8-15-2011

Quantum Diffusion-Limited Aggregation

David Bradley Johnson
Butler University

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.

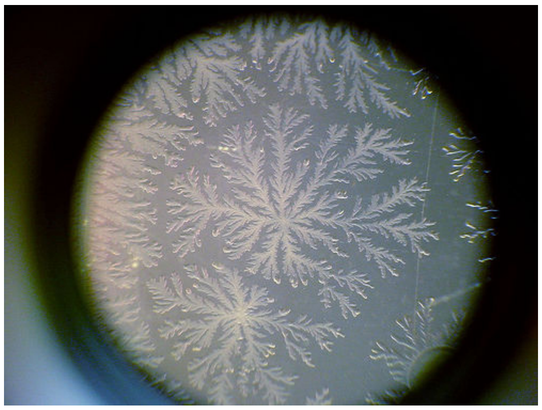

Figure 1: Bacterial Colony

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.

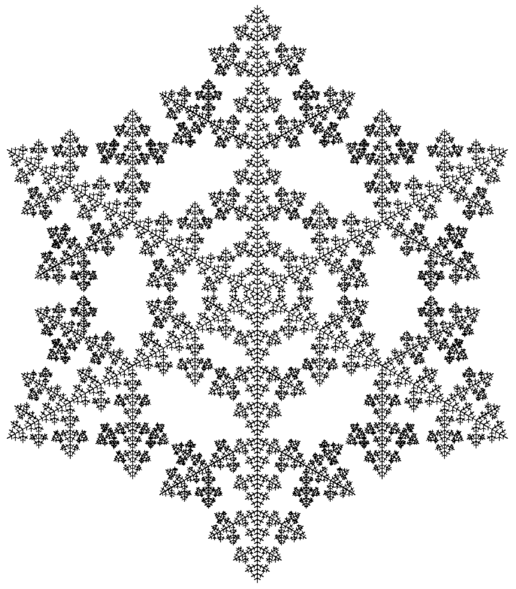

Figure 2: Snowflake Fractal

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.

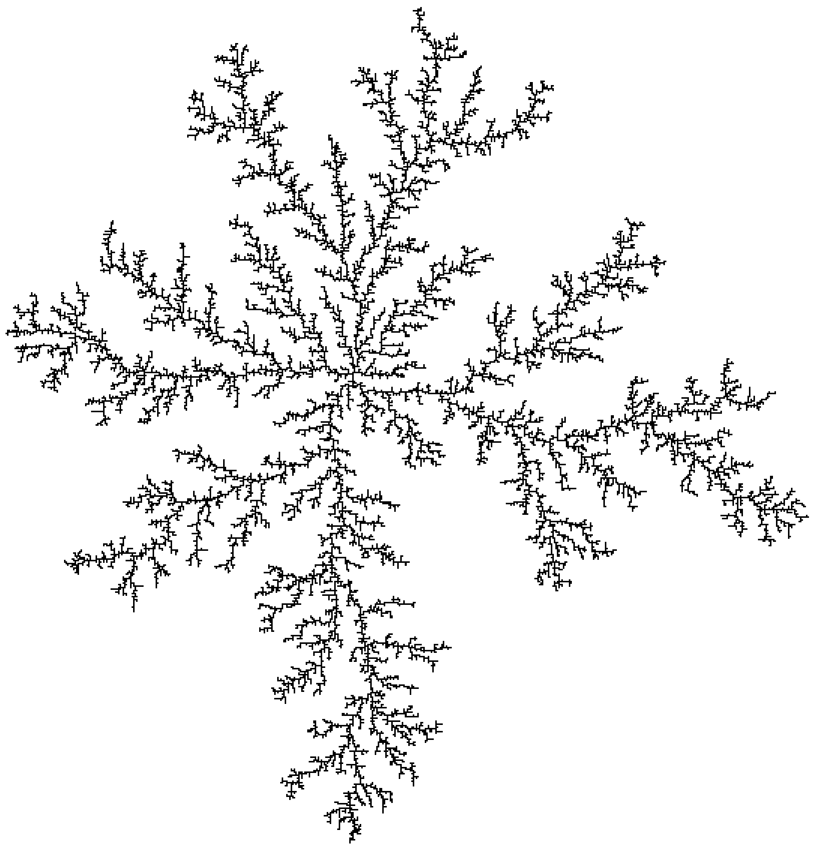

Figure 3: DLA made via Random Walk

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

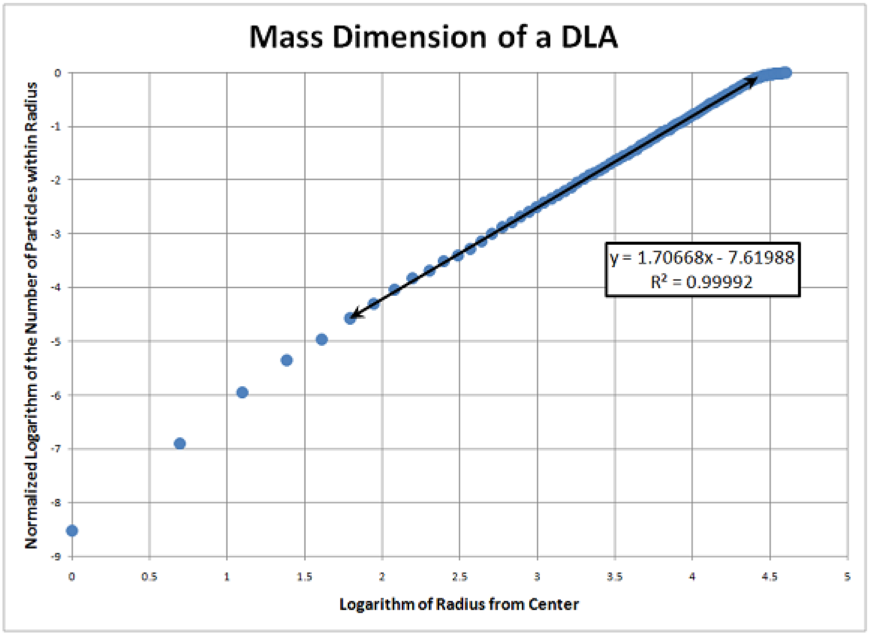

Figure 4: Log-Log Plot of the Mass within a Centered Circle vs. Radius

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 classical random walk in two dimensions can be written as Eqn. (2) where $\phi^t_{x,y}$ is the probability to find the particle at position (x, y) and time t. Likewise, $\psi^t_{x,y}$ is used to represent the complex-valued probability amplitude where $\psi^{t\,*}_{x,y}\psi^t_{x,y}$ 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).

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

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].

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

$$\frac{\Phi^{t+\Delta t}_{x,y} - \Phi^t_{x,y}}{\Delta t} = D(\frac{\Phi^t_{x+\Delta x,y} + \Phi^t_{x-\Delta x,y} - 2\Phi^t_{x,y}}{(\Delta x)^2} + \frac{\Phi^t_{x,y+\Delta y} + \Phi^t_{x,y-\Delta y} - 2\Phi^t_{x,y}}{(\Delta y)^2}) \tag{4}$$

$$\begin{gathered}\Phi^{t+1}_{x,y} = \Phi^t_{x,y} + \frac{1}{4}(\Phi^t_{x+1,y} + \Phi^t_{x-1,y} + \Phi^t_{x,y+1} + \Phi^t_{x,y-1} - 4\Phi^t_{x,y}) \\ \Delta x = 1, \Delta y = 1, \Delta t = 1, D = 1/4\end{gathered} \tag{5}$$

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.

