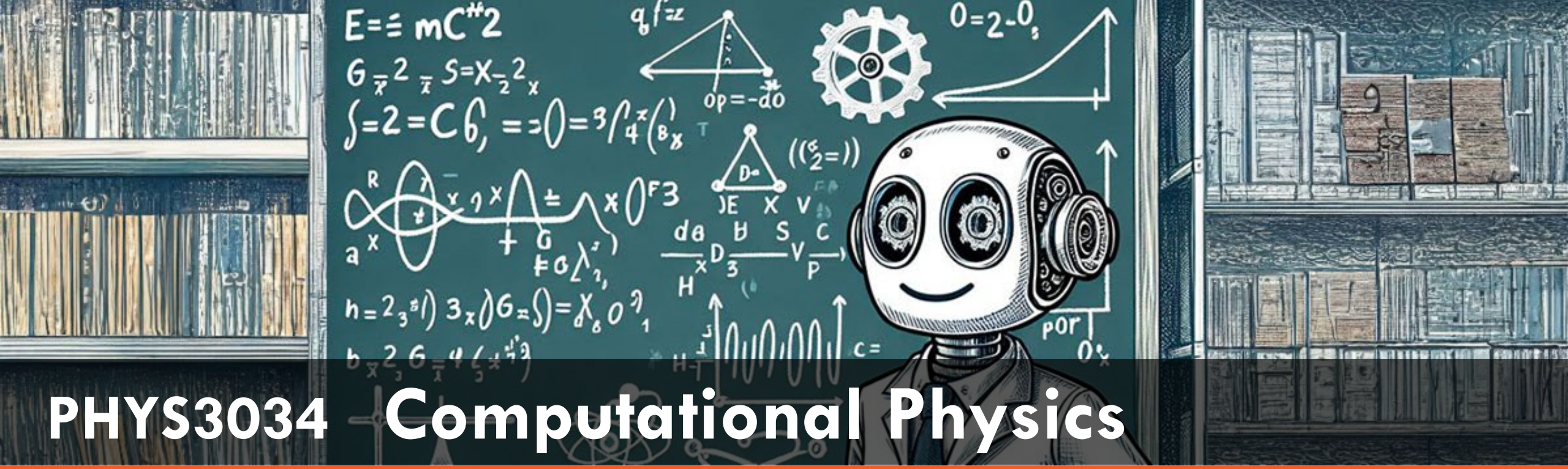


PHYS3034 Computational Physics

A/Professor Sveta Postnova

svetlana.postnova@sydney.edu.au

THE UNIVERSITY OF SYDNEY

We recognise and pay respect to the Elders and communities – past, present, and emerging – of the lands that the University of Sydney's campuses stand on. For thousands of years they have shared and exchanged knowledges across innumerable generations for the benefit of all.

THE UNIVERSITY OF SYDNEY

Celebrating 175 years

Module Structure

Five Lectures: Tuesday 9-10 AM, weeks 1-5

All materials on Canvas

Assessment

Five Computer Labs (10%)

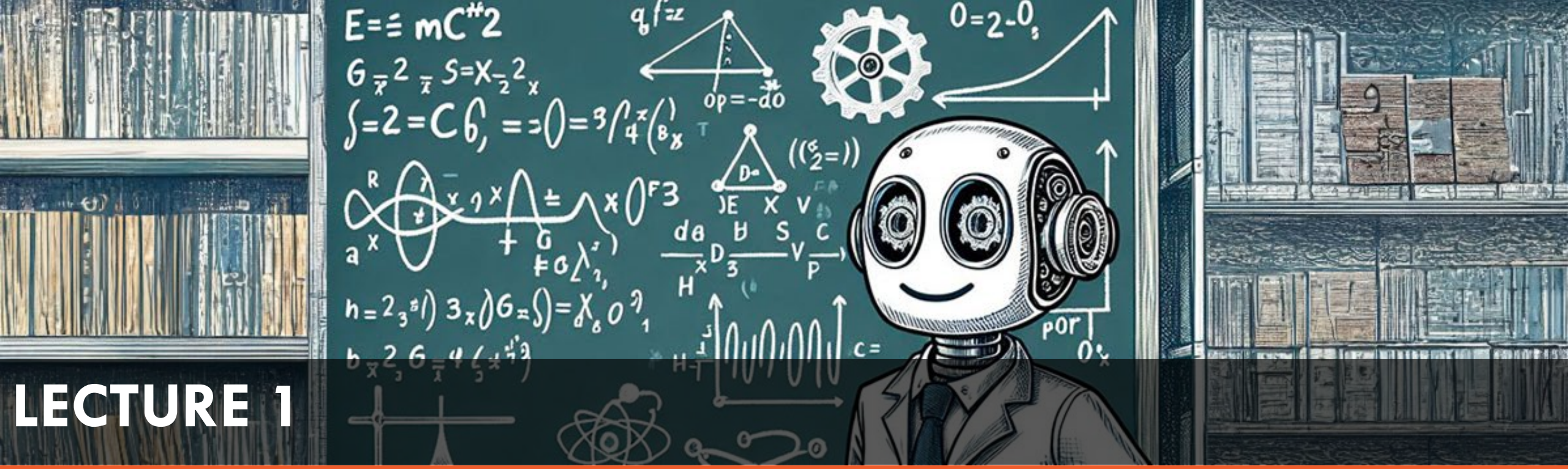
- 2-h labs, Thursday 10 AM – 12 PM, weeks 1-5
- Tutors: Christian Canete, Will Edibam, Akshaya Rajesh, Denis Furtel
- Assessed via Canvas Quiz (2% each) due 2 weeks after the associated lecture

Assignment (10%, no exam)

- Released in week 4, due in week 6

Communication and feedback

- Post questions to Ed Discussion → Computational Physics
- Only email me if you have a question that concerns only you
- All lectures and codes for this module are on GitHub (see Canvas for the link):
  - We will use Python
  - Matlab codes available on Canvas if needed



LECTURE 1

THE UNIVERSITY OF SYDNEY

Computation in Physics

In this course, we learn the theory of how to solve physics problems on computers.

- how to develop numerical algorithms to solve physical equations.
- and how to analyse them for stability and accuracy.

We don't teach coding; we focus on the algorithms and how they work.

Why do we need computation?

Physics has theory, experiment, and *computation*.

Undergrad physics is training in the simplest cases (where analytic techniques can often work). In the real world, physical equations often need to be solved numerically.

- E.g.,: molecular dynamics, N-body problems, nonlinear systems, etc.

Modern physics research relies heavily on computational methods.

- Solving for electric and magnetic fields in stars (Mike Wheatland).
- Dynamics of online social networks (Tristram Alexander).
- Biological rhythms (Svetlana Postnova), and brain dynamics (Pulin Gong, Ben Fulcher), …

Types of systems we will work with

Ordinary Differential Equations (ODEs):

- Numerical error, Euler method – *projectile motion*
- Verlet methods – *planetary motion*
- Runge-Kutta – *motion of a pendulum*

Partial Differential Equations (PDEs):

- FTCS – *heat diffusion*
- Lax method – *wave propagation*

Lecture 1: content

Types of numerical error:

- Floating point error (representation of numbers).
- Truncation error (numerical method).

Dynamics problems: the motion of a particle.

- Simple projectile motion: motion under gravity.

Non-dimensionalisation

- A standard procedure prior to numerical solution.
- Applied to projectile motion.

Euler's method

- Applied to simple projectile motion.

Global truncation error

- Estimating the error of a numerical solution.

Types of numerical error

THE UNIVERSITY OF SYDNEY

Number representation on computer (recap)

Computers don't know numbers; they operate in bits: 0s and 1s

Instead of base-10 (0–9), computers use **base-2**.

