Computation Basics II

Lecture 3, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 9/29/2022

Announcements

- ***HW1 is due TODAY***. 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

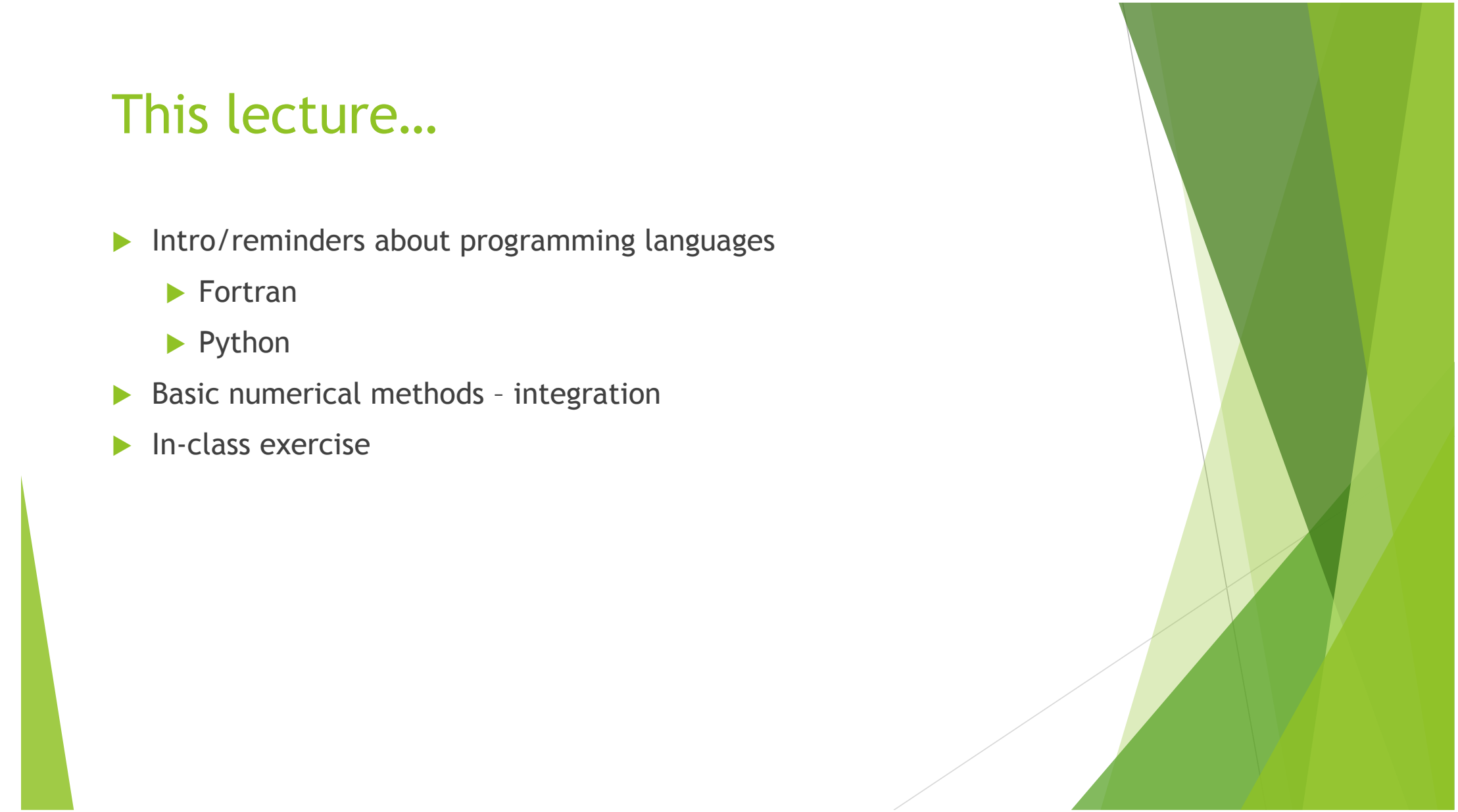
Previous lecture…

- *Validation* -- ”solving the right equations”, know the assumptions/limitations

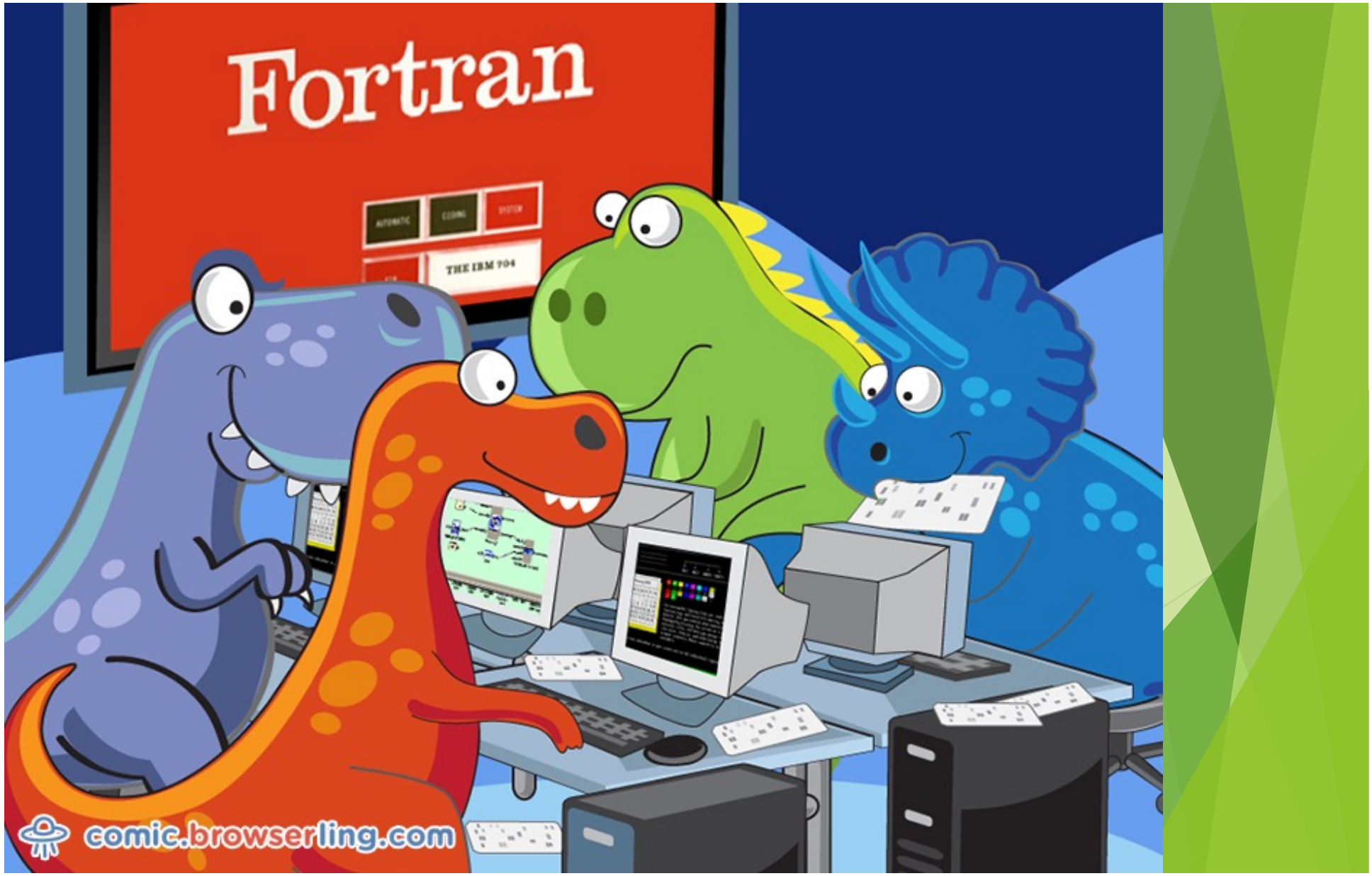
  Verification – “solving the equations right” using known physics/analytical solutions
- Computation is more than just coding and running/analyzing simulations, but require *careful planning + testing* (recall the 10 steps for designing a numerical experiment)
- How numbers are stored on the computer
  - 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

This lecture…

- Intro/reminders about programming languages
  - Fortran
  - Python
- Basic numerical methods – integration
- In-class exercise



