5-2015

Encoding Knots

Rachael Keller

Encoding Knots

by

Rachael Keller

Undergraduate honors thesis under the direction of

Professor Oliver Dasbach

Department of Mathematics

Submitted to the LSU Honors College in partial fulfillment of
the Upper Division Honors Program.

May 2015

Louisiana State University
& Agricultural and Mechanical College
Baton Rouge, Louisiana

Acknowledgments

I would like to express my gratitude to Professor Oliver Dasbach for constantly encouraging me, persistently believing in me, always keeping me and my software updated with the latest, and for proposing this project. I would also like to thank Dr. Hartmut Kaiser for instigating, developing, and encouraging my studies in C++ and programming in general. Without these two professors, I would be a much less able student. Thank you.

This project has been generously funded by the Louisiana Experimental Program to Stimulate Competitive Research (LAEPSCoR) Supervised Undergraduate Research Grant, which is funded by the NSF; LSU Chancellor's Future Leaders in Research Award; and the Tiger Athletic Foundation Robert Ogden Honors College Honors Thesis Award. Their support has made this project possible. Their funding allowed me to fully focus on the code and meet all the conditions required to develop it into an iOS app. It has been used to purchase the hardware required to test the code, which is required to submit the app for review; it has been used enroll as an Apple iOS Developer, so that I could test the program on hardware, access iOS documentation and tutorials, and submit the app for department-wide use; and it has been used to support my time working on this project. Thank you.

I would like to thank LSU and the Robert Ogden Honors College for developing such a wonderful program and passage for students to form and completely develop a thesis. It is and has been a challenging and extremely rewarding experience. Thank you.

Lastly, my family, friends, and professors have my deepest gratitude for always helping, pushing, supporting, and encouraging me throughout my undergraduate years! I'd also like to thank Professor Neal Stoltzfus, LSU Department of Mathematics, LSU Center for Computation and Technology, contributers and developers of KnotTheory, and the stackoverflow community!

Contents

Abstract

The goal of this project is to implement and extend currently available software for knot theory using the Apple iOS environment. The study of knots and their invariants is fundamental in mathematics, mathematical physics, and subareas of chemistry and biology. Knots are typically represented by translating the intrinsically three-dimensional structure onto a flat, two-dimensional representation called the knot projection. It has been shown that many characteristics of such projections can be instrumental in determining nonequivalence of knots, and thus helping expand and understand the library of known knots.

This thesis explores an an Apple iOS app that computes and outputs the Mathematica representation of knots. The app is coded in C++, a computer programming language, and Objective C, the native programming language for Apple's iOS platform. First, necessary knot terminology and coding tools will be introduced, then the algorithm for encoding a knot will be discussed in full length, and, lastly, additional features of the code will be shown. Additional work as part of this project in the computer language Python is also included. Blending C++, iOS development, and the mathematical field of topology, this project leverages user-defined data types, iOS program capabilities, and properties of knot projections in order to encode the knot into its computer-recognized representation.

Chapter 1

Background

1.1 Introduction

How often do you pick up a pair of headphones, only to find them all knotted together? If one were to fuse the ends of those headphones, the earbuds to the jack, the resulting tangle would become a classical topological knot. Such structures form the backbone of knot theory. One fundamental problem in knot theory is determining whether two knots are topologically the same. Looking at those headphones again, you know from experience that you can untangle them, but how? You set out trying to unwind and unlink the cords, but often this task is not so easy. Trying to unravel and understand knots that you have no knowledge of is a tough problem, and it must be done to determine how tangled or how knotted those headphones really are. Similarly, such ideas are applied systematically and rigorously to mathematical knots.

Currently, mathematicians can only rely on human computation for discovering new knots, and quite often calculations are faced that cannot be completed in any given time period. This limitation can be alleviated using the same tool as is used to solve massive systems of equations: the computer. Programs such as SnapPy and Mathematica's KnotTheory are being developed to process topological data and build a common library of knots.[2,4] The Knot Atlas[3] hosts KnotTheory and is an online compendium of advances in knot theory. Meanwhile, the open-source package Sage provides an environment for linking together software together to promote experimentation across mathematics.[10] My goal with this honors thesis is to further develop and intertwine these programs together into a user-friendly, open-source package using the computer languages Python, Objective C and C++ with Apple's iOS framework. The ability to use computers to determine whether two knots are the same is an important step forward in knot theory. It provides mathematicians with a reliable technique for handling computations which are often too time-consuming to be feasible by hand and makes knot theoretical applications more accessible to biologists and chemists for knots rise naturally in molecular structures.

1.2 Programming Definitions

Consider the number 2. It has multiple descriptions and representations. The number 2 is a number, a real number (i.e. $2 \in \mathbb{R}$), and a natural number ($2 \in \mathbb{N}$). It can be described as a sum $1+1$ or a fraction $\frac{4}{2}$. Indeed,

$$2 = 2.0 = \frac{4}{2} = 4 \div 2 = 2.000,$$

and so on. In fact, we can always describe 2 in a finite representation up to any magnitude of error. (In fact, for any $y \in \mathbb{R}$ and $n \in \mathbb{N}$, we can always find an $x \in \mathbb{Q}$ such that $\mid y - x \mid < 10^{-n}$.)[9] Each of these descriptions are equivalent to the number 2.

Now consider the computer: It can only store a number with finitely many digits to represent it; it may store a number represented in various ways. Each different way can be specified by *type*. A *type* defines a set of possible values and operations for an object. An *object* is a piece of memory that holds a value of a given type.[11]

For example, suppose we wanted to store 2 in a computer program. We can represent it as type `int`, `double`, or `float`. Doing so would store the value of 2 as different representations and allocate different amounts of memory. `int` 2 would store 2 as the integer, whereas `double` 2 would allow for decimal points: 1.9998. If one were to store the `double` 2 as a, then represent a as an `int`, the computer would round or truncate the value for a. The compiler, the program that processes the code into machine language, will allow this conversion without warning, though it is unsafe.[11] It is only safe to assume error will unfold. For example, suppose one sought to add 0.5 to 2 and ran the following code.

```
#include <iostream>
int main(){
    int a = 2;
    std::cout << "a's value: " << a << std::endl;
    a = a + 0.5;
    std::cout << "a's new value: " << a << std::endl;
    return 0;
}
```

Then the following output follows.

```
a's value: 2
a's new value: 2
```

So, it's important to understand *when* to allocate more room for a value. If one knows that the value will be acted on and changed by an object with more storage and values unsupported by a smaller object, one should use the smaller type. In this example, if a is stored as a `double`, then no problems would arise.

```
#include <iostream>
int main(){
    double a = 2;
    std::cout << "a's value: " << a << std::endl;
    a = a + 0.5;
    std::cout << "a's new value: " << a << std::endl;
    return 0;
}
```

Then the following output follows.

```
a's value: 2
a's new value: 2.5
```

Types limit representation, but with this determining the correct type can allow the programmer to optimize (minimize) memory usage and allocation.

More general and complex building blocks of coding exist. One such structure is called a *user-defined type*. User-defined data types are precisely as the name describes: They are containers that allow the programmer to define any set of objects and operations to one type, as named by the programmer. There are two main tools that may be used to define such containers, `struct` and `class`. `structs` were first implemented in the `C` language, and `classes` were introduced in C++. C++ allows for both `structs` and `classes`. The only difference between the two is that in structs, the object types and operations are by default *public*, meaning the user of the program or functions independent from the `struct` are able to access the data and member functions of the struct. Classes, on the other hand, are by default *private*, i.e. nothing outside the class may access it.[11]

1.3 Knot Terminology

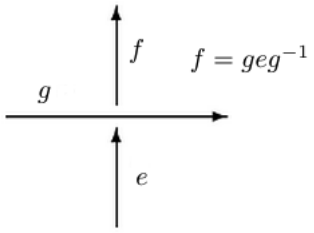
Everyday, most people wittingly or unwittingly tie a knot. A worker ties his shoe laces, and a college student is cursing the fact that his headphones are tangled into oblivion. We deal with knots on daily basis.

A topological *knot* is a smooth embedding of a circle ($\mathbb{S}^1$) into $\mathbb{R}^3$.[6] In other words, it is precisely those shoelaces, if the ends were fused together, in our three-dimensional space. The trivial knot, one with no crossings, is called the unknot. One open problem in knot theory is finding a systematic way to determine if any given knot is equivalent to the unknot. More generally, is there a way to determine the equivalence of any two knots?

Theory has developed to determine the equivalence of knots by studying their projections onto a plane. In 1932, a series of three axiomatic moves were proposed and proved by Kurt Reidemeister to answer this question. These operations, called "Reidemeister Moves," relate knots by three very natural transformations: twisting in one direction, shifting one strand over another, and moving a strand over a crossing. Applying these moves changes the outward appearance of the knot, but, as Reidemeister proved, the topological structure of the knot is preserved.[8] Reidemeister Moves were the basis for determining equivalence of two knots. However, as those headphones inevitably get more tangled, and as potentially new knots begin to be more complex in structure, using these moves becomes an increasingly difficult task. Moreover, the moves can only determine equivalence, but not complete non-equivalence.

Knot invariants such as the Fundamental Group, Alexander Polynomials, and Jones Polynomials have been developed based off of the Reidemeister Moves to help the inquiry. Using these tools, mathematicians can determine if two knots are not equivalent by comparing the associated invariants. Each invariant describes different aspects of the knot, and each has its own strengths for understanding the knot. For instance, the Fundamental Group uses strand labeling to determine the basic algebraic group structure of the complement of the knot, whereas the Jones Polynomial assigns the knot a polynomial using the amount of coiling of the knot. On the other hand, the Alexander Polynomial considers the homology of the knot complement and assigns a polynomial based on it. Computing this information is much like solving huge systems of equations; the work requires a diligent, careful hand, and for practice purposes can often be impossible to solve without the help of computers. Moreover, one knot invariant is not sufficient to determine equivalence. For example, the Alexander Polynomial of the unknot is 1, yet there exist nontrivial knots that also have Alexander Polynomial 1.[1] Mathematica's KnotTheory currently stores different knots and their associated known invariants.[2]

Due to the induced grid of Apple iOS UIKit, the framework within which the program operates, one may compute the Fundamental Group of the drawn knot. The fundamental group of a knot is an algebraic group presentation of the knot, and it is a topological invariant up to a homotopy, where a homotopy is a continuous mapping from one topological space to another. From a knot projection, it may be obtained by expressing strand relations in the following manner.[6]



Thus, it is noted in the Future Work section that this program may be further developed to include this information. Please note that, using the above image as a reference, henceforth strands of type e will be referred to as "into" undercrossings, and those of type f will be referred to as "outcoming" undercrossings. That is, the strand going "into" the crossing is e and the strand coming "out" is f.

1.4 Coding Structures

A note on notation: Types will be written in `typewriter font`. Elements of a type will be *italicized*. In addition, any function with name **function_name** will be referred to as **function_name**().

A `UIViewController` is a fundamental object and view-managing tool for iOS apps.[5] The UIView, buttons, labels, and all visible features are stored in memory and accessed from this object.

A `UIView` is the view layer that can be placed over a `UIViewController`'s screen.[5] In our work, the `UIView` in use is wrapped inside the `UIViewController`. That is, the object is a subcomponent of the `UIViewController`.

A `CGPoint` P is a C-`struct` that contains the location of the point P as x-y coordinates in the screen. A C-`struct` is a user-defined data type. Location on a screen is a mapping to pixels, and the origin is the pixel at the top left corner of the screen. The x-direction increases left-to-right, whereas y increases top-to-bottom. This format is the convention of the **UIKit**, the framework for Apple app architecture. `UIView`s have the attribute `frame` that has attributes `origin` and `size`. We leverage these components to make the code independent of screen size, so that users with an iPad or iPhone of any size may use the code. Objects are placed with respect to the elements `frame.origin` and `frame.size`.

A `UIBezierPath` is a class that allows for drawing of an object. We use the following methods:

- **addLineToPoint**(CGPoint P): Adds P to the path and draws line between last point and P to screen.
- **moveToPoint**(CGPoint P): Moves 'brush' to P and begins to draw to screen at P.
- **containsPoint**(CGPoint P): Checks area formed by current path. If contains point returns true.

A `vector<T>` is a C++ container that holds objects of type `T` linearly in memory. Thus, one may access elements randomly. For example, suppose one had a `vector` V of `int`s stored as $(57, 42, 729)$. Then $V[1] = 42$, and one may access the element 42 merely by calling $V[1]$. The i^{th} component of a `vector` is accessed and modified by writing vector$[i-1]$.

A `list<T>` is a C++ container that holds objects of type `T` and allows for no random-access memory. (That is to say, `list` does not hold a block of continuous memory. We cannot instantiate a `list` and attempt to directly access it in memory (i.e, list[0], list[1], list[2] does not work.) Instead, `list`s hold the data with `pointer`s, and `pointer`s store the address of the next element of the `list` in the computer memory (for a singly-linked forward-`list`). `list<objectType>::iterator` is used to transverse the container, where *objectType* refers to the type of object the `list` holds.

1.5 iOS Calling Actions

When a user touches the screen of an iPad or iPhone, such gestures are recognized as *event*s. The following actions are recognized as events and handled as described.

1. The action of touching the screen calls **touchesBegan**().
2. The action of moving the finger touching the screen calls **touchesMoved**().
3. The action of lifting the finger from the screen calls **touchesEnded**().

Chapter 2

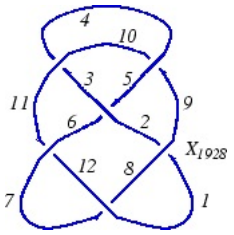
Problem Approach

2.1 Mathematica's KnotTheory

The ultimate goal of this project is to obtain information about any drawn knot via computer encoding and decomposition. Mathematica's KnotTheory package is a mathematical software that contains a library of knots, stored in a discrete representation, and their associated properties, including the Jones, Kauffman, and Alexander Polynomials. It also computes the Jones Polynomial of a knot algorithmically.[2]

Currently, the user must draw a knot's projection on paper, and number each strand according to the convention depicted in the KnotTheory manual. With that ordering, the user can input the numbers into Mathematica and access the properties associated with knots of the same structure. However, this package does not output data for all *equivalent* knots. Determining knot equivalence is a decision problem that is widely considered as hard.

The following image is an example of PD Diagram input and a possible representation of the knot.[1]



$$X_{1928}X_{3,10,4,11}X_{5362}X_{7,1,8,12}X_{9,4,10,5}X_{11,7,12,6}.$$

```
In[5]:= K = PD[
  X[1,9,2,8], X[3,10,4,11], X[5,3,6,2],
  X[7,1,8,12], X[9,4,10,5], X[11,7,12,6]
];
In[6]:= Alexander[K][-1]
Out[6]= -11
```

2.2 Method of Extraction

2.2.1 Convention

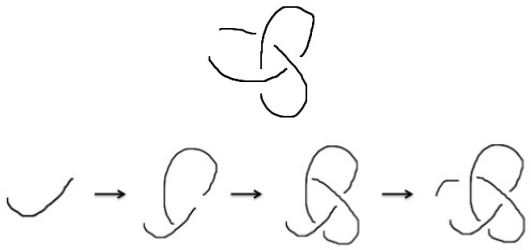
In order to discretize the knot, it is assumed that the user will draw the knot projection smoothly, i.e. lifting only at undercrossings, and will not attempt to close the knot. Each continuously drawn curve is

[1] Image from `http://katlas.org/wiki/Planar_Diagrams`

assumed to be necessary to the knot and represents a strand in the knot. The following rules are assumed.

1. The user will 'slice' the knot at some point and must draw to that point to finish.
2. The user will lift finger only at undercrossing and draw undercrossing to undercrossing.

Below is an example of this assumed method applied to the trefoil.

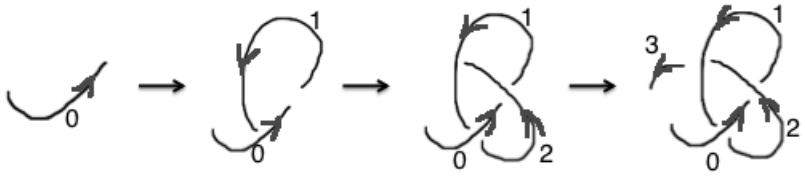


2.2.2 Consequences of the Convention

Once the finger touches the screen, a function call to **touchesBegan**() is triggered. Similarly, as the touching finger is moved, we have a function call to **touchesMoved**(), and lifting of the finger calls **touchesEnded**(). As each function is called, the coordinates are stored in a list called "pathPoints."
We explore the trefoil as an example.

Observe, then, that two pieces of information about the knot are known:

1. Direction, given by the order of input into the list.
2. Type of crossing, following the convention of lifting before an undercrossing.



2.2.3 User-Defined Data Types

We define the following data types to store information as it is processed.

```
class knot_edge{
public:
    std::list<CGPoint> FullKnotEdge;
    int parent_edge; //original edge before breaking up
    int edge_number;
    CGPoint IPa;
    CGPoint IPb;
    bool starter;
```

```
9
10      knot_edge(): FullKnotEdge(),
11      parent_edge(), edge_number(),
12      IPa(CGPointZero), IPb(CGPointZero),
13      starter(false){}
14
15  };
16
17
18  class knot_crossing{
19  public:
20
21      std::vector<int> crossing_elements;
22      // crossing_elements is stored such that:
23      //     *'into' undercrossing          --> element [0]
24      //     *counterclockwise OVERcrossing--> element [1]
25      //     *out' undercrossing            --> element [2]
26      //     *counterclockwise to [2]       --> element [3]
27      bool switched_crossing;
28
29      // Member functions:
30      void switch_crossings();
31      void shift(int shift);
32      void clear();
33
34      knot_crossing():crossing_elements(4),
35                      switched_crossing(false){}
36
37  };
38
39  // If desired, interchange overcrossings with undercrossings
40  void knot_crossing::switch_crossings(){
41
42      // First, please recall:
43      // crossing_elements is stored such that:
44      //     *'into' undercrossing          --> element [0]
45      //     *counterclockwise OVERcrossing--> element [1]
46      //     *out' undercrossing            --> element [2]
47      //     *counterclockwise to [2]       --> element [3]
48      //
49      // We will 'rotate' the elements counter-clockwise.
50
51      if(switched_crossing == false){
52          int temp_u0 = crossing_elements[0];
53          int temp_u1 = crossing_elements[2];
54
55          crossing_elements[0] = crossing_elements[1];
56          crossing_elements[2] = crossing_elements[3];
57          crossing_elements[1] = temp_u1;
58          crossing_elements[3] = temp_u0;
59
60          switched_crossing = true;
61
62      }
```

```
    else{
        // Perform inverse operation of above.

        int temp_u0 = crossing_elements[0];
        int temp_u1 = crossing_elements[2];

        crossing_elements[0] = crossing_elements[1];
        crossing_elements[2] = crossing_elements[3];
        crossing_elements[1] = temp_u0;
        crossing_elements[3] = temp_u1;

        switched_crossing = false;

    }
}

void knot_crossing::clear(){
    for(int i = 0; i < crossing_elements.size(); ++i){
        crossing_elements[i]=0;
    }
}


void knot_crossing::shift(int shift){
    for(int i = 0; i < crossing_elements.size(); ++i){
        crossing_elements[i]=crossing_elements[i]-shift;
    }

}
```

2.3 Splitting Strands

As the user draws, the computer stores the input in a `list` of `knot_edge`s. The object *IPa* is the first `CGPoint` in the strand, and the object *IPb* is the last element in the strand.

```

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    // ..


    // Store KnotEdge as knot_edge's list of path points, and
    // store this last point and the first point (from touchesBegan)
    // as the extrema (IPa, IPb) for the knot_edge.
    knot_edge ke;
    ke.FullKnotEdge = KnotEdge;
    ke.IPa = smallLP[0];
    ke.IPb = smallLP[1];
    ke.parent_edge = counter;
    ke.edge_number = counter;
    ke.starter = true; // indicates this is a drawn edge
    discrete_edges.push_back(ke);

```

```
    // Note: smallLP is a vector of the points where the
    // user lifted his/her touch

    // Clear KnotEdge to be ready for next edge.
    KnotEdge.clear();
    smallLP.clear();

    [self touchesMoved:touches withEvent:event];

}
```

The first traversal is to split the knot into the desired components. This is done by artifically creating a line between the last `CGPoint` in strand and the first `CGPoint` in the next drawn strand, finding a point in some edge E that also solves the equation of the line, synthetically splitting E into two components, and inserting the two components into the set of all edges.

```
// The following code segment define lines between
// each undercrossing space.
double a,b,c,d;
double m, yint;
int sentinel = 2*int(liftPoints.size());
std::list<knot_edge>::iterator startLine = discrete_edges.begin();
std::list<knot_edge>::iterator endLine;
edgeCount+=1;
int partitionCount = 0;
// Traverse list of edges:
while ((startLine != discrete_edges.end() ||
        endLine != discrete_edges.begin()) &&
       partitionCount <= sentinel){
    partitionCount++;
    endLine = startLine;

    // Handle boundary case.
    advance(endLine,1);
    if(endLine == discrete_edges.end()){
        endLine = discrete_edges.begin();
    }
    // Anchor at liftpoints.
    CGPoint LP1 = (*startLine).IPb;
    CGPoint LP2 = (*endLine).IPa;

    crossing.crossing_elements[0] = (*startLine).parent_edge;
    crossing.crossing_elements[2] = (*endLine).parent_edge;

    double delta = 1.5;
    double buffer = 0.5;

    CGPoint intersection_point; // to draw block about intersection points

    if ((LP1.x != LP2.x) && (LP1.y != LP2.y)){

        // Case 1: x- or y-values are equal.
        if(std::abs(LP1.x - LP2.x) < delta && LP1.y > LP2.y){
            a = LP1.x-buffer;
            b = LP2.x+buffer;
```

```
40              c = LP2.y;
41              d = LP1.y;
42              m = 0;
43          }
44          else if(std::abs(LP1.x - LP2.x) < delta && LP1.y <= LP2.y){
45              a = LP1.x-buffer;
46              b = LP2.x+buffer;
47              c = LP1.y;
48              d = LP2.y;
49              m = 0;
50          }
51          else if(std::abs(LP1.y - LP2.y) < delta && LP1.x > LP2.x){
52              a = LP2.x;
53              b = LP1.x;
54              c = LP2.y-buffer;
55              d = LP1.y+buffer;
56              m = 0;
57          }
58          else if(std::abs(LP1.y - LP2.y) < delta && LP1.x <= LP2.x){
59              a = LP1.x;
60              b = LP2.x;
61              c = LP1.y-buffer;
62              d = LP2.y+buffer;
63              m = 0;
64          }
65          // Otherwise:
66          else{
67              if (LP1.x > LP2.x){
68                  a = LP2.x;
69                  b = LP1.x;
70              }
71              if(LP1.x < LP2.x){
72                  a = LP1.x;
73                  b = LP2.x;
74              }
75              if (LP1.y > LP2.y ){
76                  c = LP2.y;
77                  d = LP1.y;
78              }
79              if (LP1.y < LP2.y ){
80                  c = LP1.y;
81                  d = LP2.y;
82              }
83              m = (LP2.y - LP1.y )/ (LP2.x - LP1.x);
84          }
85
86
87          // Obtain y-intercept in the equation of the line between extrema.
88          yint = LP2.y - m * LP2.x;
89
90          intersection_point = partitionKnot(discrete_edges,
91                                             LP1,LP2,a,b,c,d,
92                                             m , yint,
93                                             startLine, endLine);
```

```
94          intersectPoints.push_back(intersection_point);
95
96          advance(startLine,1);
97      }
98      else{
99          startLine++;
100     }
```

Once the user clicks the button labeled "Get Knot," the program begins to iterate through the list of strands, constructing the equation of a line between each pair of strands ordered as they were drawn. For each line equation, the computer traverses the set of edges until it finds an edge with a point that also solves the equation of the line, denote it p_*. This is the edge that intersects, or crosses over, the strands. Denote it $\tilde{e}$.

With $\tilde{e}$ known, the code then splits $\tilde{e}$'s list of points, *FullKnotEdge*, into two `lists`: one contains all points from the first element to p_*, and the second contains all points from p_* to the end of the `list`. Two new `knot_edges`, *ke1* and *ke2* are instantiated, each given one of the two `lists` as its associated *FullKnotEdge*. The edge $\tilde{e}$ is erased from the `list` of all edges, and *ke1* and *ke2* are inserted in its place.

```
1  // FUNCTION:   partitionKnot
2  // SCOPE:      C++. Separate component of the Objective C UIView.
3  // PARAMETERS: std::list<knot_edge>& edges, *(list of edges)*
4  //             CGPoint LP1, CGPoint LP2, *(extrema of two strands)*
5  //             int a, int b, int c, int d, *(details of the line)*
6  //             float m, float yint,        *(details of the line)*
7  //             std::list<knot_edge>::iterator& startIter,
8  //                 *(points to first strand ['into'] strand)*
9  //             std::list<knot_edge>::iterator& endIter
10 //                 *(points to second strand ['outcoming'] strand)*
11 // RETURNS:    CGPoint  *(intersection point)*
12 // PURPOSE:    Find the edge (E) which intersects the line between the two
13 //             extrema LP1 and LP2.
14 // MODIFIES:   edges (-->discrete_edges, in this case).
15 //             startIter & endIter if necessary
16 // MODS:       Splits E into two elements of edges (ie two separate edges)
17 //             Adds the new elements, in order as drawn, into edges.
18 //             Deletes E from edges.
19 CGPoint partitionKnot(std::list<knot_edge>& edges,
20                       CGPoint LP1, CGPoint LP2,
21                       double a, double b, double c, double d,
22                       float m, float yint,
23                       std::list<knot_edge>::iterator& startIter,
24                       std::list<knot_edge>::iterator& endIter ){
25
26     // Finds points that satisfy the line equation (i.e. is a crossing)
27     std::list<knot_edge>::iterator EdgeCheckIter = edges.begin();
28     std::list<CGPoint>::iterator checkIter =
29               ((*EdgeCheckIter).FullKnotEdge).begin();
30     CGPoint intersection_pt; // store intersection point
31
32     // New instances of knot_edge, to be used for the new strands.
33     knot_edge ke1;
34     knot_edge ke2;
35
36     int delta = 3;
37
38     checkIter = ((*EdgeCheckIter).FullKnotEdge).begin();
```

```
39      bool intersectFound = false;
40
41      while(EdgeCheckIter != edges.end() && intersectFound == false){
42
43          while((checkIter != ((*EdgeCheckIter).FullKnotEdge).end() &&
44                 EdgeCheckIter != edges.end() )&&
45                intersectFound == false){
46
47
48              if ( ( ((*checkIter).x != LP1.x)||((*checkIter).y != LP1.y)) &&
49                   ( ((*checkIter).x != LP2.x)||((*checkIter).y != LP2.y))){
50
51                  // Check if intersection is found.
52                  // First, check the box about the intersection.
53                  if ( ( ((*checkIter).x >= a) && ((*checkIter).x <= b) ) &&
54                       ( ((*checkIter).y >= c) && ((*checkIter).y <= d) ) ){
55
56                      // Solve equation of line with checkIter.
57                      // If checkIter's y matches the line equation,
58                      // then found intersection.
59                      double y = (m * ((*checkIter).x)) + yint;
60
61                      if ( std::abs((*checkIter).y - y) < delta ){
62
63                          KnotEdge.clear();
64
65                          // Store path points leading to the intersection.
66                          for(std::list<CGPoint>::iterator subInterval
67                              = ((*EdgeCheckIter).FullKnotEdge).begin();
68                              subInterval != checkIter; ++subInterval){
69
70                              KnotEdge.push_back(*subInterval);
71                          }
72                          ke1.FullKnotEdge = KnotEdge;
73
74                          KnotEdge.clear();
75                          // Store  points from intersection to end of path.
76                          for(std::list<CGPoint>::iterator
77                          subInterval = checkIter;
78                          subInterval!=((*EdgeCheckIter).FullKnotEdge).end();
79                          ++subInterval){
80                              KnotEdge.push_back(*subInterval);
81                          }
82
83                          ke2.FullKnotEdge = KnotEdge;
84
85                          intersection_pt = CGPointMake(checkIter->x,
86                                                        checkIter->y);
87
88                          for (std::list<knot_edge>::iterator
89                           i = edges.begin(); i !=  edges.end(); ++i){
90                              if ((*i).edge_number ==
91                                (*EdgeCheckIter).edge_number){
92
```

```
                                ke1.starter = false;
                                ke2.starter = false;

                                ++edgeCount;
                                ke1.parent_edge=(*EdgeCheckIter).parent_edge;
                                ke1.edge_number = edgeCount;
                                ++edgeCount;
                                ke2.parent_edge=(*EdgeCheckIter).parent_edge;
                                ke2.edge_number = edgeCount;

                                // knot_edges are stored in order drawn.
                                ke1.IPa = (*i).IPa;
                                ke1.IPb = (*checkIter);
                                ke2.IPa = (*checkIter);
                                ke2.IPb = (*i).IPb;
                                 // ...
                                 }
                             //..
}
```

The coordinate system is employed to determine the components of each crossing, which `knot_edge` is the "into" undercrossing, which is identified as `crossing_element[0]`, and how the strands are positioned with respect to the "into" undercrossing. For each crossing, a vector between the first strand's *IPa* and the next strand's *IPb* points is determined, denoted *vecPivot*, and a vector from some element in each of the new edges and p_* is also found. Depending on the sign of the *vecPivot* elements, the counter-clockwise adjacencies are determined for each crossing.

```
int k = int((ke1.FullKnotEdge).size());
k = k-floor(k/4);
std::list<CGPoint>::iterator it1 = (ke1).FullKnotEdge.begin();
advance(it1, k);
k = int((ke2.FullKnotEdge).size());
k = floor(k/4);
std::list<CGPoint>::iterator it2 = (ke2).FullKnotEdge.begin();
advance(it2, k);

CGPoint vecPivot;
CGPoint vec1;
CGPoint vec2;

// Use position to determine which element is
// counter-clockwise to the 'into' undercrossing.
vecPivot.x = (*endIter).IPa.x - (*startIter).IPb.x;
vecPivot.y = (*endIter).IPa.y  - (*startIter).IPb.y;
vec1.x = (*it1).x - (*checkIter).x;
vec1.y = (*it1).y - (*checkIter).y;
vec2.x = (*it2).x - (*checkIter).x;
vec2.y = (*it2).y - (*checkIter).y;


if(vecPivot.y < 0 && vecPivot.x >=0) {
    if( vec1.y>=0 || vec1.x>=0){
        crossing.crossing_elements[1] = ke1.edge_number;
        crossing.crossing_elements[3] = ke2.edge_number;

```

```
29
30      }
31      else {
32          crossing.crossing_elements[1] = ke2.edge_number;
33          crossing.crossing_elements[3] = ke1.edge_number;
34      }
35  }
36  else if(vecPivot.y < 0 && vecPivot.x <= 0) {
37      if(vec1.y <=0 || vec1.x>=0){
38          crossing.crossing_elements[1] = ke1.edge_number;
39          crossing.crossing_elements[3] = ke2.edge_number;
40      }
41      else {
42          crossing.crossing_elements[1] = ke2.edge_number;
43          crossing.crossing_elements[3] = ke1.edge_number;
44      }
45
46  }
47
48  else if(vecPivot.y > 0 && vecPivot.x <=0) {
49      if(vec1.y<=0 || vec1.x <=0){
50          crossing.crossing_elements[1] = ke1.edge_number;
51          crossing.crossing_elements[3] = ke2.edge_number;
52      }
53      else {
54          crossing.crossing_elements[1] = ke2.edge_number;
55          crossing.crossing_elements[3] = ke1.edge_number;
56      }
57
58  }
59  else if(vecPivot.y > 0 && vecPivot.x >=0) {
60      if(vec1.y>=0 || vec1.x <=0 ){
61          crossing.crossing_elements[1] = ke1.edge_number;
62          crossing.crossing_elements[3] = ke2.edge_number;
63      }
64      else {
65          crossing.crossing_elements[1] = ke2.edge_number;
66          crossing.crossing_elements[3] = ke1.edge_number;
67      }
68
69  }
70  if(vecPivot.y > 0 && vecPivot.x == 0) {
71      if(vec1.x <=0 ){
72          crossing.crossing_elements[1] = ke1.edge_number;
73          crossing.crossing_elements[3] = ke2.edge_number;
74      }
75      else {
76          crossing.crossing_elements[1] = ke2.edge_number;
77          crossing.crossing_elements[3] = ke1.edge_number;
78      }
79
80  }
81  if(vecPivot.y < 0 && vecPivot.x == 0) {
82      if(vec1.x >=0 ){
```

```
83          crossing.crossing_elements[1] = ke1.edge_number;
84          crossing.crossing_elements[3] = ke2.edge_number;
85      }
86      else {
87          crossing.crossing_elements[1] = ke2.edge_number;
88          crossing.crossing_elements[3] = ke1.edge_number;
89      }
90
91  }
92  if(vecPivot.y == 0 && vecPivot.x > 0) {
93      if(vec1.y >=0 ){
94          crossing.crossing_elements[1] = ke1.edge_number;
95          crossing.crossing_elements[3] = ke2.edge_number;
96      }
97      else {
98          crossing.crossing_elements[1] = ke2.edge_number;
99          crossing.crossing_elements[3] = ke1.edge_number;
100     }
101
102 }
103 if(vecPivot.y == 0 && vecPivot.x < 0) {
104     if(vec1.y <=0 ){
105         crossing.crossing_elements[1] = ke1.edge_number;
106         crossing.crossing_elements[3] = ke2.edge_number;
107     }
108     else {
109         crossing.crossing_elements[1] = ke2.edge_number;
110         crossing.crossing_elements[3] = ke1.edge_number;
111     }
112
113 }
114
115 if(startIter == i){
116     advance(startIter,-2);
117     advance(endIter, -1);
118 }
119 if(endIter == i){
120     advance(endIter,-1);
121 }
122
123 edges.insert(i,ke1);
124 edges.insert(i,ke2);
125 --i;
126
127 all_crossings.push_back(crossing);
128 crossing.clear();
129
130 EdgeCheckIter = i; // Now points to ke2
131
132 ++i;
133
134 // If already broke up the strand, will need to update crossings.
135 // Do this by understanding the previous crossing element to the
136 // knot being broken up.
```

```
// We're using the fact that upon going 'into,' the next strand
// label is one higher than the previous.
// So we know the orientation of the strand.
if((*i).starter == false){
    for (std::list<knot_crossing>::iterator it = all_crossings.begin();
    it != all_crossings.end(); ++it){
        int m = 0;
        for(int j = 0; j < 4; ++j){
            switch(j){
                case 0: m = 2;
                case 1: m = 3;
                case 2: m = 0;
                case 3: m = 1;
            }
            if ((*it).crossing_elements[j] == (*i).edge_number){
                if((*it).crossing_elements[j] >(*it).crossing_elements[m]){
                    (*it).crossing_elements[j] = ke1.edge_number;
                }
                else{
                    (*it).crossing_elements[j] = ke2.edge_number;
                }
            }
        }
    }
}
//..
```

The strands of each knot are found dynamically, though; as such, the elements in the crossings must also be updated, as one `knot_edge`'s *edgeNumber* changes with the splittage of the knot. This problem is the reason for the *`bool starter`* element in `knot_edge`.

