Simulations of Nanoscale Spatial Disorder

by

Ethan Gabriel Greif Howe

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by .....
Vladimir Bulović
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Simulations of Nanoscale Spatial Disorder

by

Ethan Gabriel Greif Howe

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we detail the design, implementation, and testing of two simulations of nanometer scale disorder phenomena important for electronic device fabrication. We created a kinetic simulator for the surface assembly of quantum dots into ordered or disordered monolayers. We utilized a platform for high-precision motion and collision resolution and implemented the behavior of quantum dots on a surface. The simulation demonstrated experimentally observed behavior and offers insights into future device design. We also created a material simulation of the electrochemical oxidation of a metal surface with nanoscale roughness. We demonstrated that by preserving the amount of metal and making the oxide coating conformal, anodization can highly planarize the metal surface. We verify the convergence of our results as we increase the accuracy of our model. We demonstrate differences in the rate of planarization between additive and subtractive surface features which could not be observed by experiment and make predictions about the planarization of metals with different oxide expansion coefficients.

Thesis Supervisor: Vladimir Bulović
Title: Associate Professor

Acknowledgments

I would first like to thank all my friends and coworkers in the Laboratory of Organic Optics and Electronics. I have learned so much from all of you even though I was usually coding away at my desk while you all were in the lab. I gained so much knowledge and enthusiasm about organic electronics through your presentations and impromptu discussions. Thank you Vladimir for giving me the opportunity to work with you this year, for guiding my research, and for being the most supportive and good-humored advisor. I would like to thank my family for their encouragement even though they do not quite understand what I have been working on. Thanks to my friends who were always there to distract me in between my work. I would like to specially thank my 6.170 group from a few years ago who built some of the code that I used to pursue my quantum dot simulation. I would also like to specially thank Peter Mardilovich our visiting expert from HP whose experimental data is the basis for my planarization simulation and with whom I worked in its creation. I would finally like to thank MIT for being a home to me for the past five years and for being a great environment in which to succeed academically.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We, as scientists and engineers, are everyday pushing the boundaries of understanding and application. In electrical engineering, there is a long-standing trend towards being able to accurately manipulate increasingly minute systems. By controlling nanoscale structures, we are increasing the possibilities of our systems from the ever increasing computing power of silicon, to the amazing prospect of quantum computing, to the possibility of printing any electronic device quickly and cheaply. Today, systems exist that can not only image single atoms but manipulate them on a surface. We are also able to process increasing numbers of these atomic scale features. At the same time, the increasing speed of computers is allowing us to build more intricate models of the physical processes we seek to probe. Simulation has become an indispensable part of almost all areas of scientific research and engineering. Simulation allows the researcher to test his or her hypotheses in a controlled environment. Modeling can also bridge the gap between individual phenomena and large complex systems, guiding experiments by predicting new observations. My thesis focuses on two simulations that I have created to describe nanoscale spatial disorder. The first simulation describes a novel technique for creating extremely flat metal surfaces through anodization. Our model reproduces all aspects of the experiments it describes. We have shown through the simulation an understanding of the detailed mechanism for this surface planarization. In addition, we have made predictions of how this method anodization could work to planarize other metals. We developed a second simulation to describe the

formation of an ordered monolayer of quantum dots. Several simulations of nanocrystallite motion on surfaces already exist. Most of these models describe the system as a lattice of points that can be occupied by the crystallite or some other species and describe their motion as a probability of transferring to an adjacent lattice site. Other simulations exist for how a large number of three-dimensional spheres rearrange themselves into a stable bulk solid. Our simulation blends the ideas of packing and nanoparticle motion, adding definite crystallite shape to surface simulations and adding a description of realistic motion to sphere packing simulations. Our primary objective was to examine how packing changes with the distribution of dot diameters. We created a broad framework model that can act as a tool for examining almost infinite configurations of surface composition, device features, and dot types. We hope that this simulation can be used as a tool for further experimentation and understanding. These two simulations are only a small sampling of all the possibilities in simulating spatial disorder on the nanoscale. As we improve our descriptions of these tiny phenomena in simulation, we will increase our understanding of these systems and skill in manipulating them.

Chapter 2

Quantum Dot Packing Simulation

We have created a simulation of the packing of monolayer of quantum dots on a surface. The basis for our model is kinetic motion and partially inelastic collision of the dots with surface-mediated thermal motion and van der Waals interactions between dots. The position and velocity of the dots are represented with floating-point accuracy, as is the boundary of the dots. The dots are represented as perfect circles moving in a plane. In addition to dots, our simulation includes stationary objects such as hard walls and regions with differing surface properties. The simulation is intended not only for this work but also as a tool that other researchers and designers can easily use and extend. We have included an extensive graphical user interface (GUI) and a visual representation of the progress of a simulation in real-time. We have also written our code in an easily understandable and extensible manner, using object-oriented programming and a well designed object hierarchy. We have simulated several experiments. Our first goal was to reproduce experimental observations that increasing the diameter distribution first caused quantum dot monolayers to become less well packed and then caused the monolayer order to break down. We were able to reproduce this phenomenon and we were then able to use the freedom available in our simulation to examine possible causes for this breakdown, including differences in dot effective mass and insufficient kinetic damping. We also included several calculations important for assessing both the results of our simulations and measurable characteristics of quantum dots in the lab.

2.1 Theory and Background

A large amount of research has been done on sphere packing along many different avenues. In two dimensions, Gauss proved that hexagonal packing is the densest of all plane lattice circle packings, but not until 1940 did L. Fejes Tóth prove that it is also the densest possible packing overall for same sized circles [3]. In three dimensions, the problem took even longer to solve conclusively. The assertion that the densest packing of identical spheres was so-called close packing, either in a face-centered cubic or hexagonal lattice, was called the Kepler conjecture after Johannes Kepler who posed it in 1611 [4]. Gauss was also able to show that face-centered cubic was the densest lattice packing of spheres. While few people doubted Kepler's conjecture, the final proof by Thomas C. Hales did not come until 1998, and it involved breaking down the problem into a large system of linear equations solved by computer [5]. There are a number of such geometric problems still under investigation, including random packing, sphere packing in different dimensions, random loose packing, and ellipsoid packings. The formation and motions of spherical structures has also been intensely researched [4].

Quantum dots are inorganic semiconductor nanoparticles having diameters smaller than the Bohr radius of excitons in the bulk material. This property confines electrons and holes to particular lattice excitations and quantizes the possible energy levels in the dot. This arrangement makes quantum dots of the same size highly homogeneous light emitters. Because of this property, quantum dots are being used in optical communications [6], lasers [7], light detectors [8], and even as tags in biological research [9]. There exist several types of quantum dots, including lithographically etched, epitaxially grown, and colloidal. Our simulation focuses on colloidal quantum dots that are deposited on a surface [10]. This type of quantum dot is synthesized in solution by chemical and heating processes. The quantum dot core is usually coated by a protective shell of higher bandgap semiconductor with matching lattice constant such as CdSe covered by CdS or ZnSe [11] [12]. The shell passivates the core as well as protects and physically separates the core from its surroundings. This setup also

enhances the properties of the quantum confinement by making it less prone to variations due to local environment. Finally, to allow the dots to remain easily soluble and to allow incorporation into a number of structures including organic electronic devices or biological systems, the dots can by overcoated with many types of organic molecules [13].

A large portion of the literature on packing and granular flow focuses on the packing of glass microspheres with radii between $1 - 1000\mu m$ or charge-stabilized polystyrene microspheres with radii smaller than $1\mu m$. Experimental studies and simulations of three-dimensional structures of these nanospheres has also been accomplished in air, vacuum, and liquid [14] [15]. We are interested in the subset of this theory which will apply at the smaller scale of dots. Also, since we are investigating the behavior of the dots on a surface, we can realistically ignore most interactions with the liquid, including buoyancy, and Magnus lift [15], and factor them into the random thermal motion parameter. The two remaining most important forces are collision and van der Waals interactions. The nonlinear Hertz model collision force on particle i from particle j with radii R_i, R_j, positions $\vec{r_i}$, $\vec{r_j}$, and relative velocity $\vec{v}_{ij}$ is given in Equation 2.1 where Y is Young's modulus and ν is the Poisson ratio [16] (see Figure 2-1).

$$\begin{aligned} F_{ij}^{n} &= [\frac{2Y}{3(1-\nu^2)}\sqrt{R_{ij}^{eff}}\xi_n^{3/2} - \frac{\gamma_n Y}{1-\nu^2}\sqrt{R_{ij}^{eff}\xi_n}(\vec{v}_{ij}\cdot\hat{n}_{ij})]\hat{n}_{ij} \\ R_{ij}^{eff} &= \frac{R_i R_j}{R_i + R_j} \\ \xi_n &= R_i + R_j - (|\vec{r_i} - \vec{r_j}|) \end{aligned} \tag{2.1}$$

The first term in the equation is an elastic force that accounts for the distortion and restoration of the particles. The second term is a dissipative component due to energy loss in the center of the particle from the deformation. The constant γ_n in this term is the normal damping constant that is related to the normal coefficient of restitution [14]. This component is dependent upon the rate at which the deformation occurs because it relates to shearing energies and the rate at which energy must be dissipated [16]. This force will be the main source of energy loss in our simulation.

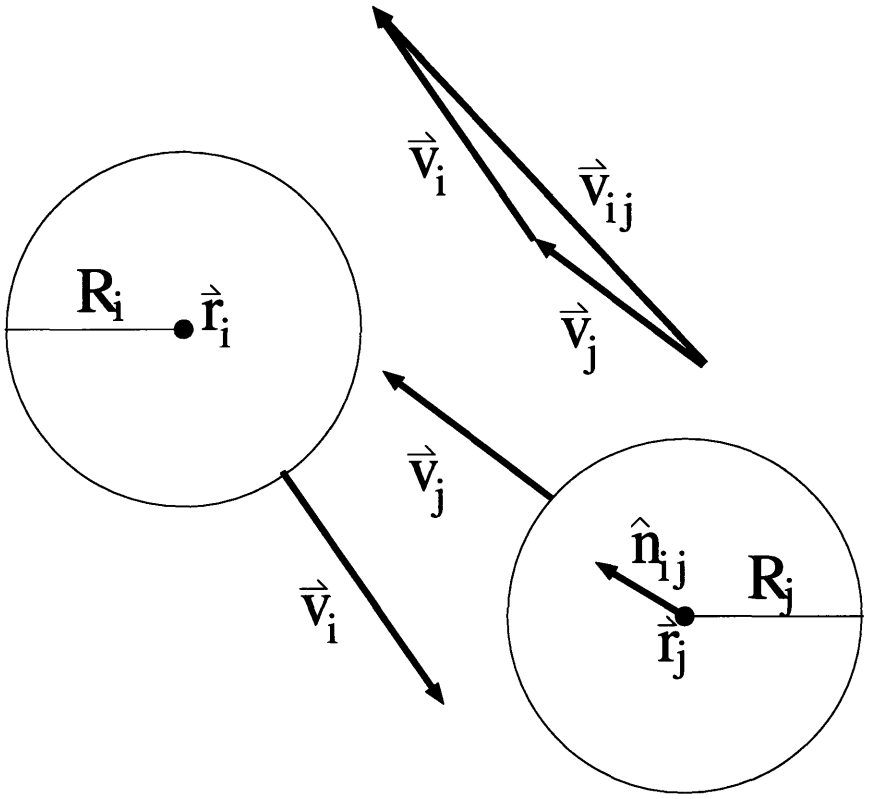


Figure 2-1: A schematic representation of two dots colliding. R_i and R_j are the radii of dots i and j, respectively. $\vec{r}_i$ and $\vec{r}_j$ are the dot positions. $\vec{v}_i$ and $\vec{v}_j$ are the velocities and $\vec{v}_{ij}$ is the relative velocity of the dots.

We will ignore tangential forces between dots during collisions because the small size of the dots make these frictional-type forces minimal as well as hard to calculate without accounting for molecular orientations and interactions. Thus the tangential component of dot collisions will be taken to be purely reflective.

The van der Waals force, core to core, is the main attractive force between quantum dots [17]. Other important forces include short-range atomic interactions and long-range electrostatic forces. The short-range atomic forces mainly serve to keep the dots separate and do not add significantly to attraction. Thus we can model these forces as a nearly-hard dot radius. The long-range electrostatic forces are negligible for uncharged dots in solution because the liquid dielectric screens this effect. The van der Waals force between two spheres, i and j, with non-negligible radii, R_i and

R_j, and a separation of h between the spheres is given in Equation 2.2 [14] [18].

$$F_{ij}^{v} = -\frac{H_a}{6} \times \frac{64R_i^3R_j^3(h+R_i+R_j)}{(h^2+2R_ih+2R_jh)^2(h^2+2R_ih+2R_jh+4R_iR_j)^2}\hat{n_{ij}} \tag{2.2}$$

The Hamaker constant, H_a, of CdSe is 0.388 [19]. The van der Waals force is exerted mainly between the cores of the quantum dots [17] so we can use this equation reasonably accurately for CdSe/ZnSe or CdSe/CdS core/shell dots coated with organic molecules. With this theory, we can build a good model of quantum dots being deposited in a monolayer on a surface.

Once we simulate a stable monolayer of quantum dots, we would like to examine some properties of that monolayer, which are strongly affected by the packing. Since quantum dots are generally used for their optical properties, exciton transfer and relaxation in the dots is particularly important. Exciton transfer in quantum dots takes place through long-range dipole-dipole interactions described by Förster theory [20]. We are concerned with singlet exciton transfer between individual dots. For this case, the transfer rate between two dots, W, is given in Equation 2.3 [21].

$$\begin{aligned} W &= \chi(R)P(E_d,E_a) \\ \chi(R) &= \frac{1}{\tau}\left(\frac{R_F}{R}\right)^6 \\ P(E_d,E_a) &= \begin{cases} exp[-(E_a-E_d)/k_BT] & E_a > E_d, \\ 1 & \text{otherwise} \end{cases} \end{aligned} \tag{2.3}$$

where E_a and E_d are the energies of the acceptor and donor molecules, respectively, k_B is Boltzmann's constant, T is temperature, τ is the radiative lifetime, R_F is the Förster radius, and R is the distance between dots. If a dot contains an exciton, then the transfer rates of all of the surrounding dots compete to determine, to which site the exciton will hop. Generally, the Förster radius for quantum dots is small enough such that excitons only hop to their nearest-neighbors. The exciton can also radiatively decay with probability $1/\tau$ at any time. Similarly, there exist non-radiative decay pathways that allow excitons to recombine without emitting light.

These pathways generally involve interaction with the media surrounding the dots but can originate from defects in the dot structure or the excitation of a charged dot. The fraction of excitons introduced into the system that decay radiatively is known as the luminescence quantum efficiency (LQE). This figure can be used to compare the efficiency of different lumophores for a given device structure if exciton formation is separate from luminescence. In our case, we want to compare the LQE of different dot packing and assess any performance degradation from poor monolayer coverage separately.

The formation of monolayers is important for the creation of efficient quantum dot light-emitting devices (QD-LEDs) [22]. We can create such a layer by spin-casting a solution of QDs and organics having the correct chemistry such that the QDs phase-separate out of the organics after deposition. Then, with the correct concentration of QDs, a complete, low-defect monolayer of QDs can be formed on top of the organic layer. The distribution of sizes in the QD sample has been shown to affect the formation and quality of the monolayer [23]. For highly monodisperse QDs having a standard deviation, σ, less than 5%, the monolayer on a particular substrate was shown to have nearly perfect hexagonal close-packed structure. Moving to broader size distributions, with $\sigma = 5 - 10\%$, the monolayer showed significant point and line defects, but it retained good surface coverage. For QD samples where $\sigma > 10\%$, the monolayer broke down completely, forming aggregates with no periodicity and large uncovered areas. While QD samples with $\sigma < 5\%$ can readily be synthesized [24], larger size distributions as well as mixtures of different dots sizes could be useful for some applications such as white light QD-LEDs.

2.2 Simulation Algorithm and Design

The code for this project was built on a program for user-reconfigurable computer games including pinball and breakout [25]. To adapt this code for our simulation, we greatly improved the accuracy of collisions, implemented new methods of interaction to describe our nanoscale model, added new analysis and output features, and

augmented the user interface. The simulation is based on a finite time step, collision look-ahead, and resolution. It includes collisions between two moving circles, a moving circle and a stationary circle or polygon, and a moving circle and a moving, non-recoiling circle or polygon. The simulation enforces a hard wall boundary between all objects and can apply a constant force to all moving objects or interaction forces between pairs of objects. The code is written exclusively in Java and is interoperable on many platforms. The simulation runs in real-time or nearly real-time on modern single-processor computers and as such is viable as a design tool.

Our main algorithm for moving forward one time step is (for relevant code excerpts see Appendix A):

1. Add random thermal velocities to all dots if the preset time has elapsed since the last randomization.

2. Update the data structure that allows us to find dots that are close to each other.

3. Add any interparticle forces by updating the velocities of pairs of dots that are within a minimum distance of each other.

4. Start loop that resolves all collisions in a given time step.

5. If there are no more collisions or a maximum number of iterations has been reached then end the loop.

6. Calculate and save the time until the first collision for each dot that needs to be updated (on the first loop all dots need to be updated).

7. Find the first collision that will occur within the remainder of the time step.

8. If no collisions will occur, then move all dots along their velocity vectors for the amount of time left in the step, and signal that there are no more collisions to look for.

9. If there is a collision then update the position of all non-colliding dots along their velocity vectors for the amount of time until the collision. Also subtract this amount of time from the saved time until collision for each dot.

10. Resolve the single collision and mark any dots involved as needing their next collision to be calculated. Subtract the time needed for this collision from the time left in the time step.

11. If the time left in the time step is greater than zero, return to step 5.

We created this algorithm as the core of all the behavior in our simulation. Before running this code, dots are introduced to the surface either through the GUI or inside other functions. The dots have several core characteristics including position, velocity, radius, mass, and color. For our later observations, we have also added secondary characteristics such as whether the dot is in an excited state and how efficiently the dot luminesses. Stationary objects are added in the same way with their own internal variables including position, rotation, and color. We can access the simulation either through the GUI or a programmatic interface. In both cases, the simulation proceeds in its own thread, which activates the above routine at the time intervals equal to the simulation time step. This time step can be adjusted, as can all of the model parameters. If the computation time of the algorithm stays below the amount of time represented, then the simulation will proceed in real time. The computational complexity of this algorithm is given in Equation 2.4 using asymptotic notation where n is the number of dots and Δt is the length of the time step.

$$f(n, \Delta t) = O(\frac{n^2}{\Delta t}) \tag{2.4}$$

The computation time depends on $1/\Delta t$ because shorter time steps require more calculations to check for collisions. The collision detection algorithm provided in the physics package is n^2 because it must check pairwise collisions. Also our algorithm is n^2 itself because we advance all n dots for each collision and the number of collisions grows approximately as the number of dots grows.

Particular details of our outlined algorithm are important for correct operation. Our difficulties arise from having to ensure that the dots behave correctly over small collision distances with many surrounding dots, as in the case of close-packing. We have chosen to advance all the dots together for each collision up to the time of that collision; that way, all non-accelerative motions of the dots can be exactly correct. Our collision detection considers only the position and velocity of the dots. Acceleration, either from attraction or thermal motion, is added only at the beginning of each time step as a change in velocity. Thus as we make our time steps increasingly small, our acceleration and forces become more accurate. In this way, we can easily calculate if and when a collision will occur between two objects, either two dots or a dot and a stationary or non-recoiling surface. If we advance time by moving all dots to just before the first calculated collision, then no other collisions should have happened and no overlap should occur. Also since we move all the dots together, each dot is at the same point in time. If we moved only the dot or dots that were colliding, then we would be advancing their times farther than the others. Then when we checked for collisions, we would have to account for the differences in time. Our method simplifies these considerations and guarantees that all collisions happen at the correct times and places. Our next concern is computational. Consider the case in which we collide two dots by placing them directly next to each other; a third dot may also be about to collide in a small distance. Since we represent positions with floating point numbers, if we continue this process many times, for instance if the dots are being pressed together, then the round-off error on the positions accumulate, and the dots may overlap, causing our model to fail. Note that when overlaps occur collisions can no longer be calculated correctly, and if dots are well packed then correcting for the overlap can be difficult without disturbing our simulation. To avoid this problem, we enforce a minimum distance around all dots. When calculating the time to collision, we increase the radius of each dot by one millionth of the average radius. Then when we find the new velocities after collisions, the dots are always a finite distance away from the other object. Once we resolve this single collision, we need to update the collisions we previously calculated. The data for the dots that just collided is

obviously invalid because their velocities have changed. Thus we need to recalculate the collisions for these one or two dots. All of the other dots still have the same velocities and have only moved forward in time. Therefore their next collision is still valid unless that next collision is with one of the dots that just collided. We first subtract the amount of time that all the dots moved forward from their saved next collision time. Second, when recalculating the collisions for previously collided dots, if the dot will collide with another dot we check if the time is less then the collision time already saved, and if so we update the other dot as well. Using this procedure, we can continue indefinitely in our simulation.

In our collision resolution procedure, we have incorporated inelastic collisions. All objects in our simulation are defined by a set of line segments and circles. Our collision detection specifies the first line segment, circle, or other dot, with which each dot will collide. We have not implemented inelastic collisions with stationary objects. When a dot collides with a line segment or circle, we simply utilize the geometric calculations of our physics package. When two dots collide, we want to approximate the forces described in Section 2.1. We have simplified the force of collisions (Equation 2.1) into a model parameter that determines the amount of elasticity in dot-dot collisions. We have not implemented inelastic collisions with stationary objects. When a dot collides with another dot, we calculate the new velocities of the dots, v_1^n and v_2^n. We separate the parallel and perpendicular components of this velocity, $v_\parallel^n$ and $v_\perp^n$, relative to the direction of the collision. Then we apply the following equations to these components:

$$\begin{aligned} v_{\parallel 1}^f &= \epsilon v_{\parallel 1}^n + (1-\epsilon)\frac{m_1 v_{\parallel 1}^n + m_2 v_{\parallel 2}^n}{m_1 + m_2} \\ v_1^f &= v_{\parallel 1}^f + v_{\perp 1}^f \\ v_{\parallel 2}^f &= \epsilon v_{\parallel 2}^n + (1-\epsilon)\frac{m_1 v_{\parallel 1}^n + m_2 v_{\parallel 2}^n}{m_1 + m_2} \\ v_2^f &= v_{\parallel 2}^f + v_{\perp 2}^f \end{aligned} \tag{2.5}$$

where ϵ is the elasticity parameter that ranges from 0 to 1 and m_1 and m_2 are the masses of the dots. We note that if $\epsilon = 1$ then the final velocities of the dots parallel to the collision will be equal, and a large amount of energy will be lost.

We apply forces between dots at the beginning of each time step. For our simulation, the main force is a van der Waals attraction between all pairs of dots. To save computational time, we only apply the force between dots that are within some distance of each other. This optimization is minimal because the van der Waals interaction fall off as h^6 where h is the separation of the dots. To find these pairs in linear time, we created a data structure that hashes the x and y locations of the dots and determines which dots from adjacent bins are within the maximum distance cutoff. Once the dots have been located, we can simply apply Equation 2.2 and change the velocities of each pair of dots. This force creates a cohesion between all the dots that tends to create one large mass of dots. The thermal velocities of the dots counter this cohesion. At some given frequency, each dot has a small velocity with a set magnitude and uniformly distributed direction added to it. This frequency of random velocity addition is related to the time taken to thermalize the dots on the surface. The magnitude of this thermal energy is the last parameter of our model, and it determines the average velocity of a non-interacting dot on the particular surface in the particular solution.

To represent the phase-separation method of monolayer creation described in Section 2.1, we add dots to the surface sequentially. We select diameters from a Gaussian distribution with the set standard deviation. We select positions uniformly over the simulation area. We then check for any overlaps between the introduced dot and existing dots. If an overlap occurs, we start over with a new random diameter and position. We wait a short time between each dot addition and we wait a longer time at the end for the sample to get closer to equilibrium. After this time has elapsed, we make our measurements.

2.3 Simulation Results

As described in the previous section, our simulation contains four model parameters, the collision inelasticity, the strength of dot to dot van der Waals interactions, the average thermal velocity on the surface, and the rate at which thermal velocities

are added to the dots (thermalization rate). While we described calculations for the first two of these quantities in Section 2.1, their accuracy, particularly in solution, is limited. The last two parameters contain information about how the quantum dots interact with the substrate and so they are not easy to calculate. These surface parameters could be resolved from observations of the motion of quantum dots on a surface. For the purposes of this thesis, the values of these parameters were estimated and then tuned using phenomenological observations. In particular, we were hoping to observe the size distribution dependence of monolayer formation. Thus we chose parameters that gave roughly the desired effects over the correct size distributions. As such, our model is still qualitative regarding packing fractions and stability across different samples. Notwithstanding these difficulties, we believe that our simulation is a good tool for understanding quantum dot monolayers.

2.3.1 Dot Packing

We demonstrated close-packing of monodisperse quantum dots on a surface and how instability arises with increasing dot size distribution. We randomly deposited a set of 300 dots into a large area with fixed boundaries. The results of the simulation are shown in Figure 2-2 for uniform-sized dots and for dots with standard deviation equal to 15% of the mean size. We observe that the uniform sample exhibits strong hexagonal packing in three distinct grains. The large distribution sample exhibits no defined lattice structure and there exist many small clusters. In addition, when we observe the motion of this sample, the dots are much less well bound to each other as the small clusters suggest. This instability translates to a physical sample that will not energetically favor a monolayer formation. Since the dots are not well-bound in the two dimensional formation, we expect the dots to decompose in three dimensions or move off the sample area.

We examined the efficacy of confining the dots in order to increase packing and reduce instability. We created a wall at 6 lattice spacings ($6d_{dot}\sqrt{3}/2$) from the perimeter along one direction. We also reduced the number of dots to 150 so that they had some freedom in the structure. In the uniform sample, we observed no

Uniform Sized Dots

15% Standard Deviation

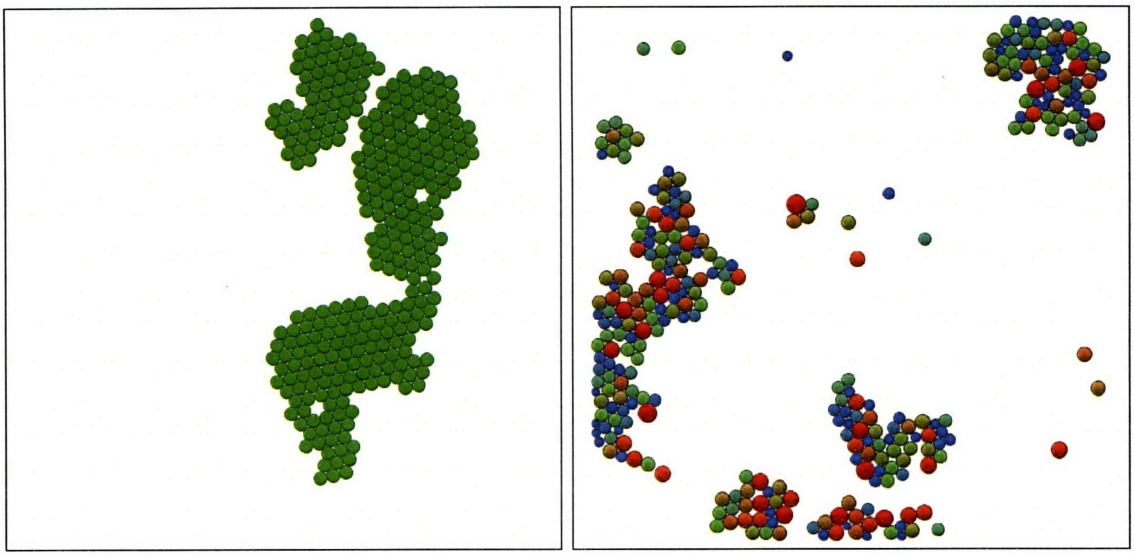


Figure 2-2: a) Shows the results of our simulation for 300 uniform sized quantum dots. The packing is hexagonally close-packed except for a few defects and grain boundaries. b) Shows the results of our simulation for 300 quantum dots whose diameters are determined by a Gaussian distribution with standard deviation equal to 15% of the mean radius. The dots are colored such that larger dots are more red, mean sized dots are green, and small dots are more blue. The dots do not form close-packing and there are several small clusters.

decrease in lattice order except for a slight increase in grain boundaries per dot (see Figure 2-3). Single dot vacancy defects were reduced. We did see however difficulty filling towards the sides of the structure due to the tight binding between the dots. This difficulty would most likely be resolved in a three dimensional situation where dots could settle into place from above. When created in this confined structure, the 15% size distribution saw significant improvement in its stability and packing. The dots formed a cohesive single aggregate and the distances between dots in aggregate form were reduced. This effect originates from a reduction in the average kinetic energy of the dots. Confining the dots increases the number of dot-dot collisions and thus increases the energy loss due to the inelasticity of those collisions.

We quantify packing quality by measuring the number of dots on the inside of a cluster. The lowest energy formation of dots would be a close-packed circle. This arrangement is also the lowest surface area configuration of dots. All interior dots

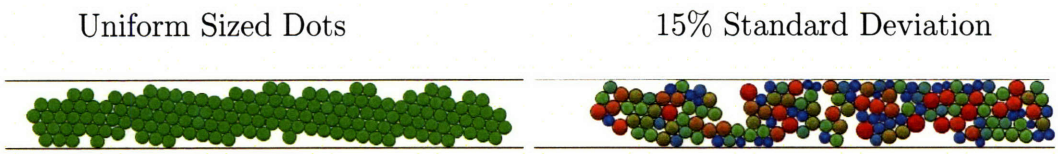


Figure 2-3: a) Shows the results of our simulation for 150 uniform sized quantum dots confined in a grating. The packing is hexagonally close-packed except for a few defects and grain boundaries. b) Shows the results of our simulation for 150 quantum dots whose diameters are determined by a Gaussian distribution with standard deviation equal to 15% of the mean radius confined in a grating. The dots are colored such that larger dots are more red, mean sized dots are green, and small dots are more blue. The dots are more cohesive because confining them increases the number of collisions and reduces the average kinetic energy.

would have 6 neighbors and only a small fraction (proportional to the perimeter of the circle over the area) would be on the exterior. For unstable packing, dots form smaller clusters with amorphous shape and thus decrease the number of internal dots. This measurement gives a good idea of the stability of the monolayer. We determine which dots are internal from the number of neighboring dots. A pair of dots are neighbors if they are separated by less than a mean radius and a dot is considered internal if it has five or more neighbors. The number of neighbors for each internal dot measures the density of packing within each cluster. Figure 2-4 compares free and confined dots on the proportion of interior dots and the mean number of neighbors for a number of size distributions. We observe the expected trends. Increasingly polydisperse dot samples show decreased stability and packing, which is mitigated by confining the dots along one dimension.

