A. Introduction to Julia

References: The Julia Documentation, The Julia–Matlab–Python Cheatsheet, Think Julia

These notes give an overview of Julia. In these notes we focus on the aspects of Julia and computing that are essential to numerical computing:

1. Integers: We discuss briefly how to create and manipulate integers, and how to see the underlying bit representation.
2. Strings and parsing: We discuss how to create and manipulate strings and characters, and how we can convert a string

of 0's and 1's to an integer or other type. 3. Vectors and matrices: We discuss how to build and manipulate vectors and matrices (which are both types of *arrays*). Later lectures will discuss linear algebra. 4. Types: In Julia everything has a type, which plays a similar role to classes in Python. Here we discuss how to make new types, for example, a complex number in radial format. 5. Loops and branches: We discuss `if`, `for` and `while`, which work similar to Python. 6. Functions: We discuss the construction of named and anonymous functions. Julia allows overloading functions for different types, for example, we can overload `*` for our radial complex type. 7. Modules, Packages, and Plotting: We discuss how to load external packages, in particular, for plotting.

1. Integers

Julia uses a math-like syntax for manipulating integers:

In [1]:
```
1 + 1 # Addition
```

Out[1]: 2

In [2]:
```
2 * 3 # Multiplication
```

Out[2]: 6

In [3]:
```
2 / 3 # Division
```

Out[3]: 0.6666666666666666

In [4]:
```
x = 5; # semicolon is optional but supresses output if used in the last line
x^2 # Powers
```

Out[4]: 25

In Julia everything has a type. This is similar in spirit to a class in Python, but much more lightweight. An integer defaults to a type `Int`, which is either 32-bit (`Int32`) or 64-

bit (`Int64`) depending on the processor of the machine. There are also 8-bit (`Int8`), 16-bit (`Int16`), and 128-bit (`Int128`) integer types, which we can construct by converting an `Int` , e.g. `Int8(3)` .

These are all "primitive types", instances of the type are stored in memory as a fixed length sequence of bits. We can find the type of a variable as follows:

In [5]:
```
typeof(x)
```

Out[5]:
```
Int64
```

For a primitive type we can see the bits using the function `bitstring` :

In [6]:
```
bitstring(Int8(1))
```

Out[6]:
```
"00000001"
```

Negative numbers may be surprising:

In [7]:
```
bitstring(-Int8(1))
```

Out[7]:
```
"11111111"
```

This is explained in detail in Chapter Numbers.

There are other primitive integer types: `UInt8` , `UInt16` , `UInt32` , and `UInt64` are unsigned integers, e.g., we do not interpret the number as negative if the first bit is `1` . As they tend to be used to represent bit sequences they are displayed in hexadecimal, that is base-16, using digits 0-9a-c, e.g., $12 = (c)_{16}$:

In [8]:
```
UInt16(12)
```

Out[8]:
```
0x000c
```

A non-primitive type is `BigInt` which allows arbitrary length integers (using an arbitrary amount of memory):

In [9]:
```
factorial(big(100))^10
```

Out[9]:
```
501229411781623769749660913457440438241423232012200322468893211086126013704
894539443344309163974791659034852365995250806062011686578246599786735232198
821804597039806782364137104356302622509999458512471445459730019563426771428
142169285986787068214579295521341498939492118275642554858413261194143086490
268931209191751598129642166571081867269800866623596546422054167388335563209
506282983082667791283399575989899227202499266538194820953638625853344859790
670876861351789336077269493142900753157564688168114708204886421032550156933
567286643950190106682511408121306047648604161917328707343138398784542423497
604981394493598166286048081979508907005988879132386487437206698804316246051
907147850704889031422801542480308658883031446705268143162836428270481946917
616560029942842623910155313531513785228134356266366120116543326234048654779
501332706634928129188477486388847112098299317511538535401423273161204152693
099733507852570701555778059731473275144712539224416914644770413696741547032
650605385759271986528123972264597981517180893649589642619662794059376815997
114926751678156467705271005657886203103034886745869986816763168755424650630
543268271783299008211100347741675970542806368972994314611089910916303943706
258492662397699718485205856883926544847011250572559721527493331235100895744
451398327234919981678691884155485959440954156071175484497209589760000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
00000
```

2. Strings and parsing

We have seen that `bitstring` returns a string of bits. Strings can be created with quotation marks

In [10]:
```
str = "hello world 😀"
```

Out[10]:
```
"hello world 😀"
```

We can access characters of a string with brackets:

In [11]:
```
str[1], str[13]
```

Out[11]:
```
('h', '😀')
```

Each character is a primitive type, in this case using 32 bits/4 bytes:

In [12]:
```
typeof(str[6]), length(bitstring(str[6]))
```

Out[12]:
```
(Char, 32)
```

Strings are not primitive types, but rather point to the start of a sequence of `Char`s in memory. In this case, there are $32 * 13 = 416$ bits/52 bytes in memory.

Strings are *immutable*: once created they cannot be changed. But a new string can be created that modifies an existing string. The simplest example is `*`, which concatenates two strings:

In [13]:
```
"hi" * "bye"
```

Out[13]:
```
"hibye"
```

(Why `*`? Because concatenation is non-commutive.) We can combine this with indexing to, for example, create a new string with a different last character:

In [14]:
```
str[1:end-1] * "😂"
```

Out[14]:
```
"hello world 😂"
```

Parsing strings

We can use the command `parse` to turn a string into an integer:

In [15]:
```
parse(Int, "123")
```

Out[15]:
```
123
```

We can specify base 2 as an optional argument:

In [16]:
```
parse(Int, "-101"; base=2)
```

Out[16]:
```
-5
```

If we are specifying bits its safer to parse as an `UInt32`, otherwise the first bit is not recognised as a sign:

In [17]:
```
bts = "11110000100111111001100110001010"
x = parse(UInt32, bts; base=2)
```

Out[17]:
```
0xf09f998a
```

The function `reinterpret` allows us to reinterpret the resulting sequence of 32 bits as a different type. For example, we can reinterpret as an `Int32` in which case the first bit is taken to be the sign bit and we get a negative number:

In [18]:
```
reinterpret(Int32, x)
```

Out[18]:
```
-257975926
```

We can also reinterpret as a `Char`:

In [19]:
```
reinterpret(Char, x)
```

Out[19]:
```
'🙊': Unicode U+1F64A (category So: Symbol, other)
```

We will use `parse` and `reinterpret` as it allows one to easily manipulate bits. This is not actually how one should do it as it is slow.

Bitwise operations (non-examinable)

In practice, one should manipulate bits using bitwise operations. These will not be required in this course and are not examinable, but are valuable to know if you have a career involving high performance computing. The `p << k` shifts the bits of `p` to the left `k` times inserting zeros, while `p >> k` shifts to the right:

In [20]:
```
println(bitstring(23));
println(bitstring(23 << 2));
println(bitstring(23 >> 2));
```

```
0000000000000000000000000000000000000000000000000000000000010111
0000000000000000000000000000000000000000000000000000000001011100
0000000000000000000000000000000000000000000000000000000000000101
```

The operations `&`, `|` and `⊻` do bitwise and, or, and xor.

3. Vectors, Matrices, and Arrays

We can create a vector using brackets:

In [21]:
```
v = [11, 24, 32]
```

Out[21]:
```
3-element Vector{Int64}:
 11
 24
 32
```

Like a string, elements are accessed via brackets. Julia uses 1-based indexing (like Matlab and Mathematica, unlike Python and C which use 0-based indexing):

In [22]:
```
v[1], v[3]
```

Out[22]:
```
(11, 32)
```

Accessing outside the range gives an error:

In [23]:
```
v[4]
```

```
BoundsError: attempt to access 3-element Vector{Int64} at index [4]

Stacktrace:
 [1] getindex(A::Vector{Int64}, i1::Int64)
   @ Base ./array.jl:924
 [2] top-level scope
   @ In[23]:1
```

Vectors can be made with different types, for example, here is a vector of three 8-bit integers:

In [24]:
```
v = [Int8(11), Int8(24), Int8(32)]
```

Out[24]:
```
3-element Vector{Int8}:
 11
 24
 32
```

Just like strings, Vectors are not primitive types, but rather point to the start of sequence of bits in memory that are interpreted in the corresponding type. In this last case, there are $3 * 8 = 24$ bits/3 bytes in memory.

The easiest way to create a vector is to use `zeros` to create a zero `Vector` and then modify its entries:

In [25]:
```
v = zeros(Int, 5)
v[2] = 3
v
```

Out[25]:
```
5-element Vector{Int64}:
 0
 3
 0
 0
 0
```

Note: we can't assign a non-integer floating point number to an integer vector:

In [26]:
```
v[2] = 3.5
```

```
InexactError: Int64(3.5)

Stacktrace:
 [1] Int64
   @ ./float.jl:788 [inlined]
 [2] convert
   @ ./number.jl:7 [inlined]
 [3] setindex!(A::Vector{Int64}, x::Float64, i1::Int64)
   @ Base ./array.jl:966
 [4] top-level scope
   @ In[26]:1
```

We can also create vectors with `ones` (a vector of all ones), `rand` (a vector of random numbers between `0` and `1`) and `randn` (a vector of samples of normal distributed quasi-random numbers).

When we create a vector whose entries are of different types, they are mapped to a type that can represent every entry. For example, here we input a list of one `Int32` followed by three `Int64`s, which are automatically converted to all be `Int64`:

In [27]:
```
[Int32(1), 2, 3, 4]
```

Out[27]:
```
4-element Vector{Int64}:
 1
 2
 3
 4
```

In the event that the types cannot automatically be converted, it defaults to an `Any` vector, which is similar to a Python list. This is bad performancewise as it does not know how many bits each element will need, so should be avoided.

In [28]:
```
[1.0, 1, "1"]
```

Out[28]:
```
3-element Vector{Any}:
 1.0
 1
  "1"
```

We can also specify the type of the Vector explicitly by writing the desired type before the first bracket:

In [29]:
```
Int32[1, 2, 3]
```

Out[29]:
```
3-element Vector{Int32}:
 1
 2
 3
```

We can also create an array using comprehensions:

In [30]:
```
[k^2 for k = 1:5]
```

Out[30]:
```
5-element Vector{Int64}:
  1
  4
  9
 16
 25
```

Matrices are created similar to vectors, but by specifying two dimensions instead of one. Again, the simplest way is to use `zeros` to create a matrix of all zeros:

In [31]:
```
zeros(Int, 4, 5) # creates a 4 × 5 matrix of Int zeros
```

Out[31]:
```
4×5 Matrix{Int64}:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
```

We can also create matrices by hand. Here, spaces delimit the columns and semicolons delimit the rows:

In [32]:
```
A = [1 2; 3 4; 5 6]
```

Out[32]:
```
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
```

We can also create matrices using brackets, a formula, and a `for` command:

In [33]:
```
[k^2+j for k=1:4, j=1:5]
```

Out[33]:
```
4×5 Matrix{Int64}:
  2   3   4   5   6
  5   6   7   8   9
 10  11  12  13  14
 17  18  19  20  21
```

Matrices are really vectors in disguise. They are still stored in memory in a consecutive sequence of bits. We can see the underlying vector using the `vec` command:

In [34]:
```
vec(A)
```

Out[34]:
```
6-element Vector{Int64}:
 1
 3
 5
 2
 4
 6
```

The only difference between matrices and vectors from the computers perspective is that they have a `size` which changes the interpretation of whats stored in memory:

In [35]:
```
size(A)
```

Out[35]:
```
(3, 2)
```

Matrices can be manipulated easily on a computer. We can multiply a matrix times vector:

In [36]:
```
x = [8; 9]
A * x
```

Out[36]:
```
3-element Vector{Int64}:
 26
 60
 94
```

or a matrix times matrix:

In [37]:
```
A * [4 5; 6 7]
```

Out[37]:
```
3×2 Matrix{Int64}:
 16  19
 36  43
 56  67
```

If you use `.*`, it does entrywise multiplication:

In [38]:
```
[1 2; 3 4] .* [4 5; 6 7]
```

Out[38]:
```
2×2 Matrix{Int64}:
  4  10
 18  28
```

We can take the transpose of a real vector as follows:

In [39]:
```
a = [1, 2, 3]
a'
```

Out[39]:
```
1×3 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  3
```

Note for complex-valued vectors this is the conjugate-transpose, and so one may need to use `transpose(a)`. Both `a'` and `transpose(a)` should be thought of as "dual-vectors", and so multiplcation with a transposed vector with a normal vector gives a constant:

In [40]:
```
b = [4, 5, 6]
a' * b
```

Out[40]:
```
32
```

One important note: a vector is not the same as an `n x 1` matrix, and a transposed vector is not the same as a `1 × n` matrix.

Accessing and altering subsections of arrays

We will use the following notation to get at the columns and rows of matrices:

```
A[a:b,k]    # returns the a-th through b-th rows of the k-th
column of A as a Vector of length (b-a+1)
A[k,a:b]    # returns the a-th through b-th columns of the k-
th row of A as a Vector of length (b-a+1)
A[:,k]      # returns all rows of the k-th column of A as a
Vector of length size(A,1)
A[k,:]      # returns all columns of the k-th row of A as a
Vector of length size(A,2)
A[a:b,c:d]  # returns the a-th through b-th rows and c-th
through d-th columns of A
            # as a (b-a+1) x (d-c+1) Matrix
```

The ranges `a:b` and `c:d` can be replaced by any `AbstractVector{Int}`. For example:

In [41]:
```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
A[[1,3,4],2]  # returns the 1st, 3rd and 4th rows of the 2nd column of A
```

Out[41]:
```
3-element Vector{Int64}:
  2
  8
 11
```

Exercise Can you guess what `A[2,[1,3,4]]` returns, using the definition of `A` as above?
What about `A[1:2,[1,3]]`? And `A[1,B[1:2,1]]`? And `vec(A[1,B[1:2,1]])`?

We can also use this notation to modify entries of the matrix. For example, we can set the `1:2` x `2:3` subblock of A to `[1 2; 3 4]` as follows:

In [42]:
```
A[1:2,2:3] = [1 2; 3 4]
A
```

Out[42]:
```
4×3 Matrix{Int64}:
  1   1   2
  4   3   4
  7   8   9
 10  11  12
```

Broadcasting

It often is necessary to apply a function to every entry of a vector. By adding `.` to the end of a function we "broadcast" the function over a vector:

In [43]:
```
x = [1,2,3]
cos.(x) # equivalent to [cos(1), cos(2), cos(3)]
```

Out[43]:
```
3-element Vector{Float64}:
  0.5403023058681398
 -0.4161468365471424
 -0.9899924966004454
```

Broadcasting has some interesting behaviour for matrices. If one dimension of a matrix (or vector) is 1, it automatically repeats the matrix (or vector) to match the size of another example.

