Dynamical-Billiards

A THESIS

submitted in partial fulfillment of the requirements

for the award of the dual degree of

Bachelor of Science - Master of Science

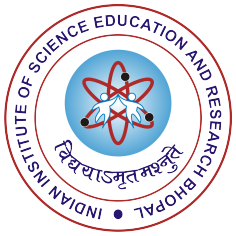
in

PHYSICS

by

Abhishek Nandekar

(14004)



DEPARTMENT OF PHYSICS
INDIAN INSTITUTE OF SCIENCE EDUCATION AND RESEARCH BHOPAL
BHOPAL - 462 066

April 2019

CERTIFICATE

This is to certify that Abhishek Nandekar, BS-MS (Dual Degree) student in Department of Physics, has completed bonafide work on the dissertation entitled ‘Dynamical Billiards’ under my supervision and guidance.

April 2019

IISER Bhopal

Dr. Kushal Shah

Committee Member	Signature	Date

ACADEMIC INTEGRITY AND COPYRIGHT DISCLAIMER

I hereby declare that this MS-Thesis is my own work and, to the best of my knowledge, that it contains no material previously published or written by another person, and no substantial proportions of material which have been accepted for the award of any other degree or diploma at IISER Bhopal or any other educational institution, except where due acknowledgement is made in the document.

I certify that all copyrighted material incorporated into this document is in compliance with the Indian Copyright (Amendment) Act (2012) and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of that law. I agree to indemnify and safeguard IISER Bhopal from any claims that may arise from any copyright violation.

April 2019

IISER Bhopal

Abhishek Nandekar

ACKNOWLEDGEMENT

Vivamus vehicula leo a justo. Quisque nec augue. Morbi mauris wisi, aliquet vitae, dignissim eget, sollicitudin molestie, ligula. In dictum enim sit amet risus. Curabitur vitae velit eu diam rhoncus hendrerit. Vivamus ut elit. Praesent mattis ipsum quis turpis. Curabitur rhoncus neque eu dui. Etiam vitae magna. Nam ullamcorper. Praesent interdum bibendum magna. Quisque auctor aliquam dolor. Morbi eu lorem et est porttitor fermentum. Nunc egestas arcu at tortor varius viverra. Fusce eu nulla ut nulla interdum consectetuer. Vestibulum gravida. Morbi mattis libero sed est.

Abhishek Nandekar

ABSTRACT

Understanding the ergodic properties of billiards with varying geometrical features is of immense interest in the domain of dynamical systems. In particular, although it is known that rational polygonal billiards are non-ergodic, the ergodic properties of irrational polygonal billiards are currently unknown and it is a very important open question with potential applications in the areas of statistical equilibration, plasma physics, optical traps and mesoscopic systems. We have taken an indirect approach to understand the ergodic properties of irrational polygons by trying to develop certain Machine Learning protocols to classify rational vs irrational polygons into two different categories.

CONTENTS

1. INTRODUCTION

1.1 General Introduction

Dynamical Billiards are Hamiltonian idealizations of the game of Billiards, wherein the containing region may not always be rectangular in shape, and can also be multidimensional. The particle (the ball) usually possess a unit speed. It undergoes elastic collisions with the boundaries, resulting in regular reflections and a constant Kinetic Energy.

Dynamical Billiards are very important in many areas of physics including nonlinear dynamics, statistical mechanics, quantum optics and even electromagnetics. Billiards in polygons are essentially studied due to their lucid nature and protean usage.

As shown in [1], they form natural models in many problems of optics,acoustics and classical mechanics. The most prominent model of statistical mechanics, the Boltzmann gas of elastically colliding hard balls in a box can be easily reduced to a billiard. 2D oscillating billiards are also used as a basis to study the exponential energy growth observed in Fermi accelerators [7] as well as in the study of statistical equilibration [6]. Many properties of quantum dynamics of billiards are also closely related to the properties of corresponding classical problem.

Many classes of billiards also demonstrate chaotic behavior. As shown in [3], billiards with strongest chaotic properties laid a foundation for the analysis of ergodic and statistical properties of hyperbolic dynamical systems with singularities. [2]

1.2 Polygonal Billiards

Billiard is a natural dynamical system, whose phase space consists of dimensionless billiard balls moving inside a plane with definite boundaries, with unit speed.

The points in the phase space collide with the boundary following the law that angle of incidence is equal to the angle of reflection from the normal. When collision occurs at a corner, the

speed is made zero.

Billiards can be of many types. Polygonal billiards are simply connected polygons; i.e. shapes with boundaries defined with straight lines. These can be classified further as rational billiards and irrational billiards.

1.2.1 Rational Billiards

*A Polygon **P** is called Rational if the angles between the sides of **P** are rational multiples of π. The billiard flow in a rational polygon is called a rational billiard*

