CONSTRUCTION OF CENTROIDAL VORONOI TESSELLATIONS USING GENETIC ALGORITHMS

ABSTRACT

Centroidal Voronoi tessellations (CVTs) are a way of partitioning sets, and genetic algorithms are a way of optimizing functions. In this paper, we discuss how to apply genetic algorithms to the problem of generating CVTs by minimizing a function associated with such partitions. We outline the ways of relating components of genetic algorithms to CVTs, we test implementations of a genetic algorithm that yields CVTs, and we compare the performance of the genetic algorithm to other methods of approximating CVTs.

by

Halley McCormick

Whitman College

May 15, 2015

Contents

1 Introduction of Problem

Given data comprised of a discrete set of points W in $\mathbb{R}^N$, we can cluster the data around points $z \in W$ (called *generators* or *centers*) using a modeling technique called *Voronoi tessellations*. If we apply a density function ρ to the points in W, then we can find cluster centers in $\mathbb{R}^N$ that also correspond to the weighted values in W, where the cluster centers are called *centroids* and the resulting tessellation is a *centroidal Voronoi tessellation (CVT)*. The task of finding these centroids involves minimizing the sum of the squared distances between the points in W and their corresponding centers; the need for optimization suggests the use of *genetic algorithms* in finding centers. Several popular methods exist for optimizing these tessellations, but little research has been conducted to explore the efficiency of genetic algorithms for the same goals.

As its name suggests, a genetic algorithm is a procedure that mimics biological evolution by defining a population of "chromosomes" (in our case, subsets of the set W), a "fitness function" that we wish to optimize over these points, and a way of prescribing the "reproduction" of points in the set W through combinations of existing points and through random "mutation" of those points. Through these steps, a genetic algorithm can converge to an optimal solution. Genetic algorithms may provide the means to push past some of the limitations of existing algorithms.

In this paper, we will review definitions and properties of Voronoi tessellations and centroidal Voronoi tessellations, we will discretize the problem in order to be able to model problems using computers, and we will outline how to use genetic algorithms to approximate centroidal Voronoi tessellations.

1.1 Centroidal Voronoi Tessellations

To understand centroidal Voronoi tessellations (CVTs), we discuss Voronoi tessellations, the conditions under which they become centroidal, and ways of characterizing CVTs. Much of the notation in defining Voronoi tessellations and CVTs comes from the work by Du, Faber, and Gunzburger in [4].

1.1.1 The General Case

Definition 1.1. Given an open set $\Omega \subseteq \mathbb{R}^N$, the set $\{V_i\}_{i=1}^k$ is called a *tessellation* of Ω if $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\cup_{i=1}^k \overline{V}_i = \overline{\Omega}$. Note that in this

paper, we will assume that Ω is bounded. Let $|\cdot|$ denote the Euclidean norm on $\mathbb{R}^N$. Given a set of points $\{\mathbf{z}_i\}_{i=1}^k$ belonging to $\overline{\Omega}$, the *Voronoi region* $\hat{V}_i$ corresponding to the point $\mathbf{z}_i$ is defined by

$$\hat{V}_i = \{\mathbf{x} \in \Omega \quad | \quad |\mathbf{x} - \mathbf{z}_i| < |\mathbf{x} - \mathbf{z}_j| \text{ for } j = 1, \ldots, k, j \neq i\}.$$

The points $\{\mathbf{z}_i\}_{i=1}^k$ are called *generators* or *centers*. The set $\{\hat{V}_i\}_{i=1}^k$ is a *Voronoi tessellation* or *Voronoi diagram* of Ω, and each $\hat{V}_i$ is referred to as the *Voronoi region* corresponding to $\mathbf{z}_i$.

If we apply a density function to the region Ω, we are interested in the case where the regions' generators $\{\mathbf{z}_i\}$, $i = 1, \ldots, k$, are equal to their centers of mass.

Definition 1.2. Given a region $V \subseteq \mathbb{R}^N$ and a density function ρ defined in V, the mass centroid, $\mathbf{z}^*$, of V is defined by

$$\mathbf{z}^* = \frac{\int_V y\rho(y)dy}{\int_V \rho(y)dy},$$

where $\mathbf{z}^*$ is a weighted average of the points in V.

Given k points $\mathbf{z}_i$, $i = 1, \ldots, k$, we can define their associated Voronoi regions $\hat{V}_i, i = 1, \ldots, k$. Furthermore, given Voronoi regions $\hat{V}_i, i = 1, \ldots, k$ and an associated density function, we can find the regions' mass centroids, $\mathbf{z}_i^*$. In this paper, we concern ourselves with the case where

$$\mathbf{z}_i = \mathbf{z}_i^*, \; i = 1, \ldots, k;$$

i.e., the mass centroids $\mathbf{z}_i$ serve as generators for the Voronoi regions. A Voronoi tessellation that uses mass centroids as generators is called a *centroidal Voronoi tessellation (CVT).*

A related problem is that if we use ρ (after it has been properly normalized) as a probability density function, we wish to choose the sets $\{\mathbf{z}_i\}_{i=1}^k$ and $\{V_i\}_{i=1}^k$ so that the quantity

$$\mathcal{E}((\mathbf{z}_i, V_i), i = 1, \ldots, k) = \sum_{i=1}^{k} \int_{\mathbf{y}_j \in V_i} \rho(\mathbf{y}_j)|\mathbf{y}_j - \mathbf{z}_i|^2 \mathrm{d}\mathbf{y} \tag{1}$$

is minimized over all possible sets of k points belonging to Ω and all possible tessellations of Ω into k regions V_i, $i = 1, \ldots, k$. We wish to minimize

this function because the function describes the weighted sum of squared distances of each point in the region to its corresponding center. This is a way of measuring how expensive it is for the point in V_i to "travel" to the center $\mathbf{z}_i$. Minimizing this quantity forces the center of each Voronoi region to be the center of mass of the region.

1.1.2 The Discrete Case

The examination of CVTs in this paper focuses on tessellations of discrete sets of points representative of a region in $\mathbb{R}^2$. Therefore, instead of examining tessellations of a general region Ω, we wish to define a tessellation of a discrete set of points, $W = \{\mathbf{y}_i\}_{i=1}^m$ in $\mathbb{R}^2$.

Definition 1.3. A set $\{V_i\}_{i=1}^k$ is a tessellation of W if $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\cup_{i=1}^k V_i = W$. Given a set of points $\{\mathbf{z}_i\}_{i=1}^k$ belonging to $\mathbb{R}^2$, Voronoi sets in $\mathbb{R}^2$ are now defined as satisfying

$$\hat{V}_i = \{\mathbf{x} \in W \quad | \quad |\mathbf{x} - \mathbf{z}_i| < |\mathbf{x} - \mathbf{z}_j| \text{ for } j = 1, \ldots, k, j \neq i\},$$

where equality holds only for $i < j$.

Since we are interested in the case where the centroids act as the generators, we need a new definition for the mass centroids of the tessellation of a discrete set of points.

Definition 1.4. Given a density function ρ defined in W, the mass centroid $\mathbf{z}^*$ of a set $V \subset W$ is now defined by

$$\sum_{\mathbf{y} \in V} \rho(\mathbf{y}) |\mathbf{y} - \mathbf{z}^*|^2 = \inf_{\mathbf{z} \in V} \sum_{\mathbf{y} \in V} \rho(\mathbf{y}) |\mathbf{y} - \mathbf{z}|^2.$$

Extending the definition of the mass centroid from the general case (a tessellation of a region Ω), we can also define the mass centroid $\mathbf{z}^* = (\mathbf{z}_x^*, \mathbf{z}_y^*)$ of a discrete set of points in $\mathbb{R}^2$ as

$$\mathbf{z}_x^* = \frac{\sum_{\mathbf{i} \in V} x_i \rho(\mathbf{i})}{\sum_{\mathbf{i} \in V} \rho(\mathbf{i})}$$
$$\mathbf{z}_y^* = \frac{\sum_{\mathbf{i} \in V} y_i \rho(\mathbf{i})}{\sum_{\mathbf{i} \in V} \rho(\mathbf{i})},$$

where $\mathbf{x}_i$ and $\mathbf{y}_i$ are the x- and y-coordinates, respectively, of $\mathbf{i} \in V$.

Using ρ (properly normalized) as a probability density function, the energy function in the discrete case is

$$\mathcal{E}((\mathbf{z}_i, V_i), i = 1, \ldots, k) = \sum_{i=1}^{k} \sum_{\mathbf{y}_j \in V_i} \rho(\mathbf{y_j})|\mathbf{y_j} - \mathbf{z_i}|^2.$$

We wish to choose the sets $\{\mathbf{z}_i\}_{i=1}^k$ and $\{V_i\}_{i=1}^k$ so that the quantity $\mathcal{E}$ is minimized over all possible sets of k points belonging to W and all possible tessellations of W into k regions V_i, $i = 1, \ldots, k$.

