Listed below is the code for the Monte Carlo Model. Several classes were created because they essentially functioned as data structures. The code for “exciton” imports previous classes.

**Explanation of Code**

**Class Boolean 3D**

This code created the 50 x 50 x 50 nm cube in which the exciton would hop/radiatively decay. Boolean 3D is one of the basic operations done for 3D models, and it sets and defines the cell on the grid. It also accounted for periodic boundary conditions.

**Int**, a fundamental type used to define variables holding numbers, set the specific cell on the grid and accounted for periodic boundary conditions.

void

Bool3D::set (int x,int y, int z, bool value)

{

int nx, ny, nz;

An important thing to realize is the **bool** data type, which declared where the exciton was in the cube based on values and coordinate being set as true (1) or false (1).

nx=x;

ny=y;

nz=z;

The nx, ny, and nz values served the purpose of accounting for the position of an element in the container. The first position of the exciton was 0.

if (nx>=Nbx) nx-=Nbx;

else if (nx<0) nx+=Nbx;

if (ny>=Nby) ny-=Nby;

else if (ny<0) ny+=Nby;

if (nz>=Nbz) nz-=Nbz;

else if (nz<0) nz+=Nbz;

The if-else statements determined the boundaries of the cubes, based on nx, ny, and nz.

if (vec[nz\*Nbx\*Nby + ny\*Nbx + nx]==true)

{

}

The vector operator **returns** to use the element at a specific position in the vector. It returns a reference to the element at position n in the vector container.

vec[nz\*Nbx\*Nby + ny\*Nbx + nx]=value;

return ;

}

**Class Quencher**

“double” initialized the presence of a decimal, in this case x2, y2, and z2 because it indicates that x2, y2, and z2 are decimal coordinate points. Class Quencher sets new coordinates for the quencher molecule (PCBM), checks for boundary conditions, and calculates the distances of the exciton from the center of the cube (a way to measure exciton diffusion length). This also randomized the position of the quencher molecule.

Brought in another class, global instance of random generator, configuration of the program, and the global\_folder which saved the output to a another location for analysis

extern ClassRandom Rnd;

extern Config \*config;

extern string global\_folder;

The next couple lines of code randomized the position of the quencher molecule within the morphology blend between the acceptor and the donor. It used a random position operator.

**Double** allowed the quencher to have more specific coordinates because it indicated that the position would be in decimals. This meant that x2, y2, and z2 would be decimal coordinate points.

**Class quencher::RandPos**

bool

ClassQuencher::RandPos ( double X, double Y, double Z )

{

x=X\*Rnd.dRnd();

y=X\*Rnd.dRnd();

z=Z\*Rnd.dRnd();

dCenter= sqrt ( pow(X/2-x,2) + pow (Y/2-y,2) + pow (Z/2-z,2) );

**pow** stands for power

dCenter was a value that represented the distance of the quencher molecule from the center. It utilized the basic 3D distance formula, \sqrt(x^2+y^2+z^2)

return true;

}

**ClassQuencher::dist** calculates the distance between the center of the quencher and quencher b. It served the purpose of examining how close the quencher molecules, and this was important because we did examine PCBM molecule aggregation in our project as well.

B was the other quencher molecule examined in comparison to quencher A.

The quencher molecule was shifted in the same way as before, and returned distance from quencher molecule A.

double

ClassQuencher::distance ( ClassQuencher& b, double X, double Y, double Z)

{

double x2, y2, z2;

x2 =b.x + X/2 - x;

y2 =b.y + Y/2 - y;

z2 =b.z + Z/2 - z;

Periodic boundary conditions using **if-else** statements.

if (x2>X) x2-=X;

else if (x2<0) x2+=X;

if (y2>Y) y2-=Y;

else if (y2<0) y2+=Y;

if (z2>Z) z2-=Z;

else if (z2<0) z2+=Z;

return sqrt ( pow(X/2-x2,2) + pow(Y/2-y2,2) + pow(Z/2-z2,2) ) ;

}

**ClassQuencher:: Set Coordinates**

This next set of code set new coordinates for a quencher, checked for boundary conditions, and recalculated the distance to the center dDistance.

void

ClassQuencher::SetCoordinates ( double x2, double y2, double z2, double X, double Y, double Z )

{

This code set the coordinates by allowing x, y, and z to equal x2, y2, and z2 after the boundary conditions were checked via **if-else** statements.

Power essentially squared whatever was in the brackets

if (x2>X) x2-=X;

else if (x2<0) x2+=X;

if (y2>Y) y2-=Y;

else if (y2<0) y2+=Y;

if (z2>Z) z2-=Z;

else if (z2<0) z2+=Z;

x = x2;

y = y2;

z = z2;

dCenter= sqrt ( pow(X/2-x,2) + pow (Y/2-y,2) + pow (Z/2-z,2) );

return ;

}

**Class Exciton**

This contained declarations of ClassExciton. There were several parts of this class, such as the constructor/generation of excitons and the movement of excitons in a random direction.

**Extern** pulled code from other classes.

Lifetime, which was measured in picoseconds, was designated as tau1. When the exciton was diffusing, it was determined to be **true**. When the exciton was quenched, it returned a **false** value.

**ClassExciton:: Class Exciton**

This constructed excitons.

extern ClassRandom Rnd;

ClassExciton::ClassExciton ()

{

lifetime = tau1;

x1=y1=z1=x2=y2=z2=x=z=y=0;

active = true;

return ;

**ClassExciton :: hop**

This moved excitons for hopsize in random directions (slower than the next class)

It is noted that the **void** type allowed us to define a function that does not return a value. It sets a value, but nothing needs to be returned. The aforementioned **double** function is used.

void

ClassExciton::hop (double bx, double by, double bz)

{

double theta, phi;

theta = Rnd.dRnd()\*PI-PI/2;

phi = Rnd.dRnd()\*2\*PI;

x = x + hopsize \* cos(theta) \* cos(phi);

y = y + hopsize \* cos(theta) \* sin(phi);

z = z + hopsize \* sin(phi);

Rnd.dRnd, which was extracted from another external file, puts a weight on the hopsize that depends on the composition of the medium.

if (x>bx) x-=bx;

else if (x<0) x+=bx;

if (y>by) y-=by;

else if (y<0) y+=by;

if (z>bz) z-=bz;

else if (z<0) z+=bz;

return ;

These were periodic boundary conditions of the medium that the exciton hopped through.

**Class Exciton :: hop2**

This moved excitons for hopsize in random directions. This used a different function called **metric** instead of the trigonometric functions in the previous code.

void

The **void** and **double** types , aforementioned, were used in this code for the coordinates.

ClassExciton::hop2 (double X, double Y, double Z)

{

double dx, dy, dz, metric;

dx = Rnd.dRnd()-0.5;

0.5 signifies the limitation of the exciton to hop, since it is in a 50 nm x 50 nm x 50 box. When you subtract this from the random generation, you ensure that the change in x,y, and z stay within the given area.

dy = Rnd.dRnd()-0.5;

dz = Rnd.dRnd()-0.5;

metric = sqrt (dx\*dx + dy\*dy + dz\*dz);

dx=hopsize\*dx/metric;

dy=hopsize\*dy/metric;

dz=hopsize\*dz/metric;

Metric is defined as the square root of the changes in the dx, dy, and dz. This calculates the movement of excitons in all 6 directions, giving it freedom.

x+=dx;

y+=dy;

z+=dz;

x2+=dx;

y2+=dy;

z2+=dz;

if (x>=X) x-=X; \

Defines periodic boundary conditions

else if (x<0) x+=X;

if (y>=Y) y-=Y;

else if (y<0) y+=Y;

if (z>=Z) z-=Z;

else if (z<0) z+=Z;

return ;

**Class Monte Carlo**

Created the global instance of random generator, configuration class, and the **global\_folder**, where all the output was saved for the whole execution so I could analyze it later.

This class contains declarations of ClassMonteCarlo

**ClassMonteCarlo :: Class Monte Carlo**

This class constructed ClassMonteCarlo.

extern ClassRandom Rnd;

extern Config \*config;

extern string global\_folder;

**ClassMonteCarlo :: ~ClassMonteCarlo**

This class destructed Class Monte Carlo.

**ClassMonteCarlo :: Init**

This class cleared up the instance of simulation.

**Class Monte Carlo :: Init2**

This class clears the array of photoluminescence decay, but keeps the medium the same.

**ClassMonteCarlo: Simulation**

This was the method of the Monte Carlo class. It used MaxGen, the maximum number of generations, to determine the duration of the calculation.

**ClassMonteCarlo :: Simulation G1**

This ran the simulation for one generation of excitons. This class was written to return the number of excitons that decayed radiatively, and the average displacements dL and dx in 3D and 1D.

**ClassMonteCarlo :: Save PL**

This class saved the array of PL data to the hard drive, so it could be extracted and analyzed later.

**Class Medium**

**Class Medium :: Class Medium**

This class constructed the ClassMedium.

**medium.cpp**

**Calculations from the Simulation**