CHAPTER

19 One-Group k-Eigenvalue Problems

OUTLINE

19.1 Nuclear System Criticality 352
19.2 Inverse Power Method 353
19.3 Function for Inverse Power Iteration 355
19.4 Solving 1-D Diffusion Eigenvalue Problems 356
19.4.1 Heterogeneous Problems 360
Coda 362
Problems 362
Programming Projects 362
1. Reflector Effective Albedo 362
2. Spherical Plutonium Reactor k-Eigenvalue 362
3. Criticality for 1-D Heterogeneous System Using the 1-Group Neutron Diffusion Equation 362

Ni vis humana quot annis
maxima quaeque manu legeret.

Put forth his hand with power, and year by year
Choose out the largest.

–*Virgil,* Georgics

CHAPTER POINTS

- We can adapt our fixed-source diffusion method to find the criticality of a nuclear system.
- To do this we use inverse power iteration, which requires repeated solution of a diffusion system.
- This method gives us the fundamental mode and k_{eff} for a nuclear system.

DOI: 10.1016/B978-0-12-812253-2.00021-2

19.1 NUCLEAR SYSTEM CRITICALITY

In the previous chapter we solved steady-state problems in source-driven, subcritical nuclear systems. A different, and perhaps more common, situation is that we want to know the degree of criticality, often quoted as k_{eff} or reactivity, of a system. In 1-D geometry with a single energy group the k-eigenvalue problem is written as

$$-\nabla \cdot D(r)\nabla\phi(r) + \Sigma_{\text{a}}(r)\phi(r) = \frac{\nu\Sigma_{\text{f}}(r)}{k}\phi(r),$$

with a vacuum, reflecting, or albedo boundary condition at the outer surface

$$\mathcal{A}\phi(r) + \mathcal{B}\frac{d\phi}{dr} = 0 \qquad \text{for } r = R,$$

and a reflecting boundary condition at $r = 0$

$$\frac{d}{dr}\phi(r) = 0 \qquad \text{for } r = 0.$$

Notice that at the outer surface, the boundary condition form is the same as that from the previous chapter, except that $\mathcal{C}$ must be zero.

Note the differences from the equation we solved in the last lecture:

- We added the eigenvalue, $1/k$, to the fission term,
- Eigenvalue problems never have sources ($Q(r) = 0$), and always have vacuum, albedo, or reflecting boundary conditions (i.e., $\mathcal{C} = 0$).

It is also useful to recall what the eigenvalue is doing physically in this equation. If the system is supercritical, then $k > 1$ which depresses the rate at which fission neutrons are produced in order to get a steady solution. On the other hand, if $k < 1$ the system is subcritical and more fission neutrons are needed to make the system not decay to 0 at steady-state. A critical system does not need more or fewer fission neutrons (i.e., $k = 1$).

Upon performing the integration over a cell inside a grid of cells, as we did last time, we get the system

$$-\frac{1}{V_i}\left[D_{i+1/2}S_{i+1/2}\frac{\phi_{i+1}-\phi_i}{\Delta r} - D_{i-1/2}S_{i-1/2}\frac{\phi_i-\phi_{i-1}}{\Delta r}\right] + \Sigma_{\text{a},i}\phi_i = \frac{\nu\Sigma_{\text{f},i}}{k}\phi_i,$$
$$i = 0, \dots, I-1, \tag{19.1}$$

and

$$\left(\frac{\mathcal{A}}{2} - \frac{\mathcal{B}}{\Delta r}\right)\phi_{I-1} + \left(\frac{\mathcal{A}}{2} + \frac{\mathcal{B}}{\Delta r}\right)\phi_I = 0.$$

We can write this as a generalized eigenvalue problem

$$\mathbf{A}\vec{\phi} = \lambda\mathbf{B}\vec{\phi},$$

where, in our case, $\lambda = 1/k$, and the matrices **A** and **B** are defined by the discrete equations above. In this case

$$\mathbf{B} = \begin{pmatrix} \nu\Sigma_{\mathrm{f},1} & & & & \\ 0 & \nu\Sigma_{\mathrm{f},2} & & & \\ 0 & 0 & \ddots & & \\ 0 & 0 & \dots & \nu\Sigma_{\mathrm{f},I} & \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}.$$

The matrix **A** is defined in the previous chapter, without the fission terms.

