Salisbury
UNIVERSITY

Honors College

Honors Thesis

An Honors Thesis Titled

Geographical Coastline Detection Through Image Processing Techniques

Submitted in partial fulfillment of the requirements for the Honors Designation to the

Honors College

of

Salisbury University

in the Major Department of

Computer Science

by

Robert Stancil

Date and Place of Oral Presentation: April 2017, Henson Science Hall (SuSRC)

Signatures of Honors Thesis Committee

	Signature	Print
Mentor:	[signature]	Dan Spicklemire
Reader 1:	[signature]	Steven M. Hetzler
Reader 2:	[signature]	Mara Chen
Director:	[signature]	James J. Buss

ABSTRACT

"Geographical Coastline Detection Through Image Processing Techniques"

This paper focuses on the steps taken and results obtained when trying to manipulate and process coastline images. Included is information about applying edge-detection algorithms to obtain the coastal boundary line that forms between the water and land at coastlines, reading in this edge to create a vector representation of the coastline, and performing analysis do determine how this representation changes over time to predict changes in coastlines. Different forms of optimization in terms of speed as well as accuracy are tested and evaluated in order to make results as precise as possible. The challenges that have occurred during research are discussed alongside the successes in order to show the design process followed.

Selecting and Preparing Images

The central focus of this research project revolved around working with coastlines and various image processing techniques in order to allow for analysis of changes over time. Geographical images were obtained through Caitlin Curry and Dr. Mara Chen from the Geography department at Salisbury University. These images came in varying geographical formats that were ultimately exported into Tagged Image File Format (TIFF) and Portable Network Graphics (PNG) files in order to preserve quality while being analyzed.

An example of an exported coastline image

These images had extra information that is unnecessary in comparing the changes in coastlines over time. The only thing that is needed to perform this analysis is an outline of the edge where the water meets the land. Several methods to obtain this edge were performed including using the GIS software IDRISI (Clark Labs), using Python (Python Software

Foundation) with the OpenCV package (Itseez), and using edge-detection algorithms we found in GIMP (The GIMP Team).

The first method that was tried in order to obtain the edge of a coastline was through using the Python OpenCV plugin. This plugin includes implementation of the Canny edge detection algorithm. The following is code used to run this algorithm along with comments on what each command does:

```
import cv2                                      # imports cv2 plugin
from matplotlib import pyplot as plt            # imports matplotlib (allows images
                                                # to be shown side to side to
                                                # compare)

img = cv2.imread('0708aer.png',0)               # loads in image of coastline
edges = cv2.Canny(img,50,150)                   # performs Canny edge detection of
                                                # img with given threshold (50,150)

plt.subplot(121),plt.imshow(img,cmap = 'gray')  # plots black and white
                                                # version of original image
plt.title('Original Image'), plt.xticks([]), plt.yticks([])

plt.subplot(122),plt.imshow(edges,cmap = 'gray')# plots image after edge
                                                # detection algorithm
has                                             # been performed
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])

cv2.imwrite('edge.png', edges);                 # saves generated image as edge.png

plt.show()                                      # displays original image next to
                                                # edge detected image
```

After running the code, the generated image is visible through the output of the program and is also saved as a new image. Due to the amount of noise in the coastline images, it is hard to get an accurate edge without a lot of extra information showing up. The threshold being used for the

Canny algorithm can be edited to try and refine what shows up in the output but it is still difficult to remove noise without losing the edge itself.

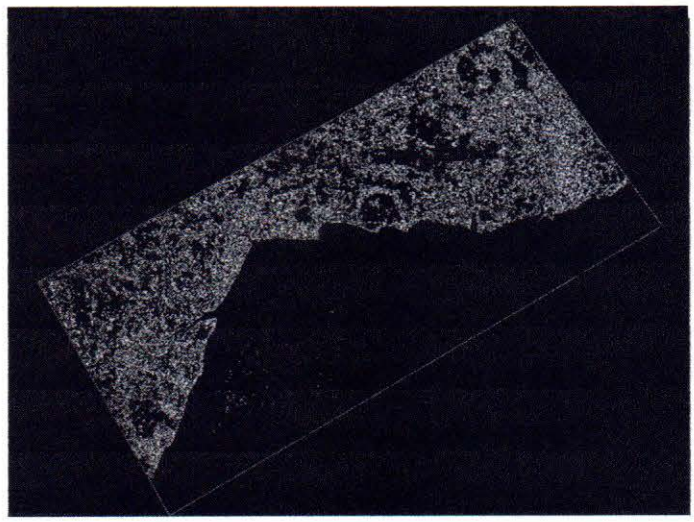


Image output from Python code

In order to accurately analyze the change in location of the coastlines over time a clear edge is required so that comparisons can be made from year to year. This would require a better edge detection method than simply using this Python code. GIMP is an open-source image manipulation program that contains a wide range of tools for working with pictures. Through trial and error the following method was found:

1. Load the image into GIMP.

2. Perform a Sobel edge detection through Filters -> Edge-Detect -> Edge and change the algorithm to Sobel and the amount to 1.0. This will perform an edge detection and generate an image.

Image in GIMP after performing Edge-Detection

3. Using the fill tool and a threshold of 15, change the color of the water to a color that doesn't appear anywhere in the image such as a bright green.

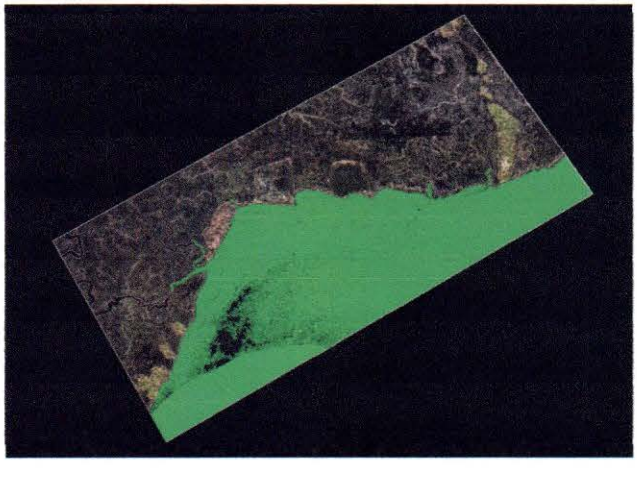
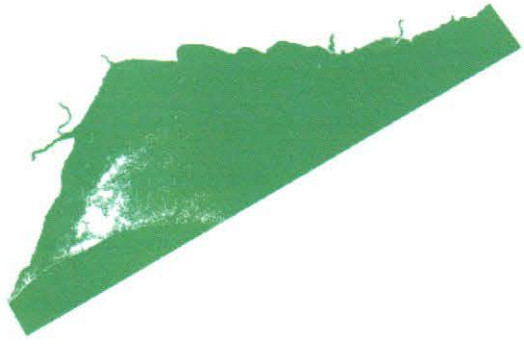


Image after filling in water bright green

4. Using the select by color tool, select all of the water. Copy this selection and copy it into a new GIMP file. This will leave you with an image of just the water.



Current image after copying water to new GIMP file

5. Still using a threshold of 15, fill the background of the image black. Then change the Image's mode to greyscale.

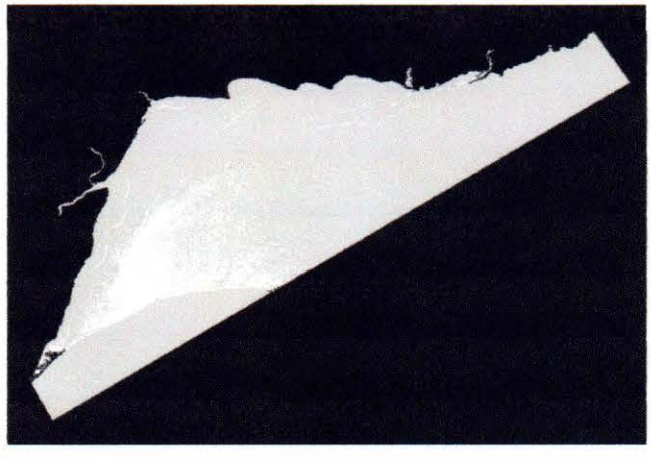
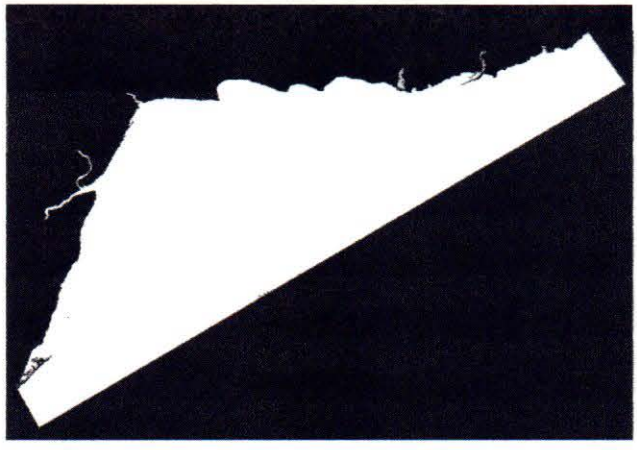


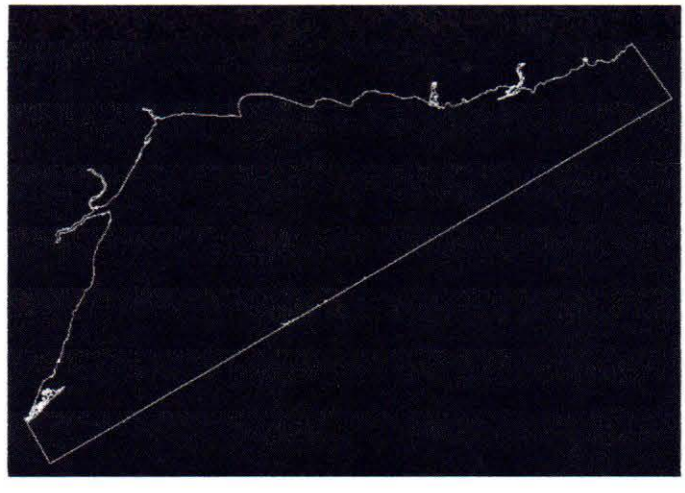
Image after changing background color and mode to greyscale

6. Using the fill tool and a high enough threshold to cover any leftover noise (such as 120), fill the water white.



After filling the water

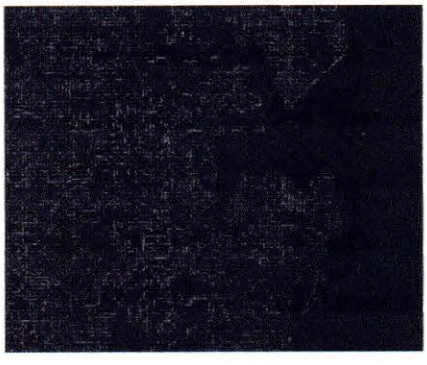
7. Finally, perform one last edge-detection using the same steps as before.



Final image after performing edge-detection

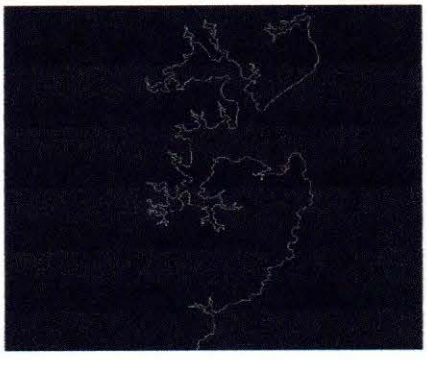
After completion of the above steps, the edge between land and water has been pulled out of the coastline image. This gives a cleaner representation that can be vectorized and manipulated. We tried each of the different methods and did our best to refine them. After testing and comparing the resulting output we found that using the edge-detection facilities in GIMP gave the cleanest coastline. Here is a side by side comparison of three of the methods we tried:

Original Image



IDRISI Edge Detection

Python OpenCV Edge Detection



GIMP Edge Detection Method

Pixel by Pixel Image Scanning

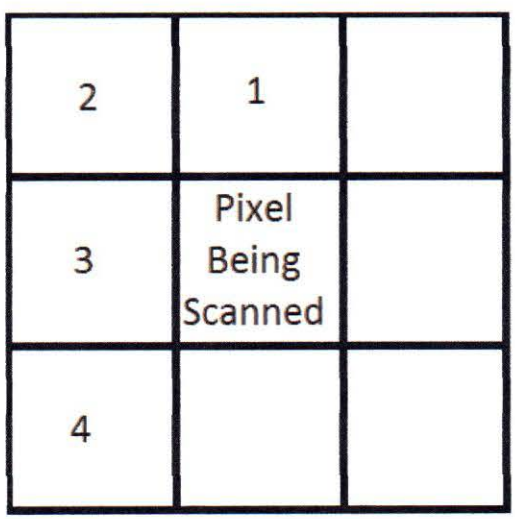
The GIMP edge-detection process generates an image that contains an edge made up of many pixels. The problem with this is that in order to find equations to monitor the change in the coastline over time a vector representation of the water-land boundary is needed. Throughout most of the generated edge there are several pixels making up this boundary. This needs to be limited to a line formed of pixels connected to another single pixel.

The first step in fixing this is to read in the image pixel by pixel and store this information in an array. Each pixel in the image will have a corresponding cell in the array to hold the value of its color. This will be whether the pixel is white and is part of the edge or is black. Using the ImageIO package in Java it is possible to programmatically open an image file. Through this it is possible to get the height and width of the image in pixels and create an array to match this size. The ImageIO package has a function getRGB(x, y) that enables getting the RGB (red, green, and blue) color values at position (x, y) in an image. Through this it is possible to loop through the coastline image and check the color of every pixel.

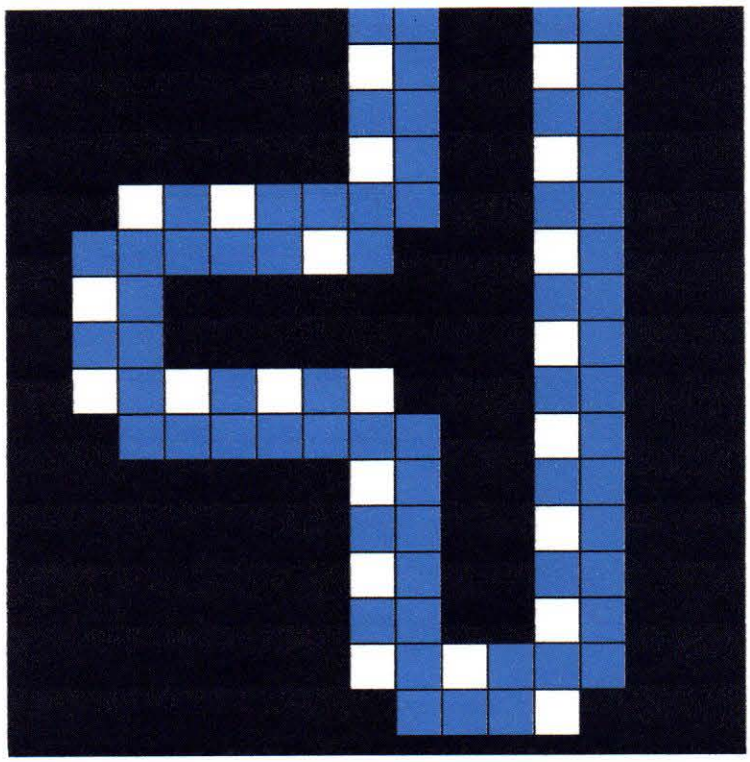
The RGB scale goes from 0 to 255. The color black has an RGB value of 0, 0, 0 while the color white will hold very high values of red, green, and blue. In order for a pixel to be white and therefore an edge the value of red, green, and blue must be high such as 220. While looping through each pixel the program looks for pixels that have a red value of at least 220. These are marked with '1' in the array representation of the image which will mark this pixel as part of the edge. All pixels that don't have a red value greater than 220 are marked with a '0' to show that they are not an edge.

This essentially turns the image being used into an array that stores whether or not each pixel is part of the coastline. However, so far there are still several pixels making up each part of the edge. One way to prevent multiple pixels being white at a particular part of the edge is by checking to see the color of neighboring pixels. One method of implementing this is to check to see if the color of the pixel to the left or above the current pixel is black. If either of those pixels is black, then a white pixel is needed to keep the edge connected. However, if they are both white then adding another white pixel would be redundant since the edge in this area has already been defined by surrounding pixels which causes there to be wasted space on information in the images generated. Removing these redundant pixel thus makes the image processing more efficient.

This code loops through this image pixel by pixel starting at the top left corner and working its way down this first column of pixels. It then goes column to column repeating this process. This means that when a pixel is being checked, the four numbered pixels shown in the image below have already been scanned.



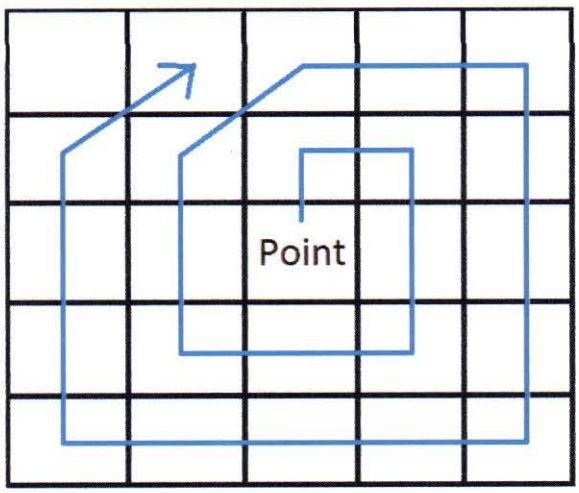
The pixel limitation is handled by checking each of these four pixels to make sure that none of them have already been stored as containing part of the coastline. If any of them are found to have already been given a non-zero color value, the pixel being currently scanned is ignored even if it does contain a piece of the coastline. The below image shows the result after performing the pixel reduction algorithm.



The image after performing pixel reduction displayed over the original coastline

Connecting Generated Points

The pixel limitation algorithm allowed images to be generated with a smaller number of points while still getting an accurate depiction of the coastline. The next step is to connect these points to one another in a sequential order that follows the path of the coastline. In order to get the best approximation as to order of the points the program picks a point containing part of the coastline and then searches for the closest pixel. This is handled by starting at the pixel above this initial point and looping clockwise around until another pixel from the coastline edge is found.



The path followed when searching for a points nearest neighbor

This algorithm allowed for points to be connected to one another but wasn't completely functional due to a couple issues. The first problem that came up was having multiple points directing themselves to a common point. In order to create a path that follows the coastline and chains all of the points together we need to ensure that each point only has two other points connected to it. One of these points is incoming (the previous pixel in the coastline) and the other one is outgoing (the next pixel in the coastline). The fix for this involved creating a new

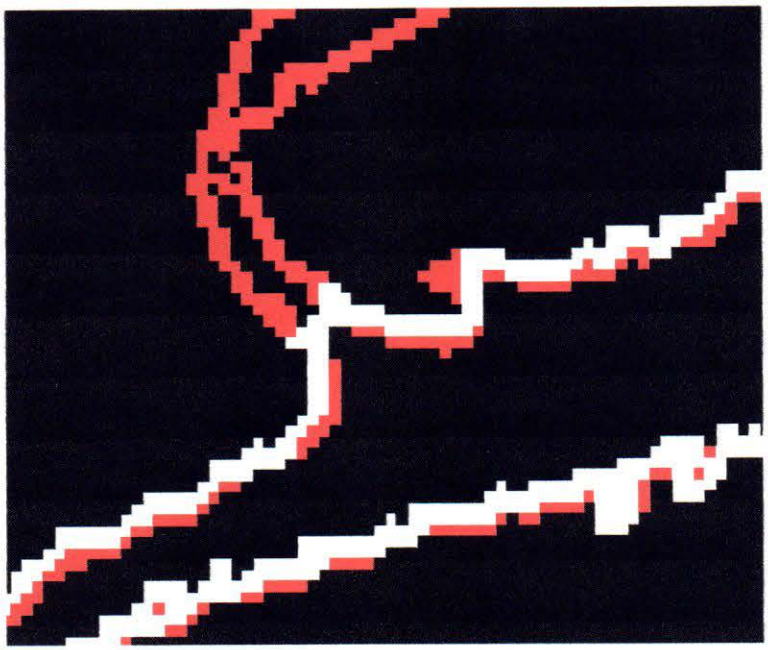
composite data type, called a class, that stores all of the information for each pixel in the image. This class, Pix, stores things such as whether or not the pixel is populated (contains a part of the coastline), whether or not there is a connection to another pixel, and the corresponding x and y coordinate for this connection. Upon starting the program, a two-dimensional array with a size corresponding to the width and height of the image being processed is initialized. During this the boolean values for each pixel being populated and for each pixel having an outgoing connection are set to false. The algorithm that connects each point now runs through this Pix class array. When a connection is to be made the boolean value for having an outgoing connection is set to true and the corresponding x and y coordinates of the point being connected to are stored. With this information stored it is possible to check to ensure that a point hasn't already been connected to when performing the nearest neighbor algorithm so that no point has multiple lines being drawn to it. One small problem that came up after implementing this was that on points such as the very last pixel at the end of the coastline there is no next point to connect to. This means that the algorithm to find the nearest point will never find a valid point to connect with. In order to prevent this from happening a count was implemented that ensures that the nearest neighbor algorithm is only ran up to a certain amount of iterations in order to prevent improper connections from being made.

Another problem that arose with the algorithm was that since it just cycled through every pixel in the image from column to column that there was no specific order in which pixels were scanned. Because of this, there is the chance that points will be hit out of order causing them to go in different directions. Since we want to get the points in order up the coastline it is important to try and get these points in sequential order. The solution for this problem involves picking an initial point through the original column sweep method but using the result from the nearest

neighbor algorithm as the next point to be analyzed instead of repeating the sweep. This causes the pixel connection to be generated by going from point to point in the image causing there to be a defined path from the beginning to the end of the coastline.

With this point to point method we are able to traverse the coastline from the first point from the left until it reaches the end of the image. This requires the initial point to be the starting pixel in the coastline edge. If this is not the case, then the path generated from this point will only cover one direction. This means that the entire other side would be lost. One way to prevent this is to have the connection algorithm perform additional sweeps for points that haven't yet been connected. The way this was implemented was having a modifiable value that determines how many times a sweep will be performed with each sweep alternating from coming from the left or right. This ensures that all points will be hit and in the event of an extreme case this threshold could be increased to include each point.

Even after putting in each of these fixes there is still the potential for other issues to pop up when running the point connection algorithm. In the case of a jagged enough coastline it is possible for there to be points that are across from the current pixel being analyzed that end up being closer than the pixel that is actually next in the coastline. This problem can cause certain parts of the coastline to be skipped and left out.

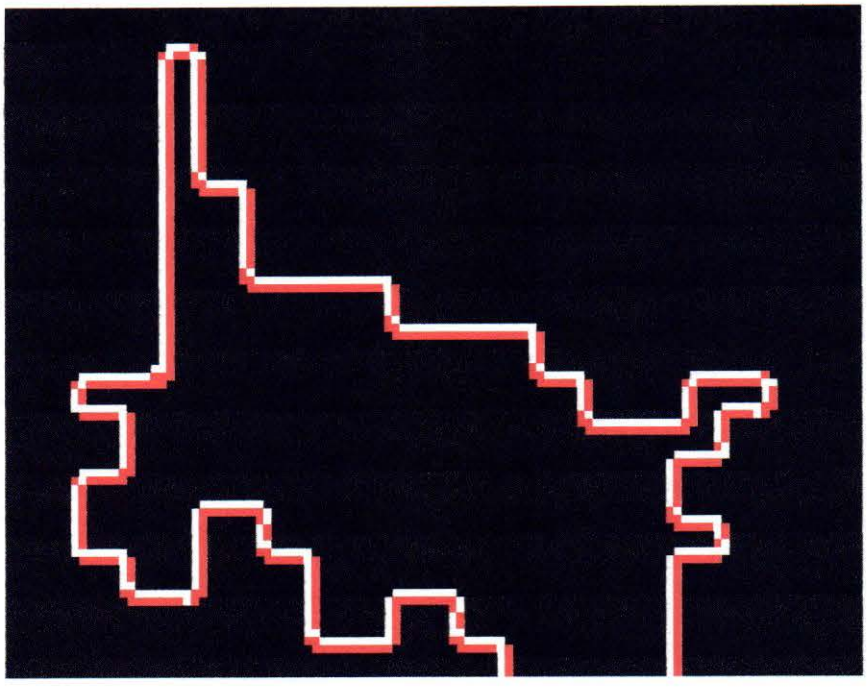


Example of a portion of the coastline being skipped

However, these left out areas will be picked up by additional sweeps and will be connected. The problem with this is that the coastline will not be able to be written as a single sequence of points and will instead be split into separate lists. One way to try and combat this is to be careful with how much point reduction is performed on the image or to link several lists together by connecting their endpoints. Another way is to link second visits together by connecting the endpoints of each list of points. I discussed the method for making sure that there is at least a pixel between each point but it is possible to expand this to create a larger gap between each pixel. This increases the likelihood of the issue occurring since there will be an increased gap between subsequent points in the coastline.

The desired output after performing the point reduction and then connecting the points is to have an image that still accurately represents the original coastline while taking up less space and therefore being easier to work with. To check how accurate of a representation is generated I output the original image and then overlaid lines connecting each of the points generated to

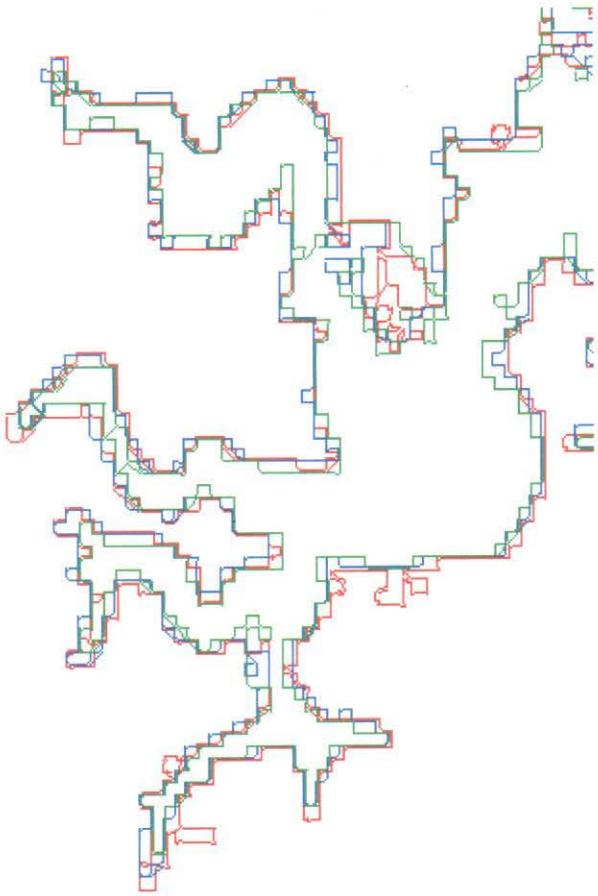
visualize what the list of points looked like. This enabled me to be sure that each of the algorithms was functioning properly and let me refine different things to debug and try and make each process as accurate as possible.



Lines drawn from points after program is run (white) over original edge (red)

Year to Year Change Detection

One of the images being used for testing the algorithms depicts an infrared portion of Wicomico County. Caitlyn Curry was able to provide me with three images of this region from the years 1996, 2006, and 2016. The most straightforward way to determine the change in the coastline between these years is to create and display the connections between the lists of points generated from each image. Overlaying each year over one another is an easy way to visually track the change over time.



Close up of coastlines from 1996 (red), 2006 (blue), and 2016 (green)

There doesn't seem to be that much of a change noticeable between these three coastlines. This is most likely due to each image being only 10 years after the previous one. Something else that is quickly noticed when looking at these close up images of the coastlines is how blocky they are. The original infrared images are large and don't have a very high resolution. This causes pixilation when they are zoomed in on. During the edge-detection and other image processing techniques each image is broken up into pixels and looks only for pixels that are bright enough to be a clear piece of the coastline. Because of this, any blending that would show the gradual shift from land to water is removed in order to preserve only the coastline. This causes the output to be blocky and pixelated due to the quality of the original images and the algorithms removing any pixels that were previously smoothing edges. It is important to note that we have not yet quantified the edge changes. Although we do not notice much change visually, once the curve changes are quantified we may be able to detect trends for the past two decades.

Fractal Dimension Analysis

One way to compare curves to one another is by looking at their fractal dimension. The fractal dimension of a curve, or set in general, is essentially how complex of a pattern the lines that make it up form. Another way at looking at this is that the more jagged the edges of an image are, the higher the dimension will be. For example, a straight line or similar curve will have a dimension of one and a plane has a dimension of two. A curve that has a dimension between one and two is showing the amount of line jaggedness to begin filling a planar region. A box-counting dimension of 1.2 shows a substantial amount of jaggedness where as a 1.5 or 1.7 shows an extreme amount.

One method of approximating the fractal dimension of a shape is by using the box-counting dimension. This is done by picking two integers n and m such that $n < m$. first splitting up the image by an n by n grid which gives n^2 cells. The next step is to count the number of cells that contain a piece of the image. The image is then split into an m by m grid and the number of cells containing the image are again counted. The fractal dimension is then estimated by:

$$\frac{\log\left(\frac{m\ \ count}{n\ \ count}\right)}{\log\left(\frac{m\ \ size}{n\ \ size}\right)}$$

where "m count" and "n count" represent the number of cells that contain a piece of the image for their respective grid sizes (Peitgen).

During the point reduction algorithm each pixel is checked for whether or not it contains part of the coastline. This means that there is already a value that can be checked in order to see if a cell contains a point. The n and m values from above are declared within the code. The image being processed is split into grids of these sizes and each of these created cells is scanned

to check if a point is present inside of it. This process is repeated until all of the cells have been searched and then the above equation is calculated to give an estimation of the fractal dimension.

Performing this box-counting algorithm can allow us to compare the fractal dimension of a coastline to look for any change over time. The following results were obtained by using an n value of 120 and an m value of 240 on the 1996, 2006, and 2016 images of Wicomico County:

```
1996: Box-Counting dimension: 1.2115332761831860
2006: Box-Counting dimension: 1.2237553832147587
2016: Box-Counting dimension: 1.2461991965863253
```

The change in these values from year to year does not necessarily mean that there is a change in the coastline. The Box-Counting method serves as a way to estimate the fractal dimension of a shape. One image's result being higher indicates that this image contains more jagged edges that the others. With enough accurate data for the coastline of a region it could be possible to detect changes by looking at this fractal dimension value. It could also be possible compare coastlines from different regions to one another based on how complex the lines that make them up are by using this method. A fractal dimension for the coastline of Great Britain including Ireland has been found to be approximately 1.31 (Peitgen). Looking at this value compared to the ones found from the Wicomico County coastline it can be seen that Great Britain's coast possesses more jagged edges.

Results Discussion and Future Work

Although statistical results were not obtained during this research period, several accomplishments were made that will allow for more work to become completed in the future. The method used to obtain an edge from coastlines by processing an image in GIMP allows for a cleaner, less noisy image than other methods that were tested. This has potential for applications outside of coastline analysis where edge-detection is required. The pixel-reduction algorithm used in this code can easily be expanded on and adjusted to work in different cases. A few adjustments can make the minimum gap required between points be any specified length. Another success was getting a functional method for connecting points in order to automatically recreate the coastline after the reduction had been performed. This incorporated detecting and storing each pixel's nearest neighbor as well as the ability to sweep over an image to ensure that every point has been hit. The last major accomplishment was building the Box-Counting dimension approximation using information obtained using other algorithms in the code. Although it currently exists inside of the coastline processing code, this can be used on any image to obtain a fractal dimension approximation.

There is still room for improvement and advances in the work that has been completed thus far. One example of this is finding a way to automate the GIMP edge-detection. Even though performing this manually doesn't take much time or effort having everything streamlined and automated would make using this code more user friendly. This will be no easy feat since this process required adjusting thresholds and other steps that can change based on the images being used. The process of connecting each of the points of the coastline after the reduction has been performed ensures that each point is hit by performing sweeps from the left and right based

on how many iterations are requested in the code. This works but can encounter problems when there are points that are skipped over by the algorithm. These points exist because they are correctly skipped over when connecting the coastline points together as they are not the nearest points to any of the other points in the sequence. However, since they still exist and have yet to be connected to another point they will be hit by sweeps when searching for more pieces of the coastline. This doesn't cause any problems since they have no points close enough to connect to and will just be removed and skipped over after being hit by a sweep. This does cause longer processing times since this causes sweeps that don't actually connect points.

There is still a lot of room for expansion on what has been done so far. Now that lists containing the points that make up a coastline have been generated, it is possible to use these to track the location of each point in the coastline over time. This allows the potential for vector equations to be created that track the movement over the years. With enough data this can be used to predict the change in a coastline over time. This leads into the next step for expanding on this research which is obtaining more data. This will give the chance to test each of the algorithm with different images to ensure that accurate results are obtained. This will also enable more drastic changes in coastline locations to be seen if imagery data taken from a large time span is used. Finally, there is much room for optimization within the code. Ensuring that everything is functioning properly as well as easily readable to others viewing the code is important in order for advancements in the techniques and methods to be made.

Source Code

```
package test;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.imageio.ImageIO;
import java.lang.Math;

class Pix {
	private boolean pop, to, from;
	private int toX, toY, fromX, fromY;

	Pix() {
		pop = false;
		to = false;
		from = false;
		toX = -1;
		toY = -1;
		fromX = -1;
		fromY = -1;
	}

	Pix(int x, int y) {
		pop = false;
		to = false;
		from = false;
		toX = -1;
		toY = -1;
		fromX = x;
		fromY = y;
	}

	public void setTo(int x, int y) {
		toX = x;
		toY = y;
		to = true;
	}

	public void setFrom(int x, int y) {
		fromX = x;
		fromY = y;
		from = true;
	}

	public void setPop() {
		pop = true;
	}

	public void removePop() {
		pop = false;
	}

	public boolean getPop() {
		return pop;
	}
```

```
	public boolean getTo() {
		return to;
	}

	public boolean getFrom() {
		return from;
	}

	public int getToX() {
		return toX;
	}

	public int getToY() {
		return toY;
	}

	public int getFromX() {
		return fromX;
	}

	public int getFromY() {
		return fromY;
	}
}

public class Pixel {
	BufferedImage image;
	BufferedImage newimage;
	BufferedImage vectorImage;
	int width;
	int height;
	Pix[][] v;

	public Pixel(String iFile, String oFile) {
		try {
			File input = new File(iFile);
			image = ImageIO.read(input);
			width = image.getWidth();
			height = image.getHeight();
			newimage = new BufferedImage(width, height,
				BufferedImage.TYPE_INT_RGB);
			vectorImage = new BufferedImage(width, height,
				BufferedImage.TYPE_INT_RGB);
			int[][] arr = new int[height][width];
			Pix[][] v = new Pix[height][width];
			for (int i = 0; i < height; i++) {
				for (int j = 0; j < width; j++) {
					v[i][j] = new Pix();
					Color c = new Color(image.getRGB(j, i));
					if (c.getRed() > 220) {
						arr[i][j] = 1;
						newimage.setRGB(j, i, Color.BLUE.getRGB());
						if (i > 0 && j > 0 && width - j > 0) {
					Color lastc = new Color(newimage.getRGB(j, i - 1));
```

```
            Color last2 = new Color(newimage.getRGB(j - 1, i));
            Color last3 = new Color(newimage.getRGB(j - 1, i - 1));
            Color last4 = new Color(newimage.getRGB(j + 1, i - 1));
                  if (lastc.getRed() < 20 && last2.getRed() < 20 &&
                        last3.getRed() < 20 && last4.getRed() < 20) {
                        newimage.setRGB(j, i, Color.WHITE.getRGB());
                              v[i][j].setPop();
                        }
                  } else if (i > 0 && j > 0) {
                  Color lastc = new Color(newimage.getRGB(j, i - 1));
                  Color last2 = new Color(newimage.getRGB(j - 1, i));
                  if (lastc.getRed() < 20 && last2.getRed() < 20) {
                        newimage.setRGB(j, i, Color.WHITE.getRGB());
                        v[i][j].setPop();
                        }
                  } else if (i > 0) {
                  Color lastc = new Color(newimage.getRGB(j, i - 1));
                        if (lastc.getRed() < 20) {
                        newimage.setRGB(j, i, Color.WHITE.getRGB());
                              v[i][j].setPop();
                              }
                  } else if (j > 0) {
                  Color lastc = new Color(newimage.getRGB(j - 1, i));
                        if (lastc.getRed() < 20) {
                        newimage.setRGB(j, i, Color.WHITE.getRGB());
                        v[i][j].setPop();
                              }
                        }
                  }
            }
}
File output = new File("outlineOutput.png");
ImageIO.write(newimage, "png", output);
boolean success = false;
int count = 1;
int checkx = -1, checky = -1;
int nexti = 0, nextj = 0;
for (int sweep = 0; sweep < 500; sweep++) {
      if (sweep % 2 == 0) {
            first: for (int j = 0; j < width; j++) {
                  for (int i = 0; i < height; i++) {
                  if (v[i][j].getPop() && !v[i][j].getTo()) {
                              nexti = i;
                              nextj = j;
                              break first;
                        }
                  }
            }
      } else {
            second: for (int j = width - 1; j >= 0; j--) {
                  for (int i = 0; i < height; i++) {
                  if (v[i][j].getPop() && !v[i][j].getTo()) {
                              nexti = i;
                              nextj = j;
                              break second;
```

```
                        }
                    }
                }
            }
            while (v[nexti][nextj].getPop() &&
                    !v[nexti][nextj].getTo()) {
                success = false;
                count = 1;
                while (!success && count < 8) {
                    checkx = -count;
                    checky = 0;
                    while (checky < count) // top right
                    {
                if (nexti + checkx >= 0 && nexti + checkx < height
                 && nextj + checky >= 0 && nextj + checky < width
                 && v[nexti + checkx][nextj + checky].getPop()
                 && !v[nexti + checkx][nextj + checky].getTo()) {
                                success = true;
                                break;
                            }
                            checky++;
                        }
                        while (checkx < count) // right
                        {
                if (nexti + checkx >= 0 && nexti + checkx < height
                 && nextj + checky >= 0 && nextj + checky < width
                 && v[nexti + checkx][nextj + checky].getPop()
                 && !v[nexti + checkx][nextj + checky].getTo()) {
                                success = true;
                                break;
                            }
                            checkx++;
                        }
                        while (checky > -count) // bottom
                        {
                if (nexti + checkx >= 0 && nexti + checkx < height
                 && nextj + checky >= 0 && nextj + checky < width
                 && v[nexti + checkx][nextj + checky].getPop()
                 && !v[nexti + checkx][nextj + checky].getTo()) {
                                success = true;
                                break;
                            }
                            checky--;
                        }
                        while (checkx > -count) // left
                        {
                if (nexti + checkx >= 0 && nexti + checkx < height
                 && nextj + checky >= 0 && nextj + checky < width
                 && v[nexti + checkx][nextj + checky].getPop()
                 && !v[nexti + checkx][nextj + checky].getTo()) {
                                success = true;
                                break;
                            }
                            checkx--;
                        }
```

```
                        while (checky < 0) {
                    if (nexti + checkx >= 0 && nexti + checkx < height
                     && nextj + checky >= 0 && nextj + checky < width
                     && v[nexti + checkx][nextj + checky].getPop()
                     && !v[nexti + checkx][nextj + checky].getTo()) {
                                    success = true;
                                    break;
                                }
                                checky++;
                            }
                            count++;
                        }
                        if (count >= 8) {
                            v[nexti][nextj].removePop();
                        }
                        if (success) {
                    v[nexti][nextj].setTo(nexti + checkx, nextj + checky);
                            nexti = nexti + checkx;
                            nextj = nextj + checky;
                        }
                    }
                }
                Graphics2D g2d = vectorImage.createGraphics();
                g2d.setBackground(Color.BLACK);
                BasicStroke bs = new BasicStroke(1);
                g2d.setStroke(bs);
                for (int i = 0; i < height; i++) {
                    for (int j = 0; j < width; j++) {
                        if (v[i][j].getPop() && v[i][j].getToX() > -1) {
                    g2d.drawLine(j, i, v[i][j].getToY(), v[i][j].getToX());
                        }
                    }
                }
                File vectorOutput = new File(oFile);
                ImageIO.write(vectorImage, "png", vectorOutput);
                int split1 = 120;
                int split2 = 240;
                int x1 = width / split1;
                int y1 = height / split1;
                int x2 = width / split2;
                int y2 = height / split2;
                double count1 = 0;
                double count2 = 0;
                for (int i = 0; i < split1; i++) {
                    for (int j = 0; j < split1; j++) {
                        if (i == split1 && j == split1) {
                ijLabel: for (int y = height - 1; y >= y1 * (split1 - 1); y--) {
                    for (int x = width - 1; x >= x1 * (split1 - 1); x--) {
                                    if (arr[y][x] == 1) {
                                        count1++;
                                        break ijLabel;
                                    }
                                }
                            }
                        } else if (i == split1) {
```

```
iLabel: for (int y = height - 1; y >= y1 * (split1 - 1); y--) {
            for (int x = (j * x1); x < ((j + 1) * x1); x++) {
                                    if (arr[y][x] == 1) {
                                          count1++;
                                          break iLabel;
                                    }
                              }
                        }
            } else if (j == split1) {
jLabel: for (int y = (i * y1); y < ((i + 1) * y1); y++) {
      for (int x = width - 1; x >= x1 * (split1 - 1); x--) {
                                    if (arr[y][x] == 1) {
                                          count1++;
                                          break jLabel;
                                    }
                              }
                        }
            } else {
normLabel: for (int y = (i * y1); y < ((i + 1) * y1); y++) {
      for (int x = (j * x1); x < ((j + 1) * x1); x++) {
                                    if (arr[y][x] == 1) {
                                          count1++;
                                          break normLabel;
                                    }
                              }
                        }
            }
      }
}
for (int i = 0; i < split2; i++) {
      for (int j = 0; j < split2; j++) {
            if (i == split2 && j == split2) {
ijLabel: for (int y = height - 1; y >= y2 * (split2 - 1); y--) {
      for (int x = width - 1; x >= x2 * (split2 - 1); x--) {
                                    if (arr[y][x] == 1) {
                                          count2++;
                                          break ijLabel;
                                    }
                              }
                        }
            } else if (i == split2) {
iLabel: for (int y = height - 1; y >= y2 * (split2 - 1); y--) {
      for (int x = (j * x2); x < ((j + 1) * x2); x++) {
                                    if (arr[y][x] == 1) {
                                          count2++;
                                          break iLabel;
                                    }
                              }
                        }
            } else if ((j + 1) == split2) {
jLabel: for (int y = (i * y2); y < ((i + 1) * y2); y++) {
      for (int x = width - 1; x >= x1 * (split2 - 1); x--) {
                                    if (arr[y][x] == 1) {
                                          count2++;
                                          break jLabel;
```

```
                                        }
                                    }
                                }
                            } else {
                    normLabel: for (int y = (i * y2); y < ((i + 1) * y2); y++) {
                                for (int x = (j * x2); x < ((j + 1) * x2); x++) {
                                    if (arr[y][x] == 1) {
                                        count2++;
                                        break normLabel;
                                    }
                                }
                            }
                        }
                    }
                    System.out.println("Box counting dimension: " + (Math.log((count2
                            / count1)) / Math.log(split2 / split1)));
            } catch (Exception e) {
                    System.out.println(e);
            }
    }

    public static void main(String[] args) {
            Pixel img1996 = new Pixel("new.png", "1996out.png");
            Pixel img2006 = new Pixel("new.png", "2006out.png");
            Pixel img2016 = new Pixel("new.png", "2016out.png");
    }
}
```

Sources and Software Used

Clark Labs. *IDRISI*. Computer software. Vers. 17.02. N.p., Mar. 2013. Web

Itseez. *Open Source Computer Vision Library (OpenCV)*. Computer software. Vers. 2.4.13. N.p., 19 May 2016. Web.

Peitgen, Heinz-Otto, H. Jùrgens, and Dietmar Saupe. *Chaos and Fractals: New Frontiers of Science*. New York: Springer-Verlag, 1992. Print.

Python Software Foundation. *Python*. Computer software. Vers. 2.7.12. N.p., 25 June 2016. Web.

The GIMP Team. *GNU Image Manipulation Program (GIMP)*. Computer software. Vers. 2.8.18. N.p., 14 July 2016. Web.