NUCLEAR REACTORS FOR DUMMIES

Jonathan Cheng, Holden Kowitt, Finnegan Wright

WHAT IS OUR GOAL?

Modeling a nuclear reactor using a monte-carlo simulation

→ Repeating random results to find a statistical output

→ No time dependence: actions happen in “ticks”

ASSUMPTIONS

1. Reactive (and non-reactive) elements are in a homogenous mixture
2. Spherical reactor
3. Neutrons released from fissions are not simulated
    a. This event would cause a never-ending cycle of reactions because fuel is assumed to be unchanging in quantity

INPUTS

01 REACTOR SIZE

02 NUMBER OF NEUTRONS

Each given a position, “velocity,” and energy

03 CROSS-SECTIONS

What is this?

```
reactorradius = 1  # cm

count = 3  # neutrons

# event counters
fcount = 0      # fission
acount = 0      # absorption
scount = 0      # scattering
ecount = 0      # escape

# mixture percentages (of total volume)
u238 = 0.25  # uranium-238
u235 = 0.25  # uranium-235
boron = 0.25  # boron-10
hw = 0.25  # heavy water
```

material	fast fission	fast absorbtion	fast scatteing	slow fission	slow absorbtion	slow scatteing	number density
uranium238	0.013	2.400	4.000	0.000	4.000	10.000	4.821
uranium235	1.100	1.900	2.900	584.000	97.000	9.000	4.820
boron10	0.000	0.250	1.000	0.000	3840.000	2.200	14.091
heavywater	0.000	0.001	10.000	0.000	0.009	10.600	3.338

03

NUCLEAR CROSS-SECTION

Fancy-speak for the probability that a nuclear event will occur

CROSS-SECTIONS ARE AREAS

They are measured in barns (10^-28 m^2)

The probability for an interaction depends on the cross-section and the number density of the material

```
import numpy as np
from scipy.integrate import quad
from pynverse import inversefunc
```

```
# for a particle density of "n" and a cross section total of "stot", the free path pdf is given by: ns * e^-xns
def pdf(x, n, stot):
    return (n * stot) * np.exp(-x * (n * stot))

# integrate the pdf to get a cdf that goes from 0<y<1
def cdf(x, n, stot):
    return quad(pdf, 0, x, args=(n,stot))[0]

# pick a random number from 0-1 to use as the y value for the cdf
# gives the distance the particle will travel
def inv(y, n, stot):
    temp = inversefunc(cdf, args=(n, stot))
    return temp(y)
```

WHAT EVENTS ARE POSSIBLE AND HOW ARE THEY PROCESSED?

Fission: The neutron collides with a reactive element with sufficient energy

→ The fission event is counted and the neutron is removed from the simulation

Escape: The distance required for the neutron to react (probabilistically), is less than the distance to the edge of the reactor

→ The escape event is counted and the neutron is removed from the simulation

WHAT EVENTS ARE POSSIBLE AND HOW ARE THEY PROCESSED?

Absorption (Inelastic Collision): The neutron collides with an atom and sticks

→ The absorption event is counted and the neutron is removed from the simulation (noticing a pattern?)

Scattering (Elastic Collision): The neutron bounces off an atom and loses energy if it bounces off a light nucleus

→ The scattering event is counted and the neutron is run through the simulation recursively

SIMULATION OUTPUTS

Number of each event: What does it tell us?

Fissions:
Amount of energy released / productivity of reactor

Absorptions / Scatterings:
Effects of control (boron) and moderator (heavy water) quantities on reaction rate / output

OUR CODE

1. INITIAL CONDITIONS

Reactor Radius

Cross-Sections

Number of Neutrons

2. NEUTRONS "SPAWNED"

Position

Velocity (Direction of Travel)

Energy Level

3. TIME "TICKED"

Neutrons are moved a distance until they (probabilistically) collide with something in the direction of "velocity"

4. CHECK (AGAIN)

Check if the distance of first collision is within the reactor

Determine which event occurs and repeat from (3) if necessary

```
def escape(neutron, count, reactorradius, f_sa, f_ss, s_sa, s_ss, f_stot, s_stot, n):

    # allow global event counters to update
    global fcount
    global acount
    global scount
    global ecount

    dist = np.arange(0,count)

    dist = np.array(list((map(lambda x:inv((np.random.uniform(0, 1)), n, f_stot) if neutron[2,0,int(x)] == 1 else inv((np.random.uniform(0, 1)), n, s_stot), dist))))    # distance travelled before interaction
    # check if particle escapes

    neutron[0] = np.add(neutron[0], dist * neutron[1])  # change neutron position by dist in movement direction
    rad = np.linalg.norm(neutron[0], axis=0)  # find final position

    newneutrons = np.zeros((3, 3, count))

    for i in range(count):
        if neutron[2,0,i] == 1:
            if rad[i] <= reactorradius:  # particle didn't leave
                prob = np.random.uniform(0, f_stot * (10**24))
                if prob < f_ss * (10**24):  # probability of elastic collision (need to account for neutron)
                    newneutrons[:, :, i] = neutron[:, :, i]
                    tempprob = np.random.uniform(0, np.sum(prpl_cs[:,2]))
                    if tempprob < (prpl_cs[:,2][1]+prpl_cs[:,2][0]):
                        newneutrons[:,:,i][2,0] = 0
                    scount += 1
                elif prob < (f_ss + f_sa) * (10**24):  # probability of inelastic collision (neutron `gone`)
                    acount += 1
                else: # fission occurs (neutron `gone`)
                    fcount += 1
            else:  # particle left reactor (neutron `gone`)
                ecount += 1
        else:
            if rad[i] <= reactorradius:  # particle didn't leave
                prob = np.random.uniform(0, s_stot * (10**24))
                if prob < s_ss * (10**24):  # probability of elastic collision (need to account for neutron)
                    newneutrons[:, :, i] = neutron[:, :, i]
                    scount += 1
                elif prob < (s_ss + s_sa) * (10**24):  # probability of inelastic collision (neutron `gone`)
                    acount += 1
                else: # fission occurs (neutron `gone`)
                    fcount += 1
            else:  # particle left reactor (neutron `gone`)
                ecount += 1

    # re-run function for remaining neutrons from elastic collisions
    for i in range(np.size(newneutrons,axis=2)+1):
        try :
            while np.all(newneutrons[:,:,i]==0):
                newneutrons = np.delete(newneutrons, i, axis=2)
        except:
            i +=1
    newcount = np.size(newneutrons,axis=2)
    if newcount != 0:
        escape(newneutrons, newcount, reactorradius, sa, ss, stot)
```

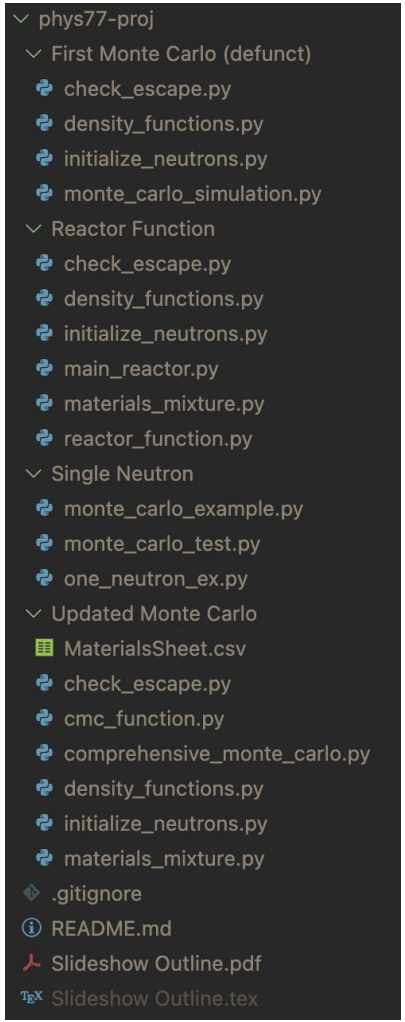
TAKE 1: SINGLE NEUTRON

Simulation of a single neutron to test functions and procedures

Issues:

Position and direction had 3 variables each

Neutron did not always spawn inside of reactor



TAKE 2: FIRST MONTE-CARLO

Position and direction were turned into a (2, 3) array

Functions looped for each neutron

Issues:

Neutron spawning (still)

Probability density functions were not efficient to use multiple times

Loops were very slow for large ‘n’

```python
def neutrons(count, reactorradius, nenergy): # nenergy is array-like w/ length `count`
    neutron = np.random.uniform(low=0, high=1.0, size=(3, 3, count))

    #   each slice is a single neutron with position, unit direction, and energy
    #   x           y           z
    #   xdir        ydir        zdir
    #   nenergy     n/a         n/a
```

SPEED

Neutrons processed in a 3d array (fewer loops

SOLVING ISSUES

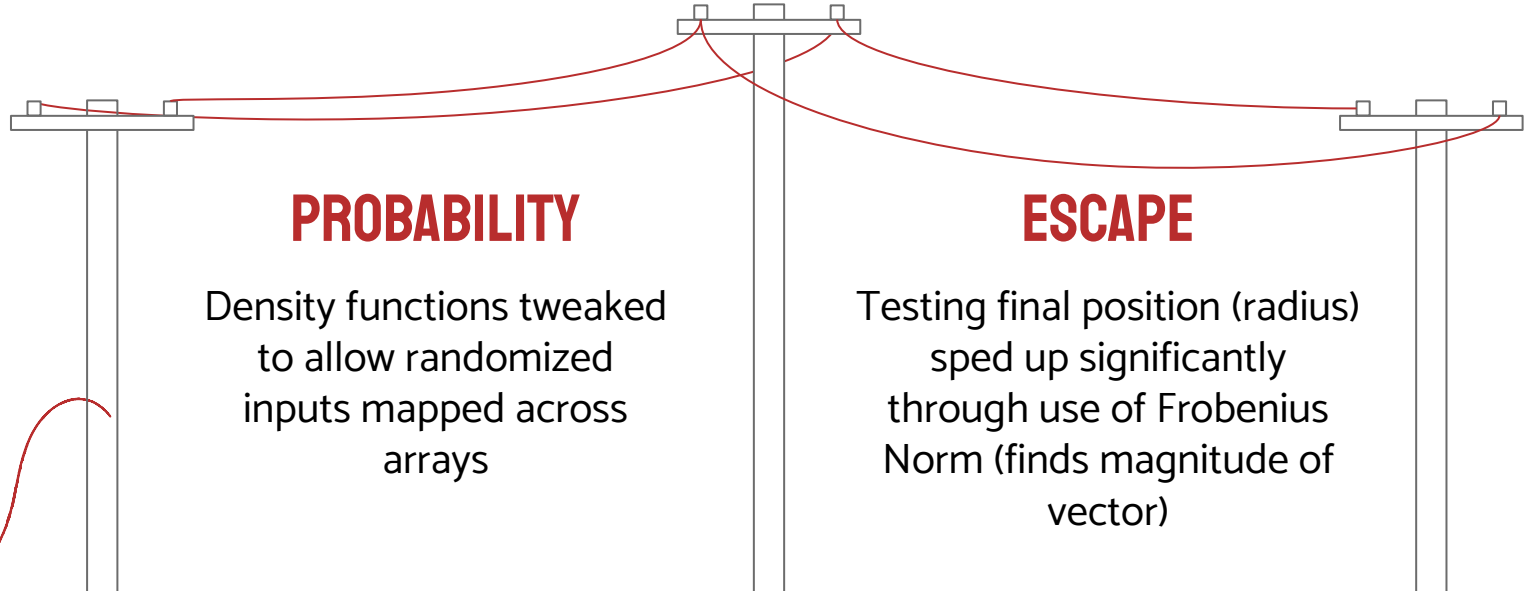
POSITION

Measure randomness in spherical rather than cartesian coordinates

```python
    x, y, z = neutron[1] # component vectors
    len = np.empty(count) # length of direction vector
    for i in range(count):
        len[i] = (np.sum(x[i] ** 2 + y[i] ** 2 + z[i] ** 2)) ** 0.5 # pythagorean theorem
    neutron[1] = neutron[1] / len  # write unit direction vector as x, y, z components

    neutron[2, 0] = nenergy # set neutron energy

    # define position in spherical coordinates (random 0-1 always within the sphere of reactor)
    rho = neutron[0, 0] * reactorradius
    phi = neutron[0, 1] * np.pi
    theta = neutron[0, 2] * 2 * np.pi

    sinphi = np.sin(phi)
    cosphi = np.cos(phi)
    sint = np.sin(theta)
    cost = np.cos(theta)

    # convert to cartesian coordinates
    neutron[0] = [rho * sinphi * cost, rho * sinphi * sint, rho * cosphi]
```

```
    dist = np.arange(0,count)

    dist = np.array(list((map(lambda x:inv((np.random.uniform(0, 1)), n, f_stot) if neutron[2,0,int(x)] == 1 else inv((np.random.uniform(0, 1)), n,
    s_stot), dist))))    # distance travelled before interaction
    # check if particle escapes

    neutron[0] = np.add(neutron[0], dist * neutron[1])  # change neutron position by dist in movement direction
