**Diffusion.cpp**

This is the heart of the code. This is where all the classes are combined for the main() of exciton diffusion.

A **namespace** is a way to add a new group name to which you can add functions and other data that remain indistinguishable. What happens is that if you have two libraries that add a function give a different function but in naming they are indistinguishable.

using namespace std;

ClassRandom Rnd;

Config \*config;

string global\_folder;

double ClassExciton::hopsize;

double ClassExciton::a1;

double ClassExciton::tau1;

A static variable has to be declared globally to allocate memory. Class Exciton defined variables such as hopsize, a1, tau1 which were previously created.

double ClassExciton::a2;

double ClassExciton::tau2;

double ClassExciton::radius;

double ClassQuencher::radius;

double ClassQuencher::ASradius;

double ClassMonteCarlo::dT;

double ClassMedium::dX;

**int**, a fundamental type used to define variables holding numbers was employed.

int

main ( int argc, char \*argv[], char\* envp[])

{

ClassMonteCarlo \* mc;

int i;

The **unsigned int** serves the purpose of holding a larger positive value and no negative value.

The **struct** worked as a public object that contained variables inside that could be manipulated by me.

// int j;

int gen\_num = 1;

unsigned int radiative = 0;

double vFrac;

double Xbox,Ybox,Zbox;

// file name related definitions

char fname[256];

char cmd[512];

time\_t rawtime;

struct tm \* timeinfo;

So essentially, **experiment.cfg** was a the file that the parameters were entered into. I put my specifications such as photoluminescence decay into the experiment.cfg and the

char time\_str[100];

// output of several

double Output[50][4];

config = new Config("experiment.cfg",envp);

ClassExciton::a1 = config->pDouble("a1");; /\* weighting \*/

This imported all the classes that were formerly written, and named variables for usage later on.

ClassExciton::tau1 = config->pDouble("tau1");; /\* in ps \*/

ClassExciton::a2 = config->pDouble("a2");; /\* weitghing \*/

ClassExciton::tau2 = config->pDouble("tau2");; /\* in ps \*/

ClassExciton::radius = config->pDouble("Eradius");; /\* in nm \*/

The defining of variables that were used in this model, as part of the classes that were imported.

ClassQuencher::radius = config->pDouble("Qradius"); /\* fullerene radius in nm \*/

ClassQuencher::ASradius = config->pDouble("QASradius"); /\* quencher action sphere radius in nm \*/

ClassMonteCarlo::dT = config->pDouble("dT"); // time discretization in ps

ClassMedium::dX = config->pDouble("dX"); // time discretization in ps

gen\_num = config->pDouble("gen\_num"); // time discretization in ps

Xbox = config->pDouble("X"); // box dimensions [nm]

Ybox = config->pDouble("Y"); // box dimensions [nm]

Zbox = config->pDouble("Z"); // box dimensions [nm]

cout.setf(ios::fixed,ios::floatfield); // floatfield not set

My partner and I decided to run this program in a unix for Windows and Linux for Mac based way.

Folders were created that stored each iteration that was run every time we modified the experiment.cfg

cout.precision(7);

// create a folder for output

time (&rawtime);

timeinfo = localtime (&rawtime);

strftime (time\_str, 100, "%Y%m%d-%H.%M.%S-", timeinfo);

// folder in unix-like format

global\_folder = config->pString("outputFolder");

global\_folder += "/";

global\_folder += time\_str;

global\_folder += config->pString("subfolder");

//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Our code was written to have 4 experiments and possibilities. The first one, case 0, examined exciton diffusion without any quencher molecules. This would give us a smooth photoluminescence decay.

case 0 : // one run without quenchers

{

int num\_points;

double Ld2; // Simulation() returns RMS displacement

ClassExciton::hopsize = config->pDouble("hopsize") /\* in nm per 1 ps \*/

\* sqrt(ClassMonteCarlo::dT);

num\_points = config->pInt("num\_points");

sprintf(fname, "One run no quenchers");

cout << "\n\n\*\*\*\*\* New simulation \*\*\*\*\* \n" << fname << "\n" << endl;

This is what was displayed on screen.

This gave us the box dimensions from the previous classes, and the num\_gen of excitons.

mc = new ClassMonteCarlo; /\* create new simulation \*/

mc->Init(fname,Xbox,Ybox,Zbox, 0); /\* args: filename; box dimensions; volume fraction of quenchers \*/

radiative = mc->Simulation(gen\_num,&Ld2); number of generations \*/

This is our second part of the code. The hopsize of the exciton was kept constant, while the concentration of the PCBM quencher molecule was varied.

cout << "\n";

cout << "Number of radiatively decayd excitons: " << radiative << endl;

cout << "\n";

delete mc; /\* free memory \*/

}

break;

case 1 : // keep the hop size constant, vary the concentration

{

**Double** ensures that the variables and numbers and possibly decimals.

double vFrac\_start, vFrac\_end, vFrac\_inc, vFrac\_offset;

int num\_points;

double Ld2; // Simulation() returns RMS displacement

ClassExciton::hopsize = config->pDouble("hopsize") /\* in nm per 1 ps \*/

\* sqrt(ClassMonteCarlo::dT);

The concentration here is taken into 3 dimensions, by calculating the volume of the sphere of the quencher using **4/3\*PI\*pow (pow meaning squared).**

num\_points = config->pInt("num\_points");

if (config->pBool("useConcentrations")){

vFrac\_start = config->pDouble("conc\_start")\*( 4.0/3.0\*PI\*pow( (ClassQuencher::radius\*1E-7), 3) );

vFrac\_end = config->pDouble("conc\_end")\*( 4.0/3.0\*PI\*pow( (ClassQuencher::radius\*1E-7), 3) );

vFrac\_offset = config->pDouble("conc\_offset")\*( 4.0/3.0\*PI\*pow( (ClassQuencher::radius\*1E-7), 3) );

}else{

The volume of the quencher is given an initial and an ending value in the user-set parameters in experiment.cfg.

vFrac\_start = config->pDouble("vFrac\_start");

vFrac\_end = config->pDouble("vFrac\_end");

vFrac\_offset = config->pDouble("vFrac\_offset");

}

switch (config->pInt("spacing"))

{

case 1: // equally spaced on log scale

vFrac\_inc = (log(vFrac\_end) - log(vFrac\_start) ) / (num\_points-1);

break;

case 2: // equally spaced on the linear scale

vFrac\_inc =(vFrac\_end - vFrac\_start ) / (num\_points-1);

break;

default:

cout << "experiment.cfg: spacing is invalid \n";

exit(0);

}

for ( i = 0; i < num\_points; i += 1 )

{

if (config->pInt("spacing") == 1 ) // equally spaced on log scale

{

vFrac = exp( i\*vFrac\_inc + log(vFrac\_start) ) + vFrac\_offset;

}else if (config->pInt("spacing") == 2 ){ // equally spaced on the linear scale

vFrac = i\*vFrac\_inc + vFrac\_start + vFrac\_offset;

}

sprintf(fname, "Volume fraction %2.5f", vFrac);

cout << "\n\n\*\*\*\*\* New simulation #" << i << " \*\*\*\*\* \n" << fname << "\n" << endl;

mc = new ClassMonteCarlo; /\* create new simulation \*/

mc->Init(fname,Xbox,Ybox,Zbox, vFrac); /\* args: filename; box dimensions; volume fraction of quenchers \*/

radiative = mc->Simulation(gen\_num,&Ld2); /\* args: number of generations \*/

Output[i][0]=vFrac; /\* volume fraction \*/

//Output[i][1]=1 - (radiative\*1.0)/(GEN\*gen\_num); /\* relative quenchign efficiency \*/

Output[i][1]=vFrac/( 4.0/3.0\*PI\*pow( (ClassQuencher::radius\*1E-7), 3) ); /\* conc in cm -3 \*/

Output[i][2]=(radiative\*1.0); /\* proportional to lifetime \*/

Output[i][3]=(Ld2\*1.0); /\* <dL^2> \*/

cout << "\n";

cout << "Number of radiatively decayd excitons: " << Output[i][2] << endl;

cout << "\n";

delete mc; /\* free memory \*/

SaveTable(Output,i+1,"z.Radiative vs Amount of Quenchers.txt",true);

}

}

break;

case 2 : // vary hop size to get specific relative quenching efficiency

{

double max\_hopsize, min\_hopsize;

double targetQ;

double currentQ;

double diffQ;

double Ld2; // Simulation() returns RMS displacement

max\_hopsize = config->pDouble("max\_hopsize") \* sqrt(ClassMonteCarlo::dT);

min\_hopsize = config->pDouble("min\_hopsize") \* sqrt(ClassMonteCarlo::dT);

if (config->pBool("useConcentrations")){

vFrac = config->pDouble("Hopsize\_qConc")\*( 4.0/3.0\*PI\*pow( (ClassQuencher::radius\*1E-7), 3) );

}else{

vFrac= config->pDouble("Hopsize\_vFrac");

}

targetQ= config->pDouble("Hopsize\_targetQ");

currentQ = 0;

mc = new ClassMonteCarlo; /\* create new simulation \*/

mc->Init("Pre-Int",Xbox,Ybox,Zbox, vFrac); /\* args: filename; box dimensions; volume fraction of quenchers \*/

do

{

ClassExciton::hopsize = (max\_hopsize + min\_hopsize)/2;

sprintf(fname, "hopsize %2.5f", ClassExciton::hopsize/sqrt(ClassMonteCarlo::dT) );

cout << "\n\n\*\*\*\*\* New simulation \*\*\*\*\* \n" << fname << "\n" << endl;

printf("Try hopsize %2.5f ..... \n", ClassExciton::hopsize/sqrt(ClassMonteCarlo::dT) );

mc->Init2(fname); /\* change the simulation ID and clear the array PL[] \*/

radiative = mc->Simulation(gen\_num,&Ld2); /\* args: number of generations \*/

currentQ=1 - (radiative\*1.0)/(GEN\*gen\_num); /\* relative quenchign efficiency \*/

diffQ = targetQ-currentQ;

if (diffQ>0)

{

// increase hopsize

min\_hopsize = ClassExciton::hopsize;

}else {

// decrease hopsize

max\_hopsize = ClassExciton::hopsize;

}

printf("targetQ-currentQ = %2.4f \n", diffQ);

}while (fabs(diffQ) > 0.005 );

delete mc; /\* free memory \*/

cout << "\n\n HopSize = " << ClassExciton::hopsize/sqrt(ClassMonteCarlo::dT) << "\n";

}

break;

case 3 : // vary the quencher size to get specific relative quenching efficiency; hopsize and quencher conc are set

{

double Rmax, Rmin;

double targetQ;

double currentQ;

double diffQ;

double qConc;

double Ld2; // Simulation() returns RMS displacement

ClassExciton::hopsize = config->pDouble("qSize\_hopsize") \* sqrt(ClassMonteCarlo::dT);

Rmax = config->pDouble("qSize\_Rmax");

Rmin = config->pDouble("qSize\_Rmin");

qConc= config->pDouble("qSize\_qConc");

targetQ= config->pDouble("qSize\_targetQ");

currentQ = 0;

do

{

ClassQuencher::radius = (Rmin + Rmax) /2;

vFrac= qConc \* PI \* pow(ClassQuencher::radius\*1E-7,3)\*4.0/3.0 ;

sprintf(fname, "qRadius %2.5f", ClassQuencher::radius);

cout << "\n\n\*\*\*\*\* New simulation \*\*\*\*\* \n" << fname << "\n" << endl;

printf("Try quencher radius %2.5f ..... \n", ClassQuencher::radius );

printf("New volume fraction is %2.5f \n", vFrac );

{ // here an instance of ClassMonteCarlo is created

mc = new ClassMonteCarlo; /\* create new simulation \*/

mc->Init(fname,Xbox,Ybox,Zbox, vFrac); /\* args: filename; box dimensions; volume fraction of quenchers \*/

radiative = mc->Simulation(gen\_num,&Ld2); /\* args: number of generations \*/

currentQ=1 - (radiative\*1.0)/(GEN\*gen\_num); /\* relative quenchign efficiency \*/

diffQ = targetQ-currentQ;

if (diffQ>0)

{

// increase hopsize

Rmin = ClassQuencher::radius;

}else {

// decrease hopsize

Rmax = ClassQuencher::radius;

}

printf("targetQ-currentQ = %2.4f \n", diffQ);

delete mc; /\* free memory \*/

}

}while (fabs(diffQ) > 0.005 );

cout << "\n\n qRadius = " << ClassQuencher::radius << "\n";

}

We are still working on **case 4**, which keeps the hopsize of the exciton constant and varies the monoexponential decay time of the exciton.

We have not run this data yet.

break;

case 4 : // keep the hop size constant, vary the monoexponential decay time

{

double tau\_start, tau\_end, tau\_inc;

double Ld2; // Simulation() returns RMS displacement

int num\_points;

ClassExciton::tau2=0;

ClassExciton::a2=0;

ClassExciton::a1=1;

ClassExciton::hopsize = config->pDouble("tau\_hopsize") /\* in nm per 1 ps \*/

\* sqrt(ClassMonteCarlo::dT);

num\_points = config->pInt("tau\_num\_points");

tau\_start = config->pDouble("tau\_start");

tau\_end = config->pDouble("tau\_end");

if (config->pBool("useConcentrations")){

vFrac = config->pDouble("tau\_qConc")\*( 4.0/3.0\*PI\*pow( (ClassQuencher::radius\*1E-7), 3) );

}else{

vFrac = config->pDouble("tau\_vFrac");

}

switch (config->pInt("tau\_spacing"))

{

case 1: // equally spaced on log scale

tau\_inc = (log(tau\_end) - log(tau\_start) ) / (num\_points-1);

break;

case 2: // equally spaced on the linear scale

tau\_inc =(tau\_end - tau\_start ) / (num\_points-1);

break;

default:

cout << "experiment.cfg: spacing is invalid \n";

exit(0);

}

cout << "tau\_start = " << tau\_start << "\ttau\_end = " << tau\_end << "\t tau\_inc = " << tau\_inc << "\n";

mc = new ClassMonteCarlo; /\* create new simulation \*/

mc->Init("Pre-Int",Xbox,Ybox,Zbox, vFrac); /\* args: filename; box dimensions; volume fraction of quenchers \*/

for ( i = 0; i < num\_points; i += 1 )

{

if (config->pInt("tau\_spacing") == 1 ) // equally spaced on log scale

{

ClassExciton::tau1 = exp( i\*tau\_inc + log(tau\_start) );

}else if (config->pInt("tau\_spacing") == 2 ){ // equally spaced on the linear scale

ClassExciton::tau1 = i\*tau\_inc + tau\_start;

}

sprintf(fname, "tau=%2.5f", ClassExciton::tau1);

cout << "\n\n\*\*\*\*\* New simulation #" << i << " \*\*\*\*\* \n" << fname << "\n" << endl;

mc->Init2(fname); /\* args: filename; \*/

radiative = mc->Simulation(gen\_num,&Ld2); /\* args: number of generations \*/

Output[i][0]=ClassExciton::tau1; /\* tau \*/

Output[i][1]=(radiative\*1.0); /\* proportional to lifetime \*/

Output[i][2]=Ld2; /\* root mean square displacement \*/

Output[i][3]=Ld2/sqrt(6); /\* <dL^2> \*/

cout << "\n";

cout << "Number of radiatively decayd excitons: " << Output[i][1] << endl;

cout << "\n";

SaveTable(Output,i+1,"z.tau-radiative-dL2-Ld(1D).txt",false);

}

delete mc; /\* free memory \*/

}

break;

return EXIT\_SUCCESS;

}

**Diffusion.h**

#include <stdlib.h>

#include <string>

#include <fstream>

#include <vector>

#include "randomc.h"

//#define dT 0.1 /\* time discretization, ps \*/

#define TN 100000 /\* TN\*dT = total simulation time \*/

//#define dX 0.05 /\* space discretization [nm] for fullerene medium \*/

//#define QRADIUS 0.5 /\* radius of a quencher in nm \*/

//#define ERADIUS 0.5 /\* radius of the exciton \*/

#define GEN 10002 /\* number of exciton per generation. "+2": Two excitons are neglected per generation \*/

// #define FOLDER "output" /\* Folder name where the calculations are saved \*/

#define MC\_2D /\* define dimensionality of simulation \*/

#define PI 3.14159265358979323846 /\* PI \*/

using namespace std;

void

SaveTable ( double arr[][4], int n, string filename, bool UseDate );

/\* saves a two column table to hard drive \*/

================================================================

\* Class: Bool3D

\* Description: container of 3D bool array

\* ================================================================

class Bool3D

{

public:

Bool3D (int bx,int by, int bz) /\* constructor \*/

: Nbx(bx), Nby(by), Nbz(bz)

{ if(bx > 0 && by > 0 && bz > 0) { vec.resize(0); vec.resize(bx \* by \* bz, false); } }

~Bool3D(){vec.clear();}; /\* destructor \*/

const bool operator () (int x,int y, int z) const /\* index operator \*/

{ return vec[z\*Nbx\*Nby + y\*Nbx + x]; }

bool get (int x,int y, int z) /\* accesses the grid \*/

{ return vec[z\*Nbx\*Nby + y\*Nbx + x]; }

void set (int x,int y, int z, bool value); /\* set value to x,y,z \*/

int Nbx, Nby, Nbz; /\* dimensions of the array \*/

private:

vector<bool> vec; /\* vector<bool> takes 1 bit per item,

while array bool[] takes 1 BYTE per item \*/

}; /\* ----- end of class Bool3D ----- \*/

================================================================

\* Class: ClassQuencher

\* Description: Container of quencher coordinates

\* ================================================================

class ClassQuencher

{

public:

ClassQuencher () {x=y=z=dCenter=0; return;}; /\* constructor \*/

double x,y,z;

double dCenter; /\* distance to the center of the box

quenchers are sorted by this parameter

is used to speed up the quencher placement\*/

static double radius;

static double ASradius; /\* quencher action sphere radius \*/

bool operator< (const ClassQuencher &q2){ /\* is needed for sort() \*/

return dCenter<q2.dCenter;

}

double distance (ClassQuencher& b, double X, double Y, double Z); /\* calculate

distance between the center of

this qencher and quencher b. X,Y,Z - box size \*/

bool RandPos(double X, double Y, double Z); /\* randomize position of a quencher \*/

void SetCoordinates (double x2, double y2, double z2, double X, double Y, double Z);

/\* args: new coordinates, box size;

\* sets new coordinates for a quencher;

\* checks for boundary conditions\*/

protected:

private:

}; /\* ----- end of class ClassQuencher ----- \*/

================================================================

\* Class: Exciton

\* Description: Exciton with its properties

\* ================================================================

\*/

class ClassExciton

{

public:

ClassExciton (); /\* constructor \*/

double lifetime; /\* exciton lifetime \*/

double x1,y1,z1; /\* original coordinates \*/

double x2,y2,z2; /\* final coordinates, disregard the periodic boundary conditions.

Needed to calculate <dL^2>, <dx^2>, etc. \*/

double x,y,z; /\* current coordinates \*/

bool active; /\* determines if exciton is alive \*/

static double hopsize; /\* hopsize, parameter that we adjust \*/

static double a1; /\* weighting \*/

static double tau1; /\* radiative lifetime of an exciton \*/

static double a2; /\* weighting \*/

static double tau2; /\* radiative lifetime of an exciton \*/

static double radius; /\* exciton radius \*/

int nexti,previ; /\* array id of next and previous active exciton; \*/

void hop(double bx, double by, double bz); /\* hop for distance hopsize \*/

void hop2(double bx, double by, double bz); /\* hop for distance hopsize \*/

private:

\* Class: ClassMedium

\* Description: This class contains

\* \* 3D bool array for quenchers maps;

\* \* box size;

\* \* operations on the medium

\*

\* medium.cpp

\* ================================================================

\*/

class ClassMedium

{

public:

ClassMedium (double bx, double by, double bz); /\* does not allocate memroy for Bool3D \*/

ClassMedium (double bx, double by, double bz, double frac); /\* constructor

bx, by, bz are box size in [nm]

frac is volume fraction of quenchers\*/

~ClassMedium (); /\* destructor, frees memory for Bool3D \*/

bool TwoInterfaceQuenching (ClassExciton& e, double bx, double by, double bz);

/\* returns true if exciton is close to z=0 \*/

bool GridQuenching (Bool3D\* grid1, ClassExciton \* e);

/\* returns true if exciton is quenched on the grid \*/

double X; /\* box size in x-direction [nm], set by Init() to value bsize\*/

double Y; /\* box size in y-direction [nm], set by Init() to value bsize\*/

double Z; /\* box size in z-direction [nm], by Init() it is set to bsize\*/

double boxdx; /\* box discretization, set by Init, typically 0.05 nm

\_size/boxdx should be an integer !!! \*/

int Nx,Ny,Nz; /\* bsize/boxdx is used to allocate memory for media \*/

int numQ; /\* number of grid points which are set as ture \*/

Bool3D \* grid; /\* a pointer to a 3D boolean array. true==quencher; false==no quencher \*/

Bool3D \* ASgrid; /\* a pointer to a 3D boolean array. Action Sphere grid. \*/

static double dX; /\* space discrimination, read from config \*/

protected:

/\* ==================== DATA MEMBERS ======================================= \*/

private:

/\* ==================== DATA MEMBERS ======================================= \*/

ClassQuencher \* q; /\* array of quenchers \*/

int qn; /\* total number of quencher molecules in a simulation\*/

vector <int> gi, gj, gk; /\* grid points that overap with a quencher

that is placed at (0,0,0) \*/

vector <int> ASgi, ASgj, ASgk; /\* ASgrid points that overap with an action sphere

that is placed at (0,0,0) \*/

vector <double> clusterX, clusterY, clusterZ; /\* coordinates of

quenchers in a cluster \*/

vector <double> clusterD; /\* distance from the center \*/

int PixelSphere ( double qR, vector<int>& vi, vector<int>& vj, vector<int>& vk );

/\* Pixel Sphere sets the list of coordinates of a pixelated sphre of a given radius qR

\* to vector vi, vj, vk. Returns number of grid cells. \*/

bool PlaceHomogeneousQ(); /\* places quencher molecules to the grid in

homogeneous manner \*/

bool PlaceClustersQ(int num); /\* places clustered quencher molecules

to the grid. Argument is a number of

quenchers per cluster

\*/

bool ReadMediumBMP(const char\* folder); /\* reads medium from BMP files, stored

in specified folder. Files should be named

1.bmp, 2.bmp, 3.bmp etc. \*/

int CheckValidQposition(int a, ClassQuencher\* quencher);

/\* Check if the quencher overlaps

with others in the array q[0..a];

returns -2 if position is invalid, otherwise

returns index in array q[] after which

this quencher should be inserted\*/

int FindQposition(int a, ClassQuencher\* quencher);

/\* finds the index in the ordered array q[0..a]

after which new element quencher must be

inserted. \*/

void SetOnGrid(ClassQuencher \*quencher); /\* sets a quencher on the grid \*/

int SetCluster (int a, ClassQuencher\* centerQ, int num);

/\* a is the index of the last element in q[];

creates a cluster of size num at central position centerQ;

inserts new elements into q[] array;

sets the cluster on the grid;

returns index of the last element in updated q[];

DOES NOT CHECK overlaps. \*/

int ReadCrystal (); /\* reads position of quenchers in a crystal from

an ordered file. Returns number of lines read\*/

void ViewGridStats(); /\* prints statistics of the grid to stdout \*/

void Save2DgridBMP(Bool3D\* grid1, int zindex, char\* filename);

/\* saves 2D cut of the grid (0..Nx,0..Ny, zindex)

\* to bmp file \*/

}; /\* ----- end of class ClassMedium ----- \*/

================================================================

\* Class: ClassMonteCarlo

\* Description: This class will handle the entire MC simulation

\* ================================================================

\*/

class ClassMonteCarlo

{

public:

ClassMonteCarlo (); /\* constructor \*/

~ClassMonteCarlo (); /\* constructor \*/

string SimulationID; /\* is used in file names \*/

unsigned PL[TN]; /\* resulting photoluminescence \*/

unsigned int iDesired; /\* desired integrated PL intensity \*/

/\* determines the duration of calculation \*/

unsigned int

Simulation(unsigned int MaxGen, double\* ret\_dL2);

/\* Max-gen maximum number of exciton generations \*/

/\* determines the duration of calculation;

\* dL2 returns mean MRS displacement of excitons \*/

void

Init(string sID, double bx, double by, double bz, double frac);

/\* calls Init(); Initializes the medium with fullerens

\* frac - volume fraction of fullerens 0..1 \*/

void Init2(string sID); // clears array PL[], but does not touch the medium

void InitMedium(string sID); // clears array PL[], update medium

static double dT; // time discretization, read from config

private:

ClassMedium \*medium; /\* medium, in wich excitons diffuse \*/

unsigned int

SimulationG1(double& dL2, double& dx2, double& dL, double& dx); /\* Function that actually performs simulation.

Returns number of excitons that decayed radiatively.

and the average displacements dL and dx in 3D and 1D, respectively \*/

void

SavePL(unsigned \* arr, string filename); /\* saves PL decay to a file \*/