1.1.3 CVTs and Minimization of $\mathcal{E}$

The tessellation with which we will be most concerned in the course of this project is a two-point tessellation of a square with unit area and over a unit density function. Therefore, we are interested in how the energy of our approximated CVTS compares to the energy of the true CVT. As discussed in [2] and [4], there are two (up to symmetry) two-point CVTs on the square: one whose boundary has opposite sides as its endpoints and one whose boundary has opposite vertices as its endpoints (see Figure 1). The former CVT has generators at the points $(0.25, 0.5)$ and $(0.75, 0.5)$ when the square has vertices at the coordinates $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. By Equation 1, we find that the energy associated with the stable two-point CVT on the square is

$$\begin{aligned}
\mathcal{E}((\mathbf{z}_i, V_i), i = 1, \ldots, k) &= \sum_{i=1}^{k} \int_{\mathbf{y}_j \in V_i} \rho(\mathbf{y}_j)|\mathbf{y}_j - \mathbf{z}_i|^2 \mathrm{d}\mathbf{y} \\
&= \int_0^1 \int_0^{0.5} |x - 0.25|^2 + |y - 0.5|^2 dxdy \\
&\quad + \int_0^1 \int_{0.5}^1 |x - 0.75|^2 + |y - 0.5|^2 dxdy \\
&\approx 0.1042.
\end{aligned}$$

On the same square, the unstable two-point CVT has generators at the points $(\frac{1}{3}, \frac{1}{3})$ and $(\frac{2}{3}, \frac{2}{3})$. By Equation 1, we find that the energy associated with the unstable two-point CVT on the square is

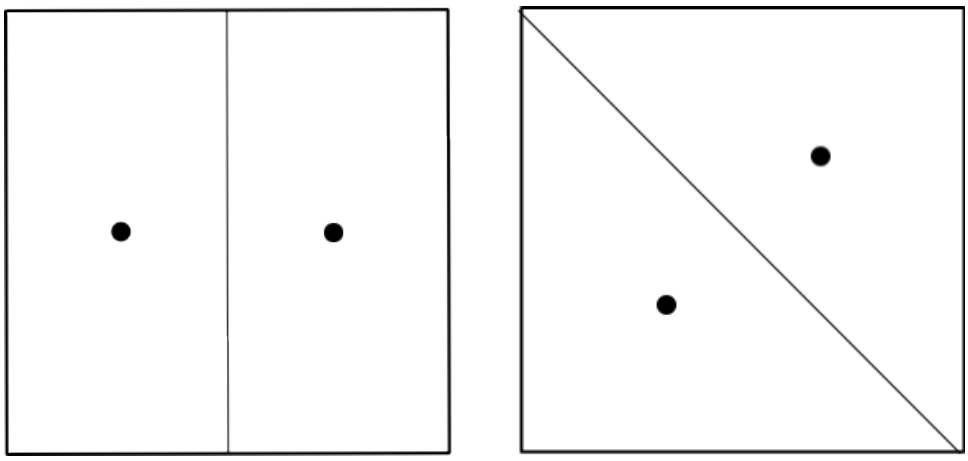

Figure 1: The two-point CVT on the left is stable while that on the right is not.

$$\begin{aligned}\mathcal{E}((\mathbf{z}_i, V_i), i = 1, \ldots, k) &= \sum_{i=1}^{k} \int_{\mathbf{y}_j \in V_i} \rho(\mathbf{y}_j)|\mathbf{y}_j - \mathbf{z}_i|^2 \mathrm{d}\mathbf{y} \\ &= \int_0^1 \int_0^{1-y} |x - \frac{1}{3}|^2 + |y - \frac{1}{3}|^2 dxdy \\ &\quad + \int_0^1 \int_{1-y}^1 |x - \frac{2}{3}|^2 + |y - \frac{2}{3}|^2 dxdy \\ &\approx 0.1111.\end{aligned}$$

Note that there are two CVTs on the square, but only one described above yields minimal energy. This leads us into the following theorem:

Theorem 1.1. *If a Voronoi tessellation minimizes $\mathcal{E}$, then it is a centroidal Voronoi tessellation [3].*

Note that the converse is not necessarily the case, as evidenced by the fact that the diagonal CVT of the square has higher energy than the vertical CVT.

1.1.4 Existing Algorithms for Generating CVTs

Methods for generating CVTs exist already, and we outline one below.

Lloyd's Method. Given a bounded, open set $\Omega \in \mathbb{R}^N$, a positive integer k, and a probability density function ρ defined on $\overline{\Omega}$,

1. select an initial set of k points $\{\mathbf{z}_i\}_{i=1}^k$;
2. construct the Voronoi tessellation $\{V_i\}_{i=1}^k$ of Ω associated with the points $\{\mathbf{z}_i\}_{i=1}^k$;
3. compute the mass centroids of the Voronoi regions $\{V_i\}_{i=1}^k$; these centroids are the new set of points $\{\mathbf{z}_i\}_{i=1}^k$;
4. if this set of points satisfies a previously determined convergence criterion, stop; otherwise, return to Step 2.

It is important to note that few theoretical results about Lloyd's algorithm exist, and there are still many open problems regarding its convergence [3]. Another common method is called MacQueen's, but it will remain unexplored in this paper [7].

Methods such as Lloyd's are already in place to generate CVTs. However, Lloyd's method does not explicitly capitalize on the fact that minimizing the quantity yielded by $\mathcal{E}$ leads to the generation of a CVT. Additionally, Lloyd's method converges logarithmically on the continuous problem and as a result will often stop converging after a point on discretized problems [3, 8]. In this paper, therefore, we explore the use of an optimization algorithm to minimize $\mathcal{E}$. One such method available to us is the *genetic algorithm*.

1.2 Genetic Algorithms

In a general sense, a genetic algorithm is an iterative process that mimics and exploits the characteristics of biological evolution's propensity for selecting for traits that optimize a particular population's effective adaptation to its environment.

Informally, a genetic algorithm is defined by the following components:

- a fitness function to be optimized,
- a population of chromosomes,
- selection of which chromosomes will reproduce (one could think of this as "survival of the fittest"),

- crossover to determine how new chromosomes are produced,
- and random mutation of chromosomes. [5]

A genetic algorithm works in a similar (if simplified) way to biological evolution. The fitness of a member of the population is determined by how well it optimizes the fitness function. The most fit members of the population then combine, with random mutation occurring, to produce new members of the population in the next generation.

In Section 1.2.1, we examine a simpler case so that we can contextualize the idea of a genetic algorithm. In this simplified case (which is exclusively present in this paper as an illustrative example), we assume that the function that we wish to optimize has domain in $\mathbb{R}^2$. After examining these ideas, we will better be able to grasp how we will apply a genetic algorithm to the task of approximating CVTs.

1.2.1 Example: Components of a Genetic Algorithm

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be a function that we wish to optimize, and let $C \subset \mathbb{R}^2$ be a randomly generated set of possible solutions that could optimize f. For instances demonstrating the form of the genetic algorithm shown below, see examples in [1] and [5]. Formally, we can define the components of a genetic algorithm in the following way:

- We call f a *fitness function*.
- We call C the *population*, where each $c \in C$ is called a *chromosome*.
- Let $c_1, c_2, \ldots, c_n$ be the elements in the set C arranged such that for $j < k$, c_j is better than or equivalent to c_k as a solution to the optimization of f. Let $P : C \to [0, 1]$ be a probability density function that assigns probabilities to each c_i based on how well it optimizes f relative to the other elements of C. Suppose that $c_k = m = (m_x, m_y)$, $c_l = p = (p_x, p_y) \in C$ for some k,l between 1 and n are chosen from C with probability $P(m)$ and $P(p)$, respectively (we call these elements m and p to represent "ma" and "pa"). The process of choosing m and p is called *mate selection*.
- At this stage, a way to proceed is to randomly choose one parameter (x or y) with equal probability, but one could imagine other ways that

involve both coordinates. Suppose that we choose the first way and suppose that x is the chosen parameter. Let β be a random number in $[0, 1]$. Define

$$x_{new1} = (1 - \beta)m_x + \beta p_x \text{ and } x_{new2} = (1 - \beta)p_x + \beta m_x$$

$$y_{new1} = m_y \text{ and } y_{new2} = p_y$$

so that

$$c_1' = (x_{new1}, y_{new1}) \text{ and } c_2' = (x_{new2}, y_{new2}).$$

The case in which y is the chosen parameter is similar. This process of creating new possible solutions to optimize f is called *one-point crossover*, and each c' is called an *offspring*.Repeat this process until the number of offspring equals n.

- Let $\mu \in [0, 1]$, where μ is chosen and fixed, not randomly generated. A good choice for μ depends on the context of the problem. Choose an offspring with probability μ and change one of its parameters (either x or y) to a random number in the parameter domain. This step is called *mutation*, where μ is the *mutation rate*.

Definition 1.5. A *genetic algorithm* is the iterative process of generating new populations of offspring according to rules like those listed above that are improved solutions to the fitness function.

1.2.2 The Importance of Mutation

A natural question that might arise is to ask why mutation is necessary when crossover (and the changes that come with it) is already occurring. If we have a way of changing our population through crossover, why introduce mutation? Mutation is necessary because it introduces a variation not obtainable through crossover alone. Given two members of the population m,$p \in C$, where $m = (m_x, m_y)$ and $p = (p_x, p_y)$, we can see that in one-point crossover the offspring of m and p will lie on the boundary of the rectangle formed when m and p are opposite vertices of the rectangle (see Figure 2).

