Random Walks on Fractals *

J. Marcus Hughes †

May 20, 2018

1 Introduction

Imagine a raindrop falling from the sky. It wanders as it falls, ever attracted to the ground by gravity. This simple behavior can be modeled as a three-dimensional random walk with an attracting plane. Even simpler, consider particles in a liquid. Their motion is seemingly random on short time scales as they are attracted by near-by particle and repelled by others. Numerous physical phenomena can be modeled with random walks of various kinds [5, 6]. Random walks exhibit structure in how they traverse space though, structure that can be characterized using a kind of fractal dimension.

Random walk theory is a well explored and old discipline. The name "random walk" is thought to originate with Pearson in 1901 regarding a game of golf [1]. Pearson presented it in Nature as a question asking for help:

> A man starts from a point 0 and walks 1 yards in a straight line; he then turns through any angle whatever and walks another 1 yards in a second straight line. He repeats this process n times. I require the probability that after these n stretches he is at a distance between r and $r + dr$ from his starting point 0. The problem is one of considerable interest, but I have only succeeded in obtaining an integrated solution for two stretches. I think, however, that a solution ought to be found, if only in the form of a series in powers of $1/n$, when n is large

*written as a final project for Math 306: Fractals and Chaos Theory at Williams College taught by Cesar Silva

†hughes.jmb@gmail.com

Lord Rayleigh had explored this phenomena for gases in the 1880s [3] and responded with to Pearson that the probability of traveling a distance between R and $R + dR$ in N steps as $N \to \infty$ was [4]:

$$P_N(R) \sim \frac{2R}{N} e^{-R^2/N} \tag{1}$$

This began a long study of random walks which would take volumes to completely account. Applications range from statistical physics (also the continuous version of Brownian motion [2]) to search engine algorithms. Despite the long history of study, there remain open questions concerning specific kinds of random walks.

For this paper, I will present a random walk contextualized as a Markov chain. The random walk is an iterated process where the next step only depends on the current location. Random walks occur in some space $\mathbb{X}$ which could be finite, such as on a finite graph, or infinite, such as $\mathbb{R}$. The walker is at some location $p \in \mathbb{X}$, such as (x, y) in the lattice $\mathbb{Z} \times \mathbb{Z}$. It can then transition to any of its neighbor states $N(p) \subset \mathbb{X}$. In the lattice, the $N((x, y)) = \{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\}$. In the case of the simple random walk, the probabilitiy of attaining any of these subsequent states is equal. However, this does not have to be the case. Let $P(p, p')$ be a function $\mathbb{X} \times \mathbb{X} \to [0, 1]$ denoting the probability of transitioning from p to p'. Then given $p \in \mathbb{X}$, it holds $\sum_{p' \in \mathbb{X}} P(p, p') = 1$. This framework is general enough to admit several interesting applications and questions.

2 Simple random walks

We will begin by considering the simple random walk on the regular lattice in k dimensions and analyzing the fractal dimensino of it. There are many ways to consider the dimension of a geometric object. The topological dimension is the most common in grade school and has applications in linear algebra and other mathematics. It is always a natural number and is defined recursively. A set A has topological dimension of zero if every point $p \in A$ has neighborhoods of arbitrarily small length whose boundary misses A. An

example is a finite set of points $\{1, 2, 3, 4\} \subset \mathbb{Z}$. A set A has topological dimension of one if is not of topological dimension 0 and every point in A has arbitrarily small neighborhoods whose boundary meets A in a set of topological dimension zero. The definition for topological dimension is recursively defined from there. This dimension does not admit the rough structure in sets we are examining, e.g. Cantor's set is the same topological dimension as a finite set.

The Hausdorff dimension admits more structure. The Hausdorff dimension is s such for

$$\lambda^t(A) = \inf\{\sum_{j=1}^{\infty} |I_j|^t : A \subset \cup_{j=1}^{\infty} I_j\}$$

it holds that $\lambda^s(A) = \infty$ for $s < t$ and $\lambda^s(A) = 0$ for $s > t$. It captures the fractal nature but is difficult to use in practice. Thus, we tend to consider the Minkowski dimension which can be extracted using the box-counting algorithm. For a compact set A and $\epsilon >, 0$, let $N(A, \epsilon)$ be the smallest number of ϵ-boxes needed to cover A. Then the Minkowski dimension is:

$$d = \lim_{\epsilon \to 0} -\frac{\log N(A, \epsilon}{\log \epsilon}$$

when the limits exists. This definition works quite well for interesting sets such as the Sierpisnki triangle. However, for the random walk on the lattice it is insufficient because when $N(A, \epsilon) < n$ where n is the number of steps in the random walk for $\epsilon < 1$ because each lattice point visited is matched to one box.

Therefore, I will follow the example of Saberi (2011) and use a different measure of the fractal dimension [7]. Thus, the dimension d of a random walk is $M \sim R^d$ where M is the number of unique lattice points visited and R is the radius of gyration for a random walk. The radius of gyration R is defined in terms of the center of mass r_{cm}:

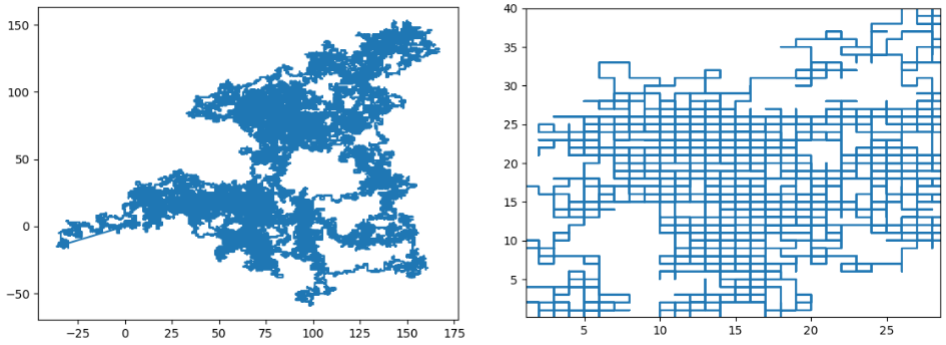
$$r_{cm} = \frac{1}{n+1} \sum_{i=0}^{n} r_i \tag{2}$$

. Here, n is the number of steps in the walk and r_i is the location at the i-th step. From this, we derive R:

$$R = \sqrt{\frac{1}{N+1}\sum_{i=0}^{n}\langle(r_{cm} - r_i)^2\rangle} \tag{3}$$

. In this case, $\langle r_i \rangle$ is the average of the entries of r_i. Thus, the dimension can be found by fitting a line to a log-log plot of M and R.

