Senior Thesis in Mathematics

SimpleX:

Software tools for visualizing functions on simplicial complexes

Author:
Dmitriy Smirnov

Advisor:
Dr. Vin de Silva

Submitted to Pomona College in Partial Fulfillment
of the Degree of Bachelor of Arts

April 13, 2017

Abstract

I introduce a web-based tool, which allows the user to dynamical input a simplicial complex with a function defined on it and to visualize associated topological operations and structures. I go over the theory behind these ideas and demonstrate my implementation and visualization contributions.

Contents

Chapter 1

Introduction

Shape is a very human concept. We can easily say that something is "round" or "straight" or "wavy" even if it is not a perfect circle or line or sine curve. And when we recognize the shape of a data point cloud, we can make inferences about the underlying dataset. For example, we might conclude that it is the result of a periodic process or that there are certain clusters of interest. However, as the number of points and their dimensionality grow, human intuition begins to fail. Fortunately, there is an area of mathematics, topology, which precisely generalizes the notion of shape. Over the past couple decades, computer scientists have begun to realize that computational topology is fundamentally compatible with many of the goals of data analysis. Its techniques find structure in messy data in order to quantify ambiguous form, and, ultimately, to visualize and understand it.

In my thesis, the central object of study is a fundamental construction from computational topology—a simplicial complex together with a function. A simplicial complex is used to approximate a topological space, and it turns out that, when a function is defined on it, a lot of interesting structure arises. In Chapter 2, I examine simplicial complexes and the functions we can define on them. In Chapter 3, I consider Reeb graphs, which reveal information about certain types of such functions. And in Chapter 4, I look at integration using a topological calculus. Most importantly, I introduce SimpleX, a web-based software tool for interactively exploring the aforementioned structures and ideas. In each chapter, after reviewing the necessary theoretical background, I demonstrate how the theory materializes in a "hands-on" way through SimpleX.

Chapter 2

Simplicial Complexes

In topology, we are interested in computing properties of smooth topological spaces. These spaces, however, can be difficult to work with algorithmically. Therefore, we would like to develop a discrete combinatorial structure, which will serve as a "good-enough" approximation of a topological space.

2.1 Definition

The structure that we will study is the simplicial complex. We can define an abstract simplicial complex purely combinatorially. This will prove to be useful when dealing with these objects computationally.

Definition 2.1 (abstract simplicial complex)**.** An *abstract simplicial complex* $\mathcal{K}$ is a pair (V, S), where V is a finite set, whose elements we call *vertices*, and S is a set of nonempty finite subsets of V, whose elements we call *abstract simplices*, such that $\{v\} \in S$ for all v, and if $\sigma \in S$ and $\tau \subset \sigma$, then $\tau \in S$.

It is often more intuitive and useful to think of a geometric realization of a simplicial complex, as a subset of $\mathbb{R}^n$. We start by introducing the notion of a k-simplex, the generalization of a triangle in k dimensions. This will serve as the building block for constructing the complex.

Definition 2.2 (convex combination)**.** Let $v_0, \ldots, v_k$ be $k+1$ points in $\mathbb{R}^n$. A point $x = \sum_i \lambda_i v_i$ is a *convex combination* of the v_i if

- $\sum_i \lambda_i = 1$ and
- $\lambda_i \geq 0$ for all i.

Figure 2.1: A vertex, edge, triangle, and tetrahedron.

Definition 2.3 (affine independence)**.** The points $v_1, \dots, v_k$ are *affinely independent* if any two linear combinations $x = \sum_i \lambda_i v_i$ and $y = \sum_i \mu_i v_i$ are equivalent if and only if $\lambda_i = \mu_i$ for all i.

Definition 2.4 (k-simplex)**.** A *k-simplex* is the set of all convex combinations of $k+1$ affinely independent points $v_0, \dots, v_k$. We say that the *dimension* of a k-simplex σ, $\dim \sigma = k$, and $\{v_0, \dots, v_k\}$ is the *vertex set* of σ.

We give special names to the 0-, 1-, 2-, and 3-dimensional simplices—*vertices*, *edges*, *triangles*, and *tetrahedra*, respectively. Figure 2.1 shows examples of each.

Definition 2.5 (face)**.** Let σ be a k-simplex with vertex set $\{v_0, \dots, v_k\}$. A *face* τ of σ is the set of all convex combinations of any (nonzero) number of the v_i. We write $\tau \leq \sigma$.

Since a set of cardinality $k+1$ has 2^{k+1} subsets, including the empty set, a k-simplex has $2^{k+1} - 1$ faces. The only face of a vertex is the vertex itself, the faces of an edge are the edge and its two incident vertices, and so on.

Now, we are ready to define a geometric simplicial complex, a well-behaved collection of "glued-together" simplices.

Definition 2.6 (geometric simplicial complex)**.** A *geometric simplicial complex* $\mathcal{K}$ is a finite collection of simplices such that

- every face of a simplex in $\mathcal{K}$ is also in $\mathcal{K}$, and
- for any two simplices $\sigma_1, \sigma_2 \in \mathcal{K}$, if $\sigma_1 \cap \sigma_2 \neq \varnothing$, then $\sigma_1 \cap \sigma_2$ is a common face of σ_1 and σ_2.

We say that the *dimension* of a simplicial complex is the maximum dimension among all of its simplices.

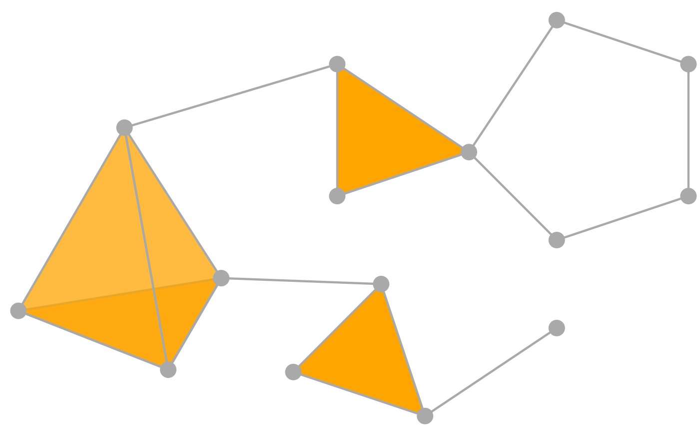


Figure 2.2: A simplicial complex

In other words, a simplicial complex is a collection of simplices that is closed under taking faces of simplices, and in which two simplices can only intersect at a face.

We can see how this corresponds to our previous combinatorial definition of an abstract simplicial complex. Given a geometric simplicial complex, we can consider a corresponding abstract simplicial complex with the same vertex set. Note that for a given abstract simplicial complex, there is an infinite number of possible geometric realizations.

Figure 2.2 shows a three-dimensional simplicial complex consisting of one tetrahedron, two triangles, twenty edges, and fourteen vertices.

Simplicial complexes allow us to create discrete, combinatorial approximations of smooth topological spaces, which facilitate concrete computations. We say that a geometric simplicial complex $\mathcal{K}$ is a *triangulation* of a topological space X if $\mathcal{K}$ and X are homeomorphic, i.e., topologically equivalent. Note that a triangulation is not unique—a topological space can admit infinitely many different triangulations.

More information about the theory behind complexes and triangulations can be found in [Munkres, 1984].

Figure 2.3 shows three topological spaces along a triangulation for each.

We define two more structures closely related to simplicial complexes, which will be useful later on.

Definition 2.7 (open k-simplex)**.** An *open k-simplex* is the set of all convex

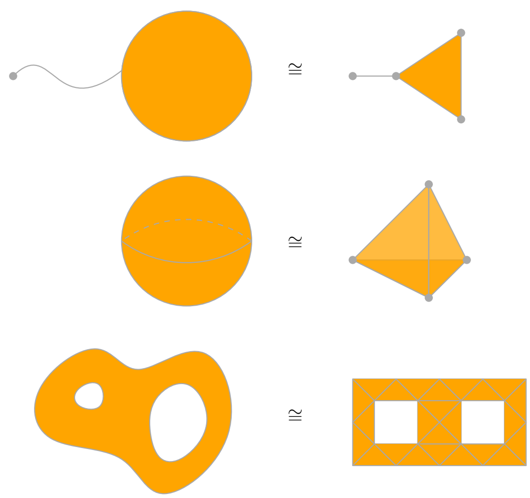


Figure 2.3: Three topological spaces (right) and their triangulations (left).

combinations of $k+1$ affinely independent points $v_0, \dots, v_k$ with strictly positive coefficients. In other words, an open k-simplex is a k-simplex without its boundary. Note that an open k-simplex σ is not an open set in $\mathbb{R}^n$, except when $\dim \sigma = n$.

We will sometimes refer to k-simplices as closed k-simplices to avoid ambiguity.

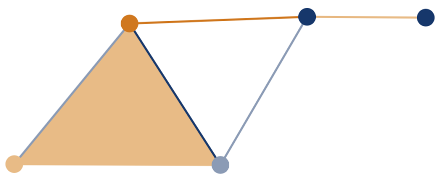
Definition 2.8 (subcomplex)**.** A *subcomplex* of a geometric simplicial complex $\mathcal{K}$ is the union of a subset of closed simplices of $\mathcal{K}$.

Definition 2.9 (definable subset of complexes)**.** Given a geometric simplicial complex $\mathcal{K}$, a *definable subset* of complexes of $\mathcal{K}$ is a union of open simplices of $\mathcal{K}$.

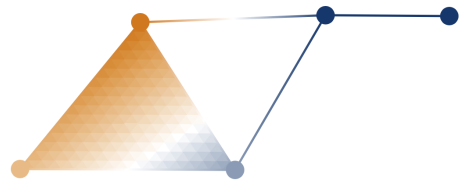
A subcomplex is itself a proper simplicial complex, but a definable subset is not necessarily a simplicial complex.

2.2 Maps

Like we define maps on arbitrary topological spaces, we would like to define maps in which the domain is a simplicial complex. In particular, we look



(a) A constructible function



(b) A piecewise-linear functions.

Figure 2.4: Two different functions defined on the same simplicial complex. Function value is represented by shade of simplex color.

at two types of such maps—constructible and piecewise linear functions, as shown in Figure 2.4.

2.2.1 Constructible functions

When defining our functions, we want to work with constructions that are "well-behaved." In particular, we wish to avoid pathological and counter-intuitive situations that may arise, especially when dealing with infinite objects. To do so, we restrict ourselves to what is known as "tame" topology by only considering an *o-minimal structure*, a sequence of subsets of $\mathbb{R}^n$ satisfying certain axioms. Each element in this sequence is a *definable* set. See [Van den Dries, 1998] for more on tame topology and o-minimality.

One common o-minimal structure is the *real semialgebraic sets*.

Definition 2.10 (real semialgebraic sets)**.** The *real semialgebraic sets* $\mathcal{SA}_n$ are the smallest class of subsets of $\mathbb{R}^n$ such that

- if $p \in \mathbb{R}[x_1, \ldots, x_n]$ is a polynomial with real coefficients, then $\{x \in \mathbb{R}^n \mid p(x) = 0\} \in \mathcal{SA}_n$ and $\{x \in \mathbb{R}^n \mid p(x) > 0\} \in \mathcal{SA}_n$ and
- if $A \in \mathcal{SA}_n$ and $B \in \mathcal{SA}_n$, then $A \cup B, A \cap B, \mathbb{R}^n \setminus A \in \mathcal{SA}_n$.

Note that the second condition makes $\mathcal{SA}_n$ a Boolean algebra.

We will be looking at simplicial complexes, which are unions of closed simplices. A closed simplex is semialgebraic, and, therefore, so is a simplicial complex.

Definition 2.11 (constructible function)**.** Given a topological space X, a function $\varphi : X \to \mathbb{Z}$ is said to be *constructible* if, for each $n \in \mathbb{Z}$, the set $\varphi^{-1}(n)$ is definable.

For $\mathcal{K}$ a geometric simplicial complex, one useful way of defining a constructible function $\varphi : \mathcal{K} \to \mathbb{Z}$ is by

$$\varphi = \sum_i C_i \cdot \mathbb{1}_{\sigma_i},$$

where $C_i \in \mathbb{Z}$ for all i, σ_i are the open simplices of $\mathcal{K}$, and $\mathbb{1}_{\sigma_i}$ is the indicator function on σ_i. Thus, we can define a constructible function on a simplicial complex by assigning an integer value to each of its simplices.

Note that a constructible function is generally not continuous—discontinuities can occur at simplex boundaries.

2.2.2 Piecewise linear functions

While constructible functions are useful in certain situations, they are not continuous. A piecewise linear function is a way to define a continuous map on a geometric simplicial complex. In general, a piecewise linear function is not constructible.

