NE250 Homework 6 **Mitch Negus**
12/1/2017

Problem 1

a.)

Expected Value/Mean:

$$\begin{aligned}\mu &= \int_{-\infty}^{\infty} x f(x)\, dx \\ &= \int_0^a \frac{x}{a}\, dx \\ &= \frac{1}{2a}\left[x^2\right]_0^a\end{aligned}$$

$$\boxed{\mu = \frac{a}{2}}$$

Variance:

$$\begin{aligned}\sigma^2 &= E[(X-\mu)^2] \\ &= \int_{-\infty}^{\infty} (x-\mu)^2 f(x)\, dx \\ &= \int_0^a \left(x - \frac{a}{2}\right)^2 \left(\frac{1}{a}\right) dx \\ &= \int_0^a \left(x^2 - ax + \frac{a^2}{4}\right)\left(\frac{1}{a}\right) dx \\ &= \int_0^a \frac{x^2}{a}\, dx - \int_0^a x\, dx + \int_0^a \frac{a}{4}\, dx \\ &= \left[\frac{x^3}{3a}\right]_0^a - \left[\frac{x^2}{2}\right]_0^a + \left[\frac{ax}{4}\right]_0^a \\ &= \frac{a^2}{3} - \frac{a^2}{2} + \frac{a^2}{4}\end{aligned}$$

$$\boxed{\sigma^2 = \frac{a^2}{12}}$$

Cumulative Distribution Function:

$$\begin{aligned}F(x) &= \int_{-\infty}^{x} f(x')dx' \\ &= \int_0^x \frac{1}{a}\, dx' \\ &= \frac{1}{a}\left[x'\right]_0^x \\ &= \frac{x}{a}\end{aligned}$$

$$\boxed{F(x) = \begin{cases} 0, & x < 0, x > a \\ \frac{x}{a}, & 0 < x < a \end{cases}}$$

b.)

Expected Value/Mean:

$$\begin{aligned}
\mu &= \int_{-\infty}^{\infty} x f(x)\,dx \\
&= \int_{0}^{\infty} \lambda x e^{-\lambda x}\,dx \\
(\text{let } u = \lambda x, du = \lambda\,dx, dv = e^{-\lambda x}, v = -e^{-\lambda x}/\lambda) \\
&= uv|_0^\infty - \int_0^\infty v\,du \\
&= -xe^{-\lambda x}\Big|_0^\infty - \int_0^\infty \frac{-\lambda e^{-\lambda x}}{\lambda}\,dx \\
&= 0 + \left[\frac{-e^{-\lambda x}}{\lambda}\right]_0^\infty
\end{aligned}$$

$$\boxed{\mu = \frac{1}{\lambda}}$$

Variance:

$$\begin{aligned}
\sigma^2 &= E[(X-\mu)^2] \\
&= E[X^2] - \mu^2 \\
&= \int_0^\infty x^2 f(x)\,dx - \frac{1}{\lambda^2} \\
&= \int_0^\infty \lambda x^2 e^{-\lambda x}\,dx - \frac{1}{\lambda^2} \\
(\text{let } u = \lambda x^2, du = 2\lambda x\,dx, dv = e^{-\lambda x}, v = -e^{-\lambda x}/\lambda) \\
&= uv|_0^\infty - \int_0^\infty v\,du - \frac{1}{\lambda^2} \\
&= \left[-x^2 e^{-\lambda x})\right]_0^\infty + \int_0^\infty 2xe^{-\lambda x}\,dx - \frac{1}{\lambda^2} \\
&= 0 + 2\int_0^\infty xe^{-\lambda x}\,dx - \frac{1}{\lambda^2} \\
&= \frac{2}{\lambda^2}\left[e^{-\lambda x}(-\lambda x - 1)\right]_0^\infty - \frac{1}{\lambda^2} \\
&= \frac{2}{\lambda^2} - \frac{1}{\lambda^2}
\end{aligned}$$

$$\boxed{\sigma^2 = \frac{1}{\lambda^2}}$$

Cumulative Distribution Function:

$$\begin{aligned}
F(x) &= \int_{-\infty}^{x} f(x')dx' \\
&= \int_0^x \lambda e^{-\lambda x'}\,dx' \\
&= \left[-e^{-\lambda x'}\right]_0^x \\
&= \left[1 - e^{-\lambda x}\right]
\end{aligned}$$

$$\boxed{F(x) = 1 - e^{-\lambda x}}$$

Problem 2

(See Jupyter notebook, attached)

Problem 3

a.)