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).

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

$$i\hbar\frac{\Psi^{t+\Delta t}_{x,y} - \Psi^{t-\Delta t}_{x,y}}{2\Delta t} = -\frac{\hbar^2}{2m}(\frac{\Psi^t_{x+\Delta x,y} + \Psi^t_{x-\Delta x,y} - 2\Psi^t_{x,y}}{(\Delta x)^2} + \frac{\Psi^t_{x,y+\Delta y} + \Psi^t_{x,y-\Delta y} - 2\Psi^t_{x,y}}{(\Delta y)^2}) \tag{7}$$

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.

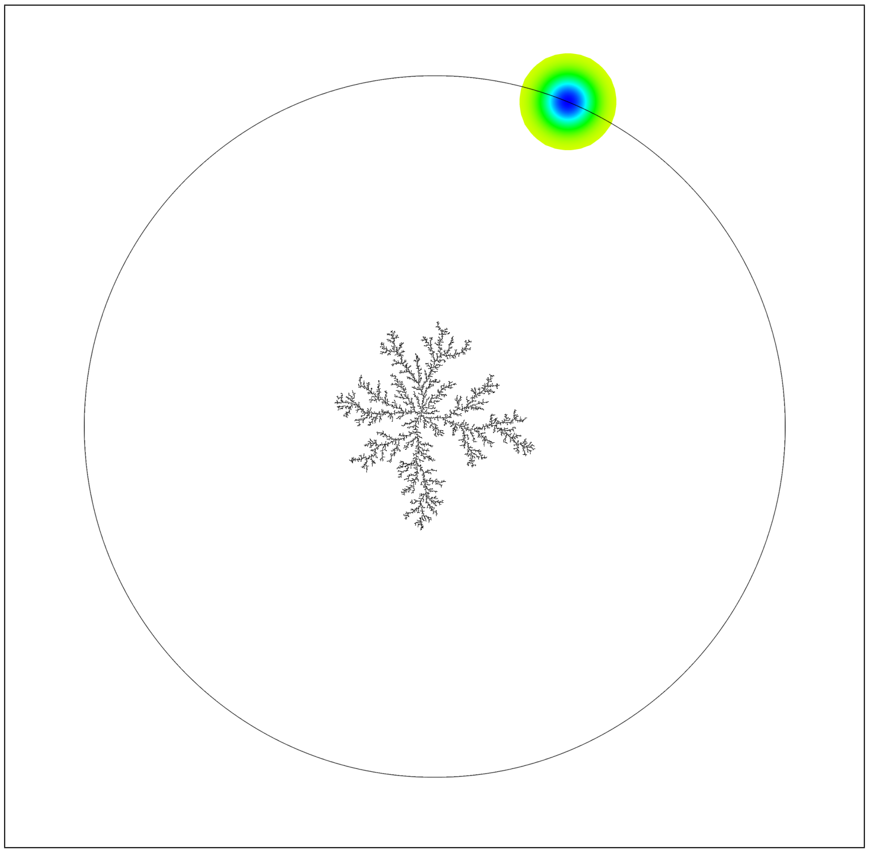

Figure 5: The Grid

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 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.

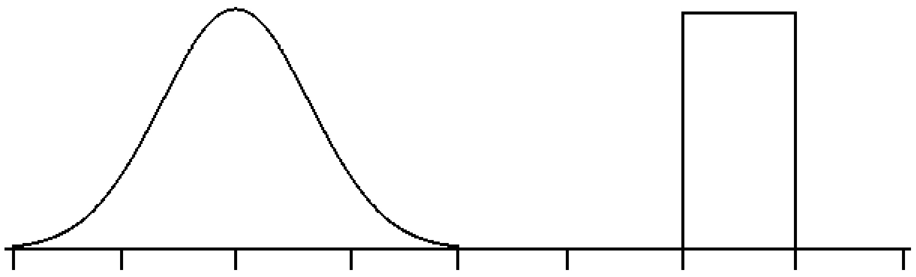

Figure 6: $T = 0$: Initial Wave Packet Far from Seed

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.

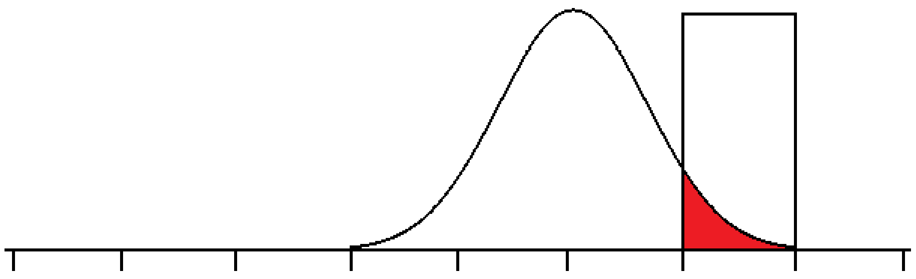

Figure 7: $T = k\Delta t$: Updated Wave Overlapping with Seed

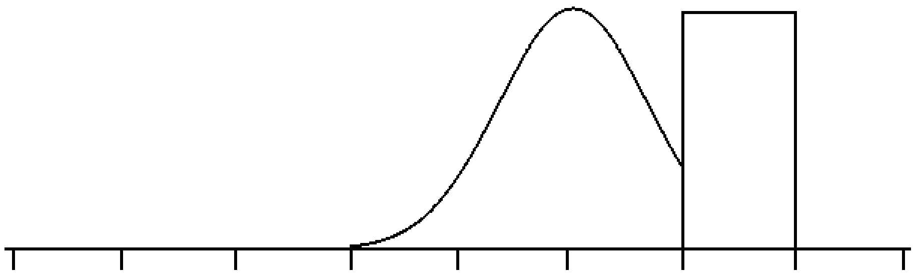

Figure 8: $T = k\Delta t$: Updated Wave Being Zeroed within Seed

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

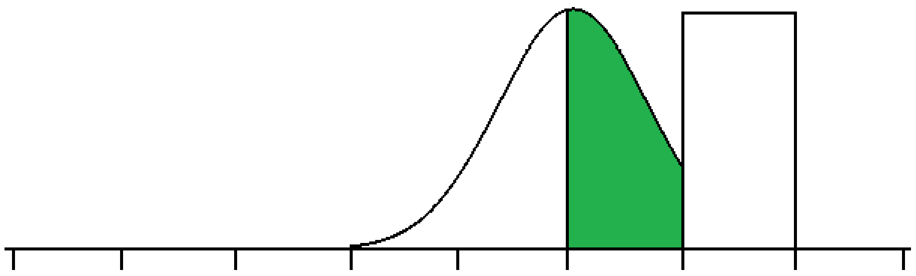

Figure 9: $T = n\Delta t = 1$: Wave Being Tested for Detection

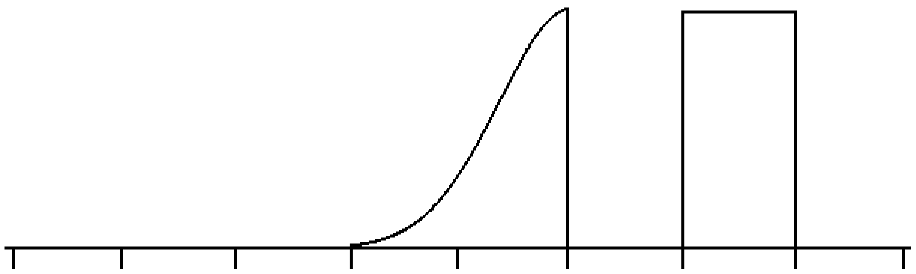

Figure 10: $T = n\Delta t = 1$: Wave Zeroed after Failing Detection

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.

