COMP220: Graphics & Simulation

11: Numerical Accuracy

Next week

Catch-up tutorials

- Last chance for feedback on your CPD task

COMP210 and COMP220 vivas

- Timetable still being finalised
- Keep an eye on Slack / email

Deadlines?!?

- Check MyFalmouth

FALMOUTH UNIVERSITY

Representing numbers

Powers of 10

Powers of 10

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

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

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

$$10^1 = 10$$

$$10^0 = 1$$

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

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

Powers of 10

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

$$10^1 = 10$$

$$10^0 = 1$$

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

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

Multiplying by powers of 10 = shifting the decimal point left/right

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 C++

Scientific notation in C++

Instead of writing $\times 10$, write `e`

Scientific notation in C++

Instead of writing $\times 10$, write `e`

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

Scientific notation in C++

Instead of writing $\times 10$, write `e`

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

This also works in Python and many other programming languages

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)
- $\pm$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)
- $\pm$mantissa $\times$ 2^{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)
- $\pm$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**

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 01111100 10011000000000000000000`

You have **5 minutes**, and you **may** use a calculator!

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**

Limitations of floating point numbers

Limitations of floating point numbers

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

Limitations of floating point numbers

- Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
  - Why? Socrative `FALCOMPED`

Limitations of floating point numbers

- Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
  - Why? Socrative `FALCOMPED`
- Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$

Limitations of floating point numbers

- Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
  - Why? Socrative `FALCOMPED`
- Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$

Limitations of floating point numbers

- Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
  - Why? Socrative `FALCOMPED`
- Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$
- This can lead to **rounding errors** with some calculations

Limitations of floating point numbers

- Precision **varies** by **magnitude**: numbers near 0 can be stored more accurately than numbers further from 0.
  - Why? Socrative `FALCOMPED`
- Many numbers cannot be represented exactly, e.g. $\frac{1}{5}$
  - Similar to how decimal notation cannot exactly represent $\frac{1}{3} = 0.3333333\ldots$
- This can lead to **rounding errors** with some calculations
  - E.g. according to Python, `0.1 + 0.2 == 0.30000000000000004`

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

Numerical accuracy in simulations

Numerical accuracy in simulations

- Errors tend to **accumulate**

Numerical accuracy in simulations

- Errors tend to **accumulate**
- Mixing **orders of magnitude** (i.e. mixing large and small numbers) is particularly bad

Fix your timestep!

Fix your timestep!

- Euler integration: $x(t+h) \approx x(t) + h \times \frac{dx}{dt}(t)$

Fix your timestep!

- Euler integration: $x(t+h) \approx x(t) + h \times \frac{dx}{dt}(t)$
- If h varies, simulation becomes **non-deterministic** (or "random")

Fix your timestep!

- Euler integration: $x(t+h) \approx x(t) + h \times \frac{dx}{dt}(t)$
- If h varies, simulation becomes **non-deterministic** (or "random")
- This is bad!

Fix your timestep!

- Euler integration: $x(t+h) \approx x(t) + h \times \frac{dx}{dt}(t)$
- If h varies, simulation becomes **non-deterministic** (or "random")
- This is bad!
- Better to use a **fixed time step** (we covered this in COMP150)

Fixed time step

```
bool running = true;
Uint32 lastUpdateTime = SDL_GetTicks();
const Uint32 timePerUpdate = 1000 / 60;

while (running)
{
    Uint32 currentTime = SDL_GetTicks();
    handleInput();

    while (currentTime - lastUpdateTime >=
        timePerUpdate)
    {
        update();
        lastUpdateTime += timePerUpdate;
    }

    render();
}
```

Further information on fixed time steps

- http://gafferongames.com/game-physics/fix-your-timestep/
- http://gameprogrammingpatterns.com/game-loop.html

Advanced physics simulation

http://www.gdcvault.com.ezproxy.falmouth.ac.uk/play/1022143/Math-for-Game-Programmers-Game

http://www.gdcvault.com.ezproxy.falmouth.ac.uk/play/1017644/Physics-for-Game-Programmers-Continuous

http://www.gdcvault.com.ezproxy.falmouth.ac.uk/play/1020603/Physics-for-Game-Programmers-Understanding