True. A normalized PDF is required so that the inverted CDF, from which we draw our sample, has a domain of 0 to 1. Then, selecting random numbers in this same interval will give us values of the PDF/CDF's independent variable distributed like the original PDF.

b.)

Deterministic Strengths

1. Fast execution (all steps contribute to a meaningful solution)
2. No statistical uncertainties, solutions are exact (within truncation limits)
3. Solutions are global over the problem space

Monte Carlo Weaknesses

1. Slow/inefficient (calculations for particles that do not contribute to the tally are essentially wasted)
2. Results contain statistical uncertainties
3. Solutions are only local (wherever tallies are collected)

Monte Carlo Strengths

1. Continuous (in space, energy, angle, etc.)
2. Easy to parallelize
3. No truncation error, solutions are exact (within uncertainties)

Deterministic Weaknesses

1. Difficult to parallelize
2. Discretization (energy, angle, space, etc.) limits accuracy
3. Solutions contain truncation error (expansions are truncated)

c.)

The relative error of an analog Monte Carlo algorithm is $R = S_{\bar{x}}/\bar{x}$. Since $S_{\bar{x}} = S/\sqrt{N}$, it becomes apparent that $R \propto 1/\sqrt{N}$.

d.)

The central limit theorem allows us to state, specifically for independently and identically distributed random variables, that for many samples of N measurements, the sample means will be distributed according to a normal distribution, with variance σ^2/N (where σ^2 is the true variance of the original variable distribution).

e.)

In a Monte Carlo algorithm, the tracking of particles through a geometry region requires the particle's mean free path in that region. At a boundary, the mean free path must be updated for proper transport through the new material.

f.)

The collision estimator is more likely to be accurate in cases where there are many collisions (such as a large and/or dense geometry region), and the majority of particles passing through the region of interest collide there. In the case that the geometry is optically thin, there will be few collisions in the region, and so a collision estimator will yield poor statistics. Then, a track length estimator will be preferable.

g.)

The Consistent Adjoint Driven Importance Sampling method is termed "consistent" because it applies the appropriate weighting (to the nominal weight) for every particle that is born, according to the importance map defined by the adjoint source.

Problem 4

(See Jupyter notebook, attached)

NE250_HW06_mnegus-prob2

December 1, 2017

1 NE 250 – Homework 6

1.1 Problem 2

12/1/2017

```
In [1]: %matplotlib inline
        import numpy as np
        import seaborn as sns
        from matplotlib import pyplot as plt
```

a.)

To sample from a distribution with an exponentially decaying probability density function

$$p(x) = e^{-x},$$

we must first determine the cumulative distribution function (CDF). This is

$$P(x) = \int_0^x e^{-x'} dx'$$
$$P(x) = \left[-e^{-x'}\right]_0^x$$
$$P(x) = 1 - e^{-x}$$

If we take a random sample from this CDF, ξ, then we can invert the function to find a random x corresponding to ξ.

$$\xi = 1 - e^{-x}$$
$$x = -\ln(1 - \xi)$$

Since ξ is chosen uniformly and ranges from 0 to 1, we can say that $1 - \xi$ is a random number with identical distribution to ξ. Then

$$x = -\ln \xi$$

```
In [2]: def sample_exp_decay(num_samples):
            '''Randomly sample from an exponential distribution, and return the sa
            xi = np.random.random(num_samples)
            x = -np.log(xi)
            return x,x.mean(),x.var()
```

Now, we want to estimate the mean and variance of samples of size 10, 40, and 160.

```
In [3]: for num_samples in [10,40,160,10_000]:
            x,mu,sig = sample_exp_decay(num_samples)
            mu = round(mu,4)
            sig = round(sig,4)
            print('{} samples \t  Mean: {} \t Variance: {} '.format(num_samples,mu,
```

```
10 samples          Mean: 1.1785          Variance: 1.1298
40 samples          Mean: 1.0652          Variance: 1.2882
160 samples          Mean: 1.1874          Variance: 1.2137
10000 samples          Mean: 1.0046          Variance: 0.9755
```

We can see that the values for the required 10, 40, and 160 are reasonably close to the values predicted analytically (We can use Problem 1 part b, with $\lambda = 1$ to see that the mean and variance of the distribution should both equal 1). Our precision increases considerably with more histories, and with 10,000 histories, we are significantly more accurate on both mean and variance.

b.)

