Introduction to Computational Physics (PHYS105)

Lecture 10: Monte Carlo Methods

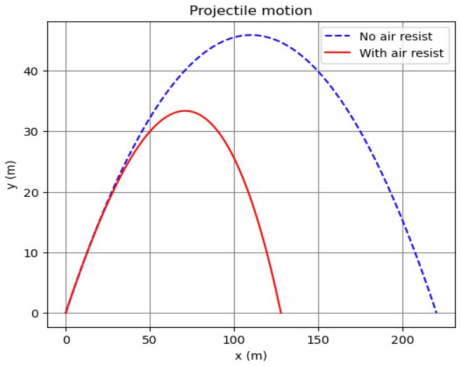
Carl Gwilliam
(C.Gwilliam@liverpool.ac.uk)

- **RECORD + attendance code !!**
- Morning everyone
  - Please register your attendance
- This week we are going to look at a new type of numeric technique known as Monte Carlo methods
  - As always, please feel free to interrupt with questions

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

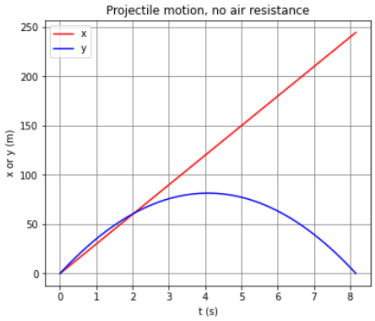
2

- **Before starting on this week's material let's first recap last week, where we looked at a real-world physics problem, projectile motion with air resistance, which …**
- As mentioned, this week, this week we'll look at …

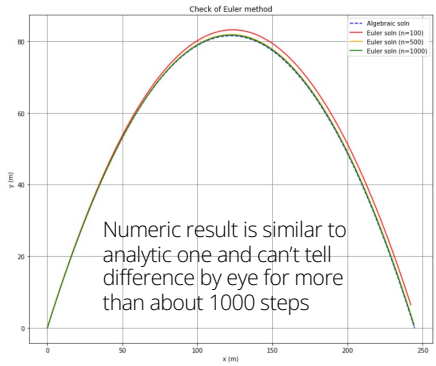
Lecture 9: Formative problem solutions

- 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



3

- First though, we will now look at the solutions to last week's formative problems
- Ex1: was to plot both the x and y positions as a function of time as a recap of our matplotlib plotting techniques
  - As usual we create a figure and give it a title and axis labels, **remembering the units**
  - We then call plot twice, each time giving it the time-array as the x values, and then either the x or y position as the y values
  - We then call show, remembering to include the legend
  - **The results, shown here, make sense since in the vertical direction the projectile goes up and comes back down as time progresses, while in the horizontal direction it just gets further away with time**
- Ex2: Was to test Euler's method worked as expected and determine the minimum number of steps needed
  - **Testing is an important part of programming and this exercise was to get you thinking about how to do that**
    - **A good way to do this is to test in cases where we know the answer**

- For numeric methods this often involves comparing to known analytic solutions for certain restricted cases
    - So, in this case we know the analytic solution for the case of no air resistance and if it is doing the correct thing the numerical method better be able to reproduce that if we turn the air resistance off
    - To do this we can just simply set our drag coefficient to 0 (or equivalently the area or air resistance to 0) so there is no drag force
- Here I have plotted the result for the no air-resistance case for different numbers of steps compared to the analytic solution
- You can see that the numeric result reproduces the analytic one and for more than 1000 steps or so we can't tell the two apart by eye
- Of course, in the real world we would use some compatibility test to compare the two and the level of agreement we require will depend on the accuracy we need

