Lecture 1:
Floating Point and ODEs

Sasha Tchekhovskoy
atchekho@northwestern.edu

Thanks K. Hahn for some slides!

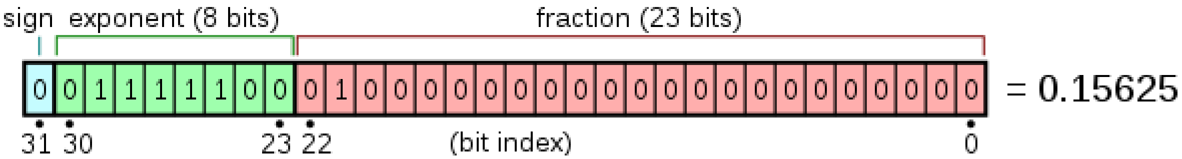
Northwestern | CIERA
CENTER FOR INTERDISCIPLINARY EXPLORATION AND RESEARCH IN ASTROPHYSICS

What's Floating? Point!

- How does computer represent the number π? Floating point!

binary32 (IEEE 754)

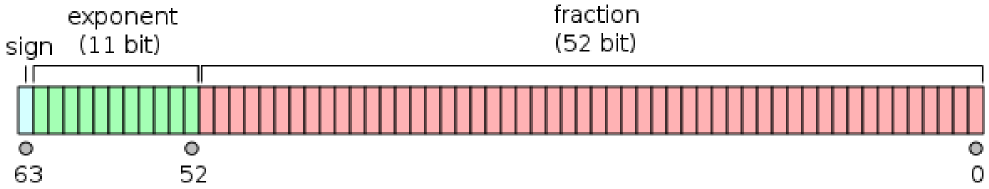
float32 (python)



$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2-127} \times (1.b_{22}b_{21}\dots b_0)_2 = 1.25 \times 2^{124-127} = 0.15625$$

binary64 (IEEE 754)

float64 (python)



$$(-1)^{\text{sign}}(1.b_{51}b_{50}\dots b_0)_2 \times 2^{e-1023}$$

Source: Wikipedia

Is sin(π) zero?

```
f = np.sin(np.pi)
print("sin(pi) = %g" % f)
print("Is it non-zero?")
if 0 == f: print("Yes!")
else: print("No")
```

```
sin(pi) = 1.22465e-16
Is it non-zero?
Yes!
```

- Comparing floating point numbers is dangerous
- Always leave room for round-off error:
  - bad: `a == b`
  - good: `fabs(a - b) < 10*eps`
- `epsm = np.finfo(np.float64).eps`
- On my machine, it is `2.220446e-16`
- Defined as the smallest number resolvable relative to unity:
  `(1.0 + eps) != 1.0 and`
  `(1.0 + eps/2) == 1.0`
- `How would you compute it?`

```
(1.0 + epsm)   != 1.0
(1.0 + epsm/2) == 1.0
```

Let us compute eps for np.float64!
Can you do it inside your jupyter notebook?
Jupyter notebook: atchekho/cofi

Let's compute a derivative!

Forward difference:

1st order truncation error

$$f'(x) \approx \frac{f(x+h)-f(x)}{h} = f' + \boxed{\frac{1}{2}hf''} + O(h^2)$$

$$f(x+h) = f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(x) + O(h^3)$$

Central difference:

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

$$= f' + \boxed{\frac{1}{6}h^2 f'''(x)} + O(h^3)$$

2nd order truncation error

Two sources of error:

1. round-off error: $\epsilon_r \sim \epsilon_m |f|/h$ $\quad$ $\epsilon_r \sim \epsilon_m |f|/h$
2. truncation error: $\epsilon_t \sim h\,|f''|$ $\quad$ $\epsilon_t \sim h^2 f'''$

Minimize $\epsilon_r + \epsilon_t$:

$\epsilon_r \sim \epsilon_t \quad \rightarrow h \sim \sqrt{\frac{\epsilon_m f}{f''}} \approx \sqrt{\epsilon_m}\, x_c$ $\quad$ $h \sim \epsilon_m^{1/3} x_c$ $\quad$ ($x_c = \sqrt{f/f''}$ = "curvature scale")

Relative error:

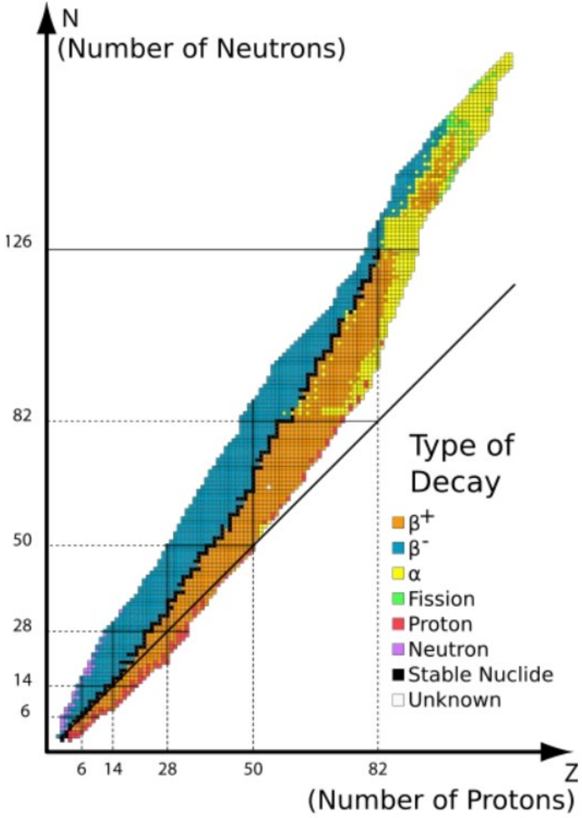
$$\frac{\epsilon_r + \epsilon_t}{|f'|} \sim \sqrt{\epsilon_m}\sqrt{\frac{f\,f''}{f'^2}} \sim \sqrt{\epsilon_m}$$

$$\frac{\epsilon_r + \epsilon_t}{|f'|} \sim \epsilon_m^{\frac{2}{3}} \left(\frac{f^2\,f''}{f'^3}\right)^{\frac{1}{3}} \sim \epsilon_f^{2/3}$$ (Numerical recipes)

Let us compute derivatives for $\sin(x)$ and compute the error!

Radioactive Decay

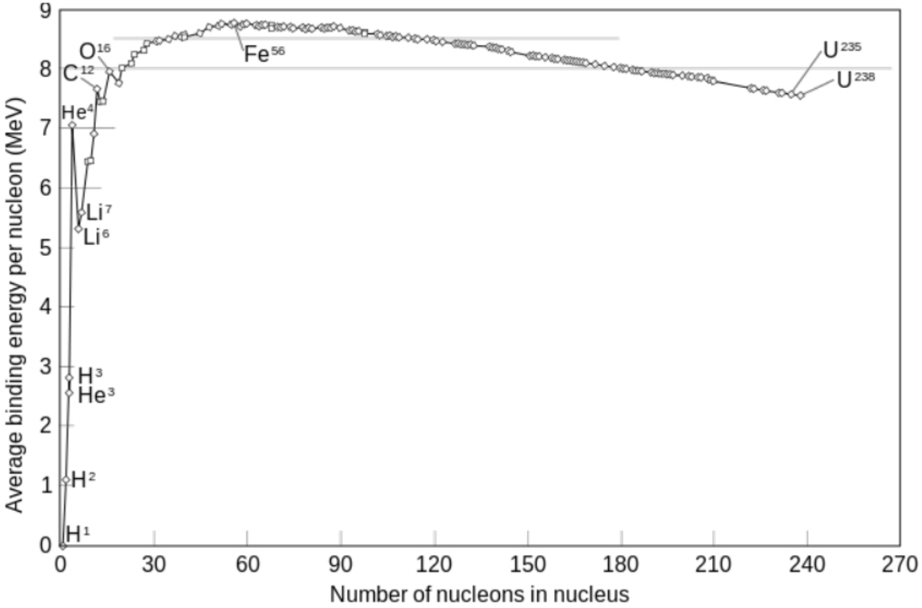
- Several types of Radioactive decay, categorized by decay products
  - Different forces at work
    - Alpha & gamma (fission) : interplay of electrostatic and residual strong nuclear
    - Beta : the weak force
- Elements with >82 protons have no stable isotopes
  - U235 has 92p, 146n



Slides courtesy K.Hahn

Why 82?

