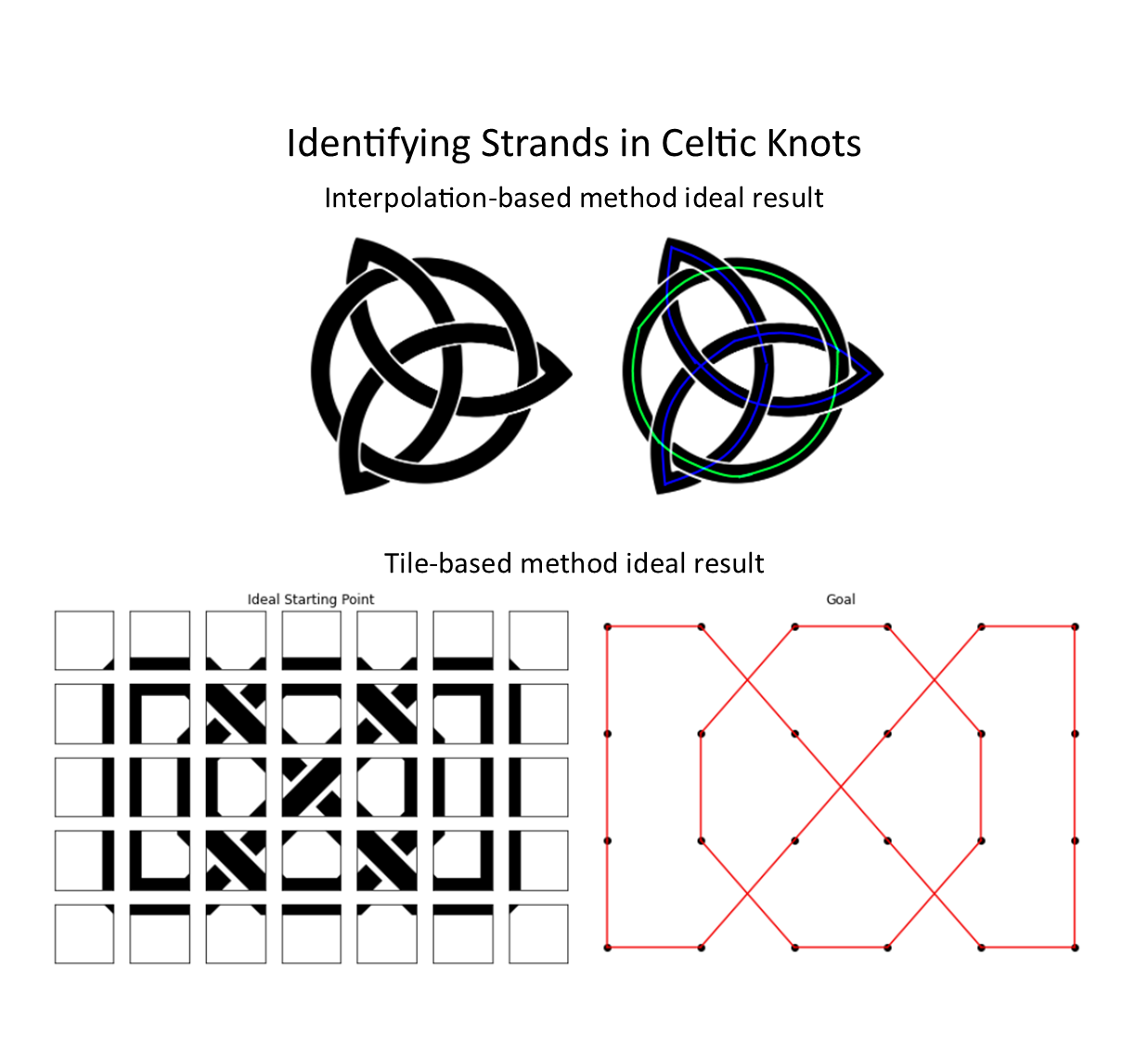
# Github repo: <https://github.gatech.edu/cv-project>

# Abstract

Celtic knots have regular structures we can analyze such as intersections, number of strands, and motifs. We use two different methods: a tile-based template matching approach assuming regular grid-like structures of celtic knots and an interpolation gradient approach. We have demonstrated that both methods can identify the strands of some celtic knots, with the interpolation method being more generalizable than the tile-based template matching approach. Studying these regular images may lend insight into how to analyze occlusion and its applications to imaging natural and man-made structures.

# Teaser Figure



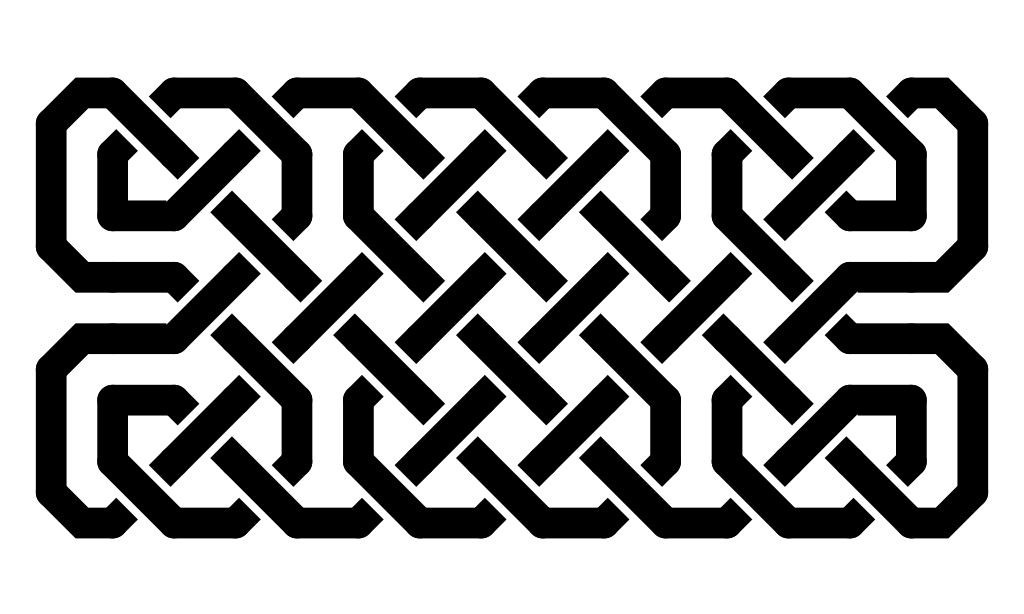
# 

# Introduction

Celtic knots are decorative knots consisting of “strands” that intertwine with each other in particular ways. They are often found in religious texts, manuscripts, and monuments.

Typical celtic knots consist of one or more strands of some fixed width that, in a series of straight paths and curves, fold over each other (or itself) in an interlocking “knot” or “weaving” pattern. In the final image, there are sections of the strands that are covered, or “occluded,” by another strand that goes over it.

From our initial research we were not able to find much research at all on the analysis of celtic knots. Most of what was found was frameworks for their generation. However, we believe this research is important as the task of reasoning with the self occlusion of celtic knots can be applied to many real analysis tasks such as overlapping roads at turnpikes and the paths of tree branches.



The structure of the knot and the continuity of strands is easily recognizable by humans. But for automated systems, occlusion can be difficult to interpret. However, given their traditionally regular shape and relatively strict rules, Celtic knots present an interesting opportunity to explore how a computer might be able to reason with uncertainty in order to find interesting properties of the knots. These include the path of the individual strands, locations of intersections and turns, and building off these properties which string is on top or bottom of an intersection and how they are tied together.

This led us to consider various approaches for how a computer may be able to detect these strands and identify parts of the image that constitute a continuous strand, and the underlying structure of the knot, such as the number of strands, and create a mathematical representation of the path of each strand.

# 

# Approach

We are testing 2 different methods to detect the underlying structure of standard celtic knots. The first is a tile-based approach, inspired by celtic knot generators that use pre-made tiles to assemble knots ([Hypatia Studio](http://hypatiastudio.com/celticknots/)). The second approach uses interpolation to connect ends of curves.

