COMP110: Principles of Computing

11: Numerical Methods

Research Journal

Peer review **tomorrow!**

Representing numbers

Powers of 10

Powers of 10

$$10^6 = 1\underbrace{000000}_{6 \text{ zeroes}}$$

Powers of 10

$$10^6 = 1\underbrace{000000}_{\text{6 zeroes}}$$

$$10^1 = 10$$

Powers of 10

$$10^6 = 1\,\underbrace{000000}_{\text{6 zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

Powers of 10

$$10^6 = 1\underbrace{000000}_{\text{6 zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

$$10^{-1} = 0.1$$

Powers of 10

$$10^6 = 1\underbrace{000000}_{\text{6 zeroes}}$$

$$10^1 = 10$$

$$10^0 = 1$$

$$10^{-1} = 0.1$$

$$10^{-6} = 0.\underbrace{00000}_{\text{5 zeroes}}1$$

Scientific notation

Scientific notation

- A way of writing **very large** and **very small** numbers

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - a $(1 \leq |a| < 10)$ is the **mantissa**

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - a ($1 \leq |a| < 10$) is the **mantissa**
  - (a is a positive or negative number with a single non-zero digit before the decimal point)

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - a ($1 \leq |a| < 10$) is the **mantissa**
  - (a is a positive or negative number with a single non-zero digit before the decimal point)
  - b (an integer) is the **exponent**

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - a $(1 \leq |a| < 10)$ is the **mantissa**
  - (a is a positive or negative number with a single non-zero digit before the decimal point)
  - b (an integer) is the **exponent**
- E.g. 1 light year = 9.461×10^{15} metres

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - a ($1 \leq |a| < 10$) is the **mantissa**
  - (a is a positive or negative number with a single non-zero digit before the decimal point)
  - b (an integer) is the **exponent**
- E.g. 1 light year = 9.461×10^{15} metres
- E.g. Planck's constant = 6.626×10^{-34} joules

Scientific notation

- A way of writing **very large** and **very small** numbers
- $a \times 10^b$, where
  - a $(1 \leq |a| < 10)$ is the **mantissa**
  - (a is a positive or negative number with a single non-zero digit before the decimal point)
  - b (an integer) is the **exponent**
- E.g. 1 light year = 9.461×10^{15} metres
- E.g. Planck's constant = 6.626×10^{-34} joules
- Socrative `FALCOMPED`

Scientific notation in code

Scientific notation in code

Instead of writing $\boxed{\times 10}$, write $\boxed{\texttt{e}}$ (no spaces)

Scientific notation in code

Instead of writing $\boxed{\times 10}$, write $\boxed{\texttt{e}}$ (no spaces)

```
lightYear = 9.461e15
plancksConstant = 6.626e-34
```

Floating point numbers

Floating point numbers

- Similar to scientific notation, but **base 2** (binary)

Floating point numbers

- Similar to scientific notation, but **base 2** (binary)
- $\pm\text{mantissa} \times 2^{\text{exponent}}$

Floating point numbers

- Similar to scientific notation, but **base 2** (binary)
- $\pm\text{mantissa} \times 2^{\text{exponent}}$
- Sign is stored as a single bit: $0 = +$, $1 = -$

Floating point numbers

- Similar to scientific notation, but **base 2** (binary)
- $\pm\text{mantissa} \times 2^{\text{exponent}}$
- Sign is stored as a single bit: $0 = +$, $1 = -$
- Mantissa is a binary number with a 1 before the point; only the digits after the point are stored

Floating point numbers

- Similar to scientific notation, but **base 2** (binary)
- $\pm\text{mantissa} \times 2^{\text{exponent}}$
- Sign is stored as a single bit: $0 = +$, $1 = -$
- Mantissa is a binary number with a 1 before the point; only the digits after the point are stored
- Exponent is a signed integer, stored with a **bias**

IEEE 754 floating point formats

Type	Sign	Exponent	Mantissa	Total
Single precision	1 bit	8 bits	23 bits	32 bits
Double precision	1 bit	11 bits	52 bits	64 bits

IEEE 754 floating point formats

Type	Sign	Exponent	Mantissa	Total
Single precision	1 bit	8 bits	23 bits	32 bits
Double precision	1 bit	11 bits	52 bits	64 bits

Exponent is stored with a **bias**:

- Single precision: store exponent $+ 127$
- Double precision: store exponent $+ 1023$