2.3.2 Luminescence Efficiency

We calculate the luminescence quantum efficiency of our dots similarly. We have opted for a simple model of non-radiative decay to demonstrate our simulations capabilities. In particular, we designated a fraction of the dots (1/3 for the cases below) to be traps that recombine excitons non-radiatively. We set all dot energies to be equal and used the transfer probabilities defined by Förster transfer with a Förster radius equal to the mean dot diameter. This model is simplistic because the size of the dots shifts

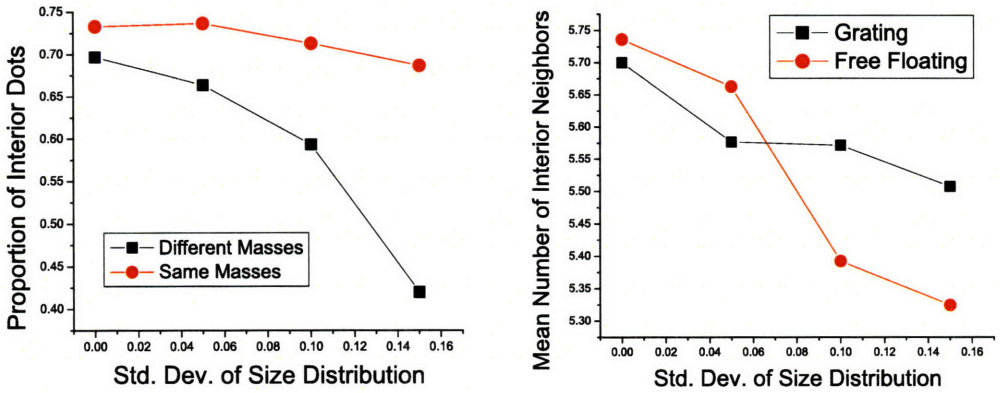


Figure 2-4: a) Shows the proportion of dots that are "interior" for a given size distribution. Circular markers give the data for unconfined dots and the square makers give the data for dots confined along one dimension to 6 lattice spacings (called a grating.). A dot is considered interior if it has 5 or more neighbors (dots separated by less than a mean radius). b) Shows the mean number of neighbors for interior dots. For a circular or monolayer uniform-sized hexagonally close-packed sample, this quantity would be 6.

their spectra and the Förster radius is a function of the spectral overlap between acceptor and donor states. We have made the approximation that all spectra are the same with the same Stokes shift in the excited state and thus have the same Förster radius. We set our time step as one hundredth of the dot radiative lifetime (note that the probability of radiating in each time step is then nearly 0.01). We start by adding an exciton to a random quantum dot. In each time step, we calculate the Förster transfer rate between the exciton and all other dots and multiply that rate by the time step to get the transfer probability. We determine if each transfer could occur by getting a random number between 0 and 1 and checking whether that number is less than the probability of the transfer. We perform the same test to determine if the dot could radiate in a time step. If we have more than one possible change in the exciton, we weight the outcomes by their probabilities and randomly determine which one occurs. If we find no possible transfers and no radiation then the exciton remains on the same dot. Otherwise, we either move the exciton to the next dot and calculate new probabilities or we note that the exciton radiated and start a new exciton on a

randomly chosen dot. If an exciton is ever on a non-radiative dot (including when it is first added), we discard the exciton and we try a new one. After a large number of random excitations, we simply divide the number of radiated excitons to the number introduced and we have a good estimate of the luminescence quantum efficiency.

Using this method, we measured the luminescence quantum efficiency (LQE) of dot packings with each size distribution. We use a simple model where a fixed fraction of the dots are quenching sites. In a disperse medium, such as in solution, where no exciton transfer occurs, the LQE would equal to the number of non-quenching dots. In Figure 2-5 c), we show the dependence of the thin-film LQE on the solution LQE. We examined this effect on the dot-packing from our uniform-sized dot simulation. The thin-film LQE is strongly dependent on the Förster radius because the more likely transfer becomes the more likely the exciton will reach a quenching site. For small Förster radii, the thin film LQE is equal to that in solution and for large Förster radii nearly all excitons are quenched. When the Förster radius equals the mean dot radius, we observe an exponential increase in the thin-film LQE as the smaller number on quenching sites has an exponential effect on how many excitons are quenched. We then examined the case with the solution LQE equal to 2/3 for our different size distribution simulations. In Figure 2-5 d), we observe that the LQE actually drops, going from the uniform sample to 5% standard deviation and then increases monotonically. This trend reflects two competing effects. First, as the size distribution increase, some dots have smaller size but their Förster radii remain the same (by our assumptions). Since the Förster transfer rate goes as R^6, we expect significant increase of exciton transfer to and from these dots. The probability that excitons will move into the smaller dots more than outweighs lower probabilities that excitons will move out of large dots. This change causes the decrease in LQE because dots are likely to move to quenching sights. We also note that the shape of our clusters affects the LQE. The bottom line on the plot in Figure 2-5 d) gives the LQE found in a large hexagonally close-packed lattice, which is significantly lower than the uniform-sized sample because even though the packing is not significantly different, all the dots have six neighbors. On the other hand, the effect that improves the LQE

as we continue to larger size distributions is the decrease in packing. From the 5% sample on, the distances between dots increase both inside aggregates and as the dots form smaller clusters. The LQE is just one of the measurements that are strongly affected by quantum dot packing that we can probe using this simulation.

2.3.3 Model Properties

We found that one of the characteristics important to our model was the effective masses of the dots. The effective mass originates from the dot to surface interactions that affect the motion of the dot. These effects can be chemical interactions between the organic caps and the substrate molecules, electronic effects involving the dot core, and drag due to the liquid. The larger dots should have larger effective mass due to increased surface area, increasing both substrate and liquid interactions. The effective masses factor into the inelastic collision calculations. The transfer of momentum between objects with different masses is less efficient and so the energy lost during inelastic collisions is also smaller. We see in Figure 2-6 a) that if we give all the dots the same mass instead of scaling the mass with the surface area of the dots then the degradation in stability is much slower. We do still observe a loss of stability with size distribution because of the difficulty in forming a good packing with dots of different sizes. To illustrate that both these effects are important in our model, we created a sample with uniform sized dots but we doubled the mass of half of the dots. We found that this configuration actually improved the packing and stability of the dot monolayer (see Table 2.1). In Figure 2-6 b), the two dot clusters have less defects and are closer to circular because they are slightly less stable than the completely uniform dots. This instability allows more freedom to find low energy configurations while keeping the dots from breaking up completely. These two effects combine when looking a large size distributions. The differences in size cause the dots to be continually in motion as finding a perfect packing is unlikely and the differences in mass keep this extra motion from being reduced. These observations have implications for trying to create a monolayer of dots with different emmissive properties. To make a close-packed monolayer of dots using this method, this analysis

suggests that it would be better to change the material composing the dots rather than changing their size in order to get a number of emission frequencies. Alternatively, the organic cap material could be changed for different dot sizes to compensate for the differences in effective mass.

Table 2.1: Comparison of packing in uniform-size, uniform-mass dots and uniform-size bimodal-mass dots. In the bimodal-mass dots, half of the dots have twice the mass of the other dots. We show the proportion of interior dots and the mean number of neighbors for interior dots. The data shows that the bimodal dots actually pack better than the uniform dots.

	Uniform Mass	Bimodal Mass
Interior Proportion	0.697	0.747
Mean Interior Neighbors	5.737	5.857

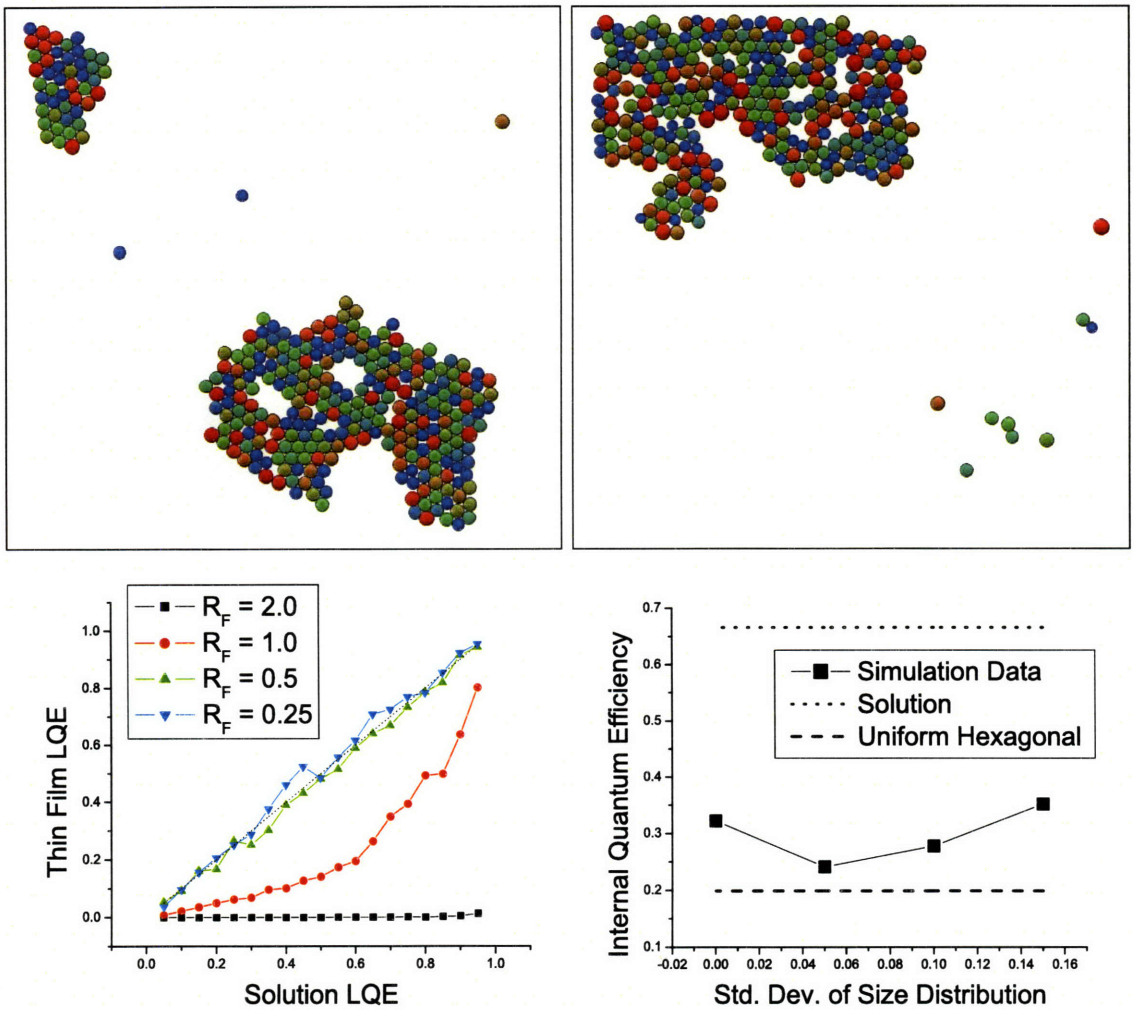


Figure 2-5: a) and b) show the simulation spatial results for 5%, and 10% size distributions. c) shows the dependence of the luminescence quantum efficiency (LQE) in thin-film on the solution LQE for the uniform-sized dot simulation with different Förster radii. The Förster radii are given in units of the mean dot radius. d) shows the dependence of the LQE on size distribution when 1/3 of the dots are designated traps, and the Förster radius equals the mean dot diameter. The lower dashed line shows the LQE (≈ 0.2) for a close-packed monolayer. The upper doted line shows the LQE for solution, which is 2/3, since we assume a dilute solution allows no energy transfer.

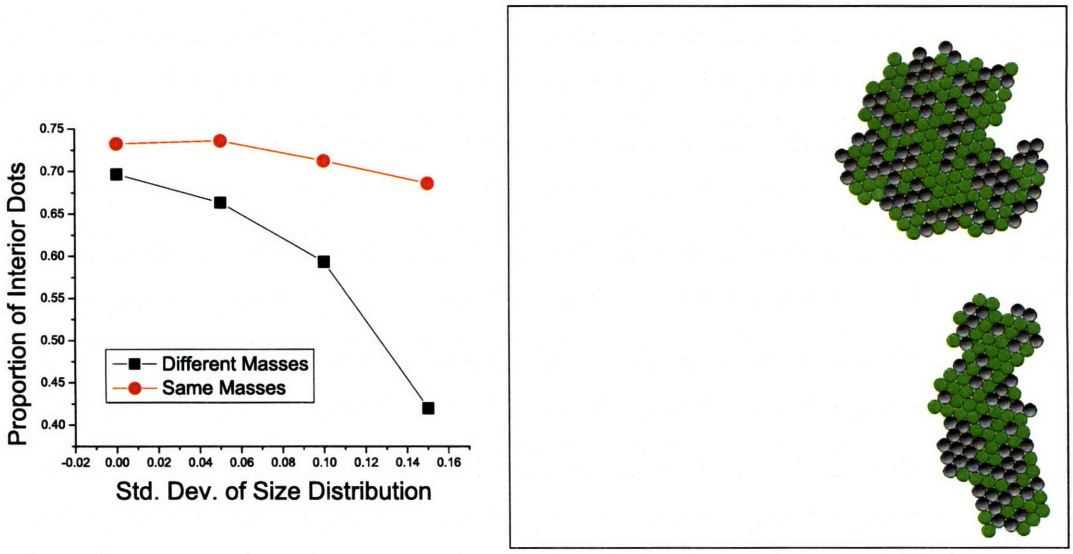


Figure 2-6: a) Compares the proportion of interior dots for equal dot masses to dot masses scaled with dot surface area. We find a much stronger degradation in monolayer stability when dot masses scale correctly with size. The difference in the first data point is due to random variations in the sample. b) Shows packing of uniform sized dots with half the dots having twice the mass of the other dots. Table 2.1 gives the packing calculations for this sample.

Chapter 3

Planarization Simulation

Our simulation of rough metal anodization is based on a discretization of the surface both in-plane and perpendicular to the surface. We create an evenly-spaced grid of points to represent an arbitrary surface topology. As the simulation progresses, each grid point can be raised or lowered only by a small step size. To simulate electrochemical oxidation, we create one surface describing the oxide and one describing the metal. Then we reduce one grid point on the metal and increase some number of oxide points so as to retain the same amount of metal, taking into account the expansion coefficient due to the incorporation of oxygen. To determine which points to modify, we find the metal point which is closest to the oxide and all its equidistant oxide points. Thus the oxide growth is conformal to the metal surface to within twice the step size used in changing the height of the metal. With this procedure we observe the same planarization of the metal surface observed in experiment [1].

3.1 Planarizing Anodization Theory

There exist few references in the literature on utilizing electrochemical oxidation (anodization) to planarize thin metal films [1] [26]. To our knowledge, there has yet to be a detailed theory of how this planarization occurs, and there does not exist a numerical simulation of the phenomenon. We believe that the reason for this oversight is that this type of planarization can be difficult to observe in many metals due to

defects that can arise during anodization and due to other process constraints. There are a large number of models and simulations for other types of oxidation, mostly for silicon [27]. The methods employed in these simulators range from the continuous approximations used in commercial process simulators such as SUPREM IV [28] to Monte Carlo bond-based simulators such as OXYSIM [29]. These challenges notwithstanding, the applications of this type of planarization are myriad. Reference [26] describes using this technique to fabricate high-quality nanodimensional inductors and capacitors. Using photoresist and etching, one of these planarized surfaces could create templates for nanoimprinting or microcontact printing. In addition, since the metal and oxide can vary in conductivity, novel fabrication techniques could be possible, for example, edge electroplating [1]. Due to the low roughness, layers of planarized metal and oxide can be deposited one on top of the other to form nearly perfect multilayer films [1]. These multilayer films could be used as exceptional x-ray defraction gratings.

Aluminum (Al) with alumina (AlO) and tantalum with Ta_2O_5 are the systems where anodization has been shown to planarize the oxide surface and the oxide-metal interface. The properties of these two metal-oxide pairs are significantly different. First, the volume expansion coefficient (the factor by which the oxide volume is larger than volume of metal consumed) for AlO from Al is low at approximately 1.3. For Ta, the expansion coefficient is much higher, between 2.3 and 2.47. This difference makes Ta much easier to planarize than Al because Al does not move as easily away from the initial metal surface. Another property that could be important for applications is tailoring of the dielectric constants of metals and their oxides. Alumina has a moderate dielectric constant $\epsilon = 9 - 11$, but Ta_2O_5 is known as a "high-k" dielectric $\epsilon \approx 25$ which makes it useful in capacitive and refractive devices such as MOS structures and the above mentioned multilayer film.

The general method followed to achieve the planarized surfaces through electrochemical oxidation is as follows [1]. DC sputter deposition is used to deposit a thin film of metal onto a flat substrate. The sample is placed in an electrolytic solution containing oxygen. The metal is used as an anode and a voltage is applied from a

platinum cathode. The electric potential drives the oxidization by inducing the diffusion of the charged oxygen species through the existing oxide to the metal surface. As the oxide grows, any area of the oxide that is thinner than another will have a larger voltage drop and a stronger electric field (V/cm). Then since the oxygen current is driven by the electric field, more oxygen will flow to the metal in this area. This process will happen quickly because the oxide is thin and the voltage is high, causing large current changes even for small defects. Consequently, oxide tends to grow at the same rate over the entire surface and self-heals any defects. This process creates a conformal coating of oxide with nearly the same thickness over the entire metal. This property is also the driving force behind the planarization of a rough starting metal surface.

To explain how planarization takes place, we examine the cases where a metal hill and a metal depression are anodized in two dimensions. Figure 3-1 compares the oxidation of a metal hill feature conformally with the consumption of an equal amount of metal at all points on the metal-oxide interface. In conformal oxidation, the distance between metal and oxide perpendicular to both surfaces must be equal at all points. Thus, any sharp edges become concentric circular sectors on both the metal and oxide surfaces. There is a larger area at the peak that needs to be filled by oxide. More metal will need to be oxidized where the surface has been bent away from itself than where the surface is flat. Since this type of roughness increases the rate at which metal is oxidized, the conformal oxidation drives the surface to lower roughness. If we examine the case of a depression in the metal, shown in Figure 3-1, we observe analogous results. At the bottom of the depression, we have a larger area of metal and a smaller area for the oxide to expand into. If the metal were oxidized at a constant rate, the oxide would become thickest above the bottom of the depression. To generate a conformal coating of oxide, little of the metal at the bottom is oxidized, and instead, the metal and oxide form the same concentric circles such that the perpendicular distance between the surfaces is equal at all points. Because less metal is oxidized where the metal is bent towards itself than where the metal is flat, the sides of the depression will eventually reach the level of the bottom, and the

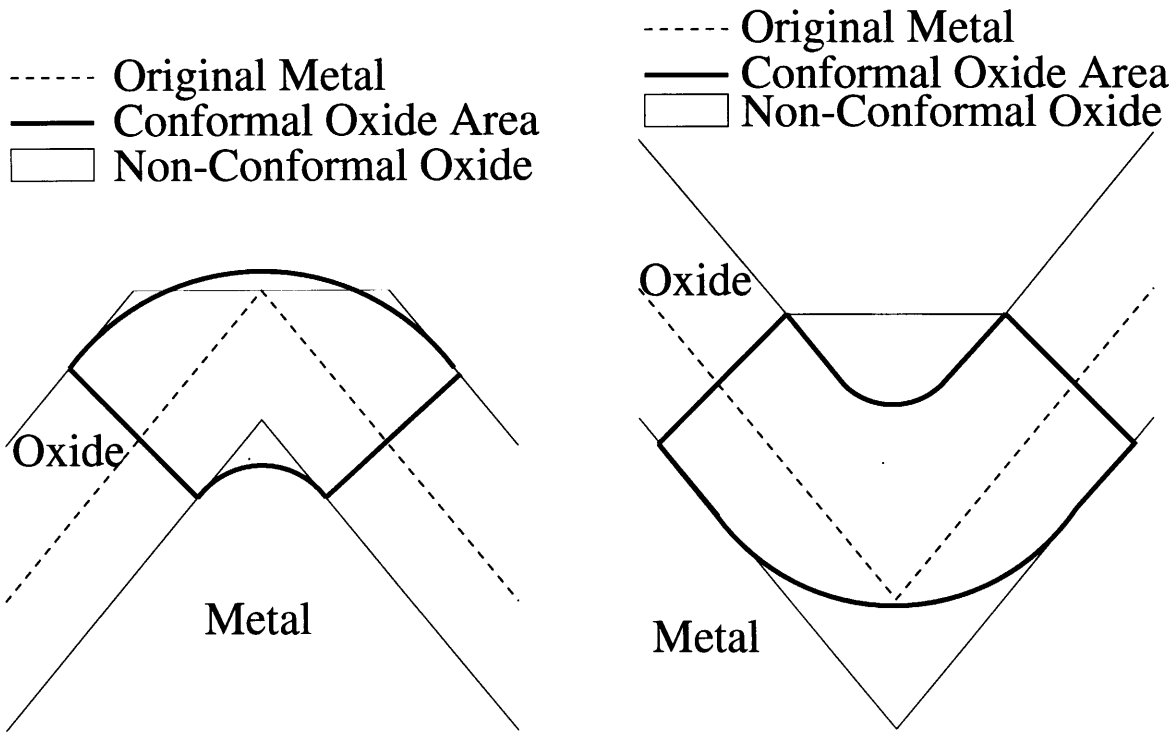


Figure 3-1: Figure comparing the conformal oxidation of a metal surface to an equal consumption of metal at the oxide growth interface. a) Shows a peak surface feature and b) shows a valley surface feature.

oxide will conform to that flat surface. Therefore, conformal oxidation decreases both types of roughness on a surface, and after a sufficient thickness of oxide is grown, the metal and oxide will both have low roughness.

Experiment shows some other interesting properties of the planarization. Figures 3-2 and 3-3 from references [1] and [2], respectively, show the progress of anodization experimentally at different oxide thicknesses. The researchers used atomic force microscopy to measure the oxide surface. They also created cross-sections of their samples using special coatings over the oxide to avoid damage and observed the metal and oxide surfaces using scanning electron microscopy. They used their AFM data to analyze the root mean squared and maximum to minimum values for different amounts of oxidation. A more detailed explanation of the importance of each of these measures can be found in Section 3.2.3. We can easily extract a number of key characteristics from this data. First, the roughness of the films increases with

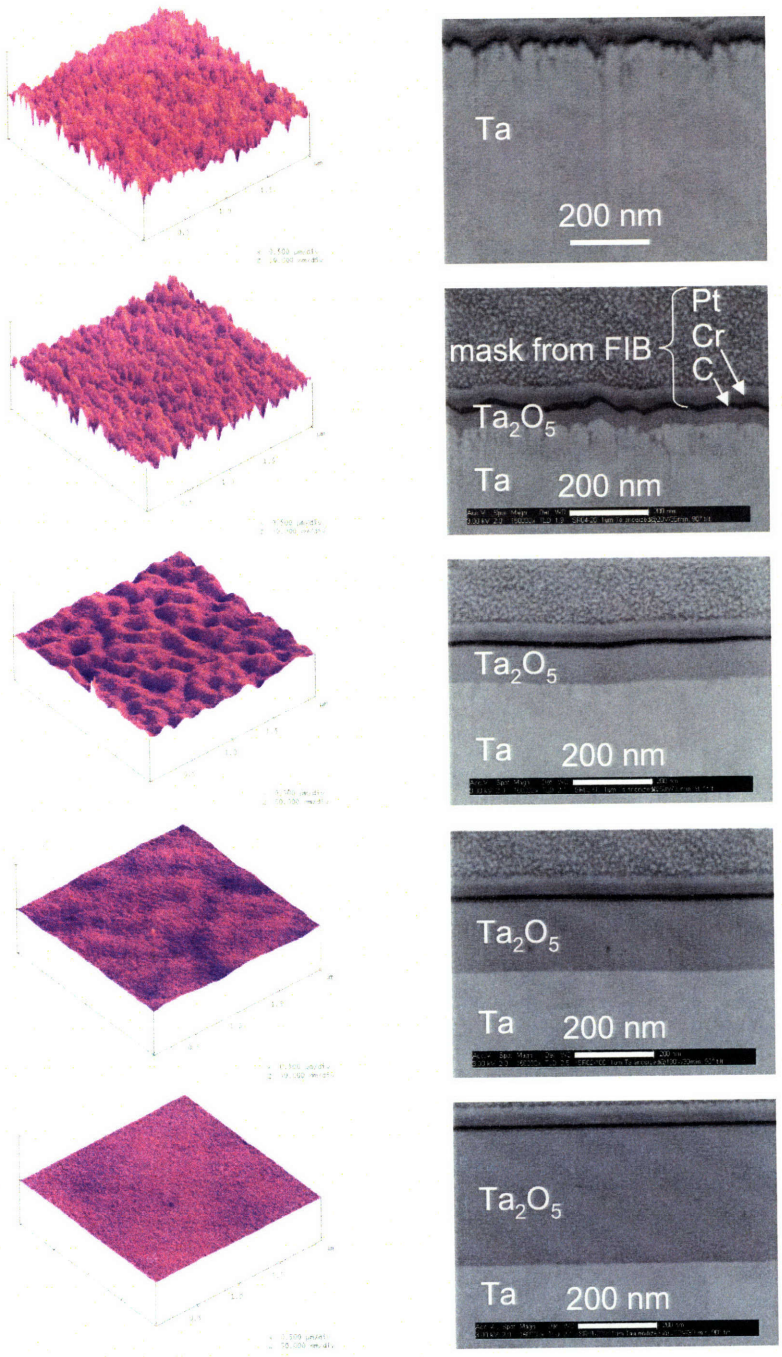


Figure 3-2: Experimental data from reference [1] detailing the progress of anodization on a $1000nm$ thick sample of Ta. Left column: AFM images of a $2\mu m$ by $2\mu m$ portion of the metal sample that clearly shows the progress towards planarization of the oxide surface as oxide thickness increases from $\approx 0 - 350nm$. Right column: SEM images of sample cross-sections for the same set of oxide thicknesses. Shows similar planarization in the metal surface as observed on the top oxide surface. Reprinted with permission from *Nano Letters* **2005**, 5 (10), 1899-1904. Copyright 2005 American Chemical Society.

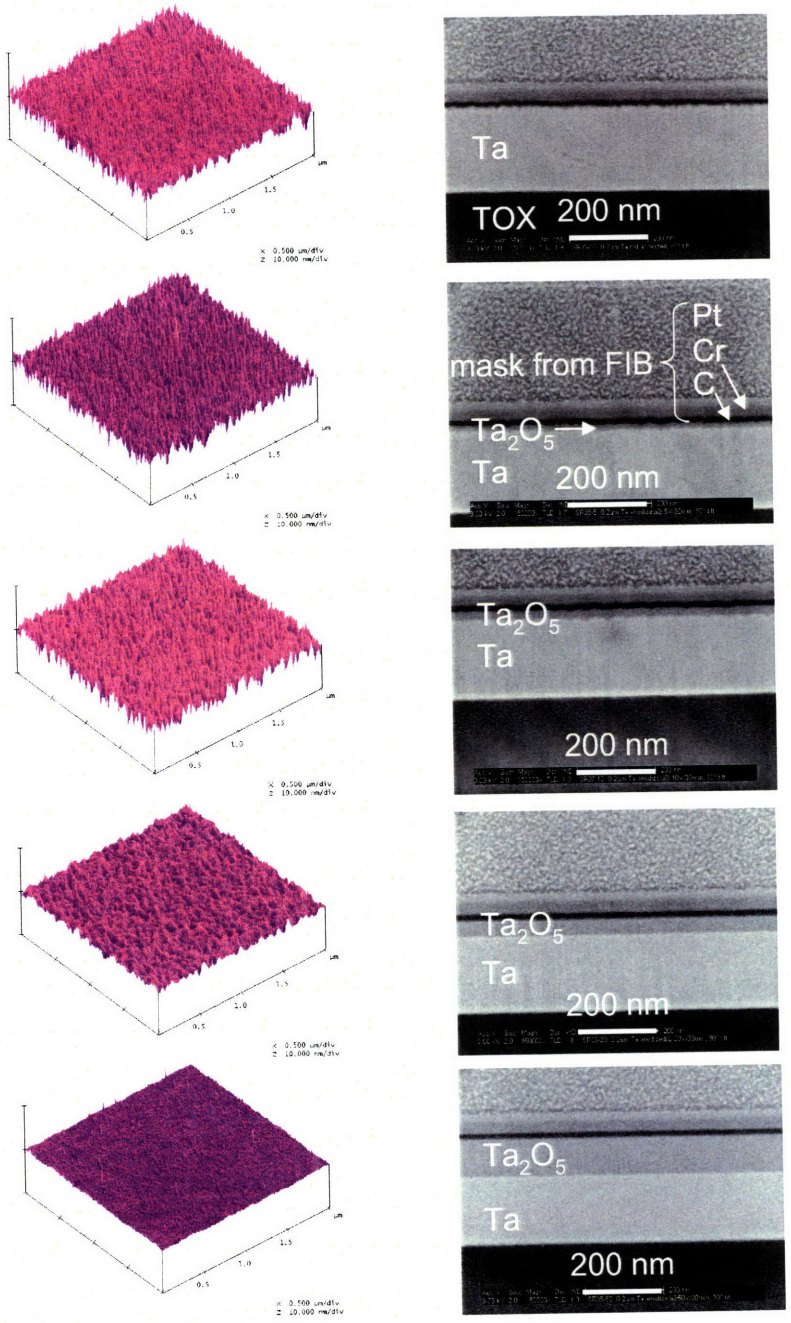


Figure 3-3: Experimental data from reference [2] detailing the progress of anodization on a $200nm$ thick sample of Ta. The dimensions of the roughness are much smaller than that observed for the $1000nm$ sample. Left column: AFM images of a $2\mu m$ by $2\mu m$ portion of the metal sample that clearly shows the progress towards planarization of the oxide surface as oxide thickness increases from $\approx 0 - 100nm$. Right column: SEM images of sample cross-sections for the same set of oxide thicknesses. Shows similar planarization in the metal surface as observed on the top oxide surface.