Each bit represents a power of 2

Bit position	7	6	5	4	3	2	1	0
Bit value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Example: $\mathbf{13_{10} = 00001101_2}$

$\rightarrow 2^3 + 2^2 + 2^0$

$= 8 + 4 + 1$

Unsigned integers:

With **N bits**, you can represent:

$0 \ to \ 2^N - 1$

- 8-bit: 0 → 255
- 16-bit: 0 → 65,535
- 32-bit: 0 → 4,294,967,295

Signed integers:

With **N bits**, you can represent:

$-2^{N-1} \ to \ 2^{N-1} - 1$

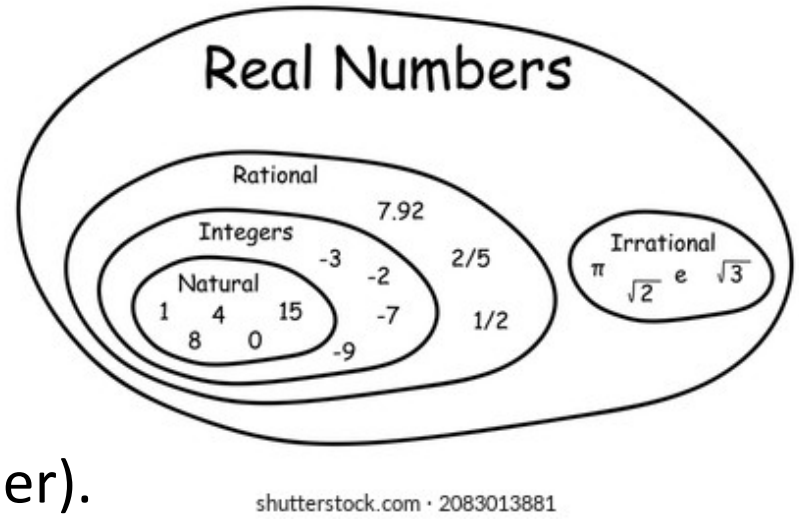
- Negative numbers are stored by inverting bits and adding 1
- The **leftmost bit** is the sign bit
- E.g., $-13_{10} = 11110011_2$

How to represent real numbers?

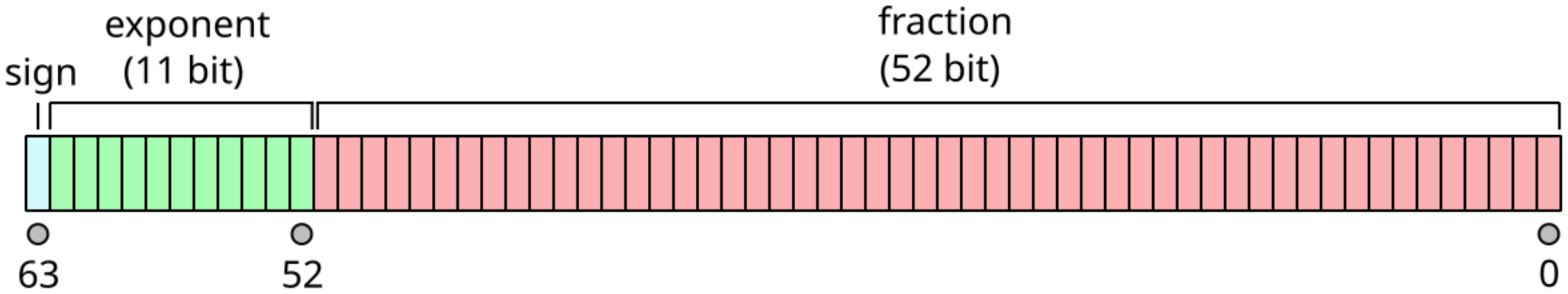
- **Floating point** is an approximate representation of real numbers.
- IEEE double precision (**binary64**) is the standard in modern coding languages.
- Uses base 2 with 64 bits (0s or 1s) per number.
- A binary64 number, $\boldsymbol{x}$, is represented as

$$\boldsymbol{x = (-1)^s \times M \times 2^E}$$

  - s is the sign bit.
  - M is a 52-bit significand (the fractional part of the number).
  - E is the exponent (represented with 11 bits).



How to represent real numbers?



- **Sign bit (1 bit)** → Determines positive (0) or negative (1).
- **Exponent (11 bits)** → 2048 → Stored in **biased** form (E = exponent + 1023).
- **Mantissa (52 bits)** → Stores the significant digits (without the leading 1).

$$x = (-1)^s \times M \times 2^E$$

There are infinitely many reals, but only a finite number of floating-point numbers...

-Yields **range error** and **rounding error**.

Overflow and Underflow errors

Overflow: Number is *too large* for binary64.

Underflow: Number is *too small* for binary64

Precision limit, machine epsilon (eps, ϵ): the fractional spacing between numbers

Quantity	Meaning	Python Value (approx.)	Python command
Real maximum	Largest finite normal number	1.7976931348623157e+308	np.finfo(np.float64).max
Real minimum (normal)	Smallest positive normal number	2.2250738585072014e-308	np.finfo(np.float64).tiny
Smallest subnormal	Smallest positive subnormal number	5e-324	np.finfo(np.float64).smallest_subnormal
Precision limit (ε)	Machine epsilon (spacing at 1)	2.220446049250313e-16	np.finfo(np.float64).eps

Rounding errors

- ***Rounding error*: Every real number is 'rounded' to the nearest floating-point number**
  - Spaced, multiplicatively, by relative factors, eps.

- Let's predict the results of:

❓ >> 1 + eps - 1?

❓ >> 1 + 0.5*eps - 1?

❓ >> 10 + eps - 10?

A real number is stored **exactly** in binary64 **if and only if** it can be written as:

$\frac{k}{2^n}$ *with integers* k, n within the representable exponent range.

That's it. Everything else is rounded.

Rounding errors: test

Calculate: 0.3 + 0.1 - 0.3 - 0.1 = ?

You: ans = 0
Computer: ans = 2.7756e-17

❓ How could this happen?

Are you still worried about computers taking over the world?

Non-Numbers (NaN, nan)

Non-numbers: Me is *too wild* for binary64:

- NaN (or nan) stands for 'not a number’.
- Example: 0/0 doesn't have a numeric answer.
- NaNs 'propagate': NaN + x gives NaN for all x.

Summary: Float-point errors

- *Range error*: due to there being the largest and smallest floats.
  - *Underflow*: $|x| \lesssim 5 \times 10^{-324}$replaced by 0.
  - *Overflow*: $|x| \gtrsim 2 \times 10^{308}$ replaced by Inf.

- *Rounding error* is due to the discrete spaces between floats.
  - Results are rounded to the nearest float.
  - The fractional error is $\leq \frac{1}{2}\epsilon$
  - $\epsilon = 2^{-52} \approx 2.2 \times 10^{-16}$is the *machine epsilon/precision*

Truncation Errors

- A *truncation error* is the error you introduce when you replace an exact mathematical process with a finite approximation.

- When we write down algorithms to *solve equations*, they make numerical approximations of quantities like derivatives. This leads to truncation error.

  - Example: *estimating the derivative* $\boldsymbol{f'(x)}$ numerically, where we can evaluate $\boldsymbol{f(x)}$ for any input, $\boldsymbol{x}$.

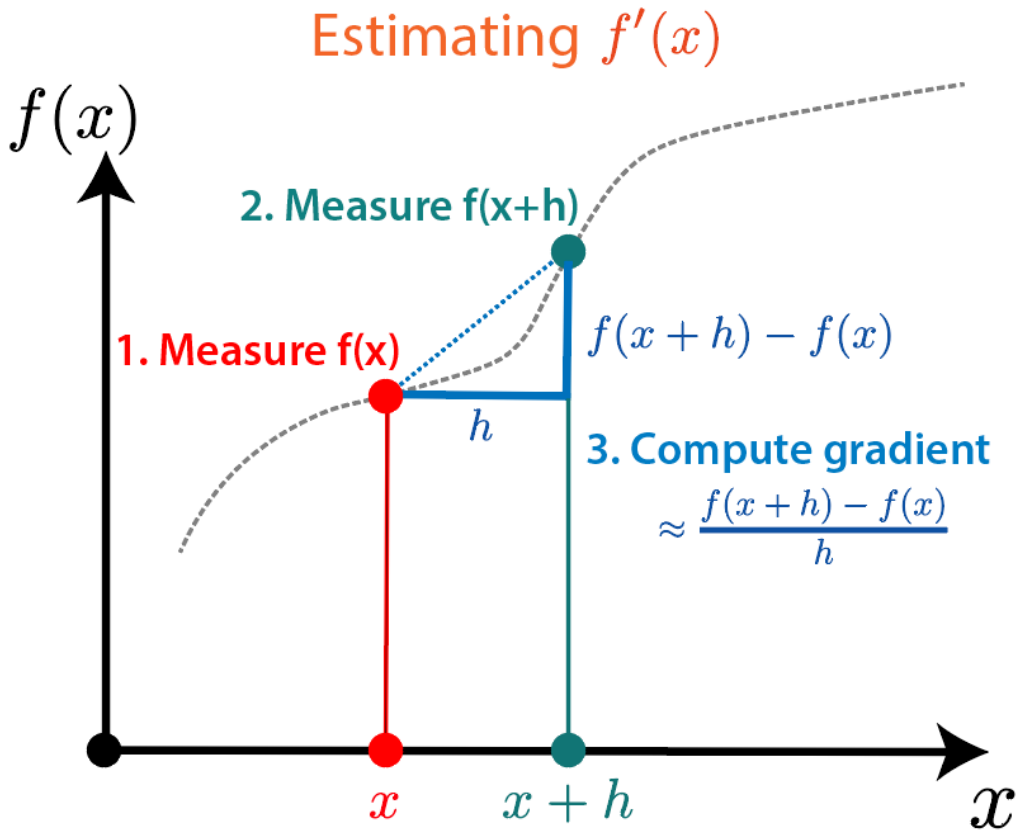
❓ What would be your strategy ('algorithm', 'numerical method')?

Estimating Derivatives: the forward difference approximation

- Formally: $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$

- We approximate: $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
  - Measure $f(x)$.
  - Measure $f(x+h)$ for some (small) increment, h .
  - Compute the gradient.

If h is small enough such that the function does not vary too much over h, this should be a good numerical approximation



Truncation Error due to forward difference approximation

- Function $f(x+h)$ is described **exactly** with Taylor expansion:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \cdots$$

- Rearranging for $f'(x)$ gives an exact expression for the derivative

$$\underbrace{f'(x) = \frac{f(x+h) - f(x)}{h}}_{\text{approximation}} \underbrace{- \frac{h}{2}f''(x) - \frac{h^2}{6}f'''(x) + \cdots}_{O(h),\ \text{truncation error}}$$

- We say that the local truncation error is of order h
- We *truncate* the full series when terms become proportional to (small) h
- The *truncation error* limits accuracy (how close you are to the correct answer)
  - And typically *dominates floating point errors*

See a detailed video on Canvas

Dynamics Problems

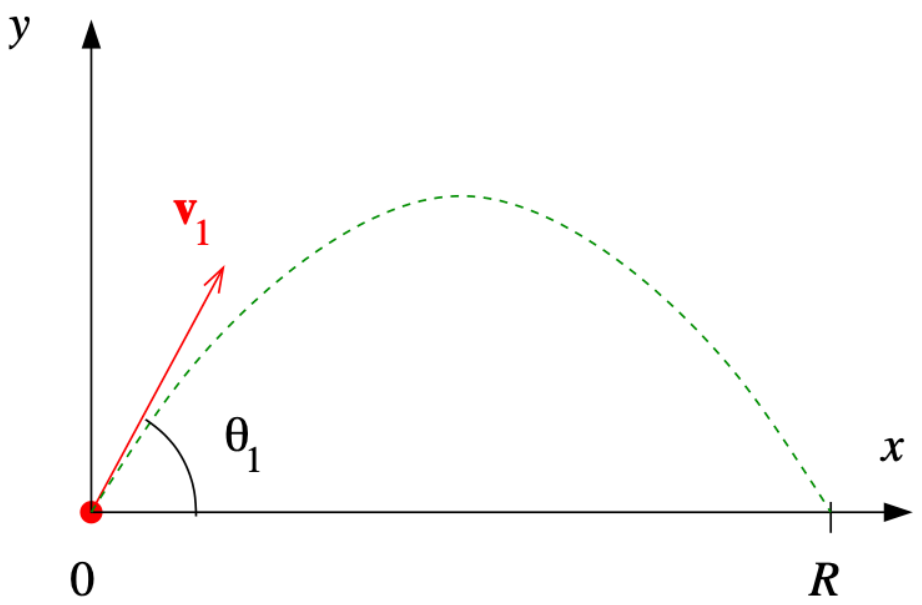
THE UNIVERSITY OF SYDNEY

Dynamics problems: solving the motion of a particle

- $\frac{d\mathbf{r}}{dt} = \mathbf{v}$, and $\frac{d\mathbf{v}}{dt} = \mathbf{a}(\mathbf{r}, \mathbf{v}, t)$
  - *position*: $\mathbf{r} = (x(t), y(t), z(t)) = \mathbf{r}(t)$,
  - *velocity*: $\mathbf{v} = \mathbf{v}(t) = (v_x(t), v_y(t), v_z(t))$
  - *acceleration*: $\mathbf{a}(\mathbf{r}, \mathbf{v}, t)$
- Six coupled Ordinary Differential Equations (ODEs).
- The problem is defined by specifying:
  - *(i)* acceleration, $\mathbf{a}(\mathbf{r}, \mathbf{v}, t)$, and *(ii)* initial condition: $\mathbf{v}(t = 0)$, $\mathbf{r}(t = 0)$.
- *Our goal*: take equations we want to solve, and construct a method for us to walk forward in time to piece together a trajectory 🏃,
  - *Our worry*: we make a truncation error at each step…

Example: Simple projectile motion

- Constant gravity: $\mathbf{a} = -g\hat{\mathbf{y}}$ (a constant vector), where
  - $g \approx 9.81\,\text{ms}^{-2}$: acceleration due to gravity close to the Earth.
  - $\hat{\mathbf{y}}$: vertical unit vector.
- Because $\mathbf{a}$ is a constant, we can solve analytically (using integration) for initial condition $\mathbf{r}(0) = \mathbf{r}_1$ and $\mathbf{v}(0) = \mathbf{v}_1$:
  - $\mathbf{r} = \mathbf{r}_1 + \mathbf{v}_1 t - \frac{1}{2} g t^2 \hat{\mathbf{y}}$
  - $\mathbf{v} = \mathbf{v}_1 - g t \hat{\mathbf{y}}$



