Introduction to Computational Physics (PHYS105)

Lecture 9: Projectile Motion using Numeric Techniques

Carl Gwilliam
(C.Gwilliam@liverpool.ac.uk)

- **RECORD and attendance code!!**
- Morning everyone
  - **Please remember to register your attendance!!**
- This week we are going to continue looking at numeric techniques and see how to apply what we have learnt to the physics case of projectile motion
  - **As always, please feel free to interrupt with questions at any point**

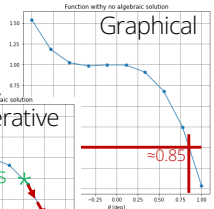
Lecture 8: Recap

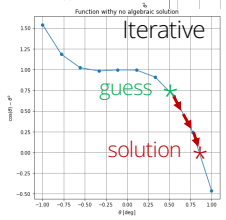
- The vast majority of calculations cannot be performed analytically (i.e. algebraically)
- Instead, we have to solve them using various numeric techniques
  - Discretisation
  - Graphical
  - Iterative

  } Just different ways of approximating result
- Last week we saw how to use these to tackle
  - Differentiation
  - Integration
  - Root finding
- This week we'll look further into numeric techniques
  - Solving the differential equations that describe projectile motion where we have to take into account air resistance

Discretisation

$$\frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$





NumPy SciPy

2

- **Before starting on this week's material let's first recap last week, where we started to look at numeric techniques**
- We saw that the vast …
- Instead we have to solve the numerically **… techniques, which are just types of approximation**
  - **Saw examples of … (point to diagrams)**
  - **… either a real graph or using numpy …**
- And we saw how to apply these to tackle …

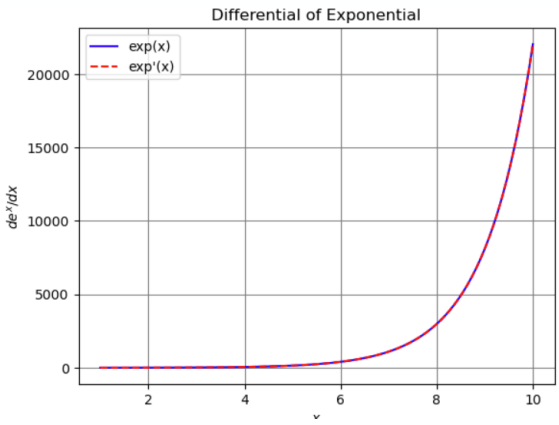
Lecture 8: Formative problem solutions (1)

- Ex 1. Plot e^x and its derivative
  - Using same method as example

$$f' = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

  - Just replacing f (x) = cos(x) by f(x) = e^x

```
# New function
def exp_diff(x, h = 0.01):
    return (np.exp(x + h) - np.exp(x - h)) / (2*h)

# Create arrays
xarray = np.linspace(0, 10, 100)
exp_diff_array = exp_diff(xarray)

# Plot
plt.figure()
plt.title("Differential of Exponential")
plt.xlabel("$x$")
plt.ylabel("$de^x/dx$")
plt.plot(xarray, np.exp(xarray), color = 'b',
         linestyle = '-', label = "exp(x)")
plt.plot(xarray, exp_diff_array, color = 'r',
         linestyle = '--', label = "exp'(x)")
plt.legend()
plt.grid(color = 'grey')
plt.show()
```



- As you should expect they are the same
  - Since d/dx(e^x) is e^x itself

3

- First, as usual, we will look at the solution to last week's formative problems …
- Exercise 1: compute the derivative of e^x using discretisation i.e. splitting into discrete steps and calculating the difference in the function over this step
  - **Now using the symmetric difference between the function at x+h and x-h over 2 * the step (h) as this turns out to be a better approximation**
- Code the equation for the difference up in a python function
  - Simply, replace the function cos(x) by exp(x) – be careful with operator order and put brackets
  - Giving a default value for h
- Create an array of x-values between 0 and 10 in 100 steps using np.linspace and pass to the function using the array-at-a-time ability of np to get corresponding derivative
  - Plot the function and it's derivative -> same

Lecture 8: Formative problem solutions (2)

