CHAPTER

23

Monte Carlo Eigenvalue Calculations

OUTLINE

"Maybe that's what it is," said Somers. "That's a useful way of putting it. I can't help my aura colliding, can I?"

–*D.H. Lawrence* Kangaroo

CHAPTER POINTS

- We can use Monte Carlo to calculate the k-eigenvalues of a nuclear system.
- One approach is to record where simulated neutrons have a fission event, and use those fission sites as the birthplace for neutrons in the next generation. By properly defining weights, the ratio of neutrons in successive generations is the estimate of k_{eff}.
- Another approach forms a matrix based on a Monte Carlo simulation. The eigenvalues and eigenvectors of this matrix are the k-eigenvalues of the system.

23.1 FISSION CYCLES

As we have discussed, we often want to compute the multiplication factor, i.e., the k-eigenvalue, for a system under consideration. We can use Monte Carlo to do this. One definition of the multiplication factor is the following limit

$$k_{\text{eff}} = \lim_{n\to\infty} \frac{\#\text{ of neutrons in generation } n}{\#\text{ of neutrons in generation } (n-1)},$$

where a generation is a step of the chain reaction, for example the neutron that started the chain reaction is in generation 1 and the fission neutrons created from generation 1 neutrons

DOI: 10.1016/B978-0-12-812253-2.00026-1

are in generation 2, and so on. To complete this calculation we need to track neutrons in generations to compute the ratio of neutrons in successive generations. To do this we use fission cycles: that is we simulate generations of neutrons by tabulating where fissions take place.

The idea of a fission cycle is that we have all of the fission locations from the previous generation of neutrons. We then source neutrons from these locations with total weights that sum to the number of fission sites times ν, the average number of neutrons born per fission event. We then track these neutrons to get the fission sites for the birth of neutrons in the next generation. The ratio between the sum of the weights of neutrons born from fission between these generations is an estimate of the eigenvalue. If we run enough cycles, we can get an estimate of the eigenvalue.

One issue with this approach is the starting of the calculation. We typically will not know where the fission sites are for the first generation. Therefore, we need to guess the initial fission sites. If we guess these sites at random, and then run several cycles it is reasonable to believe that the initial distribution of fission sites will not matter. In other words the fission sites will relax to an equilibrium. The random initial sites are analogous to the initial guess we used for the eigenvector when we used power iteration to solve diffusion eigenvalue problems.

In practice, what we do is start with an initial distribution of neutrons randomly chosen in the system. We then take some number of fission cycles that we do not use to estimate k_{eff}. These initial cycles that we do not use are called inactive cycles. The active cycles are those cycles after the inactive cycles. From the estimate of k_{eff} from each cycle we compute the mean k_{eff} and the standard deviations.

In the following code the value of k_{eff} for each fission cycle is calculated for a homogeneous slab. It returns the estimate of k_{eff} for both in the inactive and active cycles.

