NERS 544 Project Report

Aaron Graham, Mike Jarrett

April 30, 2015

1 Introduction

The purpose of this project was to apply Monte Carlo simulation techniques to some semi-realistic problem. The problem selected for this purpose was eigenvalue calculations for a pin cell. The pin cell had a radius of 1.5 cm and a height of 100 cm. The pin cell had reflecting boundary conditions radially and vacuum boundary conditions axially, making this problem an eigenvalue calculation for an infinite lattice.

The eigenvalue was calculated using both track-length and collision estimators in the fuel region, as well as a "rough estimate" of the eigenvalue calculated from the number of fissions caused by each generation. Additionally, current estimators were placed on the top and bottom surfaces of the pin cell to calculate the leakage out of the pin cell. These calculations allowed observation of the behavior properties of Monte Carlo methods as applied to an infinite lattice calculate.

2 Code Description

The code used for this project was written using C++. It was broken into five distinct pieces in order to maintain organization and a moderate degree of abstraction. Each of these pieces will be described in sections 2.1-2.5.

2.1 Utils

This module contains functions which will be needed throughout the code and are not tied to any specific functionality. Some of these functions are as follows:

- *double drand(void)* – This function uses the intrinsic C function *int rand()* to return a random double precision number between 0.0 and 1.0.
- *double Watt(void)* – This function returns a random energy for a fission neutron, sampled from the Watt spectrum.
- *bool approxeq(double, double)*, *bool approxge(double, double)*, *bool approxle(double, double)* – These functions are used for $==$, $\geq$, and $\leq$ operators, respectively, for floating point double precision numbers.

In addition to these functions, the header file for Utils also defines many constants for use in the code, such as pi, the mass of a neutron, the Boltzmann constant, and others used for various parts of the code.

The header file is found in Appendix A, and the source code in Appendix B.

2.2 Geometry

The second module is the geometry module. This defines all the functions required for two main classes: *cell* and *surface*.

The *surface* class has two sub-classes: *plane* and *cylinder*. The *plane* class assumes that the plane is perpendicular to one of the three coordinate axes. This assumption could easily be generalized, but was made to simplify this problem. The plane is then defined by a point on the plane and the normal vector for the plane. The *cylinder* class assumes that the cylinder has some origin, a radius, and an axis which extends parallel to the z-axis. Thus, no rotation of the cylinder is allowed.

Both sub-classes of the *surface* class implement several important functions:

- *double distToIntersect(double*, double*)* – This function takes a position and direction as input arguments. It then calculate the distance from the position to the surface along the given direction. The distances might be negative.

- *void reflect(double*, double*)* – This function takes a position and direction. The direction is modified as if it had reflected off the surface.
- *int getSense(double*)* – This function takes a 3D point and returns 1 or -1, depending on where the point is. For a *plane* class, 1 denotes being on the "positive" side of the plane and -1 on the "negative" side (with respect to the axis that the plane is perpendicular to), while for a *cylinder* class, 1 denotes being outside the cylinder and -1 denotes being inside it.

The *cell* class is defined mostly in terms of surfaces. It contains a vector of *surface* ID numbers, *iSurfs*, and *senses*, which contains a 1 or -1 for each surface to indicate which side of the surface the cell is on. It also contains a *material* ID number, and a *distToIntersect* function, which simply calls the corresponding function on each of the *cell*'s *surface* classes and returns the minimum positive distance.

This module also contains two vectors of pointers: one for each *surface* that has been created, and the other for each *cell* that has been created. The routines *getPtr_cell(int)* and *getPtr_surface(int)* each return a pointer from one of these lists when given an ID number.

The header file is found in Appendix C, and the source file in Appendix D.

2.3 Materials

This module contains definitions for the *moderator* and *fuel* classes. While these classes technically share a superclass *material*, it ended up being simpler for them to be mostly separate.

The *moderator* class contains variables for the oxygen and hydrogen number densities and cross-section (scattering and aborption) parameters. It also has a function *void modMacro(double, double*, double*, double*)* which, given an energy, returns the total cross-section, the probability of interaction with hydrogen, and the total absorption probability. These are used to correctly move particles around and perform the interaction physics in the moderator.

The *fuel* class has similar information, but for oxygen, U-235, and U-238. It also contains fission cross-section and resonance information, which the *moderator* class does not require. This class has three main methods defined:

- *void fuelMacro(double, double*, double*, double*)* – Similar to its *moderator* counterpart, it returns the total cross-section and other information about interaction probabilities.
- *int sample_U(double, double*, double*, double*, double*)* – This function uses a random number to determine which isotope of the fuel the neutron is interacting with, and sets appropriate probabilities of absorption and fission based on that result.
- *double fissXS(double)* – This function claculates the fission cross-section of U-235 given a particular energy.

In addition to these class definitions, this module also contains a routine *void init_materials(int& , int&)* which sets up the fuel and moderator materials. It also defines a function *void elastic(const double, int, double& , double[3])*, which performs elastic scattering off a material. There is also a *getPtr_material* function which ended up not being used due to how the materials were defined.

The header file is found in Appendix E, and the source is in Appendix F.

2.4 Particles

This module contains a definition of the *particle* and *fission* classes. The *particle* class contains the position, direction, and energy of the neutron. It also contains a logical value *isAlive* to indicate when the simulation of the particle should stop, and some information about the interaction probabilities and track-length and collision estimators. It also contains a method *int simulate()*, which simulates the entire life of the particle. The return value can represent one of three possible quantities, depending on what range it falls in:

- *return* == 0 – Particle was absorbed, so nothing is done with the result.
- *return* > 0 – The particle caused fission, and the return value contains the number of fission neutrons which is produces.
- *return* < 0 – the particle escaped, and the return value contains the negative of the surface ID across which it leaked.

The *fission* class contains a position of the neutron which caused the fission, which can be used to generate a new *particle* class for the next batch of simulations. This prevents having to pass the entire particle around once its simulation is over.

There are also two important functions defined in this module:

- *void makeSource(std::vector<fission>& , std::vector<particle>& , int)* – This function takes in the fission bank which was generated by a previous iteration and populates the new source bank with neutrons. If the fission bank is too large, it randomly samples it to get the correct number. If it is too small, it randomly samples extra fission locations from the fission bank. In either case, the resulting source bank will be a constant size.
- *double calcEntropy(std::vector<fission>)* – This function takes in the fission bank and calculates the Shannon entropy for the previous iteration.

The header file is found in Appendix G, and the source in Appendix H.

2.5 Driver

The driver contains the actual iteration logic to solve the problem. The problem was solved by performing a power iteration beginning with an initial source bank of neutrons which were uniformly sampled within the fuel region. Since the neutrons are representative of fission neutrons, their direction was sampled isotropically and their energy was sampled from the Watt Spectrum. After the first iteration, the source bank was populated from the fissions occurring in the previous iteration. The full algorithm for the code is shown below:

Iteration Algorithm	
1.	Seed random number generator using current time
2.	Accept user input of pin pitch
3.	Set up materials
4.	Set up geometry
5.	*batch_size* = 10^5
6.	*for i = 1 to batch_size, do:*
7.	Randomly sample position, direction, and energy for neutron
8.	Add neutron to source bank
9.	Initialize estimators to 0
10.	*active_cycles* = 180; *inactive_cycles* = 20
11.	*for i = 1 to active_cycles + inactive_cycles, do:*
12.	*while source bank is empty, do:*
13.	Remove neutron from bank and simulate it
14.	If neutron caused fission, at fissions to fission bank
15.	Accumulate k-eff estimators for this iteration
16.	Destroy particle
17.	Calculate Shannon Entropy
18.	If *i* >*inactive_cycles*, accumulate estimators
19.	Output updated results
20.	Exit.

All of these steps are performed using functions described in the earlier modules. The track-length and collision estimators are accumulated for each neutron as it moves throughout the pin cell and interacts with

the materials. The leakage estimators and number of fission neutrons are calculated using the return value of the *particle::simulate()* routine.

The number of active and inactive cycles used is an inexact science. The values we selected were determined from running several different problems and observing when the Shannon Entropy and leakage estimates seemed to stabilize. Running 20 inactive cycles seemed to be enough to get any bias out of the results. After 20 inactive cycles, 180 active cycles were run to obtain estimates of desired quantities with sufficiently low uncertainty.

The driver source code is found in Appendix I.

3 Results

One of the most important results of interest for this problem is how k-eff changes as the pin pitch is modified. This result is shown below in figure 1. Out of the simulations that were run, the maximum k-effs are 1.07066 $\pm$ 0.00036 (track-length) and 1.07174 $\pm$ 0.00030 (collision), which both occurs at a pin pitch of 4.5 cm.

The shape of the curve is due to the fuel-to-moderator ratio. At the smallest pin pitches, there is little moderator to slow down the neutrons, so they usually just get absorbed in the fuel instead of thermalizing and cuasing fission. This effect would be mitigated some if fast fission in U-238 were considered in this model, but it is not. For the largest pitches, there is a greater chance of neutrons being absorbed in the moderator, or simply never returning to the fuel after thermalizing. Beginning at a pitch of 5.5 cm, this effect begins to dominate the benefits of moderation.

