Section5

Elliott Ashby

November 24, 2023

1 Write a program to model the one-dimensional Uranium.

```
1
2  def one():
3      def neutron(L):
4          loc = L * random.random()
5          num = 2
6          return (loc, num)
7
8      def chain(loc, num, L):
9          R = sqrt(2 * 0.017 * 0.21)
10         newloc = []
11         temp = []
12         for i in range(num):
13             direc = bool(getrandbits(1))
14             if direc:
15                 newloc.append(loc + R)
16             else:
17                 newloc.append(loc - R)
18         for i in range(len(newloc)):
19             if newloc[i] < L and newloc[i] > 0:
20                 pass
21             else:
22                 temp.append(newloc[i])
23         for i in temp:
24             del newloc[newloc.index(i)]
25         return len(newloc)
26
27     L = 1
28     sims = 100
29     count = 0
30     for i in range(sims):
31         count += chain(*neutron(L), L=L)
```

This code has 2 functions, neutron which returns a location between 0 and L and the number of nuetrons from the fission (in this case 2). And the other function is chain, which takes a location, number of neutrons and L. It then

randomly determines which direction the neutrons travel and excludes them if they are outside the range of 0 to L. It then returns the number of total nuetrons that stay within the range.

2 Create an updated version of your program using the neutrons() function.

```
1
2  def two():
3      def neutron(L):
4          loc = L * random.random()
5          num = neutrons.neutrons()
```

Here, we simply change the number of neutrons to be set by the neutrons.neutrons() function and not a static 2.

3 Start with L = 0.1m and vary this to determine the critical value. Decide whether you need to use more than 100 initial neutrons

```
1
2      L = 0.1
3      initfissions = 100
4      avcount = 0
5      while avcount < 100:
6          temp = 0
7          for i in range(1000):
8              temp1 = 0
9              for j in range(initfissions):
10                 temp1 += chain(*neutron(L), L=L)
11             temp += temp1
12         avcount = temp / 1000
13         L += 0.001
```

Here we update neutron to include the function neutrons which returns a random number with average of 2.5 and make a new while loop in order to determine the critical value. It does this by taking an average count of secondary fissions of 1000 fissions each with 100 initial fissions. Once the average is larger than the inital fissions, it means more secondary fissions occur than initial fissions. Using this we get a critical value:

$$L_{criticalvalue} \approx 0.142 \pm 0.0005$$

Using more than 100 inital neutrons simply gets a more precision average allowing for a more more precise answer... if the small increase in L allows. In our case increasing the initial fissions doesnt increase the precision of $L_{criticalvalue}$.

4 Update your program for the 3-dimensional model.

In order to implement 3 dimensions we need to modify both our functions but not our main.

```
1  def neutron(L):
2      # array of random coords from 0 to L
3      loc = [L * random.random(), L * random.random(), L *
           ↪ random.random()]
4      # random number with average of 2
5      num = neutrons.neutrons()
6      return (loc, num)
7
8
9  def chain(loc, num, L):
10     R = sqrt(2 * 0.017 * 0.21)
11     newloc = []
12     temp = []
13     for i in range(num):
14         # generate random direction vector
15         direc = [R * neutrons.diffusion() * sin(acos(2.0
               ↪ * random.random() - 1.0)) * cos(2.0 * pi *
               ↪ random.random()),
16                  R * neutrons.diffusion() * sin(acos(2.0
                        ↪ * random.random() - 1.0)) * sin
                        ↪ (2.0 * pi * random.random()),
17                  R * neutrons.diffusion() * cos(acos(2.0
                        ↪ * random.random() - 1.0))]
18         # add random vector to the original location
19         newloc.append([direc[i] + loc[i] for i in range(
               ↪ len(loc))])
20     # if any of the locations of neutrons are outside of
           ↪ the cube, delete them from the array
21     for i in range(len(newloc)):
22         if newloc[i][0] < L and newloc[i][0] > 0 and
               ↪ newloc[i][1] < L and newloc[i][1] > 0 \
23                 and newloc[i][2] < L and newloc[i][2] >
                       ↪ 0:
24             pass
25         else:
26             temp.append(newloc[i])
27     for i in temp:
28         del newloc[newloc.index(i)]
29     # return a count of how many neutrons are still in
           ↪ the cube (hence generate new fission reactions)
30     return len(newloc)
31
32
33 def sampled(L, initfissions=100):
```

```
34      temp = 0
35      for i in range(1000):
36          temp1 = 0
37          for j in range(initfissions):
38              temp1 += chain(*neutron(L), L=L)
39          temp += temp1
40      return (temp / 1000, L)
41
42
43  def four():
44      L = 0.1
45      initfissions = 100
46      p = Pool()
47      temp = p.map(sampled, [L + 0.001 * i for i in range
            ↪ (200)])
48      p.close()
49      closest = min(temp, key=lambda x: abs(x[0] -
            ↪ initfissions))
50      print(f"Critical value in a three dimensional cube is
            ↪ {closest[1]} with {closest[0]} neutrons")
```

Since python is dynamically typed we can simply change out loc to a list instead of a float. And in chain we can randomly determine a vector as a list and add it to the the location passed into the function.
We also now need to check whether the new value is within a 3 dimensional range. We now check all dimensions are within 0 to L.
Since we are in 3 dimensions now, our computation time is much longer. Because of this I implemented rudimentary multiprocessing of the sampling function so that multiple lengths of L can be checked at once, and then the closest value of the outputted remaining neutrons to the initial value is assigned to closest to be printed.

5 Vary the value of L and find the critical value and hence the critical mass.

Using the above code we can determine a critical value and hence the critical mass, using an increment of 0.001 and initial fissions of 100:

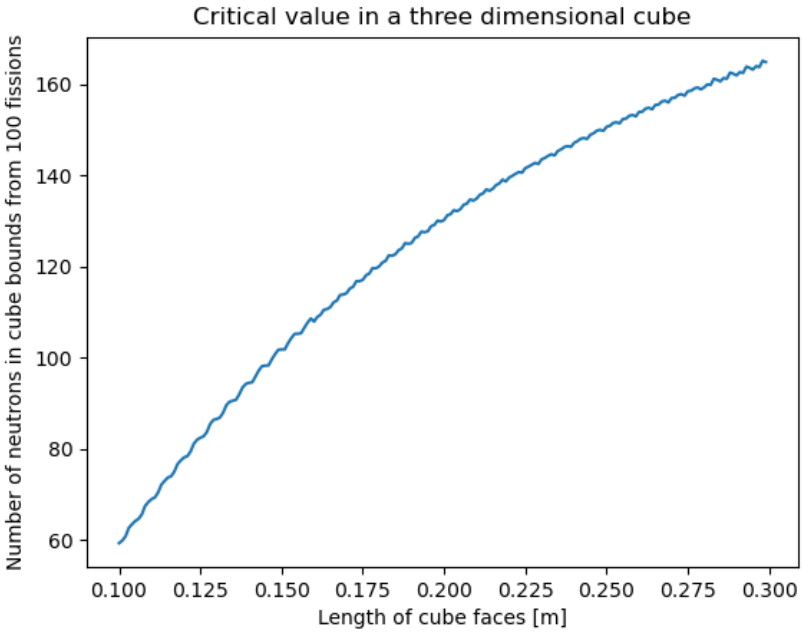
$$L_{criticalvalue} \approx 0.148 \pm 0.0005$$

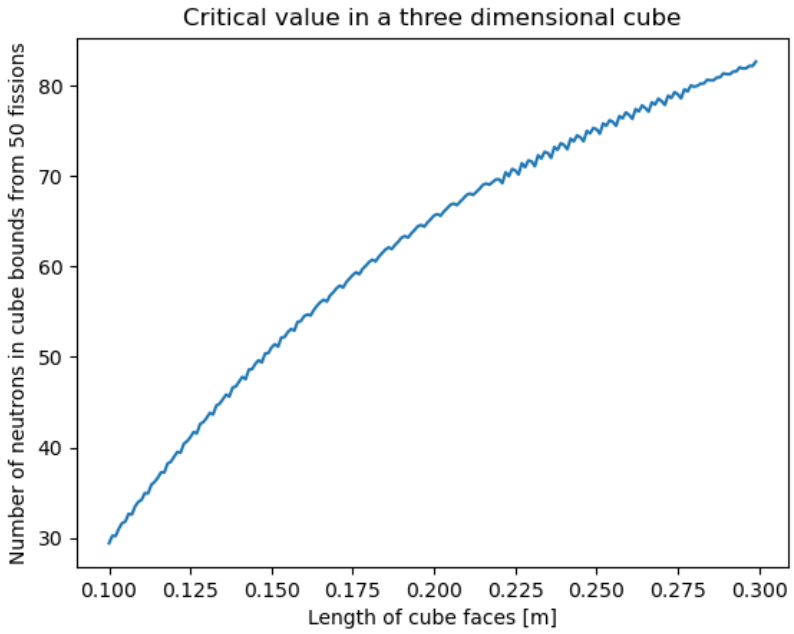
$$m_{critical} = L^3_{criticalvalue} \times \rho_{Uranium}$$

Using $\rho_{Uranium} = 18.7\text{Mgm}^{-3}$:

$$m_{critical} \approx 60.6\text{kg}$$

The expected value for the critical mass of uranium is 52kg, so our value is slightly higher. This is likely due to the fact we are using a cube instead of a sphere, meaning that the neutrons in the corner regions would not initiate a chain reaction.





From the above graphs we can see that the critical mass does not change with a change in initial fissions → this is obviously expected from a physical standpoint and it is good that the code reflects this.