To see this, suppose without loss of generality that we randomly choose to perform crossover on the x-coordinate, that $m_x \leq p_x$, and that β is the randomly chosen crossover point. We know that the offspring c_1' and c_2' are

$$c_1' = (x_{new1}, y_{new1}) \text{ and } c_2' = (x_{new2}, y_{new2}),$$

and since the x-coordinate is the location of crossover, we know that

$$x_{new1} = (1-\beta)m_x + \beta p_x \text{ and } x_{new2} = (1-\beta)p_x + \beta m_x,$$

and

$$y_{new1} = m_y \text{ and } y_{new2} = p_y.$$

It follows that

$$x_{new1} = (1-\beta)m_x + \beta p_x \geq (1-\beta)m_x + \beta m_x = m_x,$$

and

$$x_{new1} = (1-\beta)m_x + \beta p_x \leq (1-\beta)p_x + \beta p_x = p_x,$$

so

$$m_x \leq x_{new1} \leq p_x.$$

We can apply a similar argument to see that $m_x \leq x_{new2} \leq p_x$. Since the y-coordinate remains unchanged, the offspring will lie on the horizontal line between the x-coordinates of the two parents. In the case where crossover occurs on the y-coordinate, the offspring will lie on the vertical line between the parents' y-coordinates. It follows that in one-point crossover, offspring must lie on the boundary of the rectangle formed when the parent points are opposite vertices.

Another method, called *two-point crossover*, means that both coordinates undergo crossover, not just one. In two-point crossover, then, we can see that the offspring will lie in the region contained by the rectangle formed when m and p are opposite vertices of the rectangle.

2 CVTs Generated by Genetic Algorithms

This section discusses the implementation of a genetic algorithm to generate CVTs and the tests undergone to determine which parameters yield the best-performing genetic algorithm.

2.1 Genetic Algorithms and CVTs

In our case, we wish to adapt a genetic algorithm to the problem of constructing centroidal Voronoi tessellations. Our next question is: how do we

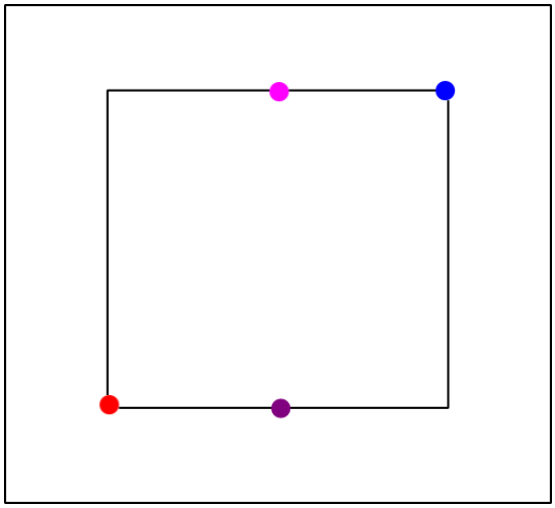

Figure 2: Consider this figure corresponding to the simplified example in Section 1.2.1. Two parent chromosomes (the red and blue points) produce offspring through crossover (the purple and pink points are examples of such offspring). In one-point crossover, these offspring must lie on the boundary of the rectangle formed when the parent points are opposite vertices. In two-point crossover, the offspring must lie within or on the boundary of the rectangle.

relate each of the components of a genetic algorithm listed above to a CVT? We will reference the simpler example in $\mathbb{R}^2$, but it is important to stress that the ideas presented below are merely analogous to those appearing in Section 1.2.1. We apply the components of the genetic algorithm discussed previously to find a parallel for each component in the CVT problem.

- Fitness function: $\mathcal{E}$, the error we are trying to minimize. We know from Theorem 1.1 that a Voronoi tessellation that minimizes $\mathcal{E}$ is a CVT.
- Population: a set C of n randomly generated Voronoi tessellations of a region, where each $c \in C$ is a particular Voronoi tessellation defined by its centers (i.e. c is a set of k points in $\mathbb{R}^2$). We define $c^{(i)}$ as the ith member of the population (so $1 \leq i \leq n$), where $c_j^{(i)} = (c_{jx}^{(i)}, c_{jy}^{(i)})$ is the jth center in member $c^{(i)}$ (so $1 \leq j \leq k$)and where $c_{jx}^{(i)}$ and $c_{jy}^{(i)}$ are the x- and y-coordinates of $c^{(i)}$, respectively.

- Mate selection: If the members of the population c_i are arranged in order from "best" to "worst" (i.e. c_1 yields the lowest value for $\mathcal{E}$ and c_n yields the highest value for $\mathcal{E}$), then we can choose population member j with probability

$$\frac{n-(j-1)}{\sum_{i=1}^{n} i}.$$

  This method of assigning probabilities to potential parents is called *rank order* [6].

- Crossover: In the case of the CVT, this occurs a little differently from in the example in Section 1.2.1. Suppose that according to rank order probability, members $c^{(k)} = m = (m_1, m_2, \ldots, m_k)$,$c^{(l)} = p = (p_1, p_2, \ldots, p_k) \in C$ are selected to mate with each other. Each element in m and p is an ordered pair in $\mathbb{R}^2$ representing one of the tessellation's k centers. Therefore, $m_i = (m_{ix}, m_{iy})$ for each i satisfying $1 \leq i \leq k$. The same is true of p. Crossover of the type we saw in Section 1.2.1 must be altered slightly to accommodate the different domain of the fitness function. In this case, we define one-point crossover as the following:

  1. Let β be a random number in $[0, 1]$.
  2. Randomly choose one parameter (x or y) with equal probability. Without loss of generality, suppose that x is the chosen parameter.
  3. Define

$$c_{ix}^{(1)'} = (1-\beta)m_{ix} + \beta p_{ix} \text{ and } c_{ix}^{(2)'} = (1-\beta)p_{ix} + \beta m_{ix},$$

$$c_{iy}^{(1)'} = m_{iy} \text{ and } c_{iy}^{(2)'} = p_{iy}$$

     so that

$$c_i^{(1)'} = (c_{ix}^{(1)'}, c_{iy}^{(1)'}) \text{ and } c_i^{(2)'} = (c_{ix}^{(2)'}, c_{iy}^{(2)'}).$$

  4. Repeat Steps 2 and 3 for all i that satisfy $1 \leq i \leq k$. This means that all centers will undergo crossover, yielding two offspring that replace the two parents.
  5. Repeat Step 6 for different parents until the new generation has n offspring in it.

- Random mutation: Once a $\mu \in [0, 1]$ is selected, perform mutation as described in Section 1.2.1.

With these characteristics in place, we need a way of measuring the effectiveness of the genetic algorithm. Since each member of the population in each generation is a set of points that defines a Voronoi tessellation of the region in question, each individual has an energy associated with it. We judge how well the genetic algorithm performs based on the value of the minimum energy in the last generation of the algorithm's run. Therefore, we can compare different implementations of the genetic algorithm by comparing the minimum energies they produce and, in turn, by testing our genetic algorithm on problems with known energies.

As can be immediately seen from the description of the use of a genetic algorithm in the context of a CVT, there are many ways of implementing crossover and mutation. In Section 2.2, we test different crossover parameters to determine how the following variations affect the performance of the algorithm.

- **Population Size.** Compare effectiveness of small populations vs. large populations.
- **Number of Generations.** Compare effectiveness of few generations vs. many generations.
- **Crossover.**
  - Location of crossover—Compare effectiveness of using same parameter β for each crossover operation in a given generation vs. using a different crossover point β for each mate pair.
  - Method of crossover—Compare effectiveness of performing crossover at a randomly selected x- or y-coordinate (one-point crossover) vs. performing crossover at both coordinates (two-point crossover).
  - Order of crossover—Compare effectiveness of crossing over in order determined by random selection of mates vs. re-ordering points according to relative proximity.
- **Mutation.**
  - Compare effectiveness of different values for the mutation rate μ.

– Compare effectiveness of mutating chosen parameters to a random number in the domain vs. mutating parameters within a neighborhood surrounding the initial parameter.

2.2 Testing Genetic Algorithm Parameters

In this section, we transition from discussing the genetic algorithm's features in the context of CVTs to testing our genetic algorithm using different parameters. With a working genetic algorithm that converges to a CVT, we are interested in finding the parameters that make it as efficient as possible. To do this, we began with a two-point tessellation of a square region. After running the genetic algorithm successive times, we compared the energy obtained by the genetic algorithm with the known minimum energy of a two-point square CVT. Therefore, we judge the performance of the genetic algorithm according to its ability to produce tessellations with lowest energy.

For a more detailed explanation of the implementation of the genetic algorithm in MATLAB, see Appendix A.1. For an outline of statistical methods when evaluating the performance of altered genetic algorithms, see Appendix A.2. For complete MATLAB code, see Appendix A.3.

2.2.1 Changing Population Size and Number of Generations

As might be expected, more generations result in producing tessellations that have energies closer to the energy of the CVT. In other words, the more generations the genetic algorithm goes through, the better CVT it produces. Similarly, larger population sizes have an improving effect on the accuracy of the generated CVT.

We can see from the tests we ran that larger populations resulted in lower energy, and that more generations resulted in lower energy (see Figure 3).

