

Computational Physics 3034/3934

•**Six/Five (1h) lectures:** Monday at 12:00pm in Weeks 7–11 in Slade Lecture Theatre (Room 217) 3034 and Room 424 3934 , Physics Building.

•**Five (2h) computational labs** in Weeks 7–12/13. In these sessions, tutors will be on-hand to help you to work through the weekly (marked) quiz. Note that **this is the only mechanism for you to get help with these quizzes**—you cannot expect to call on and get help from tutors on these quizzes outside of these hours if you choose not to attend.

Assessment (20%)

•Canvas quizzes (10%)

- Each week there is a 2% canvas quiz to test your understanding of the material.
- Quiz responses are due 2 weeks after the associated lecture.

•Computational Physics Assignment (10%)

- The Computational Physics Assignment will be released in Week 11 and is due two weeks later in Week 13.
- You should submit your assignment responses via Gradescope.

Computation in physics

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

- We will learn how to develop *numerical algorithms* to solve *physical equations*.
- And how to analyze them for stability and accuracy.
- We *don't teach coding* in this course; we focus on the algorithms and how they work.

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*

Computation in physics

- Physics has theory, experiment, and *computation*.
- Undergrad physics is training in the simplest cases (where analytic techniques can often work), but in the real world, physical equations typically need to be solved numerically.
- E.g.,: **molecular dynamics, Many-body problems, describing nonlinear systems, etc.**
- Modern physics research relies heavily on computational methods.
- Solving for electric and magnetic fields in stars (Mike Wheatland).
- **Calculating diffusion of memes on online social networks** (Tristram Alexander).
- **Solving biological rhythms** (Svetlana Postnova), brain dynamics (Pulin Gong, Ben Fulcher), ...

Problems Solved Using Ordinary Differential Equations (ODEs)

Problem	Description
Simple Harmonic Motion	Describes oscillating systems like pendulums and springs
Radioactive Decay	Models the decay rate of unstable atomic nuclei over time
Population Dynamics	Models population growth in biology, sometimes applied to physical systems
Electric Circuits	Analyzes circuits with resistors, capacitors, and inductors

Problems Solved Using Partial Differential Equations (PDEs)

Problem	Description
Heat Equation	Models heat/temperature distribution in space over time
Wave Equation	Describes wave propagation (e.g., sound, light, water waves)
Schrödinger Equation	Describes time evolution of quantum states in quantum mechanics
Navier-Stokes Equations	Governs motion of fluids (liquids and gases), essential in aerodynamics

Problems Not Directly Solvable Using ODEs/PDEs

Problem	Why It's Challenging
Nonlinear Dynamics & Chaos	Highly sensitive to initial conditions, often unpredictable
Complex Quantum Systems	Many-body systems are computationally intractable with exact methods
Turbulence	Still an unsolved problem despite being governed by Navier-Stokes
Strongly Correlated Systems	Require advanced approaches beyond standard differential equations

Computational Physics: *Lecture1*

- *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-dimensionalization*
- 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.

Number Representation

Computer store numbers in binary system

- **How do we represent real numbers on a computer?**

Binary system (array of bits)

	Column 8	Column 7	Column 6	Column 5	Column 4	Column 3	Column 2	Column 1
Base ^{exp}	2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
Weight	128	64	32	16	8	4	2	1

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 2 * 2 = 4$$

$$2^3 = 2 * 2 * 2 = 8$$

$$2^4 = 2 * 2 * 2 * 2 = 16$$

$$2^5 = 2 * 2 * 2 * 2 * 2 = 32$$

$$2^6 = 2 * 2 * 2 * 2 * 2 * 2 = 64$$

$$2^7 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 128$$

Real numbers in binary system

1x2 ³	1x2 ²	0x2 ¹	1x2 ⁰	1x2 ⁻¹	0x2 ⁻²	1x2 ⁻³	1x2 ⁻⁴
1	1	0	1	.	1	0	1 1
8	4	0	1		0.5	0	0.125 0.0625
				↑			
				Binary point			
$8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 + 0.0625 = 13.6875 \text{ (Base 10)}$							

Decimal to Binary Conversion

1. Convert the Integer Part (before the decimal point)

Steps:

Use repeated division by 2 and keep track of remainders.

- Divide the integer by 2.
- Write down the remainder (0 or 1).
- Update the number to the quotient (ignore the remainder).
- Repeat until the quotient is 0.
- The binary number is the remainders read **from bottom to top**.

2. Convert the Fractional Part (after the decimal point)

Multiply the fractional part by 2. The integer part of the result is the next binary digit. Then drop the integer part and repeat.

Steps:

- Multiply the fraction by 2.
- The whole number part (0 or 1) is the next binary digit.
- Keep the fractional part and repeat.
- Stop when the fraction becomes 0 or after enough digits for your desired precision.

Examples:

$13 \div 2 = 6$ remainder 1
 $6 \div 2 = 3$ remainder 0
 $3 \div 2 = 1$ remainder 1
 $1 \div 2 = 0$ remainder 1

Binary: 1101

$0.625 \times 2 = 1.25 \rightarrow 1$
 $0.25 \times 2 = 0.5 \rightarrow 0$
 $0.5 \times 2 = 1.0 \rightarrow 1$

Binary: $.101$

$0.1 \times 2 = 0.2 \rightarrow 0$
 $0.2 \times 2 = 0.4 \rightarrow 0$
 $0.4 \times 2 = 0.8 \rightarrow 0$
 $0.8 \times 2 = 1.6 \rightarrow 1$
 $0.6 \times 2 = 1.2 \rightarrow 1$
 $0.2 \times 2 = 0.4 \rightarrow 0$
...

$\rightarrow .0001100110011\dots$ (repeating)

Floating point error (representation of numbers)

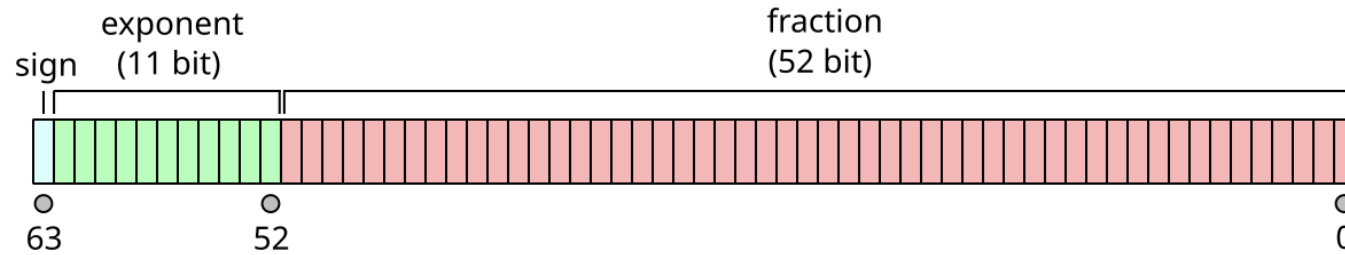
Float64 and Binary64

- *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, is represented with 64 bits as

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

- 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).
- There are infinitely many reals, but only a finite number of floating-point numbers...
- Yields *range error* and *rounding error*.

Computer store numbers in “scientific notation”



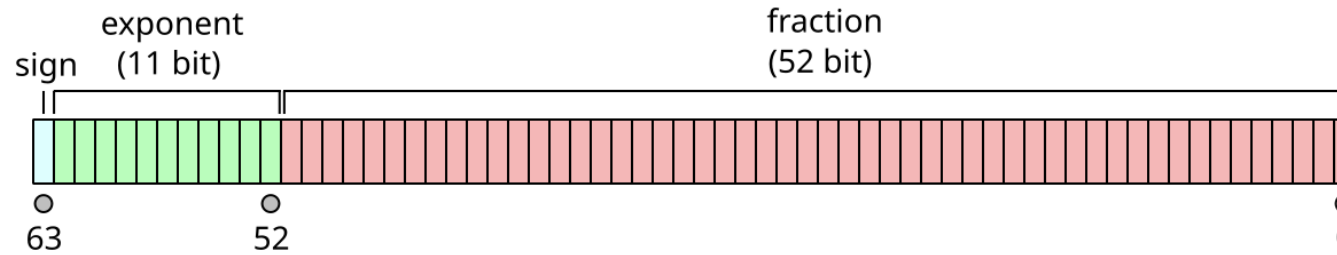
- **Sign bit (1 bit)** → Determines positive (0) or negative (1).
- **Exponent (11 bits)** → 2048 → Stored in **biased** form ($E = \text{exponent} + 1023$).
- **Mantissa (52 bits)** → Stores the significant digits (without the leading 1).

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

S=0 +

S=1 -

Computer store numbers in “scientific notation”



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

The **exponent** in binary64 is **biased** to simplify hardware implementation and to allow for both positive and negative exponents without requiring a separate sign bit.

This ensures that all exponents are stored as **positive values**, simplifying bitwise operations in hardware.

$2^{11} = 2048$ strings so you can represent numbers from 0 to 2047

one exponent bit is reserve to special cases (max exponent is 2046) and the symmetrised,

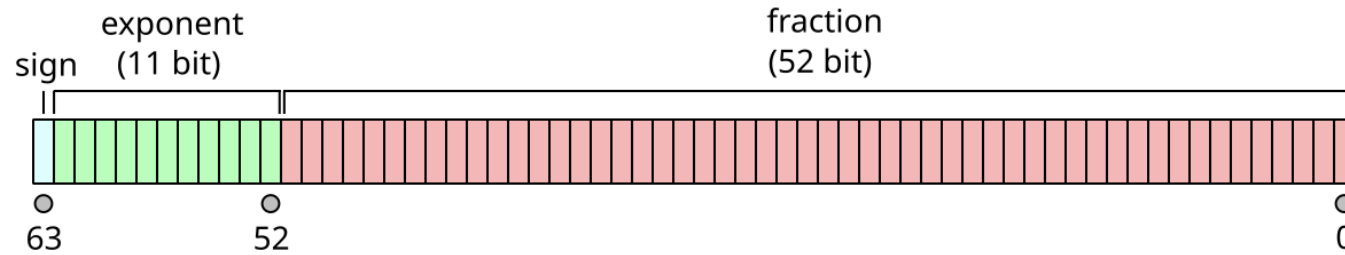
$2^{10}-1=1023$ is the maximum exponent

$$E_{\max} = 1023$$

$$E_{\text{stored}} = E_{\text{actual}} + 1023$$

- Where:
- E_{stored} is the **binary** exponent stored.
- E_{actual} is the real exponent from the floating-point representation.

Computer store numbers in “scientific notation”



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

Mantissa (significant, fraction) determines the **precision**

Smallest M indicate the distance between two consecutive numbers in binary64

$$M_{\min} = 2^{-52} = \mathbf{2.22 \times 10^{-16}}$$

$$(-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i} \right) \times 2^{e-1023}$$

Computer store numbers in “scientific notation” in binary

To better understand the binary64 to decimal conversions, use [binary64 numerical errors.ipynb](#)

- The difference is mainly in terminology: **float64** is how we use it, **binary64** is how it’s stored.

Conversion between binary and decimal involves encoding and decoding floating-point numbers using the **IEEE 754 standard**



Special Strings

Value	Sign	Exponent (bin)	Fraction (bin)	Hex
+0.0	0	00000000000	000...000	0000000000000000
-0.0	1	00000000000	000...000	8000000000000000
$+\infty$	0	11111111111	000...000	7FF0000000000000
$-\infty$	1	11111111111	000...000	FFF0000000000000
NaN	0	11111111111	\neq 000...000	7FF8000000000000 (example)

Overflow and Underflow

- **Overflow:** Number is *too large* for binary64.
- In **sys** and **numpy** Max Float is $(2-2^{-52}) \times 2^{1023}$. which is **1.79e+308**
- **sys.float (Native Python) info.max (Numpy)** in Python: the maximum number that can be represented in binary64.

- **Underflow:** Number is *too small* for binary64.
- In **sys** **2.22e-324**
- In **Numpy** **2.22e-308**
- **sys.float_info.min** in Python: the minimum number that can be represented in binary64.

- **Machine epsilon:** the fractional spacing between numbers, eps.
- **sys.float_info.epsilon** The final (52nd) bit of the significand: $2^{(-52)}$.

Concept	Python Value (approx)	Notes
float_info.max	1.7976931348623157e+308	Max representable double
float_info.min	2.2250738585072014e-308	Min normalised double
Smallest subnormal	5e-324	Min positive non-zero value
float_info.epsilon	2.220446049250313e-16	Machine epsilon (precision limit)

Rounding error

Binary-to-Decimal Rounding Error: Key Examples

Decimal	Binary (approx.)	Stored Float64	Exact?
0.1	0.0001100110011...	0.100000000000000000555	✗
0.2	0.001100110011...	0.2000000000000000001110	✗
0.3	0.010011001100...	0.2999999999999999998890	✗
0.5	0.1	0.5	✓
0.25	0.01	0.25	✓
1.1	1.0001100110011...	1.1000000000000000008882	✗
0.1+0.2	sum of imprecise values	0.3000000000000000004	✗

Not a number NaN

- *Non-numbers*: Me is *too wild* for binary64:
- NaN stands for 'not a number'.
- Example: 0/0 doesn't have a numeric answer.
- NaN is also used when a string is stored instead of a number
- NaNs 'propagate': $\text{NaN} + x$ gives NaN for all x .

Common Binary Representations of Numeric Values

h/cpp hackingcpp.com

Unsigned Integers

n bit Positional Binary

$$\text{value} = d_{n-1} \cdot 2^{n-1} + \dots + d_1 \cdot 2^1 + d_0 \cdot 2^0$$

Example: $n = 8$

0000 0000	= 0
0000 0001	= 2^0 = 1
0000 0010	= 2^1 = 2
0000 0011	= $2^1 + 2^0$ = 3
0000 0100	= 2^2 = 4
⋮	⋮
1000 0000	= 2^{n-1} = 128
⋮	⋮
1111 1111	= $2^n - 1$ = 255

Signed Integers

n bit Two's Complement

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

Example: $n = 8$

0111 1111	= $2^{n-1} - 1$ = +127
⋮	⋮
0000 0010	= +2
0000 0001	= +1
0000 0000	= ±0
1111 1111	= -1
1111 1110	= -2
⋮	⋮
1000 0000	= -2^{n-1} = -128

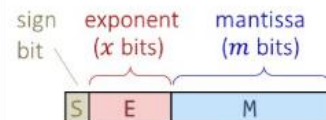
Negation:

1. invert

2. add 1

Floating-Point Numbers

IEEE 754 Format and derivatives



$$n = (1 + x + m) \text{ bits}$$

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	$= -1^0 \cdot 2^{-14} \cdot \left(0 + \frac{0}{1024}\right) = +0$
	1 00000 0000000000	$= -1^1 \cdot 2^{-14} \cdot \left(0 + \frac{0}{1024}\right) = -0$
	0 00000 0000000001	$= -1^0 \cdot 2^{-14} \cdot \left(0 + \frac{1}{1024}\right) \approx +0.0000000596$
	0 00000 1111111111	$= -1^0 \cdot 2^{-14} \cdot \left(0 + \frac{1023}{1024}\right) \approx +0.0000609756$
	S 0 : 01 M	$= (-1)^S \cdot 2^{(E - (2^{x-1}-1))} \cdot \left(1 + \frac{M}{2^m}\right)$
	S 1 : 10 M	$= (-1)^S \cdot 2^{(E - (2^{x-1}-1))} \cdot \left(1 + \frac{M}{2^m}\right)$
	0 00001 0000000000	$= -1^0 \cdot 2^{-14} \cdot \left(1 + \frac{0}{1024}\right) \approx +0.000061$
	0 01110 0000000000	$= -1^0 \cdot 2^{-1} \cdot \left(1 + \frac{0}{1024}\right) = +0.5$
	0 01110 1111111111	$= -1^0 \cdot 2^{-1} \cdot \left(1 + \frac{1023}{1024}\right) \approx +0.999512$
	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	$= -1^0 \cdot 2^0 \cdot \left(1 + \frac{1}{1024}\right) \approx +1.000977$
	0 11110 0000000000	$= -1^0 \cdot 2^{15} \cdot \left(1 + \frac{0}{1024}\right) = +32768$
	0 11110 1111111111	$= -1^0 \cdot 2^{15} \cdot \left(1 + \frac{1023}{1024}\right) = +65504$
	1 11110 1111111111	$= -1^1 \cdot 2^{15} \cdot \left(1 + \frac{1023}{1024}\right) = -65504$
normal	0 11111 0000000000	= +∞
	0 11111 0000000001	= signaling NaN
special	0 11111 1000000001	= quiet NaN

on x86

	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

negative zero

smallest positive subnormal $2^{2-m-2^{x-1}}$

largest subnormal $2^{-2^{x-1}+2} \cdot \frac{2^m - 1}{2^m}$

exponent bias: $2^{x-1} - 1 = 15$

smallest positive normal $2^{-2^{x-1}+2}$

largest less than one $\frac{1}{2} + \frac{2^m - 1}{2^{m+1}}$

smallest larger than one $\left(1 + \frac{1}{2^m}\right)$

largest normal $2^{2^{x-1}-1} \cdot \left(1 + \frac{2^m - 1}{2^m}\right)$

smallest normal

infinity

"not a number"

Truncation errors

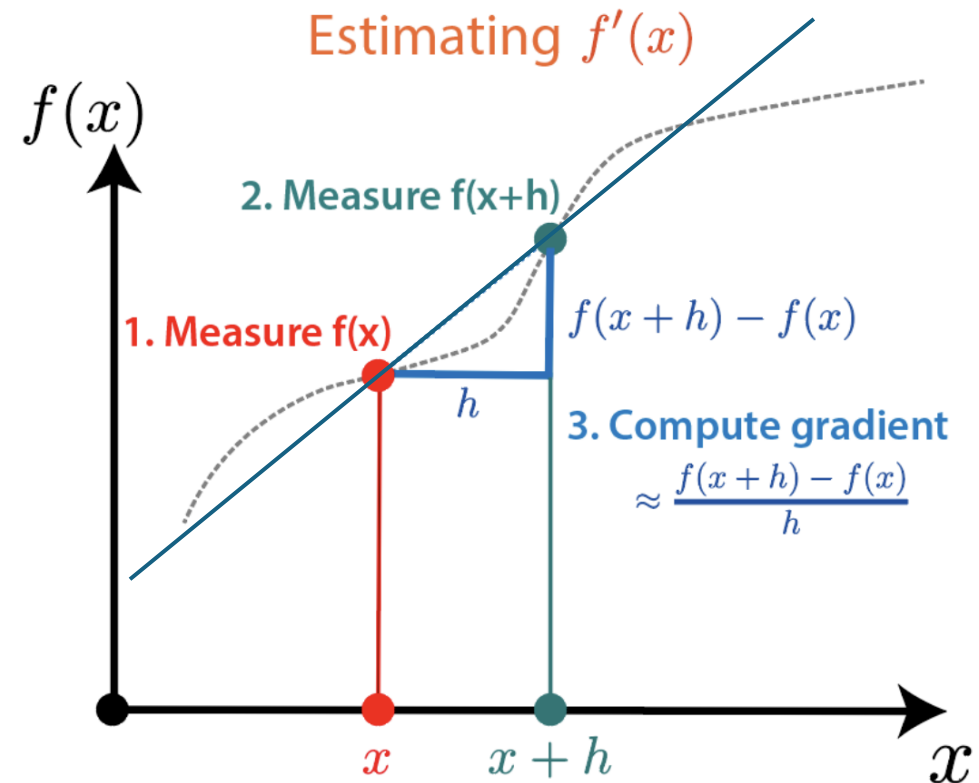
- When we write down algorithms to solve equations, they make numerical approximations of quantities like derivatives.
- The approximations in these numerical methods lead to another type of error, called *truncation error*.
- Example: *estimating the derivative* numerically, where we can evaluate for any input, .
- What would be your strategy ('algorithm', 'numerical method')?

Estimating Derivatives

The Forward difference approximation

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

- $f'(x) \approx \frac{f(x+h) - f(x)}{h}$.
 1. Measure $f(x)$.
 2. Measure $f(x+h)$, for some (small) increment, h .
 3. Compute the gradient.
- If h is small enough such that the function doesn't vary too much over h , this should be a good numerical approximation.

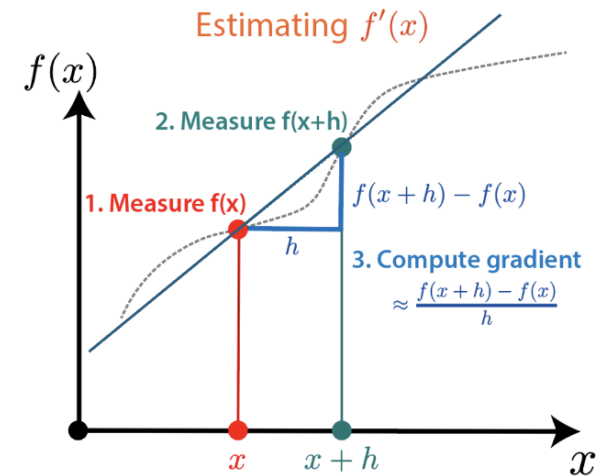


Truncation error: forward difference approximation

$$f(x+h) = \sum_{n=0}^{\infty} \frac{h^n}{n!} f^{(n)}(x)$$

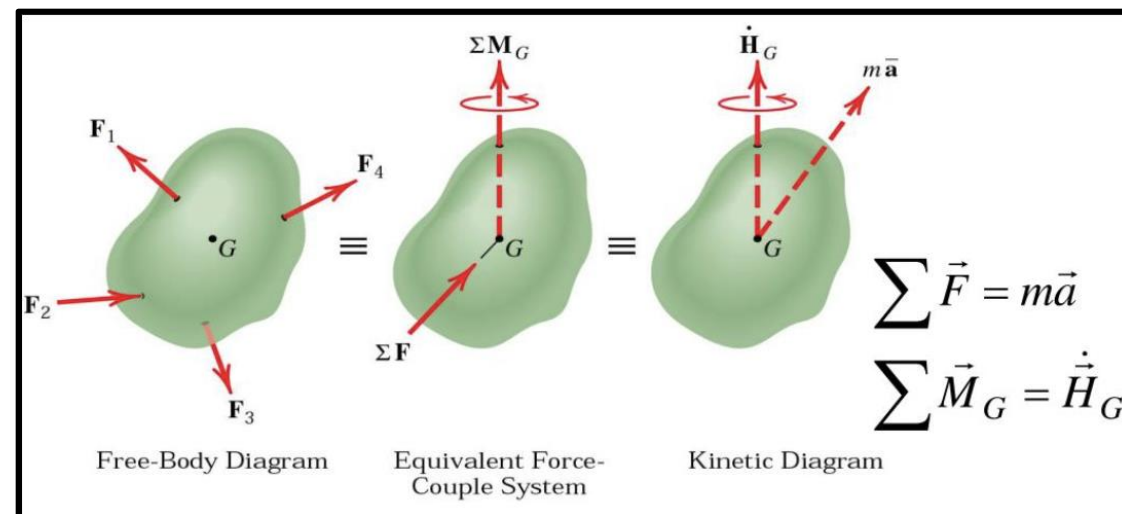
$$R_n = O(h^{n+1})$$

- Consider the Taylor series $f(x+h) = f(x) + hf'(x) + \frac{1}{2!}h^2f''(x) + \dots$
- Rearranging for $f'(x)$ gives: $f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{1}{2}hf''(x) + \dots$
- The largest error term is proportional to h and we write
$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h).$$
 - We say that the local truncation error is *of order h* .
 - We *truncate* the full series when terms become proportional to (small) h
 - This *truncation error* limits accuracy (how close you are to the correct answer).
 - And typically *dominates floating-point errors*



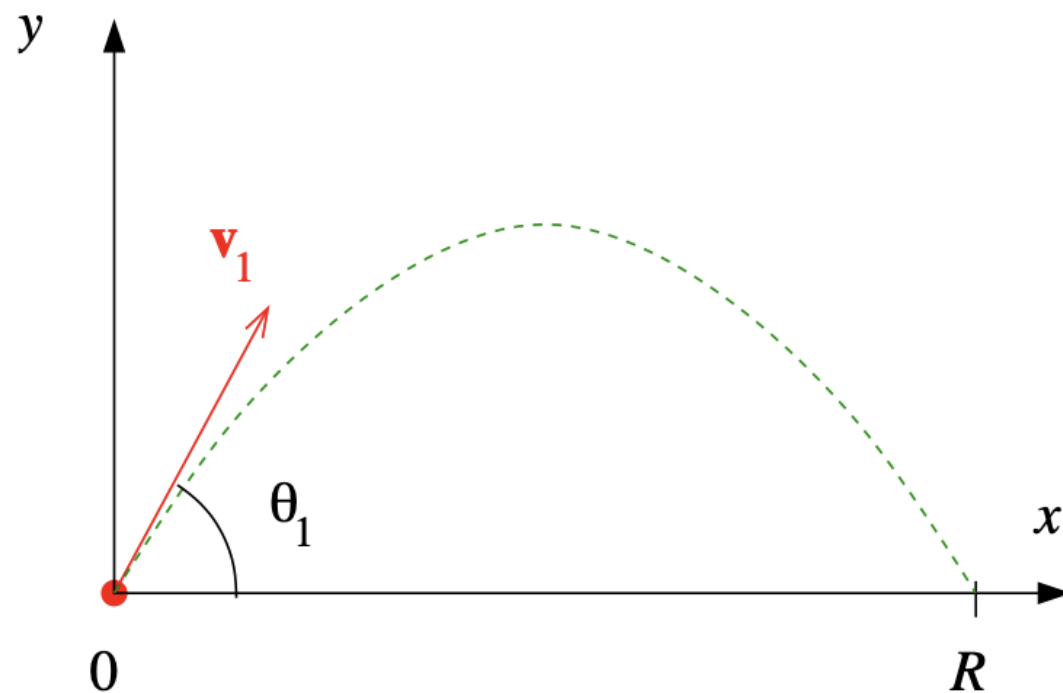
Dynamics problems: solving the motion of a particle

- $\frac{d\mathbf{r}}{dt} = \mathbf{v}$, 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...



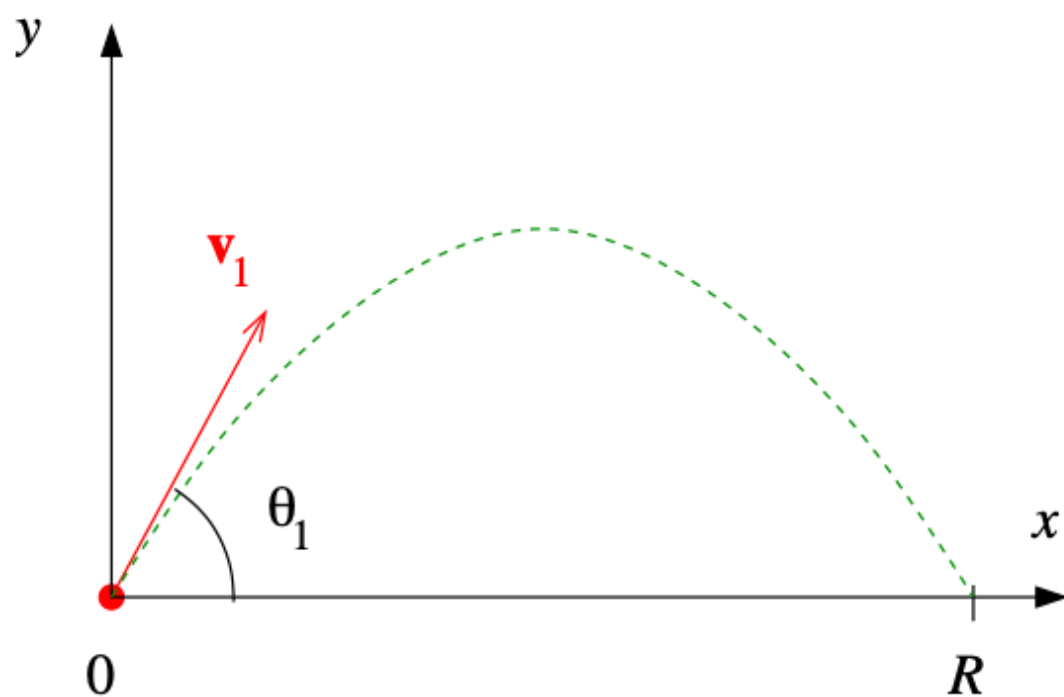
Example: Simple projectile motion

- 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 $\mathbf{v}(0) = \mathbf{v}_1$:
 - $\mathbf{r} = \mathbf{r}_1 + \mathbf{v}_1 t - \frac{1}{2}gt^2\hat{\mathbf{y}}$
 - $\mathbf{v} = \mathbf{v}_1 - gt\hat{\mathbf{y}}$



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-dimensionalizing equations

- Physical equations use *dimensional* variables:
 - Time (s), position (m), velocity (ms^{-1}) have physical units.
 - Constants can also be dimensional: e.g., $g = 9.8 \text{ ms}^{-2}$.
- In numerical work, it's simpler to *non-dimensionalize*
 - *The goal*: variables and constants in our equations have *no units*.
- *How?*: We rescale dimensional variables to *non-dimensional* (ND) ones by re-scaling.
 - 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-dimensionalizing: 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-dimensionalizing projectile motion equations

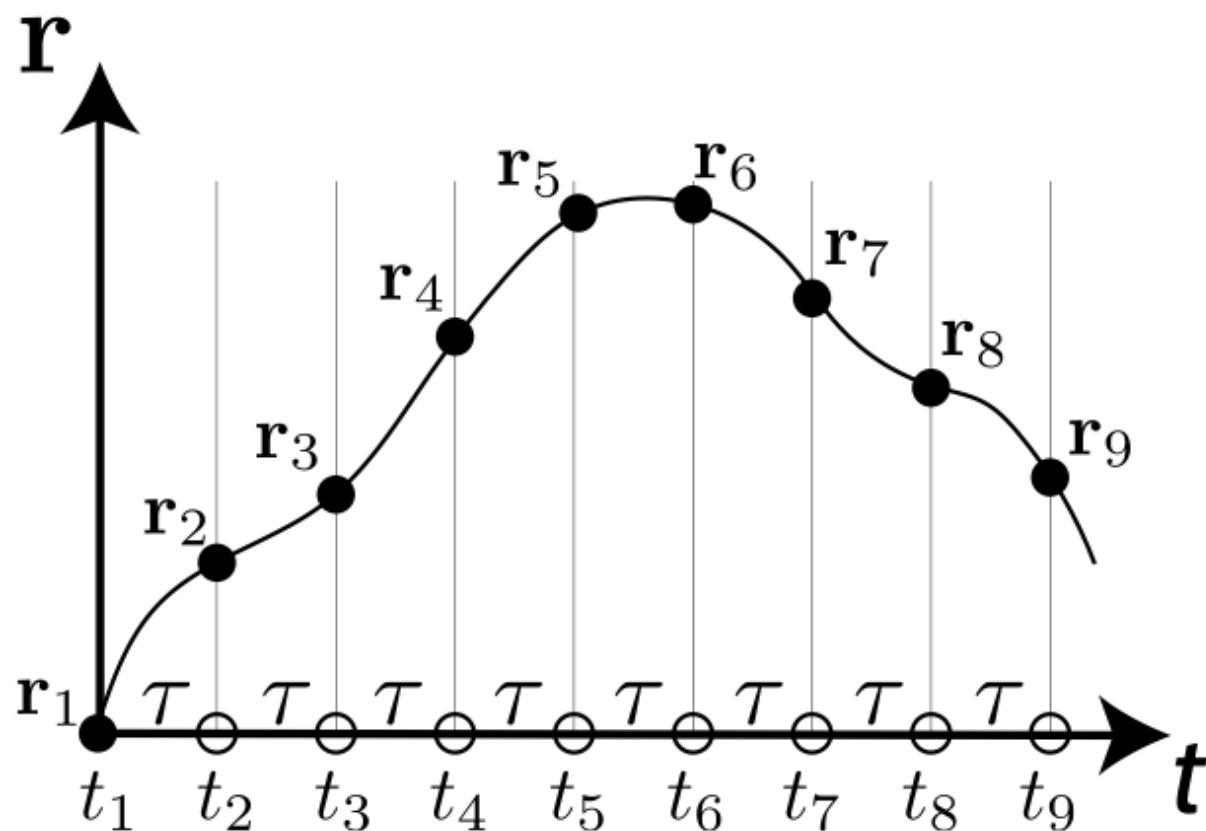
- **Velocity:** $\frac{d\mathbf{r}}{dt} = \mathbf{v} \Rightarrow \frac{L_s}{t_s} \frac{d\bar{\mathbf{r}}}{d\bar{t}} = \frac{L_s}{t_s} \bar{\mathbf{v}} \Rightarrow \frac{d\bar{\mathbf{r}}}{d\bar{t}} = \bar{\mathbf{v}}$
- **Acceleration:** $\frac{d\mathbf{v}}{dt} = -g\hat{\mathbf{y}} \Rightarrow \frac{L_s}{t_s^2} \frac{d\bar{\mathbf{v}}}{d\bar{t}} = -g\hat{\mathbf{y}} \Rightarrow \frac{d\bar{\mathbf{v}}}{d\bar{t}} = -\frac{gt_s^2}{L_s}\hat{\mathbf{y}}$
- Our constants occur 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 derivative, 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 $\frac{d^2}{dt^2} = \frac{d^2}{d(\bar{t}t_s)^2} = \frac{1}{t_s^2} \frac{d^2}{d\bar{t}^2}$