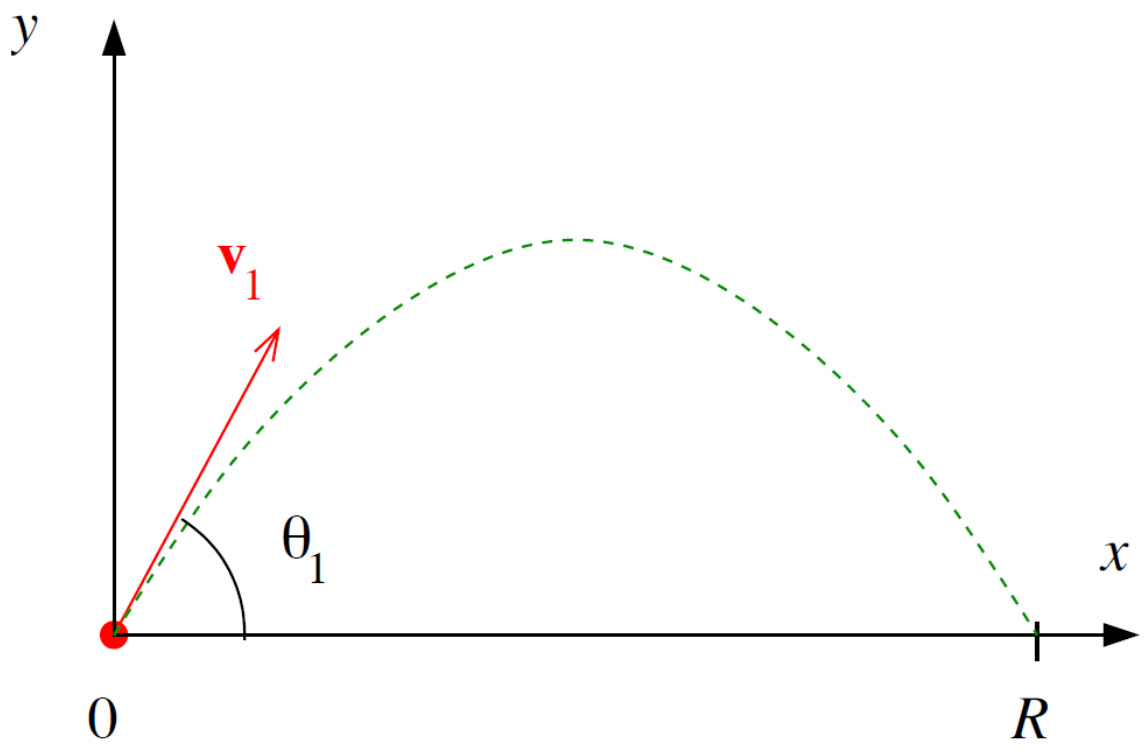
This week's test problem: projectile motion

- Fire an object at some initial velocity, $\mathbf{v}_1$...
- How far will it go? The range, R.
- *Analytic solution*: parabolic trajectory,

$$y = -\frac{g}{2v_1^2 \cos^2 \theta_1} x(x - R).$$

  - The range, $R = \frac{1}{g} v_1^2 \sin(2\theta_1)$.

- We can compare against this to *test the accuracy* of a numerical method.



Non-dimensionalisation

Non-dimensionalising equations

- Physical equations use *dimensional* variables:
  - Time (s), position (m), velocity (ms^{-1}) have physical units.
  - Constants can also be dimensional: e.g., $g = 9.8\ ms^{-2}$.

- In numerical work, it's simpler to *non-dimensionalise*
  - *The goal*: variables and constants in our equations have *no units*.

- *How?*: Rescale dimensional variables to *non-dimensional* (ND) ones:
  - We divide a distance by a distance, a time by a time, etc.
  - When the dust settles, *dimensional constants* can be eliminated and replaced by ND constants.

Non-dimensionalising: projectile motion equations

- *Projectile motion*: $\frac{d\mathbf{r}}{dt} = \mathbf{v}$ and $\frac{d\mathbf{v}}{dt} = -g\hat{\mathbf{y}}$.
- We introduce new *dependent*, $\mathbf{r}$ and $\mathbf{v}$, and *independent*, t, variables:
  - $\bar{\mathbf{r}} = \frac{\mathbf{r}}{L_s}, \quad \bar{t} = \frac{t}{t_s}, \quad \bar{\mathbf{v}} = \frac{\mathbf{v}}{L_s/t_s}.$
- L_s and t_s are a *chosen scale of length and time*.
  - *We can choose* L_s and t_s to be whatever we want, with units length and time.
- The new variables, $\bar{\mathbf{r}}$, $\bar{t}$, and $\bar{\mathbf{v}}$ are *dimensionless* (have no units).
- Careful: *We rescale variables, not constants/parameters*.
  - E.g., we do not rescale g.

Non-dimensionalising: projectile motion equations

- *Velocity*: $\frac{d\mathbf{r}}{dt} = \mathbf{v} \quad \Rightarrow \quad \frac{L_s}{t_s}\frac{d\bar{\mathbf{r}}}{d\bar{t}} = \frac{L_s}{t_s}\bar{\mathbf{v}} \quad \Rightarrow \quad \frac{d\bar{\mathbf{r}}}{d\bar{t}} = \bar{\mathbf{v}}$
- *Acceleration*: $\frac{d\mathbf{v}}{dt} = -g\hat{\mathbf{y}} \quad \Rightarrow \quad \frac{L_s}{t_s^2}\frac{d\bar{\mathbf{v}}}{d\bar{t}} = -g\hat{\mathbf{y}} \quad \Rightarrow \quad \frac{d\bar{\mathbf{v}}}{d\bar{t}} = -\frac{gt_s^2}{L_s}\hat{\mathbf{y}}$
- Our constants occur in a big 📦 as $\frac{gt_s^2}{L_s}$.
  - ❓ How might we set the timescale, t_s, to remove the other constants, g and L_s?

Non-dimensionalising: projectile motion equations

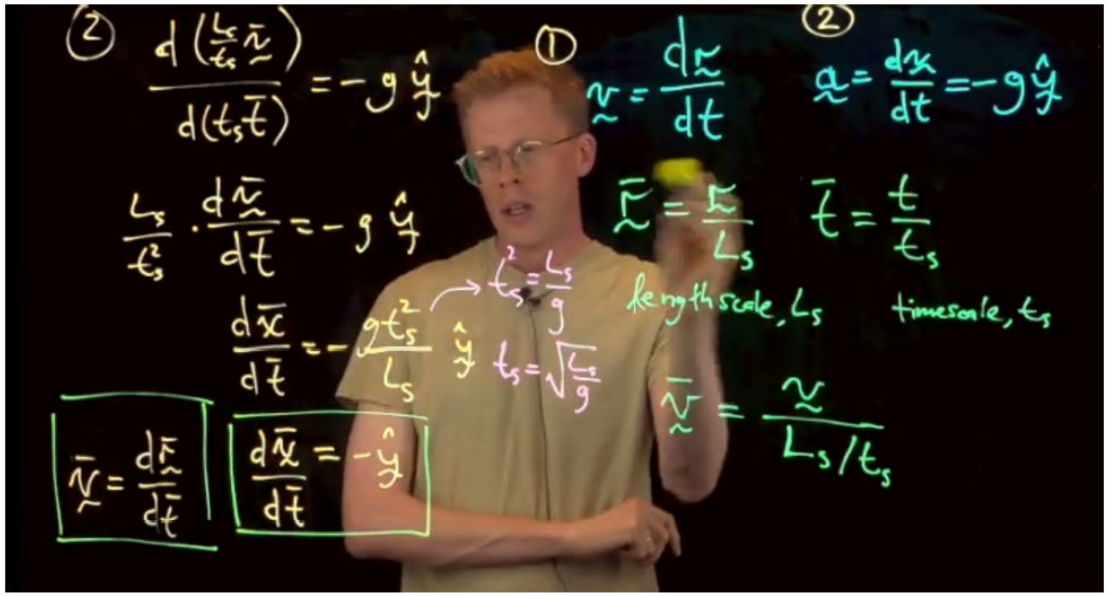
- *Velocity*: $\frac{d\mathbf{r}}{dt} = \mathbf{v} \quad \Rightarrow \quad \frac{L_s}{t_s}\frac{d\bar{\mathbf{r}}}{d\bar{t}} = \frac{L_s}{t_s}\bar{\mathbf{v}} \quad \Rightarrow \quad \frac{d\bar{\mathbf{r}}}{d\bar{t}} = \bar{\mathbf{v}}$
- *Acceleration*: $\frac{d\mathbf{v}}{dt} = -g\hat{\mathbf{y}} \quad \Rightarrow \quad \frac{L_s}{t_s^2}\frac{d\bar{\mathbf{v}}}{d\bar{t}} = -g\hat{\mathbf{y}} \quad \Rightarrow \quad \frac{d\bar{\mathbf{v}}}{d\bar{t}} = -\frac{gt_s^2}{L_s}\hat{\mathbf{y}}$
- Our constants occur in a big 📦 as $\frac{gt_s^2}{L_s}$.
  - ❓ How might we set the timescale, t_s, to remove the other constants, g and L_s?
- Choosing $t_s = \sqrt{\frac{L_s}{g}}$ yields non-dimensional projectile-motion equations: $\boxed{\frac{d\bar{\mathbf{r}}}{d\bar{t}} = \bar{\mathbf{v}}\,, \quad \frac{d\bar{\mathbf{v}}}{d\bar{t}} = -\hat{\mathbf{y}}\,.}$
  - No more annoying constants! 💖
- We perform calculations using the ND equations!
  - If we ever need dimensional values, we can always rescale back!

Non-dimensional tips

- This is a skill you need to have 💪
  - It will come up again and again throughout this course.
- *Note*: We often omit the bars for convenience, and work with non-dimensional quantities.
- When you non-dimensionalize, you should be careful to clearly write down what variables are being rescaled.
- E.g., for dynamics problems we have $\bar{t} = t/t_s$.
  - So whenever we see a t, we need to convert it to $t = \bar{t}\, t_s$.
- For a second derviative, we are rescaling $t^2 = (t_s \bar{t})^2 = t_s^2 \bar{t}^2$, noting that linear rescalings are constants that also rescale derivatives (can be 'pulled out').
  - Yielding $\dfrac{d^2}{dt^2} = \dfrac{d^2}{d(\bar{t}\, t_s)^2} = \dfrac{1}{t_s^2}\dfrac{d^2}{d\bar{t}^2}$

Non-dimensionalisation strips a model down to its essential physics: it reveals the true control parameters, reduces complexity, improves numerical behaviour, and makes results universal rather than unit-dependent.

👨‍🏫 There's a video on Canvas if you want to work through this process a little more slowly.

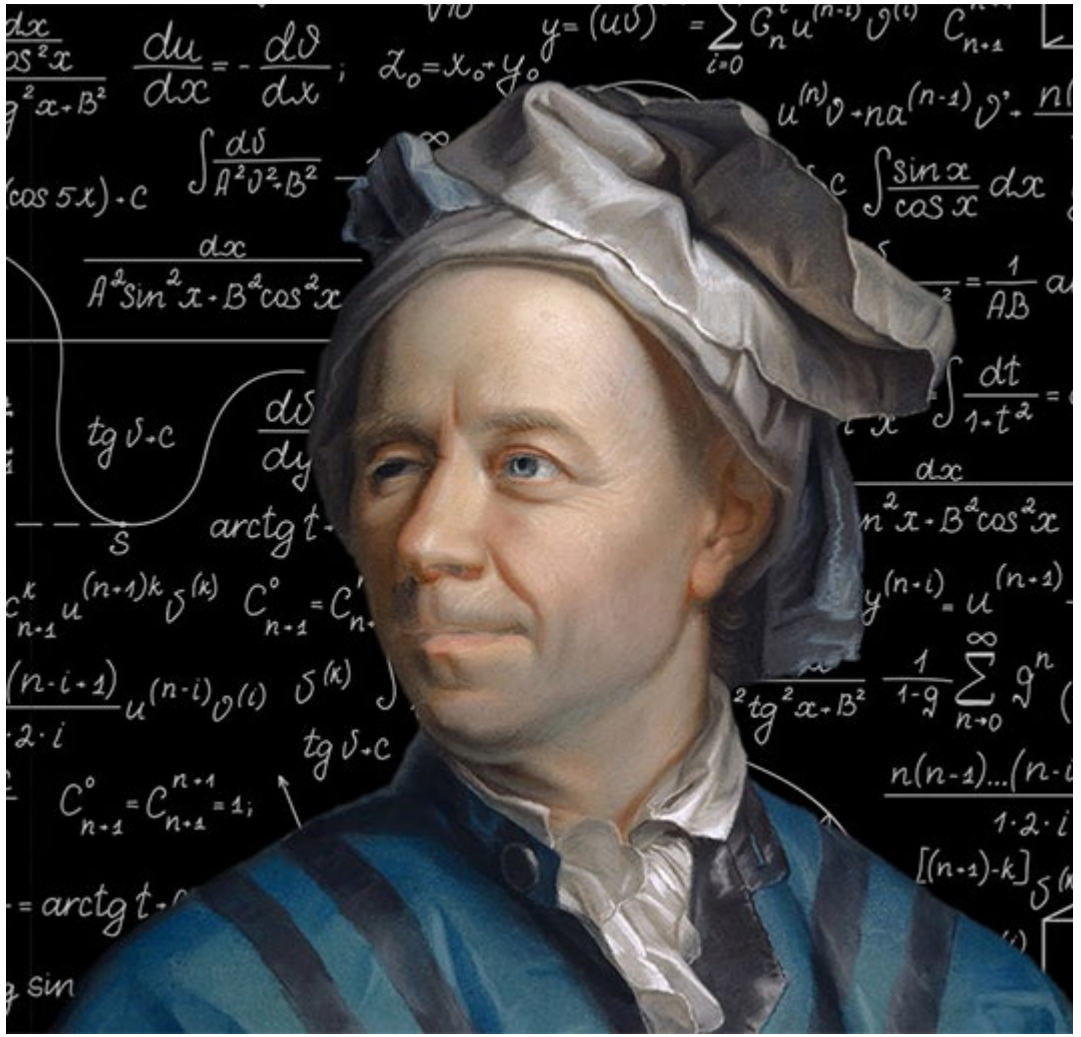


Euler’s method

THE UNIVERSITY OF SYDNEY

Euler's Method

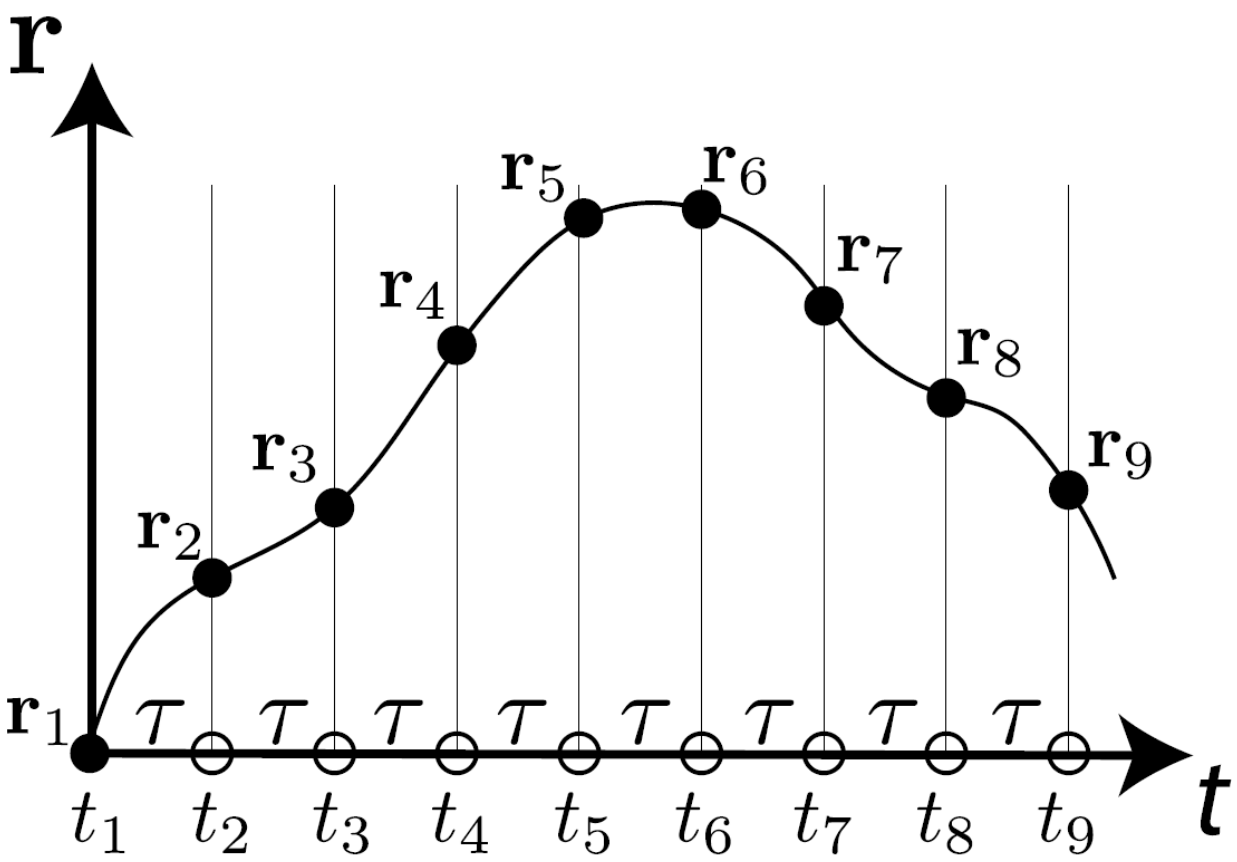
- Our very first numerical method 😄
- From Euler's book, *Institutionum calculi integralis* (1768!)
- Uses the *forward-difference approximation* to evaluate the time derivatives.



Leonhard Euler (1707–1783)

Discretising time

- Numerical methods typically treat time across a *discrete grid*.
- We evaluate at a set of times, $t_n = (n-1)\tau$ $(n = 1, 2, 3 \ldots)$
  - τ is the *time step*.
- The projectile motion equations, $\frac{d\bar{\mathbf{r}}}{d\bar{t}} = \bar{\mathbf{v}}$, $\frac{d\bar{\mathbf{v}}}{d\bar{t}} = -\hat{\mathbf{y}}$, tell us how position and velocity update over time.
  - But to solve them on a computer, *we need to compute these derivatives*.
- We want a set of rules (an *algorithm*) to march forward through time, piecing together the trajectory one τ increment at a time 🏃



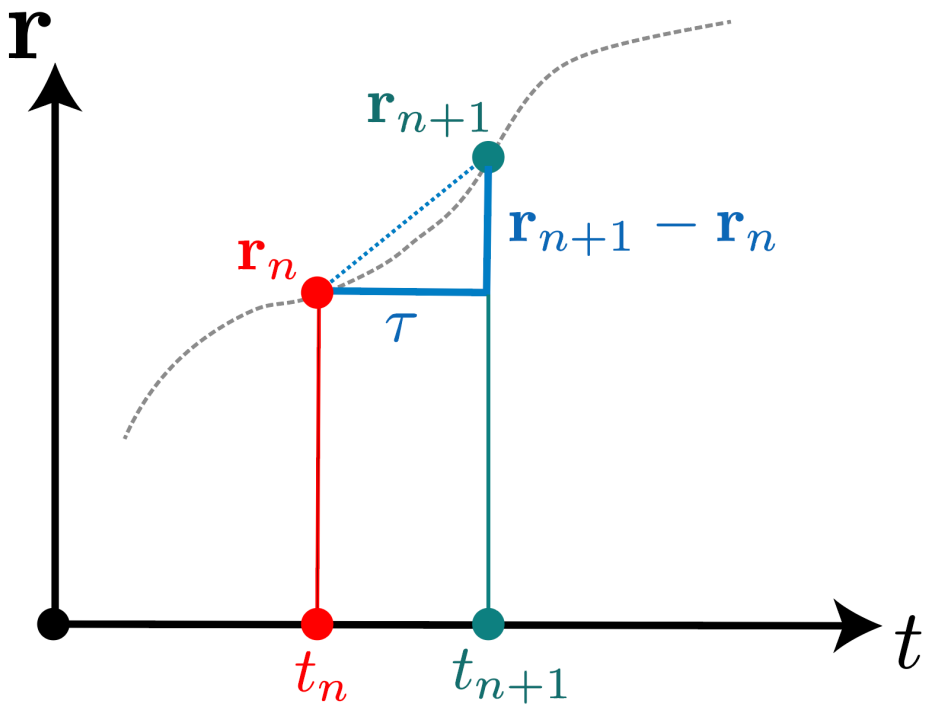
Time derivative: the forward-difference approximation

Estimate time derivative as before: $f'(t) = \frac{f(t+\tau)-f(t)}{\tau} + O(\tau)$

Simple approximation gives us a relationship between two successive time points!

$$\left.\frac{d\mathbf{r}}{dt}\right|_{t_n} = \frac{\mathbf{r}_{n+1} - \mathbf{r}_n}{\tau} + O(\tau)$$
$$\approx \frac{\mathbf{r}_{n+1} - \mathbf{r}_n}{\tau}.$$

Estimating $\left.\frac{d\mathbf{r}}{dt}\right|_{t_n}$



Euler's method for dynamics

- Applies the forward-difference approximation to the derivatives in $\mathbf{r}$ and $\mathbf{v}$:
  - $$\left.\frac{d\mathbf{r}}{dt}\right|_{t_n} = \frac{\mathbf{r}(t_{n+1}) - \mathbf{r}(t_n)}{\tau} + O(\tau)$$
    - We write as $$\left.\frac{d\mathbf{r}}{dt}\right|_{t_n} = \frac{\mathbf{r}_{n+1} - \mathbf{r}_n}{\tau} + O(\tau).$$
  - And similarly for $$\left.\frac{d\mathbf{v}}{dt}\right|_{t_n} = \frac{\mathbf{v}(t_{n+1}) - \mathbf{v}(t_n)}{\tau} + O(\tau) = \frac{\mathbf{v}_{n+1} - \mathbf{v}_n}{\tau} + O(\tau).$$

Applying the forward-difference approximation to ND dynamics equations

$$\frac{d\bar{\mathbf{r}}}{d\bar{t}} = \bar{\mathbf{v}}\,, \qquad \frac{d\bar{\mathbf{v}}}{d\bar{t}} = \bar{\mathbf{a}}\,.$$

$$\frac{1}{\tau}(\mathbf{r}_{n+1} - \mathbf{r}_n) + O(\tau) = \mathbf{v}_n\,, \qquad \frac{1}{\tau}(\mathbf{v}_{n+1} - \mathbf{v}_n) + O(\tau) = \mathbf{a}_n\,.$$

or:

$$\mathbf{r}_{n+1} = \mathbf{r}_n + \tau\mathbf{v}_n + O(\tau^2)\,, \qquad \mathbf{v}_{n+1} = \mathbf{v}_n + \tau\mathbf{a}_n + O(\tau^2)\,.$$

Euler's Method for Dynamics

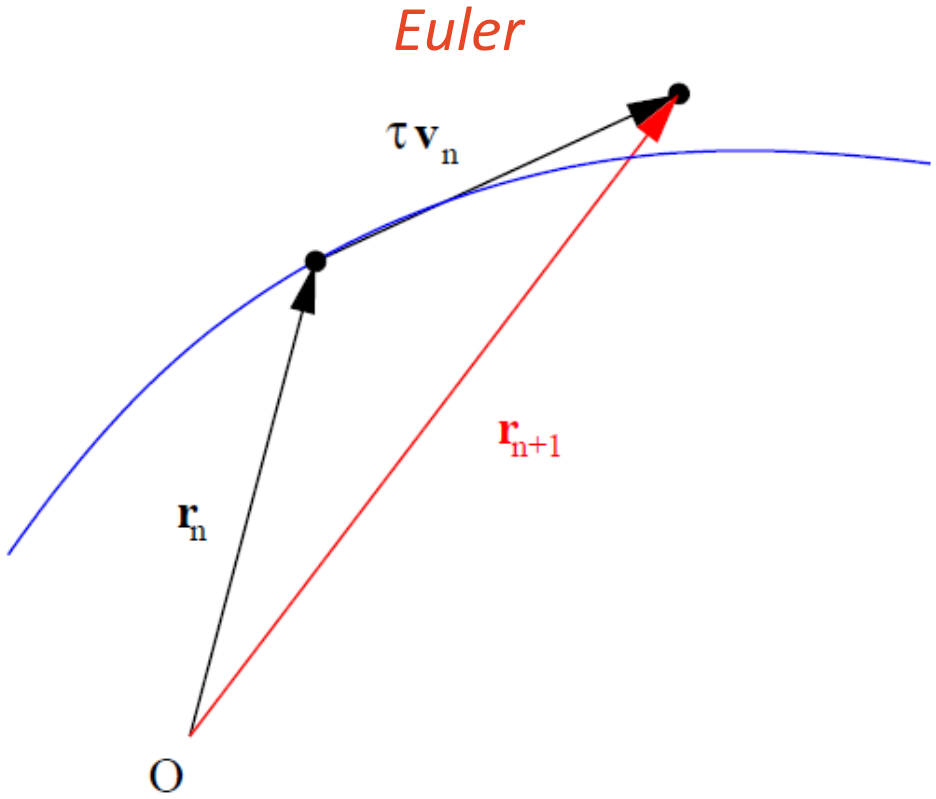
- Excludes terms of order τ^2:
  - $$\boxed{\begin{aligned}\mathbf{r}_{n+1} &= \mathbf{r}_n + \tau \mathbf{v}_n\,, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \tau \mathbf{a}_n\,.\end{aligned}}$$
- Yields a simple rule for stepping forward in time 🏃:
  - $(\mathbf{r}_1, \mathbf{v}_1) \xrightarrow{\tau} (\mathbf{r}_2, \mathbf{v}_2) \xrightarrow{\tau} \cdots$
- In general, we make a *local truncation error* in $\mathbf{r}$ and $\mathbf{v}$ at *every time step*.
  - This error is $O(\tau^2)$ per step in general
    - (but could be less in special situations: e.g., uniform motion).

Stepping forward in time

- $$\mathbf{r}_{n+1} = \mathbf{r}_n + \tau \mathbf{v}_n \,,$$
  $$\mathbf{v}_{n+1} = \mathbf{v}_n + \tau \mathbf{a}_n \,.$$
- We use the current velocity estimate, $\mathbf{v}_n$, to take a step:
  - $(\mathbf{r}_n, \mathbf{v}_n) \xrightarrow{\tau} (\mathbf{r}_{n+1}, \mathbf{v}_{n+1})$.

❓ How much worse is it to take a 'big step' (high τ) than a 'small step' (low τ)?

We make an error at each time step (of order τ^2) and they can *accumulate*.

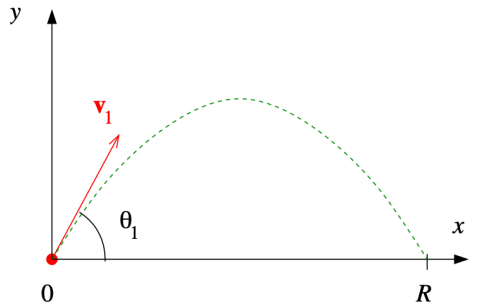


Euler's method for simple projectile motion

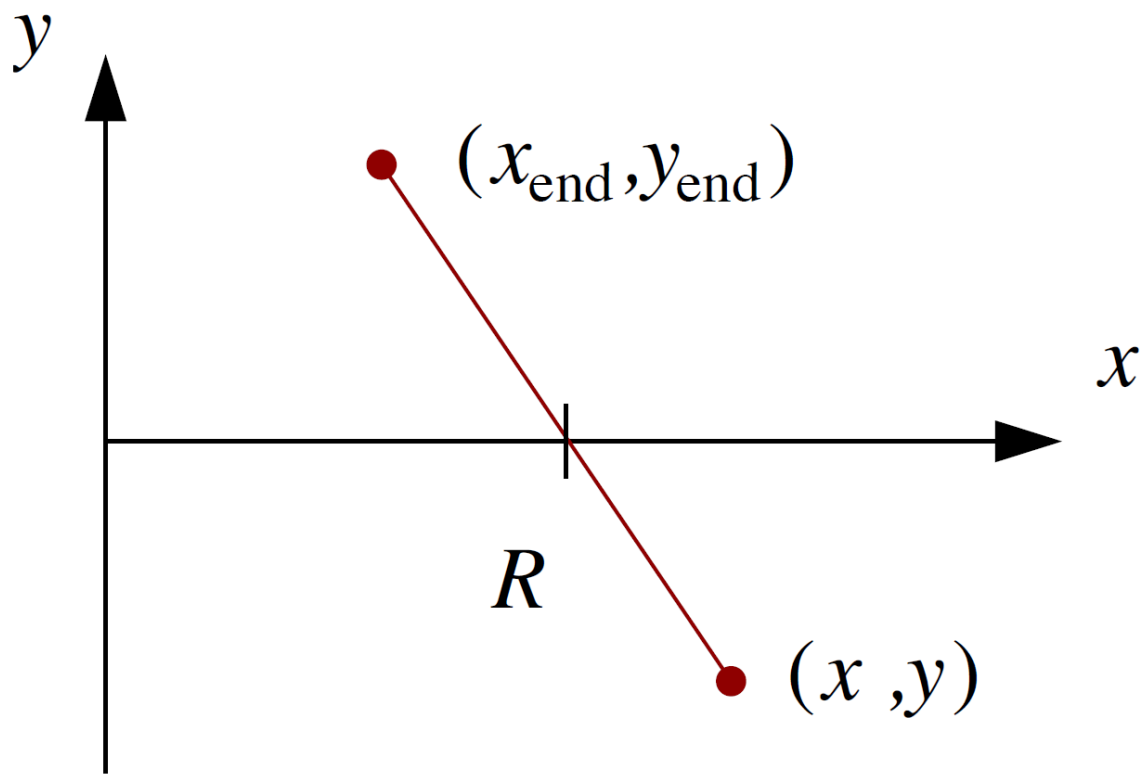
`proj_euler.ipynb` and `Euler_ODE_ipynb`

- Simple *projectile motion* has $\mathbf{a} = -\hat{\mathbf{y}}$, so our Euler algorithm for solving it is:
  - $$\boxed{\begin{aligned}\mathbf{r}_{n+1} &= \mathbf{r}_n + \tau \mathbf{v}_n\,, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n - \tau \hat{\mathbf{y}}\,.\end{aligned}}$$
- The code `proj_euler` solves these *non-dimensional* equations using Euler's method.
  - The user supplies values of v_1 (in m/s) and θ_1 (degrees).
  - Converts to non-dimensional values (L_s and τ are specified within the code).
  - Steps forward ('integrates') using a `while` loop.
  - The loop exits when $y_n \leq 0$ (projectile 'goes underground').
  - Uses *linear interpolation* to estimate the range, R (see next slide).

Estimating R



- `proj_euler` uses linear interpolation
- (*Careful to note*: this approximation is another source of numerical error).
- $$\frac{0 - y_{\text{end}}}{R - x_{\text{end}}} = \frac{y - 0}{x - R}$$
- Solving for R: $R = x - y\frac{x - x_{\text{end}}}{y - y_{\text{end}}}$.



you're linearly interpolating velocity to update position, and **linearly interpolating acceleration to update velocity.**

Global Truncation Error

THE UNIVERSITY OF SYDNEY

Global truncation error

- We know that we make *local truncation errors* at each step, say $O(\tau^k)$ (in general).
  - $O(\tau^2)$, so $k = 2$, for Euler.
- Then we can (naively) estimate the *global error*, E_g, across a total integration time T:
  - $E_g \approx (\text{number of steps}) \times (\text{error in each step})$
  - $E_g \approx \frac{T}{\tau} \times O(\tau^k)$
  - $E_g \approx O(\tau^{k-1})$
- ⭐ If local error is *order* k, then the global error is *order* $k-1$.
  - (A lower order means we're truncating early in the series: *less accurate*).
- This simple argument and calculation assumes that errors in successive steps are independent
  - (*Be careful*: errors can be non-independent).

