Computation Basics I

Lecture 2, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 9/22/2022

Announcements

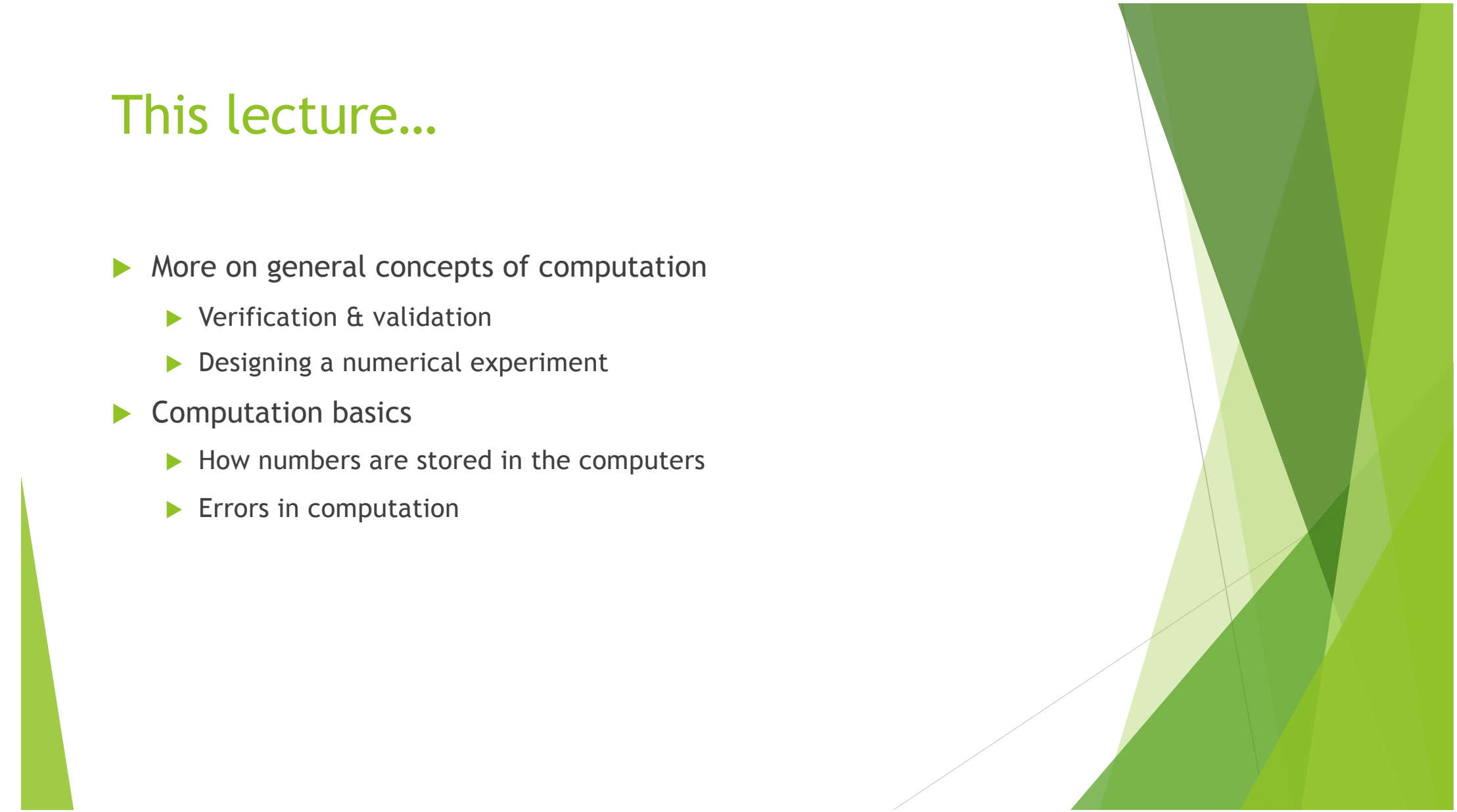
- ***HW1*** will be posted on eLearn TODAY. ***Due next Thursday (9/29) at 14:20***. Late submission within one week will receive ***75%*** of the credit
- Don't forget to reply to the form (回饋單) in the TA session on eLearn to get bonus credits if you (i) asked questions, or (ii) submitted in-class exercises
- If you're not sure what to do for the term project, please feel free to ask the instructor/TAs during the office hours

Previous lecture…

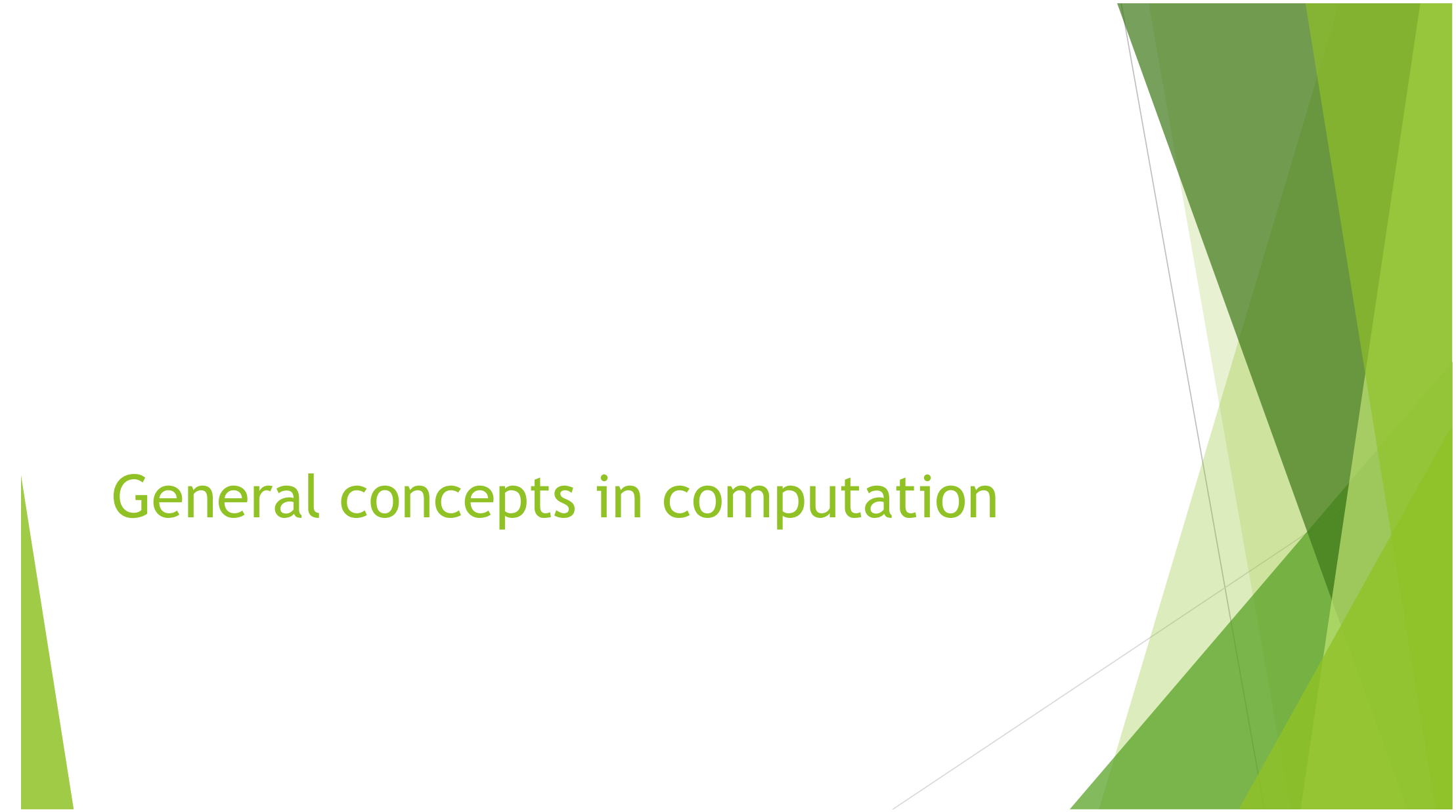
- Computational astrophysics has become one of the three major fields in astrophysical research; understanding the underlying techniques is essential for both computational/theoretical/observational astrophysicists
- Intro to the environments: Unix-like systems, CICA cluster
- Rules of thumb for computing:
  - Good habits will go a long way
  - Plan ahead before you code it
  - Your code is not done until you test it

This lecture…

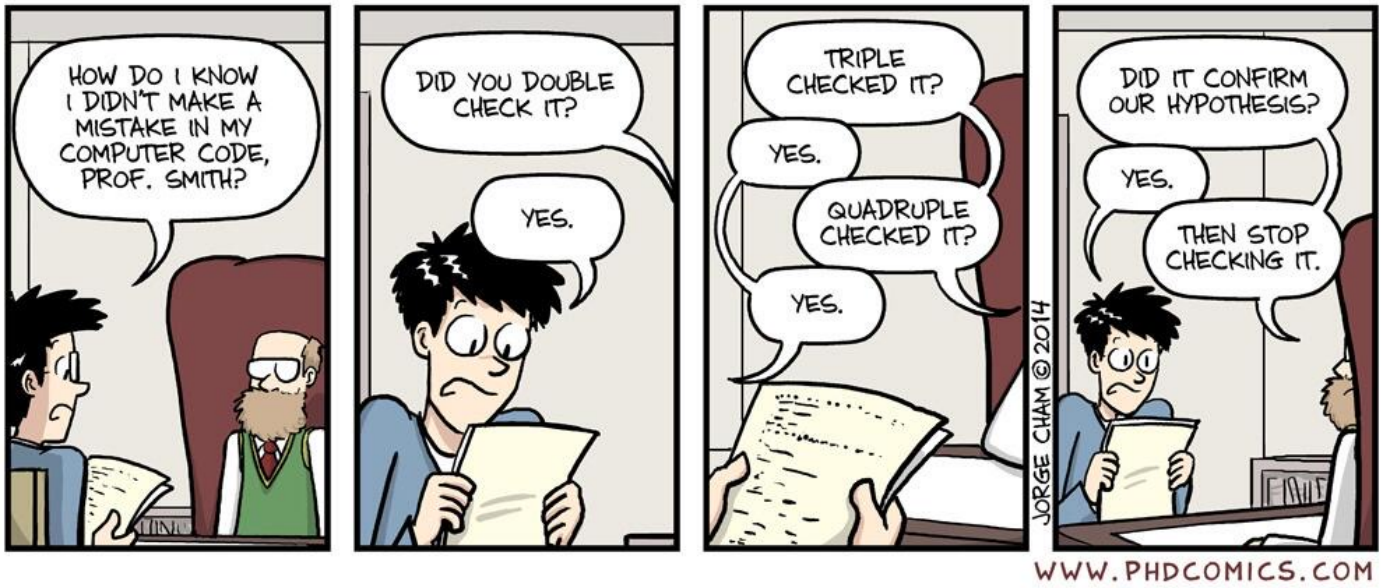
- More on general concepts of computation
  - Verification & validation
  - Designing a numerical experiment
- Computation basics
  - How numbers are stored in the computers
  - Errors in computation



General concepts in computation



Computing rule of thumbs #3:
Your code is not done until you test it

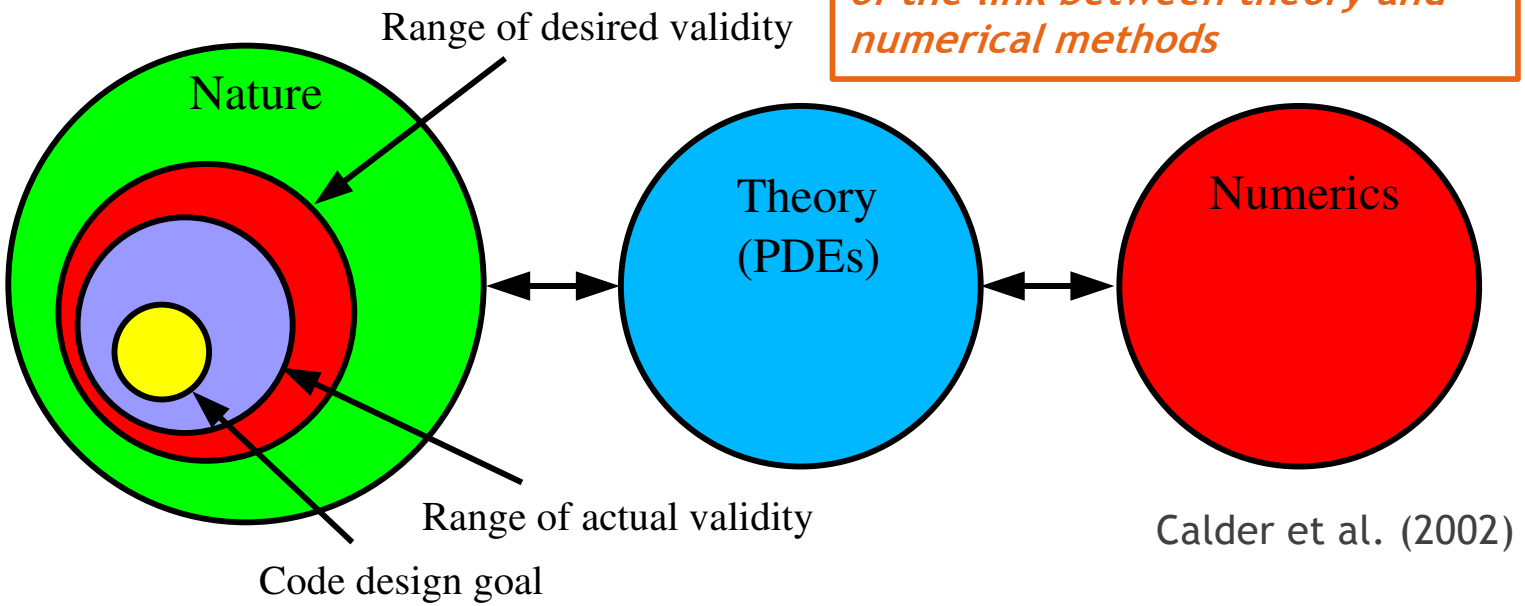


- Simulations are notorious for "*garbage in, garbage out*"
- "*Every code has a bug!*" – Never assume your code is correct before you test it
- Always *verify* and *validate* the code – and this is usually the more time-consuming part!
- Testing using known physics and known solutions

How to make sure we are getting the right answers?

- *Verify* the code
  - To determine that a model implementation accurately represents the developer's conceptual description and solution of the model
  - "*Solving the equations right*"
- *Validate* the code
  - To determine how well a model is an accurate representation of the real world for its intended use
  - In other words, computation is based on "*approximations*" of the real world using "*simplified assumptions*", and we have to know how good our approximations are
  - "*Solving the right equations*"

Verification & validation

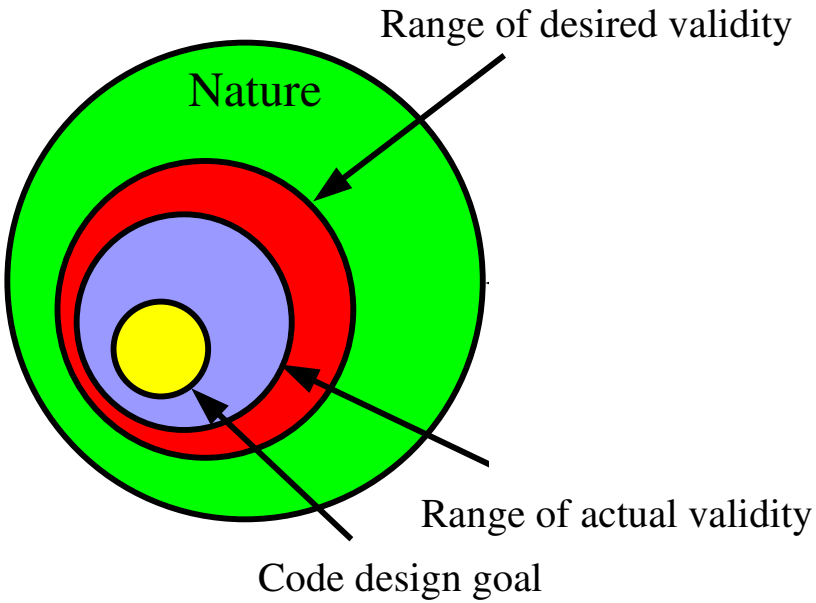


Verification probes the reliability of the link between theory and numerical methods

Validation probes the range of actual validity of the theory

Calder et al. (2002)

Validation is to examine to what extent your numerical experiment is applicable/valid



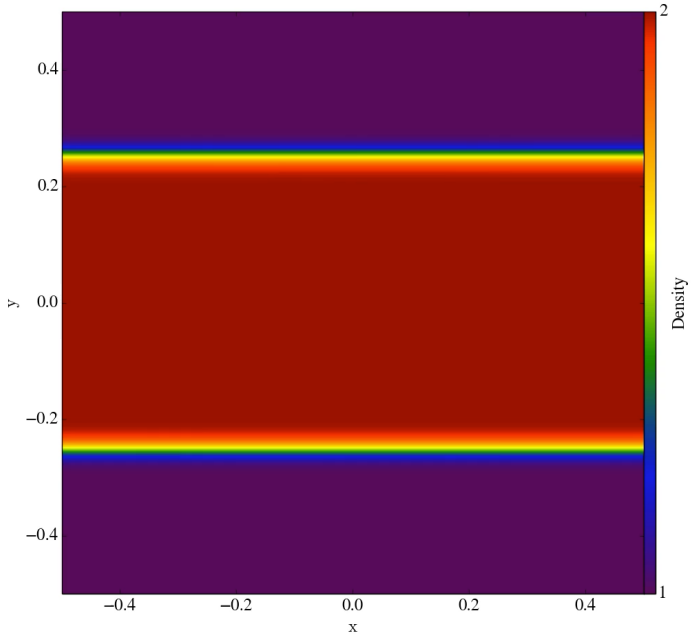
Example: cosmological hydro simulations

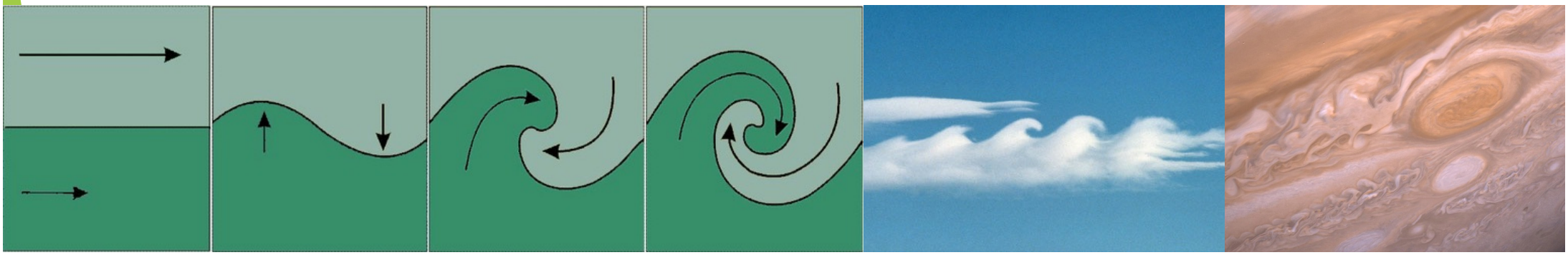
- ***Code design goal***: study the effects of AGN feedback on galaxy formation
- ***Range of desired validity***: galaxies in the universe
- ***Range of actual validity?***
  - Because it's hydro, the simulation cannot probe objects which are not well described by idealized fluids
  - Unable to probe systems with strong B fields, strong gravity, etc...
- ***Choose the numerical tool wisely and know its limitations (simulations /= truth)!!!***

Verification – test your code with known physics/analytical solutions!

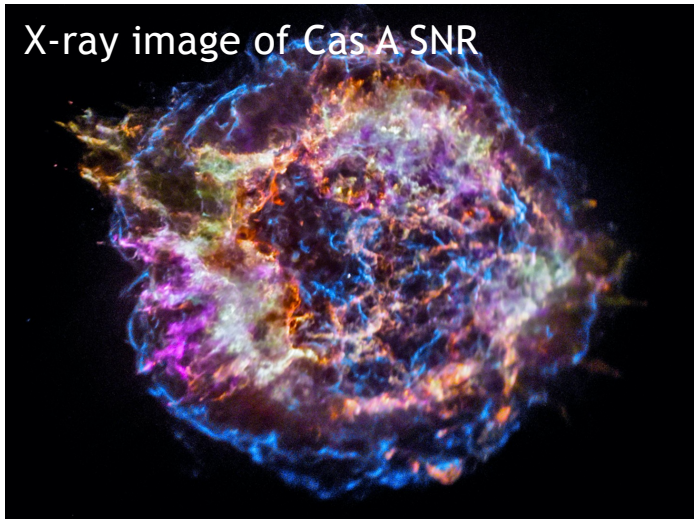
Example #1: Kelvin-Helmholtz instability

- Interface between two fluids with different densities and velocities is unstable to perturbations
- Often produces wave/ripple-like structures
- Analytical solutions for growth rate are available for comparison



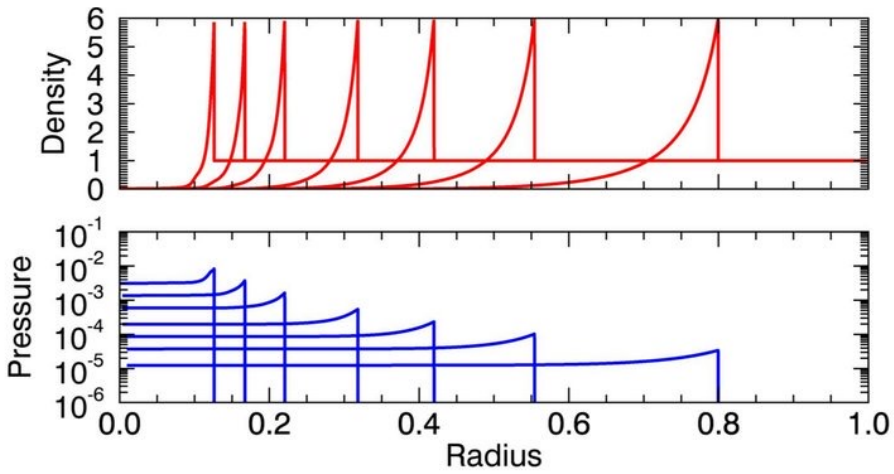


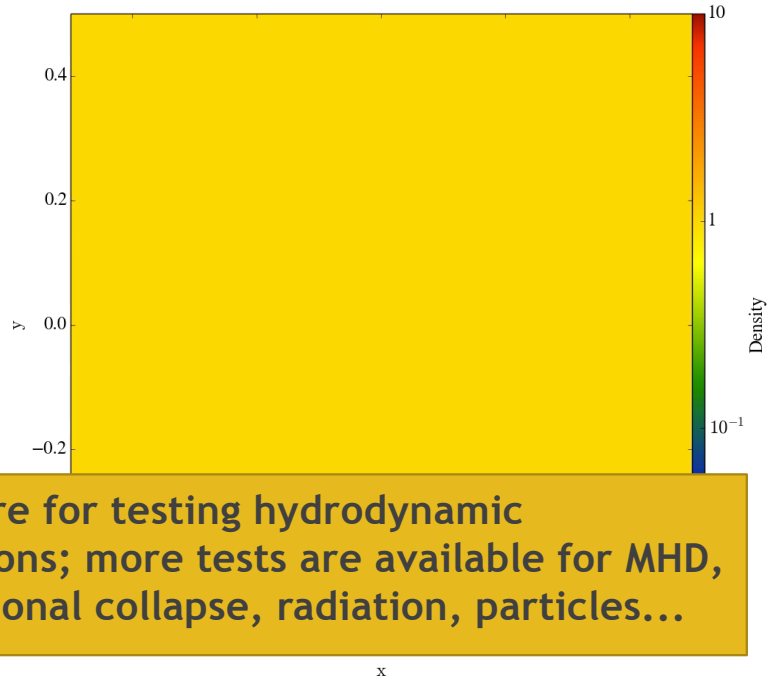
Verification – test your code with known physics/analytical solutions!



Example #2: Sedov explosion

- Propagation of shock waves driven by a powerful energy injection
- Applicable to the energy-conserving phase of supernova explosions
- Analytical solution for R(t) available for comparison





These are for testing hydrodynamic simulations; more tests are available for MHD, gravitational collapse, radiation, particles...

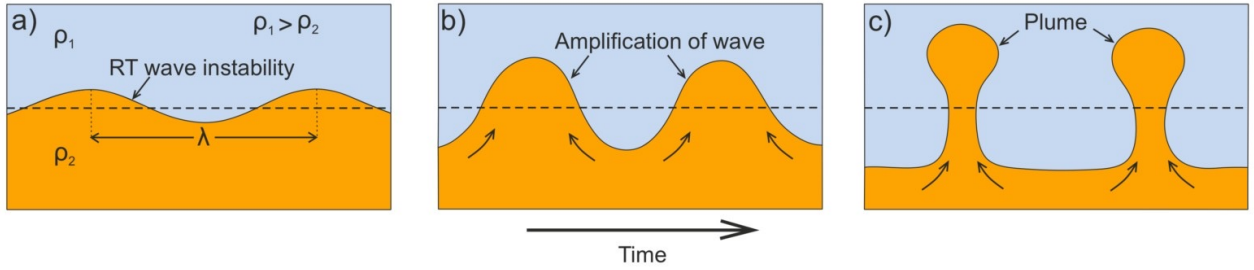
Verification tests – things to consider

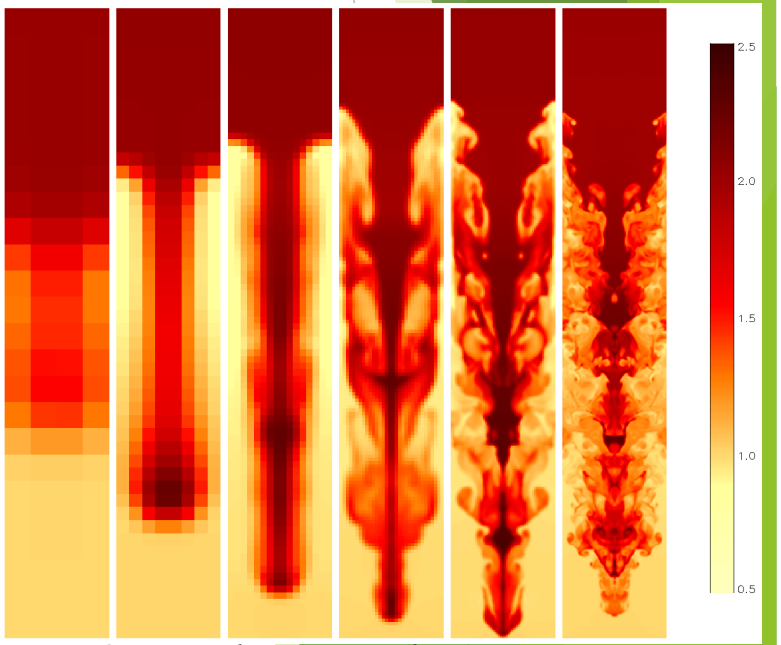
- Start from simplified models
- Test the code in the intended physical limits (e.g., high/low Mach number, with/no gravity, with/no diffusion...)
- Test symmetries (e.g., translation, rotation, Galilean/Lorentz transformations)
- Test the ***sensitivity*** of the code to the knobs/switches in the numerical algorithms (e.g., ***stability*** criteria, ***convergence***...)
  - CFL number
  - *N* in iterative methods or particle-based methods
  - Grid resolution in mesh-based methods

Convergence test on resolution

Example #3: Rayleigh-Taylor instability

- Interface between two fluids with different densities when the lighter fluid is pushing the heavier fluid
- Often produces plume/finger-like structures
- Analytical solutions for growth rate are available for comparison





λ_{eff} = 4 8 16 32 64 128 zones

cells/wavelength

Calder et al. (2002)

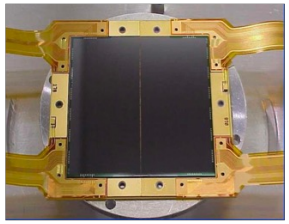
Designing a numerical experiment

What people think computational astrophysicists do

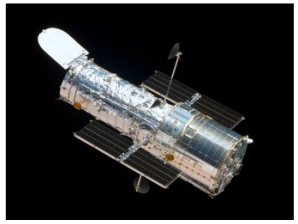
Computation is more than just pressing a button!!!

Analogies between observations & computation

Astrophysical object



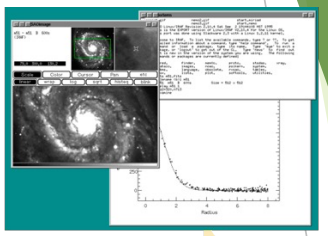
Detectors



Telescope

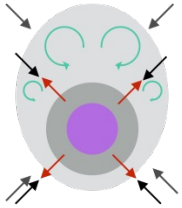
Observation

DATA

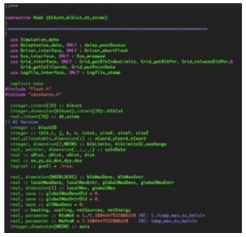


Data reduction / Calibration

Designing your obs/sims is crucial otherwise you would likely get garbage or you couldn't address the problem posed!



Mathematical Model

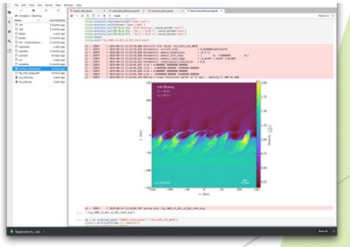


Codes

Supercomputer

Simulation

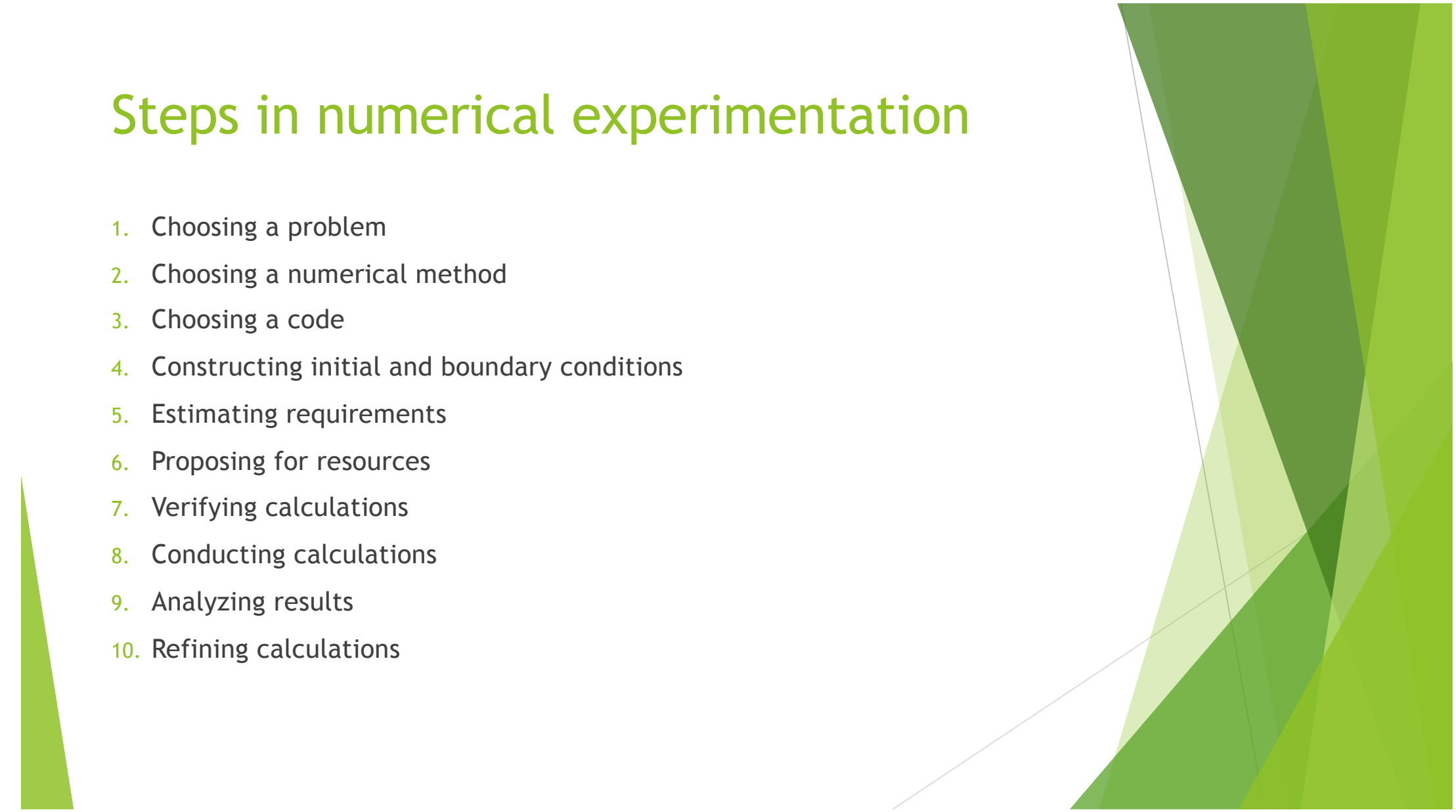
DATA



Data analysis / Visualization

Steps in numerical experimentation

1. Choosing a problem
2. Choosing a numerical method
3. Choosing a code
4. Constructing initial and boundary conditions
5. Estimating requirements
6. Proposing for resources
7. Verifying calculations
8. Conducting calculations
9. Analyzing results
10. Refining calculations



Steps in numerical experimentation

This is what we will be practicing for the term project!!!

1. Choosing a problem
2. Choosing a numerical method
3. Choosing a code
4. Constructing initial and boundary conditions
5. Estimating requirements
6. Proposing for resources
7. Verifying calculations
8. Conducting calculations
9. Analyzing results
10. Refining calculations

Midterm project proposal (preparation/planning phase)

Final project presentation (production/analysis phase)

1. Choosing a problem

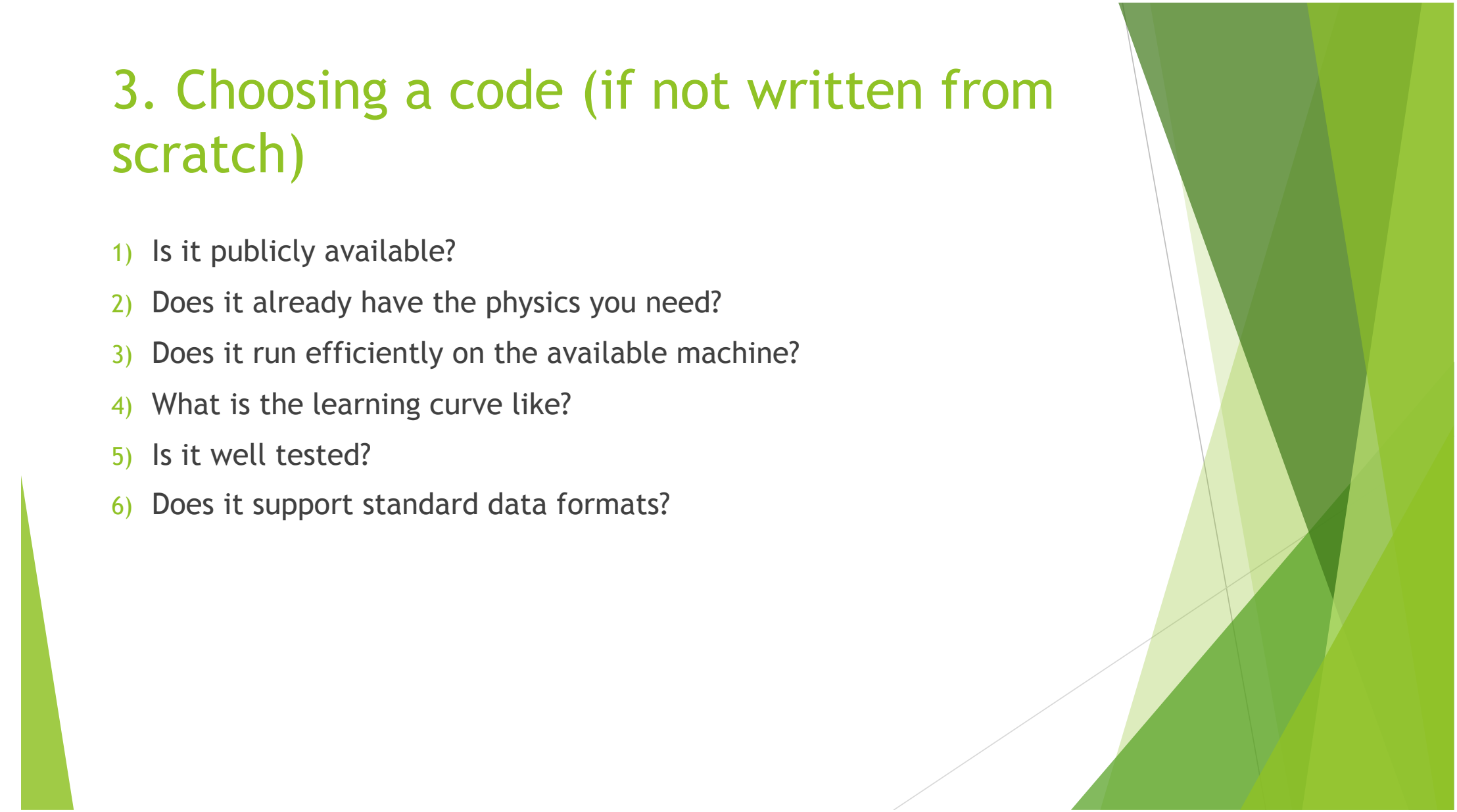
1) ***Know the state of the art*** (literature search)
2) Do ***back-of-the-envelope calculations*** (e.g., orders of magnitude estimates of energies, timescales, etc)
3) ***Determine the goals of numerical experimentation (a good numerical experiment should be able to address one of the following)!!!***
   - ***Theory testing*** – can theory X produce effect Y?
   - ***Parameter estimation*** – what is theoretical expectation for observed value of quantity Q and its errors?
   - ***Sensitivity analysis*** – what input parameters are most important for determining the outcome?
   - ***Physical insight*** – what physical models/mechanisms must be included?
   - ***Numerical insight*** – how do numerical issues (e.g., resolution) affect results?

2. Choosing a numerical method

1) ***Determine the solvers needed***
   - What physics is needed?
   - Can existing algorithms handle the necessary physics?
   - Will new algorithms need to be developed?
2) ***Determine feasibility***
   - What parts of solution can be solved directly and what parts must/can be put in by hand (e.g., subgrid physics, dynamic vs. fixed gravitational potential)?
   - Do the available methods perform well on available machines?
   - What were the requirements of the most similar published work?

3. Choosing a code (if not written from scratch)

1) Is it publicly available?
2) Does it already have the physics you need?
3) Does it run efficiently on the available machine?
4) What is the learning curve like?
5) Is it well tested?
6) Does it support standard data formats?



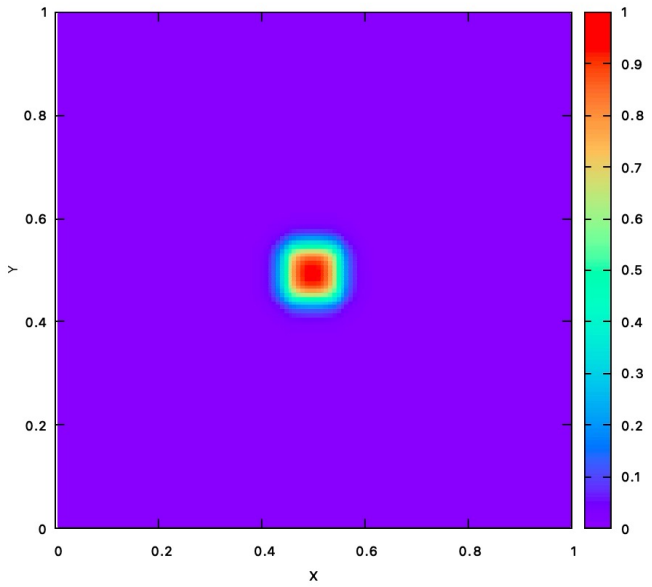
4a. Constructing initial conditions

- Map 1D profiles onto 2D/3D grids (e.g., cluster mergers, SN explosions)
- Deal with sharp features/gradients (smoothing, using adequate resolution)
- Add perturbations / Gaussian random fields
  - Numerical noise can seed perturbations
  - Better to control the initial perturbations (e.g., inverse Fourier transform from a given power spectrum)

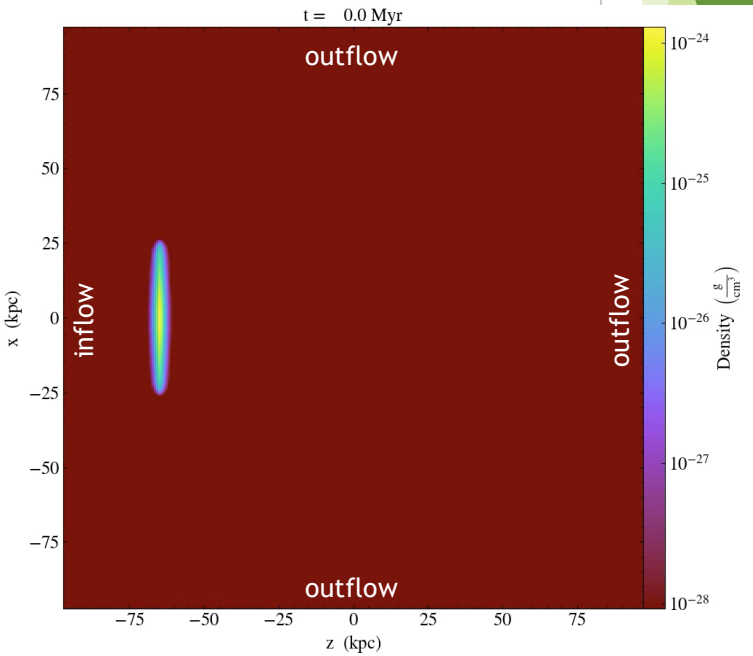
4b. Constructing boundary conditions (B.C.)