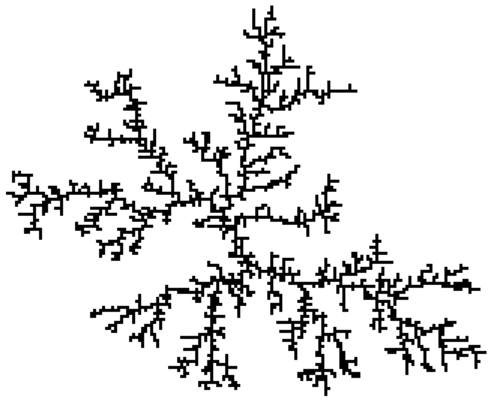

Figure 11: Fractal Generated by DLA using Diffusion Equation

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

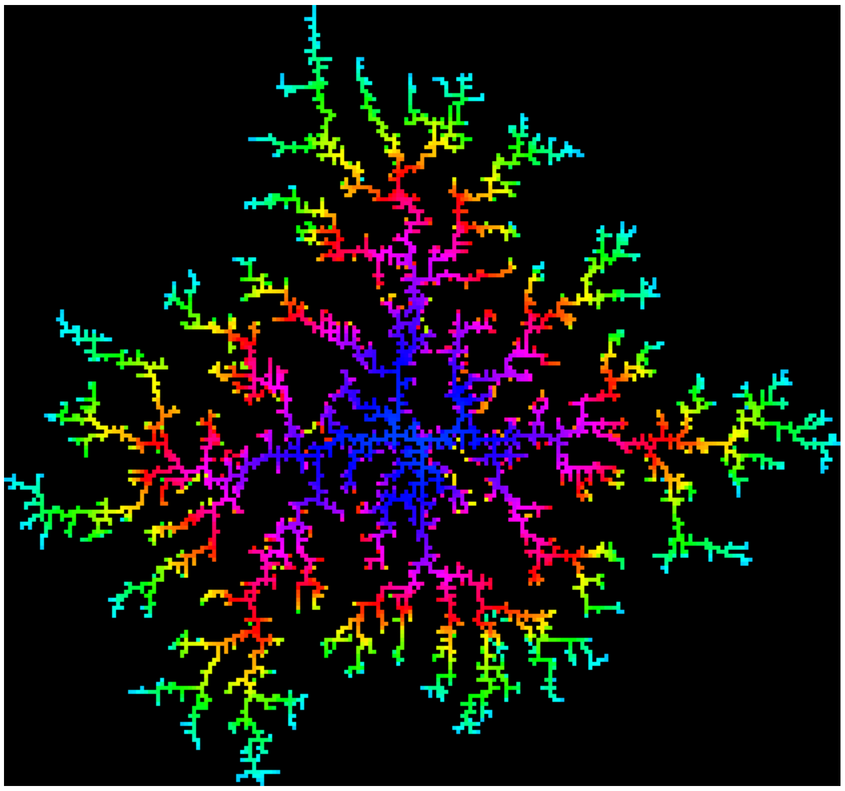

Figure 12: Fractal Generated by QDLA

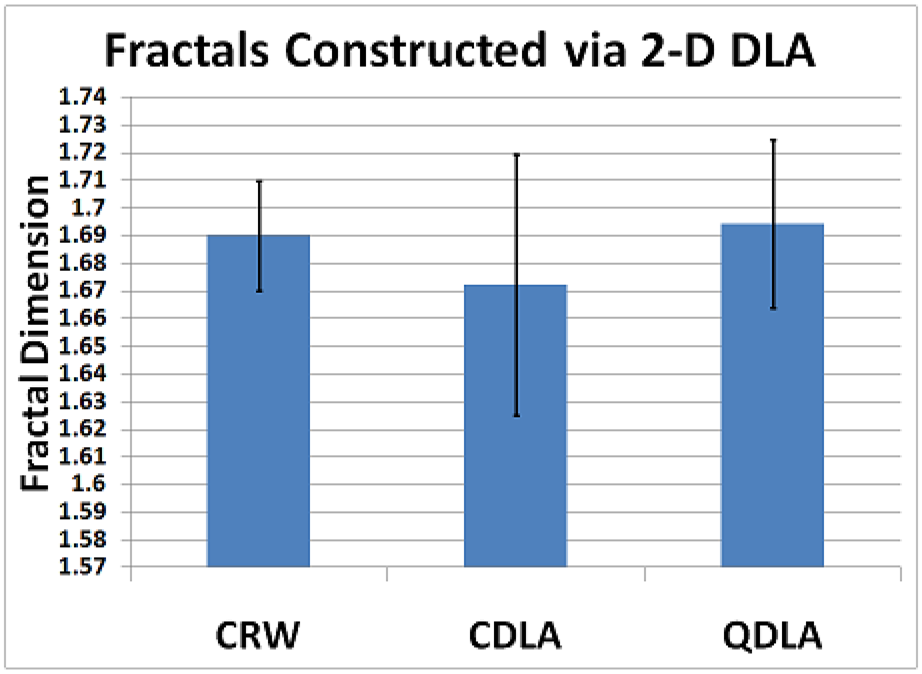

Figure 13: Average Fractal Dimensions for Several Types of 2D DLA

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.

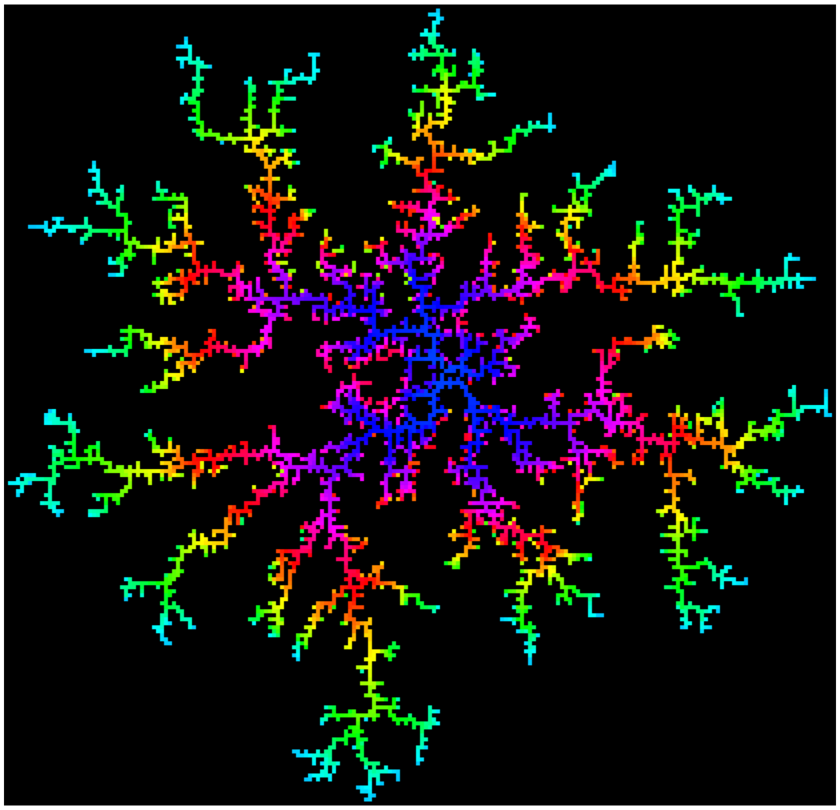

Figure 14: Fractal Generated by QDLA with Initial Wave Packet $\sigma = 16$ and $d = 1.45$

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.

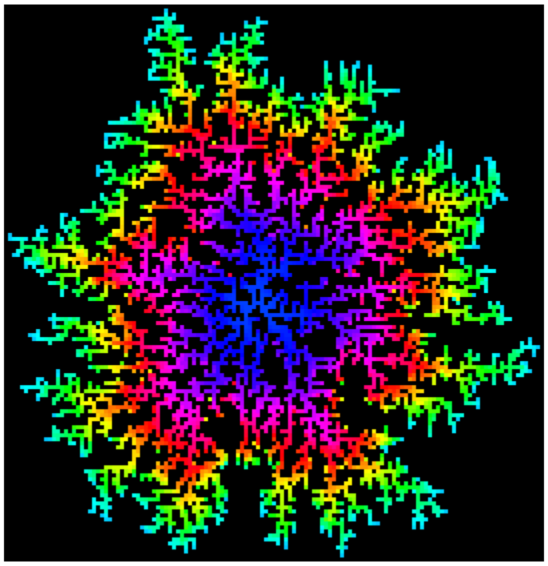

Figure 15: Fractal Generated by QDLA with Initial Wave Packet $\sigma = 1$ and $d = 1.91$

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.

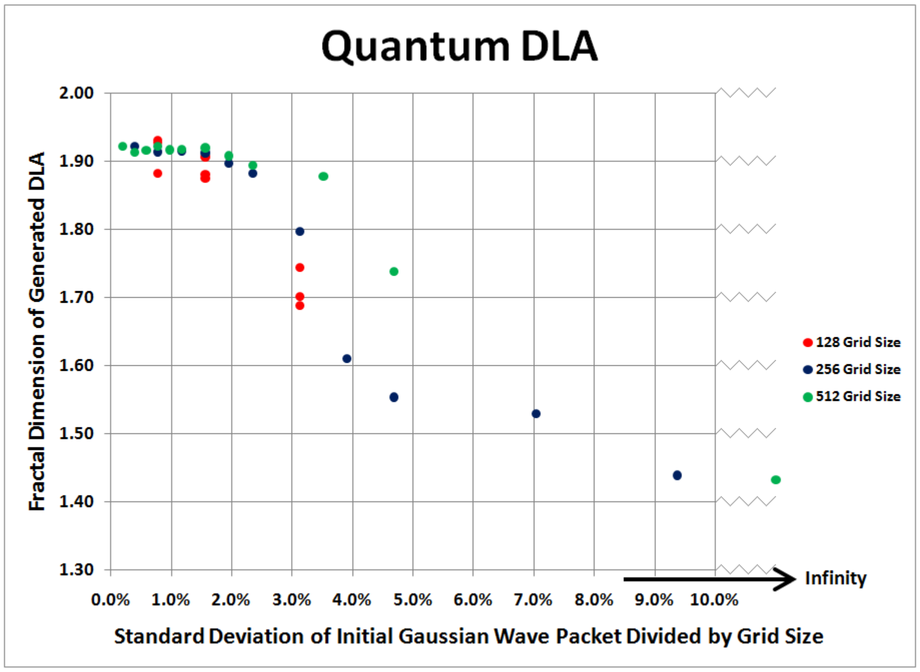

Figure 16: Fractal Dimension of Various Runs of 2D QDLA

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.

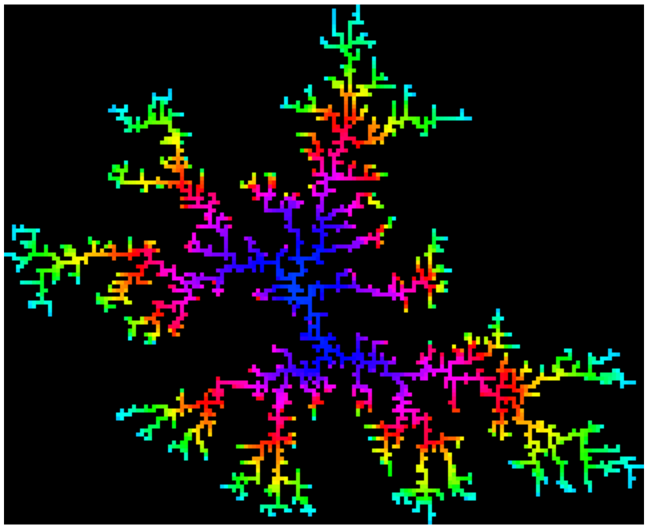

Figure 17: Fractal Generated by CDLA Showing the Relative Age of Different Regions

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, where I learned much about the peculiarities of quantum systems. I am also indebted to the Butler University Liberal Arts and Sciences College Deans Office, the Holcomb Undergraduate Grants Committee, and Dr. Han from Butler's Physics department for funding the trip to this conference.

This thesis would not be possible without the supercomputer, Big Dawg, provided through funding from Dr. Levinson and the support that came with it from Nate Partenheimer. Furthermore, special thanks goes to Drs. Sorenson and Hardikar for their help in learning MPI and their support with using the supercomputer, including the willingness to give me time on the supercomputer.

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
/**
 *  QDLA.c
 *
 *  Quantum Diffusion Limited Aggregation
 * David Johnson
 *  Butler University
 * 15 August 2011
**/

//To turn off all MPI commands at once.
#define PARALLEL
//Turn on in order to debug a program.
#define NODEBUG

//To experiment with the accuracy and speed of other data types.
#define dataType long double

#include <stdlib.h> //exit()
#include <stdio.h> //printf()
#include <string.h> //strcat()
#include <math.h> //cos(), sin(), exp()
#include <time.h> //seeding random number (it gives very weird error messages without this)
#include <fcntl.h> //shm_open, O_CREAT, O_RDWR
#include <sys/stat.h> //S_IRUSR and S_IWUSR
#include <sys/mman.h> //mmap, munmap, PROT_READ, PROT_WRITE, MAP_SHARED, and MAP_FAILED

#ifdef PARALLEL
#include "mpi.h" //allows parallel processing
#endif

void run(void);
void freeArray(char *, void *, int, int);
void *createArray(char *, int, int);
void functionNew(dataType *, dataType *, dataType *, dataType *, dataType *, dataType *);
void functionOld(dataType *, dataType *, dataType *, dataType *);
int detect(dataType *, dataType *, char *);
void borders(dataType *);
void normalize(dataType *, dataType *);
void writePsi(char *, dataType *, dataType *);
void writeSeed(int, int, double);
void init(dataType *, dataType *);
void seedBorders(char *);
int seedInit(char *);
int seedStart(int, int);
void centerParticle(long double);

#define PI 3.14159265358979323846264338327950
#define TRUE 1
#define FALSE 0
#define DIGITS 16
#define NODE_SIZE 16

#define NO_SEED 0
#define RANDOM_SEED 1
#define DOT_SEED 2
#define WALL_SEED 3

#define height 512
#define width 512
#define radius 1
#define adj (width + 2*radius)

#define timeStep ((dataType)0.05)
#define totalTime ((dataType)500000.0)
#define totalSteps ((long)10000000)
#define detectFrequency 20
#define writeFrequency 0

#define diffusionRate ((dataType)0.25)
#define hbar ((dataType)1.0)
#define mass ((dataType)1.0)

#define particleSize ((dataType)0.0)

char *directory = "run11"; //sub-directory where files will be saved to.
char *extension = "csv"; //extension of each file for file association ease.

const int numPackets = 1; //number of gaussian wave packets that a particle is initially split up
    into.
```

```c

dataType  yCenter = height/2-0.5; //y position of center of initial condition.
dataType  xCenter = width/2-0.5; //x position of center of initial condition.

const int     seedType = DOT_SEED; //determines which type of initial seeding configuration is
     used.
const int     particleCirclesSeed = TRUE; //boolean that determines if the initial particle
     position is ignored or is set to circle the seed.
const dataType  particleMinRadius = height/2-30.0; //the starting distance between the initial
     positions of particles and the seed start.
const dataType particleMaxRadius = height/2-30.0; //the ending distance between the initial
     positions of particles and the seed start.
const int     seedCenterY = height/2; //y coordinate of the center of the seed.
const int     seedCenterX = width/2; //x coordinate of the center of the seed.
const int     seedRadius = 0; //radius of the seed.
const int     particles = 25000; //the number of particles to be sent out.

double timerStart;
int id, np, node, totalNodes, *yMin, *yMax;
int x, y; // Only used on process zero for detection.

#ifdef PARALLEL
MPI_Comm comm;
#endif

int main(int argc, char *argv[])
{
  int a, b, dif, timeSeed;
  char *temp, *temp1;
  time_t rawTime, seconds1, seconds2;
  struct tm *timeInfo;
  struct stat st;
  FILE *file;

  time(&seconds1);

  //if program is not parallel, then there is only 1 process and it has id 0 and is on node 0
  id = 0;
  np = 1;
  node = 0;

  //seeds the random numbers
  time(&rawTime); //the c version of getting a time object (with updated info)
  timeInfo = localtime(&rawTime); //the c version of getting the info about the time object
  srand((*timeInfo).tm_sec); //seeds the random number generator with an int between 0 and
       RAND_MAX
  timeSeed = rand();

#ifdef PARALLEL
  long double timer = MPI_Wtime(); //Gets start time of program (according to MPI so might have
       more sig figs)
  MPI_Init(&argc, &argv); //imitialize MPI
  MPI_Comm_rank(MPI_COMM_WORLD, &id); //Gets process id# from the world comm
  MPI_Comm_size(MPI_COMM_WORLD, &np); //Gets total number of processes that are executing this
       program


  //We do this before splitting up the communicator so that the same id's on different nodes will
        have different seeds for rand()
  MPI_Bcast(&timeSeed, 1, MPI_INT, 0, MPI_COMM_WORLD);
  srand(timeSeed);

  for (a = 0; a <= id; a++) //ensures that different processes will be using uncorrelated
       pseudorandom numbers
    srand(rand());

  //Splits all the processes into their own node so each node can run independently but still be
       executed at the same time
  node = id/NODE_SIZE;
  totalNodes = (np - 1) / NODE_SIZE + 1;

  MPI_Comm_split(MPI_COMM_WORLD, node, id, &comm); //an awesome command that does all the comm
       construction for me
  MPI_Comm_rank(comm, &id); //Gets process id# from mpi
  MPI_Comm_size(comm, &np); //Gets total number of processes that are executing this program

  /**
   *  IMPORTANT NOTE: Do not think any process has its original ID anymore. They
   *  have all just been changed according to the node that they are running on.
   *  They also got a new MPI_Comm ojbect that distinguishes processes on
   *  different nodes. So, do NOT use MPI_COMM_WORLD anymore! Use the globally
   *  defined MPI_Comm object called "comm" instead. Or else, a call to
   *  MPI_Barrier(MPI_COMM_WORLD) would cause all processes to wait even though
   *  the ones on different nodes are doing something unrelated. This mistake
   *  could cause the program to crash unexpectedly with no explanatory message
   *  about it so it is important to keep an eye out for it.
  **/
#endif

#ifdef PARALLEL
  if (totalNodes > 1)
  {
    temp = (char *)calloc((strlen(directory)+5), sizeof(char));
    sprintf(temp, "%s%d", directory, node);
    directory = temp;
  }
#endif

#ifdef DEBUG
  if (id == 0)
    printf("Node%d: %s\n", node, directory);
#endif

  //Create Directory or else an exception might be thrown.
  if (id == 0 && stat(directory, &st) != 0)
  {
    printf("Making directory: %s\n", directory);
    mkdir(directory, S_IRWXU);
  }

  if (id == 0)
  {
    //Erase seed file.
    temp = (char *)calloc((strlen(directory)+strlen(extension)+10), sizeof(char));
    sprintf(temp, "%s/seed.%s", directory, extension);
    file = fopen(temp, "w");
    fprintf(file, "detections,particles,y,x,particle time,real time\n");
    fclose(file);
    free(temp);
  }

  run();

#ifdef DEBUG
  if (id == 0)
  {
#ifdef PARALLEL
    timer = MPI_Wtime() - timer;
    dif = (int)timer;

    printf("Node%d: Total time on %d cores: %d hours, %d minutes, %LG seconds.\n",
         node, np, ((int)(dif/60))/60, ((int)(dif/60))%60, timer-(dif-dif%60));
#else
    time(&seconds2);
    dif = seconds2 - seconds1;

    printf("Node%d: Total time on %d cores: %d hours, %d minutes, %d seconds.\n",
         node, np, ((int)(dif/60))/60, ((int)(dif/60))%60, dif%60);
#endif
  }
#endif

#ifdef PARALLEL
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize(); //shuts down all mpi commands for each process
#endif

  return 0;
}

void run(void)
{
  dataType *newPsiR, *newPsiI, *oldPsiR, *oldPsiI, *olderPsiR, *olderPsiI, *temp;
  char *seeds;
  char *aString;
  int a, b, d, min, max, loop = 1, detections;
  long c;
  time_t seconds; //Only used by non-mpi code.

  aString = (char *)calloc(50, sizeof(char));

  yMin = (int *)calloc(np, sizeof(int));
  yMax = (int *)calloc(np, sizeof(int));

  min = 0;
  for (a = 0;a < np;a++)
  {
    max = min+((height-1)/np)-1;
    if (a <= ((height-1)%(np)))
      max++;

    yMin[a] = min;
    yMax[a] = max;

    min = max+1;
  }

#ifdef DEBUG
  if (id == 0 && node == 0)
    printf("Initialized Seeds and Constants.\n");
#endif

#ifdef PARALLEL
  //calls createArray function which creates shared memory array and connects each process to it.
  newPsiR = (dataType *)createArray("newPsiR", (height+2*radius)*(width+2*radius), sizeof(
       dataType));
  newPsiI = (dataType *)createArray("newPsiI", (height+2*radius)*(width+2*radius), sizeof(
       dataType));
  oldPsiR = (dataType *)createArray("oldPsiR", (height+2*radius)*(width+2*radius), sizeof(
       dataType));
  oldPsiI = (dataType *)createArray("oldPsiI", (height+2*radius)*(width+2*radius), sizeof(
       dataType));
  olderPsiR = (dataType *)createArray("olderPsiR", (height+2*radius)*(width+2*radius), sizeof(
       dataType));
  olderPsiI = (dataType *)createArray("olderPsiI", (height+2*radius)*(width+2*radius), sizeof(
       dataType));
  seeds = (char *)createArray("seeds", (height+2*radius)*(width+2*radius), sizeof(char));
#else
  newPsiR = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
  newPsiI = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
  oldPsiR = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
  oldPsiI = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
  olderPsiR = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
  olderPsiI = (dataType *)calloc((height+2*radius)*(width+2*radius), sizeof(dataType));
  seeds = (char *)calloc((height+2*radius)*(width+2*radius), sizeof(char));
#endif

#ifdef PARALLEL
  //Start timer.
  timerStart = MPI_Wtime();
#else
  time(&seconds);
  timerStart = seconds;
#endif

  detections = seedInit(seeds);

#ifdef DEBUG
  if (id == 0 && node == 0)
    printf("Created and linked the all arrays.\n");
#endif

  for (d = detections; d < particles; d++)
  {
    if (particleCirclesSeed)
    {
      if (particles != 0)
        centerParticle(((long double)d)/particles);
      else
        centerParticle(0);
    }

    //Set initial conditions.
    init(olderPsiR, olderPsiI);

    for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
      for (b = a+radius; b < a+width + radius; b++)
        if (seeds[b] == TRUE)
        {
          olderPsiR[b] = 0.0;
          olderPsiI[b] = 0.0;
        }
    normalize(olderPsiR,olderPsiI);
```

```

#ifdef PARALLEL
    MPI_Barrier(comm);
#endif

    if (id == 0 && writeFrequency != 0) //Write initial condition to file.
    {
      sprintf(aString, "%d 0", d);
      writePsi(aString, olderPsiR, olderPsiI);
    }

    //Finds the second initial condition based off of the first.
    functionOld(olderPsiR, olderPsiI, oldPsiR, oldPsiI);

#ifdef PARALLEL
    MPI_Barrier(comm);
#endif

    //Write 1st Time Step to file.
    if (id == 0 && writeFrequency != 0)
    {
      sprintf(aString, "%d %.*LG", d, timeStep, DIGITS);
      writePsi(aString, oldPsiR, oldPsiI);
    }

#ifdef DEBUG
    if (id == 0 && node == 0)
      printf("Starting Particle Run.\n");
#endif

    c = 0;
    loop = TRUE;

    do //Iterates through all of the timeSteps until the totalTime is reached.
    {
      functionNew(olderPsiR, olderPsiI, oldPsiR, oldPsiI, newPsiR, newPsiI);

      for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
        for (b = a+radius; b < a+width + radius; b++)
          if (seeds[b] == TRUE)
          {
            newPsiR[b] = 0.0;
            newPsiI[b] = 0.0;
          }

      normalize(newPsiR, newPsiI);

#ifdef DEBUG
      if (id == 0 && node == 0 && c % 100 == 0)
        printf("Time: %.*LG\n", c * timeStep, DIGITS);
#endif

#if writeFrequency>0

#ifdef PARALLEL
      MPI_Barrier(comm);
#endif
      if (id == 0 && c % writeFrequency == 0)
      {
        sprintf(aString, "%d %.*LG", d, c * timeStep, DIGITS);
        writePsi(aString, newPsiR, newPsiI); //write newPsi to a file
#ifdef DEBUG
        printf("Wrote File: \"Particle %s.%s\"\n", aString, extension);
#endif
      }
#endif

#if detectFrequency>0

#ifdef PARALLEL
      MPI_Barrier(comm);
#endif
      if (c % detectFrequency == 0)
        if (detect(newPsiR, newPsiI, seeds))
        {
          loop = FALSE;
          detections++;
          writeSeed(d, detections, (double)(c * timeStep));
        }
#endif

```

```
      if (totalSteps != 0 && c > totalSteps)
        loop = FALSE;

      borders(newPsiR);
      borders(newPsiI);

      //each psi gets moved backwards as time just progressed forward.
      temp = olderPsiR;
      olderPsiR = oldPsiR;
      oldPsiR = newPsiR;
      newPsiR = temp;

      temp = olderPsiI;
      olderPsiI = oldPsiI;
      oldPsiI = newPsiI;
      newPsiI = temp;

      c++;
    } while (loop);
  }

#ifdef DEBUG
  if (id == 0 && node == 0)
    printf("Finished Calculation.\n");
#endif

#ifdef PARALLEL
  freeArray("newPsiR", newPsiR, (height+2*radius)*(width+2*radius), sizeof(dataType));
  freeArray("newPsiI", newPsiI, (height+2*radius)*(width+2*radius), sizeof(dataType));
  freeArray("oldPsiR", oldPsiR, (height+2*radius)*(width+2*radius), sizeof(dataType));
  freeArray("oldPsiI", oldPsiI, (height+2*radius)*(width+2*radius), sizeof(dataType));
  freeArray("olderPsiR", olderPsiR, (height+2*radius)*(width+2*radius), sizeof(dataType));
  freeArray("olderPsiI", olderPsiI, (height+2*radius)*(width+2*radius), sizeof(dataType));
  freeArray("seeds", seeds, (height+2*radius)*(width+2*radius), sizeof(char));
#else
  free(newPsiR);
  free(newPsiI);
  free(oldPsiR);
  free(oldPsiI);
  free(olderPsiR);
  free(olderPsiI);
  free(seeds)
#endif

#ifdef DEBUG
  if (id == 0 && node == 0)
    printf("Freed psi arrays.\n");
#endif

  free(yMin);
  free(yMax);
  free(aString);

#ifdef DEBUG
  if (id == 0 && node == 0)
    printf("Freed remaining arrays and exiting run().\n");
#endif
}

#ifdef PARALLEL
void freeArray(char *aString, void *array, int length, int size)
{
  char *temp;
  int a;

  temp = (char *)calloc(strlen(aString)+15, sizeof(char));

  //not sure this is necessary but a harmless safety precaution.
  MPI_Barrier(comm);

  munmap(array, length * size);

  //everyone must be ready before we remove the shared memory.
  MPI_Barrier(comm);

  //process zero deallocates the shared memory so that subsequent runs of the program won't
       already have values initialized.
  if (id == 0)
  {
    sprintf(temp, "%s-%d", aString, node);
    shm_unlink(temp);
```

```c
  }

  free(temp);
}

void *createArray(char *aString, int length, int size)
{
  void *array;
  char *temp;
  int fd, a;

  temp = (char *)calloc(strlen(aString)+15, sizeof(char));

  if (id == 0)
  {
    //the node identifier is there in case nodeSize is decreased so multiple virtual "nodes" are
         on the same physical compute node.
    sprintf(temp, "%s-%d", aString, node);
    //Removes any previous references to this sharedmemory (because this program crashes a lot so
          freeArray doesnt get called).
    shm_unlink(aString);
    //Opens file of a column of pointers to long doubles.
    fd = shm_open(aString, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

    if (fd == -1) //checks for errors. Dunno what exactly as this is Sorenson's code (thanks!).
    {
      perror("ERROR: createshm:shm_open:\n");
      exit(1);
    }

    //Allocates shm space sufficient to hold all those pointers (or long doubles in this case...)
         .
    if (ftruncate(fd, length * size) == -1)
    {
      perror("ERROR: createshm:ftruncate:\n");
      exit(1);
    }

    //maps the array to this shm/file.
    array = mmap(NULL, length * size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (array == MAP_FAILED)
    {
      perror("ERROR: createshm:mmap:\n");
      exit(1);
    }
  }

  //groups up all the processes. Everyone waits until ID 0 one shows up.
  MPI_Barrier(comm);

  //All Processes except process 0 run this code (which makes the above waiting kind of funny).
  if (id > 0)
  {
    //the node identifier is there in case nodeSize is decreased so multiple virtual "nodes" are
         on the same physical compute node.
    sprintf(temp, "%s-%d", aString, node);
    //this only connects to the existing shared memory as opposed to creating it like what ID 0
         did.
    fd = shm_open(aString, O_RDWR, S_IRUSR | S_IWUSR);

    if (fd == -1) //random error that I know nothing about...
    {
      perror("ERROR: shm open error in getshm\n");
      exit(1);
    }

    //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.

    //maps to the shm.
    array = mmap(NULL, length * size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    if (array == MAP_FAILED)
    {
      perror("ERROR: createshm:mmap:\n");
      exit(1);
    }
  }

  free(temp);

  return array; //returns the awesome shm memory 2D array of awesomeness.
}
#endif

//function for each iteration using only 1 previous time step.
void functionOld(dataType *oldPsiR, dataType *oldPsiI, dataType *newPsiR,
  dataType *newPsiI)
{
  int a, b;

  //Calls are being made to neighbor cells so they need to be up-to-date.
#ifdef PARALLEL
  MPI_Barrier(comm);
#endif

  for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a + radius; b < a + width + radius; b++)
    {
      newPsiR[b] = oldPsiR[b] - (hbar*timeStep/(2*mass))
        *(oldPsiI[b+1] + oldPsiI[b-1] + oldPsiI[b+adj] + oldPsiI[b-adj]
        - oldPsiI[b] - oldPsiI[b] - oldPsiI[b] - oldPsiI[b]);
      newPsiI[b] = oldPsiI[b] + (hbar*timeStep/(2*mass))
        *(oldPsiR[b+1] + oldPsiR[b-1] + oldPsiR[b+adj] + oldPsiR[b-adj]
        - oldPsiR[b] - oldPsiR[b] - oldPsiR[b] - oldPsiR[b]);
    }
}

//function for each iteration of the solution using 2 previous time steps.
void functionNew(dataType *oldPsiR, dataType *oldPsiI, dataType *psiR,
  dataType *psiI, dataType *newPsiR, dataType *newPsiI)
{
  int a, b;

  //Calls are being made to neighbor cells so they need to be up-to-date.
#ifdef PARALLEL
  MPI_Barrier(comm);
#endif

  for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a + radius; b < a + width + radius; b++)
    {
      newPsiR[b] = oldPsiR[b] - (hbar*timeStep/mass)
        *(psiI[b+1] + psiI[b-1] + psiI[b+adj] + psiI[b-adj]
        - psiI[b] - psiI[b] - psiI[b] - psiI[b]);
      newPsiI[b] = oldPsiI[b] + (hbar*timeStep/mass)
        *(psiR[b+1] + psiR[b-1] + psiR[b+adj] + psiR[b-adj]
        - psiR[b] - psiR[b] - psiR[b] - psiR[b]);
    }
}

int detect(dataType *psiR, dataType *psiI, char *seeds)
{
  int a, b, c, exit = FALSE;
  long double sum = 0, percent = ((long double)rand())/RAND_MAX;
  long double sums[np];

  //Calculate the sum of the probabilities in each region.
  for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a+radius; b < a+width + radius; b++)
      if (seeds[b] == FALSE && (seeds[b-adj] || seeds[b-1] || seeds[b+adj] || seeds[b+1]))
        sum += psiR[b]*psiR[b] + psiI[b]*psiI[b];

#ifdef PARALLEL
  MPI_Gather(&sum, 1, MPI_LONG_DOUBLE, sums, 1, MPI_LONG_DOUBLE, 0, comm);
#else
  sums[0] = sum;
#endif

  sum = 0;

  //Let process zero do the actual detection in the region with the detection.
  if (id == 0)
  {
    //Determine which region has the detection.
    for (c = 0; c < np; c++)
    {
      sum += sums[c];
      if (sum >= percent)
        break;
    }

    if (c == np) //There is no detection.
      goto nestedLoopBreak;

    sum -= sums[c]; //Went too far, so let's go back one step.


    for (a = yMin[c]*adj + radius*adj; a <= yMax[c]*adj + radius*adj; a += adj)
      for (b = a+radius; b < a+width + radius; b++)
        if (seeds[b] == FALSE && (seeds[b-adj] || seeds[b-1] || seeds[b+adj] || seeds[b+1]))
        {
          sum += psiR[b]*psiR[b] + psiI[b]*psiI[b];

          if (exit == FALSE && sum >= percent)
          {
            seeds[b] = TRUE;
            exit = TRUE;
            y = a / adj - radius;
            x = b - a - radius;

            //Top-Bottom
            if (a == radius*adj)
              seeds[(height+radius)*adj+b-a] = TRUE;
            else if (a == (height - radius)*adj)
              seeds[b-a] = TRUE;

            //Left-Right
            if (b-a == radius)
              seeds[a+(width+radius)] = TRUE;
            else if (b-a == radius + width - 1)
              seeds[a] = TRUE;

            goto nestedLoopBreak;
          }
          else
            psiR[b] = psiI[b] = 0.0;
        }
  }

  nestedLoopBreak:

#ifdef PARALLEL
  MPI_Barrier(comm); //Wait for the detection before erasing.
#endif

  for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a+radius; b < a+width + radius; b++)
      if (seeds[b] || seeds[b-adj] || seeds[b-1] || seeds[b+adj] || seeds[b+1])
        psiR[b] = psiI[b] = 0.0;


#ifdef PARALLEL
  MPI_Bcast(&exit, 1, MPI_INT, 0, comm);
#endif

  normalize(psiR, psiI);

  return exit;
}

void borders(dataType *psi)
{
  int a, b;

  //Assumes periodic boundaries.
  for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a; b < a+radius; b++)
    {
      psi[b] = psi[b+width];
      psi[b+width+radius] = psi[b+radius];
    }

#ifdef PARALLEL
  MPI_Barrier(comm); //The borders are being arranged so all the cells need to be ready
#endif

  if (id == 0)
  {
    for (a = 0; a < radius*adj; a += adj)
      for (b = a; b < a+width + radius + radius; b++)
        psi[b] = psi[b+height*adj];
  }

  if (id == np-1)
  {
    for (a = 0; a < radius*adj; a += adj)
      for (b = a; b < a+width + radius + radius; b++)
        psi[b+(height+radius)*adj] = psi[b+radius*adj];
  }
}

//writes the psi matrix to a file in comma delimited form
void writePsi(char *fileName, dataType *psiR, dataType *psiI)
{
  FILE *file;
  char *text;
  int a, b;

  text = (char *)calloc(strlen(directory)+strlen(fileName)+strlen(extension)+15, sizeof(char));
  sprintf(text, "%s/Particle %s.%s", directory, fileName, extension);
  file = fopen(text, "w");

  for (a  = radius; a < radius + height; a++)
  {
    for (b = radius; b < radius + width; b++)
      fprintf(file, "%.*LG,", psiR[(a)*adj+(b)]*psiR[(a)*adj+(b)] + psiI[(a)*adj+(b)]*psiI[(a)*
           adj+(b)], DIGITS);
    fprintf(file, "\n");
  }

  fclose(file);
  free(text);
}

void writeSeed(int particles, int detections, double particleTime)
{
  FILE *file;
  char *text;
  double realTime;

  //Only one process should write to a file at a time.
  if (id != 0)
    return;


#ifdef PARALLEL
  realTime = MPI_Wtime() - timerStart;
#else
  time_t seconds;
  time(&seconds);
  realTime = seconds - timerStart;
#endif

  text = (char *)calloc(strlen(directory)+strlen(extension)+10, sizeof(char));
  sprintf(text, "%s/seed.%s", directory, extension);

  file = fopen(text, "a");

  fprintf(file, "%d,%d,%d,%d,%f,%f\n", detections, particles, y, x, particleTime, realTime);

  fclose(file);
  free(text);
}

//Fills the matrix with the initial condition of the system
void init(dataType *psiR, dataType *psiI)
{
  long double coef;
  int a, b;

#ifdef PARALLEL
  MPI_Barrier(comm);
#endif

  for (a = yMin[id]; a <= yMax[id]; a++)
    for (b = 0; b < width; b++)
    {
      //Normal starting condition of gaussian wave packet.
      coef = expl((-(b-xCenter)*(b-xCenter)-(a-yCenter)*(a-yCenter))/(particleSize*particleSize))
           /sqrtl(particleSize*particleSize*PI/2);
      psiR[(a+radius)*adj+(b+radius)] = coef;//*cosl(Vx*b+Vy*a);
      psiI[(a+radius)*adj+(b+radius)] = 0.0;//coef*sinl(Vx*b+Vy*a);

    //Uncomment these lines for infinite width starting condition.
    //psiR[(a+radius)*adj+(b+radius)] = cosl((b + a)*4*PI/(width+height));
    //psiI[(a+radius)*adj+(b+radius)] = sinl((b + a)*4*PI/(width+height));
  }

  normalize(psiR, psiI);
  borders(psiR);
  borders(psiI);
}

void normalize(dataType *psiR, dataType *psiI)
{
  long double temp, sum = 0;
  int a, b;

  for (a = yMin[id]*adj + radius*adj; a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a+radius; b < a+width+radius; b++)
      sum += psiR[b]*psiR[b] + psiI[b]*psiI[b];

#ifdef PARALLEL
  temp = sum;

  MPI_Allreduce(&temp, &sum, 1, MPI_LONG_DOUBLE, MPI_SUM, comm);
#endif

  sum = sqrtl((long double)sum);

  for (a = yMin[id]*adj + radius*adj;a <= yMax[id]*adj + radius*adj; a += adj)
    for (b = a+radius; b < a + width + radius; b++)
    {
      psiR[b] /= sum;
      psiI[b] /= sum;
    }
}

void seedBorders(char *seeds)
{
  int a, b;

  //Assumes periodic boundaries.
  for (a = yMin[id] + radius; a <= yMax[id] + radius; a++)
    for (b = 0; b < radius; b++)
    {
      seeds[(a)*adj+(b)] = seeds[(a)*adj+(b+width)];
      seeds[(a)*adj+(b+width+radius)] = seeds[(a)*adj+(b+radius)];
    }

#ifdef PARALLEL
  MPI_Barrier(comm); //The borders are being arranged so all the cells need to be ready
#endif

  if (id == 0)
  {
    for (a = 0; a < radius; a++)
      for (b = 0; b < width + radius + radius; b++)
        seeds[(a)*adj+(b)] = seeds[(a+height)*adj+(b)];
  }

  if (id == np-1)
  {
    for (a = 0; a < radius; a++)
      for (b = 0; b < width + radius + radius; b++)
        seeds[(height+radius+a)*adj+(b)] = seeds[(a+radius)*adj+(b)];
  }
}

//Fills the matrix with the initial condition of the system
int seedInit(char *seeds)
{
  int a, b, d = 0;

  if (id == 0)
    for (a = 0; a <= height; a++)
      for (b = 0; b < width; b++)
        if (seedStart(a, b) == TRUE)
        {
          seeds[(a+radius)*adj+(b+radius)] = 1;
          y = a;
          x = b;
          d++
          writeSeed(d, d, 0.0);
        }

#ifdef PARALLEL
    //Share detection count with every process and act as a barrier.
    MPI_Bcast(&d, 1, MPI_INT, 0, comm);
#endif

  seedBorders(seeds);

  return d;
}

int seedStart(int y, int x)
{
  if (seedType == NO_SEED)
    return FALSE;
  else if (seedType == RANDOM_SEED)
    return rand() % 2;
  else if (seedType == DOT_SEED)
    if ((seedCenterX - x)*(seedCenterX - x) <= seedRadius * seedRadius && (seedCenterY - y)*(
         seedCenterY - y) <= seedRadius*seedRadius)
      return TRUE;
    else
      return FALSE;
  else if (seedType == WALL_SEED)
    if ((seedCenterX - x)*(seedCenterX - x) <= seedRadius * seedRadius)
      return TRUE;
    else
      return FALSE;
  else
    return FALSE;
}

void centerParticle(long double percent)
{
  long double theta;

  if (id == 0)
    theta = ((long double)rand()) * 2 * PI / RAND_MAX;

#ifdef PARALLEL
  MPI_Bcast(&theta, 1, MPI_LONG_DOUBLE, 0, comm);
#endif

  xCenter = cosl(theta) * (particleMinRadius + (particleMaxRadius - particleMinRadius)*percent) +
        seedCenterX;
  yCenter = sinl(theta) * (particleMinRadius + (particleMaxRadius - particleMinRadius)*percent) +
        seedCenterY;
}
```