Definition 2.12 (barycentric coordinates)**.** Let $\mathcal{K}$ be a geometric simplicial complex with vertex set $\{v_0, \ldots v_n\}$, and let $x \in \mathcal{K}$. Let $\sigma \in \mathcal{K}$ be the simplex of smallest dimension such that $x \in \sigma$. By definition, x is the convex combination of vertices v_i, i.e., $x = \sum_i b_i \cdot v_i$.

We call the number b_i the *barycentric coordinates* of $x \in \mathcal{K}$.

Definition 2.13 (piece-wise linear function)**.** Let $\mathcal{K}$ be a geometric simplicial complex with vertex set $V = \{v_0, \ldots v_n\}$. Let $f : V \to \mathbb{R}$ be a real-valued function on the vertices of $\mathcal{K}$. We extend f to all of $\mathcal{K}$ linearly, i.e., by

$$x \mapsto \sum_i b_i \cdot f(v_i),$$

where b_i are the barycentric coordinates of x. Then, f is *piece-wise linear.*

2.3 SimpleX implementation

We would like the SimpleX interface to visualize a user-inputted simplicial complex together with a function—constructible or piecewise-linear—defined on it. For visualization purposes, we only support complexes of dimension no greater than two. Our user interface design choices follow from the definitions established above.

2.3.1 Visualizing complexes

Our input process must ensure that the resulting simplicial complex satisfies the two defining conditions.

- **The simplicial complex must be closed under taking faces of simplices, i.e., for any simplex in the complex, all of that simplex's faces must be contained in the complex as well.**

  We enforce this via a three-stage input process. In the first stage, the user is able to click anywhere on the canvas in order to place a vertex. In the second stage, edges are placed. Hovering the mouse between two existing vertices highlights a potential edge, which can be added to the complex. Only a potential edge may be added, and no new vertices may be placed at this stage. Finally, in the third stage, the user places triangles. Similar to stage two, hovering over a region bound by three edges highlights a potential triangle, which may be added.

- **Any two simplices in a simplicial complex may intersect only at a face.**

  This is also ensured by the incremental nature of the input process. After an edge is placed, all potential edges that intersect its interior are removed. Similarly, potential triangles are generated only in regions that do not contain any vertices in their interior.

Figures 2.5 and 2.6 show stages two and three, respectively.

2.3.2 Visualizing functions

We would like a visual way of representing constructible and piecewise-linear functions on a user-inputted simplicial complex. Since our primary interest is in the interplay between functions are complexes, we combine the input of the simplicial complex with that of a function and determine the color in which we render a simplex based on its function value.

We require the user to specify the function value of each simplex prior to placing it. A positive-valued simplex is rendered in orange, and a negative-valued simplex is blue. The shade of the color is proportional on the magnitude of the function value—the more negative the value, the darker the blue; the more positive, the darker the orange. The darkest shade always corresponds to the extrema (positive or negative) of the function so far. So, if a

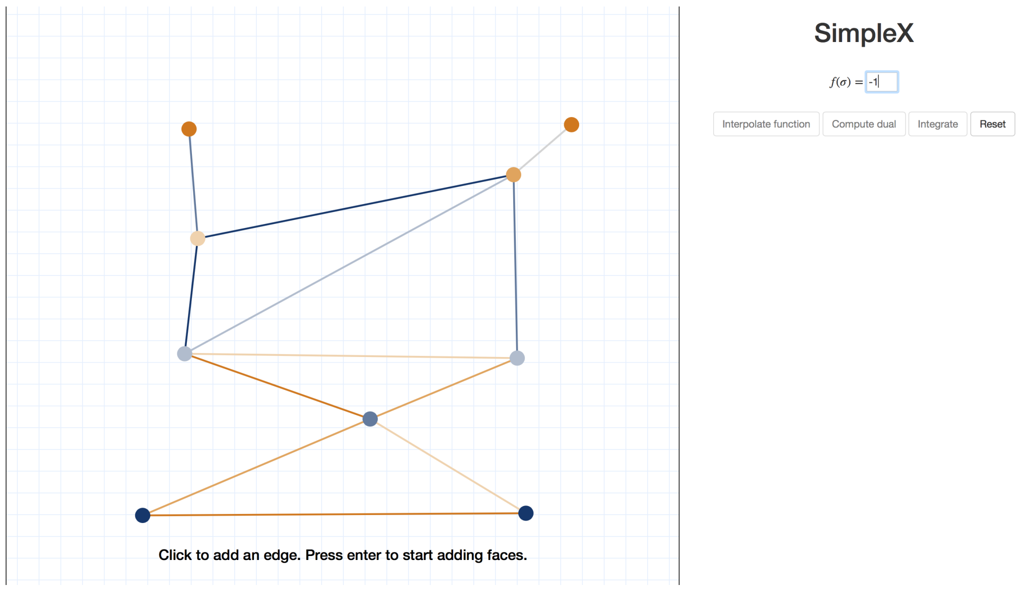


Figure 2.5: Stage two of simplicial complex input. Eleven placed edges and one potential edge are shown.

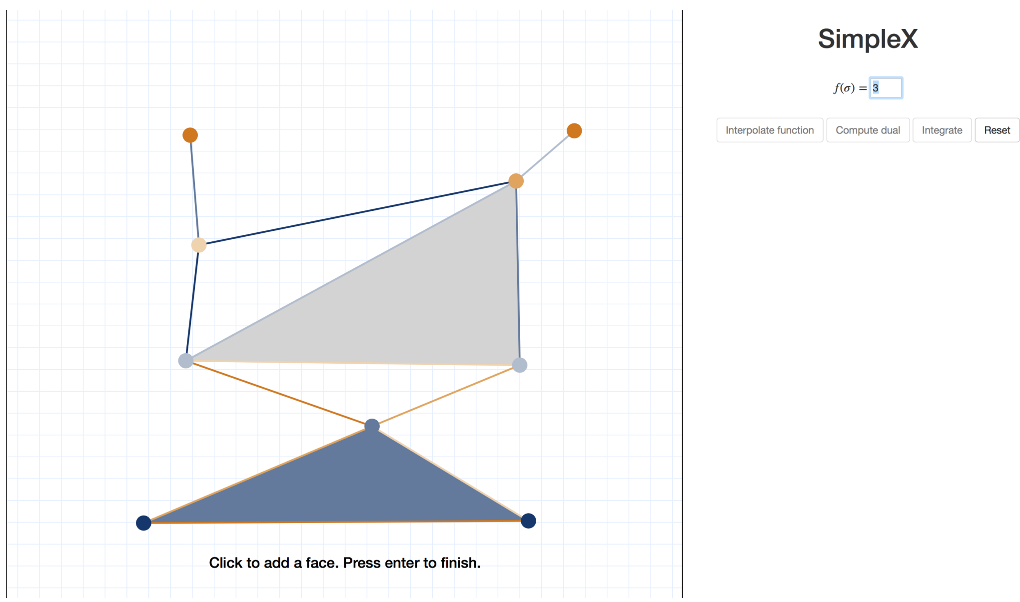


Figure 2.6: Stage three of inputting a simplicial complex.

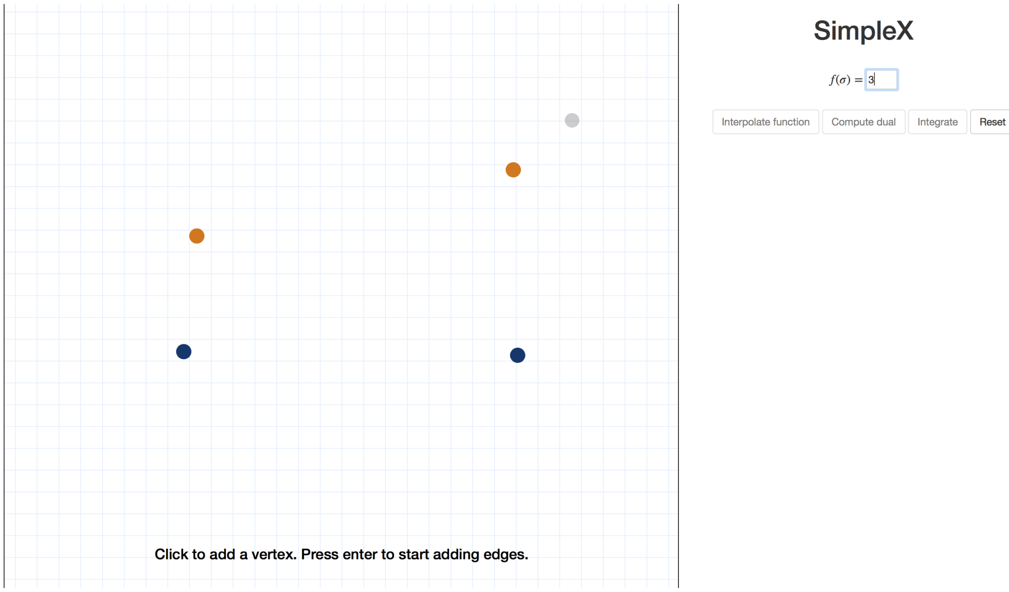
new simplex is placed with a value more negative than the current minimum or more positive than the maximum, the shades of the existing simplices get rescaled accordingly. Figure 2.7 demonstrates this reshading during stage one, and the process occurs analogously in stages two and three.

After the three stages, the structure of the simplicial complex is fixed, and the shadings of the simplices represent a constructible function defined on the complex. Hovering over each simplex displays its corresponding function value.

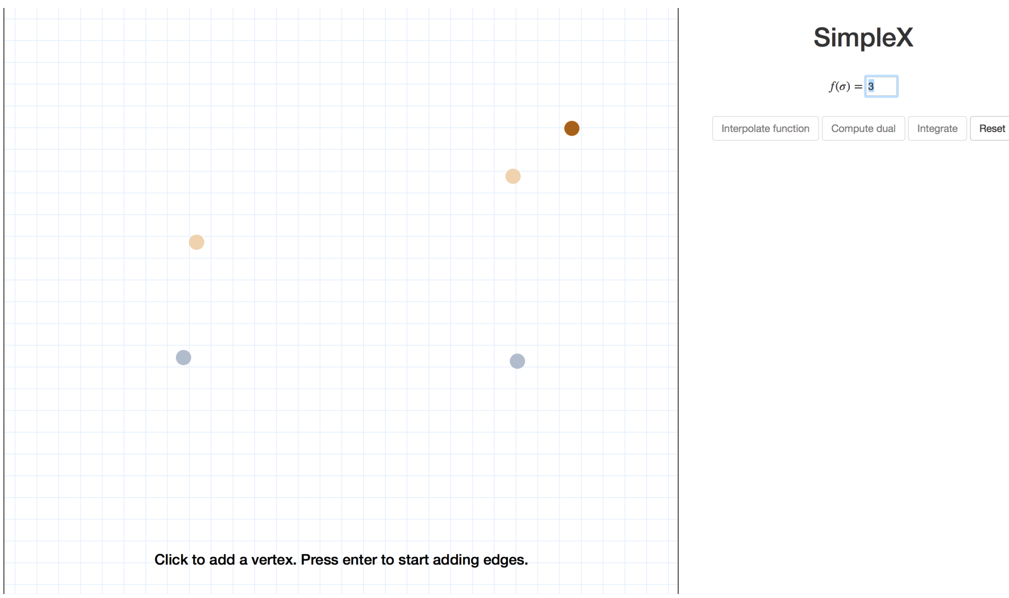
We can now choose to convert our constructible function to a piecewise-linear function by linearly interpolating the function based on vertex values. This redefines the function on the edges and triangles by computing the linear combination of the function values of their corresponding vertices. Accordingly, the edges and triangles get recolored in a gradient pattern. Hovering continues to display the precise function value.

Figure 2.8 shows an piecewise-linear function.

Note that that, although initial function values on the edges and triangles are forgotten when the function is linearly interpolated, we still require a value to be assigned to each simplex during the input process.



(a) Four vertices have been added to the simplicial complex. The bottom two (shaded blue) have function value -1, and the top two (orange) have function value 1.



(b) A new vertex with function value 3 has been added, causing the shades of the existing vertices to rescale.

Figure 2.7: The rescaling of simplex colors during vertex input.