The fundamental mode for the system is the largest value of k and its corresponding eigenvector. This eigenvalue has a special name: k_{eff}. We will now discuss a method for finding this eigenvalue/eigenvector pair.

19.2 INVERSE POWER METHOD

If we consider the generalized eigenvalue problem,

$$\mathbf{Ax} = \lambda\mathbf{Bx},$$

we can make it look like a standard eigenvalue problem by multiplying both sides by the inverse of **A** and then rearranging a bit to get

$$\mathbf{A}^{-1}\mathbf{Bx} = \frac{1}{\lambda}\mathbf{x}.$$

In other words, the eigenvalue of the generalized eigenvalue problem is the reciprocal of the eigenvalue to the eigenvalue problem

$$\mathbf{Cx} = l\mathbf{x},$$

where

$$\mathbf{C} = \mathbf{A}^{-1}\mathbf{B}, \qquad l = \frac{1}{\lambda}.$$

We can sketch out a simple algorithm, and then we will show that it should give the largest eigenvalue of C. This will be the largest value of k, which is the value of k_{eff} for the system. Here are the steps in the algorithm.

1. Start with a random initial guess for $\mathbf{x}$ of unit norm, called $\mathbf{x}_0$, set $i = 0$.
2. Compute the product $\mathbf{b}_{i+1} = \mathbf{Cx}_i$,
3. Let $l_{i+1} = ||\mathbf{b}_{i+1}||$,
4. Set

$$\mathbf{x}_{i+1} = \frac{\mathbf{b}_{i+1}}{||\mathbf{b}_{i+1}||},$$

and set $i = i + 1$ (this step normalizes $\mathbf{x}_{i+1}$ to be a unit vector),
5. if $|l_{i+1} - l_i| < \epsilon$, then stop. Otherwise, go to step 2.

The final value of l_{i+1} is an approximation to the maximum magnitude eigenvalue of **C**. To decide if the eigenvalue is positive or negative we can look at the first index of $\mathbf{b}_{i+1}$ and $\mathbf{b}_i$. The sign of the eigenvalue is equal to the sign of $b_{1,i+1}/b_{1,i}$. Also, $\mathbf{x}_{i+1}$ is the eigenvector associated with eigenvalue l_{i+1}.

Why might this algorithm work? To show that it is reasonable we will assume we know the N eigenvalues l_n and N eigenvectors $\mathbf{u}_n$ of **C**, an $N \times N$ matrix. Using this information we can write

$$\mathbf{x}_i = \sum_{n=1}^{N} \alpha_{n,i} \mathbf{u}_n, \qquad \mathbf{b}_i = \sum_{n=1}^{N} \beta_{n,i} \mathbf{u}_n,$$

because we can decompose vectors in the range of **C** into a linear combination of eigenvectors, which is what these relations say.

Using these relations we can write

$$\mathbf{b}_{i+1} = \sum_{n=1}^{N} \beta_{n,i+1} \mathbf{u}_n = \mathbf{C}\mathbf{x_i} = \mathbf{C} \sum_{n=1}^{N} \alpha_{n,i} \mathbf{u}_n.$$

Of course, $\mathbf{C}\mathbf{u}_n = l_n \mathbf{u}_n$, so we get

$$\mathbf{b}_{i+1} = \sum_{n=1}^{N} \alpha_{n,i} l_n \mathbf{u}_n.$$

If we assume, that the eigenvalues are numbered in decreasing magnitude, $|l_{n-1}| > |l_n|$, then we notice that $\mathbf{b}_{i+1}$ is larger in the $\mathbf{u}_1$ component than any of the others because $\mathbf{x}_i$ is a unit vector. After many iterations of the algorithm, we can expect that

$$\mathbf{b}_{i+1} \approx \alpha_{1,i} l_1 \mathbf{u}_1, \qquad i \gg 1,$$

because this component grows each iteration and grows faster than the others. Therefore

$$||\mathbf{b}_{i+1}|| \approx l_1,$$

because $\mathbf{x}_i$ is a unit-vector by construction.

