NUEN 629
Numerical Methods in Reactor Analysis
Homework 4 & 5 & Project

Due on:
Thursday, November 19, 2015
&
Thursday, December 3, 2015

Dr. McClarren

Paul Mendoza

Contents

Homework 4 Problem Statement

Solve the following problem and submit a detailed report, including a justification of why a reader should believe your results and a description of your methods and iteration strategies.

1. (150 points + 50 points extra credit) In class we discussed the diamond-difference spatial discretization. Another discretization is the step discretization (this has several other names from other disciplines). It writes the discrete ordinates equations with isotropic scattering as, for $\mu_n > 0$ to

$$\mu_n \frac{\psi_{i,n} - \psi_{i-1,n}}{h_x} + \Sigma_t \psi_{i,n} = \frac{\Sigma_s}{2}\phi_i + \frac{Q}{2} \tag{1}$$

and for $\mu_n < 0$

$$\mu_n \frac{\psi_{i+1,n} - \psi_{i,n}}{h_x} + \Sigma_t \psi_{i,n} = \frac{\Sigma_s}{2}\phi_i + \frac{Q}{2} \tag{2}$$

The codes provided in class should be modified to implement this discretization.

(a) (50 Points) Your task (should you choose to accept it) is to solve a problem with uniform source of $Q = 0.01$, $\Sigma_t = \Sigma_s = 100$ for a slab in vacuum of width 10 using step and diamond difference discretizations. Use, 10, 50, and 100 zones ($h_x = 1, 0.02, 0.01$) and your expert choice of angular quadratures. Discuss your results and how the two methods compare at each number of zones.

(b) (10 points) Discuss why there is a different form of the discretization for the different signs of μ.

(c) (40 points) Plot the error after each iteration using a 0 initial guess for the step discretization with source iteration and GMRES.

(d) (50 points) Solve Reed's problem (see finite difference diffusion codes). Present convergence plots for the solution in space and angle to a "refined" solution in space and angle.

(e) (50 points extra credit) Solve a time dependant problem for a slab surrounded by vacuum with $\Sigma_t = \Sigma_s = 1$ and initial condition given by $\boldsymbol{\psi(0) = 1/h_x}$ (original problem statement said $\phi(0) = 1/h_x$ and I'm not sure how to solve that). Plot the solution at $t = 1$ s, using step and diamond difference. The particles have a speed of 1 cm/s. Which discretization is better with a small time step? What do you see with a small number of ordinates compared to a really large number (100s)?

Homework 4 Problem Background

Due to the complicated nature of this course, I provided this background for the lay person (me), so that they might have some grounding for the solution and hopefully believe the results. It should be noted that most of this background information is copied from various points in Dr. McClarren's notes, and is in no way original. Anything intelligent in the following is due to this fact and for any errors, I blame myself.

Beginning with the weighty neutron transport equation.

$$\left(\frac{1}{v}\frac{\delta}{\delta t} + \hat{\Omega}\cdot\nabla + \Sigma_t\right)\psi = \int_0^\infty dE' \int_{4\pi} d\hat{\Omega}'\ K(\hat{\Omega}'\cdot\hat{\Omega}, v' \to v)\Sigma_s\psi + \frac{1}{4\pi}\chi\int_0^\infty dE'\ \bar{\nu}\Sigma_f\phi + q$$

Where $K(\hat{\Omega}'\cdot\hat{\Omega}, v' \to v)$ represents the probability of scattering from one angle and energy to another given a scattering event occured and Σ_s is the macroscopic scattering cross section. The dependencies for the variables are shown below.

$$\Sigma_t(\vec{x}, v, t)$$
$$\psi(\vec{x}, \hat{\Omega}, v, t)$$
$$\Sigma_s(\vec{x}, v, t)$$
$$\chi(\vec{x}, v)$$
$$\Sigma_f(\vec{x}, v, t)$$
$$\phi(\vec{x}, v, t)$$
$$q(\vec{x}, \hat{\Omega}, v, t)$$

There are 7 free variables (three spatial [$\vec{x}$], two angular [$\hat{\Omega}$], one energy [v] and one time [t]) in this equation. In the steady state $\left(\frac{\delta\psi}{\delta t} = 0, \text{ i.e. no time dependence}\right)$, non fissioning ($\Sigma_f = 0$) case the transport equation reduces to,

$$\left(\hat{\Omega}\cdot\nabla + \Sigma_t\right)\psi = \int_0^\infty dE' \int_{4\pi} d\hat{\Omega}'\ K(\hat{\Omega}'\cdot\hat{\Omega}, v' \to v)\Sigma_s\psi + q.$$

In order to reduce this to a single energy the following definitions are helpful (remembering all time dependence is gone).

$$\psi(\vec{x}, \hat{\Omega}) = \int_0^\infty dE\ \psi(\vec{x}, \hat{\Omega}, v(E))$$
$$\Sigma_t(\vec{x}) = \frac{\int_0^\infty dE\ \Sigma_t(\vec{x}, v(E))\psi(\vec{x}, \hat{\Omega}, v(E))}{\psi(\vec{x}, \hat{\Omega})}$$
$$K(\hat{\Omega}'\cdot\hat{\Omega}, v' \to v) = K(\hat{\Omega}'\cdot\hat{\Omega})K(v' \to v)$$
$$\Sigma_s(\vec{x}) = \frac{\int_0^\infty dE \int_0^\infty dE'\ \Sigma_s(\vec{x}, v(E))K(v' \to v)\psi(\vec{x}, \hat{\Omega}, v(E))}{\psi(\vec{x}, \hat{\Omega})}$$
$$q(\vec{x}, \hat{\Omega}) = \int_0^\infty dE\ q(\vec{x}, \hat{\Omega}, v(E))$$

Using these definitions, integrating the transport equation over all energy, and assuming cross sections and sources do not vary in space or angle, our transport equation reduces again to,

$$\left(\hat{\Omega}\cdot\nabla + \Sigma_t\right)\psi(\vec{x}, \hat{\Omega}) = \int_{4\pi} d\hat{\Omega}'\ K(\hat{\Omega}'\cdot\hat{\Omega})\Sigma_s\psi(\vec{x}, \hat{\Omega}') + q.$$

Homework 4 Problem Background continued on next page...

Where the double differential was assumed to be separable in angle and energy. The final simplification for our problem will be in space. If we assume that our geometry is infinite in y $\left(\frac{\delta}{\delta y}=0\right)$ and x $\left(\frac{\delta}{\delta x}=0\right)$. This also means that ψ depends only on z and mu, and if we recall that

$$\hat{\Omega}=(\sqrt{1-\mu^2}cos(\rho), \sqrt{1-\mu^2}sin(\rho), \mu),$$

and

$$\nabla=\left(\frac{\delta}{\delta x},\frac{\delta}{\delta y},\frac{\delta}{\delta x}\right)$$

also assuming that

$$K(\hat{\Omega}'\cdot\hat{\Omega})=\frac{1}{4\pi}\text{ Isotropic Scattering}$$

then our transport equation, and the equation I think we are trying to solve for this homework is.

$$\left(\mu\frac{\delta}{\delta z}+\Sigma_t\right)\psi(z,\mu)=\Sigma_s\frac{2\pi}{4\pi}\int_{-1}^{1}d\mu'\ \psi(z,\mu')+q.$$

Checking units,

$$\left(\mu\frac{\delta}{\delta z}+\Sigma_t\right)\left[\frac{1}{cm}\right]\psi(z,\mu)\left[\frac{n\cdot cm}{str\cdot cm^3\cdot s}\right]=\Sigma_s\frac{1}{2}\left[\frac{1}{cm\cdot rad}\right]\int_{-1}^{1}d\mu'\ \psi(z,\mu')\left[\frac{n\cdot cm}{rad\cdot cm^3\cdot s}\right]+q\left[\frac{n}{str\cdot cm^3\cdot s}\right].$$

Σ_s was moved outside the integral because it has no angular dependence integration over the azimuthal angle occured because $\psi(z,\hat{\Omega})$ is assumed to be uniform and not depend on that angle.

Using Gauss-Legendre Quadrature for the integration term

$$\phi=\int_{-1}^{1}d\mu'\psi(z,\mu')=\sum_{i=1}^{n}w_i\psi(z,\mu_i')$$

where

$$w_i=\frac{2}{(1-\mu_i^2)[P_n'(\mu_i)]^2}$$

P_n' is the differential of the legendre polynomial n, and μ_i' are the roots of P_n. The weights of even n's of the legendre polynomials should sum to 2, the value of $\int_{-1}^{1}d\mu$, which they do.

Putting this all together with time dependence:

$$\boxed{\left(\frac{1}{v}\frac{\delta}{\delta t}+\mu\frac{\delta}{\delta z}+\Sigma_t\right)\psi_n(z)=\Sigma_s\frac{1}{2}\sum_{n'=1}^{N}w_{n'}\psi_{n'}(z)+q}$$

Where n and n' denote the direction being solved for and N is the total number of angles being solved for. Also unites of w are rad.

Diamond difference discretization

$$\frac{1}{v}\frac{\psi_{n,i}^{\ell+1,j+1}-\psi_{n,i}^{L,j}}{\Delta t}+\mu_n\frac{\psi_{n,i+1/2}^{\ell+1,j+1}-\psi_{n,i-1/2}^{\ell+1,j+1}}{hz}+\Sigma_t\psi_{n,i}^{\ell+1,j+1}=\Sigma_s\frac{1}{2}\sum_{n'=1}^{N}w_i\psi_{n',i}^{\ell,j+1}+q.$$

Where n is for angle, i is the midplane of a spacial discretization, ℓ is the iteration index for spacial convergence, j is for a time step and

$$\psi_{n,i}^{\ell+1,j+1}=\frac{1}{2}(\psi_{n,i+1/2}^{\ell+1,j+1}+\psi_{n,i-1/2}^{\ell+1,j+1})$$

Writing this in terms of a steady state

$$\mu_n \frac{\psi_{n,i+1/2}^{\ell+1,j+1} - \psi_{n,i-1/2}^{\ell+1,j+1}}{hz} + \Sigma_t^* \psi_{n,i}^{\ell+1,j+1} = \Sigma_s \frac{1}{2} \sum_{n'=1}^{N} w_i \psi_{n',i}^{\ell,j+1} + q^*.$$

where

$$\Sigma_t^* = \Sigma_t + \frac{1}{v\Delta t}$$

$$q^* = q + \frac{\psi_{n,i}^{L,j}}{v\Delta t}$$

The above equation has L for the iteration index to indicate that its value was iteratively determined in the previous time step.

Step discretization

Writing this in terms of a steady state for $\mu > 0$

$$\mu_n \frac{\psi_{n,i}^{\ell+1,j+1} - \psi_{n,i-1}^{\ell+1,j+1}}{hz} + \Sigma_t^* \psi_{n,i}^{\ell+1,j+1} = \Sigma_s \frac{1}{2} \sum_{n'=1}^{N} w_i \psi_{n',i}^{\ell,j+1} + q^*.$$

and for $\mu < 0$

$$\mu_n \frac{\psi_{n,i+1}^{\ell+1,j+1} - \psi_{n,i}^{\ell+1,j+1}}{hz} + \Sigma_t^* \psi_{n,i}^{\ell+1,j+1} = \Sigma_s \frac{1}{2} \sum_{n'=1}^{N} w_i \psi_{n',i}^{\ell,j+1} + q^*.$$

where

$$\Sigma_t^* = \Sigma_t + \frac{1}{v\Delta t}$$

$$q^* = q + \frac{\psi_{n,i}^{L,j}}{v\Delta t}$$

GMRES

The generalized minimium residual (GMRES) method is an iterative method for solving linear systems of equations. The method approximates the solution by the vector in a Krylov subspace with a minimum residual (see wikipedia or Dr. McClarren's notes, I'm not really sure how this method works, but python has a solver for it).

The system $A\vec{\phi} = b$ is solved with GMRES, where for our situation,

$$A = \left(I - \sum_{n'=1}^{N} L^{-1} \Sigma_s \frac{1}{2} \right)$$

where L^{-1} is a sweep solve for our system and acts as an operator (I think), and

$$b = \sum_{n'=1}^{N} L^{-1} q^*$$

Reeds Problem

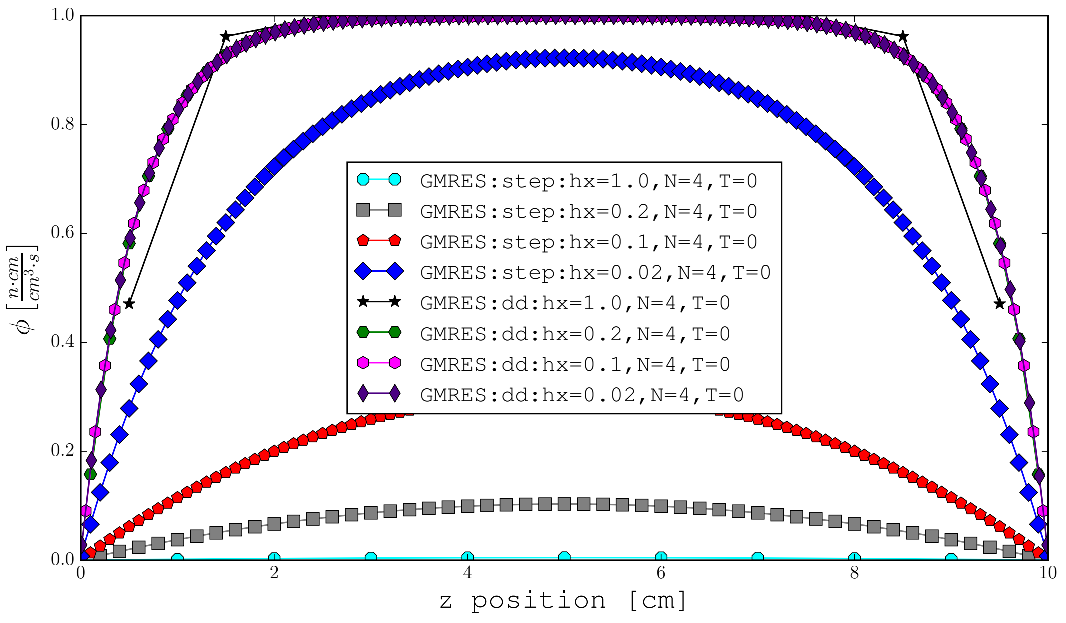
Reeds problem is a similiar system as above, except the source and scattering and total cross sections are variable in z, and the width of z is 16.

Homework 4 Problem Solution

The code for this problem will be at the end of this section. The answers are below.

(a) (50 Points) Your task is to solve a problem with uniform source of $Q = 0.01$, $\Sigma_t = \Sigma_s = 100$ for a slab in vacuum of width 10 using step and diamond difference discretizations. Use, 10, 50, and 100 zones ($h_x = 1, 0.02, 0.01$) and your expert choice of angular quadratures. Discuss your results and how the two methods compare at each number of zones.

The angular quadrature used was the Gauss-Legendre Quadrature because of the integration range. Its form was shown in the background section. The plot below was produced with the GMRES method, but the source iteration scheme produced the same results.



Both of the iterative solutions converged with max iterations of 100,000 and a slight modification on cross section ($\Sigma_t = \Sigma_t \cdot 1.0001$) to help the system converge. As the number of zones increased for the step solution, the flux magnitude kept increasing to match with the diamond difference and maintained a cosine(ish) shape. As the number of zones increased with the diamond difference, the shape started to converge towards the cosine, but maintained the proper magnitude.

Something else I would like to point out in the solution is that the step solution always had one more point plotted than the diamond difference. The reason for this is due to how each solution was solved. This is easier highlighted (for me) with an example, which is shown in the case where the number of zones is 10.

For the Diamond difference, the average locations (remember they were averaged), $\psi_{n,i}^{L,j+1}$, being solved for were,

$$z = [0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5]$$

The points for $\psi_{n,i+1/2}^{L,j+1}$ and $\psi_{n,i-1/2}^{L,j+1}$ were at the points,

$$z = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

When sweeping to the right, $\psi_n(z = 0)$ was set to zero, because the incoming flux is zero, and all points were solved for up to where $z = 10$, and $\psi_{n,i}$ values were determined with averaging. This same thing occured when sweeping to the left (except here $\psi_n(z = 10)$ was set to zero). This would yield 10 values at the points $[0.5, 1.5, ..., 9.5]$.

For the step discretization scheme, the locations (non averaged), $\psi_{n,i}^{L,j+1}$, being solved for were,

$$x = \begin{cases} [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] & \mu > 0 \\ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9], & \mu < 0 \end{cases}$$

When combining these two lists for ϕ, this was considered, and hence the step discretization scheme had one extra point (both lists have 10 points, but the location 10 is unqiue in the first list, and 0 in the second).

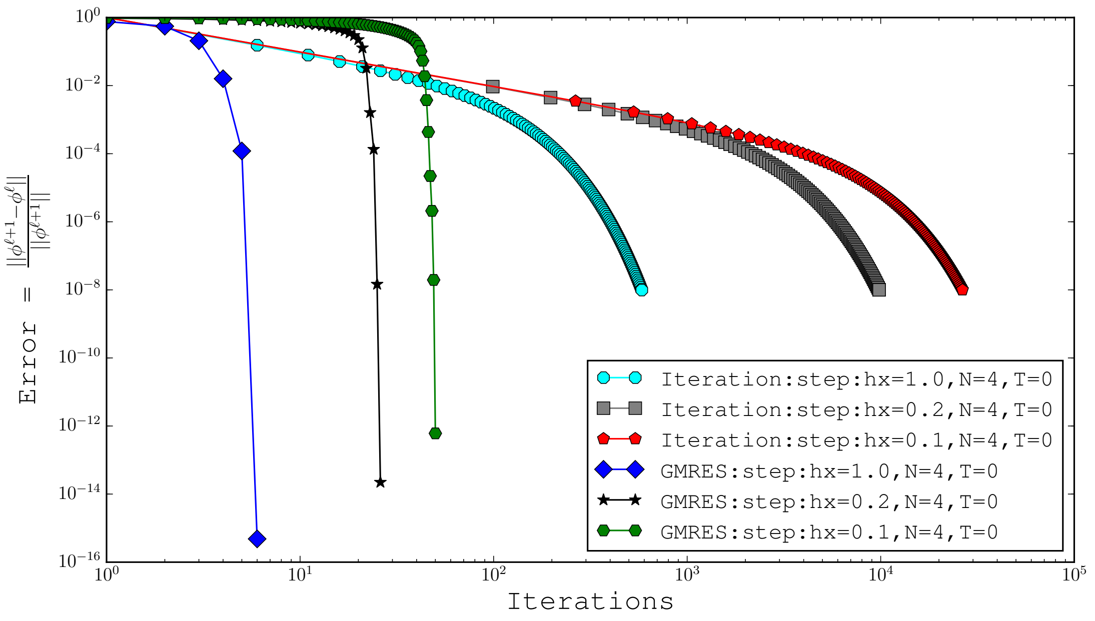
(b) (10 points) Discuss why there is a different form of the discretization for the different signs of μ.

The different forms are needed in the step discretization because in both the diamond and step approaches to the solution a value is needed from a previous zone. Our vacuum boundary condition states that the incoming neutrons are zero, which at the left side of the boundary, determines the angular flux moving to the right, and at the right side of the boundary, the angular flux moving to the left (these values are 0).

(c) (40 points) Plot the error after each iteration using a 0 initial guess for the step discretization with source iteration and GMRES.

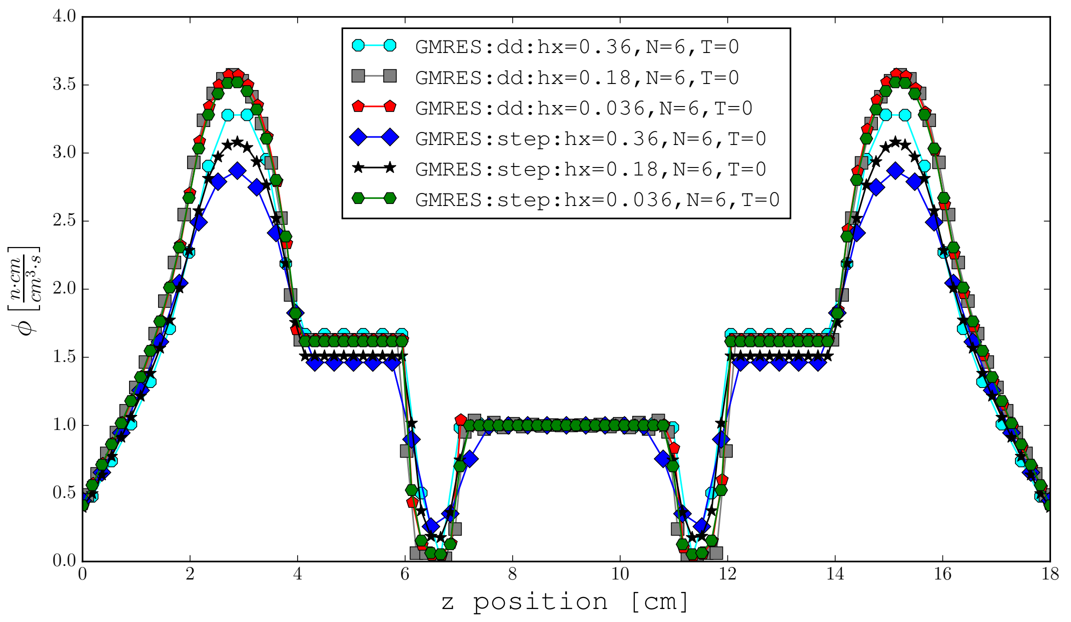
Error will be determined with the following:

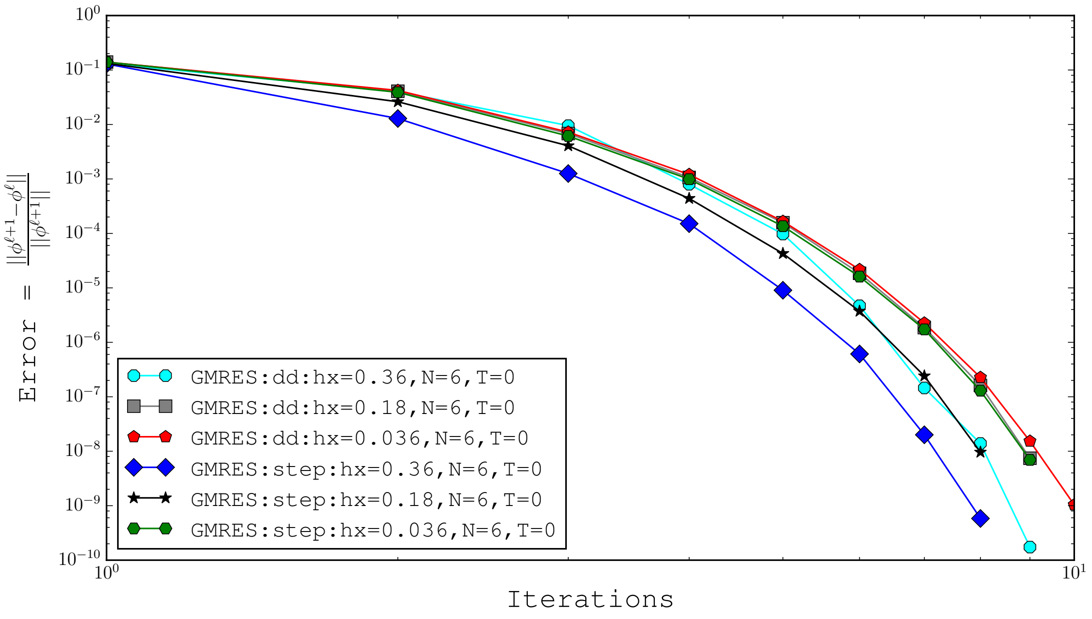
$$\text{Error} = \frac{||\boldsymbol{\phi}^{\boldsymbol{\ell}+1} - \boldsymbol{\phi}^{\boldsymbol{\ell}}||}{||\boldsymbol{\phi}^{\boldsymbol{\ell}+1}||}$$

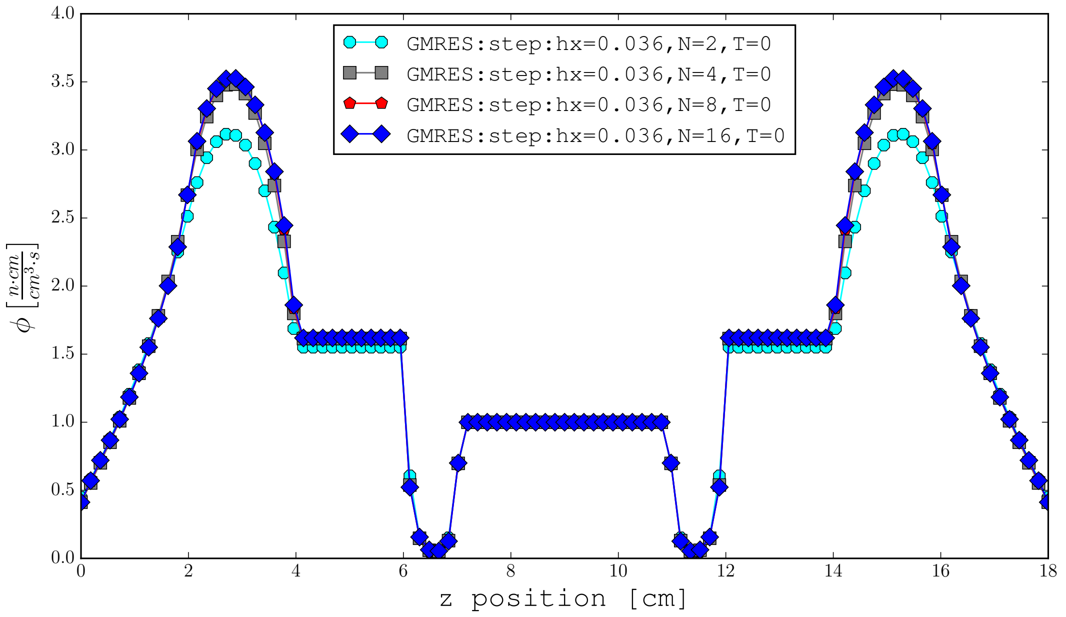


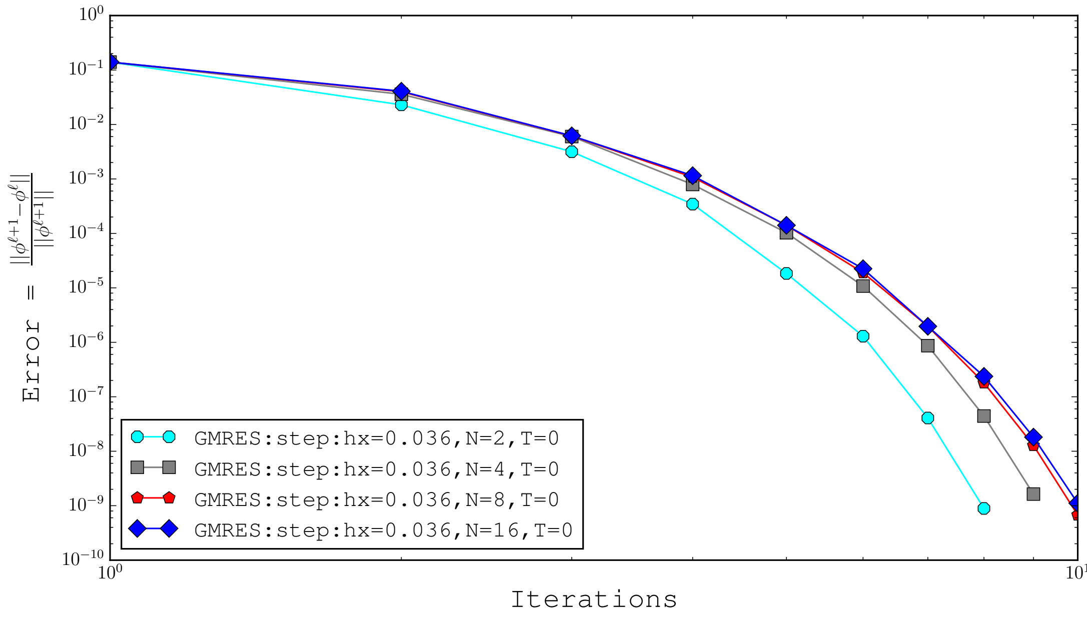
(d) (50 points) Solve Reed's problem (see finite difference diffusion codes). Present convergence plots for the solution in space and angle to a "refined" solution in space and angle.

Plots are below, reduced the number of points so that figures wouldn't take so long to load.




10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{-6}
10^{-7}
10^{-8}
10^{-9}
10^{-10}
10^{1}
Error = $\frac{||\phi^{\ell+1}-\phi^{\ell}||}{||\phi^{\ell+1}||}$
Iterations
GMRES:dd:hx=0.36,N=6,T=0
GMRES:dd:hx=0.18,N=6,T=0
GMRES:dd:hx=0.036,N=6,T=0
GMRES:step:hx=0.36,N=6,T=0
GMRES:step:hx=0.18,N=6,T=0
GMRES:step:hx=0.036,N=6,T=0


4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10
12
14
16
18
$\phi\left[\frac{n\cdot cm}{cm^3\cdot s}\right]$
z position [cm]
GMRES:step:hx=0.036,N=2,T=0
GMRES:step:hx=0.036,N=4,T=0
GMRES:step:hx=0.036,N=8,T=0
GMRES:step:hx=0.036,N=16,T=0



The solution converges with more spatial slices. Increasing the number of angular slices helps upto when N=4, but beyond that it doesn't do much.

(e) (50 points extra credit) Solve a time dependant problem for a slab surrounded by vacuum with $\Sigma_t = \Sigma_s = 1$ and initial condition given by $\boldsymbol{\psi(0) = 1/h_x}$ (original problem statement said $\phi(0) = 1/h_x$ and I'm not sure how to solve that). Plot the solution at $t = 1$ s, using step and diamond difference. The particles have a speed of 1 cm/s. Which discretization is better with a small time step? What do you see with a small number of ordinates compared to a really large number (100s)?

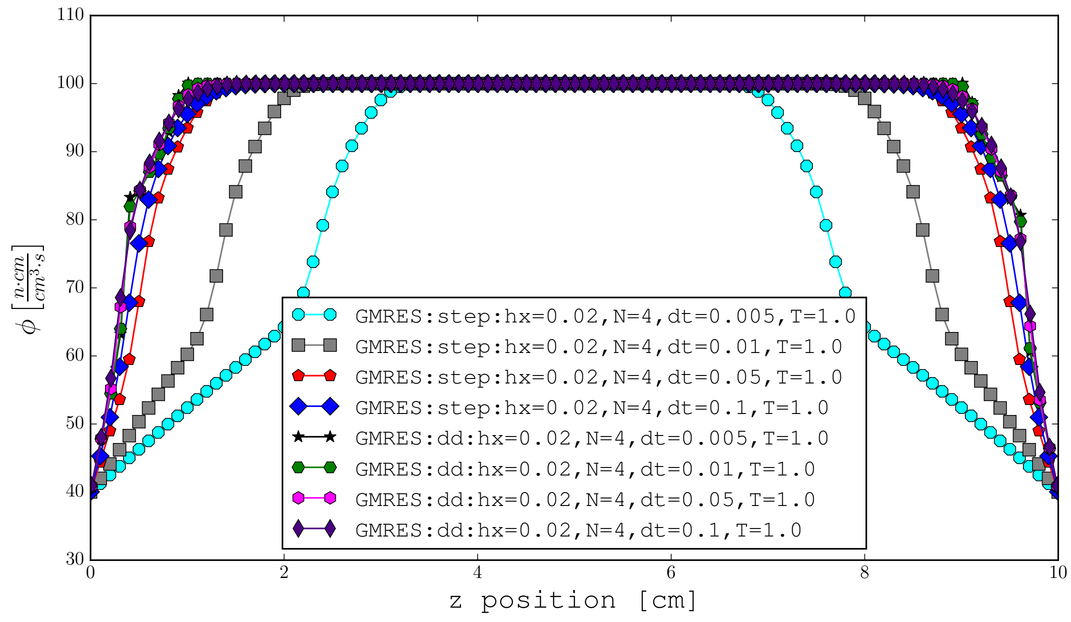


Figure 1: Q=0.01

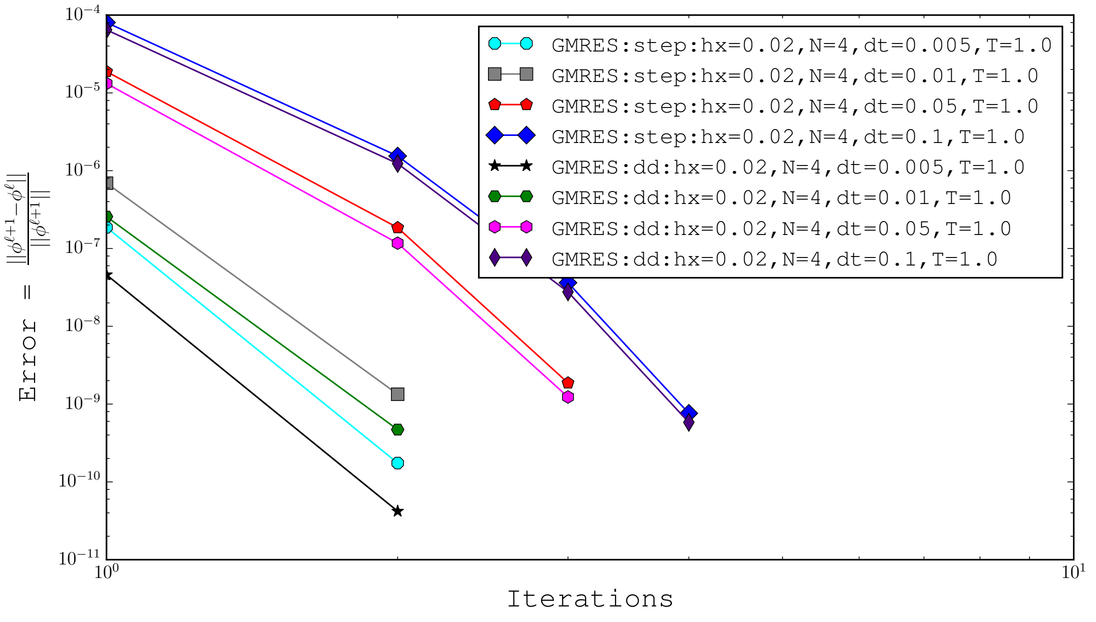
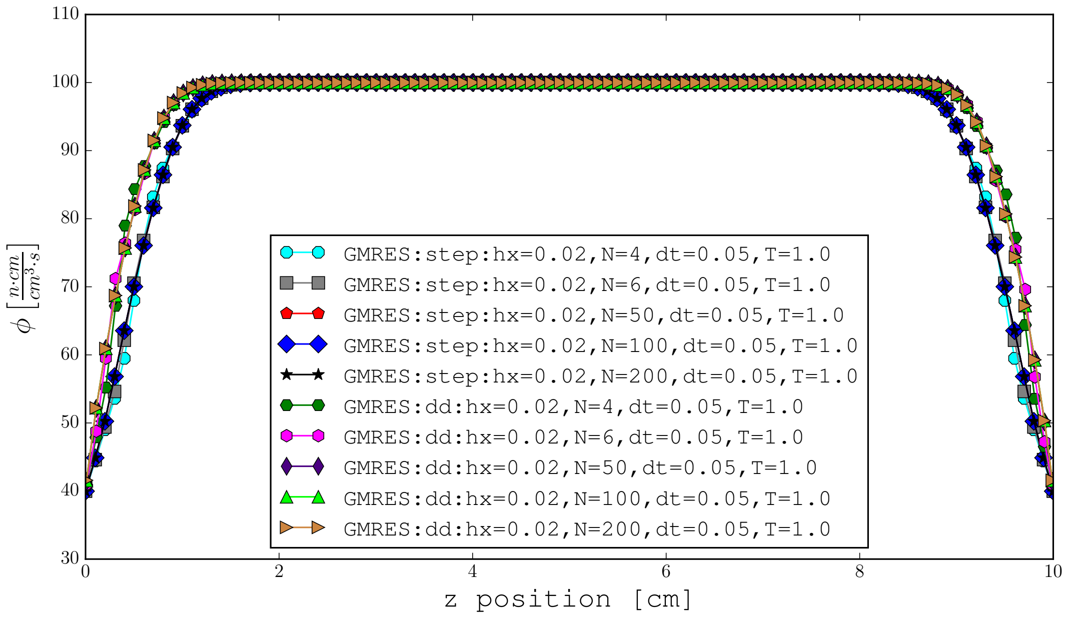
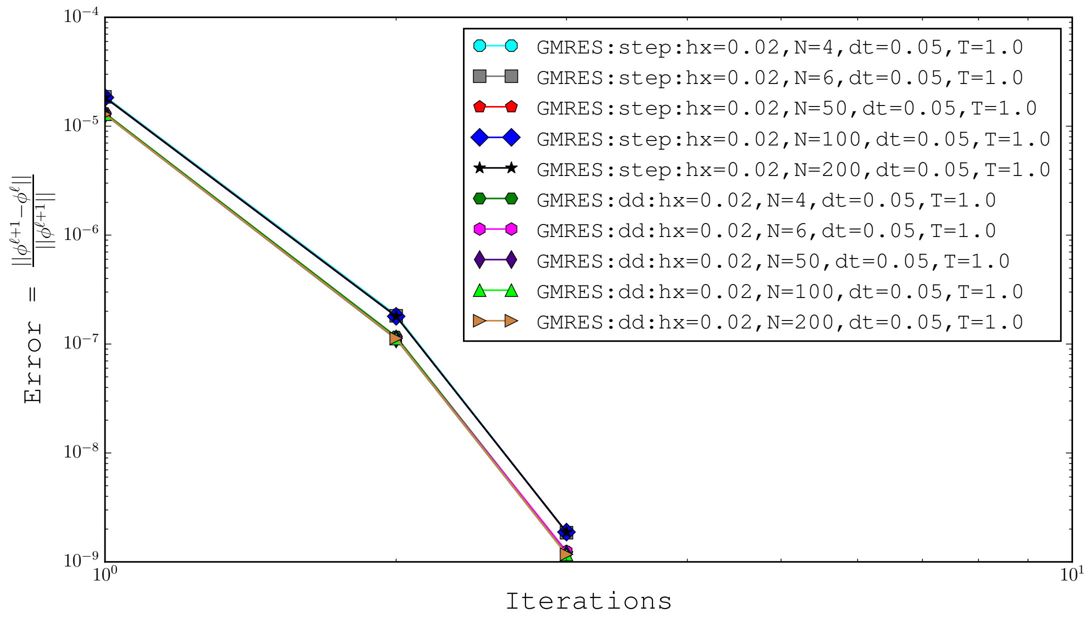


Figure 2: Q=0.01

Based on the above graphs, I am not sure which solution does better with a smaller step size. It depends on what the answer should be. I think the step solution, as the step size increases, look like they have some nonphysical bends in the solution. This could be due to lots of things, but maybe its because of the smaller step size, which makes me think the diamond difference method is better with smaller step sizes.





As the number of ordinances increase in the problem there isn't much change in the solution, which is expected because with the Quadrature rule we used, the integral can usually be expressed within around 6 terms. There was an increase in computational time though.

Homework 4 Code

Listing 1: **Main Code For Parts a,b and c**

```python
#!/usr/bin/env python3

###############################################################
#################### Import packages ########################
###############################################################

import time
start_time = time.time()
import Functions as f

###############################################################
####################### Inputs ##############################
###############################################################

# Constants
Q = 0.01
Sigma_t = 100;Sigma_s=100
# Add adsorption to help converge
if Sigma_t==Sigma_s:
    Sigma_t=Sigma_t*1.0001

# Geometry
L = 10.                         # Width of slab
slices=[10,50,100,500]          # Number of cuts in slab (looped)
N = 4                           # Number of angle slices
BCs = f.np.zeros(N)             # Zero incoming flux

#Time
T=0                             # total Time (A plot made at T)
dt=1                            # Time steps width
v=1                             # Velocity

MAXITS=100000                  # Max iterations for source iter
loud=False                      # Echo every Iteration?

#Method
Methods=['GMRES:step',           # 'Iteration' or 'GMRES'
         'GMRES:dd']           # Methods to solve with?
                               # 'step' or 'dd'

tol=1e-8

PlotError=False                # Do we plot the error?

NumOfPoints=100                # Max Number of points for plots

###############################################################
################### Initialize Figures #######################
###############################################################

```

```
Check=0
fig=f.plt.figure(figsize=f.FigureSize)    # Plot all Methods
ax=fig.add_subplot(111)
if PlotError:
    erfig=f.plt.figure(figsize=f.FigureSize) # Err Plot
    erax=erfig.add_subplot(111)              # at T=0


################################################################
##################### Calculations #############################
################################################################

for Scheme in Methods:

    Method=Scheme.split(':')[1]
    ##########################################################
    #################### Set Up ##############################
    ##########################################################

    for II in slices:
        if Method == 'step': #Step Dude needs one extra
            I=II+1
        elif Method == 'dd':
            I=II

        #Width, ang lists for materials
        hx = L/II
        q = f.np.ones(I)*Q
        Sig_t_discr = f.np.ones(I)*Sigma_t
        Sig_s_discr = f.np.ones(I)*Sigma_s

        #Initialize psi (for time steps)
        if T==0:
            psi=f.np.zeros((N,I))
            Time=[0]
        else:
            psi=f.np.ones((N,I))*(1/hx)
            Time=f.Timevector(T,dt)

        label_tmp=Scheme+":hx="+str(hx)+",N="+str(N)+",T="
        ########################################################
        ################### Determine phi ######################
        ########################################################

        for t in Time: #Loop over time

            label=label_tmp+str(t)

            #Determine phi (new psi is determined for time steps)
            x,phi,it,er,psi=f.solver(I,hx,q,Sig_t_discr,
            Sig_s_discr,N,psi,v,dt,Time,BCs,Scheme,tol,MAXITS,loud)

            ####################################################
            ################ Plot Information ###################
            #####################################################
            fig
            ax,fig=f.plot(x,phi,ax,label,fig,Check,NumOfPoints)
            if t==0 and PlotError:
                erfig
                erax,erfig=f.plotE(it,er,erax,label,erfig,
                                   Check,NumOfPoints)
            Check=Check+1

#########################################################
################### Legend/Save  #######################
#########################################################

fig
f.Legend(ax)
#f.plt.savefig('Plots/FluxPlot.pdf')
if PlotError:
    erfig
    f.Legend(erax)
    #f.plt.savefig('Plots/ErrorPlot.pdf')
    f.plt.savefig('Plots/ErrorPlotTime.pdf')
    #f.plt.clf()
    f.plt.close()
fig
f.plt.savefig('Plots/FluxPlotTime.pdf')
#f.plt.show()

#Why is tmp_psi in the GMRES going negative?

###################### Time To execute ################

print("--- %s seconds ---" % (time.time() - start_time))
```

Listing 2: **Main Code For Part d**

```
#!/usr/bin/env python3

################################################################
##################### Import packages #########################
################################################################

import time
start_time = time.time()
import Functions as f

################################################################
######################## Inputs ################################
################################################################

# Geometry
L = 18.                         # Width of slab
slices=[500]             # Number of cuts in slab (looped)
NN = [2,4,8,16]                 # Number of angle slices
```

```
#Time
T=0                              # total Time (A plot made at T)
dt=1                             # Time steps width
v=1                              # Velocity

MAXITS=1000000                   # Max iterations for source iter
loud=False                       # Echo every Iteration?

#Method
Methods=['GMRES:step']#,            # 'Iteration' or 'GMRES'
        # 'GMRES:step']          # Methods to solve with?
                                # 'step' or 'dd'

tol=1e-8

PlotError=True                 # Do we plot the error?

NumOfPoints=100                 # Max Number of points for plots

################################################################
################### Initialize Figures #######################
################################################################

Check=0
fig=f.plt.figure(figsize=f.FigureSize)    # Plot all Methods
ax=fig.add_subplot(111)
if PlotError:
    erfig=f.plt.figure(figsize=f.FigureSize) # Err Plot
    erax=erfig.add_subplot(111)              # at T=0


################################################################
#################### Calculations ############################
################################################################

for Scheme in Methods:

    Method=Scheme.split(':')[1]
    #########################################################
    #################### Set Up ###########################
    #########################################################

    for II in slices:
        if Method == 'step': #Step Dude needs one extra
            I=II+1
        elif Method == 'dd':
            I=II

        #Width, ang lists for materials
        hx = L/II
        q = f.np.zeros(I)
        Sig_t_discr = f.np.zeros(I)
        Sig_s_discr = f.np.zeros(I)

        if Method == 'step':
            x = f.np.linspace(0,(I-1)*hx,I)
        elif Method == 'dd':
            x = f.np.linspace(hx/2,I*hx-hx/2,I)

        for i in range(0,len(x)):
            q[i]=f.QReed(x[i])
            Sig_t_discr[i]=f.Sigma_tReed(x[i])
            Sig_s_discr[i]=Sig_t_discr[i]-f.Sigma_aReed(x[i])


        for N in NN:
            BCs = f.np.zeros(N)           # Zero incoming flux
            #Initialize psi (for time steps)
            if T==0:
                psi=f.np.zeros((N,I))
                Time=[0]
            else:
                psi=f.np.ones((N,I))*(1/hx)
                Time=f.Timevector(T,dt)

            label_tmp=Scheme+":hx="+str(hx)+",N="+str(N)+",T="
            #########################################################
            #################### Determine phi #####################
            #########################################################

            for t in Time: #Loop over time

                label=label_tmp+str(t)

                #Determine phi (new psi is determined for time steps)
                x,phi,it,er,psi=f.solver(I,hx,q,Sig_t_discr,
                Sig_s_discr,N,psi,v,dt,Time,BCs,Scheme,tol,MAXITS,loud)

                #####################################################
                ################ Plot Information ###################
                #####################################################
                fig
                ax,fig=f.plot(x,phi,ax,label,fig,Check,NumOfPoints)
                if t==0 and PlotError:
                    erfig
                    erax,erfig=f.plotE(it,er,erax,label,erfig,
                                       Check,NumOfPoints)
                Check=Check+1

#########################################################
################## Legend/Save  #######################
#########################################################

fig
f.Legend(ax)
```

```
#f.plt.savefig('Plots/FluxPlot.pdf')
if PlotError:
    erfig
    f.Legend(erax)
    #f.plt.savefig('Plots/ErrorPlot.pdf')
    f.plt.savefig('Plots/ErrorPlotReedVaryN.pdf')
    #f.plt.clf()
    f.plt.close()
fig
f.plt.savefig('Plots/FluxPlotReedVaryN.pdf')
#f.plt.show()

###################### Time To execute #################

print("--- %s seconds ---" % (time.time() - start_time))
```

Listing 3: **Main Code For Part e**

```
#!/usr/bin/env python3

################################################################
##################### Import packages #########################
################################################################

import time
start_time = time.time()
import Functions as f

################################################################
####################### Inputs ################################
################################################################

# Geometry
L = 10                          # Width of slab
# Constants
Q = 0.01
Sigma_t = 1;Sigma_s=1
# Add adsorption to help converge
if Sigma_t==Sigma_s:
    Sigma_t=Sigma_t*1.0001

slices=[500]            # Number of cuts in slab (looped)
NN = [4,6,50,100,200]                # Number of angle slices

#Time
T=1                             # total Time (A plot made at T)
dtt=[0.05]                       # Time steps width
v=1                             # Velocity

MAXITS=1000000                  # Max iterations for source iter
loud=False                      # Echo every Iteration?

#Method
Methods=['GMRES:step',           # 'Iteration' or 'GMRES'
```

```
         'GMRES:dd']       # Methods to solve with?
                             # 'step' or 'dd'

tol=1e-8
Ttol=1e-3

PlotError=True              # Do we plot the error?

NumOfPoints=100              # Max Number of points for plots

#################################################################
#################### Initialize Figures #######################
#################################################################

Check=0
fig=f.plt.figure(figsize=f.FigureSize)   # Plot all Methods
ax=fig.add_subplot(111)
if PlotError:
    erfig=f.plt.figure(figsize=f.FigureSize) # Err Plot
    erax=erfig.add_subplot(111)              # at T=0


#################################################################
##################### Calculations ###########################
#################################################################

for Scheme in Methods:

    Method=Scheme.split(':')[1]
    ##########################################################
    ##################### Set Up ############################
    ##########################################################

    for II in slices:
        if Method == 'step': #Step Dude needs one extra
            I=II+1
        elif Method == 'dd':
            I=II

        #Width, ang lists for materials
        hx = L/II
        q = f.np.ones(I)*Q
        Sig_t_discr = f.np.ones(I)*Sigma_t
        Sig_s_discr = f.np.ones(I)*Sigma_s


        for N in NN:
            BCs = f.np.zeros(N)           # Zero incoming flux
            for dt in dtt:
                #Initialize psi (for time steps)
                if T==0:
                    psi=f.np.zeros((N,I))
                    Time=[0]
                else:
                    psi=f.np.ones((N,I))*(1/hx)
                    Time=f.Timevector(T,dt)

                label_tmp=Scheme+":hx="+str(hx)+",N="+str(N)+",dt="+\
                           str(dt)+",T="
                ########################################################
                ################### Determine phi #####################
                ########################################################

                for t in Time: #Loop over time

                    label=label_tmp+str(round(t,3))

                    #Determine phi (new psi is determined for time steps)
                    x,phi,it,er,psi=f.solver(I,hx,q,Sig_t_discr,
                    Sig_s_discr,N,psi,v,dt,Time,BCs,Scheme,tol,MAXITS,loud)

                    ####################################################
                    ################ Plot Information ##################
                    ####################################################
                    PlotQuestion=abs(t-dt)<Ttol\
                    or abs(t-0.5)<Ttol or abs(t-1)<Ttol
                    PlotQuestion=abs(t-1)<Ttol
                    if PlotQuestion:
                        fig
                        ax,fig=f.plot(x,phi,ax,label,fig,Check,NumOfPoints)
                        if PlotError:
                            erfig
                            erax,erfig=f.plotE(it,er,erax,label,erfig,
                                               Check,NumOfPoints)
                        Check=Check+1

########################################################
################## Legend/Save  #######################
########################################################

fig
f.Legend(ax)
#f.plt.savefig(’Plots/FluxPlot.pdf’)
if PlotError:
    erfig
    f.Legend(erax)
    #f.plt.savefig(’Plots/ErrorPlot.pdf’)
    f.plt.savefig(’Plots/ErrorPlotTimeVaryN.pdf’)
    #f.plt.clf()
    f.plt.close()
fig
f.plt.savefig(’Plots/FluxPlotTimeVaryN.pdf’)
#f.plt.show()

###################### Time To execute ################
```

```
print("--- %s seconds ---" % (time.time() - start_time))
```

Listing 4: **Functions holder**

```
#!/usr/bin/env python3

#################################################################
#################### Import packages #########################
#################################################################

import sys
import numpy as np
import scipy.sparse.linalg as spla

import scipy.special as sps
import matplotlib.pyplot as plt
plt.rcParams["font.family"] = "monospace"
import matplotlib
matplotlib.rc('text',usetex=True)
matplotlib.rcParams['text.latex.preamble']=[r"\usepackage{amsmath}"]
import random as rn
import matplotlib.mlab as mlab
import copy
import os

##############################################################
######################## Variables #######################
##############################################################

# Basic information
FigureSize = (11, 6)              # Dimensions of the figure
TypeOfFamily='monospace'          # This sets the type of font for text
font = {'family' : TypeOfFamily}  # This sets the type of font for text
LegendFontSize = 12
Lfont = {'family' : TypeOfFamily}  # This sets up legend font
Lfont['size']=LegendFontSize

Title = ''
TitleFontSize = 22
TitleFontWeight = "bold"  # "bold" or "normal"

#Xlabel='E (eV)'   # X label
XFontSize=18          # X label font size
XFontWeight="normal"  # "bold" or "normal"
XScale="linear"       # 'linear' or 'log'
XScaleE='log'      # Same but for error plot

YFontSize=18                    # Y label font size
YFontWeight="normal"            # "bold" or "normal"
YScale="linear"                 # 'linear' or 'log'
YScaleE='log'

Check=0

```

```
Colors=["aqua","gray","red","blue","black",
                "green","magenta","indigo","lime","peru","steelblue",
                "darkorange","salmon","yellow","lime","black"]

# If you want to highlight a specific item
# set its alpha value =1 and all others to 0.4
# You can also change the MarkSize (or just use the highlight option below)
Alpha_Value=[1  ,1  ,1  ,1  ,1  ,1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
MarkSize=   [8  ,8  ,8  ,8  ,8  ,8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8]

Linewidth=[1  ,1  ,1  ,1  ,1  ,1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]

# Can change all these to "." or "" for nothing "x" isn't that good
MarkerType=["8","s","p","D","*","H","h","d","^",">"]

# LineStyles=["solid","dashed","dash_dot","dotted","."]
LineStyles=["solid"]

SquishGraph = 0.75
BBOXX = 1.24
BBOXY = 0.5       # Set legend on right side of graph

NumberOfLegendColumns=1

Xlabel='z position [cm]'
Ylabel="$\phi\left[\\frac{n\cdot cm}{cm^3\cdot s}\\right]$"

XlabelE='Iterations'
YlabelE="Error = $\\frac{||\phi^{\ell+1}-\phi^\ell||}{||\phi^{\ell+1}||}$"

#################################################################
######################### Functions ###########################
#################################################################

def Sigma_tReed(r):
    value = 0 + ((1.0*(r>=14) + 1.0*(r<=4)) +
                 5.0 *((np.abs(r-11.5)<0.5) or (np.abs(r-6.5)<0.5)) +
                 50.0 * (np.abs(r-9)<=2) )
    return value;
def Sigma_aReed(r):
    value = 0 + (0.1*(r>=14) + 0.1*(r<=4) +
                 5.0 *((np.abs(r-11.5)<0.5) or (np.abs(r-6.5)<0.5)) +
                 50.0 * (np.abs(r-9)<=2) )
    return value;
def QReed(r):
    value = 0 + 1.0*((r<16) * (r>14))+ 1.0*((r>2) * (r<4)) + 50.0*(np.abs(r-9)<=2)
    return value;

def Timevector(T,dt):
    Time=[dt]
    while Time[-1]<T:
        Time.append(Time[-1]+dt)
```

Homework 4 Code continued on next page...

```
    return(Time)

def diamond_sweep1D(I,hx,q,sigma_t,mu,boundary):
  """Compute a transport diamond difference sweep for a given
  Inputs:
    I:                number of zones
    hx:               size of each zone
    q:                source array
    sigma_t:          array of total cross-sections
    mu:               direction to sweep
    boundary:         value of angular flux on the boundary
  Outputs:
    psi:              value of angular flux in each zone
  """
  assert(np.abs(mu) > 1e-10)
  psi = np.zeros(I)
  ihx = 1./hx
  if (mu > 0):
    psi_left = boundary
    for i in range(I):
      psi_right = (q[i] + (mu*ihx-0.5*sigma_t[i])*psi_left)\
                  /(0.5*sigma_t[i] + mu*ihx)
      psi[i] = 0.5*(psi_right + psi_left)
      psi_left = psi_right
  else:
    psi_right = boundary
    for i in reversed(range(I)):
      psi_left = (q[i]+ (-mu*ihx-0.5*sigma_t[i])*psi_right)\
                 /(0.5*sigma_t[i] - mu*ihx)
      psi[i] = 0.5*(psi_right + psi_left)
      psi_right = psi_left
  return psi

def step_sweep1D(I,hx,q,sigma_t,mu,boundary):
  """Compute a transport step sweep for a given
  Inputs:
    I:                number of zones
    hx:               size of each zone
    q:                source array
    sigma_t:          array of total cross-sections
    mu:               direction to sweep
    boundary:         value of angular flux on the boundary
  Outputs:
    psi:              value of angular flux in each zone
  """
  assert(np.abs(mu) > 1e-10)
  psi = np.zeros(I)
  ihx = 1./hx
  if (mu > 0):
    psi_left = boundary
    psi[0] = 0
    for i in range(1,I):
      psi_right = (q[i] + mu*ihx*psi_left)/(mu*ihx + sigma_t[i])
      psi[i] = 0.5*(psi_right + psi_left)
      psi_left = psi_right
  else:
    psi_right = boundary
    psi[-1] = 0
    for i in reversed(range(0,I-1)):
      psi_left = (q[i] - mu*ihx*psi_right)/(sigma_t[i] - mu*ihx)
      psi[i] = 0.5*(psi_right + psi_left)
      psi_right = psi_left
  return psi


def source_iteration(I,hx,q,sigma_t,sigma_s,N,psipreviousttime,
                     v,dt,Time,BCs,sweep_type,
                     tolerance = 1.0e-8,maxits = 100, LOUD=False ):
  """Perform source iteration for single-group steady state problem
  Inputs:
    I:               number of zones
    hx:              size of each zone
    q:               source array
    sigma_t:         array of total cross-sections
    sigma_s:         array of scattering cross-sections
    N:               number of angles
    BCs:             Boundary conditions for each angle
    sweep_type:      type of 1D sweep to perform solution
    tolerance:       the relative convergence tolerance for the iterations
    maxits:          the maximum number of iterations
    LOUD:            boolean to print out iteration stats
  Outputs:
    x:               value of center of each zone
    phi:             value of scalar flux in each zone
  """
  iterations = []
  Errors = []
  phi = np.zeros(I)
  phi_old = phi.copy()
  converged = False
  MU, W = np.polynomial.legendre.leggauss(N)
  iteration = 1
  tmp_psi=psipreviousttime.copy()
  if len(Time)==1:
      sigma_ts=sigma_t
  else:
      sigma_ts=sigma_t+1/(v*dt)

  while not(converged):
    phi = np.zeros(I)
    #sweep over each direction
    for n in range(N):
      #qs=(q*W[n])/2+(phi_old*sigma_s)/2+psipreviousttime[n,:]/(v*dt)
      qs=(q)/2+(phi_old*sigma_s)/2+psipreviousttime[n,:]/(v*dt)
      if sweep_type == 'dd':
        tmp_psi[n,:] = diamond_sweep1D(I,hx,qs,sigma_ts,MU[n],BCs[n])
      elif sweep_type == 'step':
        tmp_psi[n,:] = step_sweep1D(I,hx,qs,sigma_ts,MU[n],BCs[n])
      else:
        sys.exit("Sweep method specified not defined in SnMethods")
      phi = phi+tmp_psi[n,:]*W[n]
    #check convergence
    change = np.linalg.norm(phi-phi_old)/np.linalg.norm(phi)
    iterations.append(iteration)
    Errors.append(change)
    #iterations.append(iteration)
    #Errors.append(change)
    converged = (change < tolerance) or (iteration > maxits)
    if (LOUD>0) or (converged and LOUD<0):
      print("Iteration",iteration,": Relative Change =",change)
    if (iteration > maxits):
      print("Warning: Source Iteration did not converge : "+\
            sweep_type+", I : "+str(I)+", Diff : %.2e" % change)
    #Prepare for next iteration
    iteration += 1
    phi_old = phi.copy()
  if sweep_type == 'step':
      x = np.linspace(0,(I-1)*hx,I)
  elif sweep_type == 'dd':
      x = np.linspace(hx/2,I*hx-hx/2,I)
  return x, phi, iterations, Errors, tmp_psi


def gmres_solve(I,hx,q,sigma_t,sigma_s,N,psipreviostime,
                v,dt,Time,BCs, sweep_type,
                tolerance = 1.0e-8,maxits = 100, LOUD=False,
                restart = 20 ):
  """Solve, via GMRES, a single-group steady state problem
  Inputs:
    I:                number of zones
    hx:               size of each zone
    q:                source array
    sigma_t:          array of total cross-sections
    sigma_s:          array of scattering cross-sections
    N:                number of angles
    BCs:              Boundary conditions for each angle
    sweep_type:       type of 1D sweep to perform solution
    tolerance:        the relative convergence tolerance for the iterations
    maxits:           the maximum number of iterations
    LOUD:             boolean to print out iteration stats
  Outputs:
    x:                value of center of each zone
    phi:              value of scalar flux in each zone
  """
  iterations = []
  Errors = []

  #compute RHS side
  RHS = np.zeros(I)
```

```
MU, W = np.polynomial.legendre.leggauss(N)
tmp_psi=psiprevioustime.copy()
if len(Time)==1:
    sigma_ts=sigma_t
else:
    sigma_ts=sigma_t+1/(v*dt)

for n in range(N):
  qs=q/2+psiprevioustime[n,:]/(v*dt)
  if sweep_type == 'dd':
    tmp_psi[n,:] = diamond_sweep1D(I,hx,qs,sigma_ts,MU[n],BCs[n])
  elif sweep_type == 'step':
    tmp_psi[n,:] = step_sweep1D(I,hx,qs,sigma_ts,MU[n],BCs[n])
  #tmp_psi = sweep1D(I,hx,q,sigma_t,MU[n],BCs[n])
  RHS += tmp_psi[n,:]*W[n]

#define linear operator for gmres
def linop(phi):
  tmp = phi*0
  #sweep over each direction
  for n in range(N):
    if sweep_type == 'dd':
      tmp_psi[n,:] = diamond_sweep1D(I,hx,(phi*sigma_s)/2,
                                sigma_ts,MU[n],BCs[n])
    elif sweep_type == 'step':
      tmp_psi[n,:] = step_sweep1D(I,hx,(phi*sigma_s)/2,
                                sigma_ts,MU[n],BCs[n])
    tmp += tmp_psi[n,:]*W[n]
  return phi-tmp
A = spla.LinearOperator((I,I), matvec = linop, dtype='d')


#define a little function to call when the iteration is called
iteration = np.zeros(1)
def callback(rk, iteration=iteration):
  iteration += 1
  if (LOUD>0):
    print("Iteration",iteration[0],"norm of residual",np.linalg.norm(rk))
  iterations.append(iteration[0])
  Errors.append(np.linalg.norm(rk))

#Do the GMRES Solve
phi,info = spla.gmres(A,RHS,x0=RHS,tol=tolerance,
                      restart=int(restart),callback=callback)

#Print important information
if (LOUD):
  print("Finished in",iteration[0],"iterations.")
if (info >0):
  print("Warning, convergence not achieved :"+str(sweep_type)+" "+str(hx))
if sweep_type == 'step':
    x = np.linspace(0,(I-1)*hx,I)
    elif sweep_type == 'dd':
        x = np.linspace(hx/2,I*hx-hx/2,I)

    #Calculate Psi for time iterations
    phi2 = np.zeros(I)
    #sweep over each direction
    for n in range(N):
        #qs=(q*W[n])/2+(phi_old*sigma_s)/2+psipreviousstime[n,:]/(v*dt)
        qs=(q)/2+(phi*sigma_s)/2+psipreviousstime[n,:]/(v*dt)
        if sweep_type == 'dd':
            tmp_psi[n,:] = diamond_sweep1D(I,hx,qs,sigma_ts,MU[n],BCs[n])
        elif sweep_type == 'step':
            tmp_psi[n,:] = step_sweep1D(I,hx,qs,sigma_ts,MU[n],BCs[n])
        else:
            sys.exit("Sweep method specified not defined in SnMethods")
        phi2 = phi2+tmp_psi[n,:]*W[n]

    return x, phi, iterations, Errors,tmp_psi

def solver(I,hx,q,Sig_t,Sig_s,N,psi,v,dt,Time,BCs,Scheme,tol,MAXITS,loud):
    Method=Scheme.split(':')[1]
    if "Iteration" in Scheme:
        x, phi, iterations, errors, psi =source_iteration(I,
            hx,q,Sig_t,Sig_s,N,psi,v,dt,Time,BCs,
            Method,tolerance=tol,maxits=MAXITS,LOUD=loud)
    elif "GMRES" in Scheme:
        x, phi, iterations, errors, psi =gmres_solve(I,
            hx,q,Sig_t,Sig_s,N,psi,v,dt,Time,BCs,
            Method,tolerance=tol,maxits=MAXITS,LOUD=loud,restart=MAXITS)
    else:
        print("Improper sweep selected")
        quit()
    return x, phi, iterations, errors,psi

#################################################################
################## Plotting Function ##########################
#################################################################

def reduceList(List,N):
    List2=[List[0]]
    Div=int(len(List)/N)
    for i in range(1,len(List)-1):
        if i % Div == 0:
            List2.append(List[i])
    List2.append(List[-1])
    return(List2)

def loop_values(list1,index):
    """
    This function will loop through values in list even if
    outside range (in the positive sense not negative)
    """
    while True:
        try:
            list1[index]
            break
        except IndexError:
            index=index-len(list1)
    return(list1[index])

def plot(x,y,ax,label,fig,Check,NumOfPoints):
    if len(x)>300:
        x=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
    #Plot X and Y
    ax.plot(x,y,
            linestyle=loop_values(LineStyles,Check),
            marker=loop_values(MarkerType,Check),
            color=loop_values(Colors,Check),
            markersize=loop_values(MarkSize,Check),
            alpha=loop_values(Alpha_Value,Check),
            label=label)

    #Log or linear scale?
    ax.set_xscale(XScale)
    ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,
                                                         ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    ax.set_ylabel(Ylabel,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)
    return(ax,fig)


def plotE(x,y,erax,label,erfig,Check,NumOfPoints):
    if len(x)>300:
        x=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
    #Plot X and Y
    erax.plot(x,y,
            linestyle=loop_values(LineStyles,Check),
            marker=loop_values(MarkerType,Check),
            color=loop_values(Colors,Check),
            markersize=loop_values(MarkSize,Check),
            alpha=loop_values(Alpha_Value,Check),
            label=label)

    #Log or linear scale?
    erax.set_xscale(XScaleE)
    erax.set_yscale(YScaleE)
    #Set Title
    erfig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,
                                                         ha='center')
    #Set X and y labels
    erax.set_xlabel(XlabelE,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    erax.set_ylabel(YlabelE,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)
    return(erax,erfig)

def Legend(ax):
    handles,labels=ax.get_legend_handles_labels()
    ax.legend(handles,labels,loc='best',
              fontsize=LegendFontSize,prop=font)
    return(ax)

# def Legend(ax):
#         handles,labels=ax.get_legend_handles_labels()
#         box=ax.get_position()
#         ax.set_position([box.x0, box.y0, box.width*SquishGraph,
#                          box.height])
#         ax.legend(handles,labels,loc='center',
#                   bbox_to_anchor=(BBOXX,BBOXY),
#                   fontsize=LegendFontSize,prop=font,
#                   ncol=NumberOfLegendColumns)
#         return(ax)
```

Homework 5 Problem Statement

Solve the following problem and submit a detailed report, including a justification of why a reader should believe your results.

Clean Fusion Energy

(100 points) Consider a thermonuclear fusion reactor producing neutrons of energy 14.1 and 2.45 MeV. The reactor is surrounded by FLiBe (a 2:1 mixture of LiF and BeF_2) to convert the neutron energy into heat. All the constituents in the FLiBe have their natural abundances. Using data from JANIS, and assuming the total neutron flux is 10^{14} n/cm^2·s. Perform the following analyses.

(a) (25 points) Write out the depletion (or in thise case activation) chains that will occur in the system.

(b) (50 points) Over a two-year cycle compute the inventory of nuclides in the system using two methods discussed in class. What is the maximum concentration of tritium?

(c) (25 points) After discharging the FLiBe blanket, how long will it take until the material is less radioactive than Brazil Nuts ? (444 Bq/kg)

Homework 5 Background

Please note, that most of this background is copied directly from Dr. McClarren's notes, but are reproduced here.

The production of an isotope is dictated by production and loss

$$\frac{dn_i}{dt} = -\lambda_i^{eff} n_i + \sum_{j=1}^{N} b_{j\to i}^{eff} \lambda_j^{eff} n_j$$

Where,

$$\lambda_i^{eff} = \lambda_i + \phi \sum_{j=1}^{N} \sigma_{i\to j}$$

and

$$b_{j\to i}^{eff} = \frac{b_{j\to i}\lambda_j + \sigma_{j\to i}\phi}{\lambda_j^{eff}}$$

For a system of isotopes, this can be reduced to:

$$\frac{d\vec{n}}{dt} = \boldsymbol{A}\vec{n}(t)$$

Where $\boldsymbol{A}$ is a matrix whose diagonal elements are $[-\lambda_1^{eff}, -\lambda_2^{eff}, ..., -\lambda_N^{eff}]$, all off diagonal elements are $b_{j\to i}^{eff}\lambda_j^{eff}$ (i for the diagonal, and j is for the off diagonal position) and $\vec{n}(t) = [n_1, n_2, ..., n_N]$.

The solution to this system is obvious (it wasn't to me at first - but that's because I'm a newb)

$$\vec{n} = e^{\boldsymbol{A}t}\vec{n}_0$$

Determing $e^{\boldsymbol{A}t}\vec{n}_0$ will be done 3 different ways,

Matrix Exponential

Analytic Solution, unstable with large N.

$$\vec{n}(t) = e^{\boldsymbol{A}t}\vec{n}_0 \approx \left[\sum_{m=0}^{\infty} \frac{1}{m!}\boldsymbol{A}^m t^m\right]\vec{n}_0$$

Backward Euler

Unstable for large Δt, but can take time steps.

$$\begin{aligned}\frac{d\vec{n}}{dt} \approx \frac{\vec{n}(\Delta t) - \vec{n}_0}{\Delta t} &\approx \boldsymbol{A}\vec{n}(\Delta t)\\ \vec{n}(\Delta t) &\approx (\boldsymbol{I} - \boldsymbol{A}\Delta t)^{-1}\vec{n}_0\end{aligned}$$

Rational Approximation

$$\vec{n}(t) = e^{\boldsymbol{A}t}\vec{n}_0 \approx -2\Re\sum_{k=1}^{N/2} c_k(z_k\boldsymbol{I} - \boldsymbol{A}t)^{-1}\vec{n}_0$$

The $\Re$ symbol means taking the real part of the solution. Further,

$$c_k = \frac{i}{N} e^{z_k} w_k$$

where z_k and w_k are both scalers defined as

$$\begin{aligned} z_k =& \phi(\theta_k) \\ w_k =& \phi'(\theta_k) \end{aligned}$$

with

$$\begin{aligned} \phi(\theta) =& N[0.1309 - 0.1194\theta^2 + 0.2500i\theta] \\ & \text{or} \\ \phi(\theta) =& 2.246N\,[1 - sin(1.1721 - 0.3443i\theta] \\ & \text{or} \\ \phi(\theta) =& N[0.5071\theta cot(0.6407\theta) - 0.6122 + 0.2645i\theta] \\ & \text{or} \\ \phi(\theta) =& \text{ Best Possible} \end{aligned}$$

and

$$\theta_k = \pm \frac{\pi}{N}(1 + 2k) \quad \text{k from 0 to N-1}$$

Where N doesn't have to go much higher than 10 to have low errors (for the best Rational Approximation). Also both plus and minus terms were written here, but the first equation in this solution method only uses the positive terms. This is because using the negative β's yields the same real part as the positive β's, with opposite complex parts (the complex cancels). Thats why the $\sum$ only goes to $N/2$ and the solution is multiplied by 2.

The solution utilized the assumption that half the neutron flux was 14 MeV and the other half was 2.45 MeV. The A matrix was built in terms of days and atoms per kg of initial fuel.

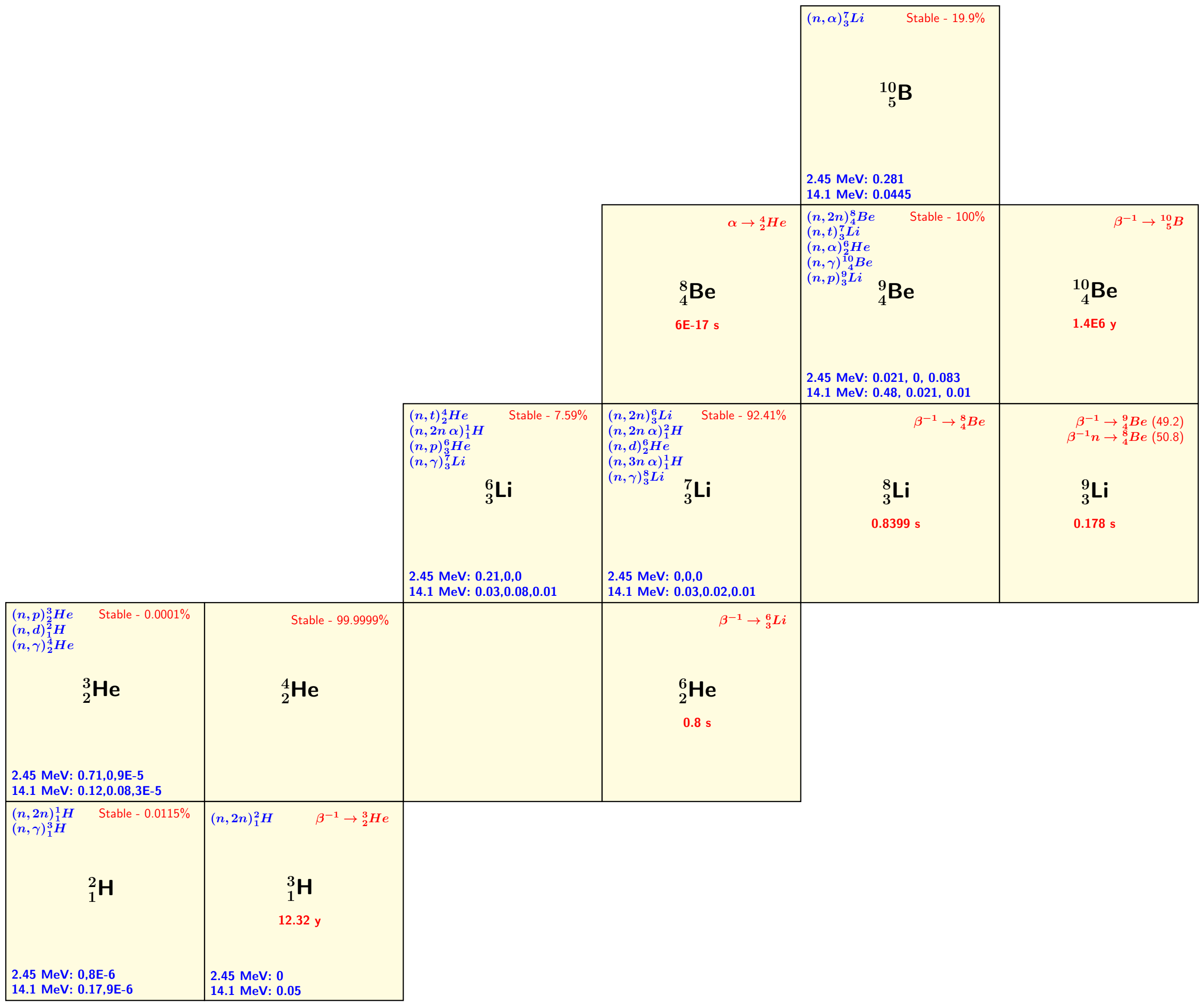
$$1 \text{ Kg FLiBe} \cdot \frac{6.022E23}{98.9 \text{ g}} = 6.09E24 \text{ atoms of FLiBe}$$

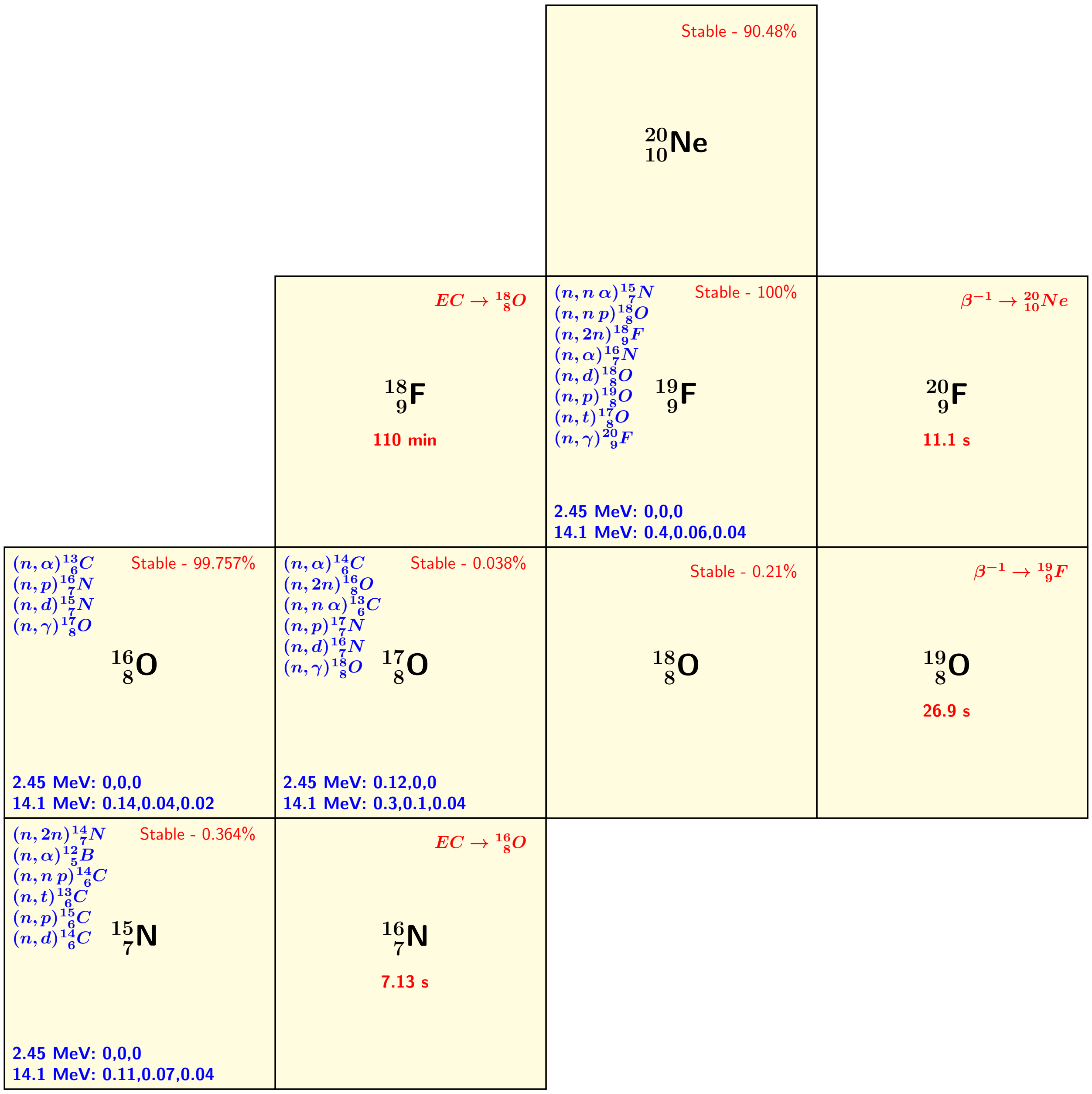
This number was the starting condition for ^{9}Be. Twice this number (with natural abundancy considerations) for ^{6}Li and ^{7}Li, and 4 times this number for ^{19}F.

Homework 5 Solution

(a) (25 points) Write out the depletion (or in thise case activation) chains that will occur in the system.

The chains were written out in a form so that the diagram would be uncluttered, and not all possible reactions are displayed.



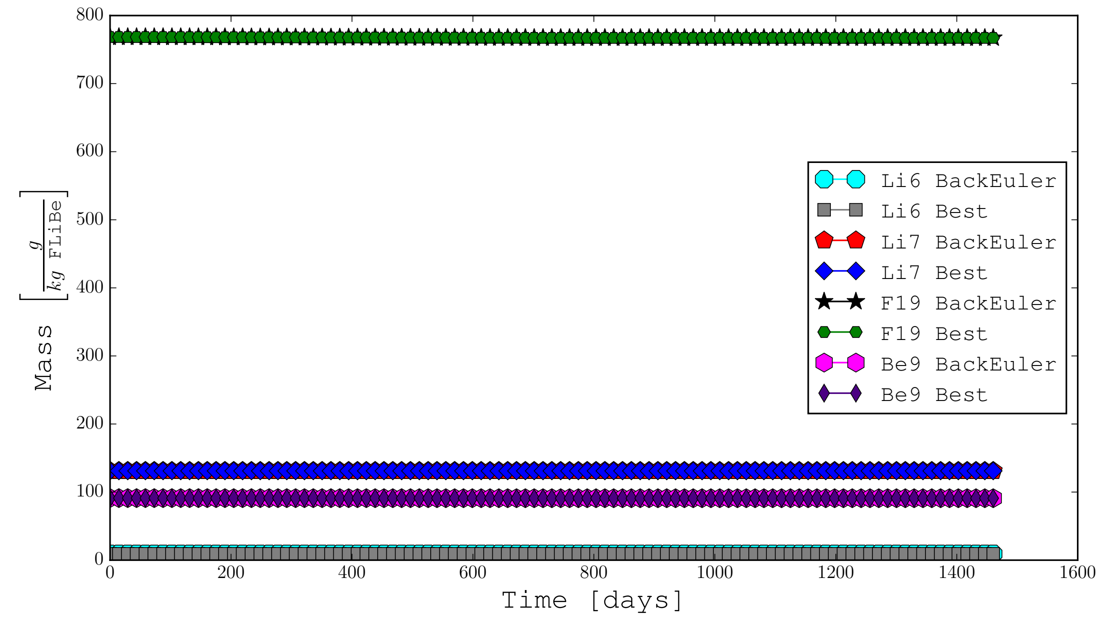


(b) (50 points) Over a two-year cycle compute the inventory of nuclides in the system using two methods discussed in class. What is the maximum concentration of tritium?

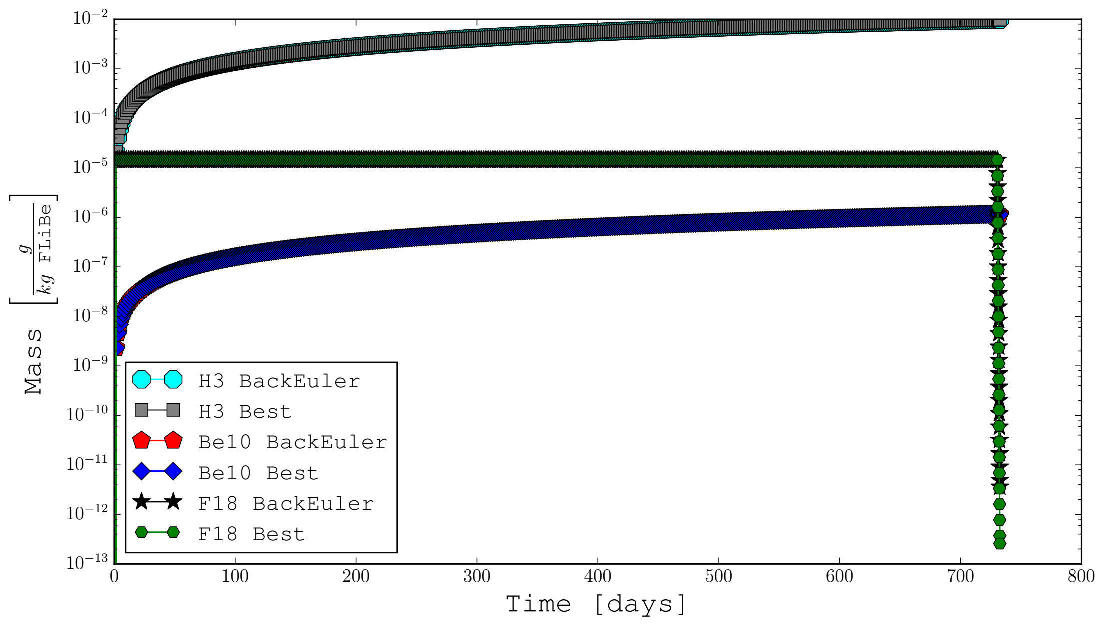
The matrix exponential method did not converge. The Backwards Euler and the Rational approximation solutions did converge. Their answers are basically the same (all plots will have both solutions, but their answers are on top of one another). It should be noted that I changed the solver from in the notes from a GMRES solver to a normal matrix inverter for the rational approach to speed up the algorithm and give better results.

Plots were split up into two groups, with initial nuclides, and product nuclides which would have appreciable activity after a couple of days. Below plots are shown in grams.

The rational approximation used the "Best" method with 10 quadrature points,

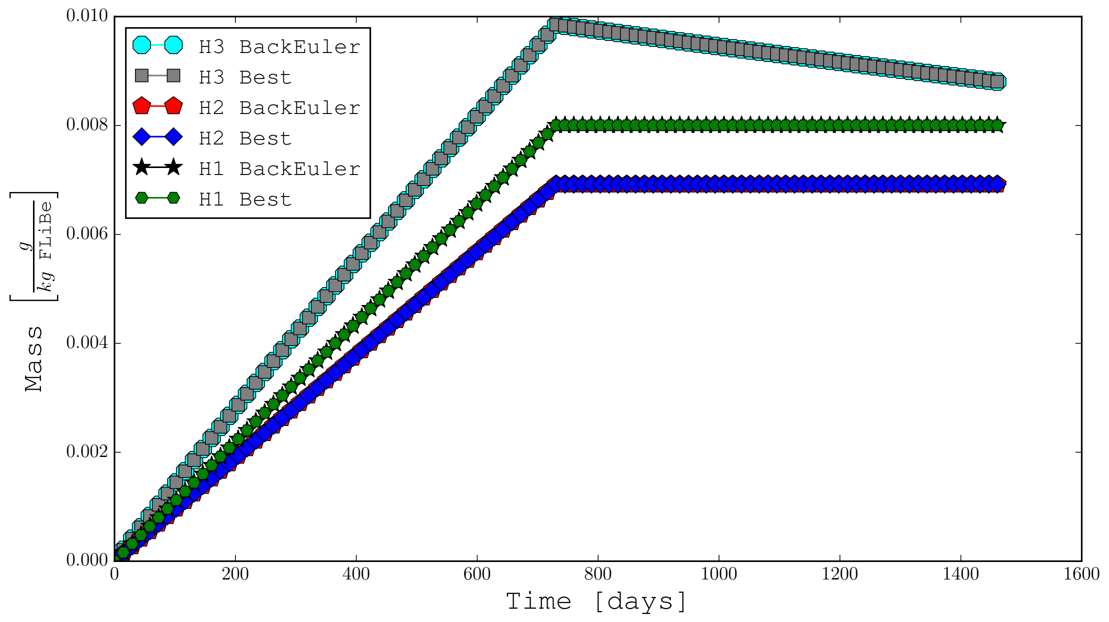


This figure shows that there wasn't sizeable amount of depletion in the coolant. Which is expected because the cross sections are less than a barn.

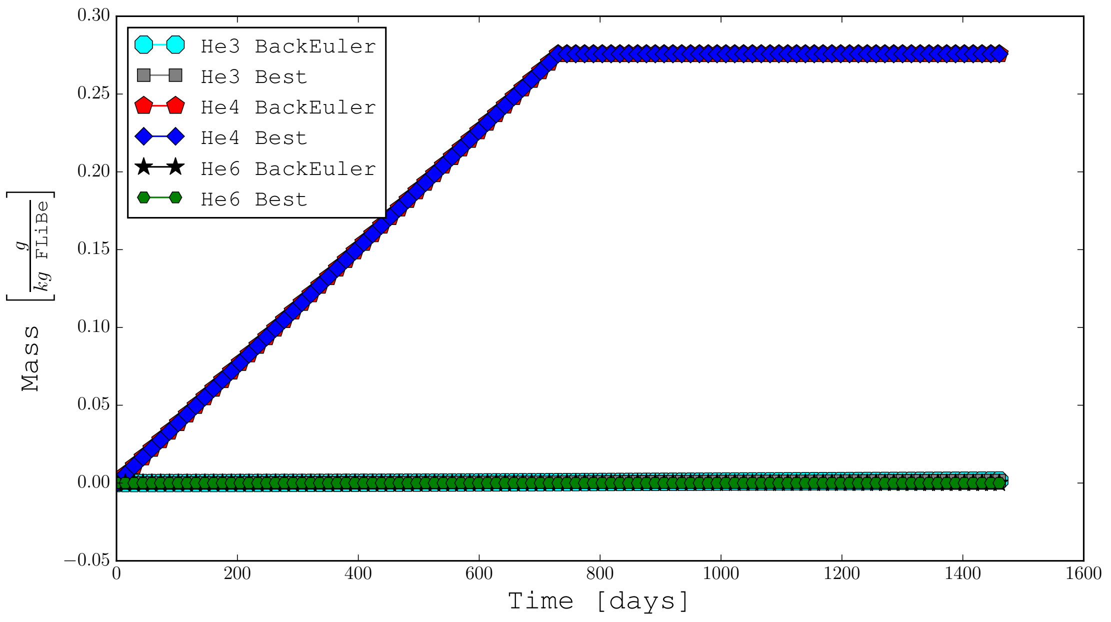


The above figure shows that the high activity elements with large half lives (all other radioactive elements have half lives less than a minute and would decay away and less than 7 minutes) are not produces in a sizeable quantity except for ^{3}H, which has a 12 year half life. The sharp decrease in ^{18}F is because the plot has about a day of decay included to show that activity is primarily from ^{3}H (Beta emitter) and ^{10}Be (alpha emitter).

Below it is shown that the hydrogen is fairly enriched in $^{3}H \approx 40\%$. If the hydrogen were removed, the dose due to the coolant would decrease dramatically but would still have a small amount of activity due to ^{10}Be.

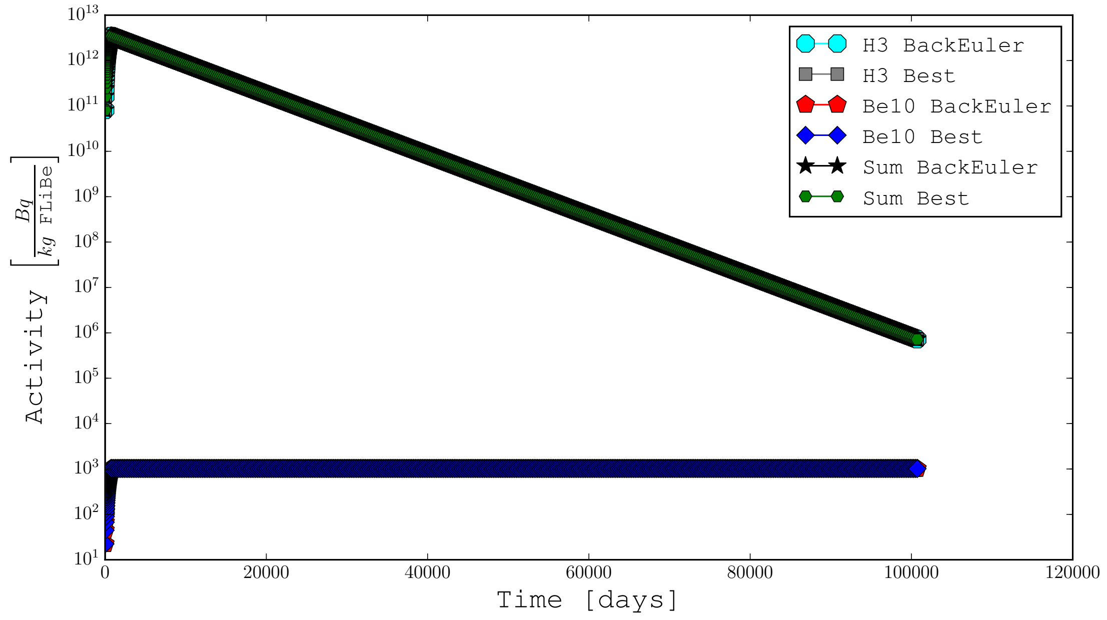


Below we look at the enrichment of ^{3}He. Which is mostly ^{4}He.



(c) (25 points) After discharging the FLiBe blanket, how long will it take until the material is less radioactive than Brazil Nuts? (444 Bq/kg)

In order to use the code to do decay, the A matrix was remade with $\phi = 0$. Looking into this problem without ^{18}F. Also the reason why this plot looks linear is because the y axis has a log scale.



It will take a long time for the ^{10}Be to decay to a point where its less radioactive than nuts.

I decided to do this analytically with

$$t = \frac{-1}{\lambda_Y} ln\left(\frac{444}{N_0\lambda_s}\right)$$

For the ^{3}H to decay to 444 Bq, it will take about 400 years, and the ^{10}Be it will take 1.8E6 years, which is a long time, and slightly over one half life for the ^{10}Be.

Homework 5 Code

In order to build the A matrix I utilized code from James Tompkins that he let me see a while back. I wanted to give him credit because building the A matrix is probably one of the harder parts of this homework. I had my own experience in building a much larger A matrix for the next problem.

Listing 5: **Main Code**

```
#!/usr/bin/env python3

################################################################
#################### Import packages #########################
################################################################

import time
start_time = time.time()
import Functions as f

################################################################
#################### Initialize System #######################
################################################################

NumOfPoints=200          # Max Number of points for plots
PlottingG1=["Li6","Li7","F19","Be9"]  #List elements you want to Plot
CompareG1=["Li6","Li7","F19","Be9"]
PlottingG2=["H3","Be10"]  #List elements you want to Plot
CompareG2=["H3","Be10"]
#List elements you want to compare between methods

#########################################################
############ Initialize Matrix ########################
#########################################################

high_flux_fraction=0.5
phi=1.0e14
A,n0=f.MakeAb(high_flux_fraction,phi)

if not A.shape[0] == A.shape[1] or not A.shape[0] == len(n0):
    print("A is not a square matrix")
    quit()

#########################################################
############ Initialize Time ##########################
#########################################################

t=730.5;  #Two years in days
Nt=1000;   #Number of Time Steps
dt=t/Nt;
Time=f.np.linspace(dt,t,Nt) #Time steps

tDecay=100000; #No time of decay
NtDecay=10000; #Number of Decay Time Steps
dtDecay=tDecay/NtDecay
TimeDecay=f.np.linspace(t+dtDecay,t+tDecay,NtDecay)
```

```
#################################################################
##################### Solve System ############################
################ And Time How long it takes ###################
################### For Each Method ###########################
################# And Plot at Each Time Step ##################
#################################################################

#Matrix Exp - Unstable
#Current_Time=time.time()
#maxits=20
# nt_Mat=f.MatExp(A,n0,t,maxits)
# Mat_Time=time.time()-Current_Time
#nt_Mat=n0.copy();TIMEOLD=0
#for TIME in Time[1:len(Time)]: #Unstable
    #nt_Mat=f.MatExp(A,n0,TIME,maxits) No Converge
    #Step through
    #nt_Mat=f.MatExp(A,nt_Mat,TIME-TIMEOLD,maxits)
    #TIMEOLD=TIME.copy()
#nt_Mat=f.RationalApprox(A,n0,t,maxits) #one Step
#Mat_Time=time.time()-Current_Time

##########################################################
############### Backward Euler ########################
##########################################################

Current_Time=time.time()
File=f.PrepFile('BackEuler.csv',n0) #Prep File
nt_Back=n0.copy();
#Irradiation Time
for TIME in Time:
    nt_Back=f.BackEuler(A,nt_Back,dt)
    File.write(str(TIME)+","+f.ListToStr(nt_Back))
#Decay Time
phi=0
A,n0=f.MakeAb(high_flux_fraction,phi)
for TIME in TimeDecay:
    nt_Back=f.BackEuler(A,nt_Back,dtDecay)
    File.write(str(TIME)+","+f.ListToStr(nt_Back))

File.close()
Back_Time=time.time()-Current_Time

##########################################################
############### Rational Approx #######################
##########################################################

#Reset A and n0
high_flux_fraction=0.5
phi=1.0e14
A,n0=f.MakeAb(high_flux_fraction,phi)

#Irradiation Time
```

```
Current_Time=time.time()
N=10;
Method="Best"  #Parabola, Cotangent, Hyperbola, Best
File=f.PrepFile(Method+".csv",n0)
ck,zk=f.RationalPrep(N,Method);nt_Rational=n0.copy();TIMEOLD=0
for TIME in Time:
    nt_Rational=f.RationalApprox(A,n0,TIME,N,ck,zk)
    File.write(str(TIME)+","+f.ListToStr(nt_Rational))
    #Step through
    #nt_Rational=f.RationalApprox(A,nt_Rational,TIME-TIMEOLD,N,ck,zk)
    #TIMEOLD=TIME.copy()
#nt_Rational=f.RationalApprox(A,n0,t,N,ck,zk) #one Step

#Non Irradiation Time
phi=0
A,n0=f.MakeAb(high_flux_fraction,phi)
n0=nt_Rational
for TIME in TimeDecay:
    nt_Rational=f.RationalApprox(A,n0,TIME-t,N,ck,zk)
    File.write(str(TIME)+","+f.ListToStr(nt_Rational))

File.close()
Rational_Time=time.time()-Current_Time

#########################################################
################# Plot Solution ########################
################# In Grams #############################
################# and activity #########################
#########################################################

dfBack = f.pd.read_csv('BackEuler.csv',index_col=False)
dfRational = f.pd.read_csv(Method+".csv",index_col=False)

# #Plot group 1 dudes Back Euler method
#f.plot(dfBack,PlottingG1,'BackEulerG1',NumOfPoints)
# f.plot(dfRational,PlottingG1,Method+'G1',NumOfPoints)

# #Plot group 2 dudes Rational method
#f.plot(dfBack,PlottingG2,'BackEulerG2',NumOfPoints)
f.plot(dfRational,PlottingG2,Method+'G2_Check',NumOfPoints)

# #Plot group 1 dudes, compare both methods
# Name='BackEuler_'+Method+"_G1"
# f.plots2(dfBack,dfRational,CompareG1,Name,
#          NumOfPoints,'BackEuler',Method)

#Plot group 2 dudes, compare both methods
# Name='BackEuler_'+Method+"_G2_NoF"
# f.plots2(dfBack,dfRational,CompareG2,Name,
#          NumOfPoints,'BackEuler',Method)

################################################################
################## Print Solution ##############################
```

```
################################################################

#f.Print("Matrix Exp","H3",nt_Mat,Mat_Time)
f.Print("Backward Euler","H3",nt_Back,Back_Time)
f.Print("Rational Approx ","H3",nt_Rational,Rational_Time)

f.Print("Backward Euler","Be10",nt_Back,Back_Time)
f.Print("Rational Approx ","Be10",nt_Rational,Rational_Time)

## Time in years to be below 444 bq
f.Years("Backward Euler","H3",nt_Back)
f.Years("Rational Approx ","H3",nt_Rational)

f.Years("Backward Euler","Be10",nt_Back)
f.Years("Rational Approx ","Be10",nt_Rational)



###################### Time To execute ################

print("--- %s seconds ---" % (time.time() - start_time))
```

Listing 6: **Functions holder**

```
#!/usr/bin/env python3

################################################################
#################### Import packages #########################
################################################################

import sys
import numpy as np
import scipy.sparse as sparse
import scipy.sparse.linalg as spla
import scipy.linalg as scil
import scipy.special as sps
import matplotlib.pyplot as plt
plt.rcParams["font.family"] = "monospace"
import matplotlib
matplotlib.rc('text',usetex=True)
matplotlib.rcParams['text.latex.preamble']=[r"\usepackage{amsmath}"]
import random as rn
import matplotlib.mlab as mlab
import copy
import os
import pandas as pd

#############################################################
######################## Variables ########################
#############################################################

# Basic information
FigureSize = (11, 6)              # Dimensions of the figure
TypeOfFamily='monospace'          # This sets the type of font for text
```

```
font = {'family' : TypeOfFamily}  # This sets the type of font for text
LegendFontSize = 12
Lfont = {'family' : TypeOfFamily}  # This sets up legend font
Lfont['size']=LegendFontSize

Title = ''
TitleFontSize = 22
TitleFontWeight = "bold"  # "bold" or "normal"

#Xlabel='E (eV)'   # X label
XFontSize=18          # X label font size
XFontWeight="normal"  # "bold" or "normal"
XScale="linear"       # 'linear' or 'log'

YFontSize=18                    # Y label font size
YFontWeight="normal"            # "bold" or "normal"
YScale="linear"                 # 'linear' or 'log'

Check=0


Colors=["aqua","gray","red","blue","black",
                "green","magenta","indigo","lime","peru","steelblue",
                "darkorange","salmon","yellow","lime","black"]

# If you want to highlight a specific item
# set its alpha value =1 and all others to 0.4
# You can also change the MarkSize (or just use the highlight option below)
Alpha_Value=[1  ,1  ,1  ,1  ,1  ,1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
MarkSize=   [8  ,8  ,8  ,8  ,8  ,8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8]

Linewidth=[1  ,1  ,1  ,1  ,1  ,1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]

# Can change all these to "." or "" for nothing "x" isn't that good
MarkerType=["8","s","p","D","*","H","h","d","^",">"]

# LineStyles=["solid","dashed","dash_dot","dotted","."]
LineStyles=["solid"]

SquishGraph = 0.75
BBOXX = 1.24
BBOXY = 0.5       # Set legend on right side of graph

NumberOfLegendColumns=1

Xlabel='Time [days]'
Ylabel="Mass $\\left[\\frac{g}{kg \\text{ FLiBe}}\\right]$"


nuclides = {  'H1':0,     'H2':1,     'H3':2,  'He3':3,  'He4':4,
              'He6':5,    'Li6':6,    'Li7':7,  'Li8':8,  'Be8':9,
              'Be9':10, 'Be10':11, 'Be11':12, 'B10':13, 'B11':14,
              'B12':15,  'C12':16,  'C13':17, 'C14':18, 'C15':19,
               'N13':20,  'N14':21,  'N15':22, 'N16':23, 'N17':24,
               'O16':25,  'O17':26,  'O18':27, 'O19':28, 'F18':29,
               'F19':30, 'F20':31, 'Ne20':32}

atom_mass = np.array([1.007825032,2.014101778,3.0160492779,  #2
                      3.016029320,4.002603254,6.151228874,   #5
                      6.015122887,7.0160034366,8.022486246,  #8
                      8.005305102,9.012183065,10.013534695,  #11
                      11.02166108,10.01293695,11.00930536,   #14
                      12.0269221, 12, 13.003354835,          #17
                      14.003241988, 15.01059926,13.00573861, #20
                      14.003074004, 15.000108898, 16.0061019,#23
                      17.008449, 15.994914619, 16.999131756, #26
                      17.999159612, 19.0035780,17.99915961286,#29
                      18.998403162, 19.999981252, 19.992440176])

nuclide_names = ('H1', 'H2', 'H3', 'He3', 'He4', 'He6', 'Li6',
                 'Li7', 'Li8', 'Be8', 'Be9', 'Be10',
                 'Be11', 'B10', 'B11', 'B12', 'C12', 'C13',
                 'C14', 'C15', 'N13', 'N14', 'N15', 'N16',
                 'N17', 'O16', 'O17', 'O18', 'O19', 'F18',
                 'F19', 'F20', 'Ne20')

decay_consts = np.array([0., 0., np.log(2)/3.887896E8, #H1 H2 H3
                         0., 0., np.log(2)/0.807, #He3 He4 He6
                         0.,0., np.log(2)/0.840,  #Li6 #Li7 #Li8
                         np.log(2)/6E-17,0.,    #Be8 #Be9
                         np.log(2)/4.73364E13,np.log(2)/13.8, # Be10,11
                         0., 0., np.log(2)/0.0202, #B10 B11 B12
                         0., 0.,np.log(2)/1.803517E11, #C12 C13 C14
                         np.log(2)/2.45,np.log(2)/598.2, #C15 N13
                         0., 0., np.log(2)/7.13, # N14 N15 N16
                         np.log(2)/4.174, 0., 0., 0., #N17 O16 O17 O18
                         np.log(2)/26.9, np.log(2)/6586.2, #O19 F18
                         0., np.log(2)/11.1, 0.]) #F19 F20 Ne20
Na=6.0221409E23

#################################################################
###################### Functions ################################
#################################################################

def MatExp(A,n0,t,maxits,tolerance=1e-12,LOUD=False):

  converged = False
  m=0
  sum_old=n0.copy()*0

  while not(converged):

    if m==0:
        APowerm=np.identity(A.shape[0])
        Factorial=1
    else:
```

```python
            APowerm=np.dot(APowerm,A)
            Factorial=Factorial*m
        Sum=sum_old+(1/Factorial)*np.dot((APowerm)*(t**m),n0)

        #Avoid dividing by zero
        if sum(Sum)==0: m+=1;sum_old=Sum.copy();continue

        change = np.linalg.norm(Sum-sum_old)/np.linalg.norm(Sum)
        converged = (change < tolerance) or (m > maxits)

        if (LOUD>0) or (converged and LOUD<0):
          print("Iteration",m,": Relative Change =",change)
        if (m > maxits):
          print("Warning: Source Iteration did not converge : "+\
                " m : "+str(m)+", Diff : %.2e" % change)
        #Prepare for next iteration
        m += 1
        sum_old = Sum.copy()

      return(Sum)

    def BackEuler(A,no,dt):
        I=np.identity(A.shape[0])
        return(np.dot(np.linalg.inv(I-A*dt),no))

    def DeterminePolesNResidues(n):
        """
        This program takes the algorithm from the reference
        and converts to a python script...I know its janky
        but it works
        """
        def Append(List1,List2):
            for item in List2:
                for item2 in item:
                    List1=np.append(List1,item2)
            return(List1)
        def absG(List):
            List2=[]
            for item in List:
                if abs(item)>1:
                    List2.append(item)
            return(List2)
        #function [zk,ck] = cf(n);
        K = 75;                                # no of Cheb coeffs
        nf = 1024;                             # no of pts for FFT
        #Roots correct?
        roots=np.arange(0,nf,1)/nf
        #w = np.exp(2i*pi*(0:nf-1)/nf);        # roots of unity

        w=np.exp(2j*np.pi*roots)
        t = np.real(w);                        # Cheb pts (twice over)
        scl = 9;                               # scale factor for stability
        #F = np.exp(scl*(t-1)./(t+1+1e-16)); # exp(x) transpl. to [-1,1]
        F = np.exp(scl*(t-1)/(t+1+1e-16)); # exp(x) transpl. to [-1,1]
        c = np.real(np.fft.fft(F))/nf;      # Cheb coeffs of F
        index=reversed(np.arange(1,K+2,1))
        partofc=[]
        for i in index:
            partofc.append(c[i-1])
        #f = np.polyval(c(K+1:-1:1),w);      # analytic part f of F
        f = np.polyval(partofc,w);      # analytic part f of F

        #[U,S,V] = svd(hankel(c(2:K+1)));    # SVD of Hankel matrix
        hankie=scil.hankel(c[1:K+1])
        U,S,V=np.linalg.svd(hankie,full_matrices=False)

        #s = S(n+1,n+1);                     # singular value
        s=S[n]
        #u = U(K:-1:1,n+1); v = V(:,n+1);  # singular vector
        u=[]
        index=reversed(np.arange(0,K,1))
        for i in index:
            u.append(U[i,n])
        #v=np.array(V[:,n].copy())
        v=np.array(V[n,:].copy())
        #zz = zeros(1,nf-K);                 # zeros for padding
        zz=np.zeros([1,nf-K])
        #b = fft([u zz])./fft([v zz]);       # finite Blaschke product
        b=np.fft.fft(Append(u,zz))/np.fft.fft(Append(v,zz))
        #rt = f-s*w.^K.*b;                   # extended function r-tilde
        rt=f-s*(w**K)*b;
        #rtc = real(fft(rt))/nf;             # its Laurent coeffs
        rtc=np.real(np.fft.fft(rt))/nf;
        #zr = roots(v); qk = zr(abs(zr)>1);  # poles
        zr=np.roots(v);qk=np.array(absG(zr));
        #qc = poly(qk);                      # coeffs of denominator
        qc=np.poly(qk);
        #pt = rt.*polyval(qc,w);             # numerator
        pt=rt*np.polyval(qc,w);
        #ptc = real(fft(pt)/nf);             # coeffs of numerator
        ptc=np.real(np.fft.fft(pt)/nf);
        #ptc = ptc(n+1:-1:1); ck = 0*qk;
        index=reversed(np.arange(0,n+1,1))
        ptc2=[]
        for i in index:  #Can I just reversed ptc?
            ptc2.append(ptc[i])
        ptc=ptc2.copy()
        ck=0*qk
        #N+1?
        #for k =1:n               # calculate residues
        #    q = qk(k); q2 = poly(qk(qk~=q));
        #    ck(k) = polyval(ptc,q)/polyval(q2,q);
        for k in range(0,n):
            if len(qk)==k:
                print("we are short a qk")
                continue
            q=qk[k];
            q2=[];
            for item in qk:
                if not q==item:
                    q2.append(item)
            q2=np.poly(q2);
            ck[k]=np.polyval(ptc,q)/np.polyval(q2,q)
        #zk = scl*(qk-1).^2./(qk+1).^2;       # poles in z-plane
        zk=scl*((qk-1)**2)/((qk+1)**2)
        #ck = 4*ck.*zk./(qk.^2-1);            # residues in z-plane
        ck=4*ck*zk/(qk**2-1)
        #Cut down ck and zk to half the original points
        ck2=[];zk2=[]
        for i in range(0,len(ck)):
            if i % 2 == 0:
                ck2=np.append(ck2,ck[i])
                zk2=np.append(zk2,zk[i])

        return(ck2,zk2)

    def RationalPrep(N,Phi):
        """Calculate constants for a rational approximation
        Inputs:
        N:                Number of Quadrature points
        Phi:              'Parabola',
                          'Cotangent', or
                          'Hyperbola' (shape of Phi)

        Outputs:
        ck:               First set of constants for approximation
        zk:              Second set of constants for approximation
        """
        theta=np.pi*np.arange(1,N,2)/N
        if Phi=='Parabola':
            zk=N*(0.1309-0.1194*theta**2+0.2500j*theta)
            w=N*(-2*0.1194*theta+0.2500j)
        elif Phi=='Cotangent':
            cot=1/np.tan(0.6407*theta)
            ncsc=-0.6407/(np.sin(0.6407*theta)**2)
            zk=N*(0.5017*theta*cot-0.6122+0.2645j*theta)
            w=N*(0.2645j+0.5017*cot+0.5017*theta*ncsc)
        elif Phi=='Hyperbola':
            zk=2.246*N*(1-np.sin(1.1721-0.3443j*theta))
            w=2.246*N*(0.3443j*np.cos(1.1721-0.3443j*theta))
        elif Phi=='Best':
            ck,zk=DeterminePolesNResidues(N)
            return(ck,zk)
        else:
            print("Did not pick proper rational approximation dude")
            print("Quiting now")
            quit()

        ck=1.0j/N*np.exp(zk)*w
        return(ck,zk)

    def RationalApprox(A,n0,t,N,ck,zk,tol=1e-12,maxits=2000):
        """
        Calculate the rational approximation solution for n(t)
        Inputs:
        A:          Matrix with system to be solved
        n0:         initial conditions of the system
        t:          time at which solution is determined
        N:          Number of quadrature points (should be less than 20)
        ck:         constants for quadrature solution
        zk:         constants for quadrature solution
        tol:        Tolerence for convergence for GMRES
        maxits:     Maximium iterations for GMRES
        Outputs:
        nt:         Solution at time t
        """
        nt=np.zeros(len(n0))
        for k in range(int(N/2)):
            if len(n0)>1:
                #phi,code=spla.gmres(zk[k]*sparse.identity(len(n0))-A*t,n0,
                #                     tol=tol,maxiter=maxits)
                phi=np.dot(np.linalg.inv(zk[k]*np.identity(len(n0))-A*t),
                           n0)
                #if (code):
                #    print(code)
            else:
                phi=(zk[k]-A*t)**(-1)*n0
            nt=nt-2*np.real(ck[k]*phi)
        return(nt)

    def MakeAb(hi_flux_frac = 0.5,phi = 1.0e14):

        """Interaction functions
        @ In, nuclides:  dictionary with isotope keywords and
                         corresponding indices
        @ In, parent:    parent nuclides undergoing a decay or interaction
        @Out, value:     new value in interaction matrix, either a half
                         life [secs] or 2.45 MeV and 14.1 MeV cross
                         sections [barns]
        """

        def betanegdecay(nuclides, parent):
            if   parent == 'F20':  return nuclides['Ne20'], 11.1 # s
            elif parent == 'O19':  return nuclides['F19'],  26.9 # s
            elif parent == 'N16':  return nuclides['O16'],  7.13 # s
            elif parent == 'N17':  return nuclides['O17'],  4.174 # s
            elif parent == 'C14':  return nuclides['N14'],  1.803517E11 # s
            elif parent == 'C15':  return nuclides['N15'],  2.45 # s
            elif parent == 'B12':  return nuclides['C12'],  0.0202 # s
            elif parent == 'Be10': return nuclides['B10'],  4.73364E13 # s
            elif parent == 'Be11': return nuclides['B11'],  13.8 # s
            elif parent == 'Li8':  return nuclides['Be8'],  0.840 # s
            elif parent == 'He6':  return nuclides['Li6'],  0.807 # s
            elif parent == 'H3':   return nuclides['He3'],  3.887896E8 # s
            else: return -1, 0.0

        def betaposdecay(nuclides, parent):
            if   parent == 'F18':  return nuclides['O18'],  6586.2 # s
            elif parent == 'N13':  return nuclides['C13'],  598.2 # s
            else: return -1, 0.0

        def twoalphadecay(nuclides, parent):
            if   parent == 'Be8':  return nuclides['He4'],  7.0E-17 # s
            else: return -1, 0.0

        def n_gamma(nuclides, parent):
            if   parent == 'F19':
                return nuclides['F20'],  8.649107E-5,    3.495035E-5
            elif parent == 'O16':
                return nuclides['O17'],  1.0E-4,         1.0E-4
            elif parent == 'O17':
                return nuclides['O18'],  2.2675E-4,      2.087114E-4
            elif parent == 'N14':
                return nuclides['N15'],  2.397479E-5,    1.679535E-5
            elif parent == 'N15':
                return nuclides['N16'],  8.121795E-6,    8.56E-6
            elif parent == 'Be9':
                return nuclides['Be10'],1.943574E-6,    1.660517E-6
            elif parent == 'Li6':
                return nuclides['Li7'],  1.106851E-5,    1.017047E-5
            elif parent == 'Li7':
                return nuclides['Li8'],  4.677237E-6,    4.105546E-6
            elif parent == 'He3':
                return nuclides['He4'],  9.28775E-5,     3.4695E-5
            elif parent == 'H2':
                return nuclides['H3'],   8.413251E-6,    9.471512E-6
            else:
                return -1, 0.0, 0.0


        def n_2n(nuclides, parent):
            if   parent == 'F19':
                return nuclides['F18'],  0.0,            0.04162
            elif parent == 'O17':
                return nuclides['O16'],  0.0,            0.066113
            elif parent == 'N14':
                return nuclides['N13'],  0.0,            0.006496
            elif parent == 'N15':
                return nuclides['N14'],  0.0,            0.112284
            elif parent == 'B11':
                return nuclides['B10'],  0.0,            0.018805
            elif parent == 'Be9':
                return nuclides['Be8'],  0.0205,         0.484483
            elif parent == 'Li7':
                return nuclides['Li6'],  0.0,            0.031603
```

```python
        elif parent == 'H3':
            return nuclides['H2'],   0.0,            0.0497
        elif parent == 'H2':
            return nuclides['H1'],   0.0,            0.166767
        else:
            return -1, 0.0, 0.0

    def n_alpha(nuclides, parent):
        if   parent == 'F19':
            return [nuclides['N16'],nuclides['He4']],2.1667E-5,0.028393
        elif parent == 'O16':
            return [nuclides['C13'], nuclides['He4']], 0.0, 0.144515
        elif parent == 'O17':
            return [nuclides['C14'], nuclides['He4']],0.117316,0.260809
        elif parent == 'N14':
            return [nuclides['B11'], nuclides['He4']],0.104365,0.080516
        elif parent == 'N15':
            return [nuclides['B12'], nuclides['He4']], 0.0,0.069240
        elif parent == 'B10':
            return [nuclides['Li7'], nuclides['He4']],0.281082,0.044480
        elif parent == 'B11':
            return [nuclides['Li8'], nuclides['He4']], 0.0,0.031853
        else:
            return [-1,-1], 0.0, 0.0

    def n_2alpha(nuclides, parent):
        if   parent == 'N14':
            return [nuclides['Li7'], nuclides['He4']],  0.0,0.031771
        elif parent == 'B10':
            return [nuclides['H3'],  nuclides['He4']],0.038439,0.095487
        else:
            return [-1,-1], 0.0, 0.0

    def n_nalpha(nuclides, parent):
        if   parent == 'F19':
            return [nuclides['N15'], nuclides['He4']], 0.0,0.3818
        elif parent == 'O17':
            return [nuclides['C13'], nuclides['He4']], 0.0,0.043420
        elif parent == 'N15':
            return [nuclides['B11'], nuclides['He4']], 0.0,0.012646
        elif parent == 'B11':
            return [nuclides['Li7'], nuclides['He4']], 0.0,0.286932
        elif parent == 'Be9':
            return [nuclides['He6'], nuclides['He4']], 0.0825,0.0104
        else:
            return [-1,-1], 0.0, 0.0

    def n_2nalpha(nuclides, parent):
        if   parent == 'Li6':
            return [nuclides['H1'], nuclides['He4']],  0.0,0.0783
        elif parent == 'Li7':
            return [nuclides['H2'], nuclides['He4']],  0.0,0.020195
        else:
            return [-1,-1], 0.0, 0.0

    def n_3nalpha(nuclides, parent):
        if   parent == 'Li7':
            return [nuclides['H1'], nuclides['He4']],  0.0,6.556330E-5
        else:
            return [-1,-1], 0.0, 0.0

    def n_p(nuclides, parent):
        if   parent == 'F19':
            return [nuclides['O19'],  nuclides['H1']], 0.0,0.018438
        elif parent == 'O16':
            return [nuclides['N16'],  nuclides['H1']], 0.0,0.042723
        elif parent == 'O17':
            return [nuclides['N17'],  nuclides['H1']], 0.0, 0.041838
        elif parent == 'N14':
            return [nuclides['C14'], nuclides['H1']],0.014102,0.043891
        elif parent == 'N15':
            return [nuclides['C15'],  nuclides['H1']], 0.0,0.019601
        elif parent == 'B10':
            return [nuclides['Be10'], nuclides['H1']],0.018860,0.034093
        elif parent == 'B11':
            return [nuclides['Be11'], nuclides['H1']], 0.0,0.005564
        elif parent == 'Li6':
            return [nuclides['He6'],  nuclides['H1']], 0.0,0.00604
        elif parent == 'He3':
            return [nuclides['H3'],nuclides['H1']],0.714941, 0.121
        else:
            return [-1,-1], 0.0, 0.0

    def n_np(nuclides, parent):
        if   parent == 'F19':
            return [nuclides['O18'],nuclides['H1']], 0.0, 0.061973
        elif parent == 'N15':
            return [nuclides['C14'],  nuclides['H1']], 0.0, 0.044827
        elif parent == 'B11':
            return [nuclides['Be10'], nuclides['H1']], 0.0, 0.001016
        else:
            return [-1,-1], 0.0, 0.0

    def n_d(nuclides, parent):
        if   parent == 'F19':
            return [nuclides['O18'], nuclides['H2']],  0.0, 0.022215
        elif parent == 'O16':
            return [nuclides['N15'], nuclides['H2']],  0.0,0.017623
        elif parent == 'O17':
            return [nuclides['N16'], nuclides['H2']],  0.0,0.020579
        elif parent == 'N14':
            return [nuclides['C13'], nuclides['H2']],  0.0, 0.042027
        elif parent == 'N15':
            return [nuclides['C14'], nuclides['H2']],  0.0,0.014926
        elif parent == 'B10':
            return [nuclides['Be9'], nuclides['H2']],  0.0, 0.031270
        elif parent == 'Li7':
            return [nuclides['He6'], nuclides['H2']],  0.0, 0.010199
        elif parent == 'He3':
            return [nuclides['H2'],  nuclides['H2']],  0.0,0.07609
        else:
            return [-1,-1], 0.0, 0.0

    def n_t(nuclides, parent):
        if   parent == 'F19':
            return [nuclides['O17'], nuclides['H3']],  0.0,0.01303
        elif parent == 'N14':
            return [nuclides['C12'], nuclides['H3']],  0.0,0.028573
        elif parent == 'N15':
            return [nuclides['C13'], nuclides['H3']],  0.0,0.020163
        elif parent == 'B11':
            return [nuclides['Be9'], nuclides['H3']],  0.0,0.015172
        elif parent == 'Be9':
            return [nuclides['Li7'], nuclides['H3']],  0.0,0.020878
        elif parent == 'Li6':
            return [nuclides['He4'], nuclides['H3']],  0.206155,0.0258
        else:
            return [-1,-1], 0.0, 0.0


    # Create Activation and Decay Matrix and initial
    # nuclide quantity vector
    A = np.zeros((len(nuclides),len(nuclides)))

    lo_flux_frac = (1.0-hi_flux_frac)

    phi = phi * 60 * 60 * 24 #10^14 1/cm^2/s in 1/cm^2 /day
    phi_hi = hi_flux_frac*phi*1.0e-24
    phi_lo = lo_flux_frac*phi*1.0e-24


    for isotope in nuclides:
        row = nuclides[isotope]
        row_betanegdecay =  betanegdecay(nuclides, isotope)
        row_betaposdecay =  betaposdecay(nuclides, isotope)
        row_2alphadecay =   twoalphadecay(nuclides, isotope)
        row_n_gamma =       n_gamma(nuclides, isotope)
        row_n_2n =          n_2n(nuclides, isotope)
        row_n_alpha =       n_alpha(nuclides, isotope)
        row_n_2alpha =      n_2alpha(nuclides, isotope)
        row_n_nalpha =      n_nalpha(nuclides, isotope)
        row_n_2nalpha =     n_2nalpha(nuclides, isotope)
        row_n_3nalpha =     n_3nalpha(nuclides, isotope)
        row_n_p =           n_p(nuclides, isotope)
        row_n_np =          n_np(nuclides, isotope)
        row_n_d =           n_d(nuclides, isotope)
        row_n_t =           n_t(nuclides, isotope)
        row_lo_act_sum = row_n_gamma[1] + row_n_2n[1] +\
                         row_n_alpha[1] + row_n_2alpha[1] +\
                          row_n_nalpha[1] + row_n_2nalpha[1] + \
                          row_n_3nalpha[1] + row_n_p[1] +\
                          row_n_np[1] + row_n_d[1] +\
                          row_n_t[1]
        row_hi_act_sum = row_n_gamma[2] + row_n_2n[2] +\
                          row_n_alpha[2] + row_n_2alpha[2] +\
                          row_n_nalpha[2] + row_n_2nalpha[2] + \
                          row_n_3nalpha[2] + row_n_p[2] +\
                          row_n_np[2] + row_n_d[2] +row_n_t[2]
        # try:
        #     if row_n_alpha[0] >= 0:
        #         print(row_n_alpha)
        #         donotuse=100
        #     continue
        # except TypeError:
        #     print(row_n_alpha)
        #     print(row_n_alpha[0][0])
        #     quit()

        if row_betanegdecay[0] >= 0:
            # [days^-1]
            row_lambda = np.log(2)*60*60*24/row_betanegdecay[1]
        elif row_betaposdecay[0] >= 0:
            # [days^-1]
            row_lambda = np.log(2)*60*60*24/row_betaposdecay[1]
        elif row_2alphadecay[0] >= 0:
            # [days^-1]
            row_lambda = np.log(2)*60*60*24/row_2alphadecay[1]
        else:
            row_lambda = 0.0

        # Diagonal Assignment
        A[row,row] = -row_lambda - phi_lo*row_lo_act_sum -\
                     phi_hi*row_hi_act_sum
        # Off Diagonal Assignment
        if row_betanegdecay[0] >= 0:
            A[row_betanegdecay[0],row] = np.log(2)*60*60*24/\
                                          row_betanegdecay[1]
        if row_betaposdecay[0] >= 0:
            A[row_betaposdecay[0],row] = np.log(2)*60*60*24/\
                                          row_betaposdecay[1]
        if row_2alphadecay[0] >= 0:
            A[row_2alphadecay[0],row] = np.log(2)*60*60*24/\
                                         row_2alphadecay[1]
        if row_n_gamma[0] >= 0:
            A[row_n_gamma[0],row] = phi_lo*row_n_gamma[1] +\
                                    phi_hi*row_n_gamma[2]
        if row_n_2n[0] >= 0:
            A[row_n_2n[0],row] = phi_lo*row_n_2n[1] +\
                                 phi_hi*row_n_2n[2]
        if row_n_alpha[0][0] >= 0:
            for i in row_n_alpha[0]:
                A[i,row] = phi_lo*row_n_alpha[1] +\
                           phi_hi*row_n_alpha[2]
        if row_n_2alpha[0][0] >= 0:
            for i in row_n_2alpha[0]:
                A[i,row] = phi_lo*row_n_2alpha[1] +\
                           phi_hi*row_n_2alpha[2]
        if row_n_nalpha[0][0] >= 0:
            for i in row_n_nalpha[0]:
                A[i,row] = phi_lo*row_n_nalpha[1] +\
                           phi_hi*row_n_nalpha[2]
        if row_n_2nalpha[0][0] >= 0:
            for i in row_n_2nalpha[0]:
                A[i,row] = phi_lo*row_n_2nalpha[1] +\
                           phi_hi*row_n_2nalpha[2]
        if row_n_3nalpha[0][0] >= 0:
            for i in row_n_3nalpha[0]:
                A[i,row] = phi_lo*row_n_3nalpha[1] +\
                           phi_hi*row_n_3nalpha[2]
        if row_n_p[0][0] >= 0:
            for i in row_n_p[0]:
                A[i,row] = phi_lo*row_n_p[1] + phi_hi*row_n_p[2]
        if row_n_np[0][0] >= 0:
            for i in row_n_np[0]:
                A[i,row] = phi_lo*row_n_np[1] + phi_hi*row_n_np[2]
        if row_n_d[0][0] >= 0:
            for i in row_n_d[0]:
                A[i,row] = phi_lo*row_n_d[1] + phi_hi*row_n_d[2]
        if row_n_t[0][0] >= 0:
            for i in row_n_t[0]:
                A[i,row] = phi_lo*row_n_t[1] + phi_hi*row_n_t[2]


    b = np.zeros(len(nuclides))

    # N_0 expressed as kg nuclide per kg FLiBe
    #b[nuclides['F19']] = 0.7685
    #b[nuclides['Be9']] = 0.0911
    #b[nuclides['Li6']] = 0.01065636
    #b[nuclides['Li7']] = 0.12974364
    AtomsofFLiBe=6.0899894727155e24
    b[nuclides['F19']] = AtomsofFLiBe*4
    b[nuclides['Be9']] = AtomsofFLiBe*1
    b[nuclides['Li6']] = AtomsofFLiBe*2*0.0759
    b[nuclides['Li7']] = AtomsofFLiBe*2*0.9241

    return(A,b)

###############################################################
################## Plotting Function ##########################
###############################################################

def reduceList(List,N):
    List2=[List[0]]
    Div=int(len(List)/N)
    for i in range(1,len(List)-1):
        if i % Div == 0:
            List2.append(List[i])
    List2.append(List[-1])
    return(List2)

def PlotPoints(Ntot,Nplot):
    t=1
def loop_values(list1,index):
    """
    This function will loop through values in list even if
    outside range (in the positive sense not negative)
    """
    while True:
        try:
            list1[index]
            break
        except IndexError:
            index=index-len(list1)
    return(list1[index])

def Legend(ax):
    handles,labels=ax.get_legend_handles_labels()
    ax.legend(handles,labels,loc='best',
              fontsize=LegendFontSize,prop=font)
    return(ax)

# def Legend(ax):
#         handles,labels=ax.get_legend_handles_labels()
#         box=ax.get_position()
#         ax.set_position([box.x0, box.y0, box.width*SquishGraph,
#                          box.height])
#         ax.legend(handles,labels,loc='center',
#                   bbox_to_anchor=(BBOXX,BBOXY),
#                   fontsize=LegendFontSize,prop=font,
#                   ncol=NumberOfLegendColumns)
#         return(ax)

def InList(item2,List):
  TF=False
  for item1 in List:
    if item1 == item2:
      TF=True
  if not TF:
    print("Invalid selection for plotting")
    print("Shuting down")
    quit()

def plot(df,Plotting,Name,NumOfPoints):
    #Plot In grams
    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)
```

Homework 5 Code continued on next page...

```
    List=list(df.columns.values)
    x=df[List[0]].values[2:-1]

    Check=0
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:-1])/Na)*df[Item].values[0]
      if len(x)>NumOfPoints:
        x=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
      ax.plot(x,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item)
      Check=Check+1


    #Log or linear scale?
    ax.set_xscale(XScale)
    ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    ax.set_ylabel(Ylabel,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_grams.pdf')

    #Plot in Bq ################################

    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:-1]


    Check=0;Sum=np.zeros(len(x))
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:-1]))*df[Item].values[1]
      Sum=Sum+y
      if len(x)>NumOfPoints:
        xP=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
      ax.plot(xP,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item)
      Check=Check+1
    if len(x)>NumOfPoints:
      Sum=reduceList(Sum,NumOfPoints)
    ax.plot(xP,Sum,
            linestyle=loop_values(LineStyles,Check),
            marker=loop_values(MarkerType,Check),
            color=loop_values(Colors,Check),
            markersize=loop_values(MarkSize,Check),
            alpha=loop_values(Alpha_Value,Check),
            label="Sum")

    #Log or linear scale?
    ax.set_xscale(XScale)
    if sum(Sum)==0:
      ax.set_yscale('linear')
    else:
      ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    YlabelBq="Activity $\\left[\\frac{Bq}{kg \\text{ FLiBe}}\\right]$"
    ax.set_ylabel(YlabelBq,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_Bq.pdf')

def plots2(df,df2,Plotting,Name,NumOfPoints,Method1,Method2):
    #Plot in grams
    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:-1]

    Check=0
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:-1])/Na)*df[Item].values[0]
      y2=((df2[Item].values[2:-1])/Na)*df2[Item].values[0]
      if len(x)>NumOfPoints:
        x=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
        y2=reduceList(y2,NumOfPoints)
      ax.plot(x,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check)*1.5,
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method1)
      Check=Check+1
      ax.plot(x,y2,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method2)
      Check=Check+1


    #Log or linear scale?
    ax.set_xscale(XScale)
    ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    ax.set_ylabel(Ylabel,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_Grams.pdf')


    #Plot in Bq
    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:-1]

    Check=0;Sum=np.zeros(len(x));Sum2=np.zeros(len(x))
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:-1]))*df[Item].values[1]
      y2=((df2[Item].values[2:-1]))*df2[Item].values[1]
      Sum=Sum+y
      Sum2=Sum2+y2
      if len(x)>NumOfPoints:
        xP=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
        y2=reduceList(y2,NumOfPoints)
      ax.plot(xP,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check)*1.5,
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method1)
      Check=Check+1
      ax.plot(xP,y2,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method2)
      Check=Check+1
    if len(x)>NumOfPoints:
      Sum=reduceList(Sum,NumOfPoints)
      Sum2=reduceList(Sum2,NumOfPoints)
    ax.plot(xP,Sum,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check)*1.5,
              alpha=loop_values(Alpha_Value,Check),
              label="Sum "+Method1)
    Check=Check+1
    ax.plot(xP,Sum2,
            linestyle=loop_values(LineStyles,Check),
            marker=loop_values(MarkerType,Check),
            color=loop_values(Colors,Check),
            markersize=loop_values(MarkSize,Check),
            alpha=loop_values(Alpha_Value,Check),
            label="Sum "+Method2)

    #Log or linear scale?
    ax.set_xscale(XScale)
    if sum(Sum)==0:
      ax.set_yscale('linear')
    else:
      ax.set_yscale(YScale)

    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    YlabelBq="Activity $\\left[\\frac{Bq}{kg \\text{ FLiBe}}\\right]$"
    ax.set_ylabel(YlabelBq,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_Bq.pdf')

def ListToStr(List):
  Str=''
  for i in range(0,len(List)):
    if not i==len(List)-1:
      Str=Str+str(List[i])+","
    else:
      Str=Str+str(List[i])+"\n"
  return(Str)

def PrepFile(Name,n0):
  File=open(Name,'w')
  File.write("Mass then Time (d),"+','.join(nuclide_names)+'\n')
  File.write("Masses,"+ListToStr(atom_mass)) #New line already included
  File.write("DecayConts,"+ListToStr(decay_consts))
  File.write("0,"+ListToStr(n0))
  return(File)

def Print(Method,nuclide,Results,Time):
  Index=nuclides[nuclide]
  MassConversion=atom_mass[Index]/Na
  string="Isotope "+nuclide_names[Index]+", Mass (g) = "
  Mass=Results[Index]*MassConversion
  Mass="%.4e" % Mass
  print(Method+" :",string,Mass,"Time=%.2f" % Time)

def Years(Method,nuclide,Results):
  Index=nuclides[nuclide]
  LambdaY=decay_consts[Index]*60*60*24*365.25
  Lambdas=decay_consts[Index]
  string="Isotope "+nuclide_names[Index]+", Years to 444 Bq = "
  Years=(-1/LambdaY)*np.log(444/(Results[Index]*Lambdas))
  print(Method+" :",string,"%.3e" % Years)
```

Project

For the project I wanted to solve a depletion problem using the algorithms developed in the previous homework. The bateman equations will be written in a different form to align more with how the system was built in homework 5, and for the project.

The production of an isotope is dictated by production and loss

$$\frac{dn_i}{dt} = -\lambda_i^{eff} n_i + \sum_{j=1}^{N} b_{j\to i}^{eff} n_j$$

Where N is the number of nuclides and,

$$\lambda_i^{eff} = \lambda_i + \phi \sum_{j=1}^{N} \sigma_{i\to j}$$

and

$$b_{j\to i}^{eff} = b_{j\to i}\lambda_j + \sigma_{j\to i}\phi + \gamma_{j\to i}\sigma_{j,f}\phi$$

where $\gamma_{j\to i}$ is the fission yield for isotope i from fission of isotope j. $b_{j\to i}$ is the fraction of radioactive disintegration by nuclide j, which leads to nuclide i. In the case of spontaneous fission $b_{j\to i}$ is the product of spontenous fission fraction for nuclide j and the yield for fission from nuclide j producing nuclide i.

For a system of isotopes, the system can be reduced to:

$$\frac{d\vec{n}}{dt} = \boldsymbol{A}\vec{n}(t)$$

Where $\boldsymbol{A}$ is a matrix whose diagonal elements are $[-\lambda_1^{eff}, -\lambda_2^{eff}, ..., -\lambda_N^{eff}]$, all off diagonal elements are $b_{j\to i}^{eff}$ (i for the row, and j is for the column) and $\vec{n}(t) = [n_1, n_2, ..., n_N]$.

The solution to this system is so obvious, I won't even write it down.

For the current problem 1 metric ton of PWR fuel will be irradiated for 400 days at a constant flux of 2.9896 $\cdot 10^{14}$ n/cm$^2\cdot$s (37.5 W/g initially) and some of the fission product masses will be determined as a function of time. If we ignore the oxygen, the initial amount of atoms for each of the heavy isotopes is

$$N_{^{234}U} = 270 \text{ g } \frac{6.022E23 \text{ atoms}}{234.0409523 \text{ g}} = 6.94741E23 \, \frac{\text{atoms of } ^{234}\text{U}}{\text{tHM}}$$

$$N_{^{235}U} = 30000 \text{ g } \frac{6.022E23 \text{ atoms}}{235.0439301 \text{ g}} = 7.6864E25 \, \frac{\text{atoms of } ^{235}\text{U}}{\text{tHM}}$$

$$N_{^{238}U} = 969730 \text{ g } \frac{6.022E23 \text{ atoms}}{238.0507884 \text{ g}} = 2.4532E27 \, \frac{\text{atoms of } ^{238}\text{U}}{\text{tHM}}$$

The results will be compared with ORIGEN2 results for a similar problem that will irradiate the same fuel with a constant power of 37.5 W/g (compare with my results from UQ).

Project Results

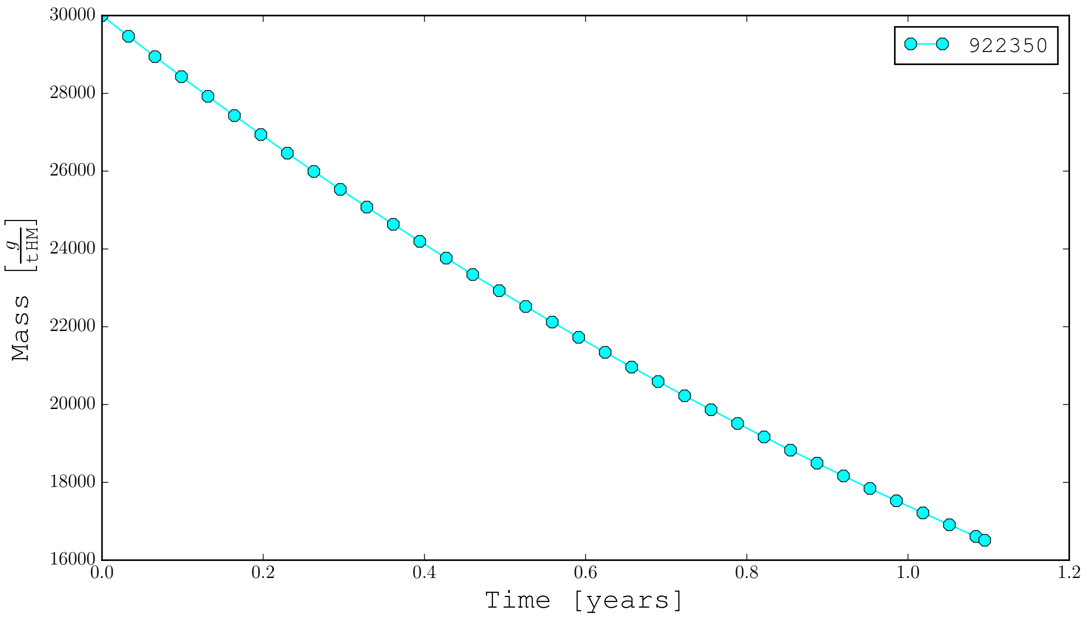
The results for the code are shown below. The hardest part of this problem was building the A matrix. In order to determine if the system was solved correctly the information fed into ORIGEN2 from its TAPE9.inp file was used so that results from the two codes could be compared. I wanted to be able to recreate what ORIGEN2 does with my own code so I could complete the project for UQ in the future after I graduate. The only part I am not sure about on how to do is how to sample the cross section spectrum.

Comparison of results from the two codes are shown in the table below, and it can be seen that they are fairly close. The main difference is from the constant power (ORIGEN2) constant flux (MINE) different assumption.

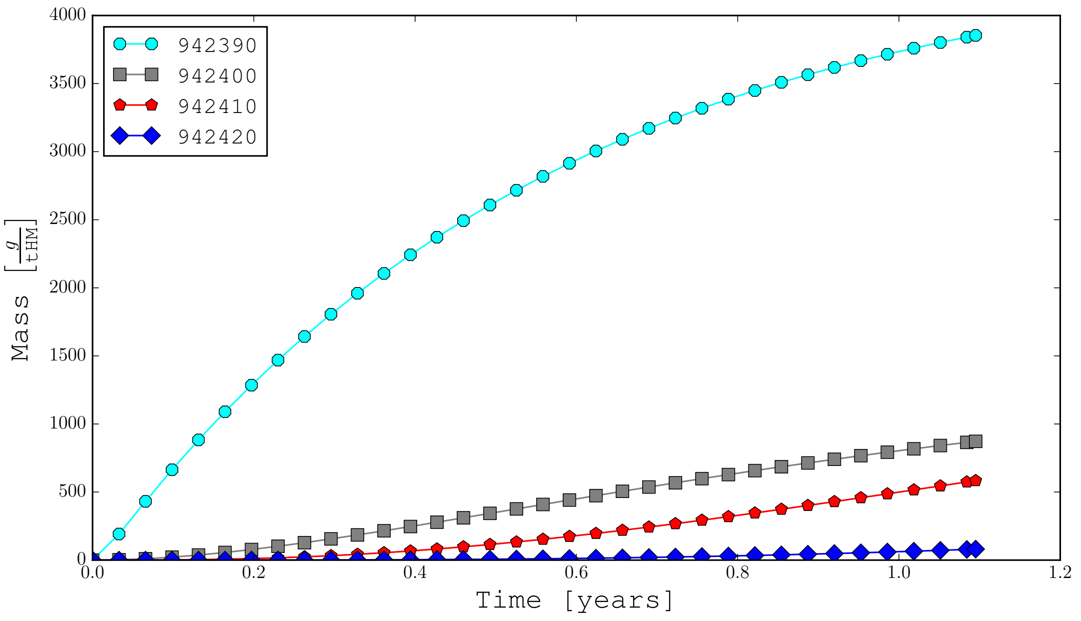
Table 1: Comparison between ORIGEN2 and my code results in grams

Nuclide	ORIGEN2	My Code
^{137}Cs	553	544
^{137}Ba	7.10	7.13
^{136}Xe	1.04E3	1.02E3
^{134}Ba	3.70	3.60
^{133}Cs	549	538
^{106}Ru	74.3	73.2
^{148}Nd	169.1	166.5
^{105}Pd	136	134
^{99}Tc	374	371
^{95}Zr	83.1	79.9
^{235}U	1.629E4	1.65E4
^{238}U	9.597E5	9.6E5
^{239}Pu	3.994E3	3.85E3
^{241}Pu	5.279E2	5.838E2

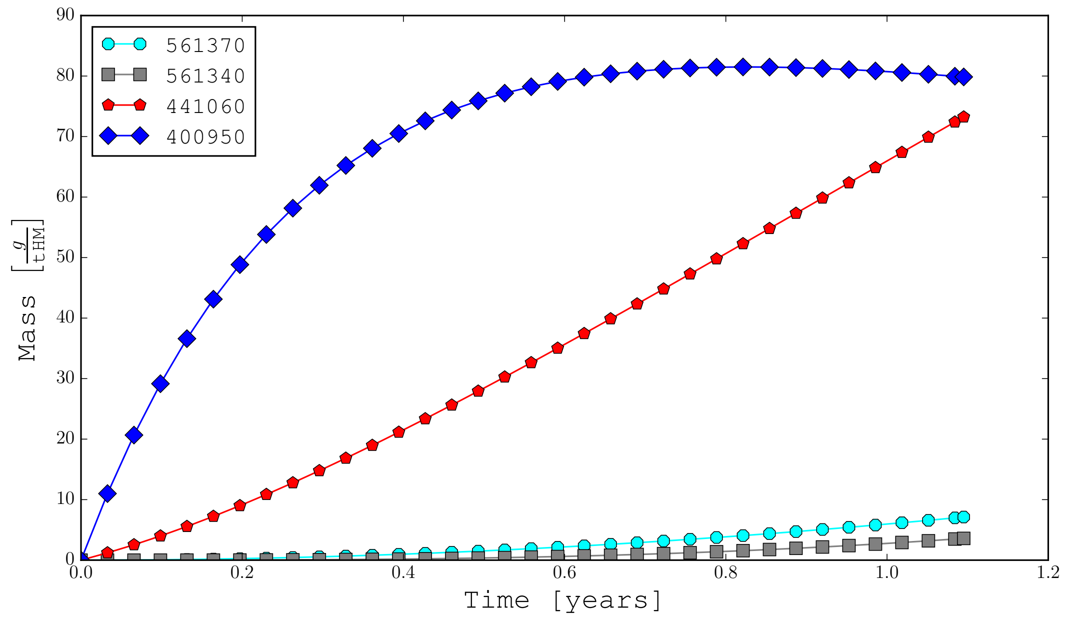
Below are some graphs of the production and loss of nuclides to show nuclide production trends. Plots are only shown with the back-Euler solution (the rational approximation was faster to get results for a single time, but to show trends, back Euler calculates the answer at each time step) to save space on the plots. The two methods for calculation agreed on the results with 10 quadrature points for the Best method (more for others).



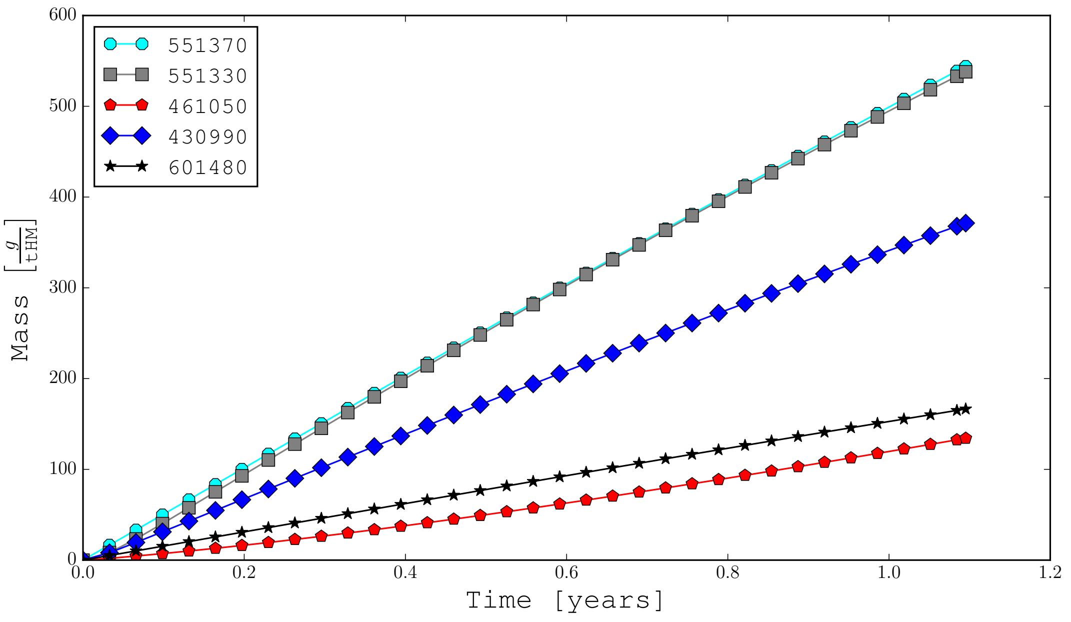
The ^{238}U didn't deplete an appreciable amount in the irradiation. The ^{235}U depleted an appreciable amount.



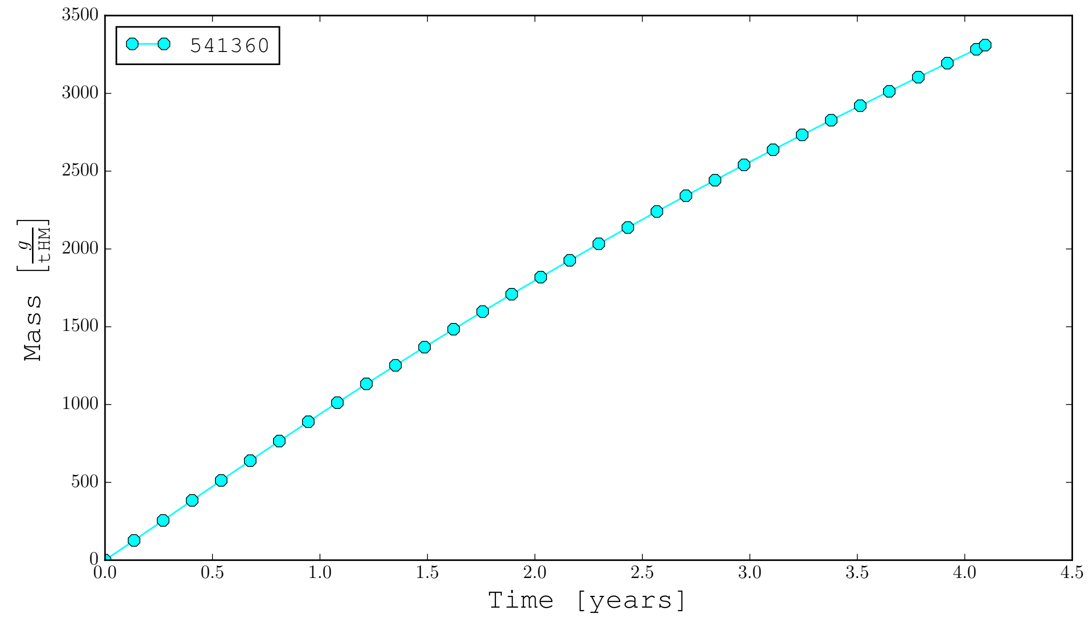
The plutonium isotopes were produced at a rate about what I would expect. It seems that ^{239}Pu is being produced at a rate that is a little higher than what I would expect.



Above are some slowly growing isotopes.



Above are moderately growing isotopes. These plots are ploted against time, not burnup, but to first order the scaling is about the same. ^{137}Cs and ^{148}Nd should be linear as a function of burnup, the others, not so much, but maybe they will turn over.



Above is another, larger yield isotope, ^{136}Xe, which is absorbed from ^{135}Xe.

