8-15-2011

Quantum Diffusion-Limited Aggregation

David Bradley Johnson
Butler University

Recommended Citation

Johnson, David Bradley, "Quantum Diffusion-Limited Aggregation" (2011). *Undergraduate Honors Thesis Collection*. 96.
https://digitalcommons.butler.edu/ugtheses/96

Quantum Diffusion-Limited Aggregation

A Thesis

Presented to the Department of Physics and Astronomy

College of Liberal Arts and Sciences

and

The Honors Program

of

Butler University

In Partial Fulfillment

of the Requirements for Graduation Honors

David Bradley Johnson

August 15, 2011

Abstract

Though classical random walks have been studied for many years, research concerning their quantum analogues, quantum random walks, has only come about recently. Numerous simulations of both types of walks have been run and analyzed, and are generally well-understood. Research pertaining to one of the more important properties of classical random walks, namely, their ability to build fractal structures in diffusion-limited aggregation, has been particularly noteworthy. However, nobody has yet pursued this avenue of research in the realm of quantum random walks.

The study of random walks and the structures they build has various applications in materials science. Since all processes are quantum in nature, it is very important to consider the quantum variant of diffusion-limited aggregation. Quantum diffusion-limited aggregation is an important step forward in understanding particle aggregation in areas where quantum effects are dominant, such as low temperature chemistry and the development of techniques for forming thin films.

Recognizing that the Schrödinger equation and a classical random walk are both diffusion equations, it is possible to connect and compare them. Using similar parameters for both equations, we ran various simulations aggregating particles. Our results show that the quantum diffusion process can create fractal structures, much like the classical random walk. Furthermore, the fractal dimensions of these quantum diffusion-limited aggregates vary between 1.43 and 2, depending on the size of the initial wave packet.

Contents

1   Introduction

Take an agar plate that has nutrients uniformly distributed across its surface but in a low concentration and put a single bacterium in the middle of the plate. Because the rate of fission of bacteria is related to the availability of nutrients, areas within the colony that have a high density of bacteria will grow slower due to there being fewer resources between the cells. In this way, thin lines of bacteria will grow and branch faster as seen in Fig. 1. Moreover, the colony can grow into a snowflake-like structure called a fractal, like in Fig. 2. Fractals exhibit a property called self-similarity, which means that the parts are like the whole, i.e. it is possible to see the same pattern at different magnifications. Fractals are found in more than just snowflakes or bacterial colonies. They can be found in clouds, river networks, fault lines, mountain ranges, crystals, lightning, and even vegetables.
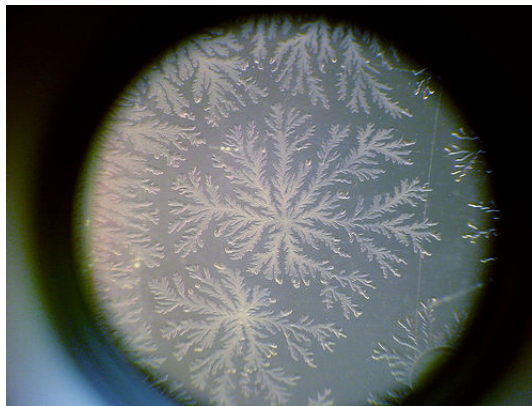

Figure 1: Bacterial Colony

The process for creating bacterial colony fractals described above is very similar to a process known as Diffusion-Limited Aggregation, which creates a Diffusion-Limited Aggregate (DLA). In a DLA, particles undergo some sort of random motion and are allowed to cluster together, forming a structure. Depending on the details, a fractal structure can be made as in Fig. 2. Computer simulations of DLA have been studied for many years leading to insights in various natural processes. For example, if the clustering property of a DLA is weakened by making aggregation less likely, the resulting structure will have a higher density.
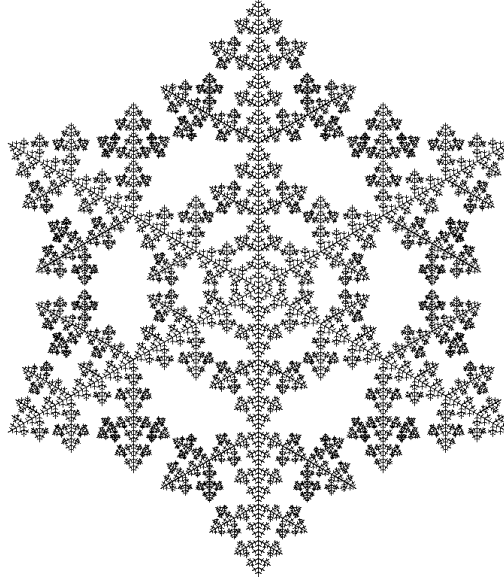
Figure 2: Snowflake Fractal

There can be several variations on the random motion that particles undergo in a DLA. However, it is only recently that there has been research into what happens when the rules of quantum mechanics govern the motion of a particle. Since all natural processes are truly quantum in origin, this would be an obvious next step to take in exploring the connection between DLA and nature. The objective of this study is to see how the structures generated by DLA are altered when the rules of quantum mechanics are incorporated into a DLA. This research could have application in areas such as the self-assembly of nanoparticles, thin film forming, and low temperature chemistry where quantum effects dominate.

2 Background

In 1905, Albert Einstein published four world changing papers. The first was about the photoelectric effect, which was fundamental in the development of quantum mechanics. The two most famous papers were about special relativity and matter/energy equivalence ($E = mc^2$). Perhaps the least well known paper was on Brownian motion, which was first observed as the random movement of particles suspended in a fluid. Einstein used the kinetic theory of fluids to explain that the fluid

consists of molecules that are numerous, invisible to the eye, and randomly moving in all directions, colliding with each other and the larger particles suspended in the fluid and therefore generating the random motion that is observed.

One way to model random motion is with something called a random walk (RW). A random walk is the trajectory resulting from taking successive random steps. The simplest example of a random walk is where you take a particle on a two dimensional square grid and at each time interval, have it randomly move up, down, left, or right with equal probabilities. A particle undergoing a random walk will meander around where it started, slowly spreading out covering a greater area with time. Random walks can be used to model the path of a foraging animal, stock prices, genetic drift, and, most notably, Brownian motion.

One important property of random walks is that they can be used in Diffusion-Limited Aggregation (DLA) to create fractals. To do a DLA simulation, consider the previous random walk example on a two dimensional grid and choose an arbitrary point on the grid labeling it the seed. Then, start a random walk sufficiently far from the seed on the grid. By adding a rule that says the randomly walking particle must stop when it comes next to the seed and become part of the structure, it is possible to aggregate particles. After sending out many particles one after the other, a fractal will begin to form in a process is called Diffusion-Limited Aggregation, as shown in Fig. 3.

Fractals built through DLA have been studied extensively. Fractals are so named because they share characteristics with objects in different dimensions, as if they exist in a fractional dimension say between the first and second dimension. This characteristic can be measured as the fractal dimension, which is a quantity that gives an indication of how completely a fractal fills a space at all scales of magnification. Ideal fractals have infinite detail and so their fractal dimensions can be calculated exactly and are usually non-integers. However, all fractals generated by DLA have finite detail and so they will have a trivial integral dimension when examined in the limit of infinite magnification. Instead, a variation of the fractal dimension must be used when examining a DLA. In this study, the mass dimension is used to calculate the fractal dimension of all structures generated via DLA.
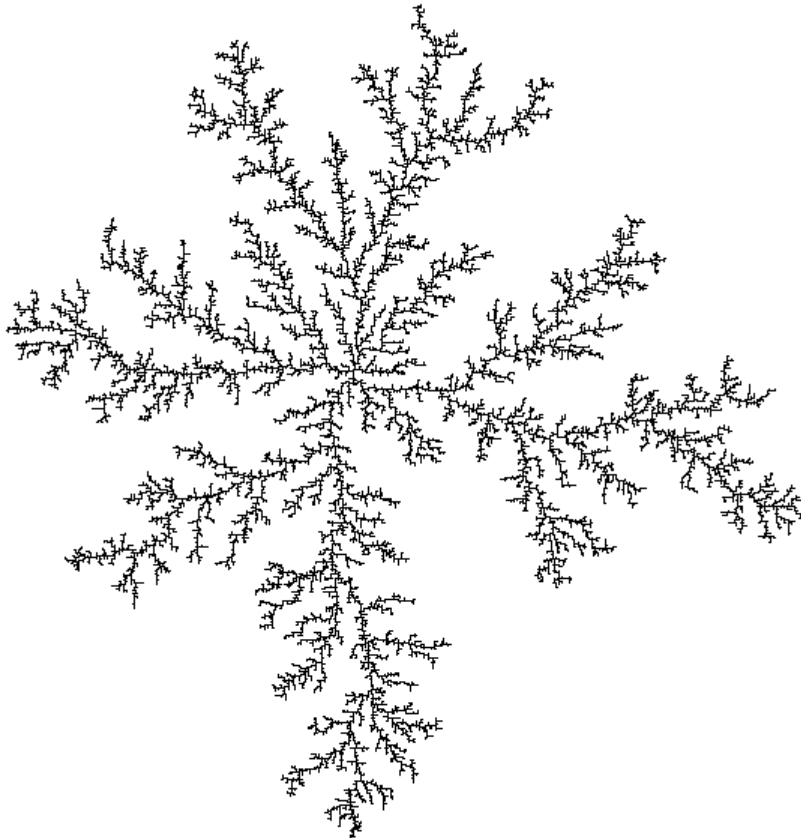
Figure 3: DLA made via Random Walk

Generally, the fractal dimension takes the form of a power law on some property of the fractal at different scales, where the exponent is the fractal dimension. When looking at finite structures such as those made via DLA, the fractal dimension obtained is only valid for a limited range of scales as shown in Fig. 4. In the figure, the arrowed line traces the limited range of scaling where the structure has a fractal dimension. The mass dimension assumes there is a power law relation between the radius from the center of the fractal r and the mass of the fractal within that radius $M(r)$ as in Eqn. (1) where d is the mass dimension and k is an arbitrary constant.