2.4 Ordering the Strands

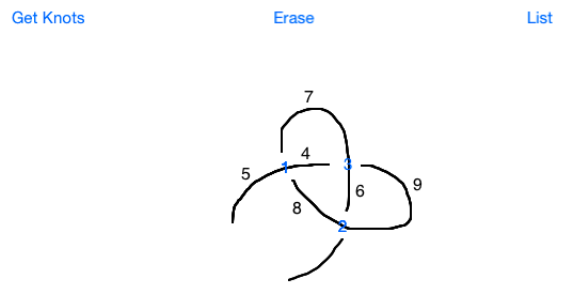
Understanding the new components of the knot, how original identifiers map to new ones, and determining the order of the crossings is the next problem. First, the final identifiers (*edgeNumber*) of each strand must be found. Second, the strands must be relabeled in canonical order; no numbers can be skipped, and the numbering must start with 1. Third, the order of the crossings must be determined. In general, KnotTheory convention is such that the undercrossings are in *increasing* order. To preserve the structure of the knot, for the ordered list of crossings, the numbering of the 'outcoming' element `crossing_element[2]` and the next crossing's "into" element `crossing_element[0]` must be non-decreasing.

To first recast the knot given the splittage and the `list` of all edges `discrete_edges`, the program first finds the first crossing's "into" element within `discrete_edges` and begins numbering from there. To do this, the `discrete_edges`'s *edge_number* is stored in a `vector edgeIds` and use the position of the *edge_number* to reassign all instances of *edge_number* with its position in `edgeIds`.

```
    //...
    minUndercrossingLocation = all_crossings.begin();
    for (std::list<knot_crossing>::iterator iter = all_crossings.begin();
     iter != all_crossings.end(); ++iter){
        if ((*iter).crossing_elements[0] <=
          (*minUndercrossingLocation).crossing_elements[0]){
            minUndercrossingLocation = iter;
        }   }

```

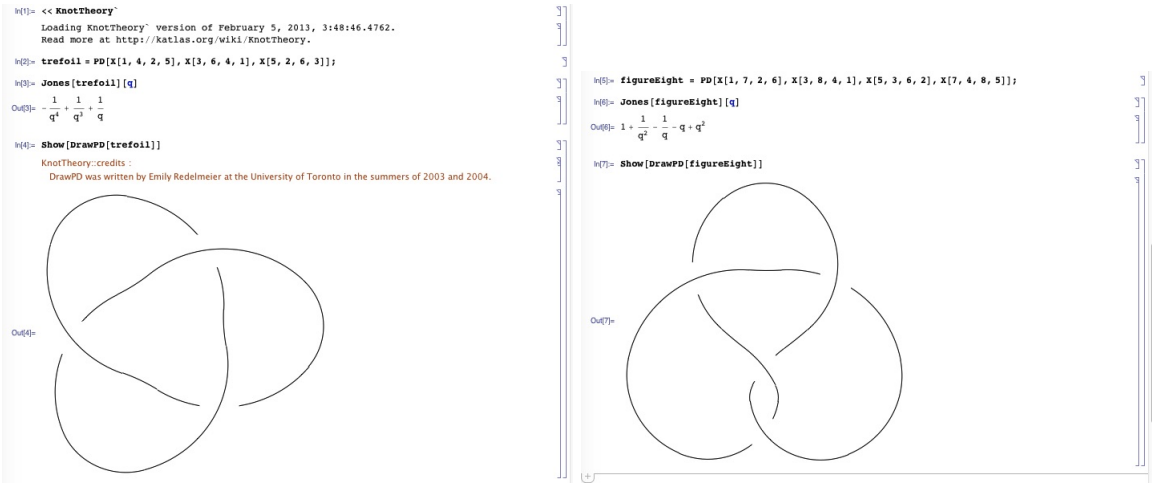
```
        int starterElement =minUndercrossingLocation->crossing_elements[0];
        std::list<knot_edge>::iterator iterStarterElement =
              discrete_edges.begin();
        int starterCounter = 0;
        while((*iterStarterElement).edge_number!= starterElement
        && starterCounter <= totalNumberStrands){
            ++iterStarterElement;
            ++starterCounter;
        }

        if(iterStarterElement != discrete_edges.end()){
            for(std::list<knot_edge>::iterator it = iterStarterElement;
              it != discrete_edges.end(); ++it){
                edgeIds.push_back((*it).edge_number);
            }
            for(std::list<knot_edge>::iterator it = discrete_edges.begin();
            it != iterStarterElement; ++it){
                edgeIds.push_back((*it).edge_number);
            }

            std::vector<int>::iterator iterEdge2;

            for(int i = 0; i < edgeIds.size(); ++i){
                for (std::list<knot_crossing>::iterator iterCross1
                 = all_crossings.begin();
                  iterCross1 != all_crossings.end(); ++iterCross1){
                    iterEdge2 =
                    std::find(((*iterCross1).crossing_elements).begin(),
                    ((*iterCross1).crossing_elements).end(),edgeIds[i]);
                    if(iterEdge2!=((*iterCross1).crossing_elements).end()){
                        (*iterEdge2) = i+1;
                    }
                    //..
                }
            }
            //..
```

In this way, we obtain the updated values and the correct ordering of the strands in the knot.

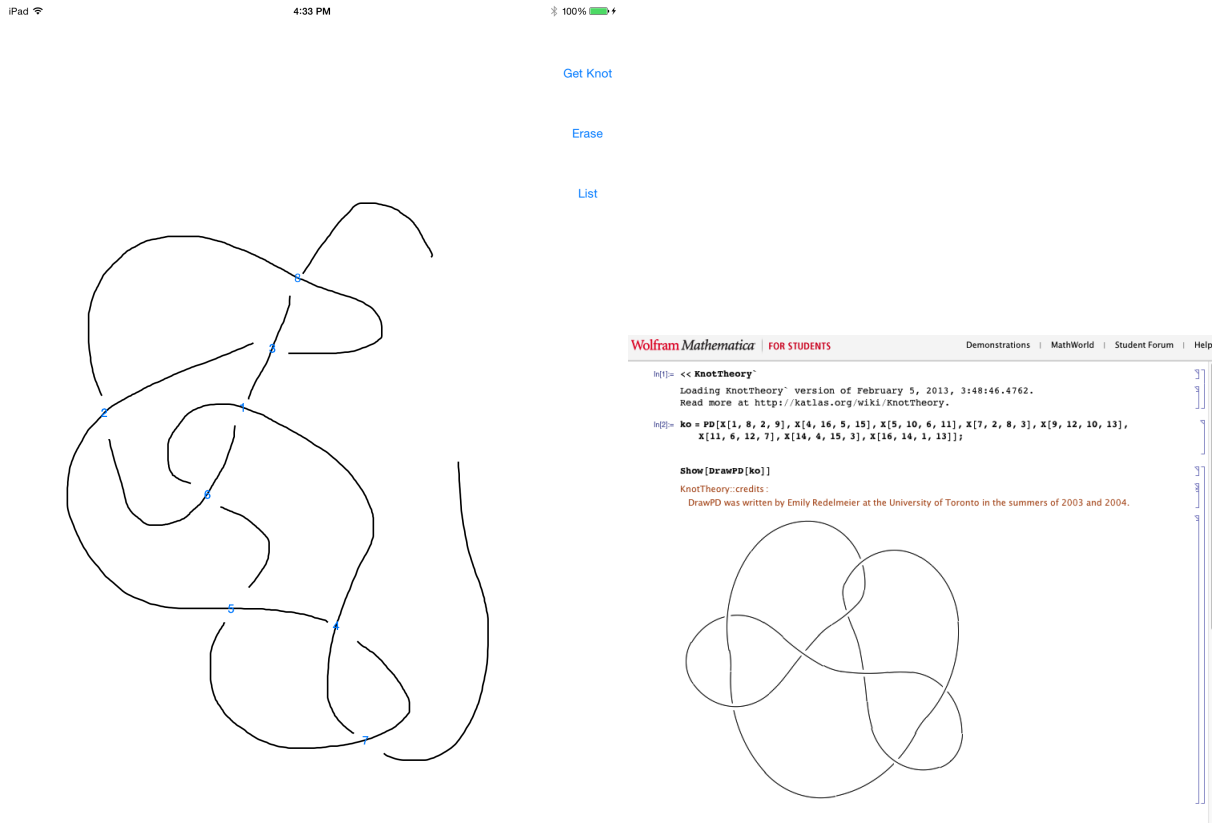


2.5 Sample Output

When the user hits "List," s/he opens an email with the Mathematica `PD` representation of the knot. Below are sample outputs generated by the code and then opened in Mathematica.



Sample input and corresponding output input into Mathematica:

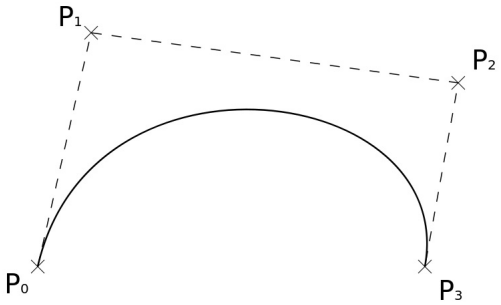


Chapter 3

Additional Aspects

3.1 Drawing to Screen

A cubic Bezier Path is used in order to draw smooth lines to the screen. Like cubic splines, cubic Bezier paths are twice continuously differentiable by construction; thus, they achieve a relatively nice level of smoothness. To do this, a total of five points will be considered: one base node and four points in succession. The following image demonstrates this idea.[1]



Note, a fifth point is tracked for the next starting place of the next Bezier path. As the user draws after the first iteration (i.e. after the first set of five points has been saved), only three points within a container are dynamically added. Two elements from the previous curve are used to draw the next curve segment. Implemented code to achieve these results (with added comments) is shown below.

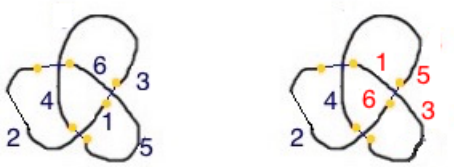
```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event

    // Obtain information about the touch event.
    UITouch *touch = [touches anyObject];
    CGPoint p = [touch locationInView:self];

    bitIndex++;

    bitMapPoints[bitIndex] = p;
    if (bitIndex == 4)
    {
        bitMapPoints[3] = CGPointMake(
                          (bitMapPoints[2].x+bitMapPoints[4].x)/2.0,
                          (bitMapPoints[2].y + bitMapPoints[4].y)/2.0);
        [path moveToPoint:bitMapPoints[0]];
```

[1] Image courtesy of Wikimedia.

```
        // Create and add the cubic Bezier using the above points.
        [path addCurveToPoint:bitMapPoints[3]
           controlPoint1:bitMapPoints[1] controlPoint2:bitMapPoints[2]];

        [self setNeedsDisplay]; // draw to screen

        bitMapPoints[0] = bitMapPoints[3]; // the next starting point
        bitMapPoints[1] = bitMapPoints[4];
        bitIndex = 1;
    }
//..
}
```

3.2 Switch Crossings

After the knot is broken up into its components, if the user wishes to switch an undercrossing to an overcrossing, he or she may do so by tapping on the crossing in question and then tapping the button "over<switch>under" that pops up. Touching this button will send out the IBAction **switchCrossings**(). Within this function, the crossing's class member function **switch_crossings**() is called, which interchanges `crossing_element[0]` with `crossing_element[3]` and similarly for the other pair. The elements are merely rotated. The inverse operation is a clockwise rotation.
A counter-clockwise rotation should be used to preserve the structure of the knot. Transposing the `crossing_element[0]` element with `crossing_element[3]` and `crossing_element[1]` with `crossing_element[2]` can be thought of as rotating the elements. The mapping may be applied in any algebraic group setting as two transpositions.




```
// Within the knot_crossing structure:
// If desired, interchange overcrossings with undercrossings
void knot_crossing::switch_crossings(){

    // First, please recall:
    // crossing_elements is stored such that:
    //    *'into' undercrossing          --> element [0]
    //    *counterclockwise OVERcrossing --> element [1]
    //    *out' undercrossing            --> element [2]
    //    *counterclockwise to [2]       --> element [3]
    //
    // We will 'rotate' the elements counter-clockwise.

    if(switched_crossing == false){
        int temp_u0 = crossing_elements[0];
        int temp_u1 = crossing_elements[2];

        crossing_elements[0] = crossing_elements[1];
        crossing_elements[2] = crossing_elements[3];
```

```
        crossing_elements[1] = temp_u1;
        crossing_elements[3] = temp_u0;

        switched_crossing = true;

    }
    else{
        // Perform inverse operation of above.

        int temp_u0 = crossing_elements[0];
        int temp_u1 = crossing_elements[2];

        crossing_elements[0] = crossing_elements[1];
        crossing_elements[2] = crossing_elements[3];
        crossing_elements[1] = temp_u0;
        crossing_elements[3] = temp_u1;

        switched_crossing = false;

    }
}
```

The following code is what is used to generate and handle the label.

```
// FUNCTION:   switch_crossings_label
// SCOPE:      Objective C UIView.
// PARAMETERS:
//          (int)cross_num       *(crossing number)*
//          touch_loc:(CGPoint)touchLocationInView *(pos of user's touch)*
//   GLOBAL PARAMETERS:
//          touch_crossing *(bool indicating if a crossing was touched)*
// RETURNS:    no value.
// PURPOSE:    Output a label for user at touch of a crossing.
-(void)switch_crossings_label:(int)cross_num
    touch_loc:(CGPoint)touchLocationInView{

    if(touch_crossing == true){
        // Delete the previous click's label.
        // (This will be true if someone clicked on one
        // crossing then clicked on another.)
        [[self viewWithTag:-1]removeFromSuperview];
    }

    double x_buff = 0.15;
    double y_buff = 0.04;

    UIButton *switchCrossingButton =
        [UIButton buttonWithType:UIButtonTypeCustom];
    switchCrossingButton = [[UIButton alloc] init];
    switchCrossingButton = [UIButton buttonWithType:
           UIButtonTypeRoundedRect];

    CGPoint positionInScene = touchLocationInView;
    CGPoint mid = CGPointMake( CGRectGetMidX(self.frame),
                               CGRectGetMidY(self.frame));
```

```
    CGFloat xPosition=positionInScene.x +x_buff*(positionInScene.x);
    CGFloat yPosition=positionInScene.y+y_buff*(positionInScene.y + mid.y);

    switchCrossingButton.frame = CGRectMake(xPosition, yPosition, 150, 75);
    [switchCrossingButton setTitle:@"over<switch>under"
           forState:UIControlStateNormal];

    switchCrossingButton.tag = -1;
    [switchCrossingButton addTarget:self action:@selector(switchCrossings:)
    forControlEvents:UIControlEventTouchUpInside];

    [self addSubview:switchCrossingButton];

}
```

```
// FUNCTION:   buttonTouched_crossing
// SCOPE:      Objective C UIView.
// PARAMETERS: (id)sender
//             ~the function is an IBAction, which means it will be
//             called by some 'sender' (UIButton) when an action occurs
//   GLOBAL PARAMETERS:
//              crossing_element *(label of crossing touched)*
//              previousCrossingElement *(label of prev. crossing touched)*
// RETURNS:    no value.
// PURPOSE:    Output a label for user at touch of a crossing.
- (IBAction)buttonTouched_crossing:(id)sender{

    int number_fixed_tags = 4;
    NSInteger tag = [sender tag];
    CGPoint button_pos = [sender position];

    // If called a second time before switching the crossing.
    previousCrossingElement = crossingElement;

    if(switch_crossing_control == true){
        previousCrossingElement +=number_fixed_tags;
    }

    crossingElement         = (int)tag; // will be used as index element
    switch_crossing_control = false;

    // If another crossing was already touched,
    // remove the button generated for that crossing.
    if (touch_crossing == true){
        [[self viewWithTag:-1]removeFromSuperview];
    }

    // Allow user to touch crossings and redefine (under/over).
    if (tag > 3){
        touch_crossing =  true;
        [self switch_crossings_label:crossingElement touch_loc:button_pos];
    }
}
```

3.3 Auto-resizing

Both Landscape and Portrait mode are supported by the project. However, in order to correctly allow the user to switch freely between Landscape and Portrait mode while maintaining positions of the buttons and labels, iOS has an autoresizing feature for subviews. There is one `UIView` within the `UIViewController` that the program occurs on. All subviews of this `UIView` will be given the ability to autoresize. Options exist for autoresizing results; we use *FlexibleLeftMargin* and *FlexibleHeight* here.

```
@implementation ViewControllerP
//...
- (void)viewDidLoad
{
    LinearInterpView *Drawview = [[LinearInterpView alloc]
                    initWithFrame:self.view.bounds];
    // Handle rotation, so that UIButtons reposition as desired.
    [Drawview setAutoresizesSubviews:YES];
    [Drawview setAutoresizingMask:
            UIViewAutoresizingFlexibleLeftMargin |
            UIViewAutoresizingFlexibleHeight];
    Drawview.userInteractionEnabled = YES;
    [self.view addSubview:Drawview];
}
//...
@end
```

3.4 Transferring Data

3.4.1 Mathematica Representation

To write the knot in the Mathematica Representation, we write two functions: **write_knot_description**() and **find_next_undercrossing**(). **write_knot_description**() develops the `string` that will be used to output the knot representation, while **find_next_undercrossing** traverses the set of crossings to find the next element.

```
// FUNCTION:   write_knot_description
// SCOPE:      C++. Separate component of the Objective C UIView.
// PARAMETERS: std::list<knot_crossing>::iterator i,
//             std::string& knot_description_full
// RETURNS:    no value.
// PURPOSE:    Modify the string knot_description_full, adding the info.
//             for crossing at address pointed to by i.
void write_knot_description(std::list<knot_crossing>::iterator i,
                std::string& knot_description_full,
                int& totalCount){

    // If this is not first crossing added.
    if(totalCount > 0){
        knot_description_full = knot_description_full + ",";

    }
    int u0 = (*i).crossing_elements[0];
    int u1 = (*i).crossing_elements[2];
    int o0 = (*i).crossing_elements[1];
    int o1 = (*i).crossing_elements[3];
```

```
21
22
23      knot_description_full = knot_description_full + "X[" +
24                  std::to_string(u0) + "," + std::to_string(o0) + "," +
25                  std::to_string(u1) + "," + std::to_string(o1) + "]";
26
27      totalCount+=1;
28
29  }
```

Now, to preserve the knot, care must be taken into deciding the order of crossings. The PD knot representation outputted from the (very different) code would yield deceptive results. The drawn `DrawPD` diagram would look exactly the same at the drawn knot, and the Alexander Polynomial representation would be identical to the known Alexander Polynomial of the drawn knot. However, the Jones Polynomial was not be the same, not even identically so. This resulted from "naive" ordering of the crossings and lack of care with respect to the order of the undercrossings and overcrossings. It is for this reason that we set the 0^{th} element of the undercrossings, `crossing_element[0]`, across the board to be the elements directed "into" a crossing, and the `crossing_element[1]` is its eastern neighbor, relative to that 0^{th} strand. Thus note that the Mathematica representation and our own subsequent ordering imposes a direction on the knot.

