Statistical Properties of the Wind-Tree Model

Presented to the S. Daniel Abraham Honors Program

in Partial Fulfillment of the

Requirements for Completion of the Program

Stern College for Women

Yeshiva University

April 27, 2021

Etta Rapp

Mentor: Professor Peter Nandori, Mathematics

Abstract

Ehrenfest's wind-tree model is a dynamical system representing the motion of a billiard on an infinite plane with periodic rectangular scatterers. Previous research has shown that the diffusion rate of the two-dimensional wind-tree model is ⅔. Our project provides computer code to simulate the wind-tree model in any dimension. We replicate prior results in two-dimensions and predict diffusion rates for the wind-tree model in higher dimensions.

1: Introduction

A dynamical system is a mathematical model in which a function describes the time dependence of the position of a point in a space. One such system is the wind-tree model, which represents the motion of a point particle on an infinite plane with periodic scatterers. The point collides elastically with the scatterers along its trajectory. This model, which can be expanded into higher dimensions, has practical applications to physics.

The diffusion rate of the wind-tree model is a statistical property of the system that measures how far a point in the system travels from its initial location as a function of time. In 2014, Delecroix, Hubert, and Lelièvre [4] proved that the diffusion rate of the two-dimensional wind-tree model is ⅔. In this project, we write computer code to simulate the trajectory of a point in the wind-tree model in any dimension. We use this code to replicate Delecroix, Hubert, and Lelièvre's result [4], and to predict exponents for the wind-tree model in higher dimensions.

In the second section of the paper, we provide relevant mathematical background to the wind-tree model. In section three, we introduce the wind-tree model and summarize prior research related to it. The final section of this paper explains our code and results.

2: Mathematical Background

Ergodic Theory and Recurrence

Ergodic theory is the study of statistical properties of deterministic dynamical systems; in other words, it is the study of the long-term average behavior of systems evolving in time. We represent all the possible states of a system with the space X, and we represent the evolution of the system with a transformation $T : X \to X$ given initial state x at $t = 0$ and state Tx at $t = 1$. In this section, I review some definition and theorems from ergodic theory using Dajani and Dirksin [3].

Consider a probability space $(X, \boldsymbol{B}, \mu)$. X is the sample space composed of all possible outcomes. For example, if we were rolling a die and studying the results, the sample space would be $\{1, 2, 3, 4, 5, 6\}$. $\boldsymbol{B}$ is the event space, a set of events where each event is a set of outcomes in the sample space. The event space for rolling a die would include events such as $\{1\}$, $\{3, 5\}$, and $\{2, 4, 6\}$. Finally, μ is the probability function which assigns a probability between 0 and 1 to each event in the event space. This probability represents the likelihood of the event occurring. The probability of the event $\{1\}$ as the result of rolling a six-sided die is 1/6. The probability of rolling an even number, corresponding to the event $\{2, 4, 6\}$, is 1/2. One common measure used as a probability function is the normalized Lebesgue measure, which corresponds to length in one dimension, area in two dimensions, and volume in three dimensions. We normalize the measure so that the total probability is equal to 1.

Given the probability space $(X, \boldsymbol{B}, \mu)$ and a transformation $T: X \to X$, the map T is called a measure preserving transformation with respect to μ if for all $A \in \boldsymbol{B}$ we have $\mu(T^{-1}A) = \mu(A)$. This means that the probability, μ, of any event in the event space is equal to the probability of its

preimage under map T. (Note that this is not equivalent to stating that $\mu(A) = \mu(T(A))$. For example, if we define $X = [0, 1)$ and $T(x) = 2x \bmod 1$ with μ the Lebesgue measure, then $\mu(T^{-1}A) = \mu(A)$ but $\mu(A) \neq \mu(T(A))$.) A measure preserving dynamical system is defined as a probability space $(X, \boldsymbol{F}, \mu)$ together with a measure preserving transformation T, combined to form a quadruple $(X, \boldsymbol{F}, \mu, T)$. We can think of a dynamical system as a model that describes the position of a point as a function of time.

The Poincaré Recurrence Theorem states that all measure preserving dynamical systems will eventually return to a state arbitrarily close to or equal to their initial state after a sufficiently long, finite amount of time. Formally posed, the theorem states that if $\mu(B) > 0$, then almost every $x \in B$ is B-recurrent. For measure preserving transformation T on probability space $(X, \boldsymbol{F}, \mu)$ with $B \in F$, $x \in B$ is defined as B-recurrent if there exists a value $k \geq 1$ such that $T^k x \in B$. Almost everywhere means on the complement of a set with measure zero. The Poincaré Recurrence Theorem also implies that almost every $x \in B$ returns to B infinitely often.

A measure preserving transformation T on a probability space (X, F, μ) is ergodic if for every measurable set A satisfying $T^{-1}A = A$, we have either $\mu(A) = 0$ or $\mu(A) = 1$. A transformation T is uniquely ergodic if there is a unique measure μ on X which is ergodic for T. The Ergodic Theorem states that given a probability space (X, F, μ) and an ergodic, measure preserving transformation $T: X \to X$, for any test function $f: X \to \mathrm{R}$ in $\mathrm{L}^1(\mu)$ we have $\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} f(T^i(x)) = \int_X f \, d\mu$. This limit converges in L^1 and converges for almost every x.

To understand this, consider a subspace of X called A, and define $f(x) = \{1$ if x is in A, else $0\}$. The ergodic theorem tells us that $\lim_{n\to\infty} \frac{1}{n} \sum_{i=0}^{n-1} f(T^i(x))$, representing the time average of

how often a point in X is in A, is equal to $\int_x f \, d\mu = \mu(A)$, the space average which represents how much of X is covered by A. In other words, if we consider a point traveling through space X, the amount of time the point spends in a subspace of X is proportional to the size of the subspace as a proportion of X. This means that the point moves through the space randomly and uniformly, and that the point eventually reaches all parts of the space. This notion of ergodicity allows us to study the average behavior of a system using the trajectory of one point in the system.

One example of an ergodic transformation is an irrational rotation. In this case we have a circle which is identified with the interval [0, 1). Transformation T: [0, 1) → [0, 1) rotates the circle by an angle of $2\pi * \theta$, formally $T_\theta x = x + \theta \pmod 1$ with $\theta \in (0, 1)$. T is a measure preserving transformation under the Lebesgue measure.

Consider the case where θ is rational, that is $\theta = p / q$ with p, q coprime and $p, q \in Z$. Rotations by θ degrees will define q points on the circle with intervals of length $1/q$ between them. Define $A = \bigcup_{k=0}^{q-1} \left[\frac{k}{q}, \frac{k}{q} + \frac{1}{2q} \right]$, in other words A is the union of the intervals given by taking the first half of each interval of length $1/q$. (See Figure 1 below.) A rotation of p/q will land A in its original position, so the condition $T^{-1}A = A$ is satisfied. However, $\mu(A) = 1/2$, not 0 or 1. Therefore, rational rotations are not ergodic. We can also see that since this transformation is not ergodic, the Ergodic Theorem is not satisfied - the rational rotation recurs, repeatedly returning to the same q points, and will not reach other parts of the space. We see that for all rational θ, the Lebesgue measure is non-ergodic with respect to transformation T.

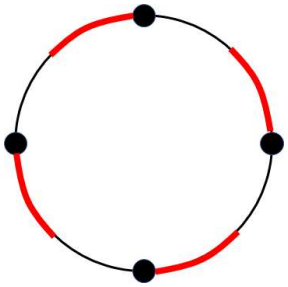

Figure 1: A rational rotation with $\theta = ¼$.
An invariant set A is indicated in red.

If θ is irrational, conversely, the orbit will be dense in the circle, uniformly visiting every open set in the space. We can prove that transformation T, defined as above, is ergodic in the case where θ is irrational. Furthermore, it is uniquely ergodic. Since rational numbers are a measure zero set, we can say that transformation T is ergodic for almost every θ.

Infinite Ergodic Theory

We now discuss the field infinite ergodic theory, referencing Bonanno [2]. Infinite ergodic theory considers ergodicity in infinite measure spaces, studying measure preserving transformations that preserve an infinite measure. Given the measure space $(X, \boldsymbol{B}, \mu)$ with infinite measure μ and a transformation $T: X \to X$, the map T is called a measure preserving transformation with respect to μ if for all $A \in \boldsymbol{B}$ we have $\mu(T^{-1}A) = \mu(A)$. A measure preserving transformation T on a measure space $(X, \boldsymbol{B}, \mu)$ with infinite measure μ is ergodic if $T^{-1}A = A$ (mod μ) for $A \in \boldsymbol{B}$ implies that either $\mu(A) = 0$ or $\mu(X \setminus A) = 0$. This is essentially equivalent to the definition in finite measure spaces, since in the finite case when the measure of A is 1 the measure of $X \setminus A$ is 0. We can alternatively define infinite ergodicity by stating that measure μ is ergodic if T-invariant measurable functions are μ-almost everywhere constant.