Example

In [44]:
```
[1,2,3] .* [4,5]'
```

Out[44]:
```
3×2 Matrix{Int64}:
  4   5
  8  10
 12  15
```

Since `size([1,2,3],2) == 1` it repeats the same vector to match the size `size([4,5]',2) == 2`. Similarly, `[4,5]'` is repeated 3 times. So the above is equivalent to:

In [45]:
```
[1 1; 2 2; 3 3] .* [4 5; 4 5; 4 5]
```

Out[45]:
```
3×2 Matrix{Int64}:
  4   5
  8  10
 12  15
```

Note we can also use broadcasting with our own functions (construction discussed later):

In [46]:
```
f = (x,y) -> cos(x + 2y)
f.([1,2,3], [4,5]')
```

Out[46]:
```
3×2 Matrix{Float64}:
 -0.91113    0.0044257
 -0.839072   0.843854
  0.0044257  0.907447
```

Ranges

We have already seen that we can represent a range of integers via `a:b`. Note we can convert it to a `Vector` as follows:

In [47]:
```
Vector(2:6)
```

Out[47]:
```
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

We can also specify a step:

In [48]:
```
Vector(2:2:6), Vector(6:-1:2)
```

Out[48]:
```
([2, 4, 6], [6, 5, 4, 3, 2])
```

Finally, the `range` function gives more functionality, for example, we can create 4 evenly spaced points between `-1` and `1`:

In [49]:
```
Vector(range(-1, 1; length=4))
```

Out[49]:
```
4-element Vector{Float64}:
 -1.0
 -0.3333333333333333
  0.3333333333333333
  1.0
```

Note that `Vector` is mutable but a range is not:

In [50]:
```
r = 2:6
r[2] = 3   # Not allowed
```

```
CanonicalIndexError: setindex! not defined for UnitRange{Int64}

Stacktrace:
 [1] error_if_canonical_setindex(#unused#::IndexLinear, A::UnitRange{Int6
4}, #unused#::Int64)
   @ Base ./abstractarray.jl:1352
 [2] setindex!(A::UnitRange{Int64}, v::Int64, I::Int64)
   @ Base ./abstractarray.jl:1343
 [3] top-level scope
   @ In[50]:2
```

4. Types

Julia has two different kinds of types: primitive types (like `Int64`, `Int32`, `UInt32` and `Char`) and composite types. Here is an example of an in-built composite type representing complex numbers, for example, $z = 1 + 2\mathrm{i}$:

In [51]:
```
z = 1 + 2im
typeof(z)
```

Out[51]:
```
Complex{Int64}
```

A complex number consists of two fields: a real part (denoted `re`) and an imaginary part (denoted `im`). Fields of a type can be accessed using the `.` notation:

In [52]:
```
z.re, z.im
```

Out[52]:
```
(1, 2)
```

We can make our own types. Let's make a type to represent complex numbers in the format

$$z = r\exp(\mathrm{i}\theta)$$

That is, we want to create a type with two fields: `r` and `θ`. This is done using the `struct` syntax, followed by a list of names for the fields, and finally the keyword `end`.

In [53]:
```
struct RadialComplex
    r
    θ
```

```
end
z = RadialComplex(1,0.1)
```

Out[53]: `RadialComplex(1, 0.1)`

We can access the fields using `.`:

In [54]:
```
z.r, z.θ
```

Out[54]: `(1, 0.1)`

Note that the fields are immutable: we can create a new `RadialComplex` but we cannot modify an existing one. To make a mutable type we use the command `mutable struct`:

In [55]:
```
mutable struct MutableRadialComplex
    r
    θ
end

z = MutableRadialComplex(1,2)
z.r = 2
z.θ = 3
z
```

Out[55]: `MutableRadialComplex(2, 3)`

Abstract types

Every type is a sub-type of an *abstract type*, which can never be instantiated on its own. For example, every integer and floating point number is a real number.
Therefore, there is an abstract type `Real`, which encapsulates many other types, including `Float64`, `Float32`, `Int64` and `Int32`.

We can test if type `T` is part of an abstract type `V` using the sytax `T <: V`:

In [56]:
```
Float64 <: Real, Float32 <: Real, Int64 <: Real
```

Out[56]: `(true, true, true)`

Every type has one and only one super type, which is *always* an abstract type.
The function `supertype` applied to a type returns its super type:

In [57]:
```
supertype(Int32)  # returns Signed, which represents all signed integers.
```

Out[57]: `Signed`

In [58]:
```
supertype(Float32) # returns `AbstractFloat`, which is a subtype of `Real`
```

Out[58]: `AbstractFloat`

An abstract type also has a super type:

In [59]:
```
supertype(Real)
```

Out[59]:
```
Number
```

Type annotation and templating

The types `RadialComplex` and `MutableRadialComplex` won't be efficient as we have not told the compiler the type of `r` and `θ`. For the purposes of this module, this is fine as we are not focussing on high performance computing. However, it may be of interest how to rectify this.

We can impose a type on the field name with `::`:

In [60]:
```
struct FastRadialComplex
    r::Float64
    θ::Float64
end
z = FastRadialComplex(1,0.1)
z.r, z.θ
```

Out[60]:
```
(1.0, 0.1)
```

In this case `z` is stored using precisely 128-bits.

Sometimes we want to support multiple types. For example, we may wish to support 32-bit floats. This can be done as follows:

In [61]:
```
struct TemplatedRadialComplex{T}
    r::T
    θ::T
end
z = TemplatedRadialComplex(1f0,0.1f0) # f0 creates a `Float32`
```

Out[61]:
```
TemplatedRadialComplex{Float32}(1.0f0, 0.1f0)
```

This is stored in precisely 64-bits.

Relationship with C structs, heap and stack (advanced)

For those familiar with C, a `struct` in Julia whose fields are primitive types or composite types built from primitive types, is exactly equivalent to a `struct` C, and can in fact be passed to C functions without any performance cost. Behind the scenes Julia uses the LLVM compiler and so C and Julia can be freely mixed.

Another thing to note is that from a programmers perspective there are three types of memory: registers, the stack and the heap. Registers only live on the CPU and are extremely fast. The stack lives in memory and has fixed memory length and is much

faster than the heap as it avoids dynamic allocation and deallocation of memory. So an instance of a type with a known fixed length (like `FastRadialComplex`) will typically live either in registers or in the stack (the compiler sorts out the details) and be much faster than an instance of a type with unknown or variable length (like `RadialComplex` or `Vector`), which must be on the heap.

5. Loops and branches

Loops such as `for` work essentially the same as in Python. The one caveat is to remember we are using 1-based indexing, e.g., `1:5` is a range consisting of `[1,2,3,4,5]`:

In [62]:
```
for k = 1:5
    println(k^2)
end
```

```
1
4
9
16
25
```

There are also `while` loops:

In [63]:
```
x = 1
while x < 5
    println("x is $x which is less than 5, incrementing!")
    x += 1
end
x
```

```
x is 1 which is less than 5, incrementing!
x is 2 which is less than 5, incrementing!
x is 3 which is less than 5, incrementing!
x is 4 which is less than 5, incrementing!
```

Out[63]:
```
5
```

If-elseif-else statements look like:

In [64]:
```
x = 5
if isodd(x)
    println("it's odd")
elseif x == 2
    println("it's 2")
else
    println("it's even")
end
```

```
it's odd
```

6. Functions

Functions are created in a number of ways. The most standard way is using the keyword `function`, followed by a name for the function, and in parentheses a list of arguments. Let's make a function that takes in a single number x and returns x^2.

In [65]:
```
function sq(x)
   x^2
end
sq(2), sq(3)
```

Out[65]:
```
(4, 9)
```

There is also a convenient syntax for defining functions on one line, e.g., we can also write

In [66]:
```
sq(x) = x^2
```

Out[66]:
```
sq (generic function with 1 method)
```

Multiple arguments to the function can be included with `,`.
Here's a function that takes in 3 arguments and returns the average.
(We write it on 3 lines only to show that functions can take multiple lines.)

In [67]:
```
function av(x, y, z)
    ret = x + y
    ret = ret + z
    ret/3
end
av(1, 2, 3)
```

Out[67]:
```
2.0
```

Variables live in different scopes. In the previous example, `x`, `y`, `z` and `ret` are *local variables*: they only exist inside of `av`.
So this means `x` and `z` are *not* the same as our complex number `x` and `z` defined above.

Warning: if you reference variables not defined inside the function, they will use the outer scope definition.
The following example shows that if we mistype the first argument as `xx`, then it takes on the outer scope definition `x`, which is a complex number:

In [68]:
```
function av2(xx, y, z)
    (x + y + z)/3
end
```

Out[68]:
```
av2 (generic function with 1 method)
```

You should almost never use this feature!!
We should ideally be able to predict the output of a function from knowing just the

inputs.

Example Let's create a function that calculates the average of the entries of a vector.

In [69]:
```
function vecaverage(v)
    ret=0
    for k = 1:length(v)
        ret = ret + v[k]
    end
    ret/length(v)
end
vecaverage([1,5,2,3,8,2])
```

Out[69]: 3.5

Julia has an inbuilt `sum` command that we can use to check our code:

In [70]:
```
sum([1,5,2,3,8,2])/6
```

Out[70]: 3.5

Functions with type signatures

functions can be defined only for specific types using `::` after the variable name. The same function name can be used with different type signatures.

The following defines a function `mydot` that calculates the dot product, with a definition changing depending on whether it is an `Integer` or a `Vector`.
Note that `Integer` is an abstract type that includes all integer types: `mydot` is defined for pairs of `Int64`'s, `Int32`'s, etc.

In [71]:
```
function mydot(a::Integer, b::Integer)
    a*b
end

function mydot(a::AbstractVector, b::AbstractVector)
    # we assume length(a)  == length(b)
    ret = 0
    for k = 1:length(a)
        ret = ret + a[k]*b[k]
    end
    ret
end

mydot(5, 6)  # calls the first definition
```

Out[71]: 30

In [72]:
```
mydot(Int8(5), Int8(6))   # also calls the first definition
```

Out[72]: 30

In [73]:
```
mydot(1:3, [4,5,6])    # calls the second definition
```

Out[73]:
```
32
```

We should actually check that the lengths of `a` and `b` match.
Let's rewrite `mydot` using an `if`, `else` statement. The following code only does the for loop if the length of a is equal to the length of b, otherwise, it throws an error.

If we name something with the exact same signature (name, and argument types), previous definitions get overriden. Here we correct the implementation of `mydot` to throw an error if the lengths of the inputs do not match:

In [74]:
```
function mydot(a::AbstractVector, b::AbstractVector)
    ret=0
    if length(a) == length(b)
        for k = 1:length(a)
            ret = ret + a[k]*b[k]
        end
    else
        error("arguments have different lengths")
    end
    ret
end
mydot([1,2,3], [5,6,7,8])
```

