2. Representing numbers

Summary of the previous class

- Logistics & introductions

- What is numerical analysis about?
- Approximation algorithms
  - Algorithms that find an approximate solution
  - As close as we want to the true solution!

- Example: Calculating $\sqrt{x}$

Outline for today's class

- Data storage
- Data types

- Representing numbers: Building towards real numbers

- Composite types in Julia

How are values stored in the computer?

- Data is stored in a computer's memory
- All data are stored as "numbers"

How are values stored in the computer?

- Data is stored in a computer's memory
- All data are stored as "numbers"

- A number is stored as a collection of **bits**
- **Bit**: *b*inary dig*it*; stores 0 or 1

How are values stored in the computer?

- Data is stored in a computer's memory
- All data are stored as "numbers"

- A number is stored as a collection of **bits**
- **Bit**: *b*inary dig*it*; stores 0 or 1

- The *same* bits can be used to represent *different* values using different **encodings**
- The program needs to tell the computer how to *use* the bits to represent different *types* of data

Data types

- How does a program tell the computer how to use the bits?

Data types

- How does a program tell the computer how to use the bits?

- The solution: **data types**

Data types

- How does a program tell the computer how to use the bits?

- The solution: **data types**
- A **data type** is an attribute or label associated to a variable
- It says **how the data behaves**

Data types

- How does a program tell the computer how to use the bits?

- The solution: **data types**
- A **data type** is an attribute or label associated to a variable
- It says **how the data behaves**

- E.g. to calculate $a + b$
  - We need to know if the data inside the variables a and b are integers or real numbers
  - This determines which version of + should be used if

Examples of numeric data types

- Some kinds of number we may want to represent:

Examples of numeric data types

- Some kinds of number we may want to represent:
- Fundamental numeric types:
  - Booleans: `true` / `false` or 0 / 1
  - Integers: -3, 17
  - Real numbers: $3.14159...$

Examples of numeric data types

- Some kinds of number we may want to represent:
- Fundamental numeric types:
  - Booleans: `true` / `false` or 0 / 1
  - Integers: -3, 17
  - Real numbers: $3.14159\ldots$
- Some other number types:
  - Rationals: $\frac{a}{b}$
  - Complex numbers: $a + ib$
  - Intervals: $[a, b]$

Booleans

- A **Boolean** value is either `true` or `false`
- This corresponds to a single bit of information

Booleans

- A **Boolean** value is either `true` or `false`
- This corresponds to a single bit of information

- But it is often stored in 1 byte $\equiv$ 8 bits
- Or even 8 bytes / 64 bits – **word size** of most modern computers

Booleans II

- We can use Julia's `bitstring` function to see the internal representation in **binary**

```
x = true
bitstring(x)
```

Booleans II

- We can use Julia's `bitstring` function to see the internal representation in **binary**

```
x = true
bitstring(x)
```

- A Boolean is often treated as an integer:

```
true == 1    # two equals signs for "equality"
```

Booleans II

- We can use Julia's `bitstring` function to see the internal representation in **binary**

```
x = true
bitstring(x)
```

- A Boolean is often treated as an integer:

```
true == 1    # two equals signs for "equality"
```

- But these are not the same object:

```
true === 1   # three equals signs for "identity"
```

Booleans III

- There are various logical operations available
  - and (`&`), or (`|`), not (`!`)

Booleans III

- There are various logical operations available
  - and (`&`), or (`|`), not (`!`)

- We can pack many Booleans efficiently using `BitArray`

Integers

- Let's look at the **integers**, $\mathbb{Z}$ ("Zahlen")
- Start by thinking about the *non-negative* integers, $\mathbb{Z}_{\geq 0}$
- How can we represent / store an integer on the computer?

Integers

- Let's look at the **integers**, $\mathbb{Z}$ ("Zahlen")
- Start by thinking about the *non-negative* integers, $\mathbb{Z}_{\geq 0}$
- How can we represent / store an integer on the computer?

- Difficulty: There are an *infinite* number of them!

Integers

- Let's look at the **integers**, $\mathbb{Z}$ ("Zahlen")
- Start by thinking about the *non-negative* integers, $\mathbb{Z}_{\geq 0}$
- How can we represent / store an integer on the computer?

- Difficulty: There are an *infinite* number of them!
- But we only have *finite* storage space
- So we can *only represent a finite subset* of the integers

Collaborative exercise

Thinking about integers

- Suppose we decide to restrict ourselves to n bits
  1. How many non-negative integers can we represent?
  2. What is the representation?
  3. What should we do to represent negative numbers?

Integers II

- For non-negative integers we use a **binary representation**
- Label the values on the n bits as $b_{n-1}, b_{n-2}, \ldots, b_0$

Integers II

- For non-negative integers we use a **binary representation**
- Label the values on the n bits as $b_{n-1}, b_{n-2}, \ldots, b_0$

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

Integers II

- For non-negative integers we use a **binary representation**
- Label the values on the n bits as $b_{n-1}, b_{n-2}, \ldots, b_0$

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

- There are 2^n possible bit patterns

- `Int8` has 8 bits, so there are $2^8 = 256$ bit patterns
- `Int64` has 8 bits, so there are $2^{64} \simeq 1.8 \times 10^{19}$ bit patterns

Integers II

- For non-negative integers we use a **binary representation**
- Label the values on the n bits as $b_{n-1}, b_{n-2}, \ldots, b_0$

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

- There are 2^n possible bit patterns

- `Int8` has 8 bits, so there are $2^8 = 256$ bit patterns
- `Int64` has 8 bits, so there are $2^{64} \simeq 1.8 \times 10^{19}$ bit patterns

Integers III

- A natural way to encode the sign is to reserve one bit for it
- E.g. $1 \leftrightarrow +$ and $0 \leftrightarrow -$