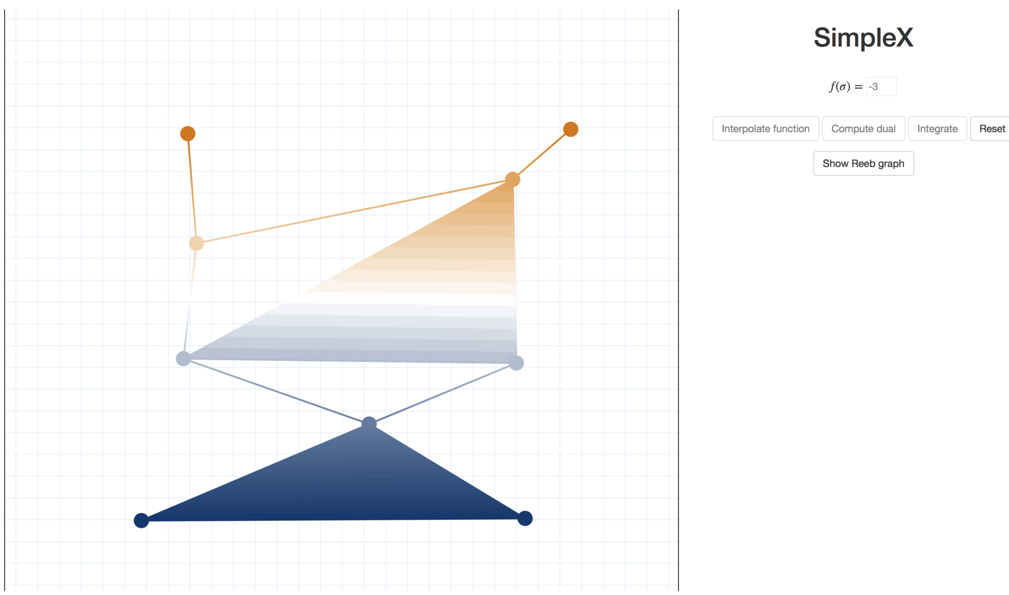


Figure 2.8: A piecewise-linear function on a simplicial complex.

Chapter 3

Euler calculus

We explore the integration theory of Euler calculus, a topological calculus with interesting application, introduced in [Schapira, 1991].

3.1 Definition

Before defining the Euler integral, we first introduce the Euler characteristic.

Given a simplicial complex, one question that we may ask is: how many connected components are there? We start by simply counting the number of vertices—if the complex contains no simplices of degree greater than zero, then, indeed, the number of vertices equals the number of components. However, as soon as we add an edge, the number of components decreases. Adding another edge again decreases the component count. But if we introduce a third edge and form a "hole," the number of components remains the same. Only when we add a triangle does that that hole get filled in. Thus, we arrive at the following formula:

$$\#\text{ components} + \#\text{ holes} = \#V - \#E + \#T,$$

where $\#V$ is the number of vertices, $\#E$ is the number of edges, and $\#T$ the number of triangles. Generalizing this count to simplicial complexes of arbitrary dimension motivates the Euler characteristic.

Definition 3.1 (Euler characteristic)**.** Let $\mathcal{K}$, and let $\mathcal{K}' = \{\sigma_i\}$ be a definable subset. Then, the *Euler characteristic* of $\mathcal{K}'$ is

$$\chi(\mathcal{K}') = \sum_i (-1)^{\dim \sigma_i}.$$

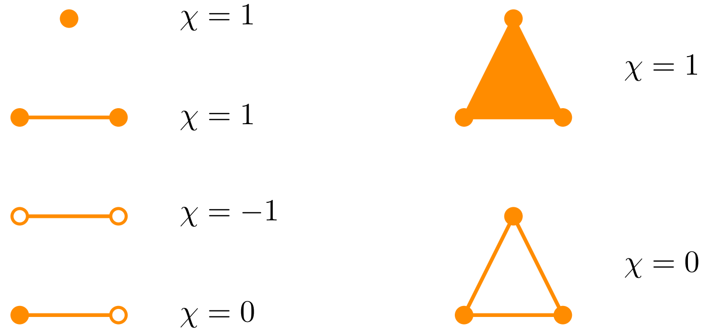


Figure 3.1: Examples of Euler characteristics.

Note that when $\mathcal{K}$ has dimension two or lower, $\chi(\mathcal{K}) = \#V - \#E + \#T$.

The Euler characteristic is a topological invariant, i.e., given a topological space, taking the Euler characteristic of any triangulation of the space will result in the same value (see [Hatcher, 2002] for more details and proof). So we can talk about the Euler characteristic of a topological space X, implicitly referring to the Euler characteristic of some triangulation of X, without it being ill-defined.

Fig 3.1 illustrates the values of the Euler characteristic for several definable subsets of simplices.

Proposition 3.2. *The Euler characteristic satisfies the property of finite additivity, i.e., for two simplicial complexes A and B,*

$$\chi(A \cup B) = \chi(A) + \chi(B) - \chi(A \cap B).$$

One may recall that finite additivity is a fundamental property of a measure.

Definition 3.3 (measure)**.** Let X be a set, and $\mathcal{B}$ a collection of subsets of X. Then, a *measure* on X is a function $\mu : \mathcal{B} \to \mathbb{R}$ that assigns to each subset a value, corresponding to its size.

Given a measure μ on X, we can integrate over subsets of X with respect to μ. Indeed, the common Lebesgue integral is computed with respect to the Lebesgue measure λ. On Euclidian space, λ corresponds to the standard notion of volume. So, for $f : \mathbb{R} \to \mathbb{R}$ with $f(x) \geq 0$ for all x,

$$\int_{-\infty}^{\infty} f(x)\ \mathrm{d}x = \int_{0}^{\infty} \ell(h)\ \mathrm{d}h = \int_{0}^{\infty} \lambda(f^{-1}(h, \infty))\ \mathrm{d}h,$$

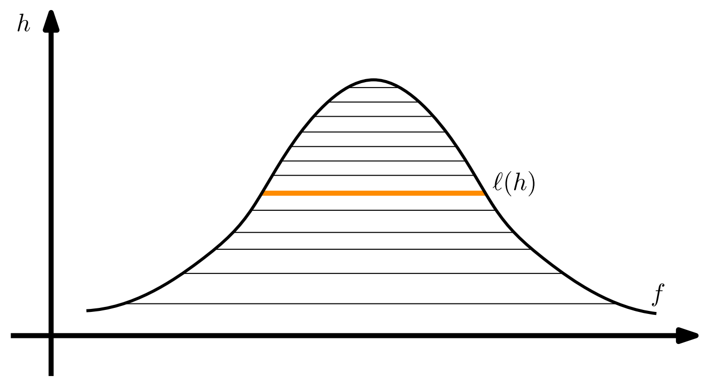


Figure 3.2: Integrating a function with respect to the Lebesgue measure.

where $f^{-1}(h, \infty)$ is the preimage of the open interval (h, ∞) under f. Here, $\ell(h)$ is the length of the interval on which f is defined at height h, as shown in Figure 3.2.

We can consider the Euler characteristic as a measure and use it for integration. Given $X \subseteq \mathbb{R}^2$ and a constructible function $h : X \to \mathbb{Z}$, we define the Euler integral in the natural way,

$$\int_X h \, \mathrm{d}\chi = \sum_{n=-\infty}^{\infty} \chi(h^{-1}(s)).$$

In practice, it is convenient to use a variant of the Fundamental Theorem of Calculus to compute Euler integrals.

Proposition 3.4. *Let $f : X \to \mathbb{Z}$ be a constructible function. Then,*

$$\int_X f \, \mathrm{d}\chi = \sum_{n=0}^{\infty} \left(\chi(f^{-1}(n, \infty)) - \chi(f^{-1}(-\infty, n)) \right),$$

where $f^{-1}(n, \infty)$ is the preimage of the open interval (n, ∞) under f, and $f^{-1}(-\infty, n)$ is analogous.

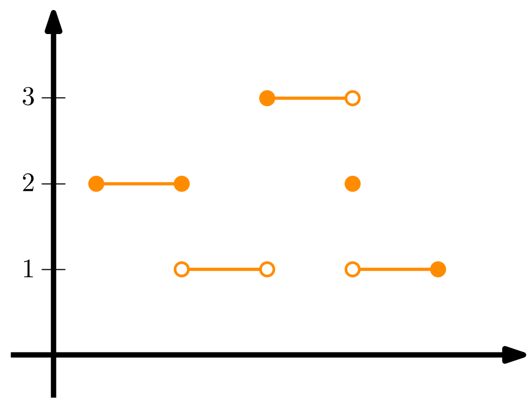


Figure 3.3: A constructible function $f : \mathbb{R} \to \mathbb{Z}$.

Proof.

$$
\begin{align}
h &= \sum_{n=-\infty}^{\infty} n \cdot \mathbb{1}_{h^{-1}(n)} \tag{3.1}\\
&= \sum_{n=0}^{\infty} n \cdot \left(\mathbb{1}_{h^{-1}[n,\infty)} - \mathbb{1}_{h^{-1}(n,\infty)}\right) + \sum_{n=0}^{-\infty} n \cdot \left(\mathbb{1}_{h^{-1}(-\infty,n]} - \mathbb{1}_{h^{-1}(-\infty,n)}\right) \tag{3.2}\\
&= \sum_{n=0}^{\infty} \left(\chi(h^{-1}(n,\infty)) - \chi(h^{-1}(-\infty,n))\right), \tag{3.3}
\end{align}
$$

where equality 3.3 holds by telescoping. $\square$

Example 3.5. Consider the constructible function $f : \mathbb{R} \to \mathbb{Z}$, as shown in Figure 3.3, where the function value corresponds to the height. Then,

$$
\begin{aligned}
\int_{\mathbb{R}} f \ \mathrm{d}\chi &= \chi(f^{-1}(0,\infty)) + \chi(f^{-1}(1,\infty)) + \chi(f^{-1}(2,\infty))\\
&= 1 + 2 + 0\\
&= 3.
\end{aligned}
$$

Example 3.6. Consider the constructible function $f : \mathbb{R}^2 \to \mathbb{Z}$, as shown in

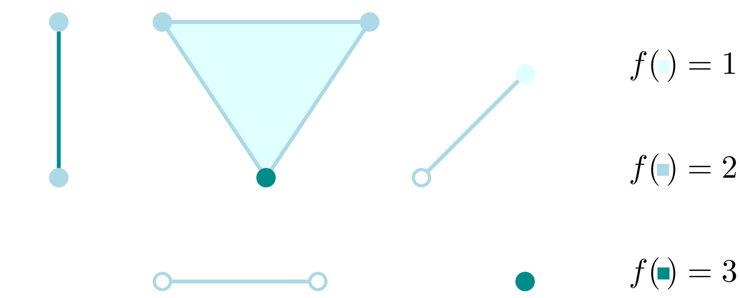


Figure 3.4: A constructible function $f : \mathbb{R}^2 \to \mathbb{Z}$.

Figure 3.4. Then,

$$\begin{aligned}\int_{\mathbb{R}^2} f \ \mathrm{d}\chi &= \chi(f^{-1}(0,\infty)) + \chi(f^{-1}(1,\infty)) + \chi(f^{-1}(2,\infty)) \\ &= (7-6+1) + (6-6) + (2-1) \\ &= 3.\end{aligned}$$

3.2 Operations

Defining integration with respect to the Euler characteristic provides a rich calculus, allowing us to compute various integral transforms with useful applications. Here, we look at two such transforms—convolution and duality. Consult [Curry et al., 2012] for more context and details.

3.2.1 Convolution and duality

The first operation that we look at, convolution, is closely related to the Minkowski sum from geometry.

Definition 3.7 (Minkowski sum)**.** Let $A, B \subset \mathbb{R}^n$. Then, the Minkowski sum of $A + B$ is the set formed by adding each vector in A to each vector in B, i.e.,

$$A \oplus B = \{a + b \mid a \in A, b \in B\}.$$

Figure 3.5 shows an example of a Minkowski sum in the plane—the entire orange region on the right is the Minkowski sum of the red and blue regions on the left.

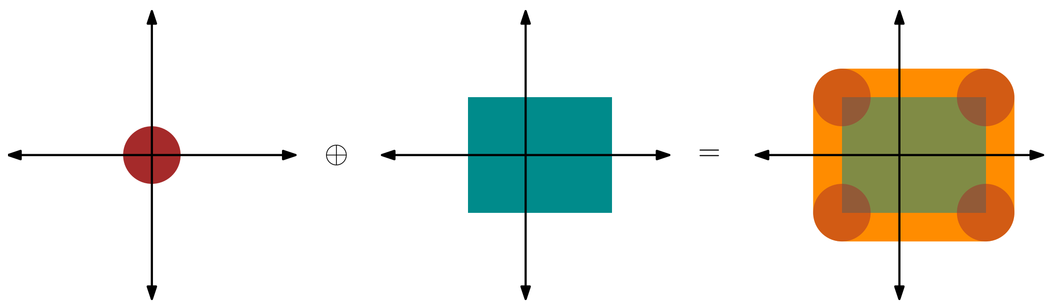


Figure 3.5: The Minkowski sum of two subsets of $\mathbb{R}^2$.

We now define the *convolution* operation with respect to the Euler characteristic on two constructible functions.