Programming languages -- Fortran

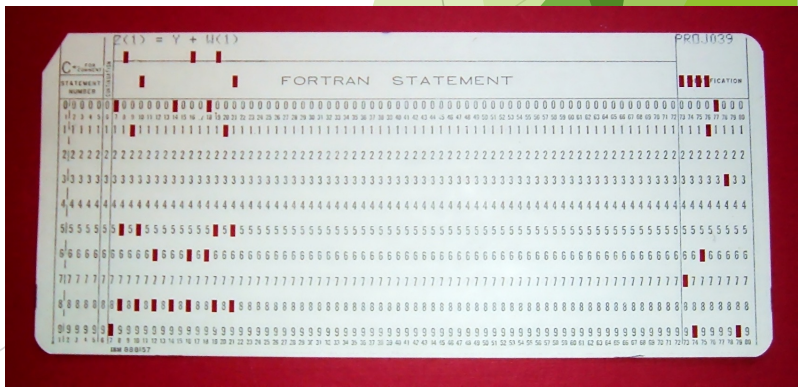

Fortran
THE IBM 704
comic.browserling.com

Introduction to Fortran

- In 1954, Fortran was designed for IBM 704 computer as a replacement for assembly language (one statement per machine instruction)
- Fortran = Formula Translation = a general-purpose, compiled programming language especially suited to numerical computation and scientific computing
- Evolution of Fortran:
  - Fortran (1954)
  - Fortran II, III, IV, 66 **C was invented in 1972
  - **Fortran 77** (1977, structured programming)
  - ***Fortran 90*** (modular programming; free-format), 95
  - Fortran 2003 (object-oriented programming)
  - Fortran 2008 (concurrent programming)

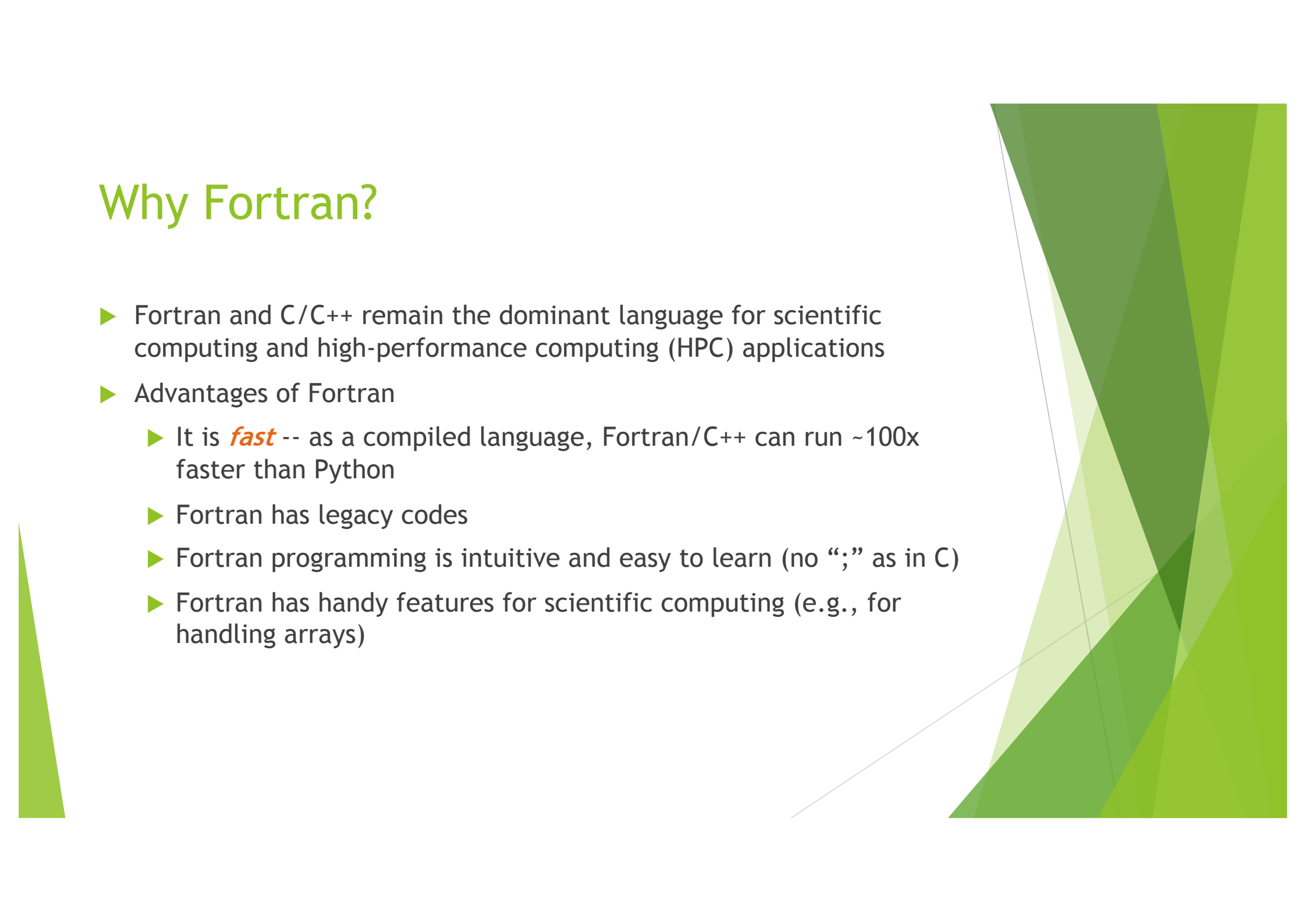
IBM 704 computer

Punchcards for Fortran statements



Why Fortran?

- Fortran and C/C++ remain the dominant language for scientific computing and high-performance computing (HPC) applications
- Advantages of Fortran
  - It is ***fast*** -- as a compiled language, Fortran/C++ can run ~100x faster than Python
  - Fortran has legacy codes
  - Fortran programming is intuitive and easy to learn (no “;” as in C)
  - Fortran has handy features for scientific computing (e.g., for handling arrays)



Array handling in Fortran

integer, dimension(3,3) : : a

$$A_{ij} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Fortran

int a[3] [3]

$$A_{ij} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

C/C++

`C=A*B` gives an element-by-element multiplication of A and B

Need a **for** loop

```
where (array .lt. 0.0) array = 0.0

double precision, dimension(-1:10) :: array

double precision, dimension(:,:), allocatable :: array_2d
allocate(array_2d(xdim, ydim))
```

Demo #1: Hello world!

- Create a file named hello.f90

```
1 program hello_world
2     print *, "Hello World!"
3 end program hello_world
4
```

- *Compile* the code by `gfortran hello.f90`
- *Execute* the program by `./a.out`
- To change the name of the executable, try

  `gfortran hello.f90 -o hello`

Let's give it a try!

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

   `ssh` *your_name* `@fomalhaut.astr.nthu.edu.tw`

2. Load the Fortran compiler:

   `module load pgi`

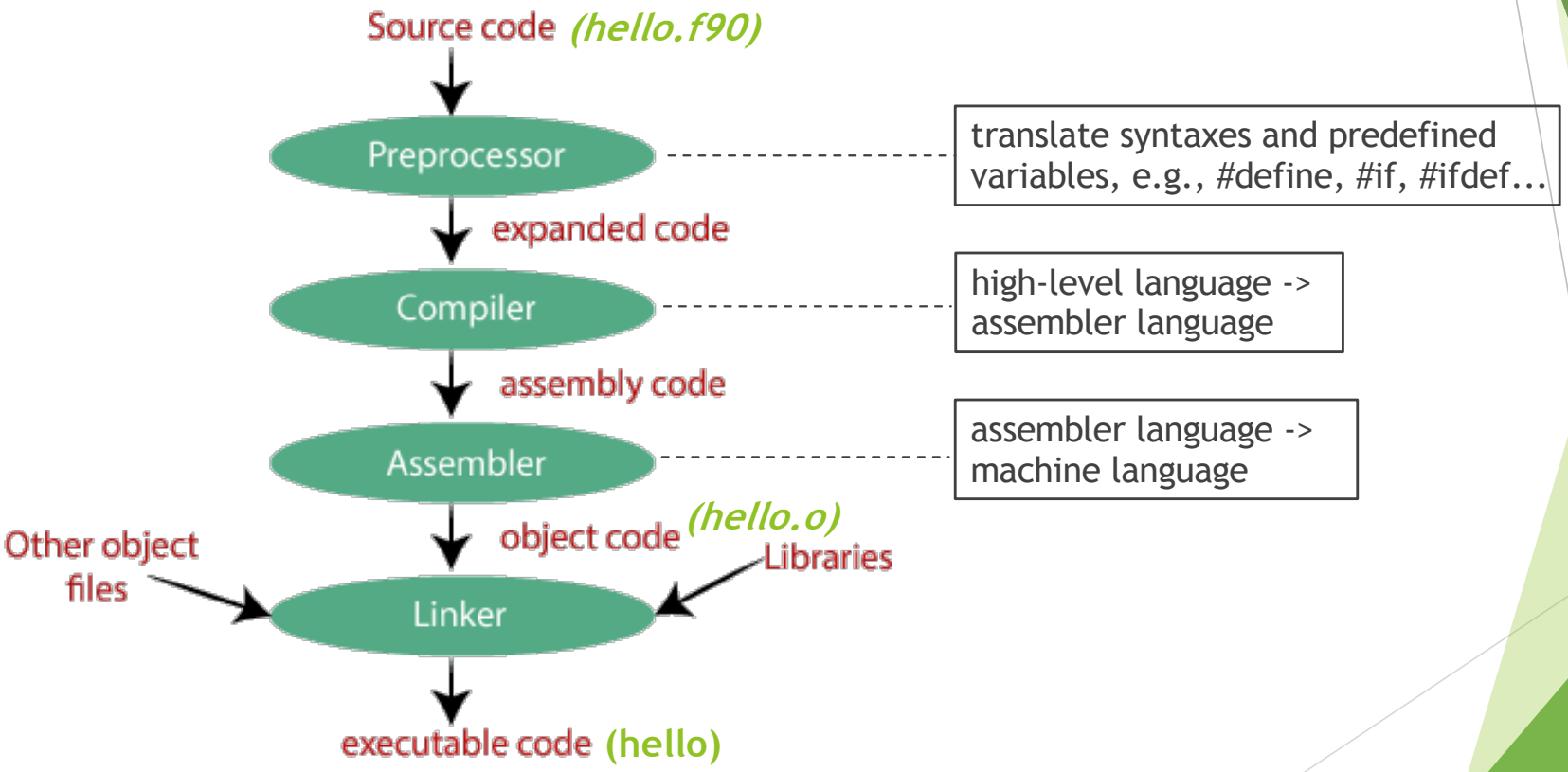
3. Move into your exercise directory and create the Fortran program:

   `cd astr660/exercise`

   `vi hello.f90`

4. Compile and execute the program.

Compiling the code



Demo #2: variables

numbers1.f90

```
1  program data_type
2      implicit none
3      integer*2 :: short
4      integer*4 :: long
5      integer*8 :: verylong
6      integer*16 :: veryverylong
7
8      real*4  :: real   = 1.0
9      real*8  :: double = 1.0
10     real*16 :: quad   = 1.0
11
12     print *, "Integers:"
13
14     print *, huge(short)
15     print *, huge(long)
16     print *, huge(verylong)
17     print *, huge(veryverylong)
18
19     print *, "Real numbers:"
20     print *, real
21     print *, double
22     print *, quad
23
24 end program data_type
```

Try it out!

- Copy the program from this directory:
  `/data/hyang/shared/astro660CompAstro/L3exercise`
- Compile and run the program:
  `gfortran [filename].f90 -o [filename]`
- What does huge(x) do?
- Are the number of significant digits consistent with what you learned in the last class?

Demo #2: variables

numbers2.f90

```
1  program data_type
2      implicit none
3      integer :: int
4      real    :: rel = 1.0
5
6      print *, "Integers:"
7      print *, huge(int)
8
9      print *, "Real numbers:"
10     print *, rel
11
12 end program data_type
13
```

Try it out!

- Copy the program from this directory:

  `/data/hyang/shared/astro660CompAstro/L3exercise`

- Compile and run the program:

  `gfortran [filename].f90 -o [filename]`

- Compile the code again using the following flags instead:

  `gfortran -fdefault-real-8 -fdefault-integer-8 [filename].f90 -o [filename]`

- Comparing with results you got for numbers1, what are the default number of bytes used to store integers and real numbers?

Demo #2: variables

numbers3.f90

```
 1 program data_type
 2     implicit none
 3     logical       :: is_person
 4     character*40  :: name
 5     complex       :: complex_var
 6     real, parameter :: msun = 1.989e33 ! a constant
 7
 8     name        = "Kuo-Chuan Pan"
 9     is_person   = .true.
10     complex_var = (3,5)
11
12     print *, name
13     print *, is_person
14     print *, complex_var
15     print *, msun
16
17 end program data_type
18
```

Demo #3: arrays & do loops

Array declaration

```
real, dimension(5) :: numbers
integer, dimension(5,5) :: matrix

real, dimension(2:6) :: numberss
integer, dimension(-3:2,0:4) :: matrixx
```

Array assignments

```
numbers(1) = 2.0

do i = 1, 5
  numbers(i) = i*2.0
end do

numbers = (/2.0, 4.0, 6.0, 8.0, 10.0/)

numbers(:) = 0.0
numbers(1:5:2) = 1.0
```

Demo #4: types, file I/O, and formatted outputs

See output.f90
& supplemental information

```
program output
    implicit none
    type Star
        character(len = 10) :: name
        integer(kind=4)     :: id
        double precision    :: age
        double precision    :: distance
        double precision    :: magnitude
        logical             :: is_double
    end type Star

    character*40                    :: file_name
    character*4                     :: id_str
    integer, parameter              :: nstars = 100

    type(Star), dimension(nstars)   :: stars
    integer :: n

    file_name = "stars.txt"
    ! create the file
    open(unit=1,file=trim(file_name))

    ! write the header
    write(1,11) "# ", "Name", "ID", "double", "Age", "Distance", &
                "Magnitude"
    write(1,11) "# ", " ", " ", " ", "[yr]", "[pc]", "[]"

    do n = 1, nstars
        ! generate the data
        write(id_str, 10) n
        stars(n)%name       = "Star "//id_str
        stars(n)%id         = n
        stars(n)%age        = sqrt(real(n)**3)
        stars(n)%distance   = real(n)**2 - 1.
        stars(n)%magnitude  = real(n)**2.5
        stars(n)%is_double  = (mod(n,5) .eq. 3)

        ! write to the file
        write(1,12) stars(n)%name, stars(n)%id, stars(n)%is_double, &
                    stars(n)%age, stars(n)%distance, stars(n)%magnitude
    enddo
    close(1)

10 format(I4)
11 format(a2, a10, a5, a7, 3a24)
12 format(2x, a10, i5, l7, 3e24.14)

end program output
```

Demo #5: the “go to” statement

```
10 40  print *, "to"
11     goto 15
12
13 30  print *, "powerful,"
14     goto 22
15
16 22  print *, "but"
17     go to 20
18
19 20  print *, "very"
20
21     if (i .eq. 0) then
22         goto 30
23     else
24         goto 40
25     endif
```

- Convenient
- Can be used to replace “if” statements or loops
- Could lead to blunders if not careful enough

```
 1          program computed goto
 2          integer day
 3          write(*,*)'Enter the day of a week'
 4          read(*,*)day
 5          goto (10,12,13,15,16,18,20),day
 6 10       write(*,*)'The day is Sunday'
 7 12       write(*,*)'The day is Monday'
 8 13       write(*,*)'The day is Tuesday'
 9 15       write(*,*)'The day is Wednesday'
10 16       write(*,*)'The day is Thursday'
11 18       write(*,*)'The day is Friday'
12 20       write(*,*)'The day is Saturdaday'
```

Demo #5: the “go to” statement

```
C     PROGRAM TO COMPUTE PRIME NUMBERS
C
      PROGRAM PRIME
      INTEGER MAXINT, N, DIVSOR
      INTEGER QUOT, PROD
      READ(*,100) MAXINT
      N=2
      WRITE(*,150) MAXINT
5     IF (MAXINT-N) 200,10,10
10    DIVSOR = 2
15    IF ((N-1)-DIVSOR) 30,20,20
20    QUOT = INT(N/DIVSOR)
      PROD = INT(QUOT*DIVSOR)
      IF (N-PROD) 25,30,25
25    DIVSOR = DIVSOR + 1
      GO TO 15
30    IF (DIVSOR-(N-1)) 40,40,35
35    WRITE (*,100) N
40    N=N+1
      GO TO 5
100   FORMAT(I5)
150   FORMAT('THE PRIME NUMBERS FROM 2 TO ',I5,' ARE: ')
200   STOP
      END
```