```
arguments have different lengths

Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:35
 [2] mydot(a::Vector{Int64}, b::Vector{Int64})
   @ Main ./In[74]:8
 [3] top-level scope
   @ In[74]:12
```

Anonymous (lambda) functions

Just like Python it is possible to make anonymous functions, with two variants on syntax:

In [75]:
```
f = x -> x^2
g = function(x)
    x^2
end
```

Out[75]:
```
#9 (generic function with 1 method)
```

There is not much difference between named and anonymous functions, both are compiled in the same manner. The only difference is named functions are in a sense "permanent". One can essentially think of named functions as "global constant anonymous functions".

Tuples

`Tuple`s are similar to vectors but written with the notation `(x,y,z)` instead of `[x,y,z]`. They allow the storage of *different types*. For example:

In [76]:
```
t = (1,2.0,"hi")
```

Out[76]:
```
(1, 2.0, "hi")
```

On the surface, this is very similar to a `Vector{Any}`:

In [77]:
```
v=[1,2.0,"hi"]
```

Out[77]:
```
3-element Vector{Any}:
 1
 2.0
  "hi"
```

The main difference is that a `Tuple` knows the type of its arguments:

In [78]:
```
typeof(t)
```

Out[78]:
```
Tuple{Int64, Float64, String}
```

The main benefit of tuples for us is that they provide a convenient way to return multiple arguments from a function. For example, the following returns both `cos(x)` and `x^2` from a single function:

In [79]:
```
function mytuplereturn(x)
    (cos(x), x^2)
end
mytuplereturn(5)
```

Out[79]:
```
(0.28366218546322625, 25)
```

We can also employ the convenient syntax to create two variables at once:

In [80]:
```
x,y = mytuplereturn(5)
```

Out[80]:
```
(0.28366218546322625, 25)
```

Modules, Packages, and Plotting

Julia, like Python, has modules and packages. For example to load support for linear algebra functionality like `norm` and `det`, we need to load the `LinearAlgebra` module:

In [81]:
```
using LinearAlgebra
norm([1,2,3]), det([1 2; 3 4])
```

Out[81]: (3.7416573867739413, -2.0)

It is fairly straightword to create ones own modules and packages, however, we will not need modules in this....module.

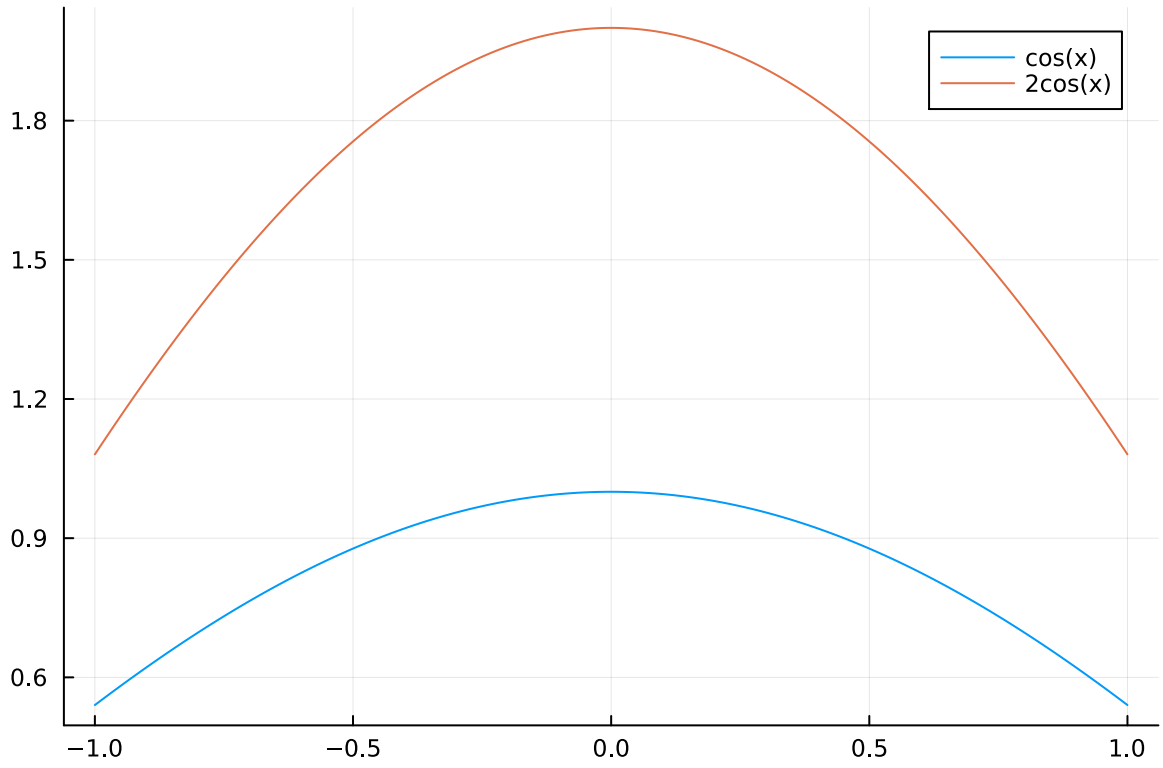
Plotting

Some important functionality such as plotting requires non-built in packages. There are many packages such as PyPlot.jl, which wraps Python's matplotlib and Makie.jl, which is a state-of-the-art GPU based 3D plotting package. We will use Plots.jl, which is an umbrella package that supports different backends.

For example, we can plot a simple function as follows:

In [82]:
```
using Plots
x = range(-1, 1; length=1000) # Create a range of a 1000 evenly spaced numbe
y = cos.(x) # Create a new vector with `cos` applied to each entry of `x`
plot(x, y; label="cos(x)")
plot!(x, 2y; label="2cos(x)")
```

Out[82]:



Note the `!` is just a convention: any function that modifies its input or global state should have `!` at the end of its name.

Installing packages (advanced)

If you choose to use Julia on your own machine, you may need to install packages. This can be done by typing the following, either in Jupyter or in the REPL: `] add Plots`.

B. 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 1 (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 2 (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 3 (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

1. $

{\cos n \over 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$.

2. $

\log n = o(n) $as \lim_{n → ∞} {\log n \over n} = 0. $

3. $

n^2 + 1 ~ n^2 as {n^2 +1 \over n^2} → 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 1 (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 4 (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 5 (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 6 (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:

In [1]:
```
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)
```

Out[1]:
```
90.95882254617737
```

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:

In [2]:
```
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)
```

Out[2]: 5377.916656174812

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.

C. Adjoints and Normal Matrices

Here we review

1. Complex inner-products
2. Adjoints
3. Normal Matrices and the spectral theorem

1. Complex-inner products

We will use bars to denote the complex-conjugate: if $z = x + \mathrm{i}y$ then $\bar{z} = x - \mathrm{i}y$.

Definition 1 (inner-product) An *inner product* $\langle \cdot, \cdot \rangle$ over $\mathbb{C}$ satisfies, for $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{C}^n$ and $a, b \in \mathbb{C}$,

1. Conjugate symmetry: $\langle \mathbf{x}, \mathbf{y} \rangle = \overline{\langle \mathbf{y}, \mathbf{x} \rangle}$
2. Linearity: $\langle \mathbf{x}, a\mathbf{y} + b\mathbf{z} \rangle = a\langle \mathbf{x}, \mathbf{y} \rangle + b\langle \mathbf{x}, \mathbf{z} \rangle$. (Some authors use linearity in the first argument.)
3. Positive-definiteness: for $\mathbf{x} \neq 0$ we have $\langle \mathbf{x}, \mathbf{x} \rangle > 0$.

We will usually use the standard inner product defined on $\mathbb{C}^n$:

$$\langle \mathbf{x}, \mathbf{y} \rangle := \bar{\mathbf{x}}^\top \mathbf{y} = \sum\nolimits_{k=1}^{n} \bar{x}_k y_k$$

Note that $\overline{zw} = \bar{z}\bar{w}$ and $\overline{z + w} = \bar{z} + \bar{w}$ together imply that:

$$\overline{A\mathbf{x}} = \bar{A}\bar{\mathbf{x}}.$$

2. Adjoints

Definition 2 (adjoint) Given an inner product, an adjoint of a matrix $A \in \mathbb{C}^{m \times n}$ is the unique matrix $A^\star$ satisfying

$$\langle \mathbf{x}, A\mathbf{y} \rangle = \langle A\mathbf{x}, \mathbf{y} \rangle$$

for all $\mathbf{x} \in \mathbb{C}^m$, $\mathbf{y} \in \mathbb{C}^n$. (Note this definition can be extended to other inner products.)

Proposition 1 (adjoints are conjugate-transposes) For the standard inner product, $A^\star = \bar{A}^\top$. If $A \in \mathbb{R}^{m \times n}$ then it reduces to the transpose $A^\star = A^\top$.

Proof

$$A_{k,j} = \langle \mathbf{e}_k, A\mathbf{e}_j \rangle = \langle A^\star \mathbf{e}_k, \mathbf{e}_j \rangle = \mathbf{e}_k^\top \overline{(A^\star)}^\top \mathbf{e}_j = \overline{A^\star_{j,k}}$$

∎

In this module we will assume the standard inner product (unless otherwise stated), that is, we will only use the standard adjoint $A^\star = \bar{A}^\top$. Note if $A = A^\star$ a matrix is called *Hermitian*. If it is real it is also called *Symmetric*.

3. Normal matrices

Definition 3 (normal) A (square) *normal matrix* commutes with its adjoint:

$$AA^\star = A^\star A$$

Examples of normal matrices include: 2. Symmetric and Hermitian: ($A^\star A = A^2 = AA^\star$) 3. Orthogonal and Unitary: ($Q^\star Q = I = QQ^\star$)

An important property of normal matrices is that they are diagonalisable using unitary eigenvectors:

Theorem 1 (spectral theorem for normal matrices) If $A \in \mathbb{C}^{n\times n}$ is normal then it is diagonalisable with unitary eigenvectors:

$$A = Q\Lambda Q^\star$$

where $Q \in U(n)$ and Λ is diagonal.

We will prove this later in the module. There is an important corollary for symmetric matrices that you may have seen before:

Corollary 1 (spectral theorem for symmetric matrices) If $A \in \mathbb{R}^{n\times n}$ is symmetric then it is diagonalisable with orthogonal eigenvectors:

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

where $Q \in O(n)$ and Λ is real and diagonal.

Proof

$A = Q\Lambda Q^\star$ since its normal hence we find that:

$$\Lambda = \Lambda^\star = (Q^\star A Q)^\star = Q^\star A^\top Q = Q^\star A Q = \Lambda$$

which shows that Λ is real. The fact that Q is real follows since the column $\mathbf{q}_k = Q\mathbf{e}_k$ is in the null space of the real matrix $A - \lambda_k I$.

∎

C. Spectral theorem for symmetric and normal matrices

Here we review the proof of the spectral theorem for symmetric and normal matrices, as well as adjoints (complex-conjugation). Here we use the standard inner product defined on $\mathbb{C}^n$:

$$\langle \mathbf{x}, \mathbf{y} \rangle := \bar{\mathbf{x}}^\top \mathbf{y} = \sum_{k=1}^n \bar{x}_k y_k$$

where the bars indicate complex conjugate: if $z = x + \mathrm{i}y$ then $\bar{z} = x - \mathrm{i}y$. Note that $\overline{zw} = \bar{z}\bar{w}$ and $\overline{z + w} = \bar{z} + \bar{w}$ together imply that:

$$\overline{A\mathbf{x}} = \bar{A}\bar{\mathbf{x}}.$$

1. Adjoints

Definition 1 (adjoint) An adjoint of a matrix $A \in \mathbb{C}^{m \times n}$ is its conjugate transpose: $A^\star := \bar{A}^\top$. If $A \in \mathbb{R}^{m \times n}$ then it reduces to the transpose $A^\star = A^\top$.

Note adjoints have the important product that for the standard inner product they satisfy:

$$\langle \mathbf{x}, A\mathbf{y} \rangle = \bar{\mathbf{x}}^\top (A\mathbf{y}) = (A^\top \bar{\mathbf{x}})^\top \mathbf{y} = (\overline{\bar{A}^\top \mathbf{x}})^\top \mathbf{y} = \langle \bar{A}\mathbf{x}, \mathbf{y} \rangle$$

2. Spectral theorem for symmetric matrices

Theorem (spectral theorem for symmetric matrices) If A is symmetric ($A = A^\top$) then

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

where Q is orthogonal ($Q^\top Q = I$) and Λ is real.

Proof

Recall every eigenvalue λ has at least one eigenvector $\mathbf{q}$ which we can normalize, but this can be complex. Thus we have

$$\lambda = \lambda \mathbf{q}^\star \mathbf{q} = \mathbf{q}^\star A \mathbf{q} =$$

∎

I.1 Integers

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. Unsigned integers: We discuss how computers represent non-negative integers using only p-bits, via modular arithmetic. 3. Signed integers: we discuss how negative integers are handled using the Two's-complement format. 4. As an advanced (non-examinable) topic we discuss `BigInt`, which uses variable bit length storage.

Before we begin its important to have a basic model of how a computer works. Our simplified model of a computer will consist of a Central Processing Unit (CPU)—the brains of the computer—and Memory—where data is stored. Inside the CPU there are registers, where data is temporarily stored after being loaded from memory, manipulated by the CPU, then stored back to memory.

Memory is a sequence of bits: `1`s and `0`s, essentially "on/off" switches. These are grouped into bytes, which consist of 8 bits. Each byte has a memory address: a unique number specifying its location in memory. The number of possible addresses is limited by the processor: if a computer has a a p-bit CPU then each address is represented by p bits, for a total of 2^p addresses (on a modern 64-bit CPU this is $2^{64} \approx 1.8 \times 10^{19}$ bytes). Further, each register consists of exactly p-bits.

A CPU has the following possible operations:

1. load data from memory addresses (up to p-bits) to a register
2. store data from a register to memory addresses (up to p-bits)
3. Apply some basic functions ("+", "-", etc.) to the bits in one or two registers

and write the result to a register.

Mathematically, the important point is CPUs only act on 2^p possible sequences of bits at a time. That is, essentially all functions f implemented on a CPU are either of the form $f : \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}$ or $f : \mathbb{Z}_{2^p} \times \mathbb{Z}_{2^p} \to \mathbb{Z}_{2^p}$, where we use the following notation:

Definition 1 ($\mathbb{Z}_m$, signed integers) Denote the

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

The limitations this imposes on representing integers is substantial. If we have an implementation of $+$, which we shall denote $\oplus_m$, how can we possibly represent $m+1$ in this implementation when the result is above the largest possible integer?

The solution that is used is straightforward: the CPU uses modular arithmetic. E.g., we have

$$(m-1) \oplus_m 1 = m \pmod m = 0.$$

In this chapter we discuss the implications of this approach and how it works with negative numbers.

We will use Julia in these notes to explore what is happening as a computer does integer arithmetic. We load an external package which implements functions `printbits` (and `printlnbits`) to print the bits (and with a newline) of numbers in colour:

In [1]:
```
using ColorBitstring
```

1. Binary representation

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

Definition 2 (binary format) For $B_0, \ldots, B_p \in \{0, 1\}$ denote an integer in *binary format* by:

$$\pm(B_p \ldots B_1 B_0)_2 := \pm \sum_{k=0}^{p} B_k 2^k$$

Example 1 (integers in binary) A simple integer example is $5 = 2^2 + 2^0 = (101)_2$. On the other hand, we write $-5 = -(101)_2$. Another example is $258 = 2^8 + 2 = (1000000010)_2$.

2. Unsigned Integers

Computers represent 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 dictate the first p binary digits: $(B_{p-1} \ldots B_1 B_0)_2$.

Integers on a computer follow modular arithmetic: 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. We denote modular arithmetic with $m = 2^p$ as follows:

$$\begin{aligned} x \oplus_m y &:= (x + y) \pmod m \\ x \ominus_m y &:= (x - y) \pmod m \\ x \otimes_m y &:= (x * y) \pmod m \end{aligned}$$

When m is implied by context we just write $\oplus, \ominus, \otimes$.

Example 2 (arithmetic with 8-bit unsigned integers) If arithmetic lies between 0 and $m = 2^8 = 256$ works as expected. For example,

$$17 \oplus_{256} 3 = 20 (\text{mod } 256) = 20$$
$$17 \ominus_{256} 3 = 14 (\text{mod } 256) = 14$$

This can be seen in Julia:

In [2]:
```
x = UInt8(17)  # An 8-bit representation of the number 255, i.e. with bits 0
y = UInt8(3)   # An 8-bit representation of the number   1, i.e. with bits 0
printbits(x); println(" + "); printbits(y); println(" = ")
printlnbits(x + y) # + is automatically modular arithmetic
printbits(x); println(" - "); printbits(y); println(" = ")
printbits(x - y) # - is automatically modular arithmetic
```

```
00010001 +
00000011 =
00010100
00010001 -
00000011 =
00001110
```

Example 3 (overflow with 8-bit unsigned integers) If we go beyond the range the result "wraps around". For example, with integers we have

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

However, the result is impossible to store in just 8-bits! So as mentioned instead it treats the integers as elements of $\mathbb{Z}_{256}$:

$$255 \oplus_{256} 1 = 255 + 1 \ (\text{mod } 256) = (00000000)_2 \ (\text{mod } 256) = 0 \ (\text{mod } 256)$$

We can see this in code:

In [3]:
```
x = UInt8(255) # An 8-bit representation of the number 255, i.e. with bits 1
y = UInt8(1)   # An 8-bit representation of the number   1, i.e. with bits 0
printbits(x); println(" + "); printbits(y); println(" = ")
printbits(x + y) # + is automatically modular arithmetic
```

