Homework 3

Simon Bolding
NUEN 629

October 26, 2015

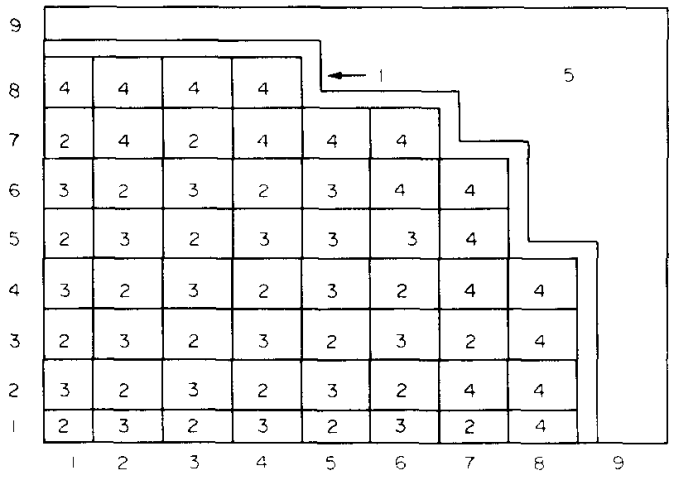
NUEN 629, Homework 3

Due Date Oct. 23

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 Goth and Hammer

(150 points + 50 points extra credit) The Zion PWR Benchmark (K.S. Smith, NSE, 1986), is a 2-D, two-group, reactor benchmark calculation where a quarter reactor is specified by the following figure:



Composition	Group, g	D_g	Σ_{ag}	$\nu\Sigma_{fg}$	$\Sigma_{gg'}$
1	1	1.02130	0.00322	0.0	0.0
	2	0.33548	0.14596	0.0	0.0
2	1	1.47160	0.00855	0.00536	0.01742
	2	0.37335	0.06669	0.10433	0.0
3	1	1.41920	0.00882	0.00601	0.01694
	2	0.37370	0.07606	0.12472	0.0
4	1	1.42650	0.00902	0.00653	0.01658
	2	0.37424	0.08359	0.14120	0.0
5	1	1.45540	0.00047	0.0	0.02903
	2	0.28994	0.00949	0.0	0.0

$X_1 = 1.0$, $X_2 = 0.0$.
Assembly pitch: 21.608 cm. Baffle thickness: 2.8575 cm.
Boundary condtions: reflective: left, bottom, zero flux: top, right.
$\Sigma_{tr,g} = 1/3D_g$ for isotropic scattering, transport problem.

Figure 1: Zion PWR benchmark from K.S. Smith, NSE, 1986.

A quarter of the reactor is a square that is divided in the figure into a 9 x 9 grid of 21.608 x 21.608 cm squares (i.e., the quarter reactor size is 194.472 x 194.472 cm). At the midline there are 8 assemblies in the x and y directions. In the problem specification there is only downscattering and what we have called Σ_{rg} is called Σ_{ag}. Also, note that all fission neutrons are born fast.

Your tasks are to:

1. Solve this problem using the finite difference method. In solving the problem you should find k_{eff} and the fission power in each assembly. Indicate what mesh resolution you need to converge k_{eff} to 1 pcm (10^{-5}). What is the time to solution for this problem?

2. Repeat part 1 using the nodal method we derived in class.

3. A critical buckling search is a method for finding the reactor height needed to make the reactor critical. We do this by adjusting the removal cross-section as

$$\Sigma_{rg} \rightarrow \Sigma_{rg} + D_g \frac{\pi^2}{H^2},$$

   and iterating on the value of H until $k_{\text{eff}} = 1.$. Using either the finite difference or nodal method, estimate the critical height of this reactor.

4. (Extra Credit) Repeat part 2 using CMFD acceleration. How does the time to solution change?

2 250 points extra credit

(150 points + 50 points extra credit) The HAFAS BWR Benchmark, say it out loud (K.S. Smith, NSE, 1986), is a 2-D, two-group, reactor benchmark calculation with pin-level homogenization where a quarter reactor is specified by different assemblies as shown in the following figure:

In the problem specification there is only downscattering and what we have called Σ_{rg} is called Σ_{ag}. Also, note that all fission neutrons are born fast.

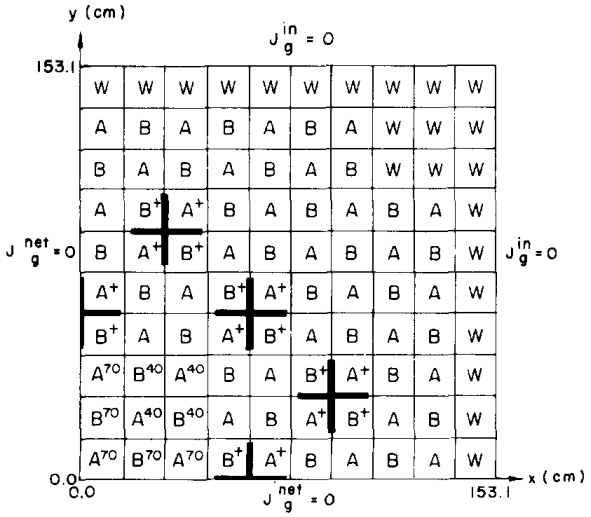
Your tasks are to:

1. (100 points) Solve this problem using the finite difference method. In solving the problem you should find k_{eff} and the fission power in each assembly. Indicate what mesh resolution you need to converge k_{eff} to 1 pcm (10^{-5}). What is the time to solution for this problem?

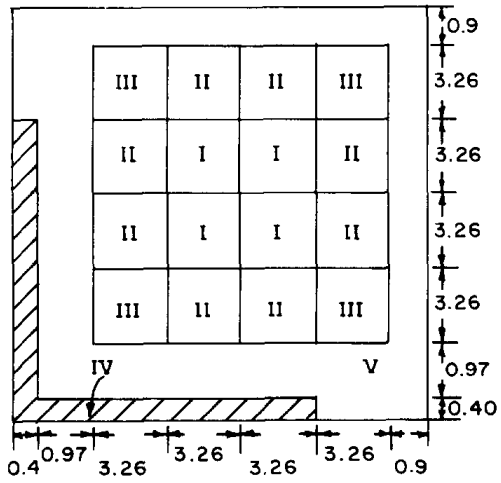
2. (100 points) Repeat part 1 using the nodal method we derived in class.

3. (50 points) Repeat part 2 using CMFD acceleration. How does the time to solution change?

Quadrant of the two-dimensional Reactor



Fuel Assembly Description



Composition-to-Zone Assignments

Zone	Assembly Type A	A^{40}	A^{70}	A^+	B	B^{40}	B^{70}	B^+	W
I	1	5	9	1	2	6	10	2	15
II	2	6	10	2	3	7	11	3	15
III	3	7	11	3	4	8	12	4	15
IV	13	13	13	14	13	13	13	14	15
V	13	13	13	13	13	13	13	13	15

Cross sections

Composition	D_1	Σ_{a1}	$\nu\Sigma_{f_1}$	$\Sigma_{1\to 2}$	D_2	Σ_{a2}	$\nu\Sigma_{f_2}$
1	1.400	0.0090	0.0065	0.0160	0.375	0.080	0.1220
2	1.400	0.0090	0.0057	0.0170	0.375	0.070	0.1000
3	1.400	0.0090	0.0051	0.0180	0.375	0.060	0.0800
4	1.400	0.0090	0.0051	0.0180	0.375	0.050	0.0700
5	1.680	0.0080	0.0063	0.0100	0.530	0.077	0.1180
6	1.680	0.0085	0.0055	0.0105	0.530	0.067	0.0960
7	1.680	0.0090	0.0049	0.0110	0.530	0.057	0.0780
8	1.680	0.0090	0.0049	0.0110	0.530	0.047	0.0680
9	2.000	0.0078	0.0061	0.0052	0.800	0.073	0.1140
10	2.000	0.0082	0.0053	0.0053	0.800	0.063	0.0920
11	2.000	0.0086	0.0047	0.0054	0.800	0.053	0.0720
12	2.000	0.0086	0.0047	0.0054	0.800	0.043	0.0620
13	1.530	0.0005	0.0	0.0310	0.295	0.009	0.0
14	1.110	0.08375	0.0	0.00375	0.185	0.950	0.0
15	2.000	0.0	0.0	0.0400	0.300	0.010	0.0

$\chi_1 = 1.0$, $\chi_2 = 0.0$, $\nu = 2.5$.
Boundary conditions: reflective: left, bottom, zero incoming flux: top, right.
$\Sigma_{tr,g} = 1/3D_g$ for isotropic scattering, transport problem.

Figure 2: HAFAS BWR benchmark from K.S. Smith, NSE, 1986.

Solution 1-1:

The 2D lattice input code provided in lab was modified to generate the input for the first benchmark problem. Because the provided code assumes a uniform mesh size, approximations must be made to resolve the baffle in the geometry. I have assumed that if a particular cells center lies with in the region of the baffle, then that cell has the baffle material properties. Also, I have assumed that the assemblies near the reflecting boundary are full assemblies, rather than half assemblies. To match the edges of the assembly exactly, I used numbers of cells in the x and y direction that are factors of 9. It is noted that this will cause a jump in convergence of the solution in terms of mesh size, caused at points where the cell widths are small enough that an additional baffle cell can be inserted.

The assembly power is averaged over each cell in a 9x9 grid over the quarter reactor, weighted by $\nu\Sigma_f$ in each group. Since we do not know Σ_f individually this is not exact since ν is larger for the fast group, generally. Since we are normalizing, this is not a great error, and produces a zero assembly power in the water regions as desired.

A convergence table versus mesh size as a function of the number of cells is given below. It appears that the solution is convergent above 3969 cells and that it would reach a tolerance of 1 pcm around 12,000 cells, noting that there is some variablity due to the baffle approximations. The time to solution at 8100 cells is 1600 seconds.

Table 1: Convergence table for finite difference method versus number of mesh cells.

81	1.2786193689	–
324	1.27611471632	0.00250465258332
729	1.27609454035	2.01759703702e-05
1296	1.27623214214	0.00013760179336
2025	1.27586573839	0.000290925744068
2916	1.27563942334	0.000177413023663
3969	1.27549513858	0.000113120586226
5184	1.27540107208	7.37544516677e-05
6561	1.27533957529	4.82199314754e-05
8100	1.27530010292	3.09514302439e05

Figure 1: Material ID's for Problem 1. There is a slight error due to how pcolor interpolates linear values at cell centers, so the entire picture is shifted

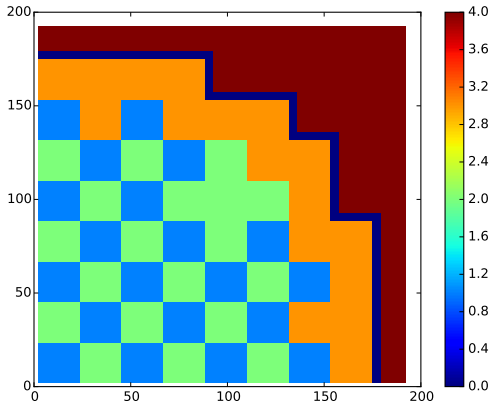


Figure 2: Normalized group 1 fluxes for finite difference solution

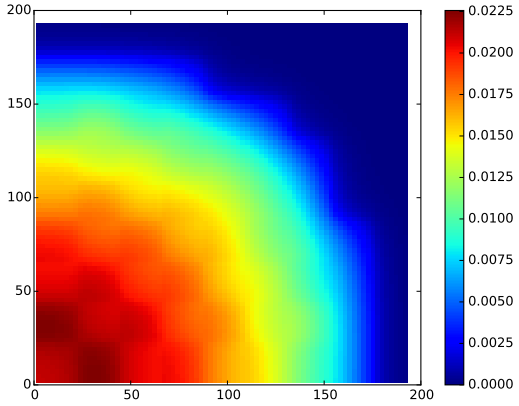


Figure 3: Normalized group 2 fluxes for finite difference solution

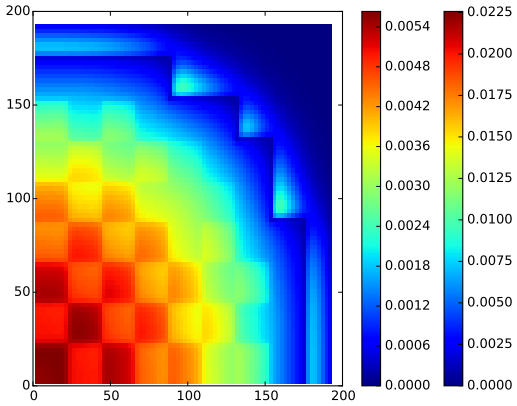
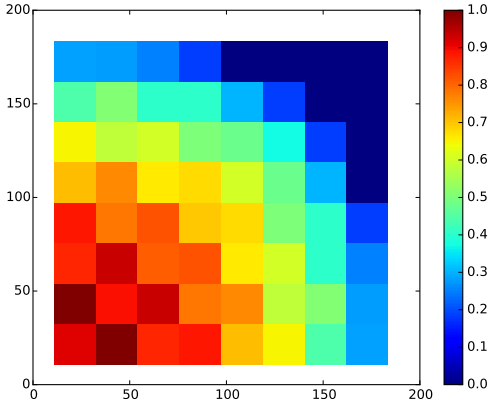


Figure 4: Assembly (assuming 9x9 grid) averaged powers for problem 1, normalized to the max assembly power, solved using finite differences. Note: there is a slight error in the way pcolor is interpolating values.



Solution 1-2:

The provided k code and nodal code were modified to solve k problems using the nodal method. The outer iterations were essentailly unmodified, with only the 2D steady fixed-source solve modified. The provided algorithms for the nodal method in class had to be modified to produce a stable algorithm. Although it is not the most efficient, the method is modified to use internal solution for computing the transverse total currents. For example, for the nodal solve in the x direction, the y transverse leakage term becomes

$$L_y, ij(x) = \frac{1}{h_y}\left(\overline{J}^y_{ij}(x_i, y_j + h_y/2) - \overline{J}^y_{ij}(x_i, y_j - h_y/2)\right)$$

where, for simplicity, the transverse current has also been assumed as a piecewise constant spatially along the direction of the nodal solve, and the L_x and L_y terms in their respective solves are lagged for the entire sweeps (Jacobi iteration). Because I have used Marshak or reflective boundary conditions, using the same equation for boundary cells also strictly enforces the boundary condition.

The iteration would likely be more efficient if the outward partial currents from the cell were used for the transverse leakages, with the inflowing currents coming from adjacent cells, however I did not test this approach for stability. If we use the total current from adjacent cells for the leakage terms, then the iteration is not stable. I believe this is due to the fact that the transverse current (which was computed by the previous nodal solve in the transverse direction) was computed using a lagged inflow to the adjacent cell. We are now using that lagged inflow as the transverse outflow within our cell. Thus, the transverse outflow to the cell of interest is 2 iterations behind, and this seems to be too inaccurate for stability.

Due to time constraints and because the nodal iteration was very slow, I did not attempt to converge k_{eff} to 1.E-05. A plot of the group fluxes and assembly powers are given below for the case of 729 cells. For this case k_{eff} was found to be 1.274. This is in agreement with the finite difference solution, noting that the baffle is extra large since there is only 2 cells in each assembly in this case. With a loose tolerance of 1.E-05 on all iteration loops, this solve took around 5000 seconds, so this method is very slow without the CMFD acceleration. The thermal flux appears to not be fully resolved near the baffle.

Figure 5: Normalized group 1 fluxes for nodal method solution with 729 cells

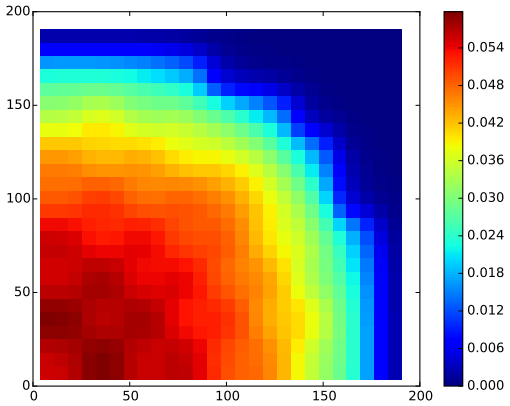


Figure 6: Normalized group 2 fluxes for nodal method with 729 cells

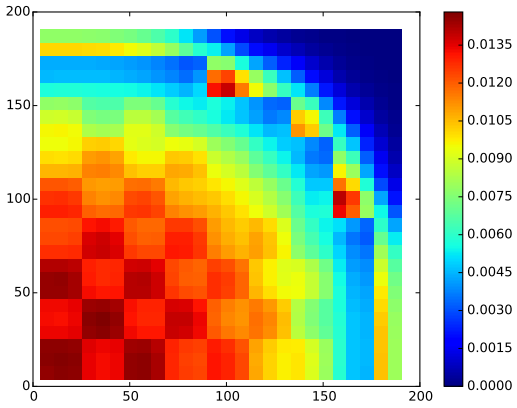
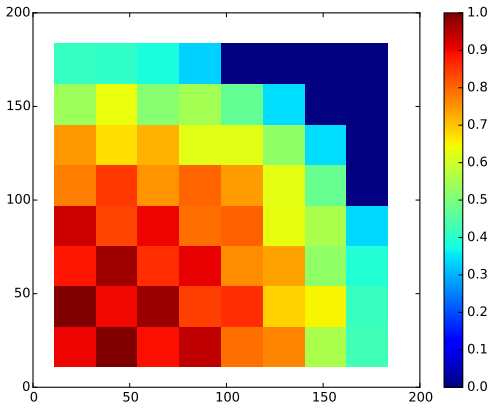


Figure 7: Assembly (assuming 9x9 grid) averaged powers for problem 1, normalized to the max assembly power, solved using nodal method. Same issues with pcolor interpolating strangely, the edge of boxes are actualy the center of assemblies



Solution 1-3:

Because the nodal code is very slow, the finite difference code was used to determine the critical reactor height. The secant method was used to determine the critical height to a relative tolerance in k_{eff} of 1.E-05, taking 11 iterations on H from an initial educated guess of 50 cm. A mesh was used with 729 elements. The critical height was found to be 46.85 cm, producing a k_{eff} of 0.999996. This is a reasonable number since the 2D reactor had a relatively high k_{eff} of $\sim$1.27.