Truncation errors: Solving dynamics using Euler's method

Consider the full Taylor series for a position update:

$$\begin{aligned}\mathbf{r}_{n+1} &= \mathbf{r}(t_{n+1}) = \mathbf{r}(t_n + \tau)\,, \\ &= \mathbf{r}(t_n) + \tau \left.\frac{d\mathbf{r}}{dt}\right|_{t_n} + \tfrac{1}{2}\tau^2 \left.\frac{d^2\mathbf{r}}{dt^2}\right|_{t_n} + \dots\,, \\ &= \mathbf{r}_n + \tau\mathbf{v}_n + \tfrac{1}{2}\tau^2\mathbf{a}_n + \dots\,, \\ &= \mathbf{r}_n + \tau\mathbf{v}_n - \tfrac{1}{2}\tau^2 g\hat{\mathbf{y}}\,.\end{aligned}$$

- The last step uses the (constant) projectile motion acceleration: $\mathbf{a} = -\frac{1}{2}\tau^2 g\hat{\mathbf{y}}$
  - Higher-order derivatives are zero in this case.
- Comparing to the Euler method update: $\mathbf{r}_{n+1} = \mathbf{r}_n + \tau\mathbf{v}_n$ 😉
  - So we find an *exact expression* for the local error we're making: $\frac{1}{2}\tau^2 g$ (in the y-direction).

Global Error, Δy

- The local error in $\mathbf{r}$ is $\frac{1}{2}\tau^2 g$ (in the y-direction).
- Using our simple assumption of independent errors to estimate the global error, Δy, after a time T:
  - T/τ steps
  - $\frac{1}{2}\tau^2 g$ error per step
  - So $\Delta y \approx \frac{1}{2} T \tau g$.
- For the figure example, $g \approx 10\,\mathrm{m\,s^{-2}}$, $\tau \approx 0.03\,\mathrm{s}$, $T \approx 2\,\mathrm{s}$:
  - global error $\approx \frac{1}{2} \times 2 \times 0.03 \times 10\,\mathrm{m} \approx 30\,\mathrm{cm}$
- This is consistent with the observed error in R.

The Computer Lab

Computational Physics Lab 1

Not available until Feb 23 at 0:00 Due Mar 9 at 10:00 2 pts

- You will simulate projectile motion using the Euler method.
- You have 2 weeks and only 1 attempt to complete the Canvas Quizz
- Lab attendance is optional, but this is your only chance to get help
- *Extra*: How are these ideas used in gaming? Check this YouTube video

1 1 point

Run the code **proj_euler.m** with initial conditions $v_1 = 50$ m/s and $\theta_1 = 40$ deg. Determine the range for the Euler's method solution with the non-dimensional time step $\tau = 0.1$.

Write your answer as a number below, in units of m.

Type your answer...

2 1 point

What is the percentage error in the Euler's method result for the range *R* compared with the (exact) analytic range given by $R = v_1^2 \sin(2\theta_1)/g$ where v_1 is the initial speed and θ_1 is the initial angle to the horizontal?

Write your answer as a number below.

Type your answer...

3 1 point

What is the error in the Euler's method result due to?

Give a brief answer in the text window below.

12pt Paragraph

supplementary

Common Binary Representations of Numeric Values

Unsigned Integers

n bit Positional Binary

value $= d_{n-1} \cdot 2^{n-1} + \cdots + d_1 \cdot 2^1 + d_0 \cdot 2^0$ (n digits)

Example: $n = 8$

00000000	$= 0$	
00000001	$= 2^0$	$= 1$
00000010	$= 2^1$	$= 2$
00000011	$= 2^1 + 2^0$	$= 3$
00000100	$= 2^2$	$= 4$
⋮	⋮	⋮
10000000	$= 2^{n-1}$	$= 128$
⋮	⋮	⋮
11111111	$= 2^n - 1$	$= 255$

Signed Integers

n bit Two's Complement

$(i + \text{2sComplement}(i) = 2^n)$

Example: $n = 8$

01111111	$= 2^{n-1} - 1$	$= +127$
⋮		⋮
00000010	Negation:	$= +2$
00000001	1. invert	$= +1$
00000000	2. add 1	$= \pm 0$
11111111		$= -1$
11111110		$= -2$
⋮		⋮
10000000	$= -2^{n-1}$	$= -128$

Floating-Point Numbers

IEEE 754 Format and derivatives

sign bit | exponent (x bits) | mantissa (m bits)

S | E | M

$n = (1 + x + m)$ bits

Example: half precision ($n = 16,\ x = 5,\ m = 10$)

subnormal	S 00000 M	$= (-1)^S \cdot 2^{-2^{x-1}+2} \cdot \left(0 + \frac{M}{2^m}\right)$		
	0 00000 0000000000	$= -1^0 \cdot 2^{-14} \cdot \left(0 + \frac{0}{1024}\right) = +0$		negative zero
	1 00000 0000000000	$= -1^1 \cdot 2^{-14} \cdot \left(0 + \frac{0}{1024}\right) = -0$		
	0 00000 0000000001	$= -1^0 \cdot 2^{-14} \cdot \left(0 + \frac{1}{1024}\right) \approx +0.0000000596$		smallest positive subnormal
	0 00000 1111111111	$= -1^0 \cdot 2^{-14} \cdot \left(0 + \frac{1023}{1024}\right) \approx +0.0000609756$		largest subnormal
normal	S 0⋮01 M S 1⋮10 M	$= (-1)^S \cdot 2^{(E - (2^{x-1}-1))} \cdot \left(1 + \frac{M}{2^m}\right)$ exponent − bias		exponent bias: $2^{x-1} - 1 = 15$
	0 00001 0000000000	$= -1^0 \cdot 2^{-14} \cdot \left(1 + \frac{0}{1024}\right) \approx +0.000061$		smallest positive normal
	0 01110 0000000000	$= -1^0 \cdot 2^{-1} \cdot \left(1 + \frac{0}{1024}\right) = +0.5$		
	0 01110 1111111111	$= -1^0 \cdot 2^{-1} \cdot \left(1 + \frac{1023}{1024}\right) \approx +0.999512$		largest less than one
	0 01111 0000000000	$= -1^0 \cdot 2^0 \cdot \left(1 + \frac{0}{1024}\right) = +1$		
	1 01111 0000000000	$= -1^1 \cdot 2^0 \cdot \left(1 + \frac{0}{1024}\right) = -1$		
	0 01111 0000000001	$= -1^0 \cdot 2^0 \cdot \left(1 + \frac{1}{1024}\right) \approx +1.000977$		smallest larger than one
	0 11110 0000000000	$= -1^0 \cdot 2^{15} \cdot \left(1 + \frac{0}{1024}\right) = +32768$		
	0 11110 1111111111	$= -1^0 \cdot 2^{15} \cdot \left(1 + \frac{1023}{1024}\right) = +65504$		largest normal
	1 11110 1111111111	$= -1^1 \cdot 2^{15} \cdot \left(1 + \frac{1023}{1024}\right) = -65504$		smallest normal
special	0 11111 0000000000	$= +\infty$		infinity
	0 11111 0000000001	= signaling NaN	on x86	"not a number"
	0 11111 1000000001	= quiet NaN	on x86	"not a number"

	n	x	m
half precision	16	5	10
bfloat16	16	8	7
TensorFloat	19	8	10
fp24	24	7	16
PXR24	24	8	15
single precision / "float"	32	8	23
double precision	64	11	52
x86 extended prec.	80	15	64
quadruple precision	128	15	112
octuple precision	256	19	236