Definition 3.8 (convolution)**.** Given two constructible functions $f, g : V \to \mathbb{Z}$ defined on a real vector space, we define the convolution operator $*$ by

$$(f * g)(x) = \int_V f(t)g(x - t)\ \mathrm{d}\chi(t).$$

It turns out that the convolution of two indicator functions is equal to the indicator function on the Minkowski sum of their respective regions, i.e., for $A, B \subset \mathbb{R}^n$ such that A and B are convex,

$$\mathbb{1}_A * \mathbb{1}_B = \mathbb{1}_{A \oplus B}.$$

Proof. Since the regions over which f and g are nonzero are convex, their Euler characteristic is equal to one. Therefore, for any x, the above integral is equal to one if $x = a + b$, where $a \in A$ and $b \in B$, and zero otherwise. $\square$

The other integral transform we consider is *duality*.

Definition 3.9 (dual)**.** Let $f : X \to \mathbb{Z}$ be a constructible function and $x_0 \in X$. Let $\varepsilon > 0$ be small enough such that the value $\int_X f \cdot \mathbb{1}_{B(x,\varepsilon)}\ \mathrm{d}\chi$, where $B(x, \varepsilon)$ denotes the ball of radius ε around X, depends only on the function f. Define the dual of f by

$$(\mathcal{D}h)(x_0) = \int_X f \cdot \mathbb{1}_{B(x,\varepsilon)}\ \mathrm{d}\chi.$$

When the domain of a constructible f is a simplicial complex, computing the dual becomes combinatorial and procedural. Indeed, the value of $\mathcal{D}f$ on a simplex σ depends only on the cofaces of σ, i.e., the higher-dimension simplices that have σ as a face as well as σ itself. Specifically, Algorithm 1 describes the procedure `ComputeDual`$(\mathcal{K}, f)$ for computing the dual of a constructible function $f : \mathcal{K} \to \mathbb{Z}$ on a simplicial complex

Algorithm 1: `ComputeDual`$(\mathcal{K},\ f)$

foreach *simplex* $\sigma \in \mathcal{K}$ **do**
    $val \leftarrow 0$
    foreach τ *such that* $\sigma \leq \tau$ **do**
        $val \leftarrow val + (-1)^{\dim \tau} \cdot f(\tau)$
    $f(\sigma) \leftarrow val$
return f

The dual can be used to define a deconvolution operator, so we can use the dual to "undo" a Minkowski sum of two subsets.

Proposition 3.10. *For a non-empty convex closed subset of a vector space* $A \subset V$*,*

$$\mathbb{1}_A * \mathcal{D}\mathbb{1}_{-A} = \delta_0,$$

where $-A$ *is the reflection of* A *about the origin, and* δ_0 *is the indicator function on the origin. In other words,* $\mathcal{D}\mathbb{1}_{-A}$ *is the convolution inverse of* $\mathbb{1}_A$.

The proof follows from sheaf theory (see [Schapira, 1991]) and is outside of the scope of this thesis.

3.3 Application to sensor networks

Another use of Euler integration is in computing information about sensor networks, introduced in [Baryshnikov and Ghrist, 2009].

Suppose we have a *sensor network*, i.e., a finite set of *targets* in Euclidian space $\{\mathcal{O}_1, \ldots, \mathcal{O}_n\} \subset \mathbb{R}^2$, where each target $\mathcal{O}_i$ is a point in the plane. Furthermore, suppose there is a *sensor* at every point $x \in \mathbb{R}^2$, which counts how many of the n targets it can detect. The count function $f : \mathbb{R}^2 \to \mathbb{Z}^{\geq 0}$

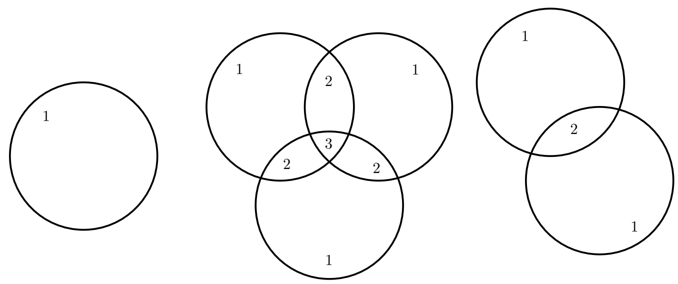


Figure 3.6: Target supports of a sensor network where each sensor can detect targets a fixed radius away.

returns the target count $f(x)$ for the sensor at x. Our goal is to determine n, the total number of targets.

Suppose, furthermore, that each sensor can sense exactly the targets that are a fixed radius r away from it, as in Figure 3.6. Then, we have that:

Proposition 3.11.

$$n = \frac{1}{\pi r^2} \int_{\mathbb{R}^2} f(x) \, \mathrm{d}x$$

Proof. This follows from Lebesgue integration.

$$\int_{\mathbb{R}^2} fx(x) \, \mathrm{d}x = \int_{\mathbb{R}^2} \sum_i \mathbb{1}_{U_i} \, \mathrm{d}x = \sum_i \int_{\mathbb{R}^2} \mathbb{1}_{U_i} \, \mathrm{d}x = \#\{\mathcal{O}_i\} \cdot \pi r^2.$$

□

Definition 3.12 (target support)**.** For each target $\mathcal{O}_i$ $(1 \leq i \leq n)$, we define the the *target support* to be

$$U_i = \{x \in X \mid \text{the sensor at } x \text{ detects } \mathcal{O}_i\}.$$

What if each sensor doesn't detect a perfect circle around itself? As long every target support is a region of some fixed area, the argument above holds—even if the target supports are of different shapes (see Figure 3.7).

But what if we the only information we have about the target supports is the Euler characteristic? We want some way to assign the same value to each region, regardless of its actual area. This can be accomplished using Euler calculus.

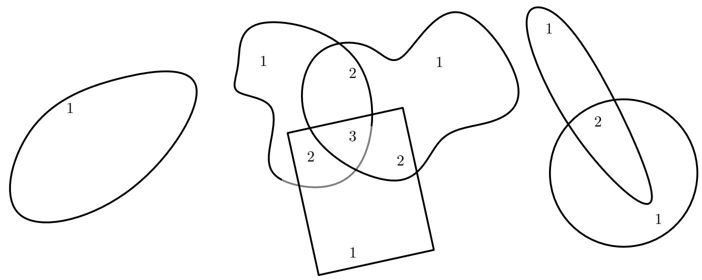


Figure 3.7: Target supports of a sensor network where each target is detected by a fixed area of sensors.

Proposition 3.13. *If $\chi(U_i) = N \neq 0$ for all i, where N is some constant, then*

$$n = \frac{1}{N}\int_X f \ \mathrm{d}\chi.$$

The proof is analogous to that of Proposition 3.11.

Example 3.14. Consider the f function represented in Figure 3.8, and suppose we know that each target support has Euler characteristic one. Then, by Proposition 3.13, the number of targets is

$$\begin{aligned} n &= \frac{1}{N}\int_X f \ \mathrm{d}\chi \\ &= \sum_{n=0}^{\infty} \big(\chi(f^{-1}(n,\infty)) - \chi(f^{-1}(-\infty,n))\big) \\ &= \sum_{n=0}^{\infty} \chi(f^{-1}(n,\infty) \\ &= 0+3+1+1 \\ &= 5. \end{aligned}$$

Indeed, as shown in Figure 3.9, there are five targets in the sensor network.

We note that the value n in Proposition 3.13 is not well defined when $N = 0$. This is not a shortcoming of the method but rather a feature. Indeed, when target supports have Euler characteristic zero, it is impossible to unambiguously compute the number of targets given the count function.

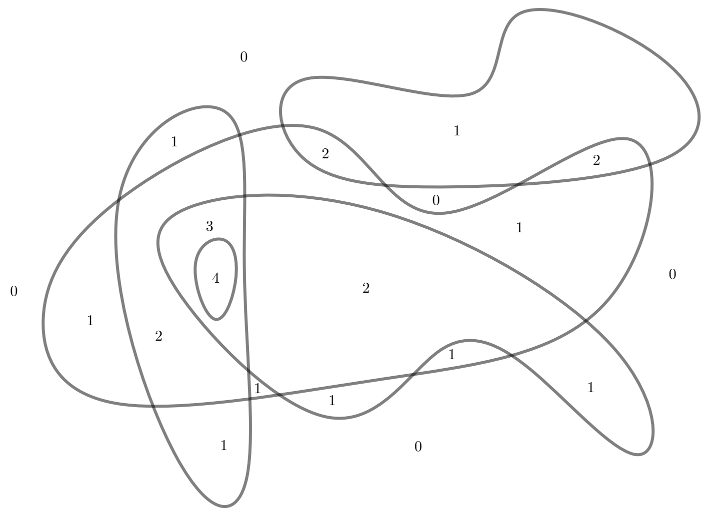


Figure 3.8: The count function values of a sensor network.

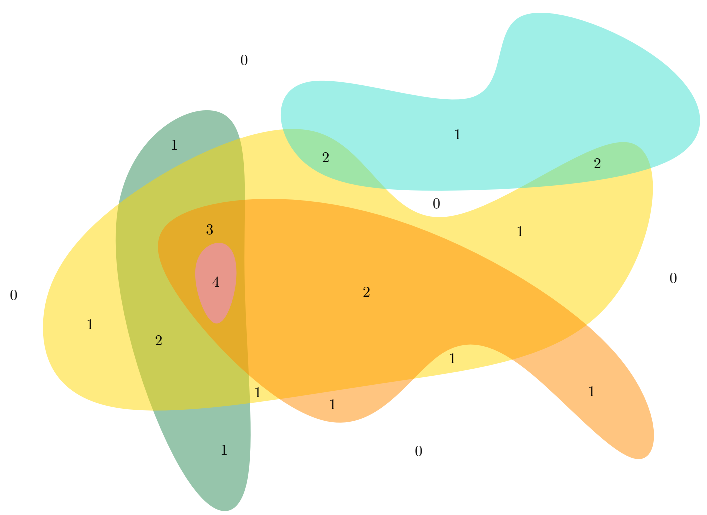
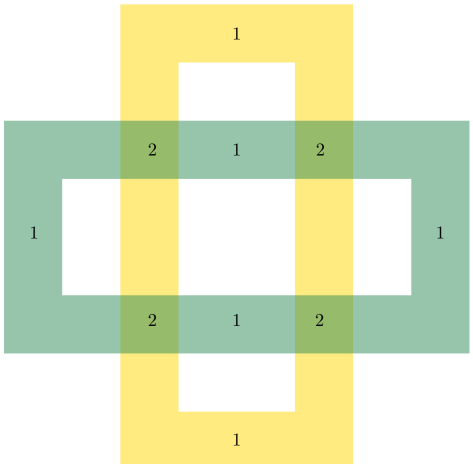
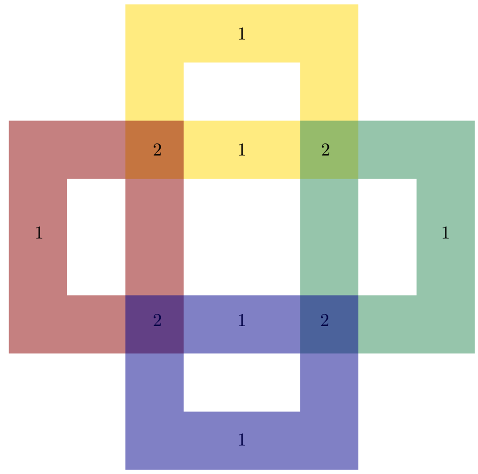


Figure 3.9: The five target supports in the sensor network.



(a) Two target supports.



(b) Four target supports.

Figure 3.10: An ambiguous case for $N = 0$.

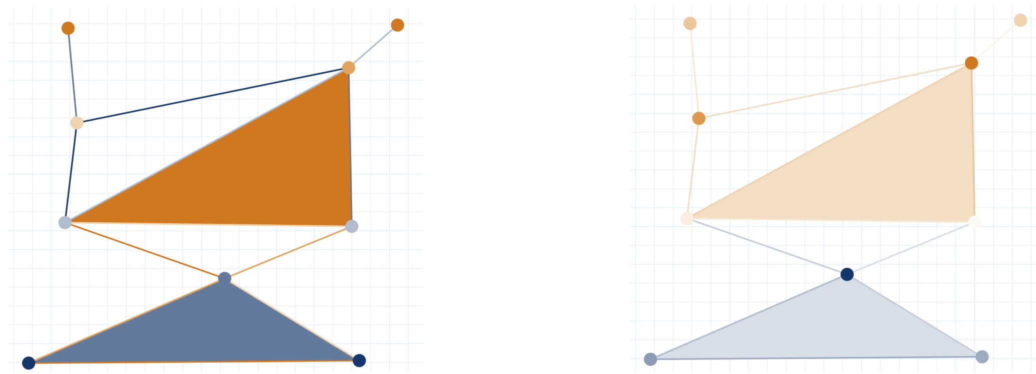
As an example, consider Figure 3.10. Both the left and right image show the same count function, but, on the left, two target supports are displayed, whereas, on the right, there are four target supports.