## Tile-based approach

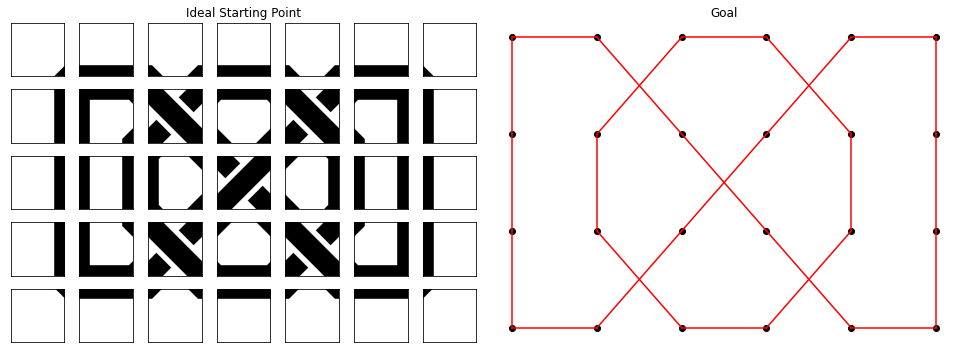
Several manual tile generators utilize existing tiles for users to assemble knots like a puzzle. We use this approach to reverse-identify the original tiles given an already completed knot with the following assumptions:

1. The strands are a single color with sharp contrast with the background
2. Strands are constant width
3. Intersections are denoted with a “break” in the strand that goes underneath. We call this gap the “overlap width.”
4. The knot work is uniformly “tight,” or that there are no inconsistently large gaps between neighboring strands
5. The knot contains at least one intersection
6. The knot must contain either a parallel vertical or horizontal lines right beside each other to determine strand width and gap width which infer the tile size
7. There are no “dead ends,” i.e., everything forms some kind of continuous loop
8. No “branching” of a single strand. i.e., no strand splits off into two separate strands
9. Can be represented by a set of standard set of tiles

The overall approach is as follows:

1. Generate a library of tile types for the particular style. We are using a fork of [rspencer01/celtic](https://github.com/rspencer01/celtic) that we modified slightly as our main image generator. We use this to manually construct a tile library
2. Identify and calculate the style parameters: the strand width, overlap width, and tile size
3. Generate the tile template images based on the style parameters
4. Apply template matching to identify the tile types
5. Use the known structure of the tiles to construct or identify features of the celtic knot

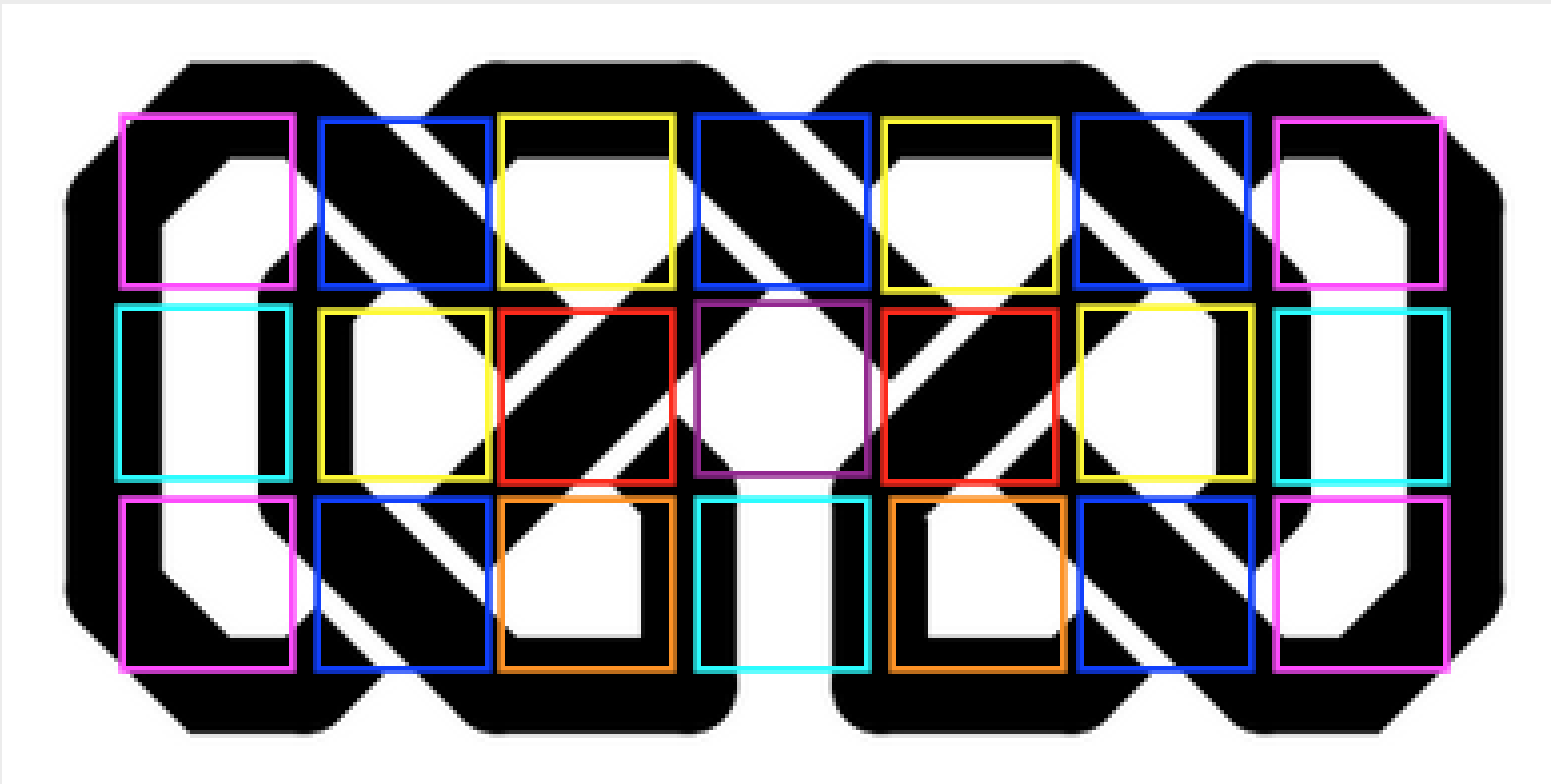
The goal is to correctly find the tiles that make up the knot, and then create a plot of the spline of the knot, as shown below.



### Generating the tile library

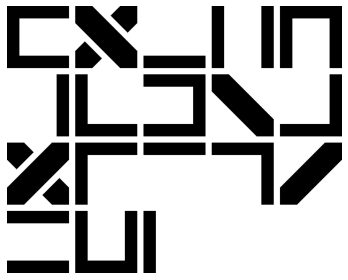
We manually curated the set of tile types we could observe in example knots generated from our fork of [rspencer01/celtic](https://github.com/rspencer01/celtic), and that we deemed was informative or could not be inferred by neighboring tiles.

First, we decided to create tiles where the “splines” of strands would lie along the edges of the tile. This was from some manual inspection of the tiles generated from [rspencer01/celtic](https://github.com/rspencer01/celtic). This provides us an easy way to map corners of tiles to points on a spline. It also affords tile types that contain intersections. Below is a manually created tile grid, each color square denoting a certain tile type.



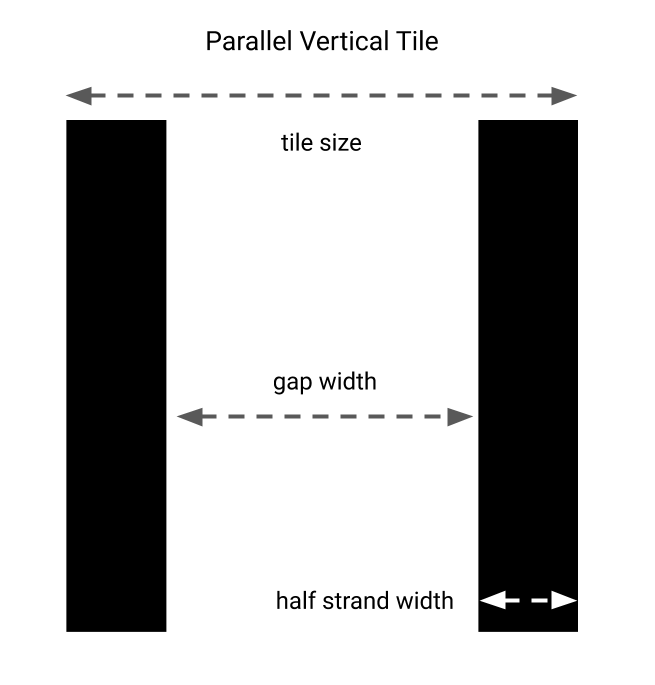
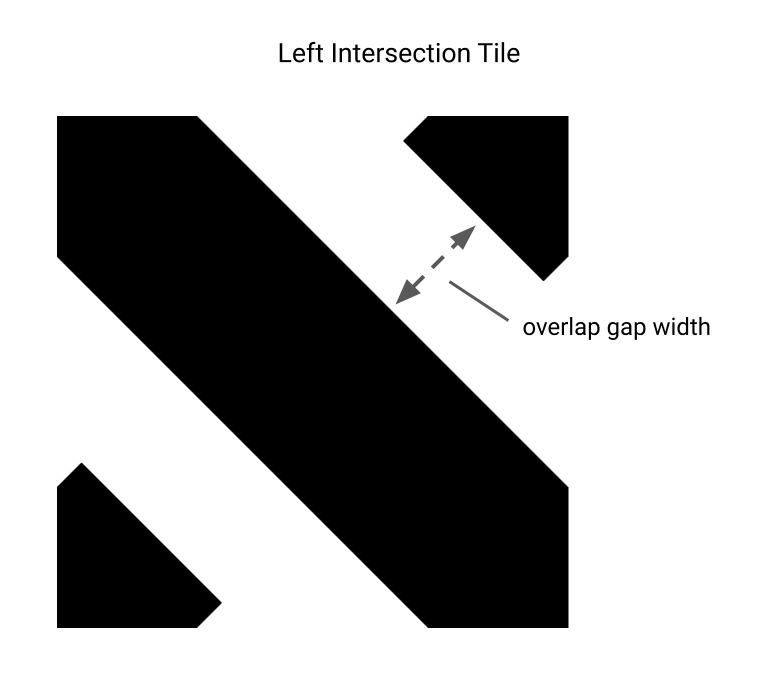
Each tile type has fields which allow us to convert detected image features to a graph we can use to analyze and visualize the underlying structure of the knot (number of independent loops, number of intersections, etc.). These fields include nodes for each of the corners of the tile type, and edges between those nodes within a tile type.

