2. Representing numbers

Last time

- Logistics; theme of course
- “Finding good approximate solutions, fast”
- (as close as desired to unknown true result)
- Implemented bisection algorithm for finding solution of $f(x) = 0$ (“root”)

Goals for today

- Storing data during a program and plotting
- Representing numbers

How are values stored in the computer?

- Each value is stored as a pattern of **bits**
- **Bit**: *b*inary dig*it*; stores 0 or 1
- Same sequence of bits can represent different values
- **Type** specifies *how to interpret* storage location
- And how operations apply to the value, i.e. its **behaviour**

Booleans

- A **Boolean** value is either `true` or `false`
- Use `bitstring` function to see internal representation:

```
x = true
bitstring(x)
x == 1    # treated as an integer
```

- Stored in 1 byte $\equiv$ 8 bits
- Logical operations:
  - and (`&`), or (`|`) and not (`!`)
- Can pack efficiently using `BitArray`

Integers

- **Integers** $\mathbb{Z}$: positive and negative whole numbers
- There are an infinite number of them – how represent?
- Finite storage space $\Rightarrow$ represent *finite* set of numbers
- Binary representation with n bits: $b_{n-1}, b_{n-2}, \ldots, b_0$:

$$x = b_0 + 2b_1 + 2^2 b_2 + \cdots + 2^{n-1} b_{n-1} = \sum_{i=0}^{n-1} b_i 2^i$$

- `Int64`: 1 bit for sign; $n = 63$ bits for number

```
bitstring(10);  bitstring(-10)
```

- Uses "two's complement" (invert 1s and 0s)

Integers II

- Maximum representable value:

```
typemax(Int32) == 2^31 - 1
typemin(Int32) == -(2^31)
```

- Julia also has *arbitrary precision* integers: `BigInt`:

```
x = BigInt(2);  # or big(2)
x^2^2^2^2
```

- But **caution**: they can be **slow**!
- Note that `big(2^2^2^2^2)` is *incorrect* – why?
- See also `BitIntegers.jl` package

Rational numbers

- Rational numbers: fractions p/q with $p, q \in \mathbb{Z}$
- Represent as pair: (p, q)
- Julia:

```julia
x = 3 // 4
typeof(x)

y = big(x)
typeof(y)
```

Making a rational number type

- How could we define our own rational number **type**?
- Make a **composite type**: “box” containing **fields**

```
struct MyRational
    p::Int   # specifies type of field `p`
    q::Int
end

x = MyRational(3, 4)  # can say x *is a* rational number

x * x  # error!
```

Operations on types

- A type tells Julia how to **interpret** data
- Rational is pair of integers with new **behaviour**:

```
import Base: *

*(x::MyRational, y::MyRational) =
    MyRational(x.p * y.p, x.q * y.q)

x * x
```

- Can specify how to display resulting object:

```
Base.show(io::IO, x::MyRational)
```

Multiple dispatch

- In Julia, `*` is just a standard **(generic) function**
- **Generic function**”: Has various **methods**
- **Method**: version of function acting on different types:

```
methods(*)
```

- Overloading `*` for our type adds a new method
- Fundamental feature of Julia (different from most other languages):
- **Multiple dispatch**: choose correct method depending on types of all arguments

Fixed-point arithmetic

- Integers with a fixed position for binary point:
- $b_7 b_6 b_5 \cdot b_4 b_3 b_2 b_1 b_0$
- Now we have real numbers! Effectively $n/2^5$ with $n = 0, \ldots, 2^7 - 1$
- “Fixed-point arithmetic”: e.g. `FixedPointNumbers.jl` package
- But can’t represent a wide *range* of numbers

Representing reals: Floating-point arithmetic

- Again we can only represent a *finite set* of real numbers.
- Fixed-point numbers are not flexible enough.
- Instead, let binary point “float”, i.e. move around
- **Floating-point numbers**: set $\mathbb{F}$ of numbers of form

$$x = \pm 2^e (1 + f)$$

- e is integer **exponent**
- f is fractional part or **mantissa**, $f = \sum_{i=1}^{d} b_i 2^{-i}$

Equal and non-equal spacing of floats

- Note that $1 \leq (1 + f) < 2$
- Same exponent $\Rightarrow$ **equally spaced** in $[2^e, 2^{e+1})$
- But changing exponent *changes spacing*:

```
using Plots

x = Float16(1.0)
xs = [x]

for i in 1:10000
    x = nextfloat(x)
    push!(xs, x)
end

scatter(xs)
```

What are floats?

- There is *nothing* mysterious about floating-point numbers: *floating-point numbers are* just special rational numbers*!*
- They actually have powers of 2 as denominators ("dyadic numbers")s
- Standard method to **approximate** real numbers

Peeling apart floats in Julia

- `Float64` is standard format: IEEE double precision ("binary64")
- Sign: 1 bit; exponent: 11 bits; mantissa: $d = 52$ bits
- Peel it apart following the above description:

```
x = 0.1
s = bitstring(x)

mantissa_string = s[end-51:end]
f = parse(Int, mantissa_string, base=2) / (2^52)
y = 2.0^(exponent(x)) * (1 + f)
```

- Exponent: integer with shift ("bias") – 1023 for `Float64`

Machine epsilon

- The smallest number greater than 1 is $1 + 2^{-d}$
- 2^{-d} is called **machine epsilon** or $□_{mach}
- Julia:

```
eps(Float64)
eps(1.0)

nextfloat(1.0) - 1.0
```

Rounding

- Given a true real number x, we can (in principle) find the *nearest* floating-point number to it, $\mathrm{fl}(x)$
- This is called **rounding**
- We have the following bound for the **relative error**:

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{1}{2}\epsilon_{\text{mach}}$$

- Equivalently, $\mathrm{fl}(x) = x(1+\epsilon)$ with $|\epsilon| \leq \frac{1}{2}\epsilon_{\text{mach}}$

Floating-point arithmetic

- Operations `+`, `-`, `*`, `/`, `sqrt` on floats are **correctly rounded**
- I.e. "do the operation in infinite precision and then round"
- Only need a few extra **guard bits** to do this
- Elementary functions like sin and exp are **much harder** to round correctly
- Julia has **faithful rounding**: returns one of the two nearest floating-point numbers
- `@edit sin(3.1)`

BigFloatS

- Julia also has arbitrary-precision floats, `BigFloat`
- This uses the MPFR library

```
setprecision(BigFloat, 1000)
x = big"0.1"
```

- `big(0.1)` shows true value represented by `0.1`.
- `pi` or `π` is special: it can calculate its value to arbitrary precision:

```
typeof(π)
```

```
big(π)
```

Overflow and underflow

- Certain operations exceed possible range of representable values
- **Overflow**: A number is produced that is too large
- **Underflow**: A number is produced that is too large
- NB: Julia's `Int` types **do not** warn you when overflow occurs (for performance):

```
x = factorial(20)  # OK
x * 21  # wrong!
```

Overflow and NaN for floats

```
x = 1e305   # scientific notation for 10^(305)
x * 10000   # gives Inf
```

- Secial values `Inf` and `-Inf` (for “*inf*inity”) for results that are too large
- Another special value is `NaN` – *N*ot *a* *N*umber:

  ```
  0.0 / 0.0   # gives NaN
  ```

- Underflow gives `0.0`.
- There is `-0.0`!

```
-1 / Inf
```

Summary

- We can represent integers and rationals exactly up to some size
- Reals are represented approximately by **floating-point** numbers
- Just a special set of real numbers with well-defined operations
- There are (slow) arbitrary-precision integer and floating-point libraries