In order to preserve the knot and obtain the correct representation, one must pay attention to the undercrossing opposite the "into" undercrossing, `crossing_element[0]`. As the path points of the knot are ordered by the way the user draws the knot, and the knot is traversed in order to break up the strands into its piecewise components, the next crossing will be the one with the `edgeNumber` of the "into" undercrossing `crossing_element[0]` next in line to the 'outgoing' crossing `crossing_element[2]` in the previous `knot_crossing` element.

```
1  // FUNCTION:   find_next_undercrossing
2  // SCOPE:      C++. Separate component of the Objective C UIView.
3  // PARAMETERS: int minUndercrossing,   *(edgeNumber of knot)*
4  //std::list<knot_crossing>::iterator&min_cross,*(iterator pointing to min)*
5  //          std::string &knot_desc  *(string description of the knot)*
6  //          std::list<knot_edge>& edges, *(list of all edges)*
7  //          std::list<knot_crossing>& crossings, *(list of all crossings)*
8  //          int totalCount *(counter for number of iterations)*
9  // RETURNS:    no value.
10 // PURPOSE:    Modify the string knot_description_full, adding the info.
11 //             for crossing at address pointed to by i.
12 void find_next_undercrossing(int minUndercrossing,
13                              std::list<knot_crossing>::iterator& min_cross,
14                              std::string &knot_desc,
15                              std::list<knot_edge>& edges,
16                              std::list<knot_crossing>& crossings,
17                              int totalCount){
18
19     // If minUndercrossing exceeds the number of strands,
20     // need to start back at 1.
21     // (Edges are labeled in canonical order.)
22     if (minUndercrossing > edges.size() ){
23         minUndercrossing = 1;
24     }
25
26     bool found = false;
27
28     // Search over the list of crossings for one with the next label.
29     for (std::list<knot_crossing>::iterator iter = crossings.begin();
```

```
30          iter != crossings.end(); ++iter){
31
32          if ( (*iter).crossing_elements[0] == minUndercrossing){
33              minUndercrossing = iter->crossing_elements[2];
34              min_cross = iter;
35              found = true;
36              // As soon as we find the strand, we're done searching.
37              break;
38          }
39
40
41      }
42
43      // If an undercrossing with label minUndercrossing is not found,
44      // iterate minUndercrossing once and search.
45      // (Using the fact that edges are in canonical order.)
46      if (found == false){
47          find_next_undercrossing(minUndercrossing+1, min_cross, knot_desc,
48                                  edges, crossings, totalCount);
49      }
50
51      else{
52          // Append the crosssing to the screen.
53          write_knot_description(min_cross ,knot_desc, totalCount);
54          // Set up and search for next crossing.
55          minUndercrossing = (*min_cross).crossing_elements[2];
56          if (totalCount < crossings.size()){
57              find_next_undercrossing(minUndercrossing, min_cross,
58                                      knot_desc, edges, crossings,
59                                      totalCount);
60          }
61      }
62
63  }
```

The controller for these two functions is the following function.

```
1   // FUNCTION:   orderedCrossingListing
2   // SCOPE:      Objective C UIView.
3   // PARAMETERS: none.
4   //   GLOBAL PARAMETERS: discrete_edges *(list of edges)*
5   //    all_crossings  *(list of crossings)*
6   //    minUndercrossingLocation *(location of smallest undercrossing label)*
7   //                          FOUND IN: shift_crossing_elements
8   // RETURNS:    no value.
9   // PURPOSE:    Finds first crossing to kick off knot_description string.
10  -(void) orderedCrossingListing{
11
12      if(discrete_edges.size()>0){
13
14          counter = 0;
15          std::string knot_description_full = "";
16          int minUndercrossing =
17              minUndercrossingLocation->crossing_elements[0];
18          minUndercrossingLocation = all_crossings.begin();
```

```
19
20          for (std::list<knot_crossing>::iterator
21          iter = all_crossings.begin();
22               iter != all_crossings.end(); ++iter){
23
24              if ( iter -> crossing_elements[0] <= minUndercrossing){
25                  minUndercrossing = iter->crossing_elements[0];
26                  minUndercrossingLocation = iter;
27              }
28          }
29
30
31          // First call to find_next_undercrossing:
32          find_next_undercrossing(minUndercrossing, minUndercrossingLocation,
33                                  knot_description_full,
34                                  discrete_edges, all_crossings, 0);
35
36          // find_next_undercrossing will iterate and
37          // modify knot_description_full until
38          // the label is complete, i.e., all X[,,,] are stored in
39          // knot_description_full.
40          knot_description = [NSString
41                              stringWithCString:knot_description_full.c_str()
42                              encoding:[NSString defaultCStringEncoding]];
43          knot_description = [NSString stringWithFormat:@"%@%@%@",
44                              @"PD[" , knot_description, @"]"];
45          [self showEmail];
46      }
47      else{
48          [self callWarning:@"Error! Knot not drawn."];
49      }
50  }
```

3.4.2 Email Delegation

The majority of the code is written inside of a `UIView` class; however, email delegates can only be assigned to `UIViewControllers`. Though the `UIView` sits atop the `UIViewController`, or the `UIView` is called from and physically layered on top of the `UIViewController` graphical object, the `UIView` and `UIViewController` are distinct objects in memory. Thus, one cannot access the functions in a `UIViewController` directly from UIView, nor is `UIView` treated as an inherited class.
Though `UIView` and `UIViewControllers` are distinct objects in memory, fear not! There is a global, abstract space that is shared by all objects in an Xcode project. It is a functional space that may also contain objects; this is a result of object-oriented programming. `NSNotifications` is a class of function types that allow the programmer the liberty to trigger processes from one UI class to another. For our project, we set the base `UIViewController` to be an observer.

```
1  @implementation ViewControllerP
2  //...
3  [[NSNotificationCenter defaultCenter] addObserver:self
4          selector:@selector(sendMail:) name:@"showMailComposer" object:nil];
5  //...
6  @end
```

Then within the UIView class:

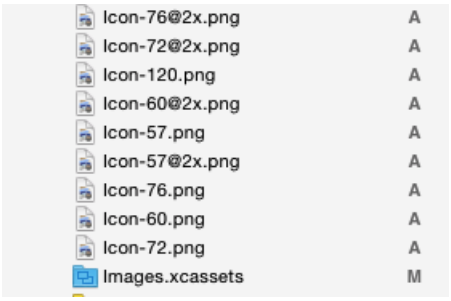
```
@implementation LinearInterpView
//...
- (IBAction)showEmail{
    [[NSNotificationCenter defaultCenter]
            postNotificationName: @"showMailComposer"
            object:knot_description];
}
//...
@end
```

Its trigger is the following `UIButton`, instantiated within the initial call to the `UIView`.

```
// FUNCTION:    initWithFrame
// SCOPE:       Objective C UIView.
// PURPOSE:     Initial setup of the frame.
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
       //...
        listButton = [UIButton buttonWithType:UIButtonTypeCustom];
        listButton = [[UIButton alloc] init];
        listButton = [UIButton buttonWithType:UIButtonTypeRoundedRect ];
        listButton.frame = CGRectMake(finishButton.frame.origin.x ,
                        eraseButton.frame.origin.y +
                        std::abs(eraseButton.frame.size.height),
                        150, 75);
        listButton.tag = 2;
        [listButton setTitle:@"List" forState:UIControlStateNormal];
        [listButton addTarget:self action:@selector(orderedCrossingListing)
             forControlEvents:UIControlEventTouchUpInside];
        //...
        }
}
```

3.5 Icons

One part of iOS coding is the icon images. The image of the figure-eight knot $(4, 1)$ that is used comes from Wikimedia Commons. In order to register an app, a number of icon sizes must be used and put into the project. The following image is used, and beside it is the list of copies of it scaled to their respective image sizes as depicted by the name. This naming is consistent with and required for Apple iOS programs.

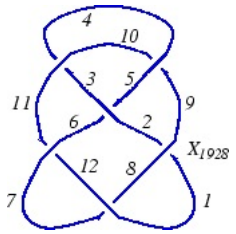


Chapter 4

Checkerboard Graphs

Writing a package that produces the checkerboard graphs of knots was an initial step into trying to bridge the gap between knot theory and computer programs. Sage has a number of developmental tools for abstract mathematics, so the package was written in Python, in hopes that it may be contributed to the Sage library. The package is written as a series of function files that will be detailed below.

This code was developed assuming that the user supplies a knot following the convention detailed in Mathematica's KnotTheory, which we use as a standard. Again recall the image:



$$X_{1928}X_{3,10,4,11}X_{5362}X_{7,1,8,12}X_{9,4,10,5}X_{11,7,12,6}.$$

Note that by inputting the crossing elements counterclockwise, components of a surface are discernible from the representation. For instance with $[1, 9, 2, 8]$, it's true that the pairs $[1, 9]$, $[9, 2]$, $[2, 8]$, and $[8, 1]$ are elements of edge lists of surfaces. This fact is leveraged in order to understand the graph.

The program assumes the user inputs an array of crossings. Here, it is assumed the user supplies an array of a 4-element array of knot strands as integers. Contrasting with Mathematica's KnotTheory, the syntax from Mathematica `PD[X[1,2,3,4], X[5, 6, 1, 7], ...]` will instead be only the integer elements, i.e. [[1,2,3,4], [5,6,1,7], ...]. One may input an array of these such arrays, so that the code can handle multiple knots in one run of the program. However, the only interface to Mathematica that is used is for the final output of the graph.

4.1 Main Program

Below is the main program (Checkerboard_Graph.py) and the function calls written explicitly. R is assumed to be an array of knots as explained above.

```
def CheckerboardGraph(R):

    list_surfaces = FindListSurfaces(R);

    for s in list_surfaces:
        s.sort();

    maxlen = 0;
```

```
    for k in list_surfaces:
        if len(k) > maxlen:
            maxlen = len(k);
            list_surfaces.remove(k);
            list_surfaces.insert(0, k);

    CrossingTypes = FindPositiveNegativeCrossings(R);

    positiveNodes = CrossingTypes[0];
    negativeNodes = CrossingTypes[1];

    SurfaceData = FindPositiveNegativeSurfaces(list_surfaces,
                                               positiveNodes,
                                               negativeNodes);

    positiveNodeSurfaces = SurfaceData[0];
    positiveNodeCrossings = SurfaceData[1];
    negativeNodeSurfaces = SurfaceData[2];
    negativeNodeCrossings = SurfaceData[3];

    MathematicaGraphStrings_PosNegGraphs =
    ExportPositiveNegativeToGraph(list_surfaces,
                          positiveNodeCrossings, positiveNodeSurfaces,
                          negativeNodeCrossings, negativeNodeSurfaces);

    list_adjacencies=[];
    for l in list_surfaces:
       list_adjacencies.append([]);
    n = len(list_surfaces);
    list_positives=[];
    list_negatives=[];
    list_connections=[];

    for i in range(0,n):
       list_positives.append([]);
       list_negatives.append([]);
       list_connections.append([]);

    for p in positiveNodeCrossings:
       list_positives[p[0]].append(p[1]);
       list_positives[p[1]].append(p[0]);

    for p in negativeNodeCrossings:
       list_negatives[p[0]].append(p[1]);
       list_negatives[p[1]].append(p[0]);

    for p in range(0,n):
       list_connections[p].extend(list_positives[p]);
       list_connections[p].extend(list_negatives[p]);

    for ind in range(0,n):
       l = list_surfaces[ind]; # start at a fixed surface l
       for i in range(0, n):
          if i != ind: #for every other surface
```

```
63              k = list_surfaces[i]; # fix the other surface
64              # Find common edge listings of l and k:
65              commonEdges = set(l).intersection(k);
66              if len(commonEdges) != 0: #if common edges
67                list_adjacencies[ind].append(i); #have adjacent surface.
68 ###Opposite surfaces:
69 #{connections} \ {adjacencies}
70    duimian=[];
71    for ind in range(0,n):
72        l = list_connections[ind];
73        v = list_adjacencies[ind];
74        # Find the common elements in connections with adjacencies
75        k = set(l).intersection(set(v));
76        opposites = [x for x in list_connections[ind] if x not in k]
77        duimian.append(opposites);
78 ##########################
79 ###Multiplicity
80 # Here we seek to find the number of edges between the same two vertices.
81    sizeList = len(duimian);
82    m = len(list_surfaces);
83    multiplicity = [];
84    smallList = [];
85
86    for v in range(0, sizeList):
87        # Obtain a null vector of size = number of surfaces (row v)
88        for r in range(0,m):
89            smallList.append(0);
90        for k in duimian[v]: #kth element of row v will be incrememnted by 1
91            smallList[k] = smallList[k] + 1;
92        multiplicity.append(smallList);
93        smallList = [];
94 ##########################Adjacency Matrix
95    coloring = [];
96    for l in list_adjacencies:
97       coloring.append(0);
98    coloring[0] = 'white';
99    i = 0;
100   coloring = BlackWhiteColor(i, list_adjacencies, coloring);
101
102   for i in range(0, n):
103      if coloring[i] != 'white':
104         coloring[i] = 'black';
105 ##########################################################
106 #White Graph
107   stringGraph = 'GraphPlot[{';
108   smallString ='';
109   count_white = 1;
110   x_old = -1;#base reference. will use to not have repeats
111   G = Graph();
112   for l in list_surfaces:
113      G.add_vertices(l);
114
115   for l in range(0,n):
116      if coloring[l] == 'white': #if white node
```

```
            count_white = count_white + 1;
            x_old = -1;
            # Look through list of connections
            for x in range(0,len(duimian[l])):
               # Finding out if haven't already dealt with node
               if duimian[l][x] > l and duimian[l][x] != x_old:
                   smallString = ' " ' + str(list_surfaces[l]) + ' " ->';
                   k = duimian[l][x];
                   x_old = k;
                   mult = multiplicity[l][k];
                   smallString = smallString + ' " ' +
                          str(list_surfaces[duimian[l][x]]) + ' " ';
                   BiggerString = smallString + ',{ ' + smallString +
                           ' , ' + str(mult) + '},';
                   stringGraph = stringGraph + BiggerString;


   stringGraph = stringGraph[:-1]; #take off last comma
   Checkerboard_stringGraph = stringGraph +
            '}, MultiedgeStyle -> False,EdgeRenderingFunction ->
            ({Line[#1], If[#3 === None, {},
            Text[#3, Mean[#1],Background -> White]]} &),
            VertexLabeling -> True, PlotLabel->"White Nodes"]';

   w = "Export[\"test.png\"," + Checkerboard_stringGraph + "]";
   mathematica(w); # NOTE: This is Sage's interface to Mathematica
############################################################
#Black Graph
   stringGraph = 'GraphPlot[{';
   smallString = '';
   count_black = 1;
   x_old = -1;
   for c in range(0, len(coloring)):
      if coloring[c] == 'black':
          count_black = count_black + 1;
          x_old = -1;
          for x in range(0,len(duimian[c])):

             if duimian[c][x] > c and duimian[c][x] != x_old:
             # (If <, we've already added it to graph.
             # If =, don't want to consider it.)
                smallString = ' " ' + str(list_surfaces[c]) + ' " ->';
                k = duimian[c][x];
                x_old = k;
                mult = multiplicity[c][k];
                smallString = smallString + ' " ' +
                            str(list_surfaces[duimian[c][x]]) + ' " ';
                BiggerString = smallString + ',{' + smallString +
                             ' , ' + str(mult) + '},';
                stringGraph = stringGraph + BiggerString;

   #v = len(stringGraph);
   stringGraph = stringGraph[:-1];#take string, subtract off last comma
   BlackCheckerboardGraph = stringGraph + '}, MultiedgeStyle -> False,
```

```
                    EdgeRenderingFunction->({Line[#1], If[#3 === None, {},
                    Text[#3, Mean[#1], Background -> White]]} &),
                    VertexLabeling -> True, PlotLabel->"Black Nodes"]';
   show(mathematica(BlackCheckerboardGraph));

#for b in B:
x = raw_input("Please input a list of edges\n>>");
[x.strip() for x in my_string.split(',')]
[int(t) for t in x];

CheckerboardGraph(x);
```