```
11111111 +
00000001 =
00000000
```

On the other hand, if we go below 0 we wrap around from above:

$$3 \ominus_{256} 5 = -2 (\text{mod } 256) = 254 = (11111110)_2$$

In [4]:
```
x = UInt8(3) # An 8-bit representation of the number   3, i.e. with bits 000
y = UInt8(5) # An 8-bit representation of the number   5, i.e. with bits 000
printbits(x); println(" - "); printbits(y); println(" = ")
printbits(x - y) # + is automatically modular arithmetic
```

```
00000011 –
00000101 =
11111110
```

Example 4 (multiplication of 8-bit unsigned integers) Multiplication works similarly: for example,

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

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

In [5]:
```
x = UInt8(254) # An 8-bit representation of the number 254, i.e. with bits 1
y = UInt8(2)   # An 8-bit representation of the number   2, i.e. with bits 0
printbits(x); println(" * "); printbits(y); println(" = ")
printbits(x * y)
```

```
11111110 *
00000010 =
11111100
```

Hexadecimal and binary format

In Julia unsigned integers are displayed in hexadecimal form: that is, in base-16. Since there are only 10 standard digits (`0–9`) it uses 6 letters (`a–f`) to represent 11–16. For example,

In [6]:
```
UInt8(250)
```

Out[6]:
```
0xfa
```

because `f` corresponds to 15 and `a` corresponds to 10, and we have

$$15 * 16 + 10 = 250.$$

The reason for this is that each hex-digit encodes 4 bits (since 4 bits have $2^4 = 16$ possible values) and hence two hex-digits are encode 1 byte, and thus the digits correspond exactly with how memory is divided into addresses. We can create unsigned integers either by specifying their hex format:

In [7]:
```
0xfa
```

Out[7]:
```
0xfa
```

Alternatively, we can specify their digits. For example, we know $(f)_{16} = 15 = (1111)_2$ and $(a)_{16} = 10 = (1010)_2$ and hence $250 = (fa)_{16} = (11111010)_2$ can be written as

In [8]:
```
0b11111010
```

Out[8]:
```
0xfa
```

3. 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. More precisely:

Definition 3 ($\mathbb{Z}^s_{2^p}$, unsigned integers)

$$\mathbb{Z}^s_{2^p} := \{-2^{p-1}, \ldots, -1, 0, 1, \ldots, 2^{p-1} - 1\}$$

Definition 4 (Shifted mod) Define for $y = x \pmod{2^p}$

$$x \pmod{^s\ 2^p} := \begin{cases} y & 0 \leq y \leq 2^{p-1} - 1 \\ y - 2^p & 2^{p-1} \leq y \leq 2^p - 1 \end{cases}$$

Note that if $R_p(x) = x(\mathrm{mod}^{\mathrm{s}}\ 2^p)$ then it can be viewed as a map $R_p : \mathbb{Z} \to \mathbb{Z}^s_{2^p}$ or a one-to-one map $R_p : \mathbb{Z}_{2^p} \to \mathbb{Z}^s_{2^p}$ whose inverse is $R_p^{-1}(x) = x \mod 2^p$.

Example 5 (converting bits to signed integers) What 8-bit integer has the bits `01001001`? Because the first bit is 0 we know the result is positive. Adding the corresponding decimal places we get:

In [9]:
```
2^0 + 2^3 + 2^6
```

Out[9]: 73

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`:

In [10]:
```
2^0 + 2^3 + 2^6 + 2^7 - 2^8
```

Out[10]: -55

We can check the results using `printbits`:

In [11]:
```
printlnbits(Int8(73)) # Int8 is an 8-bit representation of the signed intege
printbits(-Int8(55))
```

```
01001001
11001001
```

Arithmetic works precisely the same for signed and unsigned integers, e.g. we have

$$x \oplus^s_{2^p} y := x + y(\mathrm{mod}^{\mathrm{s}}\ 2^p)$$

Example 6 (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:

$$-1 \oplus_{256} 1 = -1 + 1 \pmod{2^p} = 2^p - 1 + 1 \pmod{2^p} = 2^p \pmod{2^p} = 0 \pmod{2^p}$$

Example 7 (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:

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

Example 8 (overflow) We can find the largest and smallest instances of a type using `typemax` and `typemin`:

In [12]:
```
printlnbits(typemax(Int8)) # 2^7-1 = 127
printbits(typemin(Int8)) # -2^7 = -128
```

```
01111111
10000000
```

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

In [13]:
```
typemax(Int8) + Int8(1) # returns typemin(Int8)
```

Out[13]:
```
-128
```

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

Division

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

In [14]:
```
5 ÷ 2 # equivalent to div(5,2)
```

Out[14]:
```
2
```

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

In [15]:
```
5 / 2 # alternatively 2 \ 5
```

Out[15]:
```
2.5
```

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

In [16]:
```
(1//2) + (3//4)
```

Out[16]:
```
5//4
```

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

In [17]:
```
big(102324)//132413023 + 23434545//4243061 + 23434545//42430534435
```

Out[17]: 26339037835007648477541540//476780487870754436459646 1

4. Variable bit representation (non-examinable)

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:

In [18]:
```
x = typemax(Int64) + big(1) # Too big to be an `Int64`
```

Out[18]: 9223372036854775808

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

In [19]:
```
x^100
```

Out[19]:
```
3082994025277634745700106821545665721371798533305697458855342277921093731984476404705966539412410898240561729912372038501228893141921080152404642393776599077294434061519905424124601394226943601430916434383714716724720227331596950613701661034548948388721097667275438763758128508403297199458260277707301202460980093818414167080563342761482395862435185093942443540722363151770022221783243959592531336062998494209914752408019060720805124534382646051093613814848646062038662423487504326044361203708430489305864234333801401547140023376295718383390360728662900230671437151716615826286842267917560749586018165739492101920429719261285640125596833063891562865262157026023955919873792846823095854484520920509345944712871675691790827690907778485058829248588945681685288179787963931181062068092463984296225973082494056307958089189726701678735576365394146232076917088075949053636690459581128773097212746967276496496010810878000638239143750075543163240049874489986642327436441234458040254480825038220479904594615300602390556385799245276805580024937804723029319565942013515817048714543455250235208789745701165279569026248145395218985062991831707830217974393158466067785199581037714968820628241051867119832966361530047910339065726550260741036716100932205969655083257714244071120221654679340461084001560321676025443801248355439305974923873624147980728110581452806109011739005060060604228087667499288851218705078807364237925455813890575257569981450090997117697469299234094394984840574025401463942099019413361096233909056117427663439764954916401592565651111571414769257187704568268701243082044838400201357613851006471104244828842270232637747398962711875413488415772647088571125272932490717217468263604683325933469555629785507020775366368002753612709901526248456328209643292122899677436613886360765877886748185299249994921843183573130403496311896614949399409796011301191280067209053259341918813967552543176532349157376
```

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).

I.2 Reals

Reference: Overton (https://cs.nyu.edu/~overton/book/)

In this chapter, we introduce the IEEE Standard for Floating-Point Arithmetic (https://en.wikipedia.org/wiki/IEEE_754). There are multiplies ways of representing real numbers on a computer, as well as the precise behaviour of operations such as addition, multiplication, etc.: one can use

1. Fixed-point arithmetic (https://en.wikipedia.org/wiki/Fixed-point_arithmetic): essentially representing a real number as integer where a decimal point is inserted at a fixed point. This turns out to be impractical in most applications, e.g., due to loss of relative accuracy for small numbers.
2. Floating-point arithmetic (https://en.wikipedia.org/wiki/Floating-point_arithmetic): essentially scientific notation where an exponent is stored alongside a fixed number of digits. This is what is used in practice.
3. Level-index arithmetic (https://en.wikipedia.org/wiki/Symmetric_level-index_arithmetic): stores numbers as iterated exponents. This is the most beautiful mathematically but unfortunately is not as useful for most applications and is not implemented in hardware.

Before the 1980s each processor had potentially a different representation for floating-point 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 floating-point numbers in details:

1. Floating-point arithmetic is very precisely defined, and can even be used in rigorous computations as we shall see in the labs. But it is not exact and its important to understand how errors in computations can accumulate.
2. Failure to understand floating-point arithmetic can cause catastrophic issues in practice, with the extreme example being the explosion of the Ariane 5 rocket (https://youtu.be/N6PWATvLQCY?t=86).

In this chapter we discuss the following:

1. Real numbers in binary: we discuss how binary digits can be used to represent real numbers.
2. 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*.
3. 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.

4. High-precision floating-point numbers: As an advanced (non-examinable) 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.

In [1]:
```
using SetRounding, ColorBitstring
```

1. Real numbers in binary

Reals can also be presented in binary format, that is, a sequence of `0`s and `1`s alongside a decimal point:

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

$$(B_p \ldots B_0.\, b_1 b_2 b_3 \ldots)_2 := (B_p \ldots B_0)_2 + \sum_{k=1}^{\infty} \frac{b_k}{2^k}.$$

Example 1 (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. Floating-point numbers

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

Definition 2 (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

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

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

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

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

$$F^{\text{special}} := \{\infty, -\infty, \text{NaN}\}$$

where NaN is a special symbol representing "not a number", essentially an error flag.

Note this set of real numbers has no nice *algebraic structure*: it is not closed under addition, subtraction, etc. On the other hand, we can control errors effectively hence it is extremely useful for analysis.

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 3 (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`. Alternatively, one can use scientific notation: `1e0`. `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` (this is in fact scientific notation so `1f1 ≡ 10f0`). `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 limited 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 2 (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:

In [2]:
```
printbits(1f0/3)
```

```
00111110101010101010101010101011
```

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

In [3]:
```
printbits(0.0)
```

```
00000000000000000000000000000000000000000000000000000000000000000
```

Unlike integers, we also have a negative zero:

In [4]:
```
printbits(-0.0)
```

```
10000000000000000000000000000000000000000000000000000000000000000
```

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 4 (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

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

We confirm the simple bit representations:

In [5]:
```
σ,Q,S = 127,8,23 # Float32
εₘ = 2.0^(-S)
println_bits(Float32(2.0^(1-σ))) # smallest positive normal Float32
println_bits(Float32(2.0^(2^Q-2-σ) * (2-εₘ))) # largest normal Float
```

```
00000000100000000000000000000000
01111111011111111111111111111111
```

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

In [6]:
```
eps(Float32), floatmin(Float32), floatmax(Float32)
```

Out[6]:
```
(1.1920929f-7, 1.1754944f-38, 3.4028235f38)
```

Example 3 (creating a sub-normal number) If we divide the smallest normal number by two, we get a subnormal number:

In [7]:
```
mn = floatmin(Float32) # smallest normal Float32
println_bits(mn)
print_bits(mn/2)
```

```
00000000100000000000000000000000
00000000010000000000000000000000
```

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" 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):

In [8]:
```
printlnbits(Inf16)
printbits(-Inf16)
```

```
0111110000000000
1111110000000000
```

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):

In [9]:
```
printbits(NaN16)
```

```
0111111000000000
```

These are needed for undefined algebraic operations such as:

In [10]:
```
0/0
```

Out[10]: NaN

Essentially it is a CPU's way of indicating an error has occurred.

Example 4 (many `NaN`s) What happens if we change some other b_k to be nonzero? We can create bits as a string and see:

In [11]:
```
i = 0b0111110000010001 # an UInt16
reinterpret(Float16, i)
```

Out[11]: NaN16

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

3. 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 6 (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.)

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

In [12]:
```
printlnbits(1/3)  # 64 bits
printbits(Float32(1/3))  # round to nearest 32-bit
```

```
0011111111010101010101010101010101010101010101010101010101010101
00111110101010101010101010101011
```

The default rounding mode can be changed:

In [13]:
```
printbits(Float32(1/3,RoundDown) )
```

