12

Introduction to Monte Carlo Methods

12.1 Variance Reduction

Last time we talked about how to construct particles using a computer program that were analogs of real neutrons moving through a system. We saw that in many cases it took very many simulated neutrons to get reasonable answers. Today we will see that by making our simulated particles behave in a non-analog way we can improve our answers. We will not improve the convergence rate of Monte Carlo methods. Recall that the statistical error as measured by the standard deviation of the estimate converges as $O(N^{-1/2})$, where N is the number of simulated particles. This means that the standard deviation of our estimate is, to leading order, $CN^{-1/2}$. What variance reduction does is reduce the magnitude of C, the constant in the convergence.

12.1.1 Implicit Capture and Particle Weights

In problems where radiative capture is important, it can be beneficial to remove this absorption process from the types of interactions considered. To do this we first introduce a particle weight, w. The weight is defined so that each particle represents a collection of neutrons, rather than a single one. Consider a neutron source in a volume, Q, with units neutrons per cm^3 per second. We will use N particles each with a weight w_i to represent this source in steady state calculation. The weights must satisfy

$$\sum_{i=1}^{N} w_i = \int_V dV Q.$$

From this relation we see that the units of w_i are neutrons per second.

To use implicit capture, as a particle moves a distance s in a material, we reduce its weight by a factor of $\exp(-\Sigma_\gamma s)$. This is done because this factor represents the likelihood of the neutron traveling a distance s without having a radiative capture reaction. With implicit capture, our simple shielding calculation has its procedure changed to be

The algorithm for this problem just builds on what we did before.

Create a counter, $t = 0$ to track the number of neutrons that get through.

Create neutron with μ sampled from the uniform distribution $\mu \in [0, 1]$. Set $x = 0$. Set the particle's weight to be $w = 1/N$.

Sample randomly a distance to scatter, l, from the exponential distribution.

Move the particle to $x = x + l\mu$.

Reduce the weight of the particle by a factor $\exp(-\Sigma_\gamma s)$

Check to see if $x > 3$. If so $t = t + w$, and go to 2. Otherwise, if $x < 0$ go to step 2.

Go back to step 3.

A couple of notes on this new algorithm. Now each particle has a weight and that is what we sum up to get the fraction of neutrons that leak out. Also, when we sample a distance to collision, we only sample a distance to scatter. The code to implement this algorithm is below.

```
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 slab_transmission(Sig_s,Sig_a,thickness,N,isotropic=False, implicit_capture = True):
    """Compute the fraction of neutrons that leak through a slab
    Inputs:
    Sig_s:     The scattering macroscopic x-section
    Sig_a:     The absorption macroscopic x-section
    thickness: Width of the slab
    N:         Number of neutrons to simulate
    isotropic: Are the neutrons isotropic or a beam

    Returns:
    transmission:  The fraction of neutrons that made it through
    """
    Sig_t = Sig_a + Sig_s
    transmission = 0.0
    N = int(N)
    for i in range(N):
        if (isotropic):
            mu = np.random.random(1)
        else:
            mu = 1.0
        x = 0
        alive = 1
        weight = 1.0/N
        while (alive):
            if (implicit_capture):
                #get distance to collision
                if (Sig_s > 0):
                    l = -np.log(1-np.random.random(1))/Sig_s
                else:
                    l = 10.0*thickness/mu #something that will make it through
            else:
                #get distance to collision
                l = -np.log(1-np.random.random(1))/Sig_t
            #make sure that l is not too large
            if (mu > 0):
                l = np.min([l,(3-x)/mu])
            else:
                l = np.min([l,-x/mu])
            #move particle
            x += l*mu
            if (implicit_capture):
                if not(l>=0):
                    print(l,x,mu)
                assert(l>=0)
                weight *= np.exp(-l*Sig_a)
            #still in the slab?
            if (np.abs(x-thickness) < 1.0e-14):
                transmission += weight
                alive = 0
            elif (x<= 1.0e-14):
                alive = 0
            else:
```

```
                    if (implicit_capture):
                        mu = np.random.uniform(-1,1,1)
                    else:
                        #scatter or absorb
                        if (np.random.random(1) < Sig_s/Sig_t):
                            #scatter, pick new mu
                            mu = np.random.uniform(-1,1,1)
                        else: #absorbed
                            alive = 0
        return transmission
```

The example problem of a beam incident on a pure absorber should be exact using implicit capture using only one particle.

```
In [2]: N = 1
        Sigma_s = 0.0
        Sigma_a = 2.0
        thickness = 3
        transmission = slab_transmission(Sigma_s,Sigma_a, thickness,
                                         N, isotropic=False, implicit_capture=True)
        print("The fraction that made it through using implicit capture was",
              transmission, "with a percent error of",np.abs(transmission - np.exp(-6))/np.exp(-6)*100,"%")
        transmission = slab_transmission(Sigma_s,Sigma_a, thickness, N, isotropic=False, implicit_capture=False)
        print("The fraction that made it through using analog tracking was",
              transmission, "with a percent error of",np.abs(transmission - np.exp(-6))/np.exp(-6)*100,"%")
```

```
The fraction that made it through using implicit capture was 0.00247875217667 with a percent error of 0.0 %
The fraction that made it through using analog tracking was 0.0 with a percent error of 100.0 %
```

For isotropic incident neutrons on the same slab, we expect implicit capture to be better than analog tracking as well.

```
In [3]: N = 1000
        Sigma_s = 0.0
        Sigma_a = 2.0
        thickness = 3
        true_sol = 0.0003182574636904064672 7
        transmission = slab_transmission(Sigma_s,Sigma_a, thickness,
                                         N, isotropic=True, implicit_capture=True)
        print("The fraction that made it through using implicit capture was",
              transmission, "with a percent error of",np.abs(transmission - true_sol)/true_sol*100,"%")
        transmission = slab_transmission(Sigma_s,Sigma_a, thickness, N, isotropic=True, implicit_capture=False)
        print("The fraction that made it through using analog tracking was",
              transmission, "with a percent error of",np.abs(transmission - true_sol)/true_sol*100,"%")
```

