Udai Sharma

PS-530

Computational Physics

30 March 2023

Monte Carlo Method for Criticality Conditions

The Monte Carlo Code is a tool for estimating the critical size of various objects with respect to the growth of thermal neutron density due to nuclear reactions, in particular, fission.

The aim of the assignment is majorly divided into 2 segments.

Part I: Devising an optimal strategy for finding the critical length for slab geometry, corresponding to a given computer budget (measured by the number of trajectories). By an optimal strategy, we aim to balance the trade-off between the two factors that affect the queue length (a proxy for neutron density):

- Statistical Issues
- Transient Effects

A short queue will flush out the initial neutron group rapidly, but have relatively large fluctuations in the queue length. A long queue will have a small statistical fluctuation in the queue length but will flush out the initial neutrons slowly.

Part II: Once we have devised an optimal strategy for balancing the trade-off between the statistical issues and transient effects. We aim to implement a cylindrical geometry which has its axial length equal to the radius, i.e. Height (H) = Radius (R). Then, we use the Monte Carlo code for estimating the critical length of our cylinder geometry.

WORKING

PART - I

We initially populate the queue with some sample set of neutron and investigate the path followed by the neutron and check how the count of neutrons changes. If it rises, it indicates that the we are below the critical length and if it falls, we are above the critical length. This fluctuation could be deduced to two factors. Firstly, purely statistical issues can cause the fluctuation in the neutron density. The other reason could be the transient effects induced by the initial neutrons. As the starting point of the path affects the average number of new neutrons created along the trajectory. Neutrons will not be produced uniformly in the test volume if, for instance, our beginning group is distributed evenly throughout the test container. Not before a new group of neutrons had completely replaced the initial group. So, it is probable that at the start of the calculation, we would see fluctuations in the queue length that are essentially statistical and temporary, lasting for a finite number of trajectories. Therefore, The precision with which we can establish the critical length is constrained by these effects.

It can be difficult to precisely determine the critical length using neutron transport Monte Carlo simulations. Within a certain computer budget, a number of criteria need to be taken into account in order to optimize the method for determining the critical length.

One approach might be to, manually begin with a short queue length and progressively expand it until a generally consistent behavior is seen in order to create a balance between these conflicting aspects. The critical length could be calculated based on the trend of the neutron population over a period of time, and once the behavior stabilizes. This strategy would reduce statistical instabilities and minimize transient effects while effectively utilizing the limited computational budget.

Another approach for optimizing the calculation of the critical length in Monte Carlo simulations of neutron transport is adaptive queue length. The plan is to dynamically modify the queue length throughout the simulation based on the neutron population's behavior. It can be adapted by using a feedback loop that modifies the queue length after every k trajectories, where k is a parameter that can be customized based on the problem's complexity and available computer resources. A more complex rule depending on the behavior of the neutron population can be used in the feedback loop in place of a simple rule like changing the queue length by a certain percentage.

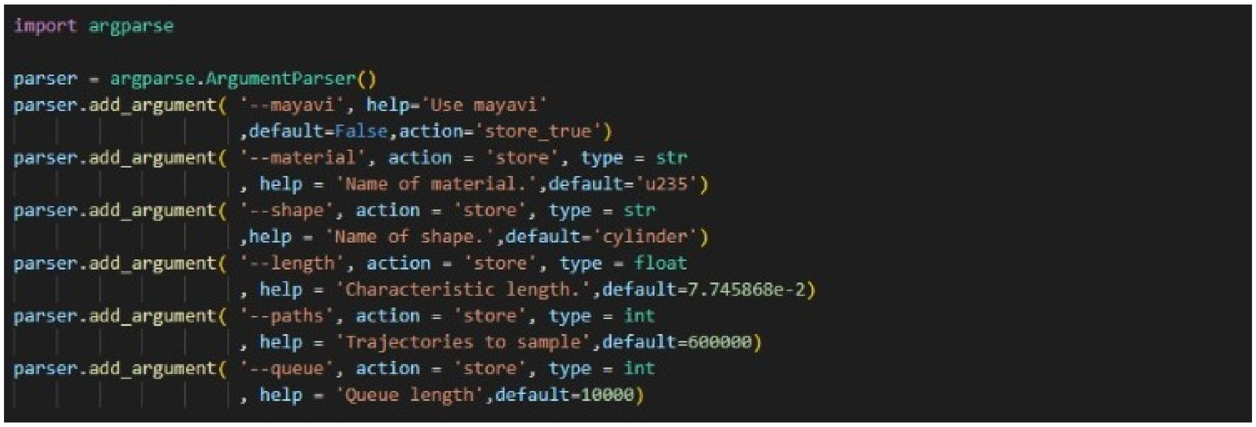
In conclusion, the effective approach for establishing the critical length in neutron transport Monte Carlo simulations would depend on a number of parameters, including the available computational budget, the desired accuracy, and the problem's complexity. a mix of techniques, such as manually selecting a short queue length and raising it gradually until a consistent behavior is established or by establishing a feedback loop-based code that dynamically modifies the queue length

PART-II

We initially modify the code to adopt it for a cylindrical geometry by creating a class of 'cylinder' object inherited from class 'shape'. After adapting the code for the cylindrical geometry, we use an iterative technique to find the critical length for the cylinder. We first pick a critical length and begin tracking the trajectories to determine whether the count of neutron particles in the queue increases or decreases. If the number decreases, the critical length estimate is short. The critical length estimate is higher if the number increases. Based on this, we increase or decrease the critical length's estimated value. Further, we continue until the critical length has been accurately determined to a satisfactory accuracy, , we can use this approach no matter how complicated the test volume's form is. The only condition necessary to determine is if a position is inside the test volume. In the results obtained for the cylindrical geometry, we get the satisfactory value of critical length at 7.745868 cm, which is equal to the diameter or twice the radius/axial length of the cylinder. Note that it is also greater than the critical length in the case of slab geometry. Unlike in a cylindrical geometry, where neutrons can take curved trajectories, in a slab geometry, neutrons move in straight lines and are more likely to leave the system. Henceforth, given that all other circumstances are identical, the critical length of a cylindrical geometry should be greater than that of a slab shape. Further, We can see the growth rate converges to a zero value but this self consistent result doesn't show up until a sufficient number of trajectories are fulfilled, which was anticipated as well due to the transient effect caused because of the replacement of the old set of neutrons by the new set of neutrons.

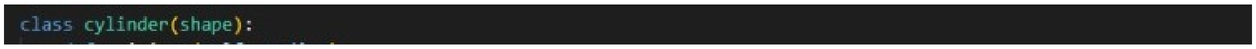
2.1 Modified Code

- By manually adjusting the characteristic length, we find an optimal solution for the criticality condition. Henceforth, the characteristic length is set to the value 7.745868e-2. The number of paths have been adjusted to 600000 according to the availability of computational resources.



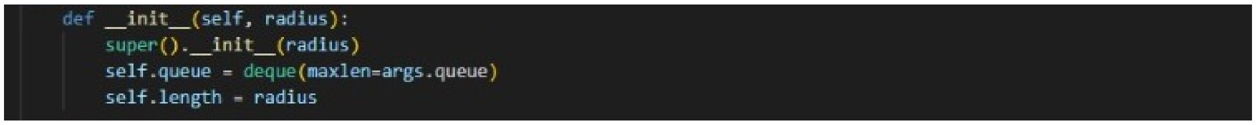
```python
import argparse

parser = argparse.ArgumentParser()
parser.add_argument( '--mayavi', help='Use mayavi'
                    ,default=False,action='store_true')
parser.add_argument( '--material', action = 'store', type = str
                    , help = 'Name of material.',default='u235')
parser.add_argument( '--shape', action = 'store', type = str
                    ,help = 'Name of shape.',default='cylinder')
parser.add_argument( '--length', action = 'store', type = float
                    , help = 'Characteristic length.',default=7.745868e-2)
parser.add_argument( '--paths', action = 'store', type = int
                    , help = 'Trajectories to sample',default=600000)
parser.add_argument( '--queue', action = 'store', type = int
                    , help = 'Queue length',default=10000)
```

- A class for object ‘cylinder’ is defined which is to be investiagated, it is inherited from the class ‘shape’. In this class, we define the set of functions that need to be implemented to define the geometry of the cylinder and the actions necessary to use Monte Carlo simulations on this geometry.



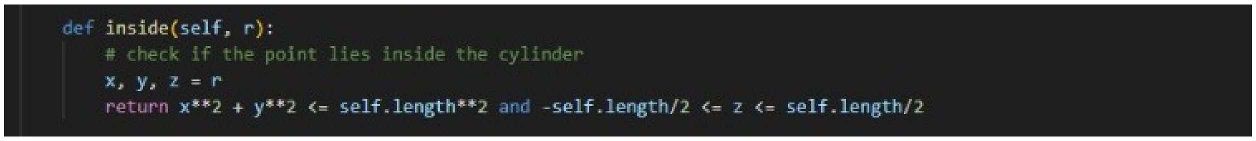
```python
class cylinder(shape):
```

- The ‘init’ function takes an input ‘radius’ and initializes it to a value. The axial length of the cylinder is also set to the radius here.



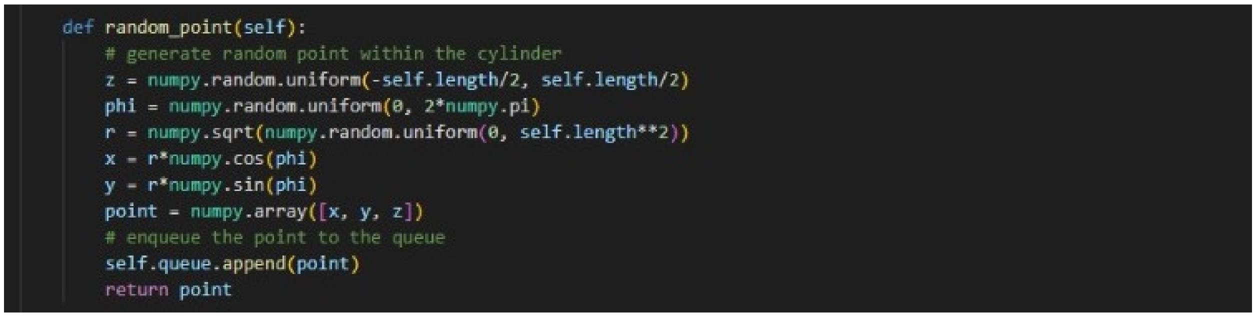
```python
    def __init__(self, radius):
        super().__init__(radius)
        self.queue = deque(maxlen=args.queue)
        self.length = radius
```

- The ‘inside‘ function takes input a single parameter ‘r’ and output’s a value, to check whether the point ‘r’ lies inside the cylinder or not. It calculates the distance ‘x**2+y**2’ and checks whether it is less than or equal to the radius



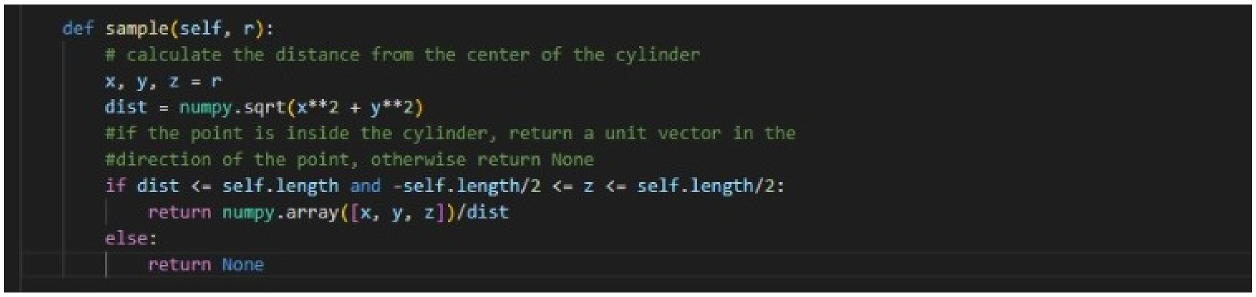
```
def inside(self, r):
    # check if the point lies inside the cylinder
    x, y, z = r
    return x**2 + y**2 <= self.length**2 and -self.length/2 <= z <= self.length/2
```

- The random_point function creates a random point inside the cylinder by choosing a random value for the z coordinate inside the cylinder's boundaries, a random angle between 0 and 2pi, a random radius value inside the cylinder's boundaries, and then computing the x and y coordinates using the formulas x=rcos(phi) and y=r*sin(phi). The point is then returned as a NumPy array. The point is also put in a queue to be used later.



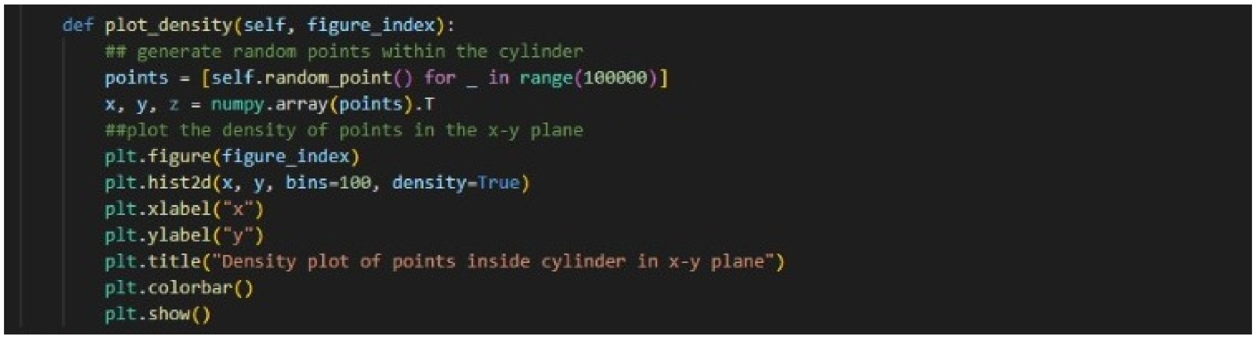
```
def random_point(self):
    # generate random point within the cylinder
    z = numpy.random.uniform(-self.length/2, self.length/2)
    phi = numpy.random.uniform(0, 2*numpy.pi)
    r = numpy.sqrt(numpy.random.uniform(0, self.length**2))
    x = r*numpy.cos(phi)
    y = r*numpy.sin(phi)
    point = numpy.array([x, y, z])
    # enqueue the point to the queue
    self.queue.append(point)
    return point
```

- This function takes in a point r as input and it determines whether or not it is inside the cylinder. It returns a unit vector in the direction of the point if it is inside the cylinder. It returns None if it lies outside the cylinder.



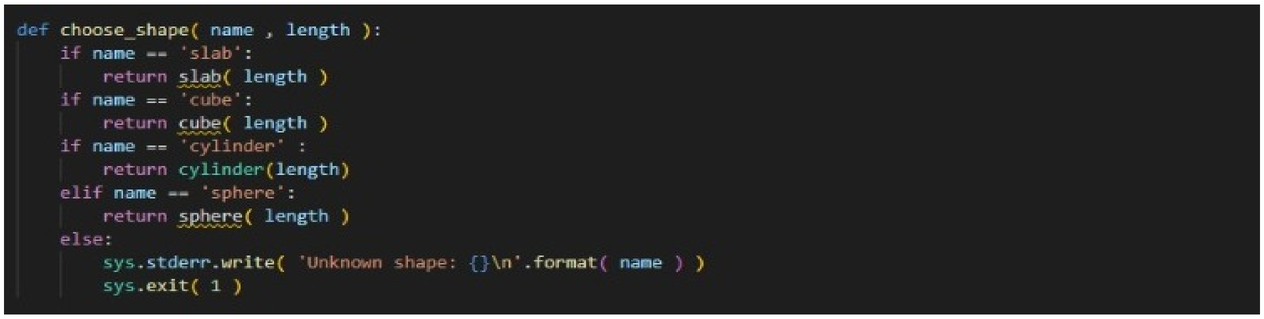
```
def sample(self, r):
    # calculate the distance from the center of the cylinder
    x, y, z = r
    dist = numpy.sqrt(x**2 + y**2)
    #if the point is inside the cylinder, return a unit vector in the
    #direction of the point, otherwise return None
    if dist <= self.length and -self.length/2 <= z <= self.length/2:
        return numpy.array([x, y, z])/dist
    else:
        return None
```

- This function uses the random point() function and creates 100,000 random points inside the cylinder and uses a 2D histogram to plot the density of those points in the x-y plane. The density flag is set to True and the histogram's number of bins is set to 100 in order to display the density of points rather than just their number. A colorbar and the axis labels for the final plot are also displayed.



```
    def plot_density(self, figure_index):
        ## generate random points within the cylinder
        points = [self.random_point() for _ in range(100000)]
        x, y, z = numpy.array(points).T
        ##plot the density of points in the x-y plane
        plt.figure(figure_index)
        plt.hist2d(x, y, bins=100, density=True)
        plt.xlabel("x")
        plt.ylabel("y")
        plt.title("Density plot of points inside cylinder in x-y plane")
        plt.colorbar()
        plt.show()
```

- This function called choose_shape that takes two arguments: name and length. Based on the value of the name input, the function selects a geometric form and returns it with the specified length. Based on the value of the name input, the function utilizes an if-elif-else statement to decide which shape to return. The function performs the slab function with the length parameter and returns the result if name is equal to "slab." The function executes the cube function with the length parameter and returns the result if name is equivalent to "cube." The function performs the cylinder function with the length parameter and returns the result if name is equal to "cylinder." The function executes the sphere function with the length parameter and returns the result if name is equivalent to "sphere."



```
def choose_shape( name , length ):
    if name == 'slab':
        return slab( length )
    if name == 'cube':
        return cube( length )
    if name == 'cylinder' :
        return cylinder(length)
    elif name == 'sphere':
        return sphere( length )
    else:
        sys.stderr.write( 'Unknown shape: {}\n'.format( name ) )
        sys.exit( 1 )
```

2.2 Results

- The growth rate of neutron is plotted against the number of trajectories. This plot shows an initial rise in the growth rate which falls after a sufficient amount of trajectories which is anticipated due to the transient effect caused to flush the initial set of neutrons completely until a new set of neutrons emerge in the queue. After, the queue is replaced with the new set of neutrons. The fluctuation in the growth rate of neutrons is close to zero, which physically means that an equilibrium is established and the criticality condition is identified with a suitable amount of accuracy.

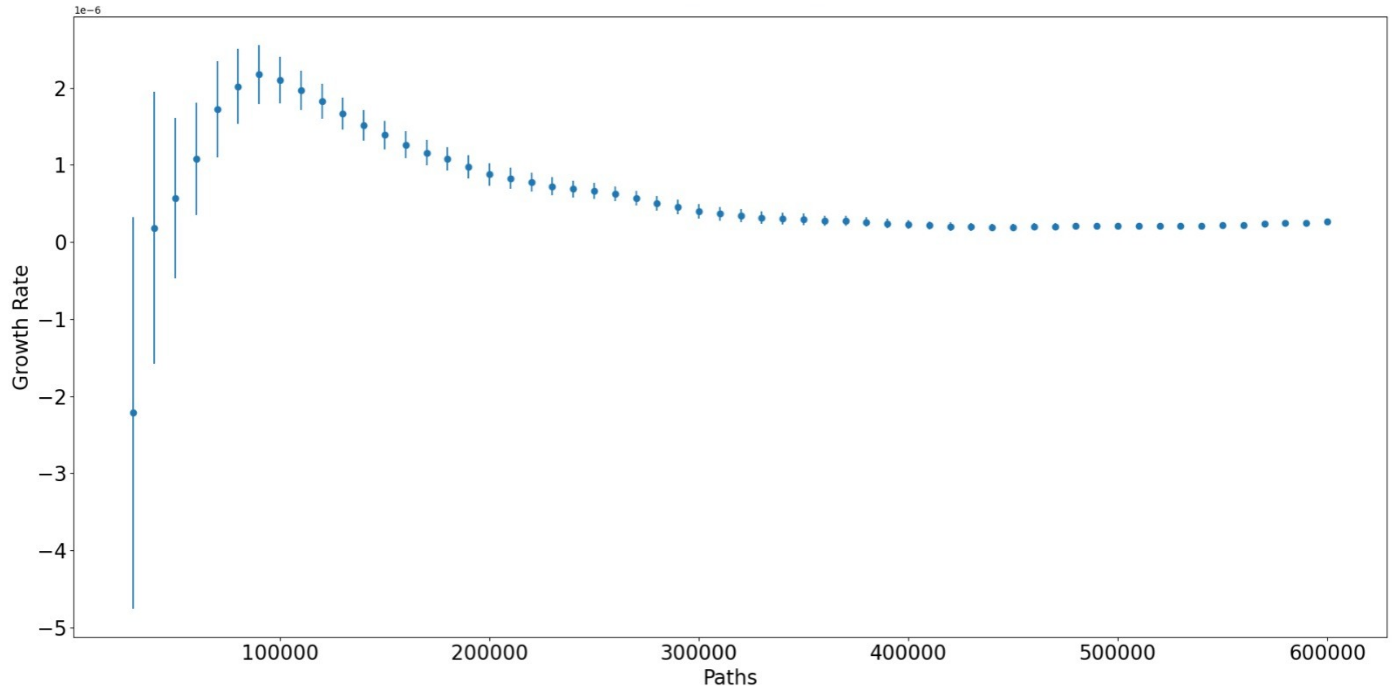


Figure 1: Growth Rate vs Path

- A 2D density plot, also known as a heatmap, is a graphical representation of the density of data points in a 2D space. Data distribution in a bivariate space can be displayed using this practical visualization tool. The plot uses color to depict the density of sample points in a 2D space, with warmer colors indicating higher density and cooler colors indicating lower density. A 2D density map can be an effective tool for visualizing and explaining the distribution of data in a 2D space, as well as for exploratory data analysis.

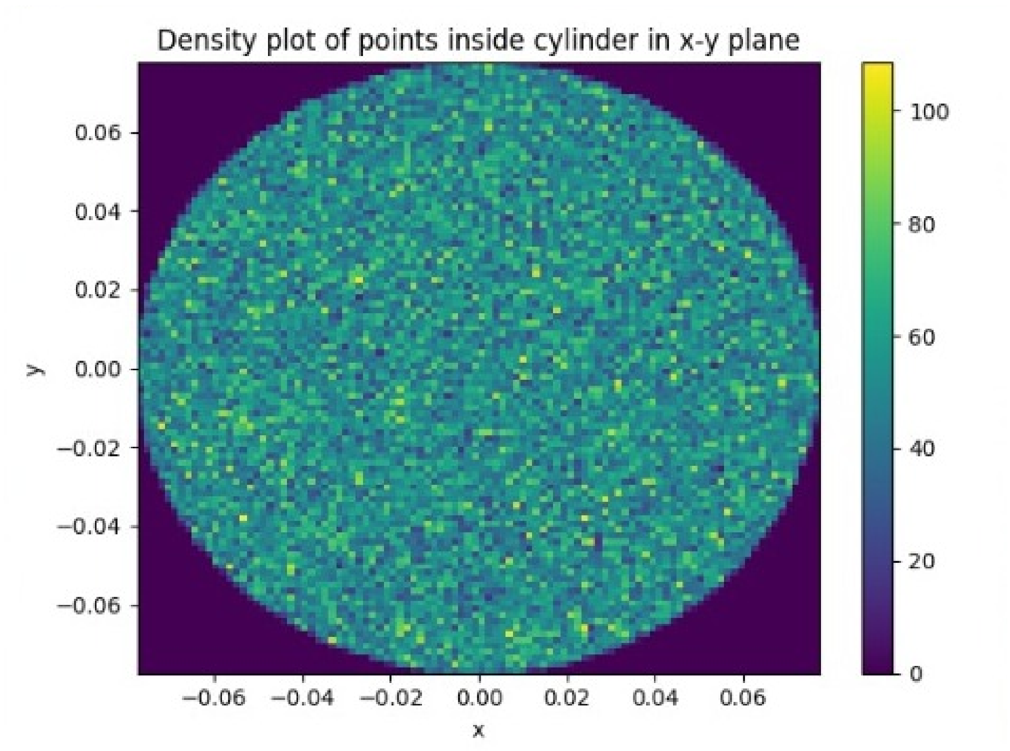


Figure 2: Density Plot

- Queue Length vs Number of Trajectories is plotted and the queue length is seen to grow as the amount of trajectories increases. This is due to the fact that fission-related neutrons may be added to the line of neutrons that are waiting to be carried through the cylinder in a neutron transport simulation. The queue will need to keep track of all of the particle's data as the simulation goes on and more of them are generated before they can be processed. So once more, the length of the queue will logically grow as more trajectories are added.

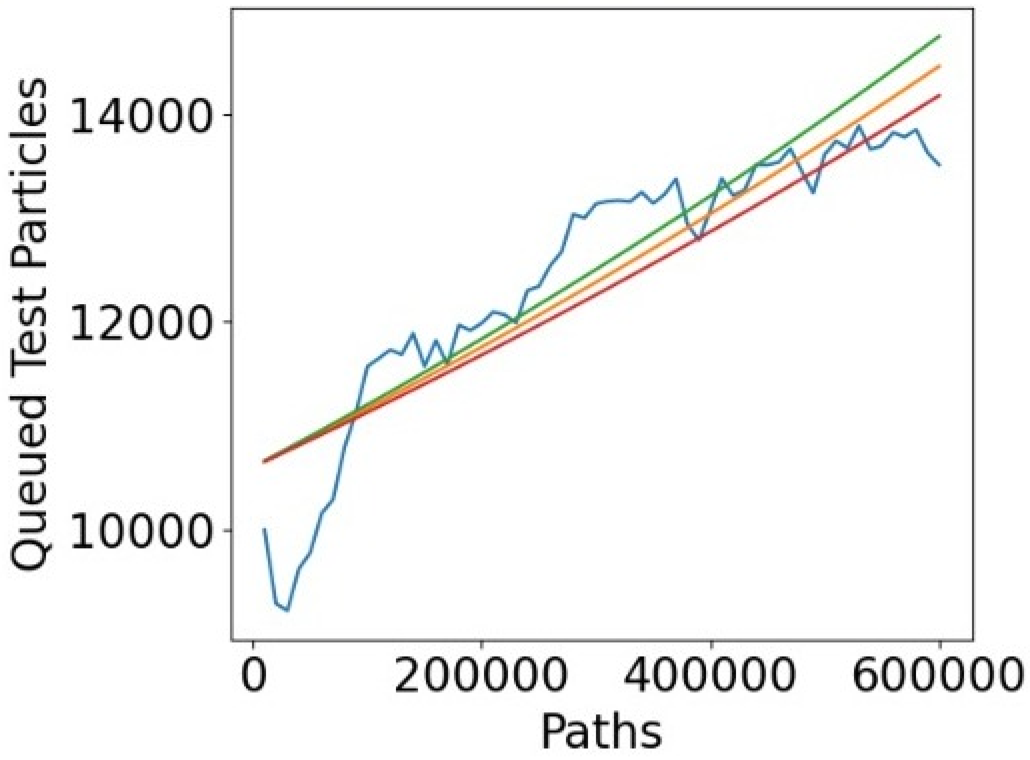


Figure 3: Queue Test Particles vs Paths