```
00111110101010101010101010101010
```

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

In [14]:
```
x = 1f0
setrounding(Float32, RoundDown) do
    x/3
end,
setrounding(Float32, RoundUp) do
    x/3
end
```

Out[14]:
```
(0.3333333f0, 0.33333334f0)
```

WARNING (compiled constants, non-examinable): 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 $\oplus, \ominus, \otimes, \oslash : F \otimes F \to F$ 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. Also, note that when we convert a Julia command with constants specified by decimal expansions we first round the constants to floats, e.g., `1.1 + 0.1` is actually reduced to

$$\mathrm{fl}(1.1) \oplus \mathrm{fl}(0.1)$$

This includes the case where the constants are integers (which are normally exactly floats but may be rounded if extremely large).

Example 5 (decimal is not exact) The Julia command `1.1+0.1` gives a different result than `1.2` :

In [15]:
```
x = 1.1
y = 0.1
x + y - 1.2 # Not Zero?!?
```

Out[15]:
```
2.220446049250313e-16
```

This is because $\mathrm{fl}(1.1) \neq 1 + 1/10$ and $\mathrm{fl}(1.1) \neq 1/10$ since their expansion in *binary* is not finite, but rather:

$$\mathrm{fl}(1.1) = (1.0001100110011001100110011001100110011001100110011010)_2$$
$$\mathrm{fl}(0.1) = 2^{-4} * (1.1001100110011001100110011001100110011001100110011010($$
$$= (0.0001100110011001100110011001100110011001100110011001100110011010)$$

Thus when we add them we get

$$\mathrm{fl}(1.1) + \mathrm{fl}(1.1) = (1.0011001100110011001100110011001100110011001100110011001100($$

where the red digits indicate those beyond the 52 representable in F_{54}. In this case we round up and get

$$\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.1) = (1.0011001100110011001100110011001100110011001100110011001100$$

On the other hand,

$$\mathrm{fl}(1.2) = (1.0110011001100110011001100110011001100110011001100110011)_2$$

which differs by 1 bit.

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:

In [16]:
```
(1.1 + 1.2) + 1.3, 1.1 + (1.2 + 1.3)
```

Out[16]:
```
(3.5999999999999996, 3.6)
```

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 7 (absolute/relative error) If $\tilde{x} = x + \delta_\mathrm{a} = x(1 + \delta_\mathrm{r})$ 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 8 (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:

$$\mathcal{N}_{\sigma,Q,S} := \{x : \min |F^{\text{normal}}_{\sigma,Q,S}| \leq |x| \leq \max F^{\text{normal}}_{\sigma,Q,S}\}$$

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 1 (round bound) If $x \in \mathcal{N}$ then

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

where the *relative error* is

$$|\delta_x^{\text{nearest}}| \le \frac{\epsilon_m}{2}$$
$$|\delta_x^{\text{up/down}}| < \epsilon_m.$$

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) $|\delta_1| \le \frac{\epsilon_m}{2}$.

Example 6 (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 $\text{fl}(1.1)$. Thus this computation becomes

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

First we find

$$(\text{fl}(1.1) \oplus \text{fl}(1.2)) = (1.1(1 + \delta_1) + 1.2(1 + \delta_2))(1 + \delta_3) = 2.3 + \underbrace{1.1\delta_1 + 1.2\delta_2 + 2}$$

In this module we will never ask for precise bounds: that is, we will always want bounds of the form $C\epsilon_m$ for a specified constant C but the choice of C need not be sharp. Thus we will tend to round up to integers. Further, while $\delta_1\delta_3$ and $\delta_2\delta_3$ are absolutely tiny we will tend to bound them rather naïvely by $|\epsilon_m/2|$. Using these rules we have the bound

$$|\delta_4| \le (1 + 1 + 2 + 1 + 1)\epsilon_m = 6\epsilon_m$$

Thus the computation becomes

$$((2.3 + \delta_4) + 1.3(1 + \delta_5))(1 + \delta_6) = 3.6 + \underbrace{\delta_4 + 1.3\delta_5 + 3.6\delta_6 + \delta_4\delta_6 + 1.3\delta_5\delta_6}_{\delta_7}$$

where the *absolute error* is

$$|\delta_7| \le (6 + 1 + 2 + 1 + 1)\epsilon_m = 11\epsilon_m$$

Indeed, this bound is bigger than the observed error:

In [17]:
```
abs(3.6 - (1.1+1.2+1.3)), 11eps()
```

Out[17]:
```
(4.440892098500626e-16, 2.4424906541753444e-15)
```

Arithmetic and special numbers

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

In [18]:
```
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
```

Out[18]:
```
true
```

Special functions (non-examinable)

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 $|\delta| < c\epsilon_{\rm m}$ for a reasonably small $c > 0$, *provided* that $x \in \mathrm{F}^{\rm normal}$. Note these special functions are written in (advanced) Julia code, for example, sin (https://github.com/JuliaLang/julia/blob/d08b05df6f01cf4ec6e4c28ad94cedda76cc62e8/ba

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:

In [19]:
```
π₆₄ = Float64(π)
πᵦ = big(π₆₄) # Convert 64-bit approximation of π to higher precisi
abs(sin(π₆₄)), abs(sin(π₆₄) - sin(πᵦ)) # only has relative accuracy
```

Out[19]:
```
(1.2246467991473532e-16, 2.99476980971833986075426382233777881143079984105459688279415867658134246764335 5e-33)
```

Another issue is when x is very large:

In [20]:
```
ε = eps() # machine epsilon, 2^(-52)
x = 2*10.0^100
abs(sin(x) - sin(big(x)))  ≤  abs(sin(big(x))) * ε
```

Out[20]:
```
true
```

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

In [21]:
```
x̃ = 2*big(10.0)^100
sin(x), sin(x̃)
```

Out[21]:
```
(-0.703969872087777, 0.691191084503746221962375159497891426040396639271694499036093734000130024296540 8)
```

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

$$2 * 10^{100} * \epsilon_m \approx 4.44 * 10^{84}$$

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

Example 7 (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

In [22]:
```
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
```

Out[22]:
```
(6.8081941268545685452715535031250016405281102332969211943236587103456258773803 71e-19, 0.00196232835409716699359705671668052673339843750000000000000000000000000000000008)
```

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):
$(x \otimes x) \oplus 4x + 1 = (x^2(1+\delta_1) + 4x)(1+\delta_2) + 1 = x^2 + 4x + 1 + \delta_1 x^2 + 4x\delta_2 +$
Using a simple bound $|x| < 1$ we get a (pessimistic) bound on the absolute error of $3\epsilon_m$. Here `f(x)` itself is less than $2\epsilon_m$ 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.)

4. High-precision floating-point numbers (non-examinable)

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:

In [23]:
```
big(1)/3
```

Out[23]:
```
0.33333333333333333333333333333333333333333333333333333333333333333333348
```

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

In [24]:
```
setrounding(BigFloat, RoundDown) do
  big(1)/3
end, setrounding(BigFloat, RoundUp) do
  big(1)/3
end
```

Out[24]:
```
(0.33333333333333333333333333333333333333333333333333333333333333333333305, 0.33333333333333333333333333333333333333333333333333333333333333333333348)
```

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

In [25]:
```julia
setprecision(4_000) do # 4000 bit precision
  setrounding(BigFloat, RoundDown) do
    big(1)/3
  end, setrounding(BigFloat, RoundUp) do
    big(1)/3
  end
end
```

Out[25]:
```
(0.333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333308, 0.33333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333346)
```

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

I.3 Divided Differences

We now get to our first computational problem: given a function, how can we approximate its derivative at a point? We consider an intuitive approach to this problem using *Divided Differences*:

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

for h small. From the definition of a derivative we know that as $h \to 0$ the approximation (the right-hand side) converges to $f'(x)$, hence a natural algorithm is use this formula for very small h. Unfortunately, the round-off errors of floating point arithmetic introduces typically limit its accuracy.

In this chapter we view f as a *Black-box function* which we can only evaluate *pointwise*: Consider a floating-point valued function $f^{\mathrm{FP}} : D \to F$ where $D \subset F \equiv F_{\sigma,Q,S}$. For example, if we have `f(x::Float64) = sqrt(x)` then our function is defined only for positive `Float64` (otherwise it throws an error) hence $D = [0, \infty) \cap F_{64}$. This is the situation if we have a function that relies on a compiled C library, which hides the underlying operations (which are usually composition of floating point operations). Since the set of floating point numbers F is discrete, f^{FP} cannot be differentiable in an obvious way, therefore we need to assume that f^{FP} approximates a differentiable function f with controllable error bounds in order to state anything precise.

In the next chapter (I.4) we wiil introduce a more accurate approach based on *dual numbers*, which requires a more general notion of a function where a formula (i.e., code) is available.

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, where the adjoints of Jacobians of each operation are constructed in a way that they can be composed (essentially the chain rule). 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. 4. Interpolation and differentiation: We can also differentiate functions *globally*, that is, in an interval instead of only a single point, which will be discussed in Part III of the module.

In [1]:
```
using ColorBitstring
```

1. Divided 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 1 The error in approximating the derivative using divided 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 divided differences. Leftside divided 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*.

2. Does divided 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 divided difference approximation to their floating point implementations f^{FP} and g^{FP}:

In [2]:
```
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
```

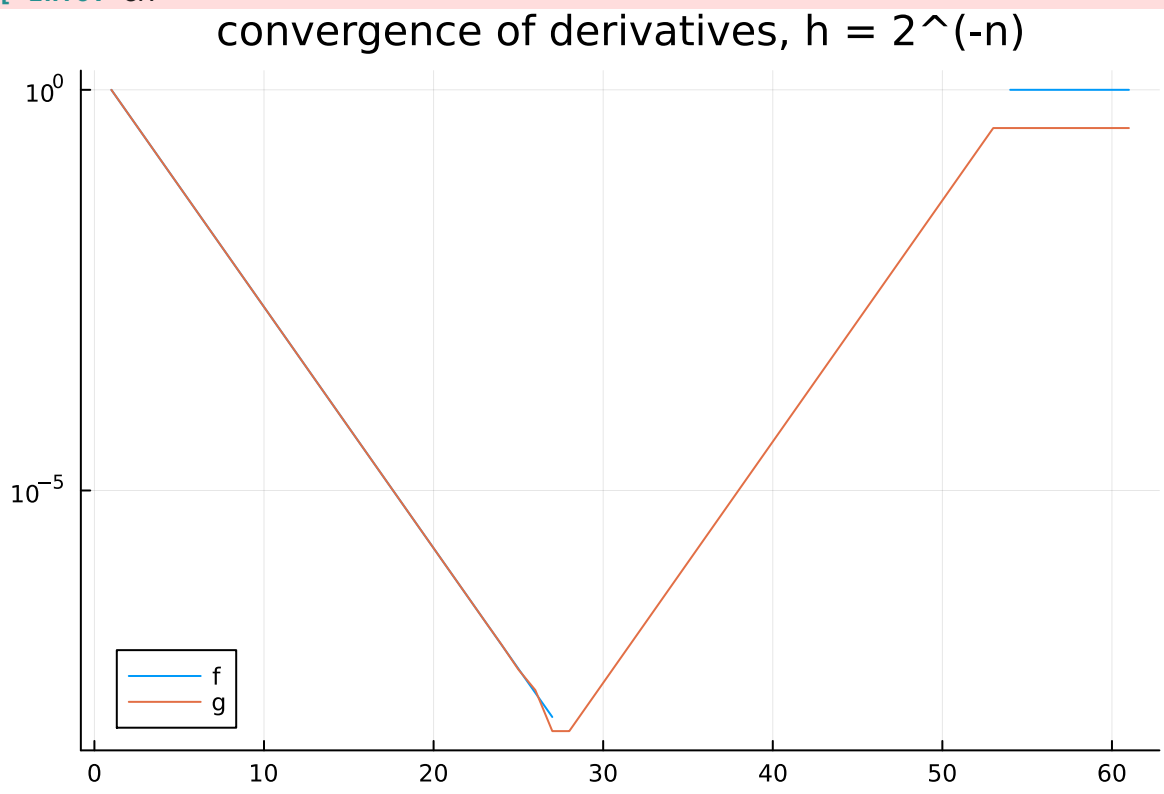
Out[2]:
```
(1.000001000006634, 0.33333433346882657)
```

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.

In [3]:
```
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 o
plot!(abs.((g.(h) .- g(0)) ./ h .- 1/3); yscale=:log10, label = "g")
```

```
[ Info: Precompiling Plots [91a5bcdd-55d7-5caf-9e0b-520d859cae80]
[ Info: GR
```

Out[3]:

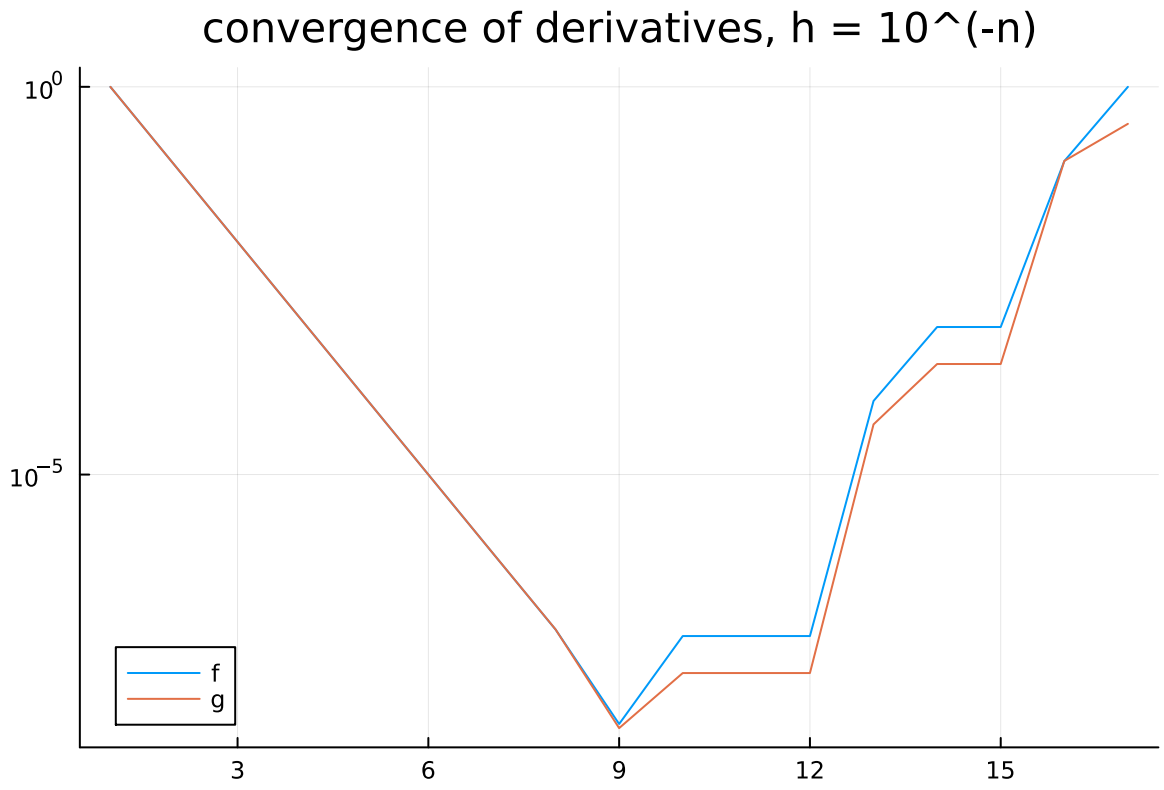


In the case of f it is a success: we approximate the true derivative *exactly* provided we take $h = 2^{-n}$ for $26 < n \leq 52$. But for g it is a huge failure: the approximation starts to converge, but then diverges exponentially fast, before levelling off!

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:

In [4]:
```
h = 10.0 .^ (0:-1:-16)  # [1,1/10,1/100,…]
plot(abs.((f.(h) .- f(0)) ./ h .- 1); yscale=:log10, title="convergence of d
plot!(abs.((g.(h) .- g(0)) ./ h .- 1/3); yscale=:log10, label = "g")
```

Out[4]:



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

Example 1 (convergence(?) of divided differences) Consider differentiating $f(x) = 1 + x + x^2$ at 0 with $h = 2^{-n}$. We consider 4 different cases with different behaviour, where S is the number of significand bits:

1. $0 \le n \le S/2$
2. $S/2 < n \le S$
3. $S \le n < S + \sigma$
4. $S + \sigma \le n$

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 ($0 \le n \le S/2$): 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:

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

```
0011110010010000
```

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

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

Case 4 ($\sigma + S \leq n$): In this case $h = 2^{-n}$ is rounded to zero and hence we get NaN.

Example 2 (divergence of divided differences) 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 $1 \oplus (h \oslash 3)$:

In [6]:
```
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)
```