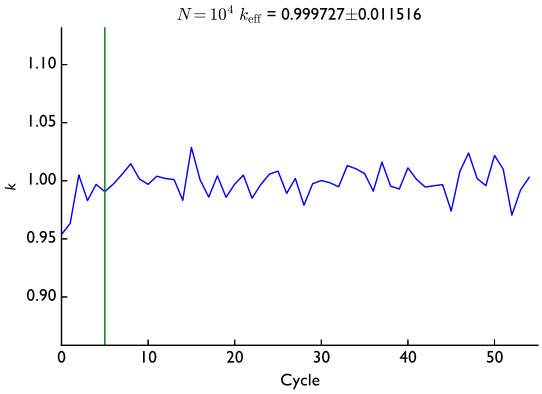
```
In [1]: def homog_slab_k(N,Sig_t,Sig_s,Sig_f,nu, thickness,
                         inactive_cycles = 5, active_cycles = 20):
            Sig_a = Sig_t - Sig_s
            iSig_t = 1/Sig_t
            iSig_a = 1/Sig_a
            #initial fission sites are random
            fission_sites = np.random.uniform(0,thickness,N)
            positions = fission_sites.copy()
            weights = nu*np.ones(N)
            mus = np.random.uniform(-1,1,N)
            old_gen = np.sum(weights)
            k = np.zeros(inactive_cycles+active_cycles)
            for cycle in range(inactive_cycles+active_cycles):
                fission_sites = np.empty(1)
                fission_site_weights = np.empty(1)
                assert(weights.size == positions.size)
                for neut in range(weights.size):
                    #grab neutron from stack
                    position = positions[neut]
                    weight = weights[neut]
                    mu = mus[neut]
                    alive = 1
                    while (alive):
```

```
                #compute distance to collision
                l = -math.log(1-random.random())*iSig_t
                #move neutron
                position += l*mu
                #are we still in the slab
                if (position > thickness) or (position < 0):
                    alive = 0
                else:
                    #decide if collision is abs or scat
                    coll_prob = random.random()
                    if (coll_prob < Sig_s*iSig_t):
                        #scatter
                        mu = random.uniform(-1,1)
                    else:
                        fiss_prob = random.random()
                        alive = 0
                        if (fiss_prob <= Sig_f*iSig_a):
                            #fission
                            fission_sites = np.append(fission_sites,position)
                            fission_site_weights
                            = np.append(fission_site_weights,weight)
        fission_sites =  np.delete(fission_sites,0,axis=0)
        #delete the initial site
        fission_site_weights =  np.delete(fission_site_weights,0,axis=0)
        #delete the initial site
        #sample neutrons for next generation from fission sites
        num_per_site = int(np.ceil(N/fission_sites.size))
        positions = np.empty(1)
        weights = np.empty(1)
        mus = np.random.uniform(-1,1,num_per_site*fission_sites.size)
        for site in range(fission_sites.size):
            site_pos = fission_sites[site]
            site_weight = fission_site_weights[site]
            positions = np.append(positions,
            site_pos*np.ones((num_per_site,1)))
            weights = np.append(weights,
            site_weight * nu/num_per_site*np.ones((num_per_site,1)))
        positions =  np.delete(positions,0,axis=0) #delete the initial site
        weights =  np.delete(weights,0,axis=0) #delete the initial site
        new_gen = np.sum(weights)
        k[cycle] = new_gen/old_gen
        old_gen = new_gen
    return k
```

To demonstrate how this works we will consider a homogeneous slab with single speed neutrons where $\Sigma_t = 1$ cm^{-1}, $\Sigma_s = 0.75$ cm^{-1}, $\Sigma_f = 0.10285$ cm^{-1}, and $\nu = 2.5$. The value of k_∞ for this system is 1.0285. This critical thickness for a slab of this material 11.331 cm [24].

Now we run a k-eigenvalue calculation for a slab of this material that is exactly 11.331 cm thick with 10^4 simulated neutrons per fission cycle and a slab thickness of 20. The expected value of k_{eff} is 1. What we expect to see is that the estimate for k_{eff} will change between iterations and that the early results could be far away from the results in later cycles because of the random initial fission sites.

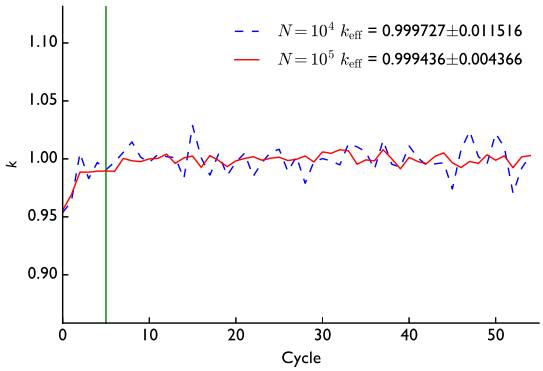
The results for 10^4 simulated neutrons per cycle are given first. In the following figure we show the ratio between the number of neutrons in successive generations as an estimate of k_{eff}. The vertical line serves to indicate where the inactive cycles stop and the active cycles begin.



In the title of this figure we give the mean of k estimates for the active cycles, as well as the standard deviations. The estimate for k_{eff}, 0.999727, is within 27.3 pcm of the correct answer; note that error in the estimate is much smaller than one standard deviation of the k_{eff} estimates over the active cycles.

It is clear to see that early on in the calculation the estimate of k is away from the true value and eventually settles into a range about the mean value. Had we included the inactive cycles in the estimate of k_{eff}, we would have had a much lower of an estimate of the eigenvalue in this case.

