Introduction to Computational Physics (PHYS105)

Lecture 9: Projectile Motion using Numeric Techniques

Carl Gwilliam

(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

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

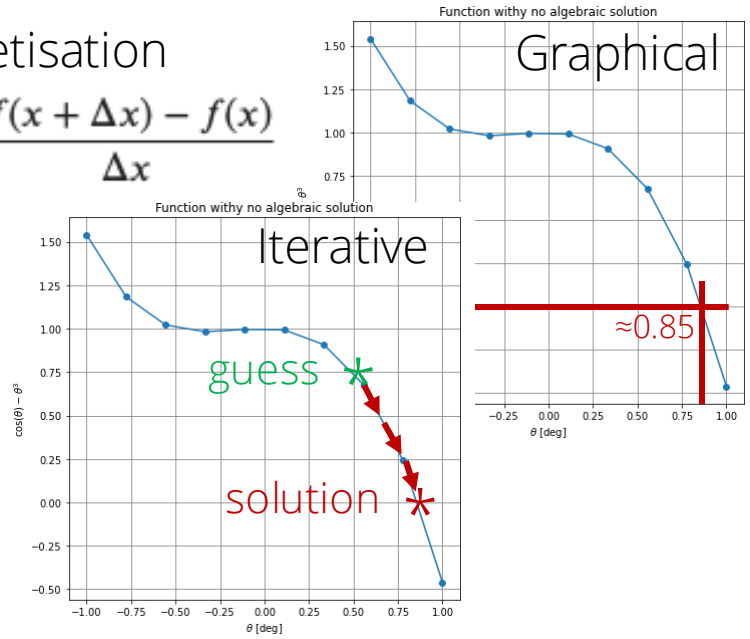
  } Just different ways of approximating result

- 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

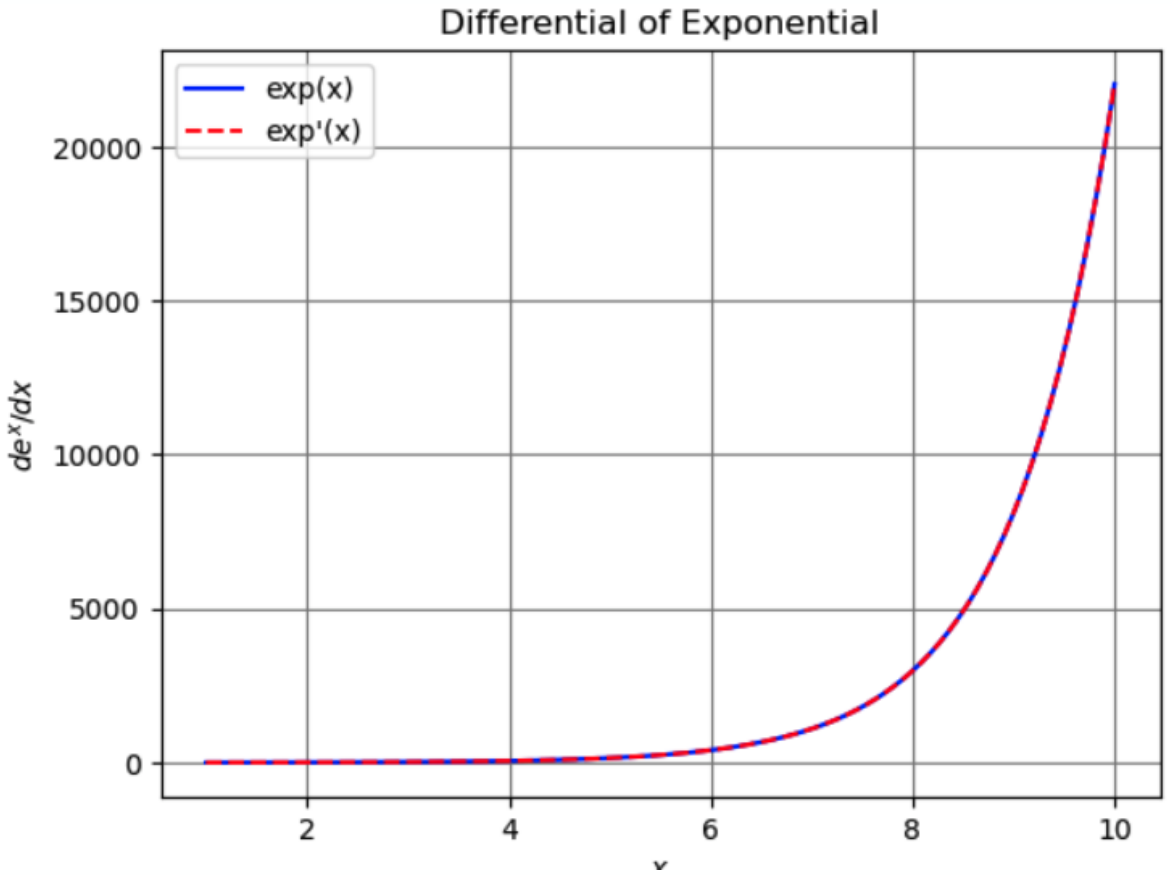
Lecture 8: Formative problem solutions (1)

- 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

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.)

```
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
```

```
# 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
```

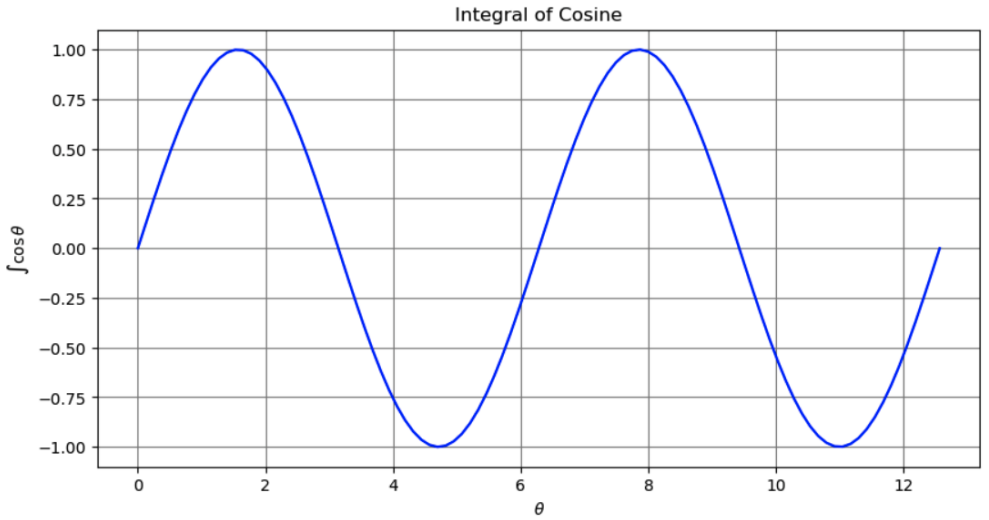
```
Integral for 01 slices is 0.00
Integral for 02 slices is 1.57
Integral for 03 slices is 1.81
Integral for 04 slices is 1.90
Integral for 05 slices is 1.93
Integral for 06 slices is 1.95
Integral for 07 slices is 1.97
Integral for 08 slices is 1.97
Integral for 09 slices is 1.98
Integral for 10 slices 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.

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

```python
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

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

```
Right hand solution = 8.34
```

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

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

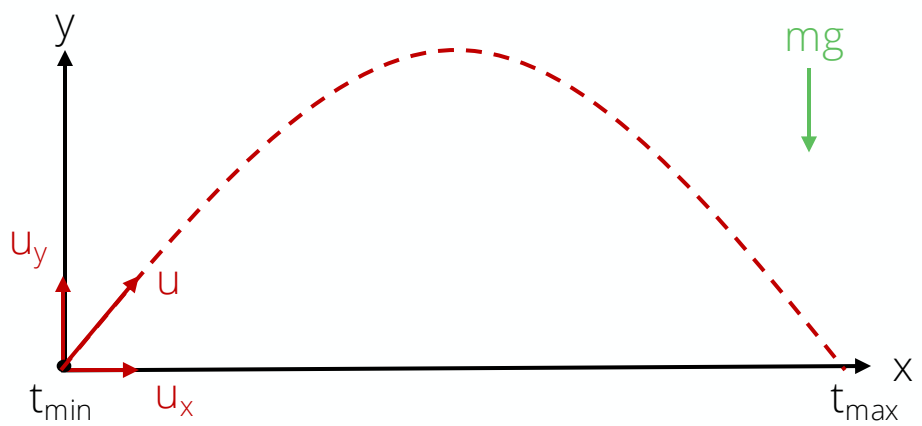
  - Vertical force is mg

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

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

$$y = \frac{1}{2}gt^2 + u_y t = t\left(\frac{gt}{2} + u_y\right) = 0$$

$$\begin{cases} t_{min} = 0 \\ t_{max} = -\dfrac{2u_y}{g} \end{cases}$$



Projectile motion (2)

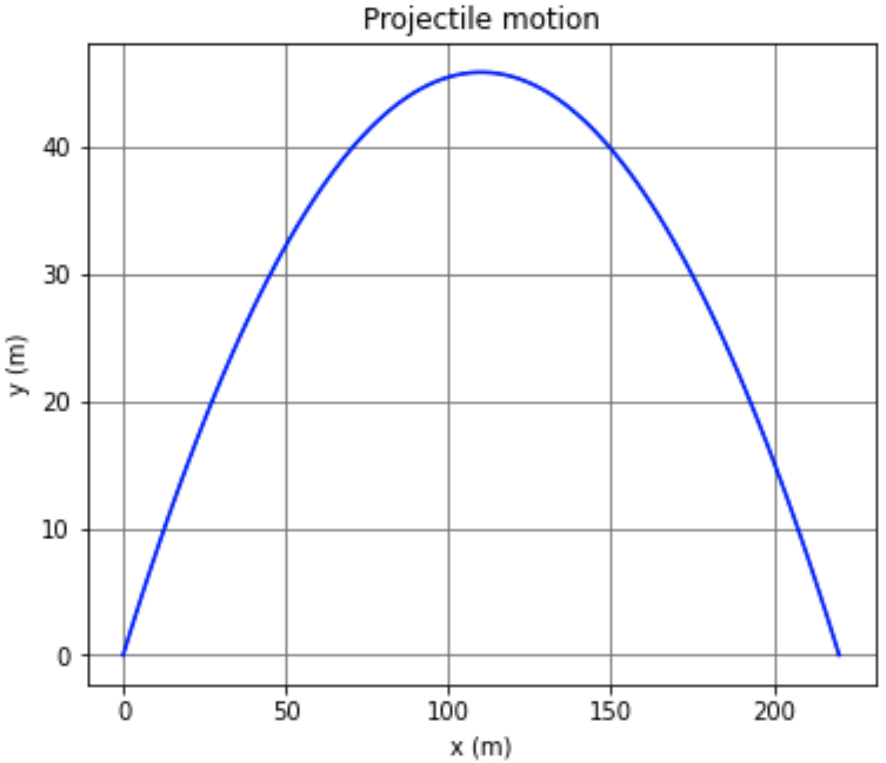
- Hence we have an analytic solution that we can plot to see the projectiles path
  - As an example, let's consider throwing a baseball from the origin i.e. $x_0 = y_0 = 0$.
  - The max speed a baseball has been thrown at is 170 km/h ≈ 47 m/s → u_x = 36 m/s and u_y = 30 m/s

```python
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

Projectile motion (2)

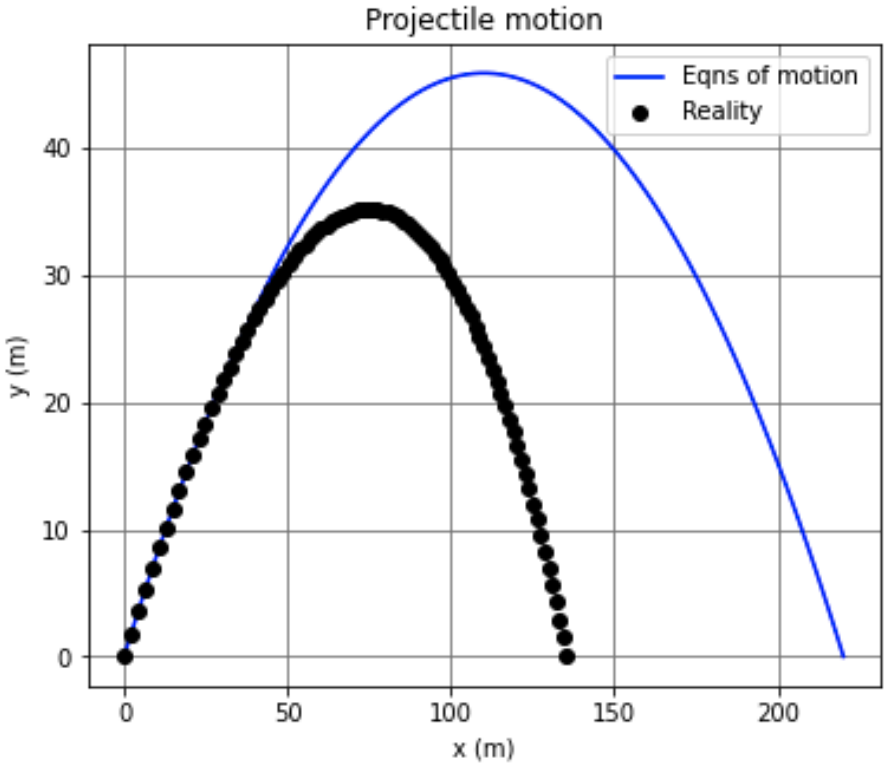
- Hence we have an analytic solution that we can plot to see the projectiles path
  - As an example, let's consider throwing a baseball from the origin i.e. $x_0 = y_0 = 0$.
  - The max speed a baseball has been thrown at is 170 km/h ≈ 47 m/s → u_x = 36 m/s and u_y = 30 m/s

```python
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 …

Air resistance

- To describe real-world projectile motion, we need to include the effect of air resistance

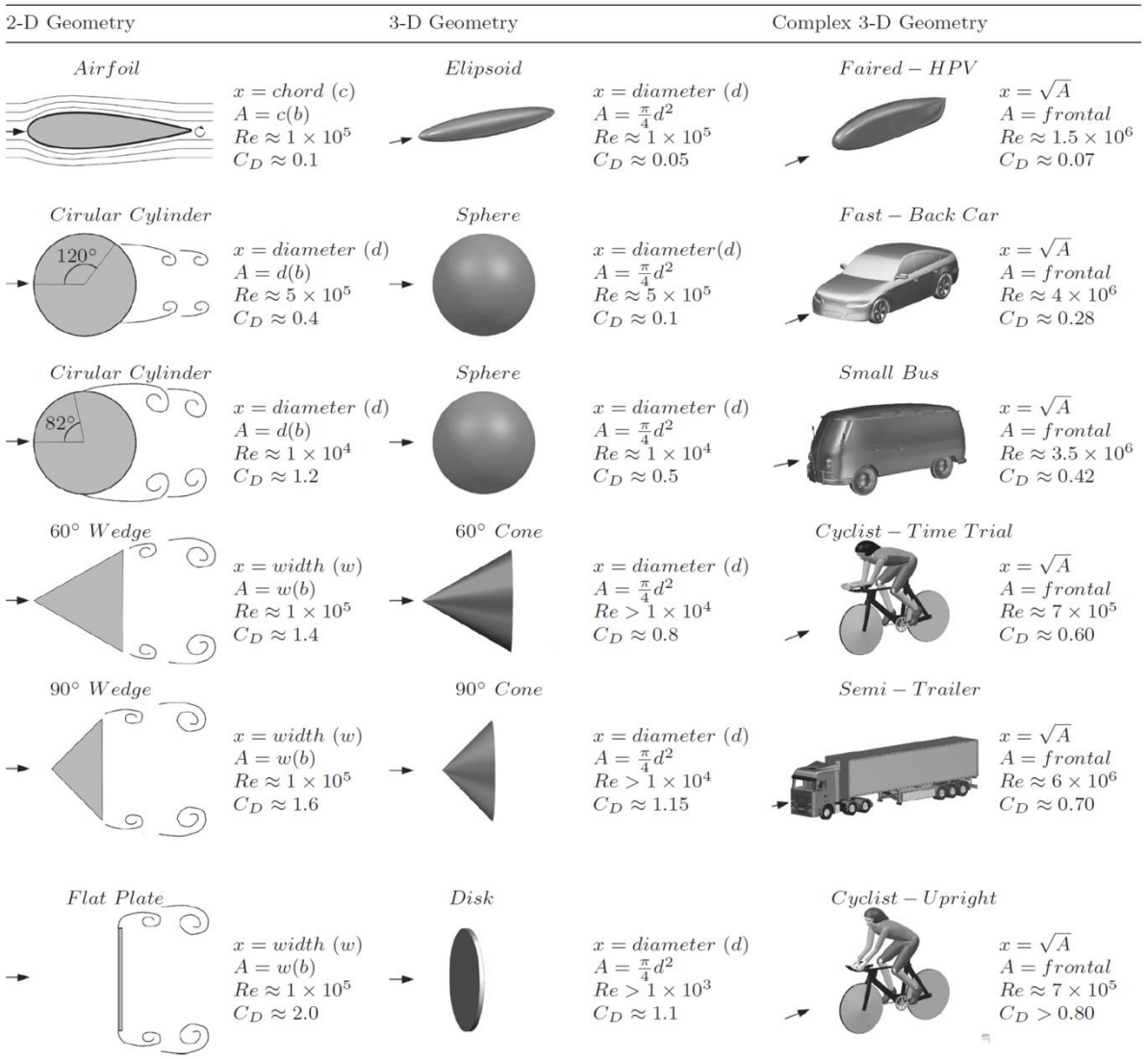
- This gives rise to an additional drag force
  - Acts in opposite direction to velocity, v
  - Magnitude is proportional to v^2

- 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



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:

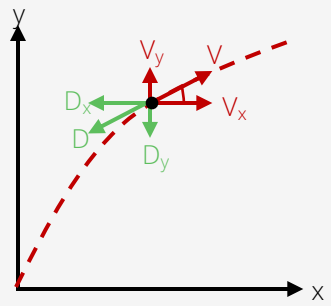
$$D = \frac{1}{2} C_D \rho_{\text{air}} A v^2$$

  - 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
```



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

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

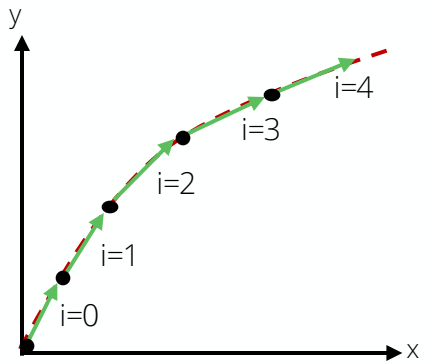
$$\Rightarrow \frac{d^2x}{dt^2} = -\frac{D}{m}\cos\theta$$

  - Vertical force now has two comps:
    - Gravity and vert. comp of D

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

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

- 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 $\theta = \theta(t)$ → 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



Projectile motion with air resistance (2)

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

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

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

  - 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

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

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

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

$$x_{i+1} = x_i + v_{x\,i}\delta t$$

$$y_{i+1} = y_i + v_{y\,i}\delta t$$

- 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 …

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

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

Projectile motion with air resistance (3)

```python
# 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

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

```python
# 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
```

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

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

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)

Projectile motion with air resistance (3)

```python
# 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

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

```python
# 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
```

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

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

Loop over steps and update position

```python
# 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)

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
```

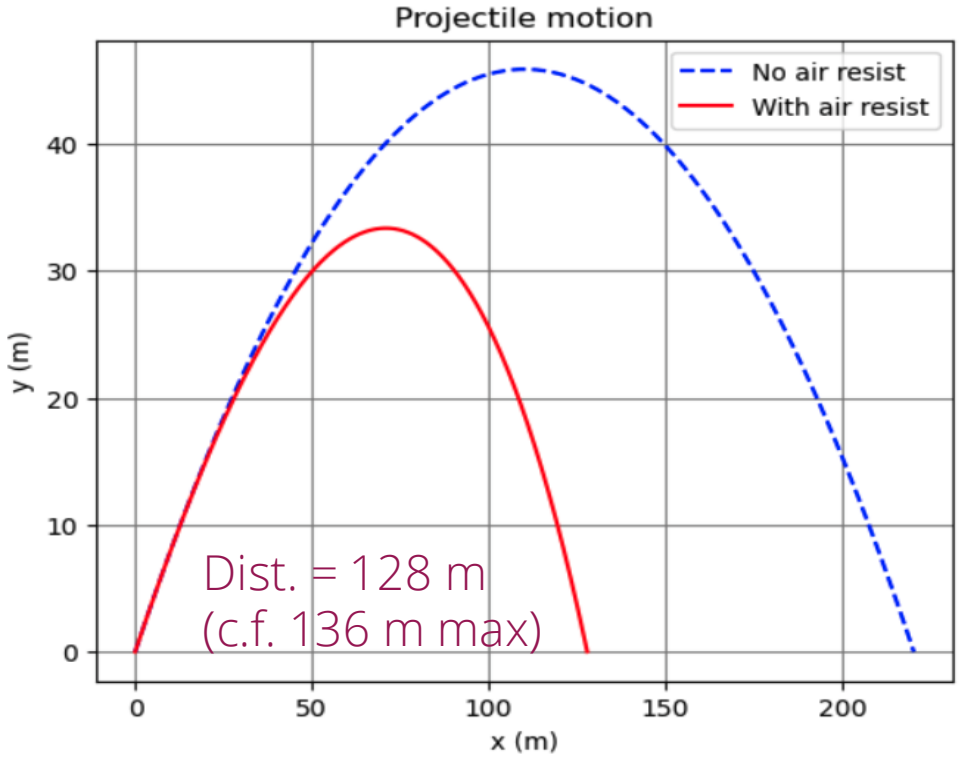
$$x_{i+1} = x_i + v_{x\,i}\delta t \qquad y_{i+1} = y_i + v_{y\,i}\delta t$$

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

Loop over steps and update position

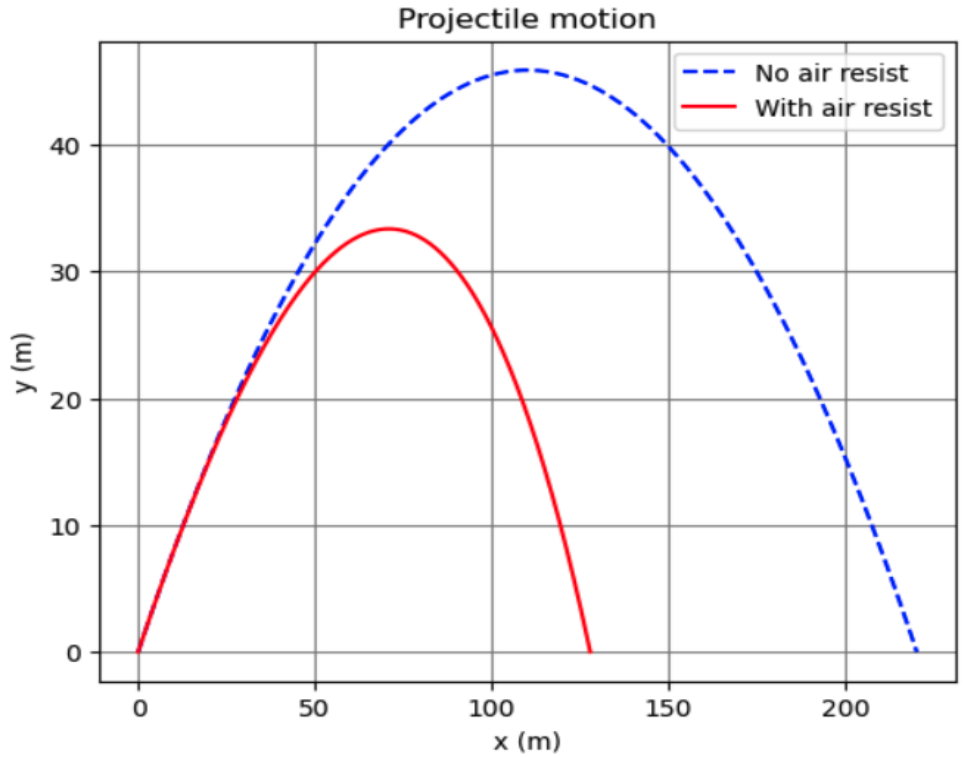
```
# Plot the resulting arrays up to last step before hits ground
# Compare result to the trajectory above w/o air resistance
plt.figure(figsize = (6, 5))
plt.title("Projectile motion")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'No air resist')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



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