From these tile properties, we can render template images and apply template matching with it on the original knot image given the estimated knot parameters described in the next section (tile size for example). Below is our current library of tile types we will use in the next steps.



### Measure Basic Knot Style Parameters

The main parameters we need to measure to render appropriate templates to match are:

1. **The strand width** - which refers to the thickness of the stand. One of our assumptions is that the width of all strands are uniform across the whole image.
2. **The gap width** - refers to the distance between parallel vertical or horizontal strands. Used in tandem with the strand width to find the tile size by assuming tile size = strand width + gap width.
3. **The overlap gap width** - This is the space between two intersecting strands, where the “bottom” strand is broken so that there is a gap of negative space (the overlap gap) and the other strand crossing it.
4. **The tile size** - The size of the tile changes based on the image. More complex knots are compressed into a similar frame, meaning our templates tiles need to be smaller to accommodate.

Our approach to find these parameters:

1. Make image grayscale with white strands and black background.
2. Take gradients in dx, dy of grayscale image.
3. Strand and gap width
   1. Filter for vertical line locations by finding points where dx = 0 and dy != 0.
   2. Filter for horizontal line locations by finding points where dx != 0 and dy = 0.
   3. Reduce vertical and horizontal line locations to extremes.
      1. For horizontal extremes, the points where dx = max(horizontal points) or min(horizontal points).
      2. For vertical extremes, the points where dy = max(vertical points) or min(vertical points).
   4. Find the minimum widths between extremes. Apply the following approach for vertical lines. For horizontal lines, do the same thing but transpose the matrix containing values for the horizontal extremes.
      1. Find rows with at least one min and max point.
      2. For each row compare each min and max point to each other:
         1. strand width = max - min
         2. gap width = min - max
      3. Keep track of the minimum strand width and gap width recorded from the previous iterations over the rows and points.
      4. The minimum values for strand and gap width from both the vertical lines and transposed horizontal lines are the final parameters.
4. Overlap gap width
   1. Filter for top left to bottom right diagonal line locations by finding points where dx = dy and dx != 0.
   2. Filter for top right to bottom left diagonal line locations by finding points where dx = -dy and dx != 0.
   3. Apply the method to find minimum widths between extremes to both sets of diagonal line locations.
   4. The minimum gap width between the two types of diagonal lines indicates the leg of a special 45 degree right triangle. By multiplying this leg by the square root of 2, the hypotenuse of this triangle gives the final overlap gap width.
5. Tile size = gap width + strand width
   1. This property comes from parallel line tiles since half a strand is present on two edges with gap width pixels between each.

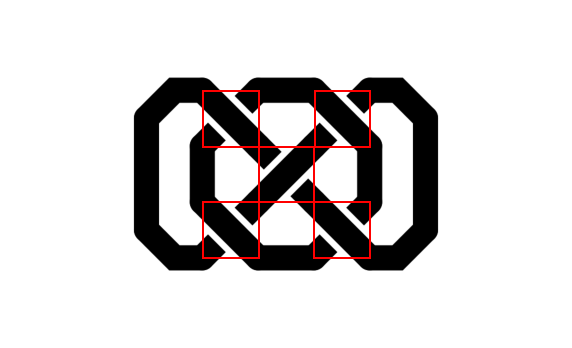
### 

### Template Matching Tiles

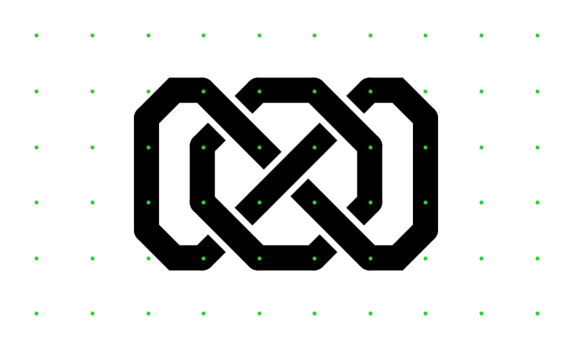
After some development, we have decided to proceed with the following steps to match different tile types to the celtic knot.

1. **Apply template matching on the intersection tiles**.

This will give us initial coordinates to align the grid. Using vary strict template matching thresholds, we can be fairly confident that the placement of these tile matches are close to the true position.

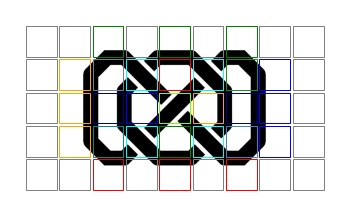


1. **Use one of the matched intersection tiles locations to align the positions and calculate a grid of where tiles should lie**.



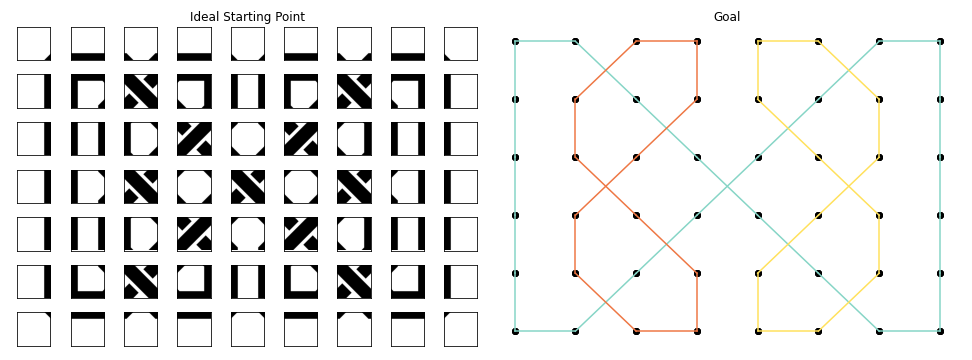
1. **For each grid area, calculate the mean squared error between each template and the patch of image at that grid area.**

Now that we can retrieve patches of the image in the correct grid areas that are the same size as the templates, we only need to make an image comparison. We use the mean square error to calculate the degree of similarity between each patch and each template. We then record which template matched the best and is within a similar threshold.



The result is a grid and the corresponding tile type (depicted with different color squares above) that aligns to that grid area.