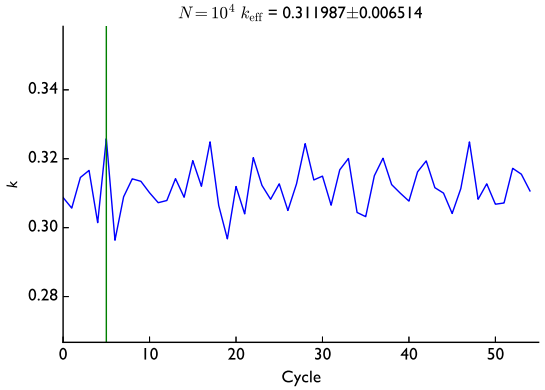
Upon increasing the number of simulated neutrons per cycle by a factor of 10, we expect that the estimate will get better. The next figure compares the fission cycles using 10^4 and 10^5 neutrons per cycle.



In these results we see that the estimate of k_{eff} went down, but the standard deviation of the estimates per fission cycle went down by a factor of $(0.011516/0.004366) \approx 2.64$. This

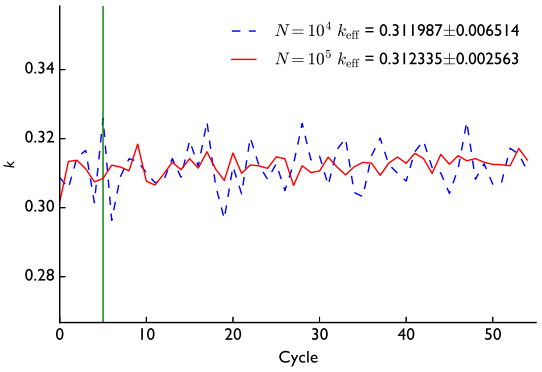
decrease in the standard deviation is close to the expected decrease in the standard deviation by increasing the number of simulated particles by a factor of 10: $\sqrt{10} \approx 3.16$. The decrease in the variation in the estimate between cycles is obvious in the figure.

We can make the slab thinner and re-run the calculation. We would expect the solution to have a smaller value for k_{eff} because more neutrons will leak out of the system. To have a baseline for comparison, we can compare the Monte Carlo solution to another transport calculation based on the discrete ordinates (S_N) method [27] with high resolution. We expect that the Monte Carlo and S_N solution should agree to several digits. The S_N estimate of k_{eff} for this problem with a thickness of 1 cm is 0.312001. The result with 10^4 neutrons per cycle is shown next.



The result is $k_{\text{eff}} = 0.311987$. Given that this is such a leaky system (most of the fission neutrons leak out of the system), we may need more neutrons per fission cycle to suppress the noise in the estimates (as observed in the figure). Also notice that in this system the number of inactive cycles could be decreased because the eigenvalue seems to be near the mean from the first cycle.

With 10^5 neutrons per cycle, the variation in the estimates of k_{eff} decrease:



Once again, the decrease in the standard deviation is about a factor of $\sqrt{10}$; in this case the decrease is about 2.56.

23.2 FISSION MATRIX METHODS

Another way of estimating the eigenvalue of a system would directly estimate the eigenvalues of the transport operator. In particular, we will discretize the system in space and then use Monte Carlo to estimate the number of fission neutrons born in one region that cause fission in another. Upon tallying where neutrons are born and where they cause fission, we will have what is known as a fission matrix. With this matrix we can use standard linear algebra techniques to estimate the eigenvalues of that matrix.

The complete derivation of the fission matrix method is outside the scope of our study, primarily because it involves a firm grasp of transport theory. Here we will give a rough justification of the approach and rely on the numerical results to demonstrate the validity of the approach *faute de mieux*.

We start by considering the integral operator $\mathcal{H}$ defined as

$$\mathcal{H}s(\mathbf{r}, E) = \int dE' \int_{V'} dV' \, F(\mathbf{r}', E' \to \mathbf{r}, E) s(\mathbf{r}', E'),$$

where $F(r', E' \to r, E)$ is the expected number of fission neutrons created at position $\mathbf{r}$ and energy E from a fission neutron born at $\mathbf{r}'$ with energy E'. In this sense, $\mathcal{H}s(\mathbf{r}, E)$ is the rate density of neutrons born from fission at location $\mathbf{r}$ and energy E due to a generic density of neutrons $s(\mathbf{r}, E)$.

Now, if $s(\mathbf{r}, E)$ gives the density of fission neutrons born in a generation, then we can write the k-eigenvalue problem as

$$\mathcal{H}s(\mathbf{r}, E) = ks(\mathbf{r}, E).$$

Therefore, we could solve for k by computing

$$k = \frac{\langle \mathcal{H}s(\mathbf{r}, E) \rangle}{\langle s(\mathbf{r}, E) \rangle}.$$

The angle brackets, $\langle \cdot \rangle$ denote integration over space and energy. This is basically what we did with the fission cycle calculation: we started with a distribution of neutrons for given generation, computed the number of fission neutrons born in the next generation and looked at the ratio.

An alternative approach would be to discretize the $\mathcal{H}$ operator in space by defining a mesh of regions. We then can write the total number of fission neutrons in region i caused by neutrons that are born in region j as

$$\mathbf{H}_{ij} = \frac{\int dE \int dE' \int_{V_i} dV \int_{V_j} dV' \, F(\mathbf{r}', E' \to \mathbf{r}, E) \hat{s}(\mathbf{r}', E')}{\int dE' \int_{V_j} dV' \, \hat{s}(\mathbf{r}', E')}.$$

The quantities $\mathbf{H}_{ij}$ form a matrix. The elements of this matrix can be estimated via Monte Carlo. Typically, one does this by assuming a flat fission source in each region of the problem. This is equivalent to stating that the fission rate density is constant in the region. We can then

solve the eigenvalue problem

$$\mathbf{Hs} = \hat{k}\mathbf{s}.$$

If the regions are small enough so that the error in making the source flat is negligible, then we can interpret $\hat{k}$ as an eigenvalue of the system. This means that the dominant eigenvalue $\hat{k}$ and its associated eigenvector are the fundamental mode for the system. We could also find the other modes in the system this way.

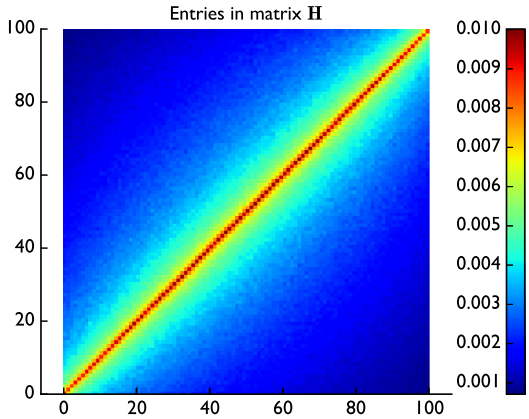
With the knowledge of the fundamental mode eigenvector, we then know the steady-state flux shape in the system. Furthermore, numerical experiments below demonstrate that the magnitudes of the imaginary parts of the eigenvalues are a measure of the uncertainty in the fundamental mode eigenvalue.

To compute the fission matrix we need to specify a mesh over the problem, and then emit neutrons in each region, and count the number of fission neutrons born in each other region. We will demonstrate this in our homogeneous slab problem.

```
In [4]: def fission_matrix(N,Sig_t,Sig_s,Sig_f,nu, thickness,Nx):
            Sig_a = Sig_t - Sig_s
            H = np.zeros((Nx,Nx))
            dx = thickness/Nx
            lowX = np.linspace(0,thickness-dx,Nx)
            highX = np.linspace(dx,thickness,Nx)
            midX = np.linspace(dx*0.5,thickness-dx*0.5,Nx)
            for col in range(Nx):
                #create source neutrons
                positions = np.random.uniform(lowX[col],highX[col],N)
                mus = np.random.uniform(-1,1,N)
                weights = np.ones(N)*(1.0/N)
                #track neutrons
                for neut in range(positions.size):
                    #grab neutron from stack
                    position = positions[neut]
                    mu = mus[neut]
                    weight = weights[neut]
                    alive = 1
                    while (alive):
                        #compute distance to collision
                        l = -np.log(1-np.random.random(1))/Sig_t
                        #move neutron
                        position += l*mu
                        #are we still in the slab
                        if (position > thickness) or (position < 0):
                            alive = 0
                        else:
                            #decide if collision is abs or scat
                            coll_prob = np.random.rand(1)
                            if (coll_prob < Sig_s/Sig_t):
                                #scatter
                                mu = np.random.uniform(-1,1,1)
                            else:
                                fiss_prob = np.random.rand(1)
                                alive = 0
```

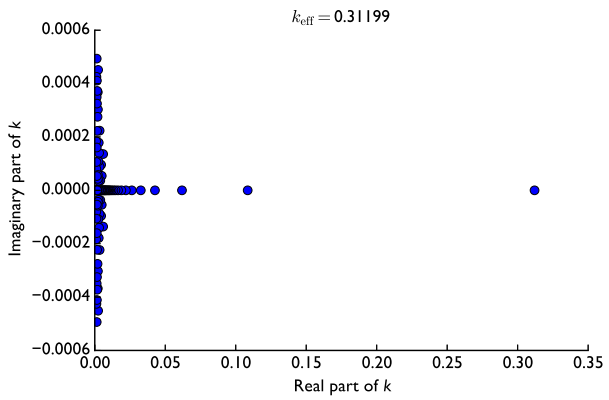
```
                    if (fiss_prob <= Sig_f/Sig_a):
                        #find which bin we are in
                        row = np.argmin(np.abs(position - midX))
                        H[row,col] += weight*nu
    return H, midX
```

With this function we will know how to build the fission matrix. Then we can use the `NumPy` function, `eig` to find all the eigenvalues of the system and take the largest of these as the value of k_{eff}. The other eigenvalues can be used to help analyze the behavior of the system during transients. The thin slab reactor analyzed above with fission cycles will have its fission matrix and eigenvalue computed in next. In the calculation we use 10^6 simulated neutrons. From these results, we plot the entries of the matrix in a color map and the eigenvalues in the complex plane.



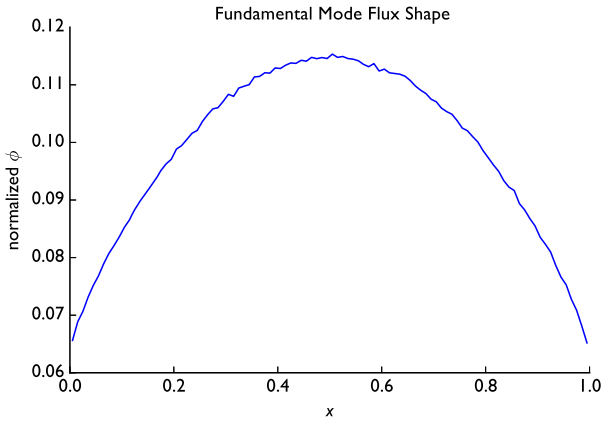
This plot of the matrix elements tells us about the physics of the reactor. The diagonal elements have the largest magnitude, and these represent fission neutrons that are born and cause fission in the same region. Also, the elements farther away from the diagonal are smaller because these represent neutrons that travel far from the birth region before causing fission. Also, there is clear statistical noise in the matrix elements.

Next, we will look at the eigenvalues of the fission matrix:



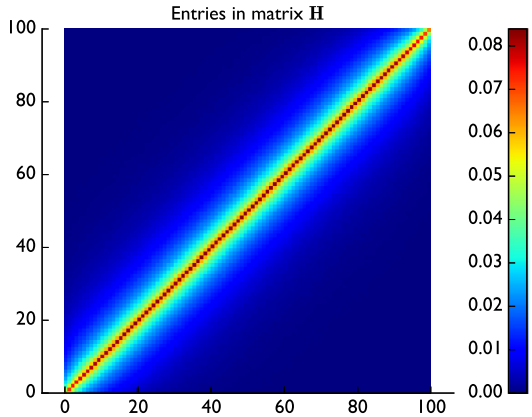
These are the eigenvalues in the complex planes. The fundamental mode eigenvalue is far to the right and is purely real. Its value is 0.31199. Closer to zero there is a cloud of complex eigenvalues. The actual k-eigenvalues of the system should all be real—the imaginary part of these eigenvalues is due to noise in the calculation of the fission matrix. Notice that the eigenvalue from the fission matrix agrees with the S_N transport calculation to five digits.

One benefit of the fission matrix approach is that we can also estimate the shape of the fundamental mode scalar flux from `np.eig`.



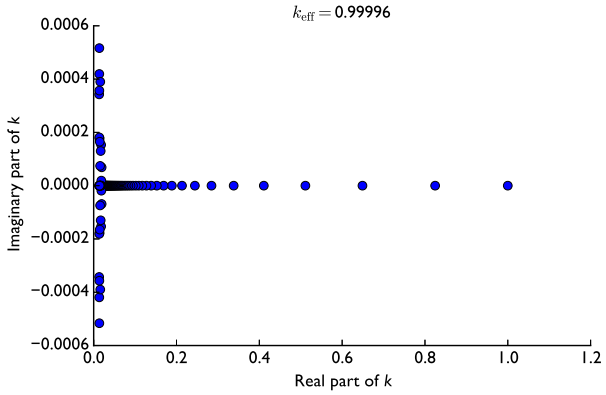
The shape of the fundamental mode scalar flux is the basic shape we would expect: peaked in the middle of the slab and falling off toward the edges of the slab. There is obvious noise present as well.

If we modify the problem to be thicker, i.e., make the slab have a thickness of 11.331, without increasing the number of regions, the noise decreases. This decrease is partially because the thickness of each region is larger, as well as the fact that fewer fission neutrons leak out. Recall that in this problem the exact answer is $k_{\text{eff}} = 1$.

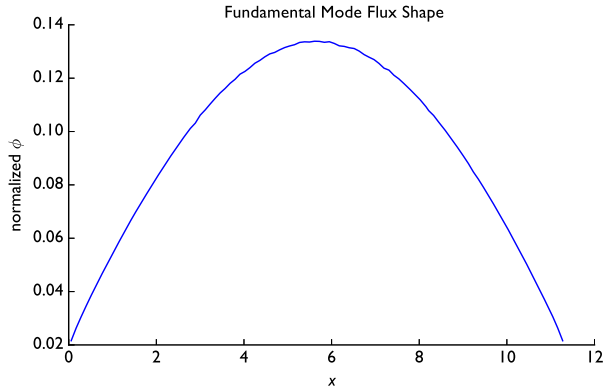


The matrix is more diagonally dominant, and neutrons appear to not move much between regions before causing fission. The k_{eff} estimate we obtain close to 1: within 4 pcm of the

actual answer. If we look at the spectrum of the eigenvalues in the complex plane, we see that the imaginary part of the eigenvalues is still present.



The fission matrix calculated for this system have fewer complex eigenvalues, Also, notice that the separation between the fundamental mode eigenvalue and the second largest eigenvalue is smaller than in the thinner system. This implies that power iteration for this problem should converge more slowly than for the thin system. Indeed, we see artifacts of this in the fission cycle calculation above: for the thicker system there was a clear need for several inactive cycles to settle on the fundamental mode. The thin system had no such slow approach to the fundamental mode.



As we saw with the fission cycles, there is less noise in the scalar flux for this thicker slab with thicker regions.

There are several benefits to the fission matrix calculation. Firstly, it relies on Monte Carlo to move the neutrons and linear algebra software to estimate the eigenvalues, whereas the fission cycle calculation combines Monte Carlo and the eigenvalue estimation in the same calculation. In the fission matrix calculation we can rely on well-tested linear algebra libraries to get the eigenvalues after we use Monte Carlo to move neutrons around. As a result we get more information including all the eigenvalues and eigenvectors. The downside of this approach is that we must approximate the fission source as flat in each region.

CODA

Monte Carlo codes are an important tool in nuclear engineering and in any application where radiation physics are important. In this chapter we applied the Monte Carlo principles from the previous two chapters to solve eigenvalue problems. We neglected any energy dependence and only solved homogeneous problems in the examples. It is straightforward, if not simple, to make these extensions to the codes above. The codes and techniques we have discussed here are only a sample of the features available in production Monte Carlo codes such as `MCNP`, `Geant 4`, and `keno`. Nevertheless, the material we have covered will give the reader a strong foundation to either explore Monte Carlo methods in more detail or to run production codes confidently.

PROBLEMS

Programming Projects

1. Pure Plutonium Reactor

In Chapter 18 we defined a pulsed reactor made of pure plutonium with the following cross-sections from the report *Reactor Physics Constants*, ANL-5800:

Quantity	Value
σ_f [b]	1.85
σ_a [b]	2.11
σ_{tr} [b]	6.8
ν	2.98

For the density of plutonium use 19.74 g/cm^3; you may assume that $\sigma_t \approx \sigma_{tr}$.

- Compute k_{eff} for a slab of thickness 7 cm made from pure plutonium-239 using fission cycles and the fission matrix method.
- Compute k_{eff} for a solid sphere of plutonium-239 with a radius of 7 cm using either method discussed in this chapter. You will have to modify the codes above to handle transport in a sphere.
- Finally, compute k_{eff} for a spherical shell of inside radius of 2 cm and an outer radius of 6 cm. You may consider the hollow part of the shell a void. In this case you will have to modify your code to handle the fact that a collision cannot take place in the hollow part of the sphere.