- Choose an appropriate boundary condition (e.g., ***periodic, reflect, outflow, inflow, diode***/one-way, arbitrary...)

Example #1: 2D advection with periodic B.C.

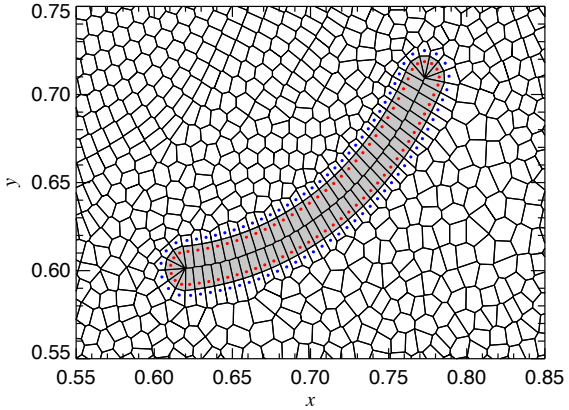


Example #2: Ram-pressure stripping with inflow/outflow B.C.

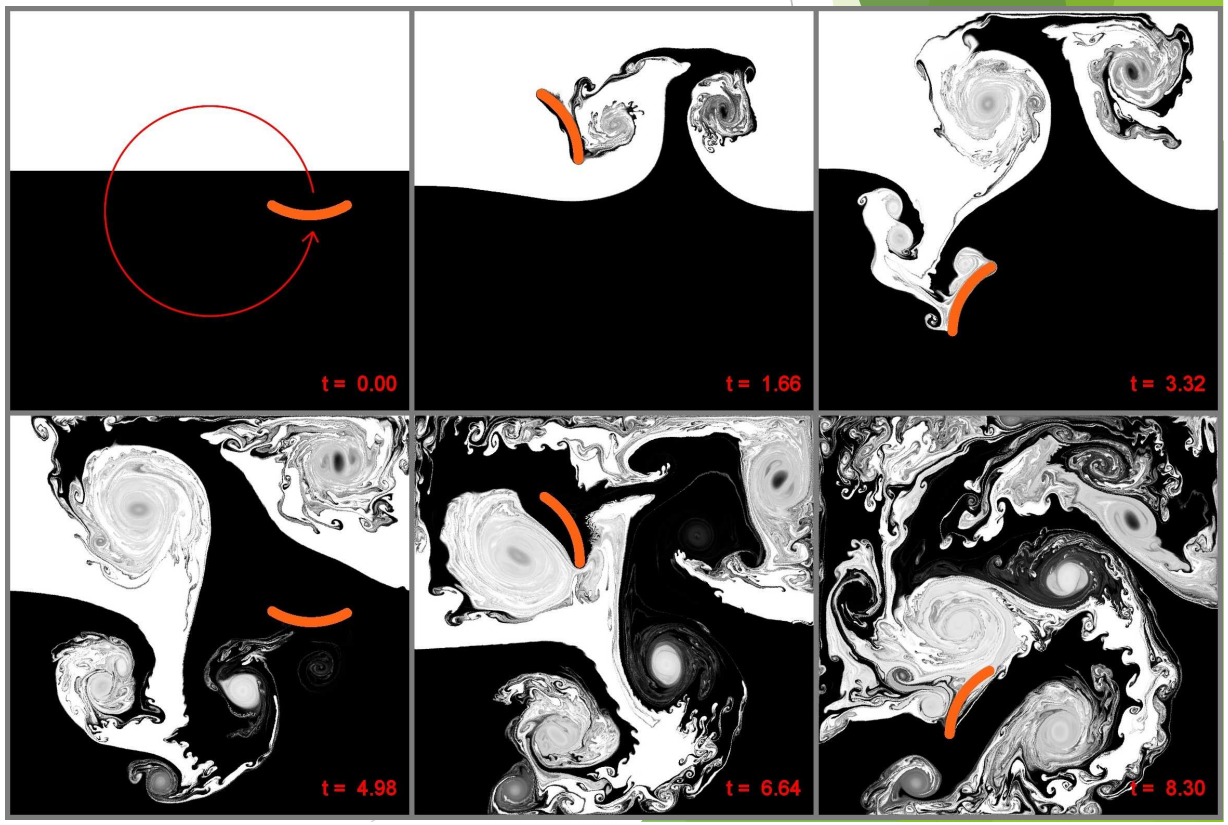


4b. Constructing boundary conditions (B.C.)

Example #3: Stirring of a spoon with reflective B.C.



- Using moving-mesh code AREPO
- Spoon treated as an inner boundary with reflective B.C.

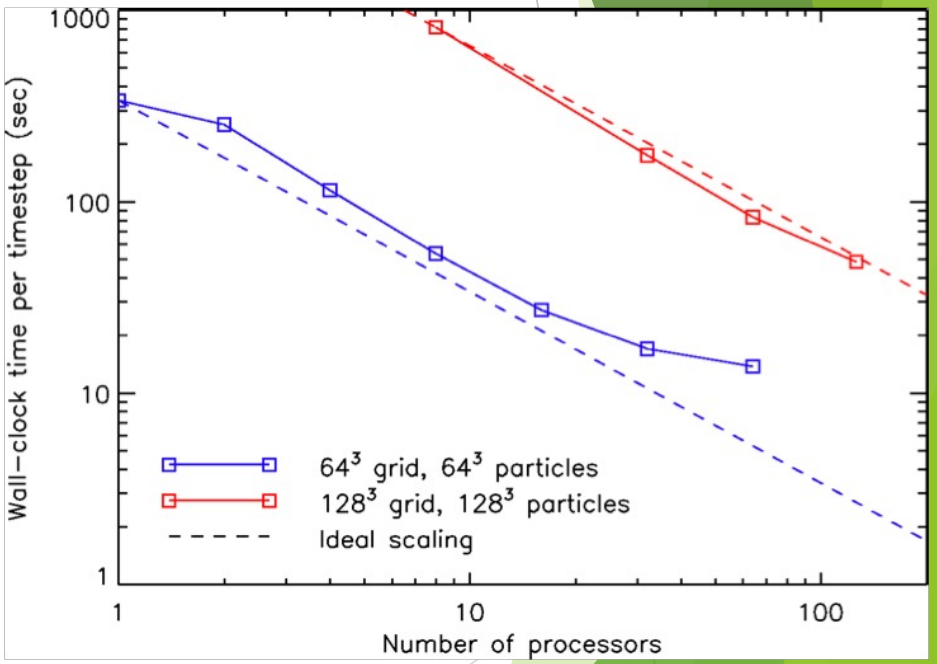


5. Estimating requirements

- Estimate ***memory*** requirement of your code
- Estimate the total ***wall-clock time*** required (often expressed in units of ***CPU-hours***)
  - Estimate # of timesteps required to complete the calculation:

$$\#\ timesteps = \frac{Simulation\ Duration}{timestep\ required\ for\ stability}$$

  - Use a short test program to estimate the wall-clock time per simulation timestep
  - Demonstrate good ***parallel scaling*** of the code as a function of # cpus

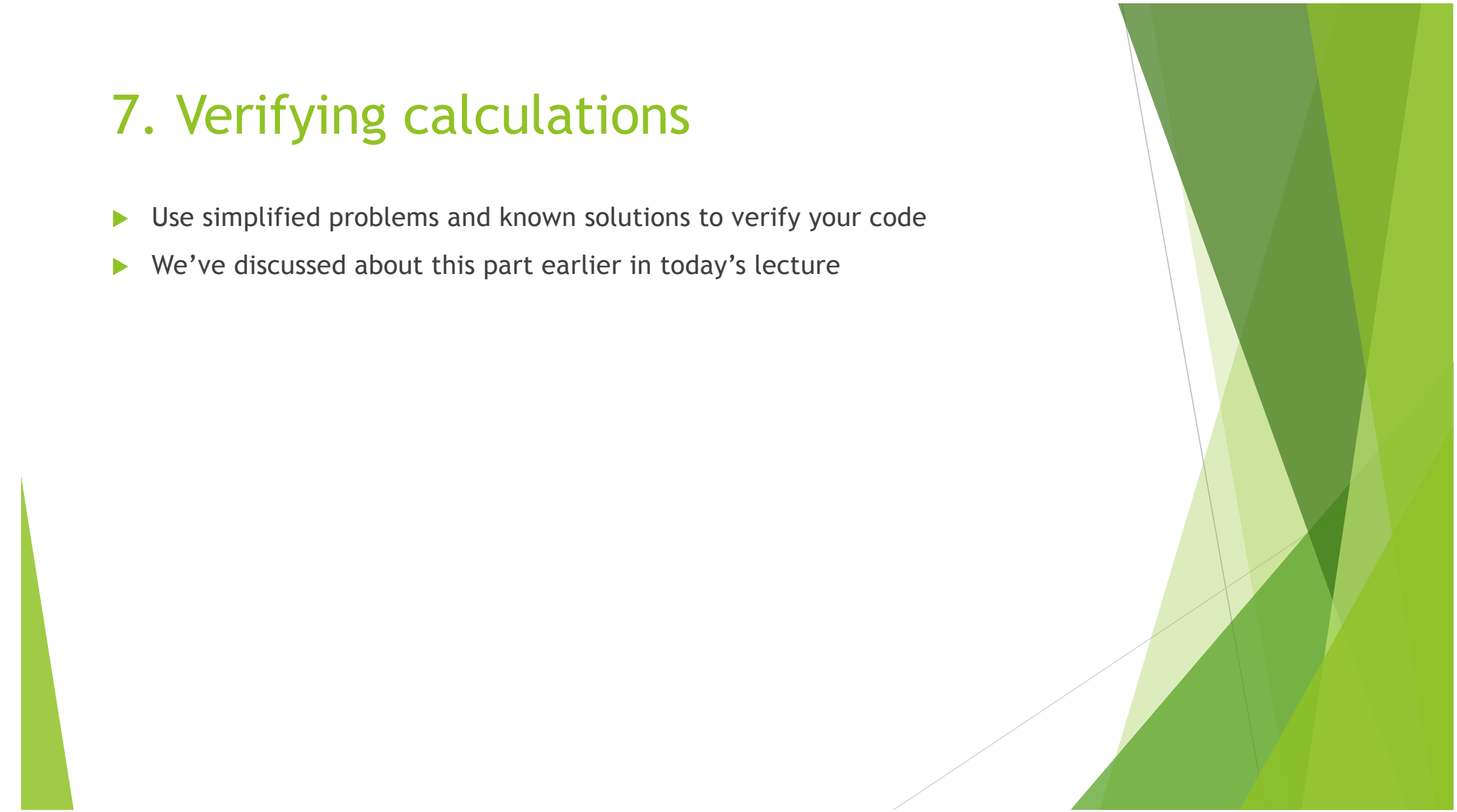


6. Proposing for resources

- For larger simulations we often need to write ***allocation proposals*** to apply for computing times on ***HPC*** platforms (e.g., supercomputers at ***NCHC***/National Center for High-Performance Computing/國網中心)
- Writing an allocation proposal
  - Summary of proposed research (scientific justification, explain experiments to be carried out)
  - Computational methodology (why choosing particular algorithms/codes, how they work, explain any new code development)
  - Preliminary results (results of previous works, published papers, or exploratory calculations)
  - Justification of resources (scaling studies, justify resources needed)
  - Local computing resources (for code development and testing)
  - Project team qualifications