### Tile Merging



After prediction, we need to create a graph of nodes using the predicted tiles in order to extract features and render different styles. In an ideal prediction, all tiles would be correctly identified without conflicts. An example of a conflicting situation would be if there was an intersection tile immediately to the right of a tile containing a right edge. This would be an impossible scenario. We do the following to avoid conflicts and produce a graph of nodes:

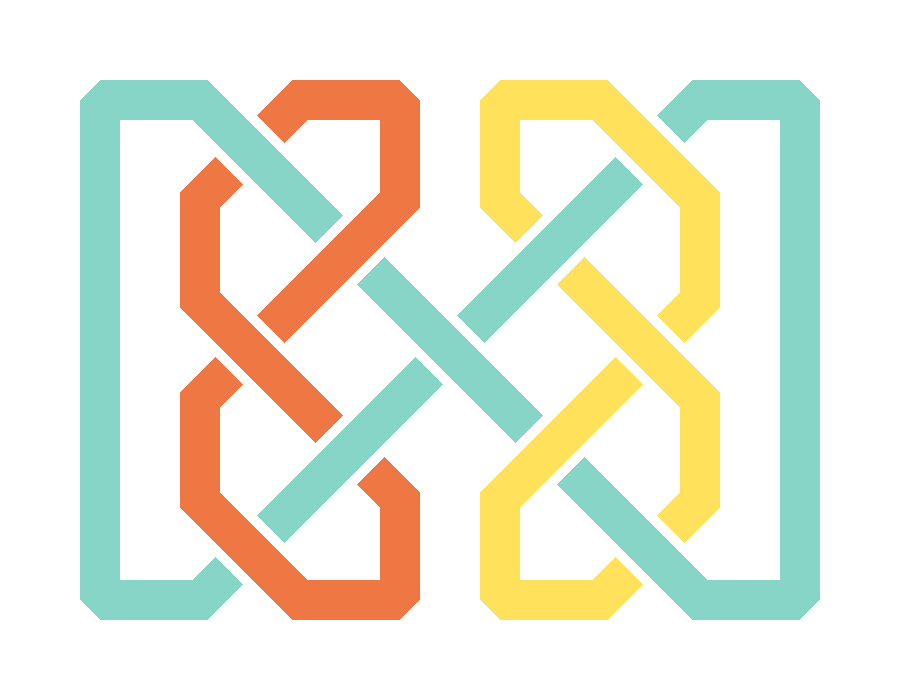
1. **Add “factors” of predicted tiles to factored graph**
   1. Factors are building blocks of tiles such as edges and corners. An intersection tile would be composed of a left diagonal edge and a right diagonal edge.
   2. Predicted tiles add their factors based in the order of the accuracy of their prediction. Intersection predictions are added in accuracy order before all other tile types.
   3. Conflicts may occur, so adding conflicting factors needs to gracefully fail.
2. **Add corresponding “factors” to neighboring tiles**, a non-diagonal edge will always have a corresponding edge in another tile, so a top edge will always have a bottom edge.
3. **Add corners to the factored graph**
   1. Diagonals generate corners in neighboring tiles.
   2. Non-diagonal edges which meet at a 90 degree angle have a corner on diagonal neighbors.
   3. Added after corresponding factors since an edge takes priority over a corner.
4. **Aggregate factors into a single tile type**, the factors are assembled to build a unified type which can be rendered and graphed.
5. **Trim aggregated graph**, the first non-empty top-left and bottom-right tiles define a bounding box for the tile map.
6. **Create and connect nodes based on tile types**
   1. Nodes are marked as present if a tile type renders the node.
   2. Two neighboring nodes are connected together.

Once the node graph is created, then the graph can be run through our rendering system to rebuild a tile image. The graph can also be further analyzed for characteristics such as strands.

### Identifying Tile Strands

By extracting the node graph successfully in the previous step, we are able to identify strands using a simple algorithm:

1. **Pick a node in 2d graph,** label node with unique strand id
2. **For each neighbor of node,** label these neighbors with node’s strand id
   1. Recursively label all of the neighbor’s neighbors with the node’s strand id as well.
3. **Repeat step 1, 2 until all nodes have been labeled**



Once all nodes have been labeled, then the number of strands is the total number of unique labels in the graph. Nodes with the same label belong to the same strand and can easily be separated into their own groups for rendering or further analysis.

## 

## Interpolation

To deal with non-tile based celtic knots, we are attempting to implement a second approach to this problem we have dubbed the interpolation approach. Although intended to be more general we still make a number of assumptions under this model:

1. The strands are a single color with sharp contrast with the background
2. Strands are constant width
3. Intersections are denoted with a “break” in the strand that goes underneath. We call this gap the “overlap width.”
4. The knot work is uniformly “tight,” or that there are no inconsistently large gaps between neighboring strands
5. There are no “dead ends,” i.e., everything forms some kind of continuous loop
6. No “branching” of a single strand. i.e., no strand splits off into two separate strands

The interpolation approach builds off the rather simple heuristic that the direction of the strand after going through an occluded section is very similar to the direction before. We leverage this by modeling the various sections of the knot as a thinned 1-pixel-wide strand then finding the endpoints of these strands and its direction. We then extend this vector a small distance past the occlusion and find the nearest endpoint of another strand with a similar direction. An overview of the steps of this method are as follows:

1. Assign each continuous segment to a cluster membership of points
2. Adaptively thin the fixed width celtic knot until every segment is only 1-pixel-wide keeping track of the distance from a resulting pixel
3. Trim resulting skeleton to create a curve without branches
4. Estimate the normal at endpoints of curve
5. Use normal to get an estimate of tangent at each endpoint. Create a vector 4 times the length of medial axis distance and in the direction of the tangent rooted at the endpoint
6. Connect to the nearest endpoint from another curve and check the angle is below a threshold. If it is, merge these two curves labeling the the new one with the cluster membership of the original
7. Repeat until there are no unconnected endpoints

### Preprocessing

1. **Binarize the image** If the image is not already a binary image make it so via thresholding
2. **Label Segments** Use connectivity tests to assign all pixels of a continuous region a cluster ID
3. **Adaptively Thin Segments** Adaptively thin the fixed width celtic knot until every segment is only 1-pixel-wide, keeping track of the distance from a resulting pixel.

### Skeleton Trimming

1. **Medial Axis Trimming** The adaptive thinning method above gives us the distance from each pixel of the resulting skeleton to the closest point on the original boundary, which is called the medial axis distance. In other words, this is how much the shape had to be thinned around that pixel. Since the curve is a fixed width, the values are very centralized and thus simple thresholding a small distance below the mode yields good results in our tests.
2. **Tree-Based Trimming** The above trimming still leaves very short flanges that would otherwise mess with the interpolation of the curve. To fix this, we find all points of degree 3 and the pixels on the path from such a point to endpoints or points of degree 1. The shortest of these subpaths is trimmed to give us our final trimmed skeleton

### Estimate Endpoint Direction

1. **Blur Skeleton** Since the curve of a 1d line is discretized, taking the gradient of it now will lead to erratic values (source: <https://stackoverflow.com/questions/48322740/curvature-of-a-one-pixel-wide-curve>). So we apply a gaussian blur to the image with a small radius
2. **Get Normal Estimate** The direction of the gradient from each pixel of the original 1d curve is now a good approximation of the normal

### Interpolate Endpoints and Connect Curves

1. **Interpolate Endpoint Direction** Take the normal vector at a given endpoint and find the endpoint that is closest to the end of that vector.
2. **Connect Endpoints** Update the cluster membership of the paired endpoint and all points within its cluster to be the same as the starting endpoint.
3. **Repeat** If this segment still has an unmatched endpoint, use that as the next starting endpoint. Otherwise, pick a random segment endpoint. Return to step 1, using the selected next starting point as the given endpoint.

### Reassemble and Identify Structures of the Knot

1. **Map cluster memberships** From the previous section, we now know the cluster IDs for the *skeletonized* segmented representation of the knot. However, to color the segments in the original image, we need to map the cluster IDs from the original image segmentation to the skeletonized segmentation.
2. **Display colorized, segmented knot** After mapping the cluster IDs, we are able to show our end result, a colorized version of the knot, with the different strands of the knot shown in different colors.

# 

# Experiments and Results

We tested the tile-based and interpolation approach’s abilities to do the following:

1. Identify the individual strands composing a celtic knot of a standard width
2. Width-invariant spline detection - The method detects the same spline regardless of the width of the strands
3. Count the number of strand components

## Datasets

We tested each approach on two different kinds of images. First, we generated a set of “standard” Celtic knots using our fork of the [rspencer01/celtic](https://github.com/rspencer01/celtic) library. The images in this test set were of varying complexities (number of strands, number of crossovers). We created three copies of this dataset using different widths, allowing us to test the performance of our two methods on knots of varying widths.

Test set: standard images:

*Include the standard test set images we used*

We had a second test set of more complicated curvy knots. These knots were designed to push the limits of our algorithm.

*Include the curvy test set images we used*

### 

### Comparing Results Across Methods

## In this section, we show a comparison in ability to correctly identify the component strands of a celtic knot using the tile-based method, the interpolation method, as well as a basic image segmentation approach as our reference basic approach.

#### Basic Image Segmentation Results

## In the trivial approach, we simply took each discrete component of the knot image as its own strand. This approach only worked for the most basic knots with just one strand made up of one component.