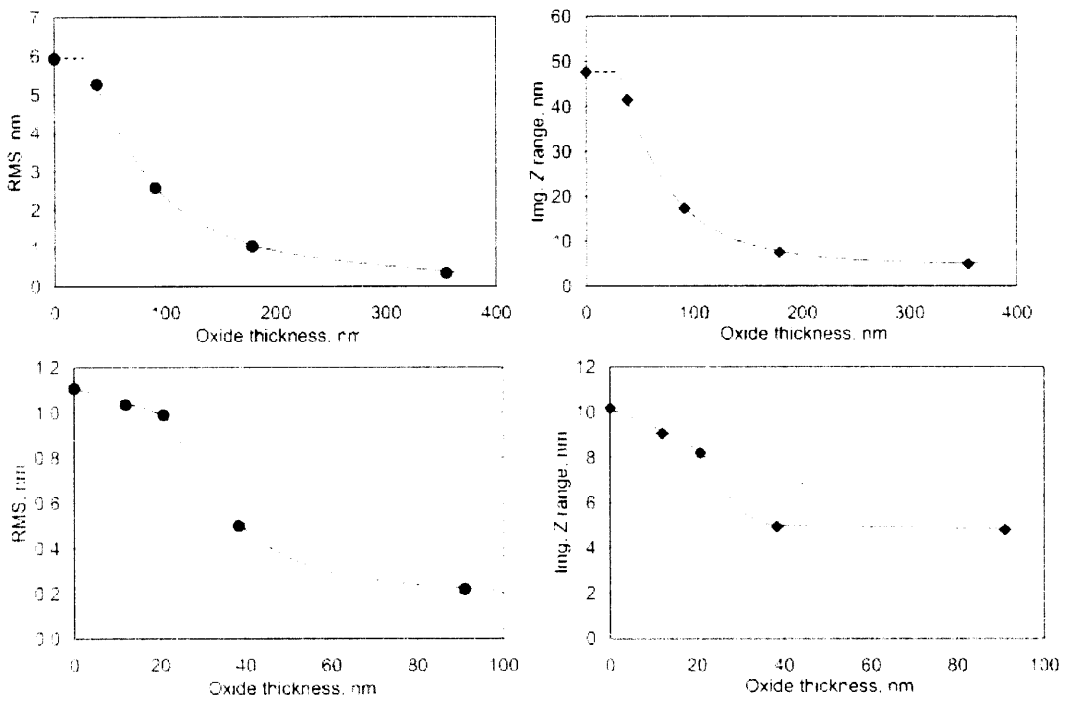


Figure 3-4: Graphs of RMS and z-range (maximum to minimum) for the oxide AFM data shown in Figures3-2 and 3-3 on the top and bottom row, respectively. The top row shows the higher roughness occuring in the $1000nm$ sample compared to the $200nm$ sample on the bottom row. Both sets of data show the clear planarization of the oxide surface.

the amount of Ta deposited. Second, the $1000nm$ thick sample contains roughness on two different scales, one larger around $200nm$ and another smaller around $30nm$. The thinner sample contains only the small-sized roughness. The contention is that the larger features result from formation of crystallite islands during deposition, and that a certain thickness of metal is necessary before the surface chemistry becomes favorable for their formation. The smaller roughness is likely a product of the sputtered deposition process, appearing unchanged at all thicknesses. Next, we should note the particular trends in roughness during the process of anodization. For a sufficiently thick oxide, both the RMS and the z-range show an exponential decrease with the amount of oxide grown. The roughness approaches an asymptote determined by the roughness of the substrate. For the lower roughness film, the experiments were able to resolve a region where the decrease in roughness is linear instead of exponen-

tial. We will discuss this effect in greater detail in Section 3.4, but in general, this results from the fact that, until the thickness of the oxide reaches some percentage of a particular feature size, those features are hard to planarize. This characteristic can also be observed in the AFM images for $1000nm$ thick Ta at oxide thickness 3 in Figure 3-2. At that thickness of oxide ($\approx 100nm$), only the small-sized roughness has been planarized, leaving the large-sized roughness nearly unchanged. These are some of the aspects of this type of anodization that we hope to observe in our simulated process model. Additionally, we hope to observe other properties that are difficult to measure by experiment. In particular, observing the metal-oxide interface is troublesome in Ta because the etches that select for Ta_2O_5 also tend to etch some fraction of the Ta. We would also like to predict how different oxide expansion coefficients affect the planarization.

3.2 Description of Key Calculations

In implementing this simulation, a few key calculations must be made correctly. First, we must determine how to change the positions of the metal and oxide surfaces so that we preserve the total mass of metal. We find that the volume change in our discretized surface is linearly proportional to the amount each point in the grid is moved up or down. Then we need to calculate the minimum distance between the two surfaces. We considered the distances between any pair of points in the metal and oxide surfaces, as well as the distance between a point in one surface and a line connecting two points in the other. Finally, we need some numerical measures of the progress made toward planarization of our sample. Our figures of merit will be the z-axis range and RMS.

3.2.1 Volume Change in Piecewise Linear Surface

In our simulation, we create a grid of points in the x-y plane with equal separation in each direction and with arbitrary z values. Between each grid point the z-axis surface values are linearly interpolated (Figure 3-5). Thus if we only have two grid

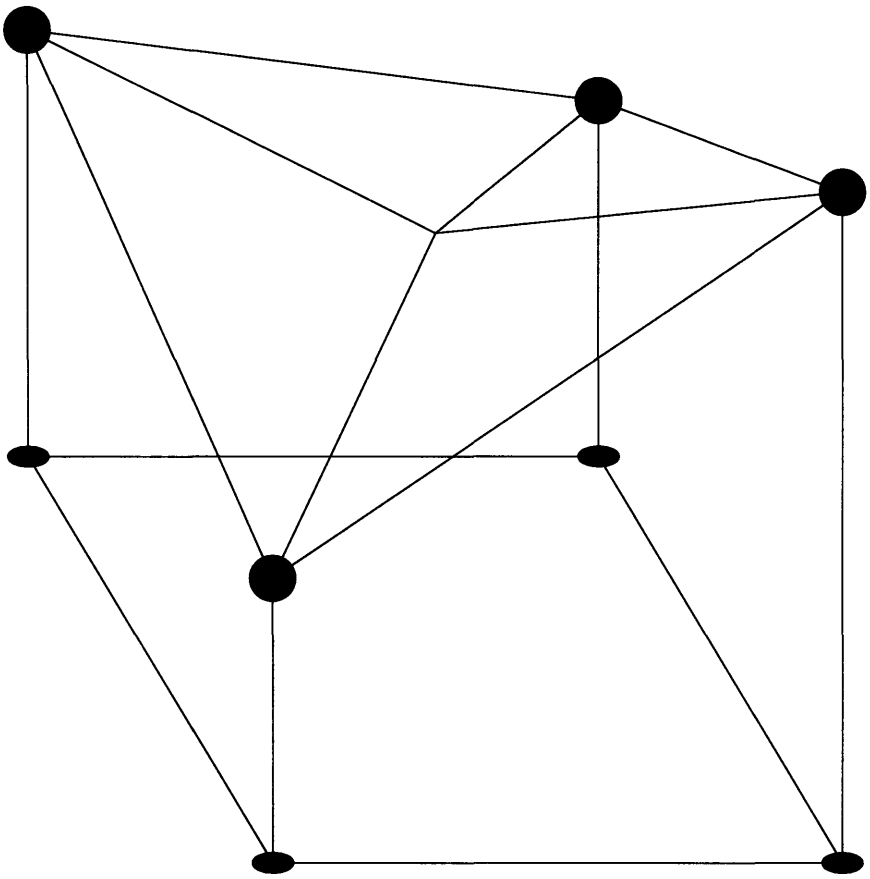


Figure 3-5: Depiction of a piece of our surface grid having equally spaced x and y coordinates and floating-point precision values in z. The figure also shows how the grid is linearly interpolated to give an analytically continuous surface for calculations.

points at $(x, y, z) = (0, 0, 0.5)$ and $(0, 1, 1)$, evaluating the height of the surface at $(x, y) = (0, 0.5)$ gives $z = 0.75$. Likewise, at the center of each square of points, the z value is the average of the four surrounding points. Between this center point and two points on the corners of the square, we define the surface as the section of the plane that passes through those three points. Thus our surface consists of tiled triangular facets defined by points of the grid with 4 facets for each point. This parameterization is as accurate as possible for an arbitrary surface with a set number of data points.

To consider how the amount of metal changes with the adjustment of a single grid point, we first calculate the volume contained under a single facet (Figure 3-6). We split the volume into two sections: a lower triangular prism and an upper pyramid. The prism extends from the x-y plane to the minimum z value (z_m) where the original

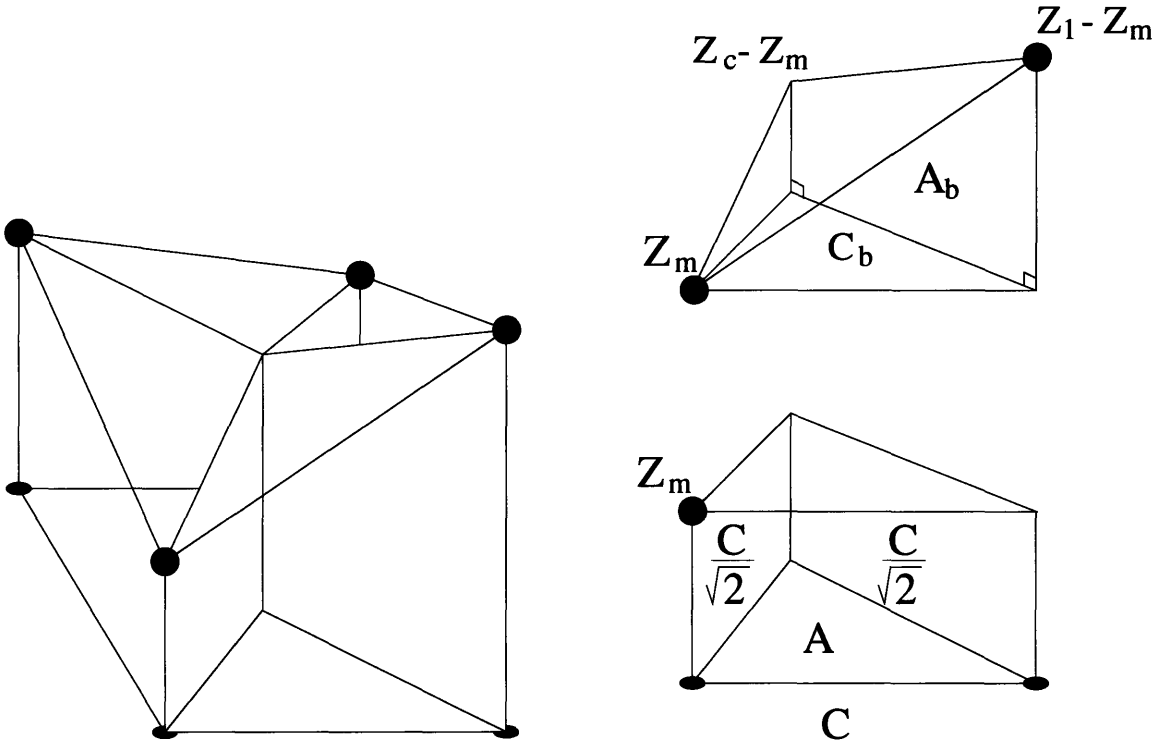


Figure 3-6: a) The volume under of a single triangular face. b) For calculation we split the volume into a triangular prism and a pyramid. The z values of the vertices are labeled by z's, side lengths are labeled by c's and areas are shaded and labeled by A.

volume has been cut. The pyramid is the rest of the volume, with the pyramid base opposite z_m. In the figure, we show z_m as one of the grid points, but it could equally well be the center point. The volume of the prism is calculated by Equation 3.1.

$$V_p = Ah = \frac{1}{4}c^2 z_m \tag{3.1}$$

For the pyramid, the base may either be a right trapezoid or a triangle in the limiting

case where $z_c = z_m$ or $z_1 = z_m$. In either case, the volume of the pyramid is given by:

$$\begin{aligned} V_{pyr} &= \frac{1}{3}A_b h = \frac{1}{6}c_b\left[(z_c - z_m) + (z_1 - z_m)\right]h \\ &= \frac{1}{12}c^2\left[(z_c + z_1) - 2z_m\right] \end{aligned}$$

where

$$\begin{aligned} c_b &= \begin{cases} c & \text{if } z_m \text{ is at the center point,} \\ \frac{c}{\sqrt{2}} & \text{otherwise.} \end{cases} \\ A_b &= \frac{1}{2}c_b\left[(z_c - z_m) + (z_1 - z_c)\right] \\ h &= \begin{cases} \frac{c}{2} & \text{if } z_m \text{ is at the center point,} \\ \frac{c}{\sqrt{2}} & \text{otherwise.} \end{cases} \end{aligned} \tag{3.2}$$

The volume of the combined solid is shown in Equation 3.3. We show that each vertex contributes linearly and equally to the volume. We must also make sure that the center point does not somehow affect the contribution to the volume under the overall surface. Equation 3.4 consists of all four volumes from Figure 3-5 added together and shows that all points on the surface contribute linearly and independently to the total volume. In fact, the volume under the entire surface as we have defined it is simply the area of the grid times the mean of all z values. Thus we have proved that when we move one point on the surface down by a certain amount, and then we move a different point on the surface up by the same amount, we retain the same volume under the surface.

$$V_t = V_p + V_{pyr} = \frac{1}{4}c^2 z_m + \frac{1}{12}c^2\left[(z_c + z_1) - 2z_m\right] = \frac{1}{12}c^2\left[z_c + z_1 + z_m\right] \tag{3.3}$$

$$\begin{aligned} z_c &= \frac{1}{4}(z_1 + z_2 + z_3 + z_4) \\ V &= V_1 + V_2 + V_3 + V_4 = \frac{1}{12}c^2\left[2z_1 + 2z_2 + 2z_3 + 2z_4 + 4z_c\right] \\ &= \frac{1}{12}c^2\left[3z_1 + 3z_2 + 3z_3 + 3z_4\right] = c^2\sum_{n=0}^{4}\frac{z_n}{4} \end{aligned} \tag{3.4}$$

3.2.2 Distance Measurements

Our simulation of anodization planarization depends heavily on measuring the distances between the metal surface and the oxide surface. These calculations also take up the bulk of the computation time, forcing us to strike a balance between accuracy and computational complexity. We also must remember that our discretization of the surface reduces the final accuracy we want to achieve, so working harder on some calculations will not noticeably improve the outcome. There are a number of distances we can consider between our metal and oxide surfaces, namely: point to point, point to line, line to line, point to plane, line to plane, and plane to plane. The line to line and plane to plane measurements would be redundant. For line to line, the grid precludes any line segment from being any closer to another line than its end points. For plane to plane, in three dimensions each plane piece can only be as close to another plane as one of its sides or one of its vertices. We have chosen to omit point to plane and line to plane calculations for our current simulation. The point to point and point to line calculations will be correct to first-order because the omitted calculations can only serve to slightly shorten the distance found between metal and oxide. The point to line measurement adds a good deal of accuracy, which can particularly be seen when the oxide thickness is small. With only point to point distances, only the oxide and metal points at the same x-y coordinates can possibly be closest to each other until the oxide height reaches the distance between points. When we additionally consider the point to line measure, the oxide or metal point will most likely be closer to the line connecting the point directly above or below and an adjacent point. This change means that we have to move more than two points to increase the metal-oxide distance correctly, and this contributes greatly to how much we expect rough spots to spread.

The equations themselves are simple Euclidean geometry. Figure 3-7 depicts the three situations for which we account. The point to point distance is given in Equation 3.5, and the oxide point to metal line distance is given in Equation 3.6. The

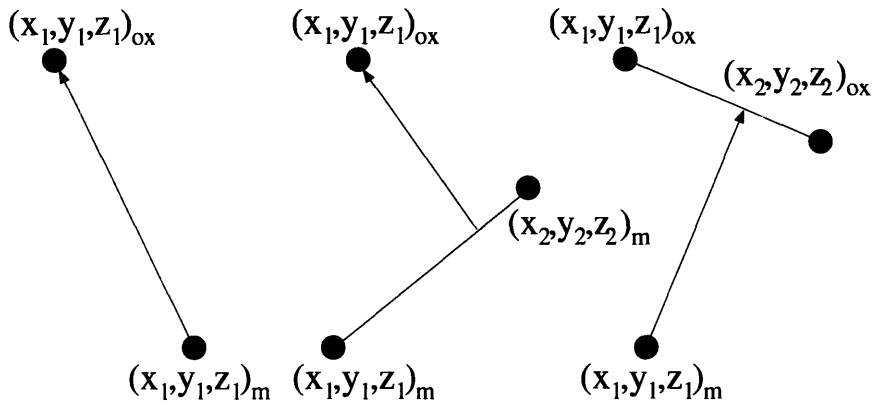


Figure 3-7: The figure shows the three distances (represented by arrows) calculated in our simulation. In point to line distances, the distance vectors meet the lines at right angles.

metal point to oxide line case is analogous.

$$d_{p-p} = ||\vec{z_{ox}} - \vec{z_m}|| \tag{3.5}$$

$$d_{p-l} = \frac{||(\vec{z_{m2}} - \vec{z_{m1}}) \times (\vec{z_{m1}} - \vec{z_{ox}})||}{||\vec{z_{m2}} - \vec{z_{m1}}||} \tag{3.6}$$

3.2.3 Surface Roughness

The surface roughness has to do with both local variations in the height of the surface as well as the long range slope of our material. The simplest way to assess overall variations in the surface is to use the root mean squared (RMS) of the z values deviation from the mean (Equation 3.8). This measure reflects discrepancies between all heights, whether they are close together or far apart. A small RMS means we have a mostly flat surface. Small defects in our surface, however, could have a large impact for many applications, and these defects could be masked by a simple RMS. The total z-range of the surface shows how the outlying points in our sample approach each

other. This measure gives us an idea of the worst-case roughness that we can expect.

$$\text{z-range} = \max_{n\in\{1\ldots N\}} z_n - \min_{n\in\{1\ldots N\}} z_n \tag{3.7}$$

$$\text{RMS} = \sqrt{\sum_{n=1}^{N} \frac{(z-\mu_z)^2}{N}} \tag{3.8}$$

3.3 Simulation Algorithm

We used the Matlab numerical analysis platform as the basis for our current implementation of this simulation. To begin, our simulation is given an arbitrary surface described by a grid of x-y points with equal spacing along both directions and a set of z-axis values. Then, depending upon the accuracy we are looking to achieve, we increase the number of points on the grid in both directions by some integer and linearly interpolate the z values from the original set. This data describes the metal surface. We assume that the oxide starts with minimal thickness such that the oxide surface is a simple copy of the metal surface data. Also important for accuracy is the step size by which we will increase the oxide surface and decrease the metal surface.

The three main data structures at the heart of our simulation are a set of oxide points, a set of metal points, and a set of distances. The set of oxide points is a matrix of variable-length arrays containing the indices of the oxide point or points that are the minimum distance away from the metal point at that index in the matrix. This matrix has the same indices as the x, y, and z data sets, which store the grid point positions. The set of metal points is a convenience structure that duplicates the oxide points data but indexing instead on the oxide point to which several metal points may be closest. This metal point set does not necessarily contain only the metal point or points equally close to a particular oxide point, but instead has any metal point that does not have any other oxide points that are closer. The set of distances is a matrix giving the minimum distance between a particular metal point and the closest oxide point and is indexed by the metal point. In other words, these are the distances associated with the set of metal to oxide points. The oxide points

set and metal points set are initialized to their own indices, and the distances are all zero.

After initializing the surface and the various data structures, our simulation proceeds through the following steps:

1. If smallest distance is greater than the desired final oxide thickness, end the simulation.

2. Find minimum distance in the set of distances. This is the index of the metal point to change.

3. Save the indices of the oxide points for that metal point.

4. Increment the z values of these oxide points by the expansion coefficient minus one times a fraction of the oxide step.

5. Decrement the z value of the metal point by the sum of all the increment fractions.

6. Recalculate the distance to the oxide for the updated metal point.

7. Recalculate the distance to the oxide for all the metal points that were previously closest to the updated oxide points.

8. Determine if the distance from the updated oxide points to any metal point is smaller than the metal point's current minimum distance.

9. Go to step 1.

Most of these steps are straightforward, but a few require further explanation. The termination condition for our simulation is fulfilled when the smallest distance between metal and oxide is greater than the goal thickness. We could alternatively do some kind of smoothing of our distances so that all distances get very close to the correct oxide thickness by reducing the oxide step size as we approach our goal. We do not believe this procedure would add much to our accuracy, however, because the step size throughout the simulation is a limit on our accuracy. Reducing it near completion

would do little for overall accuracy. In addition, if we wanted to compare the final oxide to intermediate oxides we would want to use the same procedure before taking those measurements. That would add to our computational time and make separate runs different based on the number of measurements because measuring would reduce the oxide step size temporarily.

Next we should specify which fraction we are using to increment each of the oxide points. We use the fraction that will ensure that over the entire surface at one point in time no metal point will be reduced by more than one oxide step and no oxide point will be increased by more than the expansion coefficient minus one times the oxide step. Equation 3.9 shows the calculation of that fraction.

$$F_{ox} = \frac{1}{(\text{length of oxide point set}) \times (\text{length of metal point set for this oxide point})} \tag{3.9}$$

The first term ensures that if this metal point is equidistant to multiple oxide points then the total of the oxide fractions will be less than or equal to one. The second term ensures that, if, for instance, two metal points have the same oxide point as their closest oxide point, then each point will only contribute half of an oxide step to that oxide point.

Once we have changed the metal point and the set of oxide points, any distances that refer to those points are out of date. We must therefore recalculate any distances that may have been affected by these changes. For the metal point, we have to find the closest oxide point or points, and as stated in Section 3.2.2, we consider the metal point to oxide point distances, the metal point to oxide line distances, and the metal line to oxide distances. For the metal line to oxide point calculation, we consider the four line segments with the changed point as a vertex (two along x and two along y) in this calculation. To optimize our algorithm, we start by looking at oxide points directly above the metal point and move outwards only as far in x and y directions as the minimum distance to the oxide found so far. This cutoff is a simple application of the triangle inequality because the distance between a metal point and an oxide point can at minimum be the x-y distance between the points. We then look at the

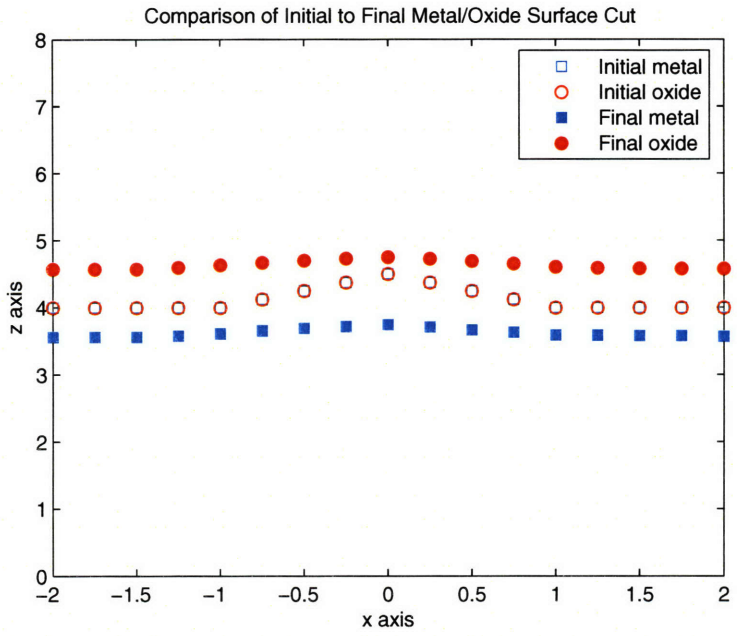
metal point lists for each of the oxide points that we changed. These lists give metal points which used to be closest to the updated oxide points, and we follow the same procedure with these metal points as with the updated metal point to correct their minimum distances. Finally, we must find any other metal points, which, after the update of the oxide, are now closest to one of the changed oxide points. Thus we follow analogous procedures, searching around each oxide point for any metal points that could be close to it. In this case, the cutoff in the number of metal points that we must consider surrounding each oxide point is slightly different. Since the minimum distances are different at each metal point, we do not have a straightforward minimum distance from the oxide point to all of the metal points. Thus we need a different measure to see how far the oxide point could possibly be from the metal surface. We remember, however, that we are conformally coating the metal surface by only incrementing the smallest distance by a small value. Thus the maximum distance we can expect to find is the current oxide thickness plus the expansion coefficient times the oxide step, since we change the metal to oxide distances by less than that amount each time. Then we only have to consider the metal points around each oxide point that are less than this distance in the x and y directions. After we have completed updating the distances, we are ready to return to the beginning of our loop to test for termination.

3.4 Planarization Results

We started testing the simulation by considering the progress of anodization on a single hill and a single depression. To create the hill we started with a 5x5 grid with the distance between grid points equal to $1nm$ and initial metal thickness of $4nm$. We then incremented the center point to $4.5nm$ and increased the sampling of the surface by four times so that the distance between grid points equaled $0.25nm$. After the interpolation, we are left with a pyramid with a base width of $2nm$ or 8 grid points. Having created the structure, we proceeded to oxidize the surface up to $1nm$ using $0.01nm$ oxide steps. Figure 3-8 compares the initial and final metal and oxide

surfaces for both the hill and the depression created by making the original center point $3.5nm$ instead of $4.5nm$. First of all, we observe significant planarization of both types of surface feature in both the oxide and metal surfaces. We also observe spreading of the features over adjacent grid points in both surfaces as we would expect to allow for planarization. The total amount of planarization is the same for the hill and the depression as can be seen in Figure 3-9.

In Figure 3-9, we observe a number of interesting phenomena that are observed experimentally (see Section 3.1). First, the metal and oxide surfaces do not planarize at exactly the same rate. For the hill feature, the metal surface RMS and z-range drop faster than the oxide RMS and z-range. Whereas for the depression feature, this trend is reversed. As the oxidation progresses, this discrepancy is reduced until both surfaces are planarized the same amount within simulation accuracy. The root of the discrepancy is a linear region in the calculations at the beginning of the planarization that appears in both RMS and z-range. The linear region is followed by an exponential region. The linear region is stronger (flatter slope) in the oxide for the hill and is stronger in the metal for the depression. Until the oxide thickness gets close to the defect height some of the metal and oxide points that should planarize each other do not have a straight line distance through the oxide. Thus planarization is slower (linear) until each metal point can see most oxide points it needs to oxidize or until each oxide point can see most metal points that need to oxide it. Now we can understand the discrepancy between the hill and valley oxidation. In the hill, the metal is closer to many oxide points at the peak of the feature, which is where the most planarization is needed. Thus the metal planarization proceeds more quickly at the beginning for the hill. The oxide catches up since it has a wider view at the peak for thicker oxides and since the rate of planarization is higher for rougher surfaces. In the depression case, the roles of metal and oxide are reversed with the oxide initially having many points contributing to it near the lowest point and the metal having less points to contribute to near the lowest point. Finally, if we were to examine the case where we have a random assortment of hills and depressions, we would observe the same, linear followed by exponential, characteristics. On the other



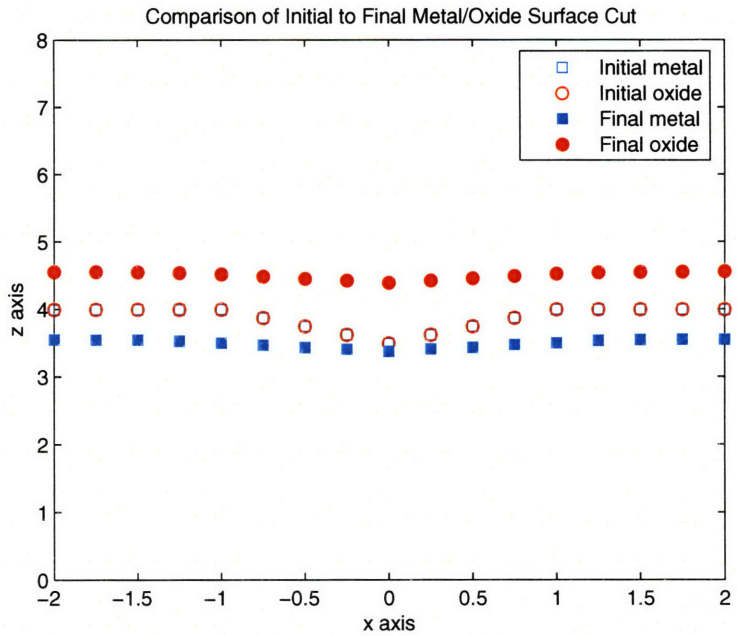


Figure 3-8: a) Shows the effects of oxidation on a hill with height $0.5nm$ after a $1.0nm$ oxide has been electrochemically grown. b) Shows the effects of oxidation on a depression with depth $0.5nm$ after a $1.0nm$ oxide has been electrochemically grown. In both figures, we have taken a slice through the center of the 3-dimensional surface. The empty symbols represent the original surface, and the solid symbols represent the final surface. The squares are for metal and the circles are for oxide.

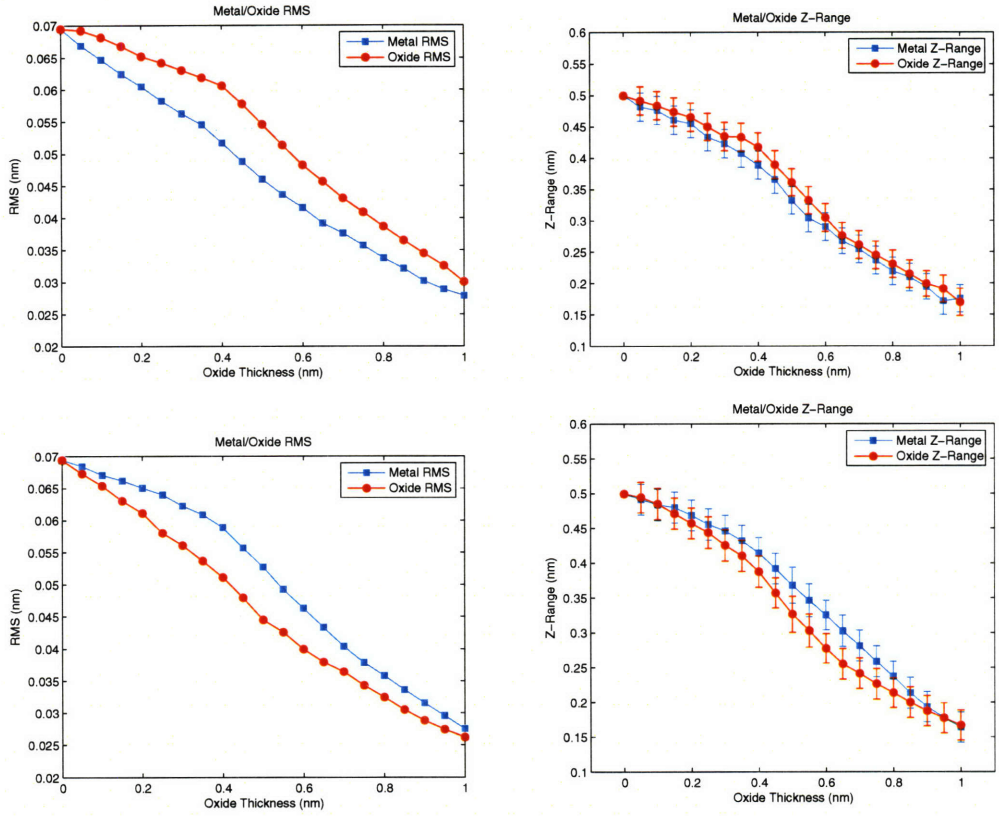


Figure 3-9: a) Shows the RMS values of both the metal and oxide as a hill feature is anodized. b) Shows the z-range (maximum to minimum z value) of both the metal and oxide as a hill feature is anodized. The error bars show the standard deviation in the metal to oxide distances. c) Shows the RMS values as in a) but for a depression instead of a hill. d) Shows the z-range as in b) but for a depression instead of a hill. The error bars show the standard deviation in the metal to oxide distances.

hand, because we would have an approximately equal number of hills and depressions, the metal and oxide progress would be nearly identical (see Figure 3-12). Although these characteristics can be theorized without the help of numerical methods, it is encouraging that our simulation exhibits the desired behavior.

We also must be concerned with the accuracy of the simulations, based on our choice of discretization. The two parameters that affect our accuracy (as well as our computation time) are the distance between grid points and the size of the steps made between the oxide and the metal. Figure 3-10 shows data for progressively smaller grid spacing and smaller step size. The progression shows that in order to achieve relatively consistent results that we must use a grid spacing that is four times

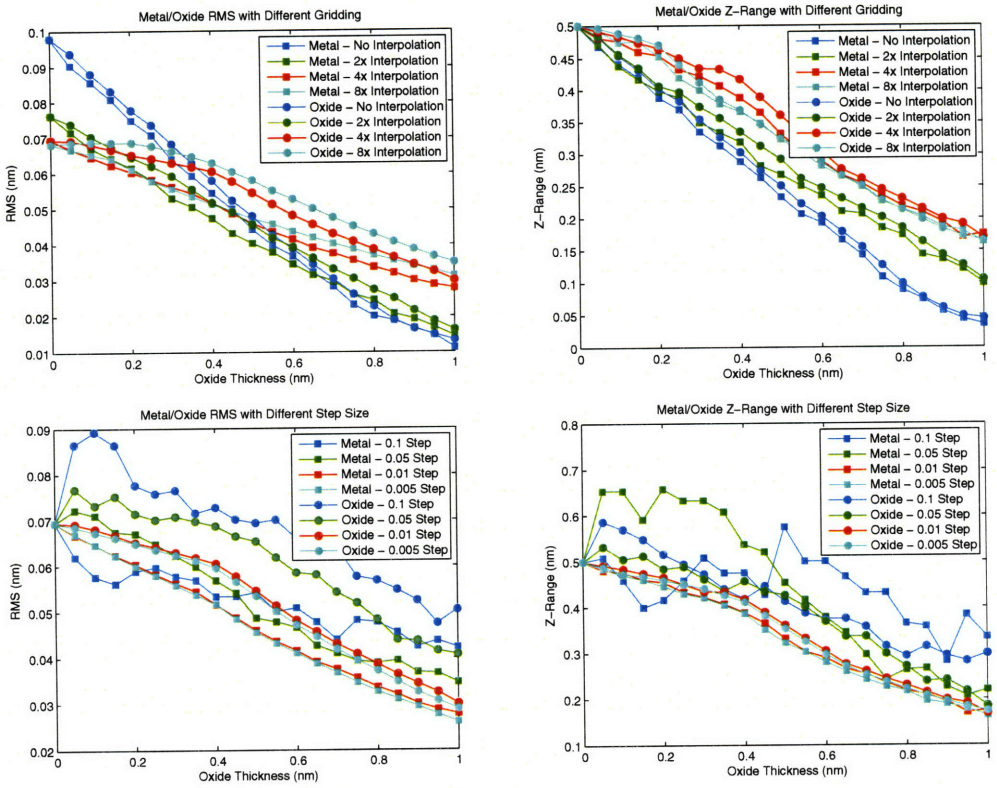


Figure 3-10: This figure demonstrates the convergence of the simulation as we decrease grid spacing and step size. a) Shows the RMS of the metal and oxide surfaces for increasingly fine grids and a step size of 0.01. We start with a 5 by 5 grid and halve the spacing between grid points for the next simulation. Thus for the first simulation the hill feature is a single grid point, for the second simulation the hill is 3 grid points, then 7 points, and 15 points. b) Shows the z-range for the same set of grids c) Shows the RMS of the metal and oxide surfaces for decreasing step sizes and a 17 by 17 grid corresponding to 7 points for the hill. We start with a step size of 0.1, which is only 1/5 the height of the hill, and go to 0.005, which is 1/100 of the hill height. d) The z-range for the same set of step sizes.

smaller than our smallest surface feature (4 times interpolation) and a step size that is approximately 1/50 of the height of our largest feature (0.01 steps for a 0.5 feature size). This data verifies that the increase in accuracy beyond these values for RMS and step size is negligible (keeping other variable constant such as the expansion coefficient) and the increase in computation time is substantial. That is why we used 4 times interpolation and 0.01 step size in our examples up to this point. Now we can continue to more complicated surfaces and have a good idea that our results would be approximately the same if we increased our accuracy.

For our simulation, we have recreated a $0.5\mu m$ by $0.5\mu m$ section of a sputtered tantalum (Ta) surface with $1000nm$ thickness. As explained in Section 3.1, for this thickness of Ta, two scales of roughness were observed, wider and slightly higher crystallite roughness and a smaller roughness due to non-uniform deposition. The larger roughness had widths of approximately $200nm$ and heights of about $30nm$ and the smaller roughness has widths of $40nm$ and heights of $20nm$. To recreate this type of roughness we used the following procedure:

1. Create a grid with spacing equal to half the width of your largest defect size ($100nm$ spacing).
2. Set all z values to the average initial metal thickness ($1000nm$).
3. Add a random number with constant distribution with zero mean and range equal to your largest defect height (random numbers from $-15nm$ to $15nm$).
4. Decrease the grid spacing until it reaches half the width of the next defect size ($20nm$ spacing).
5. Add the same type of random numbers with the new defect height (from $-10nm$ to $10nm$).
6. Continue until all defects are added.
7. After adding the last defects, decrease the grid spacing by four times to ensure accuracy.

We use a grid spacing of half the desired defect width because that allows for the random numbers to be high and then low on the correct scale. These values are then interpolated to get a smooth surface. This method does not best reproduce the experimental surface because the large crystallite structures should not be entirely random. They should instead be distinct islands pushing into each other such that the angle at the boundary between the islands is relatively sharp. The smaller defects from deposition are closer to this random configuration, having a set maximum and minimum possible addition with all heights between those two values equally likely.

We see in Figure 3-11 that we have RMS and z-range values that begin close to the experimentally observed values. Since we have this agreement in the original surfaces, we hope to see moderate agreement between our simulations and experiment.

Figure 3-11 shows our results for the simulated Ta surface described above. We observe clear planarization of the surface as the oxide thickness increases. As in experiment, the small, deposition-type roughness planarizes first. The RMS and z-range are compared to their experimental values in Figure 3-12. These measurements show reasonable agreement. We note that although the initial z-range values match well, the initial RMS is $2nm$ higher for our simulation. This fact hints that our description of the oxide surface needs to be improved to better represent the larger crystallite roughness. Our model demonstrates the expected exponential character of the planarization. As expected, the oxide and metal surfaces planarize at near exactly the same rate. In Section 3.1, we described the difficulty in experimentally measuring the metal surface when it is covered by oxide, with our model we predict that the metal planarization will correspond directly to that of the oxide. Our simulation results contain two discrepancies with the experimental data, however. First, the exponential decrease proceeds slower in our simulation. We attribute this error to the finite boundary of our simulated sample. The oxide and metal points cut off at the edges making less points to oxidize and making the planarization proceed slower. We can alleviate this error by simulating larger samples. Second, our model does not show the strong linear region at the beginning of oxidization. Two possible sources of this error are the differences in initial surfaces discussed previously and the discretization of the initial surface. The initial surface morphology affects the linear region because of differences in the oxide distances needed to get efficient planarization. Our linear interpolation of initial features could also cause this disagreement. Linear interpolation creates a jagged initial surface. Since the rate of planarization is directly proportional to roughness, the interpolation may be artificially increasing the initial planarization and suppressing the expected linear region. We can use quadratic or cubic interpolation to eliminate this effect. Overall, we conclude that our model does a good job of reproducing important experimental results.

We also studied the effects that different expansion coefficients (k_{exp}) have on the planarization process. This data can help us anticipate the planarization of different metals, which will all have different k_{exp}. We expect that for smaller k_{exp} the oxide surface will be more difficult to planarize because the surface will move up less and the metal surface will move down more. We noted some difficulty in our calculations for low k_{exp} due to the fact that the oxide surface moves only slightly for each step. We had to increase the number of grid points to get an accurate measurement. In fact, for $k_{exp} = 1.8$, we had to increase the gridding by 32 times instead of the 4 times for $k_{exp} = 2.3$. We hope in future versions we can improve this factor by slight improvements to our algorithm. Figure 3-13 shows the RMS and z-range for $k_{exp} = 1.8$ and $k_{exp} = 2.8$ on a hill surface feature. The most obvious difference is that the smaller expansion coefficient increases the disparity of planarization between the oxide and metal surfaces. For low k_{exp} the oxide surface moves less and we need to oxidize more metal to get the desired oxide thickness. This observation explains why the difference between oxide and metal RMS should be larger for lower k_{exp}. Of course, this observation also implies that for larger k_{exp}, the gap between metal and oxide RMS should be closed and could reverse in favor of the oxide RMS decreasing faster. The RMS of both surfaces also shows a slight dependence on k_{exp}. Both sets of data in Figure 3-13 have higher RMS for each surface respectively than the data in Figure 3-9 where $k_{exp} = 2.3$. Going to even higher k_{exp} we see a continued trend in increasing overall RMS. We have found that the lowest overall RMS is achieved when $k_{exp} = 2.0$. At this value, both surfaces change the same amount at each step, which may help keep the two surfaces conformal and minimize overall RMS. Of course, the changes are so slight that a more thorough investigation of accuracy is required to make sure this dependency is not a side-effect of the simulation. We can also predict the trends for depression features as a function of k_{exp}. We expect the gap between metal and oxide RMS (with the oxide RMS decreasing faster) to widen for larger k_{exp} as the oxide grows faster and the metal takes longer to be consumed. On the other hand, smaller k_{exp} should decrease the gap for a depression. We now have a good idea of how our planarization by anodization process would work for ideal

metals with different oxide expansion coefficients. This data would be hard to observe experimentally because the physical process must be fine-tuned for the chemistry of each metal. With our simulation, we can give a good indication of what to expect for different metals and whether that experimentation would be warranted.

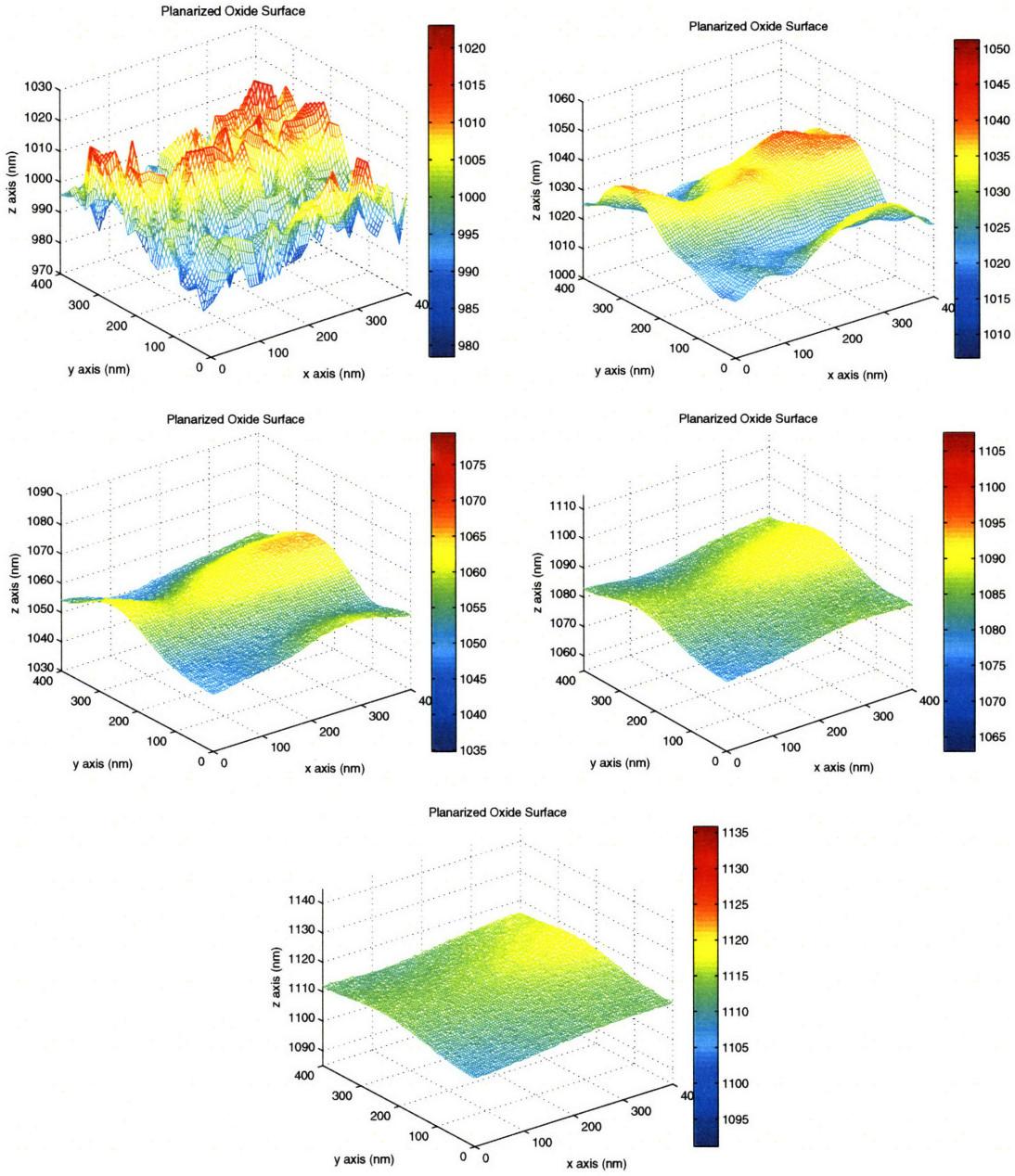


Figure 3-11: Shows the results of our simulation on a computer-generated, $1000nm$ thick Ta_2O_5 sample. Compare with results in Figure 3-2. We show the oxide surface at 5 oxide thicknesses, $t_{ox} = 0, 50, 100, 150, 200nm$, from left to right, top to bottom. We have kept the color and z scales constant across the figures and shifted the z-axis.

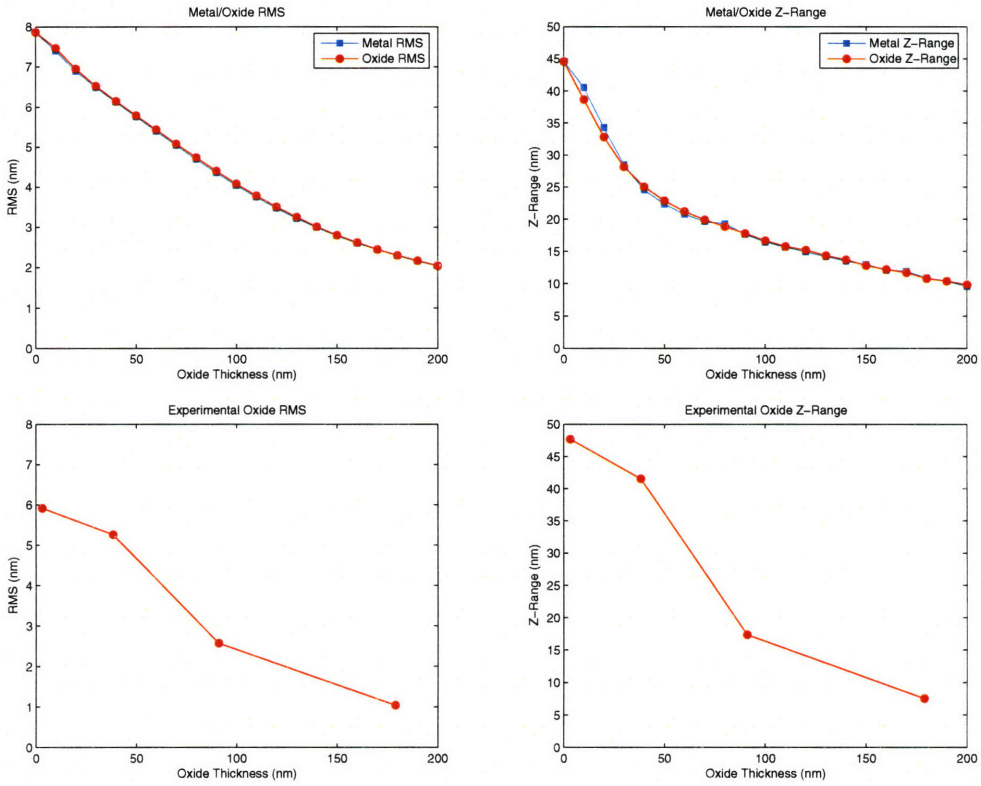


Figure 3-12: The top line shows the RMS and z-range of our simulation on a computer-generated, $1000nm$ thick Ta_2O_5 sample. We observe a generally exponential decay in RMS and z-range with a finite asymptote determined by the stability of the model (step size and grid spacing). Along the bottom row, we reproduce these calculations for the experimental data from Figure 3-2 on the same axes as our simulation results.

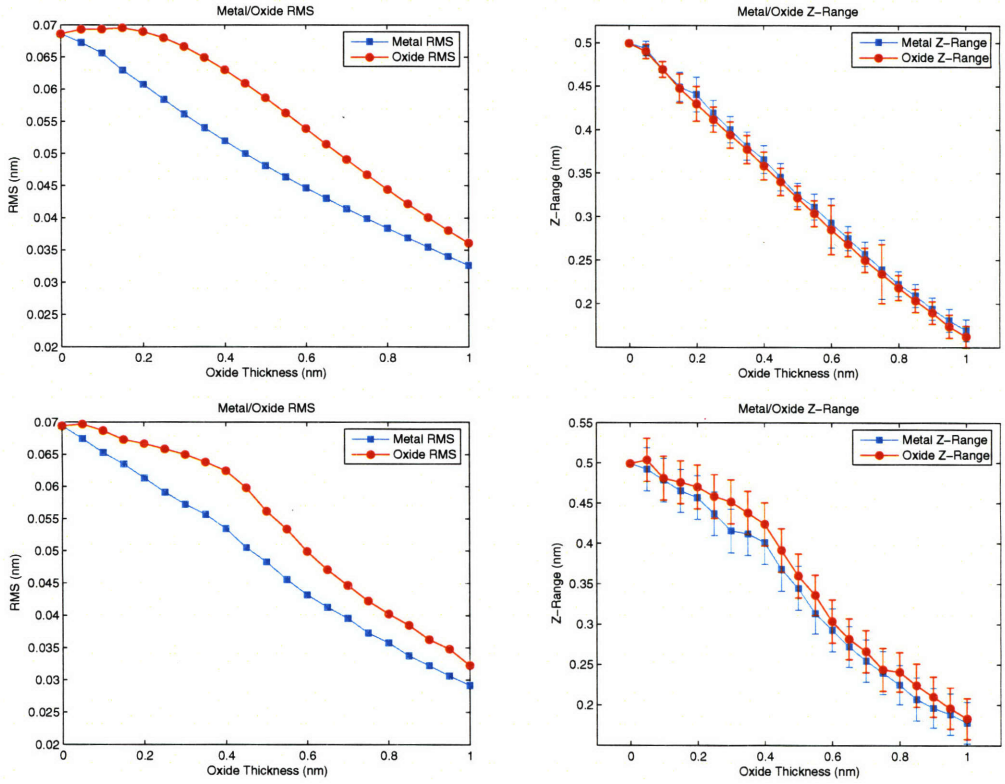


Figure 3-13: a) and b) show the RMS and z-range, respectively, of a single hill with an oxide expansion coefficient $k_{exp} = 1.8$ (see Figure 3-9 a) and b) for comparison with $k_{exp} = 2.3$). We used a 129 x 129 grid of points and an oxide step size of 0.005 to minimize inaccuracies from the small k_{exp}. c) and d) show the RMS and z-range, respectively, of a single hill with $k_{exp} = 2.8$. We used the same grid and step size as we did for $k_{exp} = 2.3$, 17 x 17 grid and an oxide step of 0.01 because no added accuracy was needed.

Chapter 4

Conclusion

This thesis detailed two simulations based on models of experimental observations. We showed that these models were able to reproduce experimental behavior with a small set of assumptions. We then applied these models to novel device structures and materials, and presented simulated measurements that are difficult to make experimentally. We have also noted avenues for improvement in both our computations and models.

Our simulation of quantum dot packing showed the significance of size distribution in monolayer formation. Our model assumed hard-sphere inelastic collisions between dots, van der Waals attractions, and random thermal motion, based on the substrate properties. For monodisperse samples, we observed stable, hexagonal close-packing of the dots. As we increased the size distribution of the dots, we observed a decrease in packing and cluster stability. We were able to mitigate these effects by confining the dots along one dimension. We investigated the effective mass as a cause of the size distribution-related packing breakdown. We also reported a simple calculation for the luminescence quantum efficiency of the dots. We plan on extending our simulation interface to make it more useful to other researchers and implement new packing-related calculations.

We demonstrated a model of planarization through electrochemical oxidization. Motivated by experimental evidence, we created a fundamental model of this process with only two assumptions: conformal oxidation and metal conservation. We demon-

strated that this model reproduces planarization of large surface defects on both the metal and oxide surfaces. We verified an expected difference in the planarization of the metal and oxide surfaces for individual additive and subtractive defects. We showed the convergence of our simulation results as we increased the accuracy of our computations. We simulated the anodization of a realistic Ta sample and found close agreement with experimental data. We also noted some discrepancies in our simulation and gave our explanation of their origins. We then reported the effect that different oxide expansion coefficients have on planarization. This data shows that other metals could be planarized by these methods. In the future, we hope to increase the accuracy and efficiency of our simulation and use it as a tool for the investigation of this type of planarization in other metals.

