5-2020

A Program For Surface Classification

Sarah Dennis
Sarah Lawrence College

A Program For Surface Classification

Sarah Dennis

May 16, 2020

Abstract

This thesis presents the classification theorem of compact connected surfaces, its proof, and a computer implementation. The theorem, first stated by Möbius in 1861 and more formally by Jordan in 1866, states that *every compact, connected surface is homeomorphic to either a 2-sphere, a connected sum of tori, or a connected sum of projective planes.* The proof we present consists of an algorithm to take a polygon representation to normal form. A polygon representation of a surface is essentially the result of cutting the surface up until it lies flat, where we identify the cut edges such that we could paste it back together again. A polygon representation in normal form has distinguishable tori or projective plane components and can be easily classified from this state. The computer implementation is written in JavaScript and includes a graphical interface that allows the user to progress through the algorithm by cut and paste operations. The program is intended as a learning tool for students in introductory topology courses.

1 Introduction

Classification sorts a set of objects into distinct categories. Through the process of classification, objects can be recognised, differentiated, and understood more easily. We put resources towards the cause of classification so that we can simplify problems and generalise knowledge in this complicated world.

One classic example of classification in mathematics is that of quadratic polynomials. An equation of the form $ax^2+bx+c=0$ with $a,b,c\in\mathbb{R}$ has at most two solutions x_1 and x_2. We can find out what form x_1 and x_2 take by analysing the discriminant $d=b^2-4ac$. If $d>0$ we know that x_1 and x_2 exist in $\mathbb{R}$ and are distinct. If $d=0$ we know that $x_1=x_2$. And if $d<0$ we know x_1 and x_2 are complex conjugates. This classification is enlightening for a number of reasons. When dealing integer coefficients, we may try to find a solution finding a solution through trial and error and this process is greatly simplified once we know the form of the solutions. Furthermore, we learn how the graph $y=ax^2+bx+c$ will interact with the x axis, either crossing it twice, or touching once or not at all. Finally, we get a sense of a certain symmetry in the quadratic equation, particularly in that complex roots come in conjugate pairs. Classifying a quadratic polynomial provides us insight to the functions behaviour with minimal work, and the classification process is independent of the form of the polynomial.

The Classification Theorem for Surfaces is central to elementary topology and to our discussion here. In a broad sense, this theorem states that we have a small number of building block surfaces from which all other surfaces may be constructed. The Classification Theorem is not dissimilar to our classification of natural numbers through prime decomposition. We know that any natural number n can be written as $n=p_1^{\alpha_1}p_2^{\alpha_2}\cdots p_k^{\alpha_k}$ where the set $\{p_i^{\alpha_i}\}$ of primes and corresponding exponents are unique to n. Then when n has 2 as a prime factor, we say n is even. And when two natural numbers $m>n$ share a prime factor, we know that m is divisible by n. Prime decomposition demonstrates the importance of prime numbers as the building blocks for our basic numerical system. The building block surfaces described in the Classification Theorem are similarly important to topology as prime numbers are to number theory in that the properties of a composite surface are uniquely determined by the building block surfaces it is composed of.

Students of mathematics will likely learn how to find a prime decomposition by hand through a division algorithm. Trying to find the prime decomposition of a very large number by hand is an arduous task however, and the increased number of steps for determining the prime decomposition of a large number leaves more room for human error. The same goes for surfaces. Classifying 'simple' surfaces by hand is not difficult once the method is learned, but the process becomes increasingly complicated

with the complexity of the surface. There are many freely available programs that can find the prime decomposition of a natural number. So why do we not also have a program for classifying surfaces?

The following paper will present a proof of the Classification Theorem, and the necessary background in topology to follow the details of this proof. This proof was chosen for is algorithmic nature, which lends itself well to constructing a program for surface classification. Following the proof, we will outline a program for classifying surfaces. This program allows for immediate surface classification and also provides a drawing window for performing cut and paste operations. We also present a thorough example of how cut and paste operations can be used to classify a surface from start to finish; this process could be easily carried out in our program. Finally, we present the underlying algorithm that allows our program to perform immediate surface classification.

2 The Classification Theorem for Surfaces

Theorem 1. *Every compact, connected surface is homeomorphic to a 2-sphere, a connected sum of tori, or a connected sum of projective planes. The Euler characteristic and orientability distinguish among these possibilities.*

The Classification Theorem for Surfaces essentially states that every 'nice' surface has either some number of holes, or some number of twists. In the physical world around us, we encounter very few real objects containing twists – most everyday objects have zero or more holes. For example, a coffee cup and a donut both have one hole. A pair of pants and a pair of scissors both have two holes. Well known examples of surfaces with twists include the möbius strip (with one twist) and Klein bottle (with two twists); I think the most common place to find these objects is on a mathematician's desk.

2.1 Preliminary definitions

We will now outline some elementary definitions from topology so we can understand the statement of The Classification Theorem more thoroughly.

Definition 1. A **homeomorphism** between two topological spaces (X, τ_X) and (Y, τ_Y) is a continuous bijection $f : X \to Y$ with a continuous inverse $f^{-1} : Y \to X$. When such a function exists, we say X is homeomorphic to Y.

The important notion behind a homeomorphism is that it is a smooth operation that can be undone. In terms of surfaces, stretching, compressing, moulding and reorienting are all smooth, continuous operations that can be described by a homeomorphism. On the other hand, cutting, tearing, and gluing are not continuous operations and cannot be undone, and hence have no corresponding homeomorphism.

Previously we have been using the term 'surface' very generally, but now we will be more specific. The Classification Theorem specifically refers to surfaces that are compact and connected.

Definition 2. A **compact connected surface** S is a topological space (X, τ) where all points in the interior of S have open balls around them of arbitrarily small radius locally homeomorphic to the plane, and all points on the boundary of S have half open balls around them of arbitrarily small radius. Furthermore, S cannot be written as the disjoint union of two open sets, and every open cover of S has some finite subcover.

Compactness is difficult to define informally. Most things we think of as finite surfaces are compact, but we must also remember that any open subset of $\mathbb{R}^n$ is not compact. Connectedness means, informally, that the surface is all in one piece. For example, a dinner plate is connected, but a dinner plate that has shattered after being dropped on the ground is not connected.

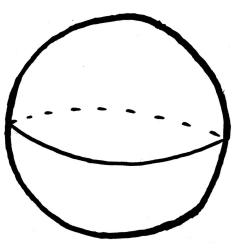
Finally, the Classification Theorem mentions the notions connected sum, orientability and Euler characteristic. We postpone a formal definition of these terms for later on. Informally, the connected sum of two compact connected surfaces M and N is the result gluing them together; the Euler characteristic of a surface is an integer determined algebraically from inherent properties of the surface; and orientability is determined by the presence of a twist in the surface.

2.2 Three fundamental surfaces

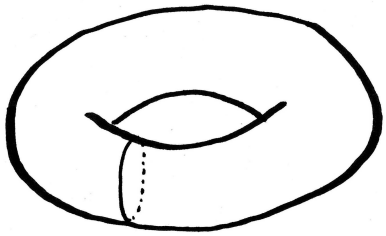
The following surfaces are referenced in the classification theorem as our three building block surfaces.

- ⋆ The **2-sphere**: we know this surface in $\mathbb{R}^3$ as $\{(x, y, z) : x^2 + y^2 + z^2 = 1\}$. Going forward, we will denote a 2-sphere by $\mathcal{S}2$.

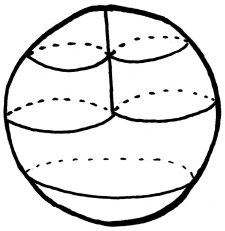
- ⋆ The **torus**: we can describe the torus in $\mathbb{R}^3$ as the surface of revolution, obtained by revolving a circle in plane E around some disjoint line also in E. We will denote a torus by $\mathcal{T}2$.
- ⋆ The **projective plane**: The projective plane can be realised by adding a line at infinity. Take a plane in $\mathbb{R}^3$, and say that parallel lines of any one specific direction in this plane meet at a unique point at infinity. Then, connecting all these points at infinity for all possible directions of parallel lines we obtain a line at infinity. The union of the plane with this line at infinity is a real projective plane. We will denote a projective plane by $\mathcal{P}2$.



A 2-sphere



A torus



A projective plane

Figure 1: Surface diagrams of three fundamental surfaces

3 Polygons

The proof of The Classification Theorem relies on representing a compact connected surface as a polygon in $\mathbb{R}^2$, and then manipulating this polygon into a form that we can recognise.

Definition 3. Given a natural number $n > 2$, a point c in $\mathbb{R}^2$ and a real number $a > 0$, place n points $v_0, v_1, ..., v_n = v_0$ on a circle centred at c with radius a such that the distance between any v_i and v_{i+1} is equal. Connect each v_i to v_{i+1} with a line segment e_i. A **polygon** P is the set of vertices $\{v_i\}_i$, the set of edges $\{e_i\}_i$ and the region of $\mathbb{R}^2$ bounded by the closed path $e_0, e_1, ...e_{n-1}$ from v_0 to $v_n = v_0$. See figure 2.[1]

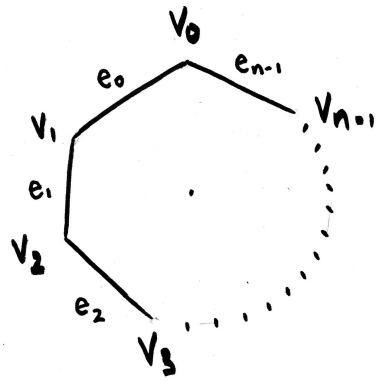


Figure 2: A polygon with edges $\{e_i\}_0^{n-1}$ and vertices $\{v_i\}_0^{n-1}$

Definition 4. We will also extend the definition of a polygon to cover $n = 2$ vertices. Then the polygon P is a **bi-gon** in $\mathbb{R}^2$ centered at the origin with vertices at $v_1 = (-1, 0)$ and $v_2 = (1, 0)$ and two 'edges' e_1 and e_2 (now half-circle arcs) between v_1 and v_2.

Definition 5. Given an edge e between vertices a and b, an **orientation** on e is an ordering of a and b. The first vertex is called the initial point of e, and the second vertex is called the terminal point of e. If a is the initial point of e, and b the terminal point, we say that e is an edge from a to b.

[1] In any of the following polygonal diagrams, dots between vertices v_i and v_j represent the presence of zero or more edges between v_i and v_j. If there are zero edges, then $v_i = v_j$.

Definition 6. If l is a line segment from a to b and l' is a line segment from c to d in $\mathbb{R}^2$, then the **positive linear map** of l onto l' is the homeomorphism h that carries a point $x = (1-t)a + tb$ on l to a point $h(x) = (1-t)c + td$ on l'.

In essence, a positive linear map is way of mapping one edge onto another such that their initial points line up, their terminal points line up, and all points in between follow as we expect.

Definition 7. A **labelling** of a polygon P is a bijective mapping between the edges of P and an alphabet.

We can now describe a polygon with a sequence of labels corresponding to its edges, given in counter-clockwise order. For example, in Figure 2 we would write $P = e_0 e_1 \cdots e_{n-1}$.

Definition 8. Let P be a polygon with oriented, labelled edges, and with vertices $v_0, v_1, ..., v_n$ where $(v_0 = v_n)$. Let $a_1, ..., a_n$ denote the edge labels for P. Then for each $k = 1, ..., n$, let a_{i_k} be the label for the edge $v_{k-1}v_k$, and let $\epsilon_k = +1$ if the edge is oriented v_{k-1} to v_k or $\epsilon_k = -1$ if the edge is oriented v_k to v_{k-1}. Then the oriented polygon P is fully described by the **labelling scheme** $w = (a_{i_1})^{\epsilon_1}(a_{i_2})^{\epsilon_2} \cdots (a_{i_k})^{\epsilon_k}$. The exponents of $+1$ are normally omitted.

