

# Homework 3

Simon Bolding  
NUEN 629

October 26, 2015

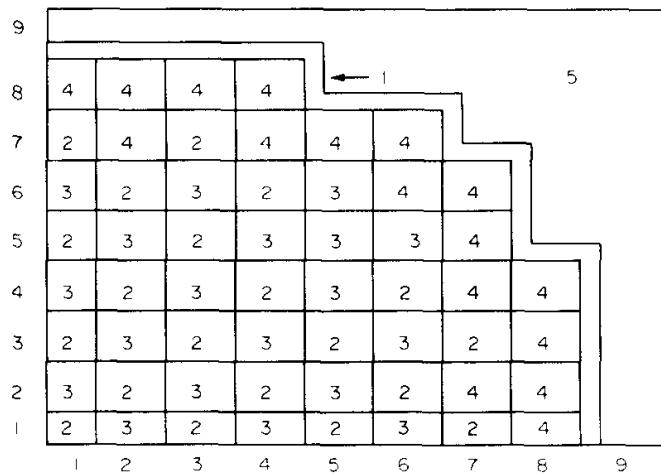
# NUEN 629, Homework 3

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, $g$	$D_g$	$\Sigma_{ag}$	$v\Sigma_{fg}$	$\Sigma_{gg'}$
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 conditions: reflective: left, bottom, zero flux: top, right.

$\Sigma_{tr,g} = 1/3 D_g$  for isotropic scattering, transport problem.

Figure 1: Zion PWR 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 cell's center lies within 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  $\Sigma_f$  individually this is not exact since  $\nu$  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 variability 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

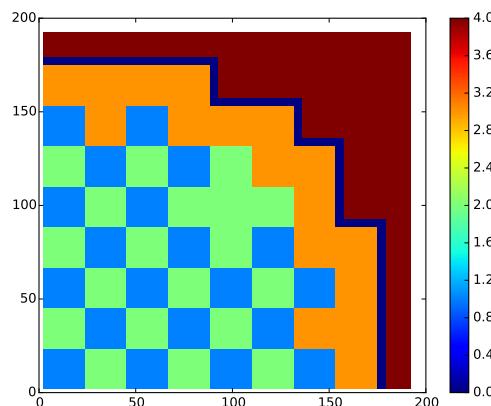


Figure 2: Normalized group 1 fluxes for finite difference solution

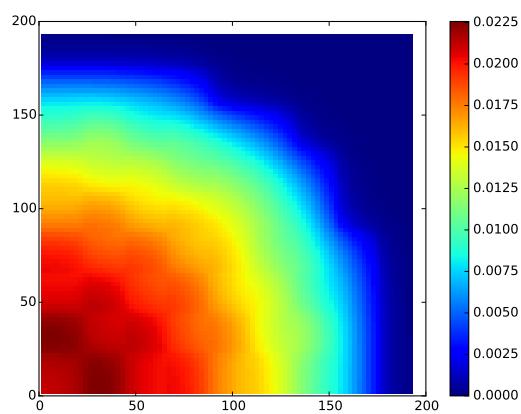


Figure 3: Normalized group 2 fluxes for finite difference solution

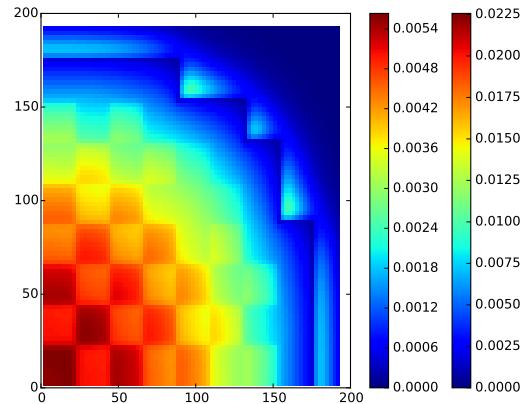
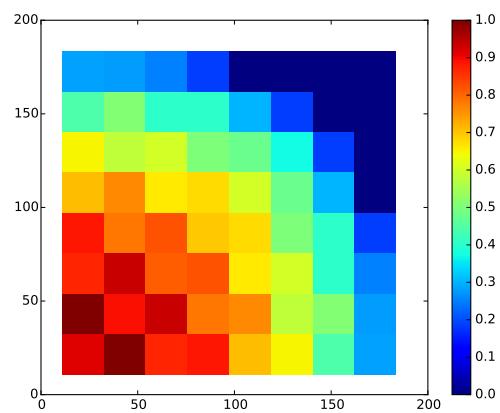


Figure 4: Assembly (assuming 9x9 grid) averaged powers for problem 1, normalized to the max assembly power, solved using finite differences. Note: there is a slight error in the way pcolor is interpolating values.



## Solution 1-2:

The provided  $k$  code and nodal code were modified to solve  $k$  problems using the nodal method. The outer iterations were essentially 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

$$L_y, ij(x) = \frac{1}{h_y} (\bar{J}_{ij}^y(x_i, y_j + h_y/2) - \bar{J}_{ij}^y(x_i, y_j - h_y/2))$$

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  $k_{\text{eff}}$  to 1.E-05. A plot of the group fluxes and assembly powers are given below for the case of 324 cells. For this case  $k_{\text{eff}}$  was found to be 1.273. 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-04 on all iteration loops, this solve took 990 seconds, so this method is very slow without the CMFD acceleration.

Figure 5: Normalized group 1 fluxes for nodal method solution with 324 cells

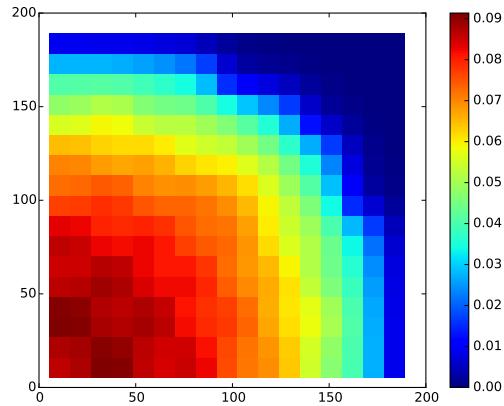


Figure 6: Normalized group 2 fluxes for nodal method with 324 cells

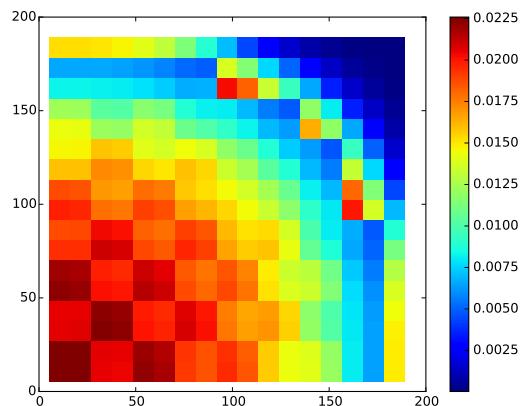
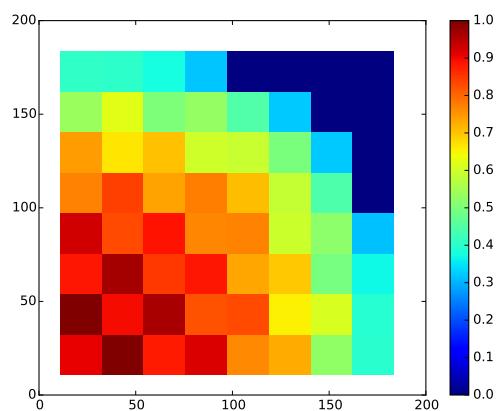


Figure 7: Assembly (assuming 9x9 grid) averaged powers for problem 1, normalized to the max assembly power, solved using nodal method.