7. Verifying calculations

- Use simplified problems and known solutions to verify your code
- We've discussed about this part earlier in today's lecture



8. Conducting calculations

- Work up from smaller calculations
  - Don’t start with a 1024^3 run!!!
- Ensure *repeatability*
  - Keep raw data, intermediate data, plots and code organized!
  - Every plot and number should have a source file from which it can be regenerated
- *Document the results/progress* (because you will forget everything in 6 months...)
  - Why did you perform this calculation?
  - Did you expect this result?
- Keep track of usage
  - Will you be able to complete your project?

9. Analyzing results

- Power spectra
- Distributions/histograms/phase diagrams
- Scatter plots
- Integral quantities
- Profiles
- Slices
- Projections
- Iso-surfaces
- Mock observations
- Visualizations (animations, volume renderings...)

Some reminders for plotting:

- Make your figures of *publication quality* (e.g., labels, fontsizes, line thickness, etc)
- Make them *concise* but *scientifically informative*
- Make them *pretty*

10. Refining calculations

- Guided by the preliminary results/analyses, perform ***production runs*** (e.g., full resolution, frequent output files for visualization)
- Add additional capabilities to aid the scientific interpretations (e.g., tracer fluid, tracer particles)

Steps in numerical experimentation

1. Choosing a problem
2. Choosing a numerical method
3. Choosing a code
4. Constructing initial and boundary conditions
5. Estimating requirements
6. Proposing for resources
7. Verifying calculations
8. Conducting calculations
9. Analyzing results
10. Refining calculations

Computation is more than just coding and running the simulations!

General concepts of computation -- summary

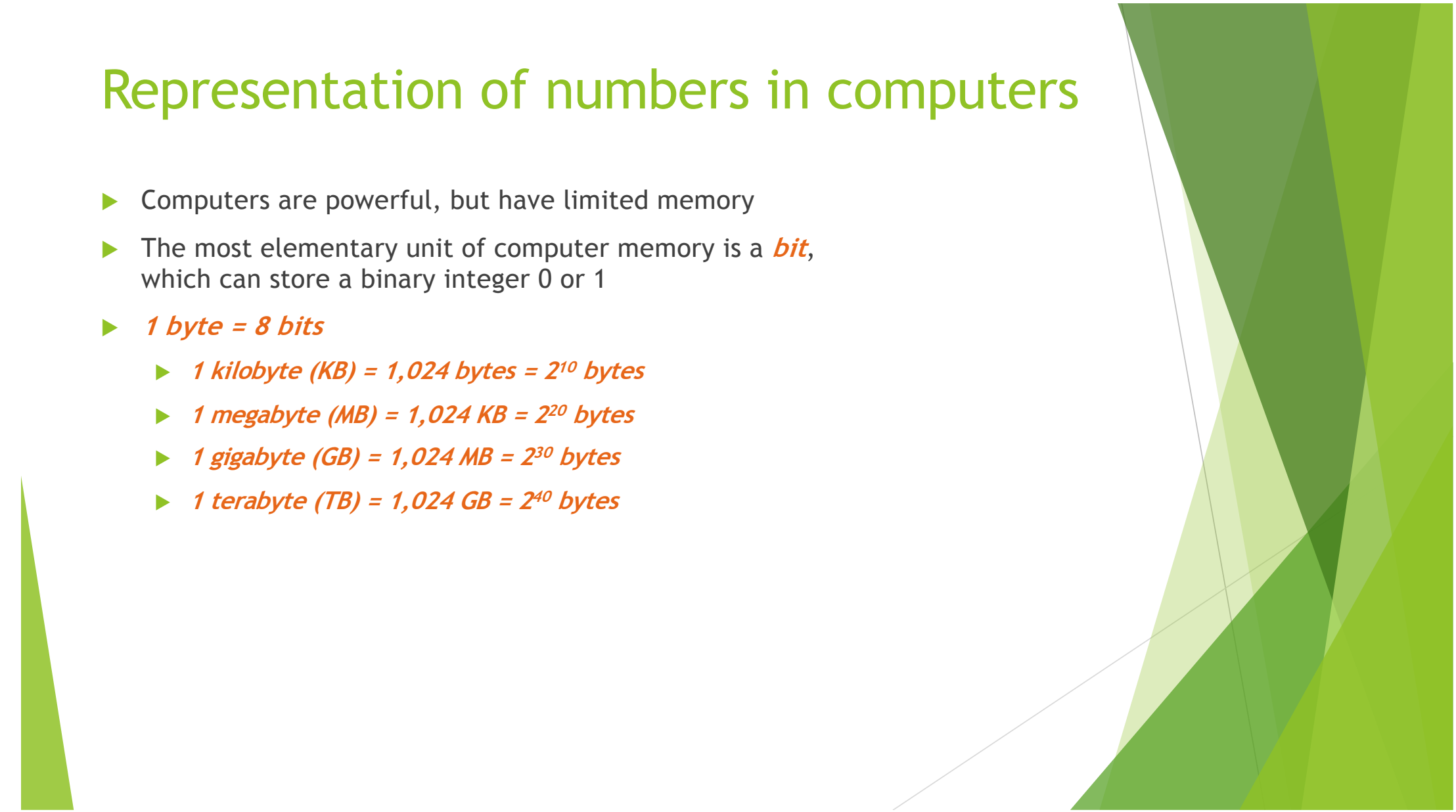
- Rules of thumb for computing:
  - Good habits will go a long way
  - Plan ahead before you code it
  - Your code is not done until you test it
- *Validation* -- ”solving the right equations”, know the assumptions/limitations

  Verification – “solving the equations right” using known physics/analytical solutions
- Computation is more than just coding and running/analyzing simulations, but require *careful planning + testing*
- Computation is more than making fancy visualizations, but has to *produce physically robust/accurate results*

How numbers are stored in computers

Representation of numbers in computers

- Computers are powerful, but have limited memory
- The most elementary unit of computer memory is a ***bit***, which can store a binary integer 0 or 1
- ***1 byte = 8 bits***
  - ***1 kilobyte (KB) = 1,024 bytes = 2^{10} bytes***
  - ***1 megabyte (MB) = 1,024 KB = 2^{20} bytes***
  - ***1 gigabyte (GB) = 1,024 MB = 2^{30} bytes***
  - ***1 terabyte (TB) = 1,024 GB = 2^{40} bytes***



Storing integers on the computer

- For storing integers
  - N bits can store integers in the range [0, 2^N]
  - The actual range is **[0, 2^{N-1}]** because the first bit stores the sign of the integer
- Different computers store numbers/words using different numbers of bits
  - On 8-bit (1-byte) systems, maximum integer is $2^7 = 128$
  - On 32-bit (4-byte) systems, maximum integer is $2^{31} \sim 2 \times 10^9$
  - On 64-bit (8-byte) systems, maximum integer is $2^{63} \sim 10^{19}$

Example -- variable declaration in a Fortran code numbers.f90:

```
integer :: a
real    :: b
```

During compilation, one can specify how many bytes to store the numbers:

```
gfortran -fdefault-integer-4 -fdefault-real-8 numbers.f90
```

Floating-point numbers

- Real numbers are stored as ***floating-point numbers***, which is a ***binary*** version of the scientific notation
- Due to finite memory/bits, not all real numbers can be stored on a machine
  - If one exceeds the maximum, an ***overflow*** occurs
  - If one falls below the minimum, an ***underflow*** occurs
- A real number ***x*** can be represented as (β = base, p = precision):

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}}\right)\beta^E, \qquad where\ 0 \le d_i \le \beta - 1$$

Floating-point numbers

$$x = \pm\left(d_0 + \frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_{p-1}}{\beta^{p-1}}\right)\beta^E$$

- For example, the speed of light expressed in decimal form (base β = 10) is

$$c = +2.99792458 \times 10^{+8}\ m/s$$

- Expressed in binary form (base β = 2), the speed of light is

$$c = +1.01111111101111 \times 2^{+1}\ m/s$$

sign

mantissa (尾數) $= d_0.d_1d_2 \ldots d_{p-1}$

exponent (指數) = E

- The way they are stored is different on different machines

The IEEE standard

- In 1987, the Institute of Electrical and Electronics Engineers (IEEE) adopted the IEEE 754 standard for floating-point arithmetic
- Following the standard, a floating-point number x is stored as

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{\text{e - bias}}$$

where ***s*** = sign

f = fractional part of the mantissa ($d_1 d_2 \ldots d_{p-1}$ in the previous expression)

e = exponent (a positive integer)

bias is used to shift the exponent so that it can store negative exponents

IEEE single precision floating-point numbers

= singles = floats = 32 bits (4 bytes)

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{\text{e - bias}}$$

Number name	Values of s, e, and f	Value of single
Normal	$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$
Subnormal	$e = 0,\ f \neq 0$	$(-1)^s \times 2^{-126} \times 0.f$
Signed Zero (± 0)	$e = 0,\ f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0,\ e = 255,\ f = 0$	+INF
$-\infty$	$s = 1,\ e = 255,\ f = 0$	−INF
Not a number	$s = u,\ e = 255,\ f \neq 0$	NaN

(u = unassigned)

(NaN = not a number, e.g., 0/0)

- Largest possible floating-point number is when

$$s = 0 \quad e = 1111\ 1110 = 254, \quad p = e - 127 = 127,$$
$$f = 1.1111\ 1111\ 1111\ 1111\ 1111\ 111 = 1 + 0.5 + 0.25 + \cdots \simeq 2,$$
$$\Rightarrow (-1)^s \times 1.f \times 2^{p=e-127} \simeq 2 \times 2^{127} \simeq 3.4 \times 10^{38}.$$

- Smallest possible floating-point number is when

$$s = 0 \quad e = 0 \quad p = e - 126 = -126$$
$$f = 0.0000\ 0000\ 0000\ 0000\ 0000\ 001 = 2^{-23}$$
$$\Rightarrow (-1)^s \times 0.f \times 2^{p=e-126} = 2^{-149} \simeq 1.4 \times 10^{-45}$$

Single-precision numbers have *6-7 decimal places of significance* in the range

$1.4 \times 10^{-45} \leq$ single precision $\leq 3.4 \times 10^{38}$.