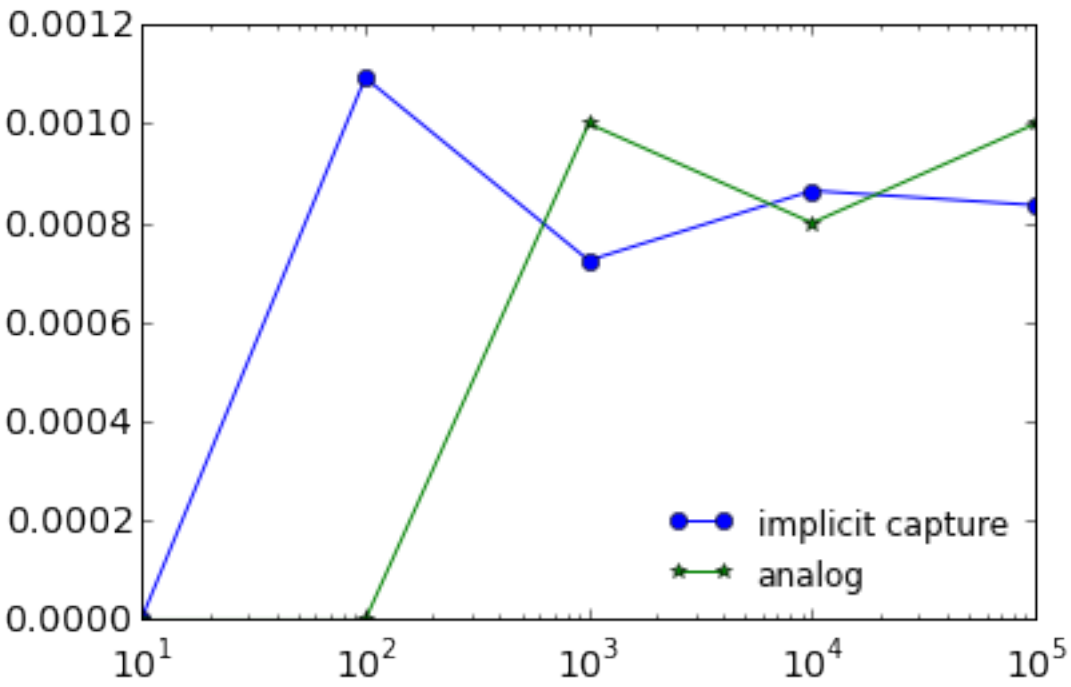
```
The fraction that made it through using implicit capture was 0.000308930894796 with a percent error of 2.9305106584
The fraction that made it through using analog tracking was 0.0 with a percent error of 100.0 %
```

Implicit capture reduces our error from 100% to less than 5% with only 1000 particles. Previously, we needed about 2 million particles to get that accuracy.

The solution to the problem with scattering is below. We will compare the convergence in this case.

```
In [4]: N_parts = [10,100,1000,10000, 1e5]
        solution_implicit = np.zeros(len(N_parts))
        solution_analog = np.zeros(len(N_parts))
        Sigma_s = 0.75
```

```
Sigma_a = 2.0 - Sigma_s
count = 0
for N in N_parts:
    solution_implicit[count] = slab_transmission(Sigma_s,Sigma_a,
                                                 thickness, N, isotropic=True, implicit_capture=True)
    solution_analog[count] = slab_transmission(Sigma_s,Sigma_a,
                                               thickness, N, isotropic=True, implicit_capture=False)
    count += 1
plt.semilogx(N_parts, solution_implicit, 'o-', label = "implicit capture")
plt.semilogx(N_parts, solution_analog, '*-', label = "analog")
plt.legend(loc="best")
plt.show()
```



12.1.2 A Figure of Merit

A way of measuring the benefit of a variance reduction technique is through the quantity called the figure of merit (FOM)

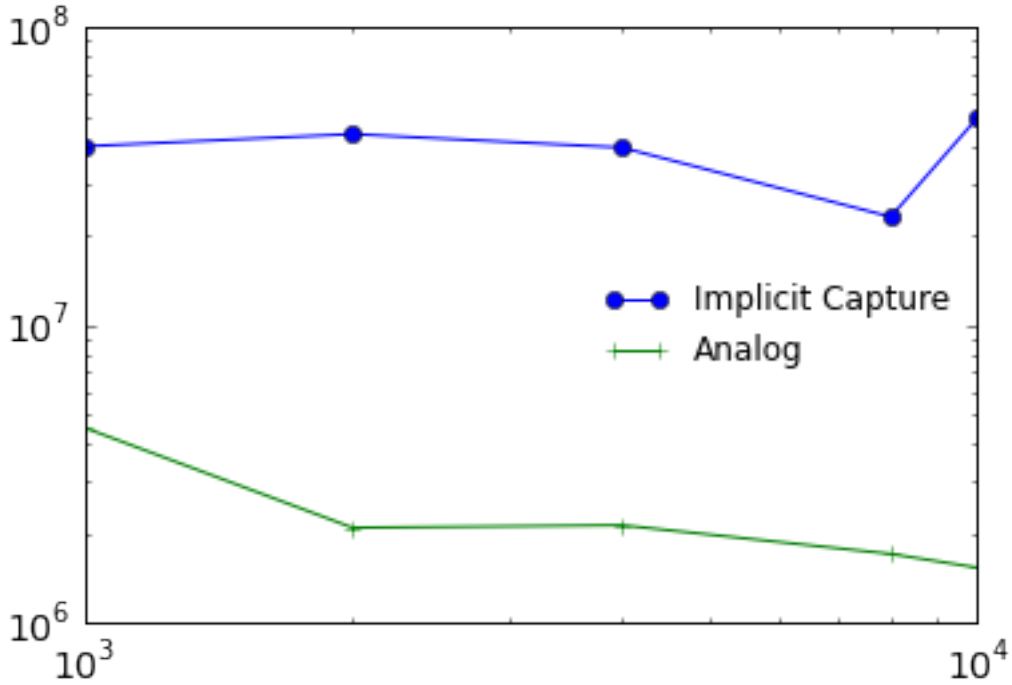
$$\text{FOM} = \frac{1}{\sigma^2 T}.$$

It should be said that this is a figure of merit, not the figure of merit because others are possible. In the expression for FOM, σ^2 is the variance in an estimate and T is the time it takes to get the estimate.

