Introduction to Computational Physics (PHYS105)

Lecture 10: Monte Carlo Methods

Carl Gwilliam

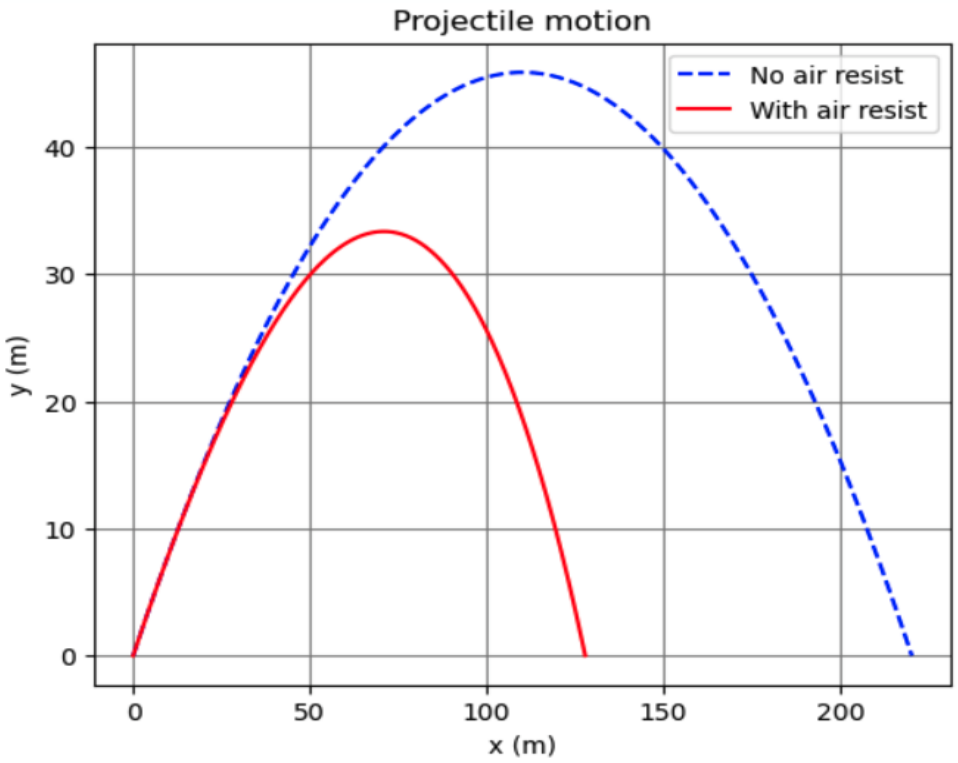
(C.Gwilliam@liverpool.ac.uk)

UNIVERSITY OF LIVERPOOL

Lecture 9: Recap

- Last week we looked at a real-world physics problem
  - Which could not be solved analytically
- We were able to solve this using Euler's method
  - Basically, just splitting it up into small steps
    - Such that angle is constant for each
  - Then iterating over the steps
    - Taking values from end of previous step as input
- Not only did this let you utilise numeric techniques
  - It also allowed you to practice writing (+ debugging!) larger programs that use all the python we've learnt
- In doing so we saw again the power of NumPy

Going from idealised situation to real world

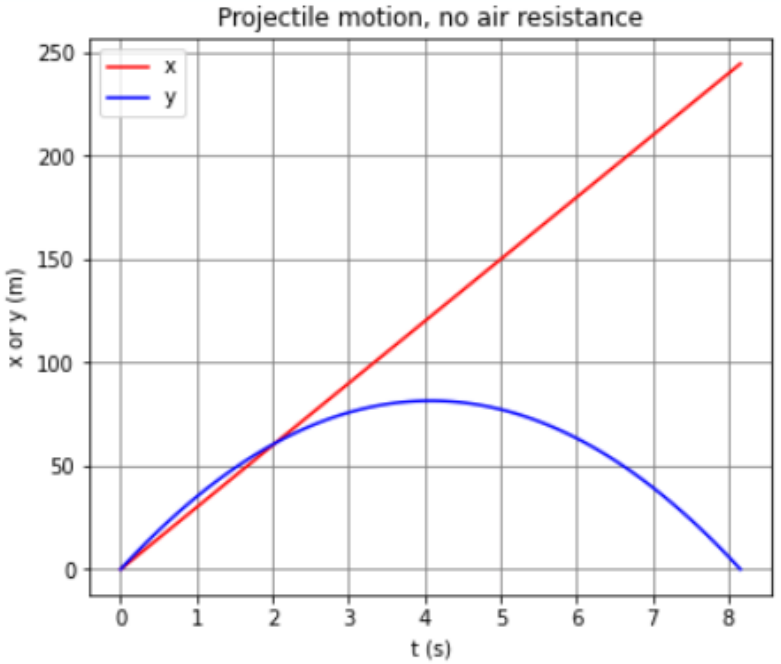


- This week we'll look at another type of numerical technique: so-called Monte Carlo methods