4.2 Finding Surfaces

Surfaces are built from the Mathematica representation.

```
# Function: findPair
# Output:    element index in set S that has element n
# Use:
def findPair(n, S): # Input node n in set of sets S
    for s in S:
        for z in range(0,4):
            v = s[z];
            if v == n:
               return s; # Return the SET that the node is in

   n = len(R);
   for index in range(0,n):
      i = R[index];
      #iterate over each element in each element of R:
      for j in range(0, FourPlanarBound):
         x = i[j]; # Take jth element of the crossing list [a,b,c,d]
         if j < 3: # Find its neighbor, (counter-clockwise pengyou)
            y = i[j+1];
         else: # At end of list in R
            y = i[0];
         surface.append(x); #start surface containing x.
         alreadyHave = False; # Used to determine if surface is unique.
         # Must search rest of input R for neighbors.
         Rmod = [k for k in R if set(k) != set(i)]; # R\{set we're in}
         NodeAdjacencyR.append(x); # Find adjacencies with crossings

         # While not back at original node (cycle)
         while y != x:
            surface.append(y);
            setPair = findPair(y, Rmod); # Find the next set that has y
            Rmod = []; j = 0;
            for ind in range(0, FourPlanarBound):
               if setPair[ind] == y: # Find element that is y
                  if ind < FourPlanarBound - 1:
                     y = setPair[ind+1];
                     break;
            else: # If statement's else belongs to the for loop
```

```
                y = setPair[0];
            Rmod = [k for k in R if set(k) != set(setPair)]; # The new R\S
        #end while loop

        for l in list_surfaces:
            if set(l) == set(surface):
                alreadyHave = True;
        if alreadyHave == False:
            list_surfaces.append(surface);
        surface = [];

    return list_surfaces;
```

4.3 Positive/Negative Surfaces and Crossings

```
def FindPositiveNegativeCrossings(R):
##########################################################################
###Finding Positive/Negative Crossings:
#This heavily relies on the implementation of the PD Diagrams in
#Mathematica's KnotTheory Package. It is tailored to the imposed
#process of input for KnotTheory's PD Diagrams.

    #NUTS AND BOLTS:
    n = len(R); #length of inputs
    positiveNodes = [];  # connected surface parts via positive crossings
    negativeNodes = []; # connected surface parts via negative crossings
    edge = []; #temporary for storing the edges
    node = []; #temporary for storing the nodes
    list_edges =[]; #container for the nodes and edges

##############################################
#FOR loop over the inputted crossing list
#
#   All crossings are inputted counter-clockwise from a starting under-
#crossing. From the first undeercrossing, we follow the base strand to the
#next undercrossing and store the counter-clockwise labels of the crossing.
#Rinse and repeat, until we end up back at the original, first crossing.
#
#   We take advantage of this knowledge. Note that since we impose a
#synthetic direction on the knot by following along one strand.
#This direction is NOT necessarily part of the knot, as we could choose
#any undercrossing to start from in any direction.
#
#   Since we know that we input each undercrossing counter-clockwise,
#  we note that we can almost think of the relationship between the strands
#in a given crossing as a relaxation of parameters. Further, the connection
#between the surfaces that touch each crossing have an actual relationship
#from the crossing-- the positive or negative connection between the
#surfaces.
#
#   We must define some type of crossing as positive, and the opposite
#as negative. Assume we have surfaces a,b,c,d, all stored in a container
```

```
#as nodes.
#  Let
#         \ a  /
#          \  /
#           \          be a  'positive' crossing between a and b
#          / \
#         / b \
#        /     \
#
#  And let
#         \ c  /
#          \  /
#            /         be a  'negative' crossing between c and d
#           / \
#          / d \
#         /     \
#
#    We note that the negative crossings are just the opposite of the
# positive crossings for a given crossing. Indeed, looking again at the
# positive crossing,
#
#         \ a  /
#          \  /
#           \
#          / \
#         / b \
#        /     \
#
# c,d could in fact be the surfaces on the other sides!
#
#         \ a  /
#          \  /
#     c     \    d
#          / \
#         / b \
#        /     \
#
#   Further, we know that we input into the computer a list of crossings,
#defined by the method of starting from an undercrossing, inputting the
#crossing edge labels  counter-clockwise, and going to the next
#undercrossing.
#
#   Looking again at our crossing, we actually have
#         \ a  /
#         4\  /3
#    c      \     d
#           / \
#        1/ b \2
#        /     \
#
#and Computer 'knows' this. We inputted as [1,2,3,4], and all programs are
#treated and handled assuming this method of input.
#
#    Now, we wish to understand the existence and types of connections
```

```
#between surface nodes. We create an edge between all surfaces that are
#directly opposite a given surface, as determined by looking at a crossing.
#So, in the above crossing, there exists an edge between a and b, and there
#exists an edge between c and d.
#
#    Which type of edge connects these surfaces is determined by the above
#definitions of positive and negative. We note then the following:
#    [i]  Edge {a,b} is positive.
#    [ii] Edge {c,d} is negative.
#
#    Crossing input: [1,2,3,4]
#
#    We make the following observations:
#    [i]  An edge {[1,2],[4,3]} is a positive crossing edge.
#    [ii] An edge {[2,3],[4,1]} is a negative crossing edge.
#    [iii]
#       (a) 1,2 are elements of the list of edges that constitute surface b
#       (b) 4,3 are elements of the list of edges that constitute surface a
#       (c) 2,3 are elements of the list of edges that constitute surface d
#       (d) 1,4 are elements of the list of edges that constitute surface c
#
#   Given the list of all surfaces, Computer can then match up the element
#combinations noted in [iii] with each element combination's own surface.
#This list of connected surfaces, then, become a list of surfaces'
#connection types.
#
# Computer's observations become:
#   [i]  Edge {a,b} is a positive crossing edge.
#   [ii] Edge {c,d} is a negative crossing edge.
# as desired.
#
#  Pseudo-code:
#  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#  Given list of crossings R, list of surfaces S:
#  positiveNodes = {};
#  negativeNodes = {};
#  for each r in R do
#      //Handeling positive nodes:
#      base_node1 <- {first element in r, second element in r}
#      base_node2 <- {last element in r , third elemenet in r}
#      edge <- {base_node1, base_node2}
#      add edge to positiveNodes
#
#      //Handeling Negative nodes:
#      base_node1 <- {second element in r, third element in r}
#      base_node2 <- {fourth element in r, first element in r}
#      edge <- {base_node1, base_node2}
#      add edge to negativeNodes
#  end
#
#  //Finding surfaces:
#  all_connections = {}
#  for each x in positiveNodes do
#     local_surface_connection = {}
```

```
#     base1 <- first element in x
#     base2 <- second element in x
#     for each s in S do
#          if base1 is contained in s then
#             add s to local_surface_connection
#          if base2 is contained in s then
#             add s to local_surface_connection
#     end
#     add local_surface_connection to all_connections
#   end
#   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#   Since each inputted crossing is an undercrossing that is inputted
#counter-clockwise from the initial strand, we can exactly follow the
#afore-mentioned method of observation for the generic crossing,
#applied in general to all inputted crossings.
   for i in R:
       node.append(i[0]);
       node.append(i[1]);
       n1 = node; node = [];
       node.append(i[2]);
       node.append(i[3]);
       n2 = node; node = [];
       edge.append(n1); edge.append(n2);
       positiveNodes.append(edge);
       edge = [];

       node.append(i[1]);
       node.append(i[2]);
       n1 = node; node = [];
       node.append(i[0]);
       node.append(i[3]);
       n2 = node; node = [];
       edge.append(n1); edge.append(n2);
       negativeNodes.append(edge);
       edge = [];

   structures = [];
   structures.append(positiveNodes);
   structures.append(negativeNodes);
   return structures;
```

```
def FindPositiveNegativeSurfaces(list_surfaces,
                          positiveNodes, negativeNodes):
#################################################################
#We have a list of edges positiveNodes and negativeNodes that connect
#surfaces labeling one type of crossing as a 'positive' crossing
#and the other as a 'negative' crossing. This is a collection
#of pieces of a surface.
#   e.g. Given a surface with edges [a,b,c,d] connected to
#   a surface [w,x,y,z], we have defined an edge between
#   some 2 elements of each, say [a,b] with [y,z]. [a,b]<->[y,z]
#
#PROBLEM: We must show that [a,b,c,d] is thus connected to
#[w,x,y,z] using this relation [a,b] <-> [y,z]
```

```
#
#Below we transverse the master container of surfaces to extract
#this data for each type of nodes, both 'positive' and 'negative.'
#
##################
#Basic idea:
#STEP 1: Find the surfaces.
#For each edge [a,b]<->[y,z]
#   Look through list_surfaces //master container of surfaces
#     IF FOUND [a,b] as a subset of surface
#         Add surface to Adj structure
#     IF FOUND [y,z] as a subset of surface
#         Add surface to Adj structure
#
#   //Okay, we've got them. Now store the edges.
#   Add to SurfaceAdj the two surfaces together Adj[0]<->Adj[1]
#   //Rinse and repeat for each edge.
#########################################################################
#########################################################################
#POSITIVE CROSSINGS:
#   Rinse and repeat. Same as the negative crossings.
#   They are just data structures!
#
   #NUTS AND BOLTS:
   MasterList = [];
   SurfaceAdj = [];
   SurfaceAdjNum = []
   Adj = [];
   AdjNum=[];
   alreadyIn = False;

   #Finding surfaces:
   for i in positiveNodes:
       alreadyIn=False;
       for jind in range(0,len(list_surfaces)):
           j = list_surfaces[jind];
           if set(i[0]).issubset(j):
               Adj.append(list(j));
               AdjNum.append(jind);
           if set(i[1]).issubset(j):
               Adj.append(list(j));
               AdjNum.append(jind);
       if alreadyIn == False:
           SurfaceAdj.append(Adj);
           SurfaceAdjNum.append(AdjNum);
       alreadyIn = False;
       Adj = [];
       AdjNum=[];

   #index 0 -> positive nodes, 1-> positive surfaces
   MasterList.append(SurfaceAdj);
   MasterList.append(SurfaceAdjNum);

   #NUTS AND BOLTS [[clearing]]:
```

```
    SurfaceAdj = [];
    SurfaceAdjNum = []
    Adj = [];
    AdjNum=[];
    alreadyIn = False;

    # Start with negativeNodes (gotta end on a positive note! ;) )
    for i in negativeNodes:
     # Transversing the master container of surfaces, list_surfaces
        for jind in range(0,len(list_surfaces)):
            j = list_surfaces[jind];

            if set(i[0]).issubset(j):
            # i.e. if edge list is a subset of the edges
            # of any surface, we found it. Add to the
            # new list that makes the adjacency between
             # the two surfaces
                Adj.append(list(j));
                AdjNum.append(jind);
            if set(i[1]).issubset(j):
                Adj.append(list(j));
                AdjNum.append(jind);
        SurfaceAdj.append(Adj);
        SurfaceAdjNum.append(AdjNum);
        Adj = [];
        AdjNum=[];

    # index 2-> negative nodes and 3 -> negative surfaces
    MasterList.append(SurfaceAdj);
    MasterList.append(SurfaceAdjNum);

    return MasterList;
```

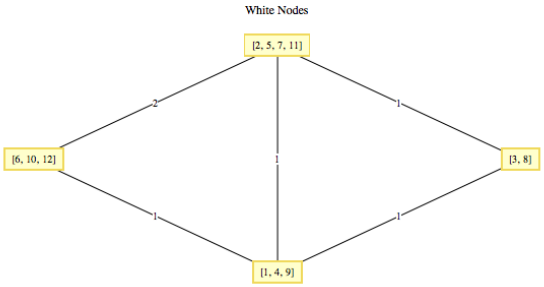
4.4 Exporting to Mathematica

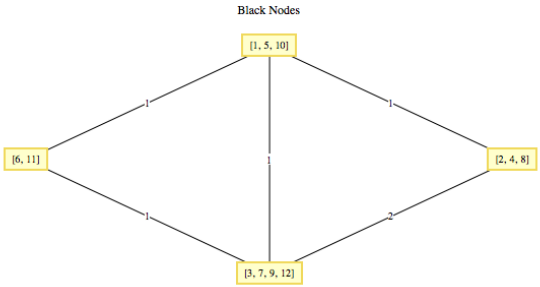
```
def ExportPositiveNegativeToGraph(list_surfaces,
                  positiveNodeCrossings,  positiveNodeSurfaces,
                  negativeNodeCrossings, negativeNodeSurfaces):
#########################################################################
#PROBLEM: Convert the list of edges into a string for Mathematica to read
#and output to a clean Checkerboard graph
    stringGraph = 'GraphPlot[{';
    smallString ='';
    for r in range(0, len(negativeNodeCrossings)):
       s = negativeNodeCrossings[r];
       smallString = smallString  + str(s[0]) + '->';
       for x in range (1, len(s)):
           smallString = smallString + str(s[x]);
           stringGraph = stringGraph + smallString;
           if r != len(negativeNodeCrossings) - 1:
              stringGraph = stringGraph + ',';
           smallString = str(s[0]) + '->';
```

```
        smallString = '';
    stringGraph = stringGraph + '}, VertexLabeling -> True]';

    NegativeCrossings = stringGraph;

    stringGraph = 'GraphPlot[{';
    smallString ='';
    for r in range(0, len(positiveNodeCrossings)):
        s = positiveNodeCrossings[r];
        smallString = smallString  + str(s[0]) + '->';
        for x in range (1, len(s)):
            smallString = smallString + str(s[x]);
            stringGraph = stringGraph + smallString;
            if r != len(positiveNodeCrossings) - 1:
                stringGraph = stringGraph + ',';
            smallString = str(s[0]) + '->';
        smallString = '';
    stringGraph = stringGraph + '}, VertexLabeling -> True]';

    PositiveCrossings = stringGraph;

    PosNegGraphs = [];
    PosNegGraphs.append(PositiveCrossings);
    PosNegGraphs.append(NegativeCrossings);
    return PosNegGraphs;
```

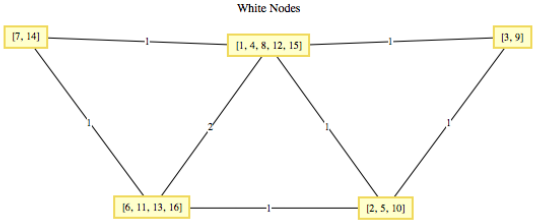
4.5 Sample Output

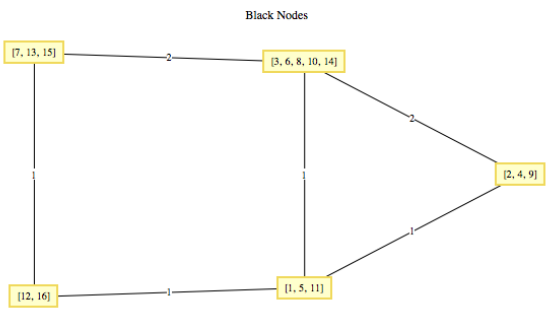
This code has been tested on all of the knots in Mathematica's KnotTheory package library. Some 477 graphs were produced; below is a sampling of the results. Note the multiplicity of the crossings is printed on the edges. First, knot (6,3) (i.e., the third in the set of knots with six crossings):





Knot (8,14):





Future Work

Next steps would be to develop algorithms for the computer to return whether a knot has certain properties. For example, the fundamental group may be outputted to the user in the email message body, along with the PD knot representation. Additionally, if iOS were integrated with Sage, then we could incorporate the checkerboard graph code into the app. The main restriction is that the Python code uses matplotlib, a library for plotting in Python, and the type of graphs that matplotlib handles and outputs are not supported by iOS as of yet, to the best of my knowledge.