IEEE double precision floating-point numbers

= doubles = 64 bits (8 bytes)

$$x_{\text{float}} = (-1)^s \times 1.f \times 2^{\text{e - bias}}$$

Number name	Values of s, e, and f	Value of double
Normal	$0 \le e \le 2047$	$(-1)^s \times 2^{e-1023} \times 1.f$
Subnormal	$e = 0,\ f \neq 0$	$(-1)^s \times 2^{-1022} \times 0.f$
Signed zero	$e = 0,\ f = 0$	$(-1)^s \times 0.0$
$+\infty$	$s = 0,\ e = 2047,\ f = 0$	`+INF`
$-\infty$	$s = 1,\ e = 2047,\ f = 0$	`−INF`
Not a number	$s = u,\ e = 2047,\ f \neq 0$	`NaN`

Double-precision numbers have ***15-16 decimal places of significance*** in the range

$4.9 \times 10^{-324} \le$ double precision $\le 1.8 \times 10^{308}$

Machine precision

- Floating-point numbers are of limited precision
  - For 32-bit machine, singles are good to 6-7 decimal places
  - For 64-bit machine, doubles are good to 15-16 decimal places
- Example -- precisions would be lost on a 32-bit machine when this operation is performed:

$$7 + 1.0 \times 10^{-7} = 7$$

- Thus, ***machine precision*** ϵ_m is defined as the maximum positive number that

$$1_c + \epsilon_m \equiv 1_c \quad (subscript\ c\ means\ computer\ representation\ of\ 1)$$

- ***For 32-bit machines,*** $\epsilon_m \simeq 10^{-7}$

  For 64-bit machines, $\epsilon_m \simeq 10^{-16}$
- The error introduced by finite machine precisions is called "***roundoff errors***"

Approximation errors / truncation errors

- Errors arising from simplifying the math so that a problem can be solved on the computer
  - Infinite series -> finite sums
  - Infinite intervals -> finite intervals
  - Variable functions -> constants
- Example:

Approximation/truncation error

$$e^{-x} = \sum_{n=0}^{\infty} \frac{(-x)^n}{n!} \simeq \sum_{n=0}^{N} \frac{(-x)^n}{n!} = e^{-x} + \mathcal{E}(x, N)$$

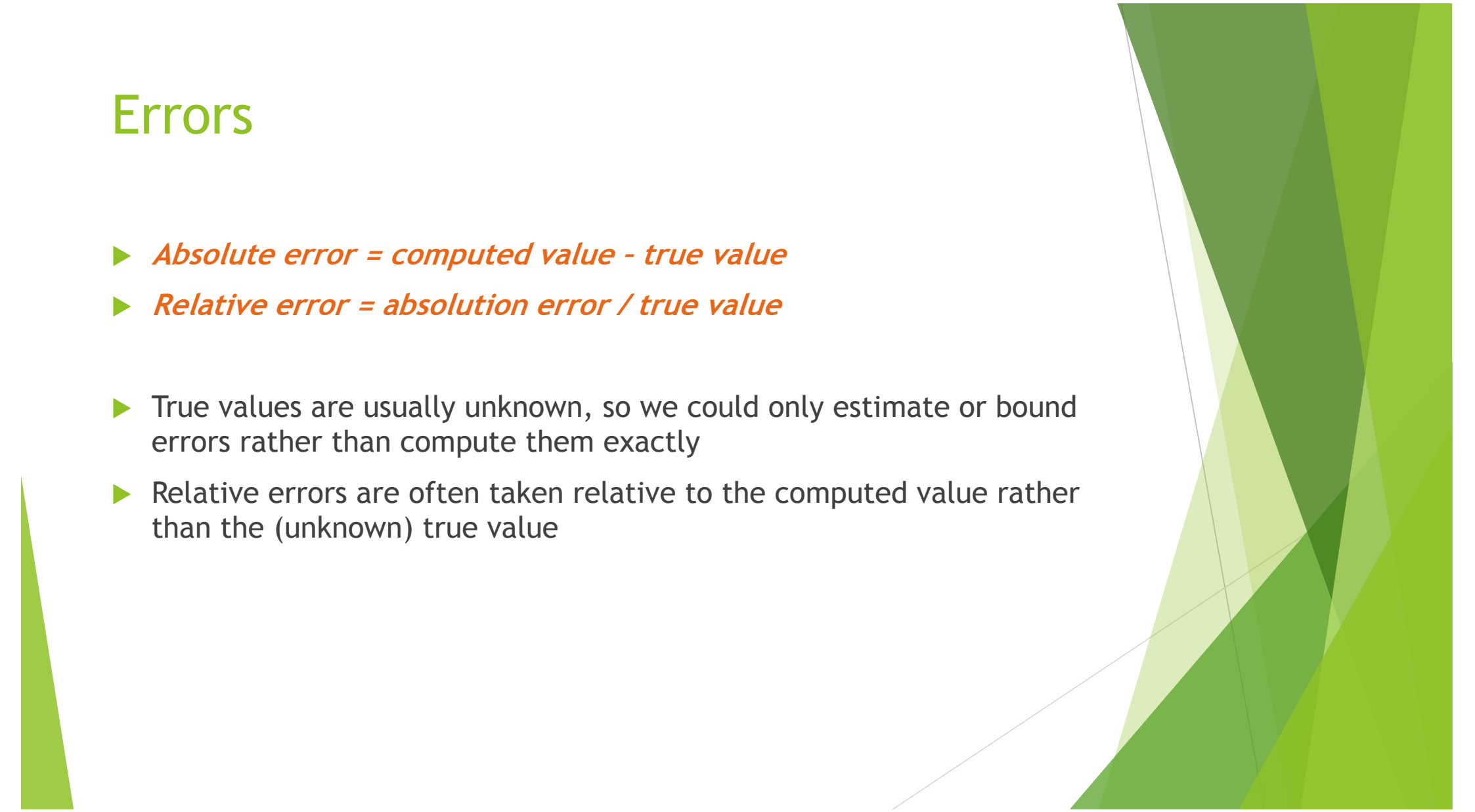
- *Approximation error decreases as N increases*

Errors in computation

Errors

- *Absolute error = computed value - true value*
- *Relative error = absolution error / true value*

- True values are usually unknown, so we could only estimate or bound errors rather than compute them exactly
- Relative errors are often taken relative to the computed value rather than the (unknown) true value



Sources of errors in computation

- ***Roundoff errors***: imprecisions arising from the finite number of digits used to store floating-point numbers
- ***Approximation/truncation errors***: errors arising from simplifying math so that a problem can be solved on the computer
- ***Random errors***: errors caused by rare, random events (e.g., fluctuation in electronics due to power surges, cosmic rays, someone pulling a plug)
  - Their likelihood increases with running time and # processors involved
  - We have no control over them; only solution is to check reproducibility
- ***Blunders or bad theory***: errors caused by human mistakes (e.g., typos of program, using the wrong data file, wrong theory/reasoning…)
  - Again, remember the 3 programming rules of thumbs
  - If they occur too frequently, it's time to go home and take a break

Consequences of roundoff errors #1 -- order of operation matters

- Example – assuming your computer stores 4 significant digits
- This computer would store 1/3 as 0.3333 and 2/3 as 0.6667
- What if you ask the computer to do the following simple operation?

$$2\left(\frac{1}{3}\right) - \frac{2}{3} =$$

- ***Be careful with the orders of operation*** when you implement equations in your codes; ***simply the equations and avoid unnecessary operations*** if possible

Consequences of roundoff errors #2 -- order of magnitude matters

- Example -- precisions would be lost on a 32-bit machine when this operation is performed:

$$7 + 1.0 \times 10^{-7} = 7$$

- ***Avoid performing addition/subtractions between numbers with very different orders of magnitudes*** (especially true in astronomy)
- A good way to mitigate this is to use quantities normalized to characteristic values in the calculation so that everything is of order 1, e.g.,

$$x \equiv \frac{r}{R}, u \equiv \frac{v}{c_s}, m \equiv \frac{M}{M_{sun}} \ldots .$$

Consequences of roundoff errors #3 -- subtractive cancellation

- Because of the roundoff error, let us call x_c the computer representation of the exact number x:

$$x_c \simeq x(1+\epsilon_x)$$

where ϵ_x is the relative error in x_c, expected to be of similar magnitudes to the machine precision ϵ_m.

- Now consider simple subtraction $a = b - c$:

$$a_c \simeq b_c - c_c \simeq b(1+\epsilon_b) - c(1+\epsilon_c)$$

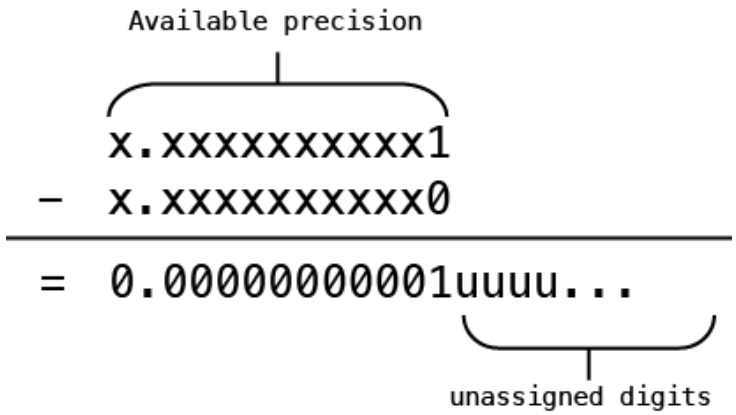
$$\frac{a_c}{a} \simeq 1 + \epsilon_b \frac{b}{a} - \frac{c}{a}\epsilon_c$$

(Note that ϵ_b and ϵ_c do not necessarily cancel because they may not have the same sign.)

- Relative error of a becomes large when $b \approx c$ (because a is small)!!

Consequences of roundoff errors #3 -- subtractive cancellation

- *If we subtract two large numbers and end with a small number, there will be fewer significant digits in the small one*
- Try the best to *avoid subtractive cancellation*!



Bird weight /= (elephant+bird) - elephant!

Errors accumulate!