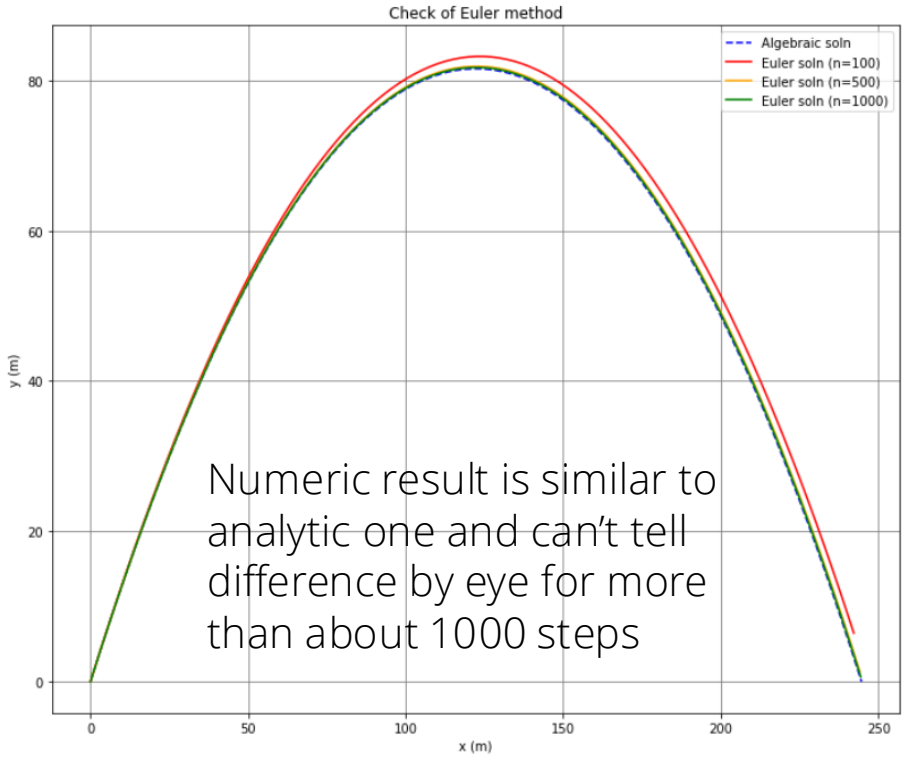
Lecture 9: Formative problem solutions

- Ex 1: plot both x and y as function of t
  - Recap of simple matplotlib plotting

```
plt.figure(figsize = (6, 5))
plt.title("Projectile motion, no air resistance")
plt.xlabel("t (s)")
plt.ylabel("x or y (m)")
plt.plot(tArr, xArr, linestyle = '-', color = 'r', label = "x")
plt.plot(tArr, yArr, linestyle = '-', color = 'b', label = "y")
plt.grid(color = 'grey')
plt.legend()
plt.show()
```



- Ex 2: Test that Euler's method works as expected + determine min number of steps
  - Testing is a key part of programming
  - Easiest way is to check with cases where you know the answer → here, we know the analytic solution for case of no resistance
    - So set $C_D = 0$ and compare to our analytic result



Lecture 9: Formative problem solutions (2)

- Ex 2 (cont.): Here we see again the advantage of using functions as reusable building blocks
  - You need to remember to recalculate dt each time as depends on number of steps

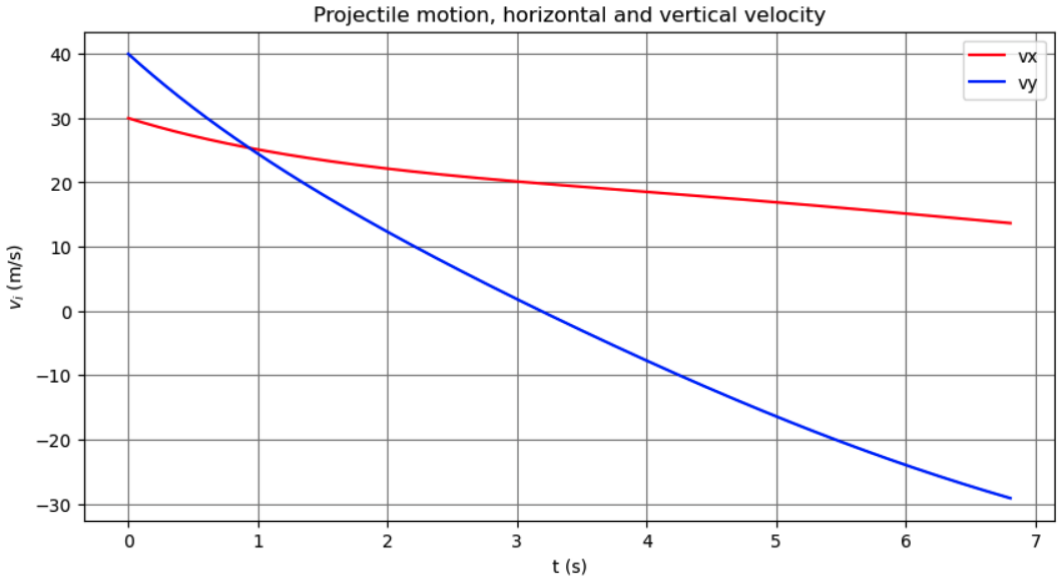
```
# Set drag coefficient to 0
CD0 = 0

# Call the function for increasing number of steps
# remembering to recalculate dt for each

nEuler1 = 100
dt1 = tMax/nEuler1
totStep1, xEuler1, yEuler1 = euler_projectile(mProj, xE, yE, vX, vY, dt1, nEuler1, g, rhoAir, CD0, Area)

nEuler2 = 500
dt2 = tMax/nEuler2
totStep2, xEuler2, yEuler2 = euler_projectile(mProj, xE, yE, vX, vY, dt2, nEuler2, g, rhoAir, CD0, Area)

nEuler3 = 1000
dt3 = tMax/nEuler3
totStep3, xEuler3, yEuler3 = euler_projectile(mProj, xE, yE, vX, vY, dt3, nEuler3, g, rhoAir, CD0, Area)

# Plot the resulting arrays for each number of steps and compare to the trajectory above w/o air resistance
plt.figure(figsize = (12, 10))
plt.title("Check of Euler method")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'Algebraic soln')
plt.plot(xEuler1[:totStep1], yEuler1[:totStep1], linestyle = '-', color = 'r', label = 'Euler soln (n=100)')
plt.plot(xEuler2[:totStep2], yEuler2[:totStep2], linestyle = '-', color = 'orange', label = 'Euler soln (n=500)')
plt.plot(xEuler3[:totStep3], yEuler3[:totStep3], linestyle = '-', color = 'g', label = 'Euler soln (n=1000)')
plt.grid(color = 'grey')
plt.legend()
plt.show()
```

Lecture 9: Formative problem solutions (3)

- Exercise 3: Plot both horizontal and vertical components of velocity (on the same plot)

```
# Calculate result
totStep, xEuler, yEuler, vXEuler, vYEuler = \
euler_projectile(mProj, xE, yE, vX, vY, dt, nEuler, g, rhoAir, CD, Area)

# Plot the x component of the velocity vs time, creating the time array
tEuler = np.linspace(0.0, tMax, nEuler)

plt.figure(figsize = (8, 5))
plt.title("Projectile motion, horizontal and vertical velocity")
plt.xlabel("t (s)")
plt.ylabel(r"$v_{i}$ (m/s)")
plt.plot(tEuler[0:totStep], vXEuler[0:totStep], color = 'r', label = 'vx')
plt.plot(tEuler[0:totStep], vYEuler[0:totStep], color = 'b', label = 'vy')
plt.grid(color = 'grey')
plt.legend()
plt.show()
```