We can extend this definition to cover the case of multiple polygons in a single space. Given a finite number $P_1, ..., P_k$ of disjoint polygons in $\mathbb{R}^2$, along with orientations and labelling of their edges, we say the collection of polygons $\{P_i\}_1^k$ has labelling scheme $w_1, ..., w_k$ where each w_i is a **subscheme** corresponding to P_i.

3.1 Surfaces as polygons

We will now describe how an arbitrary compact connected surface can be represented by a polygon in $\mathbb{R}^2$. We start with the notion of triangulation.

Definition 9. A **triangulation** of a space X is a simplicial complex K homeomorphic to X where each 1-simplex belongs to exactly two 2-simplexes.

Informally, a triangulation is a tiling of the surface using triangles (possibly of different shapes and sizes). We then have the following useful theorem:

Theorem 2. *Every compact surface admits a finite triangulation.*

The proof of this theorem is long and beyond our scope so we will omit it. As a very general overview, the proof relies heavily on the compactness of the surface. Given a compact surface S we can find an open cover C with a finite sub-cover C'. We find the finite triangulation of S by transforming each element of C' into a triangulated cell complex. The result of this theorem is very powerful and is essential for us to recognise surfaces in a polygon representation.

Lemma 1. *If S is a compact connected surface then there exists a polygon P and a proper labelling scheme w such that the quotient space arising from these is homeomorphic to S.*

The proof of this lemma requires two further definitions.

Definition 10. Given a labelling scheme w for polygons $\{P_i\}$, define **an equivalence relation** $\sim$ on $\{P_i\}$ as follows. If a point x is in the interior of some P_i then $x \sim x$ only. Given any two edges e_1 and e_2 (of the same or disjoint polygons in $\{P_i\}$) with the same label, paste e_1 to e_2 through the positive linear map h, defining each point x of the first edge to be equivalent to the point $h(x)$ on the second edge.

Definition 11. Let Q and R be polygonal regions with vertices $r_0, r_1, ...r_{k-1}, r_k, r_0$ and $q_0, q_k, q_{k+1}...q_{n-1}, q_0$ respectively. Let a be the edge a from q_0 to q_k and let b be the edge from r_0 to r_k. To **paste** together Q and R along a and b is to identify a and b through a positive linear map, obtaining the new polygonal region P with vertices $r_0, r_1, ..., r_k = q_k, ..., q_{n-1}, q_0$ with $q_0 = r_0$. See figure 3.

Now we can proceed to the proof of Lemma 1.

Proof. Let S is a compact connected surface. Take an n-triangulation $\mathcal{T}$ of S. Give each 1-simplex an orientation and label each 1-simplex uniquely. Then each 2-simplex is a polygon with a length 3 labelling scheme t_i, and our triangulation $\mathcal{T}$ has labelling scheme $t = t_1, t_2, ..., t_n$.

Starting with any triangle in our collection, for one of its edges there is a second triangle with an edge of the same label. Paste the two triangles together along this edge giving a rectangle. For one of this rectangles edges, there is a triangle with an edge of the same label, and we paste this triangle to the rectangle along this edge. We continue this process. It may be the case that for some edge, its pair has already been adjoined, in this case we move on to a different edge. When this is the case for all edges, we must have used all the n triangles because S was connected so the triangulation of S is connected. We

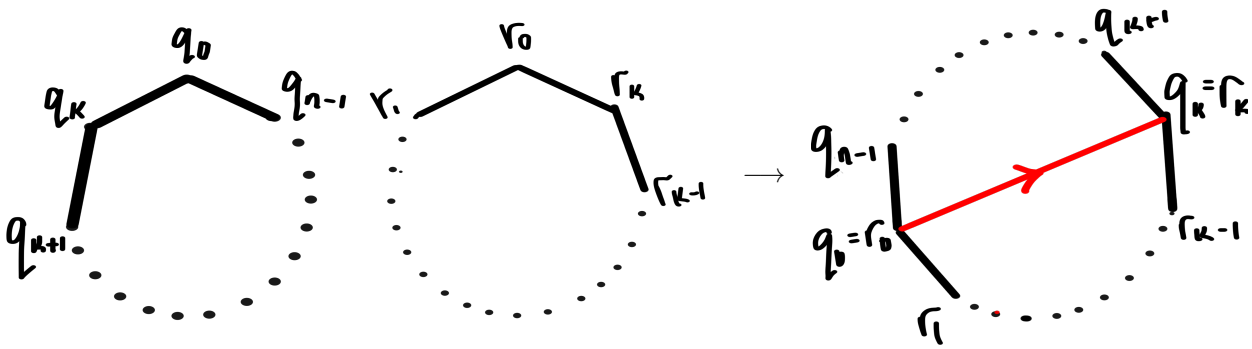


Figure 3: The paste operation along q_0q_k and r_0r_k

also have edges identified in pairs since every label was given to exactly two edges, so if an edge occurs on the boundary of our new polygon, so must its pair.

The quotient space arising from applying $\sim$ to the new polygon is homeomorphic to S since T was homeomorphic to S and, due to our labelling and orienting, reorganising the triangulation did not disrupt the quotient space. □

We have already defined a paste operation on polygons (see definition 11). We wish to define the inverse of this operation: the cut operation.

Definition 12. Let P be a polygonal region with $n > 3$ vertices $v_0, v_1, ..., v_k, ..., v_{n-1}, v_0$ where $1 < k < n-1$. To **cut** P from v_0 to v_k is to transform P into two new disjoint polygons Q and R with vertices $v_0, v_k, ..., v_{n-1}, v_0$ and $v_0, v_1..., v_k, v_0$ respectively. The polygons Q and R inherit the the relation $\sim$ as it was on the points of P, with the added identification that points on the edge $\{v_0v_k\}_Q$ are identified in the natural way with points on the edge $\{v_0v_k\}_R$. See figure 4.

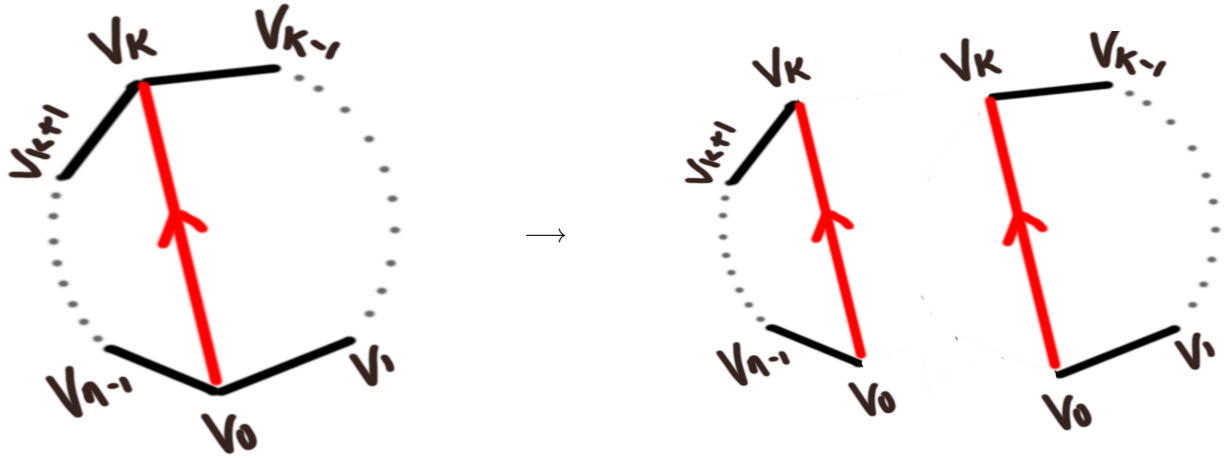


Figure 4: The cut operation from v_0 to v_k

3.2 Polygon operations and the quotient space

Now we will see why polygon representations and the cut and paste operations are useful: we can manipulate the olygon representations of surfaces without changing the surface in the resulting quotient space

Theorem 3. *Suppose $w_1, w_2, ..., w_n$ is the labelling scheme for a collection of polygons $\{P_i\}_1^n$ where w_i is the labelling scheme for P_i. Let X be the quotient space obtained from this labelling scheme. The following operations on this labelling scheme and its associated polygons do not change X up to homeomorphism.*

- ⋆ *Relabelling: Any label a can be replaced with a free label e occurring nowhere else in the scheme without affecting the resulting space. Furthermore, all exponents of a particular label in the labelling scheme can by flipped (-1 to 1 and 1 to -1).*
- ⋆ *Permuting: Any subscheme w_i can be replaced with a cyclic permutation of itself without affecting the resulting scheme.*
- ⋆ *Cutting: Take some P_i with labelling scheme $w_i = y_0y_1$ where y_0, y_1 are sub-labelling schemes with at least 2 elements. We can cut P_i from the terminal point of y_0 to the initial point of y_1 without changing the quotient space. In other words, suppose e is a label not appearing anywhere in the scheme. Then $w_1, ..., y_0ee^{-1}y_1, ..., w_n$ is a labelling scheme representing represents the same space.*
- ⋆ *Pasting: Take polygons P_i and P_j with respective labelling schemes $w_i = y_0e$ and $w_j = e^{-1}y_1$ where e appears in only these two locations in the scheme. We can paste P_i to P_j along e, then $w_1, ..., y_0y_1, ...w_n$ represents the same space.*
- ⋆ *Reflecting: Any subscheme $w_i = (a_{i_1})^{\epsilon_1}(a_{i_2})^{\epsilon_2} \cdots (a_{i_k})^{\epsilon_k}$ can be replaced by its inverse*

$$w_i^{-1} = (a_{i_k})^{-\epsilon_1} \cdots (a_{i_2})^{-\epsilon_2}(a_{i_1})^{-\epsilon_1}$$

  without affecting the resulting scheme.
- ⋆ *Folding: Suppose $w_i = y_0aa^{-1}y_1$ is a subscheme and that a does not appear anywhere else in the scheme. Then we can replace w_i with y_0y_1 without changing the resulting scheme.*

Proof.

