The Journal of Advanced Undergraduate Physics Laboratory Investigations, JAUPLI-B

Volume 3 Article 2

2014

Developing a Monte Carlo Simulation for Time- Series Analysis of Actinium-225 Decay

Victoria Wood
Agnes Scott College

Developing a Monte Carlo Simulation for Time-Series Analysis of ^{225}Ac Decay

April 19, 2016

1 Introduction

The goal of this simulation is to develop a new method by which one may determine the equilibrium ratios of isotopes in a single radioactive decay chain over time. In a radioactive decay chain, it is most common for isotopes to be produced at a different rate than they are decaying. Radioactive equilibrium occurs when the rate at which an isotope is being produced (the parent activity) is the same as the rate at which the isotope is decaying (the daughter activity) [4]. Studying the parent-to-daughter ratios at radioactive equilibrium has applications in several areas of physics. For example, in medical physics, radioactive substances are used for imaging purposes, as many radioactive decays produce photons [5]. In order to draw conclusions from the detected light at any point in time, understanding the long-term behavior of the decay and isotope ratios is crucial. Equilibrium can also be used in radioactive dating techniques. Naturally occurring isotopes, that should be in equilibrium but are not, were separated from the original substance by physical or chemical processes; the age of these fragmented isotopes can be determined by how much the isotope has restored equilibrium.

There are a few methods that can be used to determine the equilibrium ratios of isotopes in a decay chain. This can be accomplished mathematically by solving a system of first-order linear differential equations. However, to understand ratios between isotopes that are several elements down the decay chain, the derivation of the equation can be cumbersome and take a substantial amount of time. In addition, the resultant equation may be such that it is difficult to ascertain information about the long term behavior of the decay. I sought to instead develop a script using the programming language Python

that would simulate radioactive decay over time using Monte Carlo methods, taking advantage of the probabilistic nature of radioactive decay.

2 Deriving the Equilibrium Ratios of Radioactive Isotopes Mathematically

The long term behavior of the radioactive decay can be mathematically derived by solving a system of first-order linear differential equations. A differential equation is an equation that relates a function to it's derivatives [1].The change in amount of a daughter isotope at a given time is dependent on the amount of the parent isotope at that same time. Therefore, the function describing the rate of change in the daughter isotope is a differential equation dependent on the function describing the parent isotope.

Isotopes decay at a rate directly proportional to the amount present,

$$\frac{dP(t)}{dt} = -aP(t), \tag{1}$$

where $P(t)$ refers to the amount of isotope over time, t, and a refers to a proportionality constant. Equation 1 is an example of a differential equation. Solving this differential equation for $P(t)$,

$$P(t) = P(0)e^{-at}, \tag{2}$$

where $P(0)$ refers to the initial amount of the isotope. The proportionality constant a can be determined by mathematically interpreting the half-life of radioactive isotopes. All radioactive isotopes have a quantitative property known as a half-life, τ, such that

$$P(n\tau) = .5^n P(0), \tag{3}$$

where n is a positive integer. By substituting τ and $.5P(0)$ for t and $P(t)$ in Equation 2, the constant a is equal to

$$a = \frac{\ln(2)}{\tau}. \tag{4}$$

This gives a final form for the function describing the present amount of a radioactive isotope at a given time,

$$P(t) = P(0)e^{-\frac{t\ln(2)}{\tau}}. \tag{5}$$

In order to understand the ratio of parent isotope to daughter isotope over time, it is necessary to derive a function describing the amount of a daughter isotope over time, $D(t)$. If the daughter isotope is stable, then the change in amount of the daughter isotope over time is equal to the change in the amount of the parent isotope over time, and $D(t) = P(0) - P(t)$. However, if the daughter isotope is not stable, both the decay of the parent isotope and the decay of the daughter isotope must be taken into account. This yields the differential equation

$$\frac{dD(t)}{dt} = -\frac{\ln(2)}{\tau_D}D(t) + \frac{\ln(2)}{\tau_P}P(t). \tag{6}$$

This differential equation can be solved to return

$$D(t) = \frac{P(0)\tau_D}{\tau_P - \tau_D}\left(e^{\frac{-t}{\tau_P}} - e^{\frac{-t}{\tau_D}}\right). \tag{7}$$

Equation 7 is the final equation describing the present amount of an unstable daughter isotope at any point in time. However, in this form, analysis of the equilibrium behavior is difficult. If t is allowed to approach infinity, the final amount of the daughter isotope goes to zero, as expected.

