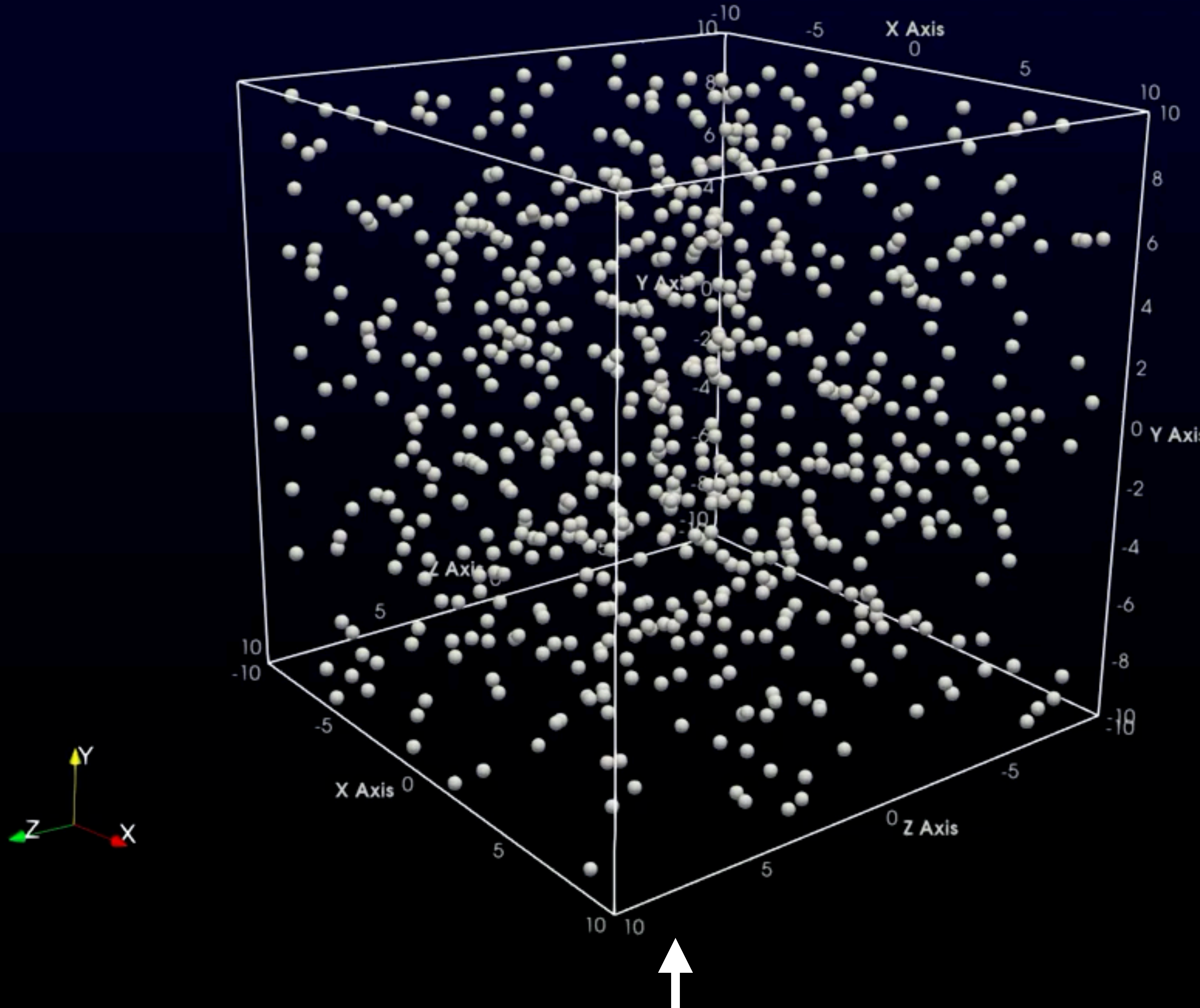


How to reproduce this video?

Why will someone want to reproduce this video?

Go to: *www.menti.com*

Use code: *9005 9647*

The world through a naive eye

This is wrong, but intuitive

```
for i in range(N):
    ax[i] = 0.0
    ay[i] = 0.0
    for j in range(N):
        if (i != j):
            xdiff = ( x[i]-x[j] )
            ydiff = ( y[i]-y[j] )
            r = math.sqrt(xdiff*xdiff + ydiff*ydiff)
            ax[i] += xdiff/(r*r*r)
            ay[i] += ydiff/(r*r*r)

for i in range(N):
    vx[i] = vx[i] + ax[i] * dt
    vy[i] = vy[i] + ay[i] * dt
    x[i] = x[i] + vx[i] * dt
    y[i] = y[i] + vy[i] * dt
```

This is wrong, but intuitive

The world through a naive eye

What we will lack?

Conservation properties are not very good

Why do we need high accuracy?

Do you want to land on the moon?

Or avoid collision with a meteor
while on spaceship?

↓

If yes,

↓

Use LeapFrog

The world through a naive eye

What we will lack?

Conservation properties are not very good

Why do we need high accuracy?

Do you want to land on the moon?

Or, avoid collision with a meteor while driving a spaceship to Mars?

↓

If yes,

↓

Use Leap-Frog / Verlet algorithm

Quiz: Compare accuracy with RK3

Ref: Durran/Moin

Have you heard of Leap-Frog or Velocity-Verlet algorithm?

Go to: *www.menti.com*

Use code: *9005 9647*

Ref!

This is better, but still wrong

```
for i in range(N):
    ux[i] = vx[i] + ax[i] * dt / 2.0
    uy[i] = vy[i] + ay[i] * dt / 2.0
    x[i] = x[i] + ux[i] * dt
    y[i] = y[i] + uy[i] * dt

for i in range(N):
    ax[i] = 0.0
    ay[i] = 0.0
    for j in range(N):
        if (i != j):
            xdiff = ( x[i]-x[j] )
            ydiff = ( y[i]-y[j] )
            r = math.sqrt(xdiff*xdiff + ydiff*ydiff)
            ax[i] += xdiff/(r*r*r)
            ay[i] += ydiff/(r*r*r)

for i in range(N):
    vx[i] = ux[i] + ax[i] * dt / 2.0
    vy[i] = uy[i] + ay[i] * dt / 2.0
```

velocity at half-time

position at full-time

velocity at full-time

This is better, but still wrong

What went bad in the last slide?

Well, we were not too bad.

Just that the particles will keep going away and away.

Why?

The potential is pure Coulomb (repulsive)

Finally, this looks useful!

```
for i in range(N):
    ux[i] = vx[i] + ax[i] * dt / 2.0
    uy[i] = vy[i] + ay[i] * dt / 2.0
    x[i] = x[i] + ux[i] * dt
    y[i] = y[i] + uy[i] * dt
    x[i] = x[i] - (int(x[i]/Lx)) * 2.0 * Lx
    y[i] = y[i] - (int(y[i]/Ly)) * 2.0 * Ly

for i in range(N):
    ax[i] = 0.0
    ay[i] = 0.0
    for j in range(N):
        if (i != j):
            xdiff = ( x[i]-x[j] ) - round((x[i]-x[j])/(2.0*Lx)) * 2.0*Lx
            ydiff = ( y[i]-y[j] ) - round((y[i]-y[j])/(2.0*Ly)) * 2.0*Ly
            r = math.sqrt(xdiff*xdiff + ydiff*ydiff)
            ax[i] += xdiff/(r*r*r)
            ay[i] += ydiff/(r*r*r)

for i in range(N):
    vx[i] = ux[i] + ax[i] * dt / 2.0
    vy[i] = uy[i] + ay[i] * dt / 2.0
```

Periodic Boundary Condition

Are we done?

Almost, but ...

Just put it inside a time loop.

And initialize the particles with some position and velocity.

And, we are good to go!

Taking a long time to run?

We need to **re-write** the code in C/C++/Fortran.

Numba-JIT compiler at rescue!

```
from numba import jit

@jit(nopython=True)

<Your python code here!>
```

Still not fast-'enough'?

```
from numba import jit, prange

@jit(nopython=True, parallel=True)

<Your python code here>
```

Run 'some of the' loops in parallel

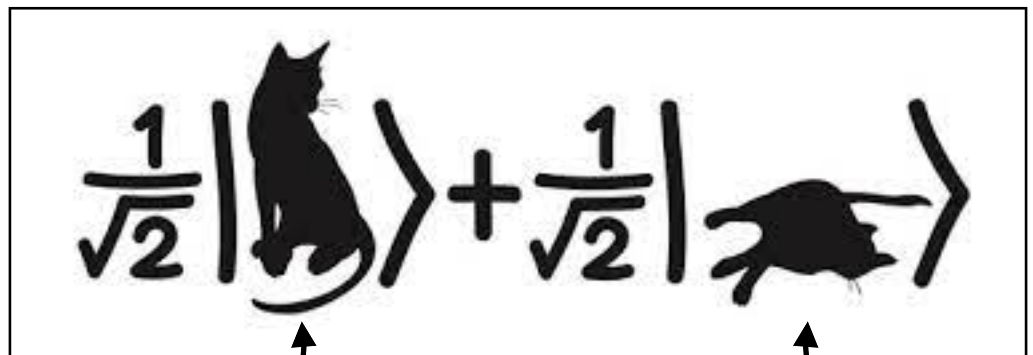
```
for i in prange(N):
    ux[i] = vx[i] + ax[i] * dt / 2.0
    uy[i] = vy[i] + ay[i] * dt / 2.0
    x[i] = x[i] + ux[i] * dt
    y[i] = y[i] + uy[i] * dt
    x[i] = x[i] - (int(x[i]/Lx)) * 2.0 * Lx
    y[i] = y[i] - (int(y[i]/Ly)) * 2.0 * Ly

for i in prange(N):
    ax[i] = 0.0
    ay[i] = 0.0
    for j in range(N):
        if (i != j):
            xdiff = ( x[i]-x[j] ) - round((x[i]-x[j])/(2.0*Lx)) * 2.0*Lx
            ydiff = ( y[i]-y[j] ) - round((y[i]-y[j])/(2.0*Ly)) * 2.0*Ly
            r = math.sqrt(xdiff*xdiff + ydiff*ydiff)
            ax[i] += xdiff/(r*r*r)
            ay[i] += ydiff/(r*r*r)

for i in prange(N):
    vx[i] = ux[i] + ax[i] * dt / 2.0
    vy[i] = uy[i] + ay[i] * dt / 2.0
```

Performance of the code iPPD:

iPPD	11725 second
iPPD (NumbaJIT)	15.88 second
iPPD (NumbaJIT + parallel)	6.87 second

Did you learn something new today?

Or, it was just a repetition of what you already knew?



You were awake

You were sleeping

Please send me your feedback:

Go to: *www.menti.com*

Use code: *7072 6275*