APPM 5380 Final Project

Modeling Rivers

Daniel Long
Sean White
Matthew Gallagher
Estelle Lindrooth
Alden Soto

APPM 5380
University of Colorado Boulder
December 2023

Contents

1 Background

1.1 River Movement

We consider changes in river shape at large time scales, and specifically, how rivers meander over time due to erosion of surrounding topography. This is a complicated phenomenon with many potential dependencies: volumetric flow (including sporadic change of flow from flash flooding), soil and sediment composition, river bottom profile, etc. The meandering characteristics of rivers are likely most significantly impacted by erosion of sediments on the outer bank toward the inner bank, driven by helical water flow created as the bulk flows around a corner. The shift of material changes the shape of the river bottom and banks of the river, altering both the velocity profile of the river and the banks of the river, producing the meandering feature of many river, and allowing for further erosion.

1.2 Oxbow Formation

A consequence of meandering is the creation of oxbow lakes: over time, meanders grow and elongate, expanding to create large sequential loops until it is possible for two nonadjacent points of the river to meet, creating a path of less resistance for water to flow down. As a result, the section of the river between these two points gets cut off, creating a standing body of water.

Oxbow creation changes the river profile drastically, which has significant consequences to the overall topography of the river, as well as the velocity profile of the water traveling through the river.

1.3 Questions

Consequently, our project and results are driven by two main guiding questions that we seek to at least partially answer:

- What is the velocity of the banks based off of the change of direction of the river at a given point? It is assumed the river will always expand in an outward normal direction.
- How can you model the formation of oxbow lakes, and can you detect when the river path is no longer in line with the oxbow lake?

2 Physics Motivated Approach

In an attempt to model the river we consider the physics of the water near the bank. We first consider the conservation of momentum of the water to help us solve for what we are truly interested in, the downstream velocity of the water around the curve.

2.1 Flow around a Curve

First we will define our coordinate system. n will denote the direction of the radius of curvature. For a point on the outer bank of the river. The positive direction will be in the direction outwards of the river, and thus the negative direction would be pointing into

the river. z will denote the river water level. $z = 0$ will denote the lowest point of the water level. For a curve this will be in the inside bank, as we will explore in an upcoming section. For a straight part of the river, all points on the river will have $z = 0$. s will denote the downstream direction. The positive direction will be in the direction of the water flow.

We will stand on the shoulders of giants to get us to a starting point. Previous study of the flow around bends shows that the equations of motion in the downstream direction, s and the outward normal direction n can be simplified [Eng74] as:

$$0 = -g\frac{dz_s}{ds} + \frac{d}{dz}(v_t\frac{dv_n}{dz}) \tag{1}$$

$$\frac{-v_s^2}{r} = -g\frac{dz_s}{dn} + \frac{d}{dz}(v_t\frac{dv_n}{dz}) \tag{2}$$

Where z is the elevation of the water above the s, n plane, g is gravity, and v_t is the eddy viscosity.

Lets consider some more simplifications for our model. Consider $\frac{dv_n}{dz}$, this would be the acceleration of the water in the radial direction. We argue that this is relatively small in comparison to the velocity near the edge of the bank and thus can be neglected for the purposes of our model. Our justification is that there exists an outer bank cell of helical motion that breaks up the larger helical flow of the water. This smaller flow is much smaller in magnitude than the overall velocity of the water [GB02].

We can then further simplify the equations for the water motion to:

$$0 = -g\frac{dz_s}{ds} \tag{3}$$

$$\frac{-v_s^2}{r} = -g\frac{dz_s}{dn} \tag{4}$$

Due to our to our simplification we end up with an assumption about the flow of the water. Looking at our first equation we can see that we are now left with:

$$\frac{dz_s}{ds} = 0 \tag{5}$$

That is that the height of the water does not change in the downstream direction.

Looking at our second equation, solving for v_s we get that:

$$v_s = (gr\frac{dz_s}{dn})^{1/2} \tag{6}$$

This is where we are going to introduce an assumption. The velocity of the water appears to grow as the distance from the center of curvature grows as we can see by the r term. We are then going to make an assumption that the velocity of the water

grows within the river at the same rate given the same radius of curvature. Thus we will transform our equation to.

$$v_s = (gn\frac{dz_s}{dn})^{1/2} \tag{7}$$

This is not quite yet useful for us, so we will explore $\frac{dz_s}{dn}$ a bit further.

2.2 Super Elevation of the Water

Due to the forces on the water we expect that the water piles up at the banks and creates deeper water at the outward bank and shallower water at the inside bank. In reality we do not see rivers with all of the water on one side suggesting there is a steady state that balances the forces applied due to the radial force on the water and the force caused by the difference in height at each bank.

We then need to figure out the inward radial force due to gravity and set it to the outward radial force in order to find our change in height.

Consider the average width of the river, W_a and the difference in height Δh. The force on the water due to gravity in the direction of the radial direction is is:

$$F_g r = \frac{F_g d}{\tan(\theta)} \tag{8}$$

Where $F_g r$ is the force due to gravity in the radial direction, $F_g d$ is the force of gravity in the downward direction, and θ is the angle formed by the water at the lowest point. We know that the downward force of gravity is $F_d = m_w g$, and $m_w = \rho_w V_w$. We also know that $\tan(\theta) = \frac{\Delta h + d}{W}$. Substituting these in we end up with a balancing force with the outward radial force. Equating them together we can start to solve for Δh. Note that when there is no radial force, we do not see a change in height, therefore we can determine that d is very small compared to the width of the river, and thus can be neglected.

$$\frac{p_w V_w v_a^2}{r_c} = \frac{p_w V_w g \Delta h + d}{W} \tag{9}$$

Where v_a is the average velocity of the water, d is the depth of the water, and g is gravitational acceleration. Note that due to lack of data, we are using the average velocity of the river. In reality we would want to use the average velocity of the river at that point which would depend on r_c. Unfortunately this assumption is necessary, as we are unable to measure the average velocity of the water at any point in the river due to lack of funding and time. We recognize that this will introduce error into our model, however, for the purposes of this class we feel that this is an appropriate assumption to make. Solving for Δh we get that:

$$\Delta h = \frac{v_a^2 W}{r_c g} \tag{10}$$

This is the change in height at the outer bank. If we consider the lowest point to be at 0 then the rate of change with respect to the radial direction would come out to be:

$$\frac{dz_s}{dn} = \frac{v_a^2 W}{r_c g W} \tag{11}$$

Inserting this back into our equation for the downstream velocity of water we get that:

$$v_s = (\frac{n v_a^2}{r_c})^{1/2} \tag{12}$$

2.3 Model

At the start, we proposed that the movement of the bank is proportional to velocity of water at the outer bank.

We then end up with:

$$d_b = k(\frac{n_b v_a^2}{r_c})^{1/2} \tag{13}$$

Where r_b is the distance to the outer bank. We will substitute this with W as the distance from the inner bank to the outer bank is equivalent to the width of the river.

Thus we end up with our final model as:

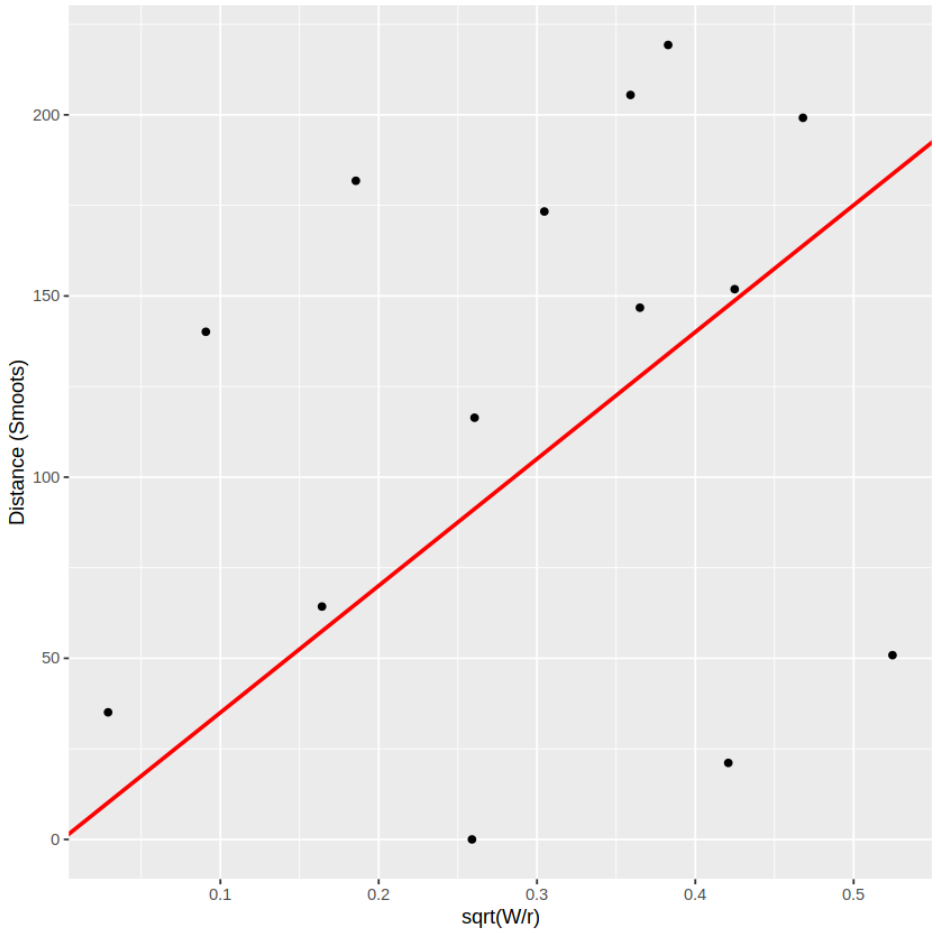
$$d_b = k(\frac{W v_a^2}{r_c})^{1/2} \tag{14}$$

$$B = k v_a \tag{15}$$

We then end up with:

$$d_b = B(\frac{W}{r_c})^{1/2} \tag{16}$$

Where B can be determined by linear regression. When running linear regression on our data we found that $B = 350.15$



3 Approaches

3.1 Approach 1

3.1.1 River and Data Collection

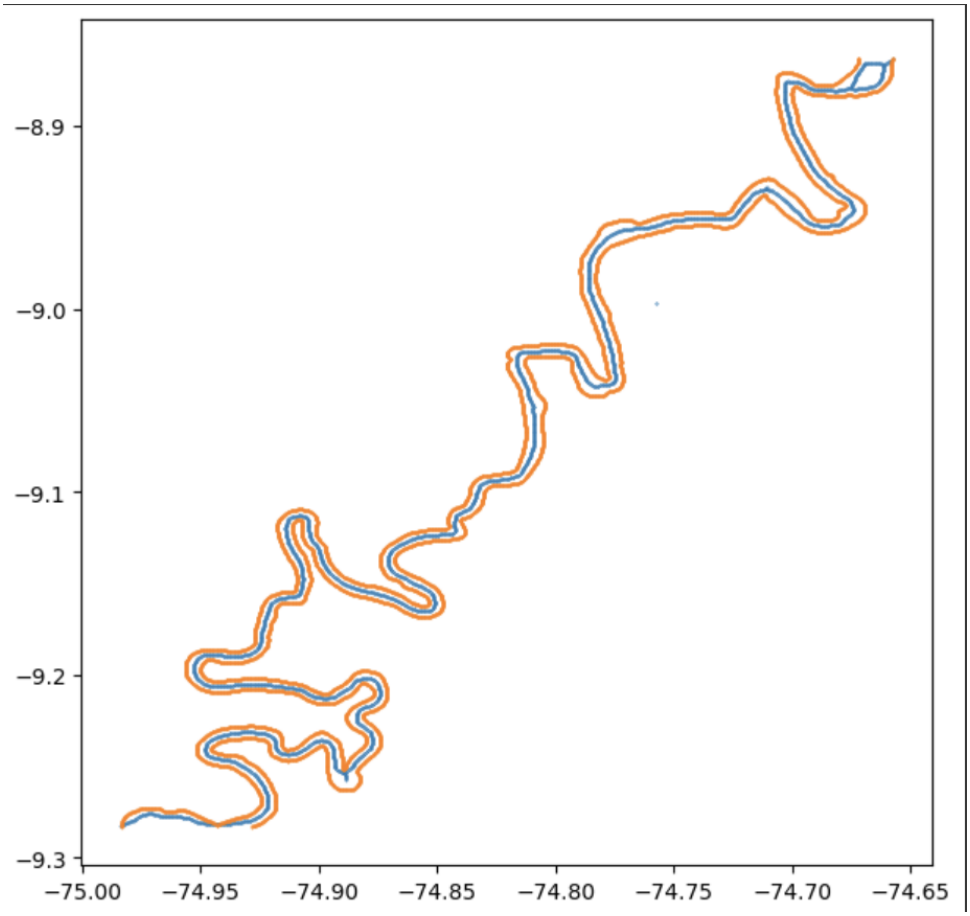
I began by gathering data on a section of the Ucayali River from years 1984 to 2023. I began in Google Earth Engine and used the composite algorithm for Landsat data to create a composite map of the study area that was relatively cloud free. From years 1984 to 2012 I used data from Landsat 5 tier 1 collection. From years 2013 I used data from Landsat 8 tier 1 collection. I then created a water mask using the Normalized Water Differencing Index. This is an equation that differences the green and near infrared (NIR) bands given by the following formula:

$$\frac{Green - NIR}{Green + NIR}$$

I then masked all the pixels that were greater than 0 as water; all pixels with an index value less than 0 were categorized as not water. Then I exported each image as a binary tif file with values 1 representing areas with water and values 0 representing areas without water. These were used for further processing in python.

In python, I used the mahotas package to blur the image using a Gaussian kernel, then label parts of the mask. Some of the mask contained some extra pixels that were not a part of the part of the river I was interested in, so the labelling was used to extract the

main parts of the river. Then I used the bwperim function to classify the perimeter of the water and used the skeleton function to find the center line. I realized that the center line was not completely accurate using this result, but it did produce a good approximations shown below:



I also tried this approach on other portions of the river. This process was tedious and did not provide the results I wanted. Some of the images had breaks in the mask that categorized the river as two different sections. Also, there were many extra parts of the river that ended up messing up my results like smaller river cut offs that were not apart of the main river. I was able to cut out most of these smaller portions, but there were still parts that did not represent the bank lines I wanted. For example, the image below represents the final product of the perimeter of the portion of the Ucayali river I was interested in. I was able to cut out a lot of the extra channels and meanderings of the river, but there is still not a smooth line where the parts were supposed to be cut out and replaced by a smoothed line representing the river banks.

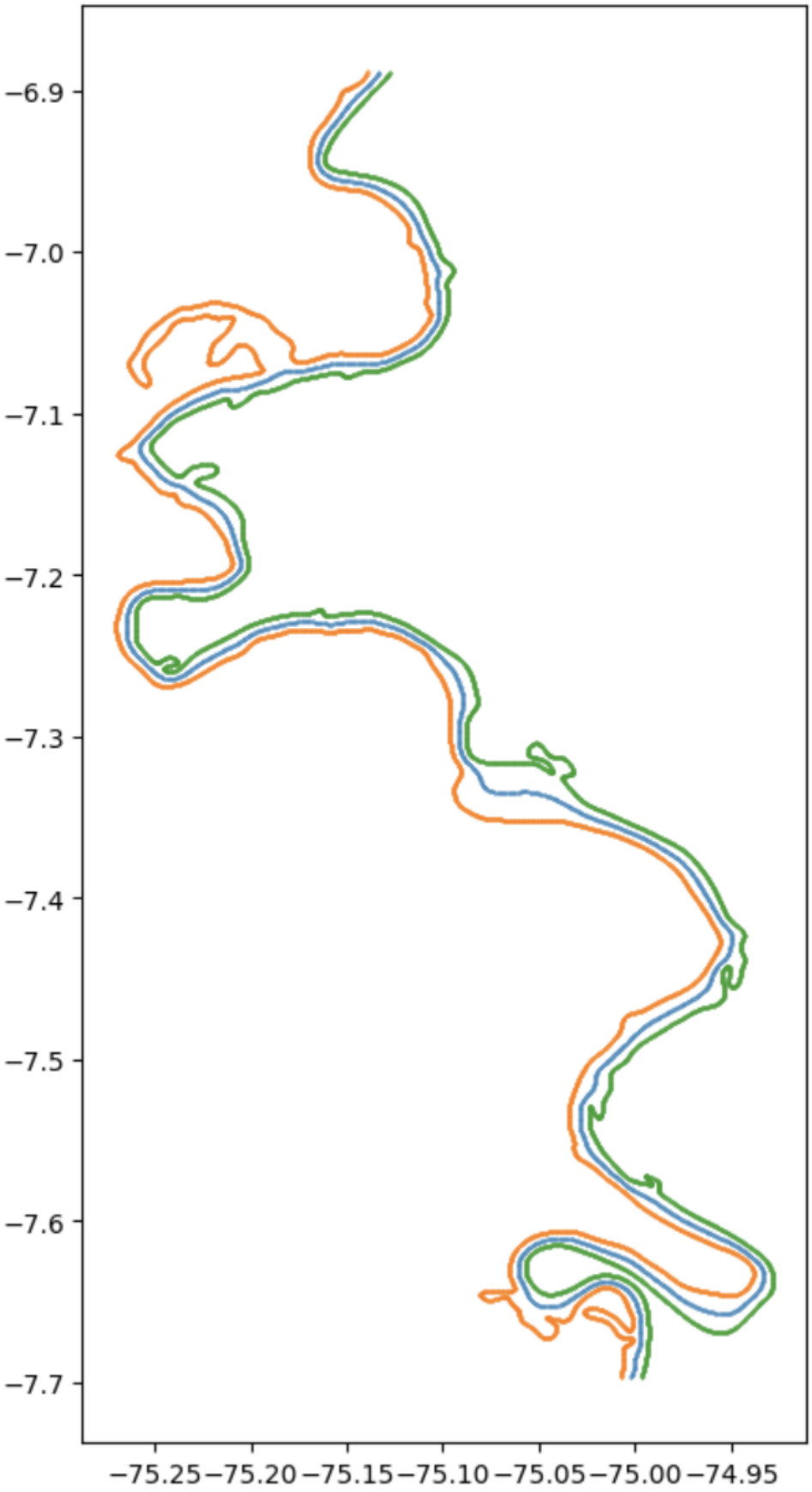


Figure 1: Portion of river perimeter with center line.

Since this did not work out and I was unable to use this data for accurate results, I used the bank line data provided by the RivMap package [Jon23] in Matlab. I exported the results from using their center line and bank line algorithms to be used in python.

3.1.2 Data Processing

Since the data was coded in cell number rather than in coordinates, I had to change back the cell numbers to coordinates. I imported a raster into python that had georeferenced cells. Then each coordinate was extracted based on the cell number in the data. Then, I smoothed the data using the Savitzky–Golay filter to smooth the bank lines. This filter takes a window of points, then fits a polynomial to those points by linear least squares. For this data, I used a window of 100 and a polynomial with degree 4. I then made each point the same distance away. I calculated the total distance of each bank line, then found 1000 equally spaced points on that line.

Next I calculated curvature based on the central differencing technique used in the RivMat algorithm. