^100
sin(x), sin(x̃)
```

This is because we commit an error on the order of roughly

$$2 * 10^{100} * \epsilon_{\mathrm{m}} \approx 4.44 * 10^{84}$$

when we round $2 * 10^{100}$ to the nearest float.

Example (polynomial near root)
For general functions we do not generally have relative accuracy.
For example, consider a simple
polynomial $1 + 4x + x^2$ which has a root at $\sqrt{3} - 2$. But

```
f = x -> 1 + 4x + x^2
x = sqrt(3) - 2
abserr = abs(f(big(x)) - f(x))
relerr = abserr/abs(f(x))
abserr, relerr # very large relative error
```

We can see this in the error bound (note that $4x$ is exact for floating point numbers and adding 1 is exact for this particular x):

$$(x \otimes x \oplus 4x) + 1 = (x^2(1+\delta_1) + 4x)(1+\delta_2) + 1 = x^2 + 4x + 1 + \delta_1 x^2 + 4x\delta_2 + x^2\delta_1\delta_2$$

Using a simple bound $|x| < 1$ we get a (pessimistic) bound on the absolute error of $3\epsilon_{\mathrm{m}}$. Here `f(x)` itself is less than $2\epsilon_{\mathrm{m}}$ so this does not imply relative accuracy. (Of course, a bad upper bound is not the same as a proof of inaccuracy, but here we observe the inaccuracy in practice.)

5. High-precision floating-point numbers (advanced)

It is possible to set the precision of a floating-point number using the `BigFloat` type, which results from the usage of `big` when the result is not an integer.
For example, here is an approximation of 1/3 accurate to 77 decimal digits:

```
big(1)/3
```

Note we can set the rounding mode as in `Float64`, e.g., this gives (rigorous) bounds on `1/3`:

```
setrounding(BigFloat, RoundDown) do
  big(1)/3
end, setrounding(BigFloat, RoundUp) do
  big(1)/3
end
```

We can also increase the precision, e.g., this finds bounds on `1/3` accurate to more than 1000 decimal places:

```
setprecision(4_000) do # 4000 bit precision
  setrounding(BigFloat, RoundDown) do
    big(1)/3
  end, setrounding(BigFloat, RoundUp) do
    big(1)/3
  end
end
```

In the problem sheet we shall see how this can be used to rigorously bound e, accurate to 1000 digits.

Week 2: Differentiation

> **YouTube Lectures:**
> Finite Differences
> Dual Numbers

We now get to our first computational problem: given a function, how can we approximate its derivative at a point? Before we begin, we must be clear what a "function" is. Consider three possible scenarios:

1. *Black-box function*: Consider a floating-point valued function $f^{\mathrm{FP}} : D \to F$ where $D \subset F \equiv F_{\sigma,Q,S}$ (e.g., we are given a double precision function that takes in a `Float64` and returns another `Float64`) which we only know *pointwise*. This is the situation if we have a function that relies on a compiled C library, which composes floating point arithmetic operations.

Since F is a discrete set such an f^{FP} cannot be differentiable in a rigorous way,
therefore we need to assume that f^{FP} approximates a differentiable function f with controlled
error in order to state anything precise.

2. *Generic function*: Consider a function that is a formula (or, equivalentally, a *piece of code*)
that we can evaluate it on arbitrary types, including
custom types that we create. An example is a polynomial:
$p(x) = p_0 + p_1 x + \cdots + p_n x^n$
which can be evaluated for x in the reals, complexes, or any other ring.
More generally, if we have a function defined in Julia that does not call any
C libraries it can be evaluated on different types.
For analysis we typically consider both a differentiable function $f : D \to \mathbb{R}$ for
$D \subset \mathbb{R}$,
which would be what one would have if we could evaluate a function exactly using real arithmetic, and
$f^{\mathrm{FP}} : D \cap F \to F$, which is what we actually compute when evaluating the function using
floating point arithmetic.
3. *Graph function*: The function is built by composing different basic "kernels" with known differentiability properties.
We won't consider this situation in this module, though it is the model used by Python machine learning toolbox's
like PyTorch and TensorFlow.

We discuss the following techniques:

1. Finite-differences: Use the definition of a derivative that one learns in calculus to approximate its value.
Unfortunately, the round-off errors of floating point arithmetic typically limit its accuracy.
2. Dual numbers (forward-mode automatic differentiation): Define a special type that when applied to a function
computes its derivative. Mathematically, this uses *dual numbers*, which are analoguous to complex numbers.

Note there are other techniques for differentiation that we don't discuss:

1. Symbolic differentiation: A tree is built representing a formula which is differentiated using
the product and chain rule.
2. Adjoints and back-propagation (reverse-mode automatic differentiation): This is similar to
symbolic differentiation but automated, to build up
a tape of operations that tracks interdependencies.
It's outside the scope of this module but is computationally preferred for computing gradients
of large dimensional functions which is critical in machine learning.
3. Interpolation and differentiation: We can also differentiate functions *globally*, that is, in an interval instead of
only a single point, which will be discussed later in the module.

```
using ColorBitstring
```

1. Finite-differences

The definition

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

tells us that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

provided that h is sufficiently small.

It's important to note that approximation uses only the *black-box* notion of a function but to obtain bounds we need more.

If we know a bound on $f''(x)$ then Taylor's theorem tells us a precise bound:

Proposition
The error in approximating the derivative using finite differences is

$$\left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \leq \frac{M}{2} h$$

where $M = \sup_{x \leq t \leq x+h} |f''(t)|$.

Proof
Follows immediately from Taylor's theorem:

$$f(x+h) = f(x) + f'(x)h + \frac{f''(t)}{2} h^2$$

for some $x \leq t \leq x + h$.

■

There are also alternative versions of finite differences. Leftside finite-differences:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

and central differences:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Composing these approximations is useful for higher-order derivatives as we discuss in the problem sheet.

Note this is assuming *real arithmetic*, the answer is drastically different with *floating point arithmetic*.

Does finite-differences work with floating point arithmetic?

Let's try differentiating two simple polynomials $f(x) = 1 + x + x^2$ and $g(x) = 1 + x/3 + x^2$ by applying the finite-difference approximation to their floating point implementations f^{FP} and g^{FP}:

```
f = x -> 1 + x + x^2     # we treat f and g as black-boxs
g = x -> 1 + x/3 + x^2
h = 0.000001
(f(h)-f(0))/h, (g(h)-g(0))/h
```

Both seem to roughly approximate the true derivatives (1 and $1/3$).
We can do a plot to see how fast the error goes down as we let h become small.

```
using Plots
h = 2.0 .^ (0:-1:-60)  # [1,1/2,1/4,…]
nanabs = x -> iszero(x) ? NaN : abs(x) # avoid 0's in log scale plot
plot(nanabs.((f.(h) .- f(0)) ./ h .- 1); yscale=:log10, title="convergence of derivatives, h =
2^(-n)", label="f", legend=:bottomleft)
plot!(abs.((g.(h) .- g(0)) ./ h .- 1/3); yscale=:log10, label = "g")
```

In the case of f it is a success: we approximate the true derivative *exactly* provided we take $h = 2^{-n}$
for $26 < n \leq 52$.
But for g it is a huge failure: the approximation starts to converge, but then diverges exponentially fast, before levelling off!

It is clear that f is extremely special. Most functions will behave like g, and had we not taken
h to be a power of two we also see divergence for differentiating f:

```
h = 10.0 .^ (0:-1:-16)  # [1,1/10,1/100,…]
plot(abs.((f.(h) .- f(0)) ./ h .- 1); yscale=:log10, title="convergence of derivatives, h = 10^(-
n)", label="f", legend=:bottomleft)
plot!(abs.((g.(h) .- g(0)) ./ h .- 1/3); yscale=:log10, label = "g")
```

For these two simple
examples, we can understand why we see very different behaviour.

Example (convergence(?) of finite difference) Consider differentiating $f(x) = 1 + x + x^2$ at 0 with $h = 2^{-n}$.
We consider 3 different cases with different behaviour, where S is the number of significand bits:

1. $0 \leq n \leq S/2$
2. $S/2 < n \leq S$
3. $S \leq n$

Note that $f^{\mathrm{FP}}(0) = f(0) = 1$. Thus we wish to understand the error in approximating $f'(0) = 1$ by

$$(f^{\mathrm{FP}}(h) \ominus 1) \oslash h \qquad \text{where} \qquad f^{\mathrm{FP}}(x) = 1 \oplus x \oplus x \otimes x.$$

Case 1 ($0 \leq n \leq S/2$): note that $f^{\mathrm{FP}}(h) = f(h) = 1 + 2^{-n} + 2^{-2n}$
as each computation is precisely a floating point number (hence no rounding). We can see this in half-precision,
with $n = 3$ we have a 1 in the 3rd and 6th decimal place:

```
S = 10 # 10 significant bits
n = 3  # 3 ≤ S/2 = 5
h = Float16(2)^(-n)
printbits(f(h))
```

Subtracting 1 and dividing by h will also be exact, hence we get

$$(f^{\mathrm{FP}}(h) \ominus 1) \oslash h = 1 + 2^{-n}$$

which shows exponential convergence.

Case 2 ($S/2 < n \leq S$): Now we have (using round-to-nearest)

$$f^{\mathrm{FP}}(h) = (1 + 2^{-n}) \oplus 2^{-2n} = 1 + 2^{-n}$$

Then

$$(f^{\mathrm{FP}}(h) \ominus 1) \oslash h = 1 = f'(0)$$

We have actually performed better than true real arithmetic and converged without a limit!

Case 3 ($n > S$): If we take n too large, then $1 \oplus h = 1$ and we have $f^{\mathrm{FP}}(h) = 1$, that is and

$$(f^{\mathrm{FP}}(h) \ominus 1) \oslash h = 0 \neq f'(0)$$

Example (divergence of finite difference) Consider differentiating $g(x) = 1 + x/3 + x^2$ at 0 with $h = 2^{-n}$ and assume n is even for simplicity and consider half-precision with $S = 10$.
Note that $g^{\mathrm{FP}}(0) = g(0) = 1$.
Recall

$$h \oslash 3 = 2^{-n-2} * (1.0101010101)_2$$

Note we lose two bits each time in the computation of $1 \oplus (h \oslash 3)$:

```
n = 0; h = Float16(2)^(-n); printlnbits(1 + h/3)
n = 2; h = Float16(2)^(-n); printlnbits(1 + h/3)
n = 4; h = Float16(2)^(-n); printlnbits(1 + h/3)
n = 8; h = Float16(2)^(-n); printlnbits(1 + h/3)
```

It follows if $S/2 < n < S$ that

$$1 \oplus (h \oslash 3) = 1 + h/3 - 2^{-10}/3$$

Therefore

$$(g^{\mathrm{FP}}(h) \ominus 1) \oslash h = 1/3 - 2^{n-10}/3$$

Thus the error grows exponentially with n.

If $n \geq S$ then $1 \oplus (h \oslash 3) = 1$ and we have

$$(g^{\mathrm{FP}}(h) \ominus 1) \oslash h = 0$$

Bounding the error

We can bound the error using the bounds on floating point arithmetic.

Theorem (finite-difference error bound) Let f be twice-differentiable in a neighbourhood of x and assume that $f^{\mathrm{FP}}(x) = f(x) + \delta_x^f$ has uniform absolute accuracy in that neighbourhood, that is:

$$|\delta_x^f| \leq c\epsilon_{\mathrm{m}}$$

for a fixed constant c. Assume for simplicity $h = 2^{-n}$ where $n \leq S$ and $|x| \leq 1$.
Then the finite-difference approximation satisfies

$$(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)) \oslash h = f'(x) + \delta_{x,h}^{\mathrm{FD}}$$

where

$$|\delta_{x,h}^{\mathrm{FD}}| \leq \frac{|f'(x)|}{2}\epsilon_{\mathrm{m}} + Mh + \frac{4c\epsilon_{\mathrm{m}}}{h}$$

for $M = \sup_{x \leq t \leq x+h} |f''(t)|$.

Proof

We have (noting by our assumptions $x \oplus h = x + h$ and that dividing by h will only change the exponent so is exact)

$$\begin{aligned}(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)) \oslash h &= \frac{f(x+h) + \delta_{x+h}^f - f(x) - \delta_x^f}{h}(1+\delta_1) \\ &= \frac{f(x+h) - f(x)}{h}(1+\delta_1) + \frac{\delta_{x+h}^f - \delta_x^f}{h}(1+\delta_1)\end{aligned}$$

where $|\delta_1| \leq \epsilon_{\mathrm{m}}/2$. Applying Taylor's theorem we get

$$(f^{\mathrm{FP}}(x+h) \ominus f^{\mathrm{FP}}(x)) \oslash h = f'(x) + \underbrace{f'(x)\delta_1 + \frac{f''(t)}{2}h(1+\delta_1) + \frac{\delta^f_{x+h} - \delta^f_x}{h}(1+\delta_1)}_{\delta^{\mathrm{FD}}_{x,h}}$$

The bound then follows, using the very pessimistic bound $|1+\delta_1| \leq 2$.

∎

The three-terms of this bound tell us a story: the first term is a fixed (small) error, the second term tends to zero as $h \to 0$, while the last term grows like ϵ_{m}/h as $h \to 0$. Thus we observe convergence while the second term dominates, until the last term takes over.
Of course, a bad upper bound is not the same as a proof that something grows, but it is a good indication of what happens *in general* and suffices to motivate the following heuristic to balance the two sources of errors:

Heuristic (finite-difference with floating-point step) Choose h proportional to $\sqrt{\epsilon_{\mathrm{m}}}$ in finite-differences.

In the case of double precision $\sqrt{\epsilon_{\mathrm{m}}} \approx 1.5 \times 10^{-8}$, which is close to when the observed error begins to increase in our examples.

Remark: While finite differences is of debatable utility for computing derivatives, it is extremely effective in building methods for solving differential equations, as we shall see later. It is also very useful as a "sanity check" if one wants something to compare with for other numerical methods for differentiation.

2. Dual numbers (Forward-mode automatic differentiation)

Automatic differentiation consists of applying functions to special types that determine the derivatives.
Here we do so via *dual numbers*.

> **Definition (Dual numbers)** Dual numbers $\mathbb{D}$ are a commutative ring over the reals generated by 1 and ϵ such that $\epsilon^2 = 0$.
> Dual numbers are typically written as $a + b\epsilon$ where a and b are real.

This is very much analoguous to complex numbers, which are a field generated by 1 and i such that $\mathrm{i}^2 = -1$. Compare multiplication of each number type:

$$\begin{aligned}(a+b\mathrm{i})(c+d\mathrm{i}) &= ac + (bc+ad)\mathrm{i} + bd\mathrm{i}^2 = ac - bd + (bc+ad)\mathrm{i} \\ (a+b\epsilon)(c+d\epsilon) &= ac + (bc+ad)\epsilon + bd\epsilon^2 = ac + (bc+ad)\epsilon\end{aligned}$$

And just as we view $\mathbb{R} \subset \mathbb{C}$ by equating $a \in \mathbb{R}$ with $a + 0\mathrm{i} \in \mathbb{C}$, we can view $\mathbb{R} \subset \mathbb{D}$ by equating $a \in \mathbb{R}$ with $a + 0\epsilon \in \mathbb{D}$.

Connection with differentiation

Applying a polynomial to a dual number $a + b\epsilon$ tells us the derivative at a:

> **Theorem (polynomials on dual numbers)** Suppose p is a polynomial. Then
>
> $$p(a + b\epsilon) = p(a) + bp'(a)\epsilon$$

Proof

It suffices to consider $p(x) = x^n$ for $n \geq 1$ as other polynomials follow from linearity. We proceed by induction:
The case $n = 1$ is trivial. For $n > 1$ we have

$$(a + b\epsilon)^n = (a + b\epsilon)(a + b\epsilon)^{n-1} = (a + b\epsilon)(a^{n-1} + (n-1)ba^{n-2}\epsilon) = a^n + bna^{n-1}\epsilon.$$

∎

We can extend real-valued differentiable functions to dual numbers in a similar manner.
First, consider a standard function with a Taylor series (e.g. $\cos$, $\sin$, $\exp$, etc.)

$$f(x) = \sum_{k=0}^{\infty} f_k x^k$$

so that a is inside the radius of convergence. This leads naturally to a definition on dual numbers:

$$\begin{aligned} f(a + b\epsilon) &= \sum_{k=0}^{\infty} f_k (a + b\epsilon)^k = \sum_{k=0}^{\infty} f_k (a^k + ka^{k-1}b\epsilon) = \sum_{k=0}^{\infty} f_k a^k + \sum_{k=0}^{\infty} f_k k a^{k-1} b\epsilon \\ &= f(a) + bf'(a)\epsilon \end{aligned}$$

More generally, given a differentiable function we can extend it to dual numbers:

Definition (dual extension) Suppose a real-valued function f is differentiable at a. If

$$f(a + b\epsilon) = f(a) + bf'(a)\epsilon$$

then we say that it is a *dual extension at* a.

Thus, for basic functions we have natural extensions:

$$\begin{aligned} \exp(a + b\epsilon) &:= \exp(a) + b\exp(a)\epsilon \\ \sin(a + b\epsilon) &:= \sin(a) + b\cos(a)\epsilon \\ \cos(a + b\epsilon) &:= \cos(a) - b\sin(a)\epsilon \\ \log(a + b\epsilon) &:= \log(a) + \frac{b}{a}\epsilon \\ \sqrt{a + b\epsilon} &:= \sqrt{a} + \frac{b}{2\sqrt{a}}\epsilon \\ |a + b\epsilon| &:= |a| + b\operatorname{sign} a\,\epsilon \end{aligned}$$

provided the function is differentiable at a. Note the last example does not have
a convergent Taylor series (at 0) but we can still extend it where it is differentiable.

Going further, we can add, multiply, and compose such functions:

Lemma (product and chain rule)
If f is a dual extension at $g(a)$ and g
is a dual extension at a, then $q(x) := f(g(x))$ is a dual extension at a.
If f and g are dual extensions at a then
$r(x) := f(x)g(x)$ is also dual extensions at a. In other words:

$$q(a + b\epsilon) = q(a) + bq'(a)\epsilon$$
$$r(a + b\epsilon) = r(a) + br'(a)\epsilon$$

Proof
For q it follows immediately:

$$q(a + b\epsilon) = f(g(a + b\epsilon)) = f(g(a) + bg'(a)\epsilon) = f(g(a)) + bg'(a)f'(g(a))\epsilon = q(a) + bq'(a)\epsilon.$$

For r we have

$$\begin{aligned} r(a + b\epsilon) &= f(a + b\epsilon)g(a + b\epsilon) = (f(a) + bf'(a)\epsilon)(g(a) + bg'(a)\epsilon) \\ &= f(a)g(a) + b(f'(a)g(a) + f(a)g'(a))\epsilon = r(a) + br'(a)\epsilon. \end{aligned}$$

∎

A simple corollary is that any function defined in terms of addition, multiplication, composition, etc.
of functions that are dual with differentiation will be differentiable via dual numbers.

Example (differentiating non-polynomial)

Consider $f(x) = \exp\left(x^2 + \mathrm{e}^x\right)$ by evaluating on the duals:

$$f(1 + \epsilon) = \exp\left(1 + 2\epsilon + \mathrm{e} + \mathrm{e}\epsilon\right) = \exp\left(1 + \mathrm{e}\right) + \exp\left(1 + \mathrm{e}\right)(2 + \mathrm{e})\epsilon$$

and therefore we deduce that

$$f'(1) = \exp\left(1 + \mathrm{e}\right)(2 + \mathrm{e}).$$

Implementation as a special type

We now consider a simple implementation of dual numbers that works on general polynomials:

```
# Dual(a,b) represents a + b*ϵ
struct Dual{T}
    a::T
    b::T
end
```

```
# Dual(a) represents a + 0*ϵ
Dual(a::Real) = Dual(a, zero(a)) # for real numbers we use a + 0ϵ

# Allow for a + b*ϵ syntax
const ϵ = Dual(0, 1)

import Base: +, *, -, /, ^, zero, exp

# support polynomials like 1 + x, x - 1, 2x or x*2 by reducing to Dual
+(x::Real, y::Dual) = Dual(x) + y
+(x::Dual, y::Real) = x + Dual(y)
-(x::Real, y::Dual) = Dual(x) - y
-(x::Dual, y::Real) = x - Dual(y)
*(x::Real, y::Dual) = Dual(x) * y
*(x::Dual, y::Real) = x * Dual(y)

# support x/2 (but not yet division of duals)
/(x::Dual, k::Real) = Dual(x.a/k, x.b/k)

# a simple recursive function to support x^2, x^3, etc.
function ^(x::Dual, k::Integer)
    if k < 0
        error("Not implemented")
    elseif k == 1
        x
    else
        x^(k-1) * x
    end
end

# Algebraic operationds for duals
-(x::Dual) = Dual(-x.a, -x.b)
+(x::Dual, y::Dual) = Dual(x.a + y.a, x.b + y.b)
-(x::Dual, y::Dual) = Dual(x.a - y.a, x.b - y.b)
*(x::Dual, y::Dual) = Dual(x.a*y.a, x.a*y.b + x.b*y.a)

exp(x::Dual) = Dual(exp(x.a), exp(x.a) * x.b)
```

We can also try it on the two polynomials as above:

```
f = x -> 1 + x + x^2
g = x -> 1 + x/3 + x^2
f(ϵ).b, g(ϵ).b
```

The first example exactly computes the derivative, and the second example is exact up to the last bit rounding!
It also works for higher order polynomials:

```
f = x -> 1 + 1.3x + 2.1x^2 + 3.1x^3
f(0.5 + ϵ).b - 5.725
```

It is indeed "accurate to (roughly) 16-digits", the best we can hope for using floating point.

We can use this in "algorithms" as well as simple polynomials. Consider the polynomial $1 + \ldots + x^n$:

```
function s(n, x)
    ret = 1 + x # first two terms
    for k = 2:n
        ret += x^k
    end
    ret
end
s(10, 0.1 + ϵ).b
```

This matches exactly the "true" (up to rounding) derivative:

```
sum((1:10) .* 0.1 .^(0:9))
```

Finally, we can try the more complicated example:

```
f = x -> exp(x^2 + exp(x))
f(1 + ϵ)
```

What makes dual numbers so effective is that, unlike finite differences, they are not prone to disasterous growth due to round-off errors.

Week 3: Structured Matrices

> **YouTube Lectures:**
> Structured Matrices
> Permutation Matrices
> Orthogonal Matrices

We have seen how algebraic operations (`+`, `-`, `*`, `/`) are well-defined for floating point numbers. Now we see how this allows us to do (approximate) linear algebra operations on structured matrices. That is, we consider the following structures:

1. *Dense*: This can be considered unstructured, where we need to store all entries in a vector or matrix. Matrix multiplication reduces directly to standard algebraic operations. Solving linear systems with dense matrices will be discussed later.
2. *Triangular*: If a matrix is upper or lower triangular, we can immediately invert using back-substitution. In practice we store a dense matrix and ignore the upper/lower entries.
3. *Banded*: If a matrix is zero apart from entries a fixed distance from the diagonal it is called banded and this allows for more efficient algorithms. We discuss diagonal, tridiagonal and bidiagonal matrices.
4. *Permutation*: A permutation matrix permutes the rows of a vector.
5. *Orthogonal*: An orthogonal matrix Q satisfies $Q^\top Q = I$, in other words, they are very easy to invert. We discuss the special cases of simple rotations and reflections.