Appendix A

Quantum Dot Packing Code

Listing A.1: Quantum dot simulation main class containing time update algorithm.

```
package se102.gb;

import se102.gb.gobjects.Ball;
import se102.gb.gobjects.Gizmo;
import se102.gb.util.Reflector;
import se102.gb.util.BallBunch;
import se102.gb.util.BallLocalizer;
import se102.gb.util.Pair;
import physics.*;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
import java.util.List;
import java.util.Iterator;
import java.util.Map;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Random;

/**
 * This class drives the updates of the game area during play.
 *
 * @specfield board | Board // updated periodically by this runner
 * @specfield timer | Timer // calls updates to go through the physics loop
 * @specfield currentDelay | time // the delay between update
 * @specfield updater | physics loop // for debugging, this is the manually manipulatable Updater
 */

public class Runner {
  private GraphArea board;

  private Timer timer;

  private int currentDelay;

  private Updater updater;
```

```
40    private boolean debug;

      private int maxIterations;

      private Random random = new Random(205);
45
      BallLocalizer bl;

      // Constructors

50    /**
       * Main constructor for making a running physics loop on the board with a delay between updates
            of milliDelay
       *
       * @requires board != null
       * @effects initializes the fields
55     */
      public Runner(GraphArea board, int milliDelay) {
        // this.board = board;
        // this.timer = new Timer(milliDelay, new Updater());
        // this.currentDelay = milliDelay;
60      // this.epsilon = milliDelay / 8;
        // this.timer.start();
        this(board, milliDelay, false);
      }

65    /**
       * Debugging constructor that allows updates to be done sequencially
       *
       * @requires board != null
       * @effects initializes all fields except timer
70     */
      public Runner(GraphArea board, int milliDelay, boolean debug) {
        this.board = board;
        this.currentDelay = milliDelay;
        // this.epsilon = 0; // milliDelay / 8;
75      Geometry.setForesight(2 * currentDelay);
        if (debug) {
          this.updater = new Updater();
        } else {
          this.timer = new Timer(milliDelay, new Updater());
80        this.timer.start();
        }
        this.debug = debug; // true if no visualization is used
        maxIterations = 2500;
        bl = new BallLocalizer(board.getSizeX(), board.getSizeY(), 15.0);
85    }

      // Getters and Setters

      public double getEpsilonSpace() {
90      return updater.getEpsilonSpace();
      }

      // Modifiers

95    /**
       * Changes the delay between each update of the board
       *
       * @modifies timer
```

```
     * @effects changes update delay
     */
    public void changeDelay(int milliDelay) {
      timer.setDelay(milliDelay);
      currentDelay = milliDelay;
      // epsilon = 0; //milliDelay / 8;
      Geometry.setForesight(2 * milliDelay);
    }

    /**
     * Stops updates of the board but remains ready to start again.
     *
     * @modifies timer
     * @effects stops the timer temporarily
     */
    public void pause() {
      timer.stop();
    }

    /**
     * Restarts updates of the board after it had been previously stopped
     *
     * @requires this.pause() had been previously called since the last construction or this.
         restart() call
     * @modifies timer
     * @effects restarts the timer after it had been paused
     */
    public void restart() {
      timer.restart();
    }

    /**
     * For debugging purposes only, runs one update cycle of the updater
     *
     * @modifies this.updater
     * @throws RuntimeException
     *               if this Runner is not in debug mode
     */
    public void runUpdate() {
      if (updater != null) {
        updater.actionPerformed(new ActionEvent(this, 0, "update"));
      } else {
        throw new RuntimeException("Not_in_debug_mode");
      }
    }

    public void setMaxIterations(int iterations) {
      maxIterations = iterations;
    }

    /**
     * Updates the state of the board through periodic calls to actionPerformed by the timer
     *
     */
    class Updater implements ActionListener {
      private double epsilonTime = 1e-6;

      private double epsilonSpace = 1e-6;

      private double epsilonVelocity = 1e-1;
```

```
    private double epsilonPrecision = 1e-10;
160

    public double getEpsilonSpace() {
      return epsilonSpace;
    }
165
    double timeSinceRandomizeVelocities = 0.0;

    /**
     * The method called be the actionPerformed method to do the updates of all the balls on the
         board to account for gravity and friction and for collision with other gizmos and balls.
170  *
     * @requires ballArray != null, reflectorArray != null, and for all i, ballArray[i] != null,
         reflectorArray[i] != null
     * @modifies all balls in ballArray
     * @effects Uses the PhysicsSimulator to update the velocity of each ball with the current
         gravity and friction. Then uses the Geometry class in the physics package to find the
         closest
     *          collision for each ball. Moves each ball to the point of collision and then uses
         the Geometry reflect methods to collide. The method loops until all balls have performed
          all the
175  *          collisions that should occur in this period.
     */
    private void arrayLoopOverlapProofMutualAttraction(Ball[] ballArray, Reflector[]
        reflectorArray) {
      int loopNum = 0;
//      System.out.println("attraction");
180   bl.clear();
      bl.addBalls(ballArray);
      boolean[] activeBalls = new boolean[ballArray.length];
      java.util.Arrays.fill(activeBalls, true);
      double delayLeft = currentDelay/1000.0;
185   double deltaDelay = 0;
      Reflector[] closestReflectors = new Reflector[ballArray.length];
      Ball[] closerBalls = new Ball[ballArray.length];
      int[] closerBallsIndex = new int[ballArray.length];
      double[] shortestTimes = new double[ballArray.length];
190 //    java.util.Arrays.fill(shortestTimes, Double.MAX_VALUE);
    //    java.util.Arrays.fill(closestReflectors, null);
    //    java.util.Arrays.fill(closerBalls, null);
    //    java.util.Arrays.fill(closerBallsIndex, 0);
      Rectangle origBallBounds;
195   Circle ballPerimeter;
      double collisionTime;
      Vect collisionDistance;
      double[] timeSinceVelocityUpdate = new double[ballArray.length];
      java.util.Arrays.fill(timeSinceVelocityUpdate, 0.0);
200
      //add random velocities
      timeSinceRandomizeVelocities += delayLeft;
      if(timeSinceRandomizeVelocities >= PhysicsSimulator.getRandomVelocityDelay()) {
        for(int i = 0; i < ballArray.length; i++) {
205       Vect randVect = new Vect(new Angle(2*Math.PI*random.nextDouble()),PhysicsSimulator.
              getRandomVelocityMagnitude(ballArray[i].getMass()));
          ballArray[i].setVelocity(ballArray[i].getVelocity().plus(randVect));
        }
        timeSinceRandomizeVelocities = 0.0;
      }
210
      bl.updateBalls();
```

```
        ArrayList<Pair<Pair<Ball,Integer>,Pair<Ball,Integer>>> pairs = bl.getAllPairsSeperatedBy(
            PhysicsSimulator.getMaximumForceDistanceThresh());
        // change the ball velocities based on their mutual attractions
        Iterator<Pair<Pair<Ball,Integer>,Pair<Ball,Integer>>> pairsIterator = pairs.iterator();
215     while(pairsIterator.hasNext()) {
//        System.out.println("found pair");
          Pair<Pair<Ball,Integer>,Pair<Ball,Integer>> balls = pairsIterator.next();
          Ball ball1 = balls.getFirst().getFirst();
//          int index1 = balls.getFirst().getSecond();
220       Ball ball2 = balls.getSecond().getFirst();
//          int index2 = balls.getSecond().getSecond();
          Vect diff = ball1.getCenter().minus(ball2.getCenter());
          double dist = diff.length()-ball1.getRadius()-ball2.getRadius();
          if(dist > PhysicsSimulator.getMinimumForceDistanceThresh()) { // precludes any negative
              dist from overlaps
225         //have to update for velocity from the previous times because we're changing it's
                velocity
//            ball1.setVelocity(PhysicsSimulator.updateVelocity(ball1.getCenter(),ball1.getVelocity
        (), timeSinceVelocityUpdate[index1]));
//            ball2.setVelocity(PhysicsSimulator.updateVelocity(ball2.getCenter(),ball2.getVelocity
        (), timeSinceVelocityUpdate[index2]));
            //add attraction
//            System.out.println("adding attraction");
230         if(ball1.getMass() == ball2.getMass()) {
              Vect update = diff.unitSize().times(PhysicsSimulator.updateVelocityAttraction(dist,
                  ball1.getMass(),delayLeft));
              ball1.setVelocity(ball1.getVelocity().plus(update.neg()));
              ball2.setVelocity(ball2.getVelocity().plus(update));
            } else {
235           // the updates made seperately for each ball
              Vect update1 = diff.unitSize().times(-PhysicsSimulator.updateVelocityAttraction(dist,
                  ball1.getMass(),delayLeft));
              Vect update2 = diff.unitSize().times(PhysicsSimulator.updateVelocityAttraction(dist,
                  ball2.getMass(),delayLeft));
              ball1.setVelocity(ball1.getVelocity().plus(update1));
              ball2.setVelocity(ball2.getVelocity().plus(update2));
240         }
//            change velocity update info if not already being updated
//            if(timeSinceVelocityUpdate[index1] > 0.0) {
//              timeSinceVelocityUpdate[index1] = 0.0;
//              updateIndexes[updateIndex] = index1;
245 //              updateIndex++;
//            }
//            if(timeSinceVelocityUpdate[index2] > 0.0) {
//              timeSinceVelocityUpdate[index2] = 0.0;
//              updateIndexes[updateIndex] = index2;
250 //              updateIndex++;
//            }
        }
      }

255   int[] updateIndexes = new int[ballArray.length];
      int updateIndex = 0;
      //initialize updateIndexes
      for(;updateIndex < updateIndexes.length; updateIndex++) {
        updateIndexes[updateIndex] = updateIndex;
260   }

      // System.out.println("At while loop");
      boolean noMoreCollisions = false;
      while (delayLeft > epsilonTime) {
```

```
265           loopNum++;
              // System.out.println("in while loop");
              // closestReflectors = new Reflector[ballArray.length];
              // closerBalls = new Ball[ballArray.length];
              // closerBallsIndex = new int[ballArray.length];
270           // shortestTimes = new double[ballArray.length];
              // java.util.Arrays.fill(shortestTimes, Double.MAX_VALUE);
              // update all velocities
              boolean noActive = true;
              for (int i = 0; i < activeBalls.length; i++) {
275             if (activeBalls[i] == false) {
    //              System.out.println("activeBalls["+i+"] is false");
                continue;
                } else {
                  noActive = false;
280               break;
                }
              }
              // System.out.println("loop " + loopnum);
              // return to the top loop if all balls are null
285           if (noActive || noMoreCollisions || loopNum > maxIterations) {
    //          System.out.println("loop number is " + loopNum);
                if(loopNum > maxIterations)
                  System.out.println("Number of iterations went over maximum of " + maxIterations + "
                       iterations");
                return;
290           }

    //          java.util.Arrays.fill(shortestTimes, Double.MAX_VALUE);
    //          java.util.Arrays.fill(closestReflectors, null);
    //          java.util.Arrays.fill(closerBalls, null);
295 //          java.util.Arrays.fill(closerBallsIndex, 0);
              // reset collision parameters for balls about to be updated
              for (int index = 0; index < updateIndex; index++) {
                int i = updateIndexes[index];
                shortestTimes[i] = Double.MAX_VALUE;
300             closestReflectors[i] = null; //not neccessary
                closerBalls[i] = null; // not neccessary
                closerBallsIndex[i] = 0; // not neccessary
              }
    //          gizmo collision detection
305           for (int index = 0; index < updateIndex; index++) {
                // if (activeBalls[i] == false) {
                // continue;
                // }
                int i = updateIndexes[index];
310             //expand size slightly to keep balls seperate for reflection step
                ballPerimeter = new Circle(ballArray[i].getCenter(),ballArray[i].getRadius()+
                     epsilonSpace);
                for (int j = 0; j < reflectorArray.length; j++) {
                  // reflector = (Reflector)reflectors.next();
                  if (reflectorArray[j].isCircle()) {
315                 if (reflectorArray[j].isTranslating()) {
                      if (reflectorArray[j].isHoriz()) {
                        collisionTime = Geometry.timeUntilCircleCollision(reflectorArray[j].getCircle()
                             , ballPerimeter, ballArray[i].getVelocity().minus(
                            new Vect(reflectorArray[j].getTransVelocity(), 0.0)));
                      } else {
320                     collisionTime = Geometry.timeUntilCircleCollision(reflectorArray[j].getCircle()
                             , ballPerimeter, ballArray[i].getVelocity().minus(
                            new Vect(0.0, reflectorArray[j].getTransVelocity())));
```

```
                    }
                  } else if (reflectorArray[j].getAngularVelocity() == 0.0) {
                    collisionTime = Geometry.timeUntilCircleCollision(reflectorArray[j].getCircle(),
                        ballPerimeter, ballArray[i].getVelocity());
                  } else {
                    collisionTime = Geometry.timeUntilRotatingCircleCollision(reflectorArray[j].
                        getCircle(), reflectorArray[j].getCenter(), Math.toRadians(reflectorArray[j]
                        .getAngularVelocity()), ballPerimeter, ballArray[i].getVelocity());
                  }
                } else {
                  if (reflectorArray[j].isTranslating()) {
                    if (reflectorArray[j].isHoriz()) {
                      collisionTime = Geometry.timeUntilWallCollision(reflectorArray[j].
                          getLineSegment(), ballPerimeter, ballArray[i].getVelocity().minus(
                          new Vect(reflectorArray[j].getTransVelocity(), 0.0)));
                    } else {
                      collisionTime = Geometry.timeUntilWallCollision(reflectorArray[j].
                          getLineSegment(), ballPerimeter, ballArray[i].getVelocity().minus(
                          new Vect(0.0, reflectorArray[j].getTransVelocity())));
                    }
                  } else if (reflectorArray[j].getAngularVelocity() == 0.0) {
                    collisionTime = Geometry.timeUntilWallCollision(reflectorArray[j].getLineSegment
                        (), ballPerimeter, ballArray[i].getVelocity());
                  } else {
                    collisionTime = Geometry.timeUntilRotatingWallCollision(reflectorArray[j].
                        getLineSegment(), reflectorArray[j].getCenter(), Math.toRadians(
                        reflectorArray[j]
                        .getAngularVelocity()), ballPerimeter, ballArray[i].getVelocity());
                  }
                }
                if (collisionTime < shortestTimes[i]) {
                  shortestTimes[i] = collisionTime;
                  closestReflectors[i] = reflectorArray[j];
                  closerBalls[i] = null;
                  closerBallsIndex[i] = 0;
                }
                // collisionDistance =
                // ballArray[i].getVelocity().times(collisionTime);
                // if(collisionDistance.length() < epsilonSpace &&
                // collisionTime > otherCloseCollisionTime[i]) {
                // otherCloseCollisionTime[i] = collisionTime;
                // }
              }
              for (int j = 0; j < ballArray.length; j++) {
    //            if (j == i) {
    //              continue;
    //            }
                Circle otherBallPerimeter = new Circle(ballArray[j].getCenter(),ballArray[j].
                    getRadius()+epsilonSpace);
                collisionTime = Geometry.timeUntilBallBallCollision(ballPerimeter, ballArray[i].
                    getVelocity(), otherBallPerimeter, ballArray[j].getVelocity());
                if (collisionTime < shortestTimes[i]) {
                  // System.out.println("collides with ball");
                  shortestTimes[i] = collisionTime;
                  closerBalls[i] = ballArray[j];
                  closerBallsIndex[i] = j;
                  closestReflectors[i] = null;
                }
                if(collisionTime < shortestTimes[j]) {
                  shortestTimes[j] = collisionTime;
                  closerBalls[j] = ballArray[i];
```

```
                closerBallsIndex[j] = i;
375             closestReflectors[j] = null;
              }
            }
          }

380       updateIndex = 0;

          // collision resolution
          /*
          double closeTimes[] = new double[ballArray.length];
385       // find dt for all balls
          for(int i = 0; i < ballArray.length; i++) {
            if(closerBalls[i] != null) {
              double curDist = ballArray[i].getCenter().minus(closerBalls[i].getCenter()).length();
              if(curDist < ballArray[i].getRadius()+closerBalls[i].getRadius()-4*epsilonSpace) {
390             closeTimes[i] = 0.0;
                continue;
              }
              double dt = 4*epsilonSpace/(ballArray[i].getVelocity().minus(closerBalls[i].
                  getVelocity()).length());
              closeTimes[i] = shortestTimes[i]-dt;
395           } else {
              if(closestReflectors[i].isCircle()) {
                double dt = 2*epsilonSpace/(ballArray[i].getVelocity().length());
                closeTimes[i] = shortestTimes[i]-dt;
              } else {
400             double dt = 2*epsilonSpace/(ballArray[i].getVelocity().times(Math.sin(
                    closestReflectors[i].getLineSegment().angle().radians()+ballArray[i].
                    getVelocity().angle().radians()))).length());
                closeTimes[i] = shortestTimes[i]-dt;
              }
              if(closeTimes[i] < 0.0) {
                closeTimes[i] = 0.0; //too close already
405           }
            }
          }
          */

410       int firstCollisionIndex = -1;
          int otherBallIndex = -1;
          double firstCollisionTime = delayLeft;// Double.MAX_VALUE;
          // boolean ballCollision = false;
          for (int i = 0; i < shortestTimes.length; i++) {
415         if (shortestTimes[i] < firstCollisionTime && activeBalls[i] != false) {
              firstCollisionIndex = i;
              firstCollisionTime = shortestTimes[i];
              if (closerBalls[i] != null)
                otherBallIndex = closerBallsIndex[i];
420           else
                otherBallIndex = -1;
            }
          }

425 //        if(firstCollisionTime == 0.0) {
    //          System.out.println("ball " + firstCollisionIndex + " overlaps with ball " +
        otherBallIndex);
    //          throw new RuntimeException("Created overlap of number " + firstCollisionIndex
         + " " + ballArray[firstCollisionIndex] + (otherBallIndex != -1 ? " with number " +
        otherBallIndex + " " + ballArray[otherBallIndex] : ""));
    //        }
```

```
430           // update all balls other than the next collision
              for (int i = 0; i < ballArray.length; i++) {
                origBallBounds = ballArray[i].getBounds();
                ballPerimeter = ballArray[i].getCircle();
                if (firstCollisionIndex == -1) {
435               if (activeBalls[i] == false)
                    continue;
                  Vect endDistance = ballArray[i].getVelocity().times(delayLeft);
    //              collisionDistance = ballArray[i].getVelocity().times(shortestTimes[i]);
                  // collisionDistance.length() is necessarily > endDistance.length() since no
                       collision within delayLeft;
440 //              if (collisionDistance.minus(endDistance).length() > minDistance - epsilonPrecision
        || endDistance.length() == 0.0) {
                    // move complete distance to the end of the delay period
                  // never have to worry about other balls actually having closer collision than
                       epsilon space
                  ballArray[i].setPosition(ballArray[i].getPosition().plus(endDistance));
                  ballArray[i].setVelocity(PhysicsSimulator.updateVelocity(ballArray[i].getCenter(),
                       ballArray[i].getVelocity(), delayLeft+timeSinceVelocityUpdate[i]));
445 //              }
                  /*else {
                    if(closerBalls[i] == null) {
                      activeBalls[i] = resolveCollisionInelastic(ballArray[i], closerBalls[i],
                           closestReflectors[i], shortestTimes[i], timeSinceVelocityUpdate[i], 0.0);
                    } else if(i > closerBallsIndex[i]) {
450                   activeBalls[i] = resolveCollisionInelastic(ballArray[i], closerBalls[i],
                           closestReflectors[i], shortestTimes[i], timeSinceVelocityUpdate[i],
                          timeSinceVelocityUpdate[closerBallsIndex[i]]);
                      activeBalls[closerBallsIndex[i]] = activeBalls[i];
                    }
                  }
455               */
                  // don't bother setting timeSinceVelocityUpdate to 0.0 for all balls since exiting
                       anyway
                } else {
                  if (i == firstCollisionIndex || i == otherBallIndex || activeBalls[i] == false)
                    continue;
                  if(closerBalls[i] != null && ballArray[i].getPosition().plus(ballArray[i].getVelocity
460                    ().times(shortestTimes[firstCollisionIndex])).minus(closerBalls[i].getPosition().
                       plus(closerBalls[i].getVelocity().times(shortestTimes[firstCollisionIndex]))).
                       length() < 0.0) {
                    System.out.println("Balls will overlap");
                  }
                  Vect firstCollisionDistance = ballArray[i].getVelocity().times(shortestTimes[
                       firstCollisionIndex]);
    //              collisionDistance = ballArray[i].getVelocity().times(shortestTimes[i]);
465               // only move the ball to within epsilonSpace of a collision
                  // collisionDistance.length() > firstCollisionDistance.length() because
                       firstCollision time is shortest
    //              if (collisionDistance.minus(firstCollisionDistance).length() > minDistance -
        epsilonPrecision || firstCollisionDistance.length() == 0.0) {
                    // move complete distance as colliding object
                  ballArray[i].setPosition(ballArray[i].getPosition().plus(firstCollisionDistance));
470 //                ballArray[i].setVelocity(PhysicsSimulator.updateVelocity(ballArray[i].getCenter()
        ,ballArray[i].getVelocity(), shortestTimes[firstCollisionIndex]));
                  timeSinceVelocityUpdate[i] += shortestTimes[firstCollisionIndex],
                  shortestTimes[i] -= shortestTimes[firstCollisionIndex];
    //            }
                  /*else {
475               // move to within epsilonSpace of closest collision
```

```
                // make sure that only one of the balls resolves the collision
                if(closerBalls[i] == null) {
                  activeBalls[i] = resolveCollisionInelastic(ballArray[i], closerBalls[i],
                      closestReflectors[i], shortestTimes[i], timeSinceVelocityUpdate[i], 0.0);
                  timeSinceVelocityUpdate[i] = 0.0;
                  updateIndexes[updateIndex] = i;
                  updateIndex++;
                } else if(i > closerBallsIndex[i]) {
                  activeBalls[i] = resolveCollisionInelastic(ballArray[i], closerBalls[i],
                      closestReflectors[i], shortestTimes[i], timeSinceVelocityUpdate[i],
                      timeSinceVelocityUpdate[closerBallsIndex[i]]);
                  activeBalls[closerBallsIndex[i]] = activeBalls[i];
                  timeSinceVelocityUpdate[i] = 0.0;
                  timeSinceVelocityUpdate[closerBallsIndex[i]] = 0.0;
                  updateIndexes[updateIndex] = i;
                  updateIndex++; // could combine the lines but don't want to
                  updateIndexes[updateIndex] = closerBallsIndex[i];
                  updateIndex++;
                }
              }*/
              // update velocity
              // ballArray[i].setVelocity(PhysicsSimulator.updateVelocity(ballArray[i].getVelocity
                  (), shortestTimes[firstCollisionIndex]));
            }
            if(!debug) {
              board.repaint(origBallBounds);
              board.repaint(ballArray[i].getBounds());
            }
          }

          if (firstCollisionIndex == -1) {
    //        System.out.println("No closest collision so loop has ended");
            noMoreCollisions = true;

            deltaDelay = delayLeft;
            delayLeft = 0;
            // return;
    //        continue;
          } else {
            // replaced long portion with function call
            activeBalls[firstCollisionIndex] = resolveCollisionInelasticNoOverlapCheck2(ballArray[
                firstCollisionIndex], closerBalls[firstCollisionIndex], closestReflectors[
                firstCollisionIndex],
                shortestTimes[firstCollisionIndex], timeSinceVelocityUpdate[firstCollisionIndex],
                    closerBalls[firstCollisionIndex] == null ? 0.0 : timeSinceVelocityUpdate[
                    closerBallsIndex[firstCollisionIndex]]);
            timeSinceVelocityUpdate[firstCollisionIndex] = 0.0;
            updateIndexes[updateIndex] = firstCollisionIndex;
            updateIndex++;
            if (closerBalls[firstCollisionIndex] != null) {
              activeBalls[closerBallsIndex[firstCollisionIndex]] = activeBalls[firstCollisionIndex
                  ];
              timeSinceVelocityUpdate[closerBallsIndex[firstCollisionIndex]] = 0.0;
              updateIndexes[updateIndex] = closerBallsIndex[firstCollisionIndex];
              updateIndex++;
            } else {
              closestReflectors[firstCollisionIndex].getGizmo().specialAction(ballArray[
                  firstCollisionIndex]);
              closestReflectors[firstCollisionIndex].getGizmo().notifyTriggerListeners();
            }
```

```
            deltaDelay = shortestTimes[firstCollisionIndex];
            delayLeft -= deltaDelay;
          }

          /*
          bl.updateBalls();
          ArrayList<Pair<Pair<Ball,Integer>,Pair<Ball,Integer>>> pairs = bl.getAllPairsSeperatedBy(
              PhysicsSimulator.getMaximumForceDistanceThresh());
          // change the ball velocities based on their mutual attractions
          Iterator<Pair<Pair<Ball,Integer>,Pair<Ball,Integer>>> pairsIterator = pairs.iterator();
          while(pairsIterator.hasNext()) {
    //        System.out.println("found pair");
            Pair<Pair<Ball,Integer>,Pair<Ball,Integer>> balls = pairsIterator.next();
            Ball ball1 = balls.getFirst().getFirst();
            int index1 = balls.getFirst().getSecond();
            Ball ball2 = balls.getSecond().getFirst();
            int index2 = balls.getSecond().getSecond();
            Vect diff = ball1.getCenter().minus(ball2.getCenter());
            double dist = diff.length()-ball1.getRadius()-ball2.getRadius();
            if(dist > PhysicsSimulator.getMinimumForceDistanceThresh()) {
              //have to update for velocity from the previous times because we're changing it's
                  velocity
              ball1.setVelocity(PhysicsSimulator.updateVelocity(ball1.getCenter(),ball1.getVelocity
                  (), timeSinceVelocityUpdate[index1]));
              ball2.setVelocity(PhysicsSimulator.updateVelocity(ball2.getCenter(),ball2.getVelocity
                  (), timeSinceVelocityUpdate[index2]));
              //add attraction
    //          System.out.println("adding attraction");
              if(ball1.getMass() == ball2.getMass()) {
                Vect update = diff.unitSize().times(PhysicsSimulator.updateVelocityAttraction(dist,
                    ball1.getMass(),deltaDelay));
                ball1.setVelocity(ball1.getVelocity().plus(update.neg()));
                ball2.setVelocity(ball2.getVelocity().plus(update));
              } else {
                // the updates made seperately for each ball
                Vect update1 = diff.unitSize().times(-PhysicsSimulator.updateVelocityAttraction(
                    dist,ball1.getMass(),deltaDelay));
                Vect update2 = diff.unitSize().times(PhysicsSimulator.updateVelocityAttraction(dist
                    ,ball2.getMass(),deltaDelay));
                ball1.setVelocity(ball1.getVelocity().plus(update1));
                ball2.setVelocity(ball2.getVelocity().plus(update2));
              }
    //          change velocity update info if not already being updated
              if(timeSinceVelocityUpdate[index1] > 0.0) {
                timeSinceVelocityUpdate[index1] = 0.0;
                updateIndexes[updateIndex] = index1;
                updateIndex++;
              }
              if(timeSinceVelocityUpdate[index2] > 0.0) {
                timeSinceVelocityUpdate[index2] = 0.0;
                updateIndexes[updateIndex] = index2;
                updateIndex++;
              }
            }
          }
          */
    //      for(int i = 0; i < ballArray.length; i++) {
    //        if(ballArray[i].getPosition().y() > 585) {
    //          System.out.println("ball fell through");
    //        }
    //      }
```

```
    }
  }

  private boolean resolveCollisionInelasticNoOverlapCheck2(Ball ball, Ball closerBall,
      Reflector reflector, double collisionTime) {
585   return resolveCollisionInelasticNoOverlapCheck2(ball,closerBall,reflector,collisionTime
        ,0,0);
  }

  private boolean resolveCollisionInelasticNoOverlapCheck2(Ball ball, Ball closerBall,
      Reflector reflector, double collisionTime, double velocityUpdateTime, double
      closerBallVelocityUpdateTime) {
    Rectangle origBallBounds = ball.getBounds();
590 Circle ballPerimeter = ball.getCircle();
    if (closerBall != null) {
      ball.setPosition(ball.getPosition().plus(ball.getVelocity().times(collisionTime)));
      if(!debug) {
        board.repaint(origBallBounds);
595     board.repaint(ball.getBounds());
      }
      origBallBounds = closerBall.getBounds();
      closerBall.setPosition(closerBall.getPosition().plus(closerBall.getVelocity().times(
          collisionTime)));
      if(!debug) {
600     board.repaint(origBallBounds);
        board.repaint(closerBall.getBounds());
      }
      /*
      if(PhysicsSimulator.isInelastic() && ball.getVelocity().minus(closerBall.getVelocity()).
          length() <= PhysicsSimulator.getBallStickThresh()) {
605     // perform totally inelastic collision
        System.out.println("inelastic collision");
        BallBunch ballBunch = bunches.get(ball);
        BallBunch closerBallBunch = bunches.get(closerBall);
        if(ballBunch == null) {
610       ballBunch = new BallBunch(ball);
          bunches.put(ball,ballBunch);
        }
        if(closerBallBunch == null) {
          closerBallBunch = new BallBunch(closerBall);
615       bunches.put(closerBall,closerBallBunch);
        }
        double ballMass = ballBunch.getMass();
        double closerBallMass = closerBallBunch.getMass();
        Vect ballMassVel = ball.getVelocity().times(ballMass);
620     Vect closerBallMassVel = closerBall.getVelocity().times(closerBallMass);
        Vect finalVelocity = ballMassVel.plus(closerBallMassVel).times(1/(ballMass+
            closerBallMass));
//        ball.setVelocity(finalVelocity);
//        closerBall.setVelocity(finalVelocity);
//        BallBunch newBunch;
625     if(ballBunch.getSize() > closerBallBunch.getSize()) {
          // currently impossible for same bunch to collide with itself
          ballBunch.addBalls(closerBallBunch);
          Iterator<Ball> iter = closerBallBunch.getBalls().iterator();
          while(iter.hasNext()) {
630         bunches.put(iter.next(),ballBunch);
          }
          //update all velocities of new bunch
          iter = ballBunch.getBalls().iterator();
          while(iter.hasNext()) {
```

```
635                 iter.next().setVelocity(finalVelocity);
                  }
                } else {
                  closerBallBunch.addBalls(ballBunch);
                  Iterator<Ball> iter = ballBunch.getBalls().iterator();
640               while(iter.hasNext()) {
                    bunches.put(iter.next(),closerBallBunch);
                  }
    //              update all velocities of new bunch
                  iter = closerBallBunch.getBalls().iterator();
645               while(iter.hasNext()) {
                    iter.next().setVelocity(finalVelocity);
                  }
                }
                // add all balls to larger ball bunch and then update map for all balls in other bunch
650           } else {
                Geometry.VectPair velocities = Geometry.reflectBalls(ball.getCenter(), ball.getMass(),
                     ball.getVelocity(), closerBall.getCenter(), closerBall.getMass(), closerBall.
                     getVelocity());
                if(PhysicsSimulator.isInelastic()) {
                  ball.setVelocity(velocities.v1.times(PhysicsSimulator.getElasticity()));
                  closerBall.setVelocity(velocities.v2.times(PhysicsSimulator.getElasticity()));
655               if(bunches.get(ball) != null) {
                    bunches.get(ball).removeBall(ball);
    //                bunches.put(ball,null);
                    bunches.remove(ball);
                  }
660               if(bunches.get(closerBall) != null) {
                    bunches.get(closerBall).removeBall(ball);
    //                bunches.put(closerBall,null);
                    bunches.remove(closerBall);
                  }
665             } else {
                  ball.setVelocity(velocities.v1);
                  closerBall.setVelocity(velocities.v2);
                }
              }
670           */
              Geometry.VectPair velocities = Geometry.reflectBalls(ball.getCenter(), ball.getMass(),
                   ball.getVelocity(), closerBall.getCenter(), closerBall.getMass(), closerBall.
                   getVelocity());
              if(PhysicsSimulator.isInelastic()) {
                // only change the velocity along the direction that the balls touch
                Vect diff = closerBall.getCenter().minus(ball.getCenter()).unitSize(); // vector from
                     closerBall to ball
675             Vect vel1para = diff.times(velocities.v1.dot(diff));
                Vect vel1perp = velocities.v1.minus(vel1para);
                Vect vel2para = diff.times(velocities.v2.dot(diff));
                Vect vel2perp = velocities.v2.minus(vel2para);
    //            stop ball from getting arbitrarily close to zero velocity so that it doesn't keep
        bouncing straight into ball
680 //          if(vel1para.length() < PhysicsSimulator.getCompletelyInelasticThresh())
    //            vel1para = Vect.ZERO;
    //          else
    //            vel1para = vel1para.times(PhysicsSimulator.getElasticity());
                Vect velInelasticPara = vel1para.times(ball.getMass()).plus(vel2para.times(closerBall.
                     getMass()).times(1/(ball.getMass()+closerBall.getMass()));
685             vel1para = vel1para.times(PhysicsSimulator.getElasticity()).plus(velInelasticPara.times
                     (1-PhysicsSimulator.getElasticity()));
                ball.setVelocity(vel1para.plus(vel1perp));
    //          if(vel2para.length() < PhysicsSimulator.getCompletelyInelasticThresh())
```

```
    //            vel2para = Vect.ZERO();
    //          else
690 //            vel2para = vel2para.times(PhysicsSimulator.getElasticity());
              vel2para = vel2para.times(PhysicsSimulator.getElasticity()).plus(velInelasticPara.times
                  (1-PhysicsSimulator.getElasticity()));
              closerBall.setVelocity(vel2para.plus(vel2perp));
    //          ball.setVelocity(velocities.v1.times(PhysicsSimulator.getElasticity()));
    //          closerBall.setVelocity(velocities.v2.times(PhysicsSimulator.getElasticity()));
695         } else {
              ball.setVelocity(velocities.v1);
              closerBall.setVelocity(velocities.v2);
            }
            if (ball.getVelocity().minus(closerBall.getVelocity()).length() < epsilonVelocity && ball
                .getVelocity().length() < epsilonVelocity && closerBall.getVelocity().length() <
                epsilonVelocity) {
700           return false;
              // activeBalls[firstCollisionIndex] = false;
            } else {
              ball.setVelocity(PhysicsSimulator.updateVelocity(ball.getCenter(),ball.getVelocity(),
                  collisionTime+velocityUpdateTime));
              closerBall.setVelocity(PhysicsSimulator.updateVelocity(closerBall.getCenter(),
                  closerBall.getVelocity(), collisionTime+closerBallVelocityUpdateTime));
705 //            ball.setVelocity(PhysicsSimulator.updateVelocity(ball.getCenter(),ball.getVelocity()
        , actualTime));
              return true;
            }
    //        return true;
          } else if (reflector.isCircle()) {
710 //        ball.setPosition(ball.getPosition().plus(ball.getVelocity().unitSize().times(
        collisionDistance.length() - epsilonSpace)));
            ball.setPosition(ball.getPosition().plus(ball.getVelocity().times(collisionTime)));
            // repaint the original and new position of the ball
            if(!debug) {
              board.repaint(origBallBounds);
715           board.repaint(ball.getBounds());
            }
            if (reflector.isTranslating()) {
              if (reflector.isHoriz()) {
                ball.setVelocity(Geometry.reflectCircle(reflector.getCircle().getCenter(), ball.
                    getCenter(), ball.getVelocity().minus(new Vect(reflector.getTransVelocity(), 0.0)
                    ), reflector
720                 .getGizmo().getReflection()));
              } else {
                ball.setVelocity(Geometry.reflectCircle(reflector.getCircle().getCenter(), ball.
                    getCenter(), ball.getVelocity().minus(new Vect(0.0, reflector.getTransVelocity())
                    ), reflector
                    .getGizmo().getReflection()));
              }
725         } else if (reflector.getAngularVelocity() == 0.0) {
              ball.setVelocity(Geometry.reflectCircle(reflector.getCircle().getCenter(), ball.
                  getCenter(), ball.getVelocity(), reflector.getGizmo().getReflection()));
            } else {
              ball.setVelocity(Geometry.reflectRotatingCircle(reflector.getCircle(), reflector.
                  getCenter(), Math.toRadians(reflector.getAngularVelocity()), ballPerimeter, ball.
                  getVelocity(),
                  reflector.getGizmo().getReflection()));
730         }

            if(PhysicsSimulator.isInelastic() && bunches.get(ball) != null) {
              bunches.get(ball).removeBall(ball);
    //          bunches.put(ball,null);
```

```
735             bunches.remove(ball);
                }

                if (ball.getVelocity().length() < epsilonVelocity) {
                  return false;
                  // activeBalls[firstCollisionIndex] = false;
                } else {
                  ball.setVelocity(PhysicsSimulator.updateVelocity(ball.getCenter(),ball.getVelocity(),
                      collisionTime+velocityUpdateTime));
                  return true;
                }
            } else {
//              ball.setPosition(ball.getPosition().plus(ball.getVelocity().unitSize().times(
        collisionDistance.length() - epsilonSpace)));
                ball.setPosition(ball.getPosition().plus(ball.getVelocity().times(collisionTime)));
                // repaint the original and new position of the ball
                if(!debug) {
                  board.repaint(origBallBounds);
                  board.repaint(ball.getBounds());
                }
                if (reflector.isTranslating()) {
                  if (reflector.isHoriz()) {
                    ball.setVelocity(Geometry.reflectWall(reflector.getLineSegment(), ball.getVelocity().
                        minus(new Vect(reflector.getTransVelocity(), 0.0)), reflector.getGizmo().
                        getReflection()));
                  } else {
                    ball.setVelocity(Geometry.reflectWall(reflector.getLineSegment(), ball.getVelocity().
                        minus(new Vect(0.0, reflector.getTransVelocity())), reflector.getGizmo().
                        getReflection()));
                  }
                } else if (reflector.getAngularVelocity() == 0.0) {
                  ball.setVelocity(Geometry.reflectWall(reflector.getLineSegment(), ball.getVelocity(),
                      reflector.getGizmo().getReflection()));
                } else {
                  ball.setVelocity(Geometry.reflectRotatingWall(reflector.getLineSegment(), reflector.
                      getCenter(), Math.toRadians(reflector.getAngularVelocity()), ballPerimeter, ball.
                      getVelocity(),
                      reflector.getGizmo().getReflection()));
                }

                if(PhysicsSimulator.isInelastic() && bunches.get(ball) != null) {
                  bunches.get(ball).removeBall(ball);
//                bunches.put(ball,null);
                  bunches.remove(ball);
                }

                if (ball getVelocity().length() < epsilonVelocity) {
                  return false;
                  // activeBalls[firstCollisionIndex] = false;
                } else {
                  ball setVelocity(PhysicsSimulator.updateVelocity(ball.getCenter(),ball.getVelocity(),
                      collisionTime+velocityUpdateTime));
                  return true.
                }
              }
            }

            /**
             * Stops the translating components of the reflectorList such as Jezzmos and Paddles
             *
             * @requires reflectorList != null
```

```
     * @effect Determines whether any of the translating reflectors intersect or will intersect
         with any of the gizmos (or balls) on the board and calls the proper method to stop
         further
     *          translation.
     */
    private void stopMovers(java.util.List reflectorList) {
790   Iterator reflectors = reflectorList.iterator();
      while (reflectors.hasNext()) {
        Reflector reflector = (Reflector) reflectors.next();
        if (reflector.isTranslating()) {
          Rectangle rect = null;
795       // System.out.println("transVel " +
          // reflector.getTransVelocity());
          if (!reflector.isCircle()) {
            // Rectangle rect = null;
            LineSegment ls = reflector.getLineSegment();
800         if (reflector.isHoriz()) {
              if (reflector.getTransVelocity() > 0) {
                rect = new Rectangle((int) ls.p1().x(), (int) ls.p1().y(), (int) (reflector.
                    getTransVelocity() * currentDelay / 1000.0), (int) (ls.p2().y() - ls.p1().y()
                    ));
              } else {
                rect = new Rectangle((int) (ls.p1().x() + (int) (reflector.getTransVelocity() *
                    currentDelay / 1000.0)), (int) ls.p1().y(), -(int) (reflector.
                    getTransVelocity()
805                 * currentDelay / 1000.0), (int) (ls.p2().y() - ls.p1().y()));
              }
              // if(board.overlaps(rect)) {
              // ((sc102.gb.gobjects.Jezzmo)reflector.getGizmo()).stopGrowing(reflector);
              // }
810         } else {
              if (reflector.getTransVelocity() > 0) {
                rect = new Rectangle((int) ls.p1().x(), (int) ls.p1().y(), (int) (ls.p2().x() -
                    ls.p1().x()), (int) (reflector.getTransVelocity() * currentDelay / 1000.0));
              } else {
                rect = new Rectangle((int) ls.p1().x(), (int) (ls.p1().y() + (int) (reflector.
                    getTransVelocity() * currentDelay / 1000.0)), (int) (ls.p2().x() - ls.p1().x
                    ()),
815                 -(int) (reflector.getTransVelocity() * currentDelay / 1000.0));
              }
            }
            // if(board.overlaps(rect)) {
            // ((sc102.gb.gobjects.Jezzmo)reflector.getGizmo()).stopGrowing(reflector);
820         // }
          } else {
            // Rectangle rect = null;
            // System.out.println("transVel " +
            // reflector.getTransVelocity());
825         Circle c = reflector.getCircle();
            // System.out.println("center is " + c.getCenter() + "
            // and radius is " + c.getRadius());
            if (reflector.isHoriz()) {
              if (reflector.getTransVelocity() > 0) {
830             rect = new Rectangle((int) (c.getCenter().x() + c.getRadius()), (int) (c.
                    getCenter().y() - c.getRadius()),
                    (int) (reflector.getTransVelocity() * currentDelay / 1000.0), (int) c.
                        getRadius() * 2);
                // System.out.println("rectangle is " + rect);
              } else {
                rect = new Rectangle((int) (c.getCenter().x() - c.getRadius() + reflector.
                    getTransVelocity() * currentDelay / 1000.0), (int) (c.getCenter().y() - c.
```

```
                    getRadius()),
                    -(int) (reflector.getTransVelocity() * currentDelay / 1000.0), (int) c.
                        getRadius() * 2);
                }
              } else {
                if (reflector.getTransVelocity() > 0) {
                  rect = new Rectangle((int) (c.getCenter().x() - c.getRadius()), (int) (c.
                      getCenter().y() + c.getRadius()),
                      (int) (reflector.getTransVelocity() * currentDelay / 1000.0), (int) c.
                          getRadius() * 2);
                } else {
                  rect = new Rectangle((int) (c.getCenter().x() - c.getRadius()), (int) (c.
                      getCenter().y() - c.getRadius() + (reflector.getTransVelocity() *
                      currentDelay / 1000.0)),
                      (int) (reflector.getTransVelocity() * currentDelay / 1000.0), (int) c.
                          getRadius() * 2);
                }
              }
              if (!rect.intersects(reflector.getGizmo().getBounds())) { // to
                // protect
                // from
                // false
                // positive
                // of
                // overlapping
                // itself
                List rectOverlaps = board.overlapsReturnList(rect, -1, board.getMode() != GraphArea.
                    JEZZ_MODE && !(reflector.getGizmo() instanceof se102.gb.gobjects.Paddle));
                if (rectOverlaps.size() > 0) {
                  // System.out.println("overlaps are " +
                  // rectOverlaps);
                  if (reflector.getGizmo() instanceof se102.gb.gobjects.Jezzmo) {
                    ((se102.gb.gobjects.Jezzmo) reflector.getGizmo()).stopGrowing(reflector,
                        rectOverlaps);
                  } else if (reflector.getGizmo() instanceof se102.gb.gobjects.Paddle) {
                    ((se102.gb.gobjects.Paddle) reflector.getGizmo()).stopMoving(reflector,
                        rectOverlaps);
                  }
                }
              }
            }
          }
        }

        // save these arrays so new one don't have to be allocated
        Ball[] lastBallArray = new Ball[0];
        Reflector[] lastReflectorArray = new Reflector[0];

        /**
         * Each time the timer goes off this method is called to update the board
         *
         * @modifies board.theGizmos, board.theBalls
         * @effects Determines if any of the gizmos needs a repaint and calls such on the correct
             portion of the board. Compiles a list of all reflectors of all the gizmos and stops any
             translating
         *          gizmos if they will overlap another gizmo in this period. Calls the loop the
             performs physics updates.
         */
        public void actionPerformed(ActionEvent evt) {
          List<Gizmo> gizmoList = board.getGizmos();
```

```
        List<Ball> ballList = board.getBalls();
        Iterator<Gizmo> gizmos = gizmoList.iterator();
885     Gizmo gizmo;
        List<Reflector> reflectorList = new ArrayList<Reflector>();
        Rectangle bounds;
        while (gizmos.hasNext()) {
          gizmo = gizmos.next();
890       if (gizmo.needsRepaint()) {
            bounds = new Rectangle(gizmo.getBounds());
            bounds.grow(1, 1);
            board.repaint(bounds);
          }
895       reflectorList.addAll(gizmo.getPerimeter(currentDelay));
        }
        stopMovers(reflectorList);

        if(ballList.size() != lastBallArray.length) {
900       lastBallArray = new Ball[ballList.size()];
        }
        if(reflectorList.size() != lastReflectorArray.length) {
          lastReflectorArray = new Reflector[reflectorList.size()];
        }
905
        arrayLoopOverlapProofMutualAttraction(ballList.toArray(lastBallArray), reflectorList.
            toArray(lastReflectorArray));

      }

910   }
  }
```

Listing A.2: Quantum dot simulation initialization and measurement class.

```
  package se102.gb;

  import java.util.*;
  import java.awt.*;
5 import java.awt.geom.*;
  import physics.*;
  import se102.gb.util.*;
  import se102.gb.gobjects.*;

10 import java.io.BufferedWriter;
  import java.io.FileWriter;
  import java.io.IOException;

  import java.text.NumberFormat;
15
  public class FileDepositer {

    private GraphArea board;
    private Runner runner;
20  private int milliFrameDelay = 20;
    private int milliMovieFrameDelay = 40;
    private int movieFrameRate = 1000/milliMovieFrameDelay;
    private double baseMass = 1.0; // nominally measured in kg but is actually arbitrary relative
        to other parameters
    private Random random;
25  private double boardDimension = 600;
  //  private FileReaderWriter rw;
    private double[] diameterArray;
```

```
    private BoardToMovie.ImageSourceStream movieSource;

30  public FileDepositer(boolean centerGravity) {
      board = new GraphArea((int)boardDimension,(int)boardDimension, centerGravity);
      PhysicsSimulator.setGravity(25);
      PhysicsSimulator.setFriction(0.025, 0.025);
      runner = new Runner(board, milliFrameDelay, true);
35    random = new Random(154);
//      rw = new FileReaderWriter();
    }

    public void reset(boolean centerGravity) {
40    reset(centerGravity, false);
    }

    public void reset(boolean centerGravity, boolean resetRandom) {
      board = new GraphArea((int)boardDimension,(int)boardDimension, centerGravity);
45    PhysicsSimulator.setGravity(25);
      PhysicsSimulator.setFriction(0.025, 0.025);
      runner = new Runner(board, milliFrameDelay, true);
      if(resetRandom) {
        random = new Random(154);
50    }
    }

    public void startMovie(String filename) {
      BoardToMovie movie = new BoardToMovie((int)boardDimension,(int)boardDimension, (float)
          movieFrameRate, BoardToMovie.createMediaLocator(filename));
55    new Thread(movie).start();
      BoardToMovie.ImageDataSource dataSource = movie.getImageDataSource();
      while(dataSource == null) {
        Thread.yield();
        dataSource = movie.getImageDataSource();
60    }
      movieSource = dataSource.getImageSourceStream();
      try {
        synchronized (movie) {
          while(!movie.isReadyToReceive()) {
65          movie.wait();
          }
        }
      } catch (InterruptedException ex) {
      }
70    System.out.println("movie is ready to receive images");
    }

    public void stopMovie() {
      movieSource.setEnded(true);
75  }

    /*
     * @param time: time in milliseconds that we want to step the board forward
     * @returns actual time stepped
80   */
    public int advance(int time) {
      int timeSoFar = 0;
      while(timeSoFar < time) {
        runner.runUpdate();
85      if(movieSource != null && timeSoFar % milliMovieFrameDelay == 0) {
          movieSource.consumeNewImage(board.getBoardImage((int)boardDimension, (int)boardDimension)
              );
```

```
            //movieSource.notify();
//          System.out.println("sent new movie frame");
          }
          timeSoFar += milliFrameDelay;
        }
        return timeSoFar;
      }

      /*
       * Adds a ball to the board at y = 100 and x chosen evenly from 100 to boardDimension-100
       * @param meanDiameter : The mean ball size to be added
       * @param stddev : the standard deviation in ball size in units of the meanSize i.e. 0.05 is
            0.05*meanSize
       * @requires : meanDiameter > 2*runner.getEpsilonSpace() which should be extremely small
       * @returns : actual diameter of the ball added
       */
      public double addRandomBallAtTop(double meanDiameter, double stddev, boolean colorize) {
        Vect pos = new Vect((boardDimension-200)*random.nextDouble()+100,100);
        Vect vel = new Vect(0,0);
        double diameter;
        if(stddev == 0) {
          diameter = meanDiameter-2*runner.getEpsilonSpace();
        } else {
          diameter = meanDiameter+meanDiameter*random.nextGaussian()*stddev-2*runner.getEpsilonSpace
              ();
        }
        if(colorize) {
          Color color = Color.GREEN;
          if(stddev != 0.0)
            color = new Color((float)Math.min(1.0,Math.exp((diameter-meanDiameter-stddev*meanDiameter
                )/(stddev*meanDiameter))),(float)Math.exp(-Math.abs(diameter-meanDiameter)/(stddev*
                meanDiameter)),(float)Math.min(1.0,Math.exp(-(diameter-meanDiameter+stddev*
                meanDiameter)/(stddev*meanDiameter))));
          board.addBall(new Ball(pos,vel,diameter,baseMass,color));
        } else {
//        board.addBall(new Ball(pos,vel,diameter,baseMass*Math.pow(diameter/2,3)/Math.pow(
       meanDiameter/2,3)));
          board.addBall(new Ball(pos,vel,diameter,baseMass));
        }
        return diameter;
      }

      /*
       * Adds a ball to the board at a distance radius from the point center
       * @param meanDiameter : The mean ball size to be added
       * @param stddev : the standard deviation in ball size in units of the meanSize i.e. 0.05 is
            0.05*meanSize
       * @requires : meanDiameter > 2*runner.getEpsilonSpace() which should be extremely small
       * @returns : actual diameter of the ball added
       */
      public double addRandomBallAtRadius(double meanDiameter, double stddev, Vect center, double
          radius, boolean colorize) {
        if(radius > center.x() || radius > center.y() || radius > boardDimension-center.x() || radius
            > boardDimension-center.y()) {
          System.out.println("Radius too large for specified center. Could attempt to add a ball off
              of board");
        }
        Vect pos = new Vect(new Angle(random.nextDouble()*2*Math.PI),radius);
        pos = pos.plus(center);
        Vect vel = new Vect(0,0);
        double diameter;
```

```
      if(stddev == 0) {
        diameter = meanDiameter-2*runner.getEpsilonSpace();
      } else {
        diameter = meanDiameter+meanDiameter*random.nextGaussian()*stddev-2*runner.getEpsilonSpace
            ();
      }
//      board.addBall(new Ball(pos,vel,diameter,baseMass*Math.pow(diameter/2,3)/Math.pow(
    meanDiameter/2,3)));
      if(colorize) {
        Color color = Color.GREEN;
        if(stddev != 0.0)
          color = new Color((float)Math.min(1.0,Math.exp((diameter-meanDiameter-stddev*meanDiameter
              )/(stddev*meanDiameter))),(float)Math.exp(-Math.abs(diameter-meanDiameter)/(stddev*
              meanDiameter)),(float)Math.min(1.0,Math.exp(-(diameter-meanDiameter+stddev*
              meanDiameter)/(stddev*meanDiameter))));
        board.addBall(new Ball(pos,vel,diameter,baseMass,color));
      } else {
        board.addBall(new Ball(pos,vel,diameter,baseMass));
      }
      return diameter;
    }

    public double addRandomBallInRect(Vect topLeft, Vect bottomRight, double meanDiameter, double
        stddev, boolean colorize, boolean relativeMasses) {
      return addRandomBallInRect(topLeft, bottomRight, meanDiameter, stddev, colorize,
          relativeMasses, baseMass);
    }

    /*
     * @returns The diameter of the ball that was added or 0.0 if the ball would have overlapped
         with another ball
     *       or been off of the board
     */
    public double addRandomBallInRect(Vect topLeft, Vect bottomRight, double meanDiameter, double
        stddev, boolean colorize, boolean relativeMasses, double mass) {
      Vect vel = new Vect(0,0);
      double diameter;
      if(stddev == 0) {
        diameter = meanDiameter-2*runner.getEpsilonSpace();
      } else {
        diameter = meanDiameter+meanDiameter*random.nextGaussian()*stddev-2*runner.getEpsilonSpace
            ();
      }
      Vect diff = bottomRight.minus(topLeft);
      if(diff.x()-diameter < 0 || diff.y()-diameter < 0) {
        System.out.println("Use a larger grating balls with diameter = " + diameter + " can't fit."
            );
      }
      Vect pos = topLeft.plus(new Vect((diff.x()-diameter)*random.nextDouble(), (diff.y()-diameter)
          *random.nextDouble()));
      if(overlapsBall(pos,diameter/2.0)) {
        return 0.0;
      }
      if(pos.x() < topLeft.x() || pos.x()+diameter > bottomRight.x() || pos.y() < topLeft.y() ||
          pos.y()+diameter > bottomRight.y()) {
        System.out.println("Position somehow fell outside of prescribed area");
        return 0.0;
      }
      //double mass = baseMass;
      if(relativeMasses) {
        mass = baseMass*Math.pow(diameter/2,3)/Math.pow(meanDiameter/2,3);
```

```
        }
        if(colorize) {
          Color color = Color.GREEN;
          if(stddev != 0.0)
            color = new Color((float)Math.min(1.0,Math.exp((diameter-meanDiameter-stddev*meanDiameter
                )/(stddev*meanDiameter))),(float)Math.exp(-Math.abs(diameter-meanDiameter)/(stddev*
                meanDiameter)),(float)Math.min(1.0,Math.exp(-(diameter-meanDiameter+stddev*
                meanDiameter)/(stddev*meanDiameter))));
          board.addBall(new Ball(pos,vel,diameter,mass,color));
        } else {
          board.addBall(new Ball(pos,vel,diameter,mass));
        }
        return diameter;
      }

      public boolean overlapsBall(Vect position, double radius) {
    //    ArrayList<Ball> balls = board.getBalls();
        Iterator<Ball> iter = board.getBalls().iterator();
        while(iter.hasNext()) {
          Ball otherBall = iter.next();
          if(otherBall.getPosition().minus(position).length() < otherBall.getRadius()+radius+2*runner
              .getEpsilonSpace()) {
            return true;
          }
        }
        return false;
      }

      public void makeSingleCluster() {
        int numballsxeven = 2;
        int numballsxodd = 3;
        double rowdistance = 15.0*Math.sin(Math.PI/3.0);
        int numballsy = 3;
        for(int i = 0; i < numballsy; i++) {
          for(int j = 0; i % 2 == 0 ? j < numballsxeven : j < numballsxodd; j++) {
            board.addBall(new Ball(new Vect(j*15.0+(i%2==0?7.5:0),rowdistance*i), new Vect(0,0),15.0,
                baseMass));
          }
        }
      }

      public void makeHexagonalLattice() {
        int numballsxeven = (int)(boardDimension/15.0);
        int numballsxodd = numballsxeven-1;
        double rowdistance = 15.0*Math.sin(Math.PI/3.0);
        int numballsy = (int)(boardDimension/rowdistance);
        for(int i = 0; i < numballsy; i++) {
          for(int j = 0; i % 2 == 0 ? j < numballsxeven : j < numballsxodd; j++) {
            board.addBall(new Ball(new Vect(j*15.0+(i%2==0?0:7.5),rowdistance*i), new Vect(0,0),15.0,
                baseMass));
          }
        }
      }

      public void changeSomeBalls(double fractionBad) {
        ArrayList<Ball> balls = board.getBalls();
        for(int i = 0; i < balls.size(); i++) {
          Ball b = balls.get(i);
          if(random.nextDouble() < fractionBad) {
            b.setMass(2.0); // denotes bad balls
            //leave ball black for non-radiating
```

```
      } else {
        b.setColor(Color.GREEN);
      }
    }
  }

  public double findRadiativeFraction() {
    ArrayList<Ball> balls = board.getBalls();
    int numRad = 0;
    int numInterior = 0;
    int numballsxeven = (int)(boardDimension/15.0);
    int numballsxodd = numballsxeven-1;
    double rowdistance = 15.0*Math.sin(Math.PI/3.0);
    int numballsy = (int)(boardDimension/rowdistance);
    //System.out.println("numballsy = " + numballsy);
    int rowNum = 0;
    int ballNum = 0;
    for(int i = 0; i < balls.size(); i++) {
      if(rowNum > 0 && ballNum > 0 && rowNum < numballsy-1 && ballNum < ((rowNum%2==0)?
          numballsxeven-1:numballsxodd-1)) {
        numInterior++;
        if(balls.get(i).getMass() != 2.0 && balls.get(i-1).getMass() != 2.0 && balls.get(i+1).
            getMass() != 2.0) {
          //calculate the position of other neighbors
          if(balls.get(i-numballsxeven).getMass() != 2.0 && balls.get(i-numballsxeven+1).getMass
              () != 2.0 &&
              balls.get(i+numballsxeven-1).getMass() != 2.0 && balls.get(i+numballsxeven).getMass
                  () != 2.0) {
            numRad++;
          }
        }
      }
      // keep track of row and column
      ballNum++;
      if(rowNum % 2 == 0) {
        if(ballNum == numballsxeven) {
          ballNum = 0;
          rowNum++;
          //System.out.println("rowNum = " + rowNum + ", i = " + i);
        }
      } else {
        if(ballNum == numballsxodd) {
          ballNum = 0;
          rowNum++;
          //System.out.println("rowNum = " + rowNum + ", i = " + i);
        }
      }
    }
    return (double)numRad/(double)numInterior;
  }

  public void refillDiameterArray() {
    ArrayList balls = board.getBalls();
    diameterArray = new double[balls.size()];
    for(int i = 0; i < balls.size(); i++) {
      diameterArray[i] = 2*((Ball)balls.get(i)).getRadius();
    }
  }

  public void colorizeBalls(double meanDiameter, double stddev) {
    ArrayList balls = board.getBalls();
```

```
        Iterator ballIter = balls.iterator();
        Ball ball;
300     if(stddev > 0.0) {
          while(ballIter.hasNext()) {
            ball = (Ball)ballIter.next();
//            ball.setColor(new Color((float)Math.exp(-Math.abs(2*ball.getRadius()-meanDiameter-
        stddev*meanDiameter)/(stddev*meanDiameter)),(float)Math.exp(-Math.abs(2*ball.getRadius()-
        meanDiameter)/(stddev*meanDiameter)),(float)Math.exp(-Math.abs(2*ball.getRadius()-
        meanDiameter+stddev*meanDiameter)/(stddev*meanDiameter))));
            ball.setColor(new Color((float)Math.min(1.0,Math.exp((2*ball.getRadius()-meanDiameter-
                stddev*meanDiameter)/(stddev*meanDiameter))),(float)Math.exp(-Math.abs(2*ball.
                getRadius()-meanDiameter)/(stddev*meanDiameter)),(float)Math.min(1.0,Math.exp(-(2*
                ball.getRadius()-meanDiameter+stddev*meanDiameter)/(stddev*meanDiameter)))));
305       }
        } else {
          while(ballIter.hasNext()) {
            ball = (Ball)ballIter.next();
            ball.setColor(Color.GREEN);
310       }
        }
      }

      public void saveBoard(String filename) {
315     FileReaderWriter.saveFile(filename,board,true);
      }

      public void loadBoard(String filename) {
        board.loadGraphArea(FileReaderWriter.loadFile(filename));
320   }

      public void saveBoardImage(String filename) {
        try {
          board.saveBoardImage(filename,(int)boardDimension,(int)boardDimension);
325     } catch (IOException ex) {
          System.out.println("Saving board image failed: " + ex.getMessage());
        }
      }

330   /*
       * Writes all the values in array to the file filename seperated by spaces and ending with a
           new line
       * or prints failure message to std output.
       */
      public void saveArray(String filename, String linePrefix, double[] array) {
335     try {
          FileWriter writ = new FileWriter(filename,true); // append to current file
          BufferedWriter writer = new BufferedWriter(writ);
          if(linePrefix != null && linePrefix.length() > 0) {
            writer.write(linePrefix);
340         writer.write(' ');
          }
          for(int i = 0; i < array.length; i++) {
            writer.write(Double.toString(array[i]));
            if(i < array.length - 1) {
345           writer.write(' ');
            }
          }
          writer.newLine();
          writer.close();
350     } catch (IOException ex) {
          System.out.println("Saving double array in file " + filename + " failed.");
```

```
    }
  }

  /*
   * Writes all the values in array to the file filename seperated by spaces and ending with a
       new line
   * or prints failure message to std output.
   */
  public void saveArrayLineSepOverwrite(String filename, double[] array) {
    try {
      FileWriter writ = new FileWriter(filename); // overwrite current file
      BufferedWriter writer = new BufferedWriter(writ);
      for(int i = 0; i < array.length; i++) {
        writer.write(Double.toString(array[i]));
        writer.newLine();
      }
      writer.close();
    } catch (IOException ex) {
      System.out.println("Saving double array in file " + filename + " failed.");
    }
  }

  /*
   * Prints the elements of array1 and array2 side by side to file filname with 2 columns
       delimited by a space
   * @requires array1.length == array2.length
   * @effects overwrites existing file
   */
  public void saveArraysAsColumns(String filename, double[] array1, double[] array2) {
    try {
      FileWriter writ = new FileWriter(filename); // overwrite current file
      BufferedWriter writer = new BufferedWriter(writ);
      for(int i = 0; i < array1.length; i++) {
        writer.write(Double.toString(array1[i]));
        writer.write('_');
        writer.write(Double.toString(array2[i])),
        writer.newLine();
      }
      writer.close();
    } catch (IOException ex) {
      System.out.println("Saving double array in file " + filename + " failed.");
    }
  }

  /*
   * Calculates the packing parameters of the board in file filename in the layers that are
       distanceFromBottom
   * @param filename : name of file that contains ball information in full board format
   * @param distanceFromBottom : distance in pixels from the bottom that should be considered in
       the packing calculations
   * @modifies board
   */
  public double[] determinePacking(String filename, double distanceFromBottom, double depth,
      double distanceFromSides) {
    loadBoard(filename);
    return determinePacking(distanceFromBottom, depth, distanceFromSides);
  }

  public double[] determinePacking(double distanceFromBottom, double depth, double
      distanceFromSides) {
    return determinePacking(new Rectangle2D.Double(distanceFromSides,boardDimension-
```

```
        distanceFromBottom, boardDimension-2*distanceFromSides, depth));
    }

    public double[] determinePacking(Vect center, double radius) {
      return determinePacking(new Ellipse2D.Double(center.x()-radius, center.y()-radius, 2*radius, 2*
          radius));
    }

    public double[] determinePacking(String filename, Vect center, double radius) {
      loadBoard(filename);
      return determinePacking(new Ellipse2D.Double(center.x()-radius, center.y()-radius, 2*radius, 2*
          radius));
    }

    public double[] determinePacking(Shape bounds) {
      ArrayList balls = board.getBalls();
      double excessSeparation = 0;
      double averageSpace = 0;
      double averageNeighbors = 0;
      double totalBalls = 0;
      Iterator currentBalls = balls.iterator();
      while(currentBalls.hasNext()) {
        Ball ball = (Ball)currentBalls.next();
        if(!bounds.contains(ball.getCenter().toPoint2D()))
          continue;
//        if(ball.getCenter().y() < boardDimension-distanceFromBottom)
//          continue;
//        if(ball.getCenter().y() > boardDimension-distanceFromBottom+depth) // if depth >
    distanceFromBottom all balls are in
//          continue;
//        if(ball.getCenter().x() < distanceFromSides || ball.getCenter().x() > boardDimension-
    distanceFromSides)
//          continue;
        totalBalls++;
        Iterator otherBalls = balls.iterator();
        int neighbors = 0;
        while(otherBalls.hasNext()) {
          Ball otherBall = (Ball)otherBalls.next();
          if(ball.equals(otherBall))
            continue;
          Vect separation = ball.getCenter().minus(otherBall.getCenter());
          // only works for reasonable size distributions (if some radii are more than twice others
              then it is incorrect)
          if(separation.length() > ball.getRadius()+2*otherBall.getRadius())
            continue;
          neighbors++;
          excessSeparation += Math.pow(separation.length()-(ball.getRadius()+otherBall.getRadius())
              ,2);
          averageSpace += separation.length()-(ball.getRadius()+otherBall.getRadius());
//          if(separation.length()-(ball.getRadius()+otherBall.getRadius()) > ball.getRadius()) {
//            System.out.println("Error: Two balls thought to be neighbors is more than the radius
    of a ball away.");
//          }
        }
        averageNeighbors += neighbors;
      }
      double rmsPerBall = Math.sqrt(excessSeparation/(averageNeighbors)); // dividing by
          averageNeighbors because we're over counting
      averageSpace /= averageNeighbors; // again averageNeighbors was the number of times we summed
          the distances
      averageNeighbors /= totalBalls;
```

```
        System.out.println("The RMS spacing per ball is " + rmsPerBall + " and the average spacing
            per ball is " + averageSpace + " on " + totalBalls + " balls each having " +
            averageNeighbors + " neighbors on average.");
        return new double[] {rmsPerBall, averageSpace, totalBalls, averageNeighbors};
    }

    public double[] determinePackingInterior() {
        ArrayList balls = board.getBalls();
        double excessSeparation = 0;
        double averageSpace = 0;
        double averageNeighbors = 0;
        double totalBalls = 0;
        Iterator currentBalls = balls.iterator();
        while(currentBalls.hasNext()) {
            Ball ball = (Ball)currentBalls.next();
//          if(!bounds.contains(ball.getCenter().toPoint2D()))
//            continue;
//          if(ball.getCenter().y() < boardDimension-distanceFromBottom)
//            continue;
//          if(ball.getCenter().y() > boardDimension-distanceFromBottom+depth) // if depth >
        distanceFromBottom all balls are in
//            continue;
//          if(ball.getCenter().x() < distanceFromSides || ball.getCenter().x() > boardDimension-
        distanceFromSides)
//            continue;
//          totalBalls++;
            Iterator otherBalls = balls.iterator();
            int neighbors = 0;
            double excessSeparationTemp = 0;
            double averageSpaceTemp = 0;
            while(otherBalls.hasNext()) {
                Ball otherBall = (Ball)otherBalls.next();
                if(ball.equals(otherBall))
                    continue;
                Vect separation = ball.getCenter().minus(otherBall.getCenter());
                // only works for reasonable size distributions (if some radii are more than twice others
                    then it is incorrect)
                if(separation.length() > ball.getRadius()+1.5*otherBall.getRadius())
                    continue;
                neighbors++;
                excessSeparationTemp += Math.pow(separation.length()-(ball.getRadius()+otherBall.
                    getRadius()),2);
                averageSpaceTemp += separation.length()-(ball.getRadius()+otherBall.getRadius());
//              if(separation.length()-(ball.getRadius()+otherBall.getRadius()) > ball.getRadius()) {
//                System.out.println("Error: Two balls thought to be neighbors is more than the radius
        of a ball away.");
//              }
            }
            if(neighbors > 4) {
                totalBalls++;
                averageNeighbors += neighbors;
                excessSeparation += excessSeparationTemp;
                averageSpace += averageSpaceTemp;
            }
        }
        double rmsPerBall = Math.sqrt(excessSeparation/(averageNeighbors)); // dividing by
            averageNeighbors because we're over counting
        averageSpace /= averageNeighbors; // again averageNeighbors was the number of times we summed
            the distances
        averageNeighbors /= totalBalls;
        System.out.println("The RMS spacing per ball is " + rmsPerBall + " and the average spacing
```

```
          per ball is " + averageSpace + " on " + totalBalls + " balls each having " +
          averageNeighbors + " neighbors on average.");
510     return new double[] {rmsPerBall, averageSpace, totalBalls, averageNeighbors};
      }

      /*
       * Method to determine the amount of space filled by balls within a certain radius of
515    * the center of the board.
       */
      public double determineFill(double radius) {
        int num = 1000000;
        int withinRadius = 0;
520     int inBall = 0;
        Random rand = new Random(83);
        Object[] balls = board.getBalls().toArray();
        for(int i = 0; i < num; i++) {
          double x = radius*(2*rand.nextDouble()-1);
525       double y = radius*(2*rand.nextDouble()-1);
          if(x*x+y*y < radius*radius) {
            withinRadius++;
            x += boardDimension/2;
            y += boardDimension/2;
530 //          System.out.println("x = " + x);
    //          System.out.println("y = " + y);
            for(int j = 0; j < balls.length; j++) {
              Vect center = ((Ball)balls[j]).getCenter();
    //            System.out.println("Ball center = " + center);
535           double ballRadius = ((Ball)balls[j]).getRadius();
    //            System.out.println("Ball radius = " + ballRadius);
    //            System.out.println("distance from center squared = " + (Math.pow(center.x()-x,2)+Math
     .pow(center.y()-y,2)));
    //            System.out.println("ball radius squared = " + ballRadius*ballRadius);
              if((Math.pow(center.x()-x,2)+Math.pow(center.y()-y,2)) < ballRadius*ballRadius) {
540 //              System.out.println("incrementing in ball");
                inBall++;
                break;
              }
            }
545       }
        }
    //    System.out.println("found fill " + inBall/withinRadius + " with " + withinRadius + " inside
        the radius");
        return (double)inBall/(double)withinRadius;
      }
550

      /*
       * Method to determine the amount of space filled by balls within a certain radius of
       * the center of the board.
       * @param : sorted array of increasing radii. The fill will be calculated within each radius
           and returned
555    */
      public double[] determineFillWithinRadius(double[] radii) {
        int num = 1000000;
        int[] withinRadius = new int[radii.length];
        int[] inBall = new int[radii.length];
560     Random rand = new Random(83);
        Object[] balls = board.getBalls().toArray();
        double radius = radii[radii.length-1];
        for(int i = 0; i < num; i++) {
          double x = radius*(2*rand.nextDouble()-1);
565       double y = radius*(2*rand.nextDouble()-1);
```

```
        int firstIndex = Arrays.binarySearch(radii,Math.sqrt(x*x+y*y));
        if (firstIndex < 0) { // unless x^2+y^2 exactly matches one of the elements it will be less
            than zero
          firstIndex = -(firstIndex+1);
        }
570     if(firstIndex != radii.length) {
          for(int k = firstIndex; k < radii.length; k++) {
            withinRadius[k]++;
          }
          x += boardDimension/2;
575       y += boardDimension/2;
    //      System.out.println("x = " + x);
    //      System.out.println("y = " + y);
          for(int j = 0; j < balls.length; j++) {
            Vect center = ((Ball)balls[j]).getCenter();
580 //        System.out.println("Ball center = " + center);
            double ballRadius = ((Ball)balls[j]).getRadius();
    //        System.out.println("Ball radius = " + ballRadius);
    //        System.out.println("distance from center squared = " + (Math.pow(center.x()-x,2)+Math
        .pow(center.y()-y,2)));
    //        System.out.println("ball radius squared = " + ballRadius*ballRadius);
585         if((Math.pow(center.x()-x,2)+Math.pow(center.y()-y,2)) < ballRadius*ballRadius) {
    //        System.out.println("incrementing in ball");
              for(int k = firstIndex; k < radii.length; k++) {
                inBall[k]++;
              }
590           break;
            }
          }
        }
      }
595 //  System.out.println("found fill " + inBall/withinRadius + " with " + withinRadius + " inside
      the radius");
      double[] ret = new double[radii.length];
      for(int i = 0; i < radii.length; i++) {
        ret[i] = (double)inBall[i]/(double)withinRadius[i];
      }
600   return ret;
    }

    public void testDetermineFill() {
      board.addBall(new Ball(new Vect(boardDimension/2-15.0/2.0,boardDimension/2-15.0/2.0), new
          Vect(0,0),15,1));
605   if(determineFill(3) != 1.0) {
        System.out.println("Found points that weren't inside a ball when all should have been
            within radius 3.");
        return;
      }
      if(determineFill(7.5) != 1.0) {
610     System.out.println("Found points that weren't inside a ball when all should have been
            within radius 7.5.");
        return;
      }
      double area30 = determineFill(15);
      double actualArea30 = Math.PI*15*7.5/(Math.PI*30*15);
615   System.out.println("Area calculated by determine fill for one ball with radius 15 in area
          with radius 30 =" + area30 + "analytically the area =" + actualArea30);
    }

    public void testDetermineFillWithinRadius() {
      board.addBall(new Ball(new Vect(boardDimension/2-15.0/2.0,boardDimension/2-15.0/2.0), new
```

```
            Vect(0,0),15,1));
620       double[] dummy = {3};
    //    dummy[0] = 3;
          if(determineFillWithinRadius(dummy)[0] != 1.0) {
            System.out.println("Found points that weren't inside a ball when all should have been
                within radius 3.");
            return;
625       }
          dummy[0] = 7.5;
          if(determineFillWithinRadius(dummy)[0] != 1.0) {
            System.out.println("Found points that weren't inside a ball when all should have been
                within radius 7.5.");
            return;
630       }
          dummy = new double[]{3, 7.5, 15, 30};
          double[] area = determineFillWithinRadius(dummy);
          double[] actualArea = {1,1,Math.PI*7.5*7.5/(Math.PI*15*15), Math.PI*7.5*7.5/(Math.PI*30*30)};
    //    double actualArea15 = Math.PI*7.5*7.5/(Math.PI*15*15);
635 //    double actualArea30 = Math.PI*7.5*7.5/(Math.PI*30*30);
          for(int i = 0; i < dummy.length; i++) {
            System.out.println("Using determineFillWithinRadius for one ball with radius 7.5 inside
                radius = " + dummy[i] + " area calculated = " + area[i] + " analytically the area = "
                 + actualArea[i]);
          }
        }
640
        public double determinePearsonSpatialCorrelation() {
          Object[] balls = (Object[])board.getBalls().toArray();
          double xsum = 0;
          double ysum = 0;
645       double xxsum = 0;
          double xysum = 0;
          double yysum = 0;
          int numdata = 0;
          double[] xs = new double[(balls.length*balls.length-balls.length)/2];
650       double[] ys = new double[(balls.length*balls.length-balls.length)/2];
          for(int i = 0; i < balls.length; i++) {
            Vect center1 = ((Ball)balls[i]).getCenter();
            double radius1 = ((Ball)balls[i]).getRadius();
            for(int j = 0; j < i; j++) {
655           Vect center2 = ((Ball)balls[j]).getCenter();
              double radius2 = ((Ball)balls[j]).getRadius();
              double x = center1.minus(center2).length()-(radius1+radius2);
              double y = Math.abs(radius1-radius2);
              xs[numdata] = x;
660           ys[numdata] = y;
              numdata++;
              xsum += x;
              ysum += y;
              xxsum += x*x;
665           xysum += x*y;
              yysum += y*y;
    //          if(numdata == 1) {
    //            System.out.println("x[0] = " + x);
    //            System.out.println("xxsum[0] = " + xxsum);
670 //          }
            }
          }
          System.out.println("numdata = " + numdata);
    //    System.out.println("average distance = " + xsum/numdata);
675 //    System.out.println("average radius difference = " + ysum/numdata);
```

```
        System.out.println("r = " + xysum/Math.sqrt(xxsum*yysum));
        System.out.println("matlab p = " + (xysum-1.0/numdata*xsum*ysum)/(Math.sqrt((xxsum-xsum*xsum/
            numdata)*(yysum-ysum*ysum/numdata))));
        //System.out.println("other mathworld r = " + (xysum-numdata*xsum*ysum)/Math.sqrt((xxsum-
            numdata*xsum*xsum)*(yysum-numdata*ysum*ysum))); wrong
//      saveArraysAsColumns("xydata.dat",xs,ys);
        double matlabValue = 0;
        for(int i = 0; i < numdata; i++) {
          double x0 = (xs[i] - xsum/numdata)/Math.sqrt(xxsum-xsum*xsum/numdata);
          double y0 = (ys[i] - ysum/numdata)/Math.sqrt(yysum-ysum*ysum/numdata);
          matlabValue += x0*y0;
        }
        System.out.println("other computation of matlab value = " + matlabValue);
        return (numdata*xysum-xsum*ysum)/Math.sqrt((numdata*xxsum-xsum*xsum)*(numdata*yysum-ysum*ysum
            ));
      }

      public double determineLuminosity(double fractionBad, int totalExcitons) {
        ArrayList<Ball> balls = board.getBalls();
        Iterator<Ball> iterator = balls.iterator();
        ArrayList<QuantumDot> dots = new ArrayList<QuantumDot>(balls.size());
        while(iterator.hasNext()) {
          Ball ball = iterator.next();
          boolean goodness = true;
          if(random.nextDouble() < fractionBad) {
            goodness = false; // denotes bad balls
            //set ball black for non-radiating
            ball.setColor(Color.BLACK);
          } else {
            // if balls uncolored previously set good ones to green
            if(ball.getCurrColor().equals(Color.BLACK))
              ball.setColor(Color.GREEN);
          }
          dots.add(new QuantumDot(ball,goodness,false));
        }
        int numRadiated = 0;
        for(int i = 0; i < totalExcitons; i++) {
          int exciteIndex = (int)Math.floor(balls.size()*random.nextDouble());
          dots.get(exciteIndex).setExcited(true);
          boolean radiated = determineRadiated(dots, exciteIndex);
          if(radiated) numRadiated++;
        }
        return (double)numRadiated/(double)totalExcitons;
      }

      /*
       * Advances time for an exciton at index using forster transfer
       * @returns : true if the exciton radiatively relaxed, false if not
       */
      public boolean determineRadiated(ArrayList<QuantumDot> dots, int intialIndex) {
        int index = intialIndex;
        double radLife = 26.0; // in nanoseconds
        double timestep = radLife/100.0;
        double radProb = timestep/radLife; // per step
        //System.out.println("rad prob = " + radProb);
        double forsterRadius = 15.0;//4.0; //in nanometers
        while(true) {
          QuantumDot currentDot = dots.get(index);
          if(!currentDot.isGood()) {
            return false;
          }
```

```
        ArrayList<QuantumDot> possibleTransfers = new ArrayList<QuantumDot>();
735     ArrayList<Double> transferProbabilities = new ArrayList<Double>();
        double totalTransferProb = 0;
        Iterator<QuantumDot> iterator = dots.iterator();
        while(iterator.hasNext()) {
          QuantumDot otherDot = iterator.next();
740       double dist = currentDot.getBall().getCenter().minus(otherDot.getBall().getCenter()).
              length();
          if(dist == 0.0) {
            continue;
          }
          double transRate = (1/radLife)*Math.pow(forsterRadius/dist,6.0);
745       double probTrans = timestep*transRate;
          if(random.nextDouble() < probTrans) {
            // transfer can occur
            possibleTransfers.add(otherDot);
            transferProbabilities.add(probTrans);
750         totalTransferProb += probTrans;
          }
        }
        //System.out.println("total transfer prob without radiate = " + totalTransferProb);
        boolean radiatePossible = false;
755     if(random.nextDouble() < radProb) {
          //System.out.println("here");
          totalTransferProb += radProb; // add probability of radiation
          radiatePossible = true;
        }
760     if(totalTransferProb > 0) {
          double choose = random.nextDouble()*totalTransferProb;
//          check if should radiate: if we get to end of transfer list without
          // transfering and totalTransferProb has been incremented by radProb
          if(radiatePossible && choose > totalTransferProb-radProb) {
765         return true;
          }
          // now check if it transfers to another dot
//          System.out.println("total transfer prob = " + totalTransferProb + " choose = " + choose
      );
          double runningTotal = 0.0;
770       iterator = possibleTransfers.iterator();
          Iterator<Double> probIterator = transferProbabilities.iterator();
          while(iterator.hasNext()) {
            QuantumDot transferDot = iterator.next();
            runningTotal += probIterator.next();
775         if(choose < runningTotal) {
              // then transfer to transferDot
              currentDot.setExcited(false);
              transferDot.setExcited(true);
              //if(transferDot.isGood()) {
780           index = dots.indexOf(transferDot);
              break;
              //} else {
              //  return false;
              //}
785         }
          }
        }
      }
    }
790
    public double[][][] runLinearGraphFillWithinRadius(double maxRadius, int numPoints) {
      double incr = maxRadius/(double)numPoints;
```

```
        double[] radialAxis = new double[numPoints];
        for(int i = 0; i < numPoints; i++) {
795       radialAxis[i] = incr*(i+1);
        }
        double[] values = determineFillWithinRadius(radialAxis);
        return new double[][]{radialAxis, values};
      }
800
      public void runUniform(double mean, int numLines) {
    //    double mean = 15;
    //    double dev = 0;
    //    int numBalls = (int)(numLines*600/mean-Math.floor(numLines/2));
805 //    for(int i = 0; i < numBalls; i++) {
    //      addRandomBallAtTop(mean,dev);
    //      advance(10000); // advance for 10 seconds
    //    }
    //    advance(100000); // advance for 100 seconds at the end to settle balls
810 //    System.out.println(determinePacking(mean*(numLines-1)));
        runRandom(mean, 0, numLines);
      }

      public double[] runRandom(double meanDiameter, double stddev, int numLines) {
815     int numBalls = (int)(numLines*boardDimension/meanDiameter-Math.floor(numLines/2));
        System.out.println("numBalls = " + numBalls);
        for(int i = 0; i < numBalls; i++) {
          addRandomBallAtTop(meanDiameter,stddev,true);
          advance(5000); // advance for 10 seconds
820     }
        System.out.println("Done introducing balls. Now running till settled.");
        advance(100000); // advance for 100 seconds at the end to settle balls
        return determinePacking(meanDiameter*(numLines-1),meanDiameter*(numLines-2),2*meanDiameter);
      }
825
      /*
       * @requires depositer must be created as center of gravity to be correct
       */
      public double[] runRandomCenterGravity(double meanDiameter, double stddev, int numBalls) {
830     /*
        PhysicsSimulator.setFriction(0.1,0.1);
        double packingInPlane = 1.0/6.0*Math.PI*Math.sqrt(3);//0.74://Math.PI/4.0;
        double radius = Math.sqrt(packingInPlane*numBalls)*meanDiameter/2.0; // average radius filled
              by hcp
        System.out.println("final radius = " + radius);
835     diameterArray = new double[numBalls];
        for(int i = 0; i < numBalls; i++) {
          radius = Math.sqrt(packingInPlane*i)*meanDiameter/2.0;
          diameterArray[i] = addRandomBallAtRadius(meanDiameter,stddev,PhysicsSimulator.
               getCenterOfGravity(),radius+3*meanDiameter+stddev*meanDiameter, true);
          advance(500); // advance 1/2 second
840     }
        System.out.println("Done introducing balls. Now running till settled.");
        advance(15000); // advance for 100 seconds at the end to settle balls
        return determinePacking(PhysicsSimulator.getCenterOfGravity(),radius-2*meanDiameter);
        */
845     PhysicsSimulator.setFriction(0.1,0.1);
        return runRandomCenter(meanDiameter,stddev,numBalls,PhysicsSimulator.getCenterOfGravity());
      }

      /*
850    * @requires depositer must be created as center of gravity to be correct
       */
```

```
    public double[] runRandomCenter(double meanDiameter, double stddev, int numBalls, Vect center)
        {
      double packingInPlane = 1.0/6.0*Math.PI*Math.sqrt(3);//0.74;//Math.PI/4.0;
      double radius = Math.sqrt(packingInPlane*numBalls)*meanDiameter/2.0; // average radius filled
          by hcp
      System.out.println("final radius = " + radius);
      diameterArray = new double[numBalls];
      for(int i = 0; i < numBalls; i++) {
        radius = Math.sqrt(packingInPlane*i)*meanDiameter/2.0;
        diameterArray[i] = addRandomBallAtRadius(meanDiameter,stddev,center,radius+3*meanDiameter+
            stddev*meanDiameter, true);
        advance(500); // advance 1/2 second
      }
      System.out.println("Done introducing balls. Now running till settled.");
      advance(15000); // advance for 100 seconds at the end to settle balls
      return determinePacking(center,radius-2*meanDiameter);
    }

    public double[] runRandomAttraction(double meanDiameter, double stddev, int numBalls, boolean
        relativeMasses) {
//    Vect center = new Vect(boardDimension/2,boardDimension/2);
//    double packingInPlane = 1.0/6.0*Math.PI*Math.sqrt(3);//0.74;//Math.PI/4.0;
      //double radius = 3*Math.sqrt(packingInPlane*numBalls)*meanDiameter/2.0;
      // place all balls uniformly on the board (addBallAt doesn't allow ball-ball overlaps or off
          of board)
      diameterArray = new double[numBalls];
      int numAdded = 0;
      while(numAdded < numBalls) {
//      double x = random.nextDouble()*radius;
//      double y = random.nextDouble()*radius;
//      double x = random.nextDouble()*(boardDimension); // could try to avoid adding off of
        board
//      double y = random.nextDouble()*(boardDimension);
        //if(x*x+y*y < radius*radius) {
//      Vect pos = new Vect(x,y);
        //pos = pos.plus(center);
        diameterArray[numAdded] = addRandomBallInRect(Vect.ZERO, new Vect(boardDimension,
            boardDimension),meanDiameter,stddev,true,relativeMasses);
        if(diameterArray[numAdded] > 0) { // if ball actually added
          numAdded++;
          advance(1000);
        }
        //}
      }
      System.out.println("Done introducing balls. Now running till settled.");
      advance(100000); // advance for 100 seconds at the end to settle balls
//    return determinePacking(center,radius-2*meanDiameter);
//    return determinePacking(new Vect(boardDimension/2,boardDimension/2),boardDimension);
      return determinePackingInterior();
    }

    public double[] runRandomAttractionGrating(double meanDiameter, double stddev, int numBalls,
        double gratingWidth, boolean relativeMasses) {
      // add grating wall first using a GeneralGizmo
      ArrayList list = new ArrayList(2);
      ArrayList<Reflector> perimeter = new ArrayList<Reflector>(1);
      perimeter.add(new Reflector(new LineSegment(0,0,boardDimension,0),null));
      list.add(perimeter);
      list.add(new Rectangle(0,0,(int)Math.floor(boardDimension),0));
      board.addGizmo(new GeneralGizmo(new Vect(0,gratingWidth), list));
      diameterArray = new double[numBalls];
```

```
905       int numAdded = 0;
          while(numAdded < numBalls) {
      //      double x = random.nextDouble()*radius;
      //      double y = random.nextDouble()*radius;
      //      double x = random.nextDouble()*(boardDimension); // could try to avoid adding off of
           board
910   //      double y = random.nextDouble()*(boardDimension);
            //if(x*x+y*y < radius*radius) {
      //      Vect pos = new Vect(x,y);
            //pos = pos.plus(center);
            diameterArray[numAdded] = addRandomBallInRect(Vect.ZERO, new Vect(boardDimension,
                gratingWidth),meanDiameter,stddev,true,relativeMasses);
915         if(diameterArray[numAdded] > 0) { // if ball actually added
              numAdded++;
              advance(1000);
            }
            //}
920       }
          System.out.println("Done introducing balls. Now running till settled.");
          advance(100000); // advance for 100 seconds at the end to settle balls
      //    return determinePacking(center,radius-2*meanDiameter);
      //    return determinePacking(new Vect(boardDimension/2,boardDimension/2),boardDimension);
925       return determinePackingInterior();
        }

        public double[] runRandomAttractionBimodal(double meanDiameter1, double stddev1, double
            meanDiameter2, double stddev2, double meanMass2, int numBalls, boolean relativeMasses) {
      //    Vect center = new Vect(boardDimension/2,boardDimension/2);
930   //    double packingInPlane = 1.0/6.0*Math.PI*Math.sqrt(3);//0.74;//Math.PI/4.0;
          //double radius = 3*Math.sqrt(packingInPlane*numBalls)*meanDiameter/2.0;
          // place all balls uniformly on the board (addBallAt doesn't allow ball-ball overlaps or off
               of board)
          boolean type1;
          diameterArray = new double[numBalls];
935       int numAdded = 0;
          int numtype1 = 0;
          int numtype2 = 0;
          while(numAdded < numBalls) {
      //      double x = random.nextDouble()*radius;
940   //      double y = random.nextDouble()*radius;
      //      double x = random.nextDouble()*(boardDimension); // could try to avoid adding off of
           board
      //      double y = random.nextDouble()*(boardDimension);
            //if(x*x+y*y < radius*radius) {
      //      Vect pos = new Vect(x,y);
945         //pos = pos.plus(center);
            type1 = random.nextBoolean();
            if(type1) {
              diameterArray[numAdded] = addRandomBallInRect(Vect.ZERO, new Vect(boardDimension,
                  boardDimension),meanDiameter1,stddev1,true,relativeMasses);
            } else {
950           diameterArray[numAdded] = addRandomBallInRect(Vect.ZERO, new Vect(boardDimension,
                  boardDimension),meanDiameter2,stddev2,false,relativeMasses,meanMass2);
            }
            if(diameterArray[numAdded] > 0) { // if ball actually added
              numAdded++;
              if(type1)
955             numtype1++;
              else
                numtype2++;
              advance(1000);
```

```
        }
960     //}
      }
      System.out.println("Done introducing balls. Now running till settled.");
      System.out.println("Added " + numtype1 + " balls of type 1 and " + numtype2 + " balls of type
          2");
      advance(100000); // advance for 100 seconds at the end to settle balls
965 //    return determinePacking(center, radius-2*meanDiameter);
    //    return determinePacking(new Vect(boardDimension/2, boardDimension/2), boardDimension);
      return determinePackingInterior();
    }

970   public void runCentralSuite(int number) {
      System.out.println("Starting central suite");

      reset(true);

975   System.out.println("Running uniform, gravity at center test");
      double[] packingArray = runRandomCenterGravity(15,0,number);
      saveArray("centralpacking.txt", "0.0", packingArray);
    //    saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);
      saveBoard("uniformcentral");
980   saveBoardImage("uniformcentral.jpg");

      reset(true);

      System.out.println("Running 15% std dev gravity at center test");
985   packingArray = runRandomCenterGravity(15,0.15,number);
      saveArray("centralpacking.txt", "0.15", packingArray);
      saveArrayLineSepOverwrite("random15percentdiameters.txt", diameterArray);
      saveBoard("random15percentcentral");
      saveBoardImage("random15percentcentral.jpg");
990
      reset(true);

      System.out.println("Running 5% std dev, gravity at center test");
      packingArray = runRandomCenterGravity(15,0.05,number);
995   saveArray("centralpacking.txt", "0.05", packingArray);
      saveArrayLineSepOverwrite("random5percentdiameters.txt", diameterArray);
      saveBoard("random5percentcentral");
      saveBoardImage("random5percentcentral.jpg");

1000  reset(true);

      System.out.println("Running 10% std dev, gravity at center test");
      packingArray = runRandomCenterGravity(15,0.10,number);
      saveArray("centralpacking.txt", "0.10", packingArray);
1005  saveArrayLineSepOverwrite("random10percentdiameters.txt", diameterArray);
      saveBoard("random10percentcentral");
      saveBoardImage("random10percentcentral.jpg");
    }

1010  public void colorizeCentralSuite() {
      loadBoard("uniformcentral");
      colorizeBalls(15,0);
      saveBoard("uniformcentralcolor");
      saveBoardImage("uniformcentral.jpg");
1015  refillDiameterArray();
      saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);

      loadBoard("random5percentcentral");
```

```
        colorizeBalls(15,0.05);
        saveBoard("random5percentcentralcolor");
        saveBoardImage("random5percentcentral.jpg");
        refillDiameterArray();
        saveArrayLineSepOverwrite("random5percentdiameters.txt", diameterArray);

        loadBoard("random10percentcentral");
        colorizeBalls(15,0.10);
        saveBoard("random10percentcentralcolor");
        saveBoardImage("random10percentcentral.jpg");
        refillDiameterArray();
        saveArrayLineSepOverwrite("random10percentdiameters.txt", diameterArray);

        loadBoard("random15percentcentral");
        colorizeBalls(15,0.15);
        saveBoard("random15percentcentralcolor");
        saveBoardImage("random15percentcentral.jpg");
        refillDiameterArray();
        saveArrayLineSepOverwrite("random15percentdiameters.txt", diameterArray);
    }

    public void calculateCentralSuiteFill() {
        loadBoard("uniformcentral");
        double fill = determineFill(100);
        System.out.println("Fill factor for uniform size balls = " + fill);

        loadBoard("random5percentcentral");
        fill = determineFill(100);
        System.out.println("Fill factor for random distribution with 5% standard deviation = " + fill
            );

        loadBoard("random10percentcentral");
        fill = determineFill(100);
        System.out.println("Fill factor for random distribution with 10% standard deviation = " +
            fill);

        loadBoard("random15percentcentral");
        fill = determineFill(100);
        System.out.println("Fill factor for random distribution with 15% standard deviation = " +
            fill);
    }

    public void calculateCentralSuiteFillWithinRadius() {
        loadBoard("uniformcentral");
        double[][] function = runLinearGraphFillWithinRadius(100,20);
        saveArraysAsColumns("uniformcentralfillradius.txt", function[0], function[1]),

        loadBoard("random5percentcentral");
        function = runLinearGraphFillWithinRadius(100,20);
        saveArraysAsColumns("random5percentfillradius.txt", function[0], function[1]);

        loadBoard("random10percentcentral");
        function = runLinearGraphFillWithinRadius(100,20);
        saveArraysAsColumns("random10percentfillradius.txt", function[0], function[1]);

        loadBoard("random15percentcentral");
        function = runLinearGraphFillWithinRadius(100,20);
        saveArraysAsColumns("random15percentfillradius.txt", function[0], function[1]);
    }

    public void calculateCentralSuiteSpatialCorrelation() {
```

```
        loadBoard("uniformcentral");
        double corrcoeff = determinePearsonSpatialCorrelation();
        System.out.println("The correlation coefficient for a uniform distribution = " + corrcoeff);

        loadBoard("random5percentcentral");
        corrcoeff = determinePearsonSpatialCorrelation();
        System.out.println("The correlation coefficient for a 5% std dev distribution = " + corrcoeff
            );

        loadBoard("random10percentcentral");
        corrcoeff = determinePearsonSpatialCorrelation();
        System.out.println("The correlation coefficient for a 10% std dev distribution = " +
            corrcoeff);

        loadBoard("random15percentcentral");
        corrcoeff = determinePearsonSpatialCorrelation();
        System.out.println("The correlation coefficient for a 15% std dev distribution = " +
            corrcoeff);
    }

    public void testMovie() {
        reset(true);

        System.out.println("Running uniform, gravity at center test with movie");
        startMovie("random15percentmovie2.mov");
        runRandomCenterGravity(15,0.15,100);
        stopMovie();
//      saveArray("centralpacking.txt", "0.0", packingArray);
////      saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);
//      saveBoard("uniformcentral");
//      saveBoardImage("uniformcentral.jpg");
    }

    public void makeCentralSuiteMovies(int number) {
        reset(true);

        System.out.println("Running uniform, gravity at center test with movie");
        startMovie("uniformmovie.mov");
        runRandomCenterGravity(15,0,number);
        stopMovie();

        reset(true);

        System.out.println("Running uniform, gravity at center test with movie");
        startMovie("random5percentmovie.mov");
        runRandomCenterGravity(15,0.05,number);
        stopMovie();

        reset(true);

        System.out.println("Running uniform, gravity at center test with movie");
        startMovie("random10percentmovie.mov");
        runRandomCenterGravity(15,0.10,number);
        stopMovie();

        reset(true);

        System.out.println("Running uniform, gravity at center test with movie");
        startMovie("random15percentmovie.mov");
        runRandomCenterGravity(15,0.15,number);
        stopMovie();
```

```
1135    }

        public void runAttractionComparison() {
          reset(false);

1140      PhysicsSimulator.setGravity(0.0);
          PhysicsSimulator.setFriction(0.0, 0.0);

          System.out.println("Running attraction test 1");
          startMovie("uniformattract.mov");
1145      double[] packingArray = runRandomAttraction(15,0.0,300,false);
          stopMovie();
    //      depositer.saveArray("centralpacking.txt", "0.0", packingArray);
    //      saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);
          saveBoard("uniformattract");
1150      saveBoardImage("uniformattract.jpg");

          reset(false);

          PhysicsSimulator.setGravity(0.0);
1155      PhysicsSimulator.setFriction(0.0, 0.0);

          System.out.println("Running attraction test 2");
          startMovie("random15percentattract.mov");
          packingArray = runRandomAttraction(15,0.15,300,false);
1160      stopMovie();
    //      depositer.saveArray("centralpacking.txt", "0.0", packingArray);
    //      saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);
          saveBoard("random15percentattract");
          saveBoardImage("random15percentattract.jpg");
1165    }

        public void runAttractionPacking() {
          reset(false);

1170      PhysicsSimulator.setGravity(0.0);
          PhysicsSimulator.setFriction(0.0, 0.0);

          System.out.println("Running attraction packing of 5% random balls");
    //      startMovie("random10percentattractmasses.mov");
1175      double[] packingArray = runRandomAttraction(15,0.05,300, false);
    //      stopMovie();
    //      saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);
          saveArray("attractpacking.txt", "0.05", packingArray);
    //      saveBoard("attractpackingrandom10masses");
1180  //      saveBoardImage("attractpackingrandom10masses.jpg");
        }

        public void runAttractionGrating() {
          reset(false);
1185
          PhysicsSimulator.setGravity(0.0);
          PhysicsSimulator.setFriction(0.0, 0.0);

          System.out.println("Running attraction packing of uniform balls in a grating");
1190      startMovie("uniformgratingmasses.mov");
          double[] packingArray = runRandomAttractionGrating(15,0.0,150,6*15*Math.sin(Math.PI/3.0),true
              );
          stopMovie();
    //      saveArrayLineSepOverwrite("uniformgratingdiameters.txt", diameterArray);
          saveArray("attractgrating.txt", "0.0 masses", packingArray);
```

```
1195        saveBoard("attractgratinguniformmasses");
            saveBoardImage("attractgratinguniformmasses.jpg");
          }

          public void runAttractionPackingBimodal() {
1200        reset(false);

            PhysicsSimulator.setGravity(0.0);
            PhysicsSimulator.setFriction(0.0, 0.0);

1205        System.out.println("Running attraction packing of balls with two different masses");
      //      startMovie("random10percentattractmasses.mov");
            double[] packingArray = runRandomAttractionBimodal(15,0.0,15,0.0,2.0,300,false);
      //      stopMovie();
      //      saveArrayLineSepOverwrite("uniformdiameters.txt", diameterArray);
1210        saveArray("attractpacking.txt", "uniform bimodal", packingArray);
            saveBoard("attractpackinguniformbimodal");
            saveBoardImage("attractpackinguniformbimodal.jpg");
          }

1215      public void testLuminosity() {
            double luminosity;
            NumberFormat format = NumberFormat.getInstance();
            format.setMinimumFractionDigits(2);

1220        makeHexagonalLattice();
            //makeSingleCluster();

            luminosity = determineLuminosity(1.0,10);
            System.out.println("for all bad dots luminosity = " + format.format(luminosity));
1225
            luminosity = determineLuminosity(0.0,10);
            System.out.println("for all good dots luminosity = " + format.format(luminosity));

            luminosity = determineLuminosity(1.0/3.0,150);
1230        System.out.println("for 2/3 good dots luminosity = " + format.format(luminosity));
          }

          public void calculateLuminosityPrevious() {
            double luminosity;
1235        NumberFormat format = NumberFormat.getInstance();
            format.setMinimumFractionDigits(3);

            loadBoard("attractpackinguniformmasses");
            luminosity = determineLuminosity(1.0/3.0,1000);
1240        System.out.println("for uniform masses luminosity = " + format.format(luminosity));

            loadBoard("attractpackingrandom5masses");
            luminosity = determineLuminosity(1.0/3.0,1000);
            System.out.println("for 5% random masses luminosity = " + format.format(luminosity));
1245
            loadBoard("attractpackingrandom10masses");
            luminosity = determineLuminosity(1.0/3.0,1000);
            System.out.println("for 10% random masses luminosity = " + format.format(luminosity));

1250        loadBoard("attractpackingrandom15masses");
            luminosity = determineLuminosity(1.0/3.0,1000);
            System.out.println("for 15% random masses luminosity = " + format.format(luminosity));
          }

1255      public void hexagonalTestTest() {
```

```
        makeSingleCluster();
        saveBoard("singlecluster");
      }

1260  public void hexagonalTest() {
        makeHexagonalLattice();
        changeSomeBalls(1.0/3.0);
        double radFrac = findRadiativeFraction();
        System.out.println(radFrac + "_fraction_of_sites_radiate");
1265    saveBoard("hexagonallattice");
      }

      /**
       * @param args
1270   */
      public static void main(String[] args) {
    //    FileDepositer depositer = new FileDepositer(false);
    //    System.out.println("Running uniform test");
    //    depositer.runUniform(15,6);
1275  //    depositer.saveBoard("uniform6layers");

    //    System.out.println("Running random test with 5% standard deviation");
    //    depositer.runRandom(15,0.05,3);
    //    depositer.saveBoard("random1");
1280
    //    System.out.println("Running random test with 10% standard deviation");
    //    depositer.runRandom(15,0.10,3);
    //    depositer.saveBoard("random10percent");

1285  //    System.out.println("Running random test with 15% standard deviation");
    //    depositer.runRandom(15,0.15,3);
    //    depositer.saveBoard("random15percent");

    //    System.out.println("Reading random test with 10% standard deviation");
1290  //    System.out.println(depositer.determinePacking("random2",5*15,4*15,2*15));

    //    System.out.println("Reading uniform1");
    //    System.out.println(depositer.determinePacking("uniform1",2*15,1*15,2*15));

1295  //    FileDepositer depositer = new FileDepositer(true);

    //    System.out.println("Running random 2.5% std dev, gravity at center test");
    //    depositer.runRandomCenterGravity(15,0.025,100);
    //    depositer saveBoard("random2.5percentcentral");
1300
    //    depositer.runCentralSuite(500);

    //    System.out.println(depositer.determinePacking("random15percentcentralquickfreeze",
        PhysicsSimulator.getCenterOfGravity(), Math.sqrt(0.74*100)*15.0/2.0-15.0));

1305  //    System.out.println("Running uniform size, gravity at center test");
    //    double[] packingArray = depositer.runRandomCenterGravity(15,0,50);
    //    depositer.saveArray("packingtest.txt", "0.0", packingArray);
    //    depositer.saveBoard("uniformcentraltest");
    //    depositer.saveBoardImage("uniformcentral.jpg");
1310
    //    depositer.loadBoard("random10percent");
    //    depositer.colorizeBalls(15,0.10);
    //    depositer.saveBoard("random10percent");

1315  //    System.out.println("Running 5% std dev, gravity at center, coloration test");
```

```
    //      double[] packingArray = depositer.runRandomCenterGravity(15,0.15,50);
    //      depositer.saveArray("packingtest.txt", "0.0", packingArray);
    //      depositer.saveBoard("colortest");
    //      depositer.saveBoardImage("colortest.jpg");
1320
    //      depositer.colorizeCentralSuite();

    //      depositer.testDetermineFill();

1325 //      depositer.calculateCentralSuiteFill();

    //      depositer.testDetermineFillWithinRadius();

    //      depositer.calculateCentralSuiteFillWithinRadius();
1330
    //      depositer.testMovie();

    //      depositer.calculateCentralSuiteSpatialCorrelation();

1335 //      depositer.hexagonalTestTest();

        FileDepositer depositer = new FileDepositer(false);

    //      depositer.runAttractionComparison();
1340
    //      depositer.runAttractionPacking();

    //      depositer.runAttractionGrating();

1345 //      depositer.runAttractionPackingBimodal();

    //      depositer.testLuminosity();

        depositer.calculateLuminosityPrevious();
1350  }

    }
```

Listing A.3: Class containing important physics information.

```
    package se102.gb;

    import physics.*;
    import java.util.Random;
 5
    /**
     * Central function library for applying the physics defined for the gizmoball environment
     *
     * @specfield gravity | number   // acceleration in pixels/second^2 representing the acceleration
          of gravity in this physics context
10   * @specfield friction1 | number // representing the constant change of friction
     * @specfield friction2 | number // representing the linear change of friction
     **/

    public class PhysicsSimulator {
15    private static int lengthToPixels = 30;
      private static double gravity;// = 25 * lengthToPixels;  // L/sec^2
      private static double friction1;// = 0.025;              // 1/sec
      private static double friction2;// = 0.025 / lengthToPixels;   // 1/L
      private static Vect centerOfGravity = null; // null then vertical gravity
20
```

```
  public static double getGravity() {
    return gravity;
  }

  public static double getFriction1() {
    return friction1;
  }

  public static double getFriction2() {
    return friction2;
  }

    /*
     * @return designated point for gravity or null if default gravity
     */
    public static Vect getCenterOfGravity() {
      return centerOfGravity;
    }

  /**
   * @requires gravity is in L/sec^2
   * @effects sets this.gravity to gravity
   **/
  public static void setGravity(double gravity) {
    PhysicsSimulator.gravity = gravity * lengthToPixels;
  }

  /**
   * @requires friction1 is in 1/sec and friction2 is in L
   * @effects sets this.gravity to gravity
   **/
  public static void setFriction(double friction1, double friction2) {
    PhysicsSimulator.friction1 = friction1;
    PhysicsSimulator.friction2 = friction2 / lengthToPixels;
  }

    public static void setCenterOfGravity(Vect gravityPoint) {
      centerOfGravity = gravityPoint;
    }

  /**
   * Returns the velocity updated for the time period deltaT in seconds and velocity in L/seconds
   * @requires velocity != null
   * @return new velocity vector in L/seconds with the effect of gravity and friction applied to
       it for
   * the specified time period
   **/
  public static Vect updateVelocity(Vect center, Vect velocity, double deltaT) {
    if(centerOfGravity == null) {
      velocity = velocity.plus(new Vect(0, gravity*deltaT));
    } else {
      velocity = velocity.plus(centerOfGravity.minus(center).unitSize().times(gravity*deltaT));
    }
    velocity = velocity.times(Math.max(1-friction1*deltaT-friction2*velocity.length()*deltaT,0.0)
        );
    return velocity;
  }

  /**
   * Returns the velocity updated for the time period deltaT in milliseconds
   * @requires velocity != null
```

```
80      * @return new velocity vector with the effect of gravity and friction applied to it for
        * the specified time period
        **/
       public static Vect updateVel(Vect velocity, double deltaT) {
         double deltaSeconds = deltaT/1000;
85       Vect velFrict = velocity.times(1-friction1*deltaSeconds-friction2*velocity.length()*
             deltaSeconds);
         Vect velGrav = velFrict.plus(new Vect(0, gravity*deltaSeconds));
         return velGrav;
       }

90     /**
        * Returns the position updated for the time period deltaT in milliseconds
        * @requires position != null && velocity != null
        * @return new position vector updated with the effects of velocity, gravity and
        * friction for the specified time period
95      **/
       public static Vect updatePos(Vect position, Vect velocity, double deltaT) {
         return position.plus(velocity.times(deltaT/1000));
       }

100    public static Geometry.VectPair updateVelandPos(Vect position, Vect velocity, double deltaT) {
         double deltaSeconds = deltaT/1000;
         Vect velFrict = velocity.times(1-friction1*deltaSeconds-friction2*velocity.length()*
             deltaSeconds);
         Vect newVel = velFrict.plus(new Vect(0, gravity*deltaSeconds));
         // use only the velocity corrected for friction to find the linear component of the position
             update
105      Vect posFrict = velFrict.times(deltaSeconds);
         // gravity is a quadratic term
         Vect posGrav = new Vect(0,gravity*(deltaSeconds*deltaSeconds));
         Vect newPos = position.plus(posFrict.plus(posGrav));
         return new Geometry.VectPair(newPos, newVel);
110    }

       // new section for microscopic physics

       private static boolean inelastic = true;
115    private static double fractionElastic = 0.8;
       private static double elasticDeviation = Math.min((1-fractionElastic)/2,fractionElastic/2);
       private static double ballStickThresh = 2*30;
       private static double reflectorStickThresh = 1*30;
       private static double completelyInelasticThresh = 1; // velocity below which balls should stop
           completely upon colliding
120    private static Random collisionVelocityRandom = new Random(913);

       private static double forceDistanceThresh = 0.1; // balls closer than this distance feel no
           force
       private static double R0 = 15.0;//15.0*Math.sqrt(1/2);
       private static double b = forceDistanceThresh-R0*Math.pow(forceDistanceThresh/(1.0*
           completelyInelasticThresh*completelyInelasticThresh/0.01),1.0/7.0);
125
       private static double randomVelocityDelay = 1/50.0;
       private static double T = 297.0;
       private static double kb = 1.38e-20; // Boltzmann's constant in m^2 g s^-2 K^-1
       private static double avgMass = 5.24e-18; // mass of average ball in g
130
       public static boolean isInelastic() {
         return inelastic;
       }
```

```
135     public static void setBallStickThresh(double bst) {
          ballStickThresh = bst;
        }

        public static double getBallStickThresh() {
140       return ballStickThresh;
        }

        public static void setReflectorStickThresh(double rst) {
          reflectorStickThresh = rst;
145     }

        public static double getReflectorStickThresh() {
          return reflectorStickThresh;
        }
150
        public static void setCompletelyInelasticThresh(double cit) {
          completelyInelasticThresh = cit;
        }

155     public static double getCompletelyInelasticThresh() {
          return completelyInelasticThresh;
        }

        private static void updateB() {
160       b = forceDistanceThresh-R0*Math.pow(forceDistanceThresh/(1.0*completelyInelasticThresh*
              completelyInelasticThresh),1.0/7.0);
        }

        public static void setForceDistanceThresh(double thresh) {
          forceDistanceThresh = thresh;
165       updateB();
        }

        public static double getMinimumForceDistanceThresh() {
          return forceDistanceThresh;
170     }

        public static double getMaximumForceDistanceThresh() {
          return R0*2; // force will be 1/2^7 times less than its strongest i.e. completely minuscule
        }
175
        public static void setVanDerWaalCoeff(double coeff) {
          R0 = coeff;
          updateB();
        }
180
        public static double getVanDerWallCoeff() {
          return R0;
        }

185     /*
         * @returns the magnitude of the velocity change in the direction between two balls
         */
        public static double updateVelocityAttraction(double dist, double mass, double time) {
          if(dist < forceDistanceThresh || dist > R0*2) {
190         return 0.0;
          } else {
            double a = Math.pow(R0/(dist-b),7.0)/mass;
            return Math.min(a*time/2.0,Math.sqrt(dist*a)/2.0);
          }
```

```
195     }

        /*
         * @param inel : The fraction inclasticity on average over many collisions. 1.0 = elastic
         */
200     public static void setElasticity(double el) {
          fractionElastic = el;
          elasticDeviation = Math.min((1-el)/2,el/2); // set std deviation of distribution so that 0
               inclasticity is 2 std devs away from mean
        }

205     public static double getElasticity() {
          return fractionElastic;
        }

        /*
210      * @returns a fraction of a balls current velocity lost in a particular collision
         */
        public static double getFractionVelocityLoss() {
          return collisionVelocityRandom.nextGaussian()*elasticDeviation+fractionElastic;
        }
215
        public static double getRandomVelocityDelay() {
          return randomVelocityDelay;
        }

220     /*
         * @param mass : Mass of the ball in the units of the average ball mass
         * @returns The magnitude to be used for the average thermal energy to be randomly added to a
              ball
         */
        public static double getRandomVelocityMagnitude(double mass) {
225       return 1.5*Math.sqrt(2*kb*T/(mass*avgMass));
        }
      }
```

Appendix B

Planarization Code

Listing B.1: Oxide smoothing main script.

```
%function oxidesmoothing6()
warning off MATLAB:divideByZero
% script to describe the electrochemical oxidized smoothing of a ruff
% surface
global samplingx samplingy lengthx lengthy xorig yorig zorig x y z xox yox zox sampleincrfinal ...
    plotstep analysisstep

%global sample

%samplingx = -2:1:2;
%samplingy = -2:1:2;
%samplingx = 0:100:2000;
%samplingy = 0:100:2000;
samplingx = 0:100:400;
samplingy = 0:100:400;

lengthx = length(samplingx);
lengthy = length(samplingy);
[x,y] = meshgrid(samplingx,samplingy);
%{
z = zeros(lengthx,lengthy)+4;
z(3,3) = 4.5;

zox = z;
axis2D = [min(samplingx) max(samplingx) 0 8];
axis3D = [min(samplingx) max(samplingx) min(samplingy) max(samplingy) 0 8];
increaseSampling(4);
%increaseSampling(sample);
%}
z = zeros(lengthx,lengthy)+1000; %now measured in nanometers
z = z+30/2*(2*rand(lengthx,lengthy)-1);
axis2D = [min(samplingx) max(samplingx) 0 2000];
axis3D = [min(samplingx) max(samplingx) min(samplingy) max(samplingy) 800 1200];
increaseSampling(5);
z = z+20/2*(2*rand(lengthx,lengthy)-1);
zox = z;
increaseSampling(4);
```

```
xorig = x;
yorig = y;
zorig = z;

plotstep = 50;%0.25;%50
analysisstep = 10;%0.05;%10
lastplot = 0;
lastanalysis = 0;

centerindex = floor((length(x)+1)/2);

slicehandle = figure;
plot(x(centerindex,:),z(centerindex,:),'o');
hold on;
plot(xox(centerindex,:),zox(centerindex,:),'*');
axis(axis2D);

figure;
mesh(xox,yox,zox);
xlabel('x_axis_(nm)');
ylabel('y_axis_(nm)');
zlabel('z_axis_(nm)');
title('Planarized_Oxide_Surface');
coloraxis = caxis;

twosurfhandle = figure;
handles1 = plot3(x,y,z,'o');
hold on;
handles2 = plot3(xox,yox,zox,'*');
axis(axis3D);
xlabel('x_axis');
ylabel('y_axis');
zlabel('z_axis');
title('Final_Metal_and_Oxide_Surfaces');
legend([handles1(1) handles2(1)],'Metal_surface','Oxide_surface',1);
hold off;

%add to the oxide according to the slope at that point
global oxidestep maxruffness expansion finaloxide distance2ox oxpoints metpoints smallestdist
oxidestep = 0.5;%0.01;
expansion = 2.3;
%maxruffness should just be expansion*oxidestep but added 1.5 to be safe
maxruffness = 1.5*(expansion)*oxidestep; %only applies for perfect beginning substrate
finaloxide = 200;%1;
smallestdist = 0;
count = 0;
%all minimum distances start as zero for this simulation
distance2ox = zeros(lengthx,lengthy);
%points to a single oxide point
oxpoints = cell(size(distance2ox)); %place to store oxide points equally
                                    %close to each point in the metal.
metpoints = cell(size(distance2ox)); %auxilliary data structure to store all the
                                     %metal points close to this oxide point
%for starting with no oxide initialize oxpoints
for i=1:length(oxpoints(:))
    oxpoints{i} = [i];
    metpoints{i} = [i];
end

oxideanalysis(1);
```

```
    while(smallestdist < finaloxide) %loop for each local oxide change
100     [smallestdist smallestind] = min(distance2ox(:));
        count = count+1;
        oxpointlist = oxpoints{smallestind};
        %oxpointlist should be a (1 by n) matrix of oxide indices
        totalchange = 0;
105     for k=1:length(oxpointlist)
            zox(oxpointlist(k)) = zox(oxpointlist(k))+(expansion-1)*oxidestep/(length(oxpointlist)*...
                length(metpoints{oxpointlist(k)}));
            totalchange = totalchange+1/(length(oxpointlist)*length(metpoints{oxpointlist(k)}));
        end
        if(totalchange > 1)
110         smallestind
            distance2met(oxpointlist(k))
            metpoints{oxpointlist(k)}
            distance2ox(smallestind)
            oxpoints{smallestind}
115         oxpointlist
            return
            'should_not_happen'
        end
        z(smallestind) = z(smallestind)-oxidestep*totalchange;
120     %done updating points, now calculate new distances for these points
        points2change = metpoints(oxpointlist);
        %now consider the metal point
        updatemetalpoint3(smallestind);
        % TBA ---- oxide point to metal plane distance
125     % now we need to update points changed in the oxide
        for k=1:length(oxpointlist)
            for i=points2change{k}
                %use the metpoints list to find the metal points that need
                %to be changed because the oxide has changed
130             %the following test didn't account for changing lists from
                %above updatemetalpoint call
                %make sure we haven't already updated this metal point
                if(i ~= smallestind)
                    updatemetalpoint3(i);
135             end
            end
            updateoxidepoint(oxpointlist(k));
        end
        if(mod(count,length(x(:)))==0)%(std(distances(:))<=oxidestep/1000)
140         %plot animation of progress
            count
            figure(twosurfhandle);
            handles1 = plot3(x,y,z,'o');
            hold on;
145         handles2 = plot3(xox,yox,zox,'*');
            axis(axis3D);
            xlabel('x_axis');
            ylabel('y_axis');
            zlabel('z_axis');
150         title('Final_Metal_and_Oxide_Surfaces');
            legend([handles1(1) handles2(1)],'Metal_surface','Oxide_surface',1);
            hold off;
        end

155     if(smallestdist > analysisstep+lastanalysis)
            lastanalysis = floor(smallestdist/analysisstep)*analysisstep;
            oxideanalysis(0),
        end
```

```
160       if(smallestdist > plotstep+lastplot)
              lastplot = floor(smallestdist/plotstep)*plotstep;

              figure;
              mesh(xox,yox,zox);
165           xlabel('x_axis_(nm)');
              ylabel('y_axis_(nm)');
              zlabel('z_axis_(nm)');
              caxis(coloraxis+(mean(zox(:))-mean(coloraxis(:))));
              title('Planarized_Oxide_Surface');
170       end
      end

      figure(slicehandle);
      plot(x(centerindex,:),z(centerindex,:),'or');
175   plot(xox(centerindex,:),zox(centerindex,:),'*r');
      axis(axis2D);
      xlabel('x_axis');
      ylabel('z_axis');
      title('Comparison_of_Initial_to_Final_Metal/Oxide_Surface_Cut');
180   legend('Initial_metal','Initial_oxide','Final_metal','Final_oxide',1);
```

Listing B.2: Function to update a metal point.

```
      function updatemetalpoint3(index)

      global samplingx samplingy lengthx lengthy x y z xox yox zox
      global oxidestep maxruffness expansion finaloxide distance2ox oxpoints distance2met metpoints
 5
      prevdist = distance2ox(index);
      clearMetalPoint(index,oxpoints{index});
      %reset its list and distance
      distance2ox(index) = Inf;
10    oxpoints{index} = [];

      maxdistindex = max(floor((prevdist+maxruffness)/(x(1,2)-x(1,1))),floor((prevdist+maxruffness)/(y...
          (2,1)-y(1,1))));
      pointset = allpoints2dist(index,maxdistindex,lengthx,lengthy);
      distset = sqrt((xox(pointset)-x(index)).^2+(yox(pointset)-y(index)).^2+(zox(pointset)-z(index))...
          .^2);
15    dist = min(distset(:));
      minind = find(distset==dist);
      points2add = pointset(minind);
      if(dist==distance2ox(index))
          oxpoints{index} = union(oxpoints{index},points2add);
20        for i=points2add
              metpoints{i} = union(metpoints{i}, index);
          end
      end
      if(dist < distance2ox(index))
25        clearMetalPoint(index,oxpoints{index});
          distance2ox(index) = dist;
          oxpoints{index} = points2add;
          for i=points2add
              metpoints{i} = union(metpoints{i}, index);
30        end
      end
      %next consider 4 lines from the metal point to find if any oxide
      %point is closest to one of those lines
      for l=1:4
```

```
        point2ind = pointaround(index,1,lengthx,lengthy);
        if(point2ind==0)
            continue
        end
        vect1 = [x(index) y(index) z(index)];
        vect2 = [x(point2ind) y(point2ind) z(point2ind)];
        diff12 = vect2-vect1;
        dist12 = sum(diff12.^2);
        maxdistindex = max(floor(distance2ox(index)/(x(1,2)-x(1,1))),floor(distance2ox(index)/(y(2,1)...
            -y(1,1))));
        pointset = union(allpoints2dist(index,maxdistindex,lengthx,lengthy),allpoints2dist(point2ind,...
             maxdistindex,lengthx,lengthy));
        vect0set = [xox(pointset(:)) yox(pointset(:)) zox(pointset(:))];
        vect1set = repmat(vect1, length(pointset),1);
        vect2set = repmat(vect2, length(pointset),1);
        diff12set = repmat(diff12,length(pointset),1);
        distset = sum((cross(diff12set,vect1set-vect0set)).^2,2)/dist12;
        %check if actually on correct segment
        dist01set = sum((vect1set-vect0set).^2,2);
        dist02set = sum((vect2set-vect0set).^2,2);
        segind = find(dist01set < (distset+dist12) & dist02set < (distset+dist12));
        if(segind)
            dist = min(distset(segind));
            minind = find(distset==dist); %find the indices of all minimum values in distset
                                          %note: distset has the same incides as pointset
            minind = intersect(minind,segind); %keeps only indices in minind that are also in segind
            dist = sqrt(dist);
            points2add = pointset(minind);
            if(dist==distance2ox(index))
                oxpoints{index} = union(oxpoints{index},points2add);
                for i=points2add
                    metpoints{i} = union(metpoints{i}, index);
                end
            end
            if(dist < distance2ox(index))
                clearMetalPoint(index,oxpoints{index});
                distance2ox(index) = dist;
                oxpoints{index} = points2add;
                for i=points2add;
                    metpoints{i} = union(metpoints{i}, index);
                end
            end
            if(dist==distance2ox(point2ind))
                origlength = length(oxpoints{point2ind});
                oxpoints{point2ind} = union(oxpoints{point2ind},points2add);
                for i=points2add
                    metpoints{i} = union(metpoints{i}, point2ind);
                end
            end
            if(dist < distance2ox(point2ind))
                clearMetalPoint(point2ind,oxpoints{point2ind});
                distance2ox(point2ind) = dist;
                oxpoints{point2ind} = points2add;
                for i=points2add
                    metpoints{i} = union(metpoints{i}, point2ind);
                end
            end
        end
    end

    %next consider 4 lines from each oxide point to find if the metal
```

```
    %point is closest to one of those lines
95
    vect0 = [x(index) y(index) z(index)];
    maxdistindex = max(floor(distance2ox(index)/(x(1,2)-x(1,1))),floor(distance2ox(index)/(y(2,1)-y...
        (1,1))));
    pointset = allpoints2distpairs(index,maxdistindex,lengthx,lengthy);
    vect0set = repmat(vect0,length(pointset),1);
100 vect1set = [xox(pointset(:,1)) yox(pointset(:,1)) zox(pointset(:,1))];
    vect2set = [xox(pointset(:,2)) yox(pointset(:,2)) zox(pointset(:,2))];
    %now we have all pairs of points in pointset between vect1set and vect2set
    diff01set = vect1set-vect0set;
    diff12set = vect2set-vect1set;
105 dist12set = sum(diff12set.^2,2);
    %will generate NaN for point same in both vect1set and vect2set but those
    %indices will be excluded from segind
    distset = sum((cross(diff12set,diff01set)).^2,2)./dist12set;
    dist01set = sum(diff01set.^2,2);
110 dist02set = sum((vect2set-vect0set).^2,2);
    segind = find(dist01set < (distset+dist12set) & dist02set < (distset+dist12set));
    if(segind)
        dist = min(distset(segind));
        minind = find(distset==dist); %find the indices of all minimum values in distset
115                                   %note: distset has the same incides as pointset
        minind = intersect(minind,segind); %keeps only indices in minind that are also in segind
        dist = sqrt(dist);
        points2change = pointset(minind,:);
        points2change = unique(points2change(:)');
120     if(dist==distance2ox(index))
            oxpoints{index} = union(oxpoints{index},points2change);
            for j=1:length(points2change)
                metpoints{points2change(j)} = union(metpoints{points2change(j)},index);
            end
125     end
        if(dist < distance2ox(index))
            clearMetalPoint(index,oxpoints{index});
            distance2ox(index) = dist;
            oxpoints{index} = points2change;
130         for j=1:length(points2change)
                metpoints{points2change(j)} = union(metpoints{points2change(j)},index);
            end
        end
    end
```

Listing B.3: Function to update an oxide point.

```
    function updateoxidepoint(index)

    global samplingx samplingy lengthx lengthy x y z xox yox zox
    global oxidestep maxruffness expansion finaloxide distance2ox oxpoints distance2met metpoints ...
        smallestdist
 5
    maxdistindex = max(floor((smallestdist+maxruffness)/(x(1,2)-x(1,1))),floor((smallestdist+...
        maxruffness)/(y(2,1)-y(1,1))));
    pointset = allpoints2dist(index,maxdistindex,lengthx,lengthy);
    distset = sqrt((x(pointset)-xox(index)).^2+(y(pointset)-yox(index)).^2+(z(pointset)-zox(index))...
        .^2);
    %indices in distset of the points to change
10  points2change = find(distset <= distance2ox(pointset));
    if(points2change)
        for i=points2change
            metindex = pointset(i);
```

```
            dist = distset(i);
            if(dist==distance2ox(metindex))
                oxpoints{metindex} = union(oxpoints{metindex},index);
                metpoints{index} = union(metpoints{index}, metindex);
            end
            if(dist < distance2ox(metindex))
                clearMetalPoint(metindex,oxpoints{metindex});
                distance2ox(metindex) = dist;
                oxpoints{metindex} = index;
                metpoints{index} = union(metpoints{index}, metindex);
            end
        end
    end
    %next consider 4 lines from the oxide point to find if any metal
    %point is closest to one of those lines
    for l=1:4
        point2ind = pointaround(index,l,lengthx,lengthy);
        if(point2ind==0)
            continue
        end
        vect1 = [xox(index) yox(index) zox(index)];
        vect2 = [xox(point2ind) yox(point2ind) zox(point2ind)];
        diff12 = vect2-vect1;
        dist12 = sum(diff12.^2);
        maxdistindex = max(floor((smallestdist+maxruffness)/(x(1,2)-x(1,1))),floor((smallestdist+...
            maxruffness)/(y(2,1)-y(1,1))));
        pointset = union(allpoints2dist(index,maxdistindex,lengthx,lengthy),allpoints2dist(point2ind,...
            maxdistindex,lengthx,lengthy));
        vect0set = [x(pointset(:)) y(pointset(:)) z(pointset(:))];
        vect1set = repmat(vect1, length(pointset),1);
        vect2set = repmat(vect2, length(pointset),1);
        diff12set = repmat(diff12,length(pointset),1);
        distset = sum((cross(diff12set,vect1set-vect0set)).^2,2)/dist12;
        %check if actually on correct segment
        dist01set = sum((vect1set-vect0set).^2,2);
        dist02set = sum((vect2set-vect0set).^2,2);
        segind = find(dist01set < (distset+dist12) & dist02set < (distset+dist12));
        if(segind)
            distset = sqrt(distset)';
            %list of indicies in segind of the points that need to change
            points2change = find(distset(segind) <= distance2ox(pointset(segind)));
            %allchanged = union(allchanged, pointset(segind(points2change)));
            %now indexed by their position in pointset
            points2change = segind(points2change)';
            if(points2change)
                for i=points2change
                    metindex = pointset(i);
                    dist = distset(i);
                    if(dist==distance2ox(metindex))
                        oxpoints{metindex} = union(oxpoints{metindex},[index point2ind]);
                        metpoints{index} = union(metpoints{index}, metindex);
                        metpoints{point2ind} = union(metpoints{point2ind}, metindex);
                    end
                    if(dist < distance2ox(metindex))
                        clearMetalPoint(metindex,oxpoints{metindex});
                        distance2ox(metindex) = dist;
                        oxpoints{metindex} = [index point2ind];
                        metpoints{index} = union(metpoints{index}, metindex);
                        metpoints{point2ind} = union(metpoints{point2ind}, metindex);
                    end
                end
```

```
        end
    end
end
%next consider 4 lines from each metal point to find if the oxide
%point is closest to one of those lines
vect0 = [xox(index) yox(index) zox(index)];
maxdistindex = max(floor((smallestdist+maxruffness)/(x(1,2)-x(1,1))),floor((smallestdist+...
    maxruffness)/(y(2,1)-y(1,1))));
pointset = allpoints2distpairs(index,maxdistindex,lengthx,lengthy);
vect0set = repmat(vect0,length(pointset),1);
vect1set = [x(pointset(:,1)) y(pointset(:,1)) z(pointset(:,1))];
vect2set = [x(pointset(:,2)) y(pointset(:,2)) z(pointset(:,2))];
%now we have all pairs of points in pointset between vect1set and vect2set
diff01set = vect1set-vect0set;
diff12set = vect2set-vect1set;
dist12set = sum(diff12set.^2,2);
%will generate NaN for point same in both vect1set and vect2set but those
%indices will be excluded from segind
distset = sum((cross(diff12set,diff01set)).^2,2)./dist12set;
dist01set = sum(diff01set.^2,2);
dist02set = sum((vect2set-vect0set).^2,2);
segind = find(dist01set < (distset+dist12set) & dist02set < (distset+dist12set));
if(segind)
    distset = sqrt(distset);
    %list of indicies in segind of the points that need to change
    points2change1 = find(distset(segind) <= distance2ox(pointset(segind,1)));
    points2change2 = find(distset(segind) <= distance2ox(pointset(segind,2)));
    %now indexed by their positions in global arrays
    points2change = union(pointset(segind(points2change1),1)',pointset(segind(points2change2),2)...
        ');
    if(points2change)
        for metindex=points2change
            allmetind = find(pointset(:,1)==metindex | pointset(:,2)==metindex);
            dist = min(distset(allmetind));
            if(dist==distance2ox(metindex))
                oxpoints{metindex} = union(oxpoints{metindex},index);
                metpoints{index} = union(metpoints{index},metindex);
            end
            if(dist < distance2ox(metindex))
                clearMetalPoint(metindex,oxpoints{metindex});
                distance2ox(metindex) = dist;
                oxpoints{metindex} = index;
                metpoints{index} = union(metpoints{index},metindex);
            end
        end
    end
end
```

Appendix C

Physics Package Copyright Notice

NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

@author: Matt Frank, MIT Laboratory for Computer Science,
mfrank@lcs.mit.edu
1999-Apr-03

@author: Rob Pinder, Phil Sarin, Lik Mui
Spring 2000
Exception handling and argument type refinemnt

@author: Jeffrey Sheldon (jeffshel@mit.edu)
Fall 2000, Spring 2001
Major rewrites and improvements to iterative solving

@author: Jeremy Nimmer (jwnimmer@alum.mit.edu)
Fall 2000, Spring 2001
Editorial role (testing and specification editing)

Bibliography

[1] P. MARDILOVICH and P. KORNILOVITCH, *Nano Letters* **5**, 1899 (2005).

[2] P. MARDILOVICH and P. KORNILOVITCH, Unpublished, Hewlett-Packard Company, Imaging and Printing Group. Corvalis, Oregon, 2005.

[3] D. WELLS, editor, *The Penguin Dictionary of Curious and Interesting Numbers*, p. 30, Penguin Books, Middlesex, England, 1986.

[4] J. H. CONWAY and N. J. A. SLOANE, *Sphere Packings, Lattices, and Groups*, Springer-Verlag, New York, second edition, 1993.

[5] N. J. A. SLOANE, *Nature* **395**, 435 (1998), Editorial review.

[6] D. BIMBERG, M. KUNTZ, and M. LAEMMLIN, *Microelectronics Journal* **36**, 175 (2005).

[7] P. CAROFF and C. P. ET AL., *Applied Physics Letters* **87** (2005), Art. No. 243107.

[8] S. A. MCDONALD and G. K. ET AL., *Nature Materials* **4**, 138 (2005).

[9] J. F. WENG and J. C. REN, *Current Medicinal Chemistry* **13**, 897 (2006).

[10] T. V. ET AL., *Journal of Physical Chemistry* **98**, 7665 (1994).

[11] C. F. H. ET AL., *Journal of Physical Chemistry* **96**, 3812 (1992).

[12] X. G. PENG, M. C. SCHLAMP, and A. V. K. ET AL., *Journal of American Chemical Society* **119**, 7019 (1997).

[13] M. L. STEIGERWALD and A. P. A. ET AL., *Journal of American Chemical Society* **110**, 3046 (1988).

[14] R. Y. Y. ET AL., *Physical Review E* **62**, 3900 (2000).

[15] K. J. DONG and R. Y. Y. ET AL., *Physical Review Letters* **96** (2006).

[16] N. V. B. ET AL., *Physical Review E* **53**, 5382 (1996).

[17] H. MATTOUSSI and A. W. C. ET AL., *Physical Review B* **58**, 7850 (1998).

[18] H. C. HAMAKER, *Physica (Amsterdam)* **4** (1937).

[19] E. RABANI, *Journal of Chemical Physics* **116**, 258 (2002).

[20] C. R. KAGAN, C. B. MURRAY, and M. G. BAWENDI, *Physical Review B* **54**, 8633 (1996).

[21] B. M. ET AL., *Physical Review B* **33**, 5545 (1986).

[22] S. COE, W. K. WOO, M. BAWENDI, and V. BULOVIĆ, *Nature* **420**, 800 (2002).

[23] S. COE, J. S. STECKEL, and W. K. W. ET AL., *Advanced Functional Materials* **15**, 1117 (2005).

[24] C. B. MURRAY, C. R. KAGAN, and M. G. BAWENDI, *Annual Review of Materials Science* **30**, 545 (2000).

[25] E. HOWE, J. YING, S. STRANSKY, and K. RUBRITZ, Unpublished, 6.170 Gizmoball, 2003.

[26] V. SURGANOV, *IEEE Transactions on Components Packaging and Manufacturing Technology Part B-Advanced Packaging* **17**, 197 (1994).

[27] N. F. MOTT, *Philosophical Magazine B-Physics of Condensed Matter Statistical Mechanics Electronic Optical and Magnetic Properties* **55**, 117 (1987).

[28] M. E. LAW, C. S. RAFFERTY, and R. W. DUTTON, SUPREM IV, Software package, Stanford University, 1991.

[29] N. G. WRIGHT, C. M. JOHNSON, and A. G. O'NEILL, *Materials Science and Engineering B-Solid State Materials for Advanced Technology* **61-62**, 468 (1999).