2.2.2 Changing Crossover Methods

Varying the Location of Crossover. Recall that in the operation of crossover, the parameter β is chosen randomly to determine how the coordinates of the parent centers combine to produce the offspring. In the original genetic algorithm, β is chosen randomly at every generation, but it stays the same throughout any given generation for every combination of coordinates.

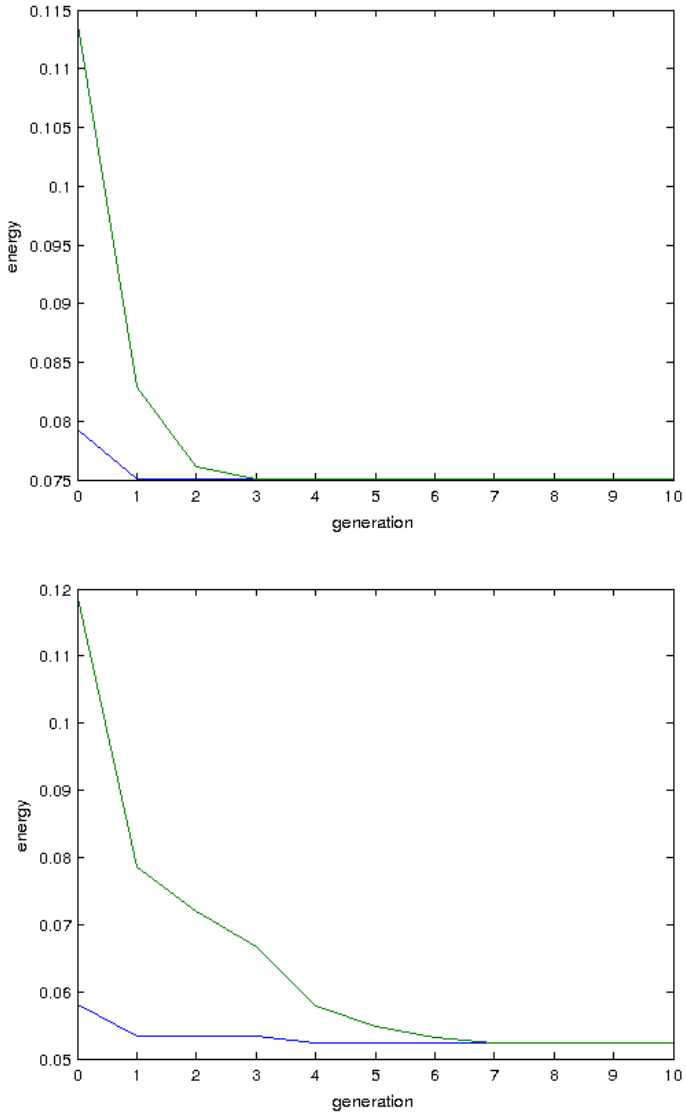

Figure 3: The minimum energy in the population at each generation (the energy of the fittest individual in the population) is shown with a blue line and the average energy for a generation is given by the green line. The energy for the smaller population (population of four on the top) is higher than the energy for a larger population (population of thirty on the bottom). The difference between energies produced by the two populations was statistically significant with a p-value of 6.861×10^{-10}.

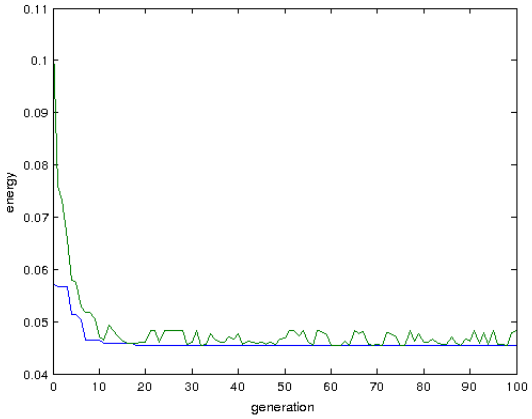

Figure 4: As the genetic algorithm progresses, the minimum energy obtained decreases over successive generations.

In this variation, β is randomly chosen for every instance of coordinate combination. In the trials that we ran, changing β within generations yielded a statistically significantly lower mean energy.

One-Point Crossover vs. Two-Point Crossover. In the crossover discussed in Section 2.1, we used a method called *one-point crossover*, which means that a coordinate—either x or y—was chosen at random and then crossover occurred at that coordinate. In *two-point crossover*, crossover occurs at both the x- and the y-coordinates. If $m = (m_1, m_2, \ldots, m_k)$ and $p = (p_1, p_2, \ldots, p_k)$ are the "ma" and "pa" chromosomes, then two-point crossover yields offspring centers of the following form:

$$c_{ix}^{(1)'} = (1-\beta)m_{ix} + \beta p_{ix} \text{ and } c_{ix}^{(2)'} = (1-\beta)p_{ix} + \beta m_{ix},$$
$$c_{iy}^{(1)'} = (1-\beta)m_{iy} + \beta p_{iy} \text{ and } c_{iy}^{(2)'} = (1-\beta)p_{iy} + \beta m_{iy}.$$

As in Section 2.1, this process is repeated for all centers in m and p and for all mating pairs that generate the new generation.

In this variation, there was no significant difference in mean energy yielded between the two methods.

Re-Ordering Points. An interesting problem we encounter is how to pair two mates once they have been selected. The way we originally set it up, we did not care which particular element of the "mother" set "mated" with another element of the "father" set. One thing we might wonder about is whether having two elements mate who are not near each other at all would result in a different convergence of the genetic algorithm. To do this, we rearrange the elements in the "mother" set so that they have the same index as the element in the "father" set which is closest to it. So far, it seems that this rearrangement would be more useful in larger populations, perhaps because there is more variation in the mother and father sets when the population is larger. See Figure 5 for an illustration of this process.

After comparing these methods of crossover, there was no significant difference in the energies yielded by the different methods when applied to approximating two-point CVTs, but the difference was significant (with p-value 0.017) for ten-point CVTs.

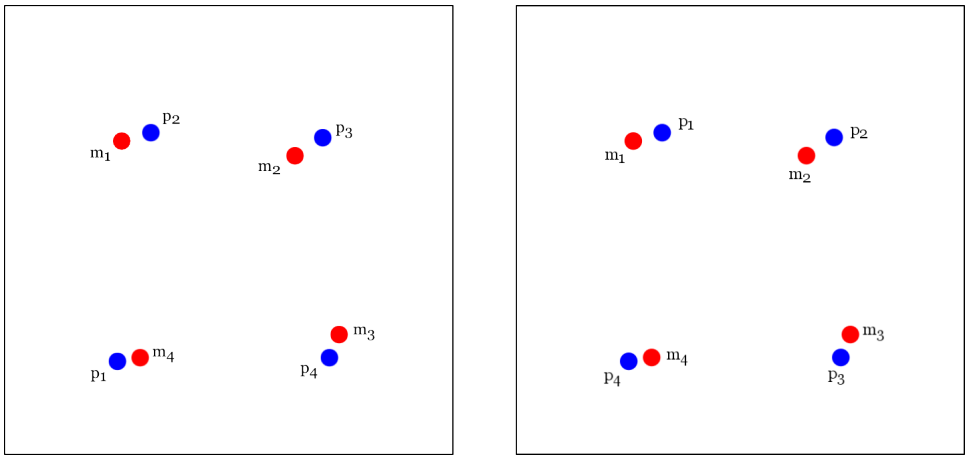

Figure 5: **Left:** This figure shows two parents, m and p in C, which have been selected to mate. Note that according to the order in which they appear in the chromosome, each center is not slated to perform crossover with the point closest to it; instead, it will perform crossover with points farther away. As can be observed by sight, this might lose some of the fitness achieved by m and p, especially since each is fairly close to the four-point CVT on a square. **Right:** We re-order the points so that each center performs crossover with the point nearest to it.

2.2.3 Changing Mutation Methods

Optimal Values of μ. We wish to examine the parameter μ when implementing our genetic algorithm because its optimal performance depends greatly on the context of the particular problem that the genetic algorithm is attempting to solve [1].

Tests were performed on a genetic algorithm approximating two-point CVTs of a square region. After testing a variety of values for μ between 0 and 1, we found that $\mu = 0.2$ had a significant effect in improving the performance of the genetic algorithm.

Restricting Mutation to a Neighborhood. In the method of mutation discussed in Section 1.2.1, if a parameter is chosen to be mutated, then it is re-set to a number in the domain of the tessellation. This might seem to have the potential to disrupt the fitness of the individual if the mutation is very far from the original point. Therefore, we implement a variation that restricts the mutation of a point to a neighborhood around the point. This variation yielded no significant difference in mean energy.

2.3 Lloyd's-Genetic Algorithm Hybrid

Since it is clear that Lloyd's method is much less computationally expensive that our genetic algorithm, there is essentially no hope of using our genetic algorithm to generate CVTs (or approximations of CVTs) faster or more efficiently than Lloyd's method does. However, since the energy yielded by tessellations that Lloyd's method produces does tend to level off, we might expect that we could use the tessellation produced by Lloyd's method as a starting point for the genetic algorithm. Then, by feeding those points into the genetic algorithm, we allow the genetic algorithm to progress and push the energy of the tessellation lower, therefore approximating a CVT more accurately.