```
using LinearAlgebra, Plots, BenchmarkTools
```

1. Dense vectors and matrices

A `Vector` of a primitive type (like `Int` or `Float64`) is stored consecutively in memory. E.g. if we have a `Vector{Int8}` of length `n` then it is stored as `8n` bits (`n` bytes) in a row.
A `Matrix` is stored consecutively in memory, going down column-by-column. That is,

```
A = [1 2;
     3 4;
     5 6]
```

Is actually stored equivalently to a length `6` vector:

```
vec(A)
```

This is known as *column-major* format.

Remark: Note that transposing `A` is done lazyily and so `A'` stores the entries by row. That is, `A'` is stored in *row-major* format.

Matrix-vector multiplication works as expected:

```
x = [7, 8]
A*x
```

Note there are two ways this can be implemented: either the traditional definition, going row-by-row:

$$\begin{bmatrix} \sum_{j=1}^n a_{1,j} x_j \\ \vdots \\ \sum_{j=1}^n a_{m,j} x_j \end{bmatrix}$$

or going column-by-column:

$$x_1 \mathbf{a}_1 + \cdots + x_n \mathbf{a}_n$$

It is easy to implement either version of matrix-multiplication in terms of the algebraic operations we have learned, in this case just using integer arithmetic:

```
## go row-by-row
function mul_rows(A, x)
```

```
    m,n = size(A)
    # promote_type type finds a type that is compatible with both types, eltype gives the type of
the elements of a vector / matrix
    c = zeros(promote_type(eltype(x),eltype(A)), m)
    for k = 1:m, j = 1:n
        c[k] += A[k, j] * x[j]
    end
    c
end

## go column-by-column
function mul(A, x)
    m,n = size(A)
    # promote_type type finds a type that is compatible with both types, eltype gives the type of
the elements of a vector / matrix
    c = zeros(promote_type(eltype(x),eltype(A)), m)
    for j = 1:n, k = 1:m
        c[k] += A[k, j] * x[j]
    end
    c
end

mul_rows(A, x), mul(A, x)
```

Either implementation will be $O(mn)$ operations. However, the implementation `mul` accesses the entries of `A` going down the column, which happens to be *significantly faster* than `mul_rows`, due to accessing memory of `A` in order. We can see this by measuring the time it takes using `@btime`:

```
n = 1000
A = randn(n,n) # create n x n matrix with random normal entries
x = randn(n) # create length n vector with random normal entries

@btime mul_rows(A,x)
@btime mul(A,x)
@btime A*x; # built-in, high performance implementation. USE THIS in practice
```

Here `ms` means milliseconds (`0.001 = 10^(-3)` seconds) and `μs` means microseconds (`0.000001 = 10^(-6)` seconds).
So we observe that `mul` is roughly 3x faster than `mul_rows`, while the optimised `*` is roughly 5x faster than `mul`.

Remark: (advanced) For floating point types, `A*x` is implemented in BLAS which is generally multi-threaded and is not identical to `mul(A,x)`, that is, some inputs will differ in how the computations are rounded.

Note that the rules of arithmetic apply here: matrix multiplication with floats will incur round-off error (the precise details of which are subject to the implementation):

```
A = [1.4 0.4;
     2.0 1/2]
A * [1, -1] # First entry has round-off error, but 2nd entry is exact
```

And integer arithmetic will be prone to overflow:

```
A = fill(Int8(2^6), 2, 2) # make a matrix whose entries are all equal to 2^6
A * Int8[1,1] # we have overflowed and get a negative number -2^7
```

Solving a linear system is done using `\`:

```
A = [1 2 3;
     1 2 4;
     3 7 8]
b = [10; 11; 12]
A \ b
```

Despite the answer being integer-valued,
here we see that it resorted to using floating point arithmetic,
incurring rounding error.
But it is "accurate to (roughly) 16-digits".
As we shall see, the way solving a linear system works is we first write `A` as a
product of simpler matrices, e.g., a product of triangular matrices.

Remark: (advanced) For floating point types, `A \ x` is implemented in LAPACK, which
like BLAS is generally multi-threaded and in fact different machines may round differently.

2. Triangular matrices

Triangular matrices are represented by dense square matrices where the entries below the
diagonal
are ignored:

```
A = [1 2 3;
     4 5 6;
     7 8 9]
U = UpperTriangular(A)
```

We can see that `U` is storing all the entries of `A`:

```
U.data
```

Similarly we can create a lower triangular matrix by ignoring the entries above the diagonal:

```
L = LowerTriangular(A)
```

If we know a matrix is triangular we can do matrix-vector multiplication in roughly half
the number of operations.
Moreover, we can easily invert matrices.
Consider a simple 3×3 example, which can be solved with `\`:

```
b = [5,6,7]
x = U \ b
```

Behind the seens, `\` is doing back-substitution: considering the last row, we have all zeros apart from the last column so we know that `x[3]` must be equal to:

```
b[3] / U[3,3]
```

Once we know `x[3]`, the second row states `U[2,2]*x[2] + U[2,3]*x[3] == b[2]`, rearranging we get that `x[2]` must be:

```
(b[2] - U[2,3]*x[3])/U[2,2]
```

Finally, the first row states `U[1,1]*x[1] + U[1,2]*x[2] + U[1,3]*x[3] == b[1]` i.e. `x[1]` is equal to

```
(b[1] - U[1,2]*x[2] - U[1,3]*x[3])/U[1,1]
```

More generally, we can solve the upper-triangular system

$$\begin{bmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ & & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

by computing $x_n, x_{n-1}, \ldots, x_1$ by the back-substitution formula:

$$x_k = \frac{b_k - \sum_{j=k+1}^n u_{kj} x_j}{u_{kk}}$$

The problem sheet will explore implementing this method, as well as forward substitution for inverting lower triangular matrices. The cost of multiplying and solving linear systems with a triangular matrix is $O(n^2)$.

3. Banded matrices

A *banded matrix* is zero off a prescribed number of diagonals.
We call the number of (potentially) non-zero diagonals the *bandwidths*:

> **Definition (bandwidths)** A matrix A has *lower-bandwidth* l if $A[k,j] = 0$ for all $k - j > l$ and *upper-bandwidth* u if $A[k,j] = 0$ for all $j - k > u$. We say that it has *strictly lower-bandwidth* l if it has lower-bandwidth l and there exists a j such that $A[j + l, j] \neq 0$. We say that it has *strictly upper-bandwidth* u if it has upper-bandwidth u and there exists a k such that $A[k, k + u] \neq 0$.

Diagonal

> **Definition (Diagonal)** *Diagonal matrices* are square matrices with bandwidths $l = u = 0$.

Diagonal matrices in Julia are stored as a vector containing the diagonal entries:

```
x = [1,2,3]
D = Diagonal(x)
```

It is clear that we can perform diagonal-vector multiplications and solve linear systems involving diagonal matrices efficiently
(in $O(n)$ operations).

Bidiagonal

> **Definition (Bidiagonal)** If a square matrix has bandwidths $(l, u) = (1, 0)$ it is *lower-bidiagonal* and if it has bandwidths $(l, u) = (0, 1)$ it is *upper-bidiagonal*.

We can create Bidiagonal matrices in Julia by specifying the diagonal and off-diagonal:

```
Bidiagonal([1,2,3], [4,5], :L)
```

```
Bidiagonal([1,2,3], [4,5], :U)
```

Multiplication and solving linear systems with Bidiagonal systems is also $O(n)$ operations, using the standard multiplications/back-substitution algorithms but being careful in the loops to only access the non-zero entries.

Tridiagonal

> **Definition (Tridiagonal)** If a square matrix has bandwidths $l = u = 1$ it is *tridiagonal*.

Julia has a type `Tridiagonal` for representing a tridiagonal matrix from its sub-diagonal, diagonal, and super-diagonal:

```
Tridiagonal([1,2], [3,4,5], [6,7])
```

Tridiagonal matrices will come up in second-order differential equations and orthogonal polynomials.
We will later see how linear systems involving tridiagonal matrices can be solved in $O(n)$ operations.

4. Permutation Matrices

Permutation matrices are matrices that represent the action of permuting the entries of a vector,
that is, matrix representations of the symmetric group S_n, acting on $\mathbb{R}^n$.
Recall every $\sigma \in S_n$ is a bijection between $\{1, 2, \dots, n\}$ and itself.
We can write a permutation σ in *Cauchy notation*:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ \sigma_1 & \sigma_2 & \sigma_3 & \cdots & \sigma_n \end{pmatrix}$$

where $\{\sigma_1, \dots, \sigma_n\} = \{1, 2, \dots, n\}$ (that is, each integer appears precisely once).
We denote the *inverse permutation* by σ^{-1}, which can be constructed by swapping the rows of the Cauchy notation and reordering.

We can encode a permutation in vector $\boldsymbol{\sigma} = [\sigma_1, \dots, \sigma_n]^\top$.
This induces an action on a vector (using indexing notation)

$$\mathbf{v}[\boldsymbol{\sigma}] = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}$$

Example (permutation of a vector)
Consider the permutation σ given by

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 2 & 5 & 3 \end{pmatrix}$$

We can apply it to a vector:

```
σ = [1, 4, 2, 5, 3]
v = [6, 7, 8, 9, 10]
v[σ] # we permutate entries of v
```

Its inverse permutation σ^{-1} has Cauchy notation coming from swapping the rows of the Cauchy notation of σ and sorting:

$$\begin{pmatrix} 1 & 4 & 2 & 5 & 3 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 4 & 3 & 5 \\ 1 & 3 & 2 & 5 & 4 \end{pmatrix}$$

Julia has the function `invperm` for computing the vector that encodes the inverse permutation:
And indeed:

```
σ⁻¹ = invperm(σ) # note that ⁻¹ are just unicode characters in the variable name
```

And indeed permuting the entries by `σ` and then by `σ⁻¹` returns us to our original vector:

```
v[σ][σ⁻¹] # permuting by σ and then σ⁻¹ gets us back
```

Note that the operator

$$P_\sigma(\mathbf{v}) = \mathbf{v}[\boldsymbol{\sigma}]$$

is linear in $\mathbf{v}$, therefore, we can identify it with a matrix whose action is:

$$P_\sigma \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_{\sigma_1} \\ \vdots \\ v_{\sigma_n} \end{bmatrix}.$$

The entries of this matrix are

$$P_\sigma[k,j] = \mathbf{e}_k^\top P_\sigma \mathbf{e}_j = \mathbf{e}_k^\top \mathbf{e}_{\sigma_j^{-1}} = \delta_{k,\sigma_j^{-1}} = \delta_{\sigma_k,j}$$

where $\delta_{k,j}$ is the *Kronecker delta*:

$$\delta_{k,j} := \begin{cases} 1 & k = j \\ 0 & \text{otherwise} \end{cases}.$$

This construction motivates the following definition:

> **Definition (permutation matrix)** $P \in \mathbb{R}^{n\times n}$ is a permutation matrix if it is equal to the identity matrix with its rows permuted.

Example (5×5 permutation matrix)
We can construct the permutation representation for σ as above as follows:

```
P = I(5)[σ,:]
```

And indeed, we see its action is as expected:

```
P * v
```

Remark: (advanced) Note that `P` is a special type `SparseMatrixCSC`. This is used to represent a matrix by storing only the non-zero entries as well as their location. This is an important data type in high-performance scientific computing, but we will not be using general sparse matrices in this module.

> **Proposition (permutation matrix inverse)**
> Let P_σ be a permutation matrix corresponding to the permutation σ. Then
>
> $$P_\sigma^\top = P_{\sigma^{-1}} = P_\sigma^{-1}$$

That is, P_σ is *orthogonal*:

$$P_\sigma^\top P_\sigma = P_\sigma P_\sigma^\top = I.$$

Proof

We prove orthogonality via:

$$\mathbf{e}_k^\top P_\sigma^\top P_\sigma \mathbf{e}_j = (P_\sigma \mathbf{e}_k)^\top P_\sigma \mathbf{e}_j = \mathbf{e}_{\sigma_k^{-1}}^\top \mathbf{e}_{\sigma_j^{-1}} = \delta_{k,j}$$

This shows $P_\sigma^\top P_\sigma = I$ and hence $P_\sigma^{-1} = P_\sigma^\top$.

∎

Permutation matrices are examples of sparse matrices that can be very easily inverted.

4. Orthogonal matrices

Definition (orthogonal matrix) A square matrix is *orthogonal* if its inverse is its transpose:

$$Q^\top Q = QQ^\top = I.$$

We have already seen an example of an orthogonal matrices (permutation matrices).
Here we discuss two important special cases: simple rotations and reflections.

Simple rotations

Definition (simple rotation)
A 2×2 *rotation matrix* through angle θ is

$$Q_\theta := \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

In what follows we use the following for writing the angle of a vector:

Definition (two-arg arctan) The two-argument arctan function gives the angle `θ` through the point $[a, b]^\top$, i.e.,

$$\sqrt{a^2+b^2}\begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

It can be defined in terms of the standard arctan as follows:

$$\operatorname{atan}(b,a) := \begin{cases} \operatorname{atan}\frac{b}{a} & a > 0 \\ \operatorname{atan}\frac{b}{a} + \pi & a < 0 \text{ and } b > 0 \\ \operatorname{atan}\frac{b}{a} + \pi & a < 0 \text{ and } b < 0 \\ \pi/2 & a = 0 \text{ and } b > 0 \\ -\pi/2 & a = 0 \text{ and } b < 0 \end{cases}$$

This is available in Julia:

```
atan(-1,-2) # angle through [-2,-1]
```

We can rotate an arbitrary vector to the unit axis. Interestingly it only requires basic algebraic functions (no trigonometric functions):

Proposition (rotation of a vector)
The matrix

$$Q = \frac{1}{\sqrt{a^2+b^2}} \begin{bmatrix} a & b \\ -b & a \end{bmatrix}$$

is a rotation matrix satisfying

$$Q \begin{bmatrix} a \\ b \end{bmatrix} = \sqrt{a^2+b^2} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Proof

The last equation is trivial so the only question is that it is a rotation matrix.
Define $\theta = -\operatorname{atan}(b,a)$. By definition of the two-arg arctan we have

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{bmatrix} = \frac{1}{\sqrt{a^2+b^2}} \begin{bmatrix} a & b \\ -b & a \end{bmatrix}.$$

■

Reflections

In addition to rotations, another type of orthognal matrix are reflections:

Definition (reflection matrix)
Given a vector $\mathbf{v}$ satisfying $\|\mathbf{v}\| = 1$, the reflection matrix is the orthogonal matrix

$$Q_{\mathbf{v}} \triangleq I - 2\mathbf{v}\mathbf{v}^\top$$

These are reflections in the direction of $\mathbf{v}$. We can show this as follows:

Proposition $Q_{\mathbf{v}}$ satisfies:

1. Symmetry: $Q_{\mathbf{v}} = Q_{\mathbf{v}}^\top$
2. Orthogonality: $Q_{\mathbf{v}}^\top Q_{\mathbf{v}} = I$
3. $\mathbf{v}$ is an eigenvector of $Q_{\mathbf{v}}$ with eigenvalue -1
4. $Q_{\mathbf{v}}$ is a rank-1 perturbation of I
5. $\det Q_{\mathbf{v}} = -1$

Proof

Property 1 follows immediately. Property 2 follows from

$$Q_{\mathbf{v}}^\top Q_{\mathbf{v}} = Q_{\mathbf{v}}^2 = I - 4\mathbf{v}\mathbf{v}^\top + 4\mathbf{v}\mathbf{v}^\top\mathbf{v}\mathbf{v}^\top = I$$

Property 3 follows since

$$Q_{\mathbf{v}}\mathbf{v} = -\mathbf{v}$$

Property 4 follows since $\mathbf{v}\mathbf{v}^\top$ is a rank-1 matrix as all rows are linear combinations of each other.
To see property 5, note there is a dimension $n-1$ space W orthogonal to $\mathbf{v}$, that is, for all $\mathbf{w} \in W$ we have $\mathbf{w}^\top\mathbf{v} = 0$, which implies that

$$Q_{\mathbf{v}}\mathbf{w} = \mathbf{w}$$

In other words, 1 is an eigenvalue with multiplicity $n-1$ and -1 is an eigenvalue with multiplicity 1, and thus the product of the eigenvalues is -1.

∎

Example (reflection through 2-vector) Consider reflection through $\mathbf{x} = [1, 2]^\top$.
We first need to normalise $\mathbf{x}$:

$$\mathbf{v} = \frac{\mathbf{x}}{\|\mathbf{x}\|} = \begin{bmatrix} \frac{1}{\sqrt{5}} \\ \frac{2}{\sqrt{5}} \end{bmatrix}$$

Note this indeed has unit norm:

$$\|\mathbf{v}\|^2 = \frac{1}{5} + \frac{4}{5} = 1.$$

Thus the reflection matrix is:

$$Q_{\mathbf{v}} = I - 2\mathbf{v}\mathbf{v}^\top = \begin{bmatrix} 1 & \\ & 1 \end{bmatrix} - \frac{2}{5}\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} = \frac{1}{5}\begin{bmatrix} 3 & -4 \\ -4 & -3 \end{bmatrix}$$

Indeed it is symmetric, and orthogonal. It sends $\mathbf{x}$ to $-\mathbf{x}$:

$$Q_{\mathbf{v}}\mathbf{x} = \frac{1}{5}\begin{bmatrix} 3-8 \\ -4-6 \end{bmatrix} = -\mathbf{x}$$

Any vector orthogonal to $\mathbf{x}$, like $\mathbf{y} = [-2, 1]^\top$, is left fixed:

$$Q_{\mathbf{v}}\mathbf{y} = \frac{1}{5}\begin{bmatrix} -6-4 \\ 8-3 \end{bmatrix} = \mathbf{y}$$

Note that *building* the matrix $Q_{\mathbf{v}}$ will be expensive ($O(n^2)$ operations), but we can apply $Q_{\mathbf{v}}$ to a vector in $O(n)$ operations using the expression:

$$Q_{\mathbf{v}}\mathbf{x} = \mathbf{x} - 2\mathbf{v}(\mathbf{v}^\top\mathbf{x}).$$

Just as rotations can be used to rotate vectors to be aligned with coordinate axis, so can reflections, but in this case it works for vectors in $\mathbb{R}^n$, not just $\mathbb{R}^2$:

Definition (Householder reflection) For a given vector $\mathbf{x}$, define the Householder reflection

$$Q_{\mathbf{x}}^{\pm,\mathrm{H}} := Q_{\mathbf{w}}$$

for $\mathbf{y} = \mp\|\mathbf{x}\|\mathbf{e}_1 + \mathbf{x}$ and $\mathbf{w} = \frac{\mathbf{y}}{\|\mathbf{y}\|}$.
The default choice in sign is:

$$Q_{\mathbf{x}}^{\mathrm{H}} := Q_{\mathbf{x}}^{-\mathrm{sign}(x_1),\mathrm{H}}.$$

Lemma (Householder reflection)

$$Q_{\mathbf{x}}^{\pm,\mathrm{H}}\mathbf{x} = \pm\|\mathbf{x}\|\mathbf{e}_1$$

Proof
Note that

$$\begin{aligned}\|\mathbf{y}\|^2 &= 2\|\mathbf{x}\|^2 \mp 2\|\mathbf{x}\|x_1,\\ \mathbf{y}^\top\mathbf{x} &= \|\mathbf{x}\|^2 \mp \|\mathbf{x}\|x_1\end{aligned}$$

where $x_1 = \mathbf{e}_1^\top\mathbf{x}$. Therefore:

$$Q_{\mathbf{x}}^{\pm,\mathrm{H}}\mathbf{x} = (I - 2\mathbf{w}\mathbf{w}^\top)\mathbf{x} = \mathbf{x} - 2\frac{\mathbf{y}\|\mathbf{x}\|}{\|\mathbf{y}\|^2}(\|\mathbf{x}\| \mp x_1) = \mathbf{x} - \mathbf{y} = \pm\|\mathbf{x}\|\mathbf{e}_1.$$

∎

Why do we choose the the opposite sign of x_1 for the default reflection? For stability.
We demonstrate the reason for this by numerical example. Consider $\mathbf{x} = [1, h]$, i.e., a small perturbation from $\mathbf{e}_1$. If we reflect to $\mathrm{norm}(\mathbf{x})\mathbf{e}_1$ we see a numerical problem:

