Numerical Methods

Lecture 4, Computational Astrophysics (ASTR660)

Hsiang-Yi Karen Yang, NTHU, 10/6/2022

Announcements

- *HW2* will be posted on eLearn today. ***Due at 14:20 on 10/13 (Thu).*** Late submission within one week will receive ***75%*** of the credit
- Please follow TAs' instructions for submission of homework assignments and bonus credits
- Schedule for the midterm presentation has been posted on eLearn. Please let me know ASAP if you have any question. Here's the link to the spreadsheet:

https://docs.google.com/spreadsheets/d/16zh4Xw3bOGANhSBlbKo6is6DRghPC3Nul8tmeOVIl8c/edit?usp=sharing

Schedule for midterm presentations (project proposal)

Date	Time	Name	Title
11/3/22	14:25-14:40	劉欣旻	
	14:40-14:55	李佳倫	
	14:55-15:10	石郡翰	
	15:30-15:45	凌志騰	
	15:45-16:00	張修瑜	
	16:00-16:15	簡嘉成	
	16:15-16:30	謝明學	
11/10/22	14:25-14:40	劉一璠	
	14:40-14:55	郭弈翔	
	14:55-15:10	胡英祈	
	15:30-15:45	周育如	
	15:45-16:00	吳耕緯	
	16:00-16:15	鄭文淇	
	16:30-16:45	龔一桓	
	16:45-17:00	考司圖巴	

Previous lecture…

- Introduction to *Fortran* & *Python*
  - *Compiled* languages like C/C++/Fortran remain widely used for scientific computing because of their efficiency
  - Compiled languages follow more strict syntaxes and variable declarations
  - Python, as an *interpreted* language, is much easier to code but slower
- Numerical integration
  - Algorithms using equally spaced intervals: *midpoint, trapezoid, Simpson's rules*
  - Truncation error decreases with increasing N => idea behind Romberg's integration
  - Algorithms using non-equally spaced intervals: Gaussian quadrature, adaptive quadrature method
  - *Monte Carlo method*: integration using random numbers, useful for higher dimensions

This lecture…

- In-class exercise: getting more familiar with Fortran & numerical integration
- More on numerical methods & exercises
  - Differentiation
  - Monte-Carlo simulations & random numbers

In-class exercise

Integration algorithms

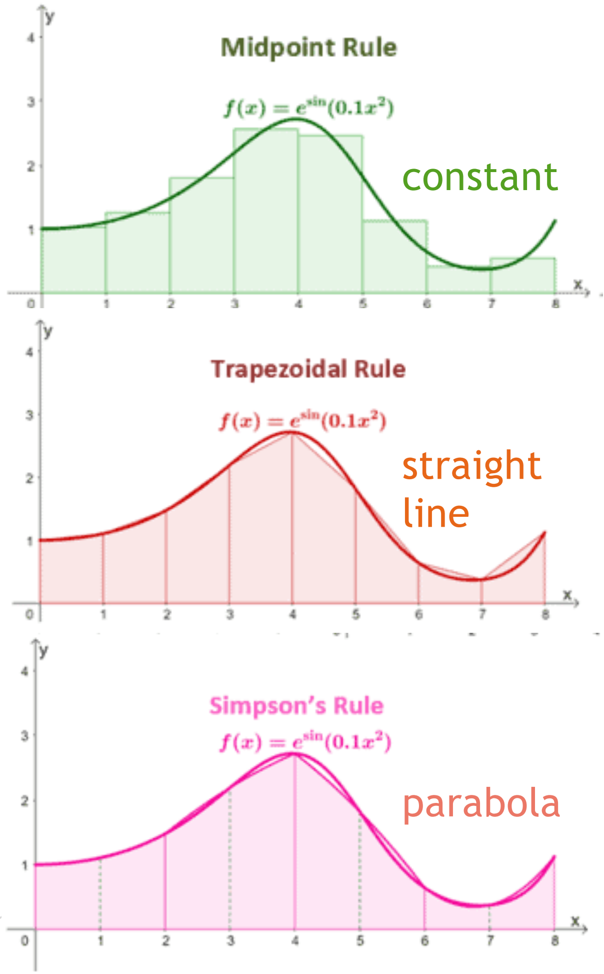
- Commonly used integration methods based on *N* equally spaced points for each ***sub-interval [a,b]***:

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

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

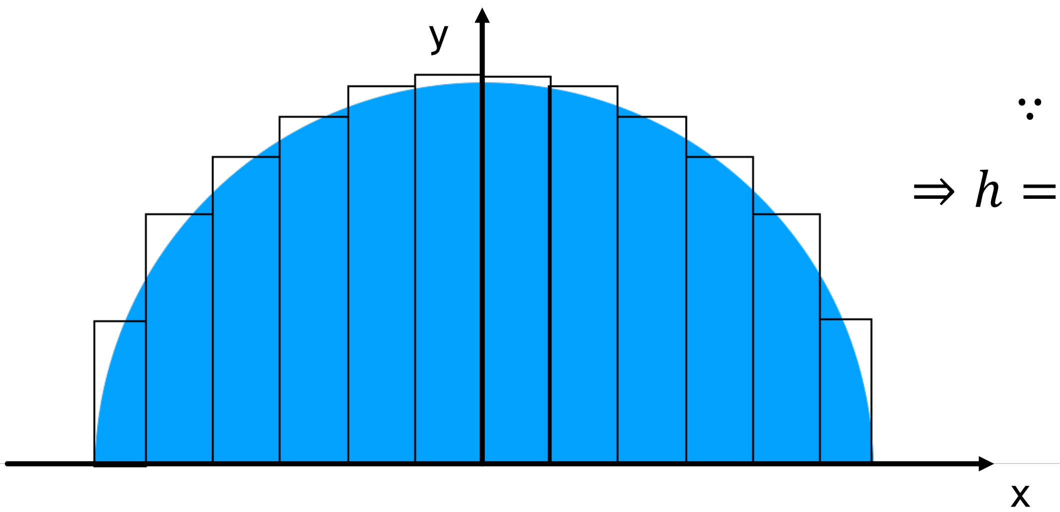
Simpson's rule: $$\int_a^b f(x)dx \sim \frac{(b-a)}{6}(f(a)+4f(\frac{a+b}{2})+f(b))$$

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



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

- Connect to CICA and load the Fortran compiler:

  ```
  module load pgi
  ```

- 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 ε as a function of N
- Output the results to a file (pi_error.dat) using ***formatted output*** (see Slide 16 in Lecture 3)
- Compile and run the code
- Use your favorite plotting routine (e.g., Python or gnuplot) to read in the file and ***plot $\log_{10}(\varepsilon)$ vs. $\log_{10}(N)$***
- To get the bonus credit, submit ***pi4.f90 and the plot*** to the TAs by the end of today (10/6/2022)

pi4.f90

```
program pi

    implicit none
    declare variables
    real, parameter :: pi = 4.0*atan(1.0)
    integer, parameter :: NMAX = 8
    integer, dimension(NMAX) :: n_iteration
    n_iteration = (/10, 100, 1000, 10000, &
                        100000, 1000000, 10000000/)
    ! 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
```