3.4 SimpleX implementation

Once a simplicial complex $\mathcal{K}$ with a constructible function f defined on it has been input into the SimpleX interface, we want to visualize the computation of the dual $\mathcal{D}f$ as well as of the the Euler integral of f over any constructible subset of $\mathcal{K}$.

3.4.1 Visualizing duality

We allow the user to toggle between the initial constructible function f and its dual $\mathcal{D}f$. Note that both f and $\mathcal{D}f$ are defined on the same domain, $\mathcal{K}$, and so only the shading of the simplicial complex may change, not its structure. Since the extrema of f and those of its dual may not be the same, a simplex with a certain shade in the visualization of f may have a different function value thanfc a simplex with the same shade in the $\mathcal{D}f$ visualization. In order to clarify the actual function values, the user can hover over each simplex. Figure 3.11 shows the initial constructible function and its dual.



(a) Initial function

(b) Dual

Figure 3.11: A constructible function defined on a simplicial complex and its dual, as displayed in SimpleX.

By experimenting we can notice empirically that duality is an involution—computing the dual twice yields the initial function. This is actually a true property of duality.

3.4.2 Visualizing Euler integration

We allow the user to build up the domain of integration by adding simplices of $\mathcal{K}$ incrementally. Upon entering integration mode, the entire simplicial complex is rendered in grayscale, to signify that the domain is initially empty. Accordingly, the integral is shown to equal zero. The user can now choose to augment the domain, one simplex at a time. Clicking on a simplex re-renders it in its original blue or orange shade, and the value of the Euler integral immediately updates over the new domain. Similarly, to remove a simplex from the integration domain, the user can click on it again. In Figure 3.12, we see the Euler integral over three vertices, two edges, and one triangle.

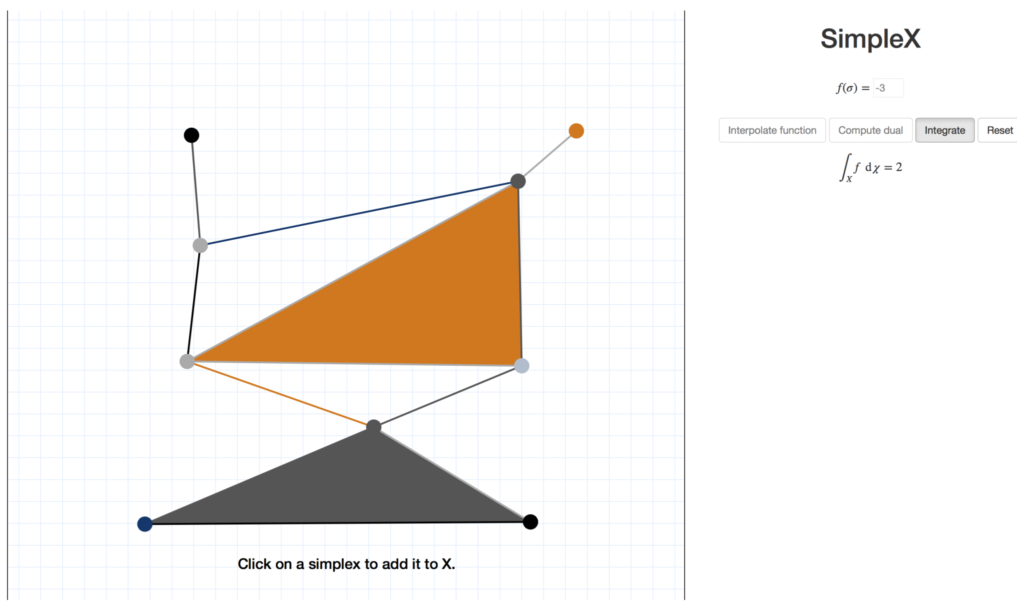


Figure 3.12: Computing the Euler integral over a constructible subset of a simplicial complex.

Chapter 4

Reeb graphs

Suppose we have a continuous function $f : X \to \mathbb{R}$, where X is a topological space. We are interested in its behavior as its value gradually changes. In particular, we would like to see how the connected components of $f^{-1}(c)$ change as c varies. This information is contained in a structure known as the Reeb graph.

4.1 Definition

We first formally define the Reeb graph.

Definition 4.1 ($\mathbb{R}$-space)**.** If X is a compact topological space and $f : X \to \mathbb{R}$ a continuous function, then the pair (X, f) is an $\mathbb{R}$-*space*.

Definition 4.2 (Reeb graph)**.** Let $\tilde{X}$ be the quotient space of X under the equivalence relation $x \sim y$ if and only if $f(x) = f(y) = c$ for some $c \in \mathbb{R}$, and there is a path from x to y in $f^{-1}(c)$. Let $\tilde{f}$ be the quotient map. Then, the *Reeb graph* of an $\mathbb{R}$-space (X, f) is the $\mathbb{R}$-space $(\tilde{X}, \tilde{f})$.

In other words, in the Reeb graph, for every $c \in \mathbb{R}$, we contract each connected component of $f^{-1}(c)$ into a single point, i.e., we consider two points in X equivalent if they have the same function value and are in the same component.

Figure 4.1 provides an example of a Reeb graph.

We have noted that a piecewise-linear function defined on a geometric simplicial complex $\mathcal{K}$ is continuous. Therefore, we restrict ourselves to geometric simplicial complexes with piecewise-linear functions.

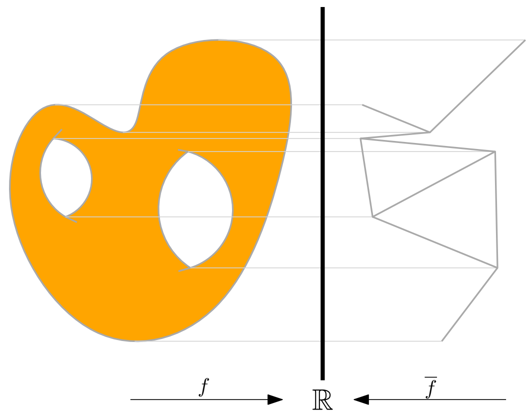


Figure 4.1: A function and its Reeb graph.

As we sweep across the Reeb graph, we notice that Reeb nodes occur where connected components of the simplicial complex are created, merge with others, split, or get destroyed.

Definition 4.3 (Reeb-critical value)**.** A value $n \in \mathbb{R}$ is *Reeb-critical* it corresponds to the function value of a node in the Reeb graph, i.e., if the number of connected components of $f(n+\varepsilon)$ is different from that of $f(n-\varepsilon)$ for a small $\varepsilon > 0$.

The following observation, which follows from the definition of a simplicial complex, is very useful for computational purposes.

Proposition 4.4. *A Reeb-critical value can only occur at a vertex of the simplicial complex.*

In the next section, we discuss an algorithm for computing Reeb graphs. Due to the nature of SimpleX, we only concern ourselves with Reeb graphs of simplicial complexes of dimension no greater than two. But it actually turns out that this restriction does not limit the algorithm.

Proposition 4.5. *The Reeb graph of a piecewise-linear function $f : \mathcal{K} \to \mathbb{R}$ depends only on the restriction of f to the simplices of $\mathcal{K}$ of dimension two and lower.*

See [Parsa, 2014] for a proof.

4.2 Computation

By tracking the connectivity of level sets, the Reeb graph is often used to quantify the perturbation necessary to eliminate a connected component of a space in a variety of applications. In [Kanongchaiyos and Shinagawa, 2000], the authors used Reeb graphs to model multimedia information and create animations. In [Biasotti et al., 2000], surface compression and reconstruction was performed using Reeb graphs for graphics rendering. In [Xiao et al., 2003], Reeb graphs were used to segment a human body into functional parts.

There have been many contributions of algorithms for computing the Reeb graph of a topological space. A runtime of $O(n \log n (\log \log n)^3)$ was achieved in [Doraiswamy and Natarajan, 2009]. Other variations, such as parallel and online computation, have also been considered.

Here, we are interested in computing the Reeb graph of a simplicial complex (with a piecewise-linear function defined on it). In particular, we look at the algorithm developed by Doraiswamy and Natarajan. We notice that in a simplicial complex, critical values may only occur at vertices. Thus, we sweep f from $-\infty$ to ∞, maintaining a graph of the preimage $f^{-1}(f(v_i))$ at each value. Notice that the preimage is indeed a graph—its nodes correspond to edges of the simplicial complex, and its edges correspond to triangles. The preimage graph changes if and only if we pass a Reeb-critical value. Thus, if we determine that a function value is critical, we add a new node to the Reeb graph.

Algorithm 2 describes this sweep procedure. This algorithm is a slight extension of Doraiswamy and Natarajan so as to handle Reeb graphs of functions where two vertices might map to the same value—the original algorithm assumes a general position in which all vertex values are distinct.

`GetLowerComps`(u, P, K) returns a list of nodes of the the preimage graph P, each representing an edge ending at u in K. `GetUpperComps`(u, P, K) functions analogously.

`UpdatePreimage`(P, u, K) updates the preimage graph P to reflect the change of passing over vertex u. In other words, it takes the preimage graph from that right before $f(u)$ to that right after.

In Algorithm 2, when we pass over a vertex u, we notice that all of the lower components merge at u, and u then splits into the upper components. If there is just a single lower and a single upper component, then $f(u)$ is not Reeb-critical. Otherwise, we add a new node ν to the Reeb graph, associate it with each of the upper components, and link it to the Reeb graph nodes

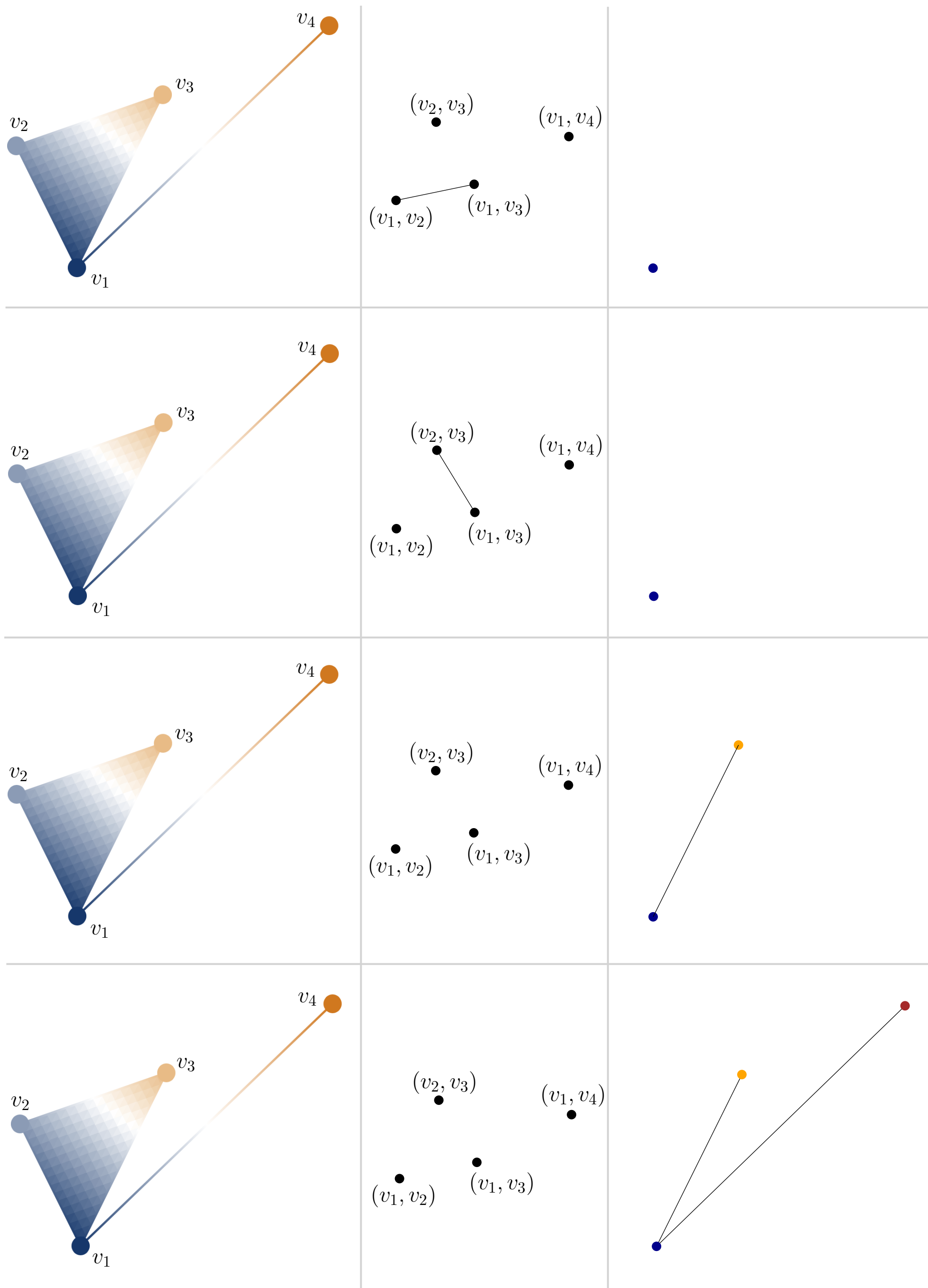


Figure 4.2: A simplicial complex (left), the preimage graph (middle), and the Reeb graph (right) over four steps of Algorithm 2, from top to bottom.

corresponding to the lower components.