The reported uncertainty on the track-length estimator ranged from 31.4 pcm to 36.7 pcm. For the collision estimator, they ranged from 27.2 pcm to 30.5 pcm. **Is this what we expect? I would have thought track length would be more consistent.**

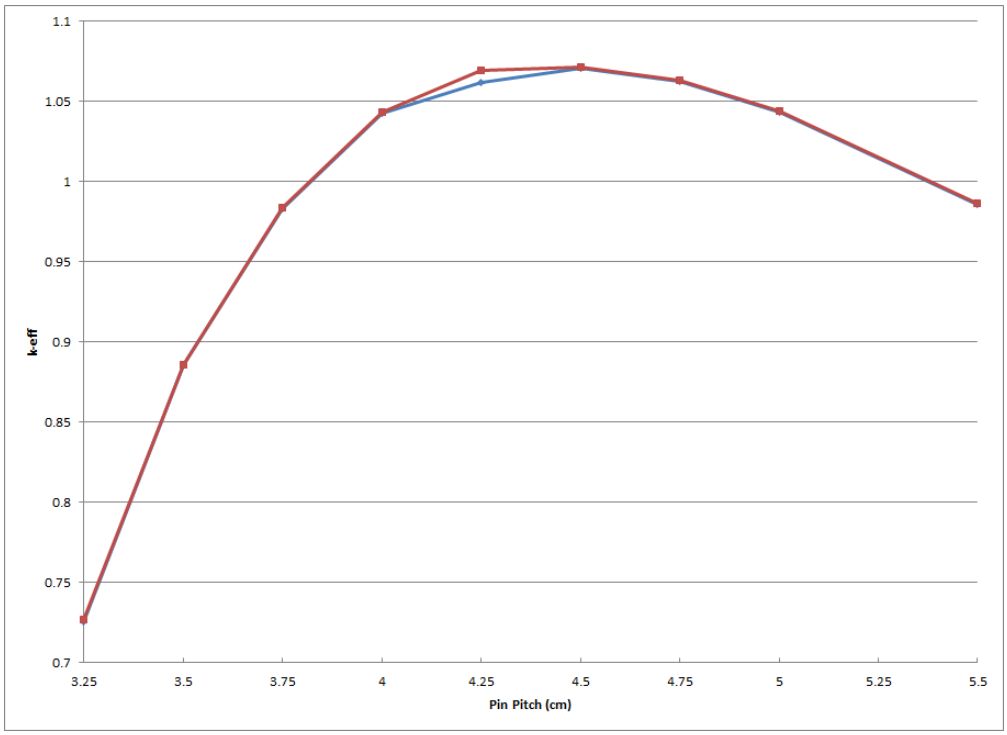

Figure 1: Plot of track-length (blue) and collision (red) estimators of k-eff versus pin pitch

A second quantity of interest is the leakage out of the pin cell, which is shown in figure 2. The leakage curves have some fluctuation in them, but generally have the same shape and similar magnitudes. This is to be expected, since the problem is axially symmetric. It also decreases with increasing pin pitch. This occurs because the increase in the amount of moderator slows down the neutrons more effectively, which decreases their mean free path and reduces the probability of escaping the pin cell.

For both figures, error bars are included, but too small to be easily visible. For figure 1, this might be acceptable, but for figure 2 we would expect to see the two curves lying within each other's uncertainties. However, in lecture it stated that the uncertainties which are reported are usually much lower than they should be. This is due to the fact that all the neutrons in the active cycles are correlated to their predecessors, but uncertainty calculations assume completely independent samples. Thus, this correlation makes the distributions appear more exact than they are. This, combined with how relatively few scores occur on each of these estimators, explains why there is some discrepancy between the top and bottom leakages.

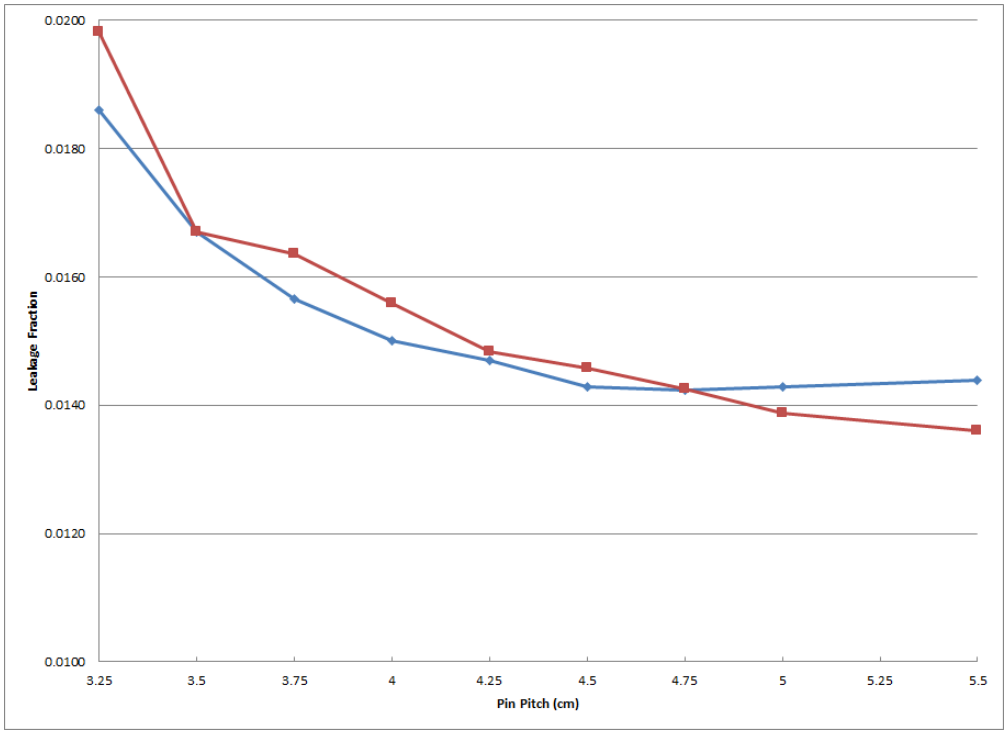

Figure 2: Plot of top (blue) and bottom (red) leakages versus pin pitch

4 Conclusion

The infinite lattice (pin cell) problem was successfully simulated using a Monte Carlo program. Estimators were used to determine the value of k-eff and axial leakages for the pin cell as a function of pin pitch. The values obtained were consistent with the expectations for this kind of problem, indicating that code is producing correct results.

Appendix A - utils.h

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include<iostream>
#include<cstdlib>
#include<limits>
#include<cmath>
#include<vector>
#include<algorithm>

//Variables
const double eps = std::numeric_limits<double>::epsilon()*100.0;
const double nudge = 1.0e-7;
const double pi = 3.14159265358979;
const double neut_mass = 939.565378; // MeV
const double kB = 8.6173324E-11; // MeV K^-1
const double nu = 2.45; // neutrons per fission
const double temp = 293; // Kelvin
const double lightspeed = 299792458.0; // m/s

// Functions
double drand(void);
double Watt(void);
bool approxeq(double, double);
bool approxge(double, double);
bool approxle(double, double);
bool softeq(double, double, double);
```

Appendix B - utils.cpp

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include<cstdlib>
#include<limits>
#include<cmath>
#include "utils.h"

// Random number generator on [0,1]
double drand(void)
{
  return static_cast<double>(rand())/static_cast<double>(RAND_MAX);
}

// Samples the Watt Spectrum
double Watt(void)
{
  double a = 0.988; // MeV
  double b = 2.249; // MeV^-1

  double x1 = drand();
  double x2 = drand();
  double x3 = drand();
  double x4 = drand();

  double W = a*(-log(x1)-log(x2)*cos(x3*pi/2)*cos(x3*pi/2));
  return W + a*a*b/4 + (2*x4-1)*sqrt(a*a*b*W);
}

// These operators are used for ==, >=, and <= for floating point values
bool approxeq(double x1, double x2)
{
  return (x1 > x2 - eps && x1 < x2 + eps);
}

bool approxge(double x1, double x2)
{
  return (x1 > x2 - eps);
}

bool approxle(double x1, double x2)
{
  return (x1 < x2 + eps);
}

bool softeq(double x1, double x2, double tol)
{
  return (x1 > x2 - tol && x1 < x2 + tol);
}
```