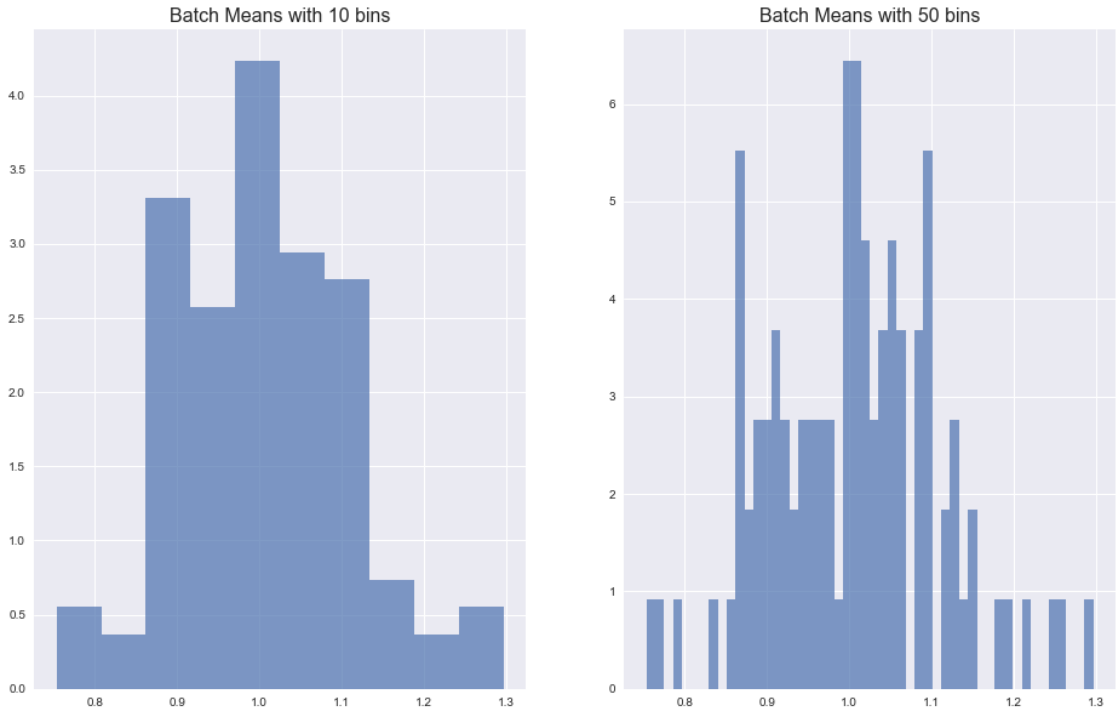
Next, we will run several batches of runs, each with a given number of histories. To show that the batch average tends to follow the central limit theorem when several batches are considered, we will plot the averages as a histogram

```
In [4]: def batch_sampling(num_batches,num_samples):
            '''Run a given number of batches, each with the given number of samples
            batch_means = [sample_exp_decay(num_samples)[1] for i in range(num_batc
            return batch_means
```

```
In [5]: def plot_hists(data,title,overlay=False,var=1,n=100):
            '''Plot a histogram with from a given dataset.'''
            binnings = 2
            fig, ax = plt.subplots(ncols=binnings,figsize=(16,10))
            bin_count = [10+40*i for i in range(binnings)]
            data=np.array(data)
            for b in range(binnings):
                ax[b].hist(data,bins=bin_count[b],normed=True,alpha=0.7)
                ax[b].set_title('{} with {} bins'.format(title,bin_count[b]),fontsi
                # Add Gaussian overlay (with mean 1 and variance 1)
                if overlay:
                    x = np.arange(0.5,1.5,0.01)
                    g = np.exp(-(x-1)**2/(2*var/n))/np.sqrt(2*np.pi*var/n)
                    ax[b].plot(x,g,'k')
            return ax
```

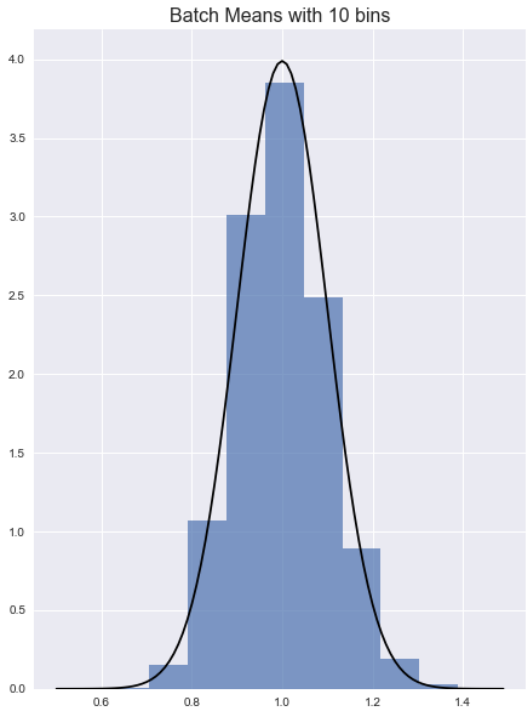
We will first run 100 batches of the sampling algorithm, each batch with 100 histories.