Code

```
def coordLookup_l(i, j, k, I, J):
    """get the position in a 1-D vector
    for the (i,j,k) index
    """
    return i + j*I + k*J*I

def coordLookup_ijk(l, I, J):
    """get the position in a (i,j,k)  coordinates
    for the index l in a 1-D vector
    """
    k = (l // (I*J)) + 1
    j = (l - k*J*I) // I + 1
    i = l - (j*I + k*J*I)-1
    return i,j,k

def hetero_node1D(N, D, Sigma_a, Q, h, BCs, maxits = 10, tol = 1.0e-10, LOUD=False, phi_solution=0):
    if (type(phi_solution) != np.ndarray):
        phi_solution = np.zeros((N,5))
    phi_new = phi_solution.copy()
    iteration = 1
    converged = 0
    localBCs = np.zeros((2,3))
    while not(converged):
        for i in range(N):
            if not(i==0):
                phi,a1,a2,a3,a4 = phi_solution[i-1,:]
                C = positive_current(phi_solution[i-1,:],h/2,h,D[i-1])
                #print("i =",i,"Cr =",C)
                localBCs[0,0:3] = [0.25,-D[i]/2,C]
            else:
                localBCs[0,:] = BCs[0,:].copy()
            if not(i==(N-1)):
                phi,a1,a2,a3,a4 = phi_solution[i+1,:]
                C = negative_current(phi_solution[i+1,:],-h/2,h,D[i+1])
                #print("i =",i,"Cr =",C)
                localBCs[1,0:3] = [.25,D[i]/2,C]
            else:
                localBCs[1,:] = BCs[1,:].copy()
            #print(localBCs)
            phi_new[i,:] = single_node1GVacuum(D[i],Sigma_a[i],Q[i,:],h,localBCs)
            phi,a1,a2,a3,a4 = phi_new[i,:]
            #print(i,"incoming current on left =", localBCs[0,2],positive_current(phi_new[i,:],-h/2,h,D[i])
            if 0*(i>0):
                print(i,"outgoing current on left =", negative_current(phi_new[i-1,:],h/2,h,D[i]),negative
            if 0*(i<N-1):
                print(i,"outgoing current on right =", positive_current(phi_new[i+1,:],-h/2,h,D[i]),positiv
            #print(i,"incoming current on right =", localBCs[1,2],negative_current(phi_new[i,:],h/2,h,D[i])
            current_left = -(D[i]*(a1/h + (3*a2)/h + a3/(2.*h) + a4/(5.*h)))
            #print("zone ",i," current in at right:",localBCs[1,2]," current out at right:",current_left)
        relchange = np.linalg.norm( np.reshape(phi_new-phi_solution, 5*N))/np.linalg.norm( np.reshape(phi_n
        converged = (relchange < tol) or (iteration >= maxits)
        if (LOUD):
            print("Iteration",iteration,": relative change =",relchange)
        iteration += 1
        phi_solution = phi_new.copy()
    return phi_solution, iteration

import numpy as np
from math import pi
import re
import scipy as sp
import scipy.sparse as sparse
import scipy.sparse.linalg as splinalg

def single_node1GVacuum(D,Sigma_a,Q,h, BCs):
```

```
    A = np.zeros((5,5))
    b = np.zeros(5)

    Al = BCs[0,0]
    Bl = BCs[0,1]
    Cl = BCs[0,2]
    Ar = BCs[1,0]
    Br = BCs[1,1]
    Cr = BCs[1,2]
    #equation 1
    A[0,:] = [Sigma_a,0,(-6*D)/(h**2),0,(-2*D)/(5.*(h**2))]
    b[0] = Q[0]
    #equation 2
    A[1,:] = [Al,-Al/2. + Bl/h,Al/2. - (3*Bl)/h,Bl/(2.*h),-Bl/(5.*h)]
    b[1] = Cl
    #equation 3
    A[2,:] = [Ar,Ar/2. + Br/h,Ar/2. + (3*Br)/h,Br/(2.*h),Br/(5.*h)]
    b[2] = Cr
    #equation 4
    A[3,:] = [0,(h*Sigma_a)/12.,0,-D/(2.*h) - (h*Sigma_a)/120.,0]
    b[3] = (h*Q[1])/12. - (h*Q[3])/120.
    #equation 5
    A[4,:] = [0,0,(h*Sigma_a)/20.,0,-D/(5.*h) - (h*Sigma_a)/700.]
    b[4] = (h*Q[2])/20. - (h*Q[4])/700.
    phi = np.linalg.solve(A,b)
    return phi

# In[459]:
def alpha_mg_diffusion(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
                       chig,D,v,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-9,maxits = 20, alpha = 1000, LOUI
    iteration = 0
    converged = False
    while not(converged):
        Sigmarg_alpha = Sigmarg + alpha/v
        x,y,z,k,iterations,phig = kproblem_mg_diffusion(G,(I,J,K),(Nx,Nx,Nx),BCGs,Sigmarg_alpha,Sigmasgg,n
                                                        chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-2,
                                                         maxits = 100, k = 1, LOUD=False)
        if (iteration > 0):
            #update via secant method
            slope = (k-kold)/(alpha-alpha_old)
        else:
            Sigmarg_alpha = Sigmarg + (alpha+tol)/v
            x,y,z,kperturb,iterations,phig = kproblem_mg_diffusion(G,(I,J,K),(Nx,Nx,Nx),BCGs,Sigmarg_alpha
                                                        chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-12,
                                                         maxits = 100, k = 1, LOUD=False)
            slope = (kperturb - k)/tol
        alpha_old = alpha
        kold = k
        alpha = (1-k)/slope + alpha
        converged = (np.abs(k - 1) < tol) or (iteration >= maxits)
        iteration += 1
        if (LOUD):
            print("=================================")
            print("Iteration",iteration,": k =",k, "alpha =",alpha)
    return x,y,z,alpha,iteration,phig

def assembly_powers(width_x, width_y, nx, ny, phi_g, Sigmafg):

    dx = width_x/nx
    dy = width_y/ny
    x_bounds = [0.]
    y_bounds = [0.]
    powers = np.zeros((nx,ny))
    x = y = 0.0
    for i in range(nx):
        x += dx
        y += dy
        x_bounds.append(x)
```

```
134             y_bounds.append(y)
135
136         hx = width_x/phi_g.shape[0]
137         hy = width_y/phi_g.shape[1]
138
139         #Loop over all cells and groups. Volume of cells doesnt matter because it will be renormalized anyways
140         for g in range(phi_g.shape[3]):
141             for i in range(phi_g.shape[0]):
142                 for j in range(phi_g.shape[1]):
143
144                     x = (i+0.5)*hx
145                     y = (j+0.5)*hy
146
147                     #Find the cell you belong to
148                     for xi in range(len(x_bounds)-1):
149                         for yi in range(len(y_bounds)-1):
150                             if x > x_bounds[xi] and x < x_bounds[xi+1]:
151                                 if y > y_bounds[yi] and y < y_bounds[yi+1]:
152                                     xid = xi
153                                     yid = yi
154                                     break
155
156                     powers[xid][yid] += phi_g[i,j,0,g]*Sigmafg[i,j,0,g]
157
158         xcents = [0.5*(x_bounds[i] + x_bounds[i+1]) for i in range(len(x_bounds)-1)]
159         ycents = [0.5*(y_bounds[i] + y_bounds[i+1]) for i in range(len(y_bounds)-1)]
160
161         #Normalize the powers
162         pmax = np.amax(powers)
163         powers *= 1./pmax
164
165         print(powers)
166
167         #plot the powers
168         plt.figure()
169         plt.pcolor(np.array(xcents),np.array(ycents),powers)
170         plt.colorbar()
171         plt.savefig("powers.pdf")
172
173
174
175 #Helper functions
176 def plot_node(phi,x,h):
177     return (phi[0] + (phi[1]*x)/h + phi[2]*(-0.25 + (3*x**2)/(h**2))
178             + phi[3]*(-x/(4.*h) + (x**3)/(h**3)) +
179             phi[4]*(0.0125 - (3*(x**2))/(10.*(h**2)) + (x**4)/(h**4)))
180
181 def current(phi,y,h,D):
182     phi,a1,a2,a3,a4 = phi
183     return -(D*(a1/h + (6*a2*y)/h**2 + a3*(-1/(4*h) + (3*y**2)/h**3) + a4*((-3*y)/(5*h**2) + (4*y**3)/h**4
184
185 def positive_current(phi,y,h,D):
186     J = current(phi,y,h,D)
187     scalar_flux = plot_node(phi,y,h)
188     return 0.25*scalar_flux + 0.5*J
189
190 def negative_current(phi,y,h,D):
191     J = current(phi,y,h,D)
192     scalar_flux = plot_node(phi,y,h)
193     return 0.25*scalar_flux - 0.5*J
194
195 def transverse_leakage_dof(neighbor_phis,y,h_trans,h_parr,D):
196
197     #These are the three phis in neighboring cell, in transverse direction,
198     #that we need to compute leakages for
199  #    print("The passed in phis are",neighbor_phis)
200     nbr_Js = [current(neighbor_phis[i],y,h_trans,D[i]) for i in range(len(neighbor_phis))]
201  #    print("The ngbr J's are",nbr_Js)
```

```
202
203     #Current on faces have the form: J_avg + l1/x*(x/h) + l2*(3x^3/h^2 - 1/4)
204     #where x is direction parallel to solve, not transverse
205     J_avg =  nbr_Js[1]
206     l1 = (nbr_Js[2] - nbr_Js[0])/(2)
207     l2 = (nbr_Js[2] + nbr_Js[0] - 2*nbr_Js[1])
208
209  #   print("THe reconstructed nbrs are",J_avg-l1/2+l2/2,J_avg,J_avg+l1+l2/2)
210
211  #   print("The returned J's are",J_avg,l1,l2)
212     return (J_avg, 0.,0.)
213     return (J_avg, l1, l2)
214
215
216 def nodal2D_steady_fixed_source(Dims,Lengths,BCs,D,Sigma,Q, tolerance=1.0e-12, phi_solution=0.,
    LOUD=False, maxits=100):
217     """Solve a steady state, single group diffusion problem with a fixed source using 2D nodal method
218     Inputs:
219         Dims:             number of zones (I,J,K)
220         Lengths:          size in each dimension (Nx,Ny,Nz)
221         BCs:              A, B, and C for each boundary, there are 8 of these
222         D,Sigma,Q:        Each is an array of size (I,J,K) containing the quantity
223         phi_old:          The cell averaged scalar flux values, dont retain shapes between solves
224     Outputs:
225         x,y,z:            Vectors containing the cell centers in each dimension
226         phi:              A vector containing the solution
227     """
228     I = Dims[0]
229     J = Dims[1]
230     K = Dims[2]
231     L = I*J*K
232     Nx = Lengths[0]
233     Ny = Lengths[1]
234     Nz = Lengths[2]
235
236     hx,hy,hz = np.array(Lengths)/np.array(Dims)
237     ihx2,ihy2,ihz2 = (1.0/hx**2,1.0/hy**2,1.0/hz**2)
238
239     if (type(phi_solution) != np.ndarray):
240         phi_solution = np.zeros((2,I,J,5))
241     phi_new = phi_solution.copy()
242     iteration = 1
243     converged = 0
244     localBCs = np.ones((2,3))
245
246     #reshape Q if necessary
247     if Q.shape != (I,J,K,5):
248         Q_new = np.zeros((I,J,K,5))
249         Q_new[:,:,:,0] = Q[:,:,:]
250         Q = Q_new
251
252     #iterate over the x directions
253     k=0
254     while not(converged):
255
256         #Solve for x direction
257         d = 0 #solv direction
258         tr_id = 1 #trans direction idx in array
259         for j in range(J): #spatial loop over J coordinates
260             for i in range(I): #spatial loop over X coordinates
261
262                 if not(i==0):
263                     phi_left = phi_solution[d,i-1,j,:]
264                     C = positive_current(phi_left,hx/2,hx,D[i-1,j,k])
265                     #print("i =",i,"Cr =",C)
266                     localBCs[0,0:3] = [0.25,-D[i,j,k]/2,C]
267                 else:
268                     localBCs[0,:] = BCs[0,:].copy()
```

```
                    localBCs[0,1] *= D[i,j,k]
                if not(i==(I-1)):
                    phi_rt = phi_solution[d,i+1,j,:]
                    C = negative_current(phi_rt,-hx/2,hx,D[i+1,j,k])
                    #print("i =",i," Cr =",C)
                    localBCs[1,0:3] = [.25,D[i,j,k]/2,C]
                else:
                    localBCs[1,:] = BCs[1,:].copy()
                    localBCs[1,1] *= D[i,j,k]

                #Compute transverse fluxes
                if i==0:
                    nbr_ids = [i,i,i+1] #Assume constant along left edge
                elif i==(I-1):
                    nbr_ids = [i-1,i,i] #assume constant along right edge
                else:
                    nbr_ids = [i-1,i,i+1] #interior cell

                if not j==(J-1):
                    top_phis = phi_solution[tr_id,nbr_ids,j,:]
                    top_Ds   = D[nbr_ids,j,k]
                    Ltop_quad = transverse_leakage_dof(top_phis,hy/2.,hy,hx,top_Ds)
                else:
                    top_phis = phi_solution[tr_id,nbr_ids,j,:]
                    top_Ds   = D[nbr_ids,j,k]
                    Ltop_quad = transverse_leakage_dof(top_phis,hy/2.,hy,hx,top_Ds)
                    #Ltop_quad = (0., 0, 0)

                if not j==0:
                    bot_phis = phi_solution[tr_id,nbr_ids,j,:]
                    bot_Ds   = D[nbr_ids,j,k]
                    Lbot_quad = transverse_leakage_dof(bot_phis,-hy/2.,hy,hx,bot_Ds)
                else:
                    bot_phis = phi_solution[tr_id,nbr_ids,j,:]
                    bot_Ds   = D[nbr_ids,j,k]
                    Lbot_quad = transverse_leakage_dof(bot_phis,-hy/2.,hy,hx,bot_Ds)
                    #Lbot_quad = (0.,0,0)

                #Add leakages to the Q_local terms
#                 print("\n X Information for element: ",i,j)
#                 print("\nThe source is: ",Q[i,j,k,0])

                Q_local = np.array(Q[i,j,k,:])
                for dof in range(len(Ltop_quad)):
                    Q_local[dof] -= 1/hy*(Ltop_quad[dof] - Lbot_quad[dof])

#                 print("The transverse leakage magnitude is: ",-1./hy*(Ltop_quad[0] - Lbot_quad[0]))
#                 print("Total RHS: ", Q_local[0], Q_local[1])

                #Compute the new x fluxes
                phi_new[0,i,j,:] = single_node1GVacuum(D[i,j,k],Sigma[i,j,k],Q_local,hx,localBCs)
                phi,a1,a2,a3,a4 = phi_new[0,i,j,:]
#                 print("The reaction magnitude: ", phi_new[0,i,j,0]*Sigma[i,j,k])
#                 print("The current magnitude: ",1./hx*(current(phi_new[0,i,j,:],hx/2,hx,D[i,j,k]) - curre
#                 print("")

                #print(i,"incoming current on left =", localBCs[0,2],positive_current(phi_new[i,:],-h/2,h,I
                if 0*(i>0):
                    print(i,"outgoing current on left =", negative_current(phi_new[0,i-1,j,:],hx/2,hx,D[i-
                          negative_current(phi_new[0,i,j,:],-hx/2,hx,D[i,j,k]) )
                if 0*(i<I-1):
                    print(i,"outgoing current on right =", positive_current(phi_new[0,i+1,j,:],-hx/2,hx,D[
                          positive_current(phi_new[0,i,j,:],hx/2,hx,D[i,j,k]) )
                #print(i,"incoming current on right =", localBCs[1,2],negative_current(phi_new[i,:],h/2,h,I
                #print("zone ",i," current in at right:",localBCs[1,2]," current out at right:",current_le


        #Solve for y direction
```

```
337         d = 1 #solv direction
338         tr_id = 0 #trans direction idx in array
339         for j in range(J): #spatial loop over J coordinates
340             for i in range(I): #spatial loop over X coordinates
341 
342                 if not(j==0):
343                     phi_left = phi_solution[d,i,j-1,:]
344                     C = positive_current(phi_left,hy/2,hy,D[i,j-1,k])
345                     #print("i =",i,"Cr =",C)
346                     localBCs[0,0:3] = [0.25,-D[i,j,k]/2,C]
347                 else:
348                     localBCs[0,:] = BCs[2,:].copy()
349                     localBCs[0,1] *= D[i,j,k]
350                 if not(j==(J-1)):
351                     phi_rt = phi_solution[d,i,j+1,:]
352                     C = negative_current(phi_rt,-hy/2,hy,D[i,j+1,k])
353                     #print("i =",i,"Cr =",C)
354                     localBCs[1,0:3] = [.25,D[i,j,k]/2,C]
355                 else:
356                     localBCs[1,:] = BCs[3,:].copy()
357                     localBCs[1,1] *= D[i,j,k]
358 
359                 #Compute transverse fluxes
360                 if j==0:
361                     nbr_ids = [j,j,j+1] #Assume constant along left edge
362                 elif j==(J-1):
363                     nbr_ids = [j-1,j,j] #assume constant along right edge
364                 else:
365                     nbr_ids = [j-1,j,j+1] #interior cell
366 
367                 if not i==(I-1):
368                     rgt_phis = phi_solution[tr_id,i,nbr_ids,:]
369                     rgt_Ds   = D[i,nbr_ids,k]
370                     Lrgt_quad = transverse_leakage_dof(rgt_phis,hx/2.,hx,hy,rgt_Ds)
371 #                    print("Leakage right",Lrgt_quad)
372 #                    print("Just the right leakage",current(phi_solution[0,i,j,:],hx/2.,hx,D[i,j,k]))
373 #                    print("Right outflow, inflow",positive_current(phi_solution[0,i,j,:],hx/2,hx,D[i,j,k])
374 #                        negative_current(phi_solution[0,i,j,:],hx/2,hx,D[i,j,k]))
375                 else:
376                     rgt_phis = phi_solution[tr_id,i,nbr_ids,:]
377                     rgt_Ds   = D[i,nbr_ids,k]
378                     Lrgt_quad = transverse_leakage_dof(rgt_phis,hx/2.,hx,hy,rgt_Ds)
379 #                    print("Leakage right",Lrgt_quad)
380 #                    print("Just the right leakage",current(phi_solution[0,i,j,:],hx/2.,hx,D[i,j,k]))
381 #                    print("Right outflow, inflow",positive_current(phi_solution[0,i,j,:],hx/2,hx,D[i,j,k])
382 #                        negative_current(phi_solution[0,i,j,:],hx/2,hx,D[i,j,k]))
383 
384                 if not i==0:
385                     lft_phis = phi_solution[tr_id,i,nbr_ids,:]
386                     lft_Ds   = D[i,nbr_ids,k]
387                     Llft_quad = transverse_leakage_dof(lft_phis,-hx/2.,hx,hy,lft_Ds)
388                 else:
389                     lft_phis = phi_solution[tr_id,i,nbr_ids,:]
390                     lft_Ds   = D[i,nbr_ids,k]
391                     Llft_quad = transverse_leakage_dof(lft_phis,-hx/2.,hx,hy,lft_Ds)
392                     #Llft_quad = (0.,0,0)
393 
394                 #Add leakages to the Q_local terms
395                 Q_local = np.array(Q[i,j,k,:])
396 #                print("\n Y Information for element: ",i,j)
397 #                print("\nThe source is: ",Q[i,j,k,0])
398                 for dof in range(len(Lrgt_quad)):
399                     Q_local[dof] -= 1/hx*(Lrgt_quad[dof] - Llft_quad[dof])
400 #                print("The transverse leakage magnitude is: ",-1./hx*(Lrgt_quad[0] - Llft_quad[0]))
401 #                print("Total RHS: ", Q_local[0], Q_local[1])
402 
403                 phi_new[1,i,j,:] = single_node1GVacuum(D[i,j,k],Sigma[i,j,k],Q_local,hy,localBCs)
404 #                print("The reaction magnitude: ", phi_new[1,i,j,0]*Sigma[i,j,k])
```

```
405 #                 print("The current magnitude: ",1./hy*(current(phi_new[1,i,j,:],hy/2,hy,D[i,j,k]) - curre
406 #                 print("")
407                 phi,a1,a2,a3,a4 = phi_new[1,i,j,:]
408                 #print(i,"incoming current on left =", localBCs[0,2],positive_current(phi_new[i,:],-h/2,h,I
409                 if 0*(i>0):
410                     print(i,"outgoing current on left =", negative_current(phi_new[i-1,:],h/2,h,D[i]),nega
411                 if 0*(i<I-1):
412                     print(i,"outgoing current on right =", positive_current(phi_new[i+1,:],-h/2,h,D[i]),po
413                 #print(i,"incoming current on right =", localBCs[1,2],negative_current(phi_new[i,:],h/2,h,I
414                 #print("zone ",i," current in at right:",localBCs[1,2]," current out at right:",current_le
415 
416 #         print("X solution", phi_new[0,:,:,0])
417 #         print("Y solution", phi_new[1,:,:,0])
418 
419         #Compute total change in x and y
420         relchange = np.linalg.norm( np.reshape(phi_new-phi_solution, 5*I*J*K*2))/np.linalg.norm( np.reshape
421         reldiff = np.linalg.norm( np.reshape(phi_new[0,:,:,0] - phi_new[1,:,:,0], I*J*K)/np.linalg.norm( np
422         converged = (relchange < tolerance) or (iteration >= maxits)
423         if (LOUD):
424             print("Iteration",iteration,": relative change total =",relchange,"relative difference X Y",re
425         iteration += 1
426         phi_solution = phi_new.copy()
427 
428 
429     x = np.linspace(hx*.5,Nx-hx*.5,I)
430     y = np.linspace(hy*.5,Ny-hy*.5,J)
431     z = np.linspace(hz*.5,Nz-hz*.5,K)
432     return x,y,z,phi_solution[0,:,:,0].reshape(I,J,1)#+phi_solution[1,:,:,0].reshape(I,J,1)))
433 
434 def diffusion_steady_fixed_source(Dims,Lengths,BCs,D,Sigma,Q, tolerance=1.0e-12, LOUD=False):
435     """Solve a steady state, single group diffusion problem with a fixed source
436     Inputs:
437         Dims:             number of zones (I,J,K)
438         Lengths:          size in each dimension (Nx,Ny,Nz)
439         BCs:              A, B, and C for each boundary, there are 8 of these
440         D,Sigma,Q:        Each is an array of size (I,J,K) containing the quantity
441     Outputs:
442         x,y,z:            Vectors containing the cell centers in each dimension
443         phi:              A vector containing the solution
444     """
445     I = Dims[0]
446     J = Dims[1]
447     K = Dims[2]
448     L = I*J*K
449     Nx = Lengths[0]
450     Ny = Lengths[1]
451     Nz = Lengths[2]
452 
453     hx,hy,hz = np.array(Lengths)/np.array(Dims)
454     ihx2,ihy2,ihz2 = (1.0/hx**2,1.0/hy**2,1.0/hz**2)
455 
456     #allocate the A matrix, and b vector
457     A = sparse.lil_matrix((L,L))
458     b = np.zeros(L)
459 
460     temp_term = 0
461     for k in range(K):
462         for j in range(J):
463             for i in range(I):
464                 temp_term = Sigma[i,j,k]
465                 row = coordLookup_l(i,j,k,I,J)
466                 b[row] = Q[i,j,k]
467                 #do x-term left
468                 if (i>0):
469                     Dhat = 2* D[i,j,k]*D[i-1,j,k] / (D[i,j,k] + D[i-1,j,k])
470                     temp_term += Dhat*ihx2
471                     A[row, coordLookup_l(i-1,j,k,I,J)]  = -Dhat*ihx2
472                 else:
```

```
                    bA,bB,bC = BCs[0,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (i<I-1):
                            temp_term += -1.5*D[i,j,k]*bA/bB/hx
                            b[row] += -D[i,j,k]/bB*bC/hx
                            A[row,  coordLookup_l(i+1,j,k,I,J)]  += 0.5*D[i,j,k]*bA/bB/hx
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hx
                            b[row] += -D[i,j,k]/bB*bC/hx
                    else:
                        temp_term += D[i,j,k]*ihx2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihx2*2.0
                #do x-term right
                if (i < I-1):
                    Dhat = 2* D[i,j,k]*D[i+1,j,k] / (D[i,j,k] + D[i+1,j,k])
                    temp_term += Dhat*ihx2
                    A[row, coordLookup_l(i+1,j,k,I,J)]  += -Dhat*ihx2
                else:
                    bA,bB,bC = BCs[1,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (i>0):
                            temp_term += 1.5*D[i,j,k]*bA/bB/hx
                            b[row] += D[i,j,k]/bB*bC/hx
                            A[row,  coordLookup_l(i-1,j,k,I,J)]  += -0.5*D[i,j,k]*bA/bB/hx
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hx
                            b[row] += -D[i,j,k]/bB*bC/hx

                    else:
                        temp_term += D[i,j,k]*ihx2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihx2*2.0
                #do y-term
                if (j>0):
                    Dhat = 2* D[i,j,k]*D[i,j-1,k] / (D[i,j,k] + D[i,j-1,k])
                    temp_term += Dhat*ihy2
                    A[row, coordLookup_l(i,j-1,k,I,J)]  += -Dhat*ihy2
                else:
                    bA,bB,bC = BCs[2,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (j<J-1):
                            temp_term += -1.5*D[i,j,k]*bA/bB/hy
                            b[row] += -D[i,j,k]/bB*bC/hy
                            A[row,  coordLookup_l(i,j+1,k,I,J)]  += 0.5*D[i,j,k]*bA/bB/hy
                        else:
                            temp_term += -0.5*D[i,j,k]*bA/bB/hy
                            b[row] += -D[i,j,k]/bB*bC/hy
                    else:
                        temp_term += D[i,j,k]*ihy2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihy2*2.0
                if (j < J-1):
                    Dhat = 2* D[i,j,k]*D[i,j+1,k] / (D[i,j,k] + D[i,j+1,k])
                    temp_term += Dhat*ihy2
                    A[row, coordLookup_l(i,j+1,k,I,J)]  += -Dhat*ihy2
                else:
                    bA,bB,bC = BCs[3,:]
                    if (np.abs(bB) > 1.0e-8):
                        if (j>0):
                            temp_term += 1.5*D[i,j,k]*bA/bB/hy
                            b[row] += D[i,j,k]/bB*bC/hy
                            A[row,  coordLookup_l(i,j-1,k,I,J)]  += -0.5*D[i,j,k]*bA/bB/hy
                        else:
                            temp_term += 0.5*D[i,j,k]*bA/bB/hy
                            b[row] += D[i,j,k]/bB*bC/hy

                    else:
                        temp_term += D[i,j,k]*ihy2*2.0
                        b[row] += D[i,j,k]*bC/bA*ihy2*2.0
                #do z-term
```

```
                    if (k>0):
                        Dhat = 2* D[i,j,k]*D[i,j,k-1] / (D[i,j,k] + D[i,j,k-1])
                        temp_term += Dhat*ihz2
                        A[row, coordLookup_l(i,j,k-1,I,J)]  += -Dhat*ihz2
                    else:
                        bA,bB,bC = BCs[4,:]
                        if (np.abs(bB) > 1.0e-8):
                            if (k<K-1):
                                temp_term += -1.5*D[i,j,k]*bA/bB/hz
                                b[row] += -D[i,j,k]/bB*bC/hz
                                A[row,  coordLookup_l(i,j,k+1,I,J)]  += 0.5*D[i,j,k]*bA/bB/hz
                            else:
                                temp_term += -0.5*D[i,j,k]*bA/bB/hz
                                b[row] += -D[i,j,k]/bB*bC/hz
                        else:
                            temp_term += D[i,j,k]*ihz2*2.0
                            b[row] += D[i,j,k]*bC/bA*ihz2*2.0
                    if (k < K-1):
                        Dhat = 2* D[i,j,k]*D[i,j,k+1] / (D[i,j,k] + D[i,j,k+1])
                        temp_term += Dhat*ihz2
                        A[row, coordLookup_l(i,j,k+1,I,J)]  += -Dhat*ihz2
                    else:
                        bA,bB,bC = BCs[5,:]
                        if (np.abs(bB) > 1.0e-8):
                            if (k>0):
                                temp_term += 1.5*D[i,j,k]*bA/bB/hz
                                b[row] += D[i,j,k]/bB*bC/hz
                                A[row,  coordLookup_l(i,j,k-1,I,J)]  += -0.5*D[i,j,k]*bA/bB/hz
                            else:
                                temp_term += 0.5*D[i,j,k]*bA/bB/hz
                                b[row] += D[i,j,k]/bB*bC/hz

                        else:
                            temp_term += D[i,j,k]*ihz2*2.0
                            b[row] += D[i,j,k]*bC/bA*ihz2*2.0
                    A[row,row] += temp_term
    phi,code = splinalg.cg(A,b, tol=tolerance)
    if (LOUD):
        print("The CG solve exited with code",code)
    phi_block = np.zeros((I,J,K))
    for k in range(K):
        for j in range(J):
            for i in range(I):
                phi_block[i,j,k] = phi[coordLookup_l(i,j,k,I,J)]
    x = np.linspace(hx*.5,Nx-hx*.5,I)
    y = np.linspace(hy*.5,Ny-hy*.5,J)
    z = np.linspace(hz*.5,Nz-hz*.5,K)
    if (I*J*K <= 10):
        print(A.toarray())
    return x,y,z,phi_block


import matplotlib.pyplot as plt

def prob1lattice2G(Lengths):

    Nx = Lengths[0]
    Ny = Lengths[1]
    Nz = Lengths[2]

    #Check multiples of 9
    if (Nx % 9 != 0) or (Ny % 9 != 0):
        raise IOError("Must be multiple of 9")

    #read in cross sections from file for each material
    sigag = [[],[]]
    dg = [[],[]]
    nusigfg = [[],[]]
```

```
    sigsgptog = [[],[]]
    with open("p1_data.csv","r") as f:
        lines = f.readlines()
        for line in lines:

            if re.search("\d+,",line):

                d = line.split(",")
                g   = int(d[1]) - 1 #get group idx
                sigag[g].append(float(d[3]))
                dg[g].append(float(d[2]))
                nusigfg[g].append(float(d[4]))
                sigsgptog[g].append(float(d[5]))

    sigsgptog = np.array(sigsgptog)

    #build sigmar for each material
    mats = [ i for i in range(len(sigag[0])) ]
    chi  = [[1.0 for i in range(len(mats))], [0.0 for i in range(len(mats))]]
    sigr = [[],[]]

    #sigr = sigs_g->g' + sig_ag
    for g in [0,1]:
        for i in range(len(sigag[g])): #loop over materials

            if g == 0:
                gprime = 1
            else:
                gprime = 0
            sigr[g].append(sigag[g][i] + sigsgptog[g,i])


    #Make numpy arrays of everything
    sigag = np.array(sigag)
    sigsgptog = np.array(sigsgptog)
    dg = np.array(dg)
    nusigfg = np.array(nusigfg)
    sigr = np.array(sigr)
    chi = np.array(chi)

    ass_w = 21.608
    baf_w = 2.8575
    width = 194.472

    baf_id = 0

    #Lengths
    Lengths = (194.472,194.472,1)

    #Create a cross section for each cell, intializing to zero
    I = Nx
    J = Ny
    K = Nz

    Sigmaag = np.zeros((I,J,K,2))
    Sigmasgg = np.zeros((I,J,K,2,2))
    nuSigmafg = np.zeros((I,J,K,2))
    chig = np.zeros((I,J,K,2))
    D = np.zeros((I,J,K,2))

    #dimensions of each cell
    hz = 1.0/Nz
    hy = width/Ny
    hx = width/Nx

    #Make a list of boxes defining edges
    x_bounds = [0.0]
    y_bounds = [0.0]
```

```
    f = lambda x: x+ass_w
    for i in range(9):
        x_bounds.append(f(x_bounds[-1]))
        y_bounds.append(f(y_bounds[-1]))

    #Make a list of the objects defining the baffle
    baf = []
    baf.append([0.0, 4*ass_w+baf_w, 8*ass_w, 8*ass_w+baf_w])
    baf.append([4*ass_w, 4*ass_w+baf_w, 7*ass_w+baf_w, 8*ass_w])
    baf.append([4*ass_w, 6*ass_w+baf_w, 7*ass_w, 7*ass_w+baf_w])
    baf.append([6*ass_w, 6*ass_w+baf_w, 6*ass_w+baf_w, 7*ass_w])
    baf.append([6*ass_w, 7*ass_w+baf_w, 6*ass_w, 6*ass_w+baf_w])
    baf.append([7*ass_w, 7*ass_w+baf_w, 4*ass_w+baf_w, 6*ass_w])
    baf.append([7*ass_w, 8*ass_w+baf_w, 4*ass_w, 4*ass_w+baf_w])
    baf.append([8*ass_w, 8*ass_w+baf_w, 0, 4*ass_w])

    #print("Forcing no baffle")
    #baf = []

    #Make a grid of the reactor configuration, index from 1 then subtract 1 everywhere
    assemblies = [[] for j in range(9)]
    assemblies[0] = [2, 3, 2, 3, 2, 3, 2, 4, 5]
    assemblies[1] = [3, 2, 3, 2, 3, 2, 4, 4, 5]
    assemblies[2] = [2, 3, 2, 3, 2, 3, 2, 4, 5]
    assemblies[3] = [3, 2, 3, 2, 3, 2, 4, 4, 5]
    assemblies[4] = [2, 3, 2, 3, 3, 3, 4, 5, 5]
    assemblies[5] = [3, 2, 3, 2, 3, 4, 4, 5, 5]
    assemblies[6] = [2, 4, 2, 4, 4, 4, 5, 5, 5]
    assemblies[7] = [4, 4, 4, 4, 5, 5, 5, 5, 5]
    assemblies[8] = [5, 5, 5, 5, 5, 5, 5, 5, 5]
    assemblies = [[i-1 for i in j] for j in assemblies]

    #Make a material guy
    mat_plot = np.zeros((I,J))

    for k in range(K): #loop over z cells
        for j in range(J): #loop over y assemblies
            for i in range(I): #loop over x assemblies

                x = (i+0.5)*hx
                y = (j+0.5)*hy
                z = (k+0.5)*hz

                #Find the cell you belong to
                for xi in range(len(x_bounds)-1):
                    for yi in range(len(y_bounds)-1):
                        if x > x_bounds[xi] and x < x_bounds[xi+1]:
                            if y > y_bounds[yi] and y < y_bounds[yi+1]:
                                id = assemblies[yi][xi] #this is right
                                break

                #Check not in a baffle
                for b in baf:
                    if x > b[0] and x < b[1]:
                        if y > b[2] and y < b[3]:
                            id = baf_id

                Sigmaag[i,j,k,(0,1)] = sigr[:,id]
                nuSigmafg[i,j,k,(0,1)] = nusigfg[:,id]
                chig[i,j,k,(0,1)] = chi[:,id]
                D[i,j,k,(0,1)] = dg[:,id]

                Sigmasgg[i,j,k,0,1] = sigsgptog[0,id] #Scattering from group 0 to group 1
                Sigmasgg[i,j,k,1,0] = sigsgptog[1,id] #Scattering from group 1 to group 0

                mat_plot[i,j] = id
```

```
    return    Sigmaag,Sigmasgg,nuSigmafg,chig,D,Lengths,  mat_plot


def steady_multigroup_diffusion(G,Dims,Lengths,BCGs,
                                Sigmatg,Sigmasgg,nuSigmafg,
                                nug,chig,D,Q,
                                lintol=1.0e-8,grouptol=1.0e-6,maxits = 12,
                                LOUD=False):
    I = Dims[0]
    J = Dims[1]
    K = Dims[2]
    iteration = 1
    converged = False
    phig = np.zeros((I,J,K,G))
    while not(converged):
        phiold = phig.copy()
        for g in range(G):
            #compute Qhat and Sigmar
            Qhat = Q[:,:,:,g].copy()
            Sigmar = Sigmatg[:,:,:,g] - Sigmasgg[:,:,:,g,g] - chig[:,:,:,g]*nuSigmafg[:,:,:,g]
            for gprime in range(0,G):
                if (g != gprime):
                    Qhat += (chig[:,:,:,g]*nuSigmafg[:,:,:,gprime] + Sigmasgg[:,:,:,gprime,g])*phig[:,:,:,
            x,y,z,phi0 = diffusion_steady_fixed_source(Dims,Lengths,BCGs[:,:,g],D[:,:,:,g],
                                                       Sigmar,Qhat, lintol)
            phig[:,:,:,g] = phi0.copy()
        change = np.linalg.norm(np.reshape(phig - phiold,I*J*K*G)/(I*J*K*G))
        if LOUD:
            print("Iteration",iteration,"Change =",change)
        iteration += 1
        converged = (change < grouptol) or iteration > maxits
    return x,y,z,phig


def inner_iteration(G,Dims,Lengths,BCGs,Sigmar,Sigmasgg,
                    D,Q,lintol=1.0e-8,grouptol=1.0e-9,maxits = 400,LOUD=True):
    I = Dims[0]
    J = Dims[1]
    K = Dims[2]
    iteration = 1
    converged = False
    phig = np.zeros((I,J,K,G))
    while not(converged):
        phiold = phig.copy()
        for g in range(G):
            #compute Qhat
            Qhat = Q[:,:,:,g].copy()
            for gprime in range(0,G):
                if (g != gprime):
                    Qhat +=   Sigmasgg[:,:,:,gprime,g]*phig[:,:,:,gprime]
            x,y,z,phi0 = diffusion_steady_fixed_source(Dims,Lengths,BCGs[:,:,g],D[:,:,:,g],
                                                       Sigmar[:,:,:,g],Qhat, lintol)
            phig[:,:,:,g] = phi0.copy()
        change = np.linalg.norm(np.reshape(phig - phiold,I*J*K*G)/(I*J*K*G))
        if LOUD:
            print("Iteration",iteration,"Change =",change)
        iteration += 1
        converged = (change < grouptol) or iteration > maxits
    return x,y,z,phig


def inner_iteration_nodal(G,Dims,Lengths,BCGs,Sigmar,Sigmasgg,
                    D,Q,lintol=1.0e-8,grouptol=1.0e-6,maxits = 2,LOUD=False):
    I = Dims[0]
    J = Dims[1]
    K = Dims[2]
    iteration = 1
```

```
    converged = False
    phig = np.zeros((I,J,K,G))
    while not(converged):
        phiold = phig.copy()
        for g in range(G):
            #compute Qhat
            Qhat = Q[:,:,:,g].copy()
            for gprime in range(0,G):
                if (g != gprime):
                    Qhat +=    Sigmasgg[:,:,:,gprime,g]*phig[:,:,:,gprime]
            x,y,z,phi0 = nodal2D_steady_fixed_source(Dims,Lengths,BCGs[:,:,g],D[:,:,:,g],
                                                     Sigmar[:,:,:,g],Qhat, tolerance=1.E-04,LOUD=False)
            phig[:,:,:,g] = phi0.copy()
        change = np.linalg.norm(np.reshape(phig - phiold,I*J*K*G)/(I*J*K*G))
        if LOUD:
            print("Iteration",iteration,"Change =",change)
        iteration += 1
        converged = (change < grouptol) or iteration > maxits
    return x,y,z,phig


def kproblem_mg_diffusion(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
                          chig,D,lintol=1.0e-11,grouptol=1.0e-10,tol=1.0e-8,maxits = 12, k = 1, LOUD=False
    I = Dims[0]
    J = Dims[1]
    K = Dims[2]
    phi0 = np.random.rand(I,J,K,G)
    phi0 = np.ones((I,J,K,G))
    phi0 = phi0 / np.linalg.norm(np.reshape(phi0,I*J*K*G))
    phiold = phi0.copy()
    converged = False
    iteration = 1
    while not(converged):
        Qhat = chig*0
        for g in range(G):
            for gprime in range(G):
                Qhat[:,:,:,g] += chig[:,:,:,g]*nuSigmafg[:,:,:,gprime]*phi0[:,:,:,gprime]
        x,y,z,phi0 = inner_iteration(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,D,Qhat,grouptol=grouptol,LOUD=Fa
        knew = np.linalg.norm(np.reshape(phi0,I*J*K*G))/np.linalg.norm(np.reshape(phiold,I*J*K*G))
        solnorm = np.linalg.norm(np.reshape(phiold,I*J*K*G))
        converged = (np.abs(knew-k) < tol  # )and
                    #  np.abs(np.linalg.norm((np.reshape(phi0,I*J*K*G)/knew)-np.reshape(phiold,I*J*K*G)))/so
< tol
                    ) or (iteration > maxits)

        k = knew
        phi0 /= k
        phiold = phi0.copy()
        if (LOUD):
            print("=======================================")
            print("Iteration",iteration,": k =",k)
        iteration += 1
    return x,y,z,k,iteration-1,phi0

def buckling_search(solver,G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
                          chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-5,maxits = 12, k = 1, LOUD=False):


    iteration = 0
    converged = False
    H = 60
    my_tol = 6.E-01
    while not(converged):
        Sigmarg_buck = Sigmarg + D*(pi**2/H**2)
        x,y,z,k,iterations,phig = solver(G,Dims,Lengths,BCGs,Sigmarg_buck,Sigmasgg,nuSigmafg,
                                                              chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=my_tol,
                                                               maxits = 100, k = 1, LOUD=True)
        if (iteration > 0):
```

```
            #update via secant method
            slope = (k-kold)/(H-H_old)
        else:
            Hhat = H + 1.
            Sigmarg_buck = Sigmarg + D*(pi**2/Hhat**2)
            x,y,z,kperturb,iterations,phig = solver(G,Dims,Lengths,BCGs,Sigmarg_buck,Sigmasgg,nuSigmafg,
                                                    chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=my_tol,
                                                    maxits = 100, k = 1, LOUD=True)
            slope = (kperturb - k)/(Hhat - H)
            H = Hhat
        H_old = H
        if H < 0:
            raise ValueError("WOOPS")
        kold = k
        print("The slope is,",slope)
        H = (1-k)/slope + H
        converged = (np.abs(k - 1) < tol) or (iteration >= maxits)
        iteration += 1
        if (LOUD):
            print("\n==========================================")
            print("Iteration",iteration,": k =",k, "H =",H)
            print("==========================================")

        #compute a new tolerance
        my_tol = min(my_tol, abs(k-1))

    return x,y,z,H,iteration,phig

def kproblem_mg_nodal(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
                      chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-8,maxits = 12, k = 1, LOUD=False):
    print("NODAL")
    I = Dims[0]
    J = Dims[1]
    K = Dims[2]
    phi0 = np.random.rand(I,J,K,G)
    phi0 = phi0 / np.linalg.norm(np.reshape(phi0,I*J*K*G))
    phiold = phi0.copy()
    converged = False
    iteration = 1
    while not(converged):
        Qhat = chig*0
        for g in range(G):
            for gprime in range(G):
                Qhat[:,:,:,g] += chig[:,:,:,g]*nuSigmafg[:,:,:,gprime]*phi0[:,:,:,gprime]
        x,y,z,phi0 = inner_iteration_nodal(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,D,Qhat,grouptol=grouptol)
        knew = np.linalg.norm(np.reshape(phi0,I*J*K*G))/np.linalg.norm(np.reshape(phiold,I*J*K*G))
        solnorm = np.linalg.norm(np.reshape(phiold,I*J*K*G))
        converged = ((np.abs(knew-k) < tol)
                     or (iteration > maxits))
        k = knew
        phi0 /= k
        phiold = phi0.copy()
        if (LOUD):
            print("==========================================")
            print("Iteration",iteration,": k =",k)
        iteration += 1
    return x,y,z,k,iteration-1,phi0



def main(ptype='diffusion'):

    buck = False
    if buck:

        I  = 9*3  #here I is number of cells in each distinct region of the problem, in x direction
        J  = 9*3   #etc.
        K  = 1
```

```
        G  = 2
        BCGs = np.ones((6,3,G))
        BCGs[0,0,:] = 0
        BCGs[0,2,:] = 0
        BCGs[2,0,:] = 0
        BCGs[2,2,:] = 0
        BCGs[1,:,0] = [0.25,1/2.,0]
        BCGs[1,:,1] = [0.25,1/2.,0]
        BCGs[3,:,0] = [0.25,1/2.,0]
        BCGs[3,:,1] = [0.25,1/2.,0]
        BCGs[(4,5),0,:] = 0.0
        BCGs[(4,5),2,:] = 0.0

        Sigmarg,Sigmasgg,nuSigmafg,chig,D,Lengths, matplot = prob1lattice2G((I,J,K))
        x,y,z,H,iterations,phig = buckling_search(kproblem_mg_diffusion,G,(I,J,K),Lengths,BCGs,Sigmarg,Sigm
                                  chig,D,lintol=1.0e-13,grouptol=1.0e-13,tol=1.0e-5,maxits = 100, k = 1, LO
        exit()

    #Find k to 5 digits
    if ptype == 'diffusion':
        kproblem = kproblem_mg_diffusion
        begin = 2
        end = 9
        LOUD = True
        tol = 1.e-08
        grouptol = 1.0e-12
    elif ptype == 'nodal':
        kproblem = kproblem_mg_nodal
        begin = 2
        end   = 2
        LOUD = True
        tol = 1.e-4
        grouptol = 1.0e-4
    keffs = []
    n_elems = []
    for nc in range(begin,end+1):

        I  = 9*nc  #here I is number of cells in each distinct region of the problem, in x direction
        J  = 9*nc   #etc.
        n_elems.append(I*J)
        K  = 1
        G  = 2
        BCGs = np.ones((6,3,G))
        BCGs[0,0,:] = 0
        BCGs[0,2,:] = 0
        BCGs[2,0,:] = 0
        BCGs[2,2,:] = 0
        BCGs[1,:,0] = [0.25,1/2.,0]
        BCGs[1,:,1] = [0.25,1/2.,0]
        BCGs[3,:,0] = [0.25,1/2.,0]
        BCGs[3,:,1] = [0.25,1/2.,0]
        BCGs[(4,5),0,:] = 0.0
        BCGs[(4,5),2,:] = 0.0

        Sigmarg,Sigmasgg,nuSigmafg,chig,D,Lengths, matplot = prob1lattice2G((I,J,K))

        x,y,z,k,iterations,phig = kproblem(G,(I,J,K),Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
                                  chig,D,lintol=1.0e-13,grouptol=grouptol,tol=tol,maxits = 400, k = 1, LOUD

        assembly_powers(194.472,194.472,9,9,phig,nuSigmafg)
        plt.figure()
        plt.pcolor(x,y,phig[:,:,0,0])
        plt.colorbar()
        plt.savefig("prob1_g1.pdf")
        plt.pcolor(x,y,phig[:,:,0,1])
        plt.colorbar()
        plt.savefig("prob1_g2.pdf")