The figure below shows an example of a simple random walk in two dimensions:

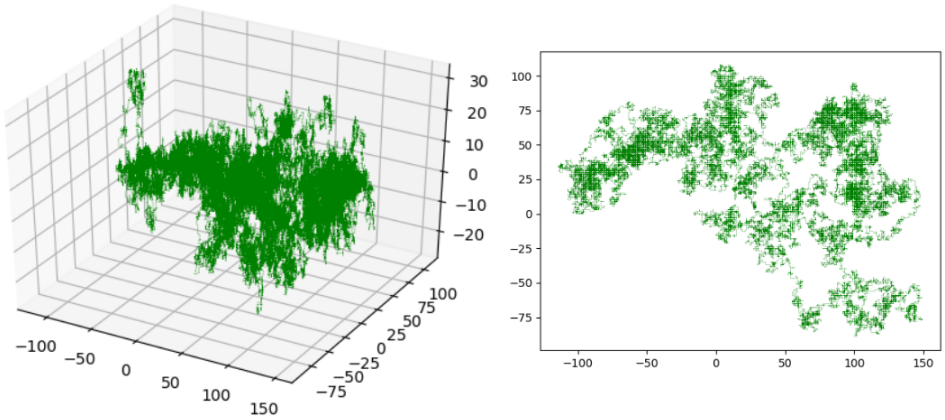


The image at left shows the full walk with 55574 steps while the right zooms in to reveal the underlying lattice structure. The fractal dimension of this motion is rather uninteresting. For $d = 1$, the Hausdorff dimension is known to be $3/2$ while it is 2 for $d \geq 2$ [7].

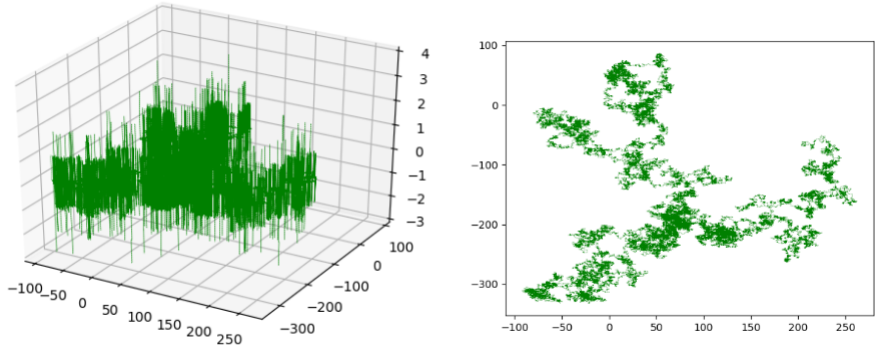
3 Attracted random walkks

Saberi (2011) propose an attracted random walk where the walker is predisposed attracted, with some attraction coefficient α, toward the xy-plane from a three-dimensional lattice. At a lattice point (x, y, z) the walker has six choices for movement: up, down, left, right, forward, backwards. Let $p = \frac{1}{\alpha+5}$ and $q = \frac{1}{4\alpha+2}$. When the walker is not on the xy-plane, i.e. $z \neq 0$, the walker has probability αq to move toward the plane and p for all other directions. When the walker is on the xy-plane, i.e. $z = 0$, the walker will move left, right, forwards, and backwards with probability of αq and up or down (thus moving off the

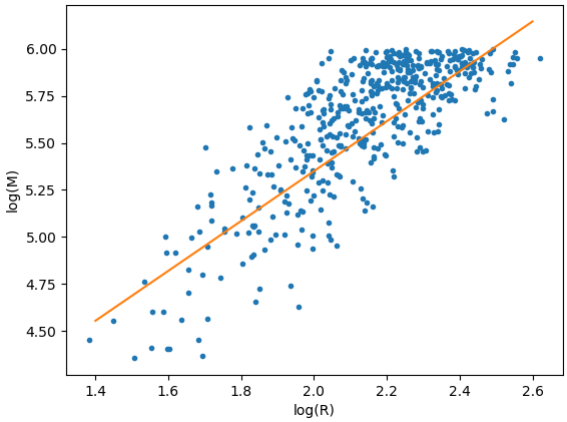
plane) with probability of q. When, $\alpha = 1$ this is a pure three-dimensional random walk but when $\alpha \to \infty$ this becomes a two-dimensional random walk on the plane. The walker begins at the origin.



Above is an example of this attracted random walk with $\alpha = 1.3$ and 10^4 steps. At left, you can see a side view in three dimensions and at right is the view directly from the top. This is lightly attracted. Below, you can see the result with $\alpha = 10$.



The walker can only get a few steps off the plane before it is pulled back onto it. Most of the interesting behavior occurs in this range of $1 < \alpha < 5$. However, by looking at very high α we can determine the limiting case. Saberi (2011) found this to be 1.83 [7]. However, I could not replicate this result.



My fit line instead had slope of 1.3. I did not have sufficient time to determine why this was different.

4 Code Contribution

Since this project was computational, I will briefly outline my original code contributions. I have developed two independent software packages in Java: jifs and jWalker. These were kept separate as jifs is being developed as a portable Java version of TeraFractal, a beautiful fractal generation tool exclusively for Mac. jifs provides iterated function systems in Java and measuring the Minkowski dimension while jWalker handles the random walks portion. This report will be made available with the jWalker code.

4.1 jWalker

This package includes the random walk code. It implements a simple arbitrary topological dimension point which is used by a generic random walk class. Then, I have implemented simple random walks as well as a plane attracted random walk. The random walk classes provide measurement tools for their dimensions.

4.2 jifs

This package has support for simple iterated function systems of affine transformations. It also supports visualizing them and has the capacity to add other kinds of iterated function systems. It provides support to measure the Minkowski dimension of a set of points.

5 Summary

Ultimately, I was a bit disappointed with my result. I had planned (and still do) to explore more random walks. For example, I was planning on adding "traps" to the attracing plane where it was difficult to esacpe, or similarly points that pushed away dramatically. I was also planning on exploring it in higher dimensions and also constraining it so the walk could not go below the plane. I sketched out a version where instead of an attracting plane there was an attracting surface. However, I had insufficient time to implement. Another plan was to allow the walk to be attracted to an iterated function system instead of a plane. Thus, it would be offered "neighbors" that were from the IFS as well spatial neighbors. I thought it would be interesting to see the attraction. I had plans to allow for the attraction to depend on the distance from the plane as well, as simple modification (which shouldn't have resulted in much change if you were already on the plane except for an occasional wandering off).

However, I had very little time with other finals and thesis work (I did apply fractal dimension in it though to find the fractal dimension of active regions on the Sun). I'm attaching an appendix with the code generated for this project (some of it did not get used because I ran out of time). I will return to this project after graduation and update this report with new results as they become available for your reading.