- Ex 2. How many slices needed to calculate integral of cosθ between –π/2 and π/2 to 2 d.p.
  - Start by putting the code in a function to be able to reuse it many times without having to copy-and-paste
  - Loop over increasing number of slices `while` deviation from 2 is > 0.005 (would round up to 0.01 at 2 d.p.)

```python
def cosine_integral(nslices):
    """
    Calculate the integral of cosine between
    -pi/2 and pi/2 using the trapezium method
    """

    # Create array of angles and cosines for each step
    angles = np.linspace(-np.pi/2, np.pi/2, nslices + 1)
    cosines = np.cos(angles)

    # Loop over the slices to find the total area
    total_area = 0
    for i in range(nslices):
        # Find coordinates of current & next point
        x1 = angles[i]
        x2 = angles[i+1]
        y1 = cosines[i]
        y2 = cosines[i+1]

        # Calculate area of slice and sum
        trapezium_area = 0.5 * (y1+y2) * (x2-x1)
        total_area = total_area + trapezium_area

    return total_area
```

```python
# Start values
nslices, diff = 1, 1.

# Loop while result - 2 > 0.005
while diff > 0.005:
    # Calculate + print result
    cosint = cosine_integral(nslices)
    print (f"Integral for {nslices:02d} slices is {cosint:.2f}")
    # Update # slices and difference
    diff = abs(2.0 - cosint)
    nslices = nslices + 1
```

```
Integral for 01 slices is 0.00
Integral for 02 slices is 1.57
Integral for 03 slices is 1.81
Integral for 04 slices is 1.90
Integral for 05 slices is 1.93
Integral for 06 slices is 1.95
Integral for 07 slices is 1.97
Integral for 08 slices is 1.97
Integral for 09 slices is 1.98
Integral for 10 slices is 1.98
Integral for 11 slices is 1.99
Integral for 12 slices is 1.99
Integral for 13 slices is 1.99
Integral for 14 slices is 1.99
Integral for 15 slices is 1.99
Integral for 16 slices is 1.99
Integral for 17 slices is 1.99
Integral for 18 slices is 1.99
Integral for 19 slices is 2.00
```

- Find we need 19 slices to be accurate to 2 d.p.

4

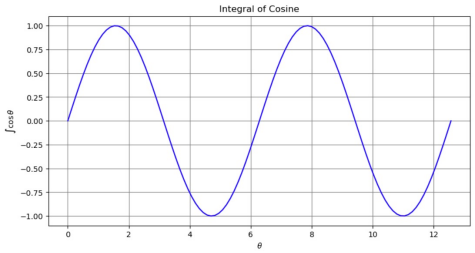
- Exercise 2: Read
- Since we will use the code several times we covert it into a function
  - That takes the number of slices as input, does the same calculation as previously
  - And returns the total area
  - Makes it easier to reason about next part
- Want to find when difference between the integral and 2 (the expected answer) is < 0.005
  - Since this would round up to 0.01 at 2dp
- So, we start with 1 slice and setting our difference to something large
- We then create the while loop that iterates while the diff > 0.005
- Inside the loop we:
  - Call our function with the current number of slices to get the area i.e. integral and print this out
  - Calculate the absolute difference from 2 and update the diff variable
  - Increase the number of slices by 1
- As we add more slices we see that the integral gets close to 2.00
  - And the loop ends when the difference is below 0.005 which we find

is for 19 slices

Lecture 8: Formative problem solutions (3)

- Ex 3. Use SciPy's `quad` function to confirm the integral of cosθ as a function of angle is sinθ
  - Loop over angles, calling `quad` with each value as the upper integral limit and saving the results in array

```
from scipy.integrate import quad

# Create array of input angles
theta_array = np.linspace(0, 4*np.pi, 100)

# Loop over angles
cos_int_array = []
for theta_max in theta_array:
    # Calculate integral using angle as upper limit
    integral, err = quad(np.cos, 0, theta_max)

    # Store each integral
    cos_int_array.append(integral)

# Plot the result vs the angle
plt.figure(figsize = (10, 5))
plt.title("Integral of Cosine")
plt.xlabel("$\\theta$")
plt.ylabel("$\\int\\cos\\theta$")
plt.plot(theta_array, cos_int_array,
         color = 'b', linestyle = '-')
plt.grid(color = 'grey')
plt.show()
```



- Ex 4. Find right-hand solution for Gaussian = 0.1
  - In this case we want the last x value above thresh
  - Which is simply the last value of xAboveThresh
    - Rather than the first value for the left-hand solution

```
xright = xAboveThresh[-1]
print(f"Right hand solution = {xright:.2f}")
```

```
Right hand solution = 8.34
```

5

- **Exercise 3: you saw in the notebook that scipy.integrate has a quad function to calculate the integral and its error**
  - **And we used this to calculate the integral of cos from –pi/2 to pi/2 showing it is 2 as we just saw from our own numerical method**
- You were then asked to do this for theta values from 0 to 4pi to show it is a sine wave
  - First create an array of angles over this range using np.linspace with say 100 steps
- Then loop over the the thetas in the array and use this as the max limit for our integral
  - Each iteration we integrate from 0 to this theta value
- We then append the result to a list that we define outside the array
- If we then plot the result we see the expected sine wave