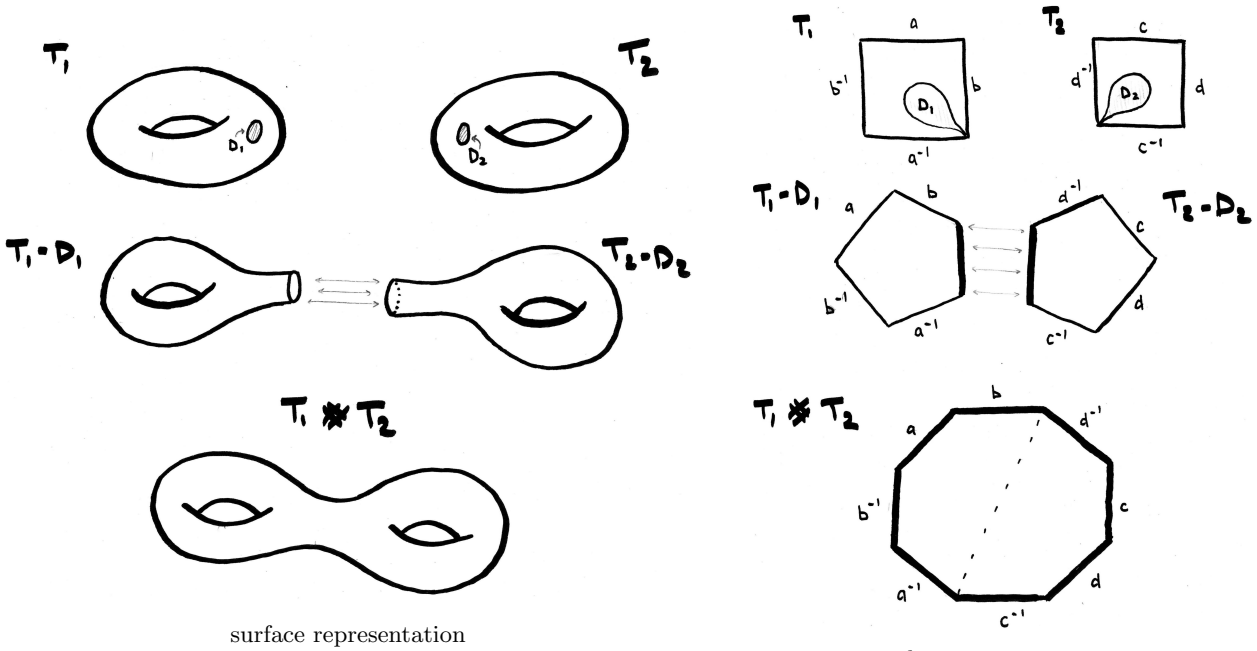
- ⋆ Relabelling is a superficial operation that will not change the resulting quotient space X.
- ⋆ Permuting is also a superficial operation that will not change the resulting quotient space X.
- ⋆ Let P be a polygonal region with vertices $v_0, ..., v_n$ where $v_n = v_0$. Cut P along the edge v_iv_k for some $0 \leq i, k < n - 1$. This gives disjoint polygons Q and r with edges $v_0, v_1, ..., v_i, v_k, ..., v_0$ and $v_k, v_i, ..., v_j, ..., v_k$ where the edge from v_i to v_k in Q is identified with the edge from v_i to v_k in R.Then P is homeomorphic to the quotient space of Q and R obtained by pasting the edge q_iq_k of Q to this edge in R using a positive linear map.
- ⋆ Suppose we have polygons Q and R with vertices $q_0, ..., q_i, q_k, q_0$ and $r_0, r_k, r_i, ..., r_n$ where $r_n = r_0$ respectively. The vertices of R lie in a circle arranged counterclockwise. Place new points $s_1, ..., s_{k-1}$ such that $r_0, s_1, ..., s_{k-1}, r_k, ..., r_n$ with $r_n = r_0$ lie on the circle in this counterclockwise order. Let S be the polygon with these $k - 1$ vertices.There is a homeomorphism between polygons S and Q that carries s_i to q_i and maps the edge q_0q_k linearly onto the edge r_0r_k of R. The quotient space obtained by gluing Q and R together is homeomorphic to the region P that is the union of R and S. So P gives rise to the same quotient space.
- ⋆ Reflecting is again simply the operation of reversing the direction of all edges and will not change the resulting quotient space.
- ⋆ Let b and c be labels that do not appear elsewhere in the total labelling scheme. First replace $y_0aa^{-1}y_1$ with the scheme y_0ab and $b^{-1}ay_1$ using the cut operation. Now combine the edges ab into a single new edge with label c (not appearing anywhere else). This operation is one of relabelling, and will not change the scheme since a and b do not appear anywhere else in the scheme. We then have the scheme y_0c and $c^{-1}y_1$ which we can paste together along c giving y_0y_1.

□

4 Connected Sum and Euler Characteristic

We previously deferred the formal definitions for connected sum, Euler characteristic, and orientability used in the Classification Theorem. We now have to appropriate terminology to understand and explain these definitions.

Definition 13. For two compact, connected surfaces M and N we define the **connected sum** $M\#N$ to be the surface we obtain through removing an open disc from both M and N, then pasting M and N together along the boundaries of these discs. An example of forming the connected sum is give in figure 5.



surface representation

polygon representation

Figure 5: Forming the connected sum of two tori

Definition 14. A surface is **non-orientable** if there exists a closed path such that the normal vector is reversed when moved around this path. In other words, the a non-orientable surface contains a möbius strip. An orientable surface is one containing no möbius strip.

Definition 15. The **Euler characteristic** of a compact connected surface S is an integer denoted $\chi(S)$. For some finite triangulation of S with e edges, v vertices, and f faces, we calculate $\chi(S) = v - e + f$.

We now present several lemmas that relate orientability, Euler characteristic and connected sum, and that we will need in proving the Classification Theorem.

Lemma 2. *The connected sum of two orientable surfaces M and N is orientable, and the connected sum of any surface Q with a non-orientable surface P is non-orientable.*

Proof. Since M and N are orientable, we know that neither contains a Möbius strip. So $M\#N$ cannot contain a Möbius strip and is orientable. Likewise, if P is non-orientable then P does contain a Möbius strip, so $P\#Q$ still contains a Möbius strip and is non-orientable. □

Lemma 3. *For two compact connected surfaces M and N, we have*

$$\chi(M\#N) = \chi(M) + \chi(N) - 2. \tag{1}$$

Proof. Take a triangulation $\mathcal{F}_M = \{T_{M_i}\}_i$ for M and a triangulation $\mathcal{F}_N = \{T_{N_j}\}_j$ for N. When forming $M\#N$, we can remove the interior of some T_{M_i} and some T_{N_j} instead open discs, then glue M and N together along the boundaries of T_{M_i} and T_{N_j} joining vertex to vertex and edge to edge. This gives us $M\#N$ as usual with a new triangulation $\mathcal{F}_{n\#m}$ where

$$\text{vertices}(\mathcal{F}_{N\#M}) = \text{vertices}(\mathcal{F}_M) + \text{vertices}(\mathcal{F}_N) - 3 \tag{2}$$

$$\text{edges}(\mathcal{F}_{N\#M}) = \text{edges}(\mathcal{F}_M) + \text{edges}(\mathcal{F}_N) - 3 \tag{3}$$

$$\text{faces}(\mathcal{F}_{N\#M}) = \text{faces}(\mathcal{F}_M) + \text{faces}(\mathcal{F}_N) - 2 \tag{4}$$

Now,

$$\chi(M\#N) = \text{vertices}(\mathcal{F}_{N\#M}) - \text{edges}(\mathcal{F}_{N\#M}) + \text{faces}(\mathcal{F}_{N\#M}) \tag{5}$$

$$= \chi(M) + \chi(N) - 2 \tag{6}$$

□

5 Proving the Classification Theorem

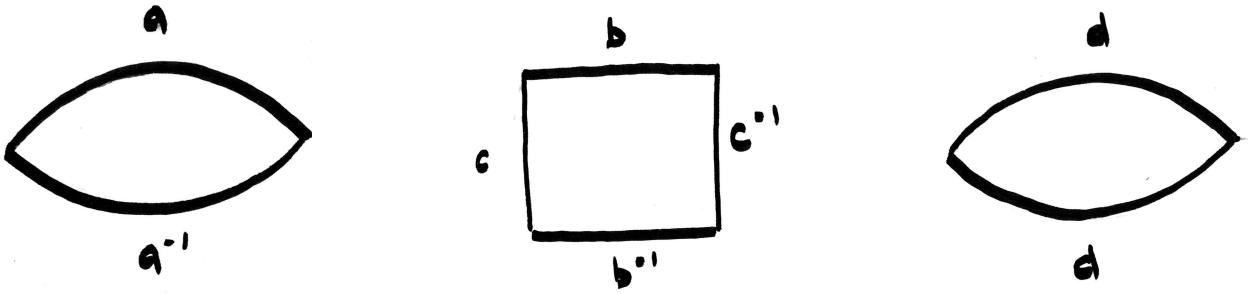
Theorem 1. *Every compact, connected surface is homeomorphic to a 2-sphere, a connected sum of tori, or a connected sum of projective planes. The Euler characteristic and orientability distinguish among these possibilities.*

The proof of this theorem relies on showing that every surface can be represented as a polygon, and that this polygon can be transformed into a normal form without changing quotient space. A polygon in normal form will then be trivial to classify.

5.1 Normal form

Previously we introduced three fundamental surfaces: $\mathcal{S}2$, $\mathcal{T}2$ and $\mathcal{P}2$. It is important that we also know their polygon representation and labelling scheme.

Definition 16. A **projective plane** can described most simply by the labelling scheme $w = aa$. A **torus** can be described by the labelling scheme $u = aba^{-1}b^{-1}$, and a **2-sphere** can be described by the labelling scheme $v = aa^{-1}$. See figure 6.



2-sphere
$v = aa^{-1}$

Torus
$u = bcb^{-1}c^{-1}$

Projective plane
$w = dd$

Figure 6: Three fundamental surfaces

As we mentioned above, part of the classification theorem is bringing polygons to normal form where classification is then trivial. Notice how each normal form is in correspondence with the three fundamental surfaces.

Definition 17. A polygon in **normal form** has a labelling scheme in one of three forms:

- $\star$ $v = aa^{-1}$ (see figure 6)
- $\star$ $u = aba^{-1}b^{-1}cdc^{-1}d^{-1}efe^{-1}f^{-1}\cdots$ with length at least 4. (see figure 7)
- $\star$ $w = aabbcc\cdots$ with length of at least 2. (see figure 7).

5.2 Distinct fundamental surfaces

We need two more lemmas for our proof of The Classification Theorem. These lemmas that will help us to show that a connected sum of n-tori is distinct from the connected sum of m-projective planes.

Lemma 4. *The connected sum of n tori is orientable with Euler characteristic $2 - 2n$.*

Proof. The torus T is orientable with $\chi(T) = 0$. By lemma 2 a connected sum of n tori will remain orientable. By lemma 3,

$$\chi(\underbrace{T\#T\#...\#T}_{n}) = n(0) - (n-1)(2) = 2 - 2n. \tag{7}$$

□

Lemma 5. *The connected sum of $n > 0$ projective planes is non-orientable with Euler characteristic $2 - n$.*

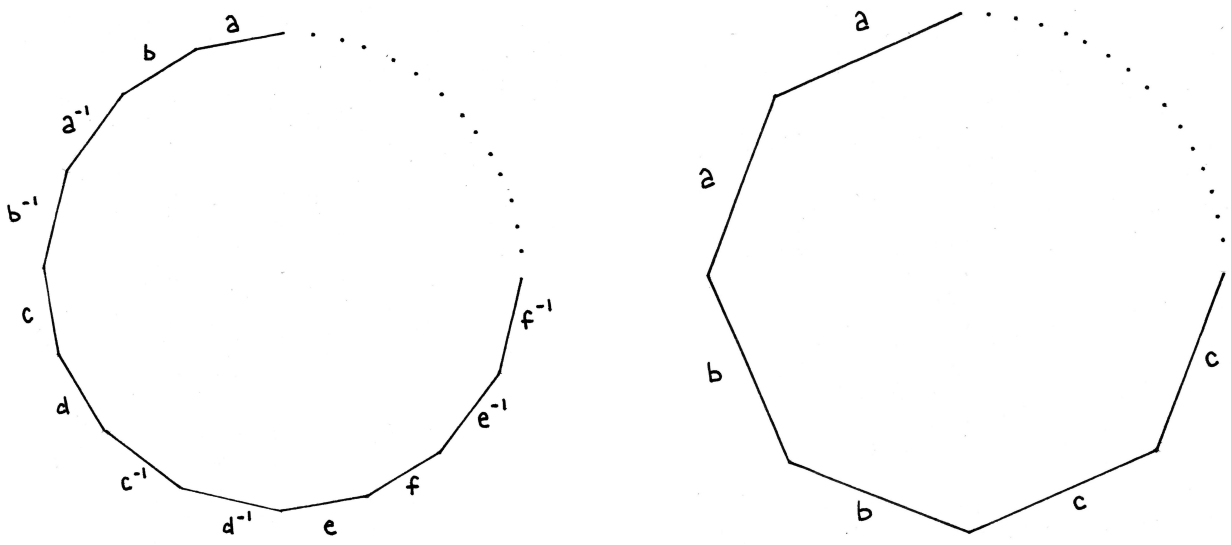


Figure 7: Polygons in normal form

Proof. The projective plane P is non-orientable with $\chi(P) = 1$. By lemma 2 a connected sum of $n > 0$ projective planes will remain non-orientable. We specify $n > 0$ since if $n = 0$ we have no projective planes and no Möbius strips so the surface will be orientable.By lemma 3,

$$\chi(\underbrace{P\#P\#...\#P}_{n}) = n(1) - (n-1)(2) = 2 - n. \tag{8}$$

□

Now note that $2 - 2n = 2 - n$ only for $n = 0$. The $n = 0$ case represents the 2-sphere, i.e. a sphere with no holes and no twists. Therefore $\mathcal{T}2$, $\mathcal{P}2$ and $\mathcal{S}2$ are all distinct.

5.3 Proof of the classification theorem

Proof. By lemmas 4 and 5, Euler characteristic and orientability together are sufficient to distinguish between a 2-sphere, a connected sum of tori, and a connected sum of projective planes.

Form a single polygon with edges identified in pairs from a triangulation $\mathcal{T}$ of the compact connected surface.

Suppose the polygon of our surface has only two edges. They are either arranged in the same direction (aa) or in opposite directions (aa^{-1}). In the first case we have the projective plane, and in the second case we have a 2-sphere.

Otherwise, suppose our polygon has more than two edges. We will provide a sequence of steps that will transform this polygon into a normal from where we can identify it up to homeomorphism as one of the three standard surfaces.

The first step is to cancel any adjacent edges of opposite directions (aa^{-1}) by gluing them together. We obtain a polygon with two fewer edges. This may bring us to a polygon with only two edges, in which case we refer back and identify it as a projective plane or a 2-sphere.

Next, identify all vertices to a single equivalence class if they are not already. This can be done by the following procedure: if an edge a has endpoints P and Q in different equivalence classes, where b is adjacent to a with endpoints A and R, make a cut from P to R, labelling this new edge c. We cut off the triangle PQR and glue it back to the polygon along the other edge a. This will reduce the number of vertices in equivalence class Q by one. We continue this process until Q has only one member, at which time we can close it up and eliminate Q altogether. We similarly eliminate all other equivalence classes until only one remains. Now that there is only one vertex present, it will continue to be the only vertex present. All further cut and past operations will not generate any new vertices. Also note that

this process may bring us to a polygon with only two edges, in which case we refer again may identify it as a projective plane or a 2-sphere. All further operations will preserve the number of edges present.

The third step is to bring all like-oriented edges into adjacent pairs. That is, $(...a...a...)$ will become $(...cc...)$. We can perform this operation by cutting from the tip of one a to the tip of the other a, labelling this cut c. Then gluing along a will bring the like-oriented c edges next to one another. Notice that any adjacent pairs already present in the polygon will remain so during this operation.

Now if any pairs of oppositely oriented edges $(...a...a^{-1}...)$ remain, they must occur in 'crossed pairs' $(...a...b...a^{-1}...b^{-1})$. Why? Suppose that $(...a...a^{-1}...)$ is not separated by any other pair of oppositely oriented edges. Let α be the sequence of edges between a and a^{-1} and let β be the sequence of edges between a^{-1} and a so that $a\alpha a^{-1}\beta = ...a...a^{-1}...$. Now each edge in α is identified with another edge in α and each edge in β is identified with another edge in β. So α and β are valid polygonal representations. Now if we were to paste along a, we would create a tube with the surface α on one end and the surface β on the other. But this implies that the tip of a is not identified with the tail of a, contrary to our previously collecting all vertices to a single equivalence class. therefore, all pairs of oppositely oriented edges occur in crossed pairs.

The fourth step is to bring all crossed pairs $(...a...b...a^{-1}...b^{-1})$ into adjacency $(...cdc^{-1}d^{-1}...)$ by the following procedure. First cut from the tip of a to the tip of a, labelling this new edge c. Now paste along b. This will give the polygon $(...aca^{-1}...c^{-1}...)$. Now cut from the tip of c to the tip of c, labelling this edge d, and glue along a. This will give $(...cdc^{-1}d^{-1}...)$. Again any adjacent edges of like-orientation will remain adjacent through this process. If there are no edges of like orientation after we collect crossed pairs, the surface is isomorphic to the connected sum of n tori where n is the number of crossed pairs.

Otherwise, if there are edges of like-orientation, we must transform $(...aa...bcb^{-1}c^{-1}...)$ into $(...eeffgg...)$. Cut from where a adjoins to a to where c adjoins to b^{-1}, labelling this edge d, then paste along a. This gives $(...dcbdbc...)$. Finally, perform the operation of collecting these like-oriented edges into adjacency. We must do this in the order b, c, d or c, b, d (see section 10.1). Now if the polygon contains any like-oriented edges, they are all like-oriented edges. We know such a surface is the connected sum of n projective planes where n is the number of edge pairs. □

6 Example: classification by cut and paste operations

Now we present a full example of how to use cut and paste operations to classify a surface from arbitrary polygon representation. Take surface represented by the labelling scheme $acbffc^{-1}a^{-1}b^{-1}gg$.

Step 1: Note that not all vertices are in the same equivalence class. The vertex at the ac intersection is not identified the other vertices. To fix this, we cut from the tail of a to the tip of c creating a new edge e. We then paste the triangle eac back along the sides $c^{-1}a^{-1}$. This buries the second vertex class in the interior of our polygon. This leaves us with the octagon $ebffe^{-1}b^{-1}gg$.

Step 2: At this point all like oriented edges are adjacent. So our next step is to bring the crossed pair $eb...e^{-1}b^{-1}...$ into adjacency. We make a cut from the tip of one e to the tip of the other e creating a new edge h. We then paste the square $hbff$ back along the side b^{-1}. This gives us the octagon $ehe^{-1}ffh^{-1}gg$.

Step 3: We are continuing to bring what was the crossed pair e, b into adjacency. We cut from the tip of one h to the other h creating a new edge k. We then paste the square $ke^{-1}ff$ back along the edge e. This gives us the octagon $ffk^{-1}hkh^{-1}gg$. Now the crossed pair is normalised.

Step 4: While the crossed pair is normalised, we have edges of like-orientation present. So we must transform the crossed pair into two like-oriented pairs. We start by cutting from the tip of the first f (at the ff vertex) to the tip of h (at the hk vertex) creating a new edge m. We now take the square $mfk^{-1}h$ and paste it back along f giving the octagon $mh^{-1}kmkh^{-1}gg$.

Step 5: Now we must bring the three pairs of like-oriented edges (m, h, and k) into adjacency, being careful to do m last. We will normalise the pair k first. To do this we cut from the tip of one k to the tip of the other k creating the edge p. We paste the triangle pmk back along the edge k giving the octagon $mhm^{-1}pphgg$.

Step 6: Now we will normalise the like-oriented pair h. Cut from the tip of one h to the tip of the other h creating the edge q. Now paste the pentagon $qm^{-1}pph$ back along the edge h giving the octagon $mppmqqgg$.

Step 7: Finally we can normalise the like-oriented pair m. Cut from the tip of one m to the tip of the other m creating an edge r. Take the square $rppm$ and paste it back along the edge m. This gives us the octagon $pprrqqgg$ which is in normal form.

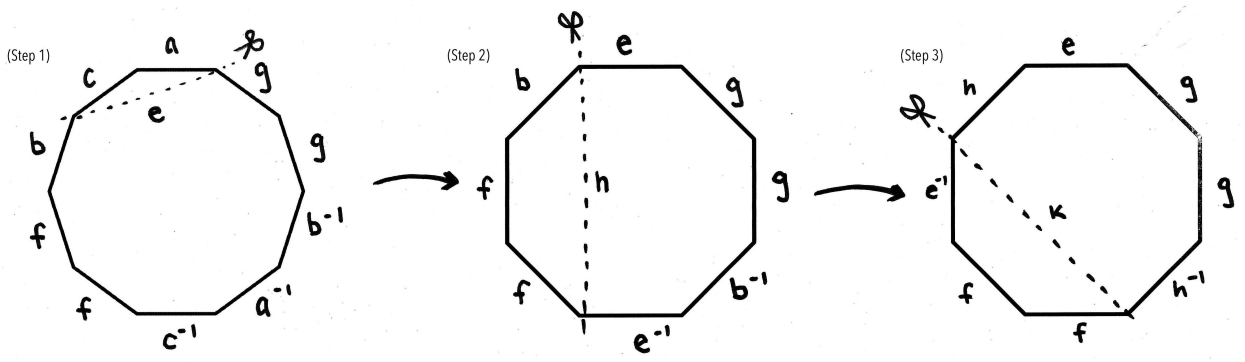


Figure 8: Steps 1-3 of Cut and Paste classification of polygon $acbffc^{-1}a^{-1}b^{-1}gg$

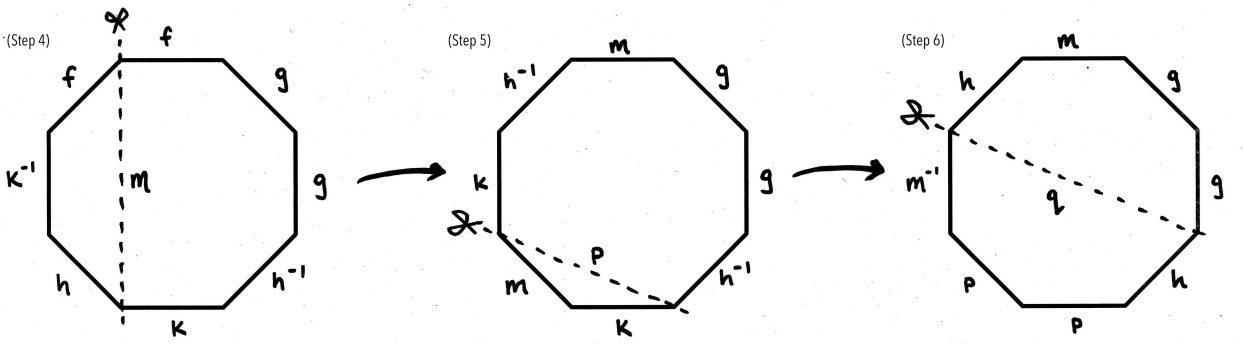


Figure 9: Steps 4-6 of Cut and Paste classification of polygon $acbffc^{-1}a^{-1}b^{-1}gg$

Step 8: A polygon in the form $pprrqqgg$ is the connected sum of four projective planes $P\#P\#P\#P$.

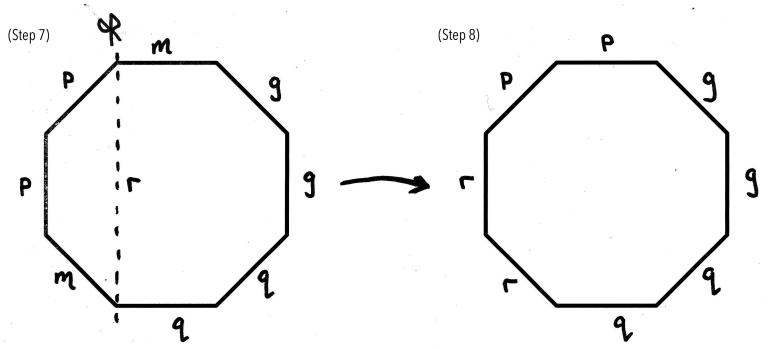


Figure 10: Steps 7, 8 of Cut and Paste classification of polygon $acbffc^{-1}a^{-1}b^{-1}gg$

7 Programming surface classification

The remainder of this paper is dedicated to presenting the program *Surface Classification* we have developed for determining surface classification from polygon representation.

7.1 Program overview

Surface Classification is a web based interface and uses JavaScript for underlying operations Given an edge list representation of a compact connected surface, the user can experiment with cut and paste operations through the drawing environment, or chose to find an immediate classification through the *classify* function.

The interface has a minimal number of interactive elements:

- An input text field for entering a polygon's string representation
- A drawing window for visualising cut and paste operations
- The *generate* button for drawing the entered string representation as a polygon.
- The *classify* button for finding an immediate classification, bypassing cut and paste operations.
- The *reset* button to erase the current drawing.
- A message field for reporting classification, cut and paste results as string representation, and any errors; A *clear* button to erase the message field.
- The *cut*, *paste* and *flip* buttons to perform cut and paste operations, further explained below.
- The *hint* button for reporting the next cut and paste operation to reach normal form
- The *undo* and *redo* buttons to undo and redo cut and paste operations in the drawing window.
- The *note* button and adjacent input field for annotating the polygon in the drawing window.

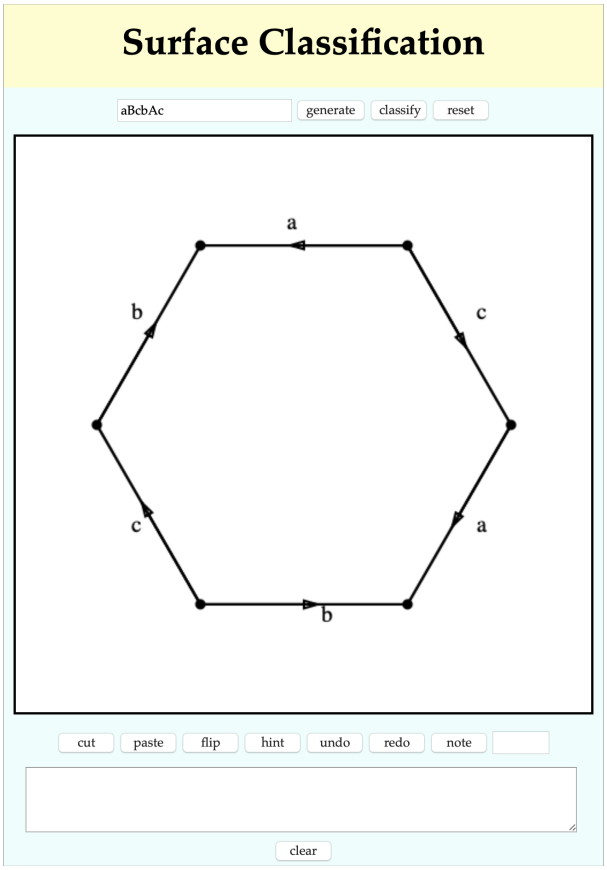


Figure 11: *Surface Classification* interface

To use the *Surface Classification* program, the user should start by entering a string representation into the text field. A string representation of a surface is a list of alphabetic characters where capitalisation of a character denotes an inverse edge, and each letter (in capital or lower case form) occurs exactly twice. Classification can then be carried out in two ways. The user can press the *classify* button and

the final classification will immediately be reported in the message window along with the sequence of cut and paste steps to reach this classification. Otherwise, the user can press the *generate* button and a polygon corresponding to the user's input will be appear in the drawing window. In either case, if the input text does not correspond to a valid polygon representation, an appropriate error will be reported in the message window and neither drawing nor classification will occur.

If the user elected to draw the polygon, the cut, paste and invert operations can be carried out in the drawing window as follows:

To perform a cut, select two vertices (they will turn blue) and press the *cut* button. A new edge will be created between the two selected vertices and will be given the next available label. The user can never select more than two vertices at a time; a vertex can be de-selected by clicking it again. Furthermore, the user cannot perform a cut between two adjacent vertices; since this operation corresponds to the trivial cutting off of an S2, it has not been implemented.

The *cut* button leads us to distinguish between two states: one where the polygon is whole, one where it is separated). No further cuts can be made while the polygon is in a separated state. Furthermore, this state influences how the *paste* and *flip* buttons operate.

To perform a paste, select two edges (they will turn red) and press the *paste* button. When the polygon is in its 'whole' state, the *paste* button can only be used to paste together pairs of adjacent inverse edges. When the polygon is in its 'separated' state, the *paste* button can only be used to paste together a pair of edges from separate segments and of opposite orientation.

The *flip* button allows the user to reverse the orientation of a polygon or cut polygon segment. When the polygon is in its 'whole' state, the user can simply press the *flip* button and the polygon will be redrawn with reversed orientation. When the polygon is in its 'separated' state, the user can select a segment to flip by clicking its interior (a purple dot will appear). Once this choice has been made, clicking the *flip* button will redraw only that segment with reversed orientation.

These three buttons correspond to the only operations needed to take a polygon to normal form. One further button has been included to allow the user to make annotations in the drawing window. Clicking the *note* button will initiate this annotation mode until it is clicked again. While annotating, whatever text is present in the *note* input field will be placed within the drawing window where the user clicks. Annotations can be especially helpful for tracking vertex equivalence classes. However, once the user performs any cut, paste or flip operation, the annotations will disappear.

Finally, while in drawing mode, pressing the *hint* button will display the recommended next step in bringing the polygon into normal form for classification. These hints are non-specific, such as 'cancel adjacent inverse edges' or 'bring to a single vertex equivalence class' or 'collect crossed pairs'. The task of determining which edges these hints are referring to is left up to the user.

7.2 Programming cut & paste operations

The follow section will outline how we represent polygons in the underlying code, and how the cut, paste and invert operations are implemented in our program.

7.2.1 Polygon representation

Many programmers will be familiar with the concept of circular linked lists. A circular linked list is an ordered set of nodes such that each node has a predecessor and a successor node. In a simple implementation of a circular list, each node has a pointer to its successor node [2]. The circular list itself keeps track of only one node, knows as the head. The list is then formed by following the successor line starting at the head; it is circular in that eventually we will reach a node whose successor is the head again.

We chose to build our own circular list construction: the *CircList* class, see code fragment 1. Each node in a *CircList* is intended to be an instance of the *Edge* class. An *Edge* has 3 properties: a pointer to its successor node, a lower-case letter denoting its label, and a boolean tag denoting if the edge is inverse or not. A *CircList* of *Edges* then represents a polygon. We opted for *CircList*s to also keep track of their length and the set of labels used by the nodes in the list. The head node of a circular list as it relates to a polygon is arbitrary. We will build our lists from a word representation of the polygon, and thus the head node will represent the first letter in the word.

[2]There are several variations on circular lists. In some constructions, each node keeps track of its predecessor or in the case of a doubly-linked circular list each node knows both its predecessor and successor nodes. The former variation is no different to a successor implementation. The latter has the advantage that traversal can operate in two directions, but subsequently the insertion and deletion process more involved.

Our circular list implementation provides two methods for adding nodes: *append* and *prepend*. Both methods take the new node to add as an argument. The main distinction is that *append* inserts the new node such that its successor is the head, whereas *prepend* inserts the new node at the new head such that the old head is the successor of the new node. While the *prepend* method is not necessary, it is very helpful in the process of inverting/flipping over polygons. Our circular lists have no delete or remove methods. If ever we need to remove a node, we simply build up a new circular list leaving out the node to be removed. Finally, we have the *buildEdges* method which takes a string as input and returns a corresponding circular list.

Now we will describe how the cut, paste, and invert operations work on these circular list polygon representations.

7.2.2 Function: paste

The *paste* function takes three arguments: two circular lists to paste together, and a single letter edge label along which to paste. Denote the two circular lists to paste by e_1 and e_2, and denote the label by λ. First, we makes a new circular list e. This will be built up to the polygon resulting from the paste operation. Traverse through e_1, pushing each node onto e, until we reach the node in e_1 with label λ. Save the successor of this node as our position in the traversal of e_1. Then traverse through e_2 until we reach the other node with label λ. Starting with the successor to this node, push nodes from e_2 onto e until we return to the node with label λ. Finally, continue the traversal of e_1 and push nodes from e_1 onto e until we return to label λ again. The *paste* function then returns the circular list e. See code fragment 2.

7.2.3 Function: cut

The *cut* function also takes three arguments: a circular list e to cut, and indices to cut from and to (denote by i and j respectively). In general, we should avoid indices in circular lists as it somewhat deconstructs the circular property. However, we found that in the case of cutting, indices were necessary because nodes in a list are not necessarily unique. The indices used for the cut operation are measured from the head of the list. To cut from 0 would be to start a cut at the vertex between the head edge and its predecessor. Likewise to cut to 2 would be to finish a cut at the vertex between the head's successor and its successor. The process of determining these indices relies on whether the cut operation is used in the classification algorithm or in the graphical interface.

First, the program will check that $j < i$. If not, i and j are switched so this inequality holds [3]. Then, the *cut* function makes two new circular lists e_1 and e_2 that will correspond to the result of the cut, and a new label λ that will mark the newly cut edge. Whenever a new label is needed, a sub-method is called that takes the set of labels in a given circular list (our lists of edges keep track of this property) and returns the first letter alphabetically that does not occur in the set.

Starting with the head element of e, iterate through e until we reach index i. Then, create a new edge with label λ and push it onto e_1. Starting with index i in the list e, push edges onto e_1 until reaching the index j. Now e_1 represents one of the cut segments. We build up e_2 through a similar process. Start by pushing a new edge with label λ onto e_2. Then, starting with index j, push edges from e onto e_2 until we reach the index i again. Now e_2 represents the second of the cut segments. The *cut* function returns a length two list containing e_1 and e_2. See code fragment 3.

7.2.4 Function: invert

The *invert* function has the simplest implementation of the three main operations. The *invert* function takes a single argument: the circular list of edges e to invert. The program begins by creating a new circular list *inv*. Starting with the head element of e, negate the inverse tag of each element in e and prepend them to the list *inv*. Then, since prepend adds to the front of the list, *inv* is a copy of e in reverse order where each edge has a reversed direction. Finally, the *invert* function returns this new list *inv*. See code fragment 4.

[3] This does prevent the user from choosing the direction of a cut, but the direction of two edges with the same label is arbitrary and this simplification allows for much more compact code.

8 Classification algorithm

In this section, we will present the underlying classification algorithm triggered by the *classify* button. This algorithm draws heavily from the proof of The Classification Theorem, and is a terminable algorithm.

8.1 Function: classify

The *classify* function serves as a crank for the classification algorithm and various external sub-methods determine and perform the necessary cut and paste operations. The *classify* function takes a string of characters as an input. In regards to the html page, clicking the *classify* button sends the exact text in the input field to the *classify* function. First, we check if this string represents a valid polygon. This is done using a sub-method *validPolygon*. If we encounter an invalid polygon, this is reported to the message box on the html page and the *classify* function ends. Otherwise, we use the function *buildEdges* to create a *CircList* of *Edge*s.

Now we can properly begin the classification process. Start by calling the function *simplify*; this will cancel any edges of the form $...aa^{-1}...$ and will write a final classification of $\mathcal{S}2$ or $\mathcal{P}2$ to the html window if our polygon has only length 2. Next, call the *collectVertices* function to bring the polygon down to a single vertex equivalence class. This can cause edges of the form $...aa^{-1}...$ to resurface, so follow up by calling *simplify* again. The next step is to transform edges of the form $...a...a...$ into $...aa....$ This is done using the *collectLikeOriented* function. Again, call *simplify* before proceeding to the next step: calling the *collectCrossedPair* function. This will transform all edges of the form $...a...b...a^{-1}...b^{-1}...$ into $...cdc^{-1}d^{-1}....$

At this point the polygon only has edges of the form $...aa...$ and $...bcb^{-1}c^{-1}....$ Now use the function *countP2* to count the number of projective planes p in the polygon (where $...aa...$ counts as one projective plane). Then, if $p > 0$, and the polygon has n sides, we know that the number of tori $T = \frac{n-2p}{4}$. Following the rule that in the presence of a projective plane, a torus translates to two additional projective planes, we now know our total number of projective planes is $p + 2T$. Otherwise, if the number of projective planes is zero, the number of tori is simply $\frac{n}{4}$. The final classification is reported to the html window and the *classify* function terminates. See code fragment 5

8.2 Function: validPolygon

Th function *validPolygon* takes a string of characters α as input and will return true if this string represents a valid polygon input, and returns false otherwise. First, if α has odd length or is an empty string the function returns false right away. Otherwise, it uses a regular expression to ensure our string only uses alphabetic characters. If α does not match the regular expression '^ [a-zA-Z]+$' it returns false. Finally, we need to check that every letter used in the string is used twice. Create a map of letters in α to their number of occurrences in α. If the count for any letter is not exactly 2, the function returns false. If α has passed all the above checks, it is a valid polygon representation and the function returns true.