References

[1] Carazza, B. The history of the random-walk problem: Considerations on the interdisciplinarity in modern physics. *Rivista Del Nuovo Cimento 7*, 3 (1977), 419–27.

[2] Einstein, A. Über die von der molekularkinetischen theorie der wrme geforderte bewegung von in ruhenden flssigkeiten suspendierten teilchen. *Annalen der physik 322*, 8, 549–560.

[3] F.R.S., L. R. Xii. on the resultant of a large number of vibrations of the same pitch and of arbitrary phase. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 10*, 60 (1880), 73–78.

[4] F.R.S., L. R. O. Xxxi. on the problem of random vibrations, and of random flights in one, two, or three dimensions. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 37*, 220 (1919), 321–347.

[5] Haw, M. Einstein's random walk. *Physics World 18*, 1 (2005), 19.

[6] Mandelbrot, B. B. *The fractal geometry of nature*. W. H. Freeman, New York, 1983.

[7] Saberi, A. A. Fractal structure of a three-dimensional brownian motion on an attractive plane. *Phys. Rev. E 84* (Aug 2011), 021113.

Appendix A Code

I included copies of the code here for your easy of viewing.

A.1 Point

```
import java.util.Vector;
import java.lang.Double;

/**
 * A representation of a point in Euclidean space.
 * @author J. Marcus Hughes
 */
public class Point {
    private Vector<Double> coord;
    private int dimension;

    /** Build a point from arbitrarily many double values */
    Point(Double...  values) {
        this.dimension = values.length;
        this.coord = new Vector<Double>();
        for (int i = 0; i < this.dimension; i++) {
            coord.addElement(values[i]);
        }
    }

    /** Built a point from arbitrarily many integer values */
    Point(Integer...  values) {
        this.dimension = values.length;
        this.coord = new Vector<Double>();
        for (int i = 0; i < this.dimension; i++) {
            coord.addElement((double) values[i]);
        }
    }

    /** Built a point from a <code> Vector </code> of coordinate values */
    Point(Vector<Double> values) {
        this.coord = values;
        this.dimension = values.size();
    }

    /**
     * Number of coordinates in point
     * <p>
     * For example, (0,0) is dimension 2 and (0,0,0) is dimension 3
```

```
     * </p>
     */
    public int dimension() {
        return coord.size();
    }

    /** Returns the point as a vector of doubles */
    public Vector<Double> getCoord() {
        return this.coord;
    }

    /**
     * Create a string version of a point
     * <p>
     * For example (3, 4, 5)
     * </p>
     */
    public String toString() {
        String s = "(";
        for (Double v : coord) {
            s += v;
            s += ", ";
        }
        s = s.substring(0, s.length()-2);
        s += ")";
        return s;
    }

    /**
     * Add a point to another point elementwise
     * @param other a point with the same dimension
     * @throws RuntimeException when points are not of the same dimension
     * @return the elementwise sum of points
     */
    public Point add(Point other) {
        if (other.dimension() != this.dimension()) {
            throw new RuntimeException("Points must have same dimension to add");
        }
        Vector<Double> result = new Vector<Double>();
        for (int i = 0; i < this.dimension; i++) {
            result.addElement(this.coord.get(i) + other.coord.get(i));
        }
        return new Point(result);
    }
```

```java
/**
 * Scales a point by a scalar double
 * @param k number to scale by
 */
public Point scale(double k) {
    Vector<Double> result = new Vector<Double>();
    for (int i = 0; i < this.dimension; i++) {
        result.addElement(this.coord.get(i) * k);
    }
    return new Point(result);
}


/**
 * Subtract other point
 * @param other point to subtract
 * @throws RuntimeException when points are not of the same dimension
 * @return this - other
 */
public Point subtract(Point other) {
    return this.add(other.scale(-1.0));
}

/**
 * Multiply points
 * @param other point to multiple
 * @throws RuntimeException when points are not of the same dimension
 * @return this * other
 */
public Point multiply(Point other) {
    if (other.dimension() != this.dimension()) {
        throw new RuntimeException("Points must have same dimension to add");
    }
    Vector<Double> result = new Vector<Double>();
    for (int i = 0; i < this.dimension; i++) {
        result.addElement(this.coord.get(i) * other.coord.get(i));
    }
    return new Point(result);
}

/** Squares */
public Point square() {
    return this.multiply(this);
}
```

```
    public double mean() {
        double sum = 0.0;
        for (double v : coord) {
            sum += v;
        }
        return sum / dimension;
    }

    /** testing method */
    public static void main(String[] args) {
        System.out.println("Testing for point");
        Point p = new Point(1, 2, 3);
        System.out.println(p);
    }
}
```

A.2 RandomWalk

```
import java.util.Vector;
import java.util.Set;
import java.util.HashSet;
import static java.lang.Math.sqrt;

/**
 * A generic representation of a random walk
 * @author J. Marcus Hughes
 */
public abstract class RandomWalk {
    int dimension; // the number of coordinates in a point, e.g. (0,0) is dimension 2
    Vector<Point> path; // the collection of points traveled by the random walk

    /**
     * A generic random walk starting from <code> source </code>
     * @param source where the random walk begins from
     */
    RandomWalk(Point source) {
        this.path = new Vector<Point>();
        this.path.addElement(source);
        this.dimension = source.dimension();
    }

    /** Take one step, i.e. move one iteration */
    abstract public void step();

    /**
```

```
     * Take <code> steps </code> and return the path
     * @param steps number of movements to make
     * @return path of random walk
     */
    public Vector<Point> walk(int steps) {
        for (int i = 0; i < steps; i++) {
            step();
        }
        return this.path;
    }

    /** Return path */
    public Vector<Point> getPath() {
        return this.path;
    }

    /** Number of points in path */
    public int length() {
        return this.path.size();
    }

    /**
     * Calculates the center of mass
     * @throws RuntimeException if the path is empty
     */
    public Point centerOfMass() {
        if (this.length() == 0) {
            throw new RuntimeException("Path must be populated before calculating");
        }
        Vector<Double> point = new Vector<Double>();
        for (int i = 0; i < this.dimension; i++) {
            point.addElement(0.0);
        }
        Point sum = new Point(point);
        for (Point p : path) {
            sum = sum.add(p);
        }
        return sum.scale(1.0 / (1.0 + length()));
    }

    /**
     * Calculates the radius of gyration
     * @throws RuntimeException if the path is empty
     */
    public double radiusOfGyration() {
```

```
        Point center = centerOfMass();
        double sum = 0.0;
        for (Point p : path) {
            sum += p.subtract(center).square().mean();
        }
        return sqrt((1.0 / (length() + 1.0)) * sum);
    }

    /**
     * Number of unique points
     */
    public int numUnique() {
        Set<Point> s = new HashSet<Point>(path);
        return s.size();
    }
}
```