- Exercise 4: was finding the RH solution for our Gaussian = 0.1
- **We had got an array of the x points where y was above the threshold**
  - **We use np's array-at-a-time functionality to ask that y is above our 0.1 threshold → gives an array of True/False**
  - **We then pass this to the x array and it only gives us the values where the Boolean is True i.e. above the threshold -> this is called Boolean indexing**
- Then for the LH solution we took the first value in this array i.e. first x value over threshold
  - So, for the RH solution we clearly just take the last value in the array i.e. the last value over threshold
  - **Any questions on the formative problems?**
- **The marks and feedback for the week 8 summative problems have been released on canvas, with an average mark of 73%**

Projectile motion

- We all know how to work out the path of a projectile from Newtons' second law F = ma
- Split into horizontal and vertical components to get the familiar equations of motion
  - Horizontal force is zero

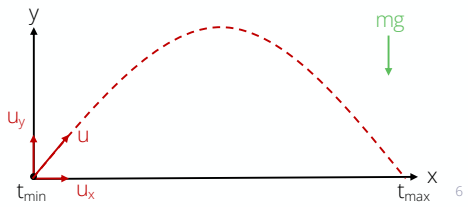
$$F_x = ma_x = m\frac{d^2x}{dt^2} = 0$$
$$\Rightarrow \frac{dx}{dt} = u_x$$
$$\Rightarrow x = u_x t + x_0$$
$$= u_x t \text{ for } x_0 = 0$$

  - Vertical force is mg

$$F_y = ma_y = m\frac{d^2y}{dt^2} = mg$$
$$\Rightarrow \frac{dy}{dt} = gt + u_y$$
$$\Rightarrow y = \frac{1}{2}gt^2 + u_y t + y_0$$
$$= \frac{1}{2}gt^2 + u_y t \text{ for } y_0 = 0$$

- Find the time at which hits the ground from y = 0

$$y = \frac{1}{2}gt^2 + u_y t = t\left(\frac{gt}{2} + u_y\right) = 0 \quad \begin{cases} t_{min} = 0 \\ t_{max} = -\dfrac{2u_y}{g} \end{cases}$$



6

- **Now, lets use the numerical methods we started to investigate last time to solve a physics problem, namely projectile motion**
  - **Which is drawn schematically here (point)**
- **First we need to set up the problem in maths before we can solve it with python**
- **We should all know how to work out the path of a projectile by using N2L: F = ma to get the familiar equations of motion**
- First, we split N2L into horizontal and vertical components
- Since there is no force acting in the horiz direction we have the x comp of the force, which depends on the x comp of the acceleration, is just 0
  - We can then write the acceleration as the second derivative of position wrt to time
  - If we integrate we get that the x component of the velocity, dx/dt, is just equal to the integration constant, which is the initial velocity ux
  - We can then integrate this again to find the position, getting another integration constant which this time represents the initial position, x0
  - **So we end up with our familiar equation of motion x = the initial x-vel(ux) * time(t) + the initial x position (x0), which we can neglect**

since we can always setup our coord system such that the projectile starts from zero (as here: point)

- We can then do the same in the vertical (i.e. y) direction except that this time we have the force of gravity given by mg
    - We get another of the equations of motion y = ½ g * time**2 + the initial y-vel (uy) * time + the initial y position (y0), which again we can always choose to be 0
- We can then use this second equation to find the time at which the projectile hits the ground by simply setting the y displacement to 0
    - If we factorise we will obv get two solutions since it is quadratic in time
    - But the first is just the trivial start point, where t=0 (point)
    - The second is where it eventually comes back to earth (point), given by -2uy/g (remember g is also negative so the result will be positive despite the minus sign)

Projectile motion (2)

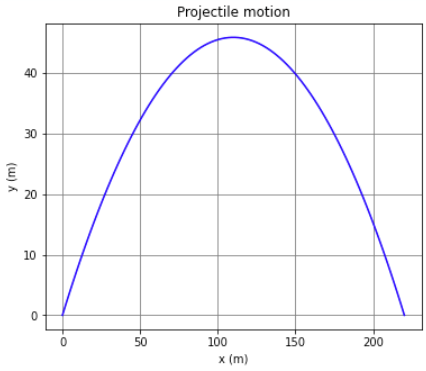
- Hence we have an analytic solution that we can plot to see the projectiles path
  - As an example, let's consider throwing a baseball from the origin i.e. $x_0 = y_0 = 0$.
  - The max speed a baseball has been thrown at is 170 km/h ≈ 47 m/s → u_x = 36 m/s and u_y = 30 m/s

```
import numpy as np
import matplotlib.pyplot as plt

ux = 36    # Initial x velocity (m/s)
uy = 30    # Initial y velocity (m/s)
g = -9.81  # Force of gravity (m/s**2)

# Create a number of time steps up to the maximum
tMax = -2*uy/g # s
nSteps = 100
tArr = np.linspace(0.0, tMax, nSteps)

# x position  = u_x*t + x_0 (with x_0 = 0)
xArr = ux*tArr

# y position = 0.5gt^2 + u_y*t + y_0 (with y_0 = 0)
yArr = 0.5*g*tArr**2 + uy*tArr

# Plot
plt.figure(figsize = (6, 5))
plt.title("Projectile motion")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '-', color = 'b')
plt.grid(color = 'grey')
plt.show()
```

$t_{max} = -\frac{2u_y}{g}$

$x = u_x t + x_0$

$y = \frac{1}{2}gt^2 + u_y t + y_0$



- Let's compare this to what we get in reality

- This gives us an analytic solution for the position that we can plot to see the path …
- As a concrete example, let's consider the case …
  - Without air resistance the max distance occurs with an angle of 45deg, but with air resistance it is slightly below this so we split it up into components …
- So, in python …
  - **We define the initial speeds ux and uy and the acceleration due to gravity (remembering to comment on the units)**
  - **We find the max time following the equation we just derived and then use linspace() to create an array of 100 time steps up to that point**
  - **We then use our formulae for the x and y position to get the corresponding x and y arrays using numpy's array-at-a-time abilities (remembering we set x0 = y0 = 0)**
- If we plot the result we get the expected parabolic flight path
- **So, if we have got the result analytically, where does the numerical part come in?**
  - Well, lets compare this to what we would get in reality

Projectile motion (2)

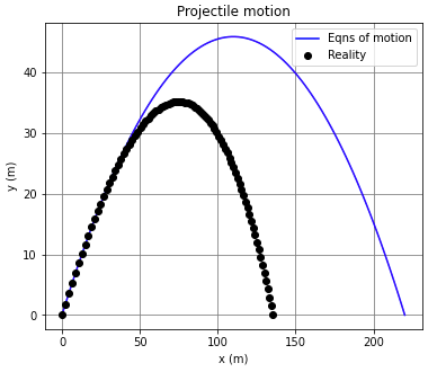
- Hence we have an analytic solution that we can plot to see the projectiles path
  - As an example, let's consider throwing a baseball from the origin i.e. $x_0 = y_0 = 0$.
  - The max speed a baseball has been thrown at is 170 km/h ≈ 47 m/s → u_x = 36 m/s and u_y = 30 m/s

```
import numpy as np
import matplotlib.pyplot as plt

ux = 36    # Initial x velocity (m/s)
uy = 30    # Initial y velocity (m/s)
g = -9.81  # Force of gravity (m/s**2)

# Create a number of time steps up to the maximum
tMax = -2*uy/g # s
nSteps = 100
tArr = np.linspace(0.0, tMax, nSteps)

# x position  = u_x*t + x_0 (with x_0 = 0)
xArr = ux*tArr

# y position = 0.5gt^2 + u_y*t + y_0 (with y_0 = 0)
yArr = 0.5*g*tArr**2 + uy*tArr

# Plot
plt.figure(figsize = (6, 5))
plt.title("Projectile motion")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '-', color = 'b')
plt.grid(color = 'grey')
plt.show()
```

$$t_{max} = -\frac{2u_y}{g}$$

$$x = u_x t + x_0$$

$$y = \frac{1}{2}gt^2 + u_y t + y_0$$



- Let's compare this to what we get in reality
  - Max baseball throw distance ≈ 136 m
  - Clearly we are missing something …

8

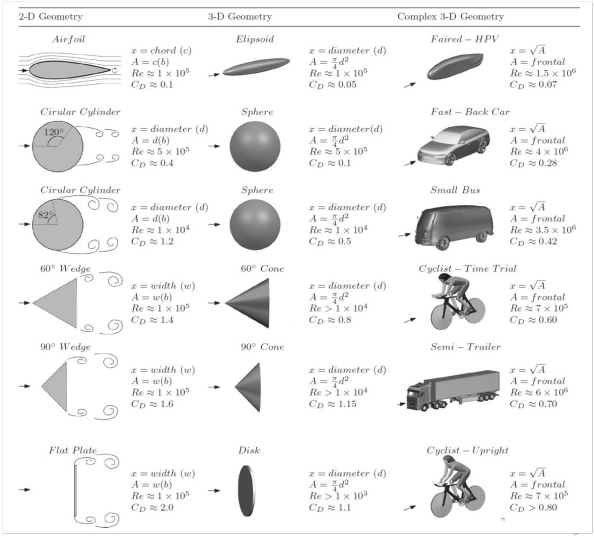
- The max distance anyone has thrown a baseball is about 136 m
  - This is way less than the ≈220 m we predicted
  - Clearly we are missing something