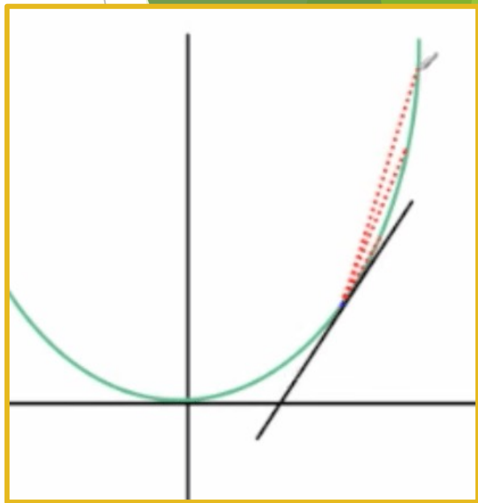
Numerical methods -- differentiation

Differentiation

- In physics/astrophysics, we deal with differentiation all the time
- For example, velocity $v(t) = dx/dt$, acceleration $\mathrm{a}(t) = \frac{dv}{dt} = \frac{d^2x}{dt^2}$
- The exact definition of a derivative is:

$$\frac{df(x)}{dx} \stackrel{\text{def}}{=} \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- However, when evaluating the derivative ***numerically***, one cannot use infinitesimally small *h*, because *h* would then fluctuate between 0 and machine precision ϵ_m due to roundoff errors



Algorithm #1: forward difference

- From Taylor expansion:

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

- We could define the ***forward-difference derivative*** to be:

$$\boldsymbol{f'(x) \equiv \frac{f(x+h) - f(x)}{h}} \simeq f'(x) + \frac{h}{2} f''(x) + \cdots$$

- ***Truncation error is $\propto \mathcal{O}(h)$ (1st order)*** and gets smaller when h is smaller (until subtraction cancellation due to roundoff error kicks in)
- **Example:** $f(x) = a + bx^2$, for which the exact derivative is $f'(x) = 2bx$

  Using forward difference we obtain:

$$f'(x) \equiv \frac{f(x+h) - f(x)}{h} = 2bx + bh$$

Algorithm #2: central difference

- From Taylor expansion: $f(x+h) = f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \ldots$

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

- We could define the ***central-difference derivative*** to be:

$$\boldsymbol{f'(x) \equiv \frac{f(x+h) - f(x-h)}{2h}} \simeq f'(x) + \frac{h^2}{3}f'''(x) + \cdots$$

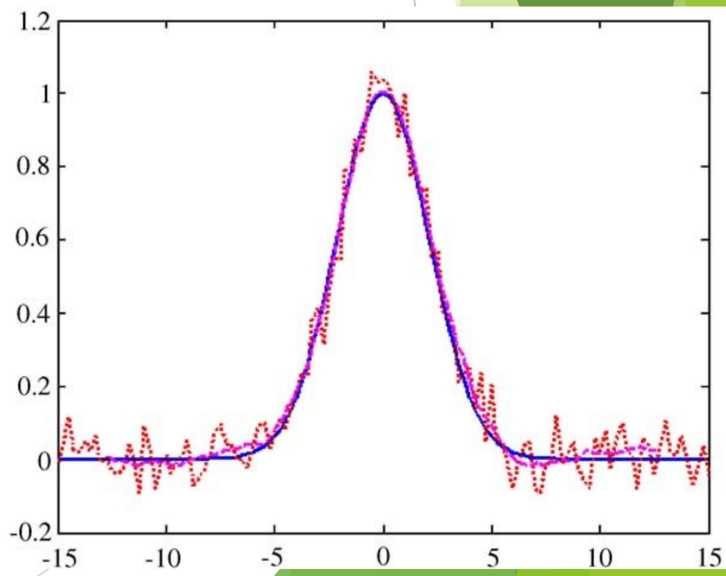
- ***Truncation error is*** $\boldsymbol{\propto \mathcal{O}(h^2)}$ ***(2nd order).*** Error smaller than forward difference if f(x) is well behaved
- **Example:** $f(x) = a + bx^2$, for which the exact derivative is $f'(x) = 2bx$

  Using central difference we obtain:

$$f'(x) \equiv \frac{f(x+h) - f(x-h)}{2h} = 2bx$$

Higher-order algorithms

- It's possible to construct algorithms which make even higher-order terms vanish
- Since truncation error is $\propto f^{(n)}(x)$, these methods work well for well-behaved functions
- However, for noisy functions/data, it may be better to first fit the data with some analytical function then perform the derivative



Second derivatives

- Second derivates are needed when we need to compute, e.g., force on a particle from its position *x(t)*:

$$F = ma = m\frac{d^2x}{dt^2}$$

- From the central-difference method:

$$f'(x) \simeq \frac{f(x+h/2) - f(x-h/2)}{h}$$

- The second derivative is then the central difference of the 1st derivative:

$$\begin{aligned} f^{(2)}(x) &\simeq \frac{f'(x+\frac{h}{2}) - f'(x-\frac{h}{2})}{h}, \\ &\simeq \frac{[f(x+h) - f(x)] - [f(x) - f(x-h)]}{h^2} \\ &\simeq \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} \end{aligned}$$

Eq. (1) -- Better expression to use

Eq. (2) -- Although more compact, this expression would be more subject to cancellation of two large numbers

In-class exercise – implement the second derivative

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 Python program:

   ```
   cd astr660/exercise
   ```

   ```
   vi ex4.py
   ```

3. In ex4.py, write a program to calculate the second derivative of cos(x) using the central difference algorithms
   - Use Eq. (1) in the last slide first. Evaluate the relative error ε at $x = 1$
   - Start with $h = \pi/10$ and keep reducing h until roundoff error dominates. Plot $\log_{10}(\varepsilon)$ vs. $\log_{10}(h)$
   - Use Eq. (2) instead and overplot the results
4. To get the bonus point, submit the ***code and the plot*** to the TAs by the end of today (10/6/2022)

If you've completed both exercises, choose one of pi4.f90 and ex4.py for the submission for bonus points

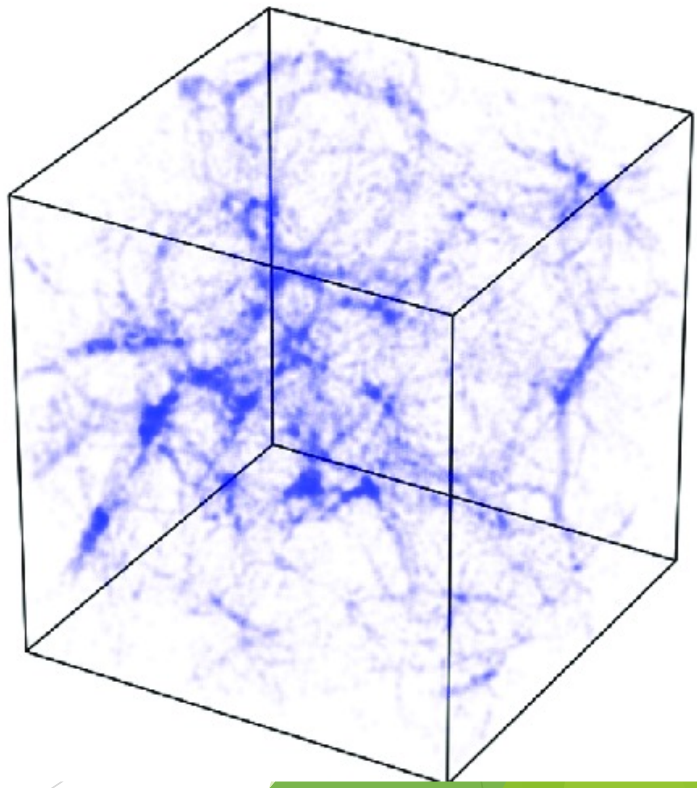
Numerical methods – Monte-Carlo simulations & random numbers

Monte Carlo methods/simulations

- Are a broad class of computational algorithms that rely on ***repeated random sampling*** to obtain numerical results
- Was developed in the 1940s during WWII by Stanislaw Ulam and John von Neumann for simulating outcomes of nuclear weapons
- Is named after the Monte Carlo Casino in Monaco
- Is particularly useful for studying
  - Nondeterministic/stochastic processes
  - Deterministic systems that are too complicated to model analytically
  - Deterministic problems whose high dimensionality makes standard discretizations infeasible (e.g., Monte Carlo integration)

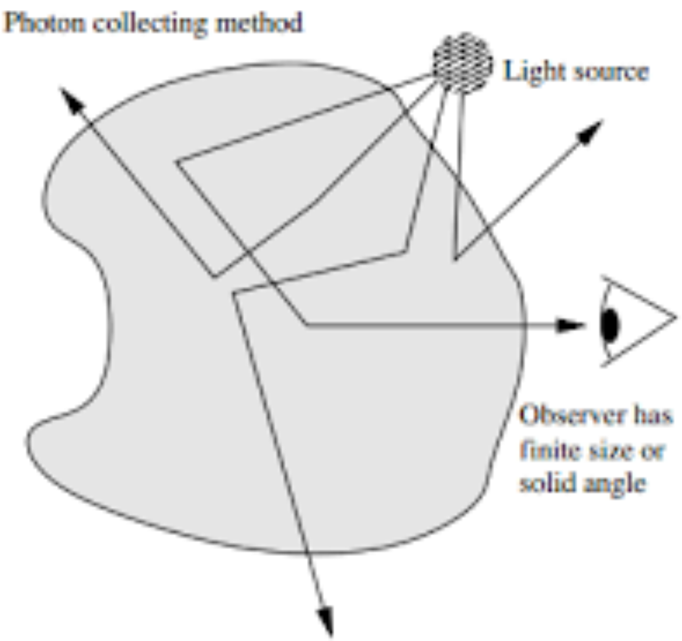
Monte Carlo methods are widely used in astrophysics

- **Example #1: N-body simulations**
- Applications: dark matter in cosmological simulations, charged particles in plasma simulations
- Motivation: it is infeasible to evolve a large grid of the distribution function of particles, $\boldsymbol{f(\vec{x}, \vec{v}, t)}$, required to solve the time-dependent 3D collisionless Boltzmann equation (also called the Vlasov equation)
- N-body simulations use *N* particles to discretize and sample $f(\vec{x}, \vec{v}, t)$ in the phase space



Monte Carlo methods are widely used in astrophysics

- **Example #2: radiative transfer (RT) simulations**
- Applications: computing emission properties/spectra from a source through a given medium
- Motivation: it is infeasible to solve the RT equation by integrating over optical depth, angle, and frequency over N^3 gird points
- Monte Carlo RT simulations use N particles to sample large numbers of photons and determine their trajectories for given optical depth & probabilities of absorption/scattering



Key requirements for Monte Carlo simulations

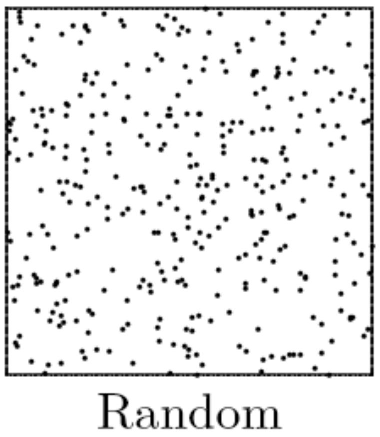
1. Knowledge of relevant probability distributions
2. Supply of random numbers
3. Using large number of trials because error $\propto 1/\sqrt{N}$

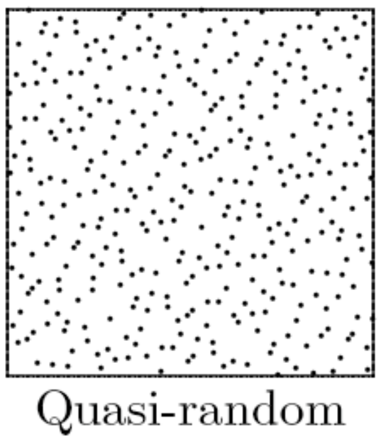
What are “random” numbers?