Figure shows the evolution of the preimage graph and the Reeb graph over four steps of the algorithm.

Algorithm 2: `ComputeReeb`(K)

```
Sort vertices V by f value
Initialize graph P with one node for each edge (u, v) ∈ E where
  f(u) ≠ f(v)
foreach vertex u ∈ V do
    Iv ← {v | (u, v) ∈ E and f(v) = f(u)}
    Lc ← ⋃_{v∈Iv} GetLowerComps(v, P, K)
    P ← UpdatePreimage(P, u, K)
    Uc ← ⋃_{v∈Iv} GetUpperComps(v, P, K)
    if ¬(#Lc = #Lc = 1) then
        Add node ν to R
        Denote ν by ν_c for each c ∈ Uc
        Add edge (ν, ν_c) to R for each c ∈ Lc
return R
```

Algorithm 3: `GetLowerComps`(u, P, K)

```
Lc ← ∅
foreach v ∈ V such that (u, v) ∈ E with f(v) < f(u) do
    Let c be the component of (u, v) in P
    Lc ← Lc ∪ {c}
return Lc
```

When using appropriate date structures, this algorithm runs in $O(m \log m)$ time, where m is the size of the simplicial complex.

4.3 SimpleX implementation

Having input a simplicial complex and linearly interpolated a function based on the vertex values, we can compute the corresponding Reeb graph. The

Algorithm 4: `GetUpperComps`(u, P, K)

```
Uc ← ∅
foreach v ∈ V such that (u, v) ∈ E with f(u) < f(v) do
    Let c be the component of (u, v) in P
    Uc ← Uc ∪ {c}
return Uc
```

Algorithm 5: `UpdatePreimage`(P, u, K)

```
foreach a, b, c ∈ V such that (a, b, c) ∈ T with f(a) ≤ f(b) ≤ f(c) and
  u ∈ {a, b, c} do
    if u = a then
        Add edge ((a, b), (a, c)) to P
    else if a = b or b = c then
        if u = c and triangle (a, b, c) is marked then
            Remove edge ((a, b), (a, c)) from P
        else
            Mark triangle (a, b, c)
    else
        if u = b then
            Add edge ((a, c), (b, c)) to P
            Remove edge ((a, b), (a, c)) from P
        else
            Remove edge ((a, c), (b, c)) from P
return P
```

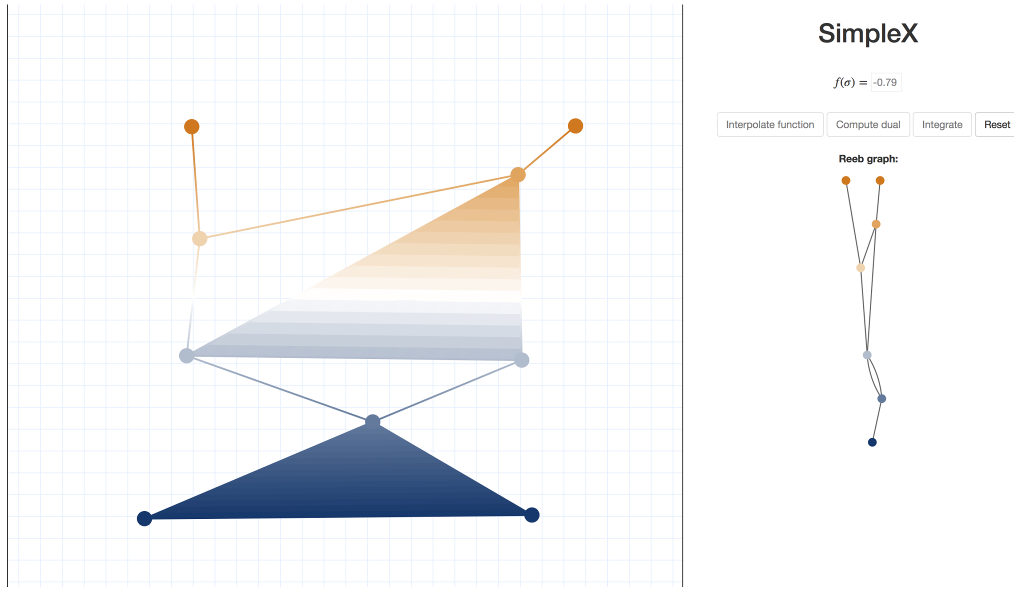


Figure 4.3: A simplicial complex and its Reeb graph

vertices of the Reeb graph are shaded according to the corresponding function value, like in the simplicial complex. The vertices' vertical positions are fixed—smaller functional values are displayed lower—but the horizontal position of a vertex can be adjusted by clicking and dragging. This can be useful in understanding the graph topology, especially when the Reeb graph is dense. As with the simplicial complex, hovering over a vertex of the Reeb graph displays its function value. Figure 4.3 shows a simplicial complex and its Reeb graph.

Chapter 5

Technical details

SimpleX is written in JavaScript with the help of the jQuery library. Visualization of simplicial complexes is implemented using the Two.js library, and D3.js is used to display Reeb graphs. Graphlib is used to maintain the underlying data structure during Reeb graph computation. User interface elements are styled and scaffolded with Twitter Bootstrap.

The interactive application is hosted at `https://dmsm.github.io/simplex`, and all code can be found on the author's Github page at `https://github.com/dmsm/simplex`. The application can be run locally offline in the browser.

Chapter 6

Conclusion and further work

The simplicial complex is a simple yet powerful object, which it serves as a basis for various useful tools, frameworks, and structures in computational topology.

The SimpleX tool provides a novel way of exploring abstract topological structures and ideas. By empowering the user to interact with and visualize simplicial complexes and functions, SimpleX serves a purpose, which is twofold. While on one hand, it can be used a teaching and learning aid, offering a tangible way of internalizing abstract concepts, on the other, it also adds a new dimension to mathematical exploration, perhaps serving an inspiration for new theoretical intuitions or discoveries.

Because the simplicial complex is so fundamental and well-studied, there are many directions for extensions of SimpleX. For instance, other structures characterizing the topology of a piecewise-linear function (such as merge trees) could be computed. Support for simplicial maps between complexes could be added. More complex Euler integral transforms could be implemented. In addition, interactive visualizations for persistent homology would fit will into the SimpleX framework.

Chapter 7

Bibliography

[Baryshnikov and Ghrist, 2009] Baryshnikov, Y. and Ghrist, R. (2009). Target enumeration via euler characteristic integrals. *SIAM Journal on Applied Mathematics*, 70(3):825–844.

[Biasotti et al., 2000] Biasotti, S., Mortara, M., and Spagnuolo, M. (2000). Surface compression and reconstruction using reeb graphs and shape analysis. In *Spring Conference on Computer Graphics*, pages 174–185.

[Curry et al., 2012] Curry, J., Ghrist, R., and Robinson, M. (2012). Euler calculus with applications to signals and sensing. In *Proceedings of Symposia in Applied Mathematics*, volume 70, pages 75–146.

[Doraiswamy and Natarajan, 2009] Doraiswamy, H. and Natarajan, V. (2009). Efficient algorithms for computing reeb graphs. *Computational Geometry*, 42(6-7):606–616.

[Hatcher, 2002] Hatcher, A. (2002). Algebraic topology. 2002. *Cambridge UP, Cambridge*, 606(9).

[Kanongchaiyos and Shinagawa, 2000] Kanongchaiyos, P. and Shinagawa, Y. (2000). Articulated reeb graphs for interactive skeleton animation. In *Multimedia Modeling: Modeling Multimedia Information and Systems*, pages 451–467. World Scientific.

[Munkres, 1984] Munkres, J. R. (1984). *Elements of algebraic topology*, volume 2. Addison-Wesley Menlo Park.

[Parsa, 2014] Parsa, S. (2014). *Algorithms for the Reeb Graph and Related Concepts*. PhD thesis, Duke University.

[Schapira, 1991] Schapira, P. (1991). Operations on constructible functions. *Journal of pure and applied algebra*, 72(1):83–93.

[Van den Dries, 1998] Van den Dries, L. (1998). *Tame topology and o-minimal structures*, volume 248. Cambridge university press.

[Xiao et al., 2003] Xiao, Y., Siebert, P., and Werghi, N. (2003). A discrete reeb graph approach for the segmentation of human body scans. In *3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings. Fourth International Conference on*, pages 378–385. IEEE.

Appendix A

JavaScript code

```
const RADIUS = 10;
const LINEWIDTH = RADIUS/4;
const GRAY = "#D3D3D3";
const RESOLUTION = 4;
const INT_TEX = "\\int_X f\\ \\operatorname{d}\\chi = "
const POS_COLOR = {
    r : 210,
    g : 120,
    b : 5
};
const NEG_COLOR = {
    r : 20,
    g : 54,
    b : 109
};

$(() => {
    var QUEUE = MathJax.Hub.queue;
    var math = null;
    QUEUE.Push(() => { math = MathJax.Hub.getAllJax("integral")[0]; });

    var canvas = document.getElementById("canvas");
    var two = new Two({
        width: $(canvas).width(),
        height: $(window).height(),
    }).appendTo(canvas);
    var $canvas = $("svg"),
        $fVal = $("#f-val");
    var offset  = $canvas.offset();
```

```
    var stage, maxF, lastF, intVal, label, vertMarker,
        auxTris, tris, edges, rects, verts, mouse;

    $(document).keypress(e => { if(e.which == 13) endStage();
         });

    reset();

    function reset() {
        two.clear();
        $("*").unbind();

        $canvas.contextmenu(e => { e.preventDefault() });
        $("#reset").click(reset);
        $("#compute-reeb").hide();

        createGrid();

        stage = 1;
        maxF = -Infinity;
        intVal = 0;

        auxTris = two.makeGroup(),
        tris = two.makeGroup(),
        edges = two.makeGroup(),
        rects = two.makeGroup(),
        verts = two.makeGroup();

        $("#dual").prop("disabled", true);
        $("#extend").prop("disabled", true);
        $("#integrate").prop("checked", false).parent().
            addClass("disabled").removeClass("active");
        $("#integral").hide().parent();
        $("#reeb").hide();
        $("#reeb svg").remove();

        $fVal.prop("disabled", false);
        $fVal.val(1).select();

        mouse = new Two.Anchor();
        vertMarker = two.makeCircle(0, 0, RADIUS);
        vertMarker.opacity = 0.2;
        vertMarker.fill = "black";
        vertMarker.noStroke();
```

```

        label = new Two.Text("Click to add a vertex. Press
            enter to start adding edges.", two.width/2, two.
            height - 50, {family: "'Helvetica Neue', Helvetica
            , Arial, sans-serif"});
        label.fill = "black";
        label.size = 20;
        two.add(label);

        $canvas.mousedown(e => {
            e.preventDefault();
            addVertex(e);
        });
        $canvas.mousemove(e => {
            mouse.x = e.clientX - offset.left;
            mouse.y = e.clientY - offset.top;
            vertMarker.translation.set(mouse.x, mouse.y);
            two.update();
        });

        two.update();
    }

    function addVertex(e) {
        var fVal = parseInt($fVal.val());
        if (!isNaN(fVal)) {
            var vert = two.makeCircle(mouse.x, mouse.y,
                RADIUS);
            lastF = fVal;
            vert.fVal = fVal;
            vert.dim = 0;
            vert.adj = [];
            vert.lowerEdges = [];
            vert.upperEdges = [];
            vert.equiEdges = [];
            vert.cotris = [];
            vert.placed = true;
            vert.processed = false;

            verts.children.forEach(vert2 => {
                var [a, b] = [vert, vert2].sort((a, b) => {
                    return a.fVal - b.fVal; });

                var edge = two.makeLine(a.translation.x, a.
                    translation.y, b.translation.x, b.
```

```
                    translation.y);

                var v = new Two.Vector(-edge.vertices[0].y, edge.vertices[0].x);
                var u = new Two.Vector(edge.vertices[0].x, edge.vertices[0].y);
                var pt = new Two.Vector();
                var rect = two.makePath();
                v.setLength(RADIUS);
                pt.add(v, u);
                v.multiplyScalar(2);
                u.multiplyScalar(2);
                rect.vertices.push(new Two.Anchor(pt.x, pt.y));
                pt.subSelf(v);
                rect.vertices.push(new Two.Anchor(pt.x, pt.y));
                pt.subSelf(u);
                rect.vertices.push(new Two.Anchor(pt.x, pt.y));
                pt.addSelf(v);
                rect.vertices.push(new Two.Anchor(pt.x, pt.y));
                rect.translation.copy(edge.translation);
                rect.noStroke().noFill();
                rects.add(rect);

                edge.stroke = GRAY;
                edge.opacity = 0;
                edge.faces = [a, b];
                edge.linewidth = LINEWIDTH;
                edge.dim = 1;
                edge.placed = false;
                edges.add(edge);

                two.update();
                edge.rect = rect;
            });

            verts.add(vert);
            recolor(fVal);
        }

        $fVal.val(lastF).select();
        two.update();
```

```
    }

    function endStage() {
        switch (stage) {
            case 1:
                verts.children.sort((u, v) => { return u.fVal - v.fVal; });

                vertMarker.opacity = 0;
                edges.children.forEach(edge => { bindEdge(edge); });

                label.value = "Click to add an edge. Press enter to start adding faces.";
                $canvas.unbind();
                stage = 2;
                break;

            case 2:
                var rectsToRemove = [];
                var edgesToRemove = [];
                edges.children.forEach(edge => {
                    if (!edge.placed) {
                        rectsToRemove.push(edge.rect);
                        edgesToRemove.push(edge);
                    }
                });
                edges.remove(edgesToRemove);
                rects.remove(rectsToRemove);

                tris.children.forEach(tri => { bindTri(tri); });

                label.value = "Click to add a face. Press enter to finish.";
                stage = 3;
                break;

            case 3:
                trisToRemove = [];
                tris.children.forEach(tri => { if (!tri.placed) trisToRemove.push(tri); });
                tris.remove(trisToRemove);