A.3 PlaneAttracted3D

```
import java.util.Vector;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class PlaneAttracted3D extends RandomWalk {

    private double alpha;
    private Random rng; // random number generator
    private double p; // probability of moving when on plane
    private double q; // probability of moving when off plane, p' in paper

    /**
     * Only constructor for <code> PlaneAttracted3D </code>
     * @param alpha attraction parameter
     */
    PlaneAttracted3D(double alpha) {
        super(new Point(0,0,0));
        this.rng = new Random();

        // attraction to plane
        this.alpha = alpha;
        if (this.alpha < 1.0) {
            throw new RuntimeException("alpha must be greater than 1");
        }

        // parameters for movement
```

```
        this.p = 1.0 / (alpha + 5.0);
        this.q = 1.0 / (4.0 * alpha + 2.0);
    }

    /** Determine where to move next when on plane */
    private Point onPlane() {
        Point prev = this.path.lastElement();
        float p = this.rng.nextFloat(); // random number

        // determine next step based on probability and selection
        // on sum compared to generated p
        if (p < this.q) {
            return prev.add(new Point(0, 0, -1)); // move off plane
        } else if (p < 2.0 * this.q) {
            return prev.add(new Point(0, 0, 1));  // move off plane
        } else if (p < (2.0 + this.alpha) * this.q) {
            return prev.add(new Point(1, 0, 0));  // move in plane
        } else if (p < (2.0 + 2.0 * this.alpha) * this.q) {
            return prev.add(new Point(-1, 0, 0)); // move in plane
        } else if (p < (2.0 + 3.0 * this.alpha) * this.q) {
            return prev.add(new Point(0, 1, 0));  // move in plane
        } else {
            return prev.add(new Point(0, -1, 0)); // move in plane
        }
    }

    /** Determine java documentationwhere to move next when off plane */
    private Point offPlane() {
        Point prev = this.path.lastElement();
        float p = this.rng.nextFloat();

        // determine next step based on probability and selection
        // on sum compared to generated p
        if (p < this.p) {
            return prev.add(new Point(1, 0, 0));
        } else if(p < 2.0 * this.p) {
            return prev.add(new Point(-1, 0, 0));
        } else if(p < 3.0 * this.p) {
            return prev.add(new Point(0, 1, 0));
        } else if(p < 4.0 * this.p) {
            return prev.add(new Point(0, -1, 0));
        } else if(p < 5.0 * this.p) {
            if (prev.getCoord().lastElement() < 0) {  // z < 0
                return prev.add(new Point(0, 0, -1)); // move away from plane
            } else {                                  // z > 0
```

```
                return prev.add(new Point(0, 0, 1));  // move toward plane
            }
        } else {
            if (prev.getCoord().lastElement() < 0) {  // z < 0
                return prev.add(new Point(0, 0, 1));  // move toward plane
            } else {                                  // z > 0
                return prev.add(new Point(0, 0, -1)); // move away from plane
            }
        }
    }

    /** Take one step */
    public void step() {
        Point prev = this.path.lastElement();
        Integer z = prev.getCoord().lastElement().intValue();

        if (z == 0) { // on plane
            this.path.addElement(onPlane());
        } else {      // z != 0, so off plane
            this.path.addElement(offPlane());
        }
    }

    /** Testing method */
    public static void main(String[] args) {
        Random rng = new Random();
        int iterations = Integer.parseInt(args[0]);
        double alpha   = Double.parseDouble(args[1]);

        int iMin = 10000;
        int iMax = 1000000;
        int length = ThreadLocalRandom.current().nextInt(iMin, iMax);

        PlaneAttracted3D rw;

        for (int i = 0; i < iterations; i++) {
            length = ThreadLocalRandom.current().nextInt(iMin, iMax);
            rw = new PlaneAttracted3D(alpha);
            rw.walk(length);
            System.out.printf("%d %f \n", rw.numUnique(),  rw.radiusOfGyration());
        }
    }
}
```

A.4 Simple2D

```
import java.util.Vector;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class Simple2D extends RandomWalk {
    private Random rng;

    /**
     * Only constructor for <code> Simple3D </code>
     * @param alpha attraction parameter
     */
    Simple2D() {
        super(new Point(0,0));
        this.rng = new Random();
    }

    /** Take one step */
    public void step() {
        Point prev = this.path.lastElement();
        float p = this.rng.nextFloat();

        // determine next step based on probability and selection
        // on sum compared to generated p
        if (p < 1.0/4.0) {
            this.path.addElement(prev.add(new Point(1, 0)));
        } else if (p < 2.0/4.0) {
            this.path.addElement(prev.add(new Point(-1, 0)));
        } else if (p < 3.0/4.0) {
            this.path.addElement(prev.add(new Point(0, 1)));
        } else {
            this.path.addElement(prev.add(new Point(0, -1)));
        }
    }

    /** Testing method */
    public static void main(String[] args) {
        int iterations = Integer.parseInt(args[0]);

        int iMin = 10000;
        int iMax = 1000000;
        int length = ThreadLocalRandom.current().nextInt(iMin, iMax);

        Simple2D rw;
```

```
        for (int i = 0; i < iterations; i++) {
            length = ThreadLocalRandom.current().nextInt(iMin, iMax);
            rw = new Simple2D();
            Vector<Point> ww = rw.walk(length);
            // if (i == 0) {
            //     for(Point p : ww) {
            //         System.out.println(p);
            //     }
            //     System.out.println("-----");
            // }
            System.out.printf("%d %f \n", rw.numUnique(),  rw.radiusOfGyration());
        }
    }
}
```

A.5 Simple3D

```
import java.util.Vector;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class Simple3D extends RandomWalk {
    private Random rng;

    /**
     * Only constructor for <code> Simple3D </code>
     * @param alpha attraction parameter
     */
    Simple3D() {
        super(new Point(0,0,0));
        this.rng = new Random();
    }

    /** Take one step */
    public void step() {
        Point prev = this.path.lastElement();
        float p = this.rng.nextFloat();

        // determine next step based on probability and selection
        // on sum compared to generated p
        if (p < 1.0/6.0) {
            this.path.addElement(prev.add(new Point(1, 0, 0)));
        } else if (p < 2.0/6.0) {
            this.path.addElement(prev.add(new Point(-1, 0, 0)));
```

```
        } else if (p < 3.0/6.0) {
            this.path.addElement(prev.add(new Point(0, 1, 0)));
        } else if (p < 4.0/6.0) {
            this.path.addElement(prev.add(new Point(0, -1, 0)));
        } else if (p < 5.0/6.0) {
            this.path.addElement(prev.add(new Point(0, 0, 1)));
        } else {
            this.path.addElement(prev.add(new Point(0, 0, -1)));
        }
    }

    /** Testing method */
    public static void main(String[] args) {
        int iterations = Integer.parseInt(args[0]);

        int iMin = 10000;
        int iMax = 1000000;
        int length = ThreadLocalRandom.current().nextInt(iMin, iMax);

        Simple3D rw;

        for (int i = 0; i < iterations; i++) {
            length = ThreadLocalRandom.current().nextInt(iMin, iMax);
            rw = new Simple3D();
            rw.walk(length);
            System.out.printf("%d %f \n", rw.numUnique(),  rw.radiusOfGyration());
        }
    }
}
```