Air resistance

- To describe real-world projectile motion, we need to include the effect of air resistance
- This gives rise to an additional drag force
  - Acts in opposite direction to velocity, v
  - Magnitude is proportional to v^2
- Drag force D is given by:

$$D = \frac{1}{2} C_D \rho_{\text{air}} A v^2$$

  - A is the cross-sectional area
  - ρ_{air} is the air density ≈ 1.2 kg/m^3
  - C_D is the drag coefficient
- Drag coefficient depends on projectile shape
  - Narrower and more streamlined → smaller C_D



- **And that of course that something is the air resistance that is present in the real world!**
- **We often find problems like this in physics where we can solve the idealised case analytically but once we add reality things get messier and we can't get an algebraic result, so instead require numerical solutions**
- Air resistance gives rise to an … force, which …
- The exact form of the drag force, which we will call D, is given by … (read) … where:
  - A: … of the projectile
  - rho: … which is about 1.2 kg/m^3 at sea level
  - and Cd is the so-called …
- This drag coefficient depends on the projectile shape and characterises how narrow and streamlined it is. The narrower and …
- **Here is a table of some example drag coefficients**
  - **Split into columns for the 2D case, then the 3D case, and then some more complex object that presents a somewhat similar shape**
  - **We can see that it starts at < 0.1 for something very streamlined (point top row)**
  - **And goes up to between 1 and 2 for something that is not (point bottom 2 rows)**
- **If we are considering a simple ball as our projectile we get something like 0.5 (point to second sphere)**

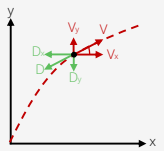
Air resistance (2)

- To describe real-world projectile motion, we need to include the effect of air resistance
- This gives rise to an additional drag force
  - Acts in opposite direction to velocity, v
  - Magnitude is proportional to v^2
- Drag force D is given by:

$$D = \frac{1}{2} C_D \rho_{\text{air}} A v^2$$

  - A is the cross-sectional area
  - ρ_{air} is the air density ≈ 1.2 kg/m^3
  - C_D is the drag coefficient
- Drag coefficient depends on projectile shape
  - Narrower and more streamlined → smaller C_D
- Can write a python function to determine D
  - Splitting into x and y components

```
[36]: def drag(cd, A, rho, velx, vely):
          '''
          Return horizontal and vertical drag force on body given
          drag coefficient (cd), area (A), air density (rho) and
          current horizontal and vertical velocity.
          '''

          # Combine comonponents to get v^2
          v2 = velx**2 + vely**2

          # Calculate angle of v to x and y
          sinTheta = vely/np.sqrt(v2)
          cosTheta = velx/np.sqrt(v2)

          # Find components of D in x and y
          Dx = -0.5*cd*rho*A*v2*cosTheta
          Dy = -0.5*cd*rho*A*v2*sinTheta

          return Dx, Dy
```



10

- So we can write a python function to determine the drag force, D, which I have imaginatively called "drag"
- **The function of course takes the coeff, area and density that D depends on and the velocity at that point as arguments**
  - **Since, as we saw, we split our problem into horiz and vertical motion, we will split the input velocity into two components as well**
- **Inside, we first calculate the velocity squared from these, since this is what D depends on**
- **We then find the angle of each component to the overall velocity (point) using simple trig**
- **We can then write equations for the horiz and vert components of D using our equation and this angle and return them**
- **Any QUESTIONS on the drag force?**

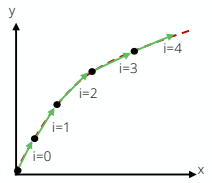
Projectile motion with air resistance

- Let's go back to our horizontal and vertical force formulae and add in the drag force
  - Horizontal force is no longer zero
    - Rather horiz. component of D
  - Vertical force now has two comps:
    - Gravity and vert. comp of D

$$F_x = m\frac{d^2x}{dt^2} = -D_x = -D\cos\theta$$
$$\Rightarrow \frac{d^2x}{dt^2} = -\frac{D}{m}\cos\theta$$

$$F_y = m\frac{d^2y}{dt^2} = -D_y + mg = -D\sin\theta + mg$$
$$\Rightarrow \frac{d^2y}{dt^2} = -\frac{D}{m}\sin\theta + g.$$

- Note that we can no longer write simple analytic expressions for v_x (or x) and v_y (or y)
  - Because the angle is itself a function of time θ = θ(t) → no algebraic form for integral
- Hence, we need to resort to numerical methods to solve
  - The method we will use is discretisation
- Break down the trajectory into small steps i
  - If steps are short enough the angle is constant across the step



11

- So we can calculate the drag but lets go back and see what adding this into our N2L equations in the horiz and vert directions does …
- In the horiz direction the force is no longer zero but minus (as it points against the direction of motion) the x comp of the drag: -Dcos(theta) (point)
- In the vertical direction it is now the gravitational force (mg) we had before minus the y component of the drag, Dsin(theta) (point)
- **So we have again two expressions for the acceleration**
  - **The diff. is that we can no longer just integrate them to get analytic expressions for the velocity or position components**
  - **This is because the angle that the equations now depend on will change with time i.e. theta is a function of time too**
  - **And there is no analytic solution for such an integral**
- Hence … solve the problem, and the method we will use is …
  - By which I mean that we …
  - **If the steps … across the step and hence we can deal with these by turning the derivative into a change in position over change in time for that small step**
    - **Like we did in our numerical differential example last lecture**

Projectile motion with air resistance (2)

- Considering the i^{th} step we can therefore write the change in velocity over the step as

$$\frac{dv_{x\,i}}{dt} = \frac{d^2x_i}{dt^2} = -\frac{D_i}{m}\cos\theta_i \quad \Rightarrow \delta v_{x\,i} = -\frac{D_i}{m}\cos\theta_i\,\delta t.$$

$$\frac{dv_{y\,i}}{dt} = \frac{d^2y_i}{dt^2} = -\frac{D_i}{m}\sin\theta_i + g \quad \Rightarrow \delta v_{y\,i} = -\frac{D_i}{m}\sin\theta_i\,\delta t + g\,\delta t.$$

  - where dt → δt due to the discretisation

- The velocity components at the end of the step are given by adding δv_i to the start value

$$v_{x\,i+1} = v_{x\,i} + \delta v_{x\,i} \qquad v_{y\,i+1} = v_{y\,i} + \delta v_{y\,i}$$

- And the position at the end of the step is given by the initial position + the velocity x time $v_i t$

$$x_{i+1} = x_i + v_{x\,i}\delta t \qquad y_{i+1} = y_i + v_{y\,i}\delta t$$

- Given these formulae, for each step we can simply iterate over the steps from the start

- This method is known as Euler's method and can be used to solve many differential equations
  - Let's use it to plot the trajectory of our projectile with air resistance …

12

- **So lets take our equations for the acceleration and now apply them to a given step i (where all I have done is add a subscript i)**
  - **Writing the acceleration as dv/dt in both cases, we can approximate the derivative as change in V over t during the course of the step in question**
  - **Rearranging, this gives us expressions for the change in velocity, in both x and y, over the step**
- The velocity at the end of the step (i+1) is then of course simply the velocity at the start of the step (i), **which for the first step we know from the initial values**, plus this change
- **And we can get the corresponding position from the position at the start, which we chose as 0 for the first step, plus the change in the position that is simply given by the velocity at the start of the step times the time span of the step**
- Given these … start to get the projectile's trajectory
- Read last 2 bullets
- **Any questions on the mathematics of this before we code it in python?**

Projectile motion with air resistance (3)

```
# Set the properties
mProj = 0.140 # kg
rad = 0.07/2 # m
Area = np.pi*rad**2 # m**2
rhoAir = 1.2 # kg/m**3
CD = 0.3

# Number of steps and size of each in time
nEuler = 10000
dt = tMax/nEuler

# Set the initial positions and velocities
xE, yE = 0.0, 0.0 # m
vX, vY = ux,uy # m/s

# Empty arrays to store the x and y positions for each step
xEuler = np.zeros(nEuler)
yEuler = np.zeros(nEuler)
```

Set up the inputs to the problem

$$t_{max} = -\frac{2u_y}{g}$$

13

- The first thing we need to do is to set up our inputs for the problem
  - For the mass and radius we take those of our baseball, which are 140g and 3.5 cm
  - The drag coeff of a baseball has been measured as around 0.3 (although with quite a large error)
  - We then assume we are at sea level, so set the density to 1.2 (if we at a higher altitude we would just change this value)
- **Let's use 10000 steps for our iteration**
  - **The length of each step in time (dt) we then set to be tmax that we defined in the simple example without air resistance divided by this number of steps**
- **We set the initial positions for the start of the first step to be 0 in each direction**
  - **And the initial velocities to be the same as the simple example**
- **We then setup arrays of the correct length, initially filled with zeros, to store the positions in x and y for each step**

Projectile motion with air resistance (3)

```
# Set the properties
mProj = 0.140 # kg
rad = 0.07/2 # m
Area = np.pi*rad**2 # m**2
rhoAir = 1.2 # kg/m**3
CD = 0.3

# Number of steps and size of each in time
nEuler = 10000
dt = tMax/nEuler

# Set the initial positions and velocities
xE, yE = 0.0, 0.0 # m
vX, vY = ux,uy # m/s

# Empty arrays to store the x and y positions for each step
xEuler = np.zeros(nEuler)
yEuler = np.zeros(nEuler)
```