8.3 Function: simplify

Between each step in the surface classification algorithm, we need to make sure that our polygon has no edges of the form $...aa^{-1}....$ Furthermore, if our polygon has been reduced to a 2-gon we can move to a final classification as S2 or $\mathbb{R}$P2. We created the *simplify* to carry out these two operations.

The function *simplify* takes a *CircList* e as input. First, it calls the function *cancelAdjInverse* on e. This submethod will return a circular list e' with no adjacent inverse pairs. If e' has length 2, the final classification of e' is written to the html window and the function returns null. Determining the classification of a 2-gon as $\mathcal{S}2$ or $\mathcal{P}2$ involves checking if the two edges or of like or opposite orientation. Otherwise, if e' has length greater than 2, *simplify* returns e'.

8.3.1 Function: cancelAdjInvs

The function *cancelAdjInvs* takes a *CircList* of *Edge*s e as input. It begins by retrieving the set of edges labels from e and then iterates through this set. For each label, it calls the function *areAdjInv*. If this function returns true, and e has length greater than 2, we perform the paste operation on e along the label λ. Then, so as to capture nested adjacent inverse pairs such as $...abb^{-1}a^{-1}...$, the function *cancelAdjInvs* calls itself again on the now modified circular list e. The function will terminate and return the circular

list e when either there are no more adjacent inverse pairs, or the number of edges has decreased to 2. See code fragment 6.

8.3.2 Function: areAdjInv

The *areAdjInv* function takes a *CircList* of *Edge*s e and a single label λ, and checks if the edges in e with label λ are of the form $...\lambda\lambda^{-1}...$. Starting with the head element, traverse through e until we reach an occurrence of the label λ. If the very next element also has label λ and is of opposite orientation, the function returns true. Otherwise, continue to iterate through e until the second occurrence of λ. Then perform the same check: if the very next element also has label λ and is of opposite orientation, return true; otherwise, return false. This method could be greatly simplified by a circular list in which nodes know both their successor and predecessor. However, the small loss in compactness here does not warrant this implementation that would make the more commonly performed insertion operations more complex. See code fragment 6.

8.4 Function: collectVertices

As in the proof of The Classification Theorem, the first major step is to collect all the polygon's vertices into a single equivalence class. This is performed using the function *collectVertices*. This function takes a circular list of edges e as input. First, it calls the function *buildVertices* on e which returns a list v of *Vertex*s. A *Vertex* object has three properties: *left* and *right* record the *Edge*s to the left and right of the *Vertex* in the polygon; the property *eqClass* is an integer denoting which equivalence class the *Vertex* belongs to. The *buildVertices*operation leaves the *eqClass* property as null. The *collectVertices* function then calls the function *makeEqClasses* on v. This returns the number of equivalence classes n in e, as well as a modified list of vertices v' where the *eqClass* property is no longer null. If $n = 1$ the function returns e without modification. Otherwise, we proceed to the process of collecting vertices to a single equivalence class.

Given a list of vertices v with marked equivalence classes and a corresponding circular list of edges e, the function 'collects' vertices in the following way. Traverse v until the first vertex with $eqClass = 0$ and record this index as i. Then traverse to the first vertex with $eqClass \neq 0$, and remember this *eqClass* as eq. Finally, traverse to the second vertex with $eqClass = 0$ and record this index as j. Now, perform a cut on e from i to j. The *cut* functionreturns two new *CircList*s e_1 and e_2. We wish to paste these two segments back together along an edge label with some specific properties: (1) as with any paste, the label must be present in both e_1 and in e_2; (2) one of this edge's two associated vertices must be of equivalence class eq, (3) this edge cannot be the one we just now created with our cut. This edge label λ is found using the sub-method *findPasteLabel*. Finally, paste e_1 to e_2 along λ creating e'. This will have decreased the number of vertices in equivalence class eq by at least one. The problem now is that the list of vertices corresponds to e, and not to e'. Instead of attempting to adjust the vertex list to match e', we will just start the process over. Perform *cancelAdjInvs* on e' and then call *collectVertices* again. Eventually the number of vertex equivalence classes will have decreased to one and the the function will return e. See code fragment 7.

8.4.1 Function: makeEqClasses

The function *makeEqClasses* takes the list of vertices v as input. It will use several helper functions to recursively mark out the vertex equivalence classes. First, define variables eq and i that initialise to 0. Then call the *mark* function on v and i. This will return a new v where all vertices in the same equivalence class as the vertex in position i have been marked as such. Then increment the value eq by 1 and update the index i using the *nextToMark* function. The *nextToMark* function searches through the list of vertices until it finds one whose eqClass property is null, it will then return the index of this vertex, or -1 if all have been marked. Hence, once i has the value -1 the *makeEqClasses* function will return the list of vertices v and the number of equivalence classes eq. See code fragment 8.

8.4.2 Function: mark

The *mark* function takes a list of vertices v, a starting index i and an integer eq denoting the equivalence class we are working in. Let x be the vertex at index i in v. First, mark x as being in equivalence class eq. Then, mark all equivalence classes that follow directly from the equivalence class of x.

Recall that a *Vertex* knows the *Edge*s to its left and right. The remainder of the *mark* function is split up into two cases to deal with the left and right edges.

If the *Edge* l to the left of x is an inverse, then (1) the vertices where l is on the left and l is not an inverse and (2) where l is on the right and l is an inverse, are also in equivalence class eq. Search through the vertex list looking for any such vertices that have not already been marked. Then recurse: let v be the result of calling *mark* with arguments v, i and eq. Note that since we only recurse when a vertex is not already marked, this process will eventually terminate.

Otherwise, if the *Edge* l to left of x is not an inverse, then (1) the vertices where l is on the left and l is an inverse and (2) where l is on the right and l is not an inverse, are also in equivalence class eq. We perform the same recursive process on all such vertices.

Now consider the *Edge* r to the right of x. If r is an inverse, then (1) the vertices where r is on the left and is an inverse and (2) where r is on the right and is not an inverse are also in equivalence class eq and recursively mark them as such. Likewise, if r is not an inverse, then (1) the vertices where r is on the left and is not an inverse and (2) where r occurs on the right and is an inverse are also in equivalence class eq. Again, recurse and mark these vertices using the function *mark*.

Eventually the recursion of will terminate having found and marked all the vertices in equivalence class eq. The function *mark* returns the list of vertices v back to *makeEqClasses*. See code fragment 9.

8.4.3 Function: findPasteLabel

The function *findPasteLabel* takes two *CircList*s e_1 and e_2, a list of vertices v and an integer eq denoting an equivalence class. First, create a new set E. Traverse through v and if a vertex has equivalence class eq, add the label of both its left and right edges to the set E. Now E is the set of all edge labels incident to vertices in equivalence class eq.

Since the second criteria for a paste label was that it not be the edge just created with a cut, we will disregard the head elements of both e_1 and e_2. Begin traversing e_1 starting at the successor to the head. For each *Edge* in e_1 perform a traversal of e_2 starting at the successor to its head. In this inner traversal, check if the two edges have same label, and check if this label is in the set E; this satisfies conditions (1) and (3) for our desired paste label. If so, return this label. Otherwise, continue the double traversal. See code fragment 11. Note that there is always an appropriate edge label to paste along, otherwise there would not be multiple equivalence classes.

8.5 Function: collectLikeOriented

At this time, the polygon should have exactly one equivalence class of vertices. The next task is to bring all pairs of edges of like orientation into adjacency. For example, $...a...a...$ should become $...bb....$ The function *collectLikeOriented* takes a *CircList* e as an argument. For each edge label used in e, we want to check if these edges have the same orientation and are non-adjacent. Start by saving the head element of e into a variable $left$ and initialise a counter variable i to 0. Then, begin a traversal by setting a variable $right$ to the successor of $left$, and set a second counter j equal to $i + 1$. We will need these counters if we want to make a cut. Then, while $left$ and $right$ have different labels, set $left$ equal to its successor. Once $left$ and $right$ have the same label, check if they have the same orientation and are not already adjacent. If so, perform a cut from i to j and then paste the cut segments along the common label giving e'. Since the order of *Edge*s in e has changed, restart by calling *collectLikeOriented* on e'. Otherwise, if $left$ and $right$ were either of opposite orientation or already adjacent, set $left$ equal to its successor and if $left$ is not yet equal to the head element of e, continue the traversal. Finally, return e. See code fragment 10.

8.6 Function: collectCrossedPair

Perhaps the most involved step in the classification process is that of collecting crossed pairs. The function *collectCrossedPairs* relies on the function *findCrossedPair* to return the pair of labels, say a, b that make up a crossed pair such as $...a...b...A...B$. The work of transforming $...a...b...A...B$ into $...cdCD...$ is then done by *collectCrossedPair*. If *findCrossedPair* finds no crossed pair, it will return an empty list, and the loop in *collectCrossedPair* terminates. The function *collectCrossedPair* begins by calling *findCrossedPair*; we will save this in the variable $pair = (a, b)$. Now start a counter n at 0. Unfortunately, our function cannot distinguish between crossed pairs that are and are not already collected. This counter n will help us to ensure we do not collect the same crossed pair twice. Then, begin a loop with the condition that $pair$ has length greater than 0. Set x to be the head element of e and set an index counter $i = 0$.

Start by looking for the two indices of the edge labeled a in e. Then, until x has the label a and x is an inverse, set x to the successor of x and increment i modulo the length of e. At this point, save the value of i as $index\_a\_1$. Set x to be its successor. Then, once again, until x has the label a and x is not an inverse, set x to the successor of x and increment i modulo the length of e. Save the current value of i into $index\_a\_2$. Now cut e from a_index_1 to a_index_2. Save these two segments as a pair $pasteA$. Then paste these segments along the second edge label in the pair b. Save this new circular list as e'.

Next, we need to know the label of the edge created in the cut we just made. This can be found as the head element of either segment in $pasteA$; save this label as c. Now we wish to search for the indices of the edges with label c. Set x to be the head of e' and set i back to 0. As before, until x has the label of the c and x is an inverse, set x to the successor of x and increment i modulo the length of e'. Save this value of i as c_index_1 and set x to be its successor. Now we will search for the second index of c: until x has the label of the c and x is not an inverse, set x to the successor of x and increment i modulo the length of e'. Now save the value of i as c_index_2. Finally, cut e' from c_index_1 to c_index_2 and paste these segments along the label a.

Before repeating the loop, ensure the pasted segments are saved back in the variable e, increment the variable n, and pass e and n into *findCrossPair* setting *pair* to be the result. Eventually, there will be no further crossed pairs and our loop will end. At this time we simply return e. See code fragment 12.

8.6.1 Function: findCrossedPair

The function *findCrossedPair* takes a *CircList* of edges e and the pair counter n. Begin by building a list of the labels of all inverse pairs in e. Iterate through the set of labels in e, and for each label λ, check if the first occurrence of λ in e is of opposite orientation to the second occurrence of λ in e. If so, add λ to the list of inverse labels. Now, we will split this list of inverse labels, removing the first n labels – these inverse pairs will already have been collected. If the list of inverse labels is empty, there are no crossed pairs remaining so *findCrossedPair* returns an empty list.

Otherwise, let a be the first label in the list of inverse pairs. We need to find the appropriate crossed pair partner b for a. First, find the two indices i and j of the label a in e and build up two lists $left$ and $right$ such that left has edges from index i up to (not including index j and $right$ has edges from index j up to (not including) index i. We need to find a label from the list of inverse pairs that occurs once in the list $left$ and once in $right$. Begin by iterating through the list of inverse labels (skipping a). Let λ denote the current label in the iteration. First, check for the presence of λ in the list $left$ and then check for λ in the list $right$. If λ is found, *findCrossedPair* terminates and returns the pair a, λ. Otherwise it continues the loop. See code fragment 13. As the proof of The Classification Theorem demonstrates, all $...a...a^{-1}...$ edges must occur in crossed pairs; hence this final loop will always find an appropriate λ.

8.7 Function: countP2

The function *countP2* takes the *CircList* of edges e and returns an integer as output. It starts by initialising a counter i to 0 and then begins a traversal of e. For each edge x in e, if x and its successor have the same label and are not inverses, then we increment i. When the successor of x is the successor of the head element of e, *countP2* terminates and returns i. See code fragment 14.

This completes the collection of functions we use to implement the surface classification algorithm. JavaScript code fragments for the majority of the functions discussed above are given in the addendum, section 10.2.

9 Discussion

The *Surface Classification* program was developed over the course of a few months, and there are several limitations to resolve and features we would like to have implemented but did not have sufficient time to do so.

Perhaps main limitation we see in the *Surface Classification* program is that only one cut can be made at a time. Much work is needed to allow for arbitrary cutting and this could not happen in our time frame. Instead, when the user cuts, they must make a paste before making a second cut. In the future, it would be beneficial for a user to be able to make several cuts to a polygon at one time. In particular, this would help with collecting vertices to one equivalence class since all vertices could be cut off and collected in one go. Likewise, there is no option to cancel adjacent inverses in a single cut

segment. This is less limiting since thee adjacent inverse edges will remain after any paste, but would certainly be a useful operation for classifying surfaces.

The main feature we would like to implement in the future is continuous animation in the graphical interface. This would help the user to develop a clearer understanding of the cut and paste operations that are taking place. Currently, when two segments are pasted back together, we do not see the paste operation taking place and are instead jump straight to the resulting polygon. Visually, this is a large transition to make and for new users or those with little experience in cut and paste operations, it can be difficult to follow what has happened. With full animation implemented in the graphics window, the program could provide a full sequential animation of the cut and paste operations for classification rather than just a text description. Implementing such animation, however, would require a full reworking of the program from its current state.

While implementing the *Surface Classification* program, and in writing this paper, we came across several interesting, related questions. In particular, what are the smallest (in terms of number of edges) and simplest (in terms of fundamental group) polygons that require every step in The Classification Theorem in order to be classified using cut and paste operations? We came across this question while trying to develop an insightful example, see section 6. The best example we could find has string representation $acbfffc^{-1}a^{-1}b^{-1}gg$, and it does require every step of The Classification Theorem. Such examples are important because they are very useful test cases for programs such as ours. We are able to verify that every operation in the *Surface Classification* program performs correctly since the polygon $acbfffc^{-1}a^{-1}b^{-1}gg$ can be correctly classified as the connected sum of four projective planes.

Other interesting questions arose in considering test cases for our program. For example, should we consider the empty string a valid polygon representation? In one sense, we could achieve the empty string by cancelling the last remaining pair of edges of the form aa^{-1}. As an operation on surfaces, we are left with a point. A point is homeomorphic to S2, but it is not generally seen as a *surface* at all. In the end, we chose not to allow the empty string as an input polygon representation in the *Surface Classification* program for a few (less mathematical) reasons. If one enters the empty string, a point could appear in the drawing window, but there are then no interesting cut and paste operations to perform. Furthermore, we decided it would be confusing to allow a lack of input to be a valid input – a user may have pressed a button accidentally and not realise this has triggered the drawing window and classification process.

Certainly the *Surface Classification* program has limitations and room for further work, but it does perform the task we initially intended. That is, the *Surface Classification* program is able to classify compact connected surfaces from a string representation with the option to visualize the classification process through cut and paste operations. Even in its current state, this program will be a valuable learning tool for students and teachers of introductory topology courses.

10 Addendum

10.1 Collecting projective planes

We noted in our proof that collecting projective planes in a polygon with scheme $...dcbdbc...$ must be done in the order b, c, d or c, b, d. There are only six possible orders to collect 3 projecting planes in, we will show here why b, c, d and c, b, d work, and why the other four orders do not.

First, here are the cut and paste diagrams for b, c, d and c, b, d. They allow us to obtain a polygon of normal form with all identified edges adjacent.

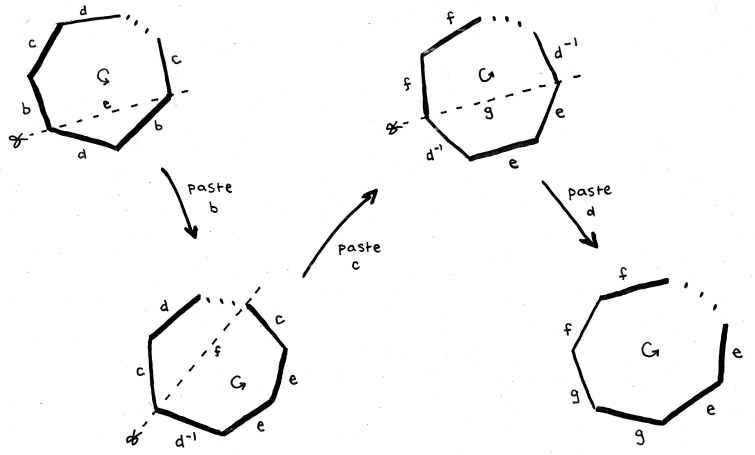


Figure 12: Cut and paste in order of b, c, d

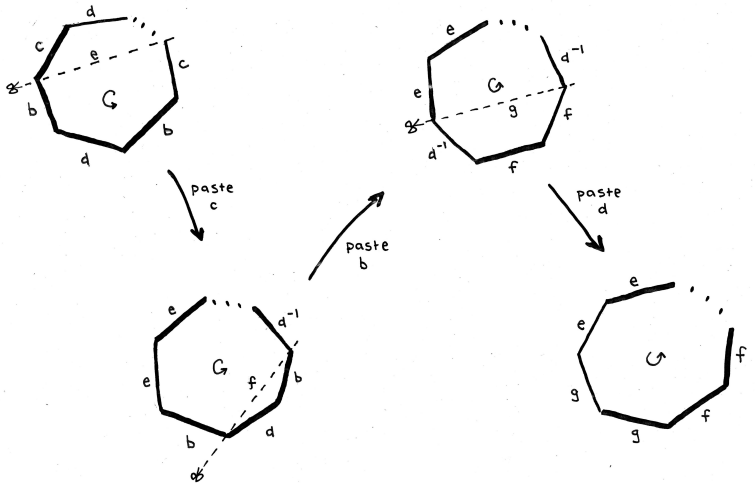
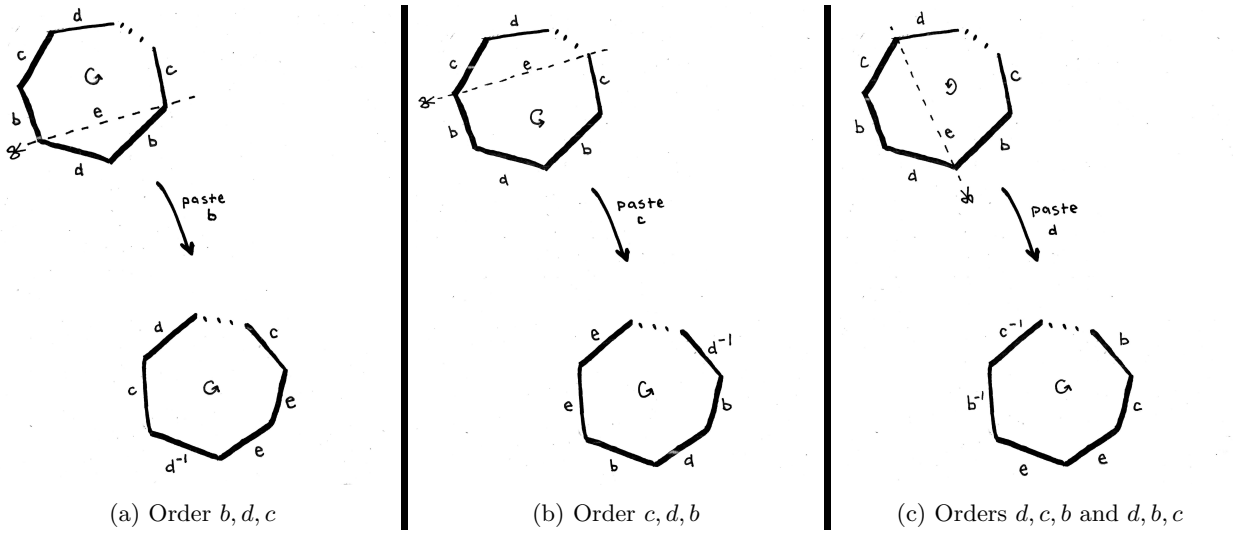


Figure 13: Cut and paste in order of c, b, d

Here are the cut and paste diagrams for what happens when we try the other four orders. In figure 14 a), we paste along b first. This gives us $...d...d^{-1}...$. When we try to make a cut f that brings d and d^{-1} into different polygons and then glue along d we obtain $...f...f^{-1}...$ so no progress has been made. Likewise, in figure 14 b) we paste along c first and also obtain $...d...d^{-1}...$ which has the same issues. Finally, in figure 14 c), we paste along d first. Now we cannot paste along b or along c next since we have both $...b...b^{-1}...$ and $...c...c^{-1}...$ so we encounter the same problems as before.

So in general, when we have a polygon of scheme $...dbcdcb...$ we must collect the projective plane d last or else we will revert to a form containing a torus.



(a) Order b, d, c (b) Order c, d, b (c) Orders d, c, b and d, b, c

Figure 14: Attempts to collect projective planes

10.2 JavaScript code fragments

```
class CircList{
  constructor(){
    this.length = 0;
    this.head = null;
    this.labels = new Set();
  }

  append(node){
    var temp = this.head;
    if (this.head !== null){
      while(temp.next !== this.head){
        temp = temp.next;
      }
      temp.next = node;
      node.next = this.head;
    }else{
      this.head = node;
      node.next = this.head;
    }
    this.length++;
    this.labels.add(node.label);
  }

  prePend(node){
    if(this.head === null){
      this.head = node;
      node.next = this.head;
    }else{
      node.next = this.head
      var temp = this.head;
      while(temp.next !== this.head){
        temp = temp.next;
      }
      temp.next = node;
      node.next = this.head;
      this.head = node;
    }
    this.length++;
    this.labels.add(node.label);
  }
```

```
}
```

Code fragment 1: CircList Class

```
function paste(e1, e2, label){
  var e = new CircList();
  var temp1 = e1.head;
  while(temp1.label !== label){
    e.push(new Edge(temp1.label, temp1.inverse));
    temp1 = temp1.next;
  }
  temp1 = temp1.next
  var temp2 = e2.head;
  while(temp2.label !== label){
    temp2 = temp2.next;
  }
  temp2 = temp2.next;
  while(temp2.label !== label){
    e.push(new Edge(temp2.label, temp2.inverse));
    temp2 = temp2.next;
  }
  while(temp1.label !== e1.head.label){
    e.push(new Edge(temp1.label, temp1.inverse));
    temp1 = temp1.next;
  }
  return e;
}
```