Therefore, we have the steps we need to find the largest magnitude eigenvector and its associated eigenvector for the matrix **C**. Recall that to compute $\mathbf{b}_{i+1} = \mathbf{C}\mathbf{x}_i$ we are actually calculating

$$\mathbf{b}_{i+1} = \mathbf{C}\mathbf{x}_i = \mathbf{A}^{-1}\mathbf{B}\mathbf{x}_i.$$

We do not want to explicitly compute the inverse of **A**, therefore we can solve the following problem to compute $\mathbf{b}_{i+1}$:

$$\mathbf{A}\mathbf{b}_{i+1} = \mathbf{B}\mathbf{x}_i.$$

Notice that at each iteration we are solving a problem with the same matrix and a changing right-hand side. This indicates LU factorization is the correct strategy. When we are solving a diffusion k-eigenvalue problem, this means we will have to solve a steady-state diffusion equation with a known source at each power iteration.

Finally, we relate the eigenvalue l to the k-eigenvalue of the original problem. Recall that $l = 1/\lambda = k$ is the largest eigenvalue of C which is the largest value of k, and therefore the fundamental mode of the system. Therefore, using inverse power iteration gives us the fundamental eigenvalue of the system, k_{eff}, and the fundamental mode eigenvector.

19.3 FUNCTION FOR INVERSE POWER ITERATION

We can translate our simple algorithm above and translate it into python.

```
In [1]: def inversePower(A,B,epsilon=1.0e-6,LOUD=False):
            """Solve the generalized eigenvalue problem
            Ax = l B x using inverse power iteration
            Inputs
            A: The LHS matrix (must be invertible)
            B: The RHS matrix
            epsilon: tolerance on eigenvalue
            Outputs:
            l: the smallest eigenvalue of the problem
            x: the associated eigenvector
            """
            N,M = A.shape
            assert(N==M)
            #generate initial guess
            x = np.random.random((N))
            x = x / np.linalg.norm(x) #make norm(x)==1
            l_old = 0
            converged = 0
            #compute LU factorization of A
            row_order = LU_factor(A,LOUD=False)
            iteration = 1;
            while not(converged):
                b = LU_solve(A,np.dot(B,x),row_order)
                l = np.linalg.norm(b)
                sign = b[0]/x[0]/l
                x = b/l
                converged = (np.fabs(l-l_old) < epsilon)
                l_old = l
                if (LOUD):
                    print("Iteration:",iteration,"\tMagnitude of l =",1.0/l)
                iteration += 1
            return sign/l, x
```

To test this method, we can solve a very simple eigenproblem:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0.1 \end{pmatrix} \mathbf{x} = l\mathbf{x}.$$

The smallest eigenvalue is 0.1, and we will solve this using our inverse power iteration method.

```
In [2]: #define A
        A = np.identity(2)
        A[1,1] = 0.1
        #define B
        B = np.identity(2)
        l, x = inversePower(A,B,LOUD=True)

Iteration: 1    Magnitude of l = 0.168384028973
Iteration: 2    Magnitude of l = 0.100930486877
Iteration: 3    Magnitude of l = 0.10000934947
Iteration: 4    Magnitude of l = 0.100000093499
Iteration: 5    Magnitude of l = 0.100000000935
Iteration: 6    Magnitude of l = 0.100000000009
```

That test worked. Now we can use our method to solve for the eigenvalue of a 1-D reactor.