```

```
$("#integrate").on("change", () => {
    if (!$("#integrate").parent().hasClass("disabled")) {
        if ($("#integrate").prop("checked")) {
            label.value = "Click on a simplex to add it to X.";
            $("#extend").prop("disabled", true);
            $("#dual").prop("disabled", true);
            $("#integral").show();
            $.merge($.merge($.merge([], verts.children), edges.children), tris.children).forEach(simp => {
                bindInt(simp);
            });
        }
        else {
            label.value = "";
            $("#extend").prop("disabled", false);
            $("#dual").prop("disabled", false);
            $("#integral").hide();
            intVal = 0;
            QUEUE.Push(["Text", math, INT_TEX+intVal]);
            $.merge($.merge($.merge([], verts.children), edges.children), tris.children).forEach(simp => {
                unbindInt(simp);
            });
            $("#eul").html(0);
        }
    }
}).parent().removeClass("disabled");
$("#extend").prop("disabled", false).click(() => {
    $("#integrate").parent().addClass("disabled");
    $("#dual").prop("disabled", true);
    edges.children.forEach(edge => {
```

```
                        extendEdge(edge); });
                    tris.children.forEach(tri => { extendTri(tri); });
                    $("#compute-reeb").show();
                    $("#extend").prop("disabled", true);
                });
                $("#compute-reeb").click(() => {
                    $("#compute-reeb").hide();
                    computeReeb();
                });
                $("#dual").prop("disabled", false).on("click", () => {
                    computeDual()
                });
                $fVal.prop("disabled", true);

                verts.children.forEach(vert => {
                    $(vert._renderer.elem).mouseover(() => { $fVal.val(vert.fVal); });
                });
                tris.children.forEach(tri => {
                    $(tri._renderer.elem).mouseover(() => { $fVal.val(tri.fVal); });
                });
                edges.children.forEach(edge => {
                    $(edge.rect._renderer.elem).mouseover(() => { $fVal.val(edge.fVal); });
                });

                label.value = "";

                stage = 4;

                break;
        }

        two.update();
        function bindEdge(edge) {
            $(edge.rect._renderer.elem).mouseover(() => {
                edge.opacity = 1;
                two.update();
            }).mouseout(() => {
                edge.opacity = 0;
                two.update();
            }).mousedown(e => {
```

```
                e.preventDefault();
                var fVal = parseInt($fVal.val());
                if (!isNaN(fVal)) {
                    edge.placed = true;
                    lastF = fVal;
                    edge.fVal = fVal;
                    edge.cofaces = [];
                    edge.isEquiedge = false;
                    recolor(fVal);

                    var i = edge.faces[0],
                        j = edge.faces[1];

                    if (i.fVal > j.fVal) {
                        i.lowerEdges.push(edge);
                        j.upperEdges.push(edge);
                    }
                    else if (i.fVal < j.fVal) {
                        j.lowerEdges.push(edge);
                        i.upperEdges.push(edge);
                    }
                    else {
                        i.equiEdges.push(edge);
                        j.equiEdges.push(edge);
                        edge.isEquiedge = true;
                    }

                    i.adj.forEach(k => {
                        if (j.adj.includes(k)) {
                            var [a, b, c] = [i, j, k].sort((a
                                , b) => { return a.fVal - b.
                                fVal; });
                            var containsVert = false;
                            verts.children.forEach(v => {
                                if (![a, b, c].includes(v))
                                    containsVert =
                                        containsVert ||
                                        pInTri(v.translation.
                                            x, v.translation.y
                                            ,
                                            a.translation.x,
                                                a.translation.
                                                y,
                                            b.translation.x,
                                                b.translation.
```

```
                                                y,
                                            c.translation.x, c.translation.y);
                            });
                            if (!containsVert) {
                                var tri = two.makePath(a.translation.x, a.translation.y, b.translation.x, b.translation.y, c.translation.x, c.translation.y);
                                tri.noStroke();
                                tri.fill = GRAY;
                                tri.opacity = 0;
                                tri.dim = 2;
                                tri.placed = false;
                                tri.processed = false;

                                var faces = [];
                                edges.children.forEach(edge => {
                                    if ([a, b, c].includes(edge.faces[0]) && [a, b, c].includes(edge.faces[1]))
                                        faces.push(edge);
                                });
                                faces.sort((a, b) => {
                                    if (a.faces[0] == b.faces[0]) return a.faces[1].fVal - b.faces[1].fVal;
                                    return a.faces[0].fVal - b.faces[0].fVal;
                                });
                                tri.oneFaces = faces;
                                tri.zeroFaces = [a, b, c];

                                tris.add(tri);
                            }
                        }
                    });