- Randomness is difficult to define, but is usually associated with
  - unpredictability
  - unrepeatability
- However, for computer algorithms, we often need a series of random numbers that are ***deterministic*** and ***repeatable*** for debugging and verification purposes
- *Pseudo random numbers*
  - a series of numbers generated by computer that appears random
  - they are in fact deterministic and reproducible
  - because only a finite sequence of numbers can be represented, they must eventually repeat

Random vs. quasi-random sequence

- Truly random sequences tend to exhibit random clumping
- However, for some applications, achieving ***reasonably uniform*** coverage of sampled volume can be more important than whether sample points are truly random
- ***Quasi-random sequences***, which are carefully constructed to give uniform coverage while maintaining random appearance, are sometimes useful





Random number generators

A good (pseudo) random number generator should have the following properties:

- *Random pattern*: passes statistical tests of randomness
- *Long period*: goes as long as possible before repeating
- *Efficiency*: executes rapidly and requires little storage
- *Repeatability*: produces same sequence if started with same initial conditions
- *Portability*: runs on different kinds of computers and is capable of producing same sequence on each

Linear congruential generator

- There are many algorithms available. One should choose an algorithm best suited for the problem at hand
- ***Linear congruential generator (LCG)*** uses the following formula to generate the pseudo random numbers:

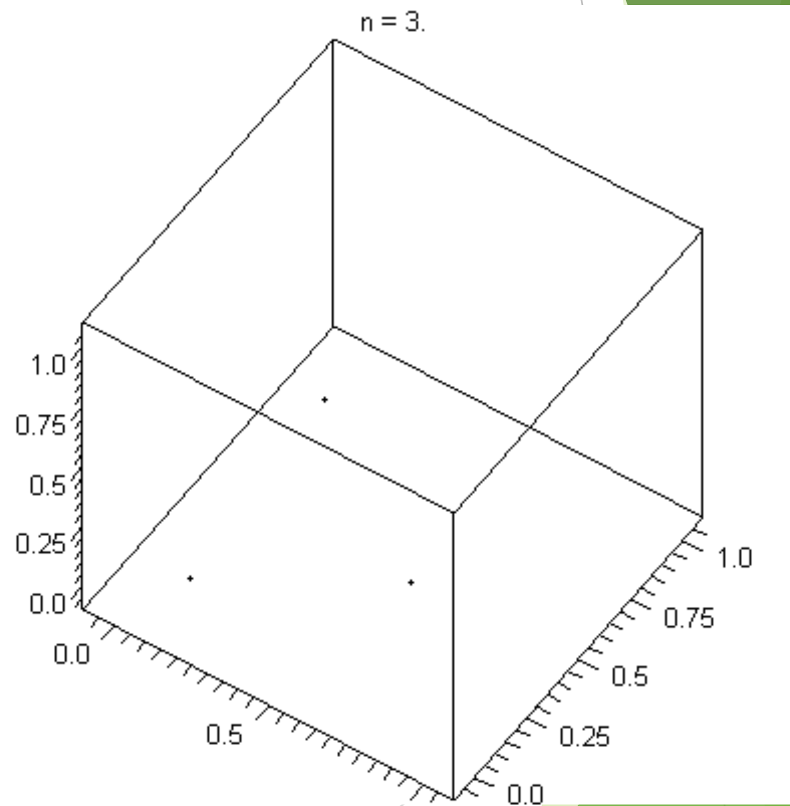
$$X_{n+1} = (aX_n + b)\ mod\ m$$

where a, b and m are large integers, which should be carefully chosen

- The start value X_0 is the "***seed***", which uniquely determines the sequence
- Maximum number this method can produce is $m-1$
- Example parameters used for IBM C/C++: $m = 2^{31}$, $a = 1103515245$, $b = 12345$

Linear congruential generator

- Advantage: simple, easy to implement, small memory requirement
- One drawback: the points generated in multi-dimensions may be correlated, forming ”hyperplanes”
- Not suitable for applications that require high-quality randomness



Other pseudo random number generators

- LGCs were widely used in the 2nd half of the 20th century; more generators were developed later to overcome its shortcomings
- Mersenne Twister (1997): efficient, period of $2^{19937} - 1$, used in IDL/R/Python/Julia/MATLAB
- Xorshift generators (2003)
- WELL generators (2006)

Words of caution when using the random number generator in numpy!!