Integers III

- A natural way to encode the sign is to reserve one bit for it
- E.g. $1 \leftrightarrow +$ and $0 \leftrightarrow -$

- But this is *not* the usual way to do it; e.g. try

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

- We use "two's complement": 0s and 1s are *inverted*

Integers IV

- Julia has some useful pre-defined functions
- E.g. the largest and smallest representable values:

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

Integers IV

- Julia has some useful pre-defined functions
- E.g. the largest and smallest representable values:

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

- What happens if we add 1 to the maximum value?
  Overflow

Integers IV

- Julia has some useful pre-defined functions
- E.g. the largest and smallest representable values:

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

- What happens if we add 1 to the maximum value?
  Overflow

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

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

Integers IV

- Julia has some useful pre-defined functions
- E.g. the largest and smallest representable values:

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

- What happens if we add 1 to the maximum value?
  Overflow

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

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

Collaborative exercise II

Thinking about rationals and complex numbers

We now know how to store an integer.

1. How can we store a **rational number** (fraction) a/b?
2. How can we multiply two rationals? And add them?
3. How can we store a **complex number** $a + ib$ with integer a and b?
4. How can we add two complexes? Add mutiply them?

Rational numbers

- A **rational number** is a fraction p/q with $p, q \in \mathbb{Z}$
- We can represent one as a *pair* of integers (p, q)

Rational numbers

- A **rational number** is a fraction p/q with $p, q \in \mathbb{Z}$
- We can represent one as a *pair* of integers (p, q)
- But different pairs can represent the *same* rational number, e.g. $6/8 == 3/4$

Rational numbers

- A **rational number** is a fraction p/q with $p, q \in \mathbb{Z}$
- We can represent one as a *pair* of integers (p, q)
- But different pairs can represent the *same* rational number, e.g. $6/8 == 3/4$

- Julia already has support for rational numbers:

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

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

Defining new number types

- Suppose we wanted to work with rational numbers

Defining new number types

- Suppose we wanted to work with rational numbers

- We could represent p/q as a pair (p, q) of integers
- And define operations like + on those pairs . . .

- But suppose we also wanted to define complex numbers
- They would also be pairs of integers…

Defining new number types

- Suppose we wanted to work with rational numbers

- We could represent p/q as a pair (p, q) of integers
- And define operations like + on those pairs . . .

- But suppose we also wanted to define complex numbers
- They would also be pairs of integers…

- We need a way to *distinguish* when a pair of numbers should **behave like** a rational or a complex number or…

Defining new number types

- Suppose we wanted to work with rational numbers

- We could represent p/q as a pair (p, q) of integers
- And define operations like + on those pairs . . .

- But suppose we also wanted to define complex numbers
- They would also be pairs of integers…

- We need a way to *distinguish* when a pair of numbers should **behave like** a rational or a complex number or…
- To do so we will **define new types**!

Making a rational number type

- How could we define our own rational number **type**?
- Make a **composite type**
  - A “box” containing various **fields** / **attributes**
  - Collects pieces of data that belong together under one name

```
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!
```

Defining arithmetic operations on types

- A type tells Julia how to **interpret** data
- A 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
```

- We can specify how to display the resulting object:

```
Base.show(io::IO, x::MyRational) = print(io, x.p, " / ", x.q
```

Multiple dispatch

- In Julia, `*` is a normal **(generic) function**
- A **generic function** has different **methods**

Multiple dispatch

- In Julia, `*` is a normal **(generic) function**
- A **generic function** has different **methods**

- A **method** is a version of function acting on certain types:

```
methods(*)
```

Multiple dispatch

- In Julia, `*` is a normal **(generic) function**
- A **generic function** has different **methods**

- A **method** is a version of function acting on certain types:

```
methods(*)
```

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

Are rationals enough?

- Maybe we can just use rationals for everything?

Are rationals enough?

- Maybe we can just use rationals for everything?

- What is $(34/56)^{20}$?
- How can we represent π?
- Or exp$(3//4)$?

Towards the reals: Fixed-point arithmetic

- Re-use an integer, but with a *fixed* position for a binary point:
- $b_7 b_6 b_5 \cdot b_4 b_3 b_2 b_1 b_0$

Towards the reals: Fixed-point arithmetic

- Re-use an integer, but with a *fixed* position for a binary point:
- $b_7 b_6 b_5 \cdot b_4 b_3 b_2 b_1 b_0$

- Now we have something like a real number!

- Effectively we have $n/2^5$ with $n = 0, \ldots, 2^7 - 1$

Towards the reals: Fixed-point arithmetic

- Re-use an integer, but with a *fixed* position for a binary point:
- $b_7 b_6 b_5 \cdot b_4 b_3 b_2 b_1 b_0$

- Now we have something like a real number!

- Effectively we have $n/2^5$ with $n = 0, \ldots, 2^7 - 1$

- Called “fixed-point arithmetic”
- E.g. the Julia `FixedPointNumbers.jl` package

Towards the reals: Fixed-point arithmetic

- Re-use an integer, but with a *fixed* position for a binary point:
- $b_7 b_6 b_5 \cdot b_4 b_3 b_2 b_1 b_0$

- Now we have something like a real number!

- Effectively we have $n/2^5$ with $n = 0, \ldots, 2^7 - 1$

- Called "fixed-point arithmetic"
- E.g. the Julia `FixedPointNumbers.jl` package

- But we can't represent a wide *range* of numbers
- We need something better

Summary

- We can represent integers using binary in the computer
- Up to some size, with **overflow**
- We can define new types to collect pieces of data together
- And hence define rationals and complex numbers
- There are (slow) libraries for arbitrary precision integers that are easy to use from Julia