A.6 plotFig1

```
import numpy as np
import matplotlib.pyplot as plt
import argparse

def get_args():
    ap = argparse.ArgumentParser()
    ap.add_argument("path", help="path to text file to open")
    return vars(ap.parse_args())

if __name__ == "__main__":
    args = get_args()
    dat = np.loadtxt(args['path'])
    m3d, rg = dat[:,0], dat[:,1]
```

```
    fig, ax = plt.subplots()
    ax.plot(np.log10(rg),
            np.log10(m3d),
            ".")
    plt.show()
```

A.7 IFS

```
import java.util.Vector;
import static java.lang.Math.random;

/**
 * A representation of an iterated function system
 * An iterated function system takes a set of functions/transformations
 * and evaluates where a given point goes under the transformation.
 * A key example is the Barnsley Fern.
 * An iterated function system can have multiple portions. Technically,
 * all of the transformations can be evaluated, but a probabalistic
 * approach is possible. Thus, each transformation has an associated
 * likelihood.
 * @author J. Marcus Hughes
 */
public class IFS {
    Vector<Transform> transforms;
    Vector<Double> probabilities;

    /**
     * Returns IFS with all transforms having equal weight
     * @param transforms a list of equally weighted transforms
     */
    IFS(Vector<Transform> transforms){
        assert transforms.size() > 0 : "transforms must have size > 0";
        this.transforms = transforms;
        this.probabilities = new Vector<Double>();
        double weight = 1.0/this.transforms.size();
        for (Transform t : this.transforms){this.probabilities.add(weight);}
        assert checkProbability() : "probability list must sum to 1.0";
    }

    /**
     * Constructor given a list of transforms and probabilities
     * @param transforms a vector of transform type objects
     * @param probabilities probability of selecting transform, must sum to 1.0
     */
```

```
IFS(Vector<Transform> transforms, Vector<Double> probabilities){
    if (transforms.size() != probabilities.size()) {
        throw new RuntimeException("Transforms and probabilities must be the same size");
    }
    this.transforms = transforms;
    this.probabilities = probabilities;
    assert checkProbability() : "probability list must sum to 1.0";
}

/**
 * copy constructor for IFS
 * @param other an IFS to copy
 */
IFS(IFS other) {
    this.transforms = other.transforms;
    this.probabilities = other.probabilities;
    assert checkProbability() : "probability list must sum to 1.0";
}

/** determine if the probability list is sums to 1.0
 * @return <code>true</code> if sum is 1.0 and <code>false</code> otherwise
 */
private boolean checkProbability(){
    double total = 0.0;
    for (double probability : this.probabilities)
        total += probability;
    return total == 1.0;
}
/**
 * Append the transform to the set of transforms
 * @param t transform to be appended
 * @param p probability of selecting that transform
 */
public void addTransform(Transform t, double p){
    this.transforms.add(t);
    this.probabilities.add(p);
    assert checkProbability() : "probability list must sum to 1.0";
}

/**
 * Select a transformation at random using the probability weighting
 */
public Transform chooseTransform() {
    double n = Math.random();
    double runningTotal = 0.0;
```

```
        int i;
        for (i = 0; i < this.probabilities.size(); i++) {
            runningTotal += this.probabilities.get(i);
            if (n <= runningTotal) {
                break;
            }
        }
        return this.transforms.get(i);
    }

    /**
     * create a string version
     */
    public String toString() {
        String out = "";
        out += "transform: \n";
        int i = 0;
        for (Transform t : transforms) {
            out += "\t";
            out += t;
            out += " at ";
            out += (int)(100 * probabilities.get(i));
            out += "%\n";
            i += 1;
        }
        return out;
    }

    /**
     * testing main
     */
    public static void main(String[] args){
        System.out.println("Test of IFS");

        Matrix shrink = new Matrix(new double[][]{{0.5, 0.0}, {0.0, 0.5}});
        AffineTransform t1 = new AffineTransform(shrink, new Matrix(0.0,0.0));
        AffineTransform t2 = new AffineTransform(shrink, new Matrix(0.5,0.0));
        AffineTransform t3 = new AffineTransform(shrink, new Matrix(0.0,0.5));
        Vector<Transform> transforms =  new Vector<Transform>();
        transforms.add(t1);
        transforms.add(t2);
        transforms.add(t3);
        IFS system = new IFS(transforms);
        Matrix p = new Matrix(0.041462, 0.408642);
        for(int i=0; i < 10000; i+=1) {
```

```
            System.out.println(p);
            Transform t = system.chooseTransform();
            p = t.transform(p);
        }


    }

}
```

A.8 AffineTransform

```
/**
 * A representation for an AffineTransform for usage in generating fractals and other
 *  iterated function systems.
 * An AffineTransform is a linear transform, in the plane this is represetable by a 2x2
 * matrix, combined with a translation. Thus, it can be fully described by 6 values:
 *
 * <pre>
 *     | a b | | x |   | e |
 * T = |     | |   | + |   |
 *     | c d | | y |   | f |
 * </pre>
 *
 * In the code below, [[a,b],[c,d]] is the matrix and [e,f] is the shift.
 * @author J. Marcus Hughes
 * @since 2018-04-22
 */
public class AffineTransform extends Transform {
    Matrix matrix;
    Matrix shift;

    /**
     * Normal constructor
     * @param matrix linear transform portion [[a,b],[c,d]]
     * @param shift translation [e,f]
     */
    AffineTransform(Matrix matrix, Matrix shift) {
        super("Affine Transform");
        this.matrix = matrix;
        this.shift = shift;
    }

    /**
     * Expanded version of Affine Transform
```

```
     * @param a element of matrix
     * @param b element of matrix
     * @param c element of matrix
     * @param d element of matrix
     * @param e element of shift
     * @param f element of shift
     */
    AffineTransform(double a, double b, double c, double d, double e, double f) {
        super("Affine Transform");
        this.matrix = new Matrix(new double[][]{{a, b}, {c, d}});
        this.shift = new Matrix(new double[][]{{e},{f}});
    }

    /** Given a point, executes the affine transformation
     * @param p a point in the plane
     * @return the point after affine transformation is applied
     */
    public Matrix transform(Matrix p) {
        return this.matrix.multiply(p).add(this.shift);
    }

    /**
     * String representation
     */
    public String toString(){
        return String.format("[[%f,%f],[%f,%f]] [%f, %f]",
                             this.matrix.get(1,1),
                             this.matrix.get(1,2),
                             this.matrix.get(2,1),
                             this.matrix.get(2,2),
                             this.shift.get(1,1),
                             this.shift.get(2,1));
    }

    /**
     * A simple testing script that runs part of a Sierpinski Triangle
     */
    public static void main(String[] args) {
        Matrix p = new Matrix(new double[][] {{0.041462}, {0.408642}});
        Matrix shrink = new Matrix(new double[][]{{0.5, 0.0}, {0.0, 0.5}});
        AffineTransform t1 = new AffineTransform(shrink, new Matrix(0.0, 0.0));
        AffineTransform t2 = new AffineTransform(shrink, new Matrix(0.5, 0.0));
        AffineTransform t3 = new AffineTransform(shrink, new Matrix(0.0, 0.5));
        for(int i=0; i < 1000; i+=1) {
            System.out.println(p);
```

```
            p = t3.transform(p);
        }
    }
}
```

A.9 LinearRegression

```
import java.util.Vector;
import java.util.function.Function;
import static java.lang.Math.random;

/**
 * Simple linear regression resulting in model of y = ax + b for data points
 * @author J. Marcus Hughes
 */
public class LinearRegression {
    double[] xs;         // input reference data
    double[] ys;         // input predicted value
    double xbar;         // mean of xs
    double xbarsquared;  // mean of xs squared i.e. sum(x^2)  / n
    double ybar;         // mean of ys
    double ybarsquared;  // mean of ys squared, see xbarsquared
    double beta0;        // constant term, b, in model y = ax + b
    double beta1;        // slope term, a, in model y = ax + b
    double xxbar;        // square of x - xbar over xs
    double yybar;        // square of y - ybar over ys
    double xybar;        // (x - xbar) * (y - ybar) over xs, xs
    double R2;           // correlation term
    double svar1;        // variance of slope
    double svar0;        // variance of constant
    double rss;          // residual sum of squares
    double ssr;          // regression sum of squares

    /**
     * Construct a <code>LinearRegression</code> object with input data
     * automatically performs the regression and the model can be found
     * <code> model() </code> or seen in String, exact values are in
     * <code>this.beta0</code> for constant and <code>this.beta1</code>
     * for slope term
     * @param xs certain values
     * @param ys predicted values
     */
    LinearRegression(double[] xs, double[] ys) {
        this.xs = xs;
        this.ys = ys;
```

```
        // calculate the means
        double[] xbars = mean(xs);
        this.xbar = xbars[0];
        this.xbarsquared = xbars[1];

        double[] ybars = mean(ys);
        this.ybar = ybars[0];
        this.ybarsquared = ybars[1];

        // determine certainty and fit
        fit();
    }

    /**
     * Calculates the mean of some input array
     * @param xs array of values to average, must be nonempty
     * @return average of values
     */
    private static double[] mean(double[] xs) {
        if (xs.length == 0)
            throw new RuntimeException("Input array must be nonempty");
        double sum = 0.0;
        double sumsquared = 0.0;
        for (double x: xs) {
            sum += x;
            sumsquared += (x * x);
        }
        return new double[]{sum/xs.length, sumsquared/xs.length};
    }


    /**
     * Performs fit and calculates summary statistics
     */
    private void fit() {
        double xxbar = 0.0, yybar = 0.0, xybar = 0.0;
        for (int i = 0; i < xs.length; i++) {
            xxbar += (xs[i] - xbar) * (xs[i] - xbar);
            yybar += (ys[i] - ybar) * (ys[i] - ybar);
            xybar += (xs[i] - xbar) * (ys[i] - ybar);
        }
        // determine fit parameters
        double beta1 = xybar / xxbar;
        double beta0 = ybar - beta1 * xbar;
```

```
        // store values
        this.xxbar = xxbar;
        this.yybar = yybar;
        this.xybar = xybar;
        this.beta0 = beta0;
        this.beta1 = beta1;

        // determine summary stats
        int df = xs.length - 2;
        this.rss = 0.0; // residual sum of squares
        this.ssr = 0.0; // regression sum of squares
        for (int i = 0; i < xs.length; i++) {
            double fit = beta1*xs[i] + beta0;
            rss += (fit - ys[i]) * (fit - ys[i]);
            ssr += (fit - ybar) * (fit - ybar);
        }
        this.R2    = ssr / yybar;
        double svar  = rss / df;
        this.svar1 = svar / xxbar;
        this.svar0 = svar/xs.length + xbar*xbar*svar1;
    }

    /**
     * Returns model as callable function. Use by calling with apply(VALUE)
     * For example:
     * Function<Double, Double> f = m.model();
     * f.apply(3.4);
     */
    public Function<Double, Double> model() {
        return (Double x) -> this.beta1 * x + this.beta0;
    }

    /**
     * Creates a string with linear model and parameters
     */
    public String toString() {
        return String.format("y = %.2f x + %.2f with R^2=%.2f", beta1, beta0, R2);
    }

    /**
     * Testing method
     */
    public static void main(String[] args) {
        double[] xs = new double[]{2.0, 3.0, 4.0};
```

```
        LinearRegression m = new LinearRegression(xs, xs);
        System.out.println(m.mean(xs)[0]);
        System.out.println(m.xbar);
        System.out.printf("y = %.2f x + %.2f\n", m.beta1, m.beta0);
        System.out.println(m);
        Function<Double, Double> f = m.model();
        System.out.println(f.apply(2.0));
    }
}
```