In infinite ergodic theory, the Poincaré Recurrence Theorem does not apply, but there is an alternate form of recurrence that does apply. Consider a transformation T on a measure space $(X, \boldsymbol{B}, \mu)$. Let T be a non-singular transformation, defined as a transformation where $\mu(T^{-1}C) = 0$ if and only if $\mu(C) = 0$ for all $C \in \boldsymbol{B}$. This is a more general definition than the definition of measure preserving transformations above. A measurable set $W \in \boldsymbol{B}$ is defined as a wandering set for T if the sets $\{T^{-n}W\}_{n=0}^{\infty}$ are disjoint. We divide space X into two parts. $D(T)$, the dissipative part of T, is the union of the wandering sets for T. $C(T)$, the conservative part of T, is defined as $X \setminus D(T)$, the complement of the dissipative part of X. We can also define the transformation itself as dissipative or conservative. If $D(T) = X \pmod{\mu}$, then T is dissipative. If $C(T) = X \pmod{\mu}$ then T is conservative.

In infinite ergodic theory, conservativity of a transformation is equivalent to validity of the Poincaré Recurrence Theorem. For example, consider the case of shifts along the number line: $X = \mathrm{R}$, $T(x) = x + 1$. Although T is Lebesgue invariant, the transformation does not recur and the Poincaré Recurrence Theorem does not apply because the measure is infinite. Applying the definitions from infinite ergodic theory listed above to this example, we can say that this transformation is dissipative, and therefore it does not recur. A measure space with a conservative transformation would recur. We can show that if T is a measure-preserving transformation on measure space $(X, \boldsymbol{B}, \mu)$ then T is conservative on that space.

Just as the Poincaré Recurrence Theorem does not apply to infinite ergodic theory but a different form of recurrence does apply, the ergodic theorem as phrased above also does not apply to transformations that preserve an infinite measure but a different formulation of the ergodic theorem called the Hopf ratio ergodic theorem does apply to infinite ergodic theory. This theorem compares two observables to produce a limit that converges in infinite spaces.

3: The Wind-Tree Model

The wind-tree model, introduced by Ehrenfest and Ehrenfest [5] in 1912, is a system representing the motion of a billiard ball, a point mass with constant velocity, on an infinite plane with scatterers placed throughout the plane. The particle (the "wind") collides elastically with scatterers (the "trees") and is reflected such that the angle of reflection is equal to the angle of incidence. The model describes the position of the point particle as a function of time. The periodic wind-tree model, introduced by Hardy and Weber [7] in 1980, is a wind-tree model with rectangular scatterers placed at integer coordinates, that is at each point in Z^2. (See Figure 2 below.) We denote the width and height of the scatterers as a and b, $0 < a, b < 1$. We refer to the subset of the plane generated by subtracting the scatterers from the plane as the billiard table $E(a, b)$. The wind-tree model is also known as the infinite billiard table.

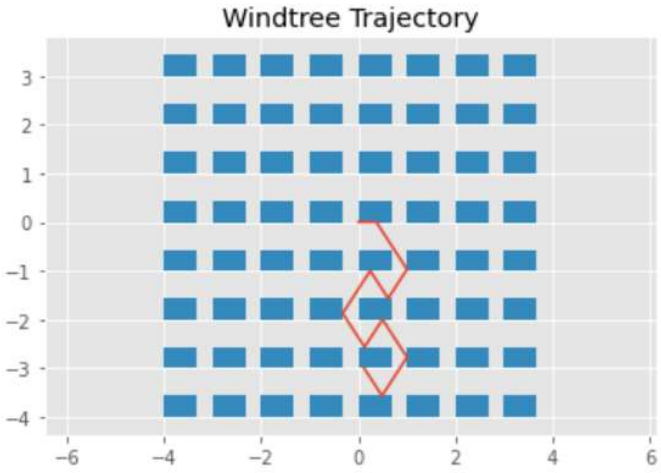

Figure 2: A sample wind-tree trajectory in dimension 2.

We can consider either the compact or the infinite case of the wind-tree model. The compact case is the case of a point moving on a torus. This torus is formed from a unit square with an enclosed a x b scatterer, and opposite sides of the square are identified with each other to

form a torus. The compact wind-tree model is an example of a rational polygonal billiard, which models the motion of a billiard ball inside a polygon, in this case a rectangle, whose angles are all rational multiples of π. To define the model, we state that for every a, b, θ, there exists a dynamical system defined by transformation $T_{a, b, \theta}$: $X \rightarrow X$ on X = the boundary of a rectangle with an outgoing velocity vector. Alternatively, instead of studying the motion of the billiard in the compact wind-tree model as a discrete set of points on the boundary of the scatterer, we can study the flow of the model, considering the position of a point particle in the wind-tree model continuously, as a function of continuous time.

The infinite case of the wind-tree model looks at the infinite plane of tesselating unit squares, each containing a scatterer. We can also study the infinite wind-tree model using either discrete or continuous time. Using discrete time, we would define the motion of the point particle as a sequence of points on the boundaries of scatterers in the plane. Using continuous time, we would model the motion of the point particle as a continuous trajectory on the infinite billiard table. To study the infinite wind-tree model, we cannot apply the Poincaré Recurrence Theorem, and we must use techniques from infinite ergodic theory.

State of the Art Results

Several studies have analyzed properties of the wind-tree model since the model was first proposed by Ehrenfest and Ehrenfest [5].

In 1986, Kerckhoff, Masur, and Smillie [8] showed that for any rational polygonal billiard, for almost every θ, the billiard flow in direction θ is uniquely ergodic. It follows from here that the compact wind-tree model is ergodic with respect to the Lebesgue measure: for every a, b, the set $\{\theta: T_{a, b, \theta}$ is ergodic$\}$ is a full measure set. This means that the wind-tree model is

ergodic for almost every direction, with the exception of specific directions that are dependent on a and b. Furthermore, it is uniquely ergodic.

Other studies produced results regarding the infinite wind-tree model and its infinite measure properties. In 2013, Frączek and Ulcigrai [6] proved that although the wind-tree model is ergodic on the torus, it is typically not ergodic nor minimal in the infinite model. They provide several criteria, and state that for all a, b that meet either of those criteria, for almost every $\theta \in S^1$, the billiard flow in direction θ in the wind-tree model $E(a, b)$ is not ergodic and has uncountably many ergodic components.

In 2014, Delecroix, Hubert, and Lelièvre [4] showed that the polynomial diffusion rate, also known as the escape rate, of the periodic wind-tree model in two dimensional space is ⅔. Formally posed, they state that for all starting positions, all scatterer sizes, and almost all starting angles, the limit superior as time goes to infinity of the log distance traveled by the point divided by the log amount of time elapsed is ⅔. This measure tells us how quickly the point particle travels away from its starting point. It tells us that after a sufficiently long time t, the distance of the point from its initial position will be approximately $t^{2/3}$.

Avila and Hubert [1] proved in 2020 that although the infinite two-dimensional periodic wind-tree model is not ergodic, it is recurrent for all parameters and for almost all directions. Formally, for every $(a, b) \in (0, 1) \times (0, 1)$, the billiard flow on the billiard table $T_{a,b}$ is recurrent for almost every direction θ. This result is only valid for almost every direction because for every a, b it is possible to choose a direction such that the model will not be recurrent. For example, see Figure 3 which displays a trajectory that will not recur.

Figure 3: An wind-tree model which does not recur.

Rational rotations provide an example of an exception to Avila and Hubert's theorem [1]. Consider Figure 3 above, where a point particle zig-zags up and down, returning to a height of b with every other collision. We can map the fractional parts of the x-coordinates of these collision points with y-coordinate b onto the unit circle, and we can then consider the transformation from one x-coordinate to the next as a rotation of the unit circle. Just as rational rotations do not recur, if the horizontal difference between two successive collision points is rational and one more criterion, which I will explain below, is met then the wind-tree model will not recur.