IEEE 754 floating point formats

Type	Sign	Exponent	Mantissa	Total
Single precision	1 bit	8 bits	23 bits	32 bits
Double precision	1 bit	11 bits	52 bits	64 bits

Exponent is stored with a **bias**:

- Single precision: store exponent $+ 127$
- Double precision: store exponent $+ 1023$

- Python uses double precision

IEEE 754 floating point formats

Type	Sign	Exponent	Mantissa	Total
Single precision	1 bit	8 bits	23 bits	32 bits
Double precision	1 bit	11 bits	52 bits	64 bits

Exponent is stored with a **bias**:

- Single precision: store exponent + 127
- Double precision: store exponent + 1023

- Python uses double precision
- Other languages have `float` (single) and `double` types

Example

Example

```
0 10000001 10100000000000000000000
```

Example

```
0 10000001 10100000000000000000000
```

- Exponent: $129 - 127 = 2$

Example

```
0 10000001 10100000000000000000000
```

- Exponent: $129 - 127 = 2$
- Mantissa: binary 1.101

Example

```
0 10000001 10100000000000000000000
```

- Exponent: $129 - 127 = 2$
- Mantissa: binary 1.101
- $1 + \frac{1}{2} + \frac{1}{8} = 1.625$

Example

```
0 10000001 10100000000000000000000
```

- Exponent: $129 - 127 = 2$
- Mantissa: binary 1.101
- $1 + \frac{1}{2} + \frac{1}{8} = 1.625$
- $1.625 \times 2^2 = 6.5$

Example

`0 10000001 10100000000000000000000`

- Exponent: $129 - 127 = 2$
- Mantissa: binary 1.101
- $1 + \frac{1}{2} + \frac{1}{8} = 1.625$
- $1.625 \times 2^2 = 6.5$
- Alternatively: $1.101 \times 2^2 = 110.1$

Example

```
0 10000001 10100000000000000000000
```

- Exponent: $129 - 127 = 2$
- Mantissa: binary 1.101
- $1 + \frac{1}{2} + \frac{1}{8} = 1.625$
- $1.625 \times 2^2 = 6.5$
- Alternatively: $1.101 \times 2^2 = 110.1$
- $= 4 + 2 + \frac{1}{2} = 6.5$

Socrative `FALCOMPED`

Socrative `FALCOMPED`

What is the value of this number expressed in IEEE 754 single precision format?

`0 10000010 01011000000000000000000`

You have **5 minutes**, and you **may** use a calculator! (Unless your calculator does IEEE 754 conversion...)

Example

Example

```
0 10000010 01011000000000000000000
```

Example

```
0 10000010 01011000000000000000000
```

- Exponent: $130 - 127 = 3$

Example

```
0 10000010 01011000000000000000000
```

- Exponent: $130 - 127 = 3$
- Mantissa: binary 1.01011

Example

`0 10000010 01011000000000000000000`

- Exponent: $130 - 127 = 3$
- Mantissa: binary 1.01011
- $1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} = 1.34375$

Example

```
0 10000010 01011000000000000000000
```

- Exponent: $130 - 127 = 3$
- Mantissa: binary 1.01011
- $1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} = 1.34375$
- $1.34375 \times 2^3 = 10.75$

Example

```
0 10000010 01011000000000000000000
```

- Exponent: $130 - 127 = 3$
- Mantissa: binary 1.01011
- $1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} = 1.34375$
- $1.34375 \times 2^3 = 10.75$
- Alternatively: $1.01011 \times 2^3 = 1010.11$

Example

```
0 10000010 01011000000000000000000
```

- Exponent: $130 - 127 = 3$
- Mantissa: binary 1.01011
- $1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} = 1.34375$
- $1.34375 \times 2^3 = 10.75$
- Alternatively: $1.01011 \times 2^3 = 1010.11$
- $= 8 + 2 + \frac{1}{2} + \frac{1}{4} = 10.75$

Precision of floating point numbers

Precision of floating point numbers

- Precision **varies** by **magnitude**

Precision of floating point numbers

- Precision **varies** by **magnitude**
- Numbers near 0 can be stored more accurately than numbers further from 0

Precision of floating point numbers

- Precision **varies** by **magnitude**
- Numbers near 0 can be stored more accurately than numbers further from 0
- Analogy: in scientific notation with 3 decimal places