Appendices

Appendix A

Terms

1. `Alexander Polynomial`[1] a knot invariant that assigns a polynomial with integer coefficients to each knot type
2. `class`[11] a user-defined C++ data type
3. `CGPoint`[5] a class containing (x, y) coordinates of a point
4. `double`[11] a type representing 8 bits in memory
5. `float`[11] a type representing 4 bits in memory ("short `double`")
6. `Fundamental Group`[6] the group representation of a knot
7. `int`[11] a type representing 4 bits in memory
8. `Jones Polynomial`[1] a polynomial representation of a knot assigned using coiling properties
9. `knot`[6] a smooth embedding of a circle ($\mathbb{S}^1$) into $\mathbb{R}^3$
10. `knot invariant`[1] a property of a knot is the same for equivalent knots
11. `list`[11] a C++ container that holds data and allows for no random-access memory
12. `member functions`[11] functions that are defined within a `class`
13. `object`[11] a piece of memory that holds a value of a given type
14. `private`[11] a keyword setting objects within a `class` to be accessible to it and its protected members
15. `public`[11] a keyword setting objects to be accessible from all parts of a code
16. `Reidemeister Moves`[1] three transformations, twisting in one direction, shifting one strand over another, and moving a strand over a crossing, that preserve the topological structure of a knot
17. `struct`[11] a user-defined C data type
18. `type`[11] defines a set of possible values and operations for an object
19. `UIBezierPath`[5] a class consisting of straight and curved line segments
20. `UIView`[5] the view layer that can be placed over a UIViewController's screen
21. `UIVewController`[5] a fundamental object- and view-managing tool for iOS apps

Appendix B

Objective C/C++ Code

Included below are functions used in the code that were not described in the paper.
The UIViewController:

```
#import <QuartzCore/QuartzCore.h>
#import <UIKit/UIKit.h>
#import <MessageUI/MessageUI.h>
#import "LinearInterpView.h"
#import <MessageUI/MFMailComposeViewController.h>

@interface ViewControllerP:UIViewController
                           <MFMailComposeViewControllerDelegate>

@property (strong,nonatomic) LinearInterpView *InterpView;
@property (weak, nonatomic) IBOutlet UIButton *emailButton;

- (IBAction)outputPath:(id)sender;
- (void)    sendMail:(NSNotification *)notification;
- (BOOL)    shouldAutorotateToInterfaceOrientation:
             (UIInterfaceOrientation)interfaceOrientation;
@end
```

The UIViewController implementation file:

```
#import "ViewControllerP.h"

@interface ViewControllerP ()
@end

@implementation ViewControllerP
@synthesize InterpView, emailButton;

- (id)initWithNibName:(NSString *)nibNameOrNil
     bundle:(NSBundle *)nibBundleOrNil
{
    self = [super initWithNibName:nibNameOrNil
          bundle:nibBundleOrNil];
    if (self) {}
    return self;
}

```

```
-(void) outputPath:(id)sender
{    self.view.clearsContextBeforeDrawing = TRUE;
}

-(void)sendMail:(NSNotification *)notification
{
    NSString *knot_desc = [notification object];
    MFMailComposeViewController *mailController =
         [[MFMailComposeViewController alloc] init];
    mailController.mailComposeDelegate = self;
    [mailController setSubject:@"Subject"];
    [mailController setMessageBody:knot_desc isHTML:NO];
    [mailController setToRecipients:[NSArray arrayWithObject:@""]];

    // Present mail view controller on screen
    [self.view.window.rootViewController
        presentViewController:mailController animated:YES completion:NULL];

}

- (void) mailComposeController:(MFMailComposeViewController *)controller
       didFinishWithResult:(MFMailComposeResult)result
       error:(NSError *)error
{
    switch (result)
    {
        case MFMailComposeResultCancelled:
            NSLog(@"Mail cancelled");
            break;
        case MFMailComposeResultSaved:
            NSLog(@"Mail saved");
            break;
        case MFMailComposeResultSent:
            NSLog(@"Mail sent");
            break;
        case MFMailComposeResultFailed:
            NSLog(@"Mail sent failure: %@", [error localizedDescription]);
            break;
        default:
            break;
    }
    // Close the mail interface.
    [self dismissViewControllerAnimated:YES completion:NULL];
}


- (void)viewDidLoad
{
    [super viewDidLoad];
    [[NSNotificationCenter defaultCenter] addObserver:self
                selector:@selector(sendMail:)
                name:@"showMailComposer" object:nil];
     LinearInterpView *Drawview = [[LinearInterpView alloc]
                                   initWithFrame:self.view.bounds];
```

```
72
73      // Handle rotation, so that UIButtons reposition as desired.
74      [Drawview setAutoresizesSubviews:YES];
75      [Drawview setAutoresizingMask:
76               UIViewAutoresizingFlexibleLeftMargin |
77               UIViewAutoresizingFlexibleHeight];
78      Drawview.userInteractionEnabled = YES;
79      [self.view addSubview:Drawview];
80  }
81
82  - (void)didReceiveMemoryWarning
83  {  [super didReceiveMemoryWarning]; }
84
85  - (BOOL)shouldAutorotateToInterfaceOrientation:
86              (UIInterfaceOrientation)interfaceOrientation {
87       return YES;  // Return YES for supported orientations
88  }
89  @end
```

The UIView header file:

```
1  #import <UIKit/UIKit.h>
2  #import <QuartzCore/QuartzCore.h>
3  #import <Foundation/Foundation.h>
4  #import <MessageUI/MessageUI.h>
5  #import <MessageUI/MFMailComposeViewController.h>
6
7  @interface LinearInterpView : UIView
8  {
9      UIView *InterpView;
10 }
11 @property (strong, nonatomic) IBOutlet UIView   *InterpView;
12 @property (strong, nonatomic) IBOutlet UIButton *finishButton;
13 @property (strong, nonatomic) IBOutlet UIButton *eraseButton;
14 @property (strong, nonatomic) IBOutlet UIButton *listButton;
15 @property (strong, nonatomic) IBOutlet UILabel  *errorLabel;
16 @property (strong, nonatomic) IBOutlet UIButton *emailButton;
17
18 -(IBAction)showEmail;
19 -(IBAction)switchCrossings:       (id)sender;
20 -(IBAction)buttonTouched_crossing:(id)sender;
21 -(void)switch_crossings_label:(int)cross_num
22          touch_loc:(CGPoint)touchLocationInView;
23 -(void) erase;
24 -(void) callWarning:(NSString *)errorMessage;
25 -(void) call_GetWorkKnot;
26 -(void) GetWorkKnot;
27 -(void) orderedCrossingListing;
28 -(void) draw_circle_about_crossings:(int)width height:(int)height;
29 -(void) shift_crossing_elements;
30 -(void) determine_minimum_edge;
31 -(void)willAnimateRotationToInterfaceOrientation:
32          (UIInterfaceOrientation)toInterfaceOrientation
33          duration:(NSTimeInterval)duration;
34 @end
```

Auxiliary functions:

```
// FUNCTION:   showEmail
// SCOPE:      Objective C UIView.
// PARAMETERS: none
// RETURNS:    no value.
// PURPOSE:    Push notification (in global space) to call email delegate
- (IBAction)showEmail{
    [[NSNotificationCenter defaultCenter]
        postNotificationName:@"showMailComposer"
        object:knot_description];

}


// FUNCTION:   resetView
// SCOPE:      Objective C UIView.
// PARAMETERS: none
//   GLOBAL PARAMETERS: all
// RETURNS:    no value.
// PURPOSE:    Resets all values if app temrinated
//     *Necessary for direct interface to iOS app.
//      Need an Objective C function to call for erase.
-(IBAction) resetView{

    path = nil;

    incrementalImage = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    path = [UIBezierPath bezierPath];

    discrete_edges.clear();
    KnotEdge.clear();
    pathPoints.clear();
    liftPoints.clear();

    all_crossings.clear();
    crossing.clear();
    beginTouchPoints.clear();
    KnotEdge.clear();
    intersectPoints.clear();
    smallLP.clear();


    knot_description = @"";
    knot_description_c = "";
    unshifted               = true;
    switch_crossing_control = false;
    touch_crossing          = false;
    gapWithoutCrossing      = false;
    crossing_firstTouch     = true;

    edgeCount = -1;
```

```
52      counter    = -1;
53      crossingElement = -1;
54
55
56      // Remove the subviews added for the previously drawn knot.
57      // Tags 0-3 are used for finishButton, eraseButton, etc.
58      for (UIView *subview in self.subviews)
59      {
60          if(subview.tag != 1 && subview.tag != 2 &&
61             subview.tag != 0 && subview.tag != 3)
62              [subview removeFromSuperview];
63      }
64      for (UIView *subview in self.superview.subviews)
65      {
66          if(subview.tag != 0 )
67              [subview removeFromSuperview];
68      }
69
70      errorLabel.hidden = true;
71
72      [self setNeedsDisplay];
73  }
74
75
76  // Procure an error message to the screen.
77  -(void) callWarning:(NSString *)errorMessage{
78      errorLabel.text = errorMessage;
79      errorLabel.hidden = false;
80  }
81
82
83  // The following block are auxiliary functions.
84  bool FoundPointEqual(CGPoint pt1, CGPoint pt2){
85      return CGPointEqualToPoint(pt1, pt2);
86  }
87
88  double dot(CGPoint p, CGPoint q){
89      CGPoint pnew;
90      pnew.x = p.x * q.x;
91      pnew.y = p.y * q.y;
92
93      return pnew.x + pnew.y;
94  }
95
96  CGPoint vectorSubtraction(CGPoint p, CGPoint q){
97      CGPoint pnew;
98      pnew.x = p.x - q.x;
99      pnew.y = p.y - q.y;
100     return pnew;
101 }
102
103 CGPoint multiply(double r, CGPoint p){
104     CGPoint pnew;
105     pnew.x = r*p.x;
```

```
    pnew.y = r*p.y;
    return pnew;
}
```

The default initial function calls:

```
// FUNCTION:   initWithFrame
// SCOPE:      Objective C UIView.
// PURPOSE:    Initial setup of the frame.
- (id)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {

        [self setBackgroundColor:[UIColor whiteColor]];
        [[NSNotificationCenter defaultCenter] addObserver:self
                                    selector:@selector(erase)
                                    name:@"Did Terminate" object:nil];
        finishButton = [UIButton buttonWithType:UIButtonTypeCustom];
        finishButton = [[UIButton alloc] init];
        finishButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
        finishButton.frame = CGRectMake((frame.origin.x +
                              0.85*frame.size.width),
                              frame.origin.y + 50,
                              150, 75);
        [finishButton setTitle:@"Get Knot" forState:UIControlStateNormal];
        finishButton.tag = 1;
        [finishButton addTarget:self action:@selector(call_GetWorkKnot)
               forControlEvents:UIControlEventTouchUpInside];

        eraseButton = [UIButton buttonWithType:UIButtonTypeCustom];
        eraseButton = [[UIButton alloc] init ];
        eraseButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
        eraseButton.frame = CGRectMake(finishButton.frame.origin.x ,
                      finishButton.frame.origin.y +
                      std::abs(finishButton.frame.size.height),
                       150, 75);
        eraseButton.tag = 2;
        [eraseButton setTitle:@"Erase" forState:UIControlStateNormal];
        [eraseButton addTarget:self action:@selector(erase)
              forControlEvents:UIControlEventTouchUpInside];


        listButton = [UIButton buttonWithType:UIButtonTypeCustom];
        listButton = [[UIButton alloc] init];
        listButton = [UIButton buttonWithType:UIButtonTypeRoundedRect ];
        listButton.frame = CGRectMake(finishButton.frame.origin.x ,
                       eraseButton.frame.origin.y +
                       std::abs(eraseButton.frame.size.height),
                       150, 75);
        listButton.tag = 2;
        [listButton setTitle:@"List" forState:UIControlStateNormal];
        [listButton addTarget:self action:@selector(orderedCrossingListing)
             forControlEvents:UIControlEventTouchUpInside];

```

```
50          errorLabel = [[UILabel alloc] initWithFrame:CGRectMake(
51                        frame.origin.x+0.35*frame.size.width,
52                        frame.origin.y +0.85*frame.size.height, 300, 100)];
53          errorLabel.numberOfLines = 0;
54          errorLabel.hidden = true;
55          errorLabel.tag = 3;
56
57
58          [self addSubview:finishButton];
59          [self addSubview:eraseButton];
60          [self addSubview:listButton];
61          [self addSubview:errorLabel];
62
63
64          // Initialization code
65          [self setMultipleTouchEnabled:NO];
66          path = [UIBezierPath bezierPath];
67          [path setLineWidth:2.0];
68          counter = -1;
69
70          intersectPoints.clear();
71
72      }
73      return self;
74  }
75
76
77  // FUNCTION:   drawRect
78  // SCOPE:      Objective C UIView.
79  // PURPOSE:    Base drawing point (brush head, if you will)
80  - (void)drawRect:(CGRect)rect
81  {
82      [incrementalImage drawInRect:rect];
83      [[UIColor blackColor] setStroke];
84      [path strokeWithBlendMode:kCGBlendModeNormal alpha:1.0 ];
85
86  }
87
88
89  // FUNCTION:   willAnimateRotationToInterfaceOrientation:
90  // PARAMETERS: (UIInterfaceOrientation)toInterfaceOrientation
91  //             duration:(NSTimeInterval)duration
92  // SCOPE:      Objective C UIView.
93  // PURPOSE:    Handles changing view (landscape, portrait).
94  - (void)willAnimateRotationToInterfaceOrientation:
95            (UIInterfaceOrientation)toInterfaceOrientation
96             duration:(NSTimeInterval)duration
97  {
98      if (toInterfaceOrientation == UIInterfaceOrientationLandscapeLeft ||
99          toInterfaceOrientation == UIInterfaceOrientationLandscapeRight)
100     {
101         finishButton.frame = CGRectMake((self.frame.origin.x +
102                                  0.95*self.frame.size.width),
103                                  self.frame.origin.y,
```

```
                                    150, 75);
        eraseButton.frame = CGRectMake(finishButton.frame.origin.x ,
                                    finishButton.frame.origin.y +
                                    std::abs(finishButton.frame.size.height),
                                    150, 75);
        listButton.frame = CGRectMake(finishButton.frame.origin.x,
                                    eraseButton.frame.origin.y +
                                    std::abs(eraseButton.frame.size.height),
                                    150, 75);
    }
    else
    {
        finishButton.frame = CGRectMake((self.frame.origin.x +
                                    0.85*self.frame.size.width),
                                    self.frame.origin.y  ,
                                    150, 75);
        eraseButton.frame = CGRectMake(finishButton.frame.origin.x ,
                                     finishButton.frame.origin.y +
                                     std::abs(finishButton.frame.size.height),
                                     150, 75);
        listButton.frame = CGRectMake(finishButton.frame.origin.x ,
                                     eraseButton.frame.origin.y +
                                     std::abs(eraseButton.frame.size.height),
                                     150, 75);
    }
}
```

Next are the methods that deal with touch events.

```
// FUNCTION:   touchesBegan
// PARAMETERS: (NSSet *)touches
//             withEvent:(UIEvent *)event
// SCOPE:      Objective C UIView.
// PURPOSE:    Called when user begins touching screen.
//             Stores path points in list KnotEdge.
//             Also stores points for a bezierPath (for smooth output).
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{

    errorLabel.hidden = true;
    KnotEdge.clear();
    counter++;

    bitIndex = 0;
    UITouch *touch = [touches anyObject];
    CGPoint p = [touch locationInView:self];
    bitMapPoints[0] = [touch locationInView:self];

    // Store the starting point of the edge.
    smallLP.push_back(p);

    pathPoints.push_back(p);
    KnotEdge.push_back(p);
    liftPoints.push_back(p);

```

```
27      [path moveToPoint:p];
28
29  }
30
31
32  // FUNCTION:   touchesMoved
33  // PARAMETERS: (NSSet *)touches
34  //             withEvent:(UIEvent *)event
35  // SCOPE:      Objective C UIView.
36  // PURPOSE:    Called as user moves after first touch.
37  //             Stores path points in list KnotEdge.
38  //             Stores points for a bezierPath (for smooth output).
39  - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
40  {
41
42      // Obtain information about the touch event.
43      UITouch *touch = [touches anyObject];
44      CGPoint p = [touch locationInView:self];
45
46      bitIndex++;
47
48      bitMapPoints[bitIndex] = p;
49      if (bitIndex == 4)
50      {
51          bitMapPoints[3] = CGPointMake(
52                            (bitMapPoints[2].x + bitMapPoints[4].x)/2.0,
53                            (bitMapPoints[2].y + bitMapPoints[4].y)/2.0);
54          [path moveToPoint:bitMapPoints[0]];
55          [path addCurveToPoint:bitMapPoints[3]
56                    controlPoint1:bitMapPoints[1]
57                    controlPoint2:bitMapPoints[2]]; // add a cubic Bezier
58
59          [self setNeedsDisplay]; // update display
60
61          bitMapPoints[0] = bitMapPoints[3];
62          bitMapPoints[1] = bitMapPoints[4];
63          bitIndex = 1;
64      }
65
66      pathPoints.push_back(p);
67      KnotEdge.push_back(p); // will pick up first & last inputted strands
68
69  }
70
71
72  // FUNCTION:   touchesEnded
73  // PARAMETERS: (NSSet *)touches
74  //             withEvent:(UIEvent *)event
75  // SCOPE:      Objective C UIView.
76  // PURPOSE:    Called after user picks up finger from screen.
77  //             Stores last point in KnotEdge
78  //             Creates a new knot_edge instance using points drawn.
79  //             Stores points for a bezierPath (for smooth output).
80  - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
```

```
{

    [self setNeedsDisplay];
    bitMapPoints[0] = [path currentPoint];
    bitIndex = 0;
    UITouch *touch = [touches anyObject]; // instantiate the touch
    CGPoint p = [touch locationInView:self]; // create CGPoint at touch
    pathPoints.push_back(p);
    liftPoints.push_back(p);
    KnotEdge.push_back(p);
    smallLP.push_back(p);
    edgeCount++;

    // Store KnotEdge as knot_edge's list of path points, and
    // store this last point and the first point (from touchesBegan)
    // as the extrema (IPa, IPb) for the knot_edge.
    knot_edge ke;
    ke.FullKnotEdge = KnotEdge;
    ke.IPa = smallLP[0];
    ke.IPb = smallLP[1];
    ke.parent_edge = counter;
    ke.edge_number = counter;
    ke.starter = true; // indicates this is a drawn edge
    discrete_edges.push_back(ke);

    // Clear KnotEdge to be ready for next edge.
    KnotEdge.clear();
    smallLP.clear();

    [self touchesMoved:touches withEvent:event];

}



// FUNCTION:   touchesCancelled
// PARAMETERS: (NSSet *)touches
//             withEvent:(UIEvent *)event
// SCOPE:      Objective C UIView.
// PURPOSE:    Handle event touches canceled.
//             Not really used that I know of,
//             just included for completion.
- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    [self touchesEnded:touches withEvent:event];
}
```