```
h = 10.0^(-10)
x = [1,h]
y = -norm(x)*[1,0] + x
w = y/norm(y)
Q = I-2w*w'
Q*x
```

It didn't work! Even worse is if `h = 0`:

```
h = 0
x = [1,h]
y = -norm(x)*[1,0] + x
w = y/norm(y)
Q = I-2w*w'
Q*x
```

This is because `y` has large relative error due to cancellation
from floating point errors in computing the first entry `x[1] - norm(x)`.
(Or has norm zero if `h=0`.)
We avoid this cancellation by using the default choice:

```
h = 10.0^(-10)
x = [1,h]
y = sign(x[1])*norm(x)*[1,0] + x
w = y/norm(y)
Q = I-2w*w'
Q*x
```

Week 4: Decompositions and least squares

YouTube Lectures:
Least squares and QR
Gram-Schmidt and Reduced QR
Householder and QR
PLU Decomposition
Cholesky Decomposition

We now look at several decompositions (or factorisations)
of a matrix into products of structured matrices, and their use in solving least squares problems and linear systems.
For a square or rectangular matrix $A \in \mathbb{R}^{m\times n}$ with more rows than columns ($m \geq n$), we consider:

1. The *QR decomposition*

$$A = QR = \underbrace{[\mathbf{q}_1|\cdots|\mathbf{q}_m]}_{m\times m} \underbrace{\begin{bmatrix} \times & \cdots & \times \\ & \ddots & \vdots \\ & & \times \\ & & 0 \\ & & \vdots \\ & & 0 \end{bmatrix}}_{m\times n}$$

where Q is orthogonal ($Q^\top Q = I$, $\mathbf{q}_j \in \mathbb{R}^m$) and R is *right triangular*, which means it
is only nonzero on or to the right of the diagonal.

2. The *reduced QR decomposition*

$$A = QR = \underbrace{[\mathbf{q}_1|\cdots|\mathbf{q}_n]}_{m\times n} \underbrace{\begin{bmatrix} \times & \cdots & \times \\ & \ddots & \vdots \\ & & \times \end{bmatrix}}_{n\times n}$$

where Q has orthogonal columns ($Q^\top Q = I$, $\mathbf{q}_j \in \mathbb{R}^m$) and R is upper triangular.

For a square matrix we consider the *PLU decomposition*:

$$A = P^\top LU$$

where P is a permutation matrix, L is lower triangular and U is upper triangular.

Finally, for a square, *symmetric positive definite* ($\mathbf{x}^\top A\mathbf{x} > 0$ for all $\mathbf{x} \in \mathbb{R}^n, \mathbf{x} \neq 0$)
matrix we consider the *Cholesky decomposition*:

$$A = LL^\top$$

The importance of these decomposition for square matrices is that their component pieces are easy to invert on a computer:

$$\begin{aligned} A = P^\top LU \Rightarrow \qquad & A^{-1}\mathbf{b} = U^{-1}L^{-1}P\mathbf{b} \\ A = QR \Rightarrow \qquad & A^{-1}\mathbf{b} = R^{-1}Q^\top \mathbf{b} \\ A = LL^\top \Rightarrow \qquad & A^{-1}\mathbf{b} = L^{-\top}L^{-1}\mathbf{b} \end{aligned}$$

and we saw last lecture that triangular and orthogonal matrices are easy to invert when applied to a vector $\mathbf{b}$, e.g., using forward/back-substitution.
For rectangular matrices we will see that they lead to efficient solutions to the *least squares problem*: find $\mathbf{x}$ that minimizes the 2-norm

$$\|A\mathbf{x} - \mathbf{b}\|.$$

In this lecture we discuss the followng:

1. QR and least squares: We discuss the QR decomposition and its usage in solving least squares problems.
2. Reduced QR and Gram–Schmidt: We discuss computation of the Reduced QR decomposition using Gram–Schmidt.
3. Householder reflections and QR: We discuss comuting the QR decomposition using Householder reflections.
4. PLU decomposition: we discuss how the LU decomposition can be computed using Gaussian elimination, and the computation of
   the PLU decomposition via Gaussian elimination with pivoting.
5. Cholesky decomposition: we introduce symmetric positive definite matrices and show that their LU decomposition can be re-interpreted
   as a Cholesky decomposition.
6. Timings: we see the relative trade-off in speed between the different decompositions.

```
using LinearAlgebra, Plots, BenchmarkTools
```

1. QR and least squares

Here we consider rectangular matrices with more rows than columns.
A QR decomposition decomposes a matrix into an orthogonal matrix Q times a right triangular matrix R.
Note the QR decomposition contains within it the reduced QR decomposition:

$$A = QR = [Q|\mathbf{q}_{n+1}|\cdots|\mathbf{q}_m]\begin{bmatrix} R \\ \mathbf{0}_{m-n\times n} \end{bmatrix} = QR.$$

We can use it to solve a least squares problem using the norm-preserving property (see PS3) of orthogonal matrices:

$$\|A\mathbf{x} - \mathbf{b}\| = \|QR\mathbf{x} - \mathbf{b}\| = \|R\mathbf{x} - Q^\top\mathbf{b}\| = \left\| \begin{bmatrix} R \\ \mathbf{0}_{m-n\times n} \end{bmatrix}\mathbf{x} - \begin{bmatrix} Q^\top \\ \mathbf{q}_{n+1}^\top \\ \vdots \\ \mathbf{q}_m^\top \end{bmatrix}\mathbf{b} \right\|$$

Now note that the rows $k > n$ are independent of $\mathbf{x}$ and are a fixed contribution. Thus to minimise this norm it suffices to drop them and minimise:

$$\|R\mathbf{x} - Q^\top \mathbf{b}\|$$

This norm is minimisable if it is attained. Provided the column rank of A is full, R will be invertible (Exercise: why is this?). Thus we have the solution

$$\mathbf{x} = R^{-1}Q^\top \mathbf{b}$$

Example (quadratic fit) Suppose we want to fit noisy data by a quadratic

$$p(x) = p_0 + p_1 x + p_2 x^2$$

That is, we want to choose p_0, p_1, p_2 at data samples $x_1, \ldots, x_m$ so that the following is true:

$$p_0 + p_1 x_k + p_2 x_k^2 \approx f_k$$

where f_k are given by data. We can reinterpret this as a least squares problem: minimise the norm

$$\left\| \begin{bmatrix} 1 & x_1 & x_1^2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} - \begin{bmatrix} f_1 \\ \vdots \\ f_m \end{bmatrix} \right\|$$

We can solve this using the QR decomposition:

```
m,n = 100,3

x = range(0,1; length=m) # 100 points
f = 2 .+ x .+ 2x.^2 .+ 0.1 .* randn.() # Noisy quadratic

A = x .^ (0:2)'  # 100 x 3 matrix, equivalent to [ones(m) x x.^2]
Q,R̂ = qr(A)
Q̂ = Q[:,1:n] # Q represents full orthogonal matrix so we take first 3 columns

p₀,p₁,p₂ = R̂ \ Q̂'f
```

We can visualise the fit:

```
p = x -> p₀ + p₁*x + p₂*x^2

scatter(x, f; label="samples", legend=:bottomright)
plot!(x, p.(x); label="quadratic")
```

Note that `\` with a rectangular system does least squares by default:

```
A \ f
```

2. Reduced QR and Gram–Schmidt

How do we compute the QR decomposition? We begin with a method
you may have seen before in another guise. Write

$$A = [\mathbf{a}_1|\dots|\mathbf{a}_n]$$

where $\mathbf{a}_k \in \mathbb{R}^m$ and assume they are linearly independent (A has full column rank).
Note that the column span of the first j columns of A
will be the same as the first j columns of Q, as
R must be non-singular:

$$\mathrm{span}(\mathbf{a}_1, \dots, \mathbf{a}_j) = \mathrm{span}(\mathbf{q}_1, \dots, \mathbf{q}_j)$$

In other words: the columns of Q are an orthogonal basis
of the column span of A.
To see this note that since `R̂` is triangular we have

$$[\mathbf{a}_1|\dots|\mathbf{a}_j] = [\mathbf{q}_1|\dots|\mathbf{q}_j]R[1:j, 1:j]$$

for all j. That is, if $\mathbf{v} \in \mathrm{span}(\mathbf{a}_1, \dots, \mathbf{a}_j)$ we have for $\mathbf{c} \in \mathbb{R}^j$

$$\mathbf{v} = [\mathbf{a}_1|\dots|\mathbf{a}_j]\mathbf{c} = [\mathbf{q}_1|\dots|\mathbf{q}_j]R[1:j, 1:j]\mathbf{c} \in \mathrm{span}(\mathbf{q}_1, \dots, \mathbf{q}_j)$$

while if $\mathbf{w} \in \mathrm{span}(\mathbf{q}_1, \dots, \mathbf{q}_j)$ we have for $\mathbf{d} \in \mathbb{R}^j$

$$\mathbf{w} = [\mathbf{q}_1|\dots|\mathbf{q}_j]\mathbf{d} = [\mathbf{a}_1|\dots|\mathbf{a}_j]R[1:j, 1:j]^{-1}\mathbf{d} \in \mathrm{span}(\mathbf{a}_1, \dots, \mathbf{a}_j).$$

It is possible to find an orthogonal basis using the *Gram–Schmidt algorithm*,
We construct it via induction:
assume that

$$\mathrm{span}(\mathbf{a}_1, \dots, \mathbf{a}_{j-1}) = \mathrm{span}(\mathbf{q}_1, \dots, \mathbf{q}_{j-1})$$

where $\mathbf{q}_1, \dots, \mathbf{q}_{j-1}$ are orthogonal:

$$\mathbf{q}_k^\top \mathbf{q}_\ell = \delta_{k\ell} = \begin{cases} 1 & k = \ell \\ 0 & \text{otherwise} \end{cases}.$$

for $k, \ell < j$.
Define

$$\mathbf{v}_j := \mathbf{a}_j - \sum_{k=1}^{j-1} \underbrace{\mathbf{q}_k^\top \mathbf{a}_j}_{r_{kj}} \mathbf{q}_k$$

so that for $k < j$

$$\mathbf{q}_k^\top \mathbf{v}_j = \mathbf{q}_k^\top \mathbf{a}_j - \sum_{k=1}^{j-1} \underbrace{\mathbf{q}_k^\top \mathbf{a}_j}_{r_{kj}} \mathbf{q}_k^\top \mathbf{q}_k = 0.$$

Then we define

$$\mathbf{q}_j := \frac{\mathbf{v}_j}{\|\mathbf{v}_j\|}.$$

which sastisfies $\mathbf{q}_k^\top \mathbf{q}_j = \delta_{kj}$ for $k \le j$.

We now reinterpret this construction as a reduced QR decomposition.
Define $r_{jj} := \|\mathbf{v}_j\|$
Then rearrange the definition we have

$$\mathbf{a}_j = [\mathbf{q}_1 | \cdots | \mathbf{q}_j] \begin{bmatrix} r_{1j} \\ \vdots \\ r_{jj} \end{bmatrix}$$

Thus

$$[\mathbf{a}_1 | \cdots | \mathbf{a}_j] \begin{bmatrix} r_{11} & \cdots & r_{1j} \\ & \ddots & \vdots \\ & & r_{jj} \end{bmatrix}$$

That is, we are computing the reduced QR decomposition column-by-column.
Running this algorithm to $j = n$ completes the decomposition.

Gram–Schmidt in action

We are going to compute the reduced QR of a random matrix

```
m,n = 5,4
A = randn(m,n)
Q,R̂ = qr(A)
Q̂ = Q[:,1:n]
```

The first column of `Q̂` is indeed a normalised first column of `A`:

```
R = zeros(n,n)
Q = zeros(m,n)
R[1,1] = norm(A[:,1])
Q[:,1] = A[:,1]/R[1,1]
```

We now determine the next entries as

```
R[1,2] = Q[:,1]'A[:,2]
v = A[:,2] - Q[:,1]*R[1,2]
R[2,2] = norm(v)
Q[:,2] = v/R[2,2]
```

And the third column is then:

```
R[1,3] = Q[:,1]'A[:,3]
R[2,3] = Q[:,2]'A[:,3]
v = A[:,3] - Q[:,1:2]*R[1:2,3]
R[3,3] = norm(v)
Q[:,3] = v/R[3,3]
```

(Note the signs may not necessarily match.)

We can clean this up as a simple algorithm:

```
function gramschmidt(A)
    m,n = size(A)
    m ≥ n || error("Not supported")
    R = zeros(n,n)
    Q = zeros(m,n)
    for j = 1:n
        for k = 1:j-1
            R[k,j] = Q[:,k]'*A[:,j]
        end
        v = A[:,j] - Q[:,1:j-1]*R[1:j-1,j]
        R[j,j] = norm(v)
        Q[:,j] = v/R[j,j]
    end
    Q,R
end

Q,R = gramschmidt(A)
norm(A - Q*R)
```

Complexity and stability

We see within the `for j = 1:n` loop that we have $O(mj)$ operations. Thus the total complexity is $O(mn^2)$ operations.

Unfortunately, the Gram–Schmidt algorithm is *unstable*: the rounding errors when implemented in floating point accumulate in a way that we lose orthogonality:

```
A = randn(300,300)
Q,R = gramschmidt(A)
norm(Q'Q-I)
```

3. Householder reflections and QR

As an alternative, we will consider using Householder reflections to introduce zeros below the diagonal.
Thus, if Gram–Schmidt is a process of *triangular orthogonalisation* (using triangular matrices to orthogonalise), Householder reflections is a process of *orthogonal triangularisation* (using orthogonal matrices to triangularise).

Consider multiplication by the Householder reflection corresponding to the first column, that is, for

$$Q_1 := Q^{\mathrm{H}}_{\mathbf{a}_1},$$

consider

$$Q_1 A = \begin{bmatrix} \times & \times & \cdots & \times \\ & \times & \cdots & \times \\ & \vdots & \ddots & \vdots \\ & \times & \cdots & \times \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & \mathbf{a}^1_2 & \cdots & \mathbf{a}^1_n \end{bmatrix}$$

where

$$r_{1j} := (Q_1 \mathbf{a}_j)[1] \qquad \text{and} \qquad \mathbf{a}^1_j := (Q_1 \mathbf{a}_j)[2:m],$$

noting $r_{11} = -\mathrm{sign}(a_{11})\|\mathbf{a}_1\|$ and all entries of $\mathbf{a}^1_1$ are zero (thus not included).
That is, we have made the first column triangular.

But now consider

$$Q_2 := \begin{bmatrix} 1 & \\ & Q^{\mathrm{H}}_{\mathbf{a}^1_2} \end{bmatrix} = Q^H_{\begin{bmatrix} 0 \\ \mathbf{a}^1_2 \end{bmatrix}}$$

so that

$$Q_2Q_1A = \begin{bmatrix} \times & \times & \times & \cdots & \times \\ & \times & \times & \cdots & \times \\ & & \vdots & \ddots & \vdots \\ & & \times & \cdots & \times \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & \cdots & r_{1n} \\ & r_{22} & r_{23} & \cdots & r_{2n} \\ & & \mathbf{a}_3^2 & \cdots & \mathbf{a}_n^2 \end{bmatrix}$$

where

$$r_{2j} := (Q_2\mathbf{a}_j^1)[1] \qquad \text{and} \qquad \mathbf{a}_j^2 := (Q_2\mathbf{a}_j^1)[2:m-1]$$

Thus the first two columns are triangular.

The inductive construction is thus clear. If we define $\mathbf{a}_j^0 := \mathbf{a}_j$ we have the construction

$$\begin{aligned} Q_j &:= \begin{bmatrix} I_{j-1\times j-1} & \\ & Q_{\mathbf{a}_j^j}^{\pm,\mathrm{H}} \end{bmatrix} \\ \mathbf{a}_j^k &:= (Q_k\mathbf{a}_j^{k-1})[2:m-k+1] \\ r_{kj} &:= (Q_k\mathbf{a}_j^{k-1})[1] \end{aligned}$$

Then

$$Q_n\cdots Q_1A = \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \\ & & 0 \\ & & \vdots \\ & & 0 \end{bmatrix}}_{R}$$

i.e.

$$A = \underbrace{Q_1\cdots Q_n}_{Q}R.$$

The implementation is cleaner. We do a naive implementation here:

```
function householderreflection(x)
    y = copy(x)
    # we cannot use sign(x[1]) in case x[1] == 0
    y[1] += (x[1] ≥ 0 ? 1 : -1)*norm(x)
    w = y/norm(y)
    I - 2*w*w'
end
function householderqr(A)
```

```
    m,n = size(A)
    R = copy(A)
    Q = Matrix(1.0I, m, m)
    for j = 1:n
        Qⱼ = householderreflection(R[j:end,j])
        R[j:end,:] = Qⱼ*R[j:end,:]
        Q[:,j:end] = Q[:,j:end]*Qⱼ
    end
    Q,R
end

m,n = 7,5
A = randn(m, n)
Q,R = householderqr(A)
Q*R ≈ A
```

Note because we are forming a full matrix representation of each Householder reflection this is a slow algorithm, taking $O(n^4)$ operations. The problem sheet will consider a better implementation that takes $O(n^3)$ operations.

Example We will now do an example by hand. Consider the 4×3 matrix

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 0 & 15 & 18 \\ -2 & -4 & -4 \\ -2 & -4 & -10 \end{bmatrix}$$

For the first column we have

$$Q_1 = I - \frac{1}{12}\begin{bmatrix} 4 \\ 0 \\ -2 \\ -2 \end{bmatrix}\begin{bmatrix} 4 & 0 & -2 & -2 \end{bmatrix} = \frac{1}{3}\begin{bmatrix} -1 & 0 & 2 & 2 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & 2 & -1 \\ 2 & 0 & -1 & 2 \end{bmatrix}$$

so that

$$Q_1 A = \begin{bmatrix} -3 & -6 & -9 \\ & 15 & 18 \\ & 0 & 0 \\ & 0 & -6 \end{bmatrix}$$

In this example the next column is already upper-triangular,
but because of our choice of reflection we will end up swapping the sign, that is

$$Q_2 = \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

so that

$$Q_2Q_1A = \begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & 0 & 0 \\ & 0 & -6 \end{bmatrix}$$

The final reflection is

$$Q_3 = \begin{bmatrix} I_{2\times 2} & \\ & I - \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} å1 & & \\ & 1 & \\ & & 0 & 1 \\ & & 1 & 0 \end{bmatrix}$$

giving us

$$Q_3Q_2Q_1A = \underbrace{\begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & & -6 \\ & & 0 \end{bmatrix}}_{R}$$

That is,

$$A = Q_1Q_2Q_3R = \underbrace{\frac{1}{3}\begin{bmatrix} -1 & 0 & 2 & 2 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & -1 & 2 \\ 2 & 0 & 2 & -1 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & & -6 \\ & & 0 \end{bmatrix}}_{R} = \underbrace{\frac{1}{3}\begin{bmatrix} -1 & 0 & 2 \\ 0 & 3 & 0 \\ 2 & 0 & -1 \\ 2 & 0 & 2 \end{bmatrix}}_{\widehat{Q}} \underbrace{\begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & & -6 \end{bmatrix}}_{\widehat{R}}$$

2. PLU Decomposition

Just as Gram–Schmidt can be reinterpreted as a reduced QR decomposition,
Gaussian elimination with pivoting can be interpreted as a PLU decomposition.

Special "one-column" lower triangular matrices

Consider the following set of $n \times n$ lower triangular
matrices which equals identity apart from one-column:

$$\mathcal{L}_j := \left\{ I + \begin{bmatrix} \mathbf{0}_j \\ \mathbf{l}_j \end{bmatrix} \mathbf{e}_j^\top : \mathbf{l}_j \in \mathbb{R}^{n-j} \right\}$$

where $\mathbf{0}_j$ denotes the zero vector of length j.
That is, if $L_j \in \mathcal{L}_j$ then it is equal to the identity matrix apart from in the jth column:

$$L_j = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & \ell_{j+1,j} & 1 & \\ & & \vdots & & \ddots \\ & & \ell_{n,j} & & 1 \end{bmatrix} =$$

These satisify the following special properties:

Proposition (one-column lower triangular inverse)
If $L_j \in \mathcal{L}_j$ then

$$L_j^{-1} = I - \begin{bmatrix} \mathbf{0}_j \\ \mathbf{l}_j \end{bmatrix} \mathbf{e}_j^\top = \begin{bmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & -\ell_{j+1,j} & 1 & \\ & & \vdots & & \ddots \\ & & -\ell_{n,j} & & 1 \end{bmatrix} \in \mathcal{L}_j.$$

Proposition (one-column lower triangular multiplication)
If $L_j \in \mathcal{L}_j$ and $L_k \in \mathcal{L}_k$ for $k \geq j$ then

$$L_j L_k = I + \begin{bmatrix} \mathbf{0}_j \\ \mathbf{l}_j \end{bmatrix} \mathbf{e}_j^\top + \begin{bmatrix} \mathbf{0}_k \\ \mathbf{l}_k \end{bmatrix} \mathbf{e}_k^\top$$

Lemma (one-column lower triangular with pivoting)
If σ is a permutation that leaves the first j
rows fixed (that is, $\sigma_\ell = \ell$ for $\ell \leq j$) and $L_j \in \mathcal{L}_j$ then

$$P_\sigma L_j = \tilde{L}_j P_\sigma$$

where $\tilde{L}_j \in \mathcal{L}_j$.

Proof
Write

$$P_\sigma = \begin{bmatrix} I_j & \\ & P_\tau \end{bmatrix}$$

where τ is the permutation with Cauchy notation

$$\begin{pmatrix} 1 & \cdots & n-j \\ \sigma_{j+1}-j & \cdots & \sigma_n - j \end{pmatrix}$$

Then we have

$$P_\sigma L_j = P_\sigma + \begin{bmatrix} \mathbf{0}_j \\ P_\tau \mathbf{l}_j \end{bmatrix} \mathbf{e}_j^\top = \underbrace{(I + \begin{bmatrix} \mathbf{0}_j \\ P_\tau \mathbf{l}_j \end{bmatrix} \mathbf{e}_j^\top)}_{\tilde{L}_j} P_\sigma$$

noting that $\mathbf{e}_j^\top P_\sigma = \mathbf{e}_j^\top$ (as $\sigma_j = j$).
∎

LU Decomposition

Before discussing pivoting, consider standard Gaussian elimation where one row-reduces to introduce zeros column-by-column. We will mimick the computation of the QR decomposition to view this as a *triangular triangularisation*.

Consider the matrix

$$L_1 = \begin{bmatrix} 1 & & & \\ -\frac{a_{21}}{a_{11}} & 1 & & \\ \vdots & & \ddots & \\ -\frac{a_{n1}}{a_{11}} & & & 1 \end{bmatrix} = I - \begin{bmatrix} 0 \\ \frac{\mathbf{a}_1[2:n]}{\mathbf{a}_1[1]} \end{bmatrix} \mathbf{e}_1^\top.$$

We then have

$$L_1 A = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & \mathbf{a}_2^1 & \cdots & \mathbf{a}_n^1 \end{bmatrix}$$

where $\mathbf{a}_j^1 := (L_1 \mathbf{a}_j)[2:n]$ and $u_{1j} = a_{1j}$. But now consider

$$L_2 := I - \begin{bmatrix} 0 \\ \frac{\mathbf{a}_2^1[2:n-1]}{\mathbf{a}_2^1[1]} \end{bmatrix} \mathbf{e}_1^\top.$$

Then

$$L_2 L_1 A = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ & u_{22} & u_{23} & \cdots & u_{2n} \\ & & \mathbf{a}_3^2 & \cdots & \mathbf{a}_n^2 \end{bmatrix}$$

where

$$u_{2j} := (\mathbf{a}_j^1)[1] \qquad \text{and} \qquad \mathbf{a}_j^2 := (L_2 \mathbf{a}_j^1)[2:n-1]$$

Thus the first two columns are triangular.

The inductive construction is again clear. If we define $\mathbf{a}_j^0 := \mathbf{a}_j$ we have the construction

$$\begin{aligned} L_j &:= I - \begin{bmatrix} \mathbf{0}_j \\ \frac{\mathbf{a}_{j+1}^j[2:n-j]}{\mathbf{a}_{j+1}^j[1]} \end{bmatrix} \mathbf{e}_j^\top \\ \mathbf{a}_j^k &:= (L_k \mathbf{a}_j^{k-1})[2:n-k+1] \\ u_{kj} &:= (L_k \mathbf{a}_j^{k-1})[1] \end{aligned}$$

Then

$$L_{n-1} \cdots L_1 A = \underbrace{\begin{bmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ & & u_{nn} \end{bmatrix}}_U$$

i.e.

$$A = \underbrace{L_1^{-1} \cdots L_{n-1}^{-1}}_L U.$$

Writing

$$L_j = I + \begin{bmatrix} \mathbf{0}_j \\ \ell_{j+1,j} \\ \vdots \\ \ell_{n,j} \end{bmatrix} \mathbf{e}_j^\top$$

and using the properties of inversion and multiplication we therefore deduce

$$L = \begin{bmatrix} 1 & & & & \\ -\ell_{21} & 1 & & & \\ -\ell_{31} & -\ell_{32} & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ -\ell_{n1} & -\ell_{n2} & \cdots & -\ell_{n,n-1} & 1 \end{bmatrix}$$

Example (computer)
We will do a numerical example (by-hand is equivalent to Gaussian elimination).
The first lower triangular matrix is:

```
n = 4
A = randn(n,n)
L₁ = I -[0; A[2:end,1]/A[1,1]] * [1 zeros(1,n-1)]
```

Which indeed introduces zeros in the first column:

```
A₁ = L₁*A
```

Now we make the next lower triangular operator:

```
L₂ = I - [0; 0; A₁[3:end,2]/A₁[2,2]] * [0 1 zeros(1,n-2)]
```

So that

```
A₂ = L₂*L₁*A
```

The last one is:

```
L₃ = I - [0; 0; 0; A₂[4:end,3]/A₂[3,3]] * [0 0 1 zeros(1,n-3)]
```

Giving us

```
U = L₃*L₂*L₁*A
```

and

```
L = inv(L₁)*inv(L₂)*inv(L₃)
```

Note how the entries of `L` are indeed identical to the negative lower entries of `L₁`, `L₂` and `L₃`.

Example (by-hand)

Consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix}$$

We have

$$L_2L_1A = L_2\begin{bmatrix} 1 & & \\ -2 & 1 & \\ -1 & & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix} = \begin{bmatrix} 1 & & \\ & 1 & \\ & -\frac{3}{2} & 1 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 \\ & 2 & 6 \\ & 3 & 8 \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 \\ & 2 & 6 \\ & & -1 \end{bmatrix}}_{U}$$

We then deduce L by taking the negative of the lower
entries of L_1, L_2:

$$L = \begin{bmatrix} 1 & & \\ 2 & 1 & \\ 1 & \frac{3}{2} & 1 \end{bmatrix}.$$

PLU Decomposition

We learned in first year linear algebra that if a diagonal entry is zero
when doing Gaussian elimnation one has to *row pivot*. For stability,
in implementation one *always* pivots: swap the largest in magnitude entry for the entry on the diagonal.
In terms of a decomposition, this leads to

$$L_{n-1}P_{n-1}\cdots P_2L_1P_1A = U$$

where P_j is a permutation that leaves rows 1 through $j-1$ fixed,
and swaps row j with a row $k \geq j$ whose entry is maximal in absolute value.

Thus we can deduce that

$$L_{n-1}P_{n-1}\cdots P_2L_1P_1 = \underbrace{L_{n-1}\tilde{L}_{n-2}\cdots\tilde{L}_1}_{L^{-1}}\underbrace{P_{n-1}\cdots P_2P_1}_{P}.$$

where the tilde denotes the combined actions of swapping the permutation and lower-triangular matrices, that is,

$$P_{n-1}\cdots P_{j+1}L_j = \tilde{L}_jP_{n-1}\cdots P_{j+1}.$$

where $\tilde{L}_j \in \mathcal{L}_j$.
The entries of L are then again the negative of the entries below the diagonal of $L_{n-1}, \tilde{L}_{n-2}, \ldots, \tilde{L}_1$.

Writing

$$\tilde{L}_j = I + \begin{bmatrix} \mathbf{0}_j \\ \tilde{\ell}_{j+1,j} \\ \vdots \\ \tilde{\ell}_{n,j} \end{bmatrix}\mathbf{e}_j^\top$$

and using the properties of inversion and multiplication we therefore deduce

$$L = \begin{bmatrix} 1 & & & & & \\ -\tilde{\ell}_{21} & 1 & & & & \\ -\tilde{\ell}_{31} & -\tilde{\ell}_{32} & 1 & & & \\ \vdots & \vdots & \ddots & \ddots & & \\ -\tilde{\ell}_{n-1,1} & -\tilde{\ell}_{n-1,2} & \cdots & -\tilde{\ell}_{n-1,n-2} & 1 & \\ -\tilde{\ell}_{n1} & -\tilde{\ell}_{n2} & \cdots & -\tilde{\ell}_{n,n-2} & -\ell_{n,n-1} & 1 \end{bmatrix}$$

Example

Again we consider the matrix

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix}$$

Even though $a_{11} = 1 \neq 0$, we still pivot: placing
the maximum entry on the diagonal to mitigate numerical errors.
That is, we first pivot and upper triangularise the first column:

$$L_1 P_1 A = L_1 \begin{bmatrix} 0 & 1 & \\ 1 & 0 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 1 & 4 & 9 \end{bmatrix} = \begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ -\frac{1}{2} & & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & 8 \\ 1 & 1 & 1 \\ 1 & 4 & 9 \end{bmatrix}$$

We now pivot and upper triangularise the second column:

$$L_2 P_2 L_1 P_1 A = L_2 \begin{bmatrix} 1 & & \\ & 0 & 1 \\ & 1 & 0 \end{bmatrix} \begin{bmatrix} 2 & 4 & 8 \\ 0 & -1 & -3 \\ 0 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 1 & & \\ & 1 & \\ & \frac{1}{2} & 1 \end{bmatrix} \begin{bmatrix} 2 & 4 & 8 \\ 0 & 2 & 5 \\ 0 & -1 & -3 \end{bmatrix} = \underbrace{\begin{bmatrix} 2 & 4 & 8 \\ 0 & 2 & 5 \\ 0 & 0 & -\frac{1}{2} \end{bmatrix}}_{U}$$

We now move P_2 to the right:

$$L_2 P_2 L_1 P_1 = \underbrace{\begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ -\frac{1}{2} & \frac{1}{2} & 1 \end{bmatrix}}_{L_2 \tilde{L}_1} \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}}_{P}$$

That is

$$L = \begin{bmatrix} 1 & & \\ \frac{1}{2} & 1 & \\ \frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix}$$

We see how this example is done on a computer:

```
A = [1 1 1;
     2 4 8;
     1 4 9]
L,U,σ = lu(A) # σ is a vector encoding the permutation
```

The permutation is

```
σ
```

Thus to invert a system we can do:

```
b = randn(3)
U\(L\b[σ]) == A\b
```

Note the entries match exactly because this precisely what `\` is using.

3. Cholesky Decomposition

Cholesky Decomposition is a form of Gaussian elimination (without pivoting)
that exploits symmetry in the problem, resulting in a substantial speedup.
It is only relevant for *symmetric positive definite*
matrices.

Definition (positive definite) A square matrix $A \in \mathbb{R}^{n\times n}$ is *positive definite* if
for all $\mathbf{x} \in \mathbb{R}^n, x \neq 0$ we have

$$\mathbf{x}^\top A\mathbf{x} > 0$$

First we establish some basic properties of positive definite matrices:

Proposition (conj. pos. def.) If $A \in \mathbb{R}^{n\times n}$ is positive definite and
$V \in \mathbb{R}^{n\times n}$ is non-singular then

$$V^\top AV$$

is positive definite.

Proposition (diag positivity) If $A \in \mathbb{R}^{n\times n}$ is positive definite
then its diagonal entries are positive: $a_{kk} > 0$.

Theorem (subslice pos. def.) If $A \in \mathbb{R}^{n\times n}$ is positive definite
and $\mathbf{k} \in \{1,\ldots,n\}^m$ is a vector of m integers where any integer appears only once,
then $A[\mathbf{k},\mathbf{k}] \in \mathbb{R}^{m\times m}$ is also
positive definite.

We leave the proofs to the problem sheets. Here is the key result:

Theorem (Cholesky and sym. pos. def.) A matrix A is symmetric positive definite if and only if it has a Cholesky decomposition

$$A = LL^\top$$

where the diagonals of L are positive.

Proof If A has a Cholesky decomposition it is symmetric ($A^\top = (LL^\top)^\top = A$) and for $\mathbf{x} \neq 0$ we have

$$\mathbf{x}^\top A\mathbf{x} = (L\mathbf{x})^\top L\mathbf{x} = \|L\mathbf{x}\|^2 > 0$$

where we use the fact that L is non-singular.

For the other direction we will prove it by induction, with the 1×1 case being trivial.
Write

$$A = \begin{bmatrix} \alpha & \mathbf{v}^\top \\ \mathbf{v} & K \end{bmatrix} = \underbrace{\begin{bmatrix} \sqrt{\alpha} & \\ \frac{\mathbf{v}}{\sqrt{\alpha}} & I \end{bmatrix}}_{L_1} \underbrace{\begin{bmatrix} 1 & \\ & K - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha} \end{bmatrix}}_{A_1} \underbrace{\begin{bmatrix} \sqrt{\alpha} & \frac{\mathbf{v}^\top}{\sqrt{\alpha}} \\ & I \end{bmatrix}}_{L_1^\top}.$$

Note that $K - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha}$ is a subslice of $A_1 = L_1^{-1} A L_1^{-\top}$, hence by the previous propositions is itself symmetric positive definite. Thus we can write

$$K - \frac{\mathbf{v}\mathbf{v}^\top}{\alpha} = \tilde{L}\tilde{L}^\top$$

and hence $A = LL^\top$ for

$$L = L_1 \begin{bmatrix} 1 & \\ & \tilde{L} \end{bmatrix}.$$

satisfies $A = LL^\top$.
∎

Note hidden in this proof is a simple algorithm form computing the Cholesky decomposition.
We define

$$
\begin{aligned}
A_1 &:= A \\
\mathbf{v}_k &:= A_k[2:n-k+1,1] \\
\alpha_k &:= A_k[1,1] \\
A_{k+1} &:= A_k[2:n-k+1,2:n-k+1] - \frac{\mathbf{v}_k \mathbf{v}_k^\top}{\alpha_k}.
\end{aligned}
$$

Then

$$
L = \begin{bmatrix}
\sqrt{\alpha_1} & & & & \\
\frac{\mathbf{v}_1[1]}{\sqrt{\alpha_1}} & \sqrt{\alpha_2} & & & \\
\frac{\mathbf{v}_1[2]}{\sqrt{\alpha_1}} & \frac{\mathbf{v}_2[1]}{\sqrt{\alpha_2}} & \sqrt{\alpha_2} & & \\
\vdots & \vdots & \ddots & \ddots & \\
\frac{\mathbf{v}_1[n-1]}{\sqrt{\alpha_1}} & \frac{\mathbf{v}_2[n-2]}{\sqrt{\alpha_2}} & \cdots & \frac{\mathbf{v}_{n-1}[1]}{\sqrt{\alpha_{n-1}}} & \sqrt{\alpha_n}
\end{bmatrix}
$$

This algorithm succeeds if and only if A is symmetric positive definite.

Example Consider the matrix

$$
A_0 = A = \begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{bmatrix}
$$

Then

$$
A_1 = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} - \frac{1}{2} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 3 & 1 & 1 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix}
$$

Continuing, we have

$$
A_2 = \frac{1}{2} \left(\begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} - \frac{1}{3} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} \right) = \frac{1}{3} \begin{bmatrix} 4 & 1 \\ 1 & 4 \end{bmatrix}
$$

Finally

$$
A_3 = \frac{5}{4}.
$$

Thus we get

$$L = L_1 L_2 L_3 = \begin{bmatrix} \sqrt{2} & & & \\ \frac{1}{\sqrt{2}} & \frac{\sqrt{3}}{2} & & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{2}{\sqrt{3}} & \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{6}} & \frac{1}{\sqrt{12}} & \frac{\sqrt{5}}{2} \end{bmatrix}$$

4. Timings

The different decompositions have trade-offs between speed and stability.
First we compare the speed of the different decompositions on a symmetric positive definite matrix, from fastest to slowest:

```
n = 100
A = Symmetric(rand(n,n)) + 100I # shift by 10 ensures positivity
@btime cholesky(A);
@btime lu(A);
@btime qr(A);
```

On my machine, `cholesky` is ~1.5x faster than `lu`,
which is ~2x faster than QR.

In terms of stability, QR computed with Householder reflections
(and Cholesky for positive definite matrices) are stable,
whereas LU is usually unstable (unless the matrix
is diagonally dominant). PLU is a very complicated story: in theory it is unstable,
but the set of matrices for which it is unstable is extremely small, so small one does not
normally run into them.

Here is an example matrix that is in this set.

```
function badmatrix(n)
    A = Matrix(1I, n, n)
    A[:,end] .= 1
    for j = 1:n-1
        A[j+1:end,j] .= -1
    end
    A
end
A = badmatrix(5)
```

Note that pivoting will not occur (we do not pivot as the entries below the diagonal are the same magnitude as the diagonal), thus the PLU Decomposition is equivalent to an LU decomposition:

```
L,U = lu(A)
```

But here we see an issue: the last column of `U` is growing exponentially fast! Thus when `n` is large
we get very large errors:

```
n = 100
b = randn(n)
A = badmatrix(n)
norm(A\b - qr(A)\b) # A \ b still uses lu
```

Note `qr` is completely fine:

```
norm(qr(A)\b - qr(big.(A)) \b) # roughly machine precision
```

Amazingly, PLU is fine if applied to a small perturbation of `A`:

```
ε = 0.000001
Aε = A .+ ε .* randn.()
norm(Aε \ b - qr(Aε) \ b) # Now it matches!
```

The big *open problem* in numerical linear algebra is to prove that the set of matrices
for which PLU fails has extremely small measure.

Week 5: Singular values and conditioning

> **YouTube Lectures:**
> Matrix Norms
> Singular Value Decomposition

In this lecture we discuss matrix and vector norms. The matrix 2-norm involves
singular values, which are a measure of how matrices "stretch" vectors.
We also show that
the singular values of a matrix give a notion of a *condition number*, which allows us
to bound errors introduced by floating point numbers in linear algebra operations.

1. Vector norms: we discuss the standard p-norm for vectors in $\mathbb{R}^n$.
2. Matrix norms: we discuss how two vector norms can be used to induce a norm on matrices. These satisfy an additional multiplicative inequality.
3. Singular value decomposition: we introduce the singular value decomposition which is related to the matrix 2-norm and best low rank approximation.
4. Condition numbers: we will see how errors in matrix-vector multiplication and solving linear systems can be bounded in terms of the *condition number*, which is defined in terms of singular values.

```
using LinearAlgebra, Plots
```

1. Vector norms

Recall the definition of a (vector-)norm:

> **Definition (vector-norm)** A norm $\|\cdot\|$ on $\mathbb{R}^n$ is a function that satisfies the following, for $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and $c \in \mathbb{R}$:

1. Triangle inequality: $\|\mathbf{x}+\mathbf{y}\| \le \|\mathbf{x}\| + \|\mathbf{y}\|$
2. Homogeneity: $\|c\mathbf{x}\| = |c|\|\mathbf{x}\|$
3. Positive-definiteness: $\|\mathbf{x}\| = 0$ implies that $\mathbf{x} = 0$.

Consider the following example:

Definition (p-norm)
For $1 \le p < \infty$ and $\mathbf{x} \in \mathbb{R}^n$, define the p-norm:

$$\|\mathbf{x}\|_p := \left(\sum_{k=1}^n |x_k|^p\right)^{1/p}$$

where x_k is the k-th entry of $\mathbf{x}$.
For $p = \infty$ we define

$$\|\mathbf{x}\|_\infty := \max_k |x_k|$$

Theorem (p-norm) $\|\cdot\|_p$ is a norm for $1 \le p \le \infty$.

Proof

We will only prove the case $p = 1, 2, \infty$ as general p is more involved.

Homogeneity and positive-definiteness are straightforward: e.g.,

$$\|c\mathbf{x}\|_p = (\sum_{k=1}^n |cx_k|^p)^{1/p} = (|c|^p \sum_{k=1}^n |x_k|^p)^{1/p} = |c|\|\mathbf{x}\|$$

and if $\|\mathbf{x}\|_p = 0$ then all $|x_k|^p$ are have to be zero.

For $p = 1, \infty$ the triangle inequality is also straightforward:

$$\|\mathbf{x}+\mathbf{y}\|_\infty = \max_k(|x_k + y_k|) \le \max_k(|x_k| + |y_k|) \le \|\mathbf{x}\|_\infty + \|\mathbf{y}\|_\infty$$

and

$$\|\mathbf{x}+\mathbf{y}\|_1 = \sum_{k=1}^n |x_k + y_k| \le \sum_{k=1}^n (|x_k| + |y_k|) = \|\mathbf{x}\|_1 + \|\mathbf{y}\|_1$$

For $p = 2$ it can be proved using the Cauchy–Schwartz inequality:

$$|\mathbf{x}^\top \mathbf{y}| \le \|\mathbf{x}\|_2 \|\mathbf{y}\|_2$$

That is, we have

$$\|\mathbf{x}+\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + 2\mathbf{x}^\top\mathbf{y} + \|\mathbf{y}\|^2 \le \|\mathbf{x}\|^2 + 2\|\mathbf{x}\|\|\mathbf{y}\| + \|\mathbf{y}\|^2 = (\|\mathbf{x}\| + \|\mathbf{y}\|)$$

■

In Julia, one can use the inbuilt `norm` function to calculate norms:

```
norm([1,-2,3]) == norm([1,-2,3], 2) == sqrt(1^2 + 2^2 + 3^2);
norm([1,-2,3], 1) == sqrt(1 + 2 + 3);
norm([1,-2,3], Inf) == 3;
```

2. Matrix norms

Just like vectors, matrices have norms that measure their "length". The simplest example is the Fröbenius norm, defined for an $m \times n$ real matrix A as

$$\|A\|_F := \sqrt{\sum_{k=1}^m \sum_{j=1}^n A_{kj}^2}$$

This is available as `norm` in Julia:

```
A = randn(5,3)
norm(A) == norm(vec(A))
```

While this is the simplest norm, it is not the most useful. Instead, we will build a matrix norm from a vector norm:

Definition (matrix-norm) Suppose $A \in \mathbb{R}^{m\times n}$ and consider two norms $\|\cdot\|_X$ on $\mathbb{R}^n$ and $\|\cdot\|_Y$ on $\mathbb{R}^n$. Define the *(induced) matrix norm* as:

$$\|A\|_{X\to Y} := \sup_{\mathbf{v}:\|\mathbf{v}\|_X=1} \|A\mathbf{v}\|_Y$$

Also define

$$\|A\|_X \triangleq \|A\|_{X\to X}$$

For the induced 2, 1, and ∞-norm we use

$$\|A\|_2, \|A\|_1 \qquad \text{and} \qquad \|A\|_\infty.$$

Note an equivalent definition of the induced norm:

$$\|A\|_{X\to Y} = \sup_{\mathbf{x}\in\mathbb{R}^n, \mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_Y}{\|\mathbf{x}\|_X}$$

This follows since we can scale $\mathbf{x}$ by its norm so that it has unit norm, that is, $\frac{\mathbf{x}}{\|\mathbf{x}\|_X}$ has unit norm.

Lemma (matrix norms are norms) Induced matrix norms are norms, that is for $\|\cdot\| = \|\cdot\|_{X\to Y}$ we have:

1. Triangle inequality: $\|A + B\| \leq \|A\| + \|B\|$
2. Homogeneneity: $\|cA\| = |c|\|A\|$
3. Positive-definiteness: $\|A\| = 0 \Rightarrow A = 0$

In addition, they satisfy the following additional properties:

1. $\|A\mathbf{x}\|_Y \leq \|A\|_{X\to Y}\|\mathbf{x}\|_X$
2. Multiplicative inequality: $\|AB\|_{X\to Z} \leq \|A\|_{Y\to Z}\|B\|_{X\to Y}$

Proof

First we show the *triangle inequality*:

$$\|A + B\| \leq \sup_{\mathbf{v}:\|\mathbf{v}\|_X=1} (\|A\mathbf{v}\|_Y + \|B\mathbf{v}\|_Y) \leq \|A\| + \|B\|.$$

Homogeneity is also immediate. Positive-definiteness follows from the fact that if $\|A\| = 0$ then $A\mathbf{x} = 0$ for all $\mathbf{x} \in \mathbb{R}^n$.
The property $\|A\mathbf{x}\|_Y \leq \|A\|_{X\to Y}\|\mathbf{x}\|_X$ follows from the definition.
Finally, the multiplicative inequality follows from

$$\|AB\| = \sup_{\mathbf{v}:\|\mathbf{v}\|_X=1} \|AB\mathbf{v}|_Z \leq \sup_{\mathbf{v}:\|\mathbf{v}\|_X=1} \|A\|_{Y\to Z}\|B\mathbf{v}| = \|A\|_{Y\to Z}\|B\|_{X\to Y}$$

∎

We have some simple examples of induced norms:

Example (1-norm) We claim

$$\|A\|_1 = \max_j \|\mathbf{a}_j\|_1$$

that is, the maximum 1-norm of the columns. To see this use the triangle inequality to find for $\|\mathbf{x}\|_1 = 1$

$$\|A\mathbf{x}\|_1 \leq \sum_{j=1}^n |x_j| \|\mathbf{a}_j\|_1 \leq \max_j \|\mathbf{a}_j\| \sum_{j=1}^n |x_j| = \max_j \|\mathbf{a}_j\|_1.$$

But the bound is also attained since if j is the column that maximises the norms then

$$\|A\mathbf{e}_j\|_1 = \|\mathbf{a}_j\|_1 = \max_j \|\mathbf{a}_j\|_1.$$

In the problem sheet we see that

$$\|A\|_\infty = \max_k \|A[k,:]\|_1$$

that is, the maximum 1-norm of the rows.

Matrix norms are available via `opnorm`:

```
m,n = 5,3
A = randn(m,n)
opnorm(A,1) == maximum(norm(A[:,j],1) for j = 1:n)
opnorm(A,Inf) == maximum(norm(A[k,:],1) for k = 1:m)
opnorm(A) # the 2-norm
```

An example that does not have a simple formula is $\|A\|_2$, but we do have two simple cases:

Proposition (diagonal/orthogonal 2-norms) If Λ is diagonal with entries λ_k then $\|\Lambda\|_2 = \max_k |\lambda_k|$.. If Q is orthogonal then $\|Q\| = 1$.

3. Singular value decomposition

To define the induced 2-norm we need to consider the following:

Definition (singular value decomposition) For $A \in \mathbb{R}^{m\times n}$ with rank $r > 0$, the *reduced singular value decomposition (SVD)* is

$$A = U\Sigma V^\top$$

where $U \in \mathbb{R}^{m\times r}$ and $V \in \mathbb{R}^{r\times n}$ have orthonormal columns and $\Sigma \in \mathbb{R}^{r\times r}$ is diagonal whose diagonal entries, which which we call *singular values*, are all positive and decreasing: $\sigma_1 \geq \cdots \geq \sigma_r > 0$. The *full singular value decomposition (SVD)* is

$$A = U\Sigma V^\top$$

where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal matrices and $\Sigma \in \mathbb{R}^{m\times n}$ has only diagonal entries, i.e., if $m > n$,

$$\Sigma = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \\ & & 0 \\ & & \vdots \\ & & 0 \end{bmatrix}$$

and if $m < n$,

$$\Sigma = \begin{bmatrix} \sigma_1 & & & & & \\ & \ddots & & & & \\ & & \sigma_m & 0 & \cdots & 0 \end{bmatrix}$$

where $\sigma_k = 0$ if $k > r$.

To show the SVD exists we first establish some properties of a *Gram matrix* ($A^\top A$):

Proposition (Gram matrix kernel) The kernel of A is the also the kernel of $A^\top A$.

Proof
If $A^\top A\mathbf{x} = 0$ then we have

$$0 = \mathbf{x}^T A^\top A\mathbf{x} = \|A\mathbf{x}\|^2$$

which means $A\mathbf{x} = 0$ and $\mathbf{x} \in \ker(A)$.
∎

Proposition (Gram matrix diagonalisation) The Gram-matrix satisfies

$$A^\top A = Q\Lambda Q^\top$$

where Q is orthogonal and the eigenvalues λ_k are non-negative.

Proof
$A^\top A$ is symmetric so we appeal to the spectral theorem for the existence of the decomposition.
For the corresponding (orthonormal) eigenvector $\mathbf{q}_k$,

$$\lambda_k = \lambda_k \mathbf{q}_k^\top \mathbf{q}_k = \mathbf{q}_k^\top A^\top A\mathbf{q}_k = \|A\mathbf{q}_k\|^2 \geq 0.$$

∎

This connection allows us to prove existence:

Theorem (SVD existence) Every $A \in \mathbb{R}^{m\times n}$ has an SVD.

Proof
Consider

$$A^\top A = Q\Lambda Q^\top.$$

Assume (as usual) that the eigenvalues are sorted in decreasing modulus, and so $\lambda_1, \dots, \lambda_r$ are an enumeration of the non-zero eigenvalues and

$$V := [\mathbf{q}_1|\cdots|\mathbf{q}_r]$$

the corresponding (orthonormal) eigenvectors, with

$$K = [\mathbf{q}_{r+1}|\cdots|\mathbf{q}_n]$$

the corresponding kernel.
Define

$$\Sigma := \begin{bmatrix} \sqrt{\lambda_1} & & \\ & \ddots & \\ & & \sqrt{\lambda_r} \end{bmatrix}$$

Now define

$$U := AV\Sigma^{-1}$$

which is orthogonal since $A^\top AV = V\Sigma^2$:

$$U^\top U = \Sigma^{-1}V^\top A^\top AV\Sigma^{-1} = I.$$

Thus we have

$$U\Sigma V^\top = AVV^\top = A\underbrace{[V|K]}_{Q}\underbrace{\begin{bmatrix} V^\top \\ K^\top \end{bmatrix}}_{Q^\top}$$

where we use the fact that $AK = 0$ so that concatenating K does not change the value.

∎

Singular values tell us the 2-norm:

Corollary (singular values and norm)

$$\|A\|_2 = \sigma_1$$

and if $A \in \mathbb{R}^{n\times n}$ is invertible, then

$$\|A^{-1}\|_2 = \sigma_n^{-1}$$

Proof

First we establish the upper-bound:

$$\|A\|_2 \le \|U\|_2\|\Sigma\|_2\|V^\top\|_2 = \|\Sigma\|_2 = \sigma_1$$

This is attained using the first right singular vector:

$$\|A\mathbf{v}_1\|_2 = \|\Sigma V^\top \mathbf{v}_1\|_2 = \|\Sigma \mathbf{e}_1\|_2 = \sigma_1$$

The inverse result follows since the inverse has SVD

$$A^{-1} = V\Sigma^{-1}U^\top = V(W\Sigma^{-1}W)U^\top$$

is the SVD of A^{-1}, where

$$W := P_\sigma = \begin{bmatrix} & & 1 \\ & \iddots & \\ 1 & & \end{bmatrix}$$

is the permutation that reverses the entries, that is, σ has Cauchy notation

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ n & n-1 & \cdots & 1 \end{pmatrix}.$$

∎

We will not discuss in this module computation of singular value decompositions or eigenvalues:
they involve iterative algorithms (actually built on a sequence of QR decompositions).

One of the main usages for SVDs is low-rank approximation:

Theorem (best low rank approximation) The matrix

$$A_k := [\mathbf{u}_1|\cdots|\mathbf{u}_k]\begin{bmatrix}\sigma_1 & & \\ & \ddots & \\ & & \sigma_k\end{bmatrix}[\mathbf{v}_1|\cdots|\mathbf{v}_k]^\top$$

is the best 2-norm approximation of A by a rank k matrix, that is, for all rank-k matrices B, we have $\|A - A_k\|_2 \le \|A - B\|_2$.

Proof

We have

$$A - A_k = U\begin{bmatrix}0 & & & & & \\ & \ddots & & & & \\ & & 0 & & & \\ & & & \sigma_{k+1} & & \\ & & & & \ddots & \\ & & & & & \sigma_r\end{bmatrix}V^\top.$$

Suppose a rank-k matrix B has

$$\|A - B\|_2 < \|A - A_k\|_2 = \sigma_{k+1}.$$

For all $\mathbf{w} \in \ker(B)$ we have

$$\|A\mathbf{w}\|_2 = \|(A - B)\mathbf{w}\|_2 \le \|A - B\|\|\mathbf{w}\|_2 < \sigma_{k+1}\|\mathbf{w}\|_2$$

But for all $\mathbf{u} \in \operatorname{span}(\mathbf{v}_1, \dots, \mathbf{v}_{k+1})$, that is, $\mathbf{u} = V[:, 1:k+1]\mathbf{c}$ for some $\mathbf{c} \in \mathbb{R}^{k+1}$ we have

$$\|A\mathbf{u}\|_2^2 = \|U\Sigma_k\mathbf{c}\|_2^2 = \|\Sigma_k\mathbf{c}\|_2^2 = \sum_{j=1}^{k+1}(\sigma_j c_j)^2 \ge \sigma_{k+1}^2\|c\|^2,$$

i.e., $\|A\mathbf{u}\|_2 \ge \sigma_{k+1}\|c\|$. Thus $\mathbf{w}$ cannot be in this span.

The dimension of the span of $\ker(B)$ is at least $n - k$, but the dimension of $\operatorname{span}(\mathbf{v}_1, \dots, \mathbf{v}_{k+1})$ is at least $k + 1$. Since these two spaces cannot intersect we have a contradiction, since $(n - r) + (r + 1) = n + 1 > n$. ∎

Here we show an example of a simple low-rank approximation using the SVD. Consider the Hilbert matrix:

```
function hilbertmatrix(n)
    ret = zeros(n,n)
    for j = 1:n, k=1:n
        ret[k,j] = 1/(k+j-1)
    end
    ret
end
hilbertmatrix(5)
```

That is, the $H[k,j] = 1/(k+j-1)$. This is a famous example of matrix with rapidly decreasing singular values:

```
H = hilbertmatrix(100)
U,σ,V = svd(H)
scatter(σ; yscale=:log10)
```

Note numerically we typically do not get a exactly zero singular values so the rank is always treated as $\min(m,n)$.
Because the singular values decay rapidly
we can approximate the matrix very well with a rank 20 matrix:

```
k = 20 # rank
Σ_k = Diagonal(σ[1:k])
U_k = U[:,1:k]
V_k = V[:,1:k]
norm(U_k * Σ_k * V_k' - H)
```

Note that this can be viewed as a *compression* algorithm: we have replaced a matrix with $100^2 = 10,000$ entries by two matrices and a vector with $4,000$ entries without losing any information.
In the problem sheet we explore the usage of low rank approximation to smooth functions.

4. Condition numbers

We have seen that floating point arithmetic induces errors in computations, and that we can typically bound the absolute errors to be proportional to $C\epsilon_{\rm m}$. We want a way to bound the effect of more complicated calculations like computing $A\mathbf{x}$ or $A^{-1}\mathbf{y}$ without having to deal with the exact nature of floating point arithmetic. Here we consider only matrix-multiplication but will make a remark about matrix inversion.

To justify what follows, we first observe that errors in implementing matrix-vector multiplication can be captured by considering the multiplication to be exact on the wrong matrix: that is, `A*x` (implemented with floating point) is precisely $A + \delta A$ where δA has small norm, relative to A.
This is known as *backward error analysis*.

To discuss floating point errors we need to be precise which order the operations happened.
We will use the definition `mul(A,x)`, which denote $\mathrm{mul}(A, \mathbf{x})$. (Note that `mul_rows` actually does the *exact* same operations, just in a different order.) Note that each entry of the result is in fact a dot-product of the corresponding rows so we first consider the error in the dot product `dot(x,y)` as implemented in floating-point, which we denote $\mathrm{dot}(A, x)$.

We first need a helper proposition:

Proposition If $|\epsilon_i| \leq \epsilon$ and $n\epsilon < 1$, then

$$\prod_{k=1}^{n}(1+\epsilon_i) = 1+\theta_n$$

for some constant θ_n satisfying $|\theta_n| \leq \frac{n\epsilon}{1-n\epsilon}$.

The proof is left as an exercise (Hint: use induction).

Lemma (dot product backward error)
For $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$,

$$\mathrm{dot}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} + \delta\mathbf{x})^\top \mathbf{y}$$

where

$$|\delta\mathbf{x}| \leq \frac{n\epsilon_{\mathrm{m}}}{2-n\epsilon_{\mathrm{m}}}|\mathbf{x}|,$$

where $|\mathbf{x}|$ means absolute-value of each entry.

Proof

Note that

$$\begin{aligned}
\mathrm{dot}(\mathbf{x}, \mathbf{y}) &= \{[(x_1 \otimes y_1) \oplus (x_2 \otimes y_2)] \oplus (x_3 \otimes y_3)] \oplus \cdots\} \oplus (x_n \otimes y_n) \\
&= \{[(x_1y_1)(1+\delta_1) + (x_2y_2)(1+\delta_2)](1+\gamma_2) \\
&+ x_3y_3(1+\delta_3)](1+\gamma_3) + \cdots + x_ny_n(1+\delta_n)\}(1+\gamma_n) \\
&= \sum_{j=1}^{n} x_jy_j(1+\delta_j)\prod_{k=j}^{n}(1+\gamma_k) \\
&= \sum_{j=1}^{n} x_jy_j(1+\theta_j)
\end{aligned}$$

where we denote the errors from multiplication as δ_k and those from addition by γ_k
(with $\gamma_1 = 0$). Note that θ_j each have at most n terms each bounded by $\epsilon_{\mathrm{m}}/2$,
Thus the previous proposition tells us

$$|\theta_j| \leq \frac{n\epsilon_{\mathrm{m}}}{2-n\epsilon_{\mathrm{m}}}.$$

Thus

$$\delta \mathbf{x} = \begin{pmatrix} x_1\theta_n^1 \\ x_2\theta_n^2 \\ x_3\theta_{n-1} \\ \vdots \\ x_n\theta_1 \end{pmatrix}$$

and the theorem follows from homogeneity:

$$\|\delta \mathbf{x}\| \le \frac{n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|\mathbf{x}\|$$

∎

Theorem (matrix-vector backward error)
For $A \in \mathbb{R}^{m\times n}$ and $\mathbf{x} \in \mathbb{R}^n$ we have

$$\mathrm{mul}(A, \mathbf{x}) = (A + \delta A)\mathbf{x}$$

where

$$|\delta A| \le \frac{n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} |A|.$$

Therefore

$$\begin{aligned} \|\delta A\|_1 &\le \frac{n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|A\|_1 \\ \|\delta A\|_2 &\le \frac{\sqrt{\min(m,n)}n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|A\|_2 \\ \|\delta A\|_\infty &\le \frac{n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|A\|_\infty \end{aligned}$$

Proof
The bound on $|\delta A|$ is implied by the previous lemma.
The 1 and ∞-norm follow since

$$\|A\|_1 = \||A|\|_1 \text{ and } \|A\|_\infty = \||A|\|_\infty$$

This leaves the 2-norm example, which is a bit more challenging as there are matrices A such that $\|A\|_2 \neq \||A|\|_2$.
Instead we will prove the result by going through the Fröbenius norm and using:

$$\|A\|_2 \le \|A\|_F \le \sqrt{r}\|A\|_2$$

where r is rank of A (see PS5)
and $\|A\|_F = \||A|\|_F$,
so we deduce:

$$\begin{aligned}\|\delta A\|_2 \le \|\delta A\| F = \||\delta A|\| F \le \frac{n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \||A|\|_F \\ = \frac{n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|A\|_F \le \frac{\sqrt{r} n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|A\|_2 \\ \le \frac{\sqrt{\min(m,n)} n\epsilon_{\mathrm{m}}}{2 - n\epsilon_{\mathrm{m}}} \|A\|_2\end{aligned}$$

∎

So now we get to a mathematical question independent of floating point:
can we bound the *relative error* in approximating

$$A\mathbf{x} \approx (A + \delta A)\mathbf{x}$$

if we know a bound on $\|\delta A\|$?
It turns out we can in turns of the *condition number* of the matrix:

Definition (condition number)
For a square matrix A, the *condition number* (in p-norm) is

$$\kappa_p(A) := \|A\|_p \|A^{-1}\|_p$$

with the 2-norm:

$$\kappa_2(A) = \frac{\sigma_1}{\sigma_n}.$$

Theorem (relative-error for matrix-vector)
The *worst-case* relative error in $A\mathbf{x} \approx (A + \delta A)\mathbf{x}$ is

$$\frac{\|\delta A\mathbf{x}\|}{\|A\mathbf{x}\|} \le \kappa(A)\varepsilon$$

if we have the relative pertubation error $\|\delta A\| = \|A\|\varepsilon$.

Proof
We can assume A is invertible (as otherwise $\kappa(A) = \infty$). Denote $\mathbf{y} = A\mathbf{x}$ and we have

$$\frac{\|\mathbf{x}\|}{\|A\mathbf{x}\|} = \frac{\|A^{-1}\mathbf{y}\|}{\|\mathbf{y}\|} \le \|A^{-1}\|$$

Thus we have:

$$\frac{\|\delta A\mathbf{x}\|}{\|A\mathbf{x}\|} \leq \|\delta A\|\|A^{-1}\| \leq \kappa(A)\frac{\|\delta A\|}{\|A\|}$$

∎

Thus for floating point arithmetic we know the error is bounded by $\kappa(A)\frac{n\epsilon_{\mathrm{m}}}{2-n\epsilon_{\mathrm{m}}}$.

If one uses QR to solve $A\mathbf{x} = \mathbf{y}$ the condition number also gives a meaningful bound on the error.
As we have already noted, there are some matrices where PLU decompositions introduce large errors, so in that case well-conditioning is not a guarantee (but it still usually works).

Week 6: Differential Equations via Finite Differences

> YouTube Lectures:
> Condition Numbers
> Indefinite integration via Finite Differences
> Euler Methods
> Poisson Equation
> Convergence of Finite Differences

We now see our first application: solving differential equations.
We will focus on the following differential equations:

1. Indefinite integration for $a \leq x \leq b$:

$$u(a) = c, u' = f(x)$$

2. Linear time-evolution problems for $0 \leq t \leq T$:

$$u(0) = c, u' - a(t)u = f(t)$$

4. Vector time-evolution problems for $0 \leq t \leq T$:

$$\mathbf{u}(0) = \mathbf{c}, \mathbf{u}' - A(t)\mathbf{u} = \mathbf{f}(t)$$

4. Nonlinear time-evolution problems for $0 \leq t \leq T$:

$$\mathbf{u}(0) = \mathbf{c}, \mathbf{u}' = f(t, \mathbf{u}(t))$$

5. Poisson equation with Dirichlet conditions for $a \leq x \leq b$:

$$u(a) = c_0, u(b) = c_1,$$
$$u'' = f(x)$$

Our approach to solving these is to

1. Approximate the solution on $[a, b]$ evaluated on a n-point grid $x_1, \ldots, x_n$ (labelled t_k for time-evolution problems) with *step size*

$$x_{k+1} - x_k = h = \frac{b-a}{n-1}$$

for a vector $\mathbf{u} \in \mathbb{R}^n$
that will be determined by solving a linear system:

$$\begin{bmatrix} u(x_1) \\ \vdots \\ u(x_n) \end{bmatrix} \approx \underbrace{\begin{bmatrix} u_1 \\ \vdots \\ u_n \end{bmatrix}}_{\mathbf{u}}$$

2. Replace the derivatives with the finite-difference approximations (here $m_k = (x_{k+1} - x_k)/2$ is the mid-point of the grid):

$$\begin{aligned}
u'(x_k) &\approx \frac{u(x_{k+1}) - u(x_k)}{h} \approx \frac{u_{k+1} - u_k}{h} && \text{(Forward-difference)} \\
u'(m_k) &\approx \frac{u(x_{k+1}) - u(x_k)}{h} \approx \frac{u_{k+1} - u_k}{h} && \text{(Central-difference)} \\
u'(x_k) &\approx \frac{u(x_k) - u(x_{k-1})}{h} \approx \frac{u_k - u_{k-1}}{h} && \text{(Backward-difference)} \\
u''(x_k) &\approx \frac{u(x_{k+1}) - 2u(x_k) + u_{k-1}}{h^2} \approx \frac{u_{k+1} - 2u_k + u_{k-1}}{h^2}
\end{aligned}$$

3. Recast the differential equation as a linear system whose solution is $\mathbf{u}$, which we solve using numerical linear algebra.
   Add the initial/boundary conditions as extra rows to make sure the system is square.

Remark: (advanced) One should normally not need to implement these methods oneselves as there are packages available, e.g. DifferentialEquations.jl. Moreover Forward and Backward Euler are only the first baby steps to a wide range of time-steppers, with Runge–Kutta being one of the most successful.
For example we can solve
a simple differential equation like a pendulum $u'' = -\sin u$ can be solved as follows (writing at a system $u' = v, v' = -\sin u$):

```
using DifferentialEquations, LinearAlgebra, Plots

u = solve(ODEProblem((u,_,x) -> [u[2], -sin(u[1])], [1,0], (0,10)))
plot(u)
```

However, even in these automated packages one has a choice of different methods with different behaviour, so it is important to understand what is happening.

1. Time-evolution problems

In this section we consider the forward and backward Euler methods, which are based on forward and backward difference approximations to the derivative. In the problem sheet we will investigate a rule that that takes the average of the two (with significant benefits). We first discuss the simplest case of indefinite integration, where central differences is also applicable, then introduce forward and backward
Euler for linear scalar, linear vector, and nonlinear differential equations.

Indefinite integration

We begin with the simplest differential equation on an interval $[a, b]$:

$$\begin{aligned} u(a) &= c \\ u'(x) &= f(x) \end{aligned}$$

Using the forward-difference (which is the standard finite-difference) approximation we choose $u_k \approx u(x_k)$ so that, for $k = 1, \ldots, n-1$:

$$f(x_k) = u'(x_k) \approx \frac{u_{k+1} - u_k}{h} = f(x_k)$$

That is, where u satisfies the differential equation exactly,
u_k satisfies the *difference equation* exactly.
We do not include $k = n$ to avoid going outside our grid.

This condition can be recast as a linear system:

$$\underbrace{\frac{1}{h}\begin{bmatrix} -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{bmatrix}}_{D_h} \mathbf{u}^{\mathrm{f}} = \underbrace{\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix}}_{\mathbf{f}^{\mathrm{f}}}$$

where the super-script f denotes that we are using
forward differences (and is leading towards the
forward Euler method).
Here $D_h \in \mathbb{R}^{n-1,n}$ so this system is not-invertible. Thus we need to add an extra row, coming from the initial condition: $\mathbf{e}_1^\top \mathbf{u}^{\mathrm{f}} = c$,
that is:

$$\begin{bmatrix} \mathbf{e}_1^\top \\ D_h \end{bmatrix} \mathbf{u}^{\mathrm{f}} = \underbrace{\begin{bmatrix} 1 & & & \\ -1/h & 1/h & & \\ & \ddots & \ddots & \\ & & -1/h & 1/h \end{bmatrix}}_{L_h} \mathbf{u}^{\mathrm{f}} = \begin{bmatrix} c \\ \mathbf{f}^{\mathrm{f}} \end{bmatrix}$$

This is a lower-triangular bidiagonal system, so can be solved using forward substitution in $O(n)$ operations.

We can also consider discretisation at the mid-point $m_k = \frac{x_{k+1}-x_k}{2}$, which is the analogue of using central-differences:

$$u'(m_k) \approx \frac{u_{k+1} - u_k}{h} = f(m_k)$$

That is, we have the exact same system with a different right-hand side:

$$\underbrace{\frac{1}{h}\begin{bmatrix} -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{bmatrix}}_{D_h} \mathbf{u}^{\mathrm{m}} = \underbrace{\begin{bmatrix} f(m_1) \\ \vdots \\ f(m_{n-1}) \end{bmatrix}}_{\mathbf{f}^{\mathrm{m}}}$$

And of course there is $\mathbf{u}^{\mathrm{B}}$ coming from the backwards-difference formula:

$$u'(x_k) \approx \frac{u_k - u_{k-1}}{h} = f(x_k)$$

which we leave as an exercise.

Example

Let's do an example of integrating $\cos x$, and see if our method matches the true answer of $\sin x$. First we construct the system as a lower-triangular, `Bidiagonal` matrix:

```
using LinearAlgebra, Plots

function indefint(x)
    h = step(x) # x[k+1]-x[k]
    n = length(x)
    L = Bidiagonal([1; fill(1/h, n-1)], fill(-1/h, n-1), :L)
end

n = 10
x = range(0, 1; length=n)
L = indefint(x)
```

We can now solve for our particular problem using both the left and mid-point rules:

```
c = 0 # u(0) = 0
f = x -> cos(x)


m = (x[1:end-1] + x[2:end])/2 # midpoints
```

```
f^f = f.(x[1:end-1]) # evaluate f at all but last points
f^m = f.(m)          # evaluate f at mid-points
u^f = L \ [c; f^f] # integrate using forward-differences
u^m = L \ [c; f^m] # integrate using central-differences

plot(x, sin.(x); label="sin(x)", legend=:bottomright)
scatter!(x, u^f; label="forward")
scatter!(x, u^m; label="mid")
```

They both are close though the mid-point version is significantly more accurate.
We can estimate how fast it converges:

```
## Error from indefinite integration with c and f
function forward_err(u, c, f, n)
    x = range(0, 1; length = n)
    u^f = indefint(x) \ [c; f.(x[1:end-1])]
    norm(u^f - u.(x), Inf)
end

function mid_err(u, c, f, n)
    x = range(0, 1; length = n)
    m = (x[1:end-1] + x[2:end]) / 2 # midpoints
    u^m = indefint(x) \ [c; f.(m)]
    norm(u^m - u.(x), Inf)
end

ns = 10 .^ (1:8) # solve up to n = 10 million
scatter(ns, forward_err.(sin, 0, f, ns); xscale=:log10, yscale=:log10, label="forward")
scatter!(ns, mid_err.(sin, 0, f, ns); label="mid")
plot!(ns, ns .^ (-1); label="1/n")
plot!(ns, ns .^ (-2); label="1/n^2")
```

This is a log-log plot:we scale both x and y axes logarithmically so that n^α becomes a straight line where the slope is dictated by α.
We seem experimentally that the error for forward-difference is $O(n^{-1})$ while for mid-point/central-differences we get faster $O(n^{-2})$ convergence.
Both methods appear to be stable.

Forward Euler

Now consider a scalar linear time-evolution problem for $0 \leq t \leq T$:

$$\begin{aligned} u(0) &= c \\ u'(t) - a(t)u(t) &= f(t) \end{aligned}$$

Label the n-point grid as $t_k = (k-1)h$ for $h = T/(n-1)$.

Definition (Restriction matrices)
Define the $n-1 \times n$ *restriction matrices* as

$$I_n^{\mathrm{f}} := \begin{bmatrix} 1 & & & \\ & \ddots & & \\ & & 1 & 0 \end{bmatrix}$$
$$I_n^{\mathrm{b}} := \begin{bmatrix} 0 & 1 & & \\ & \ddots & & \\ & & & 1 \end{bmatrix}$$

Again we can replace the discretisation using finite-differences, giving us

$$\frac{u_{k+1} - u_k}{h} - a(t_k)u_k = f(u_k)$$

for $k = 1, \dots, n-1$. We need to add the term $a(t_k)u_k$ to our differential equation,
that is. We do this using the $n-1 \times n$ *(left) restriction matrix* that takes a vector
evaluated at $x_1, \dots, x_n$ and restricts it to $x_1, \dots, x_{n-1}$,
as well as the $n \times n$ *multiplication matrix*

$$A_n = \begin{bmatrix} a(t_1) & & \\ & \ddots & \\ & & a(t_n) \end{bmatrix}$$

Putting everything together we have the system:

$$\begin{bmatrix} \mathbf{e}_1^\top \\ D_h - I_n^{\mathrm{f}} A_n \end{bmatrix} \mathbf{u}^{\mathrm{f}} = \underbrace{\begin{bmatrix} 1 & & & \\ -a(t_1) - 1/h & 1/h & & \\ & \ddots & \ddots & \\ & & -a(t_{n-1}) - 1/h & 1/h \end{bmatrix}}_{L} \mathbf{u}^{\mathrm{f}} = \begin{bmatrix} c \\ I_n^{\mathrm{f}} \mathbf{f} \end{bmatrix}$$

where $\mathbf{f} = \begin{bmatrix} f(t_1) \\ \vdots \\ f(t_n) \end{bmatrix}$.

Here is a simple example for solving:

$$u'(0) = 1, u' + tu = \mathrm{e}^t$$

which has an exact solution in terms of a special error function
(which we determined using Mathematica).

```
using SpecialFunctions
c = 1
a = t -> t
n = 2000
t = range(0, 1; length=n)
## exact solution, found in Mathematica
u = t -> -(1/2)*exp(-(1+t^2)/2)*(-2sqrt(e) + sqrt(2π)erfi(1/sqrt(2)) - sqrt(2π)erfi((1 +
t)/sqrt(2)))

h = step(t)
L = Bidiagonal([1; fill(1/h, n-1)], a.(t[1:end-1]) .- 1/h, :L)

norm(L \ [c; exp.(t[1:end-1])] - u.(t),Inf)
```

We see that it is converging to the true result.

Note that this is a simple forward-substitution of a bidiagonal system,
so we can also just construct it directly:

$$\begin{aligned} u_1 &= c \\ u_{k+1} &= (1 + ha(t_k))u_k + hf(t_k) \end{aligned}$$

Remark: (advanced) Note this can alternatively be reduced to an integral

$$u(t) = c\mathrm{e}^{at} + \mathrm{e}^{at}\int_0^t f(\tau)\mathrm{e}^{-a\tau}\mathrm{d}\tau$$

and solved as above but this approach is harder to generalise.

Backward Euler

In Backward Euler we replace the forward-difference with a backward-difference,
that is

$$\frac{u_k - u_{k-1}}{h} - a(t_k)u_k = f(u_k)$$

This leads to the system:

$$\begin{bmatrix} \mathbf{e}_1^\top \\ D_h - I_n^{\mathrm{b}} A_n \end{bmatrix} \mathbf{u}^{\mathrm{b}} = \underbrace{\begin{bmatrix} 1 & & & \\ -1/h & 1/h - a(t_2) & & \\ & \ddots & \ddots & \\ & & -1/h & 1/h - a(t_n) \end{bmatrix}}_{L} \mathbf{u}^{\mathrm{b}} = \begin{bmatrix} c \\ I_n^{\mathrm{b}} \mathbf{f} \end{bmatrix}$$

Again this is a bidiagonal forward-substitution:

$$
\begin{aligned}
u_1 &= c \\
(1 - ha(t_{k+1}))u_{k+1} &= u_k + hf(t_{k+1})
\end{aligned}
$$

That is,

$$
u_{k+1} = (1 - ha(t_{k+1}))^{-1}(u_k + hf(t_{k+1}))
$$

Systems of equations

We can also solve systems, that is, equations of the form:

$$
\begin{aligned}
\mathbf{u}(0) &= \mathbf{c} \\
\mathbf{u}'(t) - A(t)\mathbf{u}(t) &= \mathbf{f}(t)
\end{aligned}
$$

where $\mathbf{u}, \mathbf{f} : [0, T] \to \mathbb{R}^d$ and $A : [0, T] \to \mathbb{R}^{d \times d}$.
We again discretise at the grid t_k
by approximating $\mathbf{u}(t_k) \approx \mathbf{u}_k \in \mathbb{R}^d$.
This can be reduced to a block-bidiagonal system as in
the scalar case which is solved via forward-substitution. Though
it's easier to think of it directly.

Forward Euler gives us:

$$
\begin{aligned}
\mathbf{u}_1 &= c \\
\mathbf{u}_{k+1} &= \mathbf{u}_k + hA(t_k)\mathbf{u}_k + h\mathbf{f}(t_k)
\end{aligned}
$$

That is, each *time-step* consists of matrix-vector multiplication.
On the other hand Backward Euler requires inverting a matrix
at each time-step:

$$
\begin{aligned}
\mathbf{u}_1 &= c \\
\mathbf{u}_{k+1} &= (I - hA(t_{k+1}))^{-1}(\mathbf{u}_k + h\mathbf{f}(t_{k+1}))
\end{aligned}
$$

Example (Airy equation)
Consider the (negative-time) Airy equation:

$$
\begin{aligned}
u(0) &= 1 \\
u'(0) &= 0 \\
u''(t) + tu &= 0
\end{aligned}
$$

We can recast it as a system by defining

$$
\mathbf{u}(x) = \begin{bmatrix} u(x) \\ u'(x) \end{bmatrix}
$$

which satisfies

$$\begin{aligned}\mathbf{u}(0) &= \begin{bmatrix}1\\0\end{bmatrix}\\ \mathbf{u} - \begin{bmatrix}0 & 1\\ -t & 0\end{bmatrix}\mathbf{u} &= \mathbf{0}.\end{aligned}$$

It is natural to represent the *time-slices* $\mathbf{u}_k$ as
columns of a matrix $U = [\mathbf{u}_1|\cdots|\mathbf{u}_n] \in \mathbb{R}^{2\times n}$. Thus we get:

```
n = 100_000
t = range(0, 50; length=n)
A = t -> [0 1; -t 0]
h = step(t)

U = zeros(2, n) # each column is a time-slice
U[:,1] = [1.0,0.0] # initial condition
for k = 1:n-1
    U[:,k+1] = (I + h*A(t[k]))*U[:,k]
end

plot(t, U')
```

We leave implementation of backward Euler as a simple exercise.

Example (Heat on a graph)
Those who took Introduction to Applied Mathematics will recall
heat equation on a graph. Consider a simple graph of m nodes labelled
$1, \ldots, m$ where node k is connected to neighbouring nodes $k-1$ and $k+1$,
whereas node 1 is only connected to node 2 and node m only connected to
$m-1$. The graph Laplacian corresponding to this system is the matrix:

$$\Delta := \begin{bmatrix} -1 & 1 & & & \\ 1 & -2 & \ddots & & \\ & 1 & \ddots & 1 & \\ & & \ddots & -2 & 1 \\ & & & 1 & -1 \end{bmatrix}$$

If we denote the heat at time t at node k as $u_k(t)$,
which we turn into a vector

$$\mathbf{u}(t) = \begin{bmatrix} u_1(t) \\ \vdots \\ u_m(t) \end{bmatrix}$$

We consider the case of a periodic forcing at the middle node $n = \lfloor m/2 \rfloor$.

Heat equation on this lattice is defined as follows:

$$\mathbf{u}' = \Delta \mathbf{u} + \mathbf{e}_{\lfloor m/2 \rfloor} \cos \omega t$$

We can employ forward and backward Euler:

```
n = 1_000 # number of time-steps
t = range(0, 100; length=n)
h = step(t)

m = 50 # number of nodes


Δ = SymTridiagonal([-1; fill(-2.0, m-2); -1], ones(m-1))
ω = 1
f = t -> cos(ω*t) # periodic forcing with period 1

Uᶠ = zeros(m, n) # each column is a time-slice for forward Euler
Uᵇ = zeros(m, n) # each column is a time-slice for backwar Euler

Uᶠ[:,1] = Uᵇ[:,1] = zeros(m) # initial condition



for k = 1:n-1
    Uᶠ[:,k+1] = (I + h*Δ)*Uᶠ[:,k]
    Uᶠ[m÷2,k+1] += h*f(t[k]) # add forcing at e_1
end

e = zeros(m); e[m÷2] = 1;

for k = 1:n-1
    Uᵇ[:,k+1] = (I - h*Δ)\(Uᵇ[:,k] + h*f(t[k+1])e)
end

scatter(Uᶠ[:,end]; label="forward")
scatter!(Uᵇ[:,end]; label="backward")
```

Both match!

Remark: If you change the number of time-steps to be too small, for example `n = 100`, forward Euler blows up while backward Euler does not. This will be discussed in the problem sheet.

Remark: (advanced) Memory allocations are very expensive so in practice one should preallocate and use memory.

Nonlinear problems

Forward-Euler extends naturally to nonlinear equations, including the vector case:

$$\mathbf{u}' = f(t, \mathbf{u}(t))$$

becomes:

$$\mathbf{u}_{k+1} = \mathbf{u}_k + hf(x_k, \mathbf{u}_k)$$

Here we show a simple solution to a nonlinear Pendulum:

$$u'' = \sin u$$

by writing $\mathbf{u} = [u, u']$ we have:

```
n = 1000
u = fill(zeros(2), n)
x = range(0, 20; length=n)
h = step(x) # same as x[k+1]-x[k]

u[1] = [1,0]
for k = 1:n-1
    u[k+1] = u[k] + h * [u[k][2],-sin(u[k][1])]
end

plot(x, first.(u))
```

As we see it correctly predicts the oscillatory behaviour of a pendulum, and matches the simulation using DifferentialEquations.jl above.

2. Two-point boundary value problems

Here we will only consider one discretisation as it is symmetric:

$$u''(x_k) \approx \frac{u_{k-1} - 2u_k + u_{k+1}}{h^2}$$

That is we use the $n-1 \times n+1$ matrix:

$$D_h^2 := \frac{1}{h^2}\begin{bmatrix} 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \end{bmatrix}$$

Example (Poisson) Consider the Poisson equation with Dirichlet conditions:

$$
\begin{aligned}
u(0) &= c_0 \\
u'' &= f(x) \\
u(1) &= c_1
\end{aligned}
$$

which we discretise as

$$
\begin{aligned}
u_0 &= c_0 \\
\frac{u_{k-1} - 2u_k + u_{k+1}}{h^2} &= f(x_k) \\
u_1 &= c_1
\end{aligned}
$$

As a linear system this equation becomes:

$$
\begin{bmatrix} \mathbf{e}_1^\top \\ D_h^2 \\ \mathbf{e}_{n+1}^\top \end{bmatrix} \mathbf{u} = \begin{bmatrix} c_0 \\ f(x_2) \\ \vdots \\ f(x_{n-1}) \\ c_1 \end{bmatrix}
$$

Thus we solve:

```
x = range(0, 1; length = n)
h = step(x)
T = Tridiagonal([fill(1/h^2, n-2); 0], [1; fill(-2/h^2, n-2); 1], [0; fill(1/h^2, n-2)])
u = T \ [1; exp.(x[2:end-1]); 2]
scatter(x, u)
```

We can test convergence on $u(x) = \cos x^2$ which satisfies

$$
\begin{aligned}
u(0) &= 1 \\
u(1) &= \cos 1 \\
u''(x) &= -4x^2 * cos(x^2) - 2sin(x^2)
\end{aligned}
$$

We observe uniform (∞-norm) convergence:

```
function poisson_err(u, c_0, c_1, f, n)
    x = range(0, 1; length = n)
    h = step(x)
    T = Tridiagonal([fill(1/h^2, n-2); 0], [1; fill(-2/h^2, n-2); 1], [0; fill(1/h^2, n-2)])
    uᶠ = T \ [c_0; f.(x[2:end-1]); c_1]
    norm(uᶠ - u.(x), Inf)
end

u = x -> cos(x^2)
f = x -> -4x^2*cos(x^2) - 2sin(x^2)

ns = 10 .^ (1:8) # solve up to n = 10 million
```

```
    scatter(ns, poisson_err.(u, 1, cos(1), f, ns); xscale=:log10, yscale=:log10, label="error")
    plot!(ns, ns .^ (-2); label="1/n^2")
```

3. Convergence

We now study convergence of the approaches for the constant-coefficient case.
We will use *Toeplitz matrices* as a tool to simplify the explanation.

Definition (Toeplitz) A *Toeplitz matrix* has constant diagonals: $T[k,j] = a_{k-j}$.

Proposition (Bidiagonal Toeplitz inverse)
The inverse of a $n \times n$ bidiagonal Toeplitz matrix is:

$$\begin{bmatrix} 1 & & & & \\ -\ell & 1 & & & \\ & -\ell & 1 & & \\ & & \ddots & \ddots & \\ & & & -\ell & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ \ell & 1 & & & \\ \ell^2 & \ell & 1 & & \\ \vdots & \ddots & \ddots & \ddots & \\ \ell^{n-1} & \cdots & \ell^2 & \ell & 1 \end{bmatrix}$$

Theorem (Forward/Backward Euler convergence)
Consider the equation

$$u(0) = c, u'(t) + au(t) = f(t)$$

Denote

$$\mathbf{u} := \begin{bmatrix} u(t_1) \\ \vdots \\ u(t_n) \end{bmatrix}$$

Assume that u is twice-differentiable with uniformly bounded second derivative.
Then the error for forward/backward Euler is

$$\|\mathbf{u}^f - \mathbf{u}\|_\infty, \|\mathbf{u}^b - \mathbf{u}\|_\infty = O(n^{-1})$$

Proof
We prove the error bound for forward Euler as backward Euler is similar. This proof consists of two stages: (1) consistency and (2) stability.

Consistency means our discretisation approximates the true equation, that is:

$$L\mathbf{u} = \begin{bmatrix} c \\ \frac{u(t_2)-u(t_1)}{h} + au(t_1) \\ \vdots \\ \frac{u(t_n)-u(t_{n-1})}{h} + au(t_{n-1}) \end{bmatrix} = \begin{bmatrix} c \\ u'(t_1) + au(t_1) + u''(\tau_1)h \\ \vdots \\ u'(t_{n-1}) + au(t_{n-1}) + u''(\tau_{n-1})h \end{bmatrix} = \begin{bmatrix} c \\ f(t_1) + u''(\tau_1)h \\ \vdots \\ f(t_{n-1}) + u''(\tau_{n-1})h \end{bmatrix} = \begin{bmatrix} c \\ \mathbf{f}^f \end{bmatrix} + \begin{bmatrix} 0 \\ \delta \end{bmatrix}$$

where $t_k \leq \tau_k \leq t_{k+1}$, and uniform boundedness implies that $\|\delta\|_\infty = O(h)$, or in other words $\|\delta\|_1 = O(1)$.

Stability means the inverse does not blow up the error. We need to be a bit careful and first write, for $\ell = 1 + ha$,

$$L = \underbrace{\begin{bmatrix} 1 & & & \\ & h^{-1} & & \\ & & \ddots & \\ & & & h^{-1} \end{bmatrix}}_{D} \underbrace{\begin{bmatrix} 1 & & & \\ -\ell & 1 & & \\ & \ddots & \ddots & \\ & & -\ell & 1 \end{bmatrix}}_{\mathcal{L}}$$

Stability in this case is the statement that

$$\|\mathcal{L}^{-1}\|_{1\to\infty} \leq (1 + |a|/n)^{n-1} = O(1)$$

using the fact (which likely you have seen in first year) that

$$\lim_{n\to\infty} (1 + |a|/n)^{n-1} = \exp|a|.$$

We now combine stability and consistency. have

$$\|\mathbf{u}^f - \mathbf{u}\|_\infty = \|L^{-1}(L\mathbf{u}^f - L\mathbf{u})\|_\infty = \|\mathcal{L}^{-1}D^{-1}\begin{bmatrix} 0 \\ \delta \end{bmatrix}\|_\infty \leq h\|\mathcal{L}^{-1}\|_{1\to\infty}\|\delta\|_1 = O(h).$$

∎

Poisson

For 2D problems we consider Poisson. The first stage is to row-reduce to get a symmetric tridiagonal (pos. def.) matrix:

$$\begin{bmatrix} 1 & & & & & \\ -1/h^2 & 1 & & & & \\ & & 1 & & & \\ & & & \ddots & & \\ & & & & 1 & -1/h^2 \\ & & & & & 1 \end{bmatrix} \begin{bmatrix} 1 & & & & \\ 1/h^2 & -2/h^2 & 1/h^2 & & \\ & \ddots & \ddots & \ddots & \\ & & 1/h^2 & -2/h^2 & 1/h^2 \\ & & & & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ 0 & -2/h^2 & 1/h^2 & & \\ & \ddots & \ddots & \ddots & \\ & & 1/h^2 & -2/h^2 & 0 \\ & & & & 1 \end{bmatrix}.$$

Considering the right-hand side and dropping
the first and last rows our equation becomes:

$$\underbrace{\frac{1}{h^2}\begin{bmatrix} -2 & 1 & & \\ 1 & -2 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & -2 \end{bmatrix}}_{\Delta} \begin{bmatrix} u_2 \\ \vdots \\ u_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} f(x_2) - c_0/h^2 \\ f(x_3) \\ \vdots \\ f(x_{n-2}) \\ f(x_{n-1}) - c_1/h^2 \end{bmatrix}}_{\mathbf{f}_p}$$

Remark: (advanced) You may recognise Δ as a discrete Laplacian corresponding to a graph with
Dirichlet conditions,
as discussed in first year applied mathematics.
Thus one can interpret finite-differences as approximating a continuous differential equation by
a graph. This view-point extends naturally to higher-dimensional equations. In the problem sheet we also
discuss Neumann series.

> **Theorem (Poisson convergence)** Suppose that u is four-times differentiable with uniformly
> bounded fourth derivative. Then the finite difference approximation to Poisson converges like $O(n^2)$.

Proof

For consistency we need the Taylor series error of second-order finite differences. We have

$$\begin{aligned} u(x+h) &= u(x) + hu'(x) + h^2\frac{u''(x)}{2} + h^3\frac{u'''(x)}{6} + h^4\frac{u^{(4)}(t_+)}{24} \\ u(x-h) &= u(x) - hu'(x) + h^2\frac{u''(x)}{2} - h^3\frac{u'''(x)}{6} + h^4\frac{u^{(4)}(t_-)}{24} \end{aligned}$$

where $t_+ \in [x, x+h]$ and $t_- \in [x-h, x]$.
Thus

$$\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} = u''(x) + h^2\frac{u^{(4)}(t_+) + u^{(4)}(t_-)}{24} = u''(x) + O(h^2)$$

(Note this is a slightly stronger result than used in the solution of PS2.)
Thus we have *consistency*:

$$\frac{\Delta}{h^2}\begin{bmatrix} u_2 \\ \vdots \\ u_{n-1} \end{bmatrix} = \mathbf{f}^p + \delta$$

where $\|\delta\|_\infty = O(h^2)$.

Following PS5 we deduce that it has the Cholesky-like decomposition

$$\Delta = -\underbrace{\begin{bmatrix} 1 & -1 & & \\ & \ddots & \ddots & \\ & & 1 & -1 \\ & & & 1 \end{bmatrix}}_{U}\underbrace{\begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{bmatrix}}_{U^\top}$$

We can invert:

$$U^{-1} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ & \ddots & \ddots & \vdots \\ & & 1 & 1 \\ & & & 1 \end{bmatrix}$$

Thus we have *stability*: $\|h^2\Delta^{-1}\|_\infty \le h^2\|U^{-1}\|_\infty\|U^{-\top}\|_\infty = 1$.

Putting everything together we have, for $\mathbf{u} = [u(x_1), \ldots, u(x_n)]^\top$,

$$\|\mathbf{u}^f - \mathbf{u}\|_\infty = \|\Delta^{-1}(\Delta\mathbf{u}^f - \Delta\mathbf{u})\|_\infty \le \|h^2\Delta^{-1}\|_\infty\|\delta\|_\infty = O(n^{-2})$$

∎

What about the observed instability? The condition number of the matrix provides an intuition (though not a proof: condition numbers are only upper bounds!). Here we have

$$\kappa_\infty(\Delta/h^2) = \kappa_\infty(\Delta) = \|\Delta\|_\infty\|\Delta^{-1}\|_\infty \le 2n^2$$

One can show by looking at Δ^{-1} directly that the condition number is indeed growing like n^2.
Thus we *expect* floating-point errors to magnified propotional to n^2 in the linear solve.

Week 7: Fourier series

YouTube Lectures:
Fourier Series
Trapezium Rule and Fourier Coefficients

In Part III, Computing with Functions, we work with approximating functions by expansions in bases: that is, instead of approximating at a grid (as in the Differential Equations chapter), we approximate functions by other, simpler, functions. The most fundamental basis is (complex) Fourier series:

$$f(\theta) = \sum_{k=-\infty}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}$$

where

$$\hat{f}_k := \frac{1}{2\pi} \int_0^{2\pi} f(\theta) \mathrm{e}^{-\mathrm{i}k\theta} \mathrm{d}\theta$$

In numerical analysis we try to build on the analogy with linear algebra as much as possible. Therefore we write this as:

$$f(\theta) = \underbrace{[\cdots | \mathrm{e}^{-2\mathrm{i}\theta} | \mathrm{e}^{-\mathrm{i}\theta} | \underline{1} | \mathrm{e}^{\mathrm{i}\theta} | \mathrm{e}^{2\mathrm{i}\theta} | \cdots]}_{F(\theta)} \underbrace{\begin{bmatrix} \vdots \\ \hat{f}_{-2} \\ \hat{f}_{-1} \\ \underline{\hat{f}_0} \\ \hat{f}_1 \\ \hat{f}_2 \\ \vdots \end{bmatrix}}_{\hat{\mathbf{f}}}$$

where the underline indicates the zero-index location.

More precisely, we are going to build an approximation using n approximate coefficients $\hat{f}_k^n \approx \hat{f}_k$. We separate this into three cases:

1. Odd: If $n = 2m+1$ we approximate

$$\begin{aligned} f(\theta) &\approx \sum_{k=-m}^{m} \hat{f}_k{}^n \mathrm{e}^{\mathrm{i}k\theta} \\ &= \underbrace{[\mathrm{e}^{-\mathrm{i}m\theta} | \cdots | \mathrm{e}^{-2\mathrm{i}\theta} | \mathrm{e}^{-\mathrm{i}\theta} | 1 | \mathrm{e}^{\mathrm{i}\theta} | \mathrm{e}^{2\mathrm{i}\theta} | \cdots | \mathrm{e}^{\mathrm{i}m\theta}]}_{F_{-m:m}(\theta)} \begin{bmatrix} \hat{f}_{-m}^n \\ \vdots \\ \hat{f}_m^n \end{bmatrix} \end{aligned}$$

2. Even: If $n = 2m$ we approximate

$$\begin{aligned} f(\theta) &\approx \sum_{k=-m}^{m-1} \hat{f}_k{}^n \mathrm{e}^{\mathrm{i}k\theta} \\ &= \underbrace{[\mathrm{e}^{-\mathrm{i}m\theta} | \cdots | \mathrm{e}^{-2\mathrm{i}\theta} | \mathrm{e}^{-\mathrm{i}\theta} | 1 | \mathrm{e}^{\mathrm{i}\theta} | \mathrm{e}^{2\mathrm{i}\theta} | \cdots | \mathrm{e}^{\mathrm{i}(m-1)\theta}]}_{F_{-m:m-1}(\theta)} \begin{bmatrix} \hat{f}_{-m}^n \\ \vdots \\ \hat{f}_{m-1}^n \end{bmatrix} \end{aligned}$$

3. Taylor: if we know the negative coefficients vanish ($0 = \hat{f}_{-1} = \hat{f}_{-2} = \cdots$) we approximate

$$\begin{aligned} f(\theta) &\approx \sum_{k=0}^{n-1} \hat{f}_k^n \mathrm{e}^{\mathrm{i}k\theta} \\ &= \underbrace{[1|\mathrm{e}^{\mathrm{i}\theta}|\mathrm{e}^{2\mathrm{i}\theta}|\cdots|\mathrm{e}^{\mathrm{i}(n-1)\theta}]}_{F_{0:n-1}(\theta)} \begin{bmatrix} \hat{f}_0^n \\ \vdots \\ \hat{f}_{n-1}^n \end{bmatrix} \end{aligned}$$

This can be thought of as an approximate Taylor expansion using the change-of-variables $z = \mathrm{e}^{\mathrm{i}\theta}$.

1. Basics of Fourier series

In analysis one typically works with continuous functions and relates results to continuity. In numerical analysis we inheritely have to work with *vectors*, so it is more natural to focus on the case where the *Fourier coefficients* $\hat{f}_k$ are *absolutely convergent*, or in otherwords, the 1-norm of $\mathbf{f}$ is bounded:

$$\|\mathbf{f}\|_1 = \sum_{k=-\infty}^{\infty} |\hat{f}_k| < \infty$$

We first state a basic results (whose proof is beyond the scope of this module):

Theorem (convergence)
If the Fourier coeffients are absolutely convergent then

$$f(\theta) = \sum_{k=-\infty}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta},$$

which converges uniformly.

Remark: (advanced) We also have convergence for the continuous version of the 2-norm,

$$\|f\|_2 := \sqrt{\int_0^{2\pi} |f(\theta)|^2 \mathrm{d}\theta},$$

for any function such that $\|f\|_2 < \infty$, but we won't need that in what follows.

Fortunately, continuity gives us sufficient (though not necessary) conditions for absolute convergence:

Proposition (differentiability and absolutely convergence) If $f : \mathbb{R} \to \mathbb{C}$ and f' are periodic and f'' is uniformly bounded, then its Fourier coefficients satisfy

$$\|\mathbf{f}\|_1 < \infty$$

Proof

Integrate by parts twice using the fact that $f(0) = f(2\pi)$, $f'(0) = f(2\pi)$:

$$\begin{aligned} \mathfrak{f}_k &= \int_0^{2\pi} f(\theta)\mathrm{e}^{-\mathrm{i}k\theta}\mathrm{d}\theta = [f(\theta)\mathrm{e}^{-\mathrm{i}k\theta}]_0^{2\pi} + \frac{1}{\mathrm{i}k}\int_0^{2\pi} f'(\theta)\mathrm{e}^{-\mathrm{i}k\theta}\mathrm{d}\theta \\ &= \frac{1}{\mathrm{i}k}[f'(\theta)\mathrm{e}^{-\mathrm{i}k\theta}]_0^{2\pi} - \frac{1}{k^2}\int_0^{2\pi} f''(\theta)\mathrm{e}^{-\mathrm{i}k\theta}\mathrm{d}\theta \\ &= -\frac{1}{k^2}\int_0^{2\pi} f''(\theta)\mathrm{e}^{-\mathrm{i}k\theta}\mathrm{d}\theta \end{aligned}$$

thus uniform boundedness of f'' guarantees $|\mathfrak{f}_k| \leq M|k|^{-2}$ for some M, and we have

$$\sum_{k=-\infty}^{\infty} |\mathfrak{f}_k| \leq |\mathfrak{f}_0| + 2M\sum_{k=1}^{\infty} |k|^{-2} < \infty.$$

using the dominant convergence test.

∎

This condition can be weakened to Lipschitz continuity but the proof is beyond the scope of this module.
Of more practical importance is the other direction: the more times differentiable a function the faster the coefficients decay, and thence the faster Fourier series converges.
In fact, if a function is smooth and 2π-periodic its Fourier coefficients decay faster than algebraically: they decay like $O(k^{-\lambda})$ for any λ. This will be explored in the problem sheet.

Remark: (advanced) Going further, if we let $z = \mathrm{e}^{\mathrm{i}\theta}$ then if $f(z)$ is *analytic* in a neighbourhood of the unit circle the Fourier coefficients decay *exponentially fast*. And if $f(z)$ is entire they decay even faster than exponentially.

2. Trapezium rule and discrete Fourier coefficients

Let $\theta_j = 2\pi j/n$ for $j = 0, 1, \ldots, n$ denote $n+1$ evenly spaced points over $[0, 2\pi]$.
The *Trapezium rule* over $[0, 2\pi]$ is the approximation:

$$\int_0^{2\pi} f(\theta)\mathrm{d}\theta \approx \frac{2\pi}{n}\left[\frac{f(0)}{2} + \sum_{j=1}^{n-1} f(\theta_j) + \frac{f(2\pi)}{2}\right]$$

But if f is periodic we have $f(0) = f(2\pi)$ we get the *periodic Trapezium rule*:

$$\int_0^{2\pi} f(\theta)\mathrm{d}\theta \approx 2\pi\underbrace{\frac{1}{n}\sum_{j=0}^{n-1} f(\theta_j)}_{\Sigma_n[f]}$$

Define the Trapezium rule approximation to the Fourier coefficients by:

$$\mathfrak{f}_k^n := \Sigma_n[f(\theta)\mathrm{e}^{-ik\theta}] = \frac{1}{n}\sum_{j=0}^{n-1} f(\theta_j)\mathrm{e}^{-ik\theta_j}$$

Lemma (Discrete orthogonality)
We have:

$$\sum_{j=0}^{n-1} \mathrm{e}^{ik\theta_j} = \begin{cases} n & k = \ldots, -2n, -n, 0, n, 2n, \ldots \\ 0 & \text{otherwise} \end{cases}$$

In other words,

$$\Sigma_n[\mathrm{e}^{i(k-j)\theta_j}] = \begin{cases} 1 & k-j = \ldots, -2n, -n, 0, n, 2n, \ldots \\ 0 & \text{otherwise} \end{cases}.$$

Proof

Consider $\omega := \mathrm{e}^{\mathrm{i}\theta_1} = \mathrm{e}^{\frac{2\pi \mathrm{i}}{n}}$. This is an n th root of unity: $\omega^n = 1$. Note that $\mathrm{e}^{\mathrm{i}\theta_j} = \mathrm{e}^{\frac{2\pi \mathrm{i} j}{n}} = \omega^j$.

(Case 1: $k = pn$ for an integer p)
We have

$$\sum_{j=0}^{n-1} \mathrm{e}^{ik\theta_j} = \sum_{j=0}^{n-1} \omega^{kj} = \sum_{j=0}^{n-1} (\omega^{pn})^j = \sum_{j=0}^{n-1} 1 = n$$

(Case 2 $k \neq pn$ for an integer p) Recall that

$$\sum_{j=0}^{n-1} z^j = \frac{z^n - 1}{z - 1}.$$

Then we have

$$\sum_{j=0}^{n-1} \mathrm{e}^{ik\theta_j} = \sum_{j=0}^{n-1} (\omega^k)^j = \frac{\omega^{kn} - 1}{\omega^k - 1} = 0.$$

where we use the fact that k is not a multiple of n to guarantee that $\omega^k \neq 1$.

∎

Theorem (discrete Fourier coefficients)
If $\mathbf{f}$ is absolutely convergent then

$$f_k^n = \cdots + f_{k-2n} + f_{k-n} + f_k + f_{k+n} + f_{k+2n} + \cdots$$

Proof

$$\begin{aligned}\mathfrak{f}_k^n &= \Sigma_n[f(\theta)\mathrm{e}^{-ik\theta}] = \sum_{j=-\infty}^{\infty} \mathfrak{f}_j \Sigma_n[f(\theta)\mathrm{e}^{i(j-k)\theta}] \\ &= \sum_{j=-\infty}^{\infty} \mathfrak{f}_j \begin{cases} 1 & j-k = \ldots, -2n, -n, 0, n, 2n, \ldots \\ 0 & \text{otherwise} \end{cases}\end{aligned}$$

∎

Note that there is redundancy:

> **Corollary (aliasing)**
> For all $p \in \mathbb{Z}$, $\mathfrak{f}_k^n = \mathfrak{f}_{k+pn}^n$.

In other words if we know $\mathfrak{f}_0^n, \ldots, \mathfrak{f}_{n-1}^n$, we know $\mathfrak{f}_k^n$ for all k via a permutation,
for example if $n = 2m+1$ we have

$$\begin{bmatrix} \mathfrak{f}_{-m}^n \\ \vdots \\ \mathfrak{f}_m^n \end{bmatrix} = \underbrace{\begin{bmatrix} & & & & 1 \\ & & & \iddots & \\ & & & & 1 \\ 1 & & & & \\ & \ddots & & & \\ & & 1 & & \end{bmatrix}}_{P_\sigma} \begin{bmatrix} \mathfrak{f}_0^n \\ \vdots \\ \mathfrak{f}_{n-1}^n \end{bmatrix}$$

where σ has Cauchy notation (*Careful*: we are using 1-based indexing here):

$$\begin{pmatrix} 1 & 2 & \cdots & m & m+1 & m+2 & \cdots & n \\ m+2 & m+3 & \cdots & n & 1 & 2 & \cdots & m+1 \end{pmatrix}.$$

We first discuss the case when all negative coefficients are zero,
noting that the Fourier series is in fact a Taylor series if we let $z = \mathrm{e}^{\mathrm{i}\theta}$:

$$f(z) = \sum_{k=0}^{\infty} \mathfrak{f}_k z^k.$$

That is, $\mathfrak{f}_0^n, \ldots, \mathfrak{f}_{n-1}^n$ are approximations of the Taylor series coefficients by evaluating
on the boundary.

We can prove *convergence* whenever of this
approximation whenever f has absolutely summable coefficients.
We will prove the result here in the special case where the negative
coefficients are zero.

> **Theorem (Taylor series converges)**
> If $0 = \mathfrak{f}_{-1} = \mathfrak{f}_{-2} = \cdots$ and $\mathbf{f}$ is absolutely convergent then

$$f_n(\theta) = \sum_{k=0}^{n-1} \hat{f}_k^n \mathrm{e}^{\mathrm{i}k\theta}$$

converges uniformly to $f(\theta)$.

Proof

$$|f(\theta) - f_n(\theta)| = |\sum_{k=0}^{n-1} (\hat{f}_k - \hat{f}_k^n)\mathrm{e}^{\mathrm{i}k\theta} + \sum_{k=n}^{\infty} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}| = |\sum_{k=n}^{\infty} \hat{f}_k (\mathrm{e}^{\mathrm{i}k\theta} - \mathrm{e}^{\mathrm{i}\mathrm{mod}(k,n)\theta})| \leq 2\sum_{k=n}^{\infty} |\hat{f}_k|$$

which goes to zero as $n \to \infty$.

∎

For the general case we need to choose a range of coefficients that includes roughly an equal number of negative and positive coefficients (preferring negative over positive in a tie as a convention):

$$f_n(\theta) = \sum_{k=-\lceil n/2 \rceil}^{\lfloor n/2 \rfloor} \hat{f}_k \mathrm{e}^{\mathrm{i}k\theta}$$

In the problem sheet we will prove this converges provided the coefficients are absolutely convergent.

3. Discrete Fourier transform and interpolation

We note that the map from values to coefficients can be defined as a matrix-vector product using the DFT:

Definition (DFT)
The *Discrete Fourier Transform (DFT)* is defined as:

$$\begin{aligned} Q_n &:= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mathrm{e}^{-\mathrm{i}\theta_1} & \mathrm{e}^{-\mathrm{i}\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}\theta_{n-1}} \\ 1 & \mathrm{e}^{-\mathrm{i}2\theta_1} & \mathrm{e}^{-\mathrm{i}2\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}2\theta_{n-1}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{e}^{-\mathrm{i}(n-1)\theta_1} & \mathrm{e}^{-\mathrm{i}(n-1)\theta_2} & \cdots & \mathrm{e}^{-\mathrm{i}(n-1)\theta_{n-1}} \end{bmatrix} \\ &= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \cdots & \omega^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix} \end{aligned}$$

for the n-th root of unity $\omega = \mathrm{e}^{\mathrm{i}\pi/n}$. Note that

$$
\begin{aligned}
Q_n^\star &= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \mathrm{e}^{\mathrm{i}\theta_1} & \mathrm{e}^{\mathrm{i}2\theta_1} & \cdots & \mathrm{e}^{\mathrm{i}(n-1)\theta_1} \\ 1 & \mathrm{e}^{\mathrm{i}\theta_2} & \mathrm{e}^{\mathrm{i}2\theta_2} & \cdots & \mathrm{e}^{\mathrm{i}(n-1)\theta_2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \mathrm{e}^{\mathrm{i}\theta_{n-1}} & \mathrm{e}^{\mathrm{i}2\theta_{n-1}} & \cdots & \mathrm{e}^{\mathrm{i}(n-1)\theta_{n-1}} \end{bmatrix} \\
&= \frac{1}{\sqrt{n}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}
\end{aligned}
$$

That is, we have

$$
\underbrace{\begin{bmatrix} f_0^n \\ \vdots \\ f_{n-1}^n \end{bmatrix}}_{\mathbf{f}^n} = \frac{1}{\sqrt{n}} Q_n \underbrace{\begin{bmatrix} f(\theta_0) \\ \vdots \\ f(\theta_n) \end{bmatrix}}_{\mathbf{f}^n}
$$

The choice of normalisation constant is motivated by the following:

Proposition (DFT is Unitary) Q_n is unitary: $Q_n^\star Q_n = Q_n Q_n^\star = I$.

Proof

$$
Q_n Q_n^\star = \begin{bmatrix} \Sigma_n[1] & \Sigma_n[\mathrm{e}^{\mathrm{i}\theta}] & \cdots & \Sigma_n[\mathrm{e}^{\mathrm{i}(n-1)\theta}] \\ \Sigma_n[\mathrm{e}^{-\mathrm{i}\theta}] & \Sigma_n[1] & \cdots & \Sigma_n[\mathrm{e}^{\mathrm{i}(n-2)\theta}] \\ \vdots & \vdots & \ddots & \vdots \\ \Sigma_n[\mathrm{e}^{-\mathrm{i}(n-1)\theta}] & \Sigma_n[\mathrm{e}^{-\mathrm{i}(n-2)\theta}] & \cdots & \Sigma_n[1] \end{bmatrix} = I
$$

∎

In other words, Q_n is easily inverted and we also have a map from discrete Fourier coefficients back to values:

$$
\sqrt{n} Q_n^\star \mathbf{f}^n = \mathbf{f}^n
$$

Corollary (Interpolation)
$f_n(\theta)$ interpolates f at θ_j:

$$
f_n(\theta_j) = f(\theta_j)
$$

Proof
We have

$$f_n(\theta_j) = \sum_{k=0}^{n-1} \mathfrak{f}_k^n \mathrm{e}^{\mathrm{i}k\theta_j} = \sqrt{n}\mathbf{e}_j^\top Q_n^\star \mathbf{f}_n = \mathbf{e}_j^\top Q_n^\star Q_n \mathbf{f}_n = f(\theta_j).$$

∎

We will leave extending this result to the problem sheet. Note that regardless of choice of coefficients we interpolate, though some interpolations are better than others:

```
using Plots, LinearAlgebra


## evaluates f_n at a point
function finitefourier(fₙ, θ)
    m = n ÷ 2 # use coefficients between -m:m
    ret = 0.0 + 0.0im # coefficients are complex so we need complex arithmetic
    for k = 0:m
        ret += fₙ[k+1] * exp(im*k*θ)
    end
    for k = -m:-1
        ret += fₙ[end+k+1] * exp(im*k*θ)
    end
    ret
end

function finitetaylor(fₙ, θ)
    ret = 0.0 + 0.0im # coefficients are complex so we need complex arithmetic
    for k = 0:n-1
        ret += fₙ[k+1] * exp(im*k*θ)
    end
    ret
end


f = θ -> exp(cos(θ))
n = 7
θ = range(0,2π; length=n+1)[1:end-1] # θ_0, …,θ_{n-1}, dropping θ_n == 2π
Qₙ = 1/sqrt(n) * [exp(-im*(k-1)*θ[j]) for k = 1:n, j=1:n]
fₙ = 1/sqrt(n) * Qₙ * f.(θ)


fₙ = θ -> finitefourier(fₙ, θ)
tₙ = θ -> finitetaylor(fₙ, θ)

g = range(0, 2π; length=1000) # plotting grid
plot(g, f.(g); label="function", legend=:bottomright)
plot!(g, real.(fₙ.(g)); label="Fourier")
plot!(g, real.(tₙ.(g)); label="Taylor")
scatter!(θ, f.(θ); label="samples")
```

We now demonstrate the relationship of Taylor and Fourier coefficients and their discrete approximations for some examples:

Example Consider the function

$$f(\theta) = \frac{2}{2 - \mathrm{e}^{\mathrm{i}\theta}}$$

Under the change of variables $z = \mathrm{e}^{\mathrm{i}\theta}$ we know for
z on the unit circle this becomes (using the geometric series with $z/2$)

$$\frac{2}{2-z} = \sum_{k=0}^{\infty} \frac{z^k}{2^k}$$

i.e., $f_k = 1/2^k$ which is absolutely summable:

$$\sum_{k=0}^{\infty} |f_k| = f(0) = 2.$$

If we use an n point discretisation we get (using the geoemtric series with 2^{-n})

$$f_k^n = f_k + f_{k+n} + f_{k+n} + \cdots = \sum_{p=0}^{\infty} \frac{1}{2^{k+pn}} = \frac{2^{n-k}}{2^n - 1}$$

We can verify this numerically:

```
f = θ -> 2/(2 - exp(im*θ))
n = 7
θ = range(0,2π; length=n+1)[1:end-1] # θ_0, …,θ_{n-1}, dropping θ_n == 2π
Qₙ = 1/sqrt(n) * [exp(-im*(k-1)*θ[j]) for k = 1:n, j=1:n]

Qₙ/sqrt(n)*f.(θ) ≈ 2 .^ (n .- (0:n-1)) / (2^n-1)
```

4. Fast Fourier Transform

Applying Q_n or its adjoint $Q_n^\star$ to a vector naively takes $O(n^2)$ operations.
Both can be reduced to $O(n \log n)$ using the celebrated *Fast Fourier Transform* (FFT),
which is one of the Top 10 Algorithms of the 20th Century
(You won't believe number 7!).

The key observation is that hidden in Q_{2n} are 2 copies of
Q_n. We will work with multiple n we denote the n-th root as $\omega_n = \exp(2\pi/n)$.
Note that we can relate a vector of powers of ω_{2n} to two copies of vectors of powers of ω_n:

$$\underbrace{\begin{bmatrix} 1 \\ \omega_{2n} \\ \vdots \\ \omega_{2n}^{2n-1} \end{bmatrix}}_{\vec{\omega}_{2n}} = P_\sigma^\top \begin{bmatrix} I_n \\ \omega_{2n} I_n \end{bmatrix} \underbrace{\begin{bmatrix} 1 \\ \omega_n \\ \vdots \\ \omega_n^{n-1} \end{bmatrix}}_{\vec{\omega}_n}$$

where σ has the Cauchy notation

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n & n+1 & \cdots & 2n \\ 1 & 3 & 5 & \cdots & 2n-1 & 2 & \cdots & 2n \end{pmatrix}$$

That is, P_σ is the following matrix which takes the even entries and places them in the first n entries and the odd entries in the last n entries:

```
n = 4
σ = [1:2:2n-1; 2:2:2n]
P_σ = I(2n)[σ,:]
```

and so $P_\sigma^\top$ reverses the process.
Thus we have

$$\begin{aligned} Q_{2n}^\star &= \frac{1}{\sqrt{2n}} \begin{bmatrix} \mathbf{1}_{2n} | \vec{\omega}_{2n} | \vec{\omega}_{2n}^2 | \cdots | \vec{\omega}_{2n}^{2n-1} \end{bmatrix} = \frac{1}{\sqrt{2n}} P_\sigma^\top \begin{bmatrix} \mathbf{1}_n & \vec{\omega}_n & \vec{\omega}_n^2 & \cdots & \vec{\omega}_n^{n-1} & \vec{\omega}_n^n & \cdots & \vec{\omega}_n^{2n-1} \\ \mathbf{1}_n & \omega_{2n}\vec{\omega}_n & \omega_{2n}^2\vec{\omega}_n^2 & \cdots & \omega_{2n}^{n-1}\vec{\omega}_n^{n-1} & \omega_{2n}^n\vec{\omega}_n^n & \cdots & \omega_{2n}^{2n-1}\vec{\omega}_n^{2n-1} \end{bmatrix} \\ &= \frac{1}{\sqrt{2}} P_\sigma^\top \begin{bmatrix} Q_n^\star & Q_n^\star \\ Q_n^\star D_n & -Q_n^\star D_n \end{bmatrix} = \frac{1}{\sqrt{2}} P_\sigma^\top \begin{bmatrix} Q_n^\star & \\ & Q_n^\star \end{bmatrix} \begin{bmatrix} I_n & I_n \\ D_n & -D_n \end{bmatrix} \end{aligned}$$

In other words, we reduced the DFT to two DFTs applied to vectors of half the dimension.

We can see this formula in code:

```
function fftmatrix(n)
    θ = range(0,2π; length=n+1)[1:end-1] # θ_0, …,θ_{n-1}, dropping θ_n == 2π
    [exp(-im*(k-1)*θ[j]) for k = 1:n, j=1:n]/sqrt(n)
end

Q₂ₙ = fftmatrix(2n)
Qₙ = fftmatrix(n)
Dₙ = Diagonal([exp(im*k*π/n) for k=0:n-1])
(P_σ'*[Qₙ' Qₙ'; Qₙ'*Dₙ -Qₙ'*Dₙ])[1:n,1:n] ≈ sqrt(2)Q₂ₙ'[1:n,1:n]
```

Now assume $n = 2^q$ so that $\log_2 n = q$. To see that we get $O(n \log n) = O(nq)$ operations we need to count the operations.
Assume that applying F_n takes $\leq 3nq$ additions and multiplications. The first n rows takes n additions. The last n has n multiplications and n additions.
Thus we have $6nq + 3n \leq 6n(q+1) = 3(2n)\log_2(2n)$ additions/multiplications, showing by induction that we have $O(n \log n)$ operations.

Remark: The FFTW.jl package wraps the FFTW (Fastest Fourier Transform in the West) library, which is a highly optimised implementation of the FFT that also works well even when n is not a power of 2. (As an aside, the creator of FFTW Steven Johnson is now a Julia contributor and user.)

Here we approximate $\exp\left(\cos\left(\theta - 0.1\right)\right)$ using 31 nodes:

```
using FFTW
f = θ -> exp(cos(θ-0.1))
n = 31
m = n÷2
## evenly spaced points from 0:2π, dropping last node
θ = range(0, 2π; length=n+1)[1:end-1]

## fft returns discrete Fourier coefficients n*[f̂n_0, …, f̂n_(n-1)]
fc = fft(f.(θ))/n

## We reorder using [f̂n_(-m), …, f̂n_(-1)] == [f̂n_(n-m), …, f̂n_(n-1)]
##  == [f̂n_(m+1), …, f̂n_(n-1)]
f̂ = [fc[m+2:end]; fc[1:m+1]]

## equivalent to f̂n_(-m)*exp(-im*m*θ) + … + f̂n_(m)*exp(im*m*θ)
fₙ = θ -> transpose([exp(im*k*θ) for k=-m:m]) * f̂

## plotting grid
g = range(0, 2π; length=1000)
plot(abs.(fₙ.(g) - f.(g)))
```

Thus we have successfully approximate the function to roughly machine precision. The magic of the FFT is because it's $O(n \log n)$ we can scale it to very high orders. Here we plot the Fourier coefficients for a function that requires around 100k coefficients to resolve:

```
f = θ -> exp(sin(θ))/(1+1e6cos(θ)^2)
n = 100_001
m = n÷2
## evenly spaced points from 0:2π, dropping last node
θ = range(0, 2π; length=n+1)[1:end-1]

## fft returns discrete Fourier coefficients n*[f̂n_0, …, f̂n_(n-1)]
fc = fft(f.(θ))/n


## We reorder using [f̂n_(-m), …, f̂n_(-1)] == [f̂n_(n-m), …, f̂n_(n-1)]
##  == [f̂n_(m+1), …, f̂n_(n-1)]
f̂ = [fc[m+2:end]; fc[1:m+1]]

plot(abs.(fc); yscale=:log10, legend=:bottomright, label="default")
plot!(abs.(f̂); yscale=:log10, label="reordered")
```

We can use the FFT to compute some mathematical objects efficiently. Here is a simple example.

Example Define the following infinite sum (which has no name apparently, according to Mathematica):

$$S_n(k) := \sum_{p=0}^{\infty} \frac{1}{(k+pn)!}$$

We can use the FFT to compute $S_n(0), \dots, S_n(n-1)$ in $O(n \log n)$ operations.
Consider

$$f(\theta) = \exp\left(\mathrm{e}^{\mathrm{i}\theta}\right) = \sum_{k=0}^{\infty} \frac{\mathrm{e}^{\mathrm{i}\theta}}{k!}$$

where we know the Fourier coefficients from the Taylor series of e^z.
The discrete Fourier coefficients satisfy for $0 \le k \le n-1$:

$$\mathfrak{f}_k^n = \mathfrak{f}_k + \mathfrak{f}_{k+n} + \mathfrak{f}_{k+2n} + \cdots = S_n(k)$$

Thus we have

$$\begin{bmatrix} S_n(0) \\ \vdots \\ S_n(n-1) \end{bmatrix} = \frac{1}{\sqrt{n}} Q_n \begin{bmatrix} 1 \\ \exp\left(\mathrm{e}^{2\mathrm{i}\pi/n}\right) \\ \vdots \\ \exp\left(\mathrm{e}^{2\mathrm{i}(n-1)\pi/n}\right) \end{bmatrix}$$