13

Monte Carlo Eigenvalue Calculations

13.1 Fission Cycles

As we have discussed we often want to compute the k-eigenvalue for a system under consideration. We can use Monte Carlo to do this. The issue is that we need to track neutrons in generations to compute the ratio of neutrons in successive generations. To do this we use fission cycles.

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 $\bar{nu}$. We then track these neutrons to get the fission sites for the birth of neutrons in the next generation. The ratio between the number of neutrons born from fission between these generations is an estimate of the eigenvalue. If we do 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 initital fission sites. If we guess these sites, 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.

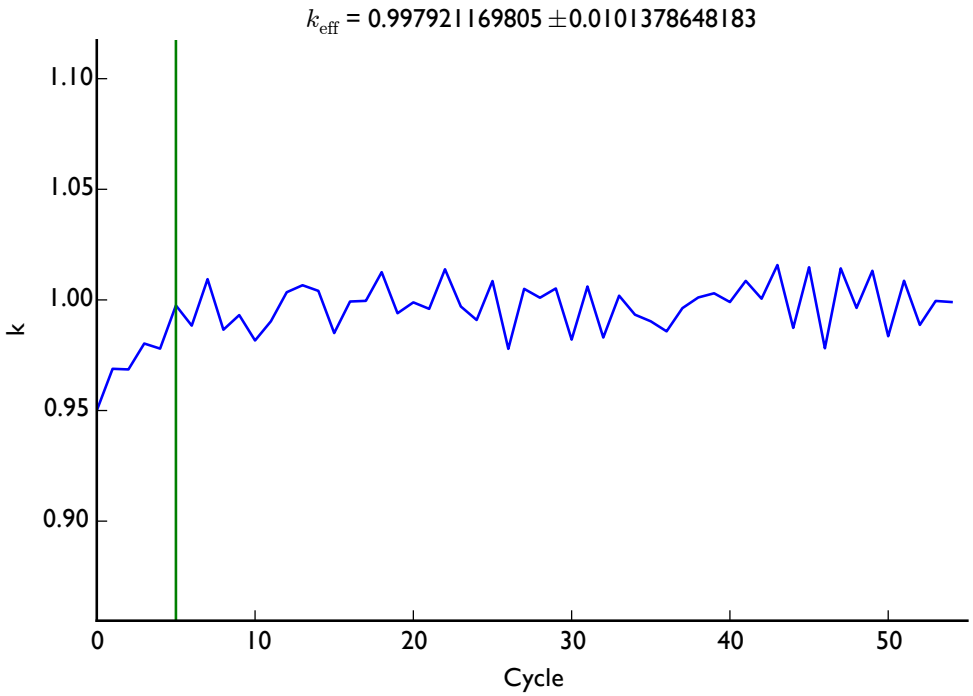
To demonstrate how this works we will consider a homogeneous slab with single speed neutrons where $\Sigma_t = 1$, $\Sigma_s = 0.75$, $\Sigma_f = 0.1$, and $\bar{\nu} = 2.5$. The value of k_∞ for this system is 1.

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        #this next line is only needed in iPython notebooks
        %matplotlib inline
        import matplotlib
        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
            #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 = -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):
                                    #fission
                                    fission_sites = np.vstack((fission_sites,position))
                                    fission_site_weights = np.vstack((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
                #plt.plot(fission_sites,1+cycle+0*fission_sites,'o')
                #sample neutrons for next generation from fission sites
                num_per_site = int(np.ceil(N/fission_sites.size))
                positions = np.zeros(1)
                weights = np.zeros(1)
                mus = np.random.uniform(-1,1,num_per_site*fission_sites.size)
                for site in np.hstack((fission_sites, fission_site_weights)):
                    positions = np.vstack((positions,site[0]*np.ones((num_per_site,1))))
                    weights = np.vstack((weights,site[1] * 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

In [2]: N = 10000
        Sig_t = 1
        Sig_s = 0.75
        Sig_f = 0.10285
        thickness = 20
        nu = 2.5
        inactive_cycles = 5
        active_cycles = 50
        k = homog_slab_k(N,Sig_t,Sig_s,Sig_f,nu, thickness, inactive_cycles, active_cycles)
        plt.plot(k)
        plt.title(str(np.mean(k[inactive_cycles:(active_cycles+inactive_cycles)]))+
                  "$\pm$" + str(np.std(k[inactive_cycles:(active_cycles+inactive_cycles)])))
        plt.xlabel("Cycle")
        plt.ylabel("k")
```

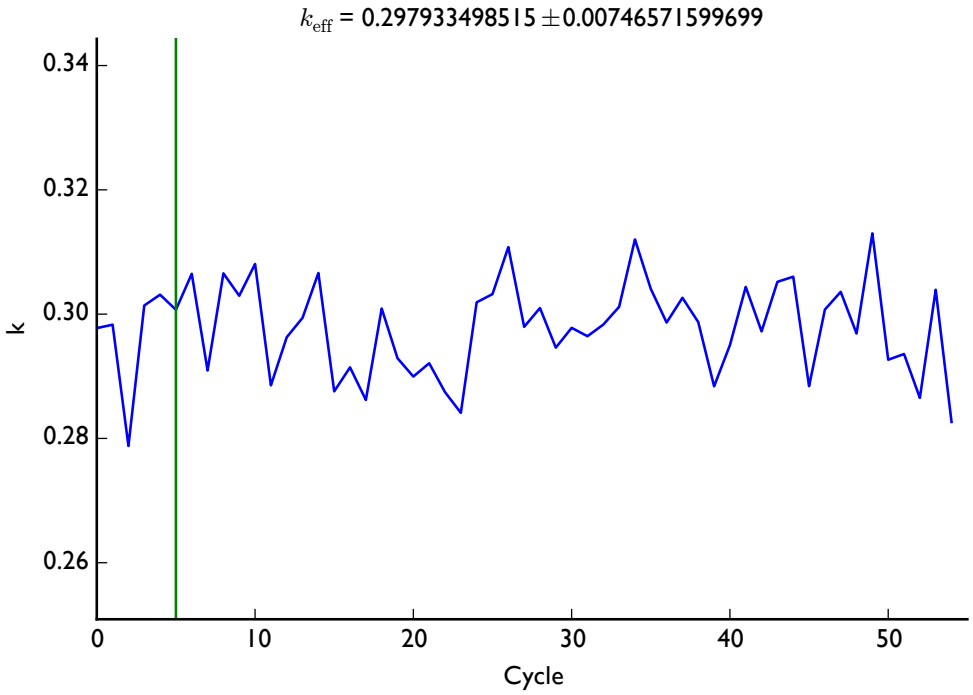
```
        plt.plot([inactive_cycles,inactive_cycles],[np.min(k)*.9,np.max(k)*1.1])
        plt.axis([0,k.size,np.min(k)*.9,np.max(k)*1.1])
        print("Sn gives k = 0.99994958525")
        plt.show()
```

```
Sn gives k = 0.99994958525
```




```
In [3]: N = 10000
        Sig_t = 1
        Sig_s = 0.75
        Sig_f = 0.10285
        thickness = 1
        nu = 2.5
        inactive_cycles = 5
        active_cycles = 50
        k = homog_slab_k(N,Sig_t,Sig_s,Sig_f,nu, thickness, inactive_cycles, active_cycles)
        plt.plot(k)
        plt.title(str(np.mean(k[inactive_cycles:(active_cycles+inactive_cycles)]))+
                  "$\pm$" + str(np.std(k[inactive_cycles:(active_cycles+inactive_cycles)])))
        plt.xlabel("Cycle")
        plt.ylabel("k")
        plt.plot([inactive_cycles,inactive_cycles],[np.min(k)*.9,np.max(k)*1.1])
        plt.axis([0,k.size,np.min(k)*.9,np.max(k)*1.1])
        print("Sn gives k = 0.297107720373")
        plt.show()
```

```
Sn gives k = 0.297107720373
```



13.2 Fission Matrix Methods

Consider the integral operator $\mathcal{H}$ defined as

$$\mathcal{H}s(\mathbf{r}, E) = \int dE' \int_V dV F(\mathbf{r}', E' \rightarrow \mathbf{r}, E) s(\mathbf{r}', E'),$$

where $F(r', E' \rightarrow r, E)$ is the expected number of fission neutrons born at position $\mathbf{r}$ and energy E from a fission neutron born at $\mathbf{r}'$ with energy E'. 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

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

The elements of this matrix can be estimated via Monte Carlo. Typically, one does this by assuming a flat source in each region of the problem. We can then solve the eigenvalue problem

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

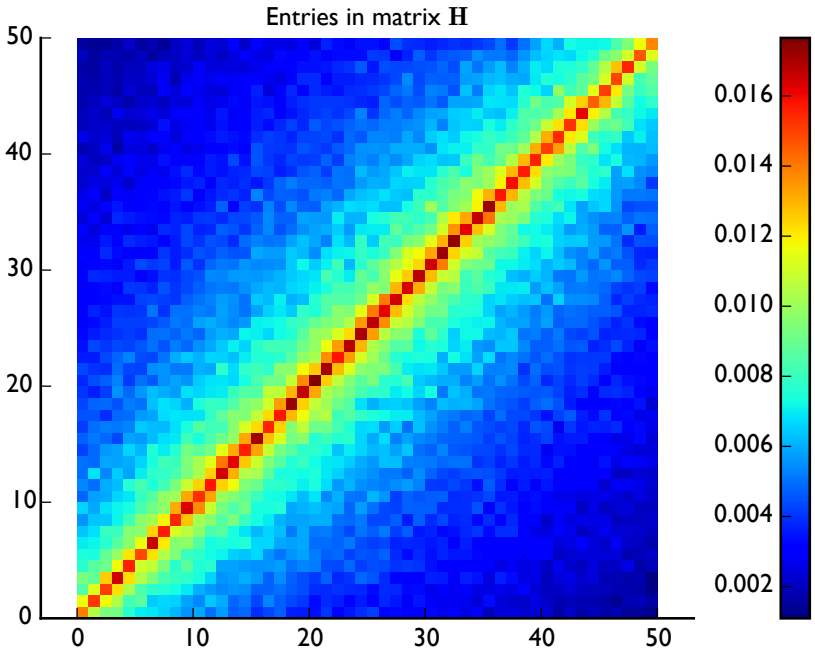
If the regions are small enough so that the error in making the source flat is negligible, then we can interpret $\hat{k}$ an eigenvalue of the system. This means that the dominant eigenvalue $\hat{k}$ and its associated eigenvector are the fundamental mode for the system. We could also find the other 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 demonstrate that the magnitude of the imaginary part of the eigenvalues is a measure of the uncertainty in the fundamental mode eigenvalue.

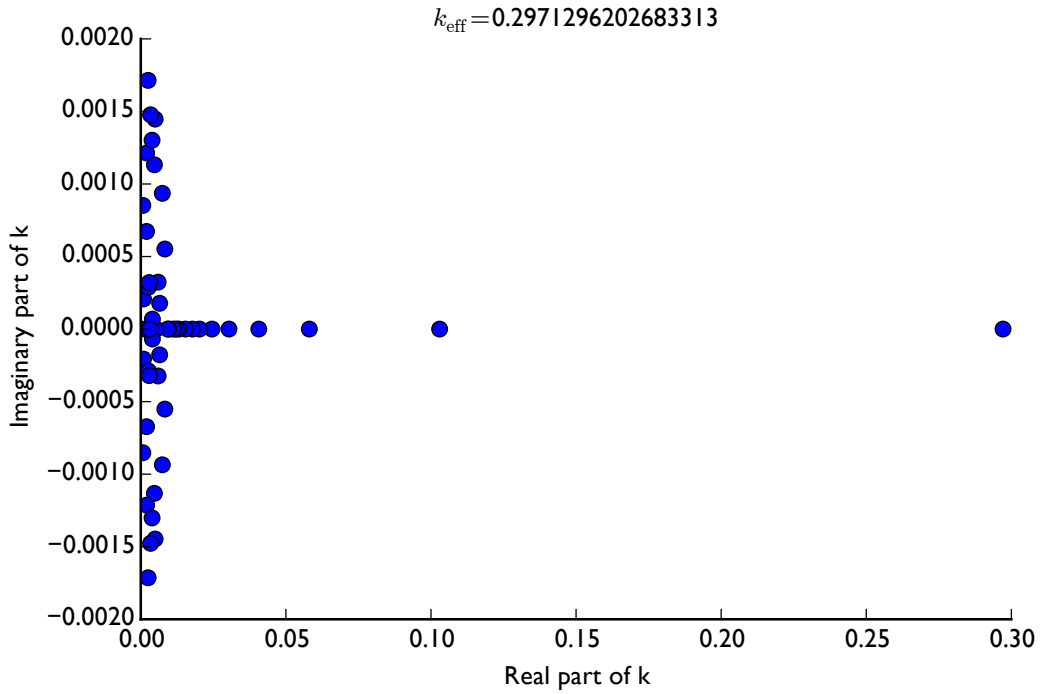
To compute the fission matrix we need to specify a mesh over the problem, and then emit neutrons in each region, and count the number of fission neutrons born in each other region. We will demonstrate this in our homogeneous slab problem.

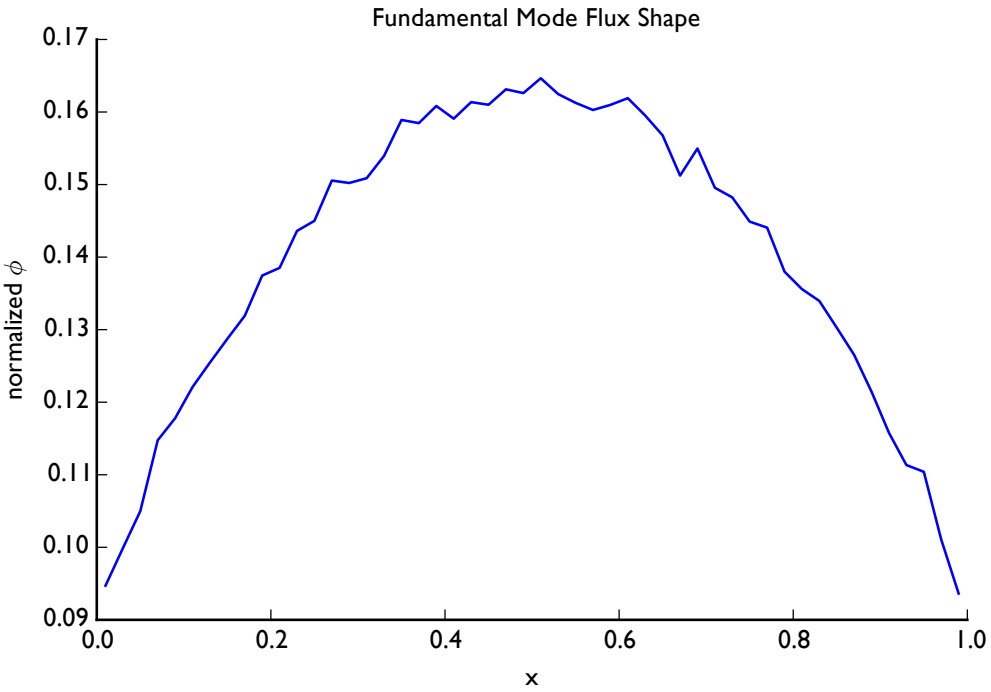
```
In [4]: def fission_matrix(N,Sig_t,Sig_s,Sig_f,nu, thickness,Nx):
            Sig_a = Sig_t - Sig_s
            H = np.zeros((Nx,Nx))
            dx = thickness/Nx
            lowX = np.linspace(0,thickness-dx,Nx)
            highX = np.linspace(dx,thickness,Nx)
            midX = np.linspace(dx*0.5,thickness-dx*0.5,Nx)
            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

In [5]: N = 80000
        Nx = 50
        Sig_t = 1
```

```
Sig_s = 0.75
Sig_f = 0.10285
thickness = 1
nu = 2.5
H,Xmid = fission_matrix(N,Sig_t,Sig_s,Sig_f,nu, thickness,Nx)
plt.pcolormesh(H)
plt.title("Entries in matrix $\mathbf{H}$")
plt.colorbar()
plt.show()
ks, v = np.linalg.eig(H)
print("Sn gives k = 0.297107720373")
plt.plot(np.real(ks),np.imag(ks),'o')
plt.title("$k_\mathrm{eff} = $" + str(np.real(np.max(ks))))
plt.xlabel("Real part of k")
plt.ylabel("Imaginary part of k")
plt.show()
plt.plot(Xmid, np.abs(v[:,np.argmax(ks)]))
plt.title("Fundamental Mode Flux Shape")
plt.xlabel("x")
plt.ylabel("normalized $\phi$")
plt.show()
```

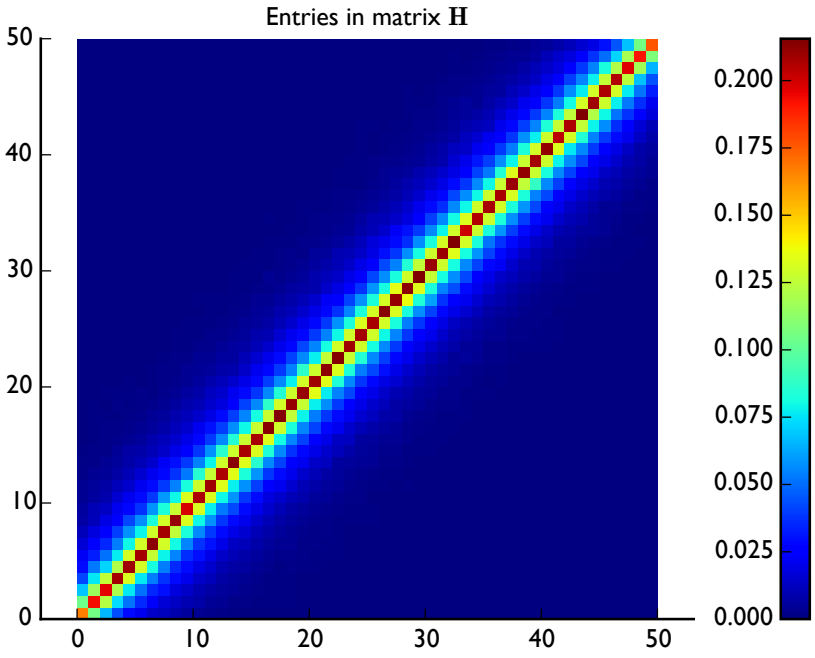



```
Sn gives k = 0.297107720373
```

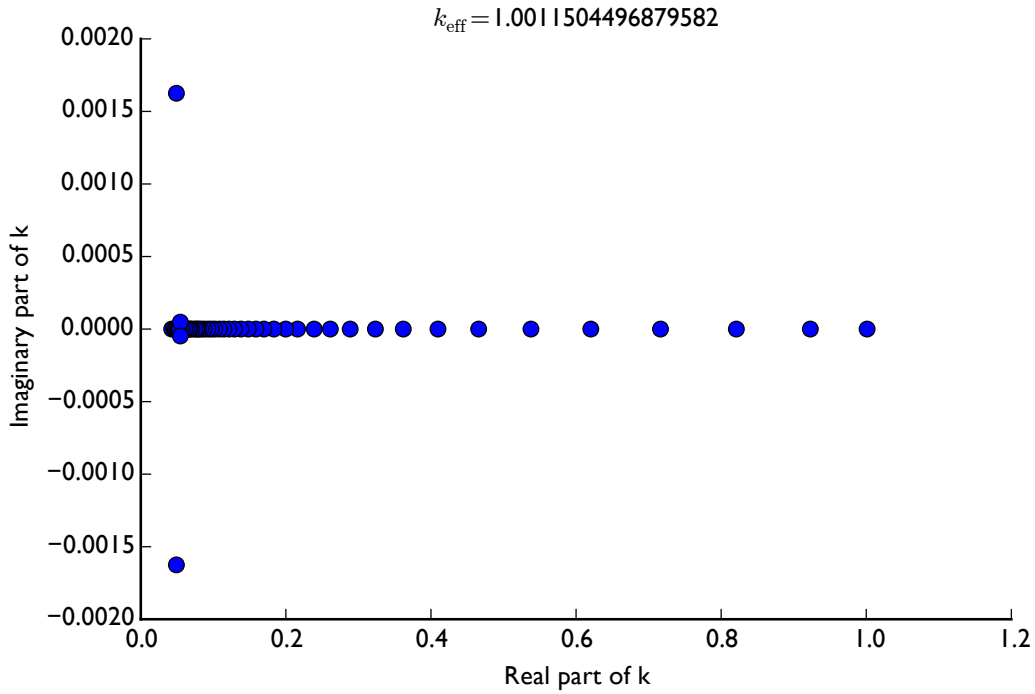

$k_{\text{eff}} = 0.2971296202683313$
Imaginary part of k
Real part of k
0.0020
0.0015
0.0010
0.0005
0.0000
−0.0005
−0.0010
−0.0015
−0.0020
0.00
0.05
0.10
0.15
0.20
0.25
0.30

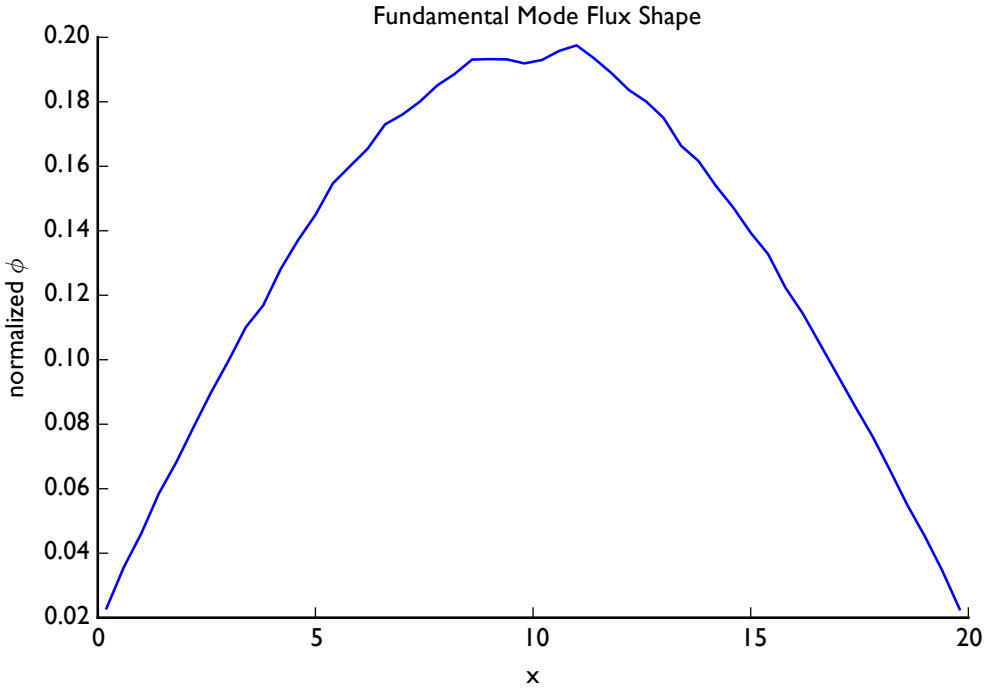

Fundamental Mode Flux Shape
normalized ϕ
x
0.17
0.16
0.15
0.14
0.13
0.12
0.11
0.10
0.09
0.0
0.2
0.4
0.6
0.8
1.0

In [6]:
```
N = 40000
Nx = 50
Sig_t = 1
Sig_s = 0.75
Sig_f = 0.10285
thickness = 20
nu = 2.5
H,Xmid = fission_matrix(N,Sig_t,Sig_s,Sig_f,nu, thickness,Nx)
plt.pcolormesh(H)
plt.title("Entries in matrix $\mathbf{H}$")
plt.colorbar()
plt.show()
ks, v = np.linalg.eig(H)
print("Sn gives k = 0.99994958525")
plt.plot(np.real(ks),np.imag(ks),'o')
plt.title("$k_\mathrm{eff} = $" + str(np.real(np.max(ks))))
plt.xlabel("Real part of k")
plt.ylabel("Imaginary part of k")
plt.show()
plt.plot(Xmid, np.abs(v[:,np.argmax(ks)]))
plt.title("Fundamental Mode Flux Shape")
plt.xlabel("x")
plt.ylabel("normalized $\phi$")
plt.show()
```




```
Sn gives k = 0.99994958525
```


$k_{\text{eff}} = 1.0011504496879582$
Imaginary part of k
Real part of k
0.0020
0.0015
0.0010
0.0005
0.0000
−0.0005
−0.0010
−0.0015
−0.0020
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Fundamental Mode Flux Shape
normalized ϕ
x
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
5
10
15
20