- Specific binding energy starts to decrease
  - Nucleus too large for residual strong force to hold together, electrostatic repulsion dominates
  - Energetically favorable to split into lighter daughters



Universal Law of Radioactive Decay

- All types of decay follow the same statistical behavior

$$dN/dt = -N/\tau$$
$$N(t) = N(0)e^{-t/\tau}$$

  - Mean lifetime of a nucleus is $1/\tau = \lambda$

- Applicable to any process with a probability of change proportional to instantaneous value
  - Capacitor discharge, heat transfer, etc

Numerical Solutions

- General method for numerically solving ODEs

involves Taylor expansion

$$N(t+\Delta t) = N(t) + (dN/dt)\,\Delta t + \tfrac{1}{2}\,(d^2N/dt^2)(\Delta t)^2 + \dots$$

- Euler method: stop expansion at first order (like in forward differencing!):

$$\frac{N(t+\Delta t) - N(t)}{\Delta t} = \frac{dN}{dt} + \frac{1}{2}\frac{d^2N}{dt^2}\Delta t + \cdots \approx -\lambda$$

  - Local truncation error in N: 2nd order, $(\Delta t)^2$
  - Global Truncation error in N: 1st order, (Δt)

Runge-Kutta Methods

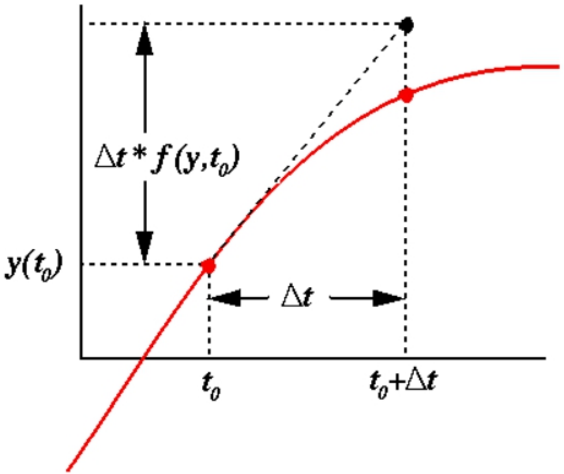
- The more general class of explicit ODE solvers
  - Euler method is a first order RK approach
  - Actually, "ODEs" means ODE initial value problems at t moment
- Re-statement of the problem:

$$dx(t)/dt = f(x,t)$$

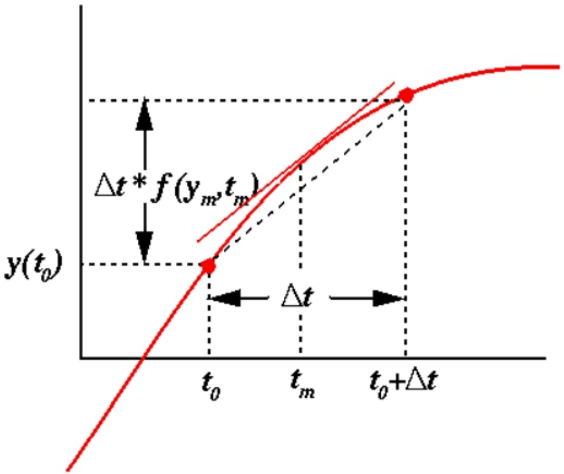
$$x(t+\Delta t) = x(t) + (dx/dt)\,\Delta t + (½)(d^2x/dt^2)(\Delta t)^2 + \ldots$$