## (Large table; scroll right to see the whole thing)

<table>

#### Tile-based Method Results

<table>

#### Interpolation Method Results

<table>

### Comparison of Methods

Segmentation only works with knots with only one strand made up of one component. The tile-based approach works well under more “average” circumstances, such as medium complexity knots and thin strands. The interpolation approach works generally well for less complicated images, but otherwise performs similarly across all the different strandwidths.

In the next sections, we will analyze the performance of the tile-based and interpolation methods.

## Tile-based Method Performance

### Tile parameter estimation for the generated knots

**Table 1.1 Expected Tile Parameters Values in Pixels**

These values were manually measured after the images were generated. Note that in all images, regardless of the strand width, that the size of the ideal tiles are 61 x 61 pixels large.

| **Strandwidth Type** | **Thickness parameter** | **Tile Size** | **Strand Width** | **Overlap Width** |
| --- | --- | --- | --- | --- |
| **Xthin** | 0.25 | 61 | 9 | 3 |
| **Thin** | 0.5 | 61 | 16 | 4 |
| **Regular** | 1 | 61 | 31 | 8 |

**Table 1.2 Tile Parameter Estimation RMSE**

We had 4 base knot patterns. Each one had 3 variations, each with different strandwidth types. For each tile parameter, we compared the true tile parameters (Table 1.1) with the parameters estimated by our method across all 4 knot patterns using RMSE.

| **Strandwidth Type** | **Tile Size** | **Strand width** | **Overlap width** |
| --- | --- | --- | --- |
| **xthin** | 0.8660254 | 1.80277564 | 1.58578644 |
| **thin** | 0.5 | 1.0 | 2.58578644 |
| **regular** | 0.70710678 | 0.8660254 | 0.48528137 |

### Tile matching accuracy

We only tested on the generated knots, since our tile set would not work on the curvy images. The generated knot test set varied in the knot pattern complexity and the strandwidth: regular, thin, and extra thin (xthin).

**Table 1.3 Grid Dimension Correctness**

In order to compare our tile prediction with the true tiles, we must have the same grid dimensions as the true tile arrangement (number of tiles). A fast check to check the accuracy of the tile approach is to determine if we got the correct number of non-empty tiles, ignoring borders.

**Table 1.3.a Grid Dimension Correctness WITHOUT Tile Size Adjustment**

| **Strandwidth Type** | **# Correct Grid Dimensions** | **# Incorrect Grid Dimensions** |
| --- | --- | --- |
| **xthin** | 0 | 2 |
| **thin** | 2 | 2 |
| **regular** | 4 | 0 |

We found that the “regular” strandwidth versions of the test knot patterns worked well, but not as well for “thin” and “xthin” (not getting any of them correct).

**Table 1.3.b Grid Dimension Correctness WITH Tile Size Adjustment**

We found that the major issues with the alignment was that the grid was not correctly sized. It was usually off by a few pixels. This is likely due to the error shown in **Table 1.2**. In smaller image the inaccuracy does not impact the tile as much, but in images with more tiles, the error compounds. We manually made some adjustments to the estimated tile size, and found it increased the number of viable grid matches. The thinner the strand, the more likely our parameter detector underestimates the tile size.

| **Strandwidth Type** | **Tile Adjustment (pixels)** | **# Correct Grid Dimensions** | **# Incorrect Grid Dimensions** |
| --- | --- | --- | --- |
| **xthin** | +3 | 2 | 4 |
| **thin** | +2 | 3 | 1 |
| **regular** | +0 | 4 | 0 |

**Tables 1.4 Tile Matching Accuracy**

Total Count refers to the number of tiles that are supposed to be in the test images collectively. % correct refers to the proportion of times our tile matching correctly matched the true tile. **This ignores the test cases where our final detected tile grid dimensions do not match the true grid dimensions.** We could only determine the number of matched tiles automatically when we knew the grids were the same size. Refer to Table 3.1.b to view how many images we did not consider for the following analysis. Note that if we included the images that did not match, the accuracies below would be lower.

**Table 1.4a Tile Matching General Accuracy Per Strandwidth**

| **Strandwidth Type** | **Accuracy** |
| --- | --- |
| **Overall** | 0.42 |
| **regular** | 0.29 |
| **thin** | 0.52 |
| **xthin** | 0.48 |

**Table 1.4b Tile Matching Accuracy Per Tile Type**

| **Strandwidth Type** | **Overall** | | **Regular** | | **Thin** | | **Xthin** | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Tile Type** | **Total count** | **% correct** | **Count** | **% correct** | **count** | **% correct** | **count** | **% correct** |
| **BOTTOM\_EDGE** | 39.0 | 1.0 | 17.0 | 1.0 | 15.0 | 1.0 | 7.0 | 1.0 |
| **LEFT\_HORSESHOE\_EDGE** | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **RIGHT\_HORSESHOE\_EDGE** | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| **RIGHT\_EDGE** | 47.0 | 0.94 | 20.0 | 0.95 | 19.0 | 1.0 | 8.0 | 0.75 |
| **TOP\_EDGE** | 41.0 | 0.78 | 18.0 | 0.5 | 16.0 | 1.0 | 7.0 | 1.0 |
| **PARALLEL\_VERTICAL** | 60.0 | 0.75 | 26.0 | 0.42 | 26.0 | 1.0 | 8.0 | 1.0 |
| **TOP\_HORSESHOE\_EDGE** | 4.0 | 0.75 | 2.0 | 0.5 | 2.0 | 1.0 | 0.0 | 0.0 |
| **LEFT\_INTERSECTION** | 78.0 | 0.69 | 34.0 | 0.38 | 33.0 | 0.94 | 11.0 | 0.91 |
| **LEFT\_EDGE** | 45.0 | 0.69 | 19.0 | 0.47 | 18.0 | 0.89 | 8.0 | 0.75 |
| **RIGHT\_INTERSECTION** | 55.0 | 0.53 | 25.0 | 0.24 | 25.0 | 0.8 | 5.0 | 0.6 |
| **BOTTOM\_HORSESHOE\_EDGE** | 2.0 | 0.5 | 1.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 |
| **PARALLEL\_HORIZONTAL** | 28.0 | 0.39 | 14.0 | 0.21 | 14.0 | 0.57 | 0.0 | 0.0 |
| **ALL\_CORNERS** | 56.0 | 0.0 | 26.0 | 0.0 | 26.0 | 0.0 | 4.0 | 0.0 |
| **LEFT\_EDGE\_W\_CORNERS** | 30.0 | 0.0 | 13.0 | 0.0 | 13.0 | 0.0 | 4.0 | 0.0 |
| **RIGHT\_EDGE\_W\_CORNERS** | 26.0 | 0.0 | 11.0 | 0.0 | 11.0 | 0.0 | 4.0 | 0.0 |
| **BOTTOM\_CORNERS** | 22.0 | 0.0 | 9.0 | 0.0 | 8.0 | 0.0 | 5.0 | 0.0 |
| **TOP\_CORNERS** | 20.0 | 0.0 | 8.0 | 0.0 | 7.0 | 0.0 | 5.0 | 0.0 |
| **BOTTOM\_EDGE\_W\_CORNERS** | 19.0 | 0.0 | 9.0 | 0.0 | 9.0 | 0.0 | 1.0 | 0.0 |
| **TOP\_EDGE\_W\_CORNERS** | 17.0 | 0.0 | 8.0 | 0.0 | 8.0 | 0.0 | 1.0 | 0.0 |
| **BR\_CORNERING\_EDGE\_W\_CORNER** | 17.0 | 0.0 | 7.0 | 0.0 | 7.0 | 0.0 | 3.0 | 0.0 |
| **TL\_CORNERING\_EDGE\_W\_CORNER** | 15.0 | 0.0 | 6.0 | 0.0 | 6.0 | 0.0 | 3.0 | 0.0 |
| **TR\_CORNERING\_EDGE\_W\_CORNER** | 15.0 | 0.0 | 6.0 | 0.0 | 6.0 | 0.0 | 3.0 | 0.0 |
| **BL\_CORNERING\_EDGE\_W\_CORNER** | 15.0 | 0.0 | 6.0 | 0.0 | 6.0 | 0.0 | 3.0 | 0.0 |
| **TL\_CORNER** | 9.0 | 0.0 | 4.0 | 0.0 | 3.0 | 0.0 | 2.0 | 0.0 |
| **TR\_CORNER** | 9.0 | 0.0 | 4.0 | 0.0 | 3.0 | 0.0 | 2.0 | 0.0 |
| **BL\_CORNER** | 9.0 | 0.0 | 4.0 | 0.0 | 3.0 | 0.0 | 2.0 | 0.0 |
| **BR\_CORNER** | 9.0 | 0.0 | 4.0 | 0.0 | 3.0 | 0.0 | 2.0 | 0.0 |
| **LEFT\_CORNERS** | 6.0 | 0.0 | 3.0 | 0.0 | 3.0 | 0.0 | 0.0 | 0.0 |
| **RIGHT\_CORNERS** | 4.0 | 0.0 | 2.0 | 0.0 | 2.0 | 0.0 | 0.0 | 0.0 |

This table excludes Tile types that did not occur in the test images that we were able to get the correct grid dimensions. The poor accuracy is partially due to the fact we do not have all the possible tile types implemented. Therefore, many tiles are matched to the “empty” type or another similar tile. In the next step to determine the splines, we try to account for this possibility.