To do this, we allow Lloyd's method to run for a number of iterations that lets it converge to something close to a CVT. Let Z be the k-by-2 matrix representing the points in the k-point tessellation produced by the last iteration of Lloyd's method. The starting population C of the genetic algorithm is composed of Z and $n-1$ other matrices whose entries are all in a neighborhood of the entries of Z and where n is the population size of

the genetic algorithm. The genetic algorithm we use has parameters based on the optimal settings we investigated in Section 2.2.

After running this hybrid optimization program, we find that the hybrid produced a lower energy that was statistically significant.

3 Conclusion

Implementing a genetic algorithm that also employs Lloyd's method to generate CVTs improves the results gained from Lloyd's method. By pushing the energy of tessellations lower, our algorithm gets us closer to a CVT.

Future Inquiry. This project was simply a scratch in the surface of what is possible to be investigated and tested in terms of the ability of a genetic algorithm (or a Lloyd's-genetic algorithm hybrid) to produce CVTs. The following are possible questions to explore further:

- Does it help to change the mutation rate over time so that the alterations to the population become more refined over successive generations?
- How does the genetic algorithm perform under different numbers of centers, non-constant density functions, differently shaped regions, higher dimensions, and non-Euclidean metrics?
- Do the statistically insignificant results obtained in Section 2.2 become significant when run over more trials?

Acknowledgments. The author would like to thank Albert Schueller for his guidance and support while composing this project, Sophie De Arment for her impeccable edits and ideas, Doug Hundley for teaching an engaging mathematical modeling course and providing the backbone of much of the code used in this project, and Patrick Keef for conducting the senior project class.

A Appendices

A.1 Appendix: Explanation of Code

This appendix examines the approaches we use to solve our specific problem. We use MATLAB, a computing environment that allows us to implement our algorithm and run many trials.

A.1.1 Initializing the Starting Population

Before the genetic algorithm can run, a starting population must be defined. Recall that a population is made up of candidate solutions to the minimization problem, so the members of the population will be Voronoi tessellations of the region. As mentioned before, any Voronoi tessellation is uniquely defined by its generators, so the population will take the form of sets of points in the region. From this, we can see that we must define a population size n, the number of centers k, and the region. For our purposes, the region is a rectangle defined by a, b, c, and d, respectively the left, right, lower, and upper boundaries of the rectangle.

In general, this population is randomly generated. Therefore, we have a population C of size n. Each member of the population (which itself generates a particular tessellation of the region) is a $c_i \in C$, where $1 \leq i \leq n$. Since each c_i must contain the information about the k generators that define it, c_i is a k-by-2 matrix, where each row contains the x- and y-coordinates of each of the k centers in the tessellation.

A.1.2 Genetic Algorithm Parameters

After setting up our population C, we turn to setting the parameters of our genetic algorithm. A mutation rate μ is chosen, where μ determines the proportion of the population that will undergo random mutation. We must also define the proportion K of the population that will be kept to the next generation; in other words, in a population of size n, the fittest Kn members will also be members of the next generation's population. Since Kn individuals will automatically make it to the next generation, the remaining ones are candidates for mating. The number of matings M is given by $(n - Kn)/2$, since each mating requires two parents.

Recall that the process of mate selection requires us to define a probability distribution. We can do this using rank order (described in Section 2.1),

where we are ranking the remaining $n - Kn$ members of the population available for mating.

A.1.3 Sorting the Initial Population

Now that we have generated a random set of Voronoi tessellations (defined by the centers of each Voronoi region) of our rectangular region, we must sort that population according to how well each member member performs when the energy functional is applied to it. Since we are looking to minimize the energy, a particular tessellation that yields a lower energy is considered to be more fit.

Accordingly, with each member of the population c_i we associate its Voronoi tessellation V_i. We can use this tessellation and the centers to define a vector $\mathbf{e}$, where each entry e_i contains the energy associated with member c_i. After we have all the energies, we can sort the population so that the member with the lowest energy (the fittest member) comes first and the least fit comes last.

A.1.4 Running the Genetic Algorithm

Now, we come to the main loop of the genetic algorithm, where mate selection, crossover, and mutation happen. According to the rank order probability distribution, we select members of the population to be mother and father sets. Let A,$B \subset C$ be sets, where each element $a_j \in A$ and $b_j \in B$ represent a particular "mother" and "father" that will mate during crossover and where $1 \leq j \leq M$.

Since each a_j and b_j is a set of k centers with x- and y-coordinates, we now perform crossover so that a_j crosses over with b_j for each j. Once we have the offspring, we replace the members of the population C that do not survive to the next generation. After this process, we have a population C comprised of the members that we kept from the last generation and the offspring of the mother and father sets.

After crossover occurs, we can randomly mutate parameters according to μ.

Once these operations are complete, we are left with the population for the following generation. We repeat the process of sorting the population according to its performance under the energy functional.

A.1.5 Table of Variable Names

The table in this section is a key for reading the variables in the MATLAB code in relation to the variables used in this text.

Mathematical Variable	Code Variable	Meaning
a,b,c,d	`a,b,c,d`	boundaries of rectangular region
A,B	`PopMa,PopPa`	centers in mother, father sets
C	`Pop`	population for a given generation
c_i	`Pop{i}`	particular set of centers
$\mathbf{e}$	`e`	vector of energies of population
K	`popKept`	proportion of population kept
k	`numcenters`	number of generators in tessellation
μ	`mutrate`	mutation rate
M	`M`	number of matings
n	`popsize`	number of tessellations per generation
V_i	`V{i}`	particular Voronoi tessellation

A.2 Appendix: Statistical Methods

This section discusses the statistical methods used to analyze the significance of the results obtained when performing the tests in Section 2.2. Each subsection below states the parameter values for each experiment, groups' mean energies and standard deviations, and statistical tests applied to each dataset. The p-value associated with each test is stated; recall that the p-value is the probability of obtaining a difference in means as extreme as or more extreme than that observed if no true difference exists.

Unless otherwise stated, each experiment employs one-point crossover, a single value for β within a particular generation, no re-ordering of points during crossover, and mutation that re-sets a single x- or y-coordinate to a random number in the region's domain.

Population Size. This experiment measured the difference in energy yielded when the genetic algorithm was run with a small population and a large population. The following parameters were used: `maxit`=10, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
`popsize`=4	0.1164908	0.008707167
`popsize`=30	0.1072654	0.002226414

We ran a two-sample t-test to detect if the mean of the larger population was less than that of the smaller population. The p-value was 6.861×10^{-10}. At a confidence level of 95%, these results are statistically significant.

Number of Generations. This experiment measured the difference in energy yielded when the genetic algorithm was run with a small number of generations and a large number of generations. The following parameters were used: `popsize`=10, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
`maxit`=5	0.1132368	0.005366084
`maxit`=50	0.1093038	0.004294878

We ran a two-sample t-test to detect if the mean of the larger number of generations was less than that of the smaller number of generations.The

p-value was 5.355×10^{-5}. At a confidence level of 95%, these results are statistically significant.

Location of Crossover. This experiment measured the difference in energy yielded when the genetic algorithm was run with a single β within a generation or a changing β. The following parameters were used: `maxit`=10, `popsize`=10, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
Changing β	0.1081566	0.002433683
Single β	0.1117608	0.004340098

We ran a two-sample t-test to detect if the mean energy of the changing β group was less than that of the single β group. The p-value was 1.087×10^{-6}. At a confidence level of 95%, these results are statistically significant.

One-Point or Two-Point Crossover. This experiment measured the difference in energy yielded when using one-point crossover or two-point crossover. The following parameters were used: `maxit`=10, `popsize`=10, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
One-Point Crossover	0.1117608	0.004340098
Two-Point Crossover	0.1107604	0.004238703

We ran a two-sample t-test to detect if the mean of the two-point crossover group was less than that of the one-point crossover group. The p-value was 0.8768. At a confidence level of 95%, these results are not statistically significant.

Re-Ordering Points, Part 1. This experiment measured the difference in energy yielded when using different methods of re-ordering points or members of the population. The following parameters were used: `maxit`=10, `popsize`=20, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
No re-ordering	0.1082850	0.002943480
Re-ordering members	0.1089898	0.003405675
Re-ordering points	0.1093876	0.003574437

We ran an analysis of variance (ANOVA) test to detect a difference in means among groups. The p-value was 0.246. At a confidence level of 95%, these results are not statistically significant.

Re-Ordering Points, Part 2. This experiment measured the difference in energy yielded when using different methods of re-ordering points or members of the population. The following parameters were used: `maxit`=20, `popsize`=20, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
No re-ordering	0.1079398	0.002972491
Re-ordering members	0.1077710	0.002226227
Re-ordering points	0.1078742	0.003070688

We ran an analysis of variance (ANOVA) test to detect a difference in means among groups. The p-value was 0.954. At a confidence level of 95%, these results are not statistically significant.

Re-Ordering Points, Part 3. This experiment measured the difference in energy yielded when re-ordering points or not on 10-point tessellations. The following parameters were used: `maxit`=10, `popsize`=10, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=10. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
No re-ordering	0.0252692	0.002348732
Re-ordering points	0.02432336	0.002041584

We ran two-sample t-test to detect whether the mean energy of the re-ordered group was lower than that of the original group. The p-value was 0.01707. At a confidence level of 95%, these results are statistically significant.