```

```
                    i.adj.push(j);
                    j.adj.push(i);

                    $(edge.rect._renderer.elem).unbind();

                    var rectsToRemove = [];
                    var edgesToRemove = [];
                    edges.children.forEach(tempEdge => {
                        if (doIntersect(i.translation, j.translation, tempEdge.faces[0].translation, tempEdge.faces[1].translation)) {
                            rectsToRemove.push(tempEdge.rect);
                            edgesToRemove.push(tempEdge);
                        }
                    });
                    edges.remove(edgesToRemove);
                    rects.remove(rectsToRemove);

                    $fVal.val(lastF).select();
                    two.update();
                }
            });
        }

        function bindTri(tri) {
            $(tri._renderer.elem).mouseover(() => {
                tri.opacity = 1;
                two.update();
            }).mouseout(() => {
                tri.opacity = 0;
                two.update();
            }).mousedown(e => {
                e.preventDefault();
                var fVal = parseInt($fVal.val());
                if (!isNaN(fVal)) {
                    lastF = fVal;
                    tri.placed = true;
                    tri.fVal = fVal;
                    tri.oneFaces.forEach(edge => { edge.cofaces.push(tri); });
                    tri.zeroFaces.forEach(vert => { vert.cotris.push(tri); });
                    recolor(fVal);
```

```
                    $fVal.val(lastF).select();

                    $(tri._renderer.elem).unbind();
                }
            });
        }
    }

    function computeReeb() {
        var reeb = new graphlib.Graph({ multigraph: true });
        var compMap = new Map();

        var preimage = new graphlib.Graph();
        edges.children.forEach(edge => {
            if (!edge.isEquiedge) preimage.setNode(edge.id);
        });

        var components = graphlib.alg.components(preimage);

        verts.children.forEach(vert => {
            if (!vert.processed) {
                vert.processed = true;

                var equiVerts = [vert];
                var stack = [vert]
                while (stack.length > 0) {
                    var currentV = stack.pop();
                    currentV.equiEdges.forEach(equiEdge => {
                        equiEdge.faces.forEach(equiV => {
                            if (!equiV.processed) {
                                equiV.processed = true;
                                equiVerts.push(equiV);
                                stack.push(equiV);
                            }
                        });
                    });
                }

                var lowerComps = new Set();
                equiVerts.forEach(equiV => {
                    lowerComps = new Set([...lowerComps, ...
                        getLowerComps(equiV, components)]);
                });

                // update preimage
```

```
            vert.cotris.forEach(tri => {
                if (vert == tri.zeroFaces[0]) preimage.setEdge(tri.oneFaces[0].id, tri.oneFaces[1].id);
                else if (tri.zeroFaces[0] == tri.zeroFaces[1] || tri.zeroFaces[1] == tri.zeroFaces[2]) {
                    if (vert == tri.zeroFaces[2] && tri.processed) preimage.removeEdge(tri.oneFaces[0].id, tri.oneFaces[1].id);
                    else tri.processed = true;
                }
                else {
                    if (vert == tri.zeroFaces[1]) {
                        preimage.removeEdge(tri.oneFaces[0].id, tri.oneFaces[1].id);
                        preimage.setEdge(tri.oneFaces[1].id, tri.oneFaces[2].id);
                    }
                    else preimage.removeEdge(tri.oneFaces[1].id, tri.oneFaces[2].id);
                }
            });

            components = graphlib.alg.components(preimage);

            var upperComps = new Set();
            equiVerts.forEach(equiV => {
                upperComps = new Set([...upperComps, ...getUpperComps(equiV, components)]);
            });

            //update reeb
            if (upperComps.size != lowerComps.size || upperComps.size != 1) {
                reeb.setNode(reeb.nodeCount(), vert.fVal);
                upperComps.forEach(upperComp => {
                    compMap.set(upperComp, reeb.nodeCount()-1);
                });
                lowerComps.forEach(lowerComp => {
                    reeb.setEdge(reeb.nodeCount()-1,
```

```
                            compMap.get(lowerComp), "",
                            lowerComp);
                    });
                }
                else if (upperComps.size == 1) {
                    compMap.set(upperComps.values().next().
                        value, compMap.get(lowerComps.values()
                        .next().value));
                }
            }
        });

        var graph_serialized = graphlib.json.write(reeb);
        var nodes = graph_serialized["nodes"];
        var links = graph_serialized["edges"];

        var width = $("#reeb").show().innerWidth(),
            height = 400;

        var y_max = d3.max(nodes, d => { return d.value; }),
            y_min = d3.min(nodes, d => { return d.value; });

        var y = d3.scale.linear()
            .domain([y_max, y_min])
            .range([20, height-20]);

        var nodesMap = d3.map();
        nodes.forEach(n => { nodesMap.set(n.v, n); });

        var linkcount = new Map();

        links.forEach(l => {
            var [from, to] = [l.v, l.w].sort();
            var id = `${from}-${to}`;
            if (linkcount.has(id)) linkcount.set(id,
                linkcount.get(id) + 1);
            else linkcount.set(id, 1);

            l.source = nodesMap.get(l.v);
            l.target = nodesMap.get(l.w);
        });

        links.sort((a, b) => {
            if (a.source > b.source) return 1;
            else if (a.source < b.source) return -1;
```

```
            else {
                if (a.target > b.target) return 1;
                if (a.target < b.target) return -1;
                else return 0;
            }
        });

        for (var i=0; i<links.length; i++) {
            if (i != 0 &&
                links[i].source == links[i-1].source &&
                links[i].target == links[i-1].target)
                links[i].linknum = links[i-1].linknum + 1;
            else links[i].linknum = 1;
        };

        var force = d3.layout.force()
            .size([width, height]);

        var svg = d3.select("#reeb").append("svg")
            .attr("width", width)
            .attr("height", height);

        var g = svg.append("g");

        force.nodes(nodes)
            .links(links)
            .start();

        var link = g.selectAll("path")
            .data(links)
            .enter().append("path")
            .attr("class", "link");

        var node = g.selectAll("circle")
            .data(nodes)
            .enter().append("circle")
            .attr("r", 6)
            .style("fill", d => { return compColor(d.value);
               } )
            .on("mouseover", d => { $fVal.val(d.value) })
            .call(force.drag);

        function linkArc(d) {
            var [from, to] = [d.source.v, d.target.v].sort();
            var count = linkcount.get(`${from}-${to}`);
```

```
            var dx = d.target.x - d.source.x,
                dy = y(d.target.value) - y(d.source.value);
            var dr;
            if (count % 2 == 1 && d.linknum == count) dr = 0;
            else dr = Math.sqrt(dx * dx + dy * dy) / (
               parseInt((d.linknum-1)/2)+1) * 2;
            var dir = (d.linknum % 2 == 0) * 1;
            return `M ${d.source.x} ${y(d.source.value)} A ${
               dr} ${dr}, 0, 0, ${dir}, ${d.target.x} ${y(d.
               target.value)}`;
        }

        force.on("tick", () => {
            link.attr("d", linkArc);

            node.attr("cx", d => { return d.x; })
                .attr("cy", d => { return y(d.value); });
        });


        function getLowerComps(vert, components) {
            var lowerComps = new Set();
            vert.lowerEdges.forEach(lowerEdge => {
                var representative;
                components.forEach(component => {
                    if (component.includes(lowerEdge.id))
                        representative = component[0];
                });
                lowerComps.add(representative);
            });
            return lowerComps;
        }

        function getUpperComps(vert, components) {
            var upperComps = new Set();
            vert.upperEdges.forEach(upperEdge => {
                var representative;
                components.forEach(component => {
                    if (component.includes(upperEdge.id))
                        representative = component[0];
                });
                upperComps.add(representative);
            });
            return upperComps;
        }
```

```
    }

    function bindInt(simp) {
        setBW(simp);
        if (simp.dim == 1) elem = $(simp.rect._renderer.elem)
            ;
        else elem = $(simp._renderer.elem);
        elem.on("mouseover.int", () => {
            if (simp.inInt) setBW(simp);
            else setColor(simp);
        }).on("mouseout.int", () => {
            if (simp.inInt) setColor(simp);
            else setBW(simp);
        }).on("mousedown.int", () => {
            var simpVal;
            if (simp.inInt) {
                simpVal = -simp.fVal;
                if (simp.dim == 1) simpVal *= -1;
                simp.inInt = false;
                setBW(simp);
            }
            else {
                simpVal = simp.fVal;
                if (simp.dim == 1) simpVal *= -1;
                simp.inInt = true;
                setColor(simp);
            }
            intVal += simpVal;

            QUEUE.Push(["Text", math, INT_TEX+intVal]);
            two.update();
        });
    }

    function unbindInt(simp) {
        setColor(simp);
        simp.inInt = false;
        if (simp.dim == 1) elem = $(simp.rect._renderer.elem)
            ;
        else elem = $(simp._renderer.elem);
        elem.unbind(".int");
    }

    function recolor(fVal) {
        maxF = Math.max(Math.abs(fVal), maxF);
```

```
        $.merge($.merge($.merge([], verts.children), edges.children), tris.children).forEach(simp => {
            if (simp.placed) setColor(simp);
        });
    }

    function setBW(simp) {
        if (simp.fVal > 0) var c = 255 - Math.round(255 * simp.fVal / maxF);
        else var c = 255 - Math.round(255 * simp.fVal / (-maxF));
        simp.stroke = simp.fill = `rgb(${c}, ${c}, ${c})`;
        two.update();
    }

    function setColor(simp) {
        simp.stroke = simp.fill = compColor(simp.fVal);
        two.update();
    }

    function compColor(fVal) {
        if (fVal > 0) {
            var ratio = fVal / maxF;
            var r = Math.round(POS_COLOR.r + (1-ratio) * (255-POS_COLOR.r));
            var g = Math.round(POS_COLOR.g + (1-ratio) * (255-POS_COLOR.g));
            var b = Math.round(POS_COLOR.b + (1-ratio) * (255-POS_COLOR.b));
        }
        else {
            var ratio = -fVal / maxF;
            var r = Math.round(NEG_COLOR.r + (1-ratio) * (255-NEG_COLOR.r));
            var g = Math.round(NEG_COLOR.g + (1-ratio) * (255-NEG_COLOR.g));
            var b = Math.round(NEG_COLOR.b + (1-ratio) * (255-NEG_COLOR.b));
        }
        return `rgb(${r}, ${g}, ${b})`;
    }

    function extendEdge(edge) {
        var fVal1 = edge.faces[0].fVal,
```

```
            fVal2 = edge.faces[1].fVal;

        var stops = [new Two.Stop(0, compColor(fVal1), 1)];
        if (fVal1 * fVal2 < 0)
            stops.push(new Two.Stop(Math.abs(fVal1)/(Math.abs
                (fVal1)+Math.abs(fVal2)), "white", 1));
        stops.push(new Two.Stop(1, compColor(edge.faces[1].
            fVal), 1));

        edge.stroke = new Two.LinearGradient(edge.vertices
            [0].x, edge.vertices[0].y, edge.vertices[1].x,
            edge.vertices[1].y, stops);

        two.update();

        $(edge.rect._renderer.elem).unbind("mouseover").
            mousemove(e => {
            mouse.x = e.clientX - offset.left;
            mouse.y = e.clientY - offset.top;
            $fVal.val(calcEdgeFVal(edge, mouse.x, mouse.y));
        });
    }

    function calcEdgeFVal(edge, x, y) {
        var trans = edge.translation;
        var a = edge.rect.vertices[0];
        var b = edge.rect.vertices[1];
        var c = edge.rect.vertices[2];
        var d = edge.rect.vertices[3];

        var d1 = pToSeg(x, y, a.x + trans.x, a.y + trans.y, b
            .x + trans.x, b.y + trans.y);
        var d2 = pToSeg(x, y, c.x + trans.x, c.y + trans.y, d
            .x + trans.x, d.y + trans.y);

        var l1 = d1 / (d1+d2);
        var l2 = d2 / (d1+d2);

        return (l2*edge.faces[0].fVal + l1*edge.faces[1].fVal
            ).toFixed(2);
    }

    function extendTri(tri) {
        tri.opacity = 0;
        subdivTri(tri, 1, tri);
```

```
        two.update();

        $(tri._renderer.elem).unbind("mouseover").mousemove(e => {
            mouse.x = e.clientX - offset.left;
            mouse.y = e.clientY - offset.top;
            $fVal.val(calcTriFVal(tri, mouse.x, mouse.y));
        });
    }

    function subdivTri(tri, i, realTri) {
        var a = new Two.Anchor(tri.vertices[0].x + tri.translation.x, tri.vertices[0].y + tri.translation.y);
        var b = new Two.Anchor(tri.vertices[1].x + tri.translation.x, tri.vertices[1].y + tri.translation.y);
        var c = new Two.Anchor(tri.vertices[2].x + tri.translation.x, tri.vertices[2].y + tri.translation.y);

        var d = new Two.Anchor((a.x+b.x)/2, (a.y+b.y)/2);
        var e = new Two.Anchor((b.x+c.x)/2, (b.y+c.y)/2);
        var f = new Two.Anchor((a.x+c.x)/2, (a.y+c.y)/2);

        var t1 = two.makePath(a.x, a.y, d.x, d.y, f.x, f.y);
        var t2 = two.makePath(d.x, d.y, e.x, e.y, f.x, f.y);
        var t3 = two.makePath(f.x, f.y, e.x, e.y, c.x, c.y);
        var t4 = two.makePath(d.x, d.y, b.x, b.y, e.x, e.y);

        auxTris.add(t1, t2, t3, t4);
        auxTris.remove(tri)

        if (i < RESOLUTION) {
            subdivTri(t1, i+1, realTri);
            subdivTri(t2, i+1, realTri);
            subdivTri(t3, i+1, realTri);
            subdivTri(t4, i+1, realTri);
        }
        else {
            t1.fVal = calcTriFVal(realTri, t1.translation.x, t1.translation.y);
            setColor(t1);
            t2.fVal = calcTriFVal(realTri, t2.translation.x,
```

```
                t2.translation.y);
            setColor(t2);
            t3.fVal = calcTriFVal(realTri, t3.translation.x,
                t3.translation.y);
            setColor(t3);
            t4.fVal = calcTriFVal(realTri, t4.translation.x,
                t4.translation.y);
            setColor(t4);
            two.update();
        }
    }

    function calcTriFVal(tri, x, y) {
        var a = tri.zeroFaces[0];
        var b = tri.zeroFaces[1];
        var c = tri.zeroFaces[2];

        var x1 = a.translation.x;
        var y1 = a.translation.y;
        var x2 = b.translation.x;
        var y2 = b.translation.y;
        var x3 = c.translation.x;
        var y3 = c.translation.y;

        var l1 = ((y2-y3)*(x-x3) + (x3-x2)*(y-y3)) / ((y2-y3)
            *(x1-x3) + (x3-x2)*(y1-y3));
        var l2 = ((y3-y1)*(x-x3) + (x1-x3)*(y-y3)) / ((y2-y3)
            *(x1-x3) + (x3-x2)*(y1-y3));
        var l3 = 1 - l1 - l2;

        return (l1 * a.fVal + l2 * b.fVal + l3 * c.fVal).
            toFixed(2);
    }

    function computeDual() {
        maxF = -Infinity;
        verts.children.forEach(vert => {
            var fVal = vert.fVal;
            $.merge($.merge([], vert.lowerEdges), vert.
                upperEdges).forEach(edge => {
                fVal -= edge.fVal;
                var triVal = 0;
                edge.cofaces.forEach(tri => { triVal += tri.
                    fVal; });
                fVal += triVal/2;
```

```
            });
            vert.fVal = fVal;
            recolor(fVal);
        });
        edges.children.forEach(edge => {
            var fVal = -edge.fVal;
            edge.cofaces.forEach(tri => { fVal += tri.fVal; });
            edge.fVal = fVal;
            recolor(fVal);
        });
    }


    function createGrid() {

        var size = 30;
        var bg = new Two({
            type: Two.Types.canvas,
            width: size,
            height: size
        });

        var a = bg.makeLine(bg.width / 2, 0, bg.width / 2, bg.height);
        var b = bg.makeLine(0, bg.height / 2, bg.width, bg.height / 2);
        a.stroke = b.stroke = "#e5efff";

        bg.update();

        $canvas.css({
            background: `url( ${bg.renderer.domElement.toDataURL("image/png")} ) 0 0 repeat`,
            backgroundSize: `${size}px ${size}px`
        });
    }
});

function distance(p, q) {
    return Math.sqrt(Math.pow(p.x-q.x, 2) + Math.pow(p.y-q.y, 2));
}

function pToSeg(x, y, x1, y1, x2, y2) {
```

```
    var A = x - x1;
    var B = y - y1;
    var C = x2 - x1;
    var D = y2 - y1;

    var dot = A * C + B * D;
    var len_sq = C * C + D * D;
    var param = -1;
    if (len_sq > 0) param = dot / len_sq;

    var xx, yy;

    if (param < 0) {
        xx = x1;
        yy = y1;
    }
    else if (param > 1) {
        xx = x2;
        yy = y2;
    }
    else {
        xx = x1 + param * C;
        yy = y1 + param * D;
    }

    var dx = x - xx,
        dy = y - yy;
    return Math.sqrt(dx * dx + dy * dy);
}

function pInTri(px, py, ax, ay, bx, by, cx, cy) {
    var v0 = [cx-ax, cy-ay];
    var v1 = [bx-ax, by-ay];
    var v2 = [px-ax, py-ay];

    var dot00 = (v0[0] * v0[0]) + (v0[1] * v0[1]);
    var dot01 = (v0[0] * v1[0]) + (v0[1] * v1[1]);
    var dot02 = (v0[0] * v2[0]) + (v0[1] * v2[1]);
    var dot11 = (v1[0] * v1[0]) + (v1[1] * v1[1]);
    var dot12 = (v1[0] * v2[0]) + (v1[1] * v2[1]);

    var invDenom = 1 / (dot00 * dot11 - dot01 * dot01);

    var u = (dot11 * dot02 - dot01 * dot12) * invDenom;
    var v = (dot00 * dot12 - dot01 * dot02) * invDenom;
```

```

    return (u >= 0) && (v >= 0) && (u + v < 1);
}

function doIntersect(p1, q1, p2, q2) {
    var o1 = orientation(p1, q1, p2);
    var o2 = orientation(p1, q1, q2);
    var o3 = orientation(p2, q2, p1);
    var o4 = orientation(p2, q2, q1);

    if (p1 == p2 || p1 == q2 || q1 == p2 || q1 == q2) return false;

    if (o1 != o2 && o3 != o4)
        return true;

    return (o1 == 0 && onSegment(p1, p2, q1)) ||
        (o2 == 0 && onSegment(p1, q2, q1)) ||
        (o3 == 0 && onSegment(p2, p1, q2)) ||
        (o4 == 0 && onSegment(p2, q1, q2));

    function onSegment(p, q, r) {
        return q.x < Math.max(p.x, r.x) && q.x > Math.min(p.x, r.x) &&
            q.y < Math.max(p.y, r.y) && q.y > Math.min(p.y, r.y);
    }

    function orientation(p, q, r) {
        var val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
        if (val == 0) return 0;
        return (val > 0) ? 1 : 2;
    }
}
```