Quick exercise – rewrite ex1.py into a Fortran program

1. Use Terminal to log onto the CICA cluster via ssh if you haven't done so:

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

2. Move into your exercise directory and create a Fortran program:

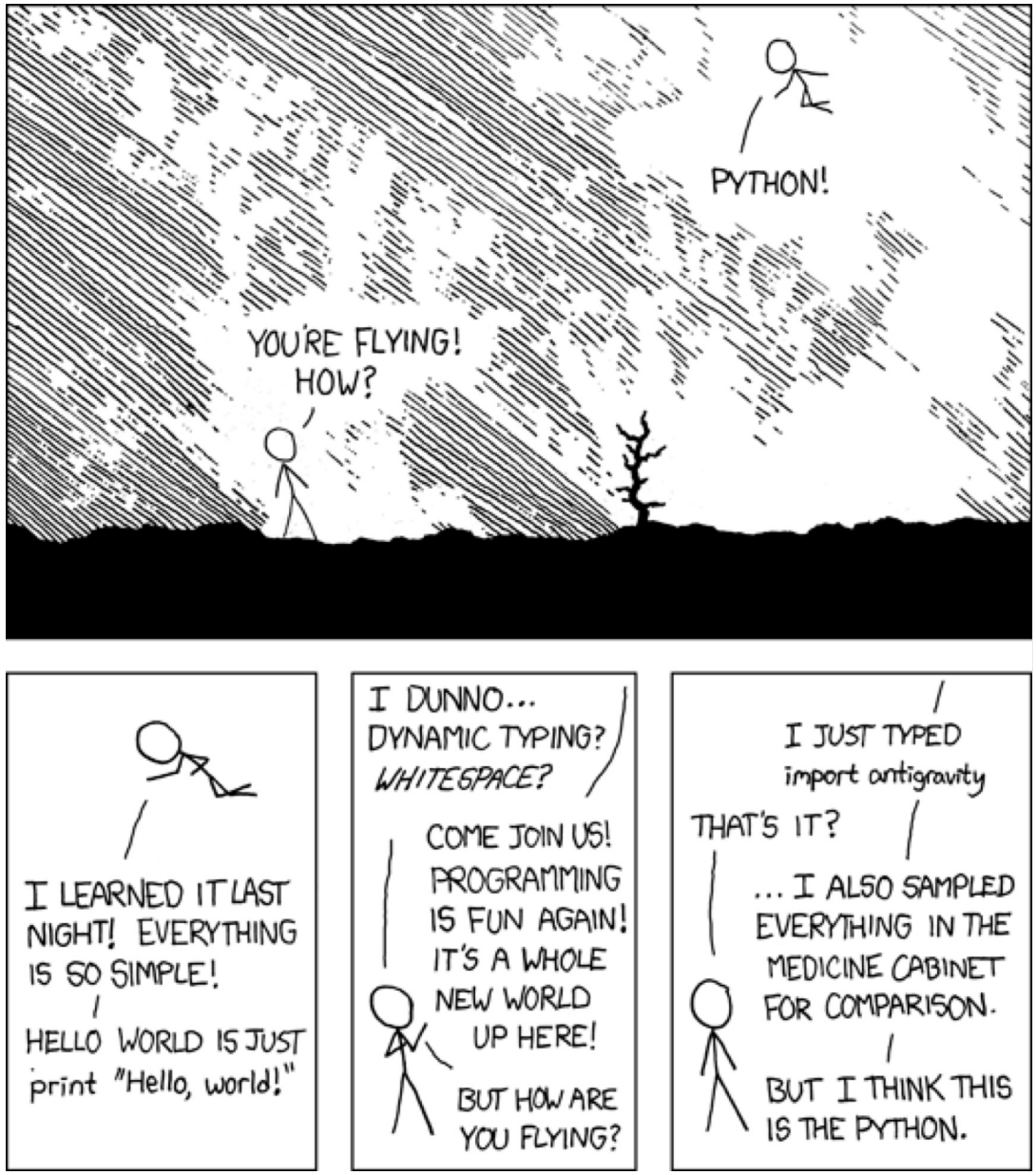
   `cd astr660/exercise`

   `vi ex1.f90`

3. In `ex1.f90`, write a short Fortran program to
   - declare and initialize an array containing numbers from 1 to 100 (using a *do* loop)
   - sum over all elements in the array using `result = sum(array)`
   - print out the result to screen

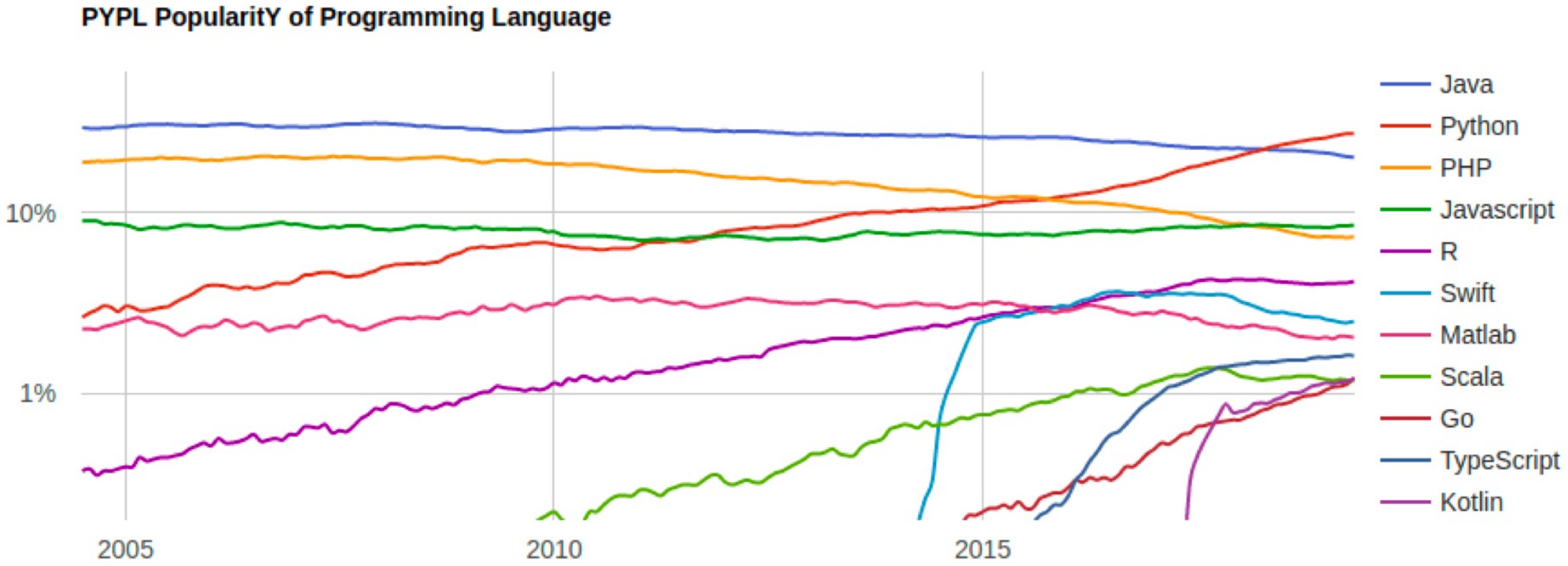
4. Compile and execute the program. Verify that your answer is correct.

Programming languages -- Python


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

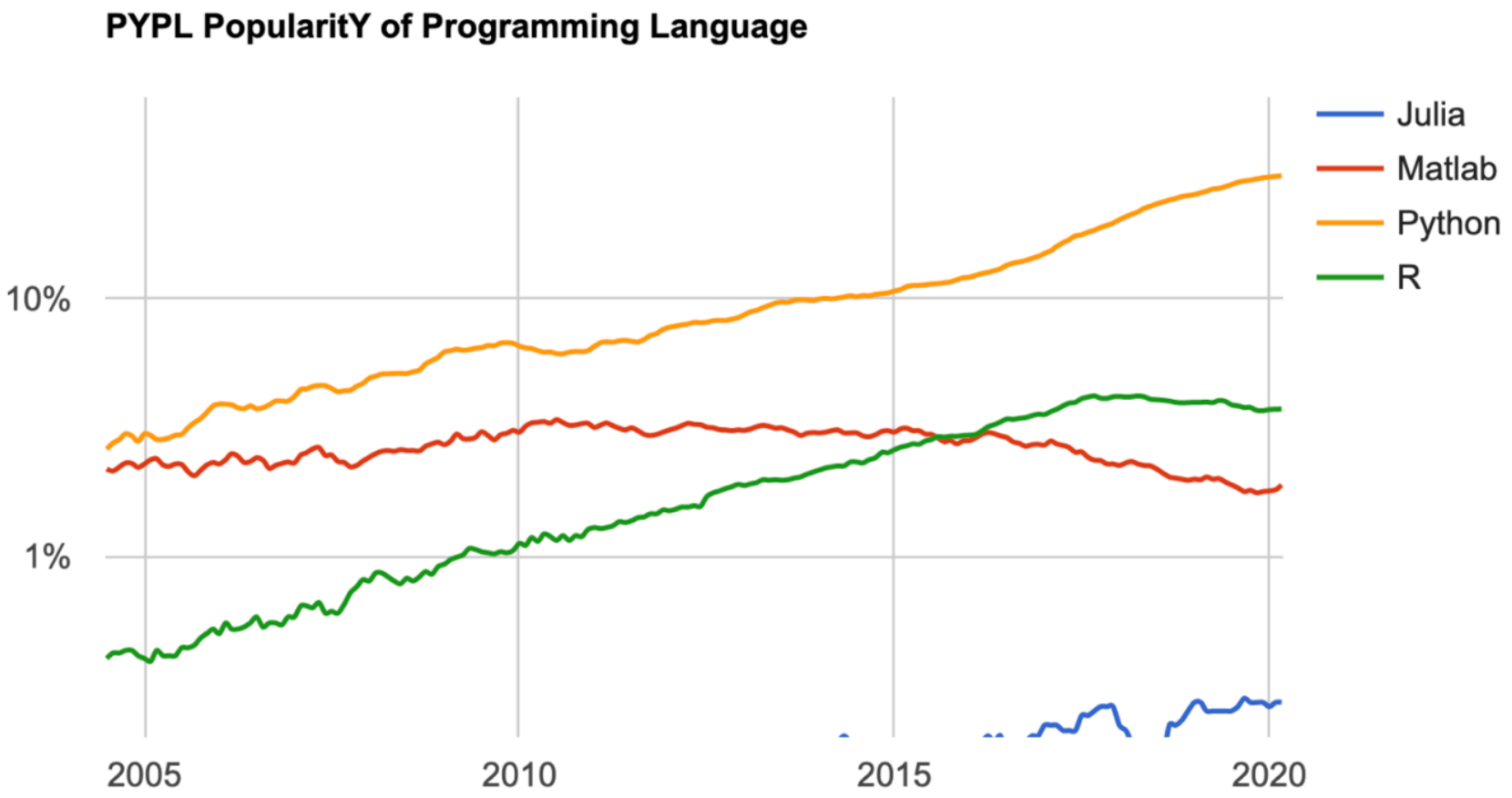
ref: xkcd.com

Top programming language trends in 2019



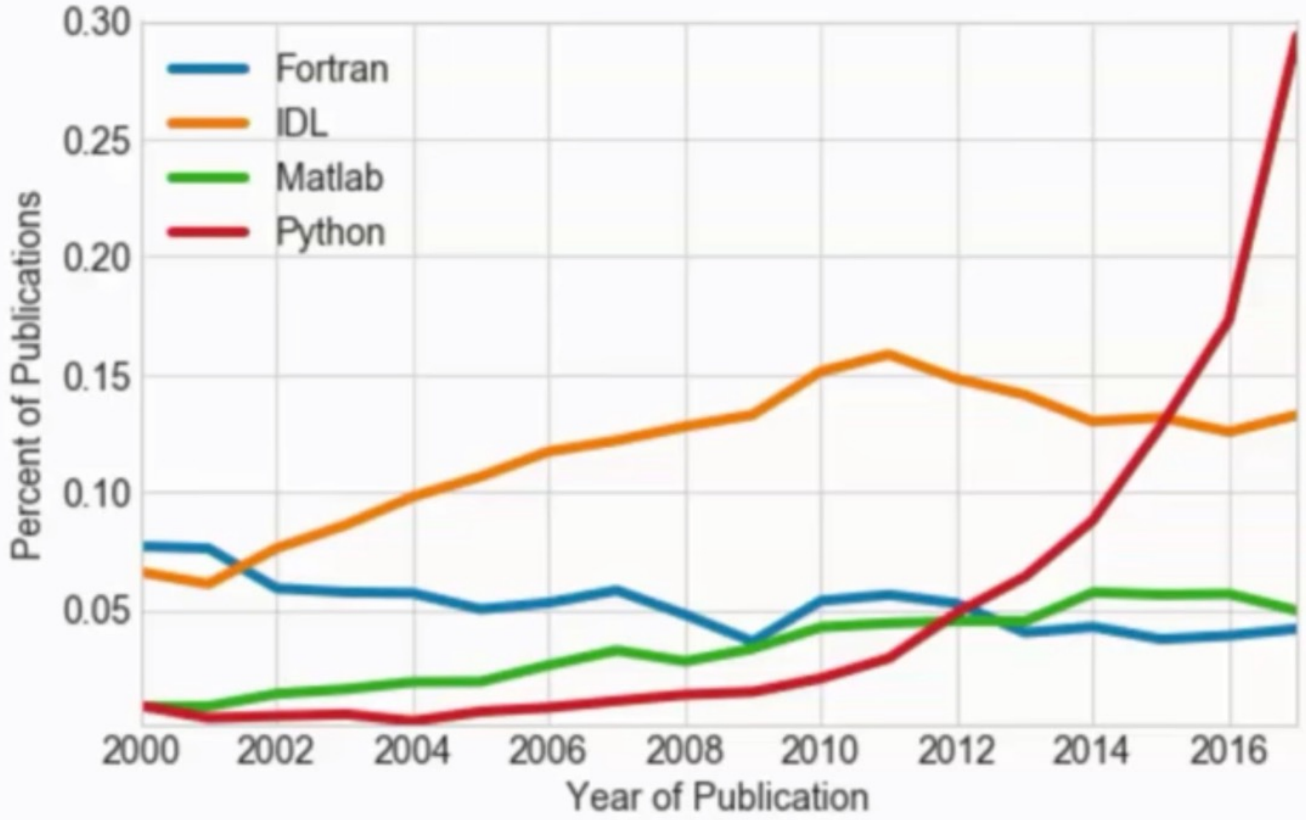
https://www.valuecoders.com/blog/technology-and-apps/programming-trends-2017-it-outsourcing-companies/

Trends for scientific computing languages



Matlab starts to die, R gets more popular but seems to plateau.

Mentions of Software in Astronomy Publications:



Thanks to Juan Nunez-Iglesias, Thomas P. Robitaille, and Chris Beaumont.

Compiled from NASA ADS (code).

https://lasseschultebraucks.com/2017/08/14/why-python-is-probably-the-programming-language-for-dealing-with-big-data.html

- First released in 1991; Python 2.0 released in 2000; Python 3.0 released in 2008)
- A modern, ***interpreted***, high-level, general purpose programming language
- Core philosophy:
  - Beautiful is better than ugly
  - Explicit is better than implicit
  - Simple is better than complex
  - Complex is better than complicated
  - Readability counts

Complex – having multiple components
Complicated – difficult to understand

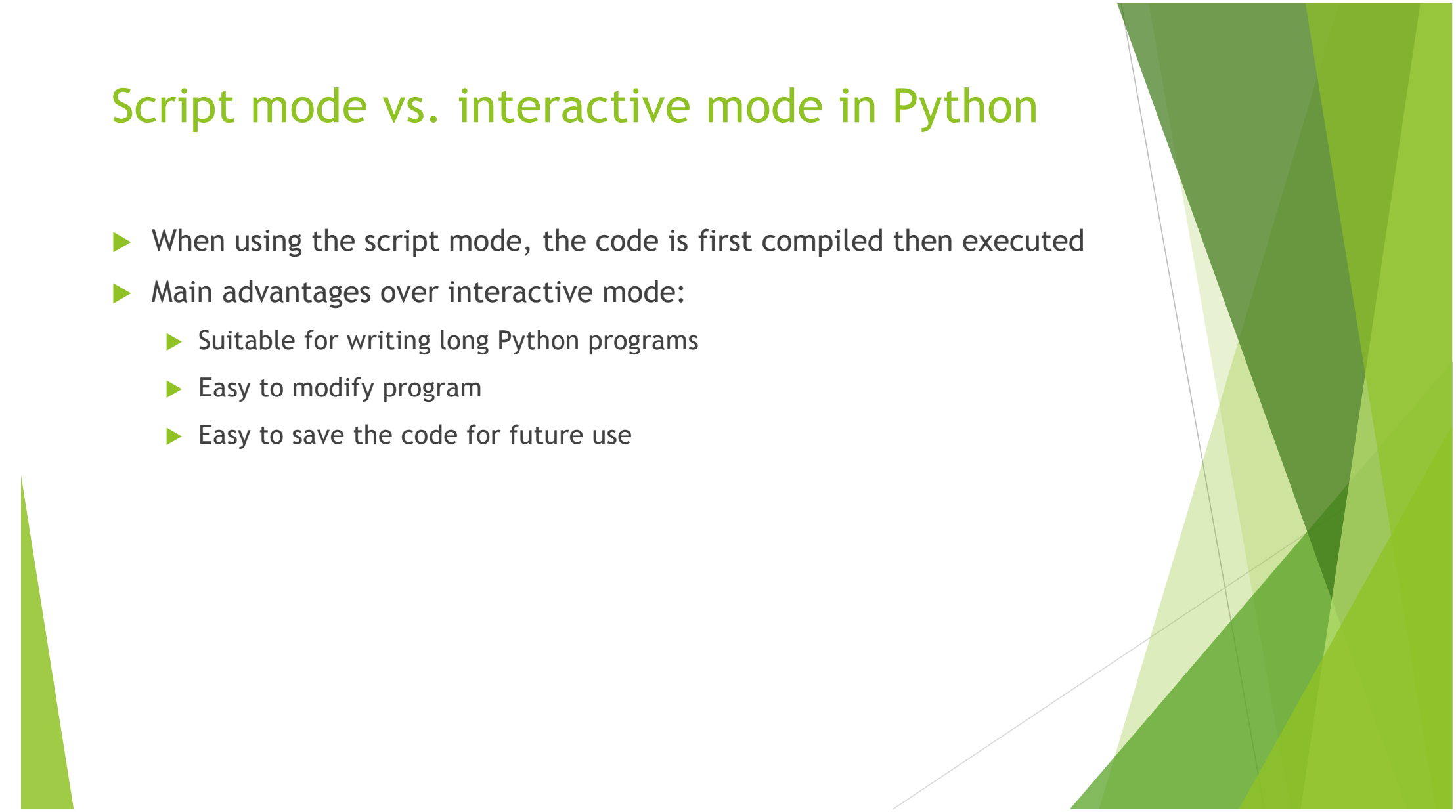
- Features:
  - Simple syntaxes: indentation rather than punctuation
  - Expressive language: fewer codes
  - Dynamic typing: no need to declare variables
  - Interpreted: no need to compile
  - Automatic memory management
- What is sacrificed – speed…