A.10 Matrix

```
import java.lang.Math;

/**
 * A matrix class with M by N real entries
 * @author J. Marcus Hughes
 */
public class Matrix {
    private final double[][] entries; // values in matrix
    private final int m; // number of rows
    private final int n; // number of columns

    /**
     * A generic operation on a matrix done by entry,
     * e.g. scalar multiplication
     */
    interface ElementMath {
        double apply(double a);
    }

    /**
     * A generic operation on a matrix with another matrix done by entry,
     * e.g. matrix addition
     */
    interface ComponentMath {
        double apply(double a, double b);
    }

    public static Matrix randomPoint() {
        double min = -1.0;
        double max = 1.0;
        double x = min + Math.random() * (max - min);
        double y = min + Math.random() * (max - min);
        return new Matrix(x,y);
```

```
}

/**
 * Constructor for <code> Matrix </code> that fills with zeroes
 * @param m number of rows
 * @param n number of columns
 */
Matrix(int m, int n) {
    this.m = m;
    this.n = n;
    this.entries = new double[m][n];
}

/**
 * Constructs column matrix corresponding to a point
 * @param a top entry
 * @param b bottom entry
 */
Matrix(double a, double b) {
    this.m = 2;
    this.n = 1;
    this.entries = new double[][]{{a},{b}};
}

/**
 * Constructs a 2x2 matrix with entries of form {{a,b},{c,d}}
 * @param a 1,1 entry
 * @param b 1,2 entry
 * @param c 2,1 entry
 * @param d 2,2 entry
 */
Matrix(double a, double b, double c, double d) {
    this.m = 2;
    this.n = 2;
    this.entries = new double[][]{{a,b},{c,d}};
}

/**
 * Construct a <code> Matrix </code>
 * @param entries a double array storing the matrix entries row by row
 */
Matrix(double [][] entries) {
    this.m = entries.length;
    this.n = entries[0].length;
    this.entries = new double[this.m][this.n];
```

```
        for (int i = 0; i < this.m; i++) {
            for (int j = 0; j < this.n; j++) {
                this.entries[i][j] = entries[i][j];
            }
        }
    }

    /**
     * Retrieve the (i,j) element from the matrix
     * @param i row index
     * @param j column index
     */
    public double get(int i, int j) {
        if (i <= 0 || i > this.m) {
            throw new RuntimeException("Row out of bounds.");
        }
        if (j <= 0 || j > this.n) {
            throw new RuntimeException("Column  out of bounds.");
        }
        return this.entries[i-1][j-1];
    }

    /**
     * Perform an element-wise operation on a matrix
     * @param f function
     */
    private Matrix elementwise(ElementMath f) {
        Matrix B = new Matrix(m, n);
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                B.entries[i][j] = f.apply(this.entries[i][j]);
        return B;

    }

    /**
     * Scale a matrix by parameter
     * @param scale real number to scale each entry by
     */
    public Matrix multiply(double scale) {
        ElementMath f = (a) -> scale * a;
        return elementwise(f);
    }

    /**
```

```
     * Add a scalar to a matrix
     * @param n number to add
     */
    public Matrix add(double n) {
        ElementMath f = (a) -> a + n;
        return elementwise(f);
    }

    /**
     * Perform an component-wise operation on a matrix
     * @param B a matrix to take other element from
     * @param f function
     */
    private Matrix componentwise(Matrix B, ComponentMath f) {
        Matrix C = new Matrix(m, n);
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                C.entries[i][j] = f.apply(B.entries[i][j], this.entries[i][j]);
        return C;
    }

    /**
     * Determines if the matrices have the same dimension. If not, throws exception.
     */
    private void checkSameDimensions(Matrix B) {
        if (B.m != this.m || B.n != this.n) {
            throw new RuntimeException("Illegal matrix dimensions. Must be same size.");
        }
    }
    /**
     * Adds a matrix
     * @param B matrix to add
     * @return this + B
     */
    public Matrix add(Matrix B) {
        checkSameDimensions(B);
        ComponentMath f = (a,b) -> a + b;
        return componentwise(B, f);
    }

    /**
     * Subtracts a matrix
     * @param B matrix to subtract
     * @return this - B
     */
```

```
public Matrix subtract(Matrix B) {
    checkSameDimensions(B);
    ComponentMath f = (a,b) -> a - b;
    return componentwise(B, f);
}

/**
 * Multiply a matrix by another
 * @param B matrix to multiply
 * return this * B
 */
public Matrix multiply(Matrix B) {
    Matrix A = this;
    // check for proper size
    if (this.n != B.m) {
        throw new RuntimeException("Illegal matrix dimensions for multiply."
                                   + "Must be (nm)(mp)=(np)");
    }

    // perform multiplication
    Matrix C = new Matrix(A.m, B.n);
    for (int i = 0; i < C.m; i++) {
        for (int j = 0; j < C.n; j++) {
            for (int k = 0; k < A.n; k++) {
                C.entries[i][j] += (A.entries[i][k] * B.entries[k][j]);
            }
        }
    }
    return C;
}

/**
 * Create a string
 */
public String toString() {
    String out = "";
    for (int i = 0; i < this.m; i++) {
        for (int j = 0; j < this.n; j++) {
            out += this.entries[i][j];
            out += " ";
        }
        out += "\n";
    }
    return out;
}
```

```
    /**
     * testing method
     */
    public static void main(String[] args) {
        double[][] myList = {{1.0, 2.0},{3.0, 3.0}};
        Matrix m = new Matrix(myList);
        System.out.println(m.multiply(4));
        double[][] myList2 = {{-1.0, -2.0},{-3.0, -3.0}};
        Matrix m2 = new Matrix(myList2);
        System.out.println(m.add(m2));
        System.out.println(m.subtract(m2));
        Matrix a = new Matrix(new double[][]{{1.0, 2.0}, {3.0, 4.0}});
        Matrix b = new Matrix(new double[][]{{3.0},{4.0}});
        System.out.println(a.multiply(b));

    }
}
```

A.11 IFSEvaluator

```
/**
 * An abstract representation of an iterated function system <code> IFS </code> evaluator.
 * @author J. Marcus Hughes
 */
abstract class IFSEvaluator {
}
```

A.12 MinkowskiDimension

```
/**
 * Calculate the box-counting/Minkowski dimension for a set of points
 */
public class MinkowskiDimension {
    int dimension;
    Vector<Vector<double>> pointSet;
    Vector<double> sizes;
    Vector<int> counts;

    /**
     *
     */
    MinkowskiDimension(int dimension, Vector<Vector<double>> pointSet) {
        this.dimension = dimension;
        this.pointSet = pointSet;
```

```
        for (Vector<double> v : pointSet) {
            if (v.length != dimension) {
                throw new RuntimeException("All points must be of the same dimension.");
            }
        }
        this.counts = new Vector<int>();
        this.sizes = new Vector<double>();

    }

    private int count_boxes() {

    }


    /**
     * testing function
     */
    public static void main(String[] args) {

    }
}
```

