3. Representing real numbers: Floating-point arithmetic

Review of the previous class

- Data are stored as a sequence of bits
- Data types tell the computer how to operate on the bits
- New data types using `struct`

- Representations of
  - Booleans
  - Integers
  - Rationals

- Rationals lead to an explosion of memory requirements and processing time

Goals for today

- Look for a solution for representing **real numbers**
- Fixed-point arithmetic
- **Floating-point numbers**
- Rounding

Real numbers

- **Real numbers**
- Ordered, uncountable set of numbers

- Think of *infinite* decimal expansions

$$x = a_m \cdots a_1 a_0 \cdot a_{-1} a_{-2} \cdots a_{-n} \cdots$$

- i.e.

$$x = \sum_{i=-\infty}^{m} a_i b^i$$

- b is the **base** or **radix**

Collaborative exercise

Towards the reals

1. How could we represent a number of dollars *and cents* using just integers?
2. How should we add daily interest by multiplying by 1.0001?
3. How does this generalise to represent numbers of scientific interest?
4. What should we do if numbers become too big or too small?

Fixed-point arithmetic

- We could **fix** the position for the radix point:
- $a_3 a_2 a_1 a_0 \cdot a_{-1} a_{-2} a_{-3} a_{-4}$

Fixed-point arithmetic

- We could **fix** the position for the radix point:
- $a_3 a_2 a_1 a_0 \cdot a_{-1} a_{-2} a_{-3} a_{-4}$
- Now we have real numbers!
- With radix 2 we have $n/2^4$ with $n = 0, \dots, 2^8 - 1$

Fixed-point arithmetic

- We could **fix** the position for the radix point:
- $a_3 a_2 a_1 a_0 \cdot a_{-1} a_{-2} a_{-3} a_{-4}$

- Now we have real numbers!
- With radix 2 we have $n/2^4$ with $n = 0, \ldots, 2^8 - 1$

- We could represent this in a single byte
- **Fixed-point arithmetic**
- e.g. `FixedPointArithmetic.jl` Julia package

Rounding

- What do we do if operations return a **non-representable** number?
  - E.g. too many decimal places (or infinitely many)
  - Or too big or small

Rounding

- What do we do if operations return a **non-representable** number?
  - E.g. too many decimal places (or infinitely many)
  - Or too big or small

- We must **round**: choose a representable value to replace it

Rounding

- What do we do if operations return a **non-representable** number?
  - E.g. too many decimal places (or infinitely many)
  - Or too big or small
- We must **round**: choose a representable value to replace it
- But a fixed-point representation cannot represent a wide *range* of numbers

Collaborative exercise II

Representing wider ranges of numbers

1. Starting from fixed-point arithmetic, how could we represent a *wider range* of numbers?
   Think about scientific notation.

Representing reals: Floating-point arithmetic

- Again, we can represent only a *finite set* approximating the real numbers
- Fixed-point numbers are not *flexible* enough, since they do not represent a wide range

Representing reals: Floating-point arithmetic

- Again, we can represent only a *finite set* approximating the real numbers
- Fixed-point numbers are not *flexible* enough, since they do not represent a wide range

- Solution: *j*Fix** the *number of digits*, but let the binary point **float**, i.e. move around
- Similar to **significant figures** and **scientific notation**:

$$3.1415$$
$$31.415$$
$$314.15$$

Relative precision

- Note that we always have the same **relative** precision
- I.e. the ratio

$$\frac{\text{smallest increment detectable}}{\text{size of the number being represented}}$$

Relative precision

- Note that we always have the same **relative** precision
- I.e. the ratio

$$\frac{\text{smallest increment detectable}}{\text{size of the number being represented}}$$

- The downside is that we have less *absolute* accuracy for larger numbers

Floating-point numbers

- Extend fixed-point arithmetic with an extra **power of 2**

Floating-point numbers

- Extend fixed-point arithmetic with an extra **power of 2**
- We define a set $\mathbb{F}$ of numbers of the form

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

- $e =$ integer **exponent**; $f =$ fractional part (**mantissa**)

Floating-point numbers

- Extend fixed-point arithmetic with an extra **power of 2**
- We define a set $\mathbb{F}$ of numbers of the form

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

- $e =$ integer **exponent**; $f =$ fractional part (**mantissa**)
- $f = \sum_{i=1}^{p} b_i 2^{-i}$
- f has a binary expansion like $0 \cdot 101 \ldots$
- $f = \frac{n}{2^p}$ with n an integer

Floating-point numbers II

- We need to store
  - 1 bit for the sign
  - n_e bits for the exponent
  - p bits for the mantissa
- E.g. IEEE double precision (“binary64” = `Float64`):

$n_e = 11$ and $p = 52$

Floating-point numbers II

- We need to store
  - 1 bit for the sign
  - n_e bits for the exponent
  - p bits for the mantissa
- E.g. IEEE double precision ("binary64" = `Float64`):

$n_e = 11$ and $p = 52$

- Note that one bit of the mantissa is **implicit** and not stored
- So the **precision** is actually $\tilde{p} = p + 1$
- E.g. double precision has 53 bits of precision

Collaborative exercise III

Spacing of floats

1. How are floats spaced along the real line?
   Think about using a small precision, say 3, and start with exponent 0.

Equal and non-equal spacing of floats

- Note that $1 \leq (1 + f) < 2$
- The floats in this range ($e = 0$) are **equally spaced**
- With distance 2^{-p} between consecutive floats

Equal and non-equal spacing of floats

- Note that $1 \leq (1 + f) < 2$
- The floats in this range ($e = 0$) are **equally spaced**
- With distance 2^{-p} between consecutive floats

- What happens if we *change* the exponent?