Discretizing time

- Numerical methods typically treat time across a *discrete grid*.
- We evaluate at a set of times, $t_n = (n - 1)\tau$ ($n = 1, 2, 3 \dots$)
 - τ is the *time step*.
- The projectile motion equations, $\frac{d\mathbf{r}}{dt} = \mathbf{\bar{v}}$, $\frac{d\mathbf{\bar{v}}}{dt} = -\hat{\mathbf{y}}$, 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 🏃



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.

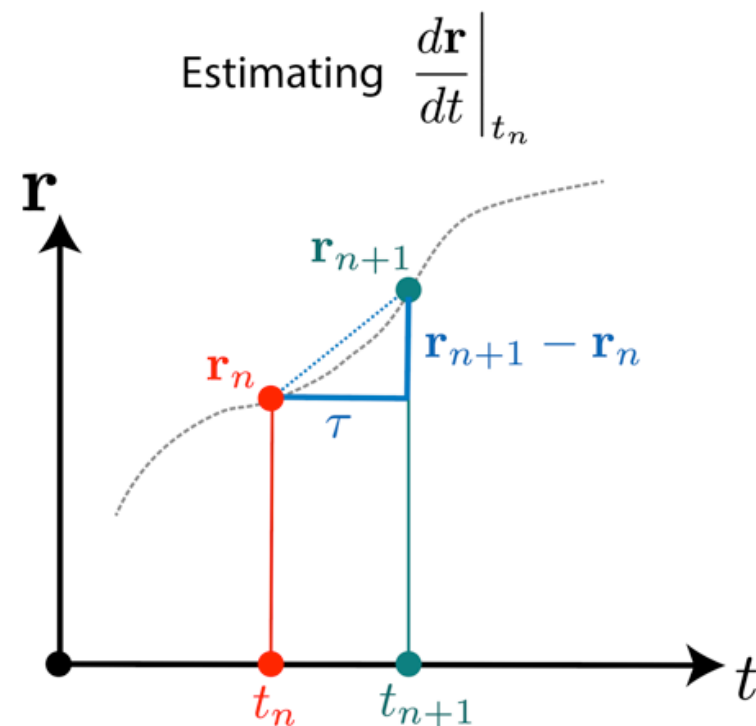


Time derivative: the forward-difference approximation

$$f'(t) = \frac{f(t + \Delta t) - f(t)}{\Delta t} + O(\Delta t)$$

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

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



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

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

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

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

or:

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

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

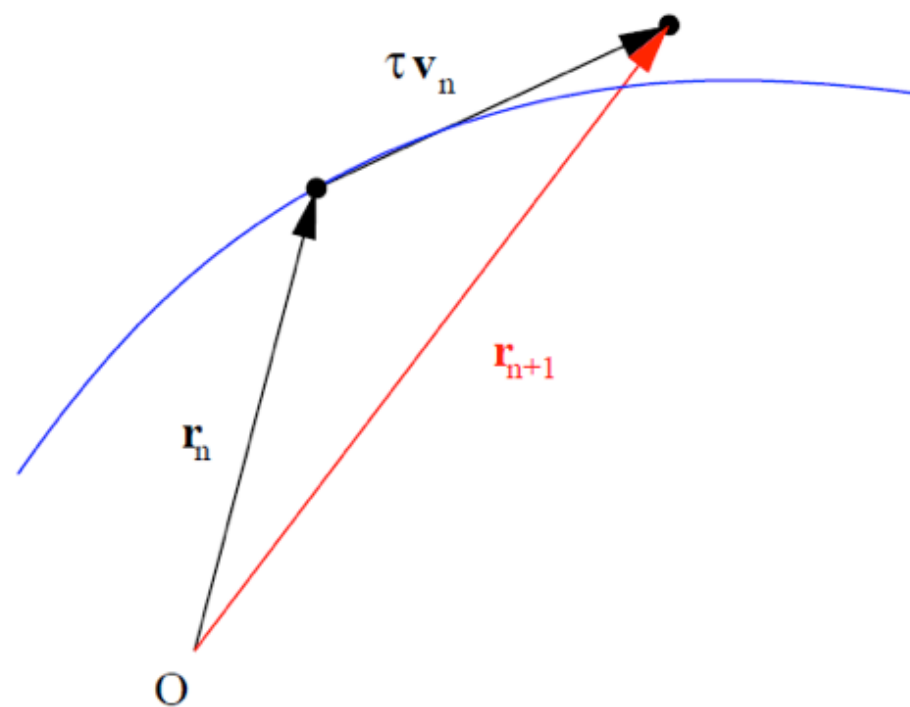
☆☆ Euler's Method for Dynamics

- Excludes terms of order τ^2 :
 - $$\begin{aligned}\mathbf{r}_{n+1} &= \mathbf{r}_n + \tau \mathbf{v}_n, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \tau \mathbf{a}_n.\end{aligned}$$
- Yields a simple rule for stepping forward in time 🏃:
 - $(\mathbf{r}_1, \mathbf{v}_1) \xrightarrow{\tau} (\mathbf{r}_2, \mathbf{v}_2) \xrightarrow{\tau} \dots$
- 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 it 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



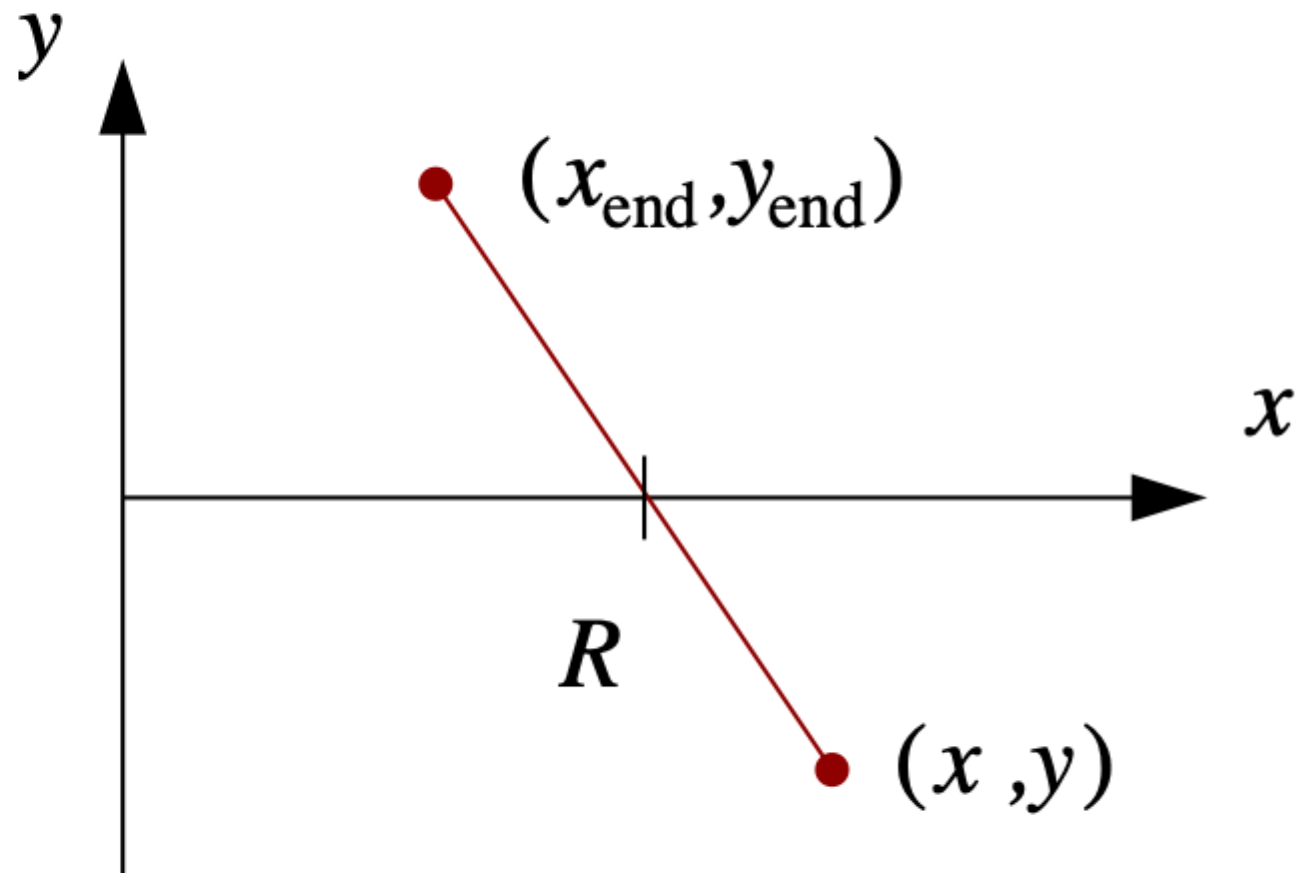
Euler's method for simple projectile motion,

proj_Euler.ipynb and Euler_ODE.ipynb

- Simple *projectile motion* has $\mathbf{a} = -\hat{\mathbf{y}}$, so our Euler algorithm for solving it is:
 - $$\begin{aligned}\mathbf{r}_{n+1} &= \mathbf{r}_n + \tau \mathbf{v}_n, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n - \tau \hat{\mathbf{y}}.\end{aligned}$$
- The code `proj_Euler.ipynb` 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_Eulernb.ipynb 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

- 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:

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

- 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, $g \approx 10 \text{ m s}^{-2}$, $\tau \approx 0.03 \text{ s}$, $T \approx 2 \text{ s}$:
 - global error $\approx \frac{1}{2} \times 2 \times 0.03 \times 10 \text{ m} \approx 30 \text{ cm}$
- This is consistent with the observed error in R .

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).

Quizzes

- **Project Notebook:**

Modify the proj_Euler.ipynb notebook with your answers.

- **Submission:**

Enter your responses on Canvas.

- **Additional Resource:**

An extra notebook is available (Euler_ODE.ipynb) to help you better understand the Euler model, including discussions on common numerical errors

Lab Computer Tutorial

- We will work through the implementation of projectile motion together.
- The session will also provide guidance to help you answer the quiz questions.

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.

← → | **B** *I* U | A ▾ **A** ▾ *I_x* | ≡ ≡ ≡ | \times^2 \times_2 ⋮ $\frac{1}{x}$ |
12pt ▾ Paragraph ▾ |     **f_x**

Computational Physics Practice Quiz 2

Quiz Type	Practice Quiz
Points	50
Shuffle Answers	Yes
Time Limit	No Time Limit
Multiple Attempts	Yes
Score to Keep	Highest
Attempts	Unlimited
View Responses	Always
Show Correct Answers	Immediately
One Question at a Time	No
Require Respondus LockDown Browser	No
Required to View Quiz Results	No
Webcam Required	No

This Will NOT count for the final mark!