Extra call to plot() with tEuler & now vYEuler




```
def euler_projectile(mass, x, y, vX, vY, dt, nSteps, g, rho, CD, area):
    "Function to calculate projectile motion with air resistance"


    # Create empty arrays to store x and y positions dn velocities for each step
    xEuler = np.zeros(nSteps)
    yEuler = np.zeros(nSteps)
    vXEuler = np.zeros(nSteps)
    vYEuler = np.zeros(nSteps)

    iStep = 0

    # Iterate up to the number of steps or until the projectile hits the ground
    while (y > 0 or iStep == 0) and iStep < nSteps:

        # Store the positions and vX before the current step
        xEuler[iStep] = x
        yEuler[iStep] = y
        vXEuler[iStep] = vX
        vYEuler[iStep] = vY

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

        # Calculate the drag given the velocities
        dragX, dragY = drag(CD, area, rho, vX, vY)

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

        # Don't forget to increment the step counter
        iStep = iStep + 1

    print("Final # step = {:d}, flight time = {:5.3f} s.\n".format(iStep, iStep*dt))

    # Return the number of steps, position arrays and vX array
    return iStep, xEuler, yEuler, vXEuler, vYEuler
```

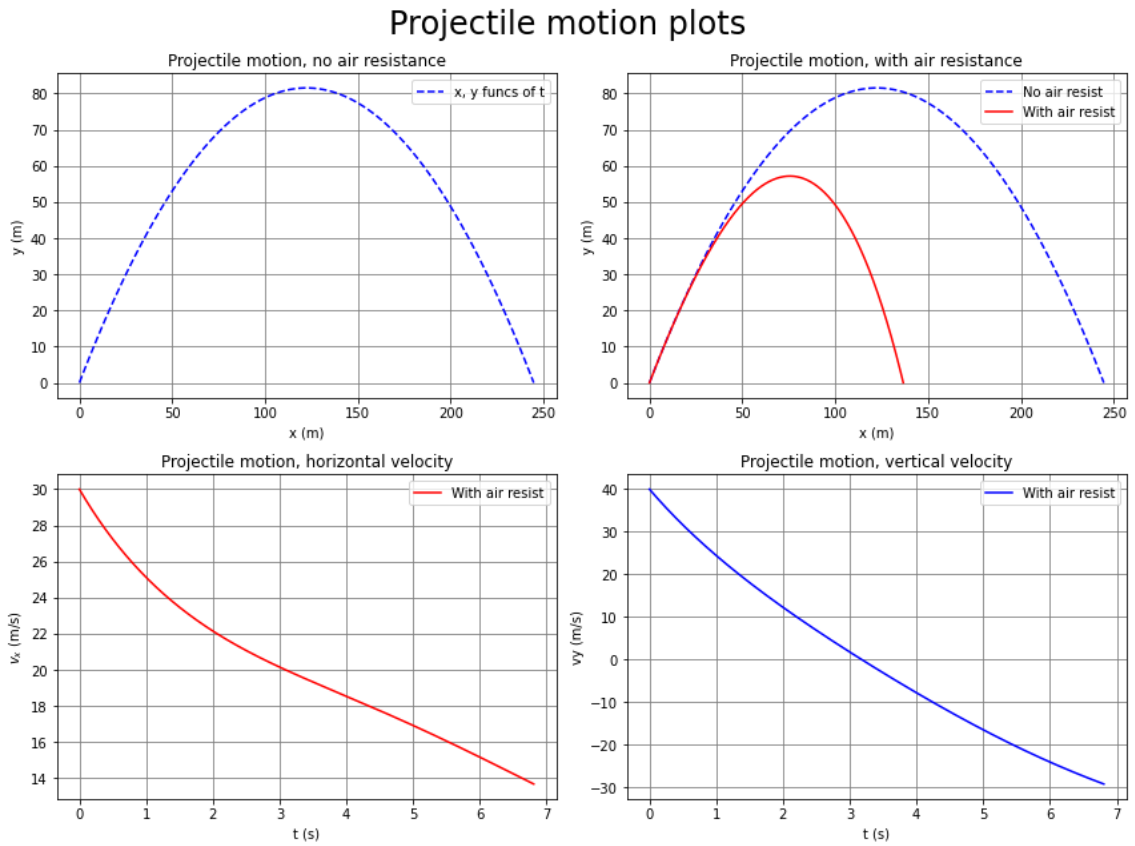
Add array for v_Y

Store value of v_Y each loop

Return vYEuler array

Lecture 9: Formative problem solutions (4)

- Exercise 4: Plotting multiple subplots in one figure → add a fourth plot to show v_y
  - Copy v_X code but change to `subplot(2,2,4)`
  - Key point is this comes before calling `show()`




```
# Open a figure and add an overall title
fig = plt.figure(figsize = (12, 9)) # opens a figure
fig.suptitle('Projectile motion plots', fontsize=24) # overall title

# Create subplots of the current figure in a 2x2 grid and move to the first one (top left)
plt.subplot(2,2,1)
plt.title("Projectile motion, no air resistance")
plt.xlabel("x (m)")
plt.ylabel("y (m)")
plt.plot(xArr, yArr, linestyle = '--', color = 'b', label = 'x, y funcs of t')
plt.legend()
plt.grid(color = 'grey')

# Move to the second square (reading from left to right, top to bottom) i.e top right.
# Note that you still have to specify the number of rows and columns
plt.subplot(2,2,2)
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:totStep], yEuler[0:totStep], linestyle = '-', color = 'r', label = 'With air resist')
plt.grid(color = 'grey')
plt.legend()

# Move to the third square i.e bottom left
plt.subplot(2,2,3)
plt.title("Projectile motion, horizontal velocity")
plt.xlabel("t (s)")
plt.ylabel("vx (m/s)")
plt.plot(tEuler[0:totStep], vXEuler[0:totStep], linestyle = '-', color = 'r', label = 'With air resist')
plt.grid(color = 'grey')
plt.legend()

# Move to the fourth square i.e bottom right
plt.subplot(2,2,4)
plt.title("Projectile motion, vertical velocity")
plt.xlabel("t (s)")
plt.ylabel("vy (m/s)")
plt.plot(tEuler[0:totStep], vYEuler[0:totStep], linestyle = '-', color = 'b', label = 'With air resist')
plt.grid(color = 'grey')
plt.legend()

# Show the whole figure
plt.tight_layout()
plt.show()
```