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()
```

4

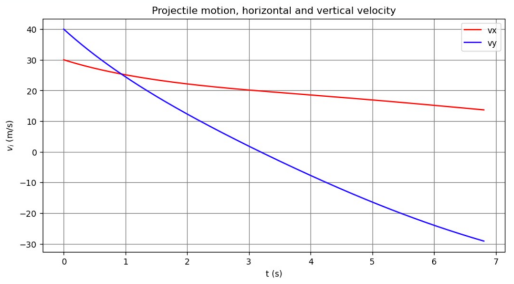
- To make this plot we simply call the function 3 times with different numbers of steps
  - No forgetting to update the time step dt
- See the advantage of functions as reusable blocks of code

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

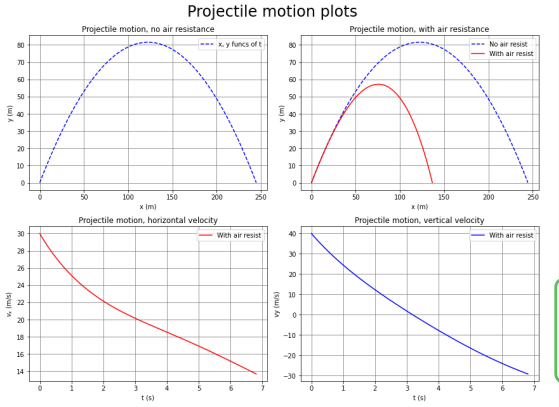
Store value of v_Y each loop

Return vYEuler array

- Ex3: was to plot the horizontal and vertical components of the velocity on one plot
  - First you had to calculate the vertical velocity by mimicking what was done for the horizontal case
    - We add an array before the loop to store the resulting values (point)
    - And then we simply add the velocity component to it during the loop (point)
    - **And add the array to the return values**
  - Then we call plot again, this time with the time vs the y-velocity component, to add an extra line to the plot
  - **Again, the result makes sense since in the vertical direction we have a larger force acting against the projectile, due to the extra gravitational force on top of the drag, so the velocity decreases faster**

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()
```

- **Ex4: The final exercise was showing you how to plot multiple subplots next to each other, which is often useful to present the full information about a result**
  - **Such as the position and the two velocity components**
  - To add a fourth plot, we can simply copy the code, just changing the last argument of the subplot so we are plotting on the fourth area of our grid (point)
    - **Remembering that the first two numbers in subplot are just the total number of plots we want in x and y**
  - **Of course, we also have to change what we are plotting**
  - The key point with subplots is that we make a **single** figure, then call subplot as many times as we want
    - This must be done **before** finally calling show() **once** for the entire figure
- **Any questions on the formative problems?**
- The Lecture 9 results have been released on canvas, with an average mark of 71%

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

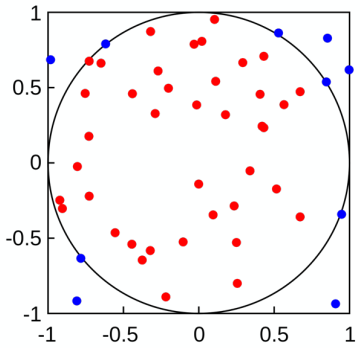
The Monte Carlo Casino in Monaco

7

- Read
- **Monte Carlo methods (linked here), which are named after the famous Monte Carlo casino, …**

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

8

- Read first bullet
- **This likely doesn't make much sense w/o context, so lets look at an example …**
- **Consider the following experiment**
  - … within it, that is centred on the middle of the square and just fits within it (point)
  - We then uniformly …. over it, for example by throwing them from a large height → random part
  - After doing so, we count … inside the circle (shown in red)
  - … and take … grains both inside (red) and outside (blue) the circle → deterministic part
- The ratio of these two will, of course, … areas (point to equation)
- Read last bullet
- **Any questions on the general idea?**

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 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 ]]
```

9

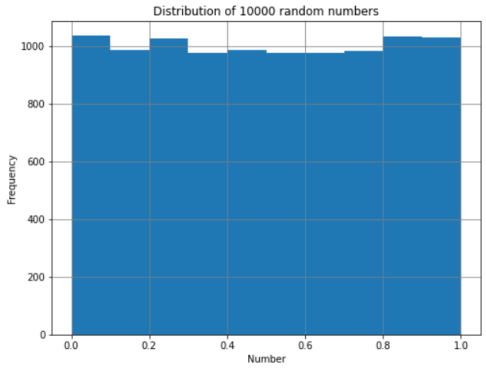
- Read first bullet
- The easiest way to do this is to use Numpy's random module, linked here, which has several types of random number generator
- **We will stick with the default one which uses an algorithm known as the Mersenne Twister (also linked) to generate random numbers**
- **The numbers are not completely random as after a certain time the sequence will begin repeating**
  - **But for Numpy's implementation of the Mersenne Twister this doesn't happen until we have drawn 2^19937 – 1 random numbers**
  - **So for our cases, we'll probably be OK :D**
- **Once we have imported numpy (point) we can obtain the random number generator by simply calling np.random and then the default_rng() function (point)**
  - **Here, rng stands for random number generator**
- **As you will see in the notebook, you can also specify the starting point in the sequence, known as the seed, as an argument**
  - **Which is useful to get repeatable sequences for comparisons or debugging**

- **The generator can produce random numbers following various distributions.**
- The most basic example is the random() function which generates random numbers uniformly, like the grains of rice in our previous example, over the range from 0 to 1
    - If we call this with no argument we get a single random number in that range (point)
    - If we call it with an argument we get an array of that many random numbers (point), for example 4 here
    - We can even pass it a tuple of numbers, in which case we get an n-dimensional array with the given number of random numbers in each dimension
        - For example a 3x2 array of random numbers here
- **Show that changes each time it is run → Random**

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()
```



10

- **If we generate a large enough set of random numbers, for example 10k here (point), and plot the result, we can see that they are indeed uniformly spread between 0 and 1**
  - You will investigate this more in the notebook
- **Any questions on the notion of random numbers?**

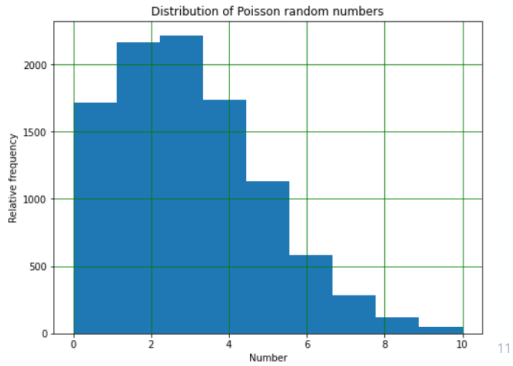
Poisson distribution

- 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()
```



11

- **In addition to being able to produce random numbers uniformly, we can generate them following several other distributions**
  - **We will look at a couple of these that are particularly useful for modelling physics problems and which you saw in your PHYS106 lectures**
- The first of these is the Poisson distribution, which …. given number of DISCRETE events, occur over a given internal in time or space
- **This can be used to model, for example, the… (read examples)**
- **Assuming … interval, given we know the mean rate (lambda) which is constant, is given by this formula:**
  - **The exponential of minus the mean (lambda) times the mean raised to the power of the number of events we are interested in (x), divided by the factorial of the**

number of events
    - You will have seen this in the stats lectures and also e.g the radiation experiment if you have done that
    - For example, if the mean number of cars passing a given point on a road is 3.2 per minute, the numbers measured each minute will follow a Poisson distribution with lambda = 3.2 as the mean
- We can generate random numbers following this distribution simply using the poisson() function from the numpy random module, in a similar way to the uniform() function we saw on the previous slide
    - As well as the number of events to generate it now also takes the mean as an input argument (point)
    - So, for our car example, if we wanted to model how many cars we'd see each minute we'd call this with 3.2 as the mean and the numbers we get each minute might be 2, 3, 2, 5, 4 etc
- In the plot here I have plotted 10k such events and the result is distributed asymmetrically around the mean with a tail to small and larger numbers
    - So, while our number of cars each minute is often round the mean it may sometimes be as low as 0 (point) or a high as 10 or more (point)

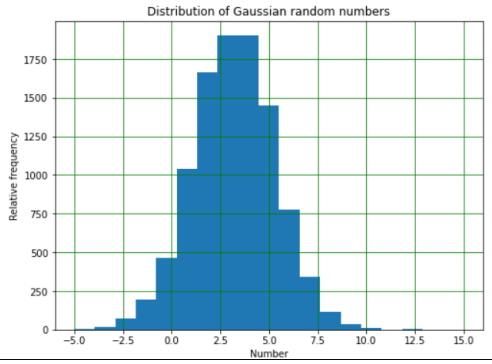
Gaussian (or Normal) distribution

- 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()
```



12

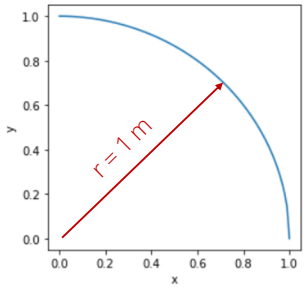
- **Another useful distribution is the Gaussian, or Normal, distribution that we have already encountered several times in the course**
- **This distribution describes the probability of a value x that is now continuous, rather than discrete as in the Poisson case**
- Given a known mean value (mu) and an RMS spread (sigma) the probability of seeing a given value x is described by the formula we have seen previously
  - Which has a characteristic bell shape
- The CLT, linked here, tell us is that the result of any …
  - (read subbullets)
- We can generate random numbers following a Gaussian distribution using the normal() function, which takes the mean, RMS and number of events (point) as arguments.
- **Here I have picked a mean of 3.2 again and an RMS of 2.1 and again generated 10k events**
  - **We can see the result is a distribution around the mean which now, unlike the Poisson case, is symmetric.**
- **Any questions on random numbers?**

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
```

13

- Now that we know how to generate random numbers in python, let's return to our circle example
  - And consider specifically a case of a 2x2m square with a inscribed circle of radius 1m
- We know that the ratio of the number of grains in the circle to the total number is the ratio of the area of the circle and the square
  - Which for this specific case it is obviously pi x 1m^2 divided by x 2x2m
  - So, the result is pi/4, which can be rearranged to estimate pi numerically
- Rather than throwing grains of rice we can now use our Python uniform random number generator to produce random x and y positions
  - Since the problem is symmetric we can cut it down to just doing one quadrant, as shown here, which aligns

with numpy's random range 0->1

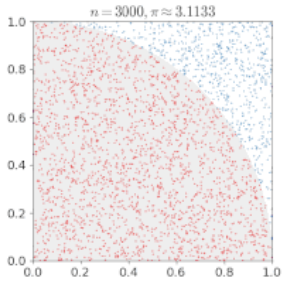
- **So, lets see how we would code this up (point to each piece)**
  - We get our default random number generator, that we will call rng
  - We then call the random function twice, each with 10k trials, to generate independent random distributions of the x and y positions
  - We can calculate the radial position for each of these x-y points using Pythagoras
  - We then use numpy's array-at-a-time functionality to find only those with a radius below 1m i.e. in the circle and sum this (T = 1, F = 0)
  - Following our formula we then take 4 times the ratio of the number of trials inside the circle to the total number of generated trials
- And, for 10k trails, we get pi as 3.1448 (point)
  - **Will change each time as random**
  - (switch slide)

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}")
```

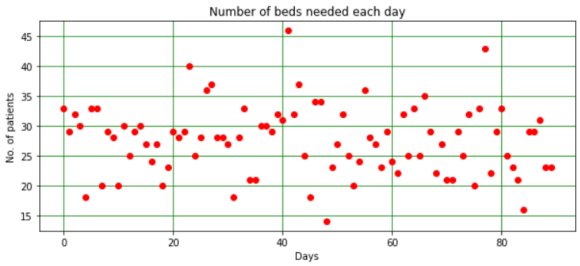
```
Performing 10000 random trials ...
of which 7862 are within the circle
Giving pi as approximately 3.14480
```

14

- Of course, the more trails we perform (i.e. generating more random numbers) the more we fill out the two areas (as shown here – point) and the more accurate the result (point)
  - You will investigate the accuracy of our estimate and how it improves with more trails in the notebook
- Any questions on MC integration?

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



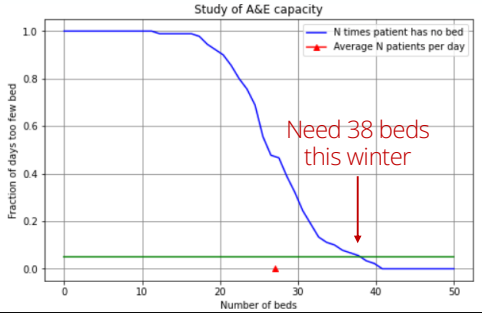

```
# Average number of patients per day
nPatsDay = 27

# 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()
```

15

- **Let's now look at a different type of problem where, instead of integration, we use Monte Carlo to start to simulate a real-life problem**
- The problem we will look at is estimating the number of A&E beds needed in the Liverpool Royal Hospital. Record show that …
  - To simplify the problem we will assume that … ward
- **Given we know the mean rate, we can use our Poisson distribution to model the number of patients, which we can use to find … day.**
  - We first setup a variable for our average number of patients per day (point)
  - We then create an array of the number of days from 0 to 90, which is the number of days in a meteorological winter, using linspace (point)
  - We then use NumPy's Poisson function, passing our mean and the number of days as arguments, to get the corresponding array of patients per day (point)
- **If we then plot this vs the time (point) we get this plot which shows the number of patients per day over the winter**
  - **Which as expected fluctuates around the mean of 27**
  - **If you project this onto the y axis, which you will do in the notebook, you will see that it follows a Poisson distribution**