Appendix C - geometry.h

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include<vector>

const int xplane = 1, yplane = 2, zplane = 3;
const int interior = -1, vacuum = 0, reflecting = 1;

class surface{
  public:
    int boundaryType;
    int id;
    virtual double distToIntersect(double*, double*) = 0;
    virtual void reflect(double*, double*) = 0;
    virtual int getSense(double*) = 0;
};

class plane : public surface{
  public:
    double point[3];
    double norm[3];
    plane(int, double, int, int);
    double distToIntersect(double*, double*);
    void reflect(double*, double*);
    int getSense(double*);
};

class cylinder : public surface{
  public:
    double origin[3];
    double radius;
    cylinder(int, double, double, double, double, int);
    double distToIntersect(double*, double*);
    void reflect(double*, double*);
    int getSense(double*);
};

class cell{
  public:
    int id; // cell id number
    std::vector<int> iSurfs; // id numbers of surfaces which create this cell
    std::vector<int> senses; // senses for each surface. 1 is +, -1 is -
    int matid;
    cell(int, int, int*, int*);
    double distToIntersect(double*, double*, double*, int&);
};

cell* getPtr_cell(int);
surface* getPtr_surface(int);

void initPinCell(double, int, int);
```

```
void clearGeom();
int getCellID(double*);
```

Appendix D - geometry.cpp

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include "geometry.h"
#include "utils.h"

std::vector<surface*> surfaceList;
std::vector<cell*> cellList;

// Constructor for plane class
plane::plane(int surfid, double position, int orientation, int bound_in)
{
  // Set surface id
  id = surfid;
  // Set normal vector depending on which axis the plane intersects
  // Only planes which have normal vectors parallel to an axia are
  // supported right now.
  switch(orientation)
  {
    case xplane:
      point[0] = position; point[1] = 0.0; point[2] = 0.0;
      norm[0] = 1.0; norm[1] = 0.0; norm[2] = 0.0;
      break;
    case yplane:
      point[0] = 0.0; point[1] = position; point[2] = 0.0;
      norm[0] = 0.0; norm[1] = 1.0; norm[2] = 0.0;
      break;
    case zplane:
      point[0] = 0.0; point[1] = 0.0; point[2] = position;
      norm[0] = 0.0; norm[1] = 0.0; norm[2] = 1.0;
      break;
    default:
      std::cout << "Error when constructing plane!" << std::endl;
      abort();
  }
  switch(bound_in)
  {
    case reflecting:
    case vacuum:
      boundaryType = bound_in;
      break;
    default:
      boundaryType = -1;
  }
}

// Function to calculate distance between a plane and a point with an
// associated direction vector
double plane::distToIntersect(double position[3], double direction[3])
{
  double distance = -1.0;
```

```
  // Take dot product of position vector and plane's normal vector
  double prod = norm[0]*direction[0] + norm[1]*direction[1] +
    norm[2]*direction[2];

  if (!approxeq(prod,0.0))
  {
    // Calculate the distance between position and the plane
    distance = ((point[0] - position[0])*norm[0] +
      (point[1] - position[1])*norm[1] + (point[2] - position[2])*norm[2])/
      prod;
  }

  return distance;
}

// Reflection routine for planar surface
void plane::reflect(double* point_in, double* direction_in)
{
  double dot_product = direction_in[0]*norm[0] + direction_in[1]*norm[1] +
    direction_in[2]*norm[2];
  direction_in[0] -= 2.0*dot_product*norm[0];
  direction_in[1] -= 2.0*dot_product*norm[1];
  direction_in[2] -= 2.0*dot_product*norm[2];

  // "Nudge" point into cell
  point_in[0] += direction_in[0]*eps;
  point_in[1] += direction_in[1]*eps;
  point_in[2] += direction_in[2]*eps;

  return;
}

// Routine to return the sense of a plane with respect to some point
int plane::getSense(double* position)
{
  double position_vec[3], dotproduct;

  position_vec[0] = position[0] - point[0];
  position_vec[1] = position[1] - point[1];
  position_vec[2] = position[2] - point[2];

  dotproduct = position_vec[0]*norm[0] + position_vec[1]*norm[1] +
    position_vec[2]*norm[2];
  if (dotproduct > 0)
  {
    return 1;
  }
  else
  {
    return -1;
  }
}

// Constructor for cylindrical surface
```

```
// It is assumed that the cylinder's axis begins at (0,0,0) and
// points along the z-axis
cylinder::cylinder(int surfid, double x, double y, double z, double R,
    int bound_in)
{
  // Set values
  id = surfid;
  origin[0] = x; origin[1] = y; origin[2] = z;
  radius = R;
  switch(bound_in)
  {
    case reflecting:
    case vacuum:
      boundaryType = bound_in;
      break;
    default:
      boundaryType = -1;
  }
}

// Function to calculate distance between a cylindrical surface and a point
// with an association direction vector
double cylinder::distToIntersect(double position[3], double direction[3])
{
  double distance = -1.0;
  double intersection[3];
  double x, y, a, b, c, dis, m;
  double xp, xm, yp, ym, d2o, tmp;

  // If vector is parallel with z-axis, return
  if (approxeq(direction[0],0.0) && approxeq(direction[1],0.0)) return -1.0;

  // Shift system so that cylinder is center at origin (in x-y)
  x = position[0] - origin[0];
  y = position[1] - origin[1];
  // Calculate 2D slope of vector
  m = direction[1]/direction[0];
  // Calculate 2D distance from position to cylinder's origin
  d2o = sqrt(x*x + y*y);

  // Calculate a, b, c, and discriminant for quadratice formula
  a = 1.0 + m*m;
  b = -2.0*m*m*x + 2.0*y*m;
  c = y*y + m*m*x*x - 2.0*y*x*m - radius*radius;
  dis = b*b - 4.0*a*c;

  // If vector is pointing straight down in 2D, slope is 0, so
  // we treat this case specially
  if (softeq(direction[0],0.0,1.0e-4))
  {
    // Calculate 2 possible intersections
    yp = sin(acos(x/radius))*radius;
    ym = -yp;
    // If inside circle and going up, or outside circle and going down
```

```
  // to intersect circle:
  if ((d2o < radius && direction[1] > 0.0) ||
    (d2o > radius && direction[1] < 0.0 && y > 0.0 && fabs(x) <= radius))
  {
    // Select top y-value.  x-value is unchanged
    intersection[1] = yp;
    intersection[0] = x;
    tmp = (intersection[0] - x)*(intersection[0] - x) +
      (intersection[1] - y)*(intersection[1] - y);
    // Calculate z-value and distance
    intersection[2] = position[2] + sqrt(tmp)*direction[2]/
      sqrt(direction[0]*direction[0] + direction[1]*direction[1]);
    distance = sqrt(tmp + (intersection[2] - position[2])*
      (intersection[2] - position[2]));
  }
  // If inside circle and going down, or outside circle and going up
  // to intersect circle:
  else if ((d2o < radius && direction[1] < 0.0) ||
    (d2o > radius && direction[1] > 0.0 && y < 0.0 && fabs(x) <= radius))
  {
    // Select bottom y-value.  x-value is unchanged
    intersection[1] = ym;
    intersection[0] = x;
    tmp = (intersection[0] - x)*(intersection[0] - x) +
      (intersection[1] - y)*(intersection[1] - y);
    // Calculate z-value and distance
    intersection[2] = position[2] + sqrt(tmp)*direction[2]/
      sqrt(direction[0]*direction[0] + direction[1]*direction[1]);
    distance = sqrt(tmp + (intersection[2] - position[2])*
      (intersection[2] - position[2]));
  }
}
// If discriminant > 0, then line intersects circle twice
else if (dis > 0.0)
{
  // Calculate both x values
  xp = (-b + sqrt(dis))/(2.0*a);
  xm = (-b - sqrt(dis))/(2.0*a);
  // If the rightmost x-value is needed
  if ((x > xm && x < xp && direction[0] > 0.0) ||
    (x > xp && direction[0] < 0.0))
  {
    // Set x-value, calculate y-value using point-slop formula
    intersection[0] = xp;
    intersection[1] = m*(xp - x) + y;
    tmp = (intersection[0] - x)*(intersection[0] - x) +
      (intersection[1] - y)*(intersection[1] - y);
    // Calculate z-value and distance
    intersection[2] = position[2] + sqrt(tmp)*direction[2]/
      sqrt(direction[0]*direction[0] + direction[1]*direction[1]);
    distance = sqrt(tmp + (intersection[2] - position[2])*
      (intersection[2] - position[2]));
  }
  // If the leftmost x-value is needed
```

```
    else if ((x > xm && x < xp && direction[0] < 0.0) ||
      (x < xm && direction[0] > 0.0))
    {
      // Set x-value, calculate y-value using point-slope formula
      intersection[0] = xm;
      intersection[1] = m*(xm - x) + y;
      tmp = (intersection[0] - x)*(intersection[0] - x) +
        (intersection[1] - y)*(intersection[1] - y);
      // Calculate z-value and distance
      intersection[2] = position[2] + sqrt(tmp)*direction[2]/
        sqrt(direction[0]*direction[0] + direction[1]*direction[1]);
      distance = sqrt(tmp + (intersection[2] - position[2])*
        (intersection[2] - position[2]));
    }
  }
  // If discriminant == 0, there is only 1 intersection
  else if (approxeq(dis,0.0))
  {
    // Calculate x value
    xp = -b/(2.0*a);
    // Ensure that the vector is pointing the correct direction to
    // use this point
    if ((x < xp && direction[0] > 0.0) || (x > xp && direction[0] < 0.0))
    {
      // Set x-value, calculate y-value with point-slope formula
      intersection[0] = xp;
      intersection[1] = m*(xp - x) + y;
      tmp = (intersection[0] - x)*(intersection[0] - x) +
        (intersection[1] - y)*(intersection[1] - y);
      // Calculate z-value and distance
      intersection[2] = position[2] + sqrt(tmp)*direction[2]/
        sqrt(direction[0]*direction[0] + direction[1]*direction[1]);
      distance = sqrt(tmp + (intersection[2] - position[2])*
        (intersection[2] - position[2]));
    }
  }

  // Shift back to global coordinates
  intersection[0] += origin[0];
  intersection[1] += origin[1];

  // Return the distance value, which is -1.0 if there was no intersection
  return distance;
}

//Reflection routine for cylindrical surface
void cylinder::reflect(double* point_in, double* direction_in)
{
  double shifted[3], norm[3];

  // Shift the point of intersection to have a (0,0,0) origin
  shifted[0] = point_in[0] - origin[0];
  shifted[1] = point_in[1] - origin[1];
  shifted[2] = point_in[2] - origin[2];
```

```
  // Calculate normal vector at the point
  norm[0] = cos(shifted[0]/radius);
  norm[1] = cos(shifted[1]/radius);
  norm[2] = 0.0; // cylinder assumed to have axis in z-direction

  // Calculate reflection direction
  double dot_product = direction_in[0]*norm[0] + direction_in[1]*norm[1] +
    direction_in[2]*norm[2];
  direction_in[0] -= 2.0*dot_product*norm[0];
  direction_in[1] -= 2.0*dot_product*norm[1];
  direction_in[2] -= 2.0*dot_product*norm[2];

  // "Nudge" point into cell
  point_in[0] += direction_in[0]*eps;
  point_in[1] += direction_in[1]*eps;
  point_in[2] += direction_in[2]*eps;

  return;
}

// Routine to return the sense of a cylinder with respect to some point
int cylinder::getSense(double* position)
{
  double shifted[2], d2o;

  shifted[0] = position[0] - origin[0];
  shifted[1] = position[1] - origin[1];

  d2o = sqrt(shifted[0]*shifted[0] + shifted[1]*shifted[1]);
  if (d2o > radius)
  {
    return 1;
  }
  else
  {
    return -1;
  }
}

// Constructor for cell class
cell::cell(int cellid, int size, int* surfs, int* sense)
{
  id = cellid;
  matid = 0;
  // Loop over surfaces, adding the surface and its sense to vectors
  for (int i = 0; i < size; i++, surfs++, sense++)
  {
    iSurfs.push_back(*surfs);
    senses.push_back(*sense);
  }
}

// Calculate the distance to the nearest surface for a cell
```

```
double cell::distToIntersect(double* position, double* direction,
  double* intersection, int& surfIntersect)
{
  double distance = -1.0;
  int surfid = -1;

  // Loop over surfaces
  for (int i = 0; i < iSurfs.size(); i++)
  {
    // Get surface id
    surfid = iSurfs.at(i);
    // Calculate distancce to that surface
    double tmp =
      surfaceList.at(surfid)->distToIntersect(position,direction);
    // If this distance is better than the best so far, assign it
    if (tmp > 0.0 && (tmp < distance || distance < 0.0))
    {
      distance = tmp;
      surfIntersect = surfid;
    }
  }

  intersection[0] = position[0] + direction[0]*distance;
  intersection[1] = position[1] + direction[1]*distance;
  intersection[2] = position[2] + direction[2]*distance;
  return distance;
}

cell* getPtr_cell(int cellid)
{
  if (cellid < 0 || cellid >= cellList.size())
  {
    std::cout << "Error returning cell ptr.  Cell id " <<
      cellid << " is invalid." << std::endl;
    exit(-4);
  }
  return cellList.at(cellid);
}

surface* getPtr_surface(int surfid)
{
  if (surfid < 0 || surfid >= surfaceList.size())
  {
    std::cout << "Error returning surface ptr.  Surface id " <<
      surfid << " is invalid." << std::endl;
    exit(-5);
  }
  return surfaceList.at(surfid);
}

// Function to initialize a pin cell
void initPinCell(double pitch, int fuelid, int modid)
{
  double halfpitch = pitch/2.0;
```

```
  double height = 100.0; // Height is hard-coded, but could easily be changed
  double radius = 1.5; // Radius is hard-coded, but could easily be changed

// Build the fuel pin
  // Construct cylinder for fuel pin
  surfaceList.push_back(new cylinder(surfaceList.size(),0.0,0.0,0.0,radius,
    -1));
  // Construct top plane
  surfaceList.push_back(new plane(surfaceList.size(),height,zplane,0));
  // Construct bottom plane
  surfaceList.push_back(new plane(surfaceList.size(),0.0,zplane,0));
  // Construct the cell
  {
    int isurfs[3] = {0,1,2};
    int sense[3] = {-1,-1,1};
    cellList.push_back(new cell(fuelid,3,isurfs,sense));
  }

// Construct the "box" for the moderator
  // Construct left plane
  surfaceList.push_back(new plane(surfaceList.size(),-halfpitch,xplane,1));
  // Construct right plane
  surfaceList.push_back(new plane(surfaceList.size(),halfpitch,xplane,1));
  // Construct front plane
  surfaceList.push_back(new plane(surfaceList.size(),-halfpitch,yplane,1));
  // Construct back plane
  surfaceList.push_back(new plane(surfaceList.size(),halfpitch,yplane,1));
  // Construct the cell
  {
    int isurfs[7] = {0,1,2,3,4,5,6};
    int sense[7] = {1,-1,1,1,-1,1,-1};
    cellList.push_back(new cell(modid,7,isurfs,sense));
  }

  return;
}

void clearGeom()
{
  while(!cellList.empty())
  {
    delete cellList.back();
    cellList.pop_back();
  }
  while(!surfaceList.empty())
  {
    delete surfaceList.back();
    surfaceList.pop_back();
  }
  return;
}

int getCellID(double* position)
{
```

```
  int surfid, j;
  int senses[surfaceList.size()];
  cell* cellptr;
  surface* surfptr;
  std::fill_n(senses, surfaceList.size(), 0);

  for (int i = 0; i < cellList.size(); i++)
  {
    cellptr = cellList.at(i);
    for (j = 0; j < cellptr->iSurfs.size(); j++)
    {
      surfid = cellptr->iSurfs[j];
      surfptr = getPtr_surface(surfid);
      // Surface has not been checked yet
      if (senses[surfid] == 0) senses[surfid] = surfptr->getSense(position);
      // position is on wrong side of surface
      if(!(senses[surfid] == cellptr->senses[j])) break;
    }
    // Checked all surfaces without a break, so return this cell
    if (j == cellptr->iSurfs.size()) return cellptr->id;
  }

  return -1;
}
```

Appendix E - materials.h

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

class material{
  public:
    int id;
};

class moderator : public material{
  private:
    double moddens[2]; // 0 = H, 1 = O
    double mod_scat[2][3]; // 0 = H, 1 = O
    double mod_cap[3]; // H only
    double macabs_H, macscat_H, macscat_O;

  public:
    moderator(int);
    void modMacro(double, double*, double*, double*);
};

class fuel : public material{
  private:
    int nres;
    double fueldens[3]; // 0 = U235, 1 = U238, 2 = O
    double fuel_scat[3][3]; // 0 = O, 1 = U235, 2 = U238
    double fuel_cap[2][3]; // 0 = U235, 1 = U238
    double U235_fiss[3];
    double U238_res[3];
    double Eres[3];
    double rwidth[3];
    double dres, macscat_U235, macscat_U238, maccap_U235, maccap_U238;
    double macfiss_U235, macscat_O;
    double res_xs, y;

  public:
    fuel(int);
    void fuelMacro(double, double*, double*, double*, double*, double*);
    int sample_U(double*, double*);
    double fissXS(double);
};

material* getPtr_material(int);
void elastic(const double, int, double&, double[3]);
void init_materials(int& fuelid, int& modid);
```

Appendix F - materials.cpp

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include<vector>
#include<cmath>
#include<cstdlib>
#include "materials.h"
#include "utils.h"

std::vector<material*> materialList;
const int O16 = 16, U235 = 235, U238 = 238;

moderator::moderator(int matid)
{
  id = matid;

  // scattering cross sections
  mod_scat[0][0] = 2.0E+01;
  mod_scat[0][1] = 3.0E-03;
  mod_scat[0][2] =-1.2E+00;
  mod_scat[1][0] = 4.0E+00;
  mod_scat[1][1] = 1.5E-04;
  mod_scat[1][2] =-6.0E-01;

  // capture cross sections
  mod_cap[0] = 0.0E+00;
  mod_cap[1] = 8.0E-05;
  mod_cap[2] = 0.0E+00;

  // isotope number densities in atoms/(b*cm)
  moddens[0] = 6.6911E-02;
  moddens[1] = 3.3455E-02;

}
fuel::fuel(int matid)
{
  id = matid;

  // scattering cross sections
  fuel_scat[0][0] = 4.0E+00;
  fuel_scat[0][1] = 1.5E-04;
  fuel_scat[0][2] =-6.0E-01;
  fuel_scat[1][0] = 1.5E+01;
  fuel_scat[1][1] = 1.5E-04;
  fuel_scat[1][2] =-4.0E-01;
  fuel_scat[2][0] = 9.0E+00;
  fuel_scat[2][1] = 1.0E-04;
  fuel_scat[2][2] =-1.6E-01;

  // capture cross sections
  fuel_cap[0][0] = 4.0E-01;
```

```
  fuel_cap[0][1] = 2.5E-03;
  fuel_cap[0][2] =-1.0E+00;
  fuel_cap[1][0] = 1.8E+00;
  fuel_cap[1][1] = 4.0E-04;
  fuel_cap[1][2] =-1.5E+00;

  // fission cross sections
  U235_fiss[0] = 8.0E-01;
  U235_fiss[1] = 6.0E-02;
  U235_fiss[2] = 0.0E+00;

  // resonance data
  nres = 3;
  // energies in MeV
  Eres[0] = 6.6E-06;
  Eres[1] = 2.2E-05;
  Eres[2] = 3.8E-05;
  // peak cross section in barns
  U238_res[0] = 7.0E+03;
  U238_res[1] = 6.0E+03;
  U238_res[2] = 6.5E+03;
  // resonance widths in MeV
  rwidth[0] = 4.0E-08;
  rwidth[1] = 3.0E-08;
  rwidth[2] = 1.0E-07;

  dres = 100; // practical width of resonance

  // isotope number densities in atoms/(b*cm)
  fueldens[0] = 4.7284E-02;
  fueldens[1] = 9.4567E-04;
  fueldens[2] = 2.2696E-02;
}

void init_materials(int& fuelid, int& modid)
{
  fuelid = 0;
  materialList.push_back(new fuel(fuelid));
  modid = 1;
  materialList.push_back(new moderator(modid));

  return;
}
material* getPtr_material(int matid)
{
  if (matid < 0 || matid >= materialList.size())
  {
    std::cout << "Error returning material ptr.  Material id " <<
      matid << " is invalid." << std::endl;
    exit(-6);
  }
  return materialList.at(matid);
}
```

```
void moderator::modMacro(double E, double *totalxs, double *H_frac,
    double *abs_frac)
{
  double sqrE = sqrt(E);
  macabs_H = moddens[0]*mod_cap[1]/sqrE;
  macscat_H = moddens[0]*(mod_scat[0][0]+mod_scat[0][1]/sqrE)*
    exp(mod_scat[0][2]*sqrE);
  macscat_O = moddens[1]*(mod_scat[1][0]+mod_scat[1][1]/sqrE)*
    exp(mod_scat[1][2]*sqrE);

  *totalxs = macscat_H+macscat_O+macabs_H;
  *H_frac = (macabs_H+macscat_H)/(*totalxs);
  *abs_frac = macabs_H/(macabs_H+macscat_H);

   return;
}

void fuel::fuelMacro(double E, double *totalxs, double *frac_U235,
    double *frac_U238, double *fiss_frac, double *abs_frac)
{
  double sqrE = sqrt(E);
  // check for proximity to a resonance
  res_xs = 0;
  for(int j = 0; j < nres; j++){
    if(fabs(E-Eres[j]) < dres*rwidth[j]){
      y = (2.0/rwidth[j])*(E-Eres[j]);
      res_xs = fueldens[2]*U238_res[j]*sqrt(Eres[j]/E)/(1+y*y);
    }
  }

  macscat_O = fueldens[0]*(fuel_scat[0][0]+fuel_scat[0][1]/sqrE)*
    exp(fuel_scat[0][2]*sqrE);
  macscat_U235 = fueldens[1]*(fuel_scat[1][0]+fuel_scat[1][1]/sqrE)*
    exp(fuel_scat[1][2]*sqrE);
  macscat_U238 = fueldens[2]*(fuel_scat[2][0]+fuel_scat[2][1]/sqrE)*
    exp(fuel_scat[2][2]*sqrE);

  maccap_U235 = fueldens[1]*(fuel_cap[0][0]+fuel_cap[0][1]/sqrE)*
    exp(fuel_cap[0][2]*sqrE);
  maccap_U238 = fueldens[2]*(fuel_cap[1][0]+fuel_cap[1][1]/sqrE)*
    exp(fuel_cap[1][2]*sqrE)+res_xs;
  macfiss_U235 = fueldens[1]*(U235_fiss[0]+U235_fiss[1]/sqrE)*
    exp(U235_fiss[2]*sqrE);

  *totalxs = macscat_O+macscat_U235+macscat_U238+maccap_U235+maccap_U238+
    macfiss_U235;
  double scatterXS = macscat_U235+macscat_U238+macscat_O;
  *frac_U235 = macscat_U235/scatterXS;
  *frac_U238 = macscat_U238/scatterXS;
  *fiss_frac = macfiss_U235/(*totalxs);
  *abs_frac = (maccap_U235+maccap_U238+macfiss_U235)/(*totalxs);

  return;
}
```

```
int fuel::sample_U(double *frac_U235, double *frac_U238)
{
  double xi = drand();
  if(xi < *frac_U235)
  {
    return U235;
  }
  else if(xi < (*frac_U235+*frac_U238))
  {
    return U238;
  }
  else
  {
    return O16;
  }
}

void elastic(const double T, int A_in, double &v_n, double d_n[3])
{
  double A = static_cast<double>(A_in);
  double beta = sqrt(neut_mass/(lightspeed*lightspeed)*A/(2*kB*T));

  bool transform1= false;
  bool transform2= false;
  double tmp = d_n[0];
  if(fabs(1-d_n[2]) < nudge*1.0E-04)
  {
    d_n[0] = d_n[2];
    d_n[2] = d_n[1];
    d_n[1] = tmp;

    transform1 = true;
  }

//  std::cout << "Incoming neutron: " << std::endl;
//  std::cout << "velocity = " << v_n << std::endl;
//  std::cout << "omegax = " << d_n[0] << std::endl;
//  std::cout << "omegay = " << d_n[1] << std::endl;
//  std::cout << "omegaz = " << d_n[2] << std::endl;

//  std::cout << "beta = " << beta <<  std::endl;
//  std::cout << std::endl;
  double x;
  double y = beta*v_n;
  double w1 = sqrt(pi)*y/(2 + sqrt(pi)*y);

  double eta = 1.0;
  double f1 = 0.0;
  double w, x1, x2, x3, Vtil, mutil;
  while(eta > f1){ // sample until Vtil, mutil are accepted
    w = drand();
    x1 = drand();
    x2 = drand();
```

```
    if(w < w1){  // sample g1(x)
      x3 = drand();
      x = sqrt(-log(x1) - log(x2)*cos(x3*pi/2)*cos(x3*pi/2));
    }
    else{ // sample g2(x)
      x = sqrt(-log(x1*x2));
    }

    Vtil = x/beta;
    mutil = 2*drand() - 1;

    // check for rejection from scaled f1(V,mu) (Lecture Module 8)
    eta = drand();
    f1 = sqrt(v_n*v_n + Vtil*Vtil - 2*v_n*Vtil*mutil)/(v_n+Vtil);
  }
//  std::cout << "isotope = " << A << std::endl;
//  std::cout << "Vtil = " << Vtil << std::endl;
//  std::cout << "mutil = " << mutil << std::endl;
//  std::cout << std::endl;

  // sample direction vector for the target nucleus Omega_T-hat
  double gamma = 2*pi*drand();
  double Tx = mutil*d_n[0] + (d_n[0]*d_n[2]*cos(gamma) - d_n[1]*sin(gamma)*
    sqrt((1-mutil*mutil)/(1-d_n[2]*d_n[2])));
  double Ty = mutil*d_n[1] + (d_n[1]*d_n[2]*cos(gamma) + d_n[0]*sin(gamma)*
    sqrt((1-mutil*mutil)/(1-d_n[2]*d_n[2])));
  double Tz = mutil*d_n[2] - cos(gamma)*sqrt((1-mutil*mutil)*
    (1-d_n[2]*d_n[2]));

  // center-of-mass velocity u_xyz
  double ux = (v_n*d_n[0] + A*Vtil*Tx)/(1+A);
  double uy = (v_n*d_n[1] + A*Vtil*Ty)/(1+A);
  double uz = (v_n*d_n[2] + A*Vtil*Tz)/(1+A);

  // neutron center-of-mass velocity
  double vcx = v_n*d_n[0] - ux;
  double vcy = v_n*d_n[1] - uy;
  double vcz = v_n*d_n[2] - uz;
  double vcn = sqrt(vcx*vcx + vcy*vcy + vcz*vcz);

  // neutron center-of-mass direction vector
  double ncx = vcx/vcn;
  double ncy = vcy/vcn;
  double ncz = vcz/vcn;

  if(fabs(1-ncz) < nudge*1.0E-04)
  {
    tmp = ncx;
    ncx = ncz;
    ncz = ncy;
    ncy = tmp;

    transform2 = true;
  }
```

```
  // outgoing neutron center-of-mass direction
  gamma = 2*pi*drand();
  double muc = 2*drand() - 1;
  double ncxp = muc*ncx + (ncx*ncz*cos(gamma) - ncy*sin(gamma))*
    sqrt((1-muc*muc)/(1-ncz*ncz));
  double ncyp = muc*ncy + (ncy*ncz*cos(gamma) + ncx*sin(gamma))*
    sqrt((1-muc*muc)/(1-ncz*ncz));
  double nczp = muc*ncz - cos(gamma)*sqrt((1-muc*muc)*(1-ncz*ncz));

  if(transform2)
  {
    tmp = ncz;
    ncz = ncx;
    ncx = ncy;
    ncy = tmp;
  }

  // finally, outgoing neutron velocity in lab frame is calculated
  double vncx = vcn*ncxp + ux;
  double vncy = vcn*ncyp + uy;
  double vncz = vcn*nczp + uz;
  v_n = sqrt(vncx*vncx + vncy*vncy + vncz*vncz);
  d_n[0] = vncx/v_n;
  d_n[1] = vncy/v_n;
  d_n[2] = vncz/v_n;
//  std::cout << "Outgoing neutron: " << std::endl;
//  std::cout << "velocity = " << v_n << std::endl;
//  std::cout << "omegax = " << d_n[0] << std::endl;
//  std::cout << "omegay = " << d_n[1] << std::endl;
//  std::cout << "omegaz = " << d_n[2] << std::endl;

//  std::cout << std::endl;

  if(transform1)
  {
    tmp = d_n[2];
    d_n[2] = d_n[0];
    d_n[0] = d_n[1];
    d_n[1] = tmp;
  }
  return;
}

double fuel::fissXS(double energy)
{
  double sqrE = sqrt(energy);
  return fueldens[1]*(U235_fiss[0]+U235_fiss[1]/sqrE)*exp(U235_fiss[2]*sqrE);
}
```

Appendix G - particles.h

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include<cstdlib>
#include<vector>

class particle{
  private:
    bool isAlive;
    int cellid;
    double position[3];
    double omega[3];
    double energy;
    double score;
    double cutoff;
    double survival;
    double totalXS,f235,f238,fH,fcap,fiss_frac,abs_frac;

  public:
    particle(const double[3], double, double, double, int);
    particle(double[3], double, double, double, int);
    double getCoord(int);
    int simulate();
    int simulate_implicit();
    bool roulette();
    void spectrumTally(double,double,int);
    double weight;
    double estimatorTL,estimatorColl;
    friend class fission;
};

class fission{
  private:
    double position[3];
    int cellid;
  public:
    fission(const particle&,int);
    friend void makeSource(std::vector<fission>&,std::vector<particle>&,int);
    friend double calcEntropy(std::vector<fission>);
};

void makeSource(std::vector<fission>&,std::vector<particle>&,int);
double calcEntropy(std::vector<fission> fissionBank);
extern double fuelSpectrum[1001];
extern double modSpectrum[1001];
extern double energyGrid[1001];
```

Appendix H - particles.cpp

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include<cstdlib>
#include<cmath>
#include<vector>
#include "geometry.h"
#include "materials.h"
#include "particles.h"
#include "utils.h"

double fuelSpectrum[1001];
double modSpectrum[1001];
double energyGrid[1001];

particle::particle(double pos_in[3], double gamma, double mu,
  double E_in, int cellid_in)
{
  cellid = cellid_in;
  isAlive = true;

  position[0] = pos_in[0];
  position[1] = pos_in[1];
  position[2] = pos_in[2];

  omega[0] = sqrt(1.0 - mu*mu)*cos(gamma);
  omega[1] = sqrt(1.0 - mu*mu)*sin(gamma);
  omega[2] = mu;

  energy = E_in;
  weight = 1.0;
  cutoff = 0.1;
  survival = 0.4;

  totalXS = 0.0;
  f235 = 0.0;
  f238 = 0.0;
  fH = 0.0;
  fcap = 0.0;
  fiss_frac = 0.0;
  abs_frac = 0.0;

  score = 0.0;
  estimatorTL = 0.0;
  //squareTL = 0.0;
  estimatorColl = 0.0;
  //squareColl = 0.0;
}

fission::fission(const particle& neutron, int cellid_in)
{
```

```
  position[0] = neutron.position[0];
  position[1] = neutron.position[1];
  position[2] = neutron.position[2];

  cellid = cellid_in;
}

int particle::simulate()
{
  int result, surfid;
  int isotope;
  const int fuelid = 0; const int modid = 1;
  double vn, xi;
  double dcoll, dsurf, intersection[3];
  cell* cellptr;
  surface* surfptr;
  fuel* thisFuel = new fuel(fuelid);
  moderator* thisMod = new moderator(modid);

  while (isAlive)
  {
    // Get pointer to the current cell
    cellptr = getPtr_cell(cellid);
    if(cellptr->id == fuelid)
    {
      thisFuel->fuelMacro(energy,&totalXS,&f235,&f238,&fiss_frac,&abs_frac);
    }
    else if(cellptr->id == modid)
    {
      thisMod->modMacro(energy,&totalXS,&fH,&fcap);
    }
    else
    {
      std::cout << "Not fuel or moderator id." << std::endl;
      exit(-3);
    }
    // get distance to next collision
    //dcoll = 500.0; // Just to test the surface/cell stuff
    dcoll = -log(drand())/(totalXS);
    // Get closest surface distance
    dsurf = cellptr->distToIntersect(position, omega, intersection, surfid);

    // Move particle to surface
    if (dsurf < dcoll)
    {
      // tally the track length estimator for keff
      if(cellptr->id == fuelid)
      {
        score = dsurf*weight*nu*fiss_frac*totalXS;
        estimatorTL = estimatorTL + score;
        //squareTL = squareTL + score*score;

      }
      // Move particle
```

```
  position[0] = intersection[0];
  position[1] = intersection[1];
  position[2] = intersection[2];
  // get pointer to the surface that the particle is colliding with
  surfptr = getPtr_surface(surfid);
  switch(surfptr->boundaryType)
  {
    // Particle hit reflecting boundary
    case reflecting:
      surfptr->reflect(intersection, omega);
      // Nudge the particle a bit to avoid floating point issues
      position[0] += omega[0]*nudge;
      position[1] += omega[1]*nudge;
      position[2] += omega[2]*nudge;
      break;
    // Particle hit vacuum boundary and escaped
    case vacuum:
      // Set return value and "kill" particle
      result = -surfid;
      isAlive = false;
      break;
    // Particle hit interior surface
    case interior:
      position[0] += omega[0]*nudge;
      position[1] += omega[1]*nudge;
      position[2] += omega[2]*nudge;

      cellid = getCellID(position);
      cellptr = getPtr_cell(cellid);
      break;
    default:
      std::cout << "Error in particle::simulate().  Particle encountered "
        << "unknown boundary type." << std::endl;
      exit(-2);
  }
}
// Move particle to collision point and sample collision
else
{
  position[0] += omega[0]*dcoll;
  position[1] += omega[1]*dcoll;
  position[2] += omega[2]*dcoll;
  switch(cellptr->id)
  {
    case fuelid:
      isotope = thisFuel->sample_U(&f235,&f238);
      // tally the track length estimator and the collision estimator
      score = dcoll*weight*nu*fiss_frac*totalXS;
      //std::cout << "Particle weight = " << weight << std::endl;
      //std::cout << "absorption fraction = " << abs_frac << std::endl;
      //std::cout << "fission fraction = " << fiss_frac << std::endl;
      //std::cout << "total XS = " << totalXS << std::endl;
      estimatorTL = estimatorTL + score;
      //squareTL = squareTL + score*score;
```

```
        score = weight*nu*fiss_frac;
        //std::cout << "Score added = " << score << std::endl;
        estimatorColl = estimatorColl + score;
        //squareColl = squareColl + score*score;

        xi = drand();
        if(xi > abs_frac) // scatter
        {
          isotope = thisFuel->sample_U(&f235,&f238);
          vn = sqrt(2.0*energy/neut_mass)*lightspeed;
          elastic(temp,isotope,vn,omega);
          energy = neut_mass*(vn/lightspeed)*(vn/lightspeed)/2.0;
        }
        else // absorption
        {
          isAlive = false;
          if(xi > fiss_frac) // capture, maybe score which isotope
          {
            result = 0;
          }
          else // fission
          {
            result = static_cast<int>(weight*nu+drand());
          }
        }
        break;
      case modid:
        if(drand() < fH) // interaction with hydrogen
        {
          if(drand() > fcap)
          {
            vn = sqrt(2.0*energy/neut_mass)*lightspeed;
            elastic(temp,1,vn,omega);
            energy = neut_mass*(vn/lightspeed)*(vn/lightspeed)/2.0;
          }
          else // capture; score estimator, end history, etc.
          {
            result = 0;
            isAlive = false;
          }
        }
        else // interaction with oxygen; all are scatters
        {
          vn = sqrt(2.0*energy/neut_mass)*lightspeed;
          elastic(temp,16,vn,omega);
          energy = neut_mass*(vn/lightspeed)*(vn/lightspeed)/2.0;
        }
        break;
      default:
        std::cout << "Not fuel or moderator id." << std::endl;
        exit(-3);
    }
  }
}
```

```
  delete thisFuel;
  delete thisMod;
  return result;
}

int particle::simulate_implicit()
{
  int result, surfid;
  int isotope;
  const int fuelid = 0; const int modid = 1;
  double vn, xi;
  double dcoll, dsurf, intersection[3];
  cell* cellptr;
  surface* surfptr;
  fuel* thisFuel = new fuel(fuelid);
  moderator* thisMod = new moderator(modid);

  while (isAlive)
  {
    // Get pointer to the current cell
    cellptr = getPtr_cell(cellid);
    if(cellptr->id == fuelid)
    {
      thisFuel->fuelMacro(energy,&totalXS,&f235,&f238,&fiss_frac,&abs_frac);
    }
    else if(cellptr->id == modid)
    {
      thisMod->modMacro(energy,&totalXS,&fH,&fcap);
    }
    else
    {
      std::cout << "Not fuel or moderator id." << std::endl;
      exit(-3);
    }
    // get distance to next collision
    //dcoll = 500.0; // Just to test the surface/cell stuff
    dcoll = -log(drand())/(totalXS);
    // Get closest surface distance
    dsurf = cellptr->distToIntersect(position, omega, intersection, surfid);

    // Move particle to surface
    if (dsurf < dcoll)
    {
      //spectrumTally(energy, dsurf*weight, cellptr->id);
      // tally the track length estimator for keff
      if(cellptr->id == fuelid)
      {
        score = dsurf*weight*nu*fiss_frac*totalXS;
        estimatorTL = estimatorTL + score;
        //squareTL = squareTL + score*score;
      }
      // Move particle
      position[0] = intersection[0];
```

```
  position[1] = intersection[1];
  position[2] = intersection[2];
  // get pointer to the surface that the particle is colliding with
  surfptr = getPtr_surface(surfid);
  switch(surfptr->boundaryType)
  {
    // Particle hit reflecting boundary
    case reflecting:
      surfptr->reflect(intersection, omega);
      // Nudge the particle a bit to avoid floating point issues
      position[0] += omega[0]*nudge;
      position[1] += omega[1]*nudge;
      position[2] += omega[2]*nudge;
      break;
    // Particle hit vacuum boundary and escaped
    case vacuum:
      // Set return value and "kill" particle
      result = -surfid;
      isAlive = false;
      break;
    // Particle hit interior surface
    case interior:
      position[0] += omega[0]*nudge;
      position[1] += omega[1]*nudge;
      position[2] += omega[2]*nudge;

      cellid = getCellID(position);
      cellptr = getPtr_cell(cellid);
      break;
    default:
      std::cout << "Error in particle::simulate().  Particle encountered "
        << "unknown boundary type." << std::endl;
      exit(-2);
  }
}
// Move particle to collision point and sample collision
else
{
  position[0] += omega[0]*dcoll;
  position[1] += omega[1]*dcoll;
  position[2] += omega[2]*dcoll;
  if(cellptr->id == fuelid)
  {
    //spectrumTally(energy,dcoll*weight,fuelid);
    isotope = thisFuel->sample_U(&f235,&f238);
    // tally the track length estimator and the collision estimator
    score = dcoll*weight*nu*fiss_frac*totalXS;
    estimatorTL = estimatorTL + score;
    //squareTL = squareTL + score*score;
    score = weight*nu*fiss_frac;
    estimatorColl = estimatorColl + score;
    //squareColl = squareColl + score*score;

    xi = drand();
```

```
        vn = sqrt(2.0*energy/neut_mass)*lightspeed;
        elastic(temp,isotope,vn,omega);
        energy = neut_mass*(vn/lightspeed)*(vn/lightspeed)/2.0;
        if(xi < fiss_frac)
        {
//          std::cout << "Particle fissioned." << std::endl;
          result = static_cast<int>(weight*nu+drand());
          isAlive = false;
        }
        weight = weight*(1.0-abs_frac);
//        std::cout << "Implicit capture: absorption fraction = " \
  << abs_frac << std::endl;
        if(weight < cutoff)
        {
          isAlive = roulette();
        }
      }
      else if(cellptr->id == modid)
      {
        //spectrumTally(energy,dcoll*weight,modid);
        if(drand() < fH) // interaction with hydrogen
        {
          isotope = 1;
        }
        else // interaction with oxygen; all are scatters
        {
          isotope = 16;
        }
        vn = sqrt(2.0*energy/neut_mass)*lightspeed;
        elastic(temp,isotope,vn,omega);
        energy = neut_mass*(vn/lightspeed)*(vn/lightspeed)/2.0;

        weight = weight*(1.0-fH*fcap);
//        std::cout << "Implicit capture fraction = " << fH*fcap << std::endl;
        if(weight < cutoff)
        {
          isAlive = roulette();
        }
      }
      else
      {
        std::cout << "Not fuel or moderator id." << std::endl;
        exit(-3);
      }
    }
  }

  delete thisFuel;
  delete thisMod;
  return result;
}

bool particle::roulette()
```

```
{
  if(drand() < weight/survival)
  {
    weight = survival;
    return true;
  }
  else
  {
//    std::cout << "The particle was killed in rouletting with a weight of "
//              << weight << std::endl;
    weight = 0.0;
    return false;
  }
}
void makeSource(std::vector<fission> &fissionBank,
    std::vector<particle> &sourceBank, int batch_size)
{
  double sourceProb, xi;
  fission* fissptr;

  if (fissionBank.size() == batch_size)
  {
    while(!fissionBank.empty())
    {
      sourceBank.push_back(particle((fissionBank.back()).position,
        2.0*pi*drand(),2.0*drand()-1.0,Watt(),0));
      fissionBank.pop_back();
    }
  }
  else if (fissionBank.size() > batch_size) // Fission bank is too large
  {
    // Add to source bank with probability batch_size/fissionBank.size()
    while(!fissionBank.empty())
    {
      xi = drand();
      sourceProb = static_cast<double>((batch_size-sourceBank.size())/
                   static_cast<double>(fissionBank.size()));
      if(xi < sourceProb)
      {
        sourceBank.push_back(particle((fissionBank.back()).position,
          2.0*pi*drand(),2.0*drand()-1.0,Watt(),0));
      }
      fissionBank.pop_back();
    }
  }
  else if (fissionBank.size() < batch_size) // Fission bank is too small
  {
    // Add to source bank with probability batch_size/fissionBank.size()
    for(int j = 0; j < (int)(batch_size/fissionBank.size()); j++)
    {
      for(int k = 0; k < fissionBank.size(); k++)
      {
        sourceBank.push_back(particle(fissionBank[k].position,2.0*pi*drand(),
          2.0*drand()-1.0,Watt(),0));
```

```
      }
    }
    while(!fissionBank.empty())
    {
      xi = drand();
      sourceProb = static_cast<double>(batch_size-sourceBank.size())/
                   static_cast<double>(fissionBank.size());
      if(xi < sourceProb)
      {
        sourceBank.push_back(particle((fissionBank.back()).position,
          2.0*pi*drand(),2.0*drand()-1.0,Watt(),0));
      }
      fissionBank.pop_back();
    }
  }
  return;
}

double calcEntropy(std::vector<fission> fissionBank)
{
  double radius = 1.5;
  int nrad = 10;
  int nz = 100;
  int nbins = nz*nrad;
  int particle_mesh[nbins];
  double dz = 100.0/nz;
  double area = radius*radius/nrad;
  for(int i = 0; i < nbins; i++)
  {
    particle_mesh[i] = 0;
  }

  int index;

  // bin all of the particles
  // uniform axial bins, equal-area radial bins
  double p_rad, pn;
  double x, y;
  int z_index, r_index;
  for(int i = 0; i < fissionBank.size(); i++){
    x = fissionBank[i].position[0];
    y = fissionBank[i].position[1];
    p_rad = x*x + y*y;
    r_index = (int)(p_rad/area);
    if(r_index >= nrad)
    {
      std::cout << "Outside of the fuel region" << std::endl;
      std::cout << "x = " << fissionBank[i].position[0] << std::endl;
      std::cout << "y = " << fissionBank[i].position[1] << std::endl;
      std::cout << "z = " << fissionBank[i].position[2] << std::endl;
    }
    z_index = (int)fissionBank[i].position[2]/dz;
    particle_mesh[nrad*z_index + r_index] =
      particle_mesh[nrad*z_index + r_index] + 1;
```

```
  }
  // calculate Shannon entropy
  double entropy = 0.0;
  for(int i = 0; i < nz; i++)
  {
    for(int j = 0; j < nrad; j++)
    {
      pn = (double)(particle_mesh[nrad*i + j])/(double)(fissionBank.size());
      if(pn > 0.0)
      {
        entropy = entropy + pn*log2(pn);
      }
    }
  }
  entropy = -entropy;
  return entropy;
}

void particle::spectrumTally(double energy, double fluxTally, int id)
{
  int g = 0;
  while(energy > energyGrid[g])
  {
    g = g+1;
  }

  if(id == 0)
  {
    fuelSpectrum[g] = fuelSpectrum[g] + fluxTally;
  }
  else if(id == 1)
  {
    modSpectrum[g] = modSpectrum[g] + fluxTally;
  }
}
```

Appendix I - NERS-544.cpp

```
// AUTHORS: Aaron Graham, Mike Jarrett
// PURPOSE: NERS 544 Course Project
// DATE   : April 30, 2015

#include <iostream>
#include <ostream>
#include <fstream>
#include <sstream>
#include "utils.h"
#include "particles.h"
#include "geometry.h"
#include "materials.h"

using namespace std;

int main()
{
//  srand(time(NULL));
  srand(101230489);
//  srand(23549);

  int fuelid, modid;
  double pitch;
  cout << "Enter the pin pitch in cm (must be greater than 3.0):";
  cin >> pitch;

  if (pitch <= 3.0)
  {
    cout << "Error!  Pin pitch must be greater than pin diameter of 3.0 cm!"
      << endl;
    exit(-1);
  }

  init_materials(fuelid, modid);

  int batch_size = 1E5;
  double En;
  double xyz[3];
  double pinrad = 1.5; // pin radius = 1.5 cm
  double r, gamma, mu;
  vector<particle> sourceBank;
  vector<fission> fissionBank;

  // estimators
  double topCurrent = 0.0, bottomCurrent = 0.0;
  int topSurf = -1;
  int bottomSurf = -2;
  double tally_TL = 0.0, tally_TLsq = 0.0;
  double tally_coll = 0.0, tally_collsq = 0.0, topleaksq = 0.0;
  double bottomleaksq = 0.0;
  double keff_TL, keff_Coll, sigTL, sigColl, active_particles;
  double topleak, bottomleak, sigtop, sigbottom, score = 0.0;
```

```
  particle neutron = particle(xyz,gamma,mu,En,fuelid);

  // outer loop over power iterations
  const int max_iters = 200, active_iters = 180, inactive_iters = 20;
  double ShannonEntropy[max_iters];
  double totalEntropy = 0.0, meanEntropy;
  int k = 0, l = 0, result, ktot = 0;

//  double pitches[npitch] = {3.25,3.5,3.75,4.0,4.25,4.5,4.75,5.0,5.5,6.0};

  // make the energy grid for flux tally
  int decades = 10;
  const int groups = 1001;
  double x = -4;
  double dg = (double)(decades)/(double)(groups-1);

  for(int i = 0; i < groups; i++)
  {
    energyGrid[i] = 20*pow(10,x);
    modSpectrum[i] = 0.0;
    fuelSpectrum[i] = 0.0;
    x = x+dg;
  }

  stringstream convert;
  convert << pitch << ".out";
  string filename = convert.str();
  ofstream myfile;
  myfile.open(filename.c_str());
/*
  ofstream spectrum;
  convert << "spectrum." << pitch;
  string spectrumfile = convert.str();
  spectrum.open(spectrumfile.c_str());
*/
    initPinCell(pitch, fuelid, modid);

    // zero all estimators
    k = 0;
    l = 0;
    ktot = 0;
    totalEntropy = 0.0;
    tally_TL = 0;
    tally_coll = 0;
    tally_TLsq = 0;
    tally_collsq = 0;
    topCurrent = 0.0;
    bottomCurrent = 0.0;
    topleaksq = 0.0;
    bottomleaksq = 0.0;
    bottomleak = 0;
    sigtop = 0.0;
    sigbottom = 0.0;
```

```
    // sample neutrons for initial source bank
    for(int i = 0; i < batch_size; i++){
      // sample energy from Watt spectrum
      En = Watt();
      // sample a radial location within the fuel cell
      gamma = 2*pi*drand();
      r = pinrad*sqrt(drand());
      xyz[0] = r*cos(gamma);
      xyz[1] = r*sin(gamma);
      // sample an axial location
      xyz[2] = 100.0*drand();
      // sample a direction
      gamma = 2*pi*drand();
      mu = 2.0*drand() - 1.0;
      // add particle to the bank
      sourceBank.push_back(particle(xyz,gamma,mu,En,fuelid));
    }

    while(k < max_iters){
      k = k+1; // total power iterations

      // inner loop over the source bank
      while(!sourceBank.empty())
      {
        // Get pointer to particle
        neutron = sourceBank.back();
        // Simulate particle
//        result = neutron.simulate_implicit();
        result = neutron.simulate();
        // Create fission neutrons (if fissions > 0)
        if(result > 0)
        {
          for(int i = 0; i < result; i++)
          {
            fissionBank.push_back(fission(neutron,fuelid));
          }
        }

        // get keff tallies for the history
        if(k > inactive_iters)
        {
          tally_TL = tally_TL + neutron.estimatorTL;
          tally_coll = tally_coll + neutron.estimatorColl;
          tally_TLsq = tally_TLsq + neutron.estimatorTL*neutron.estimatorTL;
          tally_collsq = tally_collsq + neutron.estimatorColl*
            neutron.estimatorColl;

          // Calculate Leakages
          if(result == topSurf)
          {
            score = neutron.weight;
            topCurrent = topCurrent + score;
            topleaksq = topleaksq + score*score;
          }
```

```
    else if(result == bottomSurf)
    {
      score = neutron.weight;
      bottomCurrent = bottomCurrent + score;
      bottomleaksq = bottomleaksq + score*score;
    }
  }
  // Delete pointer to neutron in sourcebank;
  sourceBank.pop_back();
}

// Calculate Shannon Entropy
ShannonEntropy[k-1] = calcEntropy(fissionBank);
// let a few cycles go by before starting to calculate the mean
if(k > inactive_iters)
{
  totalEntropy = totalEntropy + ShannonEntropy[k-1];
  meanEntropy = totalEntropy/(double)(k-inactive_iters);
  l = l+1; // power iterations with converged source
}

// Calculations for output
active_particles = static_cast<double>(batch_size*(k-inactive_iters));
keff_TL = tally_TL/active_particles;
sigTL = sqrt((tally_TLsq/active_particles -
  keff_TL*keff_TL)/active_particles);
keff_Coll = tally_coll/active_particles;
sigColl = sqrt((tally_collsq/active_particles -
  keff_Coll*keff_Coll)/active_particles);
topleak = topCurrent/active_particles;
sigtop = sqrt((topleaksq/active_particles -
  topleak*topleak)/active_particles);
bottomleak = bottomCurrent/active_particles;
sigbottom = sqrt((bottomleaksq/active_particles -
  bottomleak*bottomleak)/active_particles);
ktot = ktot + fissionBank.size();

// Do some output
cout << "Source iteration: " << k << endl;
cout << "rough keff estimate = " << (double)(ktot)/
  (double)(batch_size*k) << endl;
cout << "track length keff estimate = " << keff_TL << ", uncertainty = "
  << sigTL << endl;
cout << "collision keff estimate = " << keff_Coll << ", unceratainty = "
  << sigColl << endl;
cout << "Top leakage estimate = " << topleak << ", uncertainty = " <<
  sigtop << endl;
cout << "Bottom leakage estimate = " << bottomleak << ", uncertainty = "
  << sigbottom << endl;
cout << "Shannon Entropy: " << ShannonEntropy[k-1] << endl;
cout << "Active cycle: " << l << endl;
cout << "Fission bank has " << fissionBank.size() << " neutrons."
  << endl;
```

```
      // Make Source Bank
      cout << "Making source bank from fission bank..." << endl;
      makeSource(fissionBank,sourceBank,batch_size);
      cout << "Source bank size = " << sourceBank.size() << endl << endl;
    }
    cout << "The pin pitch was " << pitch << endl;

    // Write to output file
    myfile << "Pin pitch = " << pitch << endl;
    myfile << "Active cycles: " << active_iters << endl;
    myfile << "Inactive cycles: " << inactive_iters << endl;
    myfile << "track length keff estimate = " << keff_TL << ", uncertianty = "
      << sigTL << endl;
    myfile << "collision keff estimate = " << keff_Coll << ", uncertianty = "
      << sigColl << endl;
    myfile << "Top leakage estimate = " << topleak << ", uncertainty = "
      << sigtop << endl;
    myfile << "Bottom leakage estimate = " << bottomleak << ", uncertainty = "
      << sigbottom << endl;
    myfile << endl;
/*
    spectrum << "Pin pitch = " << endl;
    spectrum << "Active cycles: " << active_iters << endl;
    spectrum << "Inactive cycles: " << inactive_iters << endl;
    spectrum << " Energy " << " fuel spectrum " << " moderator spectrum " \
      << endl;
    for(int g = 0; groups; g++)
    {
      spectrum << energyGrid[g] << fuelSpectrum[g] << modSpectrum[g] << endl;
    }
*/
  myfile.close();
//  spectrum.close();
  return 0;
}
```