Code fragment 2: Paste operation

```
function cut(edges, i, j){
  if(j < i){
    var hold = i;
    i = j;
    j = hold;
  }
  e1 = new CircList();
  e2 = new CircList();
  var new_label = newLabel(edges.labels);
  var temp = edges.head
  var n = 0;
  while(n < i){
    temp = temp.next;
    n++;
  }
  e1.push(new Edge(new_label, false));
  while (n < j){
    e1.push(new Edge(temp.label, temp.inverse));
    temp = temp.next;
    n++;
  }
  e2.push(new Edge(new_label, true));
  while(i < edges.length){
    e2.push(new Edge(temp.label, temp.inverse));
    temp = temp.next;
    n++;
  }
  n = 0
  while(n < i){
    e2.push(new Edge(temp.label, temp.inverse));
    temp = temp.next;
    n++;
  }
  return [e1, e2];
}
```

Code fragment 3: Cut Operation

```
function invert(edges){
  var inv = new CircList();
  var temp = edges.head;
  inv.prePend(new Edge(temp.label, !temp.inverse));
  while (temp.next != edges.head){
    temp = temp.next;
    inv.prePend(new Edge(temp.label, !temp.inverse));
  }
  return inv;
}
```

Code fragment 4: Invert Operation

```
function classify(polygon){
  if(!validPolygon(polygon)){
    writeMessage("Invalid polygon " + polygon);
    return;
  }
  var edges = simplify(buildEdges(polygon));
  if(edges === null){
    return;
  }
  edges = collectVertices(edges);
  edges = simplify(edges);
  if(edges === null){
    return;
  }
  edges = collectLikeOriented(edges);
  edges = simplify(edges);
  if(edges === null){
    return;
  }
  edges = collectCrossedPair(edges);
  var numrp2 = countRP2(edges);
  if (numrp2 > 0){
    var numtori = (edges.length - (2*numrp2))/4;
    numrp2 += (numtori * 2);
    writeMessage(edges.string() + ": " + numrp2 + " P2");
  }else{
    var numtori = edges.length/4;
    if(numtori === 1){
      writeMessage(edges.string() + ": torus");
      return;
    }else{
      writeMessage(edges.string() + ": " + numtori + " tori");
      return;
    }
  }
}
```

Code fragment 5: The Classification Algorithm

```
function cancelAdjInvs(edges){
    var labelList = Array.from(edges.labels);
    var i = 0;
    var label;
    while(i < labelList.length){
      label = labelList[i];
      if(areAdjInv(edges, label) && edges.length > 2){
        edges = pasteAdjInv(edges, label)
        return cancelAdjInvs(edges);
      }
      i++;
    }
  return edges;
}

function areAdjInv(edges, label){
```

```
  var temp = edges.head;
  while(temp.label !== label){
    temp = temp.next;
  }
  if(temp.next.label === label && inverses(temp, temp.next)){
    return true;
  }else{
    temp = temp.next;
    while(temp.label !== label){
      temp = temp.next;
    }
    return (temp.next.label === label && inverses(temp, temp.next));
  }
}

function pasteAdjInv(edges, label){
  var e = new CircList();
  var temp = edges.head;
  do{
    if(temp.label !== label){
      e.push(new Edge(temp.label, temp.inverse));
    }
    temp = temp.next;
  }while(temp !== edges.head);
  return e;
}
```

Code fragment 6: The Adjacent-inverse functions

```
function collectVertices (edges){
  var v = makeEqClasses(buildVertices(edges));
  var eqClasses = v.eqClass;
  var vertices = v.vertices;
  if(eqClasses === 1){
    return edges;
  }else{
    var len = vertices.length;
    for(var j = 0; j < len; j++){
      if (vertices[j%len].eqClass === 0){
        var index0 = j%len;
        break;
      }
    }
    for(var j = index0; j < len; j++){
      if (vertices[j%len].eqClass !== 0){
        var index1 = j%len;
        var eq = vertices[j%len].eqClass;
        break;
      }
    }
    for(var j = index1; j < len; j++){
      if (vertices[j%len].eqClass === 0){
        var index2 = j%len;
        break;
      }
    }
    var cutBits = cut(edges, index0, index2);
    var pasteLabel = findPasteLabel(cutBits[0], cutBits[1], vertices, eq);
    cutBits = prePaste(cutBits[0], cutBits[1], pasteLabel);
    edges = paste(cutBits[0], cutBits[1], pasteLabel);
    edges = cancelAdjInvs(edges);
    return collectVertices(edges);
   }
}
```

Code fragment 7: The *collectVertices* Function

```
function makeEqClasses(vertices){
```

```
  var eqClass = 0;
  var index = 0;
  do{
    vertices = mark(vertices, index, eqClass);
    eqClass++;
    index = nextToMark(vertices);
  }while(index !== -1);
  return {eqClass: eqClass, vertices: vertices};
}
```

Code fragment 8: The *makeEqClasses* Function

```
function mark(vertices, index, eqClass){
  var v = vertices[index];
  v.eqClass = eqClass;
  if(v.left.inverse){
    for(var i = 0; i < vertices.length; i++){
      var u = vertices[i];
      if(u.eqClass === null){
        if(u.left.label === v.left.label && u.left.inverse){
          vertices = mark(vertices, i, eqClass)
        }else if(u.right.label === v.left.label && !u.right.inverse){
          vertices = mark(vertices, i, eqClass)
        }
      }
    }
  }else{
      for(var i = 0; i < vertices.length; i++){
        var u = vertices[i];
        if(u.eqClass === null){
          if(u.left.label === v.left.label && !u.left.inverse){
            vertices = mark(vertices, i, eqClass);
          }else if(u.right.label === v.left.label && u.right.inverse){
            vertices = mark(vertices, i, eqClass);
          }
        }
      }
  }
  if(v.right.inverse){
    for(var i = 0; i < vertices.length; i++){
      var u = vertices[i];
      if(u.eqClass === null){
        if(u.right.label === v.right.label && u.right.inverse){
          vertices = mark(vertices, i, eqClass);
        }else if(u.left.label === v.right.label && !u.left.inverse){
          vertices = mark(vertices, i, eqClass);
        }
      }
    }
  }else{
    for(var i = 0; i < vertices.length; i++){
      var u = vertices[i];
      if(u.eqClass === null){
        if(u.right.label === v.right.label && !u.right.inverse){
          vertices = mark(vertices, i, eqClass);
        }else if(u.left.label === v.right.label && u.left.inverse){
          vertices = mark(vertices, i, eqClass);
        }
      }
    }
  }
  return vertices;
}
```

Code fragment 9: The *mark* Function

```
  function collectLikeOriented(edges){
  var left = edges.head;
```

```
  var i = 0;
  var right;
  var j;
  do{
    right = left.next;
    j = i + 1;
    while(right.label !== left.label){
      right = right.next;
      j++;
    }
    if(right.inverse === left.inverse &&
       right.next !== left && left.next !== right){
      var [e0, e1] = cut(edges, i, j);
      e1 = invert(e1);
      edges = paste(e0, e1, left.label);
      return collectLikeOriented(edges);
    }
    left = left.next;
    i++;
  }while(left !== edges.head);
  return edges;
}
```

Code fragment 10: The *collectLikeOriented* Function

```
function findPasteLabel(e0, e1, vertices, eq){
  var eq_edges = new Set();
  for(var i = 0; i < vertices.length; i++){
      var v = vertices[i];
      if(v.eqClass === eq){
        eq_edges.add(v.left.label);
        eq_edges.add(v.right.label);

      }
  }
  var temp0 = e0.head.next;
  var temp1;
  while(temp0 !== e0.head){
    temp1 = e1.head.next;
    while(temp1 !== e1.head){
      if(temp0.label === temp1.label && eq_edges.has(temp0.label)){
        return temp0.label;
      }
      temp1 = temp1.next;
    }
    temp0 = temp0.next;
  }
}
```

Code fragment 11: The *findPasteLabel* Function

```
function collectCrossedPair(edges){
  var pair = findCrossPair(edges, 0);
  var n = 0;
  while(pair.length > 0){
    var temp = edges.head;
    var i = 0;
    while(temp.label !== pair[0] || temp.inverse){
      temp = temp.next;
      i = (i + 1) % edges.length;
    }
    var a_index1 = i;
    temp = temp.next;
    while(temp.label !== pair[0] || !temp.inverse){
      temp = temp.next;
      i = (i + 1) % edges.length;
    }
    var a_index2 = (i + 2) % edges.length;
```

```
    var cutA = cut(edges, a_index1, a_index2);
    var pasteB = paste(cutA[0], cutA[1], pair[1]);
    var labelC = cutA[0].head.label;
    temp = pasteB.head;
    var i = 0;
    while(temp.label !== labelC || temp.inverse){
      temp = temp.next;
      i = (i + 1) % pasteB.length;
    }
    var c_index1 = i;
    temp = temp.next;
    while(temp.label !== labelC || !temp.inverse){
      temp = temp.next;
      i = (i + 1) % pasteB.length;
    }
    var c_index2 = (i + 2) % pasteB.length;
    var cutC = cut(pasteB, c_index1, c_index2);
    var pasteA = paste(cutC[0], cutC[1], pair[0]);
    edges = pasteA;
    n++;
    pair = findCrossPair(edges, 2*n);
  }
  return edges;
}
```

Code fragment 12: The *collectCrossedPair* Function

```
function findCrossPair(edges, n){
  var labelList = Array.from(edges.labels);
  var invs = []
  for(var i = 0; i < labelList.length; i++){
    var label = labelList[i];
    var temp = edges.head;
    while(temp.label !== label){
      temp = temp.next;
    }
    var inv1 = temp.inverse;
    temp = temp.next;
    while(temp.label !== label){
      temp = temp.next;
    }
    var inv2 = temp.inverse;
    if(inv1 !== inv2){
      invs.push(label);
    }
  }
  invs = invs.slice(n);
  if (invs.length === 0){
    return [];
  }
  temp = edges.head;
  var left = [];
  var right = [];
  while(temp.label !== invs[0]){
    temp = temp.next;
  }
  left.push(temp);
  temp = temp.next;
  while(temp.label !== invs[0]){
    left.push(temp);
    temp = temp.next;
  }
  right.push(temp);
  temp = temp.next;
  while(temp.label !== invs[0]){
    right.push(temp);
    temp = temp.next;
  }
```

```
  for(var j = 1; j < invs.length; j++){
    var inLeft = false;
    var inRight = false;
    for(var k = 1; k < left.length; k++){
      if (left[k].label === invs[j]){
        inLeft = true;
      }
    }
    for(var k = 1; k < right.length; k++){
      if (right[k].label === invs[j]){
        inRight = true;
      }
    }
    if (inRight && inLeft){
      return [invs[0], invs[j]];
    }
  }
}
```

Code fragment 13: The *findCrossPair* Function

```
function countP2(edges){
  var temp = edges.head;
  var numRp2 = 0;
  do{
    if (temp.next.label === temp.label && !inverses(temp.next, temp)){
      numRp2 ++;
    }
    temp = temp.next;
  }while(temp.next != edges.head.next);
  return numRp2;
}
\label{countP2}
```

Code fragment 14: The *countP2* Function

References

[1] Mark A. Armstrong, *Basic Topology*, 3rd edition ed., Springer-Verlag, 1990.

[2] George K. Francis and Jeffrey R. Weeks, *Conway's ZIP Proof*, American Mathematical Monthly **106** (1999), no. 5.

[3] W. Randolph Franklin, *Point Inclusion in Polygon Test*, November 2018.

[4] Allen Hatcher, *Algebraic Topology*, Cambridge University Press, 2002.

[5] Marc Lackenby, *Topology and Groups*, October 2016.

[6] William S. Massey, *A Basic Course in Algebraic Topology*, Springer-Verlag, 1991.

[7] Robert Messer and Philip Straffin, *Topology Now*, The Mathematical Association of America, 2006.

[8] James R. Munkres, *Topology*, Prentice Hall Inc, 2000.

[9] Seifert and Threlfall, *A Textbook of Topology*, Academic Press, INC., 1980.

[10] Michael Siff, *Introduction to Web Programming*, 2016.

[11] John Stillwell, *Geometry of Surfaces*, Springer, 1993.

[12] Agnes Szilard, *Introduction to Topology*, 2018.