Monte Carlo (MC) methods

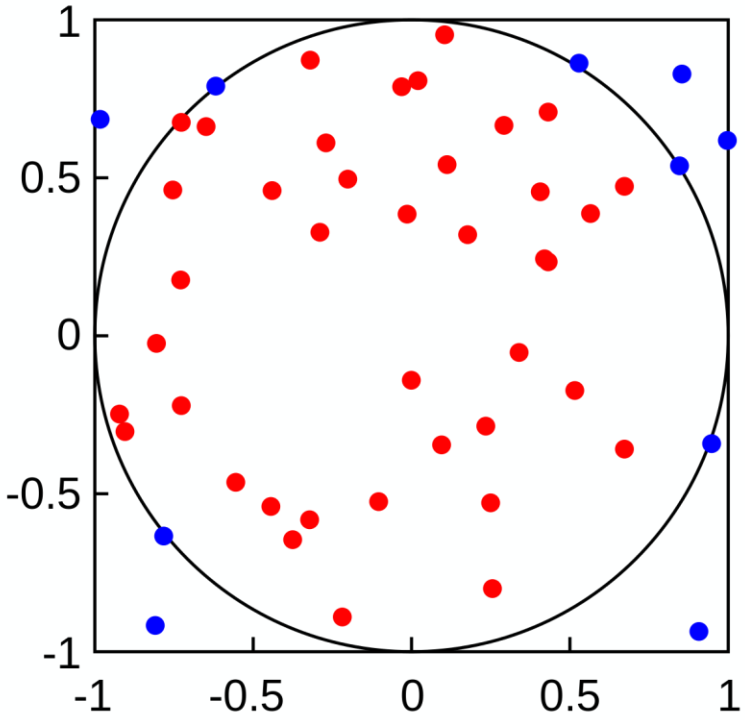
- So far, the numeric methods we have used, while approximate, are fully deterministic
  - This means the result is predictable: given the same input we always get the same result
- Monte Carlo methods are an alternate way to get numerical results using **random** sampling
  - In other words, by performing a set of random trials, or experiments, many times
- Although the underlying process is random we can still get a deterministic prediction
  - Provided we perform a sufficiently large number of trials or experiments
- Particularly useful for problems with large dimensionality, where other methods struggle
  - E.g. (Multi-dimensional) Integration & simulation

The Monte Carlo Casino in Monaco

General idea

- In simple terms, the Monte Carlo method can be summarised by the following general steps
  1. Define the range over which our inputs might possibly lie
  2. Generate many random inputs over that range
  3. Perform a deterministic computation on the inputs
  4. Use this to calculate the desired result

- For example, consider the following experiment
  1. Draw a square on the ground, then inscribe a circle within it
  2. Uniformly scatter grains of rice over it
  3. Count the number of grains inside the circle
  4. Take the ratio of this to the total number of grains



- The ratio will be equal to the ratio of the square and circle areas
  - Hence, if know the square area, we can calculate the area of the circle
  - This method may have been used by ancient Persian mathematicians

$$\frac{A_{circle}}{A_{square}} = \frac{N(\bullet)}{N(\bullet) + N(\bullet)}$$

- This idea can be extended to higher dimensions to e.g. calculate multi-dimensional integrals

NumPy random numbers

- Rather than performing real experiments, we can use random numbers generated in Python
- Can be achieved using NumPy's random module
  - Has several random number generators
- Default uses Mersenne Twister algorithm
  - Has a repetition period of $2^{19937} - 1$
- Obtained via `np.random.default_rng()`
  - Can optionally specify the starting point in the sequence via a *seed* as argument
    - Useful to get repeatable sequence
- The basic `random()` function generates uniform random numbers x over the range $0 \leq x < 1$
  - Either one value at a time or as an array

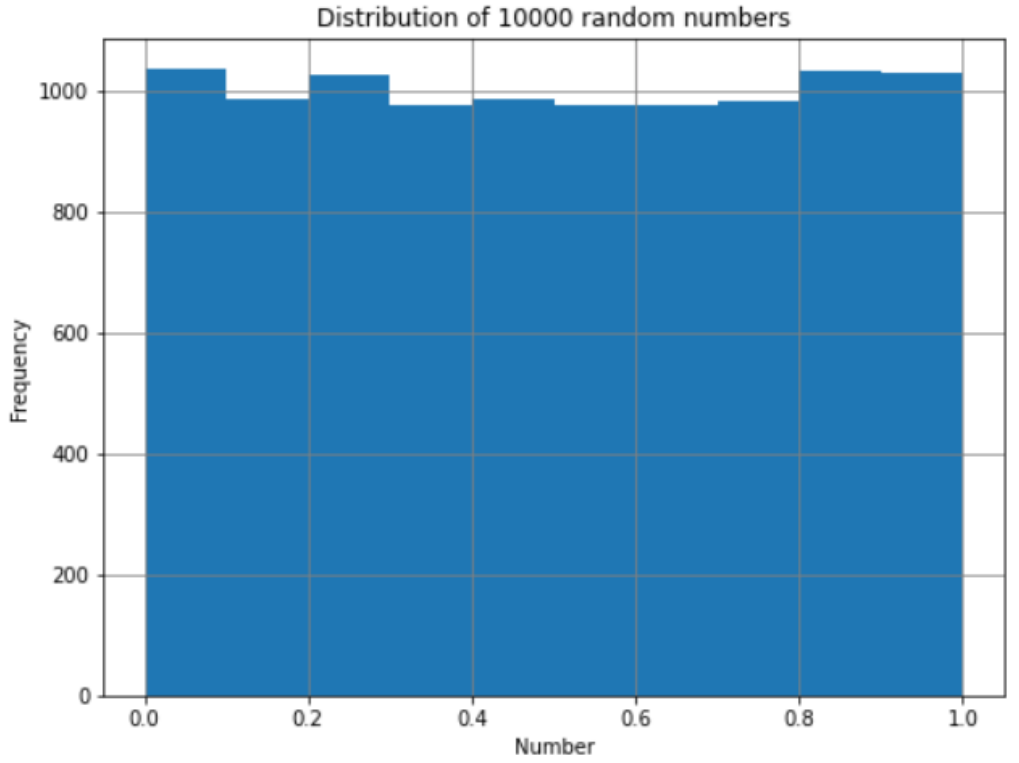
```python
import numpy as np

# Initalise random number generator
rng = np.random.default_rng()

# Generate one random number
x = rng.random()
print("1st random number:\n",x)

# Generate array of 4 random numbers
x1D = rng.random(4)
print("\n1D random number array:\n",x1D)

# Generate 3-by-2 2D array of random numbers
x2D = rng.random((3, 2))
print("\n2D random number array:\n",x2D)
```

```
1st random number:
 0.8054826104908346

1D random number array:
 [0.50980996 0.76206011 0.78524385 0.98866293]

2D random number array:
 [[0.58764682 0.5541268 ]
 [0.85872432 0.01182445]
 [0.53516147 0.1101651 ]]
```

