2 Freefall: adding gravity and drag

2.1 Let's Plan

We now have enough experience to start to plan programs before starting to code. Imagine that we wanted to create a house. If we started by laying bricks without a plan we would most likely make mistakes and have to keep restarting from scratch. A little time planning will save us a great deal of time later.

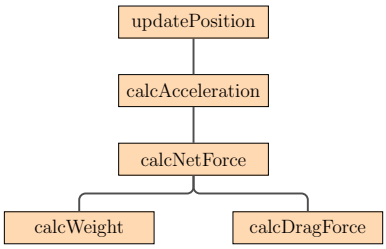
We might take a *top-down* approach. Starting with the outer walls, decide how many rooms, then work *down* to finally designing each room.

Or we could go *bottom-up*. Design each room then 'bolt' the room plans together to form the final design.

Taking the top-down approach we want our program to

- Simulate a mass falling under gravity plotting graphs of velocity, displacement and acceleration.
- The mass will accelerate under gravity.
- We want to allow for drag.

So, we could have the following functions:-



2.2 Gravity?

So far our ball is moving at a constant speed. In a gravitational field it will accelerate. When close to the earth's surface at a constant rate of 9.81 ms^{-1}. How can we add this to our model?

The simplest way is to keep updating the velocity using $v = u + a\delta t$. Every time quantum - δt (trip around the while loop) we calculate the change in velocity $g\delta t$ and add this to the velocity.

Since the velocity is a vector we should define g as a vector.

$$\mathbf{g = vec\ (0,-9.81,0)}$$

Here we are setting the acceleration in the x & z directions to be zero and in the y direction (vertically) to -9.81 i.e. 9.81 ms^{-1} downwards.

So we could add a new function **updateVelocity** - after the **updatePosition**

```
#-----------------------------------------------------
def updateVelocity(x, g):
        x.vel = x.vel + (g * dt)[
        return x
#-----------------------------------------------------
```

However... later we will also be adding drag - which is a force. Thinking ahead maybe we should determine the *Force* of gravity on the mass and calculate the acceleration using $F_{grav} = ma = mg$ and then $a = \frac{F_{grav}}{m}$.

A tricky one this ... we will be repeating an identical calculation very time we loop around - this seems pointless and will slow our simulation. Later though, when we add a drag force we can add this to F_{grav} and calculate the changing acceleration every *iteration* of the loop using $a = \frac{F_{total}}{m}$.

If we divide our code up into short functions this will be easy to do.

2.3 Drag?

Well, we now have a codebase which simulates freefall. It is pretty easy now to extend this with drag. **Stoke's Law** is a simple model of drag which sets the drag force proportional to the velocity.

$$F_{drag} = 6\pi r\eta v$$

Here η is the 'dynamic viscosity' of the fluid, v the fluid velocity over the projectile and r its radius.

Thinking on this, are we trying to explicitly model a specific object with a known radius moving though air of a given viscosity or are we simply trying to model the *form* of the motion?

In the latter case we can simplify the above equation to

$$F_{drag} = kv$$

We can simply create a new function **calcDragForce**

```
#-----------------------------------------------------
def calcDragForce():
        dragForce=vec(0,0,0)
        return dragForce
#-----------------------------------------------------
```

We haven't yet coded calculation of the drag, simply set it as a zero vector. Such as empty function is often called a **stub**. We can write it into our program and fill in the actual instruction statements later.

We could write our entire program initially using stubs - a bit like writing a book by first creating the section headings before fleshing out the actual content.

2.4 The Importance of the Time Quantum

.....add plot here.....

A falling mass accelerates constantly - not in the discrete jumps we code here[1]

The smaller the time quantum the better our model approximates to reality. However, smaller time intervals result in more calculations. This is a common trade-off in computational physics - which is why physicists like really powerful supercomputers.

2.5 Extension Activity – Wind

If we have a horizontal wind, this will add a further drag for pushing the mass sideways. So we could add another function to calculate the horizontal drag due to wind? This new drag force could simply be summed with the gravitational and vertical drag force to give the net 3d force.

However ... Since vython operates with 3d vectors we do need need to separately calculate a horizontal and vertical drag. We simply calculate the drag using $F_{drag} = 6\pi r\eta v$. This will return a force as a 3d vector. The net-force is simply $F_{net} = F_{grav} + F_{drag}$. Then $a = \frac{F_{grav}}{m}$ will return acceleration a as a 3d vector. Rather neat!

I suggest creating a new global variable for the wind speed ***windSpeed=vec(5,0,0).***

2.6 Summary

We are now getting sophisticated. In our programming we appreciate the merits of planning our programs in advance. We can break to program down into sub-programs, such as functions. These can then be 'assembled' using empty*stubs*, before we have to write the algorithms.

We have seen how vpython's inbuilt handling of 3d vectors means that we no longer need to think in simplistic terms of resolving a vector into x,y,z components rather we simply use the 3d vector description.

[1] actually the question of whether time and space are quantised are moot in fundamental physics. However - if there are fundamental lengths of distance and time these are very small indeed.