Equal and non-equal spacing of floats

- Note that $1 \leq (1 + f) < 2$
- The floats in this range ($e = 0$) are **equally spaced**
- With distance 2^{-p} between consecutive floats

- What happens if we *change* the exponent?

- We multiply the picture by 2^e
- Floats with exponent e are **equally spaced** in $[2^e, 2^{e+1})$!

Equal and non-equal spacing of floats

- Note that $1 \leq (1 + f) < 2$
- The floats in this range ($e = 0$) are **equally spaced**
- With distance 2^{-p} between consecutive floats

- What happens if we *change* the exponent?

- We multiply the picture by 2^e
- Floats with exponent e are **equally spaced** in $[2^e, 2^{e+1})$!

- But floats with different exponents have *different spacings*!

Plotting floats with Julia

```julia
using Plots

x = Float16(1.0)
xs = [x]

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

scatter(xs)
```

Peeling apart floats in Julia

- `Float64` is standard format: IEEE double precision ("binary64")
- Sign: 1 bit; exponent: 11 bits; mantissa: $p = 52$ 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) / (2.0^52)
y = 2.0^(exponent(x)) * (1 + f)
```

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

Machine epsilon

- The smallest number greater than 1 is $1 + 2^{-p}$
- 2^{-p} is called **machine epsilon** or ϵ_{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

- Modern floating-point arithmetic is governed by the **IEEE-758 standard**
- It guarantees that `+`, `-`, `*`, `/` and `sqrt` are **correctly rounded**
- I.e. "do the operation in infinite precision and then round"

Floating-point arithmetic

- Modern floating-point arithmetic is governed by the **IEEE-758 standard**
- It guarantees that `+`, `-`, `*`, `/` and `sqrt` are **correctly rounded**
- I.e. “do the operation in infinite precision and then round”
- We need only 2 or 3 extra **guard bits** to do this

Floating-point arithmetic

- Modern floating-point arithmetic is governed by the **IEEE-758 standard**
- It guarantees that `+`, `-`, `*`, `/` and `sqrt` are **correctly rounded**
- I.e. "do the operation in infinite precision and then round"
- We need only 2 or 3 extra **guard bits** to do this
- Functions like sin and exp are **difficult** to round correctly

Floating-point arithmetic

- Modern floating-point arithmetic is governed by the **IEEE-758 standard**
- It guarantees that `+`, `-`, `*`, `/` and `sqrt` are **correctly rounded**
- I.e. “do the operation in infinite precision and then round”

- We need only 2 or 3 extra **guard bits** to do this

- Functions like sin and exp are **difficult** to round correctly

- Julia has **faithful rounding**: returns one of the two nearest floating-point numbers

How much is 0.1?

- The rules we have seen lead to some interesting effects

How much is 0.1?

- The rules we have seen lead to some interesting effects

- What is `0.1` (in Julia / any programming language)?
- Is it equal to the real number $0.1 = \frac{1}{10}$?

How much is 0.1?

- The rules we have seen lead to some interesting effects

- What is `0.1` (in Julia / any programming language)?
- Is it equal to the real number $0.1 = \frac{1}{10}$?

- What is `3.3 * 1.2`?

BigFloatS

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

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

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 the 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!
```

Special values: Infinity and `NaN` (not a number)

Special values: Infinity and `NaN` (not a number)

- Special values:
  - `Inf` and `-Inf` for "**infinity**", i.e. a result that is too large

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

  - `NaN` – *N*ot *a* *N*umber, for impossible operations

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

Special values: Infinity and `NaN` (not a number)

- Special values:
  - `Inf` and `-Inf` for "**infinity**", i.e. a result that is too large

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

  - `NaN` – *N*ot *a* *N*umber, for impossible operations

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

- Underflow gives `0.0`
- There is **negative zero**!:

```
-1 / Inf
```

Sub-normal numbers

- We want to represent numbers closer to 0 than is allowed by the above description

Sub-normal numbers

- We want to represent numbers closer to 0 than is allowed by the above description
- *Sub-normal* numbers have the smallest possible exponent
- We allow initial digits of the mantissa to be 0 to represent smaller numbers

Summary: Are floats mysterious?

- There is a myth that floating-point numbers are fuzzy or approximate or have some randomness
- This is **false**!

Summary: Are floats mysterious?

- There is a myth that floating-point numbers are fuzzy or approximate or have some randomness
- This is **false**!

Floating-point numbers are special rationals*!*

- They are **dyadic numbers**: rationals with power-of-2 denominators
- They behave in well-defined and predictable ways

Summary: Are floats mysterious?

- There is a myth that floating-point numbers are fuzzy or approximate or have some randomness
- This is **false**!

Floating-point numbers are special rationals*!*

- They are **dyadic numbers**: rationals with power-of-2 denominators
- They behave in well-defined and predictable ways

- Operations usually must **round** to the nearest float
- There are (slow) arbitrary-precision float libraries