Precision of floating point numbers

- Precision **varies** by **magnitude**
- Numbers near 0 can be stored more accurately than numbers further from 0
- Analogy: in scientific notation with 3 decimal places
  - Around 3.142×10^0: can represent a difference of 0.001

Precision of floating point numbers

- Precision **varies** by **magnitude**
- Numbers near 0 can be stored more accurately than numbers further from 0
- Analogy: in scientific notation with 3 decimal places
  - Around 3.142×10^0: can represent a difference of 0.001
  - Around 3.142×10^3: can represent a difference of 1

Precision of floating point numbers

- Precision **varies** by **magnitude**
- Numbers near 0 can be stored more accurately than numbers further from 0
- Analogy: in scientific notation with 3 decimal places
  - Around 3.142×10^0: can represent a difference of 0.001
  - Around 3.142×10^3: can represent a difference of 1
  - Around 3.142×10^6: can represent a difference of 1000

Range of floating point numbers

Type	Smallest value	Largest value
Single precision	$\pm 1.175 \times 10^{-38}$	$\pm 3.403 \times 10^{38}$
Double precision	$\pm 2.225 \times 10^{-308}$	$\pm 1.798 \times 10^{308}$

Rounding errors

Rounding errors

- Many numbers cannot be represented exactly in IEEE float

Rounding errors

- Many numbers cannot be represented exactly in IEEE float
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$

Rounding errors

- Many numbers cannot be represented exactly in IEEE float
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$
- Decimal: can represent $\frac{a}{b}$ exactly iff $b = 2^m 5^n$

Rounding errors

- Many numbers cannot be represented exactly in IEEE float
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$
- Decimal: can represent $\frac{a}{b}$ exactly iff $b = 2^m 5^n$
- Binary: can represent $\frac{a}{b}$ exactly iff $b = 2^n$

Rounding errors

- Many numbers cannot be represented exactly in IEEE float
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$
- Decimal: can represent $\frac{a}{b}$ exactly iff $b = 2^m 5^n$
- Binary: can represent $\frac{a}{b}$ exactly iff $b = 2^n$
- In particular, IEEE float can't represent $\frac{1}{10} = 0.1$ exactly!

Rounding errors

- Many numbers cannot be represented exactly in IEEE float
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$
- Decimal: can represent $\frac{a}{b}$ exactly iff $b = 2^m 5^n$
- Binary: can represent $\frac{a}{b}$ exactly iff $b = 2^n$
- In particular, IEEE float can't represent $\frac{1}{10} = 0.1$ exactly!
- This can lead to **rounding errors** with some calculations

Rounding errors

- Many numbers cannot be represented exactly in IEEE float
    - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$ or $\frac{1}{7} = 0.142857\ldots$
- Decimal: can represent $\frac{a}{b}$ exactly iff $b = 2^m 5^n$
- Binary: can represent $\frac{a}{b}$ exactly iff $b = 2^n$
- In particular, IEEE float can't represent $\frac{1}{10} = 0.1$ exactly!
- This can lead to **rounding errors** with some calculations
    - E.g. according to Python, $0.1 + 0.2 - 0.3 = 5.551 \times 10^{-17}$

Testing for equality

Testing for equality

- Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea

Testing for equality

- Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- E.g. in Python, `0.1 + 0.2 == 0.3` evaluates to `False`

Testing for equality

- Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- E.g. in Python, `0.1 + 0.2 == 0.3` evaluates to `False`
- Better to check for **approximate equality**: calculate the difference between the numbers, and check that it's smaller than some threshold

Testing for equality

- Due to rounding errors, using `==` or `!=` with floating point numbers is almost always a bad idea
- E.g. in Python, `0.1 + 0.2 == 0.3` evaluates to `False`
- Better to check for **approximate equality**: calculate the difference between the numbers, and check that it's smaller than some threshold

```
THRESHOLD = 1e-5
def is_approx_equal(a, b):
    return abs(b - a) < THRESHOLD
```

Decimal types

Decimal types

- Python (and other languages) provide a `decimal` type

Decimal types

- Python (and other languages) provide a `decimal` type
- Uses base 10 rather than base 2, so avoids some of the gotchas with IEEE float

Decimal types

- Python (and other languages) provide a `decimal` type
- Uses base 10 rather than base 2, so avoids some of the gotchas with IEEE float
- ... however not natively supported by the CPU, hence much slower