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



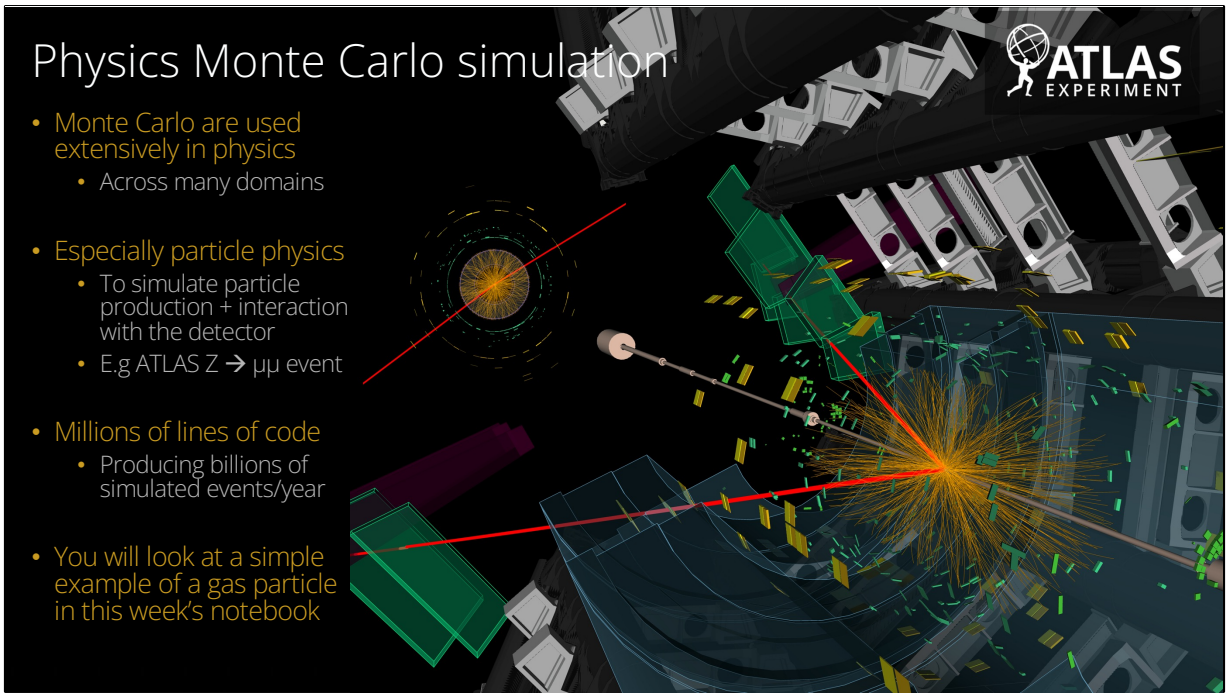

```
maxBeds = 50
tooFewBedsArray = np.zeros(maxBeds)

# 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?

16

- **Now we can use this to predict things.** Suppose …
  - The first thing is that this is not a … few, for example during an unexpected pandemic!
  - To make it more quantitative lets say we want enough beds 95% of the time (which is 2 sigma)
    - **Of course, in reality we probably want this to be much higher e.g. 99% (or 3 sigma) and you will use this in the notebook but I use 2 sigma here just so I need to generate less trials**
- **To do this we can test our distribution of patients we created above against a range of number of available beds and … out in each case**
- So let's write some code to do this ….
  - Let's test a number of beds from 0 up to a maximum number of 50 (point)
  - We first set up an array of this length to store the results for each number of beds we are going to test
- **We then want to test the various number of beds so we use a for loop to iterate over the range from 0 to our max number of 50**
  - **For each number of beds, we use numpy's array-at-a-time**

methods to see on how many days our random number of patients exceeds this number and sum this up
        - We then divide this by the total number of days to find the fraction of days where we don't have enough beds and store this in our results array
- We can then plot the results
    - We create an array of beds from 0 to 50 using linspace, and then plot our result against this
    - I also mark the mean number of patients and draw a line at 5%
- From the resulting plot we can see that as we increase the number of beds available we reduce the fraction of days we run out of beds as expected
    - Which passes 50% when we are at the average
    - And goes below 5% at 38 beds
- So we would need 38 beds to have enough beds 95% of the time
    - But this is only for this winter. What about the next one and the ones after this?
    - We clearly need to simulate more winters to be sure and again you will investigate this in the notebook.
- Any questions on MC simulation?



- Such MC method are … domains
- This is especially true in my research area of particle physics, where they are used to … detector
- **For example, this is a display of a simulated decay of a Z boson to two muons, shown by the red lines, in the ATLAS detector at the LHC**
  - **Each of the orange lines shows a charged particle track and each of the yellow and green blocks a calorimeter energy deposit**
  - **Monte Carlo is used to generate each of these particles and simulate the interactions with the material of the detector**
- **As you might expect, this is a very complex simulation and in fact takes millions of lines of code**
  - **ATLAS uses this to produce huge numbers of such simulated events: over a billion per year**
- **Of course, you won't code anything this complex, but you will look at a simple example of simulating a single 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
- 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

18

- So in summary …
- **Any final questions?**