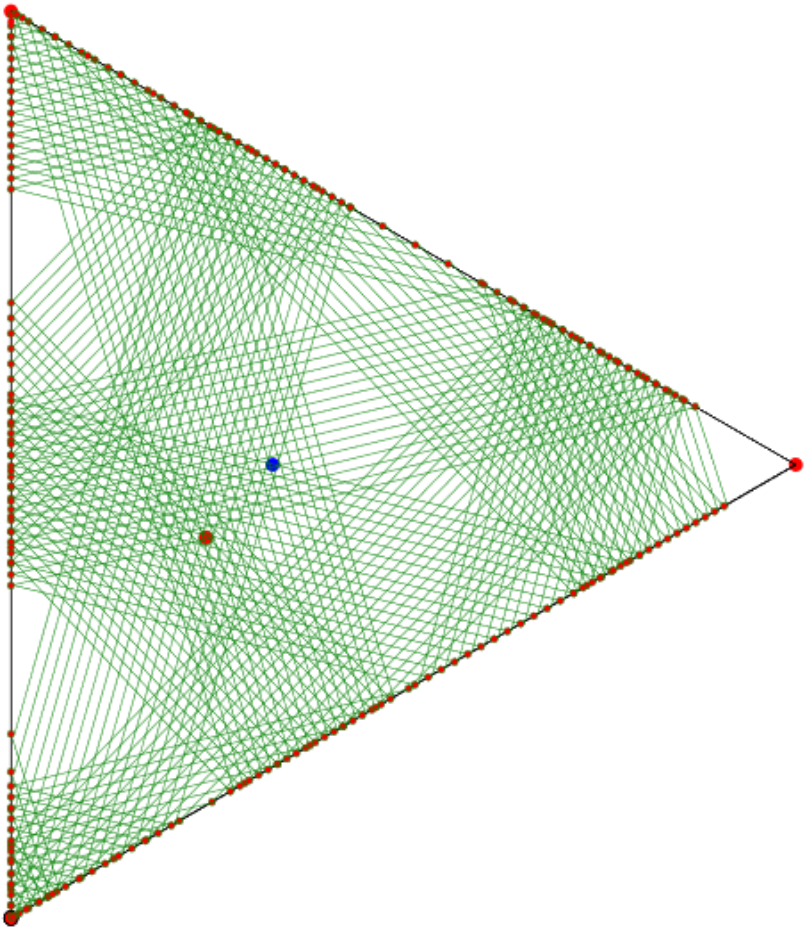


Figure 1.1: Rational Polygon. Origin is depicted in blue, particle position at genesis(in $T(P)$) and at consequent timesteps(at boundary ∂P) are depicted in red, and green depicts the particle trajectories.

We shall assume that origin lies in the center of the phase space $T(P)$ of our generated polygon P.

As a consequence of that, all simply connected convex polygons subtend an angle of 2π at the origin.

Regular Polygons are inherently rational. We can see a periodicity in the particle motion in a regular polygon. Figure 1.1 shows a regular polygon. We can observe the periodicity in particle motion as depicted in Figure 1.2 and Figure 1.3.

Invariant surfaces can be generated by tracing the billiard flow in rational polygons. Observe that only a finite number of reflections of the shape P can be drawn due to the periodicity in the billiard flow. This result is crucial in the generation of invariant shapes(Sections 4, [4]).

Rational Polygons are known to be non-ergodic.

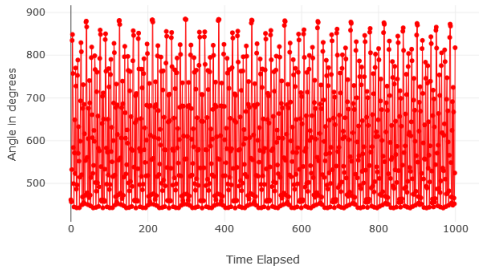


Figure 1.2: d vs Time for rational polygon. Periodicity can be observed. Total Time = 1000 steps

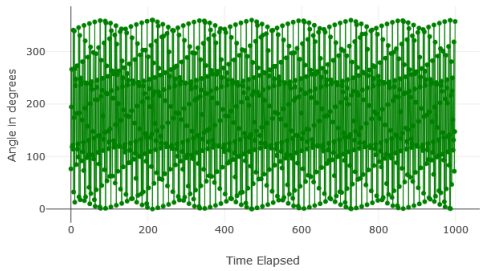


Figure 1.3: θ vs Time for the Rational Polygon. Periodicity is clearly visible here too.

1.2.2 Irrational Billiards

A polygon P is irrational if at least one angle in it is not a rational multiple of π.