- Our program involves numerous operations. Assuming each step has probability ***p*** of being correct, the joint probability P of the whole program to be correct is $P = p^n$
  - For n = 1000, p = 0.9993 => P = ½ !!
- For example, ***accumulation of roundoff errors*** could scale differently depending on the problem
  - If the error is random and behaves like a random-walk process, $\epsilon \approx \sqrt{N}\epsilon_m$
  - In cases of no cancellation, the error could be linear, $\epsilon \approx N\epsilon_m$

Taiwania 3, a supercomputer at NCHC, has peak computing speed of 2.7 PFLOPs (*FLOPs = floating-point operations per second; peta = 10^{15}*).

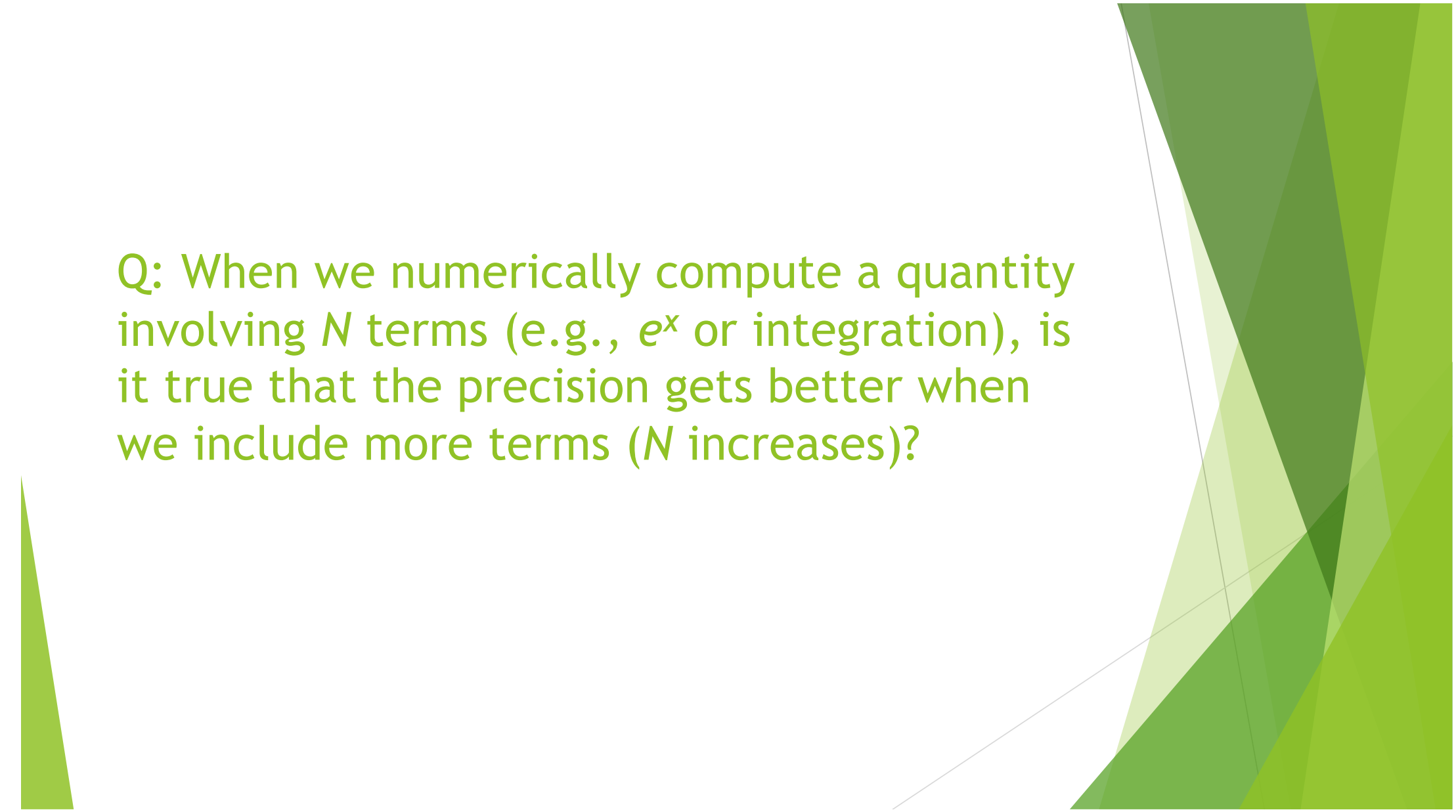
⇒ A program running for 3 hours performs ~ 10^{20} operations.

⇒ If the roundoff error accumulates randomly, after 3 hours we expect a relative error of $10^{10}\ \epsilon_m$

⇒ For the error to be smaller than the answer (relative error < 1), we need $\epsilon_m < 10^{-10}$

⇒ *Double precision is required for HPC!!*

Q: When we numerically compute a quantity involving *N* terms (e.g., e^x or integration), is it true that the precision gets better when we include more terms (*N* increases)?



The answer is *NO*!

- This is because including more terms will decrease the approximation/truncation error but increases the roundoff errors!
- In general, ***truncation error decreases with N***:

$$\epsilon_{tr} \simeq \frac{\alpha}{N^{\beta}}$$

- ***Roundoff error increases with N***:

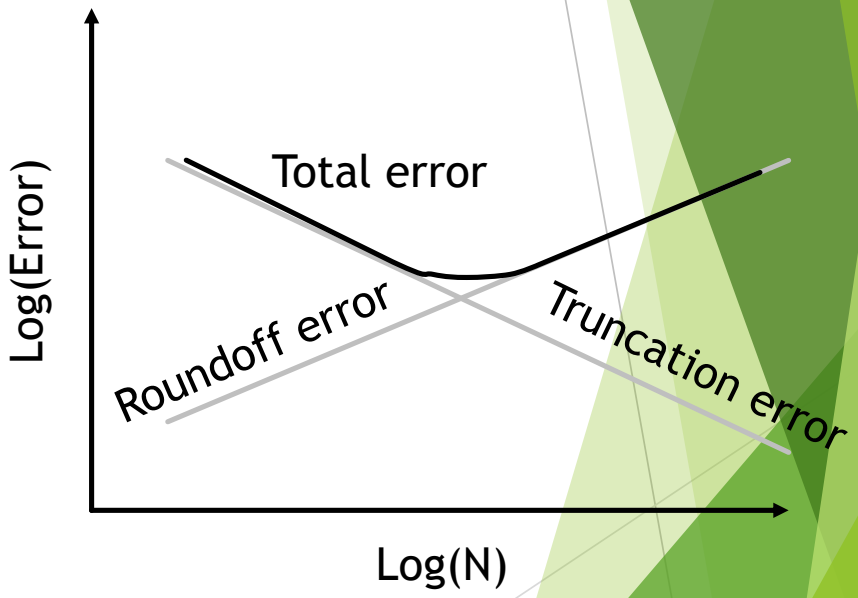
$$\epsilon_{ro} \simeq \sqrt{N}\epsilon_m$$

- Therefore, the total error is

$$\epsilon_{tot} = \epsilon_{tr} + \epsilon_{ro} \simeq \frac{\alpha}{N^{\beta}} + \sqrt{N}\epsilon_m$$

- The smallest total error would occur when

$$\epsilon_{tr} \approx \epsilon_{ro}$$



Minimizing the error

$$\epsilon_{tot} = \epsilon_{tr} + \epsilon_{ro} \simeq \frac{\alpha}{N^{\beta}} + \sqrt{N}\epsilon_m$$

Example #1: $\epsilon_{tr} \simeq \frac{1}{N^2}$

- To minimize the total error:

$$\frac{d\epsilon_{tot}}{dN} = 0 \;\Rightarrow\; N^{\frac{5}{2}} = \frac{4}{\epsilon_m}$$

- For single-precision calculation ($\epsilon_m \simeq 10^{-7}$):

$$N \simeq 1099 \qquad \Rightarrow \qquad \epsilon_{\text{tot}} \simeq 4 \times 10^{-6}$$

- For this algorithm, the minimum error is ~40x machine precision

Example #2: $\epsilon_{tr} \simeq \frac{2}{N^4}$

- To minimize the total error:

$$\frac{d\epsilon_{tot}}{dN} = 0 \;\Rightarrow\; N^{\frac{9}{2}} = \frac{16}{\epsilon_m}$$

- For single-precision calculation ($\epsilon_m \simeq 10^{-7}$):

$$N \simeq 67 \qquad \Rightarrow \qquad \epsilon_{\text{tot}} \simeq 9 \times 10^{-7}$$

- With a good algorithm, it is possible to achieve better precision with a smaller *N*!

Numbers and errors in computation -- summary

- The most elementary unit of computer memory is a ***bit***. ***1 byte = 8 bits***.
  - *N* bits could store an integer in a range **[0, 2^{N-1}]** (first bit stores the sign)
  - Real numbers are stored as ***floating-point numbers***, including their ***sign***, ***exponent***, and ***mantissa***.
  - ***Single-precision***: ***32-bit*** (4-byte), also called ***floats***, machine precision $\epsilon_m \simeq 10^{-7}$
  - ***Double-precision***: ***64-bit*** (8-byte), also called ***doubles***, machine precision $\epsilon_m \simeq 10^{-16}$
- Errors in computation
  - ***Roundoff errors*** – imprecision due to limited digits used to store floating-point numbers
  - ***Truncation errors / approximation error*** – arising due to simplified math
  - Random errors – caused by rare, random events
  - Blunders -- human errors

In-class exercise

Things to do…

1. Use Terminal to log onto the CICA cluster via ssh:

   `ssh your_account_name@fomalhaut.astr.nthu.edu.tw`

2. Load python and activate your environment:
   1. `module load python`
   2. `source activate compAstro`
3. Write a Python program (ex2.py) to test for the ***underflow*** and ***overflow*** limits of your computer system (within a factor of 2). A sample pseudo code is:

```
under = 1. over = 1.
  begin do N times
    under = under/2.
    over = over * 2.
    write out: loop number, under, over
  end do
```

Things to do…

4. Run the script (`python ex2.py`). Increase *N* until you get overflows or underflows. Take a screen shot of when that happens, e.g.,

```
N, under, over =   ###  1.1125369292536007e-308 8.98846567431158e+307
N, under, over =   ###  5.562684646268003e-309 inf
```

```
N, under, over =   ###  5e-324 inf
N, under, over =   ###  0.0 inf
```

5. Modify your code to turn the variables (under, over) into ***single-precision*** using numpy.single(). Run the script again and print the screen.
6. Verify that your code reproduces the correct ranges of double and single floating-point numbers (slides 38 & 39)
7. To get the bonus credit, submit your code as well as the above two screenshots to the TAs by the end of today (9/22/2022)