The following are functions called in the main algorithm.

```
// FUNCTION:   determine_minimum_edge
// SCOPE:      Objective C UIView.
// PARAMETERS: none.
//   GLOBAL PARAMETERS: all_crossings  *(list of crossings)*
//
// RETURNS:    no value.
```

```
// PURPOSE:    Updates undercrossing elements with new
//             edge numbers as the edge is broken up.
-(void) determine_minimum_edge{

    bool found_u0 = false;
    bool found_u1 = false;
    int original_edge, original_edge1, original_overcrossing;

    for (std::list<knot_crossing>::iterator
       it = all_crossings.begin();
       it != all_crossings.end(); ++it){
        found_u1 = false;
        std::list<knot_edge>::iterator it_edge = discrete_edges.begin();
        original_edge = (*it).crossing_elements[0];
        original_edge1 = (*it).crossing_elements[2];
        original_overcrossing = (*it).crossing_elements[1];

        while( it_edge != discrete_edges.end() &&
             (found_u0 == false || found_u1 == false) ){

            // Get last strand from parent edge of undercrossing0.
            if ((*it_edge).parent_edge == original_edge){
                (*it).crossing_elements[0] = (*it_edge).edge_number;
            }

            // Get first strand from parent edge of undercrossing1.
            if (found_u1 == false &&
               ((*it_edge).parent_edge == original_edge1)){
                (*it).crossing_elements[2] = (*it_edge).edge_number;
                found_u1 = true;
            }
            it_edge++;
        }
    }
}
```

```
// FUNCTION:   shift_crossing_elements
// SCOPE:      Objective C UIView.
// PARAMETERS: none.
//   GLOBAL PARAMETERS: discrete_edges *(list of edges)*
//                      all_crossings  *(list of crossings)*
//                      minUndercrossingLocation
//                         *(iterator pointing to min undercrossing)*
//                         FOUND IN: shift_crossing_elements
// RETURNS:    no value.
// PURPOSE:    Finds minimum crossing element,
//             shifts all edge labels so that numbering starts at 1,
//             and reshifts the labels so that they will be
//             ordered canonically *as they were drawn*.
//
//             Edges need to be ordered in this way to be consistent
//             with Mathematica PD Representation.
-(void) shift_crossing_elements{
    int minUndercrossing = (*all_crossings.begin()).crossing_elements[0];
```

```
19
20      minUndercrossingLocation = all_crossings.begin();
21
22      // Find smallested label. Store integer at minUndercrossing.
23      for (std::list<knot_crossing>::iterator
24          iter = all_crossings.begin();
25          iter != all_crossings.end(); ++iter){
26
27          if ( iter -> crossing_elements[0] <= minUndercrossing){
28              minUndercrossing = iter->crossing_elements[0];
29              minUndercrossingLocation = iter;
30
31
32          }
33          if ( iter -> crossing_elements[2] < minUndercrossing){
34              minUndercrossing = iter->crossing_elements[2];
35          }
36
37      }
38
39      // Now find smallest 'into' undercrossing.
40      minUndercrossingLocation = all_crossings.begin();
41      for (std::list<knot_crossing>::iterator
42          iter = all_crossings.begin();
43          iter != all_crossings.end(); ++iter){
44
45          if ((*iter).crossing_elements[0]
46               <= (*minUndercrossingLocation).crossing_elements[0]){
47              minUndercrossingLocation = iter;
48          }
49
50      }
51
52      // If the knot has not been shifted already, shift it.
53      if(unshifted == true){
54
55          std::vector<int> key;
56          std::list<int> edgeNumbers;
57
58          bool need_shift = false;
59          int i=0;
60
61          // Store the edge labels (edge_number) as they were drawn.
62          // To do so, traverse discrete_edges.
63          for (std::list<knot_edge>::iterator
64              it = discrete_edges.begin();
65              it != discrete_edges.end(); ++it){
66              i++;
67              key.push_back(i);
68              edgeNumbers.push_back((*it).edge_number);
69          }
70
71          edgeNumbers.sort();
72
```

```
73          // Check to see if canonical ordering already exists.
74          // That is, all elements (1,2, ..., n) will be in edgeNumbers
75          // in order, since it was sorted.
76          // (Here, 'n' is the total number of edges).
77          for (std::list<int>::iterator
78              it = edgeNumbers.begin();
79              it != edgeNumbers.end(); ++it){
80
81              // If even one element is missing, we need to map
82              // edge_numbers to new labels.
83              if((*it) != key[i]){
84                  need_shift = true;
85                  break;
86              }
87              i+=1;
88          }
89
90
91          if(need_shift == true){
92
93              // First, shift all elements so that no collisions may occur.
94              int totalNumberStrands = int(discrete_edges.size());
95              for (std::list<knot_edge>::iterator
96                  it = discrete_edges.begin();
97                  it != discrete_edges.end(); ++it){
98                  (*it).edge_number += totalNumberStrands;
99
100             }
101             for (std::list<knot_crossing>::iterator
102                 iter = all_crossings.begin();
103                 iter != all_crossings.end(); ++iter){
104                 (*iter).shift(-totalNumberStrands);
105             }
106
107             std::vector<int> key;
108             std::vector<int> edgeIds;
109             std::list<int>   edgeNumbers;
110
111             // Start at first 'into' undercrossing and traverse the
112             // elements in discrete_edges.
113
114             int starterElement =
115                  minUndercrossingLocation->crossing_elements[0];
116             std::list<knot_edge>::iterator
117                 iterStarterElement = discrete_edges.begin();
118             int starterCounter = 0;
119             while((*iterStarterElement).edge_number!= starterElement &&
120                   starterCounter <= totalNumberStrands){
121
122                 ++iterStarterElement;
123                 ++starterCounter;
124             }
125
126             // If iterStarterElement is found, continue.
```

```
            if(iterStarterElement != discrete_edges.end()){

                for(std::list<knot_edge>::iterator
                    it = iterStarterElement;
                    it != discrete_edges.end(); ++it){
                    edgeIds.push_back((*it).edge_number);
                }
                for(std::list<knot_edge>::iterator
                    it = discrete_edges.begin();
                    it != iterStarterElement; ++it){
                    edgeIds.push_back((*it).edge_number);
                }

                // Find all instances of the edge in the list of crossings
                // and update with new value.
                std::vector<int>::iterator iterEdge2;
                for(int i = 0; i < edgeIds.size(); ++i){
                    for (std::list<knot_crossing>::iterator
                         iterCross1 = all_crossings.begin();
                         iterCross1 != all_crossings.end(); ++iterCross1){

                        iterEdge2 = std::find(
                            ((*iterCross1).crossing_elements).begin(),
                            ((*iterCross1).crossing_elements).end(),
                             edgeIds[i]);

                        if(iterEdge2 !=
                        ((*iterCross1).crossing_elements).end()){
                            (*iterEdge2) = i+1;
                        }
                    }
                }
            }
            else{
                NSString *warning =
                          [NSString stringWithFormat:@"Warning! \n"
                                    "Program could not label the knot \n"
                                     "correctly."];
                [self callWarning:warning];
            }
        }
        unshifted = false;
    }
}
```

```
// FUNCTION:   draw_circle_about_crossings
// SCOPE:      Objective C UIView.
// PARAMETERS: (int)width
//             height:(int)height
//   GLOBAL PARAMETERS:
//                intersectPoints *(list of intersections)*
// RETURNS:    no value.
// PURPOSE:    Places a UIButton at each crossing.
-(void) draw_circle_about_crossings:(int)width height:(int)height{
```

```
10
11      int view_tag = 4; // 0 and 1 are reserved for the buttons
12      int countTag = 1;
13      for(std::list<CGPoint>::iterator iter =
14          intersectPoints.begin();
15          iter != intersectPoints.end(); ++iter){
16          if(!FoundPointEqual(*iter, CGPointZero)){
17
18              CGFloat centerX = (*iter).x - width/2;
19              CGFloat centerY = (*iter).y - height/2;
20
21              CGRect intersection_ball = CGRectMake(centerX, centerY,
22                                                    width, height);
23
24              NSString *string = [NSString stringWithFormat:@"%d", countTag];
25              CGContextRef ctx = UIGraphicsGetCurrentContext();
26
27              CGContextFillRect(ctx, intersection_ball);
28
29              UIButton *button = [UIButton
30                  buttonWithType:UIButtonTypeRoundedRect];
31              [button addTarget: self
32                         action: @selector(buttonTouched_crossing:)
33               forControlEvents: UIControlEventTouchDown];
34              button.frame = CGRectMake(centerX, centerY, width, height);
35              button.tag = view_tag;
36              countTag+=1;
37              view_tag+=1;
38              [button setTitle:string forState:UIControlStateNormal];
39              [self addSubview:button];
40
41          }
42      }
43  }
```

The next functions are the full method for partitioning the knot.

```
1  // FUNCTION:   partitionKnot
2  // SCOPE:      C++. Separate component of the Objective C UIView.
3  // PARAMETERS: std::list<knot_edge>& edges, *(list of edges)*
4  //             CGPoint LP1, CGPoint LP2, *(extrema of two strands)*
5  //             int a, int b, int c, int d, *(details of the line)*
6  //             float m, float yint,        *(details of the line)*
7  //             std::list<knot_edge>::iterator& startIter,
8  //                 *(points to first strand ['into'] strand)*
9  //             std::list<knot_edge>::iterator& endIter
10 //                 *(points to second strand ['outcoming'] strand)*
11 // RETURNS:    CGPoint  *(intersection point)*
12 // PURPOSE:    Find the edge (E) which intersects the line between the two
13 //             extrema LP1 and LP2.
14 // MODIFIES:   edges (-->discrete_edges, in this case).
15 //             startIter & endIter if necessary
16 // MODS:       Splits E into two elements of edges(i.e.two separate edges).
17 //             Adds the new elements, in order as drawn, into edges.
18 //             Deletes E from edges.
```

```
19
20 CGPoint partitionKnot(std::list<knot_edge>& edges,
21               CGPoint LP1, CGPoint LP2,
22               double a, double b, double c, double d,
23               float m, float yint,
24               std::list<knot_edge>::iterator& startIter,
25               std::list<knot_edge>::iterator& endIter ){
26
27 // Finds points that satisfy the line equation (i.e. is a crossing)
28 std::list<knot_edge>::iterator EdgeCheckIter = edges.begin();
29 std::list<CGPoint>::iterator checkIter =
30     ((*EdgeCheckIter).FullKnotEdge).begin();
31 CGPoint intersection_pt; //store intersection point
32
33 // New instances of knot_edge, to be used for the new strands.
34 knot_edge ke1;
35 knot_edge ke2;
36
37 int delta = 5;
38
39 checkIter = ((*EdgeCheckIter).FullKnotEdge).begin();/ /first edge
40 bool intersectFound = false;
41
42 while(EdgeCheckIter != edges.end() && intersectFound == false){
43
44 while((checkIter != ((*EdgeCheckIter).FullKnotEdge).end() &&
45        EdgeCheckIter != edges.end() )&&
46       intersectFound == false){
47
48
49     if ( ( ((*checkIter).x != LP1.x) || ((*checkIter).y != LP1.y) ) &&
50          ( ((*checkIter).x != LP2.x) || ((*checkIter).y != LP2.y))){
51
52         // Check if intersection is found.
53         // First, check the box about the intersection.
54         if ( ( ((*checkIter).x >= a) && ((*checkIter).x <= b) ) &&
55              ( ((*checkIter).y >= c) && ((*checkIter).y <= d) ) ){
56
57             // Solve equation of line with checkIter.
58             // If checkIter's y matches the line equation,
59             // then found intersection.
60             double y = (m * ((*checkIter).x)) + yint;
61
62             if ( std::abs((*checkIter).y - y) < delta ){
63
64                 KnotEdge.clear();
65
66                 // Store pathPoints leading to the intersection element.
67                 for(std::list<CGPoint>::iterator subInterval
68                     = ((*EdgeCheckIter).FullKnotEdge).begin();
69                     subInterval != checkIter; ++subInterval){
70
71                     KnotEdge.push_back(*subInterval);
72                 }
```

```
73                  ke1.FullKnotEdge = KnotEdge;
74
75                  KnotEdge.clear();
76                  // Store pathPoints from intersection element to path end.
77                  for(std::list<CGPoint>::iterator
78                      subInterval = checkIter;
79                      subInterval != ((*EdgeCheckIter).FullKnotEdge).end();
80                      ++subInterval){
81                      KnotEdge.push_back(*subInterval);
82                  }
83
84                  ke2.FullKnotEdge = KnotEdge;
85
86                  intersection_pt = CGPointMake(checkIter->x, checkIter->y);
87
88                  for (std::list<knot_edge>::iterator
89                      i = edges.begin();
90                      i !=  edges.end(); ++i){
91                      if ((*i).edge_number ==
92                          (*EdgeCheckIter).edge_number){
93
94                          ke1.starter = false;
95                          ke2.starter = false;
96
97                          ++edgeCount;
98                          ke1.parent_edge = (*EdgeCheckIter).parent_edge;
99                          ke1.edge_number = edgeCount;
100                         ++edgeCount;
101                         ke2.parent_edge = (*EdgeCheckIter).parent_edge;
102                         ke2.edge_number = edgeCount;
103
104
105                         // knot_edges are stored in order drawn.
106                         ke1.IPa = (*i).IPa;
107                         ke1.IPb = (*checkIter);
108                         ke2.IPa = (*checkIter);
109                         ke2.IPb = (*i).IPb;
110
111
112                         int k = int((ke1.FullKnotEdge).size());
113                         k = k-floor(k/4);
114                         std::list<CGPoint>::iterator
115                             it1 = (ke1).FullKnotEdge.begin();
116                         advance(it1, k);
117                         k = int((ke2.FullKnotEdge).size());
118                         k = floor(k/4);
119                         std::list<CGPoint>::iterator
120                             it2 = (ke2).FullKnotEdge.begin();
121                         advance(it2, k);
122
123                         CGPoint vecPivot;
124                         CGPoint vec1;
125                         CGPoint vec2;
126
```

```
127                         // Use position to determine which element is
128                         // counter-clockwise to the 'into' undercrossing.
129                         vecPivot.x = (*endIter).IPa.x - (*startIter).IPb.x;
130                         vecPivot.y = (*endIter).IPa.y  -(*startIter).IPb.y;
131                         vec1.x = (*it1).x - (*checkIter).x;
132                         vec1.y = (*it1).y - (*checkIter).y;
133                         vec2.x = (*it2).x - (*checkIter).x;
134                         vec2.y = (*it2).y - (*checkIter).y;
135
136
137                         if(vecPivot.y < 0 && vecPivot.x >=0) {
138                             if( vec1.y>=0 || vec1.x>=0){
139                                 crossing.crossing_elements[1]
140                                     = ke1.edge_number;
141                                 crossing.crossing_elements[3]
142                                     = ke2.edge_number;
143                             }
144                             else {
145                                 crossing.crossing_elements[1]
146                                     = ke2.edge_number;
147                                 crossing.crossing_elements[3]
148                                     = ke1.edge_number;
149                             }
150                         }
151                         else if(vecPivot.y < 0 && vecPivot.x <= 0) {
152                             if(vec1.y <=0 || vec1.x>=0){
153                                 crossing.crossing_elements[1]
154                                     = ke1.edge_number;
155                                 crossing.crossing_elements[3]
156                                     = ke2.edge_number;
157                             }
158                             else {
159                                 crossing.crossing_elements[1]
160                                     = ke2.edge_number;
161                                 crossing.crossing_elements[3]
162                                     = ke1.edge_number;
163                             }
164
165                         }
166
167                         else if(vecPivot.y > 0 && vecPivot.x <=0) {
168                             if(vec1.y<=0 || vec1.x <=0){
169                                 crossing.crossing_elements[1]
170                                     = ke1.edge_number;
171                                 crossing.crossing_elements[3]
172                                     = ke2.edge_number;
173                             }
174                             else {
175                                 crossing.crossing_elements[1]
176                                     = ke2.edge_number;
177                                 crossing.crossing_elements[3]
178                                     = ke1.edge_number;
179                             }
180
```

```
181                         }
182                         else if(vecPivot.y > 0 && vecPivot.x >=0) {
183                             if(vec1.y>=0 || vec1.x <=0 ){
184                                 crossing.crossing_elements[1]
185                                     = ke1.edge_number;
186                                 crossing.crossing_elements[3]
187                                     = ke2.edge_number;
188                             }
189                             else {
190                                 crossing.crossing_elements[1]
191                                     = ke2.edge_number;
192                                 crossing.crossing_elements[3]
193                                     = ke1.edge_number;
194                             }
195
196                         }
197                         if(vecPivot.y > 0 && vecPivot.x == 0) {
198                             if(vec1.x <=0 ){
199                                 crossing.crossing_elements[1]
200                                     = ke1.edge_number;
201                                 crossing.crossing_elements[3]
202                                     = ke2.edge_number;
203                             }
204                             else {
205                                 crossing.crossing_elements[1]
206                                     = ke2.edge_number;
207                                 crossing.crossing_elements[3]
208                                     = ke1.edge_number;
209                             }
210
211                         }
212                         if(vecPivot.y < 0 && vecPivot.x == 0) {
213                             if(vec1.x >=0 ){
214                                 crossing.crossing_elements[1]
215                                     = ke1.edge_number;
216                                 crossing.crossing_elements[3]
217                                     = ke2.edge_number;
218                             }
219                             else {
220                                 crossing.crossing_elements[1]
221                                     = ke2.edge_number;
222                                 crossing.crossing_elements[3]
223                                     = ke1.edge_number;
224                             }
225
226                         }
227                         if(vecPivot.y == 0 && vecPivot.x > 0) {
228                             if(vec1.y >=0 ){
229                                 crossing.crossing_elements[1]
230                                     = ke1.edge_number;
231                                 crossing.crossing_elements[3]
232                                     = ke2.edge_number;
233                             }
234                             else {
```

```
235                                 crossing.crossing_elements[1]
236                                     = ke2.edge_number;
237                                 crossing.crossing_elements[3]
238                                     = ke1.edge_number;
239                             }
240
241                         }
242                         if(vecPivot.y == 0 && vecPivot.x < 0) {
243                             if(vec1.y <=0 ){
244                                 crossing.crossing_elements[1]
245                                     = ke1.edge_number;
246                                 crossing.crossing_elements[3]
247                                     = ke2.edge_number;
248                             }
249                             else {
250                                 crossing.crossing_elements[1]
251                                     = ke2.edge_number;
252                                 crossing.crossing_elements[3]
253                                     = ke1.edge_number;
254                             }
255
256                         }
257
258                         if(startIter == i){
259                             advance(startIter,-2);
260                             advance(endIter, -1);
261                         }
262                         if(endIter == i){
263                             advance(endIter,-1);
264                         }
265
266                         edges.insert(i,ke1);
267                         edges.insert(i,ke2);
268                         --i;
269
270                         all_crossings.push_back(crossing);
271                         crossing.clear();
272
273                         EdgeCheckIter = i; // Now points to ke2
274
275                         ++i;
276
277                         // If already broke up the strand,
278                         // will need to update crossings.
279                         // Do this by understanding the previous
280                         // crossing element to the
281                         // knot being broken up.
282                         // We're using the fact that upon going 'into,'
283                         // the next strand
284                         // label is one higher than the previous.
285                         // So we know the orientation of the strand.
286                         if((*i).starter == false){
287
288                             for (std::list<knot_crossing>::iterator
```

```
                                    it = all_crossings.begin();
                                    it != all_crossings.end(); ++it){

                                     int m = 0;
                                     for(int j = 0; j < 4; ++j){
                                         k = 0;
                                         switch(j){
                                             case 0:
                                                 m = 2;
                                             case 1:
                                                 m = 3;
                                             case 2:
                                                 m = 0;
                                             case 3:
                                                 m = 1;
                                         }


                                         if ((*it).crossing_elements[j]
                                             == (*i).edge_number){
                                             if((*it).crossing_elements[j] >
                                                 (*it).crossing_elements[m]){
                                             (*it).crossing_elements[j]
                                                 = ke1.edge_number;
                                             }
                                             else{
                                             (*it).crossing_elements[j]
                                                 = ke2.edge_number;
                                             }
                                         }
                                     }
                                 }
                             }
                             edges.erase(i);
                             intersectFound = true;

                             break;
                         }
                     }
                 }
             }
           }
         ++checkIter;
         }

         advance(EdgeCheckIter,1);
         checkIter = ((*EdgeCheckIter).FullKnotEdge).begin();

         }

         if(intersectFound == true){
             return intersection_pt;
         }
         else{
```

```
        gapWithoutCrossing = true;
        return CGPointZero;
    }
}
```

