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 cur
                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 track
                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 track
            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.