Compiler	Interpreter
Processes whole programs at once	Processes programs one instruction at a time
Translates programs to binary machine code	Executes programs by loading and translating instructions one by one on the fly
Needed only once after the program is completed	Runs each time the program is executed
Allows for detection of some errors prior to execution	All the errors are caught during execution
Need not be present in RAM during execution	Must be in RAM during a program's execution
Compiled programs usually run faster	Interpreted programs are usually slower

*This mainly refers to the interactive mode of Python

Script mode vs. interactive mode in Python

- When using the script mode, the code is first compiled then executed
- Main advantages over interactive mode:
  - Suitable for writing long Python programs
  - Easy to modify program
  - Easy to save the code for future use



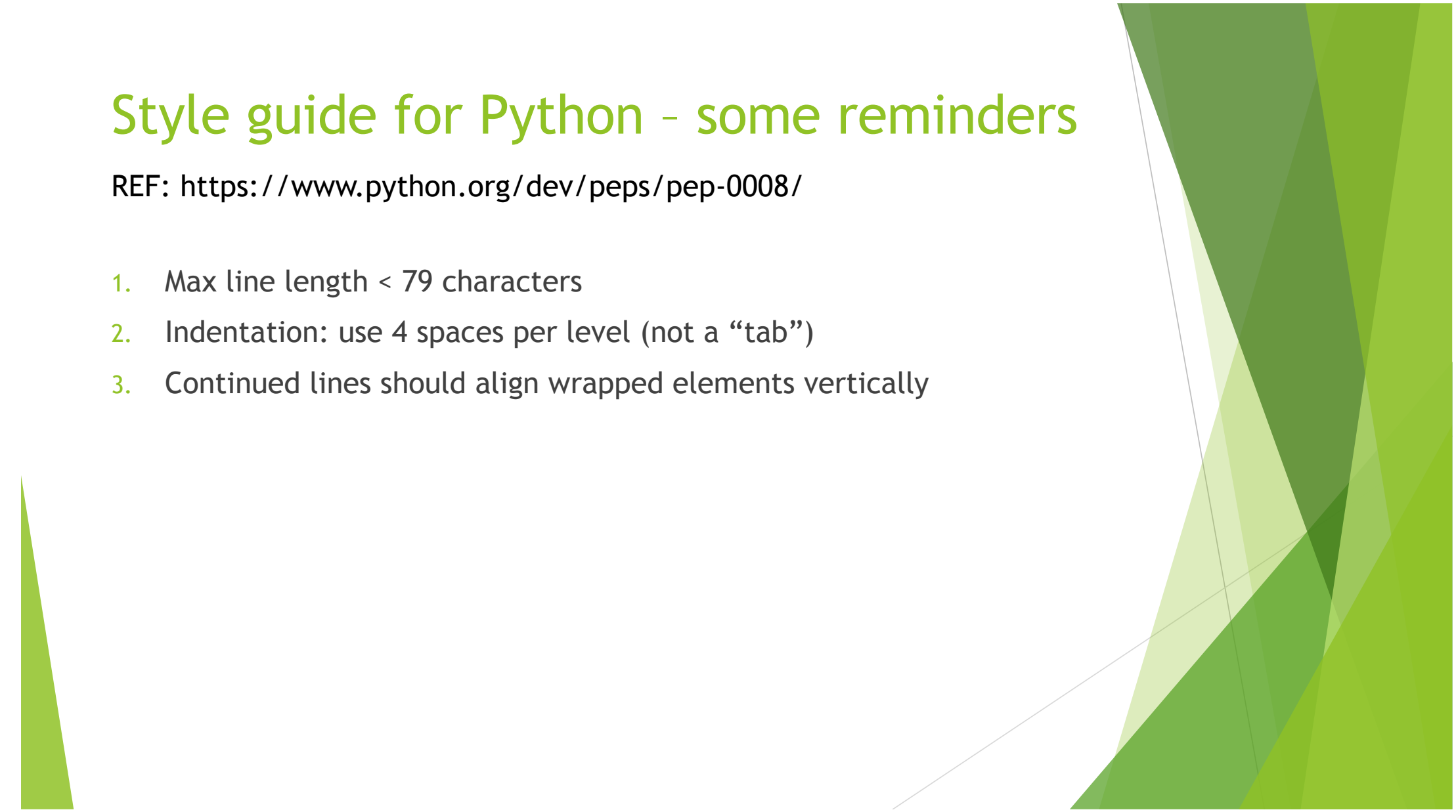
Why Python for scientific computing?

- Large community of users
- Plenty of scientific libraries and environments (e.g., numpy, scipy, matplotlib, scikit-learn, astropy…)
- Easy integration with highly optimized codes written in C/Fortran
- Good support for parallel programming and GPU computing
- Open sourced

Style guide for Python – some reminders

REF: https://www.python.org/dev/peps/pep-0008/

1. Max line length < 79 characters
2. Indentation: use 4 spaces per level (not a “tab”)
3. Continued lines should align wrapped elements vertically



Style guide for Python – some reminders

REF: https://www.python.org/dev/peps/pep-0008/

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the
rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

Style guide for Python – some reminders

REF: https://www.python.org/dev/peps/pep-0008/

```
# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

Style guide for Python – some reminders

REF: https://www.python.org/dev/peps/pep-0008/

4. Line break before a binary operator

```
# Yes: easy to match operators with operands
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Style guide for Python – some reminders

REF: https://www.python.org/dev/peps/pep-0008/

5. Import on the top of the files line by line

```
import os
import sys
```

```
import sys, os
```

```
from subprocess import Popen, PIPE
```

Switching between Python and Fortran – reminder #1

division.f90

```
program test

integer :: a = 1
print*, a/2

end program
```

division.py

```
a = 1
print(a/2)
```

- In Fortran, division between two integers (e.g., a/2) would result in an ***integer.*** Therefore, an integer of “1/2” would give “0” instead of “0.5”
- How would you modify division.f90 so that it gives the desired result of “0.5”?
- For scientific computing in Fortran, better use “ ***1.***” instead of “ ***1***” in the equations!

Switching between Python and Fortran – reminder #2

loop.f90

```
do i = 1, 5
  print*, "i = ", i
end do
```

loop.py

```
for i in range(1, 5):
  print("i = ", i)
```

- In Python, `range(start, end)` does not include "`end`"
- Similarly, `a[0:5]` in Python does not include `a[5]` but `a(1:5)` in Fortran does include `a(5)`
- By default, first array element is `a[0]` in Python but is `a(1)` in Fortran

Switching between Python and Fortran – more resources

- See a useful cheat sheet of commonly used syntaxes for Python programmers: https://github.com/wusunlab/fortran-vs-python *Highly recommended!*
- Fortran tutorial:
  - https://fortran-lang.org/en/learn/
  - https://www.tutorialspoint.com/fortran/index.htm

Numerical methods -- integration

Numerical integration

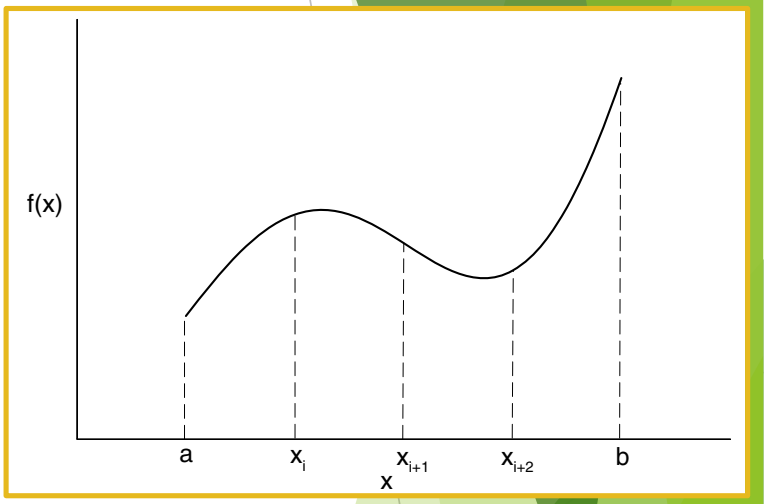
- Let's consider an integral of f(x) within the interval [a, b]. By definition:

$$\int_a^b f(x)dx = \lim_{h\to 0}\left[h\sum_{i=1}^{(b-a)/h} f(x_i)\right]$$

- To evaluate the integral numerically, the above equation is approximated as a finite sum over boxes of ***height f(x)*** and ***width w_i***:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} f(x_i)w_i$$