**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  $k_{\text{eff}}$  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  $k_{\text{eff}}$  of 0.999996. This is a reasonable number since the 2D reactor had a relatively high  $k_{\text{eff}}$  of  $\sim 1.27$ .

## Code

```

1 def coordLookup_l(i, j, k, I, J):
2     """get the position in a 1-D vector
3     for the (i,j,k) index
4     """
5     return i + j*I + k*I*I
6
7 def coordLookup_ijk(l, I, J):
8     """get the position in a (i,j,k) coordinates
9     for the index l in a 1-D vector
10    """
11    k = (l // (I*j)) + 1
12    j = (l - k*I) // I + 1
13    i = l - (j*I + k*I)-1
14    return i,j,k
15
16 def hetero_node1D(N, D, Sigma_a, Q, h, BCs, maxits = 10, tol = 1.0e-10, LOUD=False, phi_solution=0):
17     if (type(phi_solution) != np.ndarray):
18         phi_solution = np.zeros((N,5))
19     phi_new = phi_solution.copy()
20     iteration = 1
21     converged = 0
22     localBCs = np.zeros((2,3))
23     while not(converged):
24         for i in range(N):
25             if not(i==0):
26                 phi,a1,a2,a3,a4 = phi_solution[i-1,:]
27                 C = positive_current(phi_solution[i-1,:],h/2,h,D[i-1])
28                 #print(" i =",i," Cr =",C)
29                 localBCs[0,0:3] = [0.25,-D[i]/2,C]
30             else:
31                 localBCs[0,:] = BCs[0,:].copy()
32             if not(i==(N-1)):
33                 phi,a1,a2,a3,a4 = phi_solution[i+1,:]
34                 C = negative_current(phi_solution[i+1,:],-h/2,h,D[i+1])
35                 #print(" i =",i," Cr =",C)
36                 localBCs[1,0:3] = [.25,D[i]/2,C]
37             else:
38                 localBCs[1,:] = BCs[1,:].copy()
39             #print(localBCs)
40             phi_new[i,:] = single_node1GVacuum(D[i],Sigma_a[i],Q[i,:],h,localBCs)
41             phi,a1,a2,a3,a4 = phi_new[i,:]
42             #print(i," incoming current on left =", localBCs[0,2],positive_current(phi_new[i,:],-h/2,h,D[i]))
43             if 0*(i>0):
44                 print(i,"outgoing current on left =", negative_current(phi_new[i-1,:],h/2,h,D[i]),negative_
45             if 0*(i<N-1):
46                 print(i,"outgoing current on right =", positive_current(phi_new[i+1,:],-h/2,h,D[i]),positiv_
47             #print(i," incoming current on right =", localBCs[1,2],negative_current(phi_new[i,:],h/2,h,D[i]))
48             current_left = -(D[i]*(a1/h + (3*a2)/h + a3/(2.*h) + a4/(5.*h)))
49             #print("zone ",i," current in at right:",localBCs[1,2]," current out at right:",current_left)
50             relchange = np.linalg.norm( np.reshape(phi_new-phi_solution, 5*N))/np.linalg.norm( np.reshape(phi_n_
51             converged = (relchange < tol) or (iteration >= maxits)
52             if (LOUD):
53                 print("Iteration",iteration,": relative change =",relchange)
54             iteration += 1
55             phi_solution = phi_new.copy()
56     return phi_solution, iteration
57
58 import numpy as np
59 from math import pi
60 import re
61 import scipy as sp
62 import scipy.sparse as sparse
63 import scipy.sparse.linalg as splinalg
64
65 def single_node1GVacuum(D,Sigma_a,Q,h, BCs):

```

```

66 A = np.zeros((5,5))
67 b = np.zeros(5)
68
69 Al = BCs[0,0]
70 Bl = BCs[0,1]
71 Cl = BCs[0,2]
72 Ar = BCs[1,0]
73 Br = BCs[1,1]
74 Cr = BCs[1,2]
75 #equation 1
76 A[0,:] = [Sigma_a, 0, (-6*D)/(h**2), 0, (-2*D)/(5.*h**2))]
77 b[0] = Q[0]
78 #equation 2
79 A[1,:] = [Al, -Al/2. + Bl/h, Al/2. - (3*Bl)/h, Bl/(2.*h), -Bl/(5.*h)]
80 b[1] = Cl
81 #equation 3
82 A[2,:] = [Ar, Ar/2. + Br/h, Ar/2. + (3*Br)/h, Br/(2.*h), Br/(5.*h)]
83 b[2] = Cr
84 #equation 4
85 A[3,:] = [0, (h*Sigma_a)/12., 0, -D/(2.*h) - (h*Sigma_a)/120., 0]
86 b[3] = (h*Q[1])/12. - (h*Q[3])/120.
87 #equation 5
88 A[4,:] = [0, 0, (h*Sigma_a)/20., 0, -D/(5.*h) - (h*Sigma_a)/700.]
89 b[4] = (h*Q[2])/20. - (h*Q[4])/700.
90 phi = np.linalg.solve(A,b)
91 return phi
92
93 # In [459]:
94 def alpha_mg_diffusion(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
95                         chig,D,v,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-9,maxits = 20, alpha = 1000, LOUD=False):
96     iteration = 0
97     converged = False
98     while not(converged):
99         Sigmarg_alpha = Sigmarg + alpha/v
100        x,y,z,k,iterations,phig = kproblem_mg_diffusion(G,(I,J,K),(Nx,Nx,Nx),BCGs,Sigmarg_alpha,Sigmasgg,
101                                              chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-2,
102                                              maxits = 100, k = 1, LOUD=False)
103        if (iteration > 0):
104            #update via secant method
105            slope = (k-kold)/(alpha-alpha_old)
106        else:
107            Sigmarg_alpha = Sigmarg + (alpha+tol)/v
108            x,y,z,kperturb,iterations,phig = kproblem_mg_diffusion(G,(I,J,K),(Nx,Nx,Nx),BCGs,Sigmarg_alpha,
109                                              chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-12,
110                                              maxits = 100, k = 1, LOUD=False)
111            slope = (kperturb - k)/tol
112            alpha_old = alpha
113            kold = k
114            alpha = (1-k)/slope + alpha
115            converged = (np.abs(k - 1) < tol) or (iteration >= maxits)
116            iteration += 1
117            if (LOUD):
118                print("-----")
119                print("Iteration",iteration," : k =",k, "alpha =",alpha)
120    return x,y,z,alpha,iteration,phig
121
122 def assembly_powers(width_x, width_y, nx, ny, phi_g, Sigmafg):
123
124     dx = width_x/nx
125     dy = width_y/ny
126     x_bounds = [0.]
127     y_bounds = [0.]
128     powers = np.zeros((nx,ny))
129     x = y = 0.0
130     for i in range(nx):
131         x += dx
132         y += dy
133         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 nbr 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.,
217 LOUD=False, maxits=100):
218     """Solve a steady state, single group diffusion problem with a fixed source using 2D nodal method
219     Inputs:
220         Dims:           number of zones (I,J,K)
221         Lengths:        size in each dimension (Nx,Ny,Nz)
222         BCs:            A, B, and C for each boundary, there are 8 of these
223         D,Sigma,Q:      Each is an array of size (I,J,K) containing the quantity
224         phi_old:        The cell averaged scalar flux values, dont retain shapes between solves
225     Outputs:
226         x,y,z:          Vectors containing the cell centers in each dimension
227         phi:            A vector containing the solution
228     """
229     I = Dims[0]
230     J = Dims[1]
231     K = Dims[2]
232     L = I*J*K
233     Nx = Lengths[0]
234     Ny = Lengths[1]
235     Nz = Lengths[2]
236
237     hx,hy,hz = np.array(Lengths)/np.array(Dims)
238     ihx2,ihy2,ihz2 = (1.0/hx**2,1.0/hy**2,1.0/hz**2)
239
240     if (type(phi_solution) != np.ndarray):
241         phi_solution = np.zeros((2,I,J,5))
242     phi_new = phi_solution.copy()
243     iteration = 1
244     converged = 0
245     localBCs = np.ones((2,3))
246
247     #reshape Q if necessary
248     if Q.shape != (I,J,K,5):
249         Q_new = np.zeros((I,J,K,5))
250         Q_new[:, :, :, 0] = Q[:, :, :, :]
251         Q = Q_new
252
253     #iterate over the x directions
254     k=0
255     while not (converged):
256
257         #Solve for x direction
258         d = 0 #solv direction
259         tr_id = 1 #trans direction idx in array
260         for j in range(J): #spatial loop over J coordinates
261             for i in range(I): #spatial loop over X coordinates
262
263                 if not(i==0):
264                     phi_left = phi_solution[d,i-1,j,:]
265                     C = positive_current(phi_left,hx/2,hx,D[i-1,j,k])
266                     #print("i =",i,"Cr =",C)
267                     localBCs[0,0:3] = [0.25,-D[i,j,k]/2,C]
268                 else:
269                     localBCs[0,:] = BCs[0,:].copy()