See test_random1.py under directory /data/hyang/shared/astro660CompAstro/L4exercise

```
import numpy as np

N = 10

print(np.random.random(N))

for i in range(N):
  print(np.random.random())
```

Q1: Do you get the same sequence by calling np.random.random(N) vs. calling np.random.random() for *N* times?

Q2: What should you do if you want to generate the same sequence using these two methods or if you want to make the code reproducible?

Use np.random.seed() to make the results reproducible (see test_random2.py)

Words of caution when using the random number generator in numpy!!

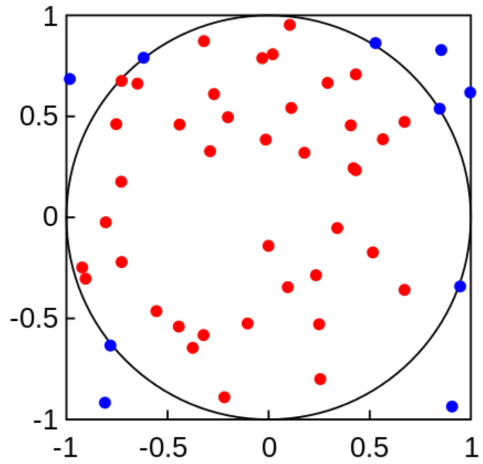
See test_random3.py under directory /data/hyang/shared/astro660CompAstro/L4exercise

```
import numpy as np

N = 10
seed = 100 # this is arbitrary

np.random.seed(seed)
print(np.random.random(N))

np.random.seed(seed)
for i in range(int(N/2)):
  x = np.random.random()
  y = np.random.random()
  print(x, y)
```

Say you'd like to use this code for Monte Carlo integration

Q1: What is the potential problem of the *x* and *y* values generated in this example?

A: They are not totally independent of each other since they are from the same sequence. They are random when combined but not necessarily random individually



Q2: How would you modify this code if you'd like to generate independent sequence for *x* and *y*?

Summary

- Numerical differentiation
  - Forward difference: $f'(x) \equiv \dfrac{f(x+h) - f(x)}{h}$*, error is* $\propto \mathcal{O}(h)$ *(1st order)*
  - Central difference: $f'(x) \equiv \dfrac{f(x+h) - f(x-h)}{2h}$*, error is* $\propto \mathcal{O}(h^2)$ *(2nd order)*
  - 2nd derivatives (central difference): $f^{(2)}(x) \equiv \dfrac{[f(x+h) - f(x)] - [f(x) - f(x-h)]}{h^2}$
- ***Monte Carlo simulations*** uses repeated random sampling to obtain numerical results, requiring (i) known probability distributions, (ii) random numbers, (iii) large sample size
- ***Pseudo random numbers*** are sequence of numbers generated by computers that appear to be random but are deterministic and repeatable. Generators must be chosen carefully to suit your applications

Quick exercise – Monte Carlo integration

1. On CICA, move into your exercise directory and create a Python program:

   ```
   cd astr660/exercise
   ```

   ```
   vi pi5.py
   ```

2. In pi5.py, write a program to use the *Monte Carlo* method for evaluating the area of a unit circle
   - Throw *N = 10000* points onto the square (please draw *x* and *y* in two *independent* random sequences)
   - Count # points within the circle
   - $$\frac{Area\ of\ circle}{Area\ of\ square} = \frac{\#\ points\ within\ circle}{N}$$