    rad = np.linalg.norm(neutron[0], axis=0)  # find final position

    newneutrons = np.zeros((3, 3, count))
```

SOLVING ISSUES



PROBABILITY

Density functions tweaked to allow randomized inputs mapped across arrays

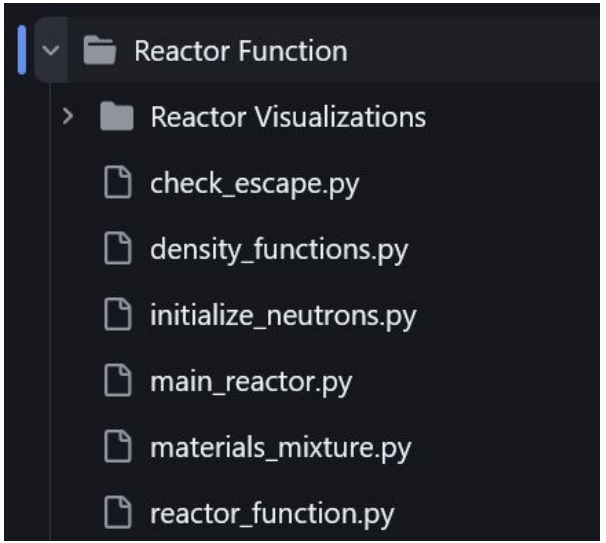
ESCAPE

Testing final position (radius) sped up significantly through use of Frobenius Norm (finds magnitude of vector)

CONCLUSIONS FOR REACTOR BASE

1. Reactors involving mixed materials

2. Reactor function and putting everything together



MATERIALS_MIXTURE

Rather than just using the data for one material, we wanted to be able to the the interactions with a mixed reactor, allow us to account for enrichment, moderators, and control rods.

```python
import numpy as np
import pandas as pd

# unpack data from csv
materials = pd.read_csv('MaterialsSheet.csv')

numden = materials['numden'].to_numpy()    # extract specifically the numerical de
materials = materials.iloc[0:, 1:7].to_numpy()    # extract the rest of the data f

# establish function to be used in comprehensive_monte_carlo
def mixture(u238, u235, boron, hw):    # input volume percentages for each materia
    densities = np.array([u238, u235, boron, hw])    # turn inputs into an array

    # process the volume densities for each material into weighted proportions of
    eff_den = (numden.T * densities).T
    weigh_den = eff_den/sum(eff_den)

    # multiply each row of cross-sections from the csv by their weighted proportio
    prpl_cs = (materials.T * weigh_den).T
    total_cs = np.array([sum(i) for i in zip(*prpl_cs)])    # sum each column for

    # unpack total_cs list into fast & slow cm^2 fission, absorption, and scatteri
    f_sf, f_sa, f_ss, s_sf, s_sa, s_ss = total_cs * (10 ** (-24))    # although un

    f_stot = f_sf + f_sa + f_ss    # total fast cross-section
    s_stot = s_sf + s_sa + s_ss    # total slow cross-section
    n_per = eff_den * (10**22)    # atoms/cm^3 for each material
    n = np.sum(n_per)    # total atoms/cm^3

    return prpl_cs, f_sf, f_sa, f_ss, s_sf, s_sa, s_ss, f_stot, s_stot, n_per, n
```