- Different algorithms use different ways of choosing f(x) and w_i
- There is no universally best algorithm because the result depends on f(x)
- Generally, precision increases as *N* gets larger until roundoff error dominates



Deal with singularities first

- One should avoid numerical integration of an integrand that contains a singularity
- Common ways of removing singularities:
  - breaking the interval down to several subintervals
  - by change of variables

$$I = \int_0^1 \frac{dt}{t^{1/2}}$$

$$x \equiv 1/t$$

$$= \int_1^\infty \frac{dx}{x^{3/2}} \qquad dx = -\frac{1}{t^2}dt = -x^2 dt$$

Deal with singularities first

- One should avoid numerical integration of an integrand that contains a singularity
- Common ways of removing singularities:
  - breaking the interval down to several subintervals
  - by change of variables
  - by subtracting them away

$$\int_{0.01}^{1} \frac{dx}{e^x - 1}$$

$$x \to 0 \qquad \frac{1}{e^x - 1} \to \frac{1}{x}$$

$$I = \int_{0.01}^{1} \left(\frac{1}{e^x - 1} - \frac{1}{x} \right) dx + \int_{0.01}^{1} \frac{dx}{x}$$

Can do it analytically

Integration algorithms

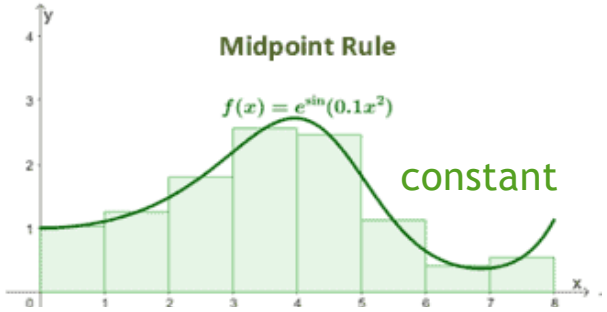
- Commonly used integration methods based on *N* equally spaced points for each ***sub-interval [a,b]***:

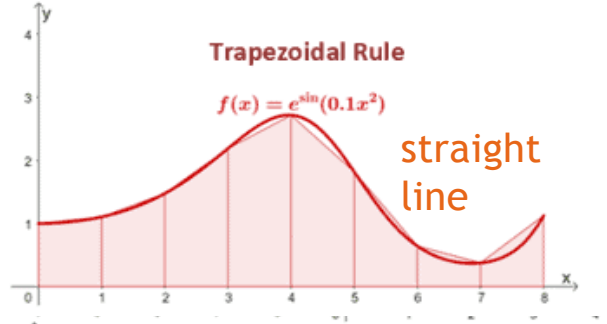
Midpoint rule: $$\int_a^b f(x)dx \sim (b-a)f(\frac{a+b}{2})$$

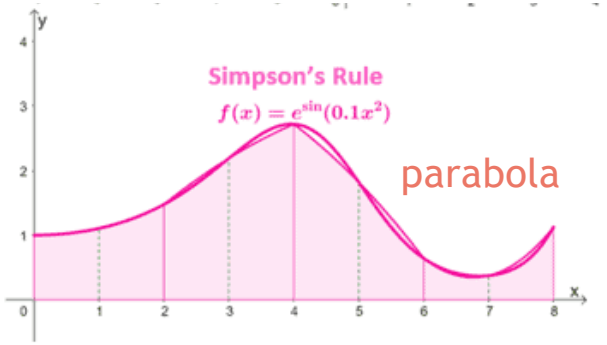
Trapezoid rule: $$\int_a^b f(x)dx \sim (b-a)(\frac{f(a)+f(b)}{2})$$

Simpson's rule: $$\int_a^b f(x)dx \sim \frac{(b-a)}{6}(f(a)+4f(\frac{a+b}{2})+f(b))$$

**To integrate over the whole range in x, one needs to sum over the areas for all sub-intervals*







Truncation error of the integration algorithms

- For the above methods, one can estimate the truncation errors using a *Taylor series expansion* about the midpoint $m = \frac{a+b}{2}$ of the interval [a,b]:

$$f(x) = f(m) + f'(m)(x-m) + \frac{f''(m)}{2}(x-m)^2 + \frac{f^{(3)}(m)}{6}(x-m)^3 + \frac{f^{(4)}(m)}{24}(x-m)^4 + \cdots.$$

- Integrating this expression from a to b, the odd-order terms drop out:

$$I(f) = f(m)(b-a) + \frac{f''(m)}{24}(b-a)^3 + \frac{f^{(4)}(m)}{1920}(b-a)^5 + \cdots$$

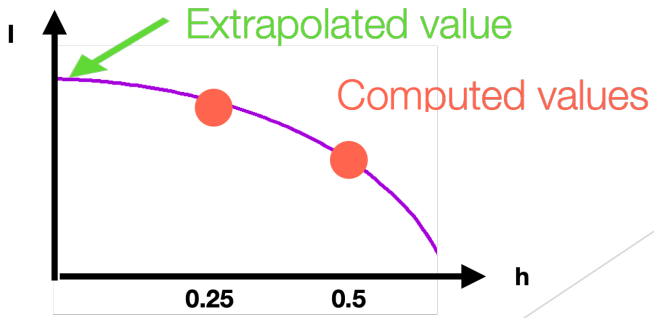
- The truncation error for the midpoint rule is thus $E_m = \mathcal{O}(\frac{[b-a]^3}{N^2})f^{(2)}$
- Similarly, the errors for Trapezoid and Simpson's rules are

$$E_t = O\left(\frac{[b-a]^3}{N^2}\right)f^{(2)} \qquad E_s = O\left(\frac{[b-a]^5}{N^4}\right)f^{(4)} \qquad \epsilon_{t,s} = \frac{E_{t,s}}{f}$$

Truncation error of the integration algorithms

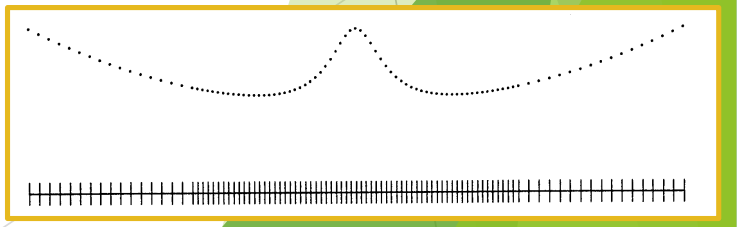
$$E_m = \mathcal{O}\left(\frac{[b-a]^3}{N^2}\right) f^{(2)}, E_t = \mathcal{O}\left(\frac{[b-a]^3}{N^2}\right) f^{(2)}, E_s = \mathcal{O}(\frac{[b-a]^5}{N^4}) f^{(4)}$$

- The error for the midpoint rule is on the same order as the trapezoid rule
- For functions that have well-behaved high derivatives, Simpson's rule should converge more rapidly than the other two methods
- Decreasing the interval $\boldsymbol{h = \frac{b-a}{N}}$ would decrease the truncation errors
- ***Romberg integration*** uses this behavior with $\boldsymbol{h}$ to iteratively obtain the true answer by extrapolation



Other integration algorithms that don't require equally spaced intervals

- ***Gaussian quadrature***: $\int_a^b f(x)dx \equiv \int_a^b W(x)g(x)dx \approx \sum_{i=1}^{N} w_i g(x_i)$
  - Separate a weighting function *W(x)* from the integrand
  - Specific distributions of *N* points and weights are chosen to make the truncation error vanish for *g(x)*.
- ***Adaptive quadrature method:***
  - Use two methods to evaluate integral (e.g., midpoint & trapezoid)
  - If their difference ($\propto$ error) is more than desired tolerance, divide into subintervals and repeat the procedure
  - This strategy leads to non-uniform sampling with many points in regions where the function is difficult to integrate



Integral of tabular data

- Sometimes we need to perform integral on ***discrete*** data instead of a continuous function, i.e., only $f(x_i)$ are available
- A typical approach is to integrate tabular data using ***piecewise interpolation,*** i.e., to find polynomials to approximate $f(x_i)$ in each subinterval separately
- For example, piecewise linear interpolation is essentially the trapezoid rule
- Higher order interpolation schemes (e.g., Hermit cubic, cubic spline interpolation) are excellent methods for this task