To formally pose this case, consider Figure 4 below which shows a segment of the trajectory of a wind-tree model defined by parameters a, b, θ. In this figure, the horizontal distance between the collision point at (x, b) and the successive collision of height b at (x', b) is 1 + x' - x. The tangent of the initial angle θ is (1 - b), the vertical distance between scatterers, divided by half of 1 + x' - x. If 1 + x' - x is a rational number, i.e. $\frac{tan(\Theta)}{2(1-b)} = \frac{p}{q}$ with $p, q \in Z^+$ coprime, and 1 - a < 1 / q, then the windtree model will not recur. The condition that 1 - a < 1 / q

is necessary so that the point particle does not "fall into" the horizontal gap between scatterers. We also must choose our initial x value in a way that prevents the point from falling into the gap.

This case of a wind-tree model that does not recur provides an example of an exception to Avila and Hubert's thesis [1]. We can generally say that given values a, b, θ defining a wind-tree model, if $\tan(\theta) / (1 - b) = p / q$ with q positive, p and q coprime integers, and $1 - a < 1 / q$, then we have a positive measure set of trajectories that will not recur. Therefore, for every a, b, it is possible to choose countably many directions θ for which the wind-tree model defined by a, b, θ will not recur. Not only is this example not recurrent, it also is not ergodic and does not have a diffusion rate of ⅔. Since rational numbers are countably infinite, they are a measure zero set, so although there are infinitely many cases where the wind-tree model does not recur, we can say that it is recurrent almost always, as stated by Avila and Hubert [1].

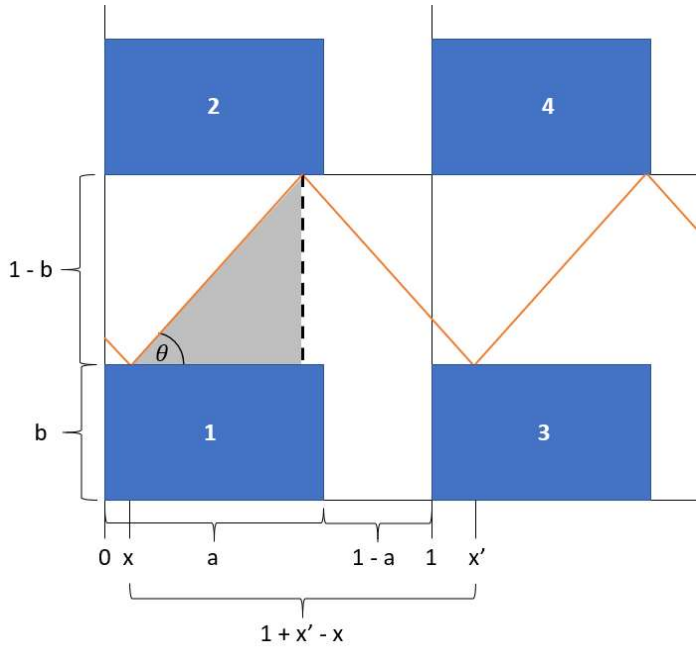

Figure 4: A wind-tree trajectory that does not recur.

4: Methodology and Results

In this project, we write code to simulate the wind-tree model in various dimensions, and use it to predict the diffusion rate of the model. We begin with code for a 3-dimensional wind-tree model, and then increase the level of abstraction, expanding into higher dimensions. Our code is available in appendices A and B of this paper.

Collision Map Code

The main part of our three-dimensional wind-tree code is a function called collision_map. The function has eight inputs, a, b, c, x, y, z, side, and v. Inputs a, b, and c are values between 0 and 1, respectively representing the width, height, and depth of the cubic scatterers in the model. Values x, y, and z are the coordinates of an initial point in 3-dimensional space, located on a face of a scatterer in the model. The side input, which represents the face of the scatterer that the point (x, y, z) is on, is one of the enumerated values NORTH, SOUTH, WEST, EAST, TOP, and BOTTOM. (We use the Python Enum class to ensure that side is one of these discrete values.) The final input, v, is a unit vector representing the velocity of the point.

Given these inputs representing an initial point's position and direction, the collision_map function will return the next point along the trajectory of the point particle where the particle hits a scatterer. The code follows the velocity vector, registering whenever the velocity vector intersects with a face of the unit cube, iterating until the point particle lands on a scatterer. It then returns values x, y, z, side, and v, representing the position, side, and velocity of the new point.

We use the convention that NORTH, EAST, and TOP refer to faces of the scatterer that do not coincide with faces of the unit cube enclosing the scatter. Sides SOUTH, WEST, and BOTTOM refer to faces of the scatterer that are subsets of faces of the unit cube. (See Figure 5 below.) In our collision map, we refer to the case where a point particle is on the NORTH, EAST, or TOP of a scatter as a base case, and refer to sides SOUTH, WEST, and BOTTOM as recursives cases. In a base case, the point particle is on the scatterer, and the velocity vector must point away from the inside of the scatterer, in other words the corresponding component of the velocity vector must be positive. In a recursive case, the point particle may or may not be on the scatterer. We also identify the six faces of the scatterer with tuples, using a dictionary to represent this mapping. NORTH corresponds to (0, 1, 0), SOUTH to (0, -1, 0), WEST to (-1, 0, 0), EAST to (1, 0, 0), TOP to (0, 0, 1), and BOTTOM to (0, 0, -1).

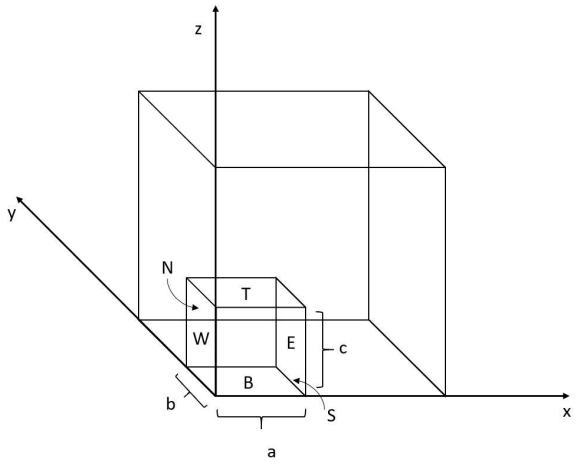

Figure 5: One unit cube in the 3D wind-tree model with labeled faces and dimensions of the enclosed scatterer.

Our code for collision_map first checks if it is dealing with a base case, a case where the initial point is on the NORTH, EAST, or TOP of a scatterer and the velocity vector is pointing

outwards from the scatterer. In this case, the code increments x, y, and z to produce a new point which is on the face of the unit cube. In other words, the point particle continues traveling in the direction of its velocity vector until it reaches the boundary of the unit cube enclosing it. The code then updates the side variable. The new side variable will correspond to a recursive case, and the code continues with that case. The recursive case code will first check if the new point is on a scatterer. If so, this new point will be returned by collision_map; if not, the recursive case code will continue moving the point along until it reaches a scatterer.

The code for the base case is straightforward. For example, assume the side variable is TOP. This means that the z-coordinate of the point particle modulo 1 equals c, the height of the scatterer, and that the velocity vector is pointing upwards. We know that the point particle will now continue along its trajectory until its z-coordinate becomes a whole number. (It is possible that the trajectory will intersect with other faces of the unit cube before this point, but the trajectory will not intersect with any scatterers on this stretch since the fractional part of the z-coordinate of the point is between c and 1 on this segment of the trajectory, so we can disregard these intermediate intersections.)

We can easily calculate the scalar that we must increment the position of the point by in order for its z-coordinate to reach the next integer as (1 - c) / v[2]. Since Python uses 0-based array indexing, v[2] refers to the third element of the velocity vector. We increment the position of the point particle by this scalar value, and then adjust the value of the side variable. In the case of TOP, where we continue upwards until reaching the bottom of a unit cube, the new side variable will be BOTTOM. The cases of EAST and NORTH are similar to the case of TOP with modified scalar computations and side variables. We increment the position of the point using the increment_by_scalar function which takes as input x, y, z, v, and a scalar, adds to each

coordinate the product of the corresponding component of the velocity vector and the scalar value, and returns the updated x, y, and z.

Recursive Case Code

Our code for the recursive case is in a separate function called recursive_case. This function takes the same inputs as the collision_map function: x, y, and z representing the location of the point particle in three-dimensional space, a, b, and c representing the size of the scatterers in the model, side representing the location of the point particle with respect to the scatterers, and v representing the velocity vector of the point. The output values of the function are x, y, z, side, v, and done. Variables x, y, z, side, and v represent the new location and velocity vector of the point particle. The done output is a boolean variable indicating whether the returned point represents a collision with the scatterer or just with a face of the unit cube.