```
0011110101010101
0011110001010101
0011110000010101
0011110000000001
```

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 $S \leq n < \sigma + S$ then $1 \oplus (h \oslash 3) = 1$ and we have

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

3. Bounding the error

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

Theorem 1 (divided 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 $c \geq 0$. Assume for simplicity $h = 2^{-n}$ where $n \leq S$ and $|x| \leq 1$. Assuming that all calculations result in normal floating point numbers, the divided 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 (divided difference with floating-point step) Choose h proportional to $\sqrt{\epsilon_{\mathrm{m}}}$ in divided 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 divided 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.

I.4 Dual Numbers

Int his chapter we introduce a mathematically beautiful alternative to divided differences for computing derivatives: *dual numbers*. As we shall see, these are a commutative ring that *exactly* compute derivatives, which when implemented in floating point give very high-accuracy approximations to derivatives. They underpin forward-mode automatic differentation.

Before we begin, we must be clear what a "function" is. Consider three possible scenarios:

1. *Black-box function*: this was considered last chapter where we can only input floating-point numbers

and get out a floating-point number. 2. *Generic function*: Consider a function that is a formula (or, equivalentally, a *piece of code*) that we can evaluate 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. 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.

Definition 1 (Dual numbers) Dual numbers $\mathbb{D}$ are a commutative ring (over $\mathbb{R}$) 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 $a + 0\epsilon \in \mathbb{D}$.

1. Differentiating polynomials

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

Theorem 1 (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

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

∎

Example 1 (differentiating polynomial) Consider computing $p'(2)$ where

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

We can use Dual numbers to differentiating, avoiding expanding in monomials or rules of differentiating:

$$p(2+\epsilon) = (1+\epsilon)\epsilon + (2+\epsilon)^2 = \epsilon + 4 + 4\epsilon = 4 + \underbrace{5}_{p'(2)}\epsilon$$

2. Differentiating other functions

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:

$$\begin{aligned} f(a+b\epsilon) &= \sum_{k=0}^{\infty} f_k (a+b\epsilon)^k = f_0 + \sum_{k=1}^{\infty} f_k (a^k + ka^{k-1}b\epsilon) = \sum_{k=0}^{\infty} f_k a^k + \sum_k^{\infty} \\ &= f(a) + bf'(a)\epsilon \end{aligned}$$

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

Definition 2 (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:

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

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 1 (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:

$$
\begin{aligned}
q(a+b\epsilon) &= q(a) + bq'(a)\epsilon \\
r(a+b\epsilon) &= r(a) + br'(a)\epsilon
\end{aligned}
$$

Proof For q it follows immediately:

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

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 2 (differentiating non-polynomial)

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

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

and therefore we deduce that

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

3. Implementation as a special type

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

In [1]:
```
# 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)
```

Out[1]:
```
exp (generic function with 15 methods)
```

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

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

Out[2]:
```
(1, 0.3333333333333333)
```

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:

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

Out[3]:
```
8.881784197001252e-16
```

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 $1 + \ldots + x^n$:

In [4]:
```
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
```

Out[4]:
```
1.2345678999999998
```

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

In [5]:
```
sum((1:10) .* 0.1 .^(0:9))
```

Out[5]:
```
1.2345678999999998
```

Finally, we can try the more complicated example:

In [6]:
```
f = x -> exp(x^2 + exp(x))
f(1 + ϵ)
```

Out[6]:
```
Dual{Float64}(41.193555674716116, 194.362805189629)
```

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

II.1 Structured Matrices

We have seen how algebraic operations (`+` , `-` , `*` , `/`) are defined exactly in terms of rounding ($\oplus$, $\ominus$, $\otimes$, $\oslash$) for floating point numbers. Now we see how this allows us to do (approximate) linear algebra operations on matrices.

A matrix can be stored in different formats. Here 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.

In the next chapter we consider more complicated orthogonal matrices.

In [1]:
```
# LinearAlgebra contains routines for doing linear algebra
# BenchmarkTools is a package for reliable timing
using LinearAlgebra, Plots, BenchmarkTools, Test
```

---

1. Dense vectors and matrices

A `Vector` of a primitive type (like `Int` or `Float64`) is stored consecutively in memory: that is, a vector consists of a memory address (a *pointer*) to the first entry and a length. E.g. if we have a `Vector{Int8}` of length `n` then it is stored as `8n` bits (`n` bytes) in a row. That is, if the memory address of the first entry is `k` and the type is `T` , the memory address of the second entry is `k + sizeof(T)` .

Remark (advanced) We can actually experiment with this (NEVER DO THIS IN PRACTICE!!), beginning with an 8-bit type:

In [2]:
```
a = Int8[2, 4, 5]
p = pointer(a) # pointer(a) returns memory address of the first entry, which
# We can think of a pointer as simply a UInt64 alongside a Type to interpret
```

Out[2]:
```
Ptr{Int8} @0x000000014539fd78
```

We can see what's stored at a pointer as follows:

In [3]:
```
Base.unsafe_load(p) # loads data at `p`. Knows its an `Int8` because of type
```

Out[3]:
```
2
```

Adding an integer to a pointer gives a new pointer with the address incremented:

In [4]:
```
p + 1 # memory address of next entry, which is 1 more than first
```

Out[4]:
```
Ptr{Int8} @0x000000014539fd79
```

We see that this gives us the next entry:

In [5]:
```
Base.unsafe_load(p) # loads data at `p+1`, which is second entry of the vect
```

Out[5]:
```
2
```

For other types we need to increment the address by the size of the type:

In [6]:
```
a = [2.0, 1.3, 1.4]
p = pointer(a)
Base.unsafe_load(p + 8) # sizeof(Float64) == 8
```

Out[6]:
```
1.3
```

Why not do this in practice? It's unsafe because there's nothing stopping us from going past the end of an array:

In [7]:
```
Base.unsafe_load(p + 3 * 8) # whatever bits happened to be next in memory, u
```

Out[7]:
```
2.405888678e-314
```

This may even crash Julia! (I got lucky that it didn't when producing the notes.)

A `Matrix` is stored consecutively in memory, going down column-by- column (*column-major*). That is,

In [8]:
```
A = [1 2;
     3 4;
     5 6]
```

Out[8]:
```
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
```

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

In [9]:
```
vec(A)
```

Out[9]:
```
6-element Vector{Int64}:
 1
 3
 5
 2
 4
 6
```

which in this case would be stored using in `8 * 6 = 48` consecutive memory addresses. That is, a matrix is a pointer to the first entry alongside two integers dictating the row and column sizes.

Remark (advanced) Note that transposing `A` is done lazyily and so `transpose(A)` (which is equivalent to the adjoint/conjugate-transpose `A'` when the entries are real), is just a special type with a single field: `transpose(A).parent == A`. This is equivalent to *row-major* format, where the next address in memory of `transpose(A)` corresponds to moving along the row.

Matrix-vector multiplication works as expected:

In [10]:
```
x = [7, 8]
A * x
```

Out[10]:
```
3-element Vector{Int64}:
 23
 53
 83
```

Note there are two ways this can be implemented:

Algorithm 1 (matrix-vector multiplication by rows) For a ring R (typically $\mathbb{R}$ or $\mathbb{C}$), $A \in R^{m\times n}$ and $\mathbf{x} \in R^n$ we have

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

In code this can be implemented for any types that support `*` and `+` as follows:

In [11]:
```
function mul_rows(A, x)
    m,n = size(A)
    # promote_type type finds a type that is compatible with both types, elt
    T = promote_type(eltype(x), eltype(A))
    c = zeros(T, m) # the returned vector, begins of all zeros
    for k = 1:m, j = 1:n
        c[k] += A[k, j] * x[j] # equivalent to c[k] = c[k] + A[k, j] * x[j]
    end
    c
end
```

Out[11]: `mul_rows (generic function with 1 method)`

Algorithm 2 (matrix-vector multiplication by columns) For a ring R (typically $\mathbb{R}$ or $\mathbb{C}$), $A \in R^{m \times n}$ and $\mathbf{x} \in R^n$ we have

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

where $\mathbf{a}_j := A\mathbf{e}_j \in R^m$ (that is, the j-th column of A). In code this can be implemented for any types that support `*` and `+` as follows:

In [12]:
```
function mul_cols(A, x)
    m,n = size(A)
    # promote_type type finds a type that is compatible with both types, elt
    T = promote_type(eltype(x),eltype(A))
    c = zeros(T, m) # the returned vector, begins of all zeros
    for j = 1:n, k = 1:m
        c[k] += A[k, j] * x[j] # equivalent to c[k] = c[k] + A[k, j] * x[j]
    end
    c
end
```

Out[12]: `mul_cols (generic function with 1 method)`

Both implementations match exactly for integer inputs:

In [13]:
```
mul_rows(A, x), mul_cols(A, x) # also matches `A*x`
```

Out[13]: `([23, 53, 83], [23, 53, 83])`

Either implementation will be $O(mn)$ operations. However, the implementation `mul_cols` 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`:

In [14]:
```
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_cols(A,x)
@btime A*x; # built-in, high performance implementation. USE THIS in practic
```

```
  1.667 ms (1 allocation: 7.94 KiB)
  755.646 μs (1 allocation: 7.94 KiB)
  220.887 μs (1 allocation: 7.94 KiB)
```

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_cols(A,x)`, that is, some inputs will differ in how the computations are rounded.

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

In [15]:
```
A = [1.4 0.4;
     2.0 1/2]
A * [1, -1] # First entry has round-off error, but 2nd entry is exact
```

Out[15]:
```
2-element Vector{Float64}:
 0.9999999999999999
 1.5
```

And integer arithmetic will be subject to overflow:

In [16]:
```
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
```

Out[16]:
```
2-element Vector{Int8}:
 -128
 -128
```

Solving a linear system is done using `\`:

In [17]:
```
A = [1 2 3;
     1 2 4;
     3 7 8]
b = [10; 11; 12]
A \ b
```

Out[17]:
```
3-element Vector{Float64}:
  41.000000000000036
 -17.000000000000014
   1.0
```

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 matrices that are easy to invert, e.g., a product of triangular matrices or a product of an orthogonal and triangular matrix.

2. Triangular matrices

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

In [18]:
```
A = [1 2 3;
     4 5 6;
     7 8 9]
U = UpperTriangular(A)
```

Out[18]:
```
3×3 UpperTriangular{Int64, Matrix{Int64}}:
 1  2  3
 ·  5  6
 ·  ·  9
```

We can see that `U` is storing all the entries of `A` in a field called `data`:

In [19]:
```
U.data
```

Out[19]:
```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

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

In [20]:
```
L = LowerTriangular(A)
```

Out[20]:
```
3×3 LowerTriangular{Int64, Matrix{Int64}}:
 1  ·  ·
 4  5  ·
 7  8  9
```

If we know a matrix is triangular we can do matrix-vector multiplication in roughly half the number of operations by skipping over the entries we know are zero:

Algorithm 3 (upper-triangular matrix-vector multiplication by columns)

In [21]:
```
function mul_cols(U::UpperTriangular, x)
    n = size(U,1)
    # promote_type type finds a type that is compatible with both types, elt
    T = promote_type(eltype(x),eltype(U))
    b = zeros(T, n) # the returned vector, begins of all zeros
    for j = 1:n, k = 1:j # k = 1:j instead of 1:m since we know U[k,j] = 0 i
        b[k] += U[k, j] * x[j]
    end
    b
end

x = [10, 11, 12]
# matches built-in *
@test mul_cols(U, x) == U*x
```

Out[21]:
```
Test Passed
```

Moreover, we can easily invert matrices. Consider a simple 3×3 example, which can be solved with `\`:

In [22]:
```
b = [5, 6, 7]
x = U \ b # Excercise: why does this return a float vector?
```

Out[22]:
```
3-element Vector{Float64}:
 2.1333333333333333
 0.2666666666666666
 0.7777777777777778
```

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:

In [23]:
```
b[3] / U[3,3]
```

Out[23]:
```
0.7777777777777778
```

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:

In [24]:
```
(b[2] - U[2,3]*x[3])/U[2,2]
```

Out[24]:
```
0.2666666666666666
```

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

In [25]:
```
(b[1] - U[1,2]*x[2] - U[1,3]*x[3])/U[1,1]
```

Out[25]:
```
2.1333333333333333
```

More generally, we can solve the upper-triangular system using *back-substitution*:

Algorithm 4 (back-substitution) Let $\mathbb{F}$ be a field (typically $\mathbb{R}$ or $\mathbb{C}$). Suppose $U \in \mathbb{F}^{n\times n}$ is upper-triangular and invertible. Then for $\mathbf{b} \in \mathbb{F}^n$ the solution $\mathbf{x} \in \mathbb{F}^n$ to $U\mathbf{x} = \mathbf{b}$, that is,

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

is given by computing $x_n, x_{n-1}, \ldots, x_1$ via:

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

In code this can be implemented for any types that support `*`, `+` and `/` as follows:

In [26]:
```
# ldiv(U, b) is our implementation of U\b
function ldiv(U::UpperTriangular, b)
    n = size(U,1)

```

```
    if length(b) != n
        error("The system is not compatible")
    end

    x = zeros(n)  # the solution vector

    for k = n:-1:1  # start with k=n, then k=n-1, ...
        r = b[k]  # dummy variable
        for j = k+1:n
            r -= U[k,j]*x[j] # equivalent to r = r - U[k,j]*x[j]
        end
        # after this for loop, r = b[k] - ∑_{j=k+1}^n U[k,j]x[j]
        x[k] = r/U[k,k]
    end
    x
end

@test ldiv(U, x) ≈ U\x
```

Out[26]: **Test Passed**

The problem sheet will explore implementing multiplication and forward substitution for 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 1 (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 2 (Diagonal) *Diagonal matrices* are square matrices with bandwidths $l = u = 0$.

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

In [27]:
```
x = [1,2,3]
D = Diagonal(x) # the type Diagonal has a single field: D.diag
```

Out[27]:
```
3×3 Diagonal{Int64, Vector{Int64}}:
 1  ·  ·
 ·  2  ·
 ·  ·  3
```

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

Bidiagonal

Definition 3 (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:

In [28]:
```
L = Bidiagonal([1,2,3], [4,5], :L) # the type Bidiagonal has three fields: L
```

Out[28]:
```
3×3 Bidiagonal{Int64, Vector{Int64}}:
 1  ·  ·
 4  2  ·
 ·  5  3
```

In [29]:
```
Bidiagonal([1,2,3], [4,5], :U)
```

Out[29]:
```
3×3 Bidiagonal{Int64, Vector{Int64}}:
 1  4  ·
 ·  2  5
 ·  ·  3
```

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 4 (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:

In [30]:
```
T = Tridiagonal([1,2], [3,4,5], [6,7]) # The type Tridiagonal has three fiel
```

Out[30]:
```
3×3 Tridiagonal{Int64, Vector{Int64}}:
 3  6  ·
 1  4  7
 ·  2  5
```

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

II.2 Orthogonal and Unitary Matrices

A very important class of matrices are *orthogonal* and *unitary* matrices:

Definition 1 (orthogonal/unitary matrix) A square real matrix is *orthogonal* if its inverse is its transpose:

$$O(n) = \{Q \in \mathbb{R}^{n\times n} : Q^\top Q = I\}$$

A square complex matrix is *unitary* if its inverse is its adjoint:

$$U(n) = \{Q \in \mathbb{C}^{n\times n} : Q^\star Q = I\}.$$

Here the adjoint is the same as the conjugate-transpose: $Q^\star := \overline{Q}^\top$.