Let's try this on our slab problem. We will run the problem 10 times with each number of particles, compute the time to solution, and the variance of the runs, and then plot the FOM. Remember that a larger number is better (lower variance and/or time). We expect that the implicit capture method should be better because it appears to give less error (see above) and the exponentials we evaluate should not be costly relative to the particle tracking.

```
In [5]: import time
        N_parts = [1000,2000,4000,8000,10000]
```

```
solution_implicit = np.zeros((len(N_parts),10))
solution_analog = np.zeros((len(N_parts),10))
times_implicit = np.zeros(len(N_parts))
times_analog = np.zeros(len(N_parts))
var_implicit = np.zeros(len(N_parts))
var_analog = np.zeros(len(N_parts))
Sigma_s = 0.75
Sigma_a = 2.0 - Sigma_s
count = 0
for N in N_parts:
    for replicate in range(10):
        tmp = time.clock()
        solution_implicit[count,replicate] = slab_transmission(Sigma_s,Sigma_a,
                                                               thickness, N,
                                                               isotropic=True, implicit_capture=True)
        times_implicit[count] += time.clock()-tmp
        tmp = time.clock()
        solution_analog[count,replicate] = slab_transmission(Sigma_s,Sigma_a,
                                                             thickness, N,
                                                             isotropic=True, implicit_capture=False)
        times_analog[count] += time.clock()-tmp
    var_implicit[count] = np.std(solution_implicit[count,:])**2
    var_analog[count] = np.std(solution_analog[count,:])**2
    count += 1
plt.loglog(N_parts,1.0/(var_implicit*times_implicit),'o-',label="Implicit Capture")
plt.loglog(N_parts,1.0/(var_analog*times_analog),'+-',label="Analog")
plt.legend(loc="best")
plt.show()
```



12.1.3 Collision Estimators

Now that we have introduced our first variance reduction technique, we will discuss scalar flux estimators before moving on to other techniques. The first that we will consider is the collision estimator. Consider the reaction rate in a volume, defined by

$$R = \int_V dV\, \Sigma_t(\mathbf{r})\phi(\mathbf{r}).$$

If insider the region the cross-section is constant, we can compute the average scalar flux via the relation

$$\bar{\phi} = \frac{1}{V}\int_V dV\, \phi(\mathbf{r}) = \frac{R}{V\Sigma_t}.$$

Therefore, if we sum, or tally, the weight from each collision inside the volume and divide that count by the total cross-section, we get an estimate of the scalar flux.

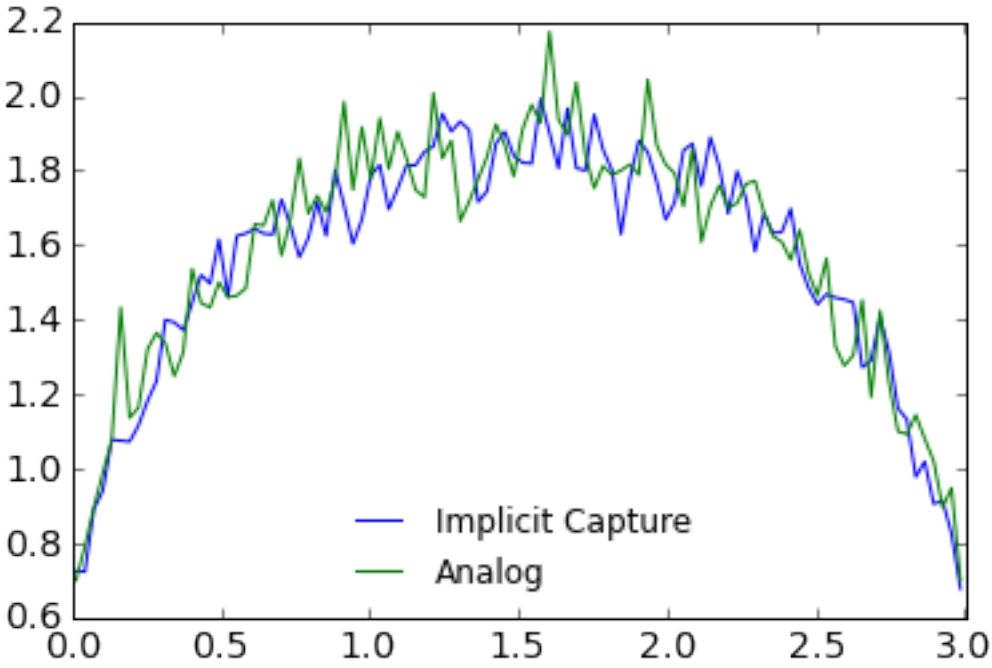
Notice, however, that we cannot use this in voids. We can, however, use other processes to estimate the scalar flux. For example, we could compute the scattering rate and divide by Σ_s, for implicit capture this is what we will do.