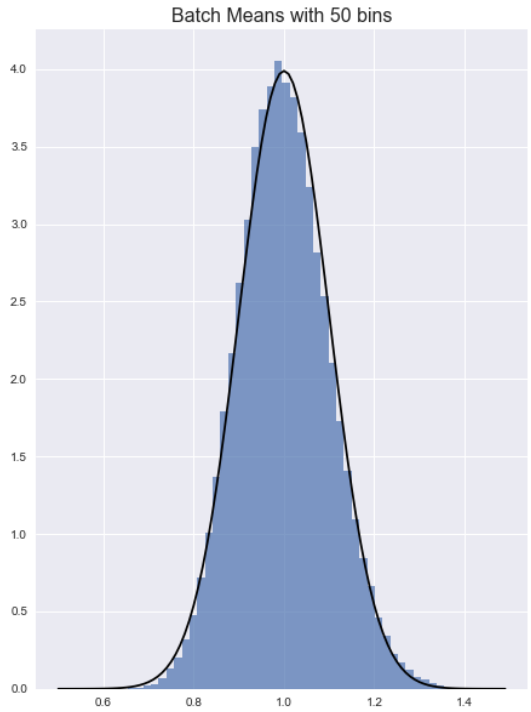
```
In [6]: batch_means = batch_sampling(100,100)
        mean_plots = plot_hists(batch_means,'Batch Means')
```



The Gaussian behavior is fairly clear in these graphics, but we can make it more apparent if we increase the number of batches. Let's try 100,000 batches instead, but each with still 100 histories. We can overlay the predicted Gaussian distribution (with mean $\mu = 1$ and variance $\frac{\sigma^2}{n} = 0.01$) given by the central limit theorem on the plots.

```
In [7]: batch_means = batch_sampling(100_000,100)
        mean_plots = plot_hists(batch_means,'Batch Means',overlay=True)
```


Batch Means with 10 bins
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.6
0.8
1.0
1.2
1.4


Batch Means with 50 bins
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.6
0.8
1.0
1.2
1.4

NE250_HW06_mnegus-prob4

December 1, 2017

1 NE 250 – Homework 6

1.1 Problem 4

12/1/2017

```
In [1]: import numpy as np
```

We are considering an infinite, steady-state, monoenergetic, two-region Monte Carlo problem with the following characteristics: (note thate we have renamed region 1 and region 2 as region 0 and region 1 respectively; this allows for simpler calculations) * 1-D problem geometry * Region 0 has $\Sigma_s = 0.5\,\text{cm}^{-1}$ and $\Sigma_t = 1.0\,\text{cm}^{-1}$ (in both regions the only interactions are scattering or absorption). * Region 1 has $\Sigma_s = 0.75\,\text{cm}^{-1}$ and $\Sigma_t = 0.9\,\text{cm}^{-1}$. * Region 0 has $w_{nom} = 1$ and Region 1 has $w_{nom} = 2$. w_{max} and w_{min} can be found as $(w_{nom} \times 2.5)$ or $(w_{nom} \,/\, 2.5)$, respectively. * All particles are born in Region 0 with weight 1 at a location that is 1 cm to the left of the interface between Regions 0 and 1. The source is isotropic. * Isotropic scattering.

Problem Statement *Write the algorithm for a Monte Carlo code to solve this specific problem. Include the PDFs required for sampling as well as algorithms for conducting sampling. Use a collision estimator to tally the flux. Include implicit capture, rouletting, and splitting.*

1.1.1 Underlying parameters

Using the above information, we can create a set of dictionaries to describe our physical situation. Each fundamental parameter gets a dictionary, and each dictionary has entries corresponding to each region.

```
In [2]: # Macroscopic Total Cross Sections
        Sigma_t = {0:1.0,1:0.9}

        # Macroscopic Scattering Cross Sections
        Sigma_s = {0:0.5,1:0.75}

        # Nominal Weights
        w_nom = {0:1,1:2}

        # Max/Min Weights
        w_ext = {region:(w_nom[region]*2.5,w_nom[region]/2.5) for region in w_nom.k
```

```
In [3]: w_ext
```

```
Out[3]: {0: (2.5, 0.4), 1: (5.0, 0.8)}
```