NumPy random numbers

- Rather than performing real experiments, we can use random numbers generated in Python
- Can be achieved using NumPy's random module
  - Has several random number generators
- Default uses Mersenne Twister algorithm
  - Has a repetition period of $2^{19937} - 1$
- Obtained via `np.random.default_rng()`
  - Can optionally specify the starting point in the sequence via a *seed* as argument
    - Useful to get repeatable sequence
- The basic `random()` function generates uniform random numbers x over the range $0 \leq x < 1$
  - Either one value at a time or as an array

```
import matplotlib.pyplot as plt

rng = np.random.default_rng()

plt.figure(figsize = (8, 6))
plt.title('Distribution of 10000 random numbers')
plt.xlabel('Number')
plt.ylabel('Frequency')
plt.hist(rng.random(10000))
plt.grid(color = 'grey')
plt.show()
```



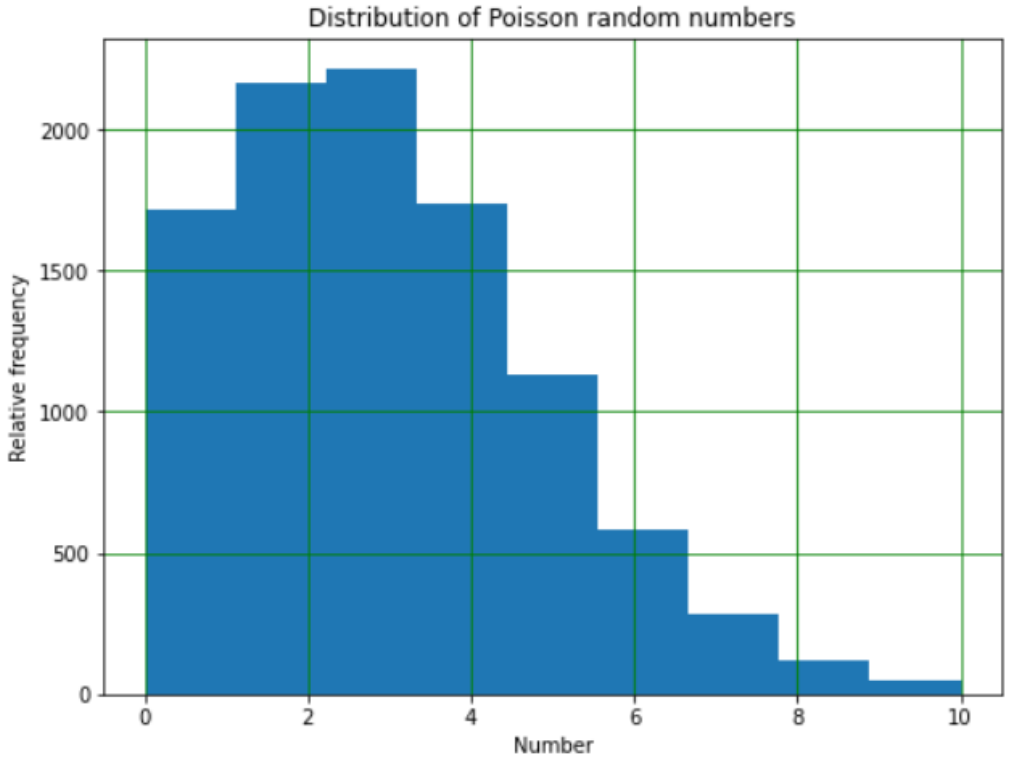
Poisson distribution

- Poisson distribution describes the probability that a given number of events occurs in a fixed interval
  - Events are discrete and can be over time or space
- Assuming events are independent of each other, the probability of seeing x events in the interval is

$$P(x, \lambda) = \frac{e^{-\lambda}\lambda^{x}}{x!}$$

  - Where λ is the known mean rate, which is constant
- Can generate random numbers following this via the `numpy.random` module's `poisson()` function
  - Takes mean and number of events as arguments
- Examples:
  - Number of cars passing a given point per minute
  - Number of traffic accidents on a given road per mile

```
# Generate Poisson random numbers about mean
mean = 3.2
nEvents = 10000
randArr = rng.poisson(mean, nEvents)

# Plot the result
plt.figure(figsize = (8, 6))
plt.title('Distribution of Poisson random numbers')
plt.xlabel('Number')
plt.ylabel('Relative frequency')
plt.hist(randArr, bins = np.linspace(0, 10, 10))
plt.grid(color = 'g')
plt.show()
```



Gaussian (or Normal) distribution

- The Gaussian (or Normal) distribution describes the probability of a continuous value x based on the mean value (μ) and its RMS spread (σ)

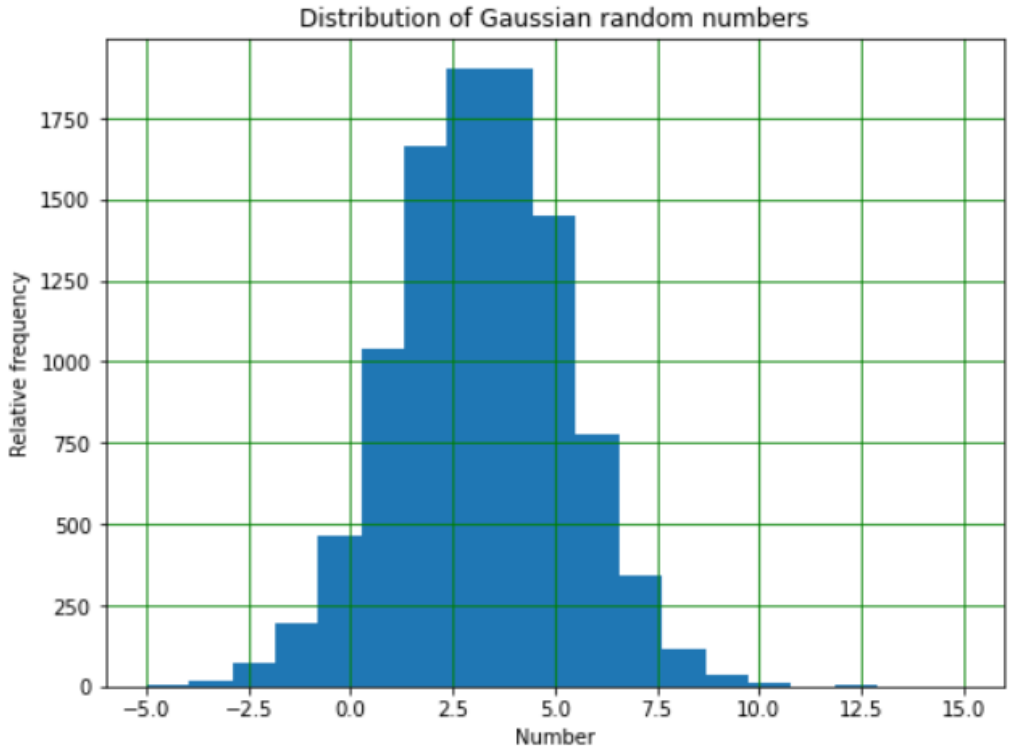
$$P(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}}$$

  - Has a characteristic symmetric bell shape

- Based on the Central Limit Theorem, the result of any sufficiently large number of random trials tends towards a Gaussian distribution
  - Hence it can describe many natural phenomena
  - E.g. heights/weights/IQs/etc of a population

- Can generate random numbers following this via the `numpy.random` module's `normal()` function
  - Takes mean, RMS & number of events as arguments

```
# Generate Gaussian random numbers about mean with given RMS
mean = 3.2
RMS = 2.1
nEvents = 10000
randArr = rng.normal(mean, RMS, nEvents)

# Plot the result
plt.figure(figsize = (8, 6))
plt.title('Distribution of Gaussian random numbers')
plt.xlabel('Number')
plt.ylabel('Relative frequency')
plt.hist(randArr, bins = np.linspace(-5, 15, 20))
plt.grid(color = 'g')
plt.show()
```

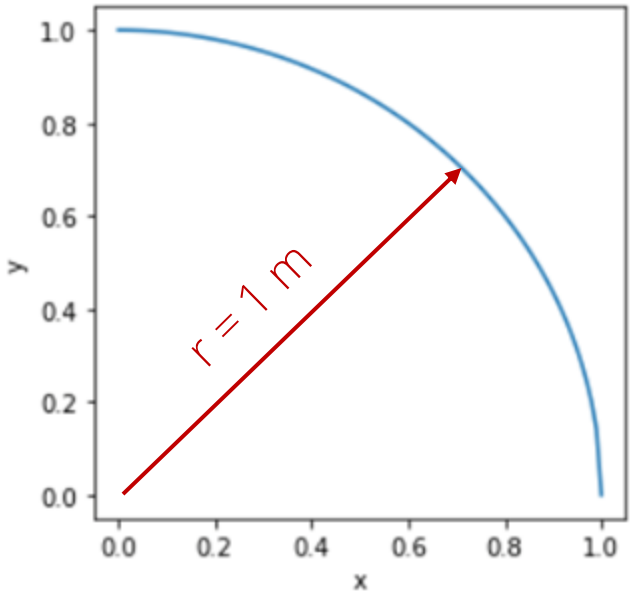


Monte Carlo integration

- Let's return to our earlier circle area example and use it to estimate the value of π numerically
  - Specifically consider the case of a 2x2m square with an inscribed circle of radius 1m

$$\frac{N_{\text{circle}}}{N_{\text{tot}}} = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi \times 1^2}{2 \times 2} = \frac{\pi}{4} \quad \Rightarrow \pi = 4\frac{N_{\text{circle}}}{N_{\text{tot}}}$$

- We can code this in python and since the problem is symmetrical we only need to model 1 quadrant
  - Use random number generator to create indep. random nums uniformly in x and y
  - Count how many lie within the circle



- The more random trials the more accurate π is
  - Will investigate this in the notebook

```
# Generate arrays of independent random x and y
# values in the range [0,1) for nGrains of rice
nGrains = 10000
print(f"Performing {nGrains:d} random trials ...")
rng = np.random.default_rng()
riceX = rng.random(nGrains)
riceY = rng.random(nGrains)

# Calculate the radial position of the grains
# from this using r**2 = x**2 + y**2
riceR = np.sqrt(riceX**2 + riceY**2)

# Sum number inside the radius of the circle
nCircle = np.sum(riceR < 1)
print(f"of which {nCircle:d} are within the circle")

# Calculate pi from derived formula
pi = 4*nCircle/nGrains
print(f"Giving pi as approximately {pi:6.5f}")
```

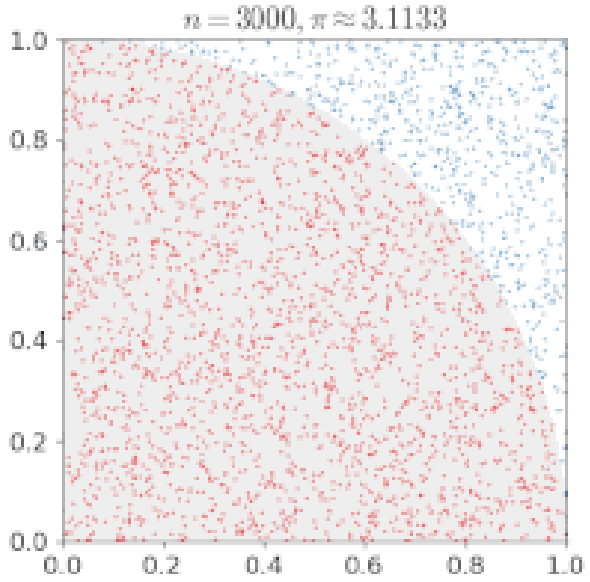
```
Performing 10000 random trials ...
of which 7862 are within the circle
Giving pi as approximately 3.14480
```

Monte Carlo integration

- Let's return to our earlier circle area example and use it to estimate the value of π numerically
  - Specifically consider the case of a 2x2m square with an inscribed circle of radius 1m

$$\frac{N_{\text{circle}}}{N_{\text{tot}}} = \frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi \times 1^2}{2 \times 2} = \frac{\pi}{4} \quad \Rightarrow \pi = 4\frac{N_{\text{circle}}}{N_{\text{tot}}}$$

- We can code this in python and since the problem is symmetrical we only need to model 1 quadrant
  - Use random number generator to create indep. random nums uniformly in x and y
  - Count how many lie within the circle



- The more random trials the more accurate π is
  - Will investigate this in the notebook

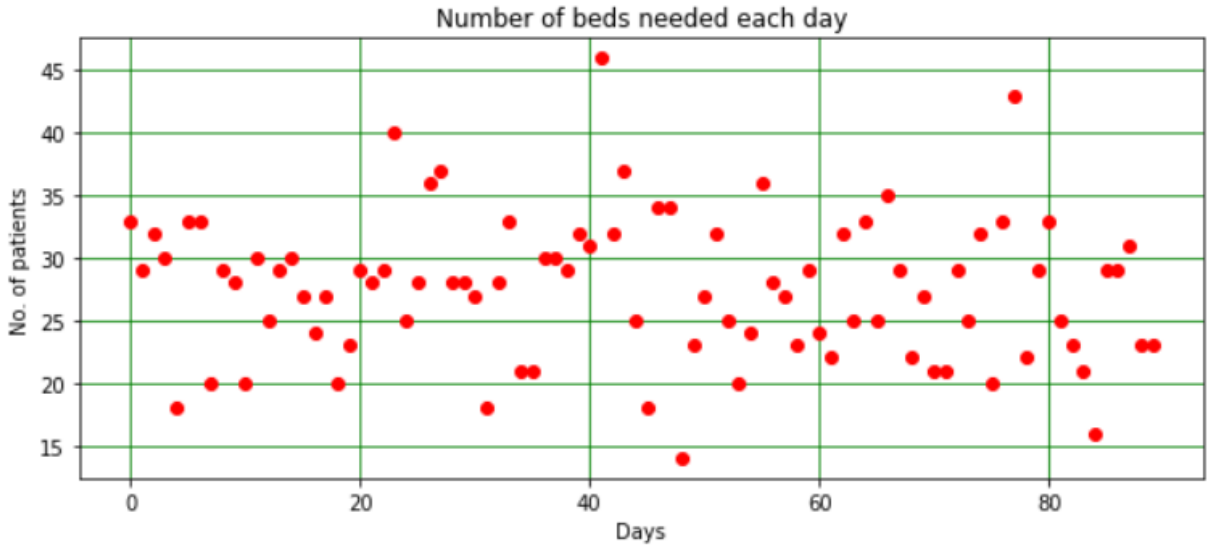
```
# Generate arrays of independent random x and y
# values in the range [0,1) for nGrains of rice
nGrains = 10000
print(f"Performing {nGrains:d} random trials ...")
rng = np.random.default_rng()
riceX = rng.random(nGrains)
riceY = rng.random(nGrains)

# Calculate the radial position of the grains
# from this using r**2 = x**2 + y**2
riceR = np.sqrt(riceX**2 + riceY**2)

# Sum number inside the radius of the circle
nCircle = np.sum(riceR < 1)
print(f"of which {nCircle:d} are within the circle")

# Calculate pi from derived formula
pi = 4*nCircle/nGrains
print(f"Giving pi as approximately {pi:6.5f}")
```

```
Performing 10000 random trials ...
of which 7862 are within the circle
Giving pi as approximately 3.14480
```

Monte Carlo simulation

- Let's now look at simulating the number of A&E beds needed in the Liverpool Royal Hospital
  - Given that records show the average number of people requiring a bed is 27/day in winter
  - Assuming that after 1 day, each A&E patient is moved to another ward to simplify the problem

- We can model this using a Poisson distribution
  - We know the average number of patients
  - Can use this to find expected number of patients (and hence beds) needed, each winter's day

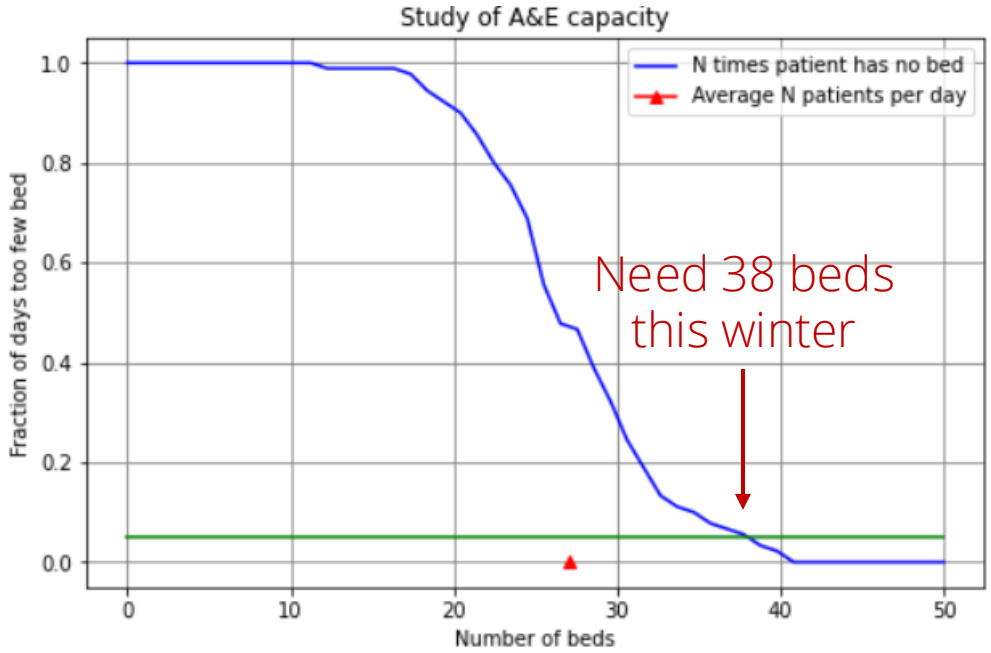



```
# Average number of patients per day
nPatsDay = 27

# Create an array of days for winter
nDays = 90
tArray = np.linspace(0, nDays - 1, nDays)

# Number of patients (i.e. beds) per day
# based on Poisson-distributed trials
patsPerDay = rng.poisson(nPatsDay, nDays)

# Plot number of beds needed per day
plt.figure(figsize = (10, 4))
plt.title("Number of beds needed each day")
plt.xlabel("Days")
plt.ylabel("No. of patients")
plt.plot(tArray, patsPerDay, color = 'r',
         linestyle = '', marker = 'o')
plt.grid(color = 'g')
plt.show()
```

Monte Carlo simulation (2)

- Suppose the hospital manager asks how many A&E beds they need so they never run out?
  - Not a well-formed problem: even an enormous number of beds will one day be too few
  - Need to make it more quantitative: let's say we want enough beds 95% of the time (i.e. 2σ)

- We can test the above distribution of patients against a range of different numbers of beds
  - And see what fraction of the times we run out

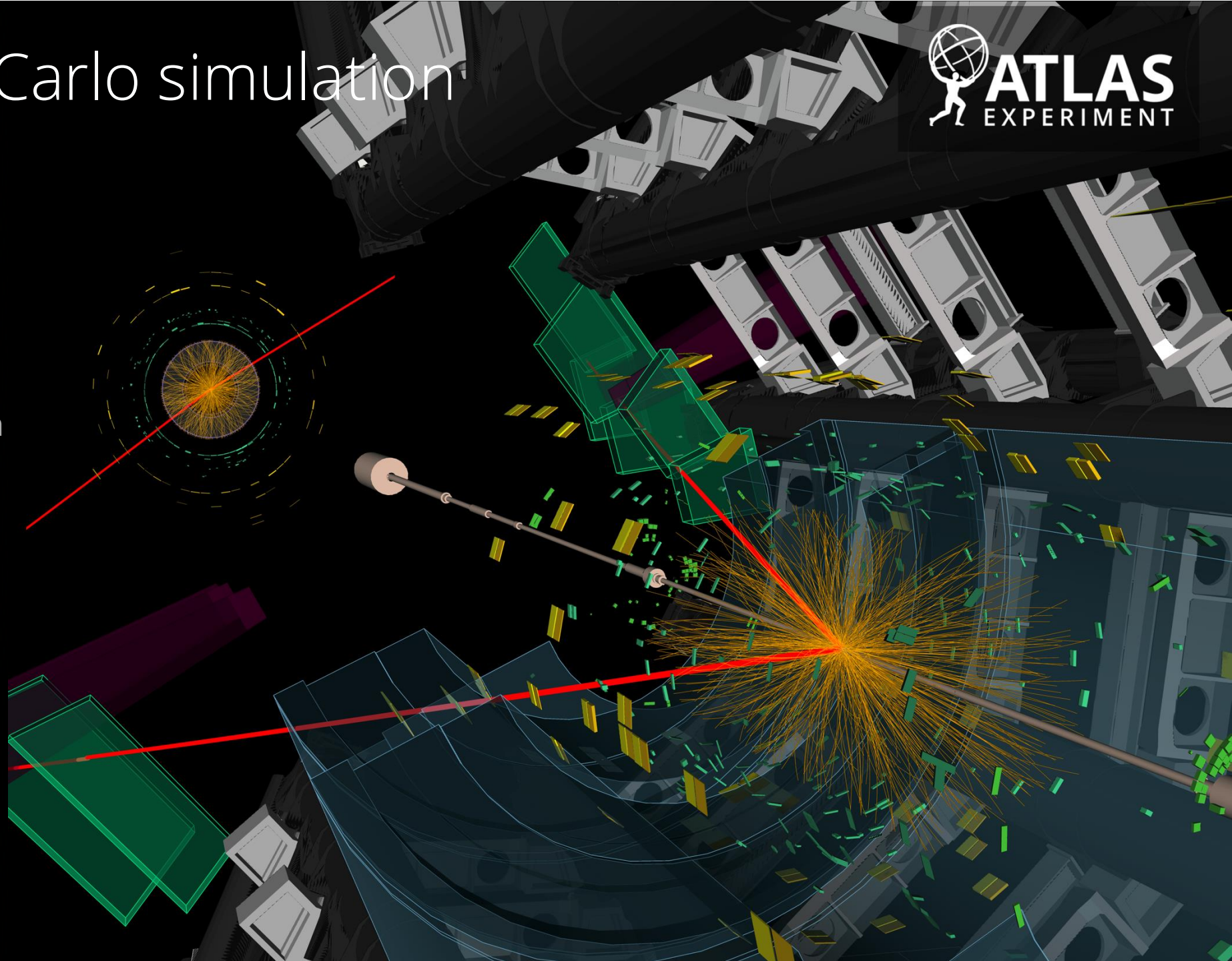



```
maxBeds = 50
tooFewBedsArray = np.zeros(maxBeds)

# Test various numbers of beds
for nBeds in range(0, maxBeds):
    # Check if enough beds each day
    nDaysNotEnoughBeds = np.sum(patsPerDay > nBeds)
    # Convert to fraction
    fDaysNotEnoughBeds = nDaysNotEnoughBeds/nDays
    # Store result for each number of beds
    tooFewBedsArray[nBeds] = fDaysNotEnoughBeds

# Plot fraction
bedNumbers = np.linspace(0, maxBeds, maxBeds)
plt.figure(figsize = (8, 5))
plt.title("Study of A&E capacity")
plt.xlabel("Number of beds")
plt.ylabel("Fraction of days too few bed")
plt.plot(bedNumbers, tooFewBedsArray, color = 'b',
         label = "N times patient has no bed")
plt.plot(nPatsDay, 0.0, color = 'r', marker = '^',
         label = "Average N patients per day")
plt.plot(bedNumbers, np.full(maxBeds, 0.05), color = 'g')
plt.legend()
plt.grid(color = 'grey')
plt.show()
```

- But what about next winter & the ones after?

Physics Monte Carlo simulation

- Monte Carlo are used extensively in physics
  - Across many domains
- Especially particle physics
  - To simulate particle production + interaction with the detector
  - E.g ATLAS Z → μμ event
- Millions of lines of code
  - Producing billions of simulated events/year
- You will look at a simple example of a gas particle in this week's notebook



Summary

- This week we looked at MC: a different type of numeric method based on random trials
  - Which can be very useful in cases of large dimensions, particularly for integration and simulation
- In doing so we introduced Python's random number generators
  - Seeing how to generate trials following different distributions
- In the notebook, you will look further at the examples we started in the lectures
  - And then, as said, simulate the motion of a simple gas particle
- **Reminder: answers must be your own, individual work and not created using generative AI**
  - Suspected breaches are taken seriously and can have severe consequences (up to being expelled from the course) as detailed in COPA Appendix L,
- Next week we will revisit fitting and pull things together into your own python module
  - That you can use In lab going forward and continue to add to over time