```

```
            keffs.append(k)
            print("*****************************")
            print(n_elems[-1],"elements, k for mg diffusion",keffs[-1])
            print("*****************************")

            if len(keffs) >1:
                if (abs(keffs[-1] - keffs[-2]) < 1.0E-05):
                    break

        print(r"$N_{\textsf{cells}}$ & $\keff$  &  $\Delta \keff$ (pcm)")
        for i in range(len(n_elems)):
            if i > 0:
                print(n_elems[i], keffs[i], abs(keffs[i]-keffs[i-1])/abs(keffs[i]))
            else:
                print(n_elems[i], keffs[i], "--")

        #for timing return now
        return
        exit()

        #nodal stuff
        x,y,z,k,iterations,phig = kproblem_mg_nodal(G,(I,J,K),Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
                                  chig,D,lintol=1.0e-13,grouptol=1.0e-13,tol=1.0e-4,maxits = 200, k = 1, LOUD="

        print("k =",k,"Number of iterations =",iterations)
        plt.figure()
        plt.plot(x,phig[:,0,0,0], label="group 1")
        plt.pcolor(x,y,phig[:,:,0,0])
        plt.colorbar()
        plt.savefig("prob1_modal.pdf")
        exit()

        plt.figure()
        plt.pcolor(x,y,matplot)
        plt.colorbar()
        plt.savefig("geom.pdf")



#def main(): #test easier problems
#
#    #solve in x direction
#    solve_x = True
#    solve_y = False
#    if solve_x:
#        print("Solving 1D Problem in X direction")
#        I = 4
#        J = 4
#        K = 1
#        Nx = 1
#        Ny = 1
#        Sigma = np.ones((I,J,K))
#        D = Sigma.copy()
#        Q = np.zeros((I,J,K,5))
#        Q[:,:,:,0] = 1.
#        BCs = np.ones((6,3))
#        BCs[:,0] = 0 #Reflective in Y
#        BCs[:,2] = 0
#        BCs[0,:] = [0.25,-1/2,0]
#        BCs[1,:] = [0.25,1/2,0]
#        BCs[2,:] = [0.25,-1/2,0]
#        BCs[3,:] = [0.25,1/2,0]
#
#
#        x,y,z,phi_x = nodal2D_steady_fixed_source((I,J,K),(Nx,Nx,Nx),BCs,D,Sigma,Q,LOUD=True, maxits=500,
#        plt.plot(x,phi_x[:,0],label='x')
#        solution = (np.exp(1-x) + np.exp(x) + 1 - 3*np.exp(1))/(1-3*np.exp(1))
#        plt.plot(x,solution,'o-',label='Analytic')
```

```
#         x,y,z,phi_x_diff = diffusion_steady_fixed_source((I,J,K),(Nx,Nx*1,Nx),BCs,D,Sigma,Q[:,:,:,0],LOUD=
#         phi_x_derp,its = hetero_node1D(I, D[:,0,0], Sigma[:,0,0], Q[:,0,0,:], Nx/I, BCs, maxits = 100)
#         print("My solution",phi_x[:,:,0])
#         print("Diff solution",phi_x_diff[:,0,0])
#         print("Nodal Regular",phi_x_derp[:,0])
#         print("Analytic",solution)
#         print("Error",np.linalg.norm(phi_x[:,0,0].reshape(I) - solution))
#         print("Error Diff",np.linalg.norm(phi_x_diff[:,0,0].reshape(I) - solution))
#
#     if solve_y:
#         print("Solving Problem in Y direction")
#         I = 4
#         J = 4
#         K = 1
#         Nx = 1
#         Ny = 1
#         Sigma = np.ones((I,J,K))
#         D = Sigma.copy()
#         Q = np.zeros((I,J,K,5))
#         Q[:,:,:,0] = 1.
#         BCs = np.ones((6,3))
#         BCs[:,0] = 0
#         BCs[:,2] = 0
#         BCs[2,:] = [0.25,-1/2,0]
#         BCs[3,:] = [0.25,1/2,0]
#
#         x,y,z,phi_y = nodal2D_steady_fixed_source((I,J,K),(Nx,Nx,Nx),BCs,D,Sigma,Q,LOUD=True, maxits=500,
#         solution = (np.exp(1-y) + np.exp(y) + 1 - 3*np.exp(1))/(1-3*np.exp(1))
#         plt.plot(y,solution,'o-',label='Analytic')
#         x,y,z,phi_y_diff = diffusion_steady_fixed_source((I,J,K),(Nx,Nx*1,Nx),BCs,D,Sigma,Q[:,:,:,0],LOUD=
#         phi_y_derp,its = hetero_node1D(J, D[0,:,0], Sigma[0,:,0], Q[0,:,0,:], Ny/J, BCs[(2,3),:], maxits =
#         print("My solution",phi_y[:,:,0])
#         print("Diff solution",phi_y_diff[:,0,0])
#         print("Nodal Regular",phi_y_derp[:,0])
#         print("Analytic",solution)
#         print("Error",np.linalg.norm(phi_y[0,:,0].reshape(J) - solution))
#         print("Error Diff",np.linalg.norm(phi_y_diff[0,:,0].reshape(J) - solution))

# In[ ]:
if __name__ == "__main__":
    import cProfile
    time = False
    ptype = "diffusion"
    if time:
        cProfile.run("main(ptype='nodal')",sort="cumulative")
    else:
        main(ptype=ptype)
```