For some decays, approximations can be made that simplify Equation 7 and allow us to glean information about the equilibrium state. One such decay is the beta decay by which ^{90}Sr decays into ^{90}Y. For this particular decay, a few approximations can be made based on the following assumptions:

1. $\tau_Y \ll \tau_{Sr}$
2. $\tau_Y \ll t$

The first proposition states that the half-life of ^{90}Y, approximately 64 hours, is far less than the half-life of ^{90}Sr, approximately 28 years [6]. Because of this, the approximation $\tau_Y - \tau_S r \approx \tau_P$ can be made, which simplifies Equation 7 to

$$Y(t) = \frac{Sr(0)\tau_Y}{\tau_{Sr}}\left(e^{\frac{-t}{\tau_{Sr}}} - e^{\frac{-t}{\tau_Y}}\right). \tag{8}$$

The second proposition, that the amount of time past is greater than the half-life of ^{90}Y, allows us to say that as the time increases, the value $e^{\frac{-t}{\tau_Y}}$ goes to zero, while $e^{\frac{-t}{\tau_{Sr}}}$ does not. This further simplifies Equation 8 to a final form of

$$Y(t) = \frac{Sr(0)\tau_Y}{\tau_{Sr}} e^{\frac{-t}{\tau_{Sr}}}. \tag{9}$$

Notice that this expression contains Equation 5, the function for $Sr(t)$. Substituting in $Sr(t)$ and performing a little algebra, the proportion

$$\frac{Sr(t)}{Y(t)} = \frac{\tau_{Sr}}{\tau_Y}, \tag{10}$$

is obtained. This expression directly gives us the equilibrium ratio of ^{90}Sr to ^{90}Y over time. In order to get such a distinct and clear solution, approximations must be made that are reliant on a Strontium-90 decay. While many decay chains can make the same approximation, the majority cannot. In addition, understanding the equilibrium ratio of a parent isotope to the eighth isotope down the decay chain would require a time consuming calculation and most likely yield an equation that is impractical to try to analyze.

3 Monte Carlo Methods

Because deriving an equation to analyze the equilibrium ratios of a decaying isotope over time is time consuming and not always illustrative, I sought an alternative method in the form of simulation. Our simulation relies on Monte Carlo methods to model the decay of radioactive isotopes. A Monte Carlo method refers to any algorithm that relies on drawing from a random distribution to obtain numerical results [2]. The produced random numbers simulate a physical process and information about the process is inferred from the results. This approach works best for probabilistic processes, such as radioactive decay. While half of the atoms of a radioactive substance will decay after one half-life, individual atoms may decay much earlier or much later than one half-life. The time at which an individual atom decays is determined by a random distribution dependent on the half-life of the isotope. Treating the function in Equation 5 as a distribution, it is possible to randomly generate a

decay time from said distribution. This would simulate the amount of time it takes for a particular atom to decay into its daughter isotope.