Set up the inputs to the problem

$$t_{max} = -\frac{2u_y}{g}$$

```
# Loop over the steps until nEuler or until the projectile hits the ground (yE = 0)
iStep = 0
while (yE > 0 or iStep == 0) and iStep < nEuler:
    # Set the positions before current step
    xEuler[iStep] = xE
    yEuler[iStep] = yE

    # Calculate the positions after current step (using v dt)
    xE = xE + vX*dt
    yE = yE + vY*dt

    # Calculate the drag given the velocities
    dragX, dragY = drag(CD, Area, rhoAir, vX, vY)

    # Calculate the new velocities from the old ones,
    # taking into account the drag
    vX = vX + dragX/mProj*dt
    vY = vY + dragY/mProj*dt + g*dt

    iStep = iStep + 1
```

$$x_{i+1} = x_i + v_{x\,i}\delta t \qquad y_{i+1} = y_i + v_{y\,i}\delta t$$

$$\delta v_{x\,i} = -\frac{D_i}{m}\cos\theta_i\,\delta t, \quad \delta v_{y\,i} = -\frac{D_i}{m}\sin\theta_i\,\delta t + g\,\delta t$$

Loop over steps and update position

- Just 10 lines of python!
- Since we are likely to reuse this it is a prime candidate for a function (see notebook)

14

- We then need to loop over the steps and implement our Euler equations we derived
- The first thing we do is setup a while loop to loop while our step number, which starts at zero, is less than the number of steps we defined
    - We also add an extra condition that the vertical position must be > 0 after the first step since there is no point in continuing calculating if it has already hit the ground before the number of steps we are doing
    - Remember that tmax was from the non-air-resistance case and hence the time for the case with air resistance will, of course, be shorter
- Within the loop we first store the current position before the step at that iteration (which will be 0 for the first step / iteration of course) at the correct point in the array, using istep as the index
- We then calculate the new position at the end of the step using the formulae we derived
- We then need to calculate our drag, which we can just do by passing the current velocity components into the function we defined earlier

- We must then update the velocity using our formulae with the correct drag component
- Finally, we must remember to increment the step to avoid an infinite loop
- So, the loop will simply start with the initial position and velocity and then iterate over the requested number of steps
    - Updating the values taking into account drag and assuming the angle is constant over the small step
- Not including comments, this is just 10 lines of code to implement Euler's method for a real physics case!
    - And since we will reuse this …

Projectile motion with air resistance (3)

```
# Set the properties
mProj = 0.140 # kg
rad = 0.07/2 # m
Area = np.pi*rad**2 # m**2
rhoAir = 1.2 # kg/m**3
CD = 0.3

# Number of steps and size of each in time
nEuler = 10000
dt = tMax/nEuler

# Set the initial positions and velocities
xE, yE = 0.0, 0.0 # m
vX, vY = ux,uy # m/s

# Empty arrays to store the x and y positions for each step
xEuler = np.zeros(nEuler)
yEuler = np.zeros(nEuler)
```

Set up the inputs to the problem

$$t_{max} = -\frac{2u_y}{g}$$

```
# Loop over the steps until nEuler or until the projectile hits the ground (yE = 0)
iStep = 0
while (yE > 0 or iStep == 0) and iStep < nEuler:
    # Set the positions before current step
    xEuler[iStep] = xE
    yEuler[iStep] = yE

    # Calculate the positions after current step (using v dt)
    xE = xE + vX*dt
    yE = yE + vY*dt

    # Calculate the drag given the velocities
    dragX, dragY = drag(CD, Area, rhoAir, vX, vY)

    # Calculate the new velocities from the old ones,
    # taking into account the drag
    vX = vX + dragX/mProj*dt
    vY = vY + dragY/mProj*dt + g*dt

    iStep = iStep + 1
```

$$x_{i+1} = x_i + v_{x\,i}\delta t \qquad y_{i+1} = y_i + v_{y\,i}\delta t$$

$$\delta v_{x\,i} = -\frac{D_i}{m}\cos\theta_i\,\delta t, \quad \delta v_{y\,i} = -\frac{D_i}{m}\sin\theta_i\,\delta t + g\,\delta t$$

Loop over steps and update position

```
# Plot the resulting arrays up to last step before hits ground
# Compare result to the trajectory above w/o air resistance
plt.figure(figsize = (6, 5))
plt.title("Projectile motion")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'No air resist')
plt.plot(xEuler[0:iStep], yEuler[0:iStep], linestyle = '-',
         color = 'r', label = 'With air resist')
plt.grid(color = 'grey')
plt.legend()
plt.show()
```

Plot the result, compared to no air resistance

- Just 10 lines of python!
- Since we are likely to reuse this it is a prime candidate for a function (see notebook)

