Lec3

- Graphing, multiple displays
- Many particles - Newtonian gravity
- Higher order integrators

Graphing

- So far we have used Python to *animate* a simulation of simple motion.

- For more quantitative work need to be able to *plot* aspects of the motion.

- Eg. for the 1D harmonic oscillator problem graph solution `x(t)`
  Luckily, VPython provides the `gcurve` and `gdisplay` objects to facilitate this

- We can also create more than 1 display screen

Drawing graphs - I

```
# stuff to initialize graphics
from visual import *
from visual.graph import *

scene=display(x=0,y=0,width=400,height=400,
title="simulation")
scene.autoscale=0
scene.range=10.0

pic=gdisplay(x=400,y=0,width=400,height=400,
             title="x vs t",
     xtitle="t",ytitle="x",
     xmax=20.0,xmin=0,
     ymax=8.0,ymin=-8.0)
xplot=gcurve(color=color.blue)
```

Drawing graphs - II

```
# simulation/plotting code
# use def force(pos,vel,t) from before here ....

ball=sphere(radius=0.5,pos=vector(4.0,0.0,0),
track=curve(radius=0.1,display=scene),
mass=1.0,display=scene)
ball.vel=vector(0,0,0)

dt=0.01
t=0

while (t<20.0) :
    rate(100)
    t=t+dt
    ball.pos=ball.pos+ball.vel*dt
    ball.vel=ball.vel+
    (force(ball.pos,ball.vel,t)/ball.mass)*dt
    ball.track.append(pos=ball.pos)
    xplot.plot(pos=(t,ball.pos.x))
```

Time step errors

- The Euler method we have used so far has its limitations – solution accurate to $O(dt)$ only

- See this by computing *energy* $E = 1/2mv^2 + 1/2kx^2$

- Plot as function of time ...

Energy (non)conservation

Just add a line to compute the energy and plot it now instead of `x(t)`

```
energy=0.5*ball.mass*ball.vel.x*ball.vel.x+
0.5*ball.pos.x*ball.pos.x
xplot.plot(pos=(t,energy))
```

- Should see that E is *not* constant.

- $\frac{\Delta E}{E} \sim dt$ (Euler)

- Here, error remains *finite* as $t \to \infty$ - not always so. Often large enough $dt > dt_c$ *discrete* equations *unstable* – `x(t)` blows up ..

- Solution ? Better algorithm than Euler

More accurate integrators

Using Taylor ....

$$x(t + dt) = x(t) + v(t)dt + a(t)dt^2/2 + O(dt^3)$$

leading to

$$x_{n+1} = x_n + v_n dt + a_n dt^2/2$$

Also taking velocity from *symmetric difference*

$$v_{n+1} = \frac{x_{n+2} - x_n}{2dt}$$

Substiting for x_{n+1} using previous equation:

$$v_{n+1} = v_n + \frac{dt}{2}(a_n + a_{n+1})$$

Verlet or leap-frog algorithm
Accurate to $O(dt^2)$

Code needed

```
a1=force(ball.pos,ball.vel,t)/ball.mass
ball.pos=ball.pos+ball.vel*dt+a1*0.5*dt*dt
a2=force(ball.pos,ball.vel,t)/ball.mass
ball.vel=ball.vel+(a1+a2)*dt*0.5
```

See *much* smaller errors in energy. Consistent with $\frac{\Delta E}{E} \sim dt^2$

Many particles

- Consider two masses a and b interacting via some mutual force

- Denote force on a due to b as F_{ab}.

- Likewise force on b due to a as F_{ba}

- By Newton's third law $F_{ab} = -F_{ba}$ *vector* statement

- Given a specific force law can we solve Newton's 2nd law for both particles numerically – *simulate* the system ?

Python `lists` and `for`

Useful to introduce a `list` to store the objects which are interacting

```
system=[balla,ballb]
```

In general lists can comprise arbitrary abstract objects enclosed in square brackets eg.

```
a=[1,2,3]
b=[4,5,6]
```

The statement `c=a+b` concatenates the lists. To process lists we often use the `for` command

```
for i in list:
....
```

Modules

- Useful to package related functions and data into `modules`.

- Typically a module (eg the `visual` module used for graphics) contains extensions to Python to help code some new functionality.

- Simply make a text file with the new commands and save it with the `.py` extension eg. `usefulstuff.py`

- Then to use it in some other piece of code use the command

```
{\tt from usefulstuff import *}
```

Integrator Module I

```
from visual import *
G=1.0
## Force on a due to b
def force(a,b):
    diff=b.pos-a.pos
    return G*b.mass*a.mass*norm(diff)/diff.mag2

## Finds acceleration of a due to all objects b

def totalacc(a,objlist):
    sum_acc=vector(0,0,0)
    for b in objlist:
        if (a!=b):
            sum_acc=sum_acc+force(a,b)/a.mass
    return sum_acc
```

Integrator Module II

```
## Finds total acceleration on all objects

def update_acceleration(objlist):
    for i in objlist:
        i.acc=totalacc(i,objlist)

## updates positions and track of each object

def update_position(objlist, dt):
    for i in objlist:
        i.pos=i.pos+dt*i.velocity
        i.track.append(pos=i.pos)

## update velocity of each object

def update_velocity(objlist, dt):
    for i in objlist:
        i.velocity=i.velocity+dt*i.acc
```

Gravity code

```
from visual import *
from integrator import *

scene.autoscale=0
scene.range=1
balla=sphere(..)
ballb=sphere(..)

# create list of gravitating objects
system=[balla,ballb]

dt=0.01
while True:
    rate(100)
    update_position(system,dt)
    update_acceleration(system)
    update_velocity(system,dt)
```