Optimal Value of μ. This experiment measured the difference in energy yielded when using different values for the mutation rate. The following parameters were used: `maxit`=10, `popsize`=10, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
`mutrate`=0.01	0.1125336	0.0048183
`mutrate`=0.05	0.1092922	0.002872124
`mutrate`=0.1	0.1079248	0.002539257
`mutrate`=0.2	0.1075016	0.00180067
`mutrate`=0.4	0.109065	0.002887795
`mutrate`=0.6	0.1133278	0.005265273
`mutrate`=0.8	0.1150554	0.006061004
`mutrate`=1.0	0.118436	0.008242524

We ran an analysis of variance (ANOVA) test to detect a difference in means among groups. The p-value was 2×10^{-16}. At a confidence level of 95%, these results are not statistically significant.

Mutation in a Neighborhood, Part 1. This experiment measured the difference in energy yielded when using different methods of mutating. The following parameters were used: `maxit`=10, `popsize`=20, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
Normal mutation	0.1083520	0.002950079
`mutrad`=0.1, new x or y	0.1082132	0.002470340
`mutrad`=0.1, new point	0.1079226	0.002883508

We ran an analysis of variance (ANOVA) test to detect a difference in means among groups. The p-value was 0.733. At a confidence level of 95%, these results are not statistically significant.

Mutation in a Neighborhood, Part 2. This experiment measured the difference in energy yielded when using different methods of mutating. The following parameters were used: `maxit`=20, `popsize`=20, `mutrate`=0.01, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 50.

Group	Group's Mean Energy	Group's Std. Dev.
Normal mutation	0.1078716	0.002494082
`mutrad`=0.1, new x or y	0.1076344	0.002439304
0.1<`mutrad`<1, new x or y	0.1071652	0.002070959
0.1<`mutrad`<1, new point	0.1081428	0.002584948

We ran an analysis of variance (ANOVA) test to detect a difference in means among groups. The p-value was 0.22. At a confidence level of 95%, these results are not statistically significant.

Lloyd's-Genetic Algorithm Hybrid. This experiment measured the difference in energy yielded when using Lloyd's method alone and using a Lloyd's-genetic algorithm hybrid. The following parameters were used: `maxitLloyds`=10, `maxit`=20, `popsize`=20, `mutrate`=0.2, `popKept`=0.5, `res`=1000, `numcenters`=2. Each group had sample size 100.

Group	Group's Mean Energy	Group's Std. Dev.
Lloyd's	0.1046979	0.000448
Lloyd's-GA hybrid	0.104622	0.000109

We ran a paired (also called "dependent") two-sample t-test to detect whether the mean energy of the hybrid algorithm was lower than that of Lloyd's method. The p-value was 0.02024. At a confidence level of 95%, these results are statistically significant.

A.3 Appendix: MATLAB Code for Genetic Algorithms

Written with guidance from the work in [6].

A.3.1 Helper Function: Make VTs Given Centers

```
function P=assignregion(X,C)
% function C=kmeansUpdate(X,C)
%  Performs 1 step of the k-means algorithm where X is p by n (data)
%  and C is k by n (k clusters)
% P holds the sorted data
% C is the new matrix of clusters
% Err is a vector of distortion errors

%Check to see if dimensions are correct:
[m,n]=size(X);
[k,p]=size(C);

% Compute the distances from all the points to each center
D=edm(X(:,1:2),C);

% Find the minimum distance for each and sort
[Vals,Idx]=min(D,[],2);

% Resort the data to get the new centers.
for j=1:k
    idx=find(Idx==j);
    if length(idx)>0
        P{j}=X(idx,:);
    end
end
```

A.3.2 Helper Function: Calculate Squared Distances

```
function z=edm(w,p)
% A=edm(w,p)
% Input: w, number of points by dimension
% Input: p is number of points by dimension
% Ouput: Matrix z, number points in w by number pts in p
% which is the squared distance from one point to another
```

```
%Check dimensions to make sure data was input properly:
[S,R] = size(w);
[Q,R2] = size(p);
p=p';
if (R ~= R2)
    error('Inner matrix dimensions do not match.');
end


z = zeros(S,Q);   %Allocate space for the EDM
if (Q<S)
    p = p';
    copies = zeros(1,S);
    for q=1:Q
        z(:,q) = sum((w-p(q+copies,:)).^2,2);
    end
else
    w = w';
    copies = zeros(1,Q);
    for i=1:S
        z(i,:) = sum((w(:,i+copies)-p).^2,1);
    end
end
```

A.3.3 Helper Function: Calculate Energy

```
function e = energy(P,C,res)
% function e=energy(P,C)
%   Finds the energy (or distortion) associated with the Voronoi
%   tessellation formed by the points in P associated with the centers
%   C. P must be a cell array, with each cell containing the points for
%   respective centers according to their indices.

[m,k]=size(P);
e=0;

for i=1:k
    [ncluster,p]=size(P{i});
    z=edm(P{i}(:,1:2),C(i,:));
```

```
    for j=1:ncluster
        e=e+P{i}(j,3)*z(j,1);
    end
end

e=e/res^2;
```

A.3.4 Helper Function: Rank Order Probability

```
function Action=RandChooseN(P,N)
% function Action=RandChooseN(P,N)
%   Choose N numbers from 1 to length(P) using the
%   probabilities in P.  For example, if P=[0.1,0.9],
%   we choose "1" 10% of the time, and "2" 90% of
%   the time.   Selection is done WITH replacement,
%   so, for example, if N=3, we could return [2, 2, 2]

%Set up the bins
BinEdges=[0, cumsum(P(:)')];
Action=zeros(1,N);

for i=1:N
    x=rand;
    Counts=histc(x,BinEdges);
    Action(i)=find(Counts==1);
end
```

A.3.5 Helper Function: Calculate Mass Centroids

```
function Z = masscentroid(V)
% function Z = masscentroid(V)
%   Input is cell array, V, that defines a Voronoi tessellation of the
%   region, with each cell containing the points for
%   respective centers according to their indices.
%
%   Output is an kx2 array Z, where k is the number of regions in the
%   tessellation. Z contains the mass centroids of each region, ordered
%   according to their indices.

[m,k]=size(V);
```

```
Z=zeros(k,2);

for i=1:k
    zxnum=0;
    zynum=0;
    [p,n]=size(V{i});
    for j=1:p
        zxnum=zxnum+V{i}(j,1)*V{i}(j,3);
        zynum=zynum+V{i}(j,2)*V{i}(j,3);
    end
    zweight=sum(V{i}(:,3))
    Z(i,1)=zxnum/zweight;
    Z(i,2)=zynum/zweight;
end
```