Double integrals

- Simplest strategy is to evaluate as a series of one dimensional integrals:

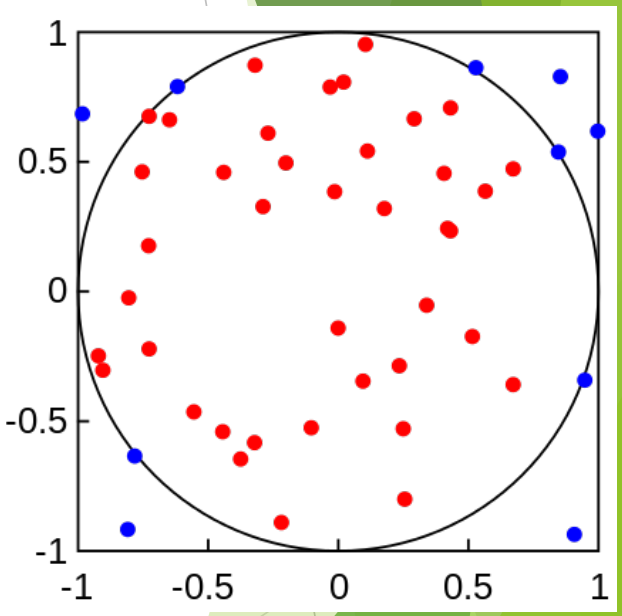
$$\int_{x=a}^{x=b}\int_{y=c}^{y=d} f(x,y)dxdy = \int_{x=a}^{x=b} g(x)dx$$

$$g(x) = \int_{c}^{d} f(x,y)dy$$

- Drawback – very inefficient for evaluating integrals of even higher (>3) dimensions

Monte Carlo integration

- A technique for numerical integration using ***random numbers***
- This method is ***non-deterministic***
- Very useful for higher-dimensional integrals
- **Example:** compute the area of a unit circle
  - Throw *N* points onto the square
  - $\frac{Area\ of\ circle}{Area\ of\ square} = \frac{\#\ red\ points}{N} = \frac{40}{50} = 0.8$
  - Area of circle = 0.8*(area of square) = 0.8*4 = 3.2 ~ π



Monte Carlo integration

- To perform an integral: $I = \int_{\Omega} f(\vec{x}) d^n \vec{x}$

  where the volume of a region Ω of dimension *n* is: $V = \int_{\Omega} d^n \vec{x}$

- Draw *N* samples in Ω, then the integral can be approximated by

$$I \simeq V \frac{1}{N} \sum_{i=1}^{N} f(\vec{x}) = V \langle f \rangle$$

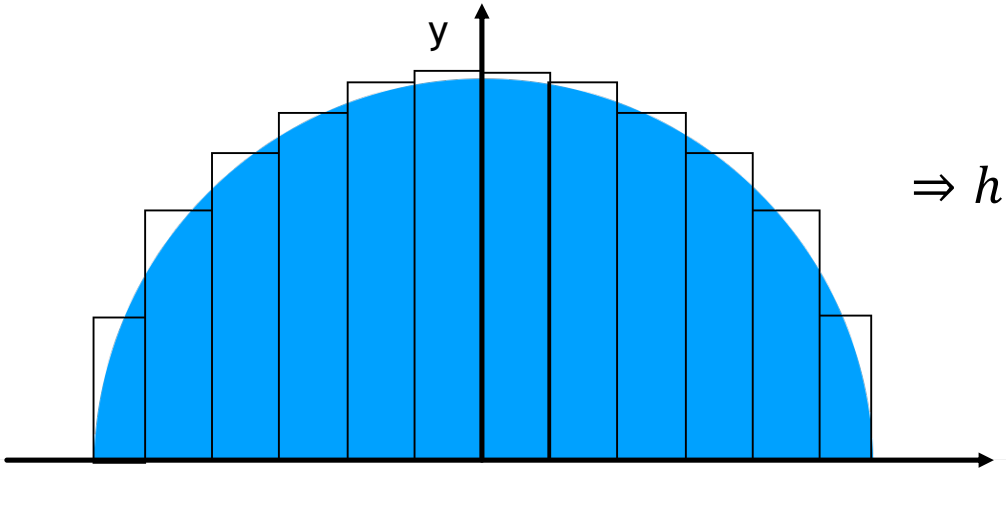
*In the previous example, $f(\vec{x}) = \begin{cases} 1, & for\ \vec{x} \in \Omega \\ 0, & otherwise \end{cases}$

- Error is from sample variance: $\propto 1/\sqrt{N}$
- The approximation gets more accurate as *N* increases

In-class exercise

Let's write a Fortran program to integrate the area of a unit circle

- The answer is π
- Let's use the ***midpoint rule*** for the integration, i.e., approximating the area by summing over the rectangles **x 2**
- Area of each rectangle is **dA = dx * h**



$$\because x^2 + y^2 = 1$$
$$\Rightarrow h = y(x) = \sqrt{1 - x^2}$$

Step 1

- Go to your exercise directory on CICA:

  ```
  cd astr660/exercise
  ```

- Write a Fortran program (pi1.f90) to evaluate the area of a unit circle using the midpoint rule. An example pseudo code is shown on the right
- Compile and execute the program:

  ```
  gfortran pi1.f90 -o pi1
  ./pi1
  ```

pi1.f90

```
program pi

   implicit none
   declare variables

   initialize N, dx

   area = 0.
   do i = 1, N
      x = midpoint of rectangle i
      h = height of rectangle i
      dA = dx * h
      area = area + dA
   enddo

   print*, "PI = ", 2.*area

end program
```

Step 2: use functions

- `cp pi1.f90 pi2.f90`
- Modify your code to compute the integral using ***functions*** in Fortran
- Compile and run the code

pi2.f90

```
program pi

   implicit none
   declare variables
   real :: my_func
   ....
   do i = 1, N
      x = midpoint of rectangle i
      h = my_func(x)
      ....
   enddo
   ....
end program

real function my_func(x)
   real :: x
   my_func = sqrt(1.0- x**2.)
   return
end function
```

Step 3: use subroutines

- cp pi2.f90 pi3.f90
- Modify your code to compute the integral using ***subroutines*** in Fortran
- Compile and run the code

pi3.f90

```
program pi

   implicit none
   declare variables
   ....
   ! replace the do loop
   call compute_integral(N, area)
   ....
end program

subroutine compute_integral(N, A)
   implicit none
   integer, intent(in) :: N
   real, intent(out) :: A
   ! declare other variables
   ! perform the do loop

   return
end subroutine compute_integral
```

Step 4: check convergence

- `cp pi3.f90 pi4.f90`
- Modify your code to compute the integral for ***different numbers of N***
- Compute the relative errors as a function of *N*
- Output the results to a file (pi_error.dat) using ***formatted output*** (see Slide 15)
- Compile and run the code
- Use your favorite plotting routine (e.g., Python or gnuplot) to read in the file and ***plot log(relative error) vs. log(N)***
- To get the bonus credit, submit ***pi4.f90 and the plot*** to the TAs by the end of today (9/29/2022)

pi4.f90

```
program pi

   implicit none
   declare variables
   real, parameter :: pi = 4.0*atan(1.0)
   integer, parameter :: NMAX
   integer, dimension(NMAX) :: n_iteration
   n_iteration = (/10, 100, 1000, 10000, &
                          100000, 1000000/)
   ! open a file "pi_error.dat"

   do i = 1, NMAX
      ! call the subroutine
      ! compute the relative error
      ! write results to file
   enddo

   ! close file
end program
```

Supplemental information

Formatted output in Fortran

Purpose		*Edit Descriptors*	
Reading/writing **INTEGER**s		**I***w*	**I***w.m*
Reading/writing **REAL**s	Decimal form	**F***w.d*	
	Exponential form	**E***w.d*	**E***w.d***E***e*
	Scientific form	**ES***w.d*	**ES***w.d***E***e*
	Engineering form	**EN***w.d*	**EN***w.d***E***e*
Reading/writing **LOGICAL**s		**L***w*	
Reading/writing **CHARACTER**s		**A**	**A***w*
Positioning	Horizontal	***n*X**	
	Tabbing	**T***c*	**TL***c* and **TR***c*
	Vertical	**/**	
Others	Grouping	***r*(....)**	
	Format Scanning Control	**:**	
	Sign Control	**S**, **SP** and **SS**	
	Blank Control	**BN** and **BZ**	

- ***w***: the number of positions to be used
- ***m***: the minimum number of positions to be used
- ***d***: the number of digits to the right of the decimal point
- ***e***: the number of digits in the exponent part