4 Development of Simulation to Model Radioactive Decay

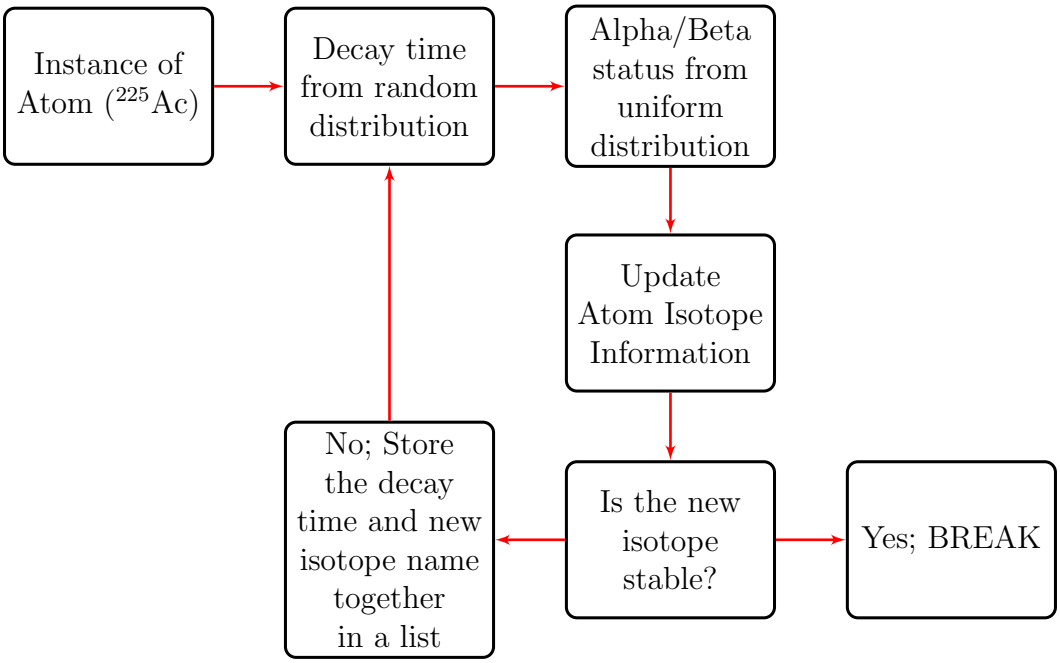
4.1 Inputs

In order to implement the Monte Carlo approach for determining equilibrium ratios, I developed a programmed script that simulates the decay of N atoms of a single isotope to a stable isotope in the decay chain. To do this, I used Python: a high-level, object oriented programming language[7]. The script takes an input of a CSV file containing information on the half-life and daughters of an isotope. For our purposes, I used only the information for the ^{225}Ac decay chain, obtained from the LBNL Nuclear Data Search [6]. The script also requires arguments for the number of atoms of the original isotope (in this case ^{225}Ac) and resolution information for the resultant plot. The script outputs a stacked histogram illustrating the amounts of each isotope in the decay chain over a period of time. The combined height of all isotope histograms is equal to the amount of atoms originally specified in the argument and the resolution of the plot is determined by the width of the bins.

4.2 Overview of Algorithms

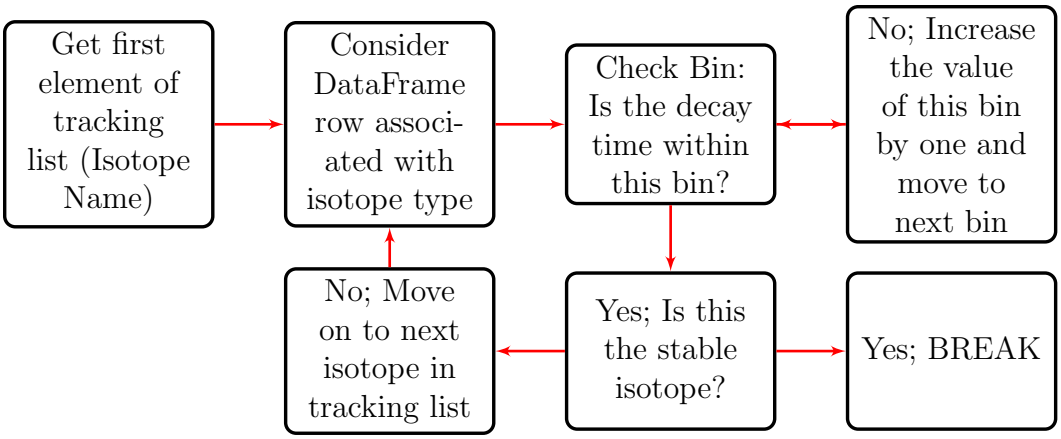
After reading in the CSV file, the program will generate a number of objects from the Atom Class equal to the `atom_count` argument provided upon running. Objects of the Atom Class contain variables storing the isotope name, half-life, daughter produced by alpha decay, daughter produced by beta decay, probability of alpha decay, and probability of beta decay. The Atom Class also contains one key method that utilizes a Monte Carlo approach by generating the time of the next decay based on the aforementioned distribution. This process is illustrated in Figure 4.2.1. The method also determines whether or not the object decays into it's alpha-daughter or beta-daughter, using a uniform distribution. A random number is chosen from a uniform distribution between 0 and 1; if the chosen number is less than the probability that an alpha decay occurs, then the isotope is changed to the alpha daughter. Else, the isotope is changed to the beta daughter. This method does not take into account other types of decay, but the probability of a decay in the ^{225}Ac chain that is not alpha or beta is negligible.