The strand identification results can be seen in Table 0.2.

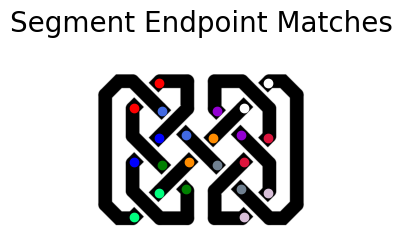
### Tile Based Spline Detection Results

## Interpolation Method Performance

Results:

See [Experiment Results](https://docs.google.com/spreadsheets/d/1bpmPJ0GhHn7oT2m9m0D-THFt7trEb4a4pc-PwLf7bXw/edit#gid=0)

In reviewing the performance of the interpolation method, we noticed that its performance decreased as knots became either more complex or curvier. Some of the reasons for this include:

* **Skeletonization errors.** To get accurate results, the skeletonization of the image must be correct. As shown in the qualitative results section, curvy and/or complex knots can throw off our skeletonization or tree-trimming steps. If the skeleton of the knot is incorrect, this can result in incorrect segment endpoint matching, explained below.
* **Incorrect segment endpoint matching.** To generate the final image, we must first identify associations between the different discrete segments that form the knot. In this step of our algorithm, we iterate through each segment endpoint and use our direction estimation vectors to find the most likely match for a given segment. Ideally, this results in something like the one below, where we successfully connect the segments across knots because our gradient estimation gave us an accurate direction estimation. We show in-depth examples of segment endpoint matching failure cases in the qualitative results section. However, common reasons that caused segment endpoint matching to fail (i.e., an endpoint was not matched with the endpoint it should have been matched with) were:
  + Incorrectly placed endpoints due to skeletonization errors
  + Incorrect direction vectors (often the result of skeletonization errors or sharp turns in the strand)

However, there were also some strengths of the interpolation algorithm:

* **Flexibility.** In contrast to the tile-matching method, the interpolation method was able to gracefully handle knots of varying strand widths. In fact, we noticed that performance improved slightly as the strand width decreased.
  + Additionally, while the interpolation algorithm as we wrote it struggled on the curvy images, this was primarily due to the skeletonization of the image. If we had more time, there are additional improvements we could make to further refine the accuracy of the interpolation algorithm on curvy knots.
* **Ease of use.** The interpolation algorithm only requires the image as its input and requires no extra parameters for black and white images. On the other hand, the tile-method requires forming a complex set of tiles to represent the different possible components of a knot.

# Qualitative Results

## Tile-based Method

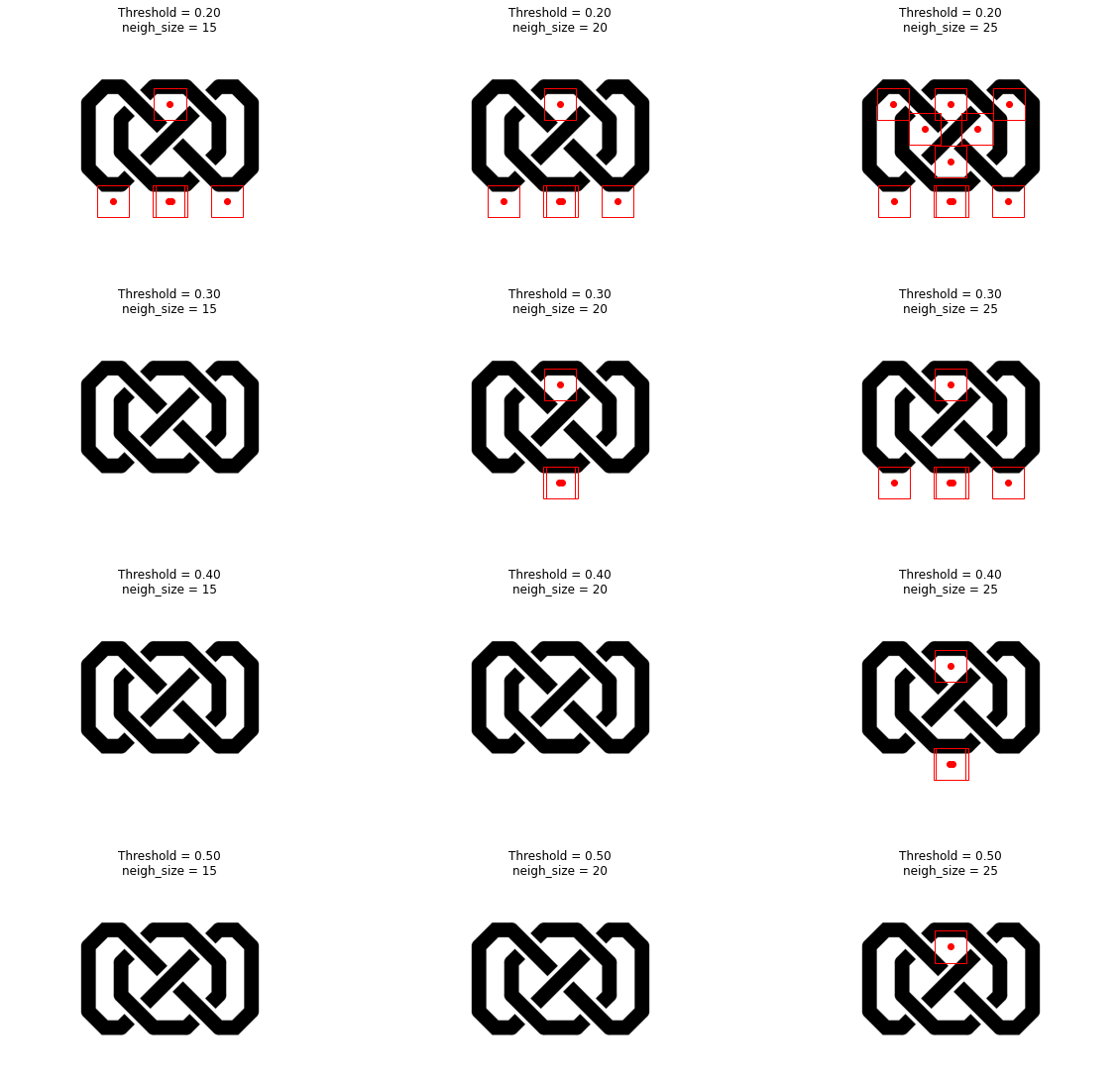
### Tile template matching viability

Our initial approach to tile template matching was to apply scikit-image’s match\_template function to all tiles. For each tile, match\_template would return a results matrix with “scores” representing how well the tile matched at each location in the image. We then used scipy.ndimage.filters maximum and minimum following [this suggestion on Stackoverflow](https://stackoverflow.com/questions/9111711/get-coordinates-of-local-maxima-in-2d-array-above-certain-value) to detect all the peaks in the results. Maximum and minimum take two parameters: threshold (difference between the maximum and minimum values) and neighborhood size (window size). These parameters needed to be adjusted so we could properly detect only the parts of the image that matched a particular tile. Assuming that template matching worked well, each part of the knot would match to at most one tile type.

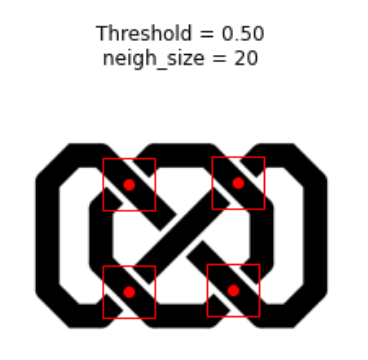
However, the main problems we have with this approach run into are:

1. **Tile positioning ambiguity**. For example, there can be a long stretch of just a horizontal strand. A template recording a horizontal line can match at any point along that horizontal strand. Additionally, we can get matches of a tile that would not be inline with other tiles (grid like formation).
2. **Multiple or incorrect matches.** Multiple tiles may match at the same place. Clearly, there can only be one match. It is difficult to directly resolve which one should match when matches are determined on a per-tile basis.
3. **Different tile types would require different parameters**. Due to the level of “complexity” some tiles are more likely to match in areas it is not supposed to be in.

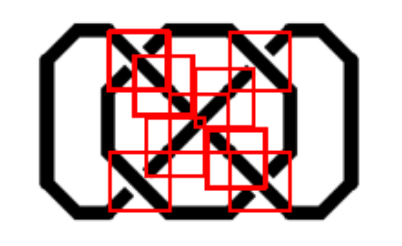
We could play with the parameters to prevent nearby matches and incorrect matches for each tile. However, below we demonstrate below that finding the right parameters is tedious, if not impossible, and not robust. Additionally, tile matches may not be properly aligned in a grid like fashion due to noise (as seen by two heavily overlapping tiles matching in similar locations in the bottom center of the figures).



We could also apply some clustering algorithm to resolve overlapping tile matches, but that does not resolve the grid and alignment problem. We found that more complex tiles, such as the intersection tile types, appear to be more reliable to map accurately given reasonable parameters.



We attempted to apply a calibration stage to align the located intersection points. However, it was highly sensitive to instances where the insterection tiles were incorrectly matched. In some test images, we would pick up many extraneous “intersections” such as the one below.



Here, we see that with a thinner strand, our template matching was more likely to mistaken diagonals for intersections, even with a higher template matching and peak matching threshold. In either case, the calibration phase was not helpful, so we decided to remove it and run on the heuristic that the top left detected intersection is actually an intersection.

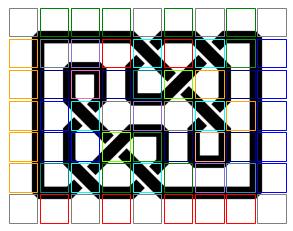
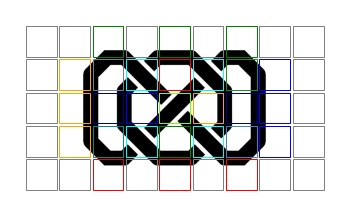
This leads us to the method described in the Tile-based approach section.

### 

### Testing different knots

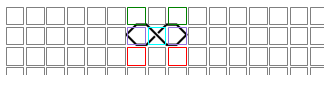
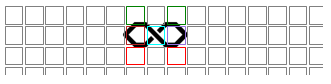
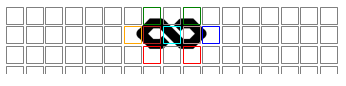
Below, we tested different knots with varying degrees of complexity, namely the number of intersections, irregular knot patterns, and size. We used the resulting knots from making the tile size adjustments as noted in **Table 1.3.b.**

Each color square represents a different tile type. The intersection tile styles are in cyan (top left to bottom right diagonal on top) and lime green (top right to bottom left diagonal on top). The knot parameter estimation stage and tile matching appears to work fairly well (given the assumptions) for small, simple knots.



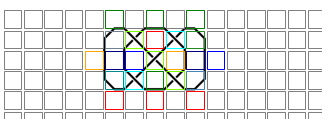
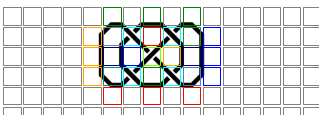
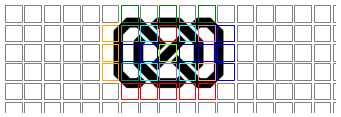
**Simple Knot**

This knot is a single strand and contains only a single intersection. We see that with thinner strands, we are more likely to “lose” the outside edges of the knot.



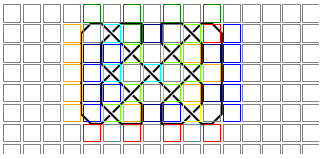
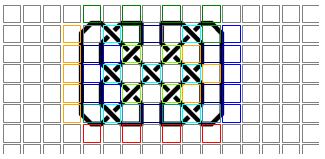
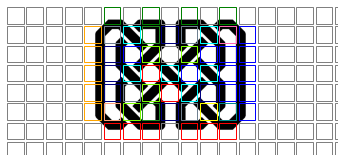
**Infinity + Ring Knot**

This is a slightly more complicated knot pattern with two intertwined strands. With a thicker “regular” strand, we see the method is more likely to mismatch with another non-empty tile (the red bottom edge). This may throw off the spline detecting stage when we are detecting connected nodes.



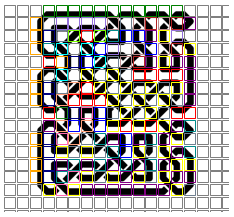
**Triple Infinity Knot**

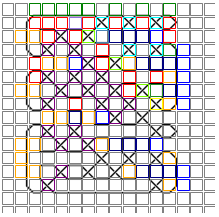
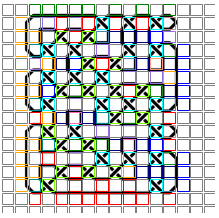
This knot has 3 intertwined “infinity” loops. In the thicker “regular” knot, we can see that the intersections were not all properly detected. There is a yellow square at the bottom right, which denotes a “top left to bottom right diagonal” tile type. This is likely due to the slight misalignment of the grid. However, the thinner strand versions were more robust to that misalignment.



**Maze Knot**

We created this knot to be (1) non-regular, (2) large, and (3) contain 4 independent strands.The tile grid on the thicker “regular” strand version is much more misaligned (the entire right edge is within a tile) compared to the thinner versions. However, the xthin version is more likely to miss the outer edge tiles (it misses bottom edge) since the strand itself is so thin.

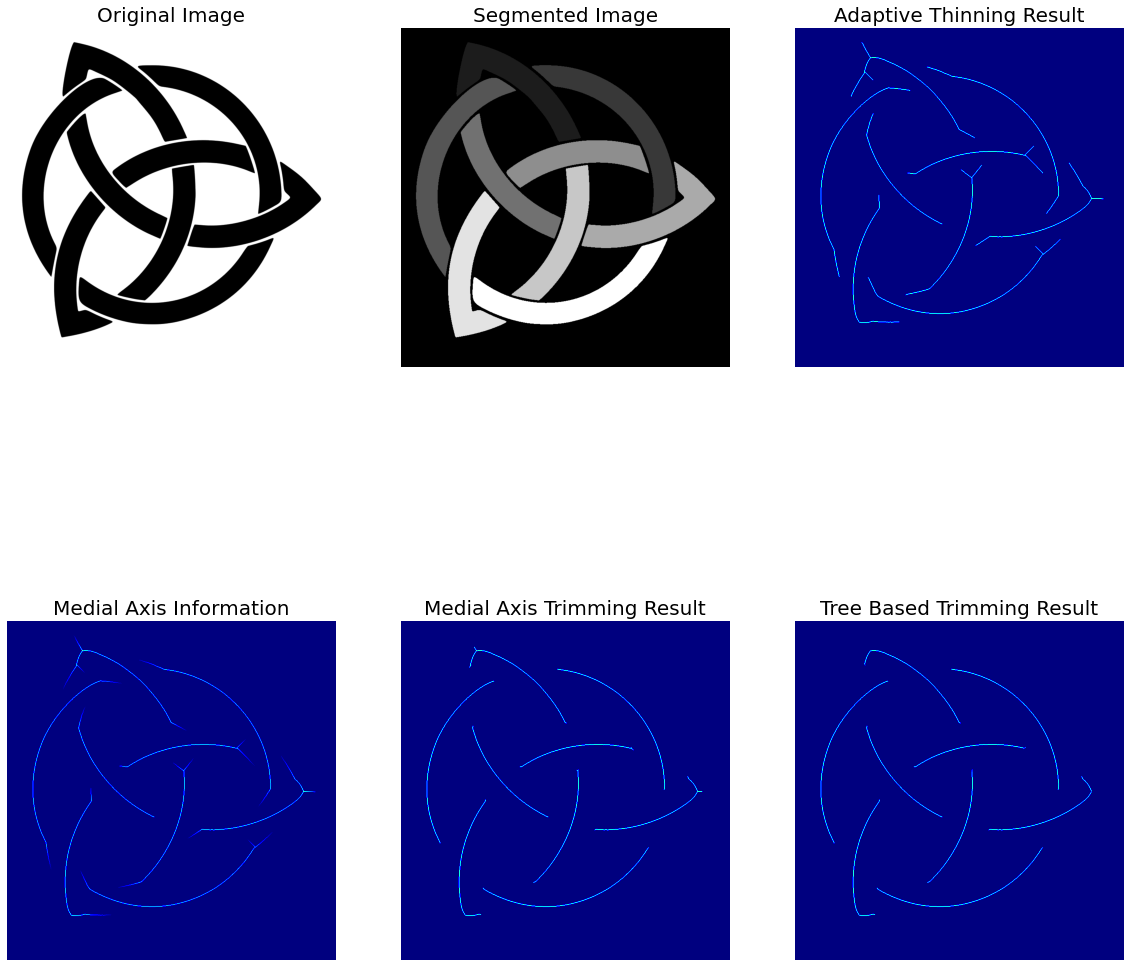




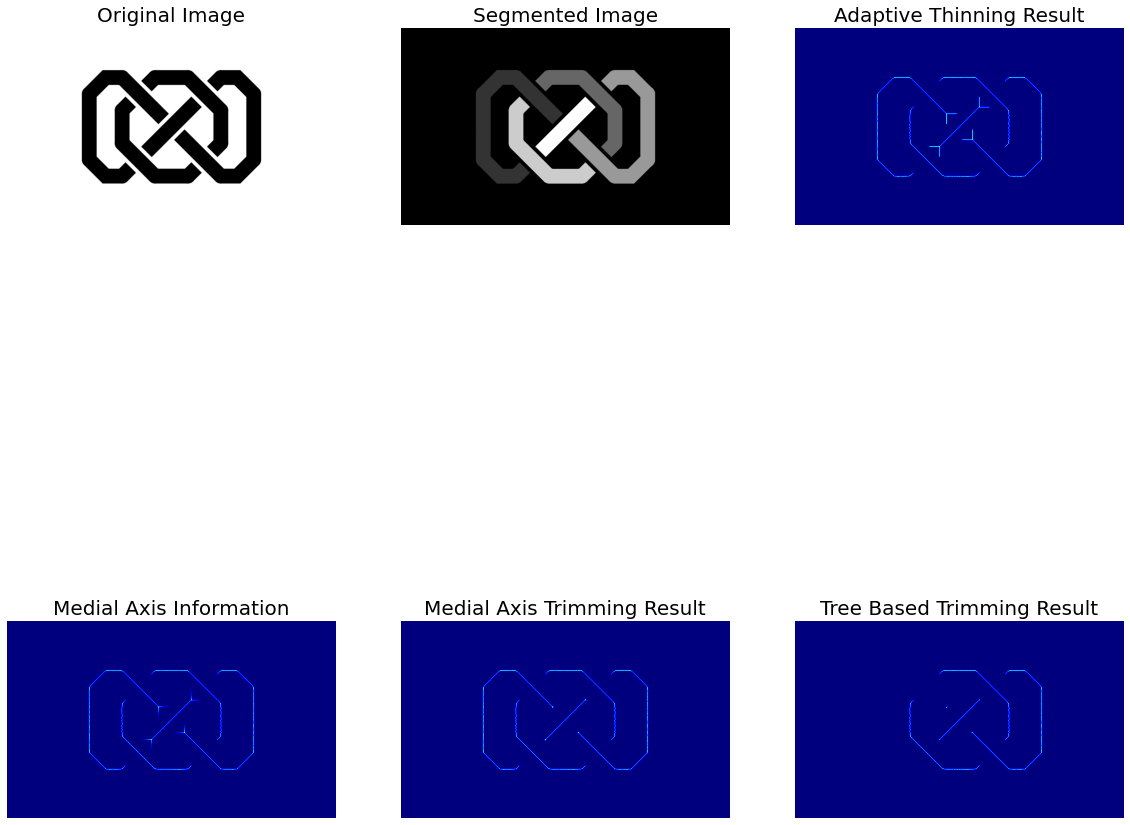
## Interpolation method

Right now, our interpolation method is complete through the Skeleton Trimming description described earlier.

A successful example of skeleton trimming is shown below. As you can see, the adaptive thinning step left a number of extra slanted ends present that should not be present. However, after the medial axis and tree-based trimming, the ends of the paths shown are cleaner, setting the stage for interpolating the line segments to fill in the empty spaces.



However, skeleton trimming is not always successful. For example, on the image below, you can see a case where tree-based trimming was overly aggressive and removed a very large path. You can also see that in this case, skeleton trimming may not even be necessary. We plan to investigate reasons for this failure case to continue refining our skeleton trimming processes.



# 

# Conclusion and Future Work

In working on this project, we have found that while it is difficult and complicated to identify the strands making up a celtic knot, it is very much possible to do. We have found two approaches, one tile-based and one using interpolation, that are both promising solutions to this problem and have different strengths and weaknesses.

The tile-based approach works theoretically, but has several dependent stages. The success of the tile matching is dependent on the parameter searching to return the correct tile parameters. The success of the tile merging and spline detection depends on the tile matching stage to correctly find the grid and match the tiles. In the results, we see that with larger complexity and varying width, the performance at each of these stages are impacted, and require manual adjustments. In all, the approach works well under perfect conditions, but would quickly begin to fail once those conditions introduce noise.

Perhaps there are ways to refine each of these stages and make them robust to errors in teh previous stage. For example, the tile merging stage does implement a priority based predictor to determine which tiles can be reliably used to create the final spline. Further work would also include making more tile templates, which would likely increase our tile matching accuracy. However, this approach is extremely specific, and would at best only work on the images generated by the rspencer/celtic library. If we were to continue this method, we would focus on making a generalized approach for generating the tile templates. For example, given a library of knots of a similar style, we can make a method that automatically generates the library of tile types. This may make our current method more generalizable to all kinds of knots. Additionally, if this tile method improves, we could also theoretically identify large motifs of repeated knot elements, denoted by a particular arrangement of tiles.

We found that the interpolation approach was more flexible but reliant on our ability to accurately skeletonize an image, which was one of the most difficult parts for us to implement. Our results for interpolation were very promising, showing a high degree of flexibility in terms of the strand width of the image. One of the biggest findings in our results however, was that the interpolation approach didn’t work well on most knots with curves or on larger generated knots. In the future, more work could be done to improve the performance of the interpolation approach on these images. One area of improvement is the skeletonization algorithm, which is one of the biggest points of failure, especially on curvy images. A second area of improvement is the segment endpoint matching stage; adding checks to validate that the angles of a proposed match are similar and preventing endpoints on the same segment from being identified as a match would both improve the performance of this stage.

We have shown that it is possible to analyze celtic knots, opening the door to future applications of these methodologies. In the future, additional work could go into not only improving our algorithms, but also using them to analyze parts of the real world, such as complex highway interchanges, tree branches, and rope knots.

All of the source code can be found at <https://github.gatech.edu/cv-project>.

# 

# References

* numpy
* scikit-image
* opencv
* matplotlib
* imageio
* scipy
* [rspencer01/celtic: Display of various celtic knots](https://github.com/rspencer01/celtic)
* [skan](https://jni.github.io/skan/index.html)
* <https://homepages.inf.ed.ac.uk/rbf/HIPR2/fourier.htm>
* <http://scipy-lectures.org/intro/scipy/auto_examples/solutions/plot_fft_image_denoise.html>
* <https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_transforms/py_fourier_transform/py_fourier_transform.html>
* <https://peteris.rocks/blog/extrapolate-lines-with-numpy-polyfit/>
* <https://stackoverflow.com/questions/48322740/curvature-of-a-one-pixel-wide-curve>
* <https://www.pyimagesearch.com/2016/02/01/opencv-center-of-contour/>
* <http://opencvpython.blogspot.com/2012/05/skeletonization-using-opencv-python.html?m=1>
* <https://stackoverflow.com/questions/33095476/is-there-any-build-in-function-can-do-skeletonization-in-opencv>
* <https://scipy-cookbook.readthedocs.io/items/RadialBasisFunctions.html>
* <https://stackoverflow.com/questions/61103400/fit-a-spline-on-image>
* <https://pomax.github.io/bezierinfo/>
* [[PDF] An Algorithm for Curve Identification in the Presence of Curve Intersections](https://www.semanticscholar.org/paper/An-Algorithm-for-Curve-Identification-in-the-of-Gnecco/9a205572bea220b4afc61eb92a65904dcdabc2b3)
* [CURVE TRACING AND CURVE DETECTION IN IMAGES](https://www.semanticscholar.org/paper/CURVE-TRACING-AND-CURVE-DETECTION-IN-IMAGES-Raghupathy/48a12fb85ad6f1f5f0f43f8e48a8ee5f836892f6?p2df)
* <http://hypatiastudio.com/celticknots/>
* <https://stackoverflow.com/questions/9111711/get-coordinates-of-local-maxima-in-2d-array-above-certain-value>