REACTOR_FUNCTION

Put all the separate pieces together into a shared module

- Interesting issue: global variables
- Optimization extension

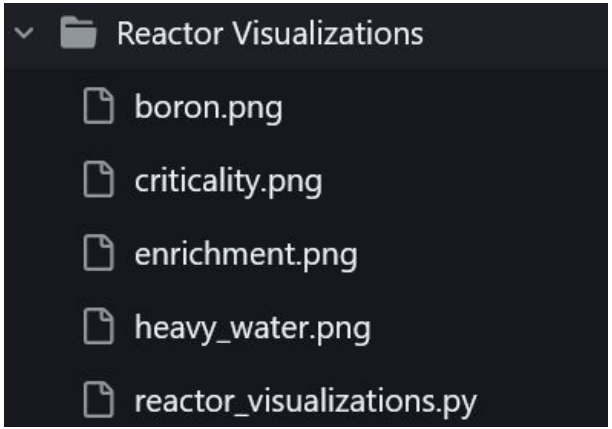
```
In [3]: reactor(1000, 0.8, 0.2, 0, 0)
Out[3]: (97, 97, 138, 806)
```

```python
# second function with purpose of running the first function, but reseting all the global counts first t
def baseline_escape(neutron, count, reactorradius, f_sa, f_ss, s_sa, s_ss, f_stot, s_stot, prpl_cs, n):

    # ties function into global counts
    global fcount
    global acount
    global scount
    global ecount

    # resets event counters
    fcount = 0      # fission
    acount = 0      # absorption
    scount = 0      # scattering
    ecount = 0      # escape

    # runs escape function and returns outputs
    fcount, acount, scount, ecount = \
        escape(neutron, count, reactorradius, f_sa, f_ss, s_sa, s_ss, f_stot, s_stot, prpl_cs, n)

    return fcount, acount, scount, ecount
```

```python
def reactor(count, u238, u235, boron, hw, reactorradius = 1):

    # initiates overall mixture
    prpl_cs, f_sf, f_sa, f_ss, s_sf, s_sa, s_ss, f_stot, s_stot, n_per, n = mixture(u238, u235, boron, hw)
    # outputs all needed atom totals and cross-sections

    nenergy_initial = np.ones(count)

    neutron = neutrons(count, reactorradius, nenergy_initial) # initialize neutrons for monte carlo

    # runs baseline_escape function with inputs, giving output counts returned below
    fcount, acount, scount, ecount = \
        baseline_escape(neutron, count, reactorradius, f_sa, f_ss, s_sa, s_ss, f_stot, s_stot, prpl_cs, n) \

    return fcount, acount, scount, ecount
```

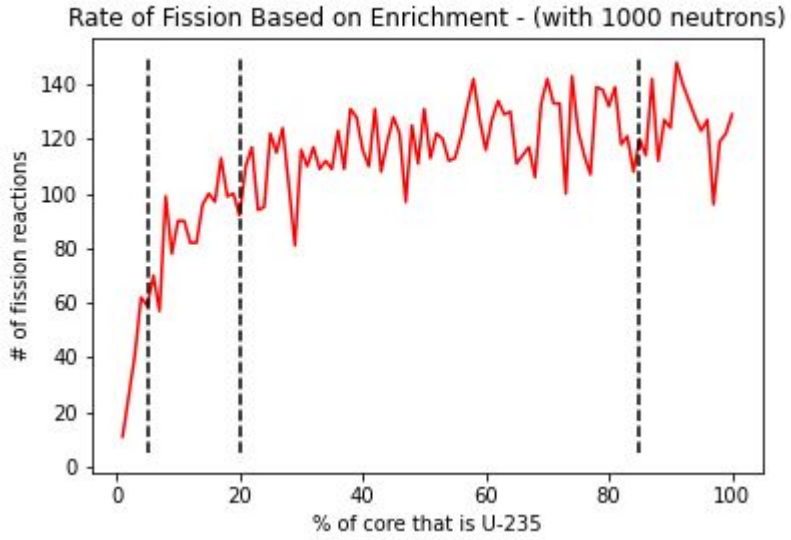
EXTENSIONS

1. Graphical representations
2. Antineutrinos
3. Criticality and other extensions



ENRICHMENT VISUALIZATION

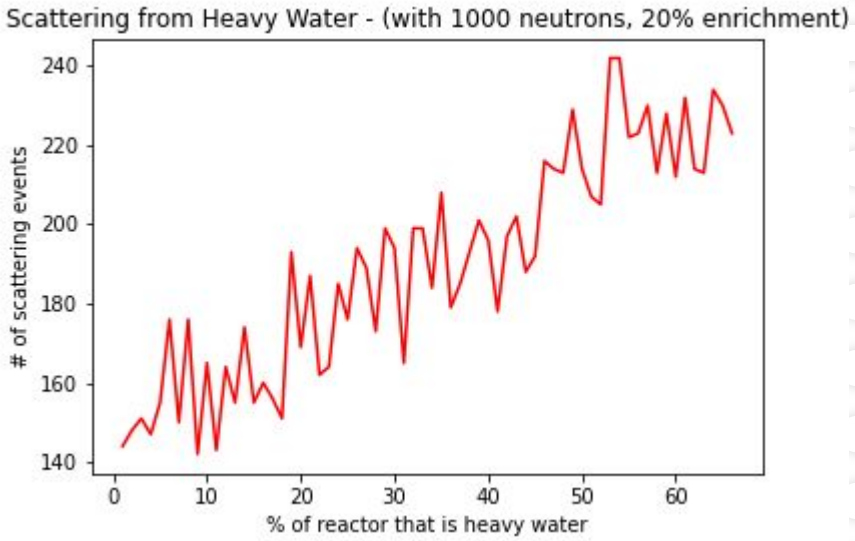
```python
def enrichment(n):

    power = []

    for i in range(1, 101):
        fcount, acount, scount, ecount = reactor(n, 1-0.01*i, 0.01*i, 0, 0)
        power.append(fcount)

    X = np.arange(1,101)

    plt.plot(X, power, color = 'red')
    plt.vlines([5, 20, 85], 5, 150, color = 'black', linestyle = '--')
    plt.title('Rate of Fission Based on Enrichment - (with ' + str(n) + ' neutrons)')
    plt.xlabel('% of core that is U-235')
    plt.ylabel('# of fission reactions')

    plt.savefig('enrichment.png')
    plt.show()

```



Clearly demonstrates the importance of different levels of enrichment: Reactor vs. weapons-grade uranium

HEAVY WATER VISUALIZATION

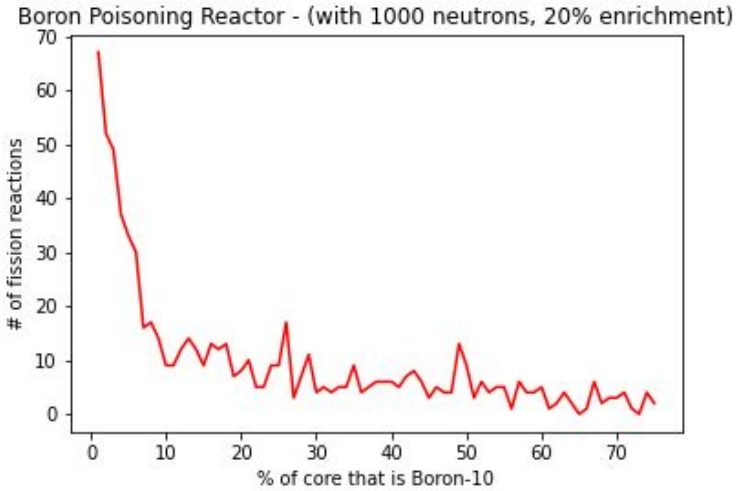
```
24  def heavy_water(n):
25
26      events = []
27
28      for i in range(1, 67):
29          fcount, acount, scount, ecount = reactor(n, 0.8*(1-0.01*i), 0.2*(1-0.01*i), 0, 0.01*i)
30          events.append(scount)
31
32      X = np.arange(1,67)
33
34      plt.plot(X, events, color = 'red')
35      plt.title('Scattering from Heavy Water - (with ' + str(n) + ' neutrons, 20% enrichment)')
36      plt.xlabel('% of reactor that is heavy water')
37      plt.ylabel('# of scattering events')
38
39      plt.savefig('heavy_water.png')
40      plt.show()
```



Key insights about temperature, and interesting impacts on scattering

BORON VISUALIZATION

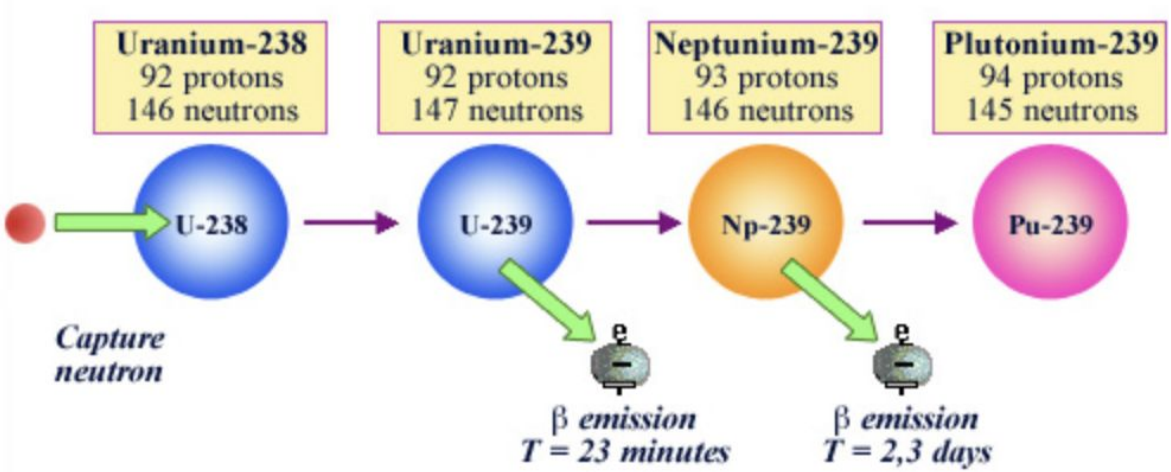
```
42  def boron(n):
43
44      power = []
45
46      for i in range(1, 76):
47          fcount, acount, scount, ecount = reactor(n, 0.8*(1-0.01*i), 0.2*(1-0.01*i), 0.01*i, 0)
48          power.append(fcount)
49
50      X = np.arange(1,76)
51
52      plt.plot(X, power, color = 'red')
53      plt.title('Boron Poisoning Reactor - (with ' + str(n) + ' neutrons, 20% enrichment)')
54      plt.xlabel('% of core that is Boron-10')
55      plt.ylabel('# of fission reactions')
56
57      plt.savefig('boron.png')
58      plt.show()
```



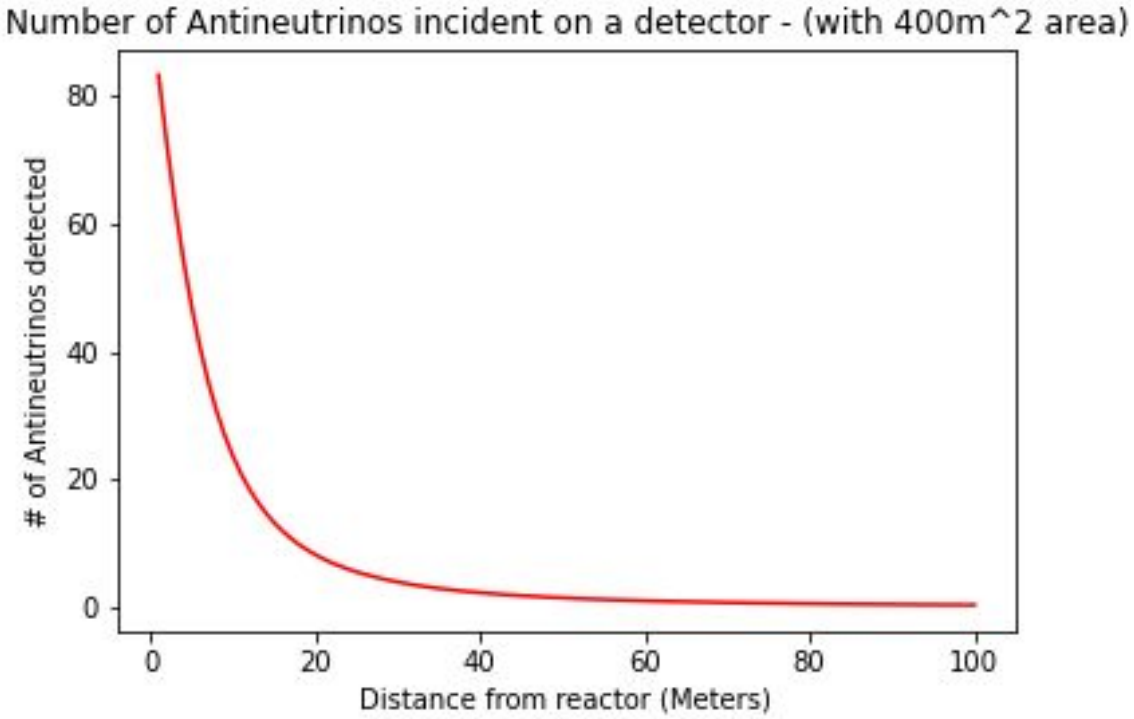
Poisoning the reactor!

ANTINEUTRINOS

Fission products and uranium that absorbs a neutron will undergo beta decays, which release antineutrinos



For a simulation with 1000 initial neutrons



CRITICALITY AND OTHER EXTENSIONS

Criticality

- Subcritical, critical and supercritical
- Seemingly not simulating a critical reactor
- Clearly greater complexity involved; however we were able to demonstrate various important principles

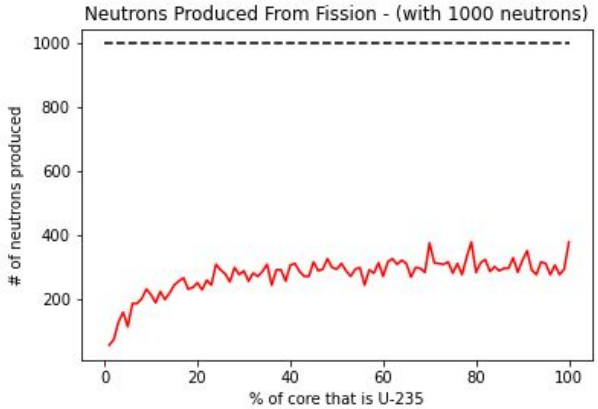
Temperature

- Heavy water vs. water
- Changing cross-sections w/ temp

Geometry

- Shaping reactor beyond sphere
- Control rods vs control substance
- Water shell

```
In [4]: reactor(1000, 0.8, 0.2, 0, 0)
Out[4]: (121, 115, 156, 764)

In [5]: 121/1000
Out[5]: 0.121

In [6]: reactor(10000, 0.8, 0.2, 0, 0)
Out[6]: (1070, 1056, 1468, 7874)

In [7]: reactor(100000, 0.8, 0.2, 0, 0)
Out[7]: (10494, 10160, 14395, 79346)
```



THANK YOU!