$$M(r) = k\, r^d \tag{1}$$

Recently, there has been work on a new kind of random walk which attempts to incorporate quantum mechanics, called the quantum random walk. In a Quantum Random Walk (QRW), the particle is in a superposition of positions instead of a single position like with the classical random
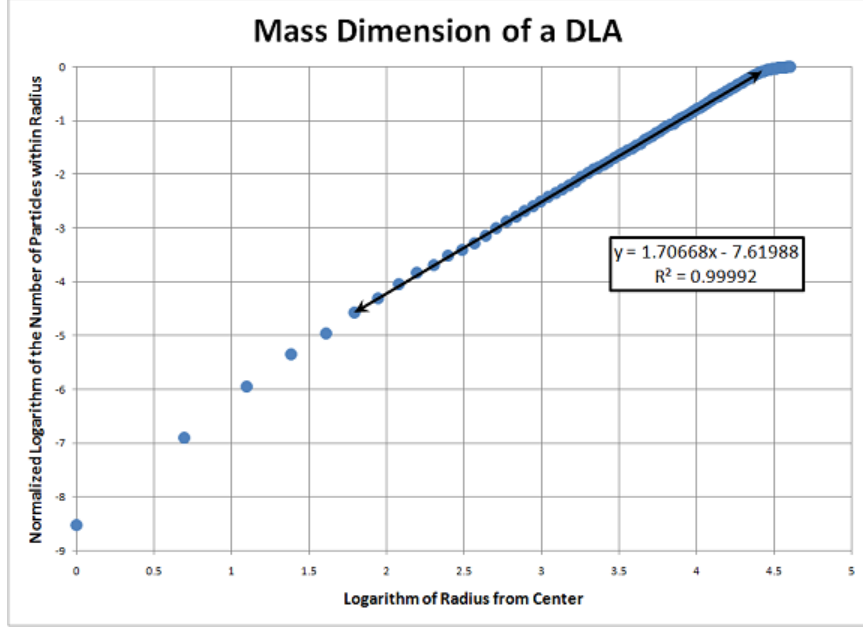
Figure 4: Log-Log Plot of the Mass within a Centered Circle vs. Radius

walk. The probabilities amplitudes for each position are then propagated in a wave-like fashion. Previous work [6] has shown that a quantum random walk is capable of producing fractals when used in a DLA. This work however only produced qualitative results and lacked a precise measurement of the fractal dimension of the structure formed by the quantum random walk. In this study, quantum random motion will be implemented using the Schrödinger equation instead of a quantum random walk for reasons discussed later.

3   Methodology

This study will compare the structures generated in a DLA where the particles follow classical versus quantum random motion.

3.1   Classical Diffusion-Limited Aggregation

To simulate the classical random motion of the particle, two different methods have been implemented: a random walk of a particle and a diffusion of probabilities (see reference for source of the idea to compare random walks to diffusion [1]). The equation of motion for a particle undergoing a

7

classical random walk in two dimensions can be written as Eqn. (2) where $\phi_{x,y}^t$ is the probability to find the particle at position (x, y) and time t. Likewise, $\psi_{x,y}^t$ is used to represent the complex-valued probability amplitude where $\psi_{x,y}^{t\ *}\psi_{x,y}^t$ gives the probability of to find the particle at position (x, y) and time t. As a random walk takes a step, the possible destination is evenly split between the four possible directions for a probability of one fourth in each direction. Likewise, the probability of the particle ending up in a given location is a quarter of the combined probability from all neighbor locations; this logic is captured in Eqn. (2).

$$\Phi_{x,y}^{t+1} = \frac{1}{4}(\Phi_{x+1,y}^t + \Phi_{x-1,y}^t + \Phi_{x,y+1}^t + \Phi_{x,y-1}^t) \tag{2}$$

The classical random walk equation is not different from the diffusion equation (3) when it is written in a numerical form (4). By choosing the right parameters, the original equation for a classical random walk (5) can be recovered from the diffusion equation. This means that the classical random walk is a diffusion process and that it can be modeled by a probability distribution via a diffusion equation [1, p. 44-3].

$$\frac{\partial \Phi}{\partial t} = D\nabla^2 \Phi \tag{3}$$

$$\frac{\Phi_{x,y}^{t+\Delta t} - \Phi_{x,y}^t}{\Delta t} = D\left(\frac{\Phi_{x+\Delta x,y}^t + \Phi_{x-\Delta x,y}^t - 2\Phi_{x,y}^t}{(\Delta x)^2} + \frac{\Phi_{x,y+\Delta y}^t + \Phi_{x,y-\Delta y}^t - 2\Phi_{x,y}^t}{(\Delta y)^2}\right) \tag{4}$$

$$\Phi_{x,y}^{t+1} = \Phi_{x,y}^t + \frac{1}{4}(\Phi_{x+1,y}^t + \Phi_{x-1,y}^t + \Phi_{x,y+1}^t + \Phi_{x,y-1}^t - 4\Phi_{x,y}^t)$$
$$\Delta x = 1, \Delta y = 1, \Delta t = 1, D = 1/4 \tag{5}$$

However, due to stability issues, it is not practical to numerically solve the diffusion equation using the parameters of equation (5). Instead, a different diffusion coefficient D is selected, which allows for stable solutions to be numerically computed. It is assumed that this does not affect the structures generated by aggregation.

8

3.2 Quantum Diffusion-Limited Aggregation

Schrödinger equation (6) is also a diffusion equation, except with an imaginary diffusion coefficient, D, and so can be compared to the traditional random walk. An explicit integration method (7) is used to solve Schrödinger Equation. Whereas this scheme is unstable for real diffusion coefficients, it was selected because, it becomes a stable method with the imaginary coefficient in Schrödinger Equation. This is achieved by using a formula that is symmetrical in both space and in time, the latter of which is not the case with Eqn. (4).

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m}\left(\frac{\partial^2 \Psi}{\partial x^2} + \frac{\partial^2 \Psi}{\partial y^2}\right) + V(x,y)\Psi \tag{6}$$

$$i\hbar \frac{\Psi_{x,y}^{t+\Delta t} - \Psi_{x,y}^{t-\Delta t}}{2\Delta t} = -\frac{\hbar^2}{2m}\left(\frac{\Psi_{x+\Delta x,y}^{t} + \Psi_{x-\Delta x,y}^{t} - 2\Psi_{x,y}^{t}}{(\Delta x)^2} + \frac{\Psi_{x,y+\Delta y}^{t} + \Psi_{x,y-\Delta y}^{t} - 2\Psi_{x,y}^{t}}{(\Delta y)^2}\right) \tag{7}$$

Since Schrödinger equation and a classical random walk are both diffusion equations, it is possible to connect and compare them. Two programs were written: one performing a classical diffusion and the other using Schrödinger equation. Similar parameters were used for the quantum simulation as for the classical, when running various simulations where particles were aggregated in a Quantum Diffusion-Limited Aggregation (QDLA).

Pietronero et al. [5] have obtained a theoretical value for the fractal dimension of structures created from DLA. They considered models where particles are aggregated with a probability $P(x,y) = \phi(x,y)^n$ where $\phi(x,y)$ obeys Laplace equation. They conclude that all such models will form a fractal for $0 \leq n \leq 2$ with a fractal dimension ranging from 2 to 1.43, respectively. Under stationary conditions, both the classical diffusion equation and Schrödinger equation are Laplace equations. For the classical DLA, n will equal 1 while n is 2 for the quantum DLA (permitting a complex $\phi(x,y)$). Therefore, it is expected that the QDLA will generate fractals much like the CDLA.

3.3   Implementation Details

A square grid was created with a single point in the center designated as the seed. Initially, a size of 256x256 was used for the grid but for later simulations, the grid was expanded to 512x512 to allow larger fractals to grow. The boundaries were set to be periodic (i.e. a torus) so that computational time was not wasted because a particle randomly leaves the grid and must be thrown away. Particles were released one at a time and allowed to run for a time period up to $T_{MAX} = 500,000$ before being discarded. This value was found experimentally by allowing a free particle in an empty grid to diffuse for a long time. When the sum of the probabilities for the particle grew significantly different from unity, the accumulated error from the numerical solution to the diffusion equations was deemed too great. A fraction of this time was selected for T_{MAX} to ensure the validity of the simulation.
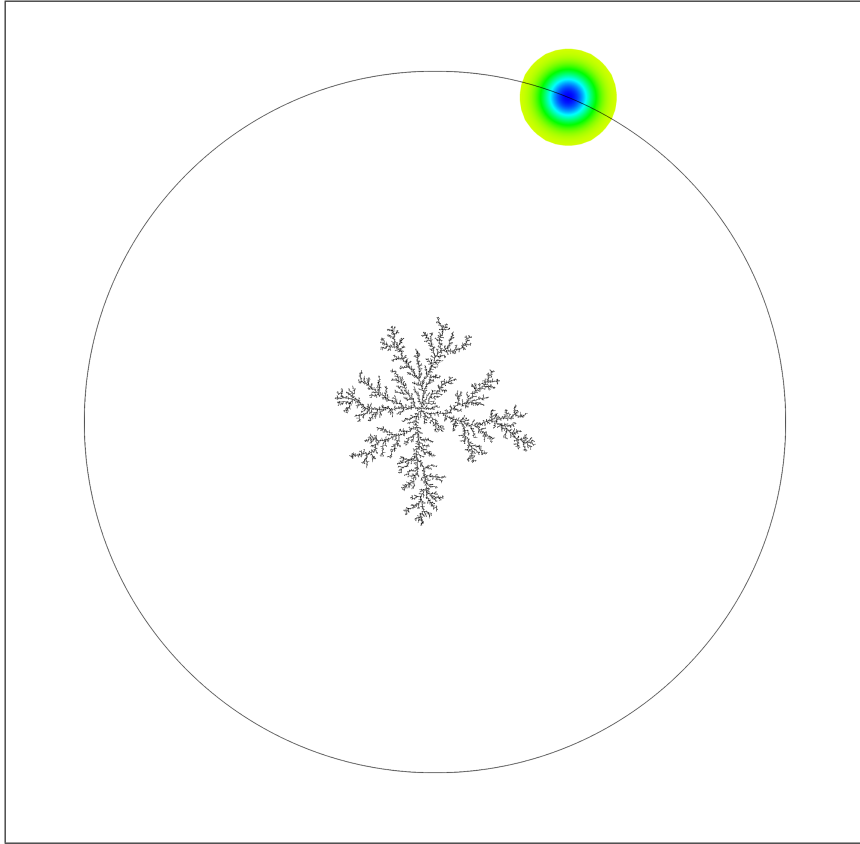

Figure 5: The Grid

Each particle was initialized as a 2D Gaussian distribution with the standard deviations of $\sigma_x = \sigma_y = 10$, which were arbitrarily selected. As with Sanbergs work [6], a starting distribution

10

that is too small will cause the particle to interfere with itself, generating waves due to grid effects. So, a larger particle must be selected to prevent this but it cannot be too large because the grid has a limited size and the particle must not start out interacting with the aggregated structure. In addition, it was noted by Kempe [2] that the starting condition of random walks can bias the particle in a single direction. This, in effect, gives the particle an initial velocity, which will alter the fractal dimension of the generated structure. These were all issues with the original QRW-based DLA [6] but are resolved in this study by using a 2D Gaussian distribution with no initial velocity. Every particle is placed so that it is centered randomly on the circumference of a circle centered on the seed as shown in Fig. 5. The circumference is as wide as possible while ensuring that the particle is at least one standard deviation away from the edge.
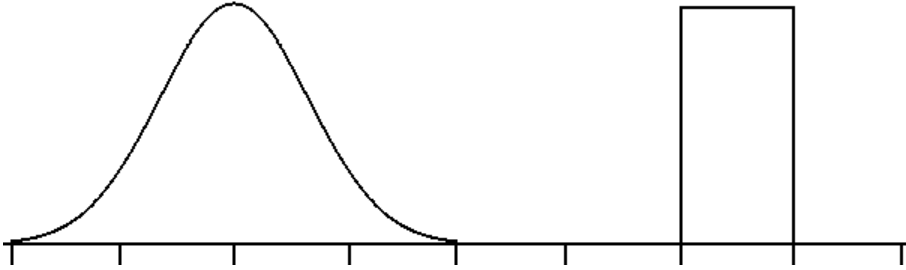

Figure 6: $T = 0$: Initial Wave Packet Far from Seed

As mentioned before, the time step must be less than one ($\Delta t < 1$) in order to manage the error in the numerical simulations. This requires special treatment of the propagation and detection of the particle. A 1D toy model is shown here to demonstrate the concept, which can be thought of as an exaggerated cross-section of the real simulation. When the total running time for the particle is zero ($T = 0$), the probability distribution of the particle should be sufficiently far from all parts of the DLA as shown in Fig. 6.

As time is incremented by the time step ($T_{new} = T_{old} + \Delta t$), the probability amplitudes (QDLA) or probabilities (CDLA) are erased at the grid locations where part of the structure is located as shown in Fig. 7 and Fig. 8. This, in effect, treats the seeds as infinite potentials where the probability of the particle entering them is zero. Consequently, the probabilities and probability amplitudes over the entire grid must be renormalized each time step.
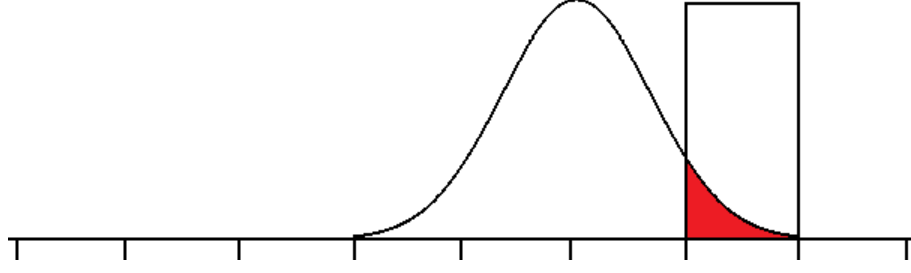
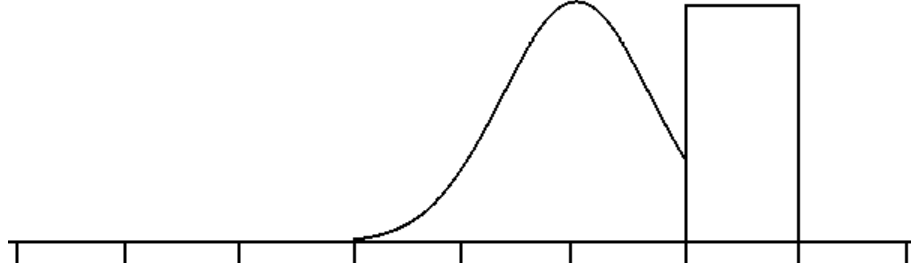Figure 7: $T = k\Delta t$: Updated Wave Overlapping with Seed

Figure 8: $T = k\Delta t$: Updated Wave Being Zeroed within Seed

Every n^{th} time step (where the time step was selected as $\Delta t = 1/n$), an attempt is made to detect the particle next to any of the seeds as in Fig. 9. If it is detected, the particle is localized to that position and another particle is released. If there is no detection, all locations next to seeds have the probability amplitudes or probabilities zeroed there (requiring renormalization again) as can be seen in Fig. 10. This is done because we know that the particle is definitely not at any of the locations examined.

Detection is handled the same way as in Sanbergs paper [6]. First, a pseudorandom number is generated between 0% and 100%. The calculated probability of each grid location that is adjacent to a part of the DLA is added to a running total until this sum exceeds the pseudorandom number that was just generated. The grid location that causes the sum to exceed the number is where the particle is aggregated. If the total probability does not ever exceed the number generated, there is no detection.

The simulations were all run at Butler University on the clustered supercomputer, BigDawg. BigDawg is comprised of several compute nodes, each containing four AMD 2.0 GHz quad-cores with 8GB of RAM that are all interconnected through an InfiniBand connection. The simulations
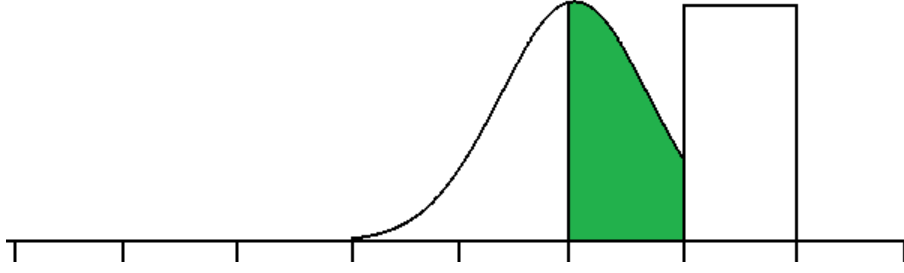
12

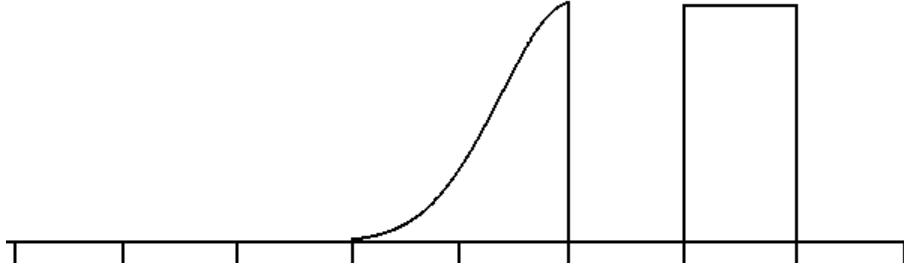Figure 9: $T = n\Delta t = 1$: Wave Being Tested for Detection

Figure 10: $T = n\Delta t = 1$: Wave Zeroed after Failing Detection

were written in C, using the Message Passing Interface (MPI) in order to leverage the parallel capabilities of BigDawg. Furthermore, multiple instances of each simulation were run in order to average the simulations and thus enhance the precision of the results.

Each simulation was restricted to a single compute node so that the 16 cores could share memory between them. By using shared memory, fewer MPI function calls were needed, thus localizing communication, which is good because traffic over InfiniBand is relatively slow compared to communication over a shared bus. The program was parallelized by dividing the rows of the grid between each of the cores and running the calculations in parallel. Besides needing to synchronize to ensure they remain in step together, the different cores avoided communication by relying on having concurrent read access to all needed memory. The only exception is when normalizing the wave function or performing a detection where minimal communication is necessary.

Detection and renormalization require the sum of probabilities over the entire grid be shared between all cores. This can be done sequentially but it was parallelized in order to speed up the calculation. Every core performs the sum for its section of the grid before using a special MPI function that sums and shares the values from all cores. For grid sizes such as 512x512, it was

much faster to parallelize this calculation than to have only one core perform it. If a particle is detected, a second pass over a fraction of the grid must be performed to actually determine which grid location the particle will be located. These techniques maximized parallelization and minimized communication, making the program as efficient as possible.

A utility program was written that finds the fractal dimension of a DLA. The program specifically finds the mass dimension by generating the data shown in Fig. 4. Clearly, it is not possible to just use all the data points in the graph to find the slope since the entire graph is not a straight line. Data points related to small radii suffer from grid effects, while larger radii skew the results because of the limited size of the DLA. The linear region within the curve must be identified so that its slope can be measured using least squares linear regression. Techniques developed by Kroll et al. [3] were used to have the program algorithmically determine the linear region instead of relying on human intuition. Then, the slope of the best fit line of the points within this linear region was used to calculate the mass dimension.

Unfortunately, it is not a simple matter to calculate the error of the mass dimension using these techniques. Although a least squares regression allows for calculation of an error for all terms of the best fit line, there is a much larger error from selecting different points within the linear region. Therefore, providing the standard error of the slope as the error of the fractal dimension is misleading. Instead, it is better to perform numerous simulations under the exact same parameters and then present the statistics over those.

4 Results

Because a comparison needs to be made between a classical and quantum generated DLA, this study makes the assumption that a classical random walk can be simulated as a diffusion equation without changing the resulting DLA. However, this assumption must be verified before continuing. According to Meakin [4], the fractal dimension of a two dimensional DLA generated via random walk is 1.69 ± 0.02. This number has been confirmed with the generation and analysis of fractals like the one in Fig. 3.
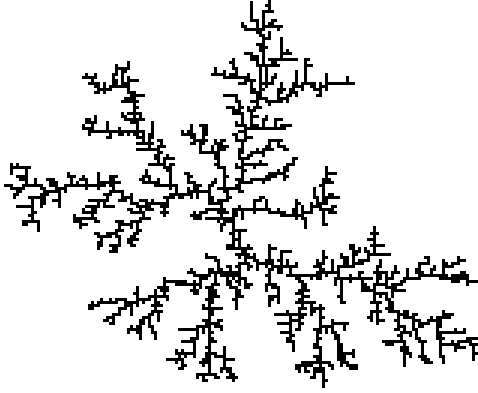
Figure 11: Fractal Generated by DLA using Diffusion Equation

Using identical parameters, 13 simulations of a classical DLA (CDLA) were performed. In a CDLA, a diffusion equation is used to govern the movement of the particle instead of a random walk. This was the first patch of runs, where the grid was of the size 256x256 and the particles started randomly on a circle of radius of 113. These particles were giving an initial Gaussian distribution with standard deviations $\sigma_x = \sigma_y = 10$. The time step Δt used was 0.05, which means there is an attempt at detection every $n = 20$ iterations and a diffusion constant of $D = 0.25$ was used. The result of these simulations is a fractal dimension of 1.67 ± 0.04, confirming that the choice of time step does not alter the generated fractals so long as the detection frequency maintains the relation $n = 1/\Delta t$. As shown in Fig. 11, it is possible to qualitatively confirm the result that fractals generated by a diffusion equation are no different than those made via random walk.

Using the same parameters as the CDLA, a quantum DLA (QDLA) simulation based on Schrödinger equation was performed in 13 identical simulations as well. In the case of the QDLA, there were two possible expectations. The first was that the particles would be capable of diffracting around the structure and thus will fill in the gaps between the branches of the fractal. This would lead to a fractal dimension closer to 2. The other possibility is that that diffraction does not occur and the classical squared probability amplitudes would dominate leading to a fractal dimension of 1.43 as predicted by Pietronero et al. [5]. From Sanbergs work [6], it is reasonable to expect that a fractal would be generated but the fractal dimension is unpredictable. However, the average fractal dimension of the QDLA simulations is 1.69 ± 0.03 as can be visually confirmed with Fig. 12. All
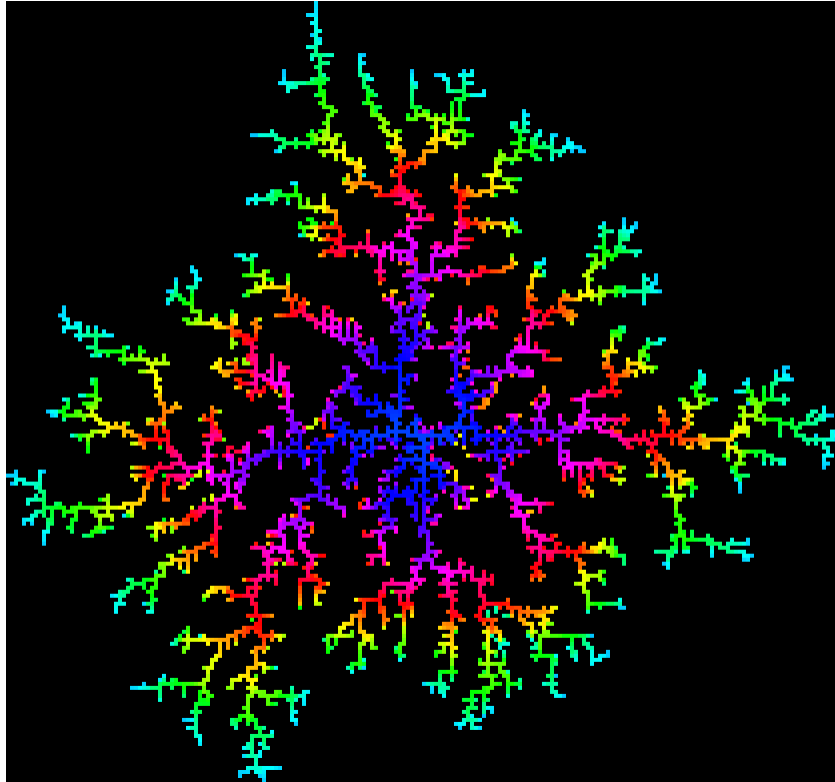
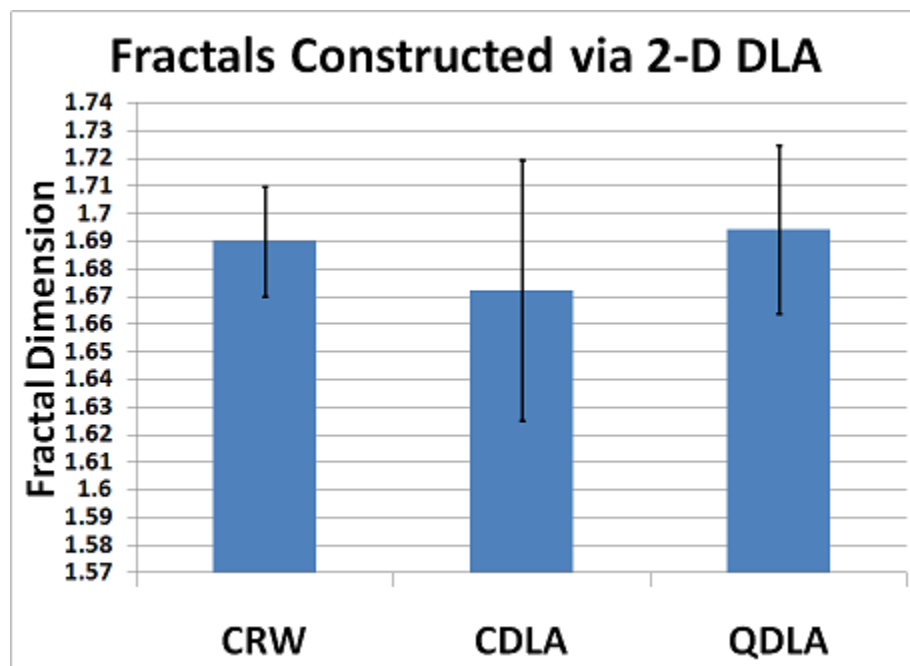Figure 12: Fractal Generated by QDLA

Figure 13: Average Fractal Dimensions for Several Types of 2D DLA

three types of simulations generated values very close to each other as shown in Fig. 13. This result was not expected and there is not an obvious explanation for why Schrödinger equation would create fractals of the same fractal dimension as a classical random walk.
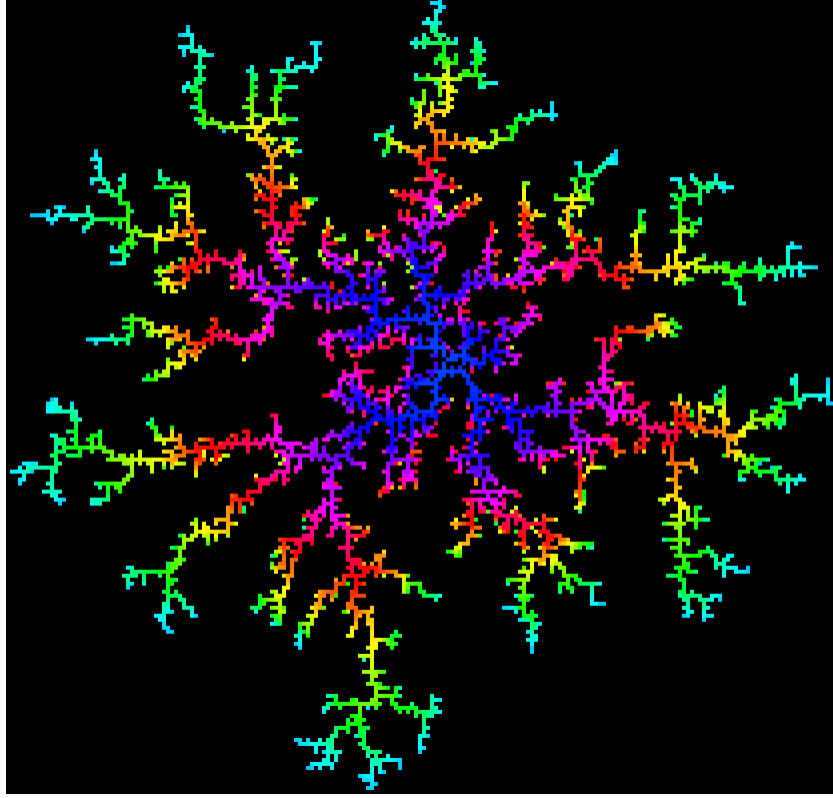

Figure 14: Fractal Generated by QDLA with Initial Wave Packet $\sigma = 16$ and $d = 1.45$

It is very peculiar that this study would come to such a conclusion and so these results were investigated, while participating in the StatPhys 24 Satellite Meeting in Tokyo in August 2010. By examining the wave function of the QDLA interacting with the structure, it was observed that at the boundary the particles probability amplitudes were interacting with the seeds just like how the classical diffusion equation did. The quantum particle was too spread out to be able to move between the branches. Therefore, the detections occurred in a similar fashion to the classical version. However, it was suspected that if there was a change made to the initial size of the Gaussian distribution used when initializing the particles, the particles would have different energies and thus be able to diffuse around the branches more easily. So, another set of simulations was started where all of the parameters were the same but the initial wave packet size changed and the grid size was

expanded to 512x512. Fig. 14 and Fig. 15 show that suddenly two very different types of fractals can result with such a change.
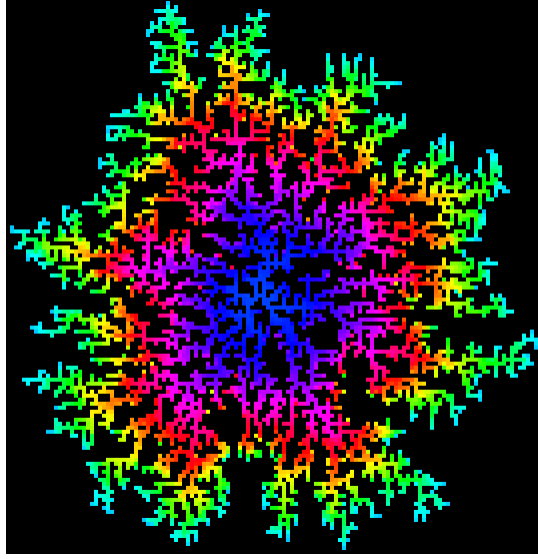

Figure 15: Fractal Generated by QDLA with Initial Wave Packet $\sigma = 1$ and $d = 1.91$

An additional 12 simulations were started on a 512x512 grid. Each simulation had a starting wave packet with a different size in an attempt to better characterize the relationship between the energy of the particle and the fractal dimension generated. One simulation was given a special initial configuration. There is a time invariant solution to Schrödinger equation in a grid with periodic boundaries, such that the particle starts with equal probability everywhere. This can be thought of as equivalent to a wave packet with infinite width. This is an important configuration to consider because the particle satisfies Laplace Equation when there is no seed present, which is a condition specified by Pietronero et al. [5]. It was expected that this run would approach the fractal dimension 1.43 that was specified.

The fractal dimension of all QDLA runs are shown in Fig. 16. The wave packet sizes are reduced by the size of the grid so that they can be compared fairly. Unfortunately, the infinite width simulation only aggregated 768 particles after running for months. From these simulations, it was learned that the larger the wave packet, the less likely it will detect and the longer it takes to grow a DLA of significant size. So, the three 512x512 simulations with the largest wave packets should

not be trusted as they did not have sufficient time to aggregate particles. Otherwise, the data seems largely consistent with some sort of curve.
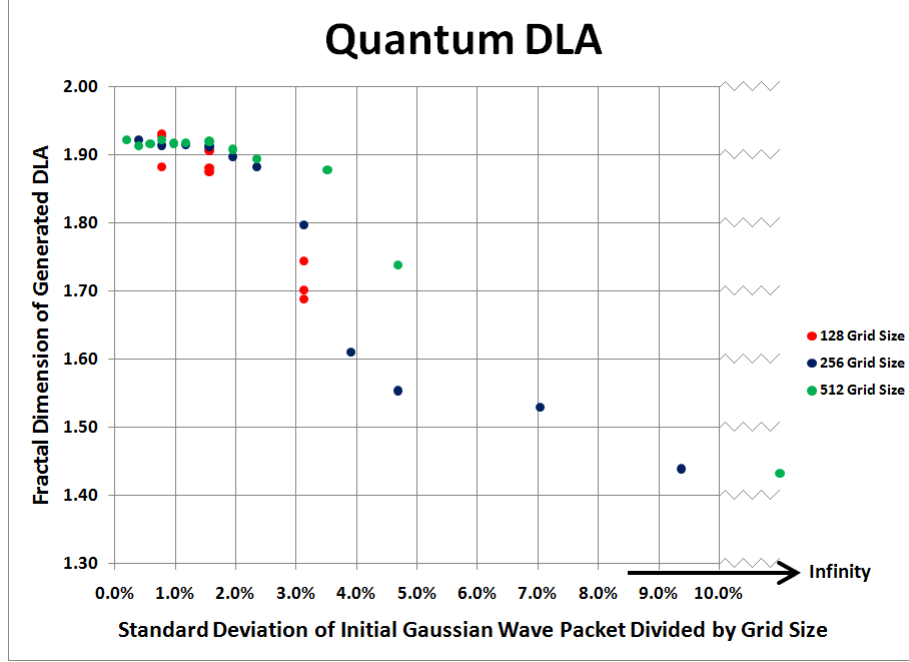

Figure 16: Fractal Dimension of Various Runs of 2D QDLA

5   Conclusion

The data indicates that a QDLA based on Schrödinger equation will indeed create fractals. Furthermore, it seems that depending on the initial width of the wave packet, a fractal dimension between 1.43 and 2 can be created. It is also interesting to note that these ranges have limits that are predicted by Pietronero et al. [5].

The growth of these diffusion equation based fractals was also investigated. For example, Fig. 17 shows the same fractal as Fig. 11, except it is color coded to show the relative ages of different regions with the fractal. The regions get progressively younger as the radius increases. Typically, no detections occur between the branches because the particle is too big and is deflected away by the tips of the branches.

For future work on this project, it is worth taking the time to better filling in the curve of Fig. 16.
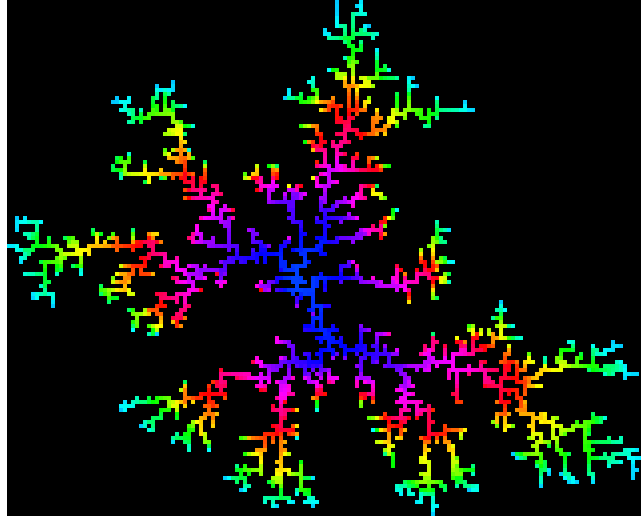
Figure 17: Fractal Generated by CDLA Showing the Relative Age of Different Regions

It is suspected that there is some sort of inflection point where the tunneling of high energy particles is in equilibrium with the deflection that occurs with low energy particles. It would be interesting to research the meaning of such a point, if it exists. It is important to not only fill in the curve but to also use an average of runs with identical starting conditions to determine the characteristic fractal dimension as well as to provide error bars. This problem is well suited to the framework of a graphics processing unit (GPU) such as NVIDAs Compute Unified Device Architecture so perhaps that may be best hardware to use for those long simulations in the future.

6 Acknowledgements

I would like to thank the many people who have supported me throughout this project. I owe my deepest gratitude to my thesis advisor and fellow researcher, Dr. Ordóñez, who inspired the topic and was there supporting me through the entire process. I would like to thank Bob Holm, the Butler Institute for Research and Scholarship, and the Butler Summer Institute for funding and supporting this research when I first began.

I am extraordinarily grateful to Dr. Hatano, Dr. Petrosky, and the University of Tokyo for facilitating my trip to the International Workshop on Statistical Physics of Quantum Systems,

References

[1] H. M. Dixon. *Path Integrals in Field Theory and Statistical Mechanics*, volume 6 of *Introduction to Natural Philosophy Logic in Physical Science*. Butler University, August 2007.

[2] J. Kempe. Quantum random walks - an introductory overview. *Contemporary Physics*, 44:0303081, 2003.

[3] M H Kroll and K Emancipator. A theoretical evaluation of linearity. *Clin Chem*, 39(3):405–13, 1993.

[4] Paul Meakin. Diffusion-controlled cluster formation in two, three, and four dimensions. *Phys. Rev. A*, 27(1):604–607, Jan 1983.

[5] L. Pietronero, A. Erzan, and C. Evertsz. Theory of fractal growth. *Phys. Rev. Lett.*, 61(7):861–864, Aug 1988.

[6] Colin F. Sanberg. Implementing quantum random walks in two-dimensions with application to diffusion-limited aggregation. *Undergraduate Honors Thesis Collection*, 14, May 2007.

A Code

```c
1 /**
2  *  QDLA.c
3  *
4  *  Quantum Diffusion Limited Aggregation
5  * David Johnson
6  *  Butler University
7  * 15 August 2011
8 **/
9
10 //To turn off all MPI commands at once.
11 #define PARALLEL
12 //Turn on in order to debug a program.
13 #define NODEBUG
14
15 //To experiment with the accuracy and speed of other data types.
16 #define dataType long double
17
18 #include <stdlib.h> //exit()
19 #include <stdio.h> //printf()
20 #include <string.h> //strcat()
21 #include <math.h> //cos(), sin(), exp()
22 #include <time.h> //seeding random number (it gives very weird error messages without this)
23 #include <fcntl.h> //shm_open, O_CREAT, O_RDWR
24 #include <sys/stat.h> //S_IRUSR and S_IWUSR
25 #include <sys/mman.h> //mmap, munmap, PROT_READ, PROT_WRITE, MAP_SHARED, and MAP_FAILED
26
27 #ifdef PARALLEL
28 #include "mpi.h" //allows parallel processing
29 #endif
30
31 void run(void);
32 void freeArray(char *, void *, int, int);
33 void *createArray(char *, int, int);
34 void functionNew(dataType *, dataType *, dataType *, dataType *, dataType *, dataType *);
35 void functionOld(dataType *, dataType *, dataType *, dataType *);
36 int detect(dataType *, dataType *, char *);
37 void borders(dataType *);
38 void normalize(dataType *, dataType *);
39 void writePsi(char *, dataType *, dataType *);
40 void writeSeed(int, int, double);
41 void init(dataType *, dataType *);
42 void seedBorders(char *);
43 int seedInit(char *);
44 int seedStart(int, int);
45 void centerParticle(long double);
46
47 #define PI 3.1415926535897932384626433832795
48 #define TRUE 1
49 #define FALSE 0
50 #define DIGITS 16
51 #define NODE_SIZE 16
52
53 #define NO_SEED 0
54 #define RANDOM_SEED 1
55 #define DOT_SEED 2
56 #define WALL_SEED 3
57
58 #define height 512
59 #define width 512
60 #define radius 1
61 #define adj (width + 2*radius)
62
63 #define timeStep ((dataType)0.05)
64 #define totalTime ((dataType)500000.0)
65 #define totalSteps ((long)10000000)
66 #define detectFrequency 20
67 #define writeFrequency 0
68
69 #define diffusionRate ((dataType)0.25)
70 #define hbar ((dataType)1.0)
71 #define mass ((dataType)1.0)
72
73 #define particleSize ((dataType)0.0)
74
75 char *directory = "run11"; //sub-directory where files will be saved to.
76 char *extension = "csv"; //extension of each file for file association ease.
77
78 const int numPackets = 1; //number of gaussian wave packets that a particle is initially split up
       into.
```

```c
79
80  dataType   yCenter = height/2-0.5; //y position of center of initial condition.
81  dataType   xCenter = width/2-0.5; //x position of center of initial condition.
82
83  const int      seedType = DOT_SEED; //determines which type of initial seeding configuration is
          used.
84  const int      particleCirclesSeed = TRUE; //boolean that determines if the initial particle
          position is ignored or is set to circle the seed.
85  const dataType  particleMinRadius = height/2-30.0; //the starting distance between the initial
          positions of particles and the seed start.
86  const dataType particleMaxRadius = height/2-30.0; //the ending distance between the initial
          positions of particles and the seed start.
87  const int      seedCenterY = height/2; //y coordinate of the center of the seed.
88  const int      seedCenterX = width/2; //x coordinate of the center of the seed.
89  const int      seedRadius = 0; //radius of the seed.
90  const int      particles = 25000; //the number of particles to be sent out.
91
92  double timerStart;
93  int id, np, node, totalNodes, *yMin, *yMax;
94  int x, y; // Only used on process zero for detection.
95
96  #ifdef PARALLEL
97  MPI_Comm comm;
98  #endif
99
100 int main(int argc, char *argv[])
101 {
102   int a, b, dif, timeSeed;
103   char *temp, *temp1;
104   time_t rawTime, seconds1, seconds2;
105   struct tm *timeInfo;
106   struct stat st;
107   FILE *file;
108
109   time(&seconds1);
110
111   //if program is not parallel, then there is only 1 process and it has id 0 and is on node 0
112   id = 0;
113   np = 1;
114   node = 0;
115
116   //seeds the random numbers
117   time(&rawTime); //the c version of getting a time object (with updated info)
118   timeInfo = localtime(&rawTime); //the c version of getting the info about the time object
119   srand((*timeInfo).tm_sec); //seeds the random number generator with an int between 0 and
          RAND_MAX
120   timeSeed = rand();
121
122 #ifdef PARALLEL
123   long double timer = MPI_Wtime(); //Gets start time of program (according to MPI so might have
          more sig figs)
124   MPI_Init(&argc, &argv); //imitialize MPI
125   MPI_Comm_rank(MPI_COMM_WORLD, &id); //Gets process id# from the world comm
126   MPI_Comm_size(MPI_COMM_WORLD, &np); //Gets total number of processes that are executing this
          program
127
128
129   //We do this before splitting up the communicator so that the same id's on different nodes will
            have different seeds for rand()
130   MPI_Bcast(&timeSeed, 1, MPI_INT, 0, MPI_COMM_WORLD);
131   srand(timeSeed);
132
133   for (a = 0; a <= id; a++) //ensures that different processes will be using uncorrelated
          pseudorandom numbers
134     srand(rand());
135
136   //Splits all the processes into their own node so each node can run independently but still be
          executed at the same time
137   node = id/NODE_SIZE;
138   totalNodes = (np - 1) / NODE_SIZE + 1;
139
140   MPI_Comm_split(MPI_COMM_WORLD, node, id, &comm); //an awesome command that does all the comm
          construction for me
141   MPI_Comm_rank(comm, &id); //Gets process id# from mpi
142   MPI_Comm_size(comm, &np); //Gets total number of processes that are executing this program
143
144   /**
145    *  IMPORTANT NOTE: Do not think any process has its original ID anymore. They
146    *  have all just been changed according to the node that they are running on.
147    *  They also got a new MPI_Comm ojbect that distinguishes processes on
148    *  different nodes. So, do NOT use MPI_COMM_WORLD anymore! Use the globally
```

```
149    *   defined MPI_Comm object called "comm" instead. Or else, a call to
150    *   MPI_Barrier(MPI_COMM_WORLD) would cause all processes to wait even though
151    *   the ones on different nodes are doing something unrelated. This mistake
152    *   could cause the program to crash unexpectedly with no explanatory message
153    *   about it so it is important to keep an eye out for it.
154    **/
155 #endif
156
157 #ifdef PARALLEL
158   if (totalNodes > 1)
159   {
160     temp = (char *)calloc((strlen(directory)+5), sizeof(char));
161     sprintf(temp, "%s%d", directory, node);
162     directory = temp;
163   }
164 #endif
165
166 #ifdef DEBUG
167   if (id == 0)
168     printf("Node%d: %s\n", node, directory);
169 #endif
170
171   //Create Directory or else an exception might be thrown.
172   if (id == 0 && stat(directory, &st) != 0)
173   {
174     printf("Making directory: %s\n", directory);
175     mkdir(directory, S_IRWXU);
176   }
177
178   if (id == 0)
179   {
180     //Erase seed file.
181     temp = (char *)calloc((strlen(directory)+strlen(extension)+10), sizeof(char));
182     sprintf(temp, "%s/seed.%s", directory, extension);
183     file = fopen(temp, "w");
184     fprintf(file, "detections,particles,y,x,particle time,real time\n");
185     fclose(file);
186     free(temp);
187   }
188
189   run();
190
191 #ifdef DEBUG
192   if (id == 0)
193   {
194 #ifdef PARALLEL
195     timer = MPI_Wtime() - timer;
196     dif = (int)timer;
197
198     printf("Node%d: Total time on %d cores: %d hours, %d minutes, %LG seconds.\n",
199           node, np, ((int)(dif/60))/60, ((int)(dif/60))%60, timer-(dif-dif%60));
200 #else
201     time(&seconds2);
202     dif = seconds2 - seconds1;
203
204     printf("Node%d: Total time on %d cores: %d hours, %d minutes, %d seconds.\n",
205           node, np, ((int)(dif/60))/60, ((int)(dif/60))%60, dif%60);
206 #endif
207   }
208 #endif
209
210 #ifdef PARALLEL
211   MPI_Barrier(MPI_COMM_WORLD);
212   MPI_Finalize(); //shuts down all mpi commands for each process
213 #endif
214
215   return 0;
216 }
217
218 void run(void)
219 {
220   dataType *newPsiR, *newPsiI, *oldPsiR, *oldPsiI, *olderPsiR, *olderPsiI, *temp;
221   char *seeds;
222   char *aString;
223   int a, b, d, min, max, loop = 1, detections;
224   long c;
225   time_t seconds; //Only used by non-mpi code.
226
227   aString = (char *)calloc(50, sizeof(char));
228
229   yMin = (int *)calloc(np, sizeof(int));
```

```
230    yMax = (int *)calloc(np, sizeof(int));
231
232    min = 0;
233    for (a = 0;a < np;a++)
234    {
235      max = min+((height-1)/np)-1;
236      if (a <= ((height-1)%(np)))
237        max++;
238
239      yMin[a] = min;
240      yMax[a] = max;
241
242      min = max+1;
243    }
244
245 #ifdef DEBUG
246    if (id == 0 && node == 0)
247      printf("Initialized Seeds and Constants.\n");
248 #endif
249
250 #ifdef PARALLEL
251    //calls createArray function which creates shared memory array and connects each process to it.
252    newPsiR = (dataType *)createArray("newPsiR", (height+2*radius)*(width+2*radius), sizeof(
           dataType));
253    newPsiI = (dataType *)createArray("newPsiI", (height+2*radius)*(width+2*radius), sizeof(
           dataType));
254    oldPsiR = (dataType *)createArray("oldPsiR", (height+2*radius)*(width+2*radius), sizeof(
           dataType));
255    oldPsiI = (dataType *)createArray("oldPsiI", (height+2*radius)*(width+2*radius), sizeof(
           dataType));
256    olderPsiR = (dataType *)createArray("olderPsiR", (height+2*radius)*(width+2*radius), sizeof(
           dataType));
257    olderPsiI = (dataType *)createArray("olderPsiI", (height+2*radius)*(width+2*radius), sizeof(
           dataType));
258    seeds = (char *)createArray("seeds", (height+2*radius)*(width+2*radius), sizeof(char));
259 #else
260    newPsiR = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
261    newPsiI = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
262    oldPsiR = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
263    oldPsiI = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
264    olderPsiR = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
265    olderPsiI = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
266    seeds = (char *)calloc((height+2*radius)*(width+2*radius), sizeof(char));
267 #endif
268
269 #ifdef PARALLEL
270    //Start timer.
271    timerStart = MPI_Wtime();
272 #else
273    time(&seconds);
274    timerStart = seconds;
275 #endif
276
277    detections = seedInit(seeds);
278
279 #ifdef DEBUG
280    if (id == 0 && node == 0)
281      printf("Created and linked the all arrays.\n");
282 #endif
283
284    for (d = detections; d < particles; d++)
285    {
286      if (particleCirclesSeed)
287      {
288        if (particles != 0)
289          centerParticle(((long double)d)/particles);
290        else
291          centerParticle(0);
292      }
293
294      //Set initial conditions.
295      init(olderPsiR, olderPsiI);
296
297      for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
298        for (b = a+radius; b < a+width + radius; b++)
299          if (seeds[b] == TRUE)
300          {
301            olderPsiR[b] = 0.0;
302            olderPsiI[b] = 0.0;
303          }
304      normalize(olderPsiR,olderPsiI);
```

```
305
306 #ifdef PARALLEL
307     MPI_Barrier(comm);
308 #endif
309
310     if (id == 0 && writeFrequency != 0) //Write initial condition to file.
311     {
312       sprintf(aString, "%d 0", d);
313       writePsi(aString, olderPsiR, olderPsiI);
314     }
315
316     //Finds the second initial condition based off of the first.
317     functionOld(olderPsiR, olderPsiI, oldPsiR, oldPsiI);
318
319 #ifdef PARALLEL
320     MPI_Barrier(comm);
321 #endif
322
323     //Write 1st Time Step to file.
324     if (id == 0 && writeFrequency != 0)
325     {
326       sprintf(aString, "%d %.*LG", d, timeStep, DIGITS);
327       writePsi(aString, oldPsiR, oldPsiI);
328     }
329
330 #ifdef DEBUG
331     if (id == 0 && node == 0)
332       printf("Starting Particle Run.\n");
333 #endif
334
335     c = 0;
336     loop = TRUE;
337
338     do //Iterates through all of the timeSteps until the totalTime is reached.
339     {
340       functionNew(olderPsiR, olderPsiI, oldPsiR, oldPsiI, newPsiR, newPsiI);
341
342       for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
343         for (b = a+radius; b < a+width + radius; b++)
344           if (seeds[b] == TRUE)
345           {
346             newPsiR[b] = 0.0;
347             newPsiI[b] = 0.0;
348           }
349
350       normalize(newPsiR, newPsiI);
351
352 #ifdef DEBUG
353       if (id == 0 && node == 0 && c % 100 == 0)
354         printf("Time: %.*LG\n", c * timeStep, DIGITS);
355 #endif
356
357 #if writeFrequency >0
358
359 #ifdef PARALLEL
360       MPI_Barrier(comm);
361 #endif
362       if (id == 0 && c % writeFrequency == 0)
363       {
364         sprintf(aString, "%d %.*LG", d, c * timeStep, DIGITS);
365         writePsi(aString, newPsiR, newPsiI); //write newPsi to a file
366 #ifdef DEBUG
367         printf("Wrote File: \"Particle %s.%s\"\n", aString, extension);
368 #endif
369       }
370 #endif
371
372 #if detectFrequency >0
373
374 #ifdef PARALLEL
375       MPI_Barrier(comm);
376 #endif
377       if (c % detectFrequency == 0)
378         if (detect(newPsiR, newPsiI, seeds))
379         {
380           loop = FALSE;
381           detections++;
382           writeSeed(d, detections, (double)(c * timeStep));
383         }
384 #endif
385
```

```
386        if (totalSteps != 0 && c > totalSteps)
387          loop = FALSE;
388
389        borders(newPsiR);
390        borders(newPsiI);
391
392        //each psi gets moved backwards as time just progressed forward.
393        temp = olderPsiR;
394        olderPsiR = oldPsiR;
395        oldPsiR = newPsiR;
396        newPsiR = temp;
397
398        temp = olderPsiI;
399        olderPsiI = oldPsiI;
400        oldPsiI = newPsiI;
401        newPsiI = temp;
402
403        c++;
404      } while (loop);
405    }
406
407 #ifdef DEBUG
408    if (id == 0 && node == 0)
409      printf("Finished Calculation.\n");
410 #endif
411
412 #ifdef PARALLEL
413    freeArray("newPsiR", newPsiR, (height+2*radius)*(width+2*radius), sizeof(dataType));
414    freeArray("newPsiI", newPsiI, (height+2*radius)*(width+2*radius), sizeof(dataType));
415    freeArray("oldPsiR", oldPsiR, (height+2*radius)*(width+2*radius), sizeof(dataType));
416    freeArray("oldPsiI", oldPsiI, (height+2*radius)*(width+2*radius), sizeof(dataType));
417    freeArray("olderPsiR", olderPsiR, (height+2*radius)*(width+2*radius), sizeof(dataType));
418    freeArray("olderPsiI", olderPsiI, (height+2*radius)*(width+2*radius), sizeof(dataType));
419    freeArray("seeds", seeds, (height+2*radius)*(width+2*radius), sizeof(char));
420 #else
421    free(newPsiR);
422    free(newPsiI);
423    free(oldPsiR);
424    free(oldPsiI);
425    free(olderPsiR);
426    free(olderPsiI);
427    free(seeds)
428 #endif
429
430 #ifdef DEBUG
431    if (id == 0 && node == 0)
432      printf("Freed psi arrays.\n");
433 #endif
434
435    free(yMin);
436    free(yMax);
437    free(aString);
438
439 #ifdef DEBUG
440    if (id == 0 && node == 0)
441      printf("Freed remaining arrays and exiting run().\n");
442 #endif
443 }
444
445 #ifdef PARALLEL
446 void freeArray(char *aString, void *array, int length, int size)
447 {
448    char *temp;
449    int a;
450
451    temp = (char *)calloc(strlen(aString)+15, sizeof(char));
452
453    //not sure this is necessary but a harmless safety precaution.
454    MPI_Barrier(comm);
455
456    munmap(array, length * size);
457
458    //everyone must be ready before we remove the shared memory.
459    MPI_Barrier(comm);
460
461    //process zero deallocates the shared memory so that subsequent runs of the program won't
                already have values initialized.
462    if (id == 0)
463    {
464      sprintf(temp, "%s-%d", aString, node);
465      shm_unlink(temp);
```

```
466    }
467
468    free(temp);
469 }
470
471 void *createArray(char *aString, int length, int size)
472 {
473    void *array;
474    char *temp;
475    int fd, a;
476
477    temp = (char *)calloc(strlen(aString)+15, sizeof(char));
478
479    if (id == 0)
480    {
481       //the node identifier is there in case nodeSize is decreased so multiple virtual "nodes" are
                on the same physical compute node.
482       sprintf(temp, "%s-%d", aString, node);
483       //Removes any previous references to this sharedmemory (because this program crashes a lot so
                freeArray doesnt get called).
484       shm_unlink(aString);
485       //Opens file of a column of pointers to long doubles.
486       fd = shm_open(aString, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
487
488       if (fd == -1) //checks for errors. Dunno what exactly as this is Sorenson's code (thanks!).
489       {
490          perror("ERROR: createshm:shm_open:\n");
491          exit(1);
492       }
493
494       //Allocates shm space sufficient to hold all those pointers (or long doubles in this case...)
                .
495       if (ftruncate(fd, length * size) == -1)
496       {
497          perror("ERROR: createshm:ftruncate:\n");
498          exit(1);
499       }
500
501       //maps the array to this shm/file.
502       array = mmap(NULL, length * size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
503       if (array == MAP_FAILED)
504       {
505          perror("ERROR: createshm:mmap:\n");
506          exit(1);
507       }
508    }
509
510    //groups up all the processes. Everyone waits until ID 0 one shows up.
511    MPI_Barrier(comm);
512
513    //All Processes except process 0 run this code (which makes the above waiting kind of funny).
514    if (id > 0)
515    {
516       //the node identifier is there in case nodeSize is decreased so multiple virtual "nodes" are
                on the same physical compute node.
517       sprintf(temp, "%s-%d", aString, node);
518       //this only connects to the existing shared memory as opposed to creating it like what ID 0
                did.
519       fd = shm_open(aString, O_RDWR, S_IRUSR | S_IWUSR);
520
521       if (fd == -1) //random error that I know nothing about...
522       {
523          perror("ERROR: shm open error in getshm\n");
524          exit(1);
525       }
526
527       //Notice there is no call to ftruncate()? That only needs to be done when the shm is created.
                Everyone just needs to map to it.
528
529       //maps to the shm.
530       array = mmap(NULL, length * size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
531
532       if (array == MAP_FAILED)
533       {
534          perror("ERROR: createshm:mmap:\n");
535          exit(1);
536       }
537    }
538
539    free(temp);
540
```

```
541    return array; //returns the awesome shm memory 2D array of awesomeness.
542  }
543  #endif
544
545  //function for each iteration using only 1 previous time step.
546  void functionOld(dataType *oldPsiR, dataType *oldPsiI, dataType *newPsiR,
547    dataType *newPsiI)
548  {
549    int a, b;
550
551    //Calls are being made to neighbor cells so they need to be up-to-date.
552  #ifdef PARALLEL
553    MPI_Barrier(comm);
554  #endif
555
556    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
557      for (b = a + radius; b < a + width + radius; b++)
558      {
559        newPsiR[b] = oldPsiR[b] - (hbar*timeStep/(2*mass))
560          *(oldPsiI[b+1] + oldPsiI[b-1] + oldPsiI[b+adj] + oldPsiI[b-adj]
561          - oldPsiI[b] - oldPsiI[b] - oldPsiI[b] - oldPsiI[b]);
562        newPsiI[b] = oldPsiI[b] + (hbar*timeStep/(2*mass))
563          *(oldPsiR[b+1] + oldPsiR[b-1] + oldPsiR[b+adj] + oldPsiR[b-adj]
564          - oldPsiR[b] - oldPsiR[b] - oldPsiR[b] - oldPsiR[b]);
565      }
566  }
567
568  //function for each iteration of the solution using 2 previous time steps.
569  void functionNew(dataType *oldPsiR, dataType *oldPsiI, dataType *psiR,
570    dataType *psiI, dataType *newPsiR, dataType *newPsiI)
571  {
572    int a, b;
573
574    //Calls are being made to neighbor cells so they need to be up-to-date.
575  #ifdef PARALLEL
576    MPI_Barrier(comm);
577  #endif
578
579    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
580      for (b = a + radius; b < a + width + radius; b++)
581      {
582        newPsiR[b] = oldPsiR[b] - (hbar*timeStep/mass)
583          *(psiI[b+1] + psiI[b-1] + psiI[b+adj] + psiI[b-adj]
584          - psiI[b] - psiI[b] - psiI[b] - psiI[b]);
585        newPsiI[b] = oldPsiI[b] + (hbar*timeStep/mass)
586          *(psiR[b+1] + psiR[b-1] + psiR[b+adj] + psiR[b-adj]
587          - psiR[b] - psiR[b] - psiR[b] - psiR[b]);
588      }
589  }
590
591  int detect(dataType *psiR, dataType *psiI, char *seeds)
592  {
593    int a, b, c, exit = FALSE;
594    long double sum = 0, percent = ((long double)rand())/RAND_MAX;
595    long double sums[np];
596
597    //Calculate the sum of the probabilities in each region.
598    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
599      for (b = a+radius; b < a+width + radius; b++)
600        if (seeds[b] == FALSE && (seeds[b-adj] || seeds[b-1] || seeds[b+adj] || seeds[b+1]))
601          sum += psiR[b]*psiR[b] + psiI[b]*psiI[b];
602
603  #ifdef PARALLEL
604    MPI_Gather(&sum, 1, MPI_LONG_DOUBLE, sums, 1, MPI_LONG_DOUBLE, 0, comm);
605  #else
606    sums[0] = sum;
607  #endif
608
609    sum = 0;
610
611    //Let process zero do the actual detection in the region with the detection.
612    if (id == 0)
613    {
614      //Determine which region has the detection.
615      for (c = 0; c < np; c++)
616      {
617        sum += sums[c];
618        if (sum >= percent)
619          break;
620      }
621
```

```
622      if (c == np) //There is no detection.
623        goto nestedLoopBreak;
624
625      sum -= sums[c]; //Went too far, so let's go back one step.
626
627
628      for (a = yMin[c]*adj + radius*adj; a <= yMax[c]*adj + radius*adj; a += adj)
629        for (b = a+radius; b < a+width + radius; b++)
630          if (seeds[b] == FALSE && (seeds[b-adj] || seeds[b-1] || seeds[b+adj] || seeds[b+1]))
631          {
632            sum += psiR[b]*psiR[b] + psiI[b]*psiI[b];
633
634            if (exit == FALSE && sum >= percent)
635            {
636              seeds[b] = TRUE;
637              exit = TRUE;
638              y = a / adj - radius;
639              x = b - a - radius;
640
641              //Top-Bottom
642              if (a == radius*adj)
643                seeds[(height+radius)*adj+b-a] = TRUE;
644              else if (a == (height - radius)*adj)
645                seeds[b-a] = TRUE;
646
647              //Left-Right
648              if (b-a == radius)
649                seeds[a+(width+radius)] = TRUE;
650              else if (b-a == radius + width - 1)
651                seeds[a] = TRUE;
652
653              goto nestedLoopBreak;
654            }
655            else
656              psiR[b] = psiI[b] = 0.0;
657          }
658  }
659
660    nestedLoopBreak:
661
662 #ifdef PARALLEL
663    MPI_Barrier(comm); //Wait for the detection before erasing.
664 #endif
665
666    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
667      for (b = a+radius; b < a+width + radius; b++)
668        if (seeds[b] || seeds[b-adj] || seeds[b-1] || seeds[b+adj] || seeds[b+1])
669          psiR[b] = psiI[b] = 0.0;
670
671
672 #ifdef PARALLEL
673    MPI_Bcast(&exit, 1, MPI_INT, 0, comm);
674 #endif
675
676    normalize(psiR, psiI);
677
678    return exit;
679 }
680
681 void borders(dataType *psi)
682 {
683    int a, b;
684
685    //Assumes periodic boundaries.
686    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
687      for (b = a; b < a+radius; b++)
688      {
689        psi[b] = psi[b+width];
690        psi[b+width+radius] = psi[b+radius];
691      }
692
693 #ifdef PARALLEL
694    MPI_Barrier(comm); //The borders are being arranged so all the cells need to be ready
695 #endif
696
697    if (id == 0)
698    {
699      for (a = 0; a < radius*adj; a += adj)
700        for (b = a; b < a+width + radius + radius; b++)
701          psi[b] = psi[b+height*adj];
702    }
```

```
703
704    if (id == np-1)
705    {
706      for (a = 0; a < radius*adj; a += adj)
707        for (b = a; b < a+width + radius + radius; b++)
708          psi[b+(height+radius)*adj] = psi[b+radius*adj];
709    }
710  }
711
712  //writes the psi matrix to a file in comma delimited form
713  void writePsi(char *fileName, dataType *psiR, dataType *psiI)
714  {
715    FILE *file;
716    char *text;
717    int a, b;
718
719    text = (char *)calloc(strlen(directory)+strlen(fileName)+strlen(extension)+15, sizeof(char));
720    sprintf(text, "%s/Particle %s.%s", directory, fileName, extension);
721    file = fopen(text, "w");
722
723    for (a  = radius; a < radius + height; a++)
724    {
725      for (b = radius; b < radius + width; b++)
726        fprintf(file, "%.*LG,", psiR[(a)*adj+(b)]*psiR[(a)*adj+(b)] + psiI[(a)*adj+(b)]*psiI[(a)*
                adj+(b)], DIGITS);
727      fprintf(file, "\n");
728    }
729
730    fclose(file);
731    free(text);
732  }
733
734  void writeSeed(int particles, int detections, double particleTime)
735  {
736    FILE *file;
737    char *text;
738    double realTime;
739
740    //Only one process should write to a file at a time.
741    if (id != 0)
742      return;
743
744
745  #ifdef PARALLEL
746    realTime = MPI_Wtime() - timerStart;
747  #else
748    time_t seconds;
749    time(&seconds);
750    realTime = seconds - timerStart;
751  #endif
752
753    text = (char *)calloc(strlen(directory)+strlen(extension)+10, sizeof(char));
754    sprintf(text, "%s/seed.%s", directory, extension);
755
756    file = fopen(text, "a");
757
758    fprintf(file, "%d,%d,%d,%d,%f,%f\n", detections, particles, y, x, particleTime, realTime);
759
760    fclose(file);
761    free(text);
762  }
763
764  //Fills the matrix with the initial condition of the system
765  void init(dataType *psiR, dataType *psiI)
766  {
767    long double coef;
768    int a, b;
769
770  #ifdef PARALLEL
771    MPI_Barrier(comm);
772  #endif
773
774    for (a = yMin[id]; a <= yMax[id]; a++)
775      for (b = 0; b < width; b++)
776      {
777        //Normal starting condition of gaussian wave packet.
778        coef = expl((-(b-xCenter)*(b-xCenter)-(a-yCenter)*(a-yCenter))/(particleSize*particleSize))
                /sqrtl(particleSize*particleSize*PI/2);
779        psiR[(a+radius)*adj+(b+radius)] = coef;//*cosl(Vx*b+Vy*a);
780        psiI[(a+radius)*adj+(b+radius)] = 0.0;//coef*sinl(Vx*b+Vy*a);
781
```

```
782            //Uncomment these lines for infinite width starting condition.
783            //psiR[(a+radius)*adj+(b+radius)] = cosl((b + a)*4*PI/(width+height));
784            //psiI[(a+radius)*adj+(b+radius)] = sinl((b + a)*4*PI/(width+height));
785        }
786
787    normalize(psiR, psiI);
788    borders(psiR);
789    borders(psiI);
790 }
791
792 void normalize(dataType *psiR, dataType *psiI)
793 {
794    long double temp, sum = 0;
795    int a, b;
796
797    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
798      for (b = a+radius; b < a+width+radius; b++)
799        sum += psiR[b]*psiR[b] + psiI[b]*psiI[b];
800
801 #ifdef PARALLEL
802    temp = sum;
803
804    MPI_Allreduce(&temp, &sum, 1, MPI_LONG_DOUBLE, MPI_SUM, comm);
805 #endif
806
807    sum = sqrtl((long double)sum);
808
809    for (a = yMin[id]*adj + radius*adj;a <= yMax[id]*adj + radius*adj; a += adj)
810      for (b = a+radius; b < a + width + radius; b++)
811      {
812        psiR[b] /= sum;
813        psiI[b] /= sum;
814      }
815 }
816
817 void seedBorders(char *seeds)
818 {
819    int a, b;
820
821    //Assumes periodic boundaries.
822    for (a = yMin[id] + radius; a <= yMax[id] + radius; a++)
823      for (b = 0; b < radius; b++)
824      {
825        seeds[(a)*adj+(b)] = seeds[(a)*adj+(b+width)];
826        seeds[(a)*adj+(b+width+radius)] = seeds[(a)*adj+(b+radius)];
827      }
828
829 #ifdef PARALLEL
830    MPI_Barrier(comm); //The borders are being arranged so all the cells need to be ready
831 #endif
832
833    if (id == 0)
834    {
835      for (a = 0; a < radius; a++)
836        for (b = 0; b < width + radius + radius; b++)
837          seeds[(a)*adj+(b)] = seeds[(a+height)*adj+(b)];
838    }
839
840    if (id == np-1)
841    {
842      for (a = 0; a < radius; a++)
843        for (b = 0; b < width + radius + radius; b++)
844          seeds[(height+radius+a)*adj+(b)] = seeds[(a+radius)*adj+(b)];
845    }
846 }
847
848 //Fills the matrix with the initial condition of the system
849 int seedInit(char *seeds)
850 {
851    int a, b, d = 0;
852
853    if (id == 0)
854      for (a = 0; a <= height; a++)
855        for (b = 0; b < width; b++)
856          if (seedStart(a, b) == TRUE)
857          {
858            seeds[(a+radius)*adj+(b+radius)] = 1;
859            y = a;
860            x = b;
861            d++
862            writeSeed(d, d, 0.0);
```

```
863          }
864
865 #ifdef PARALLEL
866     //Share detection count with every process and act as a barrier.
867     MPI_Bcast(&d, 1, MPI_INT, 0, comm);
868 #endif
869
870   seedBorders(seeds);
871
872   return d;
873 }
874
875 int seedStart(int y, int x)
876 {
877   if (seedType == NO_SEED)
878     return FALSE;
879   else if (seedType == RANDOM_SEED)
880     return rand() % 2;
881   else if (seedType == DOT_SEED)
882     if ((seedCenterX - x)*(seedCenterX - x) <= seedRadius * seedRadius && (seedCenterY - y)*(
              seedCenterY - y) <= seedRadius*seedRadius)
883       return TRUE;
884     else
885       return FALSE;
886   else if (seedType == WALL_SEED)
887     if ((seedCenterX - x)*(seedCenterX - x) <= seedRadius * seedRadius)
888       return TRUE;
889     else
890       return FALSE;
891   else
892     return FALSE;
893 }
894
895 void centerParticle(long double percent)
896 {
897   long double theta;
898
899   if (id == 0)
900     theta = ((long double)rand()) * 2 * PI / RAND_MAX;
901
902 #ifdef PARALLEL
903   MPI_Bcast(&theta, 1, MPI_LONG_DOUBLE, 0, comm);
904 #endif
905
906   xCenter = cosl(theta) * (particleMinRadius + (particleMaxRadius - particleMinRadius)*percent) +
              seedCenterX;
907   yCenter = sinl(theta) * (particleMinRadius + (particleMaxRadius - particleMinRadius)*percent) +
              seedCenterY;
908 }
```