Project Code

Listing 7: **Main Code**

```
#!/usr/bin/env python3
#Means for improving speed. Make dir and save all variables
#made for making a, so that no parsing is needed,
#Just multiplying by phi

#################################################################
##################### Import packages #########################
#################################################################

import time
start_time = time.time()
import Functions as f

#################################################################
#################### Initialize System #######################
#################################################################

NumOfPoints=30          # Max Number of points for plots
PlottingG1=["922350","922380","942390","942410"]  #List elements you want to Plot
PlottingG1=["942390","942400","942410","942420"]  #List elements you want to Plot
PlottingG1=["922350"]  #List elements you want to Plot
#CompareG1=["922350","922380","942390"]
CompareG1=["922350","922380","942390","942410"]
PlottingG2=["551370","561370","541360","561340","551330","441060","461050","430990","400950"]
PlottingG2=["561370","561340","441060","400950"]
PlottingG2=["551370","551330","461050","430990","601480"]
PlottingG2=["541360"]
#List elements you want to Plot
#CompareG2=["551370","631540","601480"]
CompareG2=["551370","561370","541360","561340","551330","441060","461050","430990","400950"]
#List elements you want to compare between methods

##########################################################
############# Initialize Matrix ########################
##########################################################

phi=1.0e14;Power = 37.5 # MW (bc 1 ton HM power den is W/g)

Fissile_Isotopes=["922350","922380","942390","942410"]
FissionXSections=f.FindFissionXSection(Fissile_Isotopes)
MeVperFission=f.CalMevPerFiss(Fissile_Isotopes)
ToAdd=[]
Nuclides,Nuclide_Names=f.Isotopes(ToAdd)
n0=f.Makeb(Nuclides)
phi=f.Calculatephi(FissionXSections,MeVperFission,n0,Power,Fissile_Isotopes,Nuclides)

#This variable takes a while to make, so if its already made
#Just load it
#f.os.system("rm Saved_Variables/Masses.npy")  #To start over and remake
if not f.os.path.isfile('Saved_Variables/Masses.npy'):
    #Grab atomic weight information
    df=f.pd.read_csv('Data/AtomicWeights.csv')
    Atom_Mass=f.GatherMasses(df,Nuclides)
    f.np.save('Saved_Variables/Masses',Atom_Mass)
else:
    Atom_Mass=f.np.load('Saved_Variables/Masses.npy')

#Load up all the decay constants
#f.os.system("rm Saved_Variables/Decay_Consts.npy") #To start over and remake
if not f.os.path.isfile('Saved_Variables/Decay_Consts.npy'):
    Decay_Conts=f.GatherDecay(Nuclide_Names)
    f.np.save('Saved_Variables/Decay_Consts',Decay_Conts)
else:
    Decay_Conts=f.np.load('Saved_Variables/Decay_Consts.npy')

#load up A and b with phi
#f.os.system("rm Saved_Variables/Amatrix.npy") #To start over
#f.os.system("rm Saved_Variables/n0vector.npy") #to start over
if not f.os.path.isfile('Saved_Variables/Amatrix.npy'):
    #A is in terms of years
    A,n0=f.MakeAb(phi,Nuclides,Nuclide_Names,Decay_Conts)
    f.np.save('Saved_Variables/Amatrix',A)
    f.np.save('Saved_Variables/n0vector',n0)
else:
    A=f.np.load('Saved_Variables/Amatrix.npy')
    n0=f.np.load('Saved_Variables/n0vector.npy')

phi=0
#load up A and b without phi (decay)
#f.os.system("rm Saved_Variables/Amatrixdecay.npy") #To start over
#f.os.system("rm Saved_Variables/n0vectordecay.npy") #to start over
if not f.os.path.isfile('Saved_Variables/Amatrixdecay.npy'):
    #A is in terms of years
    Adecay,n0=f.MakeAb(phi,Nuclides,Nuclide_Names,Decay_Conts)
    f.np.save('Saved_Variables/Amatrixdecay',A)
    f.np.save('Saved_Variables/n0vectordecay',n0)
else:
    Adecay=f.np.load('Saved_Variables/Amatrixdecay.npy')
    n0decay=f.np.load('Saved_Variables/n0vectordecay.npy')

if not A.shape[0] == A.shape[1] or not A.shape[0] == len(n0):
    print("A is not a square matrix")
    quit()

print("A is complete")
##########################################################
############# Initialize Time ###########################
##########################################################

t=1.09514;  #years
Nt=100;   #Number of Time Steps
dt=t/Nt;
Time=f.np.linspace(dt,t,Nt) #Time steps
```

Project Code continued on next page...

```
tDecay=0; #No time of decay
NtDecay=1; #Number of Decay Time Steps
dtDecay=tDecay/NtDecay
TimeDecay=f.np.linspace(t+dtDecay,t+tDecay,NtDecay)

#################################################################
##################### Solve System ############################
################ And Time How long it takes ###################
################### For Each Method ###########################
################# And Plot at Each Time Step ##################
#################################################################

#Matrix Exp - Unstable
#Current_Time=time.time()
#maxits=20
# nt_Mat=f.MatExp(A,n0,t,maxits)
# Mat_Time=time.time()-Current_Time
#nt_Mat=n0.copy();TIMEOLD=0
#for TIME in Time[1:len(Time)]: #Unstable
    #nt_Mat=f.MatExp(A,n0,TIME,maxits) No Converge
    #Step through
    #nt_Mat=f.MatExp(A,nt_Mat,TIME-TIMEOLD,maxits)
    #TIMEOLD=TIME.copy()
#nt_Mat=f.RationalApprox(A,n0,t,maxits) #one Step
#Mat_Time=time.time()-Current_Time

##########################################################
############### Backward Euler #######################
##########################################################

Current_Time=time.time()
File=f.PrepFile('Data/BackEuler.csv',n0,Nuclide_Names,Atom_Mass,Decay_Conts) #Prep File
nt_Back=n0.copy();
#Irradiation Time
for TIME in Time:
    nt_Back=f.BackEuler(A,nt_Back,dt)
    File.write(str(TIME)+","+f.ListToStr(nt_Back))
#Decay Time
#for TIME in TimeDecay:
#    nt_Back=f.BackEuler(Adecay,nt_Back,dtDecay)
#    File.write(str(TIME)+","+f.ListToStr(nt_Back))

File.close()
Back_Time=time.time()-Current_Time

##########################################################
############### Rational Approx #######################
##########################################################


#Irradiation Time
Current_Time=time.time()
```

```
N=10;
Method="Best"  #Parabola, Cotangent, Hyperbola, Best
File=f.PrepFile("Data/"+Method+".csv",n0,Nuclide_Names,Atom_Mass,Decay_Conts)
ck,zk=f.RationalPrep(N,Method);nt_Rational=n0.copy();TIMEOLD=0
#for TIME in Time:
#    nt_Rational=f.RationalApprox(A,n0,TIME,N,ck,zk)
#    File.write(str(TIME)+","+f.ListToStr(nt_Rational))
    #Step through
    #nt_Rational=f.RationalApprox(A,nt_Rational,TIME-TIMEOLD,N,ck,zk)
    #TIMEOLD=TIME.copy()
nt_Rational=f.RationalApprox(A,n0,t,N,ck,zk) #one Step
File.write(str(TIME)+","+f.ListToStr(nt_Rational))

#Non Irradiation Time
n0decay=nt_Rational

#for TIME in TimeDecay:
#    nt_Rational=f.RationalApprox(Adecay,n0decay,TIME-t,N,ck,zk)
#    File.write(str(TIME)+","+f.ListToStr(nt_Rational))

File.close()
Rational_Time=time.time()-Current_Time

########################################################
################# Plot Solution ########################
################# In Grams #############################
################# and activity #########################
########################################################


dfBack = f.pd.read_csv('Data/BackEuler.csv',index_col=False)
dfRational = f.pd.read_csv("Data/"+Method+".csv",index_col=False)

# #Plot group 1 dudes Back Euler method and rational
f.plot(dfBack,PlottingG1,'BackEulerG1U',NumOfPoints)
#f.plot(dfRational,PlottingG1,Method+'G1',NumOfPoints)

# #Plot group 2 dudes Rational method and back
#f.plot(dfBack,PlottingG2,'BackEulerG2A1000',NumOfPoints)
# f.plot(dfRational,PlottingG2,Method+'G2',NumOfPoints)

#Plot group 1 dudes, compare both methods
#Name='BackEuler_'+Method+"_G1"
#f.plots2(dfBack,dfRational,CompareG1,Name,
#         NumOfPoints,'BackEuler',Method)

#Plot group 2 dudes, compare both methods
#Name='BackEuler_'+Method+"_G2"
#f.plots2(dfBack,dfRational,CompareG2,Name,
#         NumOfPoints,'BackEuler',Method)

###############################################################
################## Print Solution #############################
```

```
#################################################################

#f.Print("Matrix Exp","H3",nt_Mat,Mat_Time)
#f.Print("Backward Euler","601480",nt_Back,Back_Time,Nuclides,Atom_Mass,Nuclide_Names)
#f.Print("Rational Approx ","541360",nt_Rational,Rational_Time,Nuclides,Atom_Mass,Nuclide_Names)


###################### Time To execute ################

print("--- %s seconds ---" % (time.time() - start_time))
```

Listing 8: **Functions holder**

```
#!/usr/bin/env python3

#################################################################
##################### Import packages #########################
#################################################################

import sys
import numpy as np
import scipy.sparse as sparse
import scipy.sparse.linalg as spla
import scipy.linalg as scil
import scipy.special as sps
import matplotlib.pyplot as plt
plt.rcParams["font.family"] = "monospace"
import matplotlib
matplotlib.rc('text',usetex=True)
matplotlib.rcParams['text.latex.preamble']=[r"\usepackage{amsmath}"]
import random as rn
import matplotlib.mlab as mlab
import copy
import os
import pandas as pd
import os.path

##############################################################
######################### Variables ########################
##############################################################

# Basic information
FigureSize = (11, 6)              # Dimensions of the figure
TypeOfFamily='monospace'          # This sets the type of font for text
font = {'family' : TypeOfFamily}  # This sets the type of font for text
LegendFontSize = 12
Lfont = {'family' : TypeOfFamily}  # This sets up legend font
Lfont['size']=LegendFontSize

Title = ''
TitleFontSize = 22
TitleFontWeight = "bold"  # "bold" or "normal"

#Xlabel='E (eV)'   # X label
```

```
XFontSize=18          # X label font size
XFontWeight="normal"  # "bold" or "normal"
XScale="linear"       # 'linear' or 'log'

YFontSize=18                    # Y label font size
YFontWeight="normal"            # "bold" or "normal"
YScale="linear"                 # 'linear' or 'log'

Check=0


Colors=["aqua","gray","red","blue","black",
                "green","magenta","indigo","lime","peru","steelblue",
                "darkorange","salmon","yellow","lime","black"]

# If you want to highlight a specific item
# set its alpha value =1 and all others to 0.4
# You can also change the MarkSize (or just use the highlight option below)
Alpha_Value=[1  ,1  ,1  ,1  ,1  ,1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]
MarkSize=   [8  ,8  ,8  ,8  ,8  ,8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8]

Linewidth=[1  ,1  ,1  ,1  ,1  ,1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1]

# Can change all these to "." or "" for nothing "x" isn't that good
MarkerType=["8","s","p","D","*","H","h","d","^",">"]

# LineStyles=["solid","dashed","dash_dot","dotted","."]
LineStyles=["solid"]

SquishGraph = 0.75
BBOXX = 1.24
BBOXY = 0.5       # Set legend on right side of graph

NumberOfLegendColumns=1

#Xlabel='Time [days]'
Xlabel='Time [years]'
Ylabel="Mass $\\left[\\frac{g}{\\text{tHM}}\\right]$"


Na=6.0221409E23


################################################################
################## Functions for building ####################
################## Lists                  ####################
################################################################

def Returnfloat(string):
    """
    string has format   238.023249814(23)
            or format   [15.99903-15.99977]
            or format   235.04+/-0.0000019

    Returns just the number, no uncertainties
    """
    if "(" in string:
        Number=str(string.split('(')[0])
        LastErrorNumber=str(string.split("(")[1].replace(")",""))
        NumberOfZeros=len(Number.split(".")[1])-len(LastErrorNumber)
        Error="0."
        for i in range(0,NumberOfZeros):
            Error=Error+"0"
        Error=Error+LastErrorNumber
    elif "[" in string:
        FirstNum=float(string.split('-')[0].replace("[",''))
        SecondNum=float(string.split('-')[1].replace(']',''))
        Number=str((FirstNum+SecondNum)/2)
        Error=str(float(Number)-FirstNum)
    elif "+/-" in string:
        Number=string.split("+/-")[0]
        Error=string.split("+/-")[1]

    return(float(Number))

def Isotopes(ToAdd):
  """
  This function will create a dictionary 'Nuclides'
  with nuclides found in tape9.inp, excluding activation isotopes
  """
  Nuclides={}
  Nuclide_Names=()

  with open('Data/tape9.inp') as f:
    TAPE9Content=f.readlines()

  count=0
  for line in TAPE9Content:
    hold=line.split()

    #No activation products or the -1 between libraries
    if not '1' in hold[0] and not '601' in hold[0] and "-" not in hold[0]:

      #No repeats                  #No decimals              #No text
      if hold[1] not in Nuclides and "." not in hold[1] and hold[1].isdigit():
        #Filter out lower mass isotopes
        if len(hold[1])==6:
          Nuclides[hold[1]]=count
          Nuclide_Names=Nuclide_Names+(hold[1],)
          count=count+1

  for isotope in ToAdd:
      Nuclides[isotope]=count
      Nuclide_Names=Nuclide_Names+(isotope,)
      count=count+1
```

Project Code continued on next page. . .

```
  return(Nuclides,Nuclide_Names)

def GatherDecay(Nuclide_Names):
  """
  This function will create an array 'Decay_Consts'
  that has all the half-life information for our system.
  """

  Decay_Consts = np.zeros(len(Nuclide_Names))


  with open('Data/tape9.inp') as f:
    TAPE9Content=f.readlines()

  for i in range(0,len(Nuclide_Names)):
      Nuclide=Nuclide_Names[i]
      for line in TAPE9Content:
          hold=line.split()

          #Look for half life information, and decay type information
          #No activation products or the -1 between libraries
          if hold[0]=='2' or hold[0]=='3':
              if hold[1] == Nuclide:
                  Thalf=float(hold[3])
                  if hold[2]=='1':  #seconds
                      const=np.log(2)/Thalf
                  elif hold[2]=='2':  #minutes
                      const=np.log(2)/(Thalf*60)
                  elif hold[2]=='3': #hours
                      const=np.log(2)/(Thalf*60*60)
                  elif hold[2]=='4': #days
                      const=np.log(2)/(Thalf*60*60*24)
                  elif hold[2]=='5': #years
                      const=np.log(2)/(Thalf*60*60*24*365.25)
                  elif hold[2]=='6': #Stable
                      const=-1
                  elif hold[2]=='7':
                      const=np.log(2)/(Thalf*60*60*24*365.25*10**3)
                  elif hold[2]=='8':
                      const=np.log(2)/(Thalf*60*60*24*365.25*10**6)
                  elif hold[2]=='9':
                      const=np.log(2)/(Thalf*60*60*24*365.25*10**9)
                  else:
                      print("could not find a proper halflife")
                      print(line)
                      quit()
                  Decay_Consts[i]=const

  return(Decay_Consts)



def FindAtomicMass(df,proton,Isotope):
```

Project Code continued on next page...

```
    """
    This function will take in a dataset 'df' look through the
    'df.Protons' column and find the column that matches with
    'proton'. If the row that contains 'proton' also contains
    'Isotope' in the 'df.Isotope' column, then the value stored
    in 'df.Relative_Atomic_Mass' is reported for that row.
    Because the proton numbering scheme can have a format
    '10' for hydrogen and '10' for neon (following MCNP ZAID
    naming conventions) if we don't find a value with the whole
    string of 'proton' then the program looks through the first
    element of string and tries to match that 'proton[0]'
    If no matches are found, and error is thrown out.

    df = dataset with columns 'Protons' 'Isotopes' and
    'Relative_Atomic_Mass'. Dataset created with pandas

    proton = string with proton number (follow MCNP zaid format)

    Isotope = string with isotope number (just put the atomic mass
    do not follow MCNP format - different for few cases)
    """
    #print(df)
    for i in range(0,len(df.Protons)):
        dfPro=str(df.Protons[i])
        if proton==dfPro:
            dfIso=str(df.Isotope[i])
            if Isotope==dfIso:
                Mass=df.Relative_Atomic_Mass[i]
                break
    try:
        Mass
    except NameError:
        for i in range(0,len(df.Protons)):
            dfPro=str(df.Protons[i])
            if proton[0]==dfPro:
                dfIso=str(df.Isotope[i])
                if Isotope==dfIso:
                    Mass=df.Relative_Atomic_Mass[i]
                    break
    try:
        Mass
    except NameError:
        print("Could not find atomic mass for proton = "\
              +proton+" and for Isotope = "+Isotope)
        Mass='10000.09(23)'
    Mass=Returnfloat(Mass)
    return(Mass)

def GatherMasses(df,Nuclides):
  """
  Make numpy array of masses which correspond to isotopes in Nuclides
  Nuclides is a dictionary. Where each key is a zaid number
  """
```

Project Code continued on next page...

```
  Atom_Mass=np.zeros(len(Nuclides))
  for key,value in Nuclides.items():
    if not len(key)==6:
      print("Did not filter out the lower mass isotopes, quitting")
      quit()
    proton=key[0:2]
    if key[2]=="0":
        Isotope=key[3:5]
    elif key[4]=="0" and float(proton)*3<100:
        Isotope=key[2:4]
    elif key[2]!="0":
        Isotope=key[2:5]

    Mass=FindAtomicMass(df,proton,Isotope)
    if Mass>9000:
        print("Mass is over 9,000!!!")
        print(key)
        quit()
    Atom_Mass[value]=Mass
  return(Atom_Mass)

#################################################################
####### Functions for solving the system Ax=b (kind of) ########
#################################################################

def MatExp(A,n0,t,maxits,tolerance=1e-12,LOUD=False):

  converged = False
  m=0
  sum_old=n0.copy()*0

  while not(converged):

    if m==0:
        APowerm=np.identity(A.shape[0])
        Factorial=1
    else:
        APowerm=np.dot(APowerm,A)
        Factorial=Factorial*m
    Sum=sum_old+(1/Factorial)*np.dot((APowerm)*(t**m),n0)

    #Avoid dividing by zero
    if sum(Sum)==0: m+=1;sum_old=Sum.copy();continue

    change = np.linalg.norm(Sum-sum_old)/np.linalg.norm(Sum)
    converged = (change < tolerance) or (m > maxits)

    if (LOUD>0) or (converged and LOUD<0):
      print("Iteration",m,": Relative Change =",change)
    if (m > maxits):
      print("Warning: Source Iteration did not converge : "+\
            " m : "+str(m)+", Diff : %.2e" % change)
        #Prepare for next iteration
        m += 1
        sum_old = Sum.copy()

    return(Sum)

def BackEuler(A,no,dt):
    I=np.identity(A.shape[0])
    return(np.dot(np.linalg.inv(I-A*dt),no))

def DeterminePolesNResidues(n):
    """
    This program takes the algorithm from the reference
    and converts to a python script...I know its janky
    but it works
    """
    def Append(List1,List2):
        for item in List2:
            for item2 in item:
                List1=np.append(List1,item2)
        return(List1)
    def absG(List):
        List2=[]
        for item in List:
            if abs(item)>1:
                List2.append(item)
        return(List2)
    #function [zk,ck] = cf(n);
    K = 75;                                # no of Cheb coeffs
    nf = 1024;                             # no of pts for FFT
    #Roots correct?
    roots=np.arange(0,nf,1)/nf
    #w = np.exp(2i*pi*(0:nf-1)/nf);      # roots of unity

    w=np.exp(2j*np.pi*roots)
    t = np.real(w);                        # Cheb pts (twice over)
    scl = 9;                               # scale factor for stability
    #F = np.exp(scl*(t-1)./(t+1+1e-16)); # exp(x) transpl. to [-1,1]
    F = np.exp(scl*(t-1)/(t+1+1e-16)); # exp(x) transpl. to [-1,1]
    c = np.real(np.fft.fft(F))/nf;       # Cheb coeffs of F
    index=reversed(np.arange(1,K+2,1))
    partofc=[]
    for i in index:
        partofc.append(c[i-1])
    #f = np.polyval(c(K+1:-1:1),w);       # analytic part f of F
    f = np.polyval(partofc,w);       # analytic part f of F

    #[U,S,V] = svd(hankel(c(2:K+1)));     # SVD of Hankel matrix
    hankie=scil.hankel(c[1:K+1])
    U,S,V=np.linalg.svd(hankie,full_matrices=False)

    #s = S(n+1,n+1);                       # singular value
    s=S[n]
    #u = U(K:-1:1,n+1); v = V(:,n+1);  # singular vector
    u=[]
    index=reversed(np.arange(0,K,1))
    for i in index:
        u.append(U[i,n])
    #v=np.array(V[:,n].copy())
    v=np.array(V[n,:].copy())
    #zz = zeros(1,nf-K);                # zeros for padding
    zz=np.zeros([1,nf-K])
    #b = fft([u zz])./fft([v zz]);      # finite Blaschke product
    b=np.fft.fft(Append(u,zz))/np.fft.fft(Append(v,zz))
    #rt = f-s*w.^K.*b;                  # extended function r-tilde
    rt=f-s*(w**K)*b;
    #rtc = real(fft(rt))/nf;            # its Laurent coeffs
    rtc=np.real(np.fft.fft(rt))/nf;
    #zr = roots(v); qk = zr(abs(zr)>1);  # poles
    zr=np.roots(v);qk=np.array(absG(zr));
    #qc = poly(qk);                     # coeffs of denominator
    qc=np.poly(qk);
    #pt = rt.*polyval(qc,w);            # numerator
    pt=rt*np.polyval(qc,w);
    #ptc = real(fft(pt)/nf);            # coeffs of numerator
    ptc=np.real(np.fft.fft(pt)/nf);
    #ptc = ptc(n+1:-1:1); ck = 0*qk;
    index=reversed(np.arange(0,n+1,1))
    ptc2=[]
    for i in index:  #Can I just reversed ptc?
        ptc2.append(ptc[i])
    ptc=ptc2.copy()
    ck=0*qk
    #N+1?
    #for k =1:n              # calculate residues
    #    q = qk(k); q2 = poly(qk(qk~=q));
    #    ck(k) = polyval(ptc,q)/polyval(q2,q);
    for k in range(0,n):
        if len(qk)==k:
            print("we are short a qk")
            continue
        q=qk[k];
        q2=[];
        for item in qk:
            if not q==item:
                q2.append(item)
        q2=np.poly(q2);
        ck[k]=np.polyval(ptc,q)/np.polyval(q2,q)
    #zk = scl*(qk-1).^2./(qk+1).^2;     # poles in z-plane
    zk=scl*((qk-1)**2)/((qk+1)**2)
    #ck = 4*ck.*zk./(qk.^2-1);          # residues in z-plane
    ck=4*ck*zk/(qk**2-1)
    #Cut down ck and zk to half the original points
    ck2=[];zk2=[]
    for i in range(0,len(ck)):
        if i % 2 == 0:
            ck2=np.append(ck2,ck[i])
            zk2=np.append(zk2,zk[i])

    return(ck2,zk2)

def RationalPrep(N,Phi):
    """Calculate constants for a rational approximation
    Inputs:
    N:                Number of Quadrature points
    Phi:              'Parabola',
                      'Cotangent', or
                      'Hyperbola' (shape of Phi)

    Outputs:
    ck:               First set of constants for approximation
    zk:              Second set of constants for approximation
    """
    theta=np.pi*np.arange(1,N,2)/N
    if Phi=='Parabola':
        zk=N*(0.1309-0.1194*theta**2+0.2500j*theta)
        w=N*(-2*0.1194*theta+0.2500j)
    elif Phi=='Cotangent':
        cot=1/np.tan(0.6407*theta)
        ncsc=-0.6407/(np.sin(0.6407*theta)**2)
        zk=N*(0.5017*theta*cot-0.6122+0.2645j*theta)
        w=N*(0.2645j+0.5017*cot+0.5017*theta*ncsc)
    elif Phi=='Hyperbola':
        zk=2.246*N*(1-np.sin(1.1721-0.3443j*theta))
        w=2.246*N*(0.3443j*np.cos(1.1721-0.3443j*theta))
    elif Phi=='Best':
        ck,zk=DeterminePolesNResidues(N)
        return(ck,zk)
    else:
        print("Did not pick proper rational approximation dude")
        print("Quiting now")
        quit()

    ck=1.0j/N*np.exp(zk)*w
    return(ck,zk)

def RationalApprox(A,n0,t,N,ck,zk,tol=1e-12,maxits=2000):
    """
    Calculate the rational approximation solution for n(t)
    Inputs:
    A:          Matrix with system to be solved
    n0:         initial conditions of the system
    t:          time at which solution is determined
    N:          Number of quadrature points (should be less than 20)
    ck:         constants for quadrature solution
    zk:         constants for quadrature solution
    tol:        Tolerence for convergence for GMRES
    maxits:     Maximium iterations for GMRES
    Outputs:
```

```
    nt:         Solution at time t
    """
    nt=np.zeros(len(n0))
    for k in range(int(N/2)):
        if len(n0)>1:
            #phi,code=spla.gmres(zk[k]*sparse.identity(len(n0))-A*t,n0,
            #                    tol=tol,maxiter=maxits)
            phi=np.dot(np.linalg.inv(zk[k]*np.identity(len(n0))-A*t),
                       n0)
            #if (code):
            #    print(code)
        else:
            phi=(zk[k]-A*t)**(-1)*n0
        nt=nt-2*np.real(ck[k]*phi)
    return(nt)

################################################################
################## Plotting Functions ########################
################################################################

def reduceList(List,N):
    List2=[List[0]]
    Div=int(len(List)/N)
    for i in range(1,len(List)-1):
        if i % Div == 0:
            List2.append(List[i])
    List2.append(List[-1])
    return(List2)

def PlotPoints(Ntot,Nplot):
    t=1
def loop_values(list1,index):
    """
    This function will loop through values in list even if
    outside range (in the positive sense not negative)
    """
    while True:
        try:
            list1[index]
            break
        except IndexError:
            index=index-len(list1)
    return(list1[index])

def Legend(ax):
    handles,labels=ax.get_legend_handles_labels()
    ax.legend(handles,labels,loc='best',
              fontsize=LegendFontSize,prop=font)
    return(ax)

# def Legend(ax):
#         handles,labels=ax.get_legend_handles_labels()
#         box=ax.get_position()
#         ax.set_position([box.x0, box.y0, box.width*SquishGraph,
#                          box.height])
#         ax.legend(handles,labels,loc='center',
#                   bbox_to_anchor=(BBOXX,BBOXY),
#                   fontsize=LegendFontSize,prop=font,
#                   ncol=NumberOfLegendColumns)
#         return(ax)

def InList(item2,List):
  TF=False
  for item1 in List:
    if item1 == item2:
      TF=True
  if not TF:
    print("Invalid selection for plotting")
    print("Shuting down")
    quit()

def plot(df,Plotting,Name,NumOfPoints):
    #Plot In grams
    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:]

    Check=0
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:])/Na)*df[Item].values[0]
      if len(x)>NumOfPoints:
        x=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
      ax.plot(x,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item)
      Check=Check+1


    #Log or linear scale?
    ax.set_xscale(XScale)
    ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
```

Project Code continued on next page. . .

```
    ax.set_ylabel(Ylabel,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_grams.pdf')

    #Plot in Bq ################################

    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:]

    Check=0;Sum=np.zeros(len(x))
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:]))*df[Item].values[1]
      Sum=Sum+y
      if len(x)>NumOfPoints:
        xP=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
      else:
        xP=x.copy()
      ax.plot(xP,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item)
      Check=Check+1
    if len(x)>NumOfPoints:
      Sum=reduceList(Sum,NumOfPoints)
    ax.plot(xP,Sum,
            linestyle=loop_values(LineStyles,Check),
            marker=loop_values(MarkerType,Check),
            color=loop_values(Colors,Check),
            markersize=loop_values(MarkSize,Check),
            alpha=loop_values(Alpha_Value,Check),
            label="Sum")

    #Log or linear scale?
    ax.set_xscale(XScale)
    if sum(Sum)==0:
      ax.set_yscale('linear')
    else:
      ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
```

```python
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    YlabelBq="Activity $\\left[\\frac{Bq}{\\text{tHM}}\\right]$"
    ax.set_ylabel(YlabelBq,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_Bq.pdf')

def plots2(df,df2,Plotting,Name,NumOfPoints,Method1,Method2):
    #Plot in grams
    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:]
    x2=df2[List[0]].values[2:]

    Check=0
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:])/Na)*df[Item].values[0]
      y2=((df2[Item].values[2:])/Na)*df2[Item].values[0]
      if len(x)>NumOfPoints:
        x=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
      if len(x2)>NumOfPoints:
        x2=reduceList(x2,NumOfPoints)
        y2=reduceList(y2,NumOfPoints)
      ax.plot(x,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check)*1.5,
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method1)
      Check=Check+1
      ax.plot(x2,y2,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method2)
      Check=Check+1


    #Log or linear scale?
    ax.set_xscale(XScale)
    ax.set_yscale(YScale)
    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    ax.set_ylabel(Ylabel,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_Grams.pdf')


    #Plot in Bq
    fig=plt.figure(figsize=FigureSize)
    ax=fig.add_subplot(111)

    List=list(df.columns.values)
    x=df[List[0]].values[2:]
    x2=df2[List[0]].values[2:]

    Check=0;Sum=np.zeros(len(x));Sum2=np.zeros(len(x2))
    for Item in Plotting:
      InList(Item,List) #Check if we have the isotope
      y=((df[Item].values[2:]))*df[Item].values[1]
      y2=((df2[Item].values[2:]))*df2[Item].values[1]
      Sum=Sum+y
      Sum2=Sum2+y2
      if len(x)>NumOfPoints:
        xP=reduceList(x,NumOfPoints)
        y=reduceList(y,NumOfPoints)
      else:
        xP=x.copy()
      if len(x2)>NumOfPoints:
        xP2=redcueList(x2,NumOfPoints)
        y2=reduceList(y2,NumOfPoints)
      else:
        xP2=x2.copy()
      ax.plot(xP,y,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check)*1.5,
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method1)
      Check=Check+1
      ax.plot(xP2,y2,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
```

Project Code continued on next page...

```
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check),
              alpha=loop_values(Alpha_Value,Check),
              label=Item+" "+Method2)
      Check=Check+1
    if len(x)>NumOfPoints:
      Sum=reduceList(Sum,NumOfPoints)
    if len(x2)>NumOfPoints:
      Sum2=reduceList(Sum2,NumOfPoints)
    ax.plot(xP,Sum,
              linestyle=loop_values(LineStyles,Check),
              marker=loop_values(MarkerType,Check),
              color=loop_values(Colors,Check),
              markersize=loop_values(MarkSize,Check)*1.5,
              alpha=loop_values(Alpha_Value,Check),
              label="Sum "+Method1)
    Check=Check+1
    ax.plot(xP2,Sum2,
            linestyle=loop_values(LineStyles,Check),
            marker=loop_values(MarkerType,Check),
            color=loop_values(Colors,Check),
            markersize=loop_values(MarkSize,Check),
            alpha=loop_values(Alpha_Value,Check),
            label="Sum "+Method2)

    #Log or linear scale?
    ax.set_xscale(XScale)
    if sum(Sum)==0:
      ax.set_yscale('linear')
    else:
      ax.set_yscale(YScale)

    #Set Title
    fig.suptitle(Title,fontsize=TitleFontSize,
                 fontweight=TitleFontWeight,fontdict=font,ha='center')
    #Set X and y labels
    ax.set_xlabel(Xlabel,
                  fontsize=XFontSize,fontweight=XFontWeight,
                  fontdict=font)
    YlabelBq="Activity $\\left[\\frac{Bq}{\\text{tHM}}\\right]$"
    ax.set_ylabel(YlabelBq,
                  fontsize=YFontSize,
                  fontweight=YFontWeight,
                  fontdict=font)

    Legend(ax)
    plt.savefig("Plots/"+Name+'_Bq.pdf')

def ListToStr(List):
  Str=''
  for i in range(0,len(List)):
    if not i==len(List)-1:
      Str=Str+str(List[i])+","
    else:
      Str=Str+str(List[i])+"\n"
  return(Str)

def PrepFile(Name,n0,nuclide_names,atom_mass,decay_consts):
  File=open(Name,'w')
  File.write("Mass then Time (d),"+','.join(nuclide_names)+'\n')
  File.write("Masses,"+ListToStr(atom_mass)) #New line already included
  File.write("DecayConts,"+ListToStr(decay_consts))
  File.write("0,"+ListToStr(n0))
  return(File)

def Print(Method,nuclide,Results,Time,nuclides,atom_mass,nuclide_names):
  Index=nuclides[nuclide]
  MassConversion=atom_mass[Index]/Na
  string="Isotope "+nuclide_names[Index]+", Mass (g) = "
  Mass=Results[Index]*MassConversion
  Mass="%.4e" % Mass
  print(Method+" :",string,Mass,"Time=%.2f" % Time)


######################################################################
################# for calculatin phi ############################
######################################################################

def FindFissionXSection(Fissile_Isotopes):

    FissionXSections=np.zeros(len(Fissile_Isotopes))

    with open('Data/tape9.inp') as f:  #Save all X-section data to variable
        TAPE9Content=f.readlines()

    for i in range(0,len(Fissile_Isotopes)): #Loop through fissile isos
        parent=Fissile_Isotopes[i]

        for line in TAPE9Content: #Loop through x-section data
            hold=line.split()

            if '602' == hold[0] and hold[1]==parent:  #Find x-section
                FissionXSections[i]=hold[5]
                break
    return(FissionXSections)

def CalMevPerFiss(Fissile_Isotopes):
    """
    Given a list of fissile isotopes
    return a list of MeV/fission numbers
    calculated from an equation
    """
    MeVperFission=np.zeros(len(Fissile_Isotopes))
    for i in range(0,len(Fissile_Isotopes)):
        isotope=Fissile_Isotopes[i]
        proton=isotope[0:2]
        if isotope[2]=="0":
            Anum=isotope[3:5]
        elif isotope[4]=="0" and float(proton)*3<100:
            Anum=isotope[2:4]
        elif isotope[2]!="0":
            Anum=isotope[2:5]
        else:
            print("Missed logic in finding A number")
            quit()
        if int(Anum)<int(proton):
            print("Something is wrong, more protons than neutrons")
            print("Proton: ",proton,"A: ",Anum,"ZAID :",isotope)
            quit()
        MeVperFission[i]=1.29927*(10**-3)*(float(proton)**2)*(float(Anum)**0.5)+33.12
    return(MeVperFission)


def Calculatephi(FissionXSections,MeVperFission,n0,Power,Fissile_Isotopes,Nuclides):
    Sum=0
    for i in range(0,len(FissionXSections)):
        sigma=FissionXSections[i]*10**-24
        E=MeVperFission[i]
        N=n0[Nuclides[Fissile_Isotopes[i]]]
        Sum=Sum+sigma*E*N
    phi=(6.2414959617521E18*Power)/Sum
    return(phi)

def Makeb(Nuclides):

    b = np.zeros(len(Nuclides))
    b[Nuclides['922340']] = 6.94741E23
    b[Nuclides['922350']] = 7.6864E25
    b[Nuclides['922380']] = 2.4532E27
    return(b)



######################################################################
################# For Building A and b ############################
######################################################################

class DecayClass:
    def __init__(self):
        self.FBX    =  0. # 'The fraction of negatron beta decay transitions that results in
                          # in the daughter nuclide being in a relatively long-lived state'
                          # I think this should read the fraction of all decay events which are.
        self.IDFBX  = ''  # ZAID for daughter for FBX
        self.FPEC   =  0. # Fraction of all decay events which are positron or EC
        self.IDFPEC = ''  # ZAID for daughter for FPEC
        self.FPECX  =  0. # Fraction of all EC or positron decays which result in excited state
        self.IDFPECX= ''  # ZAID for daughter for FPECX
        self.FA     =  0. # Fraction of all decay events which are alpha
        self.IDFA   = ''  # ZAID for daughter for alpha
        self.FIT    =  0. # 'fraction of all the decay events of an excited nuclear state
                          # which result in the production of the ground state of the same nucli
        self.IDFIT  = ''  # ZAID for daughter of FIT
        self.FSF    =  0. # Fraction that decay events that are spontaneous fission
        # No daughter listed, will loop through all elements for yields
        self.FN     =  0. # Fraction of all decay events that are beta + neutron decays
        self.IDFN   = ''  # ZAID for daughter for FN
        #Note: Negatron beta decay = 1 - FBX - FPEC - FA - FIT - FSF - FN
        self.FB     =  0. # Fraction of all decay events which are beta
        self.IDFB   = ''


def FindPotentialMatch(List,protons,A,Fraction,Excited,LOUD=False):
    if(Fraction<0 or Fraction>1):
        print("Fraction of decays is too low or high : ",Fraction)
        print("Inquire further")
        quit()
    if Fraction>0:
        for item in List:
            if protons in item[0:2] and A in item[:-1] and item[-1]==Excited:
                Toreturn=item
    else:
        Toreturn=''
    try: #To make sure its defined
        Toreturn
    except NameError:
        if LOUD:
            print("Could not find daughter in list of isotopes when expecting one")
            print("Looking for Protons : ",protons," Total Nucleons : ",A," Fraction",Fraction)
            #print("Close items are")
            #for item in List:
            #    if protons in item[:-1] and A in item[:-1]:
            #        print(item)
        if (Fraction<2):
            if LOUD:
                print("Assuming it doesn't matter, will let slide")
            Toreturn=''
        else:
            quit()

    return(Toreturn)


def DecayInfo(Nuclide_Names,parent,Lambda,proton,A):
    """
    This function will store and return decay information from
    the tape9.inp file
    """

    Info=DecayClass()

    with open('Data/tape9.inp') as f:
        TAPE9Content=f.readlines()
    Found=False
    for line in TAPE9Content:
        hold=line.split()

        #Looking at second line in each library (put this if statement above below)
        if Found:
            #Spontaneous fission
            Info.FSF=float(hold[1])
            #Beta plus neutron
            Info.FN=float(hold[2])
            Info.IDFN=FindPotentialMatch(Nuclide_Names,str(int(proton)+1),str(int(A)-1),Info.FN,
            break

        #Looking for fission product and actinide decay information
        #The libraries we are looking through for this information are '2' and '3'
        if ('2' == hold[0] or '3' == hold[0]) and hold[1]==parent:
            Found=True

            #Beta minus to excited
            Info.FBX=float(hold[4])
            Info.IDFBX=FindPotentialMatch(Nuclide_Names,str(int(proton)+1),A,Info.FBX,'1')
            #positron or EC
            Info.FPEC=float(hold[5])  #Total positron or EC
            Info.FPECX=float(hold[6]) #percent of above to excited
            if Info.FPECX < 1:
                Info.IDFPEC=FindPotentialMatch(Nuclide_Names,str(int(proton)-1),A,Info.FPEC,'0')
            Info.IDFPECX=FindPotentialMatch(Nuclide_Names,str(int(proton)-1),A,Info.FPECX,'1')
            Info.FA=float(hold[7])    #Fraction of events that are alpha
            Info.IDFA=FindPotentialMatch(Nuclide_Names,str(int(proton)-2),str(int(A)-4),Info.FA,
            #Excited state to ground state
            Info.FIT=float(hold[8])
            Info.IDFIT=FindPotentialMatch(Nuclide_Names,proton,A,Info.FIT,'0')

    #Calculate the beta
    Info.FB=1-Info.FBX-Info.FPEC-Info.FA-Info.FIT-Info.FSF-Info.FN

    if(Info.FB<0 or Info.FB>1):
        #print("Fraction of beta decays is too low or high : ",Info.FB)
        if abs(Info.FB)<7e-4 or abs(Info.FB-1)<7e-4:
            #print("But I'll let it slide and set to zero")
            Info.FB=0
        else:
            print("I can't let this slide, not small enough")
            print("Beta excited",Info.FBX)
            print("EC",Info.FPEC)
            print("Alpha decay ground",Info.FA)
            print("Excited to ground",Info.FIT)
            print("Spontaneous fission",Info.FSF)
            print("Beta Plus neutron",Info.FN)
            quit()

    if Lambda>0:
            Info.IDFB=FindPotentialMatch(Nuclide_Names,str(int(proton)+1),A,Info.FB,'0')

    #Make sure all decay fractions add to one (for some reason the EC to excited is
    # probability given a EC
    FPEC=(1-Info.FPECX)*Info.FPEC
    FPECX=Info.FPECX*Info.FPEC
    Info.FPEC=FPEC
    Info.FPECX=FPECX

    return(Info)


def FindPotentialMatchX(List,protons,A,XSection,Excited,Lambda,LOUD=False):
    if(XSection<0):
        print("Fraction of decays is too low",XSection)
        print("Inquire further")
        quit()
    if XSection>0:
        for item in List:
            if protons in item[0:2] and A in item[:-1] and item[-1]==Excited:
                Toreturn=item
    else:
        Toreturn=''
    try: #To make sure its defined
        Toreturn
    except NameError:
        if LOUD:
            print("Could not find daughter in list of isotopes when expecting one")
            print("Looking for Protons : ",protons," Total Nucleons : ",A," XSection ",XSection)
            print("Lambda = ",Lambda)
            #print("Close items are")
            #for item in List:
            #    if protons in item[:-1] and A in item[:-1]:
            #        print(item)
        if (XSection<100000 or Lambda>0.1):
            if LOUD:
                print("Assuming it doesn't matter, will let slide")
            Toreturn=''
            XSection=0
        else:
            quit()

    return(Toreturn,XSection)

class XSectionClass:
    def __init__(self):
        self.SNG   = 0     # the effective, one groupn (n,y) x-section leadin to ground state
        self.IDSNG = ''    # ZAID ID for the above
        self.SN2N    =  0. # the effective, one groupn (n,2n) x-section leading to ground state
        self.IDSN2N = ''   # ZAID ID for the above
        self.SN3N  =   0   # effective to ground
        self.IDSN3N  = ''  # ZAID for daughter for above
        self.SNA   =  0.   # effective to ground n,alpha
        self.IDSNA = ''     # ZAID for daughter for above
        self.SNF  =  0.
        self.SNP     =  0. #
        self.IDSNP   = ''  # ZAID for daughter for above
        self.SNGX    =  0. #
        self.IDSNGX  = ''  # ZAID for daughter of above
        self.SN2NX     =  0. #
        self.IDSN2NX   = ''  # ZAID for daughter for above


def XSectionInfo(Nuclide_Names,parent,L,proton,A):
    """
    This function will store and return decay information from
    the tape9.inp file
    """

    Info=XSectionClass()

    with open('Data/tape9.inp') as f:
        TAPE9Content=f.readlines()

    Found=False
    for line in TAPE9Content:
        hold=line.split()

        #Looking for fission product and actinide cross section information
        #The libraries we are looking through for this information are '602' and '603'
        #set a cross section to zero if daughter is not found (second return of FindPotentialMat
        if ('602' == hold[0] or '603' == hold[0]) and hold[1]==parent:

            #(n,gamma)
            Info.SNG=float(hold[2])
            Info.IDSNG,Info.SNG=FindPotentialMatchX(Nuclide_Names,proton,str(int(A)+1),Info.SNG,
            #n,2n
            Info.SN2N=float(hold[3])
            Info.IDSN2N,Info.SN2N=FindPotentialMatchX(Nuclide_Names,
                                                      proton,str(int(A)-1),Info.SN2N,'0',L)
            if '602' == hold[0]: #Actinides
                #n,3n
                Info.SN3N=float(hold[4])
                Info.IDSN3N,Info.SN3N=FindPotentialMatchX(Nuclide_Names,
                                                          proton,str(int(A)-2),Info.SN3N,'0',L)
                #n,f
                Info.SNF=float(hold[5])
            if '603' == hold[0]: #Fission products
                #n,alpha
                Info.SNA=float(hold[4])
                Info.IDSNA,Info.SNA=FindPotentialMatchX(Nuclide_Names,str(int(proton)-2),str(int
                                                    Info.SNA,'0',L)
                #n,p
                Info.SNP=float(hold[5])
                Info.IDSNP,Info.SNP=FindPotentialMatchX(Nuclide_Names,
                                                        str(int(proton)-1),A,Info.SNP,'0',L)
            #(n,gamma)excited
            Info.SNGX=float(hold[6])
            Info.IDSNGX,Info.SNGX=FindPotentialMatchX(Nuclide_Names,proton,
                                                     str(int(A)+1),Info.SNGX,'1',L)
            #(n,2n)excited
            Info.SN2NX=float(hold[7])
            Info.IDSN2NX,Info.SN2NX=FindPotentialMatchX(Nuclide_Names,proton,
                                                       str(int(A)-1),Info.SN2NX,'1',L)

    return(Info)

def YieldInfo(yieldiso,holdIndex,LOUD=False):
    with open('Data/tape9.inp') as f:
        TAPE9Content=f.readlines()

    Found=False;Yield=False
    for line in TAPE9Content:
        hold=line.split()

        if Found:
            if Yield:
                returnyield=float(hold[holdIndex])
            else:
                returnyield=0
            break
        if '603' == hold[0] and hold[1]==yieldiso:
            Found=True
            if float(hold[8])>0:
                Yield=True
            else:
                Yield=False

    if not Found:
        if LOUD:
            print("Did not find a yield for ",yieldiso)
        returnyield=0
    return(returnyield)

def AddFission(A,Nuclides,isotope,c1,c2,row,LOUD=False):
    if isotope=="922320": #Th232
        #print("Th232")
        holdIndex=1
        element="Th232"
    elif isotope=="922330": #U233
        #print("U233")
        holdIndex=2
        element="U233"
    elif isotope=="922350": #U235
        #print("U235")
        holdIndex=3
        element="U235"
    elif isotope=="922380": #U238
        #print("U238")
```

```
            holdIndex=4
            element="U238"
        elif isotope=="942390": #Pu239
            #print("Pu239")
            holdIndex=5
            element="Pu239"
        elif isotope=="942410": #Pu241
            #print("Pu241")
            holdIndex=6
            element="Pu241"
        elif isotope=="962450": #Cm245
            #print("Cm245")
            holdIndex=7
            element="Cm245"
        elif isotope=="982490": #Cf249
            #print("Cf249")
            holdIndex=8
            element="Cf249"
        else:
            if LOUD:
                print("Did not find yields for ",isotope," because not provided")
            holdIndex=100

        if holdIndex<40:
            YieldSum=0 #Check what the yields sum up to
            for yieldiso in Nuclides:
                actualrow=Nuclides[yieldiso]
                Yield=YieldInfo(yieldiso,holdIndex,LOUD=False)
                A[actualrow,row]=A[actualrow,row]+c1*c2*Yield/100
                YieldSum=YieldSum+Yield
            if LOUD:
                print("Yield Sum for ",element," = ",YieldSum)
        return(A)

    def MakeAb(phi,Nuclides,Nuclide_Names,Decay_Conts):

        # Create Activation and Decay Matrix and initial
        # nuclide quantity vector
        A = np.zeros((len(Nuclides),len(Nuclides)))

        #10^14 1/cm^2/s in 1/cm^2 /year #Bars included
        phi = phi * 60 * 60 * 24 * 365.25 *(10**(-24))


        for isotope in Nuclides:
            row = Nuclides[isotope]
            proton=isotope[0:2]
            if isotope[2]=="0":
                Anum=isotope[3:5]
            elif isotope[4]=="0" and float(proton)*3<100:
                Anum=isotope[2:4]
            elif isotope[2]!="0":
                Anum=isotope[2:5]
            else:
                print("Missed logic in finding A number")
                quit()
            if int(Anum)<int(proton):
                print("Something is wrong, more protons than neutrons")
                print("Proton: ",proton,"A: ",Anum,"ZAID :",isotope)
                quit()

            #print(row,isotope,Decay_Conts[row])
            Lambda=Decay_Conts[row]
            #Store all decay and x section information for row (I think I mean column)
            row_decay =  DecayInfo(Nuclide_Names,isotope,Lambda,proton,Anum)
            row_XSection=XSectionInfo(Nuclide_Names,isotope,Lambda,proton,Anum)

            #Convert LAmbda to years^-1 (If radioactive)
            if Lambda>0:
                Lambday=Lambda*60*60*24*365.25
            else:
                Lambday=0

            XList=["SN2N","SN2NX","SN3N","SNA","SNG","SNGX","SNP","SNF"]
            DList=["FA","FB","FBX","FIT","FN","FPEC","FPECX","FSF"]

            #USeful to see things
            #for a in dir(row_XSection):
            #    if not a.startswith('__'):
            #        print(a)
            #        print(getattr(row_XSection,a))
            #quit()
            #for a in dir(row_decay):
            #    if not a.startswith('__'):
            #        print(a)
            #        print(getattr(row_decay,a))
            #quit()

            #Sum up all the sigma abs
            sigma_sum=0
            for xsec in XList:
                sigma_sum=sigma_sum+getattr(row_XSection,xsec)

            # Diagonal Assignment
            A[row,row] = -Lambday - phi*sigma_sum

            ##does a single isotope produce another isotope through more than one path
            ##like EC and NP
            #for xsec in XList:
            #    for decay in DList:
            #        if not decay[-1]=='F' and not "F" in xsec and len(getattr(row_decay,"ID"+decay)
            #            if getattr(row_XSection,"ID"+xsec) == getattr(row_decay,"ID"+decay):
            #                print("Isotope ",getattr(row_XSection,"ID"+xsec),"Produced from ",
            #                      isotope," with RNS ",decay,xsec)
            #                print("Both will be used")

```

Project Code continued on next page...

```
            #Off diagonal assignment adding x-sec productions except from fission
            for xsec in XList:
                if not "F" in xsec: #Don't do fission yet
                    Product=getattr(row_XSection,"ID"+xsec)
                    if len(Product)>1:
                        actualrow=Nuclides[Product]
                        A[actualrow,row]=A[actualrow,row]+phi*getattr(row_XSection,xsec)
            #Off diagonal assignment adding all decay except from spontaneous fission
            for decay in DList:
                if not decay[-1]=='F':
                    Product=getattr(row_decay,"ID"+decay)
                    if len(Product)>1:
                        actualrow=Nuclides[Product]
                        A[actualrow,row]=A[actualrow,row]+Lambday*getattr(row_decay,decay)

            #Now for xfission (these next two if statements take the longest)
            if row_XSection.SNF>0:
                A=AddFission(A,Nuclides,isotope,row_XSection.SNF,phi,row,LOUD=False)
            #Now for spontaneous fission
            if row_decay.FSF>0:
                A=AddFission(A,Nuclides,isotope,row_decay.FSF,Lambday,row,LOUD=False)

    b = np.zeros(len(Nuclides))
    b[Nuclides['922340']] = 6.94741E23
    b[Nuclides['922350']] = 7.6864E25
    b[Nuclides['922380']] = 2.4532E27

    return(A,b)
```