Euler says: $x(t_{i+1}) \approx x(t_i) + f(x(t_i),t_i)\,\Delta t$

- The (local) error in our projection is 2nd order and related to the curvature of the function x(t) between the steps



- We could get the exact solution if we knew just the right point (t_m) at which to evaluate the slope



- RK methods are iterative approaches that try to zero-in on the solution

O(2) Runge-Kutta

$$t' = t + (1/2)\,\Delta t$$

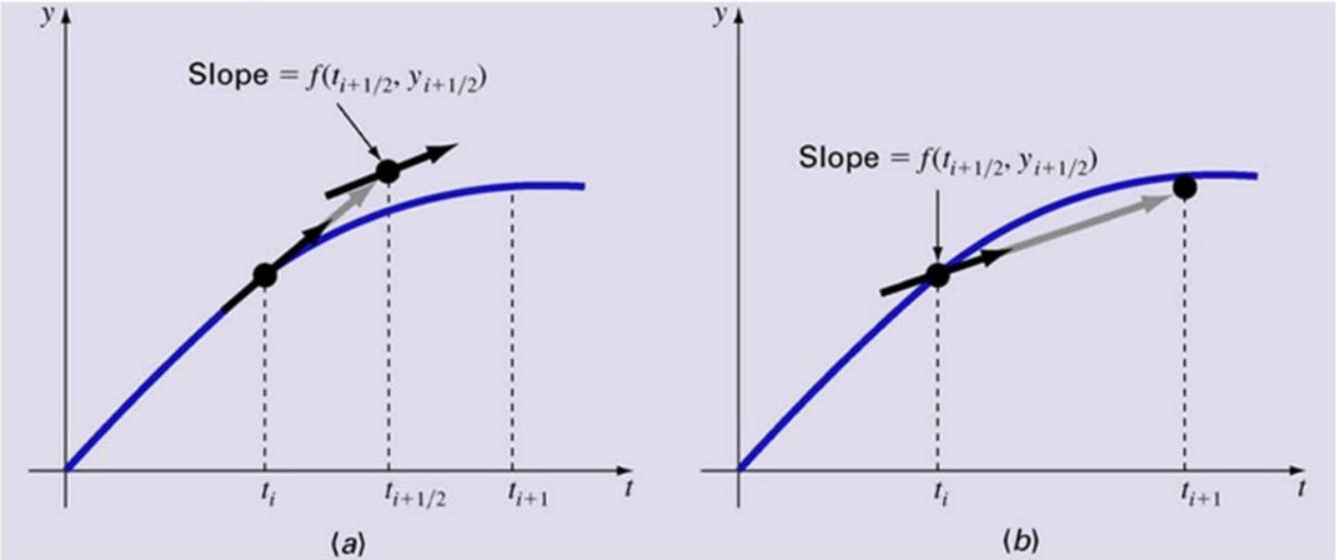
$$x' = x(t) + (1/2) f(x(t),t)\,\Delta t$$

$$x(t+\Delta t) = x(t) + f(x',t')\,\Delta t$$

where

- t' is the midpoint of the interval
- x' is the Euler approximation of x(t) at t'

- Computational costs increase by ~2x
  - But LTE, GTE goes as Δt^3, Δt^2 !


y
Slope = $f(t_{i+1/2}, y_{i+1/2})$
t_i
$t_{i+1/2}$
t_{i+1}
t
(a)
y
Slope = $f(t_{i+1/2}, y_{i+1/2})$
t_i
t_{i+1}
t
(b)

O(4) Runge-Kutta

$x_1' = x(t)$

$x_2' = x(t) + (1/2)f(x_1',t_1')\Delta t$

$x_3' = x(t) + (1/2)f(x_2',t_2')\Delta t$

$x_4' = x(t) + f(x_3',t_3')\Delta t$

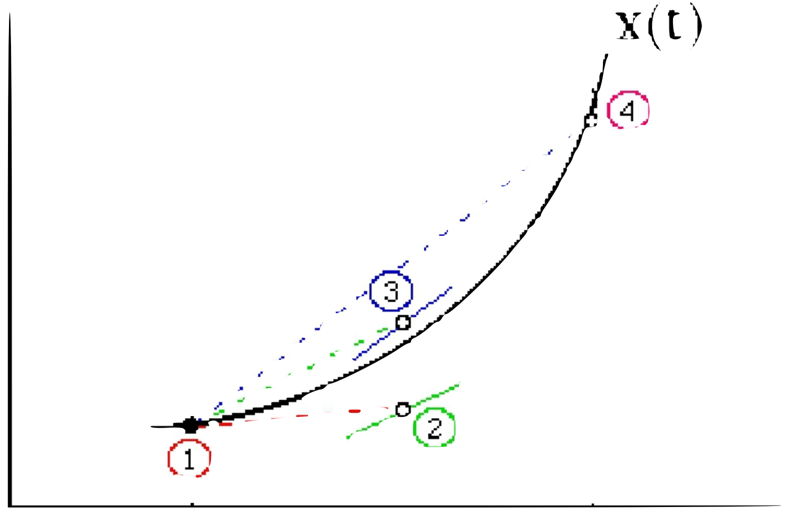
$t_1' = t$

$t_2' = t + (1/2)\Delta t$

$t_3' = t + (1/2)\Delta t$

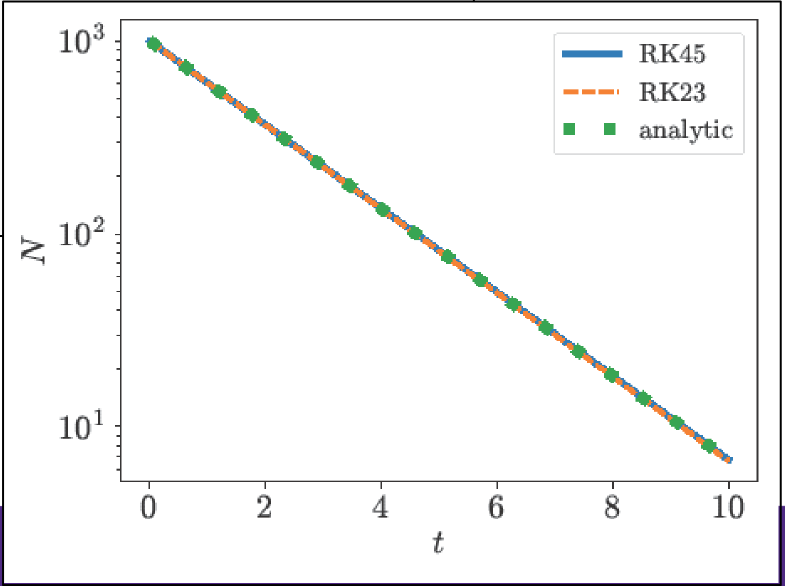
$t_4' = t + \Delta t$

$$x(t+\Delta t) = x(t) + (1/6)[f(x_1',t_1') + 2f(x_2',t_2') + 2f(x_3',t_3') + f(x_4',t_4')]\,\Delta t$$



- Radioactive decay code in python

```
import numpy as np
from scipy.integrate import solve_ivp
def exponential_decay(t, y): return -0.5 * y
sol45 = solve_ivp(exponential_decay, [0, 10], [1e3],method='RK45')
sol23 = solve_ivp(exponential_decay, [0, 10], [1e3],method='RK23')
plt.plot(sol45.t,sol45.y.T,label="RK45",lw=3)
l,=plt.plot(sol23.t,sol23.y.T,"--",label="RK23",lw=2)
l2,=plt.plot(sol23.t,1e3*np.exp(-sol23.t/2),lw=6,label="analytic")
l2.set_dashes([1,3])
plt.yscale("log")
plt.ylabel(r"$N$",fontsize=20)
plt.xlabel(r"$t$",fontsize=20)
plt.legend()
plt.tight_layout()
plt.savefig("decay.pdf")
```



Source: scipy manual

Let’s compare RK23 and RK45 method solutions against the analytic one: which one is more accurate? How can you control the accuracy of the method (see scipy manual)?