A.3.6 GA: Generate CVTs

```
%% Using a Genetic Algorithm to Compute a CVT

clear   % makes sure to clear variable assignments

%% Initializing the starting population in the first generation

% Stopping criteria
maxit=10;  %Max number of iterations
mincost=-99999999;  %Minimum cost

% Domain of VT
a=0;   % left endpoint in x-direction
b=1;   % right endpoint in x-direction
c=0;   % lower endpoint in y-direction
d=1;   % upper endpoint in y-direction

% Initialization parameters
popsize = 10;   % population size
                %(number of VTs created per generation)
numcenters = 2; % number of centroids in each VT

for i=1:popsize    % this loop randomly generates
                   % a starting population of centers (determined by
```

```
                    % popsize and numcenters)
    Pop{i} = rand(numcenters,2);
    for j=1:numcenters
        Pop{i}(j,1)=a+(b-a)*Pop{i}(j,1);
        Pop{i}(j,2)=c+(d-c)*Pop{i}(j,2);
    end
end

% Genetic algorithm parameters
mutrate=0.01;  %Mutation rate
popKept=0.5;  %Fraction of the population to keep
keep=floor(popKept*popsize);  %How many individuals are kept
if mod(popsize-keep,2)==1     %Makes number of matings work.
    keep=keep-1;
end
M=ceil((popsize-keep)/2); % number of matings;
                          % 2 mates create 2 offspring
crossprob=round(rand(maxit,numcenters)); %0= x-coord, 1= y-coord
nmut=ceil((popsize-1)*2*numcenters*mutrate);  %Number of mutations

% Probability distribution for mate selection
probs=(keep:-1:1)/sum(1:keep);  %Probability is rank ordering

% Set up grid of reference points to mimic continuous surface
res=1000;   % number of increments in each dimension
n=res+1;    % makes dimensions of dataset work

[X,Y]=meshgrid(a:(b-a)/res:b,c:(b-a)/res:d);    % implement grid
                                                %   over dimensions
                                                %   of VT's domain
Xnew=reshape(X,[n*n,1]);     % see Matlab documentation on
                             %   meshgrid and reshape for more info
Ynew=reshape(Y,[n*n,1]);

data=zeros(n*n,2);   % puts coordinates of grid points
                     %   into a single n^2 X 2 matrix
for i=1:n*n
    data(i,1)=Xnew(i,1);
    data(i,2)=Ynew(i,1);
end
```

```
% Density function
RHO=ones(n*n,1);      % an n^2 X 1 matrix that stores
                      %   the weights for each grid point;
                      %   this one is for a constant
                      %   density function

% Associate grid points with density function
data=[data,RHO];      % grid points side-by-side with weights

%{
% Plot Voronoi cells for each member of the population
figure
for i=1:popsize
    subplot(1,popsize,i);
    voronoi(Pop{i}(:,1),Pop{i}(:,2))  % Built-in Matlab command
                                      %   for Voronoi cells only
                                      %   works with 3 or more centers
end
%}

%% Sort initial population according to performance

e = zeros(popsize,1);

for i=1:popsize
    V{i}=assignregion(data,Pop{i});
    e(i,1)=energy(V{i},Pop{i},res);
end

[e,idx]=sort(e);  % Default sort is from small to large
for i=1:popsize
    temp{i}=Pop{idx(i)};
end
for i=1:popsize
    Pop{i}=temp{i};
end

minenergy(1)=min(e);  % Minimum energy, for plotting later
meanenergy(1)=mean(e);  %Mean cost for this population
```

```
                        %  (for plotting later)

good=[0.25,0.5;0.75,0.5];
refVor=assignregion(data,good);
refenergy=energy(refVor,good,res);

%% Main Loop
%  This is where the genetic algorithm is implemented.

iga=0;

while iga<maxit

    iga=iga+1;

    % Pair up and mate:
    ma=RandChooseN(probs,M);
    pa=RandChooseN(probs,M);

    % Set up crossover and mutation:
    idx2=keep+1:popsize;
    beta=rand;

    for i=1:M
        PopMa{i}=Pop{ma(i)};
        PopPa{i}=Pop{pa(i)};
    end

    for j=1:M
        for i=1:numcenters
            if crossprob(iga,i)==0  %Crossover the x-coordinate
                Pop{idx2(2*j-1)}(i,1)=(1-beta)*PopMa{j}(i,1)+
                    beta*PopPa{j}(i,1);
                Pop{idx2(2*j-1)}(i,2)=PopMa{j}(i,2);
                Pop{idx2(2*j)}(i,1)=(1-beta)*PopPa{j}(i,1)+
                    beta*PopMa{j}(i,1);
                Pop{idx2(2*j)}(i,2)=PopPa{j}(i,2);
            else                    %Crossover the y-coordinate
                Pop{idx2(2*j-1)}(i,1)=PopMa{j}(i,1);
                Pop{idx2(2*j-1)}(i,2)=(1-beta)*PopMa{j}(i,2)+
```

```
                beta*PopPa{j}(i,2);
            Pop{idx2(2*j)}(i,1)=PopPa{j}(i,1);
            Pop{idx2(2*j)}(i,2)=(1-beta)*PopPa{j}(i,2)+
                beta*PopMa{j}(i,2);
        end
    end
end

% Mutation
mPop=sort(round(rand(1,nmut)*(popsize-1))+1)
mcol=ceil(rand(1,nmut)*2);
for ii=1:nmut
    [r,c]=size(Pop{mPop(ii)});
    mcenter=round(rand(1,nmut)*(r-1))+1
    Pop{mPop(ii)}(mcenter(ii),mcol(ii))=rand;  % only works for
                                               %   domain specified
                                               %   above
end


% New cost, set up for next iteration:
for i=1:popsize
    V{i}=assignregion(data,Pop{i});
    e(i,1)=energy(V{i},Pop{i},res);
end

[e,idx]=sort(e);  % Default sort is from small to large
    for i=1:popsize
        temp{i}=Pop{idx(i)};
    end
    for i=1:popsize
        Pop{i}=temp{i};
    end
minenergy(iga+1)=min(e);  % Minimum energy, for plotting later
meanenergy(iga+1)=mean(e);  %Mean cost for this population
                            %  (for plotting later)

%{
figure
voronoi(Pop{1}(:,1),Pop{1}(:,2))
```

```
    %}

    % Stopping criteria
    if iga>maxit || e(1)<mincost
        break
    end

    end % End of the while loop

figure
iters=0:length(minenergy)-1;
plot(iters,minenergy,iters,meanenergy,'-');
xlabel('generation');ylabel('energy');
```

A.3.7 GA: Varying Location of Crossover

```
%% This section replaces crossover in the original program.

% Set up crossover and mutation:
idx2=keep+1:popsize;

for i=1:M
    PopMa{i}=Pop{ma(i)};
    PopPa{i}=Pop{pa(i)};
end

for j=1:M
    for i=1:numcenters
        if crossprob(iga,i)==0  %Crossover the x-coordinate
            Pop{idx2(2*j-1)}(i,1)=(1-rand)*PopMa{j}(i,1)+
                rand*PopPa{j}(i,1);
            Pop{idx2(2*j-1)}(i,2)=PopMa{j}(i,2);
            Pop{idx2(2*j)}(i,1)=(1-rand)*PopPa{j}(i,1)+
                rand*PopMa{j}(i,1);
            Pop{idx2(2*j)}(i,2)=PopPa{j}(i,2);
        else                    %Crossover the y-coordinate
            Pop{idx2(2*j-1)}(i,1)=PopMa{j}(i,1);
            Pop{idx2(2*j-1)}(i,2)=(1-rand)*PopMa{j}(i,2)+
                rand*PopPa{j}(i,2);
```

```
            Pop{idx2(2*j)}(i,1)=PopPa{j}(i,1);
            Pop{idx2(2*j)}(i,2)=(1-rand)*PopPa{j}(i,2)+
                rand*PopMa{j}(i,2);
        end
    end
end
```

A.3.8 GA: Two-Point Crossover

```
%% This section replaces crossover in the original program.

% Set up crossover and mutation:
idx2=keep+1:popsize;
beta=rand;

for i=1:M
    PopMa{i}=Pop{ma(i)};
    PopPa{i}=Pop{pa(i)};
end

for j=1:M
    for i=1:numcenters
        Pop{idx2(2*j-1)}(i,1)=(1-beta)*PopMa{j}(i,1)+beta*PopPa{j}(i,1);
        Pop{idx2(2*j-1)}(i,2)=(1-beta)*PopMa{j}(i,2)+beta*PopPa{j}(i,2);
        Pop{idx2(2*j)}(i,1)=(1-beta)*PopPa{j}(i,1)+beta*PopMa{j}(i,1);
        Pop{idx2(2*j)}(i,2)=(1-beta)*PopPa{j}(i,2)+beta*PopMa{j}(i,2);
    end
end
```

A.3.9 GA: Re-Order Members of Population

```
%% This section is implemented before crossover.

MaAvg=zeros(M,2);
PaAvg=zeros(M,2);
for i=1:M
    MaAvg(i,1)=sum(PopMa{i}(:,1))/M;
    MaAvg(i,2)=sum(PopMa{i}(:,2))/M;
    PaAvg(i,1)=sum(PopPa{i}(:,1))/M;
    PaAvg(i,2)=sum(PopPa{i}(:,2))/M;
```

```
end

distances=edm(MaAvg,PaAvg)
for i=1:M-1
    [mindist,minindex]=min(distances(i:M,i));
    tempdist=distances(i,:);
    distances(i,:)=distances(i+minindex-1,:);
    distances(i+minindex-1,:)=tempdist
    temp2{i}=PopMa{i};
    PopMa{i}=PopMa{i+minindex-1};
    PopMa{i+minindex-1}=temp2{i};
end
```

A.3.10 GA: Re-Order Points

```
%% This section is implemented before crossover.

for i=1:M
    distances=edm(PopMa{i},PopPa{i});
    for j=1:numcenters-1
        [mindist,minindex]=min(distances(j:numcenters,j));
        tempdist=distances(j,:);
        distances(j,:)=distances(j+minindex-1,:);
        distances(j+minindex-1,:)=tempdist;
        temp2=PopMa{i}(j,:);
        PopMa{i}(j,:)=PopMa{i}(j+minindex-1,:);
        PopMa{i}(j+minindex-1,:)=temp2;
    end
end
```

A.3.11 GA: Mutation in a Neighborhood—Single Parameter

```
%% This section replaces mutation in the original program.

% Mutation
mPop=sort(round(rand(1,nmut)*(popsize-1))+1);
mcol=round(rand(1,nmut)*2);
for ii=1:nmut
    [r,c]=size(Pop{mPop(ii)});
```

```
    mcenter=ceil(rand(1,r-1))+1;
    oldcenter=Pop{mPop(ii)}(mcenter(ii),mcol(ii));
    newcenter=oldcenter+(rand-0.5)*2*mutrad;
    if newcenter<0
        newcenter=0;
    end
    if newcenter>1
        newcenter=1;
    end
    Pop{mPop(ii)}(mcenter(ii),mcol(ii))=newcenter;
end
```