Note that $O(n) \subset U(n)$ as for real matrices $Q^\star = Q^\top$. Because in either case $Q^{-1} = Q^\star$ we also have $QQ^\star = I$ (which for real matrices is $QQ^\top = I$). These matrices are particularly important for numerical linear algebra for a number of reasons (we'll explore these properties in the problem sheets):

1. They are norm-preserving: for any vector $\mathbf{x} \in \mathbb{C}^n$ we have

$\|Q\mathbf{x}\| = \|\mathbf{x}\|$ where $\|\mathbf{x}\|^2 := \sum_{k=1}^n x_k^2$ (i.e. the 2-norm). 2. All eigenvalues have absolute value equal to 1 3. For $Q \in O(n)$, $\det Q = \pm 1$. 2. They are trivially invertible (just take the transpose). 3. They are generally "stable": errors are controlled. 4. They are *normal matrices*: they commute with their adjoint ($QQ^\star = QQ^\star$). See Chapter C for why this is important.

On a computer there are multiple ways of representing orthogonal/unitary matrices, and it is almost never to store a dense matrix storing the entries. We shall therefore investigate three classes:

1. *Permutation*: A permutation matrix permutes the rows of a vector and is a representation of the symmetric group.
2. *Rotations*: The simple rotations are also known as 2×2 special orthogonal matrices ($SO(2)$) and correspond to rotations in 2D.
3. *Reflections*: Reflections are $n \times n$ orthogonal matrices that have simple definitions in terms of a single vector.

We remark a very similar concept are rectangular matrices with orthogonal columns, e.g.

$$U = [\mathbf{u}_1 | \cdots | \mathbf{u}_n] \in \mathbb{R}^{m\times n}$$

where $m \geq n$ such that $U^\top U = I_n$ (the $n \times n$ identity matrix). In this case we must have $UU^\top \neq I_m$ as the rank of U is n. These will play an important role in the Singular Value Decomposition.

1. 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 $\{1, 2, \ldots, n\}$ 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 $\{\sigma_1, \ldots, \sigma_n\} = \{1, 2, \ldots, n\}$ (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 $\boldsymbol{\sigma} = [\sigma_1, \ldots, \sigma_n]$. 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 1 (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:

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

Out[1]:
```
5-element Vector{Int64}:
  6
  9
  7
 10
  8
```

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:

In [2]:
```
σ⁻¹ = invperm(σ) # note that ⁻¹ are just unicode characters in the variable
```

```
Out[2]: 5-element Vector{Int64}:
 1
 3
 5
 2
 4
```

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

```
In [3]: v[σ][σ⁻¹] # permuting by σ and then σⁱ gets us back
```

```
Out[3]: 5-element Vector{Int64}:
  6
  7
  8
  9
 10
```

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 2 (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 2 (5×5 permutation matrix) We can construct the permutation representation for σ as above as follows:

```
In [4]: P = I(5)[σ,:]
```

```
Out[4]: 5×5 Matrix{Bool}:
 1  0  0  0  0
 0  0  0  1  0
 0  1  0  0  0
 0  0  0  0  1
 0  0  1  0  0
```

And indeed, we see its action is as expected:

```
In [5]: P * v
```

```
Out[5]: 5-element Vector{Int64}:
  6
  9
  7
 10
  8
```

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 1 (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 $P_\sigma^{-1} = P_\sigma^\top$.

∎

2. Rotations

We begin with a general definition:

Definition 3 (Special Orthogonal and Rotations) *Special Orthogonal Matrices* are

$$SO(n) := \{Q \in O(n) | \det Q = 1\}$$

And (simple) *rotations* are $SO(2)$.

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

Definition 4 (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 via the function `atan(y,x)`.

We show $SO(2)$ are exactly equivalent to standard rotations:

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

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

We have $Q \in SO(2)$ iff $Q = Q_\theta$ for some $\theta \in \mathbb{R}$.

Proof

We will write $c = \cos\theta$ and $s = \sin\theta$. Then we have

$$Q_\theta^\top Q_\theta = \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} c & -s \\ s & c \end{pmatrix} = \begin{pmatrix} c^2 + s^2 & 0 \\ 0 & c^2 + s^2 \end{pmatrix} = I$$

and $\det Q_\theta = c^2 + s^2 = 1$ hence $Q_\theta \in SO(2)$.

Now suppose $Q = [\mathbf{q}_1, \mathbf{q}_2] \in SO(2)$ where we know its columns have norm 1 $\|\mathbf{q}_k\| = 1$ and are orthogonal. Write $\mathbf{q}_1 = [c, s]$ where we know $c = \cos\theta$ and $s = \sin\theta$ for $\theta = \operatorname{atan}(s, c)$. Since $\mathbf{q}_1 \cdot \mathbf{q}_2 = 0$ we can deduce $\mathbf{q}_2 = \pm[-s, c]$. The sign is positive as $\det Q = \pm(c^2 + s^2) = \pm 1$.

∎

We can rotate an arbitrary vector in $\mathbb{R}^2$ to the unit axis using rotations, which are useful in linear algebra decompositions. Interestingly it only requires basic algebraic functions (no trigonometric functions):

Proposition 3 (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 ($Q \in SO(2)$) 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. This follows immediately:

$$Q^\top Q = \frac{1}{a^2+b^2} \begin{bmatrix} a^2+b^2 & 0 \\ 0 & a^2+b^2 \end{bmatrix} = I$$

and $\det Q = 1$.

∎

3. Reflections

In addition to rotations, another type of orthogonal/unitary matrix are reflections:

Definition 5 (reflection matrix) Given a unit vector $\mathbf{v} \in \mathbb{C}^n$ (satisfying $\|\mathbf{v}\| = 1$), the *reflection matrix*

$$Q_{\mathbf{v}} := I - 2\mathbf{v}\mathbf{v}^\star$$

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

Proposition 4 (Householder properties) $Q_{\mathbf{v}}$ satisfies:

1. $Q_{\mathbf{v}} = Q_{\mathbf{v}}^\star$ (Symmetry)
2. $Q_{\mathbf{v}}^\star Q_{\mathbf{v}} = I$ (Orthogonality $Q_{\mathbf{v}} \in U(n)$)
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. $\det Q_{\mathbf{v}} = -1$ ($Q_{\mathbf{v}} \notin SO(n)$)

Proof

Property 1 follows immediately. Property 2 follows from

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

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 $\mathbf{w}^\star \mathbf{v} = 0$, 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 3 (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:

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

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 $\mathbb{C}^n$, not just $\mathbb{R}^2$:

Definition 6 (Householder reflection, real case) For a given vector $\mathbf{x} \in \mathbb{R}^n$, 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 1 (Householder reflection maps to axis) For $\mathbf{x} \in \mathbb{R}^n$,

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

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:

In [6]:
```
h = 10.0^(-10)
x = [1,h]
y = -norm(x)*[1,0] + x
w = y/norm(y)
Q = I - 2w*w'
Q*x
```

Out[6]:
```
2-element Vector{Float64}:
  1.0
 -1.0e-10
```

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

In [7]:
```
h = 0
x = [1,h]
y = -norm(x)*[1,0] + x
w = y/norm(y)
Q = I - 2w*w'
Q*x
```

Out[7]:
```
2-element Vector{Float64}:
 NaN
 NaN
```

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:

In [8]:
```
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
```

Out[8]:
```
2-element Vector{Float64}:
 -1.0
  0.0
```

We can extend this definition for complexes:

Definition 7 (Householder reflection, complex case) For a given vector $\mathbf{x} \in \mathbb{C}^n$, define the Householder reflection as

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

for $\mathbf{y} = \operatorname{csign}(x_1)\|\mathbf{x}\|\mathbf{e}_1 + \mathbf{x}$ and $\mathbf{w} = \frac{\mathbf{y}}{\|\mathbf{y}\|}$, for $\operatorname{csign}(z) = \mathrm{e}^{\mathrm{i}\arg z}$.

Lemma 2 (Householder reflection maps to axis, complex case) For $\mathbf{x} \in \mathbb{C}^n$,

$$Q_{\mathbf{x}}^{\mathrm{H}}\mathbf{x} = -\operatorname{csign}(x_1)\|\mathbf{x}\|\mathbf{e}_1$$

Proof Denote $\alpha := \operatorname{csign}(x_1)$. Note that $\bar{\alpha}x_1 = \mathrm{e}^{-\mathrm{i}\arg x_1}x_1 = |x_1|$. Now we have

$$\begin{aligned}
\|\mathbf{y}\|^2 &= (\alpha\|\mathbf{x}\|\mathbf{e}_1 + \mathbf{x})^\star(\alpha\|\mathbf{x}\|\mathbf{e}_1 + \mathbf{x}) = |\alpha|\|\mathbf{x}\|^2 + \|\mathbf{x}\|\alpha\bar{x}_1 + \bar{\alpha}x_1\|\mathbf{x}\| + \|\mathbf{x}\|^2 \\
&= 2\|\mathbf{x}\|^2 + 2|x_1|\|\mathbf{x}\| \\
\mathbf{y}^\star\mathbf{x} &= \bar{\alpha}x_1\|\mathbf{x}\| + \|\mathbf{x}\|^2 = \|\mathbf{x}\|^2 + |x_1|\|\mathbf{x}\|
\end{aligned}$$

Therefore:

$$Q_{\mathbf{x}}^{\mathrm{H}}\mathbf{x} = (I - 2\mathbf{w}\mathbf{w}^\star)\mathbf{x} = \mathbf{x} - 2\frac{\mathbf{y}}{\|\mathbf{y}\|^2}(\|\mathbf{x}\|^2 + |x_1|\|\mathbf{x}\|) = \mathbf{x} - \mathbf{y} = -\alpha\|\mathbf{x}\|\mathbf{e}_1.$$

∎

II.3 QR factorisation

Let $A \in \mathbb{C}^{m\times n}$ be a rectangular or square matrix such that $m \geq n$ (i.e. more rows then columns). In this chapter we consider two closely related factorisations:

1. The *QR factorisation*

$$A = QR = \underbrace{[\,\mathbf{q}_1|\cdots|\mathbf{q}_m\,]}_{Q\in U(m)} \underbrace{\begin{bmatrix} \times & \cdots & \times \\ & \ddots & \vdots \\ & & \times \\ & & 0 \\ & & \vdots \\ & & 0 \end{bmatrix}}_{R\in\mathbb{C}^{m\times n}}$$

where Q is unitary (i.e., $Q \in U(m)$, satisfying $Q^\star Q = I$, with columns $\mathbf{q}_j \in \mathbb{C}^m$) and R is *right triangular*, which means it is only nonzero on or to the right of the diagonal ($r_{kj} = 0$ if $k > j$).

2. The *reduced QR factorisation*

$$A = \hat{Q}\hat{R} = \underbrace{[\,\mathbf{q}_1|\cdots|\mathbf{q}_n\,]}_{\hat{Q}\in\mathbb{C}^{m\times n}} \underbrace{\begin{bmatrix} \times & \cdots & \times \\ & \ddots & \vdots \\ & & \times \end{bmatrix}}_{\hat{R}\in\mathbb{C}^{n\times n}}$$

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

Note for a square matrix the reduced QR factorisation is equivalent to the QR factorisation, in which case R is *upper triangular*. The importance of these decomposition for square matrices is that their component pieces are easy to invert:

$$A = QR \qquad \Rightarrow \qquad A^{-1}\mathbf{b} = R^{-1}Q^\top\mathbf{b}$$

and we saw in the last two chapters 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

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

Note in the rectangular case the QR decomposition contains within it the reduced QR decomposition:

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

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 computing the QR decomposition using Householder reflections.

In [1]:
```
using LinearAlgebra, Plots, BenchmarkTools
```

1. QR and least squares

Here we consider rectangular matrices with more rows than columns. Given $A \in \mathbb{C}^{m\times n}$ and $\mathbf{b} \in \mathbb{C}^m$, least squares consists of finding a vector $\mathbf{x} \in \mathbb{C}^n$ that minimises the 2-norm: $\|A\mathbf{x} - \mathbf{b}\|$.

Theorem 1 (least squares via QR) Suppose $A \in \mathbb{C}^{m\times n}$ has full rank. Given a QR decomposition $A = QR$ then

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

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

Proof

The norm-preserving property (see PS4 Q3.1) of unitary matrices tells us

$$\|A\mathbf{x} - \mathbf{b}\| = \|QR\mathbf{x} - \mathbf{b}\| = \|Q(R\mathbf{x} - Q^\star\mathbf{b})\| = \|R\mathbf{x} - Q^\star\mathbf{b}\| = \left\| \begin{bmatrix} \hat{R} \\ \mathbf{0}_{m-n\times n} \end{bmatrix} \mathbf{x} - \begin{bmatrix} (\\ \mathbf{q} \\ \\ \\ (\end{bmatrix} \right.$$

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:

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

This norm is minimised if it is zero. Provided the column rank of A is full, $\hat{R}$ will be invertible (Exercise: why is this?).