1.1.2 Tracking the particle

We can track a particle using a particle class, which can model the behavior of each particle as it proceeds through it's lifetime. This particle class will contain methods for each action that the particle will undergo * birth (the class' _init__ method) * transport * boundary encounter * collision * scoring

```
In [4]: class particle:
            """
            A class to model a single Monte Carlo particle over it's lifetime
            """
            def __init__(self):
                """The particle is born. Assign a position [cm], angle (cos θ), ene
                self.x = -1
                self.mu = 2*np.random.random()-1
                self.E = 1
                self.w = 1
                self.region = 0
                self.score = np.zeros(2)

            def transport(self,sample=True):
                """Transport the particle through the problem geometry"""
                # Sample to find the number of mean free paths that traveled by the
                if sample:
                    xi = np.random.random()
                    self.mfp_x = -np.log(xi)*self.mu
                # Determine the number of mean free paths to a boundary in the curr
                boundary_mfp = -self.x*Sigma_t[self.region]
                # If the particle reaches a boundary before the collision, stop and
                if boundary_mfp == 0:  # (the particle was at the boundary, so must
                    self.collision()
                elif self.mfp_x > boundary_mfp:
                    self.boundary(boundary_mfp)
                else:
                    self.collision()

            def boundary(self,boundary_mfp):
                """The particle reached a boundary: reevaluate the particle's trach
                self.x = 0
                self.region = 1-self.region
                self.mfp_x -= boundary_mfp
                self.transport(sample=False)

            def collision(self):
```

```
            """The particle collided: score and determine the collision type"""
            self.score_particle()
            xi = np.random.random()
            if xi*Sigma_t[self.region] > Sigma_s[self.region]: # particle scatt
                self.x += self.mfp_x
                self.mu = 2*np.random.random()-1
                self.transport(sample=True)

        def score_particle(self):
            self.score[self.region] += self.w
```

```
In [5]: ## Analog MC run
        N = 20000
        tally = np.zeros(2)
        for i in range(N):
            n = particle()
            n.transport()
            tally += n.score
        rel_flux = tally/N
```

```
In [6]: print('Relative flux, region 0: {} \t Relative flux, region 1: {}'.format(r
        print('Ratio: ',rel_flux[0]/rel_flux[1])
```

```
Relative flux, region 0: 1.7293          Relative flux, region 1: 0.16225
Ratio:  10.6582434515
```

1.2 Survival Biasing

```
In [7]: class particle:
            """
            A class to model a single Monte Carlo particle over it's lifetime
            """
            def __init__(self):
                """The particle is born. Assign a position [cm], angle (cos θ), ene
                self.x = -1
                self.mu = 2*np.random.random()-1
                self.E = 1
                self.w = 1
                self.region = 0
                self.score = np.zeros(2)

            def transport(self,sample=True):
                """Transport the particle through the problem geometry"""
                # Sample to find the number of mean free paths that traveled by the
                if sample:
                    xi = np.random.random()
                    self.mfp_x = -np.log(xi)*self.mu
```

```
            # Determine the number of mean free paths to a boundary in the curr
            boundary_mfp = -self.x*Sigma_t[self.region]
            # If the particle reaches a boundary before the collision, stop and
            if boundary_mfp == 0:  # (the particle was at the boundary, so must
                self.collision()
            elif self.mfp_x > boundary_mfp:
                self.boundary(boundary_mfp)
            else:
                self.collision()

        def boundary(self,boundary_mfp):
            """The particle reached a boundary: reevaluate the particle's trach
            self.x = 0
            self.region = 1-self.region
            self.mfp_x -= boundary_mfp
            self.transport(sample=False)

        def collision(self):
            """The particle collided: use survival biasing to continue followin
            tally_weight = self.w*(Sigma_t[self.region]-Sigma_s[self.region])/S
            self.score_particle(tally_weight)
            self.w -= tally_weight
            if self.w < 0.0001:
                return
            self.x += self.mfp_x
            self.mu = 2*np.random.random()-1
            self.transport(sample=True)

        def score_particle(self,tally_weight):
            self.score[self.region] += tally_weight
```

In [8]:
```
## Survival biasing MC run
N = 20000
tally = np.zeros(2)
for i in range(N):
    n = particle()
    n.transport()
    tally += n.score
rel_flux = tally/N
```

In [9]:
```
print('Relative flux, region 0: {} \t Relative flux, region 1: {}'.format(r
print('Ratio: ',rel_flux[0]/rel_flux[1])
```

```
Relative flux, region 0: 0.9446255195935787          Relative flux, region 1: 0.055
Ratio:  17.0807326028
```

An error most likely exists in the code since the relative fluxes ratios do not match between the analog and survival biasing runs.