A.13 Transform

```
import static java.lang.Math.random;
import static java.lang.Math.round;
import java.awt.Color;

/**
 * a requirement for any transformation for the IFS
 * each transform needs:
 *<ul>
 *<li> a name for logging purposes
 *<li> an initial starting color
 *<li> a definition of how to transform
 *<li> a string representation
 *</ul>
 * @author J. Marcus Hughes
 */
abstract class Transform {
    String transformName;
    Color color;

    /**
```

```
     * Basic constructor
     */
    Transform(String name) {
        this.transformName = name;

        this.color = new Color(randomColorChannel(),
                               randomColorChannel(),
                               randomColorChannel());
    }

    /**
     * utitlity function to get a random RGB color channel
     */
    private static int randomColorChannel() {
        return (int) Math.round(Math.random() * 255.0);
    }

    /**
     * how colors should change as the function evaluates
     * @param c what color is being considered
     */
    public void transform_color(Color c) {
        this.color = new Color((this.color.getRed() + c.getRed()) / 2,
                               (this.color.getGreen() + c.getGreen()) / 2,
                               (this.color.getBlue() + c.getBlue()) / 2);
    }

    /**
     * the definition of where <code> p </code> should go under this transform
     * @param p starting point
     * @return location after starting from <code> p </code>
     */
    abstract public Matrix transform(Matrix p);

    /**
     * simple string method
     */
    abstract public String toString();
}
```

A.14 RandomIFSEvaluator

```
import java.io.*;
import java.util.Vector;
import java.awt.image.*;
```

```java
import javax.imageio.*;
import java.awt.Color;

/**
 * An iterated function system <code> IFS </code> can be evaluated either
 * deterministically, where every point is evaluated in a straight forward
 * manner as described by the system, or probabilistically.
 * This is a probabilistic approach. A set of random points is selected
 * and iterated forward. These characterize the behavior of the iterated
 * function system without as much computational work.
 * @author J. Marcus Hughes
 */
public class RandomIFSEvaluator extends IFSEvaluator {
    IFS ifs;
    int numPoints;
    int iterations;
    Vector<Vector<Matrix>> results;
    /** Set up the random iterated function system evaluator
     * @param ifs an initialized iterated function system
     * @param numPoints how many randomly selected initial starting locations to run
     * @param how many iterations should be evaluated for each point
     * @see IFS
     */
    RandomIFSEvaluator(IFS ifs, int numPoints, int iterations) {
        assert numPoints > 0 : "numPoints must be positive";
        assert iterations > 0 : "iterations must be positive";
        this.ifs = ifs;
        this.iterations = iterations;
        this.numPoints = numPoints;
    }

    /**
     * Simulate running on a single point
     * @param p the initial point
     */
    public Vector<Matrix> runPoint(Matrix p) {
        Vector<Matrix> ps = new Vector<Matrix>();
        ps.add(p);
        for (int it = 0; it < this.iterations; it++) {
            Transform t = ifs.chooseTransform();
            p = t.transform(p);
            ps.add(p);
        }
        return ps;
    }
```

```
/**
 * Simulate running on many points
 */
public Vector<Vector<Matrix>> run() {
    Vector<Vector<Matrix>> result = new Vector<Vector<Matrix>>();
    for (int it = 0; it < this.numPoints; it++) {
        Matrix p = Matrix.randomPoint();
        result.add(runPoint(p));
    }
    this.results = result;
    return result;
}

/**
 * Plot the IFS in region (-1.0, 1.0) x (-1.0, 1.0) for last iteration
 * @param filename where to save image
 * @param width number of pixels wide for image
 * @param height number of pixels high for image
 */
public void plot(String filename, int width, int height) {
    plot(filename, width, height, iterations, -1.0, 1.0, -1.0, 1.0);
}

/**
 * Plot the IFS in region (-1.0, 1.0) x (-1.0, 1.0) for requested iteration
 * @param filename where to save image
 * @param width number of pixels wide for image
 * @param height number of pixels high for image
 * @param iteration step to plot
 */
public void plot(String filename, int width, int height, int iteration) {
    assert iteration < iterations : "not a valid iteration number";
    plot(filename, width, height, iteration, -1.0, 1.0, -1.0, 1.0);
}

/**
 * Plot the IFS in requested region for requested iteration
 * @param filename where to save image
 * @param width number of pixels wide for image
 * @param height number of pixels high for image
 * @param iteration step to plot
 * @param xmin least x value to show
 * @param xmax greatest x value to show
 * @param ymin least y value to show
```

```
     * @param ymax greatest y value to show
     */
    public void plot(String filename, int width, int height, int iteration,
                     double xmin, double xmax, double ymin, double ymax) {
        assert iteration < iterations : "not a valid iteration number";
        Image img = new Image(width, height, xmin, xmax, ymin, ymax);
        Vector<Vector<Matrix>> results = run();
        for(Vector<Matrix> r: results) {
            img.plot(r.get(iteration));
        }
        img.save(filename);
    }


    /**
     * testing main
     */
    public static void main(String[] args) {
        System.out.println("Test of random ifs evaluator");
        Vector<Transform> transforms = new Vector<Transform>();
        Vector<Double> probabilities = new Vector<Double>();

        //Sierpinski triangle
        /*
        Matrix shrink = new Matrix(0.5, 0.0, 0.0, 0.5);
        AffineTransform t1 = new AffineTransform(shrink, new Matrix(0.0,0.0));
        AffineTransform t2 = new AffineTransform(shrink, new Matrix(1.0,0.0));
        AffineTransform t3 = new AffineTransform(shrink, new Matrix(0.0,1.0));
        transforms.add(t1);
        probabilities.add(1.0/3.0);
        transforms.add(t2);
        probabilities.add(1.0/3.0);
        transforms.add(t3);
        probabilities.add(1.0/3.0);
        */


        // Barnsley Fern
        //stem
        AffineTransform t1 = new AffineTransform(new Matrix(0.0, 0.0, 0.0, 0.16),
                                                 new Matrix(0.0, 0.0));
        transforms.add(t1);
        probabilities.add(0.01);

        //smaller leaflets
```

```
        AffineTransform t2 = new AffineTransform(new Matrix(0.85, 0.04, -0.04, 0.85),
                                                 new Matrix(0.0, 1.6));
        transforms.add(t2);
        probabilities.add(0.85);

        //largest left leaflet
        AffineTransform t3 = new AffineTransform(new Matrix(0.20, -0.26, 0.23, 0.22),
                                                 new Matrix(0.0, 1.6));
        transforms.add(t3);
        probabilities.add(0.07);


        //largest right leaflet
        AffineTransform t4 = new AffineTransform(new Matrix(-0.15, 0.28, 0.26, 0.24),
                                                 new Matrix(0.0, 0.44));
        transforms.add(t4);
        probabilities.add(0.07);


        IFS system = new IFS(transforms, probabilities);

        RandomIFSEvaluator ifsRunner = new RandomIFSEvaluator(system, 10000, 100);
        ifsRunner.run();
        System.out.println("FINISHED RUNNING!");
        for (int i = 0; i < 100; i+=1){
            System.out.println(i);
            String fn = String.format("imgs/trial%03d.jpeg", i, 0.0, 3.0, 0.0, 3.0);
            ifsRunner.plot(fn, 500, 500, i);
        }
    }

}
```