■

Example 1 (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:

In [2]:
```
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 column

p₀,p₁,p₂ = R̂ \ Q̂'f
```

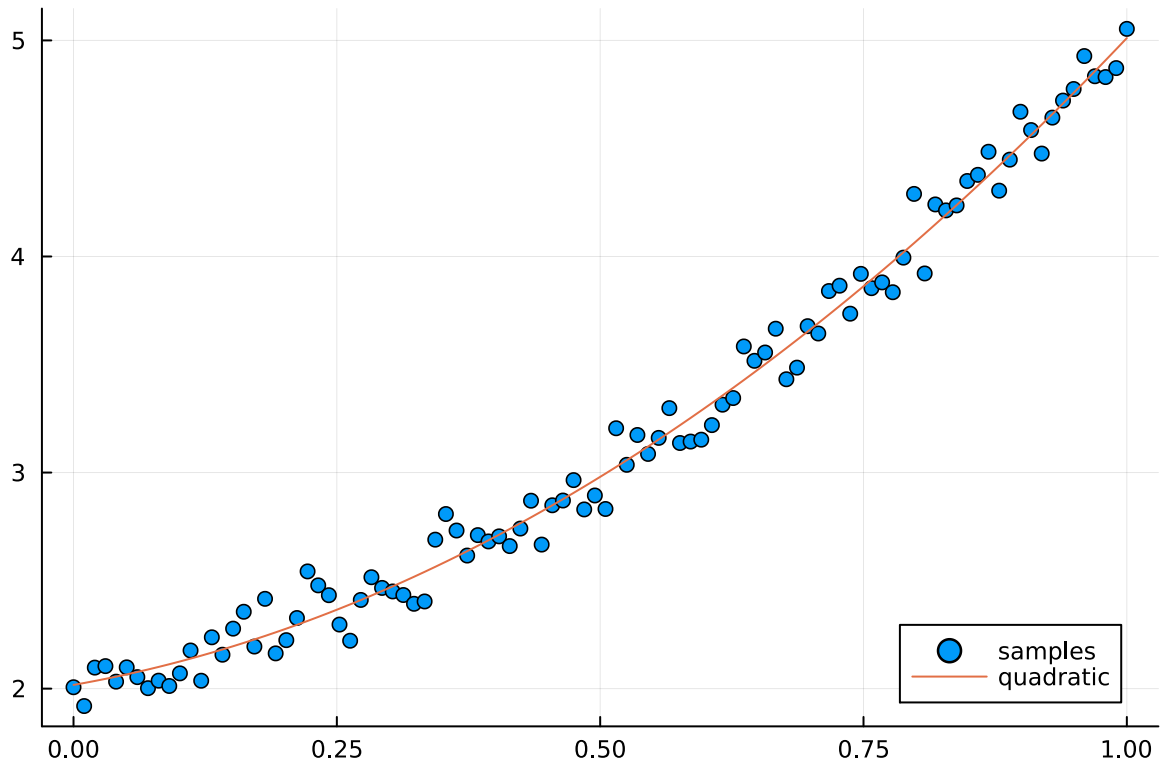
Out[2]:
```
3-element Vector{Float64}:
 2.0172265583556173
 0.857813720251557
 2.1358409286607394
```

We can visualise the fit:

In [3]:
```
p = x -> p₀ + p₁*x + p₂*x^2

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

Out[3]:



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

In [4]:
```
A \ f
```

Out[4]:
```
3-element Vector{Float64}:
 2.0172265583556177
 0.8578137202515559
 2.13584092866074
```

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

$$A = \begin{bmatrix} \mathbf{a}_1 | \cdots | \mathbf{a}_n \end{bmatrix}$$

where $\mathbf{a}_k \in \mathbb{C}^m$ and assume they are linearly independent (A has full column rank).

Proposition 1 (Column spaces match) Suppose $A = \hat{Q}\hat{R}$ where $\hat{Q} = [\mathbf{q}_1 | \ldots | \mathbf{q}_n]$ has orthogonal columns and $\hat{R}$ is upper-triangular, and A has full rank. Then the first j columns of $\hat{Q}$ span the same space as the first j columns of A:

$$\operatorname{span}(\mathbf{a}_1, \ldots, \mathbf{a}_j) = \operatorname{span}(\mathbf{q}_1, \ldots, \mathbf{q}_j).$$

Proof

Because A has full rank we know $\hat{R}$ is invertible, i.e. its diagonal entries do not vanish: $r_{jj} \neq 0$. If $\mathbf{v} \in \operatorname{span}(\mathbf{a}_1, \ldots, \mathbf{a}_j)$ we have for $\mathbf{c} \in \mathbb{C}^j$

$$\mathbf{v} = \left[\, \mathbf{a}_1 \middle| \cdots \middle| \mathbf{a}_j \,\right] \mathbf{c} = \left[\, \mathbf{q}_1 \middle| \cdots \middle| \mathbf{q}_j \,\right] \hat{R}[1:j, 1:j]\mathbf{c} \in \operatorname{span}(\mathbf{q}_1, \ldots, \mathbf{q}_j)$$

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

$$\mathbf{w} = \left[\, \mathbf{q}_1 \middle| \cdots \middle| \mathbf{q}_j \,\right] \mathbf{d} = \left[\, \mathbf{a}_1 \middle| \cdots \middle| \mathbf{a}_j \,\right] \hat{R}[1:j, 1:j]^{-1}\mathbf{d} \in \operatorname{span}(\mathbf{a}_1, \ldots, \mathbf{a}_j).$$

∎

It is possible to find $\hat{Q}$ and $\hat{R}$ the using the *Gram–Schmidt algorithm*. We construct it column-by-column:

Algorithm 1 (Gram–Schmidt) For $j = 1, 2, \ldots, n$ define

$$\begin{aligned} \mathbf{v}_j &:= \mathbf{a}_j - \sum_{k=1}^{j-1} \underbrace{\mathbf{q}_k^\star \mathbf{a}_j}_{r_{kj}} \mathbf{q}_k \\ r_{jj} &:= \|\mathbf{v}_j\| \\ \mathbf{q}_j &:= \frac{\mathbf{v}_j}{r_{jj}} \end{aligned}$$

Theorem 2 (Gram–Schmidt and reduced QR) Define $\mathbf{q}_j$ and r_{kj} as in Algorithm 1 (with $r_{kj} = 0$ if $k > j$). Then a reduced QR decomposition is given by:

$$A = \underbrace{\left[\, \mathbf{q}_1 \middle| \cdots \middle| \mathbf{q}_n \,\right]}_{\hat{Q} \in \mathbb{C}^{m \times n}} \underbrace{\begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \end{bmatrix}}_{\hat{R} \in \mathbb{C}^{n \times n}}$$

Proof

We first show that $\hat{Q}$ has orthogonal columns. Assume that $\mathbf{q}_\ell^\star \mathbf{q}_k = \delta_{\ell k}$ for $k, \ell < j$. For $\ell < j$ we then have

$$\mathbf{q}_\ell^\star \mathbf{v}_j = \mathbf{q}_\ell^\star \mathbf{a}_j - \sum_{k=1}^{j-1} \mathbf{q}_\ell^\star \mathbf{q}_k \mathbf{q}_k^\star \mathbf{a}_j = 0$$

hence $\mathbf{q}_\ell^\star \mathbf{q}_j = 0$ and indeed $\hat{Q}$ has orthogonal columns. Further: from the definition of $\mathbf{v}_j$ we find

$$\mathbf{a}_j = \mathbf{v}_j + \sum_{k=1}^{j-1} r_{kj} \mathbf{q}_k = \sum_{k=1}^{j} r_{kj} \mathbf{q}_k = \hat{Q}\hat{R}\mathbf{e}_j$$

∎

Gram–Schmidt in action

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

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

Out[5]:
```
5×4 Matrix{Float64}:
 -0.70943    -0.454043   0.41333    0.344949
 -0.0439827  -0.275512  -0.320392  -0.138942
 -0.360728    0.548527  -0.493729   0.56686
 -0.419119    0.592301   0.369346  -0.541286
 -0.434729   -0.257365  -0.588492  -0.497378
```

The first column of `\hat Q` is indeed a normalised first column of `A`:

In [6]:
```
R = zeros(n,n)
Q = zeros(m,n)
R[1,1] = norm(A[:,1])
Q[:,1] = A[:,1]/R[1,1]
```

Out[6]:
```
5-element Vector{Float64}:
 0.7094298199902744
 0.043982737231058715
 0.36072796028194515
 0.4191187281458683
 0.4347294327768107
```

We now determine the next entries as

In [7]:
```
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]
```

Out[7]:
```
5-element Vector{Float64}:
 -0.4540426563561427
 -0.27551156613455136
  0.5485266393297886
  0.5923009397834829
 -0.2573650438818667
```

And the third column is then:

In [8]:
```
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]
```

Out[8]:
```
5-element Vector{Float64}:
 -0.41332990163830907
  0.3203921940446482
  0.49372888242968505
 -0.3693457774967121
  0.5884919045382065
```

(Note the signs may not necessarily match.)

We can clean this up as a simple algorithm:

In [9]:
```
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)
```

Out[9]:
```
7.850462293418876e-17
```

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:

In [10]:
```
A = randn(300,300)
Q,R = gramschmidt(A)
norm(Q'Q-I)
```

Out[10]:
```
1.6958205615505476e-12
```

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

$$Q_1 A = \begin{bmatrix} \times & \times & \cdots & \times \\ & \times & \cdots & \times \\ & \vdots & \ddots & \vdots \\ & \times & \cdots & \times \end{bmatrix} = \begin{bmatrix} \alpha & \mathbf{w}^\top \\ & A_2 \end{bmatrix}$$

where

$$\alpha := -\mathrm{csign}(a_{11})\|\mathbf{a}_1\|, \mathbf{w} = (Q_1 A)[1, 2:n] \qquad \text{and} \qquad A_2 = (Q_1 A)[2:m, 2:n],$$

$\mathrm{csign}(z) := \mathrm{e}^{\mathrm{i}\arg z}$. That is, we have made the first column triangular. In terms of an algorithm, we then introduce zeros into the first column of A_2, leaving an A_3, and so-on. But we can wrap this iterative algorithm into a simple proof by induction:

Theorem 3 (QR) Every matrix $A \in \mathbb{C}^{m\times n}$ has a QR factorisation:

$$A = QR$$

where $Q \in U(m)$ and $R \in \mathbb{C}^{m\times n}$ is right triangular.

Proof

Assume $m \geq n$. If $A = [\mathbf{a}_1] \in \mathbb{C}^{m\times 1}$ then we have for the Householder reflection $Q_1 = Q^{\mathrm{H}}_{\mathbf{a}_1}$

$$Q_1 A = [\alpha \mathbf{e}_1]$$

which is right triangular, where $\alpha = -\mathrm{sign}(a_{11})\|\mathbf{a}_1\|$. In other words

$$A = \underbrace{Q_1}_{Q} \underbrace{[\alpha \mathbf{e}_1]}_{R}.$$

For $n > 1$, assume every matrix with less columns than n has a QR factorisation. For $A = [\mathbf{a}_1 | \dots | \mathbf{a}_n] \in \mathbb{C}^{m\times n}$, let $Q_1 = Q^{\mathrm{H}}_{\mathbf{a}_1}$ so that

$$Q_1 A = \begin{bmatrix} \alpha & \mathbf{w}^\top \\ & A_2 \end{bmatrix}$$

where $A_2 = (Q_1 A)[2:m, 2:n]$ and $\mathbf{w} = (Q_1 A)[1, 2:n]$. By assumption $A_2 = \tilde{Q}\tilde{R}$. Thus we have

$$A = Q_1 \begin{bmatrix} \alpha & \mathbf{w}^\top \\ & \tilde{Q}\tilde{R} \end{bmatrix}$$
$$= \underbrace{Q_1 \begin{bmatrix} 1 & \\ & \tilde{Q} \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} \alpha & \mathbf{w}^\top \\ & \tilde{R} \end{bmatrix}}_{R}.$$

∎

This proof by induction leads naturally to an iterative algorithm. Note that $\tilde{Q}$ is a product of all Householder reflections that come afterwards, that is, we can think of Q as:

$$Q = Q_1\tilde{Q}_2\tilde{Q}_3\cdots\tilde{Q}_n \qquad \text{for} \qquad \tilde{Q}_j = \begin{bmatrix} I_{j-1} & \\ & Q_j \end{bmatrix}$$

where Q_j is a single Householder reflection corresponding to the first column of A_j. This is stated cleanly in Julia code:

Algorithm 2 (QR via Householder) For $A \in \mathbb{C}^{m\times n}$ with $m \geq n$, the QR factorisation can be implemented as follows:

In [11]:
```
function householderreflection(x)
    y = copy(x)
    if x[1] == 0
        y[1] += norm(x)
    else # note sign(z) = exp(im*angle(z)) where `angle` is the argument of
        y[1] += sign(x[1])*norm(x)
    end
    w = y/norm(y)
    I - 2*w*w'
end
function householderqr(A)
    T = eltype(A)
    m,n = size(A)
    if n > m
        error("More columns than rows is not supported")
    end

    R = zeros(T, m, n)
    Q = Matrix(one(T)*I, m, m)
    Aⱼ = copy(A)

    for j = 1:n
        a₁ = Aⱼ[:,1] # first columns of Aⱼ
        Q₁ = householderreflection(a₁)
        Q₁Aⱼ = Q₁*Aⱼ
        α,w = Q₁Aⱼ[1,1],Q₁Aⱼ[1,2:end]
        Aⱼ₊₁ = Q₁Aⱼ[2:end,2:end]

        # populate returned data
        R[j,j] = α
        R[j,j+1:end] = w
```

```
        # following is equivalent to Q = Q*[I 0 ; 0 Qⱼ]
        Q[:,j:end] = Q[:,j:end]*Q₁

        Aⱼ = Aⱼ₊₁ # this is the "induction"
    end
    Q,R
end

m,n = 100,50
A = randn(m,n)
Q,R = householderqr(A)
@test Q'Q ≈ I
@test Q*R ≈ A
```

Out[11]: **Test Passed**

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

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

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

$$\tilde{Q}_2 = \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

so that

$$\tilde{Q}_2 Q_1 A = \begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & 0 & 0 \\ & 0 & -6 \end{bmatrix}$$

The final reflection is

$$\tilde{Q}_3 = \begin{bmatrix} I_{2\times 2} & \\ & I - \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} å1 & & \\ & 1 & \\ & & 0 & 1 \\ & & 1 & 0 \end{bmatrix}$$

giving us

$$\tilde{Q}_3 \tilde{Q}_2 Q_1 A = \underbrace{\begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & & -6 \\ & & 0 \end{bmatrix}}_{R}$$

That is,

$$A = Q_1 \tilde{Q}_2 \tilde{Q}_3 R = \underbrace{\frac{1}{3}\begin{bmatrix} -1 & 0 & 2 & 2 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & -1 & 2 \\ 2 & 0 & 2 & -1 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} -3 & -6 & -9 \\ & -15 & -18 \\ & & -6 \\ & & 0 \end{bmatrix}}_{R}$$

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