Figure 4.2.1: Decay of One Object of the Atom Class



Each Atom object created by the script undergoes a full "decay" into the stable isotope. The time of decay at each stage is recorded in a list unique to each object, called the "tracking list". These lists are eventually used to fill the bins of the final output histogram. First, for each isotope in the decay chain, there is a Pandas DataFrame row with number of elements E, which is default 50. Pandas is a python package used for data analysis, while the DataFrame is one of the data structures available through the Pandas library [8]. Each element of the row represents a period of time corresponding to the bin width of the final histogram and initially contains a value of zero. For example, if the bin width of the histogram was 30 seconds, the first element of each row represents the time interval from 0 seconds to 30 seconds. Likewise, the second element represents 30 seconds to 60 seconds. The number of bins E multiplied by the bin width returns the full time-interval represented in the histogram. Figure 4.2.2 illustrates how each tracking list is used to fill the DataFrame.

Figure 4.2.2: Filling the Bins for One Atom



4.3 Output

Once all tracking lists have been used, the resulting DataFrame contains the information needed to create histograms representing the amount present of each isotope over time. A histogram is created for each isotope, then all histogram are stacked one atop the other to ensure the total number of atoms remains constant.

Figure 1 displays an example of what is output by the script, after feeding through the information for a ^{90}Sr decay to compare to the earlier derived equations. The thin blue line separating the white ^{90}Zr and green ^{90}Sr illustrates the amount of ^{90}Y present. The amount of ^{90}Y present at any given time agrees with our expected ratio of ^{90}Sr to ^{90}Y of about 3942:1.

4.4 Challenges

The chosen method of displaying the simulated information poses a few challenges. First, because some isotopes in both the ^{225}Ac and ^{90}Sr decay chains have vastly larger half-lives than others, the isotopes with smaller half-lives are difficult to accurately represent visually. This can be seen in Figure 1. Analytically, I account for the smaller half-lives with "partial binning", a process done when filling the bins. If an atom decays into a particular isotope, then decays into its daughter during the same bin, the bin is filled with only a fraction of an atom, proportional to the amount of time it spent in that state. This provides more accurate values for the amount of a given isotope at any

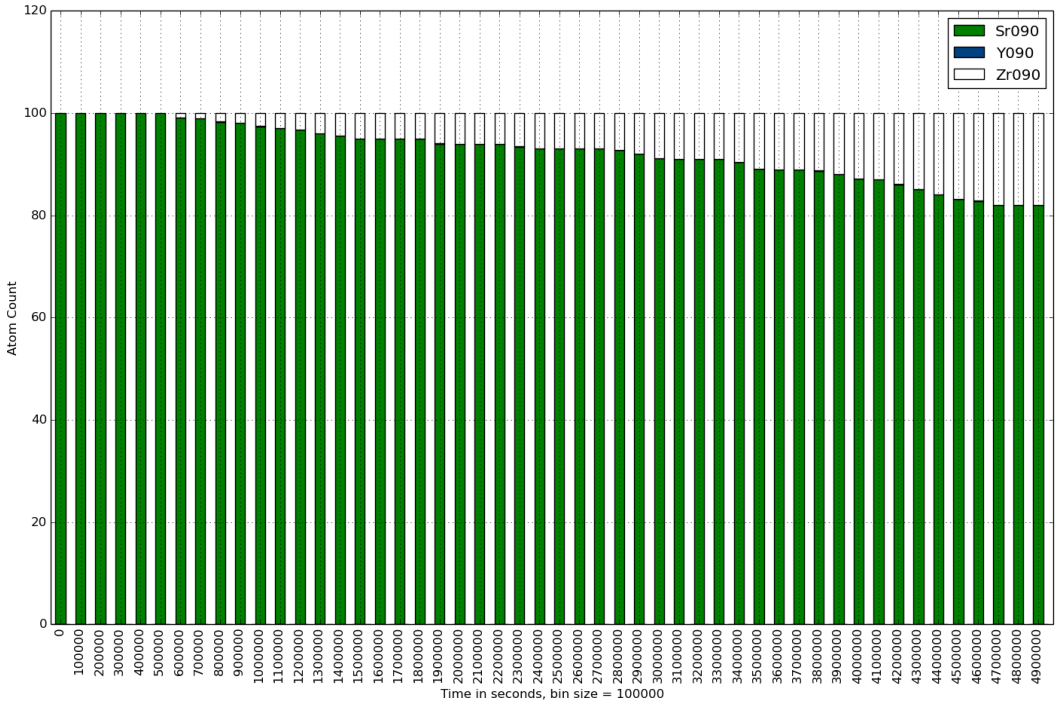


Figure 1: **Atoms Present Over Time in the Decay of ^{90}Sr**.An example output histogram, using the Sr-90 chain for ease of comparison to earlier equations. The atom count was 100 atoms and the bin size was 100000 seconds, spanning an interval of 5000000 seconds.

point in time.

5 Evaluation of Results

5.1 Comparison Between Methods

Comparing the mathematical calculation to the Monte Carlo simulation, the Monte Carlo simulation can provide both a visual representation of the long-term behavior and specific values for the present amounts of isotopes. The simulation provides a more realistic view of the phenomenon. The mathematical approach is the perfect method for deriving theoretical equations and ideal values. However, in nature, the behavior of the system has more irregularity than can be accounted for in an equation. The probabilistic Monte Carlo approach more closely models the true behavior of a radioactive decay chain in that it allows room for reasonable irregularity in the results.

5.2 Continuing Research

The Monte Carlo approach better suits the plans for continued work with this decay chain as well. As previously mentioned, understanding the equilibrium behavior of isotopes in a radioactive decay chain is important in imaging for medicine. For decay chains that produce light, understanding the ratios of specific isotopes over time is important in determining what the measured values of light correspond to. Continued work on this project will incorporate the production of Cerenkov light, which is produced when a charged particle passes through a medium faster than the speed of light in that medium [3]. By simulating the production of Cerenkov light for the Actinium-225 decay chain as well as other medically relevant decay chains, the script will contribute to the vast body of knowledge utilized in progressing cancer treatment research.

References

[1] Paul Blanchard, Robert Delaney, and Glen Hall. *Ordinary Differential Equations*. Brooks/Cole, 2002.

[2] J. Hammersley. *Monte Carlo Methods*. Springer Science and Business Media, 2013.

[3] Michael F. L'Annunziata. *Handbook of Radioactivity Analysis*. Academic Press, Sand Diego, California, 2012.

[4] Gerhart Lowenthal and Peter Airey. *Practical Applications of Radioactivity and Nuclear Radiations*. Cambridge University Press, 2001.

[5] Ervin Podgorsak. *Radiation Physics for Medical Physicists*. 2010.

[6] L.P. Ekström S.Y.F. Chu and R.B. Firestone. *The Lund/LBNL Nuclear Data Search*. Worldwatch Institute, Berkley, USA, 1999.

[7] G. van Rossum. Python tutorial. Report CS-R9526, Centrum voor Wiskunde en Informatica (CWI), 1995.

[8] PyData Development Team Wes McKinney. pandas: powerful python data analysis toolkit. Report, NUMFocus organization, 2016.

"DecayChainPlotting.py"

```
#-----------------------------------------------
#SETUP ******** Initial Module Imports:
#-----------------------------------------------

import random
from math import exp,log
import matplotlib.pyplot as plt
import numpy.random as np
import numpy as npy
import colorsys
import pylab as P
import pandas as pd

#-----------------------------------------------
#SETUP ******** ACCESSING THE DATA ON THE ISOTOPE DECAY CHAIN:
#-----------------------------------------------

#Opens the Decay Chain file
f = open("DecayChains.csv","r")

#Reads the lines of the file into a list, with each isotopes's
    information
#being an element in the list
DecayChain_List = f.readlines()
```

```
#-----------------------------------------------
#SETUP ******** MODIFYING THE DATA TO A USABLE FORM:
#-----------------------------------------------

#Takes out the new line character in each element of the
    DecayChain_List
#Ex. ['Ac225 information','Fr221 information',...,'Tl205
    information']
DecayChain_List = [item.strip("\n") for item in DecayChain_List]

#Makes each element of the DecayChain_List its own list within the
     original list
#Ex. [['Ac225','some info','more info'],['Fr221','some info','more
     info']...]
DecayChain_List = [item.split(",") for item in DecayChain_List]

#Prompts for first isotope of the Decay Chain
isotope_initial = 'Sr090'#input("First Isotope of Chain: \n")
#Prompts for simulated number of atoms to work with
atom_count = 100#int(input("Number of Atoms: \n"))

#Prompts for desired width of plot bin
bin_width = 100000#float(input("Bin width in minutes: \n"))

#-----------------------------------------------
#SIMULATION ******** DEFINITION OF ATOM CLASS:
#-----------------------------------------------

class Atom:
    def __init__(self):
        """Creates atom with initial isotope status."""
        #Loops through the Decay Chain List to find the isotope
            specified prior
        for idx in range(len(DecayChain_List)):
            if DecayChain_List[idx][0] == isotope_initial:
                idx_init = idx
        #Creates class-global variables based on the initial
            isotope's data:
        #Isotope Type - String
        self.isotope = DecayChain_List[idx_init][0]
        #Probabilty that it will decay alpha - Float
        self.alpha_prob = float(DecayChain_List[idx_init][1])
        #Probabilty that it will decay beta - Float
        self.beta_prob = float(DecayChain_List[idx_init][2])
        #Resulting isotope of alpha decay - String
        self.alpha_daughter = DecayChain_List[idx_init][3]
        #Resulting isotope of beta decay - String
        self.beta_daughter = DecayChain_List[idx_init][4]
```

```
        #Isotope Halflife - Float
        self.halflife = float(DecayChain_List[idx_init][5])
        #Creates the "Time Tracker" list to store the decay-
            isotope pairs for
        #each atom
        self.timetracker = [[0,self.isotope]]

    def decay_once(self):
        """Simulates one instance of decay of a single atom."""
        #Determines the probabilty value associated with the time
            of decay

        t = np.exponential(self.halflife/log(2))
        #Determines new isotope type and updates object
            information accordingly
        self.abcheck()
        #Creates a temporary list containing decay time and new
            isotope type
        templist = [t,self.isotope]
        #Appends the temporary list to the larger "Time Tracker"
            list that
        #stores all method-generated data for the object's decay
        self.timetracker.append(templist)

    def abcheck(self):
        """Determines the new isotope type and updates atom
            information
        accordingly."""
        #Chooses random probabilty from a uniform distribution
        r1 = random.random()
        #Determines whether the isotope underwent alpha or beta
            decay
        if r1 < self.alpha_prob:
                #In the case of alpha decay
                isotope_next = self.alpha_daughter
        else:
            #In the case of beta decay
            isotope_next = self.beta_daughter
        #Updates the isotope information based on what it decayed
            into
        for idx in range(len(DecayChain_List)):
            if DecayChain_List[idx][0] == isotope_next:
                idx_next = idx
                self.isotope = DecayChain_List[idx_next][0]
                self.alpha_prob = float(DecayChain_List[idx_next
                    ][1])
                self.beta_prob = float(DecayChain_List[idx_next
                    ][2])
```

```
                self.alpha_daughter = DecayChain_List[idx_next][3]
                self.beta_daughter = DecayChain_List[idx_next][4]
                self.halflife = float(DecayChain_List[idx_next
                    ][5])

    def decay_full(self):
        """Simulates the full decay to stable of an individual
            atom."""
        #Decays until the atom is stable
        while self.halflife != 0:
            self.decay_once()
        #Adjusts the time to be continuous as different isotope
            types decay
        for set in range(1,len(self.timetracker)):
            self.timetracker[set][0]+= self.timetracker[set-1][0]
        return self.timetracker

#-----------------------------------------
#SIMULATION ******** PROCESSING THE SIMULATED DATA:
#-----------------------------------------

#Creation of "All Lists" that contains one list for each isotope
    in the chain
timeinfo2 = []
for atm in range(atom_count):
    #Creates an object of the Atom class
    atom = Atom()
    #Retrieves the object's "Time Tracker" data
    timeinfo = atom.decay_full()
    timeinfo2.append(timeinfo)

Time = [c*bin_width for c in range(100)]
#actinium = [0.0 for c in range(100)]
#francium = [0.0 for c in range(100)]
#astatine = [0.0 for c in range(100)]
#bismuth213 = [0.0 for c in range(100)]
#polonium = [0.0 for c in range(100)]
#bismuth209 = [0.0 for c in range(100)]
#thallium205 = [0.0 for c in range(100)]
#thallium209 = [0.0 for c in range(100)]
yttrium90 = [0.0 for c in range(100)]
sr90 = [0.0 for c in range(100)]
zr90 = [0.0 for c in range(100)]
lead = [0.0 for c in range(100)]
data = {#'Ac225' : pd.Series(actinium, index=Time),
        #'Fr221' : pd.Series(francium, index=Time),
        #'At217' : pd.Series(astatine, index=Time),
        #'Bi213' : pd.Series(bismuth213, index=Time),
```

```
        #'Po213' : pd.Series(polonium, index=Time),
        #'Pb209' : pd.Series(lead, index=Time),
        #'Bi209' : pd.Series(bismuth209, index=Time),
        #'Tl205' : pd.Series(thallium205, index=Time),
        #'Tl209' : pd.Series(thallium209, index=Time),
        'Sr090' : pd.Series(sr90, index=Time),
        'Y090' : pd.Series(yttrium90, index=Time),
        'Zr090' : pd.Series(zr90, index=Time)}

df = pd.DataFrame(data)

#-----------------------------------------
for timeinfo in timeinfo2:
    for pair in range(len(timeinfo)):
        #First Pair
        if pair == 0:
            #"isotope" is set to be the first isotope
            isotope = timeinfo[0][1]
            #"decaytime" is set to be the first decay time
            decaytime = timeinfo[1][0]
            #"next" is set to be how much time the atom remains
                in the next bin
            next = decaytime%bin_width
            #"a" is used to iterate through bins
            a=0.0
            #If it decays within the first bin
            if decaytime < bin_width:
                #Put a fraction of an atom into the first bin
                df[isotope][0.0] += float((decaytime)/
                    bin_width)
            #Populate with ones
            else:
                #while a is not the bin of decay and we arent
                    out of time
                while a < decaytime-next and a < bin_width*(
                    len(df[isotope])):
                    #Put a one in the bin
                    df[isotope][a] += 1.0
                    #Increment a
                    a+=bin_width
                #Partial Binning
                if a == decaytime-next:
                    df[isotope][a] += float(next/bin_width)
#-----------------------------------------
        #Last Pair
        elif pair == len(timeinfo)-1:
            isotope = timeinfo[pair][1]
            decaytime = timeinfo[pair][0]
```

```
                #Jump to last used time
                a=decaytime-decaytime%bin_width
                if a<=bin_width*(len(df[isotope])):
                    df[isotope][a] += (a+bin_width-decaytime)/
                        bin_width
                    a+=bin_width
                    while a!=bin_width*(len(df[isotope])):
                        df[isotope][a] += 1
                        a += bin_width
#-----------------------------------------
            #Middle Pairs
            else:
                #"isotope" is the current isotope
                isotope = timeinfo[pair][1]
                #"decaytime1" is the time the current isotope was
                    decayed into
                decaytime1 = float(timeinfo[pair][0])
                #"decaytime2" is the time the current isotope is
                    decayed out of
                decaytime2 = float(timeinfo[pair+1][0])
                #fraction of current isotope in last bin
                next1 = float(decaytime1%bin_width)
                #fraction of current isotope in next bin
                next2 = float(decaytime2%bin_width)
                #Jump to last used time
                a=decaytime1-next1
                #In the case of decay in/out same bin
                if a<bin_width*(len(df[isotope])):
                    if decaytime1-next1 == decaytime2-next2:
                        df[isotope][a] += float((decaytime2-
                            decaytime1)/bin_width)
                    else:
                        df[isotope][a] += 1.0-float((next1)/
                            bin_width)
                        a+=bin_width
                        while a < decaytime2-next2 and a<
                            bin_width*(len(df[isotope])):
                            df[isotope][a] += float(1)
                            a+=bin_width
                    #Partial Binning
                        if a==decaytime2-next2:
                            df[isotope][a] += float(next2/(
                                bin_width))

#-----------------------------------------
#PLOTTING ********
#-----------------------------------------
```

```
ax = df.plot(kind="bar",stacked=True, colormap = "ocean", sharex =
    False)
ax.set_xlabel("Time in seconds, bin size = 100000")
ax.set_ylabel("Atom Count")
plt.show()
```