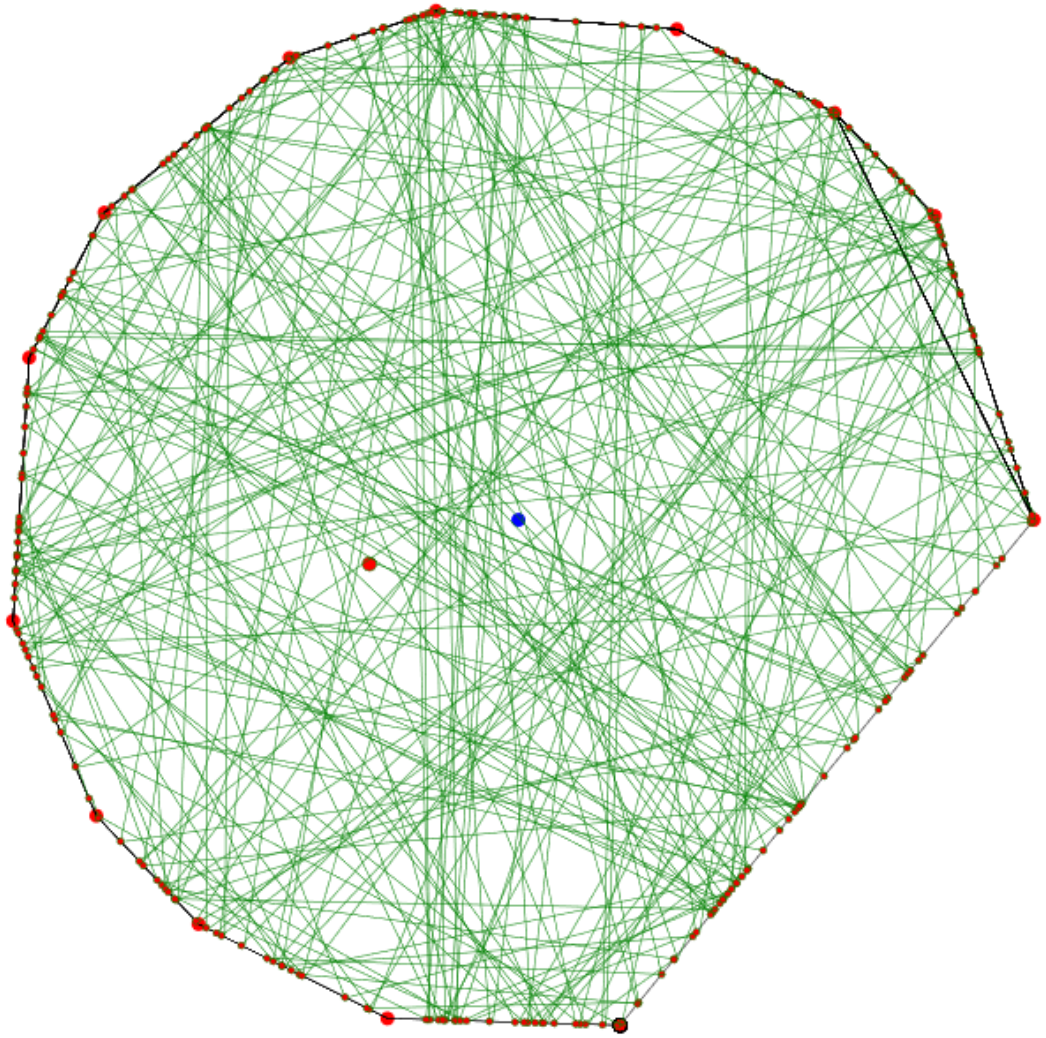


Figure 1.4: Irrational Polygon. Origin is depicted in blue, particle position at genesis(in $T(P)$) and at consequent timesteps(at ∂P) are depicted in red, and green depicts the particle trajectories. The black line inside the polygon is due to a bug in the code, and hence should be ignored.

Irrational Polygons may be symmetric, but not regular. Figure 1.4 shows a non-symmetric irrational polygon.

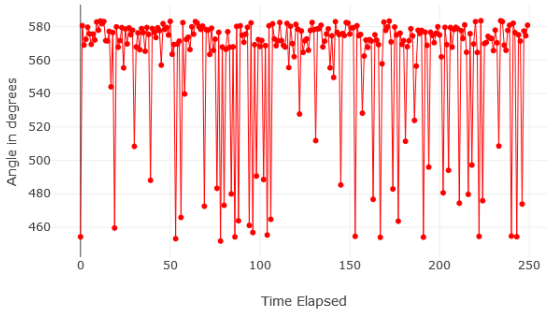


Figure 1.5: d vs Time for irrational polygon. Periodicity can not be observed. Total Time = 250 steps (reduced for image clarity.)

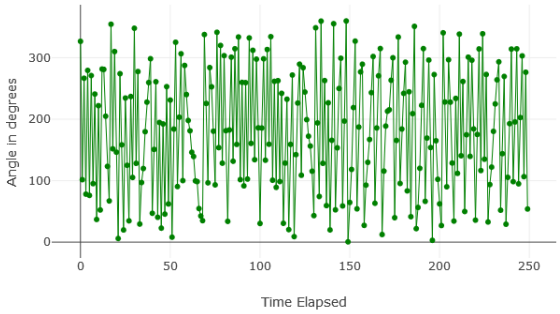


Figure 1.6: θ vs Time for the irrational Polygon. Periodicity is absent here too.

1.3 Genus of a Surface

In simple words, the genus $g(R)$ of a closed surface R is the number of holes inside R. The surface with genus g looks like a pretzel with g holes in it (refer fig 1.7 and fig 1.8).

The *Euler* number χ of the surface of genus g is given by:

$$\chi = 2 - 2g$$

The way R is constructed in Section 4 of [4] makes calculation of its *Euler* number convenient.The usual counting of faces, edges and vertices gives:

$$g(R) = 1 + \frac{N}{2} \sum \frac{m_i - 1}{n_i}$$

where vertex angles are $\pi m_i/n_i$, and $i = 1, \ldots, N$.

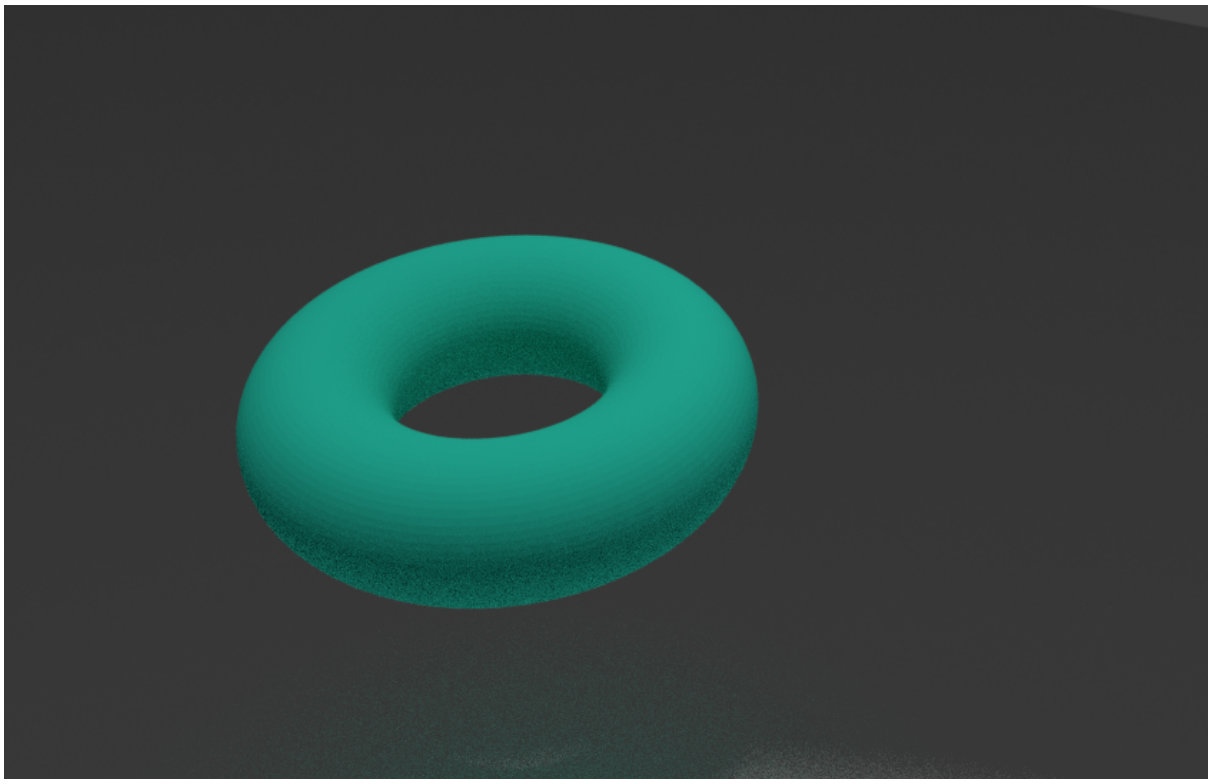


Figure 1.7: The surface of genus $g(R) = 1$

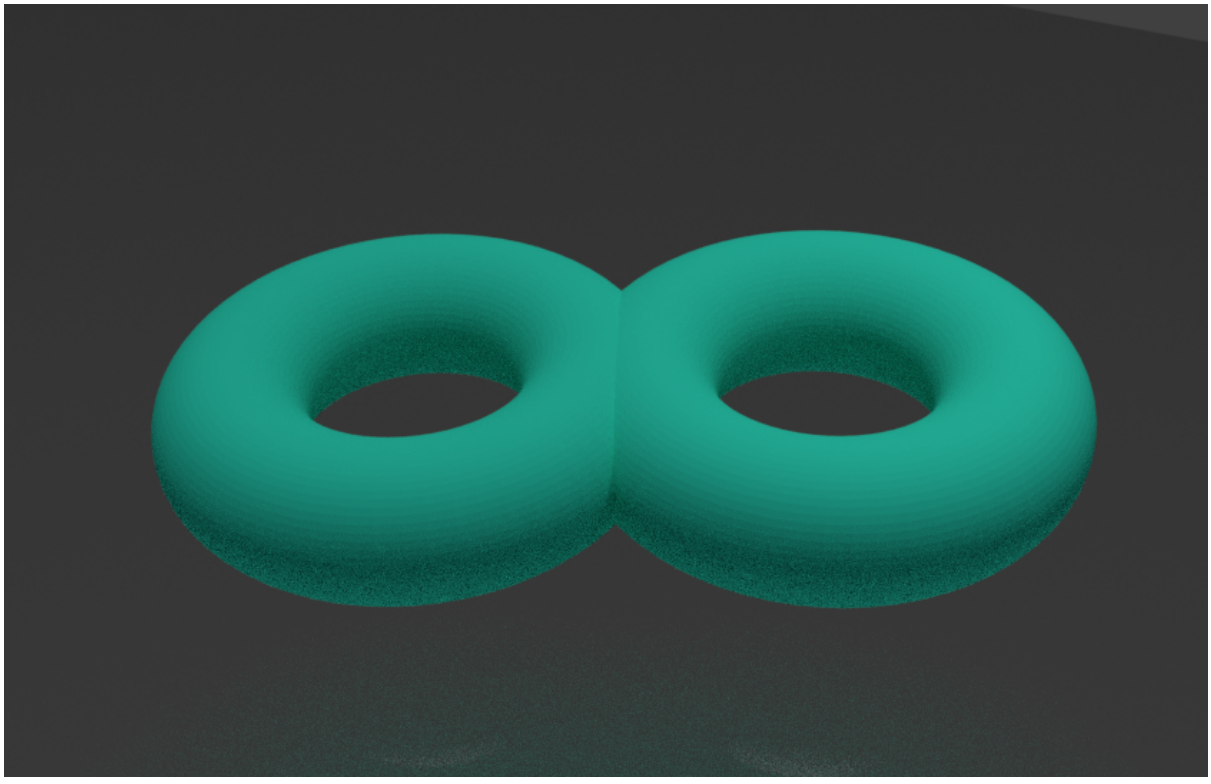


Figure 1.8: the surface of genus $g(R) = 2$

2. MACHINE LEARNING

Machine Learning is a form of Artificial Intelligence where statistical techniques are applied to algorithms such that their performance improves proportionally to the size of the input dataset. Machine learning can be classified into three broad categories, namely, Supervised Learning, Unsupervised Learning and Reinforcement Learning. Typical problems where ”learning” is involved are prediction problems, reward-maximization/error-minimization and classification problems.

Supervised Learning is the simplest of all the approaches. A large input data set I with known results R is used to ”teach” the algorithm.As Subset $V \epsilon I$ is excluded while training. V is then used as a ”validation” set to tune the ”hyperparameters” for improving the accuracy of the algorithm. Prior to any training process, a set $T \epsilon I$ is left out, as the ”test” set of the algorithm. T is then used after the training is complete to test the accuracy. Once it’s accurate enough, the algorithm is ready to predict the unknown results $R‘$ of any input data set $I‘$. Supervised Learning is broadly used in pattern recognition e.g. Handwriting detection, Natural Language Processing, and Computer Vision.

In Unsupervised Learning, the algorithm is not ”trained” based upon the input data-set. Instead, it groups similar results. Unsupervised Learning is usually seen in action in search engines, where webpages with similar metadata are displayed in the search results.

Reinforcement Learning is the most complicated approach. In this approach, the algorithm works its way up to maximize the reward. Explanation of this approach is beyond the scope of this project. This technique is usually seen in action in chess engines, where algorithm tries to maximize its possibility of winning the game based on the opposition player’s move.

We will now discuss a few basic heuristics generally used in Machine Learning.

2.1 General Machine Learning Techniques

2.1.1 Linear Regression

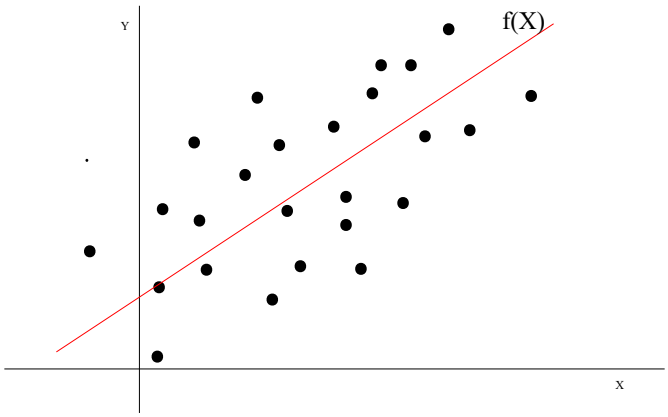


Figure 2.1: Typical Linear regression model

In linear regression, our aim is to predict the tentative value of a quantitative measure, which is dependent upon a given measure. Suppose the set $X = \{x_1, \ldots, x_n\}$ carries the values of our independent measure, and $Y = \{y_1, \ldots, y_n\}$ is in some way dependent upon X, our aim is to find a tentative value of some point y which is not already in the set Y based upon some given value $x \epsilon X$. We usually start with the method of Least-Squares approximation (refer fig 2.1). Consider a function f acting on the set X:

$$f(x) = ax + b \qquad\qquad x \epsilon X$$

such that $(y - f(x))^2$ is minimized for all $y \epsilon Y$. Let A be the set of all such a's and B be the set of all such b's .

At genesis, random values are assigned to all the elements of A and B. our algorithm then finds out the optimal values of all the elements in the sets A and B using the least-square method. Once that is done, our algorithm is ready to make predictions.

Linear regression is not a perfect model for prediction since it assumes linearity in all the data sets. Hence, generally, "better" statistical prediction techniques are used.

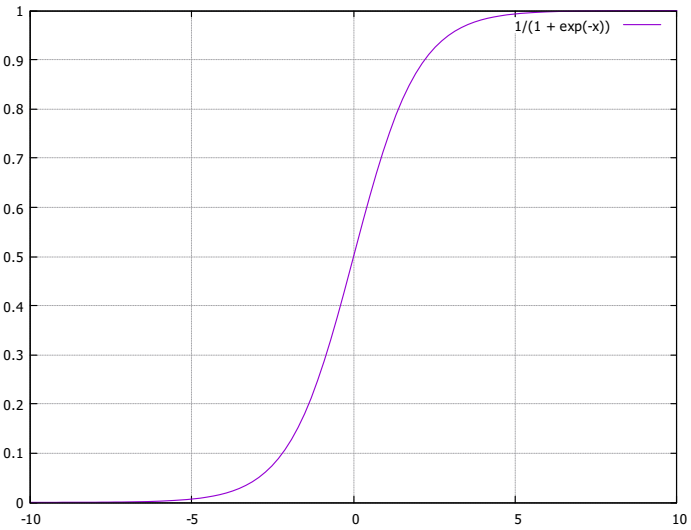


Figure 2.2: Typical logistic regression curve. Y axis ranges is [0,1]. X range can be anything; here it is [-10, 10]

2.1.2 Logistic Regression

In Logistic Regression we consider classification problems of dichotomous nature. We get the probability of input belonging to either class as the outcome.

We use the logistic function, also known as the sigmoid function to get the final outcome:

$$y = \frac{1}{1 + e^x}$$

This justifies the curve we get in fig 2.2

The elements of the dependent set Y here has only two values: either 0 or 1. Denote $P(x)$, $x \epsilon X$ as the probability that the corresponding value of $y \epsilon Y$ for the given $x \epsilon X$ is 1.

We can now write for f as given below:

$$f(x) = log(\frac{P(x)}{1 - P(x)})$$

This result can be easily derived from the logistic function:

$$P(x) = \frac{1}{1 + e^{f(x)}}$$

Hence,

$$P(x) = \frac{e^{ax+b}}{1 + e^{ax+b}}$$

Here, the optimal values of the sets A and B are calculated by a heuristic called *backpropagation*. The explanation of this heuristic is explained in detail in [5].

2.1.3 k-Nearest Neighbours

The kNN algorithm is one of the simplest classification algorithms. In kNN, along with the input data-set, a positive integer k is also defined. A database is created from that input data-set. When a new input is given, the algorithm calculates the Mahalanobis distance of the recently charted input point from k nearest neighbours. We then categorize the input with the class where the maximum number of these k nearest neighbours lie.

It is a non-parametric algorithm, which means it does not make any assumptions about the input dataset, unlike linear regression, where we assume the linearity of the model, and logistic regression, where dichotomy is assumed.

The kNN algorithm creates a database of all the input data, making it fast, but computationally expensive. It requires no training or testing. It still produces outcomes with good-enough accuracy.

This makes kNN versatile; it can also be used for non-linear data inputs. The cons of kNN classification are visible in the space and memory requirements of the algorithm. For huge datasets (which is usually the case in Machine Learning), storing the full dataset in the memory is usually not a good idea. We'll want to sparsify the input dataset in most real-world applications.

2.1.4 Support Vector Machines

Support Vector Machines (SVMs) is one of the most widely used models for binary classification. SVM is a maximum margin classifier, i.e. in case of a binary classification, our aim is to find out a hyperplane that lies in between the two classes such that it's distance from the closest extremums of the two classes is maximum.

Suppose we have an input dataset $I \epsilon \mathbb{R}^n$ and a corresponding label set $Y \epsilon \{-1, 1\}$. Lets say we have a total of m data points, such that $I = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_m\}$.

Any hyperplane in this space can be written as

$$wx - b = 0$$

where w is the normal vector to the hyperplane and $\frac{b}{||w||}$ determines the offset of the hyperplane from the origin.

If we are lucky enough to get a linearly separable dataset, we can easily find out two hyper-

planes such that

$$wx_i - b \geq 1 \qquad \forall y_i = 1$$
$$wx_i - b \leq 1 \qquad \forall y_i = -1$$

The distance between these two hyperplanes is $\frac{2}{||w||}$, hence, to maximise the distance, we need to minimise the value of w.

The above equations can then be reduced to

$$y_i(wx_i - b) \geq 1 \qquad \forall 1 \leq i \leq m \qquad (*)$$

The problem then reduces to the optimization problem that can be stated as:

Minimize $||w||$ *constrained by* (*)

The resulting w and b then build our classifier. It's visible that the x_i's closest to the maximum margin hyperplane determine it's location. Therefore, these x_i's are called the "Support Vectors".

In case the problem is not linearly separable, We can define a hinge loss function, $max(0, 1 - y_i(wx_i - b))$, where y_i is from the label set and $(wx_i - b)$ is the current output. The function takes the value zero when x_i lies on the correct side of the margin. For the wrong values, the value of hinge loss is proportional to the distance of the outlier from the margin.

The optimisation problem can then be reframed as:

Minimize $\frac{1}{m}\Big[\Sigma max(0, 1 - y_i(wx_i - b))\Big] + \eta||w||^2$

η parameter helps us determine the tradeoff between margin size and correctness. For a sufficiently small value of η, we can end up with a good classification algorithm.

SVM's can use the **kernel trick** for non-linear classifications. We usually increase the dimensionality of the input dataset, which allows the representation of the hyperplane in a higher dimensional feature space, which, in the original dimensions, may transform to a non-linear classifier. Implementation of this idea is beyond the scope of this project.

2.1.5 Principal Component Analysis

As stated above, in Machine Learning, we want to compress the dataset size without losing much precision from it. That way, memory requirements for running the algorithm will be reduced and we'll be able to predict/classify in a more efficient manner.

Suppose we have an input dataset $I \epsilon \mathbb{R}^n$, our aim is to obtain a new dataset $I' \epsilon \mathbb{R}^l$ where $l < n$, without losing much precision. The basic idea is to device an encoding function $f : \mathbb{R}^n \Longrightarrow \mathbb{R}^l$ and a decoding function $g : \mathbb{R}^l \Longrightarrow \mathbb{R}^n$ such that $I = g(f(I))$.

Principal Component Analysis (PCA) is used to apply lossy compression to the dataset. Dimensionality reduction at it's core, PCA is a widely used algorithm for sparsification.

In PCA, we try to capture a screenshot of our input data from a lower dimension. For example, suppose we have a classroom full of students. Our aim is to take a photograph of the class such that maximum faces are visible. By capturing this photograph, we reduce the three dimensional data representing the co-ordinates of each student's face in a Euclidean space to the two-dimensional photograph.

PCA uses basic linear algebra to achieve this. Let us denote the input I in form of a matrix or a tensor. For the sake of simplicity, consider $I \epsilon \mathbb{R}^n$. This means that I can be denoted as a simple two-dimensional matrix with each column indicating a unique independent property and each row indicating the value of that property.

We will first calculate the mean value of every property in I. Denote this with $\overline{I}$. $\overline{I}$ will then be a matrix with just one row. We then subtract all values in every column of I with their respective means in $\overline{I}$. Denote this new matrix as $I - \overline{I}$.

Now we will calculate the variance of every property in I. Denote this with $Var(I)$. Let $\sigma = \sqrt{Var(I)}$. σ is hence the standard deviation of I. We will now divide all values in every column of $I - \overline{I}$ with the value in that column of σ. Let this new matrix be denoted by Z. Then:

$$Z = \frac{I - \overline{I}}{\sigma}$$

Create a new matrix M such that:

$$M = Z^T Z$$

M will be a symmetric $n \times n$ matrix. Eigendecomposition is then performed on M. For this purpose, we have used the QR Decomposition technique. [8] is a nice article on QR Decomposition.

Basically, the `Gram-Schmidt process` is applied to M, after which M can be denoted as

$$M = QR$$

where Q is an orthogonal matrix and R is upper triangular. We then calculate the dot product RQ. Denote this with M_1. We then apply the `Gram-Schmidt process` to M_1 and continue the same process. Let

$$M_1 = Q_1 R_1$$
$$\therefore M_2 = R_1 Q_1$$
$$M_2 = Q_2 R_2$$
$$.$$
$$.$$
$$.$$
$$\therefore M_n = R_{n-1} Q_{n-1}$$
$$M_n = Q_n R_n$$

Then, $QQ_1Q_2 \dots Q_n$ is a diagonal matrix with eigenvalues in a decreasing order. Let

$$E = QQ_1Q_2 \dots Q_n$$
$$Z^* = ZE$$

The first l columns of Z^* is the output of this PCA. We have hence reduced an n-dimensional input to l dimensions, where, $l < n$.
In this project, $n = 2$ and $l = 1$

2.1.6 Artificial Neural Networks

Artificial neural networks are inspired by the actual biophysical neural networks. They mimic their functioning, while being much simpler to implement.
To visualize an ANN, we need a weighted directed graph data structure. Each neuron in the neural network has some weighted input, performs some activation and predicts weighted output. Using some sophisticated techniques, the neural networks have the ability to learn, which is why they are interesting. Learning means, suppose we have a task at hand, and a hypothesis space of functions, a set of known observations can be used to find the optimal value for the task using a function from the hypothesis space.
The most basic neuron in called a perceptron. It can act as a good linearly separable binary

classifier. It can be defined as:

$$f_{perceptron} = \begin{cases} 1 & \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{P1}$$

where **x** is the vector of all the inputs to the perceptron, **w** is the weight matrix of all the connections to the next layer. This function, termed as the "activation" function, determines the operation the neuron will perform on the given set of input data. The perceptron algorithm uses the Heaviside Step function (the function shown in equation P1) as the activation.

Multi-Layer Perceptron

Multi-Layer Perceptrons (MLP's) are a more flexible class of feedforward networks, which can classify data that are not linearly separable. It is a network of at least three layers, namely, an input layer, a hidden layer, and an output layer. These layers are densely connected; i.e. all the nodes of each layer are connected to all the nodes of the next layer until the output layer is reached. This is a supervised learning approach and Backpropogation is used for training the network.

2.1.7 Backpropagation

A nice introduction to backpropagation can be found in Chapter 2 of [5].

2.1.8 Stochastic Gradient Descent

2.1.9 Adaptive Moments Estimator

2.1.10 Types of Errors

Bayes Error

Generalization Error

3. ALGORITHM AND RESULTS

In this chapter, we will discuss the algorithm that forms the core of this project. This project has been implemented primarily in Javascript, and TensorFlow.js is used as the primary Machine Learning framework.

This is a Machine Learning project, so, Python might sound like a more intuitive choice for carrying out the programming, due to the vast number of frameworks it provides and it's popularity in the Machine Learning research community. But there are a few reasons why Javascript was chosen over Python for this:

- **Visualisation:** Visualising and animating Polygonal Billiards is much easier in Javascript, paired with the HTML5 Canvas functionality. Javascript allows this out-of-the-box, without using any third-party libraries. That solves a dual purpose: it saves us time and effort required to learn a new framework, while making the code usable for a wider range of audience.

- **Speed:** Python is one of the slowest languages out there. In an Intel i5 4th generation U series processor, it takes around 10 minutes for a Python interpreter to run a loop of a hundred thousand timesteps. Javascript takes around 2 minutes for the same.

- **Versatility:** Since Javascript runs in a browser, and doesn't need any extra interpreters to be installed on the system. As a consequence, it is cross-platform out-of-the-box.

Let's now have a look at what we are doing:

3.1 Particle Physics

The particle is implemented as class, encapsulating some basic properties like position, velocity and acceleration. To represent these vector quantities, a separate vector class has been

implemented, which has methods for vector addition, scalar product, vector product etc. Additionally, the particle class contains methods to update its position based on some given time, while calculating the time it will require for the next collision with the boundary. This is important, as we will not have to animate the particle position at every frame in that case.
The update methods in the particle class also handle the reflections of the particle when it collides with the boundary of a polygonal billiard. These are specular reflections, i.e., particle reflects like a beam of light hitting on the mirror. So, the angle of incidence with respect to the normal boundary of the billiard is equal to the angle of reflection, without any change in the particle speed.

3.2 Billiard Generation

We need to able to generate polygonal billiards in different shapes and sizes to build a viable classifier. For that purpose, a `shape` class was implemented, with the number of sides as an input parameter. Different logics were considered for the generation of Rational and Irrational Polygons. Earlier, Regular polygons were generated to represent a rational polygon, and any irregular shape was considered as an irrational polygon. The generation logic was as follows for Rational Polygons:

- The first vertex is drawn on the horizontal axis at some distance x from the origin.
- The second vertex is drawn at some angle θ_1 from the horizontal axis, but at the same distance x. Here $\theta_1 = \pi m_1/n_1$.
- The third vertex is drawn at some angle θ_2 from the horizontal axis, such that $\theta_2 > \theta_1$, $\theta_2 = \pi m_2/n_2$, again at the same distance x from the origin.
- Same procedure continues until all the vertices are placed. We then add up all the θ_i's to get 2π.

For Irrational Polygons, the same procedure was followed. Only difference is, here we choose θ_i randomly such that

$$\theta_i = \theta_{i-1} + r_i(\frac{\pi}{N} - \frac{\pi}{18}) + \frac{\pi}{18} \tag{1}$$

where r_i is a random value between 0 and 1.

Since, in a computer, there is no way we can generate an irrational number because of limited precision, we approximate one of these random r_i's to be irrational in nature and carry forward the simulation.

Recently, the shape generation logic was completely reformulated. A new restrictive random number generator was designed. In the new logic, an array `a` is maintained such that:

$$\{a[i] = i + 2\} \qquad 0 \leq i \leq 999$$

Another array, `a_sqrt` is maintained that stores the squared root values of all the numbers in `a`. After that, all the elements of `a_sqrt` that are also present in `a` are stripped off, so that we are left with an array of irrational numbers.

Now, we take `N` as the input to the generator. where `N` is the total number of sides we want in the polygon. We want the generator to generate an array of `N` elements which sum up to 2. Let's denote the output by an array `op`. Then,

$$\sum_{n=1}^{n=N} op[i] = 2$$

For implementing this, we take a dividend `1/(N/4)`. When `N` is even, we fill up half the output array by `dividend/f()` where `f()` is

$$f() = \text{any random element of A} \qquad where\ A\ \epsilon\{a, a_{sqrt}\} \tag{2}$$

We fill in the next half of the array with `dividend - op[k]` where $k\,\epsilon\,\{1, \ldots, N/2\}$ hence completing the output array. For an odd value of `N`, an output array of `N+1` elements is generated, then the last element is popped out of the array, divided by `N` and then added to every element of the resulting `N` membered array. That way, we can maintain the constant summation of elements.

Now, we can re-write the equation for generating the angles i.e. equation 1 as:

$$\theta_i = \theta_{i-1} + \pi op[i] \qquad \forall\, i\,\epsilon\,\{0, \ldots, N-1\} \tag{3}$$

such that on adding up `N` such elements, we get 2π as a result. Now, for generating such an array of rational numbers, in equation 2, `A` will take the value `a`. On the other hand, for generating irrational numbers, `A` will take the value `a_sqrt`. This way, we can generate a small set of rational as well as irrational angles, but fairly enough for our classification.

Apart from this, the shape class also implements a method that takes a particle object as the input to calculate the side that the particle will encounter the first for collision. For this purpose,

we had to represent the sides as a separate class to house methods that could calculate the Y intercept etc.

3.3 Data Formulation and Machine Learning

As we can see, so far, our algorithm is generating a lot of data. We need to clean up the data and formulate it in a certain manner to apply machine learning to it.

Since we are using `TensorFlow.js` to carry out the machine learning operations, we need to make out data compatible to the library. It is optimum to have the input dataset values between zero and one for obtaining the best results with `TensorFlow.js`. `TensorFlow.js` needs all our arrays to be converted to ”tensors”. A function has been implemented to apply PCA to our data, which also outputs mean and second moments. Also, a separate function has been implemented that shuffles our data, separates our train and test sets and then converts them all to the required ”tensors”.

Now let’s have a look at the hottest topic of this project: the machine learning implementation. Since its a simple, non-linear binary classification, we need to use Multi-layer perceptrons to classify our data. The reason why Multi-layer Perceptrons were chosen over Support Vector Machines for this classification is that we know out dataset will not be linearly seperable. The MLP is usually known to be more efficient in that case. Also, implementing a MLP Algorithm will be much simpler than the more complicated kernel trick we will need to use to implement a viable SVM classifier.

The Input dataset has six properties, namely, PCA-ed distance from the origin, PCA-ed angles with respect to the X-axis, mean distance from the origin, mean angle with respect to the X-axis, second moment(squared root of variance) distance from the origin and second moment angle with respect to the X-axis. Hence, we’ll need six perceptrons in the input layer. Since it’s a binary classification, we’ll need two outputs in the output layer. We have a hidden layer whose job is tweaking the parameters in some way to facilitate the classification, and that layer can have any number of perceptrons. Around ten to fifteen perceptrons is optimum for our problem. Too many neurons in the hidden layer will lead to overfitting and too less will lead to underfitting, and we will not be able to build a proper classifier.

In our classifier, we have used twelve hidden perceptrons, densely connected with the input and the output layers. The input layer and the hidden layer are conneceted using the sigmoid

activation function, the same function used in logistic regression discussed above. The hidden layer is densely conneted to the output layer, using the softmax activation function, again discussed above. We are bound to use the Categorical Cross-entropy as the loss function, since it works best with the softmax classifier. Finally, we are using the 'adam' optimizer to tweak our hyper-parameters.

After compiling this model, we feed it with the training set, and the test set that is used once the training process is complete. We are training the data for a thousand epochs, the training set is shuffled every time and a random ten percent of the set is used as the validation set.

As we can see, there is a lot of flexibility here. We can use sigmoid as the activation function in all the layers, or softmax in all the layers, or some other activation function listed in `TensorFlow.js` documentation, or we can change the number of hidden perceptrons, or we can add a few more hidden in hopes of getting a better classification. Since machine learning was never applied to this problem, it is all hit-and-trial. We will have to experiment a lot.

3.4 Results

In this section, we'll discuss some results of all the experiments we performed with the simulation.

3.4.1 Polygonal Billiards

Data generation usually takes the most amount of time in any machine learning application. We were no exception. It took us around five months to come up with the code that could simulate Polygonal Billiards the way we wanted, generating a random shape every time we run the simulation. Refer Fig 1.1 for the result of the generation of a regular polygon, Fig 1.4 for the result of the generation of an irregular or the old irrational polygon. Fig and Fig show the rational and irrational polygons generated using the most recent logic.

3.4.2 Classification: Regular and Irregular Polygons

Even though this classification has no physical significance, and is not useful as such, but we have followed baby-steps: build a simple classifier first, and then move on to the more complex one. Using the classifier we created as discussed above, we were able to get more than ninety-five percent accuracy on average, for a dataset size of five thousand trained over a thousand

epochs. The accuracy was almost hundered percent on lower dataset sizes. The screenshots shown in Fig show the accuracy results.

3.4.3 Classification: Rational and Irrational Polygons

After changing the shape generation logic, we started getting variable results, without changing parameters over different runs. The screenshot can be seen in Fig

4. CONCLUSION

As of now, we have not succeeded in building a classifier that classifies better than 70 percent accuracy. We have not been able to build a robust classifier that gives the same accuracy for the same input parameters. We are still experimenting, doing everything we can to build a robust classifier that classifies with a good accuracy.

4.1 Future scope

We plan to implement the SVM classification if we get enough time. As of now, we are tweaking the Deep Neural Network we have created, experimenting to find out where the problem lies.

BIBLIOGRAPHY

[1] Leonid Bunimovich. Dynamical billiards. *Scholarpedia*, 08 2007.

[2] Jr. Charles P. Poole, Ruslan Prozorov, Horacio A. Farach, and Richard J. Creswick. *Superconductivity*. Elsevier, 2014.

[3] Yakov G Sinai. Dynamical systems with elastic reflections. *Russian Mathematical Surveys*, 25:137, 10 2007.

[4] Eugene Gutkin. Billiards in polygons. *Physica D: Nonlinear Phenomena*, 19:311–333, 04 1986.

[5] M.Nielson. *Neural Networks and Deep Learning, Chapter 2*. Online Book, 2014.

[6] Kushal Shah, Dmitry Turaev, Vassili Gelfreich, and Vered Rom-Kedar. Equilibration of energy in slow–fast systems. *Proceedings of the National Academy of Sciences*, 114(49):E10514–E10523, nov 2017.

[7] Kushal Shah, Dmitry Turaev, and Vered Rom-Kedar. Exponential energy growth in a fermi accelerator. *Physical Review E*, 81(5), may 2010.

[8] Igor Yanovsky. Qr decomposition with gram-schmidt.