If the done variable returned by the recursive_case function is True, this means that the point particle has landed on a scatterer. In this case, the collision_map function will return output to its caller, returning the new location and velocity of the point. If the done variable is False, this means that the point particle is currently on the boundary of a unit cube but is not on a scatterer. In this case, the collision_map function will continue iterating, calling the recursive_case function as necessary, and the point particle will continue traveling along its trajectory until it reaches a scatterer and then exits the collision_map function.

The recursive_case function is written to handle the case where the side variable is BOTTOM. In the other recursive cases, SOUTH and WEST, collision_map permutes the input and output values to the recursive_case function. For example, in the case of WEST, we would permute the inputs to the recursive_case function to symbolically represent rotating the space so

that the WEST face of the unit cube takes the place of the BOTTOM face. (See Figure 6 below.) The recursive_case function would then calculate the next point for the case of BOTTOM, and the collision_map function would permute the outputs of the recursive_case function, rotating the three-dimensional space back to its original orientation.

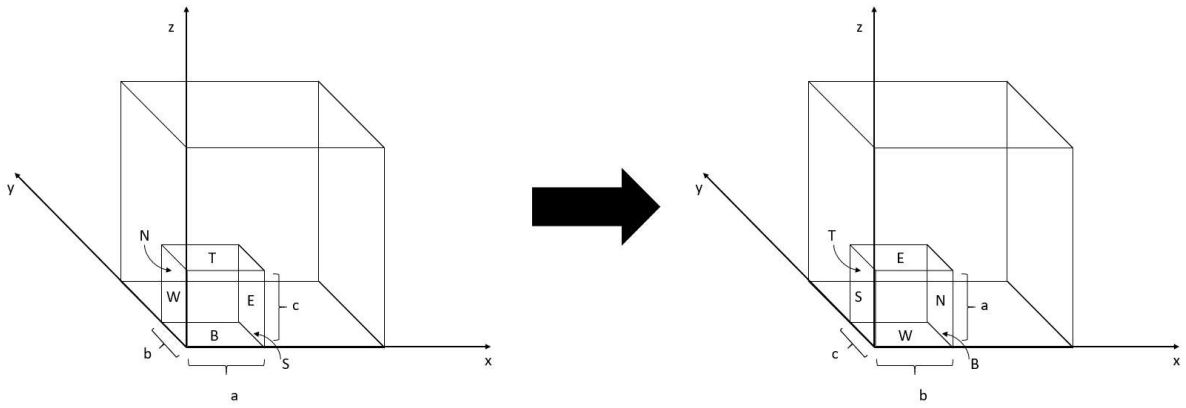

Figure 6: Permuting 3D space so that the WEST face of the scatterer takes the place of BOTTOM. Scatterer faces and dimensions are relabeled accordingly.

Note that when we rotate the space the coordinates of the point particle, the lengths of the scatterer sides, the components of the velocity vector, and the side variable must all be permuted. We permute these values before inputting them to the recursive_case function, and then apply the inverse permutation to the output from the recursive_case function. To permute the side variable, we use the map described above, mapping the side to the corresponding tuple, permuting the elements of the tuple, and then mapping the result back to a side value.

The code for recursive_case has four parts, handling four different cases. The first case is the case where the point particle is currently on a scatterer. The second case is the case where the next collision point along the particle's trajectory is on the TOP face of a scatterer, and the third case is when the next collision point is on the NORTH or EAST face of a scatterer. The

final case is when the next collision point is also a recursive case, on side BOTTOM, WEST, or SOUTH of a scatterer.

Recursive Case Part 1: The Point Particle is on a Scatterer

The code for the recursive case first checks if the point particle is currently on a face of a scatterer. Since recursive_case assumes the case of BOTTOM, this involves checking if the point is on the BOTTOM face of a scatterer. Computationally this means checking if the x-coordinate of the point modulo 1 is less than a, the width of the scatterer, and the y-coordinate modulo 1 is less than b, the depth of the scatterer in the y-direction.

If the point is on a BOTTOM face of the scatterer, we reflect the point off the scatterer, changing the sign of the z-component of the velocity vector. We then return the unchanged x, y, and z values, the new side value which is BOTTOM, the updated velocity vector, and the boolean value True indicating that we have reached a scatterer along the point's trajectory. These position, side, and velocity values will be permuted by the calling collision_map cases for cases other than BOTTOM.

Recursive Case Part 2: The Next Collision Point is on Side TOP

The next eventuality that recursive_case considers is the case where the point is heading downwards in the negative z direction and the next collision point is on the TOP face of a scatterer. We consider this case first because the point is currently on a BOTTOM face, so if we follow the velocity vector leading from the point particle and it intersects with several scatterer faces, the first scatterer face it reaches will be a TOP face. For example, see Figure 7 below with

an initial point in red and its velocity vector in blue. In this figure the point's velocity vector intersects with TOP and EAST faces of the scatterer in its path, but TOP is reached first.

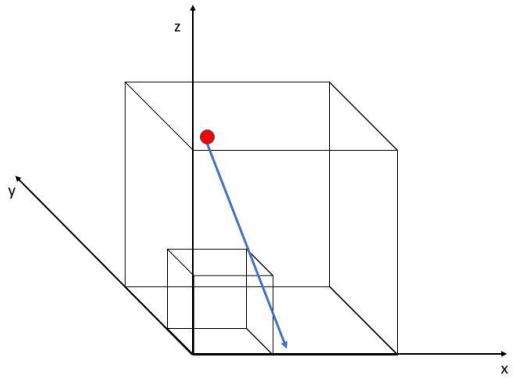

Figure 7: The next collision point is on the TOP of a scatterer.

To compute whether the next collision point is on the TOP face of a scatterer, we calculate the time to travel a vertical distance of 1-c as (1-c) / v[2]. We increment the position of the point by this scalar value, and then check if we are on the TOP of a scatterer. If so, the velocity vector is reflected, the side variable becomes TOP, and the new position, side, and velocity are returned along with a done value of True since we have reached a scatterer.

Recursive Case Part 3: The Next Collision Point is on Side EAST or NORTH

If the point did not hit TOP, we then check whether its next collision will be on the EAST or NORTH face of a scatterer. For example, see Figure 8 below. In this pictured case, after traveling a vertical distance of 1 - c the point is not on a TOP face of a scatterer, so it continues traveling and reaches the EAST face of a scatterer. It is possible for the velocity vector to intersect with both EAST and NORTH faces of the next scatterer in its path, so in addition to calculating whether each intersection occurs, in the case where both intersections occur we must also calculate which happens first.

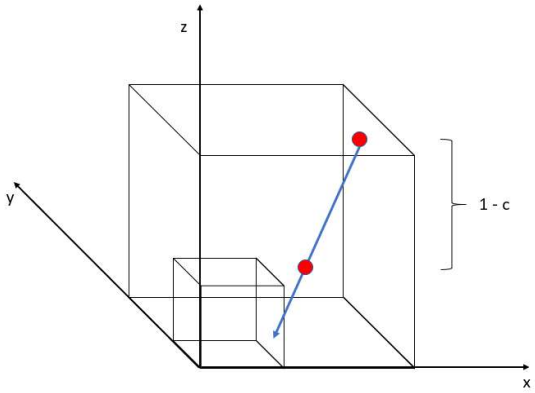

Figure 8: The next collision point is on the EAST of a scatterer.

The code to calculate whether intersections with EAST and NORTH faces take place is similar to the code that checks for an intersection with TOP. To calculate which intersection happens first in the case where they both take place, we simply compare the scalars which represent the length of time until each collision. The return value in the cases where the point reaches EAST or NORTH are similar to the return value for TOP, and they also have a done value of True.

Recursive Case Part 4: The Next Collision Point is Also a Recursive Case

This final category of cases dealt with by the recursive_case function is when the point particle is not currently on a scatterer and its next collision point is not on the TOP, EAST, or NORTH of a scatterer. In other words, after the point travels until its x-coordinate is a, y-coordinate is b, or z-coordinate is c, it still has not reached a scatterer. In this case, we follow the velocity vector of the point until it reaches a face of the unit cube which will be BOTTOM, WEST, or SOUTH. Since this point is not necessarily on a scatterer, it is not the next collision point along the trajectory of the point particle but is just an intermediate step in calculating it.

This point is returned to the collision_map function which will pass it back to the recursive_case function to continue recursing.

In this case, we calculate the time it will take for the point to reach the next integer x-plane, y-plane, and z-plane, and then find the smallest of the three. In calculating the time to reach the next x-plane and y-plane, we need to take into account whether the velocity vector is pointing in the negative or positive direction along the x and y axes so that we can find the distance from the point to the plane. In the z-direction, however, since we are currently on side BOTTOM, on an integer z-plane, we know that the distance to the next integer z-plane will be 1, regardless of direction. Once we have calculated the distance from the point to the next integer plane in each direction, we divide the distance by the corresponding component of the velocity vector to yield the time it will take to reach that plane.

Once we have the three times, we find the minimum of the three, and then increment the position of our point by the distance corresponding to that amount of time. We update the side variable according to our determination of which plane was reached first: for example, if the next integer plane reached is a y-plane, then the new side is SOUTH. Because we have not collided with a scatterer, the velocity vector remains unchanged and the done variable is False.

Modeling a Wind-Tree Trajectory

The collision_map and recursive_case functions model the behavior of the wind-tree model and constitute the bulk of our code. The next step in our project was to repeatedly call the collision_map function to generate a trajectory of points and to study the long-term behavior of the wind-tree model. A segment of a sample trajectory is shown in Figure 9. (Scatterers are omitted so as not to obstruct the view of the trajectory.)

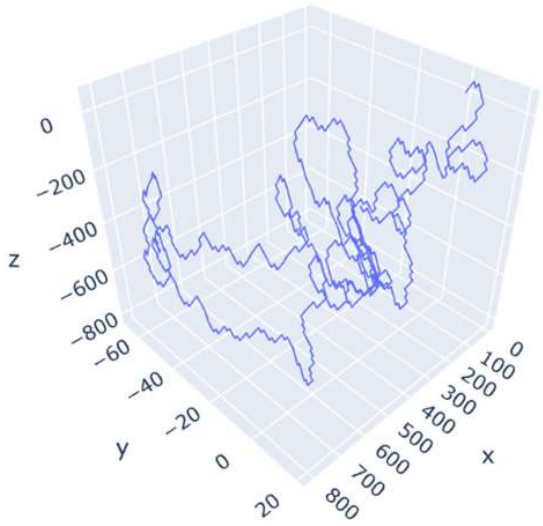

Figure 9: A sample 3D wind-tree trajectory.

Measuring Statistical Properties of the Wind-Tree Model

We tested several methods of estimating diffusion rates of the wind-tree model. As described above, the diffusion rate is the limit superior of the log distance traveled by the point divided by the log amount of time elapsed. We wrote code, mainly consisting of trigonometric calculations, to simulate the two-dimensional wind-tree model, for which the diffusion rate has already been shown to be ⅔. We were then able to use this code to determine which simulation method was best for approximating diffusion rates by determining which result came closest to ⅔.

In the first method we used to calculate the diffusion rate of the wind-tree model, we computed a trajectory of collisions starting at a randomly chosen initial point. With each collision we incremented time, and whenever time was a power of ten we logged the distance from the initial point. We then plotted the logs of the distances versus the logs of the times at

which they were recorded, fit a line to the plot using numpy's least squares polynomial fit with degree 1, and estimated the diffusion rate as the slope of this line. Figure 10 below shows a plot with a few of these lines, each using 10,000,000 (10^7) iterations of collision_map. In this log-log plot, with powers of time on the x-axis, lines 1, 2, and 3 have slopes of approximately 0.6697, 0.8025, and 0.6460 respectively.

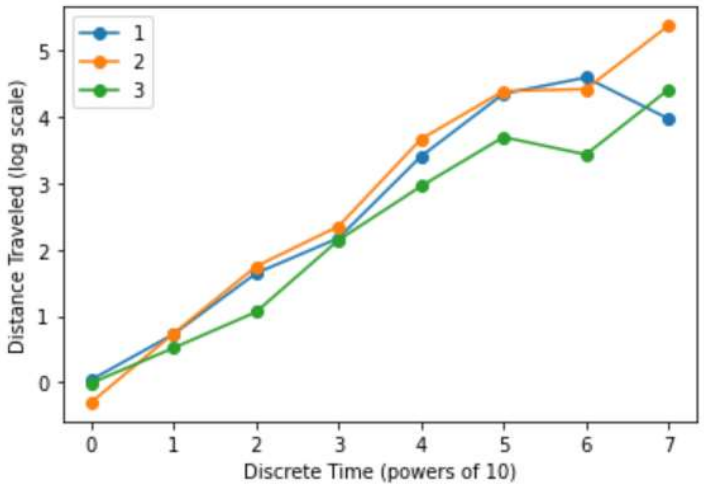

Figure 10: Estimations of the diffusion rate of the wind-tree model. This is a log-log plot with time, in powers of 10, on the x-axis, and distance, also on a log scale, on the y-axis. Lines 1, 2, and 3 have slopes of approximately 0.6697, 0.8025, and 0.6460 respectively.

In the second method we used, we calculated the exponent directly as log of distance divided by log of time using the distance and time from the end of the trajectory. We tested both of these methods of computing diffusion rates using both limits and limit superiors. We found that of these four combinations (slope vs direct calculation and limit vs limit superior), the most accurate method was estimating the diffusion rate as the limit superior of the directly calculated exponent.

Simulating the Wind-Tree Model in Higher Dimensions

The next step in our project was to expand the concepts from the three-dimensional wind-tree code into higher dimensions. We translated the three-dimensional code to a higher level of abstraction, enabling the code to compute wind-tree trajectories for any dimension d. We use lists and for loops to generalize the concepts from our three-dimensional wind-tree code. For example, just as our three-dimensional wind-tree code has a function increment_by_scalar which updates inputs x, y, and z with an input scalar, our d-dimensional code has a function increment_by_scalar which updates all the elements of input list pos by an input scalar using a for loop.

Just like our three-dimensional code, our d-dimensional wind-tree code is split into collision_map and recursive_case functions. The inputs to the d-dimensional collision_map function are d, scatterer_sizes, pos, v, side, and side_bool. d is the dimension of the wind-tree model. scatterer_sizes is a list of length d describing the dimensions of the scatterers in the model. The components of this list would correspond to a, b, and c in our three-dimensional code. pos describes the initial position of the point particle. This is also represented as a list of d elements, corresponding to x, y, and z in our three-dimensional code. v, also a list of length d, is the velocity vector of the point.

In our three-dimensional code, we mapped sides of the scatterer to tuples. Each of these tuples had one non-zero element which was either 1 or -1, depending whether the side represented a base case or a recursive case. In our d-dimensional code, we generalize this idea. We use the variable side, an integer between 0 and d-1, to indicate which element is non-zero. The boolean variable side_bool then indicates whether the non-zero value is 1 or -1. A side_bool

value of True indicates that the non-zero element of side is 1, and we are in a base case. A side_bool value of False indicates that the non-zero element is -1 and we are in a recursive case. These two variables, side and side_bool, are sufficient to represent the side tuple instead of storing all of its elements.

The collision_map code operates similarly to the three-dimensional code. In a base case, the code follows the velocity vector of the point until it reaches a boundary of the unit d-dimensional cube enclosing it. (This will be a square in dimension two, cube in dimension three, hypercube in dimension four, etc.) The time the point travels along its velocity vector until it reaches a boundary is calculated as (1 - scatterer_sizes[side]) / v[side]. The position of the point is incremented by this scalar amount, and the side_bool variable becomes False since the point is no longer on a scatterer.

In a recursive case, the code rotates the space to line up with the case where side equals d-1 (the d-dimensional equivalent of BOTTOM). This is accomplished with a function called permute which rotates the elements in the lists scatterer_sizes, pos, v, and the implied side tuple. The collision_map function passes the permuted values to recursive_case which calculates the next point and returns it to collision_map. Then collision_map rotates the elements in scatterer_sizes, pos, v, and side again to restore the space to its original orientation.

The code for recursive_case is similar to the recursive_case function in our three-dimensional code. It is divided into four cases which parallel the four cases in the three-dimensional code, and uses lists and for loops to generalize concepts from 3D to any dimension. Our code to generate trajectories in higher dimensions is also similar to the parallel three-dimensional code.

Results

We ran our d-dimensional wind-tree code to produce trajectories and predict diffusion rates for the wind-tree model in several dimensions and with various numbers of iterations and sample sizes. We found that for a given dimension and number of iterations, diffusion rate estimations varied widely between samples, indicating that the number of iterations that our computers could compute in a reasonable amount of time is insufficient for the diffusion rate of the model to converge. We therefore report average diffusion rate estimations, averaged over several samples, together with standard deviations of the results. The results of several runs of our code are in Table 1.

Table 1: Simulations of the Wind-Tree Model

Dimension	Sample Size	Iterations per Sample, log base 10	Average Estimated Diffusion Rate (lim sup)	Estimated Diffusion Rate Standard Deviation (lim sup)	Average Estimated Diffusion Rate (lim)	Estimated Diffusion Rate Standard Deviation (lim)
2	10	8	0.6619	0.0604	---	---
2	10	8	0.7274	0.0839	0.6539	0.0776
2	5	9	0.7433	0.0970	0.6817	0.1208
3	10	7	---	---	0.6315	0.0628
3	10	8	0.6479	0.0639	0.5933	0.0867
3	5	9	0.7011	0.0645	0.6109	0.0669
4	10	5	0.9650	0.0341	0.8234	0.0660
4	10	6	0.6676	0.0331	0.6070	0.0449
4	10	7	---	---	0.6180	0.0474
6	10	5	0.9758	0.0282	0.8695	0.0392

Since with increasing dimension the speed of the code decreases, we ran the code with smaller numbers of iterations per sample in higher dimensions. Smaller numbers of iterations also corresponded to increased standard deviations, so these results are less reliable.

Conclusion and Directions for Future Work

Our code confirms the ⅔ diffusion rate of the two-dimensional wind-tree model proven by Delecroix, Hubert, and Lelièvre [4]. In higher dimensions, our code generally shows a trend that diffusion rates of the wind-tree model increase with increasing dimension. We cannot provide precise estimations of the diffusion rate in higher dimensions because our code runs slowly in high dimensions and the diffusion rate of the wind-tree model converges slowly.

Future work in this area could optimize our code to run faster in order to generate more precise estimates of the diffusion rate of the wind-tree model in higher dimensions. For example, plots of wind-tree trajectories frequently display repeating shapes along the trajectory (see Figure 11). Code could be written to take advantage of this structure and produce wind-tree trajectories in sub-linear time. Additionally, future research could attempt to prove diffusion rates of the higher-dimensional wind-tree model mathematically instead of by computer simulation.

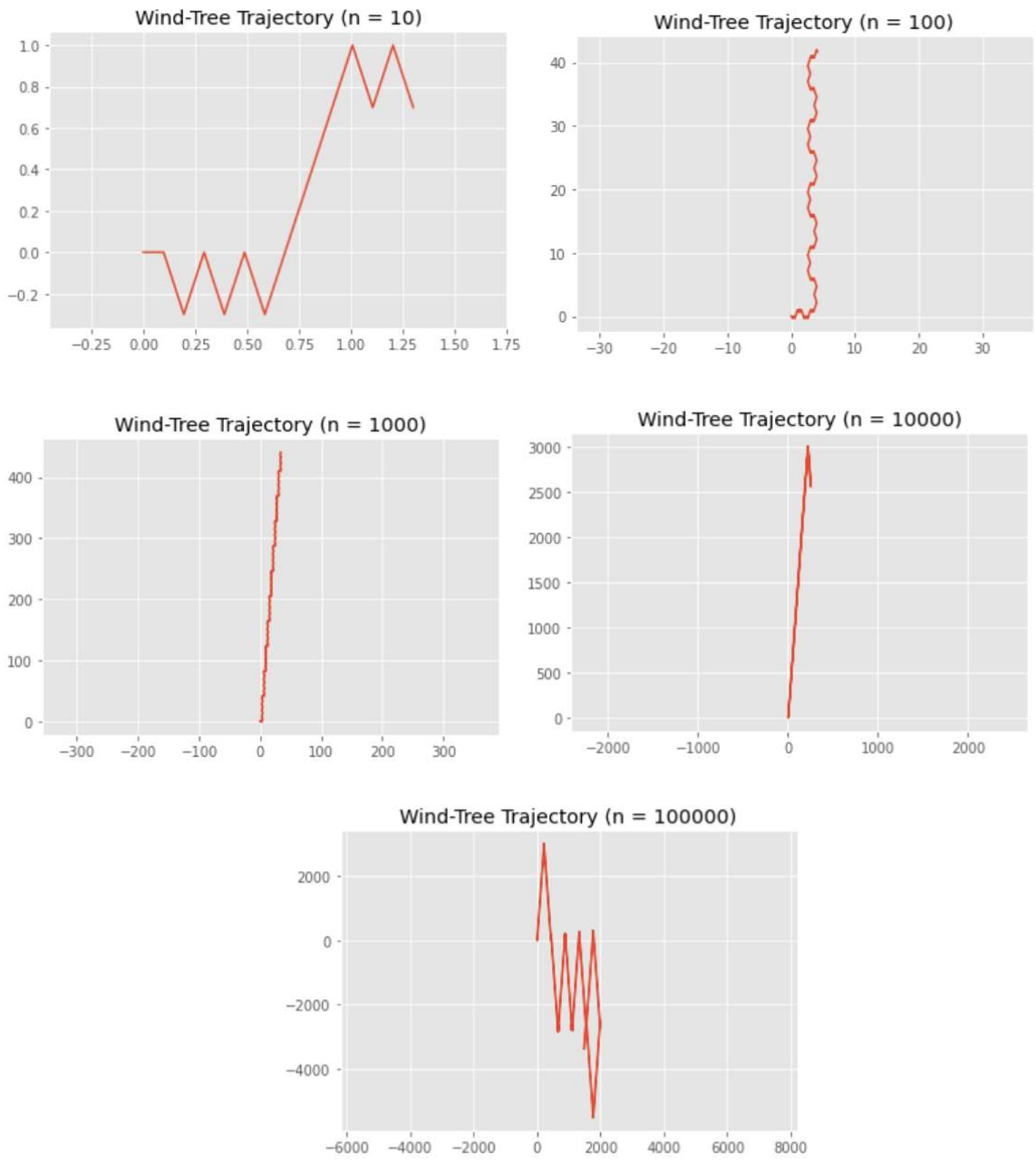

Figure 11: Several wind-tree trajectories with fixed initial conditions (start point, velocity, and scatterer size) and an increasing number of iterations, n.

Acknowledgements

Thank you to my thesis advisor and mentor Professor Peter Nandori for all of his encouragement and assistance throughout the past few months. I am grateful to Professor Nandori for explaining and reexplaining concepts to me, and for guiding me throughout the research process. It has been an honor and pleasure to work with him, and I have learned so much from this experience.

Thank you to my mathematics and computer science professors at Stern College for all that they have taught me and for the time and energy that they invest in their students. Thank you to Dr. Cynthia Wachtell for directing the honors program, which I feel privileged to be a part of.

Last but not least, thank you to my family for their support as I've worked on this project. I am so grateful for everything that they do for me always.

References Cited

[1] Avila, A., & Hubert, P. (2020). Recurrence for the wind-tree model. *Annales De L'Institut Henri Poincaré C, Analyse Non Linéaire, 37*(1), 1-11. doi:10.1016/j.anihpc.2017.11.006

[2] Bonanno, C. (2018). Infinite Ergodic Theory. Retrieved April 25, 2021, from http://www.crm.sns.it/media/event/432/Bonanno-pisahokkaido2018.pdf

[3] Dajani, K., & Dirksin, S. (2008, December 18). A Simple Introduction to Ergodic Theory. Retrieved April 25, 2021, from https://webspace.science.uu.nl/~kraai101/lecturenotes2009.pdf

[4] Delecroix, V., Hubert, P., & Lelièvre, S. (2014). Diffusion for the periodic wind-tree model. *Annales Scientifiques De L'École Normale Supérieure, 47*(6), 1085-1110. doi:10.24033/asens.2234

[5] Ehrenfest, P., & Ehrenfest, T. (1912). Begriffliche Grundlagen der statistischen Auffassung in der Mechanik. *Encykl. D. Math. Wissench.* IV2 II Heft (6), 90. (in German, translated in:) The conceptual foundations of the statistical approach in mechanics. Cornell University Press, Ithaca (1959), 10-13

[6] Frączek, K., & Ulcigrai, C. (2013). Non-ergodic Z-periodic billiards and infinite translation surfaces. *Inventiones Mathematicae, 197*(2), 241-298. doi:10.1007/s00222-013-0482-z

[7] Hardy, J., & Weber, J. (1980). Diffusion in a periodic wind-tree model. *Journal of Mathematical Physics, 21*(7), 1802-1808. doi:10.1063/1.524633

[8] Kerckhoff, S., Masur, H., & Smillie, J. (1986). Ergodicity of billiard flows and quadratic differentials. *The Annals of Mathematics, 124*(2), 293. doi:10.2307/1971280

Appendix A: Three-Dimensional Windtree Code

```
from enum import Enum, auto
import math
import random
import plotly.express as px
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt


# enum to represent side as a member of a discrete set of options
class Side(Enum):
    NORTH = auto()
    SOUTH = auto()
    WEST = auto()
    EAST = auto()
    TOP = auto()
    BOTTOM = auto()


side_to_num = {Side.NORTH: (0, 1, 0),
               Side.SOUTH: (0, -1, 0),
               Side.WEST: (-1, 0, 0),
               Side.EAST: (1, 0, 0),
               Side.TOP: (0, 0, 1),
               Side.BOTTOM: (0, 0, -1)}

num_to_side = {v: k for k, v in side_to_num.items()}


def increment_by_scalar(x, y, z, v, scalar):
    x += scalar * v[0]
    y += scalar * v[1]
    z += scalar * v[2]
    return (x, y, z)


'''
a, b, and c are the side lengths of the scatterer in the x, y, and z
directions
x, y, and z are the coordinates of the billiard in R3
v is the unit vector indicating the direction of the billiard's motion

There are 3 base cases - the billiard is on the scatterer but not on an
edge of the unit cube that the scatterer is in
There are 3 recursive cases - that the billiard is on one of the 3 faces
of the cube

NORTH, EAST, and TOP refer to sides of the scatterer.  SOUTH, WEST, and
BOTTOM refer to faces of the unit cube.  Therefore, when side is TOP, v[2]
is always positive, but when side is BOTTOM v[2] can be positive or
negative.
```

```
'''

def collision_map(a, b, c, x, y, z, side, v):

    while True:

        # Base Cases - starting on the scatterer in the middle of the cube

        # Top heading up
        if side == Side.TOP:
            scalar = (1 - c) / v[2]
            x, y, z = increment_by_scalar(x, y, z, v, scalar)
            side = Side.BOTTOM

        # East heading right
        elif side == Side.EAST:
            scalar = (1 - a) / v[0]
            x, y, z = increment_by_scalar(x, y, z, v, scalar)
            side = Side.WEST

        # North heading up
        elif side == Side.NORTH:
            scalar = (1 - b) / v[1]
            x, y, z = increment_by_scalar(x, y, z, v, scalar)
            side = Side.SOUTH

        # Recursive cases

        # Top/Bottom
        if side == Side.BOTTOM:
            x, y, z, side, v, done = recursive_case(x, y, z, a, b, c,
side, v)

        # North/South
        elif side == Side.SOUTH:
            z, x, y, side, v, done = recursive_case(z, x, y, c, a, b,
side, [v[2], v[0], v[1]])
            v = [v[1], v[2], v[0]]
            side_value = side_to_num[side]
            permuted_side = (side_value[1], side_value[2], side_value[0])
            side = num_to_side[permuted_side]

        # West/East
        elif side == Side.WEST:
            y, z, x, side, v, done = recursive_case(y, z, x, b, c, a,
side, [v[1], v[2], v[0]])
            v = [v[2], v[0], v[1]]
            side_value = side_to_num[side]
            permuted_side = (side_value[2], side_value[0], side_value[1])
            side = num_to_side[permuted_side]


        if done:
            return x, y, z, side, v
```

```
# RECURSIVE CASE IS WRITTEN FOR CASE OF BOTTOM, WILL USE PERMUTED INPUT
# FOR OTHER DIRECTIONS.  RECURSIVE CASE DEALS WITH CASES OF GOING UP OR
# DOWN.
def recursive_case(x, y, z, a, b, c, side, v):

    if x % 1 < a and y % 1 < b and v[2] > 0:
        v[2] = -v[2]
        # This is BOTTOM because the recursive case is written for the
        # case of BOTTOM; the output will be permuted for other cases
        return (x, y, z, Side.BOTTOM, v, True)

    # First, check if we hit the scatterer

    if v[2] < 0:
        # Hitting top of scatterer
        # (May be a point in a different cube)
        c_scalar = (c - 1) / v[2]
        C = increment_by_scalar(x, y, z, v, c_scalar)
        if (0 < C[0] % 1 < a) and (0 < C[1] % 1 < b):
            v[2] = -v[2]
            return (C[0], C[1], C[2], Side.TOP, v, True)

    # If we didn't hit the top of the scatterer, check if we hit east or
    # north. If we hit both, we must check which is closer.

    hit_east = False

    # If we are heading left, towards east of the scatterer
    # This works regardless of whether v[2] is + or -
    if v[0] < 0 and x % 1 > a:
        a_scalar = (a - x % 1) / v[0]
        A = increment_by_scalar(x, y, z, v, a_scalar)
        if (0 < A[1] % 1 < b) and (0 < A[2] % 1 < c):
            hit_east = True
            v[0] = -v[0]
            a_return = (A[0], A[1], A[2], Side.EAST, v, True)

    # Hitting north of scatterer
    # This works regardless of whether v[2] is + or -
    if v[1] < 0 and y % 1 > b:
        b_scalar = (b - y % 1) / v[1]
        B = increment_by_scalar(x, y, z, v, b_scalar)

        if (0 < B[0] % 1 < a) and (0 < B[2] % 1 < c):
            v[1] = -v[1]
            b_return = (B[0], B[1], B[2], Side.NORTH, v, True)

            if not hit_east or (hit_east and b_scalar < a_scalar):
                return b_return

    if hit_east:
        return a_return
```

```
    # If we didn't hit the scatterer, then we now need to see which edge
    # of the cube we will hit next to continue the recursion - find the
    # closest of the 3 possibilities (p, q, r)

    # hitting next x-plane
    if v[0] > 0:
        p_scalar = (1 - x % 1) / v[0]
    else:
        p_scalar = - (x % 1) / v[0]

    # hitting next y-plane
    if v[1] > 0:
        q_scalar = (1 - y % 1) / v[1]
    else:
        q_scalar = - (y % 1) / v[1]

    # hitting next z-plane
    r_scalar = abs(1 / v[2])

    if p_scalar <= q_scalar and p_scalar <= r_scalar:
        (x, y, z, side) = increment_by_scalar(x, y, z, v, p_scalar) +
(Side.WEST,)
    elif q_scalar <= p_scalar and q_scalar <= r_scalar:
        (x, y, z, side) = increment_by_scalar(x, y, z, v, q_scalar) +
(Side.SOUTH,)
    else:
        (x, y, z, side) = increment_by_scalar(x, y, z, v, r_scalar) +
(Side.BOTTOM,)

    # recursive case, continue to the top of the while loop
    return (x, y, z, side, v, False)


def compute_trajectory(a, b, c, x, y, z, v, iterations):
    trajectory = [(x, y, z)]
    side = Side.TOP
    for i in range(iterations):
        x, y, z, side, v = collision_map(a, b, c, x, y, z, side, v)
        trajectory.append((x, y, z))
    return trajectory


def norm(x1, y1, z1, x2, y2, z2):
    return math.sqrt((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)


def plot_trajectory(trajectory):
    df = pd.DataFrame(trajectory, columns =['x', 'y', 'z'])
    fig = px.line_3d(df, x="x", y="y", z="z")
    fig.show()


def compute_exp_with_limsup(sample_size, iterations):
```

```
    a = random.random()
    b = random.random()
    c = random.random()
    limsup_exponents = []
    lim_exponents = []

    for i in range(sample_size):
        start_x = random.uniform(0, a)
        start_y = random.uniform(0, b)
        start_z = c
        x = start_x
        y = start_y
        z = start_z

        # choose vector randomly
        first = random.gauss(0, 1)
        second = random.gauss(0, 1)
        third = random.gauss(0, 1)
        length = math.sqrt(first ** 2 + second ** 2 + third ** 2)
        v = [first / length, second / length, abs(third / length)]

        side = Side.TOP
        max_exp = 0

        for t in range(1, 10**7 + 1):
            x, y, z, side, v = collision_map(a, b, c, x, y, z, side, v)

        for t in range(10**7 + 1, iterations + 1):
            x, y, z, side, v = collision_map(a, b, c, x, y, z, side, v)
            distance = norm(start_x, start_y, start_z, x, y, z)
            exp = math.log10(distance) / math.log10(t)
            if exp > max_exp:
                max_exp = exp

        limsup_exponents.append(max_exp)

        distance = norm(start_x, start_y, start_z, x, y, z)
        exp = math.log10(distance) / math.log10(iterations)
        lim_exponents.append(exp)

    print("Computing exponent using limsup and lim")
    print("Limsup exponents are: ", limsup_exponents)
    print("Limit exponents are: ", lim_exponents)
    print("Average limsup exponent is: ",
sum(limsup_exponents)/len(limsup_exponents))
    print("Average limit exponent is: ",
sum(lim_exponents)/len(lim_exponents))
    print()
```

Appendix B: d-Dimensional Windtree Code

```
import math
import random
import sys


def increment_by_scalar(d, pos, v, scalar):
    pos2 = [None] * d
    for i in range(d):
        pos2[i] = pos[i] + scalar * v[i]
    return pos2


'''
cycle elements clockwise
E.g. [a, b, c] -> permute(1) -> [c, a, b]
'''
def permute(d, array, amount):
    array2 = array.copy()
    for i in range(amount):
        temp = array2[-1]
        for j in range(d-1, 0, -1):
            array2[j] = array2[j-1]
        array2[0] = temp
    return array2


'''
d = dimension
scatterer_sizes = list of length d - e.g. [a, b, c]
pos = position, list of length d - e.g. [x, y, z]
v = direction, unit vector - a list of length d
side = an integer between 0 and d-1 indicating which element of side is
non-zero
side_bool = a boolean indicating whether
    - true - the non-0 element of side is 1, we're in a base case
    - false - the non-0 element of side is -1, we're in a recursive case
'''
def collision_map(d, scatterer_sizes, pos, v, side, side_bool):

    while True:

        # Base Cases - starting on the scatterer in the middle of the cube
        if side_bool:
            scalar = (1 - scatterer_sizes[side]) / v[side]
            pos = increment_by_scalar(d, pos, v, scalar)
            side_bool = False

        # Recursive cases
        amount = side + 1
        inverse_amount = d - amount
        inv_pos = permute(d, pos, inverse_amount)
```

```
        inv_v = permute(d, v, inverse_amount)
        inv_scatterers = permute(d, scatterer_sizes, inverse_amount)
        pos, side, side_bool, v, done = recursive_case(d, inv_pos,
inv_scatterers, inv_v)
        pos = permute(d, pos, amount)
        v = permute(d, v, amount)
        # cycle side by amount
        side = (side + amount) % d

        if done:
            return pos, side, side_bool, v


# RECURSIVE CASE
def recursive_case(d, pos, scatterer_sizes, v):

    # If we're on a scatterer, flip angle and return
    on_scatterer = True
    for i in range(d-1):
        if pos[i] % 1 > scatterer_sizes[i]:
            on_scatterer = False
            break
    if on_scatterer and v[d-1] > 0:
        v[d-1] = -v[d-1]
        new_side = d-1
        new_side_bool = False
        return (pos, new_side, new_side_bool, v, True)


    # Check if we hit the scatterer
    scalars = [None] * d
    new_pts = [None] * d
    hit_side = [False for i in range(d)]

    # First check if we hit the equivalent of TOP
    if v[d-1] < 0:
        scalars[d-1] = (scatterer_sizes[d-1] - 1) / v[d-1]
        new_pts[d-1] = increment_by_scalar(d, pos, v, scalars[d-1])
        hit_side[d-1] = True
        for i in range(d-1):
            if not (0 < new_pts[d-1][i] % 1 < scatterer_sizes[i]):
                hit_side[d-1] = False
                break
        if hit_side[d-1]:
            v[d-1] = -v[d-1]
            return(new_pts[d-1], d-1, True, v, True)

    # If we don't hit top, check if we hit any other side of the
    # scatterer, and if we hit a few, find the hit side that has the
    # minimum scalar (time)
    for i in range(d-1):

        if v[i] < 0 and pos[i] % 1 > scatterer_sizes[i]:
```

```
            scalars[i] = (scatterer_sizes[i] - pos[i] % 1) / v[i]
            new_pts[i] = increment_by_scalar(d, pos, v, scalars[i])

            hit_side[i] = True
            for j in range(d):
                if i != j:
                    if not (0 < new_pts[i][j] % 1 < scatterer_sizes[j]):
                        hit_side[i] = False
                        break


    # find the hit side that has the minimum scalar
    min_scalar = sys.maxsize
    closest_side = None
    for i in range(d):
        if hit_side[i]:
            if scalars[i] < min_scalar:
                min_scalar = scalars[i]
                closest_side = i
    if min_scalar != sys.maxsize:
        v[closest_side] = -v[closest_side]
        return (new_pts[closest_side], closest_side, True, v, True)


    # If we didn't hit the scatterer, then we now need to see which edge
    # of the cube we will hit next to continue the recursion - find the
    # closest of the possibilities and recurse

    scalars = [None] * d
    for i in range(d-1):
        if v[i] > 0:
            scalars[i] = (1 - pos[i] % 1) / v[i]
        else:
            scalars[i] = - (pos[i] % 1) / v[i]
    scalars[d-1] = abs(1 / v[d-1])

    min_scalar = sys.maxsize
    closest_side = None
    for j in range(d):
        if scalars[j] < min_scalar:
            min_scalar = scalars[j]
            closest_side = j
    new_pt = increment_by_scalar(d, pos, v, min_scalar)
    return (new_pt, closest_side, False, v, False)


''' Initialize scatterer sizes '''
def initializing_scatterers(d):
    scatterer_sizes = []
    for i in range(d):
        scatterer_sizes.append(random.random())
    return scatterer_sizes
```

```
''' Initialize position, start vector, side, and side_bool '''
def initializing(d, scatterer_sizes):
    start_pos = []
    for i in range(d-1):
        start_pos.append(random.uniform(0, scatterer_sizes[i]))
    start_pos.append(scatterer_sizes[d-1])

    v_dims = []
    for i in range(d):
        v_dims.append(random.gauss(0, 1))
    length = 0
    for i in range(d):
        length += v_dims[i] ** 2
    length = math.sqrt(length)
    v = []
    for i in range(d-1):
        v.append(v_dims[i] / length)
    v.append(abs(v_dims[d-1] / length))

    side = d-1
    side_bool = True

    return d, start_pos, v, side, side_bool


''' Calculate distance between two points in dimension d '''
def norm(d, pos1, pos2):
    sum = 0
    for i in range(d):
        sum += (pos2[i] - pos1[i]) ** 2
    return math.sqrt(sum)


def compute_exp_with_limsup(dim, sample_size, iterations):
    scatterer_sizes = initializing_scatterers(dim)
    limsup_exponents = []
    lim_exponents = []

    for i in range(sample_size):
        d, start_pos, v, side, side_bool = initializing(dim,
scatterer_sizes)
        pos = start_pos.copy()
        max_exp = 0

        for t in range(1, 10**7 + 1):
            pos, side, side_bool, v = collision_map(d, scatterer_sizes,
pos, v, side, side_bool)

        for t in range(10**7 + 1, iterations + 1):
            pos, side, side_bool, v = collision_map(d, scatterer_sizes,
pos, v, side, side_bool)
            distance = norm(dim, start_pos, pos)
            exp = math.log10(distance) / math.log10(t)
            if exp > max_exp:
```

```
                max_exp = exp

        limsup_exponents.append(max_exp)
        print("limsup exp: ", max_exp)

        distance = norm(dim, start_pos, pos)
        exp = math.log10(distance) / math.log10(iterations)
        lim_exponents.append(exp)

        print("lim exp: ", exp)

    print("Computing exponent using limsup and lim in dimension", dim,
"with sample size", sample_size, "and", iterations, "iterations")
    print("Limsup exponents are: ", limsup_exponents)
    print("Limit exponents are: ", lim_exponents)
    print("Average limsup exponent is: ",
sum(limsup_exponents)/len(limsup_exponents))
    print("Average limit exponent is: ",
sum(lim_exponents)/len(lim_exponents))
    print()


if __name__ == '__main__':

    dim = int(input("Enter dimension number: "))
    sample_size = int(input("Enter sample size: "))
    iterations = int(input("Enter number of iterations: "))

    compute_exp_with_limsup(dim, sample_size, iterations)
```