The slab problem from above will be modified for this purpose. We will introduce a mesh onto the problem and count the reactions in each mesh cell. Also, instead of a source on the boundary we will add a volumetric source to the problem. The source will be uniform throughout. Therefore, we need to sample a position of the neutron's birth as well as an angle in $\mu \in [-1, 1]$.

```
In [6]: def slab_source(Nx,Sig_s,Sig_a,thickness,N,Q,isotropic=False, implicit_capture = True):
            """Compute the fraction of neutrons that leak through a slab
            Inputs:
            Nx:        The number of grid points
            Sig_s:     The scattering macroscopic x-section
            Sig_a:     The absorption macroscopic x-section
            thickness: Width of the slab
            N:         Number of neutrons to simulate
            isotropic: Are the neutrons isotropic or a beam

            Returns:
            transmission:  The fraction of neutrons that made it through
            scalar_flux:   The scalar flux in each of the Nx cells
            X:             The value of the cell centers in the mesh
            """
            dx = thickness/Nx
            X = np.linspace(dx*0.5, thickness - 0.5*dx,Nx)
            scalar_flux = np.zeros(Nx)
            Sig_t = Sig_a + Sig_s
            leak_left = 0.0
            leak_right = 0
            N = int(N)
            for i in range(N):
                if (isotropic):
                    mu = np.random.uniform(-1,1,1)
                else:
                    mu = 1.0
                x = np.random.random(1)*thickness
                alive = 1
                weight = Q*thickness/N
                while (alive):
                    if (implicit_capture):
                        #get distance to collision
```

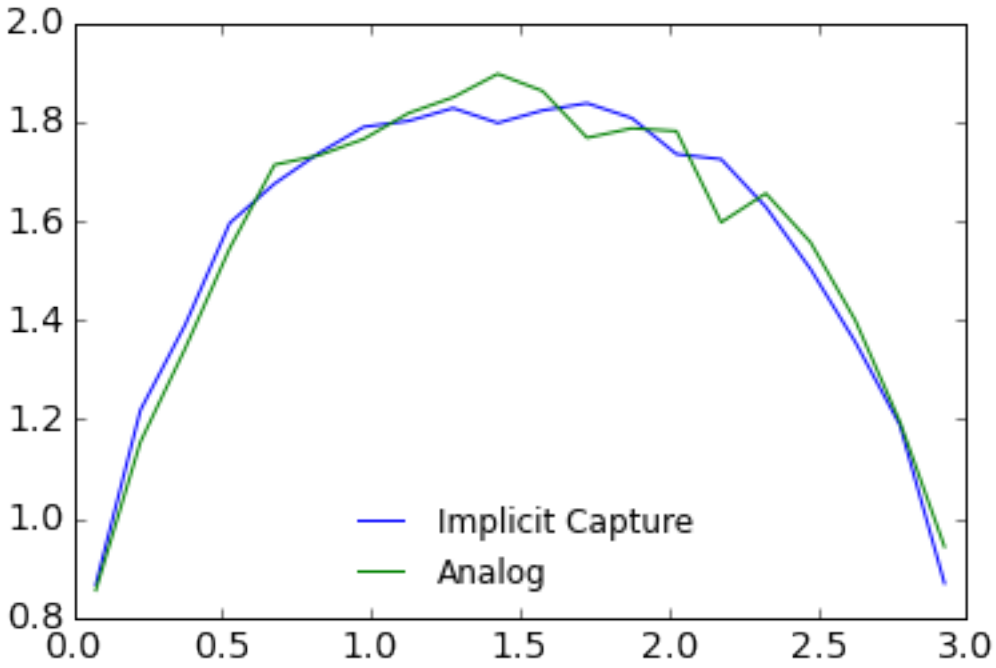
```
                    if (Sig_s > 0):
                        l = -np.log(1-np.random.random(1))/Sig_s
                    else:
                        l = 10.0*thickness/mu #something that will make it through
                else:
                    #get distance to collision
                    l = -np.log(1-np.random.random(1))/Sig_t
                #make sure that l is not too large
                if (mu > 0):
                    l = np.min([l,(3-x)/mu])
                else:
                    l = np.min([l,-x/mu])
                #move particle
                x += l*mu
                if (implicit_capture):
                    if not(l>=0):
                        print(l,x,mu)
                    assert(l>=0)
                    weight_old = weight
                    weight *= np.exp(-l*Sig_a)
                #still in the slab?
                if (np.abs(x-thickness) < 1.0e-14):
                    leak_right += weight
                    alive = 0
                elif (x<= 1.0e-14):
                    alive = 0
                    leak_left += weight
                else:
                    #compute cell particle collision is in
                    cell= np.argmin(np.abs(X-x))
                    if (implicit_capture):
                        mu = np.random.uniform(-1,1,1)
                        scalar_flux[cell] += weight/Sig_s/dx
                    else:
                        #scatter or absorb
                        scalar_flux[cell] += weight/Sig_t/dx
                        if (np.random.random(1) < Sig_s/Sig_t):
                            #scatter, pick new mu
                            mu = np.random.uniform(-1,1,1)
                        else: #absorbed
                            alive = 0
        return leak_left,leak_right, scalar_flux, X


In [7]: N = 10000
        Nx = 100
        Sigma_s = 2.0
        Sigma_a = 0.5
        Q = 1
        leak_left,leak_right, scalar_flux, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                          thickness, N,Q,
                                                          isotropic=True, implicit_capture=True)
        plt.plot(X,scalar_flux,label='Implicit Capture')
        leak_left,leak_right, scalar_flux, X= slab_source(Nx,Sigma_s,Sigma_a,
```

```
                                                    thickness, N,Q, isotropic=True,
                                                    implicit_capture=False)
        plt.plot(X,scalar_flux,label='Analog')
        plt.legend(loc="best")
        plt.show()
```



If we use fewer mesh cells, the answer looks better.

```
In [8]: N = 10000
        Nx = 20
        Sigma_s = 2.0
        Sigma_a = 0.5
        Q = 1
        leak_left,leak_right, scalar_flux, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                    thickness, N,Q, isotropic=True,
                                                    implicit_capture=True)
        plt.plot(X,scalar_flux,label='Implicit Capture')
        leak_left,leak_right, scalar_flux, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                    thickness, N,Q,
                                                    isotropic=True, implicit_capture=False)
        plt.plot(X,scalar_flux,label='Analog')
        plt.legend(loc="best")
        plt.show()
```



It is hard to tell which of the two methods (implicit capture or analog) is doing better. The figure of merit can help.

```
In [9]: N = 10000
        Nx = 20
        Sigma_s = 2.0
        Sigma_a = 0.5
        solution_implicit = np.zeros(10)
        solution_analog = np.zeros(10)
        times_implicit = 0.0
        times_analog = 0.0
        for replicate in range(10):
            tmp = time.clock()
            leak_left,leak_right, scalar_flux, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                              thickness, N,Q,
                                                              isotropic=True, implicit_capture=True)
            solution_implicit[replicate] = scalar_flux[Nx/2]
            times_implicit += time.clock()-tmp
            tmp = time.clock()
            leak_left,leak_right, scalar_flux, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                              thickness, N,Q,
                                                              isotropic=True, implicit_capture=False)
            solution_analog[replicate] = scalar_flux[Nx/2]
            times_analog += time.clock()-tmp
        var_implicit = np.std(solution_implicit)**2
        var_analog = np.std(solution_analog)**2
        print("Analog FOM =",1.0/var_analog/times_analog)
        print("Implicit Capture FOM =",1.0/var_implicit/times_implicit)
```

```
Analog FOM = 12.4525667248
Implicit Capture FOM = 3.23671833969
```

12.2 Track-Length Estimators

Another type of estimator for the scalar flux uses that definition of the scalar flux to estimate it. Recall that the scalar flux is the rate-density at which neutrons generate track length. Therefore, for a given region, every time a neutron moves inside it we sum the weight of the neutron times its path length in the region. We then divide this by the volume of the region. We can write this in equation form as

$$\bar{\phi} = \frac{1}{V} \sum_{\text{neutrons}} \text{weight} \times \text{path length}.$$

For implicit capture, the weight is changing while the neutron moves in the region. Therefore, we integrate the weight over the track to decide the contribution. For a neutron traveling a distance s inside a region will contribute

$$\text{contribution} = \int_0^s ds' w_0 e^{-\Sigma_a s'} = \frac{1}{\Sigma_a} w_0 (1 - e^{-\Sigma_a s}),$$

where w_0 is the initial weight.

To implement this estimator we will reformulate how we do our tracking. We will make our method work by checking the distance to collision against the distance to the edge of a region. Whichever distance is shorter, that event occurs: either the neutron has a collision or it moves to the next region and a new distance to collision is sampled. This means that the neutrons will step through the problem region by region.

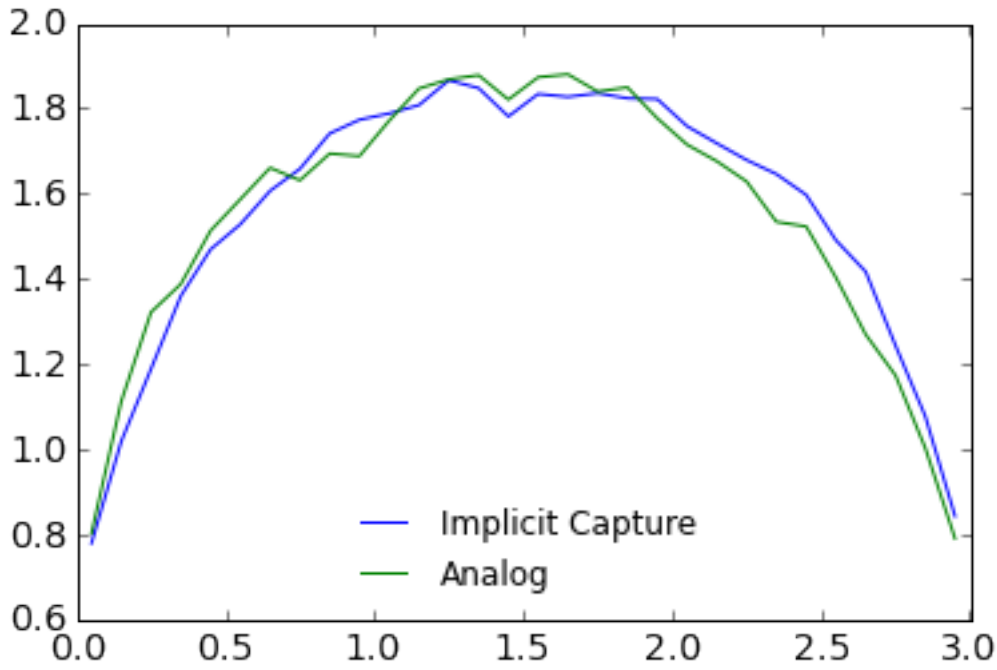
The resulting code for our slab problem is given below.

```
In [10]: import math
         def slab_source(Nx,Sig_s,Sig_a,thickness,N,Q,isotropic=False, implicit_capture = True):
             """Compute the fraction of neutrons that leak through a slab
             Inputs:
             Nx:        The number of grid points
             Sig_s:     The scattering macroscopic x-section
             Sig_a:     The absorption macroscopic x-section
             thickness: Width of the slab
             N:         Number of neutrons to simulate
             isotropic: Are the neutrons isotropic or a beam

             Returns:
             transmission:  The fraction of neutrons that made it through
             scalar_flux:   The scalar flux in each of the Nx cells
             scalar_flux_tl:   The scalar flux in each of the Nx cells from track length estimator
             X:             The value of the cell centers in the mesh
             """
             dx = thickness/Nx
             X = np.linspace(dx*0.5, thickness - 0.5*dx,Nx)
             scalar_flux = np.zeros(Nx)
             scalar_flux_tl = np.zeros(Nx)
             Sig_t = Sig_a + Sig_s
             leak_left = 0.0
             leak_right = 0
             N = int(N)
             for i in range(N):
                 if (isotropic):
                     mu = np.random.uniform(-1,1,1)
                 else:
                     mu = 1.0
                 x = np.random.random(1)*thickness
                 alive = 1
```

```
weight = Q*thickness/N
#which cell am I in
cell = np.argmin(np.abs(X-x))
while (alive):
    if (implicit_capture):
        #get distance to collision
        if (Sig_s > 0):
            l = -np.log(1-np.random.random(1))/Sig_s
        else:
            l = 10.0*thickness/np.abs(mu) #something that will make it through
    else:
        #get distance to collision
        l = -np.log(1-np.random.random(1))/Sig_t
    #compare distance to collision to distance to cell edge
    distance_to_edge = ((mu > 0.0)*( (cell+1)*dx - x) +
                        (mu<0.0)*( x - cell*dx) + 1.0e-8)/np.abs(mu)
    if (distance_to_edge < l):
        l = distance_to_edge
        collide = 0
    else:
        collide = 1
    #move particle
    x += l*mu
    #score track length tally
    if (implicit_capture):
        scalar_flux_tl[cell] += weight*(1.0 - np.exp(-l*Sig_a))/(Sig_a + 1.0e-14)
    else:
        scalar_flux_tl[cell] += weight*l
    if (implicit_capture):
        if not(l>=0):
            print(l,x,mu,cell,distance_to_edge)
        assert(l>=0)
        weight_old = weight
        weight *= np.exp(-l*Sig_a)
    #still in the slab?
    if (np.abs(x-thickness) < 1.0e-14) or (x > thickness):
        leak_right += weight
        alive = 0
    elif (x<= 1.0e-14):
        alive = 0
        leak_left += weight
    else:
        #compute cell particle collision is in
        cell= np.argmin(np.abs(X-x))
        if (implicit_capture):
            if (collide):
                mu = np.random.uniform(-1,1,1)
            scalar_flux[cell] += weight/Sig_s/dx
        else:
            #scatter or absorb
            scalar_flux[cell] += weight/Sig_t/dx
            if (collide) and (np.random.random(1) < Sig_s/Sig_t):
                #scatter, pick new mu
                mu = np.random.uniform(-1,1,1)
```

```
                elif (collide): #absorbed
                    alive = 0
            #print(x,mu,alive,l*mu,weight*l)
        return leak_left,leak_right, scalar_flux, scalar_flux_tl/dx, X
```