15

- **We can then take the resulting x and y arrays for the steps after the loop and plot them as normal using plot()**
  - **The only thing we do is make sure we stop at the last step of the loop, if it less than the requested number of steps, to stop drawing once it has hit the ground**
- We also plot the analytic formula for the case of no air resistance on top for comparison

Projectile motion with air resistance (3)

```
# Set the properties
mProj = 0.140 # kg
rad = 0.07/2 # m
Area = np.pi*rad**2 # m**2
rhoAir = 1.2 # kg/m**3
CD = 0.3

# Number of steps and size of each in time
nEuler = 10000
dt = tMax/nEuler

# Set the initial positions and velocities
xE, yE = 0.0, 0.0 # m
vX, vY = ux,uy # m/s

# Empty arrays to store the x and y positions for each step
xEuler = np.zeros(nEuler)
yEuler = np.zeros(nEuler)
```

Set up the inputs to the problem

$$t_{max} = -\frac{2u_y}{g}$$

```
# Loop over the steps until nEuler or until the projectile hits the ground (yE = 0)
iStep = 0
while (yE > 0 or iStep == 0) and iStep < nEuler:
    # Set the positions before current step
    xEuler[iStep] = xE
    yEuler[iStep] = yE

    # Calculate the positions after current step (using v dt)
    xE = xE + vX*dt
    yE = yE + vY*dt

    # Calculate the drag given the velocities
    dragX, dragY = drag(CD, Area, rhoAir, vX, vY)

    # Calculate the new velocities from the old ones,
    # taking into account the drag
    vX = vX + dragX/mProj*dt
    vY = vY + dragY/mProj*dt + g*dt

    iStep = iStep + 1
```

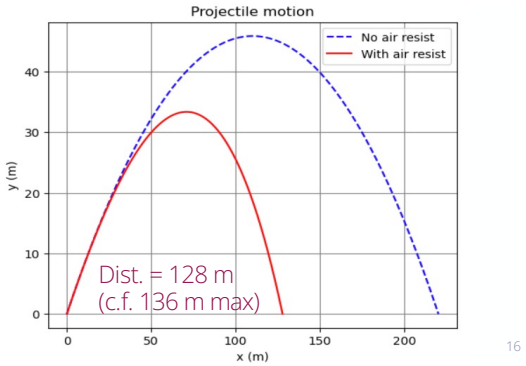
$$x_{i+1} = x_i + v_{x\,i}\delta t \qquad y_{i+1} = y_i + v_{y\,i}\delta t$$

$$\delta v_{x\,i} = -\frac{D_i}{m}\cos\theta_i\,\delta t, \quad \delta v_{y\,i} = -\frac{D_i}{m}\sin\theta_i\,\delta t + g\,\delta t$$

Loop over steps and update position

```
# Plot the resulting arrays up to last step before hits ground
# Compare result to the trajectory above w/o air resistance
plt.figure(figsize = (6, 5))
plt.title("Projectile motion")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'No air resist')
plt.plot(xEuler[0:iStep], yEuler[0:iStep], linestyle = '-',
         color = 'r', label = 'With air resist')
plt.grid(color = 'grey')
plt.legend()
plt.show()
```

Plot the result, compared to no air resistance

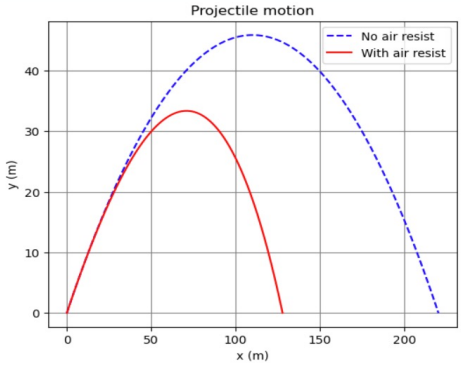


16

- The result looks much more like the reality!
  - We predict 128 m now (rather than 220 m) which is within ≈ 10% of the real value of 136 m
  - Not bad given we have not taken into account things like spin etc
- ANY QUESTIONS on this?

Summary

- This week we have looked at a real-life physics problem
  - That cannot be solved analytically
- Saw how to solve this numerically using Euler's method
  - A fancy name for simply splitting it down into small steps
- Coded this using the python we have learnt in PHYS105
  - With help from NumPy arrays
  - And array-at-a-time calculations
- Brought together the various concepts we have covered
  - Data structures, plotting, functions, conditionals, loops, etc
- You will investigate this further in this week's notebook



Your coding is already enough to solve real-world problems!

- Next week we will look at another numerical approach: so-called Monte Carlo techniques

17

- Read mostly
- **Coded this up … calculations**
  - **So your coding is already enough to solve real-world physics problems.**
- … loops etc, allowing you to practice what you have learnt
- You will investigate this problem …
- **Any final questions?**