```

```

269     localBCs[0,1] *= D[i,j,k]
270 if not(i==(I-1)):
271     phi_rt = phi_solution[d,i+1,j,:]
272     C = negative_current(phi_rt,-hx/2,hx,D[i+1,j,k])
273     #print("i =",i,"Cr =",C)
274     localBCs[1,0:3] = [.25,D[i,j,k]/2,C]
275 else:
276     localBCs[1,:] = BCs[1,:].copy()
277     localBCs[1,1] *= D[i,j,k]
278
279 #Compute transverse fluxes
280 if i==0:
281     nbr_ids = [i,i,i+1] #Assume constant along left edge
282 elif i==(I-1):
283     nbr_ids = [i-1,i,i] #assume constant along right edge
284 else:
285     nbr_ids = [i-1,i,i+1] #interior cell
286
287 if not j==(J-1):
288     top_phis = phi_solution[tr_id,nbr_ids,j,:]
289     top_Ds   = D[nbr_ids,j,k]
290     Ltop_quad = transverse_leakage_dof(top_phis,hy/2.,hy,hx,top_Ds)
291 else:
292     top_phis = phi_solution[tr_id,nbr_ids,j,:]
293     top_Ds   = D[nbr_ids,j,k]
294     Ltop_quad = transverse_leakage_dof(top_phis,hy/2.,hy,hx,top_Ds)
295     #Ltop_quad = (0., 0, 0)
296
297 if not j==0:
298     bot_phis = phi_solution[tr_id,nbr_ids,j,:]
299     bot_Ds   = D[nbr_ids,j,k]
300     Lbot_quad = transverse_leakage_dof(bot_phis,-hy/2.,hy,hx,bot_Ds)
301 else:
302     bot_phis = phi_solution[tr_id,nbr_ids,j,:]
303     bot_Ds   = D[nbr_ids,j,k]
304     Lbot_quad = transverse_leakage_dof(bot_phis,-hy/2.,hy,hx,bot_Ds)
305     #Lbot_quad = (0.,0,0)
306
307 #Add leakages to the Q_local terms
308 #
309 # print("\n X Information for element: ",i,j)
310 # print("\nThe source is: ",Q[i,j,k,0])
311
312 Q_local = np.array(Q[i,j,k,:])
313 for dof in range(len(Ltop_quad)):
314     Q_local[dof] -= 1/hy*(Ltop_quad[dof] - Lbot_quad[dof])
315
316 # print("The transverse leakage magnitude is: ",-1./hy*(Ltop_quad[0] - Lbot_quad[0]))
317 # print(" Total RHS: ", Q_local[0], Q_local[1])
318
319 #Compute the new x fluxes
320 phi_new[0,i,j,:] = single_node1GVacuum(D[i,j,k],Sigma[i,j,k],Q_local,hx,localBCs)
321 phi,a1,a2,a3,a4 = phi_new[0,i,j,:]
322 # print("The reaction magnitude: ", phi_new[0,i,j,0]*Sigma[i,j,k])
323 # print("The current magnitude: ",1./hx*(current(phi_new[0,i,j,:],hx/2,hx,D[i,j,k]) - current(phi_new[0,i-1,j,:],hx/2,hx,D[i-1,j,k])))
324
325 #print(i," incoming current on left =", localBCs[0,2],positive_current(phi_new[i,:],-h/2,h))
326 if 0*(i>0):
327     print(i,"outgoing current on left =", negative_current(phi_new[0,i-1,j,:],hx/2,hx,D[i-1,j,k]))
328     print(i,"negative current on left =", negative_current(phi_new[0,i,j,:],-hx/2,hx,D[i,j,k]))
329 if 0*(i<I-1):
330     print(i,"outgoing current on right =", positive_current(phi_new[0,i+1,j,:],-hx/2,hx,D[i+1,j,k]))
331     print(i,"positive current on right =", positive_current(phi_new[0,i,j,:],hx/2,hx,D[i,j,k]))
332 #print(i," incoming current on right =", localBCs[1,2],negative_current(phi_new[i,:],h/2,h))
333 #print("zone ",i," current in at right:",localBCs[1,2]," current out at right:",current(phi_new[1,:],h/2,h))
334
335 #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             #
372             #
373             #
374             #
375             print("Leakage right",Lrgt_quad)
376             print("Just the right leakage",current(phi_solution[0,i,j,:],hx/2.,hx,D[i,j,k]))
377             print("Right outflow, inflow",positive_current(phi_solution[0,i,j,:],hx/2,hx,D[i,j,k])
378             #
379             #
380             #
381             #
382             #
383             negative_current(phi_solution[0,i,j,:],hx/2,hx,D[i,j,k]))
384
385         if not i==0:
386             lft_phis = phi_solution[tr_id,i,nbr_ids,:]
387             lft_Ds = D[i,nbr_ids,k]
388             Llft_quad = transverse_leakage_dof(lft_phis,-hx/2.,hx,hy,lft_Ds)
389         else:
390             lft_phis = phi_solution[tr_id,i,nbr_ids,:]
391             lft_Ds = D[i,nbr_ids,k]
392             Llft_quad = transverse_leakage_dof(lft_phis,-hx/2.,hx,hy,lft_Ds)
393             #
394             #
395             #
396             #
397             #
398             #
399             Q_local = np.array(Q[i,j,k,:])
400             print("\n Y Information for element: ",i,j)
401             print("\nThe source is: ",Q[i,j,k,0])
402             for dof in range(len(Lrgt_quad)):
403                 Q_local[dof] -= 1/hx*(Lrgt_quad[dof] - Llft_quad[dof])
404             print("The transverse leakage magnitude is: ",-1./hx*(Lrgt_quad[0] - Llft_quad[0]))
405             print("Total RHS: ", Q_local[0], Q_local[1])
406
407             phi_new[1,i,j,:] = single_node1GVacuum(D[i,j,k],Sigma[i,j,k],Q_local,hy,localBCs)
408             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]) - current(phi_new[0,i,j,:],hy/2,hy,D[i,j,k])) )
406 #
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,D[i]))
409 if 0*(i>0):
410     print(i,"outgoing current on left =", negative_current(phi_new[i-1,:],h/2,h,D[i]),negative_current(phi_new[i,:],h/2,h,D[i]))
411 if 0*(i<I-1):
412     print(i,"outgoing current on right =", positive_current(phi_new[i+1,:],-h/2,h,D[i]),positive_current(phi_new[i,:],h/2,h,D[i]))
413 #print(i,"incoming current on right =", localBCs[1,2],negative_current(phi_new[i,:],h/2,h,D[i]))
414 #print("zone ",i," current in at right:",localBCs[1,2]," current out at right:",current_left(phi_new[i,:],h/2,h,D[i]))
415
416 #
417 print("X solution", phi_new[0,:,:,:,0])
418 print("Y solution", phi_new[1,:,:,:,0])
419
420 #Compute total change in x and y
421 relchange = np.linalg.norm( np.reshape(phi_new-phi_solution , 5*I*K*2))/np.linalg.norm( np.reshape(phi_new-phi_solution , 5*I*K*2))
422 reldiff = np.linalg.norm( np.reshape(phi_new[0,:,:,:,0] - phi_new[1,:,:,:,0] , I*K))/np.linalg.norm( np.reshape(phi_new[0,:,:,:,0] - phi_new[1,:,:,:,0] , I*K))
423 converged = (relchange < tolerance) or (iteration >= maxits)
424 if (LOUD):
425     print("Iteration",iteration ,": relative change total =",relchange , "relative difference X Y",reldiff)
426 iteration += 1
427 phi_solution = phi_new.copy()
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*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:

```

```

473 bA,bB,bC = BCs[0,:]
474 if (np.abs(bB) > 1.0e-8):
475     if (i<I-1):
476         temp_term += -1.5*D[i,j,k]*bA/bB/hx
477         b[row] += -D[i,j,k]/bB*bC/hx
478         A[row, coordLookup_l(i+1,j,k,I,J)] += 0.5*D[i,j,k]*bA/bB/hx
479     else:
480         temp_term += -0.5*D[i,j,k]*bA/bB/hx
481         b[row] += -D[i,j,k]/bB*bC/hx
482     else:
483         temp_term += D[i,j,k]*ihx2*2.0
484         b[row] += D[i,j,k]*bC/bA*ihx2*2.0
485 #do x-term right
486 if (i < I-1):
487     Dhat = 2* D[i,j,k]*D[i+1,j,k] / (D[i,j,k] + D[i+1,j,k])
488     temp_term += Dhat*ihx2
489     A[row, coordLookup_l(i+1,j,k,I,J)] += -Dhat*ihx2
490 else:
491     bA,bB,bC = BCs[1,:]
492     if (np.abs(bB) > 1.0e-8):
493         if (i>0):
494             temp_term += 1.5*D[i,j,k]*bA/bB/hx
495             b[row] += D[i,j,k]/bB*bC/hx
496             A[row, coordLookup_l(i-1,j,k,I,J)] += -0.5*D[i,j,k]*bA/bB/hx
497         else:
498             temp_term += -0.5*D[i,j,k]*bA/bB/hx
499             b[row] += -D[i,j,k]/bB*bC/hx
500
501     else:
502         temp_term += D[i,j,k]*ihx2*2.0
503         b[row] += D[i,j,k]*bC/bA*ihx2*2.0
504 #do y-term
505 if (j>0):
506     Dhat = 2* D[i,j,k]*D[i,j-1,k] / (D[i,j,k] + D[i,j-1,k])
507     temp_term += Dhat*ihy2
508     A[row, coordLookup_l(i,j-1,k,I,J)] += -Dhat*ihy2
509 else:
510     bA,bB,bC = BCs[2,:]
511     if (np.abs(bB) > 1.0e-8):
512         if (j<J-1):
513             temp_term += -1.5*D[i,j,k]*bA/bB/hy
514             b[row] += -D[i,j,k]/bB*bC/hy
515             A[row, coordLookup_l(i,j+1,k,I,J)] += 0.5*D[i,j,k]*bA/bB/hy
516         else:
517             temp_term += -0.5*D[i,j,k]*bA/bB/hy
518             b[row] += -D[i,j,k]/bB*bC/hy
519         else:
520             temp_term += D[i,j,k]*ihy2*2.0
521             b[row] += D[i,j,k]*bC/bA*ihy2*2.0
522     if (j < J-1):
523         Dhat = 2* D[i,j,k]*D[i,j+1,k] / (D[i,j,k] + D[i,j+1,k])
524         temp_term += Dhat*ihy2
525         A[row, coordLookup_l(i,j+1,k,I,J)] += -Dhat*ihy2
526 else:
527     bA,bB,bC = BCs[3,:]
528     if (np.abs(bB) > 1.0e-8):
529         if (j>0):
530             temp_term += 1.5*D[i,j,k]*bA/bB/hy
531             b[row] += D[i,j,k]/bB*bC/hy
532             A[row, coordLookup_l(i,j-1,k,I,J)] += -0.5*D[i,j,k]*bA/bB/hy
533         else:
534             temp_term += 0.5*D[i,j,k]*bA/bB/hy
535             b[row] += D[i,j,k]/bB*bC/hy
536
537     else:
538         temp_term += D[i,j,k]*ihy2*2.0
539         b[row] += D[i,j,k]*bC/bA*ihy2*2.0
540 #do z-term

```

```

541     if (k>0):
542         Dhat = 2* D[i ,j ,k]*D[i ,j ,k-1] / (D[i ,j ,k] + D[i ,j ,k-1])
543         temp_term += Dhat*ihz2
544         A[row, coordLookup_l(i ,j ,k-1,I ,J)] += -Dhat*ihz2
545     else:
546         bA,bB,bC = BCs[4 ,:]
547         if (np.abs(bB) > 1.0e-8):
548             if (k<K-1):
549                 temp_term += -1.5*D[i ,j ,k]*bA/bB/hz
550                 b[row] += -D[i ,j ,k]/bB*bC/hz
551                 A[row, coordLookup_l(i ,j ,k+1,I ,J)] += 0.5*D[i ,j ,k]*bA/bB/hz
552             else:
553                 temp_term += -0.5*D[i ,j ,k]*bA/bB/hz
554                 b[row] += -D[i ,j ,k]/bB*bC/hz
555             else:
556                 temp_term += D[i ,j ,k]*ihz2*2.0
557                 b[row] += D[i ,j ,k]*bC/bA*ihz2*2.0
558         if (k < K-1):
559             Dhat = 2* D[i ,j ,k]*D[i ,j ,k+1] / (D[i ,j ,k] + D[i ,j ,k+1])
560             temp_term += Dhat*ihz2
561             A[row, coordLookup_l(i ,j ,k+1,I ,J)] += -Dhat*ihz2
562         else:
563             bA,bB,bC = BCs[5 ,:]
564             if (np.abs(bB) > 1.0e-8):
565                 if (k>0):
566                     temp_term += 1.5*D[i ,j ,k]*bA/bB/hz
567                     b[row] += D[i ,j ,k]/bB*bC/hz
568                     A[row, coordLookup_l(i ,j ,k-1,I ,J)] += -0.5*D[i ,j ,k]*bA/bB/hz
569                 else:
570                     temp_term += 0.5*D[i ,j ,k]*bA/bB/hz
571                     b[row] += D[i ,j ,k]/bB*bC/hz
572                 else:
573                     temp_term += D[i ,j ,k]*ihz2*2.0
574                     b[row] += D[i ,j ,k]*bC/bA*ihz2*2.0
575             A[row, row] += temp_term
576 phi ,code = sp.linalg.cg(A,b, tol=tolerance)
577 if (LOUD):
578     print("The CG solve exited with code",code)
579 phi_block = np.zeros((I,J,K))
580 for k in range(K):
581     for j in range(J):
582         for i in range(I):
583             phi_block[i,j,k] = phi[coordLookup_l(i ,j ,k,I ,J)]
584 x = np.linspace(hx*.5,Nx-hx*.5,I)
585 y = np.linspace(hy*.5,Ny-hy*.5,J)
586 z = np.linspace(hz*.5,Nz-hz*.5,K)
587 if (I*j*k <= 10):
588     print(A.toarray())
589 return x,y,z,phi_block
590
591
592
593 import matplotlib.pyplot as plt
594
595 def prob1lattice2G(Lengths):
596
597     Nx = Lengths[0]
598     Ny = Lengths[1]
599     Nz = Lengths[2]
600
601 #Check multiples of 9
602 if (Nx % 9 != 0) or (Ny % 9 != 0):
603     raise IOError("Must be multiple of 9")
604
605 #read in cross sections from file for each material
606 sigag = [[],[]]
607 dg = [[],[]]
608 nusigfg = [[],[]]

```

```

609     sigsgptog = [[],[]]
610     with open("pl_data.csv", "r") as f:
611         lines = f.readlines()
612         for line in lines:
613             if re.search("\d+", line):
614                 d = line.split(",")
615                 g = int(d[1]) - 1 #get group idx
616                 sigag[g].append(float(d[3]))
617                 dg[g].append(float(d[2]))
618                 nusigfg[g].append(float(d[4]))
619                 sigsgptog[g].append(float(d[5]))
620
621     sigsgptog = np.array(sigsgptog)
622
623 #build sigmar for each material
624 mats = [ i for i in range(len(sigag[0])) ]
625 chi = [[1.0 for i in range(len(mats))], [0.0 for i in range(len(mats))]]
626 sigr = [[],[]]
627
628 #sigr = sigs_g->g' + sig_ag
629 for g in [0,1]:
630     for i in range(len(sigag[g])): #loop over materials
631
632         if g == 0:
633             gprime = 1
634         else:
635             gprime = 0
636         sigr[g].append(sigag[g][i] + sigsgptog[g,i])
637
638 #Make numpy arrays of everything
639 sigag = np.array(sigag)
640 sigsgptog = np.array(sigsgptog)
641 dg = np.array(dg)
642 nusigfg = np.array(nusigfg)
643 sigr = np.array(sigr)
644 chi = np.array(chi)
645
646 ass_w = 21.608
647 baf_w = 2.8575
648 width = 194.472
649
650 baf_id = 0
651
652 #Lengths
653 Lengths = (194.472,194.472,1)
654
655 #Create a cross section for each cell , intializing to zero
656 I = Nx
657 J = Ny
658 K = Nz
659
660 Sigmaag = np.zeros((I,J,K,2))
661 Sigmasgg = np.zeros((I,J,K,2,2))
662 nuSigmaafg = np.zeros((I,J,K,2))
663 chig = np.zeros((I,J,K,2))
664 D = np.zeros((I,J,K,2))
665
666 #dimensions of each cell
667 hz = 1.0/Nz
668 hy = width/Ny
669 hx = width/Nx
670
671 #Make a list of boxes defining edges
672 x_bounds = [0.0]
673 y_bounds = [0.0]
674
```

```

677     f = lambda x: x+ass_w
678     for i in range(9):
679         x_bounds.append(f(x_bounds[-1]))
680         y_bounds.append(f(y_bounds[-1]))
681
682     #Make a list of the objects defining the baffle
683     baf = []
684     baf.append([0.0, 4*ass_w+baf_w, 8*ass_w, 8*ass_w+baf_w])
685     baf.append([4*ass_w, 4*ass_w+baf_w, 7*ass_w+baf_w, 8*ass_w])
686     baf.append([4*ass_w, 6*ass_w+baf_w, 7*ass_w, 7*ass_w+baf_w])
687     baf.append([6*ass_w, 6*ass_w+baf_w, 6*ass_w+baf_w, 7*ass_w])
688     baf.append([6*ass_w, 7*ass_w+baf_w, 6*ass_w, 6*ass_w+baf_w])
689     baf.append([7*ass_w, 7*ass_w+baf_w, 4*ass_w+baf_w, 6*ass_w])
690     baf.append([7*ass_w, 8*ass_w+baf_w, 4*ass_w, 4*ass_w+baf_w])
691     baf.append([8*ass_w, 8*ass_w+baf_w, 0, 4*ass_w])
692
693     #print("Forcing no baffle")
694     #baf = []
695
696     #Make a grid of the reactor configuration , index from 1 then subtract 1 everywhere
697     assemblies = [[] for j in range(9)]
698     assemblies[0] = [2, 3, 2, 3, 2, 3, 2, 4, 5]
699     assemblies[1] = [3, 2, 3, 2, 3, 2, 4, 4, 5]
700     assemblies[2] = [2, 3, 2, 3, 2, 3, 2, 4, 5]
701     assemblies[3] = [3, 2, 3, 2, 3, 2, 4, 4, 5]
702     assemblies[4] = [2, 3, 2, 3, 3, 3, 4, 5, 5]
703     assemblies[5] = [3, 2, 3, 2, 3, 4, 4, 5, 5]
704     assemblies[6] = [2, 4, 2, 4, 4, 4, 5, 5, 5]
705     assemblies[7] = [4, 4, 4, 4, 5, 5, 5, 5, 5]
706     assemblies[8] = [5, 5, 5, 5, 5, 5, 5, 5, 5]
707     assemblies = [[i-1 for i in j] for j in assemblies]
708
709     #Make a material guy
710     mat_plot = np.zeros((I,J))
711
712     for k in range(K): #loop over z cells
713         for j in range(J): #loop over y assemblies
714             for i in range(I): #loop over x assemblies
715
716                 x = (i+0.5)*hx
717                 y = (j+0.5)*hy
718                 z = (k+0.5)*hz
719
720                 #Find the cell you belong to
721                 for xi in range(len(x_bounds)-1):
722                     for yi in range(len(y_bounds)-1):
723                         if x > x_bounds[xi] and x < x_bounds[xi+1]:
724                             if y > y_bounds[yi] and y < y_bounds[yi+1]:
725                                 id = assemblies[yi][xi] #this is right
726                                 break
727
728                 #Check not in a baffle
729                 for b in baf:
730                     if x > b[0] and x < b[1]:
731                         if y > b[2] and y < b[3]:
732                             id = baf_id
733
734                 Sigmaag[i,j,k,(0,1)] = sigr[:,id]
735                 nuSigmafg[i,j,k,(0,1)] = nusigfg[:,id]
736                 chig[i,j,k,(0,1)] = chi[:,id]
737                 D[i,j,k,(0,1)] = dg[:,id]
738
739                 Sigmasgg[i,j,k,0,1] = sigsgptog[0,id] #Scattering from group 0 to group 1
740                 Sigmasgg[i,j,k,1,0] = sigsgptog[1,id] #Scattering from group 1 to group 0
741
742                 mat_plot[i,j] = id
743
744

```

```

745     return Sigmaag , Sigmasgg , nuSigmafg , chig ,D,Lengths , mat_plot
746
747
748
749 def steady_multigroup_diffusion (G,Dims,Lengths,BCGs,
750                                     Sigmatg , Sigmasgg , nuSigmafg ,
751                                     nug , chig ,D,Q,
752                                     lintol=1.0e-8,grouptol=1.0e-6,maxits = 12 ,
753                                     LOUD=False ):
754     I = Dims[0]
755     J = Dims[1]
756     K = Dims[2]
757     iteration = 1
758     converged = False
759     phig = np.zeros((I,J,K,G))
760     while not (converged):
761         phiold = phig .copy()
762         for g in range(G):
763             #compute Qhat and Sigmar
764             Qhat = Q[:, :, :, g].copy()
765             Sigmar = Sigmatg[:, :, :, g] - Sigmasgg[:, :, :, g, g] - chig[:, :, :, g]*nuSigmafg[:, :, :, g]
766             for gprime in range(0,G):
767                 if (g != gprime):
768                     Qhat += (chig[:, :, :, g]*nuSigmafg[:, :, :, gprime] + Sigmasgg[:, :, :, gprime, g])*phig[:, :, :, gprime]
769             x,y,z,phi0 = diffusion_steady_fixed_source(Dims,Lengths,BCGs[:, :, g],D[:, :, :, g],
770                                               Sigmar ,Qhat , lintol)
771             phig[:, :, :, g] = phi0 .copy()
772             change = np.linalg.norm(np.reshape(phig - phiold ,I*J*K*G)/(I*J*K*G))
773             if LOUD:
774                 print("Iteration",iteration,"Change =",change)
775             iteration += 1
776             converged = (change < grouptol) or iteration > maxits
777     return x,y,z,phig
778
779
780 def inner_iteration (G,Dims,Lengths,BCGs,Sigmar ,Sigmasgg ,
781                      D,Q,lintol=1.0e-8,grouptol=1.0e-9,maxits = 400,LOUD=True ):
782     I = Dims[0]
783     J = Dims[1]
784     K = Dims[2]
785     iteration = 1
786     converged = False
787     phig = np.zeros((I,J,K,G))
788     while not (converged):
789         phiold = phig .copy()
790         for g in range(G):
791             #compute Qhat
792             Qhat = Q[:, :, :, g].copy()
793             for gprime in range(0,G):
794                 if (g != gprime):
795                     Qhat += Sigmasgg[:, :, :, gprime, g]*phig[:, :, :, gprime]
796             x,y,z,phi0 = diffusion_steady_fixed_source(Dims,Lengths,BCGs[:, :, g],D[:, :, :, g],
797                                               Sigmar[:, :, :, g],Qhat , lintol)
798             phig[:, :, :, g] = phi0 .copy()
799             change = np.linalg.norm(np.reshape(phig - phiold ,I*J*K*G)/(I*J*K*G))
800             if LOUD:
801                 print("Iteration",iteration,"Change =",change)
802             iteration += 1
803             converged = (change < grouptol) or iteration > maxits
804     return x,y,z,phig
805
806
807 def inner_iteration_nodal (G,Dims,Lengths,BCGs,Sigmar ,Sigmasgg ,
808                           D,Q,lintol=1.0e-8,grouptol=1.0e-6,maxits = 2,LOUD=False ):
809     I = Dims[0]
810     J = Dims[1]
811     K = Dims[2]
812     iteration = 1

```

```

813     converged = False
814     phig = np.zeros((I,J,K,G))
815     while not(converged):
816         phiold = phig.copy()
817         for g in range(G):
818             #compute Qhat
819             Qhat = Q[:, :, :, g].copy()
820             for gprime in range(0,G):
821                 if (g != gprime):
822                     Qhat += Sigmasgg[:, :, :, gprime]*phig[:, :, :, gprime]
823             x,y,z,phi0 = nodal2D_steady_fixed_source(Dims,Lengths,BCGs[:, :, :, g],D[:, :, :, g],
824                                         Sigmar[:, :, :, g],Qhat, tolerance=1.E-04,LOUD=False)
825             phig[:, :, :, g] = phi0.copy()
826             change = np.linalg.norm(np.reshape(phig - phiold,I*J*K*G)/(I*J*K*G))
827             if LOUD:
828                 print("Iteration",iteration,"Change =",change)
829             iteration += 1
830             converged = (change < grouptol) or iteration > maxits
831     return x,y,z,phig
832
833
834 def kproblem_mg_diffusion(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
835                             chig,D,lintol=1.0e-11,grouptol=1.0e-10,tol=1.0e-8,maxits = 12, k = 1, LOUD=False):
836     I = Dims[0]
837     J = Dims[1]
838     K = Dims[2]
839     phi0 = np.random.rand(I,J,K,G)
840     phi0 = np.ones((I,J,K,G))
841     phi0 = phi0 / np.linalg.norm(np.reshape(phi0,I*J*K*G))
842     phiold = phi0.copy()
843     converged = False
844     iteration = 1
845     while not(converged):
846         Qhat = chig*0
847         for g in range(G):
848             for gprime in range(G):
849                 Qhat[:, :, :, g] += chig[:, :, :, g]*nuSigmafg[:, :, :, gprime]*phi0[:, :, :, gprime]
850             x,y,z,phi0 = inner_iteration(G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,D,Qhat,grouptol=grouptol,LOUD=False)
851             knew = np.linalg.norm(np.reshape(phi0,I*J*K*G))/np.linalg.norm(np.reshape(phiold,I*J*K*G))
852             solnorm = np.linalg.norm(np.reshape(phiold,I*J*K*G))
853             converged = (np.abs(knew-k) < tol # )and
854                         # np.abs(np.linalg.norm((np.reshape(phi0,I*J*K*G)/knew)-np.reshape(phiold,I*J*K*G)))/solnorm
855             < tol
856                         ) or (iteration > maxits)
857
858             k = knew
859             phi0 /= k
860             phiold = phi0.copy()
861             if (LOUD):
862                 print("-----")
863                 print("Iteration",iteration,": k =",k)
864             iteration += 1
865     return x,y,z,k,iteration-1,phi0
866
867 def buckling_search(solver,G,Dims,Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
868                      chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-5,maxits = 12, k = 1, LOUD=False):
869
870     iteration = 0
871     converged = False
872     H = 60
873     my_tol = 6.E-01
874     while not(converged):
875         Sigmarg_buck = Sigmarg + D*(pi**2/H**2)
876         x,y,z,k,iterations,phig = solver(G,Dims,Lengths,BCGs,Sigmarg_buck,Sigmasgg,nuSigmafg,
877                                           chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=my_tol,
878                                           maxits = 100, k = 1, LOUD=True)
879     if (iteration > 0):

```

```

880     #update via secant method
881     slope = (k-kold)/(H-H_old)
882 else:
883     Hhat = H + 1.
884     Sigmarg_buck = Sigmarg + D*(pi**2/Hhat**2)
885     x,y,z,kperturb ,iterations ,phig = solver(G,Dims,Lengths,BCGs,Sigmarg_buck,Sigmasgg,nuSigmafg,
886                                                 chig,D,lintol=1.0e-8,grouptol=1.0e-6,tol=my_tol,
887                                                 maxits = 100, k = 1, LOUD=True)
888     slope = (kperturb - k)/(Hhat - H)
889     H = Hhat
890     H_old = H
891     if H < 0:
892         raise ValueError("WOOPS")
893     kold = k
894     print("The slope is ,",slope)
895     H = (1-k)/slope + H
896     converged = (np.abs(k - 1) < tol) or (iteration >= maxits)
897     iteration += 1
898     if (LOUD):
899         print("\n")
900         print("Iteration",iteration ,": k =",k, "H =",H)
901         print("====")
902
903     #compute a new tolerance
904     my_tol = min(my_tol, abs(k-1))
905
906 return x,y,z,H,iteration ,phig
907
908 def kproblem_mg_nodal(G,Dims,Lengths,BCGs,Sigmarg ,Sigmasgg ,nuSigmafg ,
909                         chig ,D,lintol=1.0e-8,grouptol=1.0e-6,tol=1.0e-8,maxits = 12, k = 1, LOUD=False):
910     print("NODAL")
911     I = Dims[0]
912     J = Dims[1]
913     K = Dims[2]
914     phi0 = np.random.rand(I,J,K,G)
915     phi0 = phi0 / np.linalg.norm(np.reshape(phi0 ,I*J*K*G))
916     phiold = phi0.copy()
917     converged = False
918     iteration = 1
919     while not(converged):
920         Qhat = chig*k0
921         for g in range(G):
922             for gprime in range(G):
923                 Qhat[:, :, :, g] += chig[:, :, :, g]*nuSigmafg[:, :, :, gprime]*phi0[:, :, :, gprime]
924             x,y,z,phi0 = inner_iteration_nodal(G,Dims,Lengths,BCGs,Sigmarg ,Sigmasgg ,D,Qhat ,grouptol=grouptol)
925             knew = np.linalg.norm(np.reshape(phi0 ,I*J*K*G))/np.linalg.norm(np.reshape(phiold ,I*J*K*G))
926             solnorm = np.linalg.norm(np.reshape(phiold ,I*J*K*G))/np.linalg.norm(np.reshape(phi0 ,I*J*K*G))
927             converged = ((np.abs(knew-k) < tol)
928                         or (iteration > maxits))
929             k = knew
930             phi0 /= k
931             phiold = phi0.copy()
932             if (LOUD):
933                 print("====")
934                 print("Iteration",iteration ,": k =",k)
935             iteration += 1
936     return x,y,z,k,iteration -1,phi0
937
938
939
940 def main(ptype='diffusion'):
941
942     buck = False
943     if buck:
944
945         I = 9*3 #here I is number of cells in each distinct region of the problem, in x direction
946         J = 9*3 #etc.
947         K = 1

```

```

948 G = 2
949 BCGs = np.ones((6,3,G))
950 BCGs[0,0,:] = 0
951 BCGs[0,2,:] = 0
952 BCGs[2,0,:] = 0
953 BCGs[2,2,:] = 0
954 BCGs[1,:,:0] = [0.25,1/2.,0]
955 BCGs[1,:,:1] = [0.25,1/2.,0]
956 BCGs[3,:,:0] = [0.25,1/2.,0]
957 BCGs[3,:,:1] = [0.25,1/2.,0]
958 BCGs[(4,5),0,:] = 0.0
959 BCGs[(4,5),2,:] = 0.0
960
961 Sigmarg,Sigmasgg,nuSigmafg,chig,D,Lengths, matplot = prob1lattice2G((I,J,K))
962 x,y,z,H,iterations,phig = buckling_search(kproblem_mg_diffusion,G,(I,J,K),Lengths,BCGs,Sigmarg,Sig
963 chig,D,lintol=1.0e-13,grouptol=1.0e-13,tol=1.0e-5,maxits = 100, k = 1, LOU
964 exit()
965
966 #Find k to 5 digits
967 if ptype == 'diffusion':
968     kproblem = kproblem_mg_diffusion
969     begin = 2
970     end = 9
971     LOUD = True
972     tol = 1.e-08
973     grouptol = 1.0e-12
974 elif ptype == 'nodal':
975     kproblem = kproblem_mg_nodal
976     begin = 2
977     end = 2
978     LOUD = True
979     tol = 1.e-4
980     grouptol = 1.0e-4
981 keffs = []
982 n_elems = []
983 for nc in range(begin,end+1):
984
985     I = 9*nc #here I is number of cells in each distinct region of the problem, in x direction
986     J = 9*nc #etc.
987     n_elems.append(I*J)
988     K = 1
989     G = 2
990     BCGs = np.ones((6,3,G))
991     BCGs[0,0,:] = 0
992     BCGs[0,2,:] = 0
993     BCGs[2,0,:] = 0
994     BCGs[2,2,:] = 0
995     BCGs[1,:,:0] = [0.25,1/2.,0]
996     BCGs[1,:,:1] = [0.25,1/2.,0]
997     BCGs[3,:,:0] = [0.25,1/2.,0]
998     BCGs[3,:,:1] = [0.25,1/2.,0]
999     BCGs[(4,5),0,:] = 0.0
1000    BCGs[(4,5),2,:] = 0.0
1001
1002 Sigmarg,Sigmasgg,nuSigmafg,chig,D,Lengths, matplot = prob1lattice2G((I,J,K))
1003
1004 x,y,z,k,iterations,phig = kproblem(G,(I,J,K),Lengths,BCGs,Sigmarg,Sigmasgg,nuSigmafg,
1005 chig,D,lintol=1.0e-13,grouptol=grouptol,tol=tol,maxits = 400, k = 1, LOU
1006
1007 assembly_powers(194.472,194.472,9,9,phig,nuSigmafg)
1008 plt.figure()
1009 plt.pcolor(x,y,phig[:, :, 0, 0])
1010 plt.colorbar()
1011 plt.savefig("prob1_g1.pdf")
1012 plt.pcolor(x,y,phig[:, :, 0, 1])
1013 plt.colorbar()
1014 plt.savefig("prob1_g2.pdf")
1015

```

```

1016     keffs . append(k)
1017     print("*****")
1018     print(n_elems[-1], "elements , k for mg diffusion",keffs[-1])
1019     print("*****")
1020
1021     if len(keffs) >1:
1022         if (abs(keffs[-1] - keffs[-2]) < 1.0E-05):
1023             break
1024
1025     print(r"\$N_{\text{cells}}\$ & \$\text{keff} \$ & \$\Delta \text{keff} \$ (\text{pcm})")
1026     for i in range(len(n_elems)):
1027         if i > 0:
1028             print(n_elems[i], keffs[i], abs(keffs[i]-keffs[i-1])/abs(keffs[i]))
1029         else:
1030             print(n_elems[i], keffs[i], "--")
1031
1032     #for timing return now
1033     return
1034     exit()
1035
1036     #nodal stuff
1037     x,y,z,k,iterations ,phig = kproblem_mg_nodal(G,(I,J,K),Lengths,BCGs,Sigmarg ,Sigmasgg ,nuSigmafg ,
1038                                         chig ,D,lintol=1.0e-13,grouptol=1.0e-13,tol=1.0e-4,maxits = 200, k = 1, LOUD=
1039
1040     print("k =",k,"Number of iterations =",iterations)
1041     plt . figure()
1042     plt . plot(x,phig[:,0,0,0], label="group 1")
1043     plt . pcolor(x,y,phig[:, :,0,0])
1044     plt . colorbar()
1045     plt . savefig("prob1_modal.pdf")
1046     exit()
1047
1048     plt . figure()
1049     plt . pcolor(x,y,matplotlib)
1050     plt . colorbar()
1051     plt . savefig("geom.pdf")
1052
1053
1054
1055     #def main(): #test easier problems
1056     #
1057     #    #solve in x direction
1058     #    solve_x = True
1059     #    solve_y = False
1060     #    if solve_x:
1061     #        print("Solving 1D Problem in X direction")
1062     #        I = 4
1063     #        J = 4
1064     #        K = 1
1065     #        Nx = 1
1066     #        Ny = 1
1067     #        Sigma = np.ones((I,J,K))
1068     #        D = Sigma.copy()
1069     #        Q = np.zeros((I,J,K,5))
1070     #        Q[:, :, :, 0] = 1.
1071     #        BCs = np.ones((6,3))
1072     #        BCs[:, 0] = 0 #Reflective in Y
1073     #        BCs[:, 2] = 0
1074     #        BCs[0, :] = [0.25,-1/2,0]
1075     #        BCs[1, :] = [0.25,1/2,0]
1076     #        BCs[2, :] = [0.25,-1/2,0]
1077     #        BCs[3, :] = [0.25,1/2,0]
1078     #
1079     #
1080     #        x,y,z,phi_x = nodal2D_steady_fixed_source((I,J,K),(Nx,Nx,Nx),BCs,D,Sigma,Q,LOUD=True , maxits=500,
1081     #        plt . plot(x,phi_x[:,0],label='x')
1082     #        solution = (np.exp(1-x) + np.exp(x) + 1 - 3*np.exp(1))/(1-3*np.exp(1))
1083     #        plt . plot(x,solution,'o-',label='Analytic')
```

```

1084 #           x,y,z,phi_x_diff = diffusion_steady_fixed_source((I,J,K),(Nx,Nx*1,Nx),BCs,D,Sigma,Q[:, :, :, 0],LOUD=True)
1085 #           phi_x_derp,its = hetero_node1D(I, D[:, 0, 0], Sigma[:, 0, 0], Q[:, 0, 0, :], Nx/I, BCs, maxits = 100)
1086 #           print("My solution",phi_x[:, :, 0])
1087 #           print("Diff solution",phi_x_diff[:, 0, 0])
1088 #           print("Nodal Regular",phi_x_derp[:, 0])
1089 #           print("Analytic",solution)
1090 #           print("Error",np.linalg.norm(phi_x[:, 0, 0].reshape(I) - solution))
1091 #           print("Error Diff",np.linalg.norm(phi_x_diff[:, 0, 0].reshape(I) - solution))
1092 #
1093 # if solve_y:
1094 #     print("Solving Problem in Y direction")
1095 #     I = 4
1096 #     J = 4
1097 #     K = 1
1098 #     Nx = 1
1099 #     Ny = 1
1100 #     Sigma = np.ones((I,J,K))
1101 #     D = Sigma.copy()
1102 #     Q = np.zeros((I,J,K,5))
1103 #     Q[:, :, :, 0] = 1.
1104 #     BCs = np.ones((6,3))
1105 #     BCs[:, 0] = 0
1106 #     BCs[:, 2] = 0
1107 #     BCs[2, :] = [0.25,-1/2,0]
1108 #     BCs[3, :] = [0.25,1/2,0]
1109 #
1110 #     x,y,z,phi_y = nodal2D_steady_fixed_source((I,J,K),(Nx,Nx,Nx),BCs,D,Sigma,Q,LOUD=True, maxits=500,
1111 #                                                 solution = (np.exp(1-y) + np.exp(y) + 1 - 3*np.exp(1))/(1-3*np.exp(1)))
1112 #     plt.plot(y,solution,'o-',label='Analytic')
1113 #     x,y,z,phi_y_diff = diffusion_steady_fixed_source((I,J,K),(Nx,Nx*1,Nx),BCs,D,Sigma,Q[:, :, :, 0],LOUD=True)
1114 #     phi_y_derp,its = hetero_node1D(J, D[0, :, 0], Sigma[0, :, 0], Q[0, :, 0, :], Ny/J, BCs[(2,3),:], maxits = 100)
1115 #     print("My solution",phi_y[:, :, 0])
1116 #     print("Diff solution",phi_y_diff[:, 0, 0])
1117 #     print("Nodal Regular",phi_y_derp[:, 0])
1118 #     print("Analytic",solution)
1119 #     print("Error",np.linalg.norm(phi_y[0, :, 0].reshape(J) - solution))
1120 #     print("Error Diff",np.linalg.norm(phi_y_diff[0, :, 0].reshape(J) - solution))
1121 #
1122 # In[ ]:
1123 if __name__ == "__main__":
1124     import cProfile
1125     time = False
1126     ptype = "diffusion"
1127     if time:
1128         cProfile.run("main(ptype='nodal')", sort="cumulative")
1129     else:
1130         main(ptype=ptype)

```