A.3.12 GA: Mutation in a Neighborhood—New Point

```
%% This section replaces mutation in the original program.

% Mutation
mPop=sort(round(rand(1,nmut)*(popsize-1))+1);
for ii=1:nmut
    [r,c]=size(Pop{mPop(ii)});
    mcenter=ceil(rand(1,r-1))+1;
    oldcenter1=Pop{mPop(ii)}(mcenter(ii),1);
    oldcenter2=Pop{mPop(ii)}(mcenter(ii),2);
    newcenter1=oldcenter1+(rand-0.5)*2*mutrad;
    newcenter2=oldcenter2+(rand-0.5)*2*mutrad;
    if newcenter1<0
        newcenter1=0;
    end
    if newcenter1>1
        newcenter1=1;
    end
    if newcenter2<0
        newcenter2=0;
    end
    if newcenter2>1
        newcenter2=1;
    end
    Pop{mPop(ii)}(mcenter(ii),1)=newcenter1;
    Pop{mPop(ii)}(mcenter(ii),2)=newcenter2;
end
```

A.3.13 Hybrid Algorithm

```
%% Initializing the starting population in the first generation

% Stopping criteria
maxitLloyd=10;  %Max number of iterations

% Domain of VT
a=0;   % left endpoint in x-direction
b=1;   % right endpoint in x-direction
c=0;   % lower endpoint in y-direction
d=1;   % upper endpoint in y-direction

% Initialization parameters
numcenters = 2;      % number of centroids in each VT

PopLloyd=rand(numcenters,2);  % this loop randomly generates a
                              %   starting set of centers
                              %   (determined by numcenters)
for i=1:numcenters
    PopLloyd(i,1)=a+(b-a)*PopLloyd(i,1);
    PopLloyd(i,2)=c+(d-c)*PopLloyd(i,2);
end

% Set up grid of reference points to mimic continuous surface
res=1000;   % number of increments in each dimension
n=res+1;    % makes dimensions of dataset work

[X,Y]=meshgrid(a:(b-a)/res:b,c:(b-a)/res:d);  % implement grid
                                              %   over dimensions of %   VT's domain
Xnew=reshape(X,[n*n,1]);
Ynew=reshape(Y,[n*n,1]);

data=zeros(n*n,2);     % puts coordinates of grid points into
                       %   a single n^2 X 2 matrix
for i=1:n*n
    data(i,1)=Xnew(i,1);
    data(i,2)=Ynew(i,1);
end
```

```
% Density function
RHO=ones(n*n,1);       % an n^2 X 1 matrix that stores the
                       %   weights for each grid point
                       % this one is for a constant density function

% Associate grid points with density function
data=[data,RHO];        % grid points side-by-side with weights

% Start Lloyd's alg.
VLloyd=assignregion(data,PopLloyd);
eLloyd=energy(VLloyd,PopLloyd,res);


%% Main Loop
%  This is where Lloyd's method is implemented.

igaLloyd=0;
while igaLloyd<maxitLloyd

    igaLloyd=igaLloyd+1

    Z=masscentroid(VLloyd);
    PopLloyd=Z;
    VLloyd=assignregion(data,PopLloyd);
    eLloyd=energy(VLloyd,PopLloyd,res);

end % End of the while loop

%% Initializing the starting population in the first generation of GA

% Stopping criteria
maxit=20;  %Max number of iterations
mincost=-99999999;  %Minimum cost

% Initialization parameters
popsize = 20;        % population size (number of VTs created
                     %   per generation)

Pop{1}=PopLloyd;
```

```
for i=2:popsize      % centers in neighborhood of Lloyd's output
    for j=1:numcenters
        oldcenter1=PopLloyd(j,1);
        oldcenter2=PopLloyd(j,2);
        newcenter1=oldcenter1+(rand-0.5)*2*0.005;
        newcenter2=oldcenter2+(rand-0.5)*2*0.005;
        Pop{i}(j,1)=newcenter1;
        Pop{i}(j,2)=newcenter2;
    end
end

% Genetic algorithm parameters
mutrate=0.2;  %Mutation rate
popKept=0.5;  %Fraction of the population to keep
keep=floor(popKept*popsize);  %How many individuals are kept
if mod(popsize-keep,2)==1     %Makes number of matings work
    keep=keep-1;
end
M=ceil((popsize-keep)/2);
crossprob=round(rand(maxit,numcenters)); %0= x-coord, 1= y-coord
nmut=ceil((popsize-1)*2*numcenters*mutrate);  %Number of mutations

% Probability distribution for mate selection
probs=(keep:-1:1)/sum(1:keep);  %Probability is rank ordering

% Set up grid of reference points to mimic continuous surface
res=1000;   % number of increments in each dimension
n=res+1;    % makes dimensions of dataset work

[X,Y]=meshgrid(a:(b-a)/res:b,c:(b-a)/res:d);
Xnew=reshape(X,[n*n,1]);
Ynew=reshape(Y,[n*n,1]);

data=zeros(n*n,2);
for i=1:n*n
    data(i,1)=Xnew(i,1);
    data(i,2)=Ynew(i,1);
end

% Density function
```

```
RHO=ones(n*n,1);        % an n^2 X 1 matrix that stores the
                        %   weights for each grid point
                        % this one is for a constant density function

% Associate grid points with density function
data=[data,RHO];        % grid points side-by-side with weights


%% Sort initial population according to performance

e = zeros(popsize,1);

for i=1:popsize
    V{i}=assignregion(data,Pop{i});
    e(i,1)=energy(V{i},Pop{i},res);
end

[e,idx]=sort(e);  % Default sort is from small to large
for i=1:popsize
    temp{i}=Pop{idx(i)};
end
for i=1:popsize
    Pop{i}=temp{i};
end
minenergy(1)=min(e);  % Minimum energy, for plotting later
meanenergy(1)=mean(e);  %Mean cost for this population


%% Main Loop
%  This is where the genetic algorithm is implemented.

iga=0;
while iga<maxit

    iga=iga+1

    % Pair up and mate:
    ma=RandChooseN(probs,M);
    pa=RandChooseN(probs,M);
```

```
% Set up crossover and mutation:
idx2=keep+1:popsize;
beta=rand;

for i=1:M
    PopMa{i}=Pop{ma(i)};
    PopPa{i}=Pop{pa(i)};
end

for j=1:M
    for i=1:numcenters
        if crossprob(iga,i)==0  %Crossover the x-coordinate
            Pop{idx2(2*j-1)}(i,1)=(1-beta)*PopMa{j}(i,1)+beta*PopPa{j}(i,1);
            Pop{idx2(2*j-1)}(i,2)=PopMa{j}(i,2);
            Pop{idx2(2*j)}(i,1)=(1-beta)*PopPa{j}(i,1)+beta*PopMa{j}(i,1);
            Pop{idx2(2*j)}(i,2)=PopPa{j}(i,2);
        else                    %Crossover the y-coordinate
            Pop{idx2(2*j-1)}(i,1)=PopMa{j}(i,1);
            Pop{idx2(2*j-1)}(i,2)=(1-beta)*PopMa{j}(i,2)+beta*PopPa{j}(i,2);
            Pop{idx2(2*j)}(i,1)=PopPa{j}(i,1);
            Pop{idx2(2*j)}(i,2)=(1-beta)*PopPa{j}(i,2)+beta*PopMa{j}(i,2);
        end
    end
end

% Mutation
mPop=sort(round(rand(1,nmut)*(popsize-1))+1);
mcol=ceil(rand(1,nmut)*2);
for ii=1:nmut
    [r,c]=size(Pop{mPop(ii)});
    mcenter=round(rand(1,nmut)*(r-1))+1;
    Pop{mPop(ii)}(mcenter(ii),mcol(ii))=rand;
end


% New cost, set up for next iteration:
for i=1:popsize
    V{i}=assignregion(data,Pop{i});
    e(i,1)=energy(V{i},Pop{i},res);
end
```

```
    [e,idx]=sort(e);  % Default sort is from small to large
    for i=1:popsize
        temp{i}=Pop{idx(i)};
    end
    for i=1:popsize
        Pop{i}=temp{i};
    end
    minenergy(iga+1)=min(e);  % Minimum energy, for plotting later
    meanenergy(iga+1)=mean(e);  %Mean cost for this population

    % Stopping criteria
    if iga>maxit || e(1)<mincost
        break
    end

end % End of the while loop

%{
figure
iters=0:length(minenergy)-1;
plot(iters,minenergy,iters,meanenergy,'-');
xlabel('generation');ylabel('energy');
%}
```

References

[1] Jenna Carr. "An Introduction to Genetic Algorithms." Whitman College, 2014.

[2] Sophie De Arment. "2–Point Centroidal Voronoi Diagrams." Whitman College, 2015.

[3] Qiang Du, Maria Emelianenko, and Lili Ju. "Convergence of the Lloyd Algorithm for Computing Centroidal Voronoi Tessellations." *SIAM Journal of Numerical Analysis*. Vol. 44, No. 1, p. 104.

[4] Qiang Du, Vance Faber, and Max Gunzburger. "Centroidal Voronoi Tessellations: Applications and Algorithms." *SIAM Review*. Vol. 41, No. 4, pp. 637–676.

[5] Randy L. Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms.* Hoboken: Wiley, 2004.

[6] Doug Hundley. "Math Modeling Class Notes." Whitman College, 2014.

[7] James MacQueen. "Some methods for classification and analysis of multivariate observations." Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: *Statistics*, 281–297, University of California Press, Berkeley, Calif., 1967.

[8] Xiao Xiao. "Over-relaxation Lloyd method for computing centroidal Voronoi tessellations." (2010).