The below function `GetWorkKnot()` is the main function. Listed first is its call function.

```
// FUNCTION:   call_GetWordKnot
// SCOPE:      Objective C UIView.
// PARAMETERS: none
// RETURNS:    no value.
// PURPOSE:    Checks to see if user inputted enough
//             information to proceed with program.
//             If not, outputs message to user and
//             clears variables.
-(void) call_GetWorkKnot{
    if(discrete_edges.size() > 1){
        [self GetWorkKnot];
    }
    else if(discrete_edges.size() == 1){
        errorLabel.text = @"Please input at least two strands.";
        [self erase];
        errorLabel.hidden = false;

    }
    else{
        errorLabel.text = @"Error! Please draw a knot projection.";
        [self erase];
        errorLabel.hidden = false;

    }

}
```

```
// FUNCTION:   GetWorkKnot
// SCOPE:      Objective C UIView.
// PARAMETERS: none.
//   GLOBAL PARAMETERS: discrete_edges *(list of knot_edges)*
//                      all_crossings  *(list of knot_crossings)*
//                      KnotEdge       *(list of CGPoints*)
// RETURNS:    no value.
// PURPOSE:    Takes originally drawn constructs.
//             'Glue's the first and last strands together,
//               >i.e. puts them in same knot_edge element.
//             Removes the liftPoints stored for that element.
//               >The end of the last strand and beginning of first strand.
//             Determines the parameters of the line between knot_edge
//             extreme values (IPb of one and IPa of the next).
//
//             Begins the partitioning of the knot and the
//             creation of the knot strand.
// FUNCTION CALLS:
//    * partitionKnot
//    * draw_circle_about_crossings
//    * shift_crossing_elements
```

```
//    * callWarning      [[<<if error occurs]]
-(void) GetWorkKnot{

    // 'Glue' first and last strands together.
    KnotEdge.clear();
    int k = edgeCount;
    {
        std::list<knot_edge>::iterator iterLastEle= discrete_edges.begin();

        // Copy last strand.
        advance(iterLastEle, k);
        for (std::list<CGPoint>::iterator
             iter = ((*iterLastEle).FullKnotEdge).begin();
             iter != ((*iterLastEle).FullKnotEdge).end(); ++iter){
            KnotEdge.push_back(*iter);
        }


        CGPoint end_of_first_strand;

        // Copy first strand.
        for (std::list<CGPoint>::iterator
            iter = ((*discrete_edges.begin()).FullKnotEdge).begin();
            iter !=((*discrete_edges.begin()).FullKnotEdge).end(); ++iter){
            KnotEdge.push_back(*iter); //storing points in path
            end_of_first_strand = (*iter);
        }
     }


    discrete_edges.pop_back();
    discrete_edges.pop_front();

    liftPoints.pop_front();
    liftPoints.pop_back();

    knot_edge ke;
    ke.FullKnotEdge = KnotEdge;
    ke.IPb = (end_of_first_strand);
    end_of_first_strand = (*KnotEdge.begin());

    ke.IPa = end_of_first_strand;
    ke.parent_edge = edgeCount;
    ke.edge_number = edgeCount;
    ke.starter = true;
    discrete_edges.push_back(ke);

    KnotEdge.clear();


    // Define lines between each undercrossing space.
    double a,b,c,d;
    double m, yint;
    int sentinel = 2*int(liftPoints.size());
```

```
76      std::list<knot_edge>::iterator startLine = discrete_edges.begin();
77      std::list<knot_edge>::iterator endLine;
78      edgeCount+=1;
79      int partitionCount = 0;
80      while ((startLine != discrete_edges.end() ||
81              endLine != discrete_edges.begin()) &&
82             partitionCount <= sentinel){
83          partitionCount++;
84          endLine = startLine;
85
86          // Handle boundary case.
87          advance(endLine,1);
88          if(endLine == discrete_edges.end()){
89              endLine = discrete_edges.begin();
90          }
91          // Anchor at liftpoints.
92          CGPoint LP1 = (*startLine).IPb;
93          CGPoint LP2 = (*endLine).IPa;
94
95          crossing.crossing_elements[0] = (*startLine).parent_edge;
96          crossing.crossing_elements[2] = (*endLine).parent_edge;
97
98          double delta = 1.5;
99          double buffer = 0.5;
100
101         CGPoint intersection_point; // to draw block abt intersections
102
103         if ((LP1.x != LP2.x) && (LP1.y != LP2.y)){
104
105             // Case 1: x- or y-values are equal.
106             if(std::abs(LP1.x - LP2.x) < delta && LP1.y > LP2.y){
107                 a = LP1.x-buffer;
108                 b = LP2.x+buffer;
109                 c = LP2.y;
110                 d = LP1.y;
111                 m = 0;
112             }
113             else if(std::abs(LP1.x - LP2.x) < delta && LP1.y <= LP2.y){
114                 a = LP1.x-buffer;
115                 b = LP2.x+buffer;
116                 c = LP1.y;
117                 d = LP2.y;
118                 m = 0;
119             }
120             else if(std::abs(LP1.y - LP2.y) < delta && LP1.x > LP2.x){
121                 a = LP2.x;
122                 b = LP1.x;
123                 c = LP2.y-buffer;
124                 d = LP1.y+buffer;
125                 m = 0;
126             }
127             else if(std::abs(LP1.y - LP2.y) < delta && LP1.x <= LP2.x){
128                 a = LP1.x;
129                 b = LP2.x;
```

```
130                 c = LP1.y-buffer;
131                 d = LP2.y+buffer;
132                 m = 0;
133             }
134             // Otherwise:
135             else{
136                 if (LP1.x > LP2.x){
137                     a = LP2.x;
138                     b = LP1.x;
139                 }
140                 if(LP1.x < LP2.x){
141                     a = LP1.x;
142                     b = LP2.x;
143                 }
144                 if (LP1.y > LP2.y ){
145                     c = LP2.y;
146                     d = LP1.y;
147                 }
148                 if (LP1.y < LP2.y ){
149                     c = LP1.y;
150                     d = LP2.y;
151                 }
152                 m = (LP2.y - LP1.y )/ (LP2.x - LP1.x);
153             }
154 
155 
156             // Obtain final part of equation of the line between extrema.
157             yint = LP2.y - m * LP2.x;
158 
159             intersection_point = partitionKnot(discrete_edges,
160                                                LP1,LP2,a,b,c,d,
161                                                m , yint,
162                                                startLine, endLine);
163             intersectPoints.push_back(intersection_point);
164 
165             advance(startLine,1);
166         }
167         else{
168             startLine++;
169         }
170 
171         int rect_width = 100;
172         int rect_height = 100;
173         [self draw_circle_about_crossings:rect_width height:rect_height];
174 
175     }
176 
177     if(partitionCount == sentinel){
178         NSString *warning = [NSString stringWithFormat:@"Error! \n"
179                              "Program could not read the knot. \n"
180                              "Please re-input knot."];
181         [self callWarning:warning];
182     }
183 
```

```
    if(gapWithoutCrossing == true){
        NSString *warning = [NSString stringWithFormat:@"Warning! \n"
                             "One or more gaps is not a crossing.  \n"
                             "Could result in error."];
        [self callWarning:warning];
    }

    // Controls for finding the undercrossings' new identifier values
    // (after breakage).
    // This procedure uses the fact that the user draws
    // undercrossing to undercrossing.
    // We need to do this before we consider switching
    // over/undercrossings, etc.
    [self determine_minimum_edge];

    // Call the function to centralize the edge numbers,
    // so that the minimum edge number is 1.
    [self shift_crossing_elements];

}
```

The following two functions obtain the Mathematica PD Representation string for the knot.

```
// FUNCTION:   write_knot_description
// SCOPE:      C++. Separate component of the Objective C UIView.
// PARAMETERS: std::list<knot_crossing>::iterator i,
//             std::string& knot_description_full
// RETURNS:    no value.
// PURPOSE:    Modify the string knot_description_full, adding the info
//             for crossing at address pointed to by i.
void write_knot_description(std::list<knot_crossing>::iterator i,
                            std::string& knot_description_full,
                            int& totalCount){

    // If this is not first crossing added.
    if(totalCount > 0){
        knot_description_full = knot_description_full + ",";

    }
    int u0 = (*i).crossing_elements[0];
    int u1 = (*i).crossing_elements[2];
    int o0 = (*i).crossing_elements[1];
    int o1 = (*i).crossing_elements[3];


    knot_description_full = knot_description_full + "X[" +
                    std::to_string(u0) + "," + std::to_string(o0) + "," +
                    std::to_string(u1) + "," + std::to_string(o1) + "]";

    totalCount+=1;

}
// FUNCTION:   find_next_undercrossing
// SCOPE:      C++. Separate component of the Objective C UIView.
// PARAMETERS: int minUndercrossing,   *(edgeNumber of knot)*
```

```
//              std::list<knot_crossing>::iterator& min_cross,
//                        *(iterator pointing to min)*
//              std::string &knot_desc  *(string description of the knot)*
//              std::list<knot_edge>& edges, *(list of all edges)*
//              std::list<knot_crossing>&crossings,*(list of all crossings)*
//              int totalCount *(counter for number of iterations)*
// RETURNS:     no value.
// PURPOSE:     Modify the string knot_description_full, adding the info
//              for crossing at address pointed to by i.
void find_next_undercrossing(int minUndercrossing,
                             std::list<knot_crossing>::iterator& min_cross,
                             std::string &knot_desc,
                             std::list<knot_edge>& edges,
                             std::list<knot_crossing>& crossings,
                             int totalCount){

    // If minUndercrossing exceeds the number of strands,
    // need to start back at 1.
    // (Edges are labeled in canonical order.)
    if (minUndercrossing > edges.size() ){
        minUndercrossing = 1;
    }
    bool found = false;
    // Search over the list of crossings for one with the next label.
    for (std::list<knot_crossing>::iterator iter = crossings.begin();
         iter != crossings.end(); ++iter){

        if ( (*iter).crossing_elements[0] == minUndercrossing){
            minUndercrossing = iter->crossing_elements[2];
            min_cross = iter;
            found = true;
            // As soon as we find the strand, we're done searching.
            break;
        }
    }
    // If an undercrossing with label minUndercrossing is not found,
    // iterate minUndercrossing once and search.
    // (Using the fact that edges are in canonical order.)
    if (found == false){
        find_next_undercrossing(minUndercrossing+1, min_cross, knot_desc,
                                edges, crossings, totalCount);
    }
    else{
        // Append the crosssing to the screen.
        write_knot_description(min_cross ,knot_desc, totalCount);
        // Set up and search for next crossing.
        minUndercrossing = (*min_cross).crossing_elements[2];
        if (totalCount < crossings.size()){
            find_next_undercrossing(minUndercrossing, min_cross,
                                    knot_desc, edges, crossings,
                                    totalCount);
        }
    }
}
```

The following function makes the intial call to the above two processes.

```
// FUNCTION:   orderedCrossingListing
// SCOPE:      Objective C UIView.
// PARAMETERS: none.
//   GLOBAL PARAMETERS:
//            discrete_edges *(list of edges)*
//            all_crossings  *(list of crossings)*
//            minUndercrossingLocation
//                   *(location of smallest undercrossing label)*
//                   FOUND IN: shift_crossing_elements
// RETURNS:    no value.
// PURPOSE:    Finds first crossing to kick off knot_description string.
-(void) orderedCrossingListing{

    if(discrete_edges.size()>0){
        counter = 0;
        std::string knot_description_full = "";
        int minUndercrossing =
            minUndercrossingLocation->crossing_elements[0];
        minUndercrossingLocation =all_crossings.begin();

        for (std::list<knot_crossing>::iterator
            iter = all_crossings.begin();
             iter != all_crossings.end(); ++iter){
            if ( iter -> crossing_elements[0] <= minUndercrossing){
                minUndercrossing = iter->crossing_elements[0];
                minUndercrossingLocation = iter;
            }
        }

        // First call to find_next_undercrossing:
        find_next_undercrossing(minUndercrossing, minUndercrossingLocation,
                                knot_description_full,
                                discrete_edges, all_crossings, 0);

        // find_next_undercrossing will iterate and modify
        // knot_description_full until the label is complete,
        // i.e., all X[,,,] are stored in knot_description_full.
        knot_description =
                 [NSString stringWithCString:knot_description_full.c_str()
                     encoding:[NSString defaultCStringEncoding]];
        knot_description =
                 [NSString stringWithFormat:@"%@%@%@", @"PD[" ,
                                            knot_description, @"]"];
        [self showEmail];
    }
    else{
        [self callWarning:@"Error! Knot not drawn."];
    }
}
```

Bibliography

[1] Adams, Colin. 2013. *The Knot Book* Providence: American Mathematical Society, 2004. Print.

[2] Bar-Natan, Dror. *The Mathematica Package KnotTheory*. 5 Jan 2009. Web. 2013-2015. `http://katlas.org/wiki/The_Mathematica_Package_KnotTheory%60`.

[3] Bar-Natan, Dror and Scott Morrison. *The Knot Atlas*. University of Toronto: 29 Apr 2013. Web. 2013-2015. `http://katlas.org`.

[4] Culler, Marc, Nathan Dunfield, and Jeffrey Weeks. *SnapPy, a Computer Program for Studying the Topology of* 3*-manifolds*. 2009-2015. Web. 2013-2015. `http://snappy.computop.org`

[5] iOS Developer Library. 2013. *iOS 7 Design Resources*. Apple.com. Accessed 2013-2015. https://developer.apple.com/library/ios/navigation/

[6] Manturov, Vassily. *Knot Theory*. Boca Raton: Chapman and Hall, 2004. Google Books. Web. Aug. 2014.

[7] Murasugi, Kunio. *Knot Theory and Its Applications*. Trans. Bohdan Kurpita. Birkhauser: Boston, 1996. Pg. 267-296. Web. 2 Sept. 2013. `http://www.maths.ed.ac.uk/~aar/papers/murasug3.pdf`.

[8] Reidemeister, Kurt. *Knotentheorie*. Berlin: Springer, 1932. Print.

[9] Royden, H.L. and P.M. Fitzpatrick. *Real Analysis*, 4^{th} ed. Upper Saddle River: Pearson Education, Inc. 2010. Print.

[10] Stein, W. A. et al., *Sage Mathematics Software*, The Sage Development Team, `http://www.sagemath.org`

[11] Stroustrup, Bjarne. *Programming– Principles and Practice Using C++*. Addison-Wesley 2009. Print.