19.4 SOLVING 1-D DIFFUSION EIGENVALUE PROBLEMS

We will now modify our code from the previous lecture to deal with the differences in eigenvalue problems. In particular we need to remove the sources, and move the fission terms to the **B** matrix.

```
In [3]: def DiffusionEigenvalue(R,I,D,Sig_a,nuSig_f,BC, geometry,epsilon = 1.0e-8):
            """Solve a neutron diffusion eigenvalue problem in a 1-D geometry
            using cell-averaged unknowns
            Args:
                R: size of domain
                I: number of cells
                D: name of function that returns diffusion coefficient for a given r
                Sig_a: name of function that returns Sigma_a for a given r
                nuSig_f: name of function that returns nu Sigma_f for a given r
                BC: Boundary Condition at r=R in form [A,B]
                geometry: shape of problem
                        0 for slab
                        1 for cylindrical
                        2 for spherical

            Returns:
                k: the multiplication factor of the system
                phi:  the fundamental mode with norm 1
                centers: position at cell centers

            """
            #create the grid
            Delta_r, centers, edges = create_grid(R,I)
            A = np.zeros((I+1,I+1))
            B = np.zeros((I+1,I+1))
            #define surface areas and volumes
            assert( (geometry==0) or (geometry == 1) or (geometry == 2))
            if (geometry == 0):
                #in slab it's 1 everywhere except at the left edge
```

```
        S = 0.0*edges+1
        S[0] = 0.0 #to enforce Refl BC
        #in slab its dr
        V = 0.0*centers + Delta_r
    elif (geometry == 1):
        #in cylinder it is 2 pi r
        S = 2.0*np.pi*edges
        #in cylinder its pi (r^2 - r^2)
        V = np.pi*( edges[1:(I+1)]**2
                   - edges[0:I]**2 )
    elif (geometry == 2):
        #in sphere it is 4 pi r^2
        S = 4.0*np.pi*edges**2
        #in sphere its 4/3 pi (r^3 - r^3)
        V = 4.0/3.0*np.pi*( edges[1:(I+1)]**3
                   - edges[0:I]**3 )

    #Set up BC at R
    A[I,I] = (BC[0]*0.5 + BC[1]/Delta_r)
    A[I,I-1] = (BC[0]*0.5 - BC[1]/Delta_r)


    #fill in rest of matrix
    for i in range(I):
        r = centers[i]
        A[i,i] = (0.5/(Delta_r * V[i])*((D(r)+D(r+Delta_r))*S[i+1]) +
                  Sig_a(r))
        B[i,i] = nuSig_f(r)
        if (i>0):
            A[i,i-1] = -0.5*(D(r)+D(r-Delta_r))/(Delta_r * V[i])*S[i]
            A[i,i] += 0.5/(Delta_r * V[i])*((D(r)+D(r-Delta_r))*S[i])
        A[i,i+1] = -0.5*(D(r)+D(r+Delta_r))/(Delta_r * V[i])*S[i+1]

    #find eigenvalue
    l,phi = inversePower(A,B,epsilon)
    k = 1.0/l
    #remove last element of phi because it is outside the domain
    phi = phi[0:I]
    return k, phi, centers
```

To test this code we should compute the eigenvalue and fundamental mode for a homogeneous 1-D reactor. We will set up the problem with $D = 3.850204978408833$ cm, $\nu\Sigma_\mathrm{f} = 0.1570$ cm^{-1}, and $\Sigma_\mathrm{a} = 0.1532$ cm^{-1}. We also know that in spherical geometry the critical size can be found from

$$\frac{\nu\Sigma_\mathrm{f} - \Sigma_\mathrm{a}}{D} = \left(\frac{\pi}{R}\right)^2,$$

which leads to

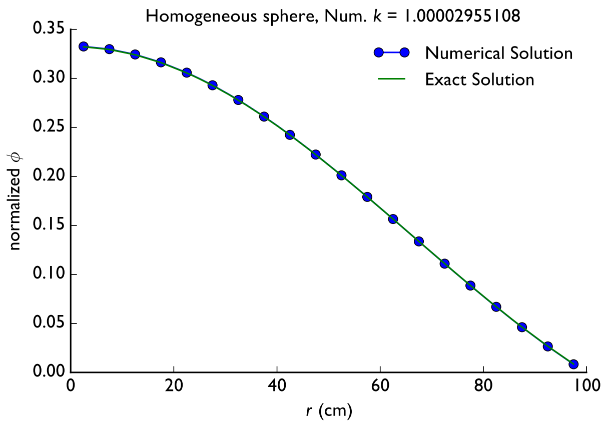
$$R^2 = \frac{\pi^2 D}{\nu\Sigma_\mathrm{f} - \Sigma_\mathrm{a}}.$$

For this system the critical size is 100 cm.

When we run our eigenvalue solver on this problem, we should see that the eigenvalue converges to 1 as the number of mesh points is refined, also the eigenvector should be the fundamental mode, which has the shape

$$\phi(r) = \frac{A}{r}\sin\left(\frac{\pi r}{R}\right).$$

```
In [4]: nuSigmaf_func = lambda r: 0.1570
        D_func = lambda r: 3.850204978408833
        Sigmaa_func = lambda r: 0.1532
        R = 100
        I = 20
        #solution in spherical geometry with 100 cells
        k, phi,centers = DiffusionEigenvalue(R,I,D_func,
                                            Sigmaa_func,nuSigmaf_func,
                                            [1,0],
                                            2, epsilon=1.0e-10)
```



The flux shape looks correct, but the eigenvalue is slightly off. We can see how the eigenvalue solution converges to the right answer.

```
In [5]: Points = np.array((10,20,40,80,160,320))
        for I in Points:
            k, phi,centers = DiffusionEigenvalue(R,I,D_func,Sigmaa_func,
                                                nuSigmaf_func,[1,0],
                                                2,epsilon=1.0e-9)
            print("I =", I, "\t\tk =",k,"\tError =",np.fabs(k-1))
```

```
I = 10      k = 1.00011764407   Error = 0.000117644067299
I = 20      k = 1.00002956354   Error = 2.95635412424e-05
I = 40      k = 1.00000740964   Error = 7.40964406831e-06
I = 80      k = 1.00000186269   Error = 1.86269399705e-06
I = 160     k = 1.00000047537   Error = 4.75371085829e-07
I = 320     k = 1.00000012886   Error = 1.28862679416e-07
```

We can do the same check for slab geometry. In this case, the critical half-size of the reactor is

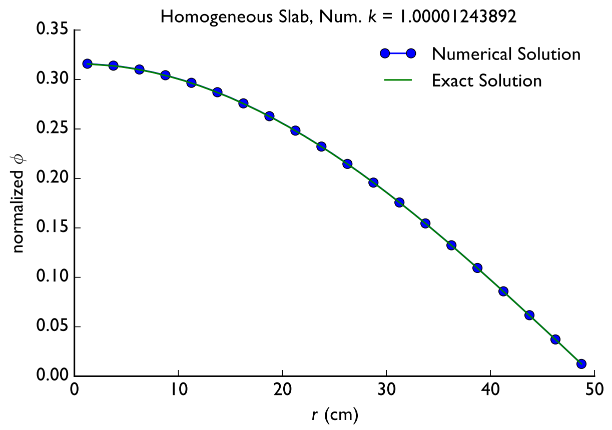
$$4R^2 = \frac{\pi^2 D}{\nu \Sigma_\mathrm{f} - \Sigma_\mathrm{a}},$$

giving a critical size of 50 cm.

The shape in this case is

$$\phi(r) = A \cos\left(\frac{\pi r}{R}\right).$$

The solution with 20 cells is



Again, we match the exact solution pretty well with only 20 cells.

The final check is the cylinder. For a cylinder

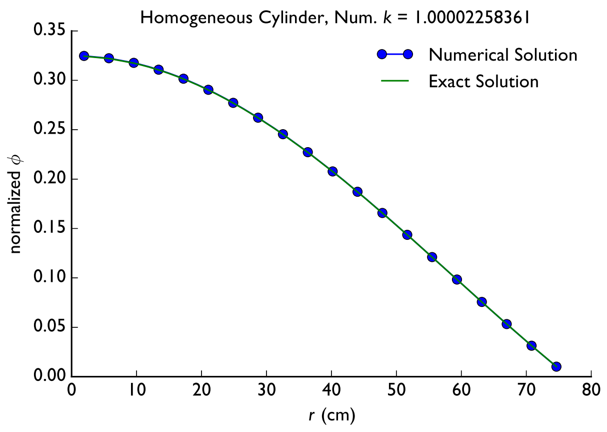
$$R^2 = \frac{2.405^2 D}{\nu \Sigma_\mathrm{f} - \Sigma_\mathrm{a}},$$

and the fundamental mode is

$$\phi(r) = C J_0\left(\frac{2.405 r}{R}\right),$$

where J_0 is a Bessel function of the first kind. The critical size is about 76.5535 cm.

Once again, 20 cells gives us a reasonable approximation to the exact solution:

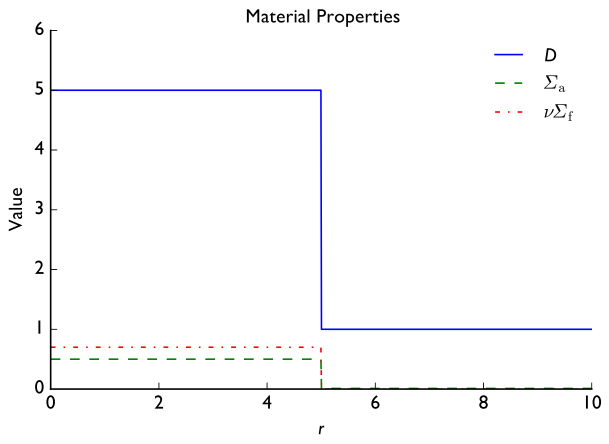


We have not done enough homogeneous problems to call this a thorough verification of the code, but we do have some confidence that the code is working correctly.

19.4.1 Heterogeneous Problems

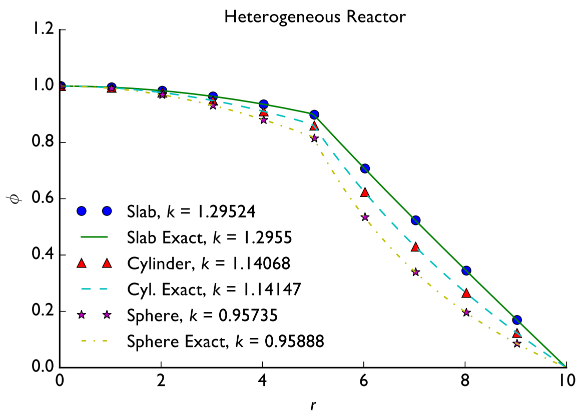
Previously, we looked at a problem with fuel on the interior and a reflector on the outside. The fuel extends from $r = 0$ to $r = 5$ cm and the reflector extends out to $R = 10$.

```
In [6]: def D(r):
            value = 5.0*(r<=5) + 1.0*(r>5)
            return value;
        def Sigma_a(r):
            value = 0.5*(r<=5) + 0.01*(r>5)
            return value;
        def nuSigma_f(r):
            value = 0.7*(r<=5) + 0.0*(r>5)
            return value;
```



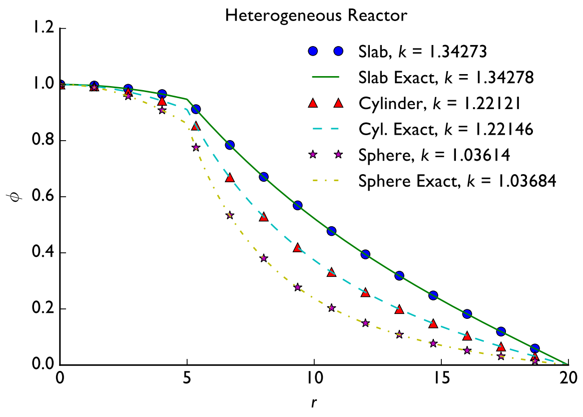
For this problem the eigenvalue in each geometry is found in the next code snippet. We then plot the solution and compare it with the exact solution and its eigenvalue.

```
In [7]:  R = 10
         I = 200
         #Solve Diffusion Problem in slab geometry
         k, phi_slab,centers = DiffusionEigenvalue(R, I,D, Sigma_a,
                                                   nuSigma_f,[1,0], 0)
         #Solve Diffusion Problem in cylindrical geometry
         kc, phi_cyl,centers_cyl = DiffusionEigenvalue(R, I,D, Sigma_a,
                                                       nuSigma_f,[1,0], 1)
         #Solve Diffusion Problem in spherical geometry
         ks, phi_sphere,centers_sp = DiffusionEigenvalue(R, I,D,Sigma_a,
                                                         nuSigma_f,[1,0], 2)
```



With 200 cells the eigenvalues are approximated to about 4 digits of accuracy. We see that the eigenvalue goes down as we go from slab to cylinder to sphere. This is not a surprise because the slab geometry is infinite in 2 directions, cylinder is infinite in 1, and the sphere is a finite size. Therefore, the leakage goes up as I make the problem "smaller".

One thing we should see is that the eigenvalue goes up as we make the moderator larger. Doubling the thickness of the reflector should increase k_{eff}:



This does depress the leakage and increases the multiplication factor, as expected. With 600 cells we get 5 digits of accuracy in the slab and 4 digits in cylinders and spheres for the eigenvalue.

CODA

In this chapter we adapted our diffusion code for source driven problems to solve k-eigenvalue problems to find the multiplication factor for a system. The basis for this is inverse power iteration. At each step of the iterative process we have to solve a steady-state diffusion problem where the source is the fission neutrons created by the previous iteration's solution.

Eigenvalue calculations are important in reactor design, but the one-group approximation is a bit dubious for thermal, light water reactors. A better model for these systems is a two-group diffusion model where we solve for the scalar flux of fast and thermal neutrons separately. This is done in the next chapter.

PROBLEMS

Programming Projects

1. Reflector Effective Albedo

For the heterogeneous reactor example used above, determine the effective albedo of the reflector in the slab case. To do this you will solve for the eigenvalue of a slab reactor with $R =$ 5 cm with an albedo boundary condition. Then your task is to find a value of α, the fraction of neutrons reflected back, that produces a value of k_{eff} that matches that of the reactor with a reflector. You can use a nonlinear solver to accomplish this by defining a function that calls the eigenvalue solver with a particular value of α in the albedo boundary condition and returns the difference between k_{eff} and the eigenvalue for the reactor with a reflector.

2. Spherical Plutonium Reactor k-Eigenvalue

For the burst reactor made of plutonium in the defined in the "super-critical excursion" problem of Chapter 18, compute the value of k_{eff} with and without the plug inserted.

3. Criticality for 1-D Heterogeneous System Using the 1-Group Neutron Diffusion Equation

Consider a 1-D heterogeneous cylinder of thickness R. The medium is made of 5 regions 10 cm thick (total domain size is $R = 50$ cm):

Region	1	2	3	4	5
Medium	fuel 1	fuel 3	fuel 1	fuel 2	reflector 1

Use a uniform mesh size of $\Delta r = 1$ cm (i.e., 10 cells per region, i.e., a total of 50 cells). Find the value of k_{eff} for this reactor. Plot the solution for $\phi(r)$ in the reactor and comment on the shape of the solution.

The data you will need:

Medium	D **(cm)**	Σ_a **(cm^{-1})**	$\nu\Sigma_f$ **(cm^{-1})**
reflector 1	2	0.020	0
fuel 1	1.1	0.070	0.095
fuel 2	1.2	0.065	0.095
fuel 3	1.5	0.085	0.095