In [11]:
```
N = 10000
Nx = 30
Sigma_s = 2.0
Sigma_a = 0.5
thickness = 3
Q = 1
leak_left,leak_right, scalar_flux, scalar_flux_tl, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                                  thickness, N,Q,
                                                                  isotropic=True, implicit_capture=True)
plt.plot(X,scalar_flux_tl,label='Implicit Capture')
leak_left,leak_right, scalar_flux, scalar_flux_tl, X= slab_source(Nx,Sigma_s,Sigma_a,
                                                                  thickness, N,Q,
                                                                  isotropic=True, implicit_capture=False)
plt.plot(X,scalar_flux_tl,label='Analog')
plt.legend(loc="best")
plt.show()
```



In [12]:
```
N = 10000
Nx = 30
Sigma_s = 2.0
Sigma_a = 0.5
solution_implicit = np.zeros(10)
solution_analog = np.zeros(10)
```

```
        times_implicit = 0.0
        times_analog = 0.0
        for replicate in range(10):
            tmp = time.clock()
            leak_left,leak_right, scalar_flux, scalar_flux_tl, X= slab_source(Nx,
                                                                              Sigma_s,Sigma_a,
                                                                              thickness, N,Q,
                                                                              isotropic=True, implicit_capture=Tru
            solution_implicit[replicate] = scalar_flux_tl[Nx/2]
            times_implicit += time.clock()-tmp
            tmp = time.clock()
            leak_left,leak_right, scalar_flux, scalar_flux_tl, X= slab_source(Nx,
                                                                              Sigma_s,Sigma_a,
                                                                              thickness, N,Q,
                                                                              isotropic=True, implicit_capture=Fal
            solution_analog[replicate] = scalar_flux_tl[Nx/2]
            times_analog += time.clock()-tmp
        var_implicit = np.std(solution_implicit)**2
        var_analog = np.std(solution_analog)**2
        print("Analog FOM =",1.0/var_analog/times_analog)
        print("Implicit Capture FOM =",1.0/var_implicit/times_implicit)
```

```
Analog FOM = 4.62265989803
Implicit Capture FOM = 1.49589589295
```

12.3 Stratified Sampling

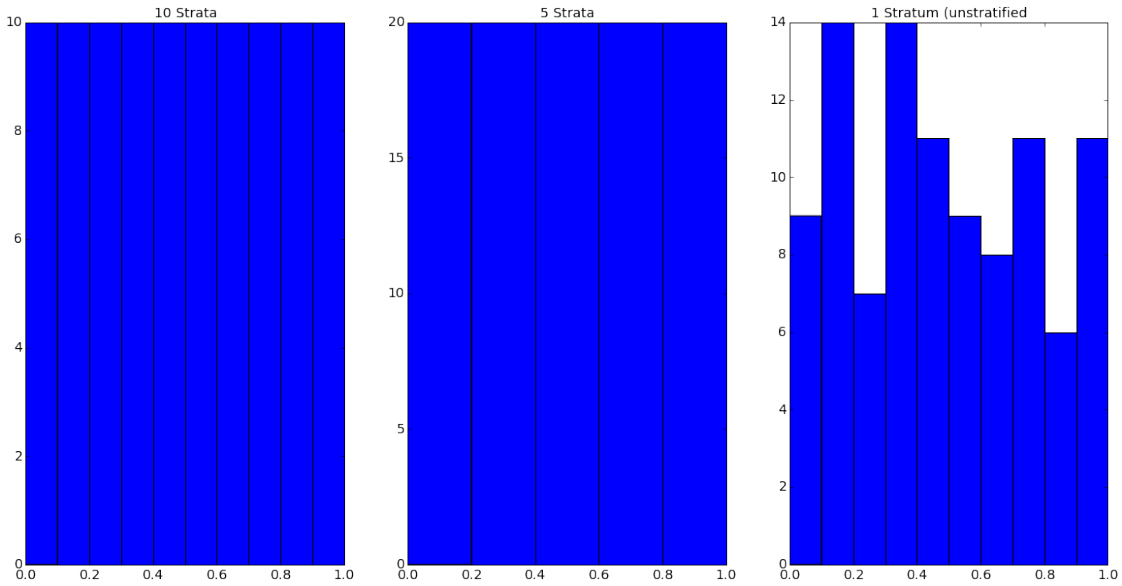
The idea behind stratified sampling is to control the randomness in the simulation. We want to use random numbers to simulate neutron interactions, but there is no guarantee that random numbers will not be close together. Stratified sampling is a way to spread out the numbers.

It is easiest to think about stratification in terms of a single random variable uniformly distributed between 0 and 1. There are several possible formulations, but the most straightforward to use divides the range between 0 and 1 into S bins of equal size. We then pick a bin at random by generating a random integer between 0 and $S-1$. Then inside that bin we randomly pick a location. We can even do this so that the same bin is not picked again until all the others have a sample inside them (this is called sampling without replacement).

For this simple example, the code to produce the stratified samples is straightforward.

```
In [16]: def strat_sample(N,S):
             N = N + (N % S)
             assert(N%S == 0 )
             dS = 1.0/S
             bins = np.zeros(N,dtype=int)
             count = 0
             for i in range(N//S):
                 bins[count:count+S] = np.random.permutation(S)
                 count += S
             place_in_bin = np.random.uniform(-0.5*dS,0.5*dS,N) + (bins+0.5)*dS
             return place_in_bin
         plt.figure(figsize=(20,10))
         plt.subplot(1, 3, 1)
         plt.hist(strat_sample(100,10),bins=10,range=(0,1))
         plt.title("10 Strata")
         plt.subplot(1,3,2)
```

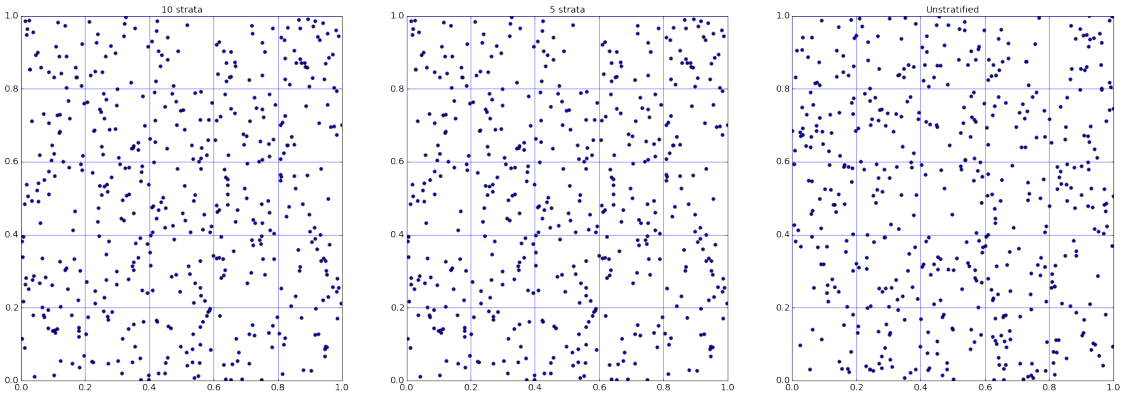
```
plt.hist(strat_sample(100,5),bins=5,range=(0,1))
plt.title("5 Strata")
plt.subplot(1,3,3)
plt.hist(np.random.random(100),range=(0,1))
plt.title("1 Stratum (unstratified")
plt.show()
```

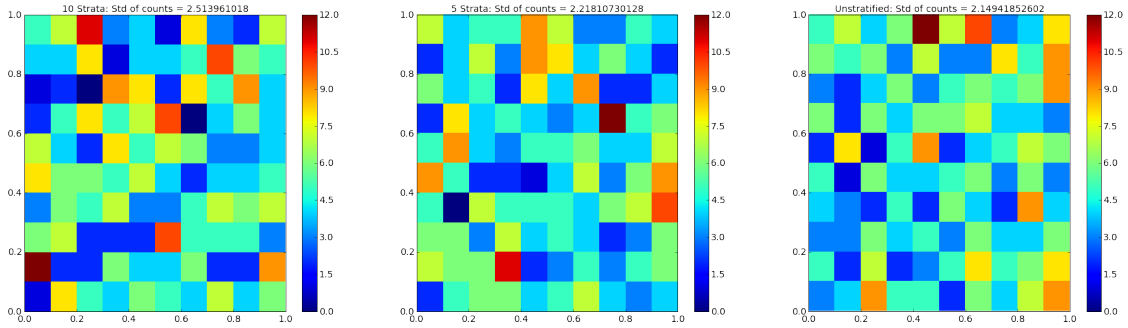


If we pick the bins randomly, we can use this stratified sampling algorithm for two (or more) uncorrelated random variables. It will not guarantee perfect stratification, but it will be better than simple random sampling. Here is an algorithm for this.

```
In [17]: def strat_sample_ndim(D,N,S):
             samples = np.zeros((N,D))
             for d in range(D):
                 samples[:,d] = strat_sample(N,S)
             return samples
         plt.figure(figsize=(30,10))
         plt.subplot(1, 3, 1)
         xy = strat_sample_ndim(2,500,10)
         plt.scatter(xy[:,0], xy[:,1])
         plt.axis([0,1,0,1])
         plt.grid(b=True, which='major', color='b', linestyle='-')
         plt.title("10 strata")
         plt.axis([0,1,0,1])
         plt.subplot(1, 3, 2)
         plt.scatter(xy[:,0], xy[:,1])
         plt.axis([0,1,0,1])
         xy = strat_sample_ndim(2,500,5)
         plt.grid(b=True, which='major', color='b', linestyle='-')
         plt.title("5 strata")
         plt.subplot(1, 3, 3)
         plt.scatter(xy[:,0], xy[:,1])
         plt.axis([0,1,0,1])
```

```
xy = strat_sample_ndim(2,500,1)
plt.grid(b=True, which='major', color='b', linestyle='-')
plt.title("Unstratified")
plt.show()


plt.figure(figsize=(30,8))
plt.subplot(1, 3, 1)
xy = strat_sample_ndim(2,500,10)
counts, xedges, yedges, Image = plt.hist2d(xy[:,0], xy[:,1],bins=10)
plt.title("10 Strata: Std of counts = "+ str(np.std(counts)))
plt.axis([0,1,0,1])
plt.colorbar()
plt.clim(0,12)
plt.subplot(1, 3, 2)
xy = strat_sample_ndim(2,500,5)
counts, xedges, yedges, Image = plt.hist2d(xy[:,0], xy[:,1],bins=10)
plt.title("5 Strata: Std of counts = "+ str(np.std(counts)))
plt.axis([0,1,0,1])
plt.clim(0,12)
plt.colorbar()
plt.subplot(1, 3, 3)
xy = strat_sample_ndim(2,500,1)
counts, xedges, yedges, Image = plt.hist2d(xy[:,0], xy[:,1],bins=10)
plt.title("Unstratified: Std of counts = "+ str(np.std(counts)))
plt.axis([0,1,0,1])
plt.clim(0,12)
plt.colorbar()
plt.show()
```





Notice that the standard deviation went down as the number of strata increased: this means that the number of particles in each histogram bin. We could use, for example, stratified sampling to make sure that particles are spread out in space, angle, and energy. This will decrease the variance in the calculation.

12.4 Russian Roulette

On aspect of Monte Carlo is the we can spend time tracking particles that are not important. This could happen when using implicit capture and the neutron weight gets very low. It could also happen that the neutron moves to a region of the problem that is not important to us. To do this we can probabilitistically terminate a neutron when its weight is below some threshold.

Let's call this threshold, w_L. We will also pick a survival weight w_A. When a neutron exits a collision or crosses an interface and has a weight less than the threshold, $w < w_L$. We define a probability of being killed as

$$p_k = 1 - \frac{w}{w_A}.$$

To determine if the neutron survives, we select a random number between 0 and 1. If that number is less than, p_k the neturon is killed. If it survives, it then is assined weight w_A.

To demonstrate that we have not committed any statistical violence in our roulette game, let's look at N particles undergoing roulette. The total weight of the particles before roulette is

$$\sum_{i=1}^{N} w_i.$$

After roulette the total weight of particles remaining, on average, will be $1 - p_{ki}$ times w_A for each particle. This means that the total final weight will be, on average,

$$\sum_{i=1}^{N}(1 - p_{ki})w_A = \sum_{i=1}^{N} \frac{w_i}{w_A} w_A = \sum_{i=1}^{N} w_i.$$

Therefore, on average, the total weight before and after russian roulette will be preserved. Russian roulette will allow our calculations to go faster if we pick the values of w_L and w_A appropriately to terminate the tracking of unimportant neutrons. The definition of unimportant will vary from problem to problem and could even vary within a problem. Also, note that the ratio w_L/w_A will control the number of neutrons that survive: if w_L/w_A is small, then very few neutrons will survive roulette because p_k will be close to 1. Killing many neutrons will make the calculation faster but could increase noise. On the other hand, if w_L/w_A is too large, not enough neutrons will be killed and performance will not be improved.

12.5 Splitting

Splitting is the opposite of roulette in that important neutrons are each split into several neutrons. There are two reasons for this. The first is that if each neutron in the problem has the same weight, the overall variance in the problem will be

decreased. Furthermore, in a region of the problem where few neutrons reach, we may desire to make more neutrons when a neutrons enters that part of the problem. For example, consider a source next to a shield. Very few neutrons will penetrate the shield (if it is a good shield). If we are interested in what happens to neutrons after they leave the shield, it may be beneficial to split neutrons as they move deeper into the shield.

To split neutrons, we consider a desired neutron weight, w_D. We also define a threshold, w_H, above which a neutron will be split. If a neutron leaves a collision or enters a region with $w > w_H$. We would then split the neutron into w/w_D rounded to the nearest integer particles, each with the weight

$$w_{\text{split}} = \frac{w}{\text{round}\left(\frac{w}{w_D}\right)}.$$

The splitting of particles will increase runtime because we are adding more work. However, when combined with russian roulette, it is possible to focus the work to regions of the problem that are important to the desired quantities we are estimating.

12.6 Weight Windows

A weight window is a range of weights where neutron tracking is done as normal. When neutrons fall below the bottom of the window (w_L), they are subject to russian roulette, and when neutrons have a weight above the top of the window (w_A) the neutrons are split. Typically, the upper and lower cutoffs are chosen so that the after particles are split they will be close to w_D and when a neutron wins roulette, it is promoted to have weight w_D (this is done by making $w_A = w_D$). The ideal neutron weight is sometimes called the weight window center.

The ideal choice of weight windows is not straightforward because it depends on the relative importance of neutrons to score in the desired region of interest. In problems where the solution in localized region is of interest, the weight window center, w_D can be picked to be proportional to the inverse of the adjoint scalar flux:

$$w_D = \frac{C}{\phi^\dagger},$$

where $\phi^\dagger$ is the adjoint scalar flux. The adjoint scalar flux is a measure of the importance of a neutron to the response. Notice that if the adjoint scalar flux goes to zero, then the weight window center goes to infinity and all particles in that region are subject to roulette.

In problems where we care about the solution everywhere, the approach for weight windows is slightly different. In this case we would divide our problem up into regions where we want to know the scalar flux. Then in each region we set the weight window center to be proportional to the average scalar flux in that region:

$$w_D = C\phi.$$

Of course, if we knew the scalar flux we would not have to solve the problem via Monte Carlo. Nevertheless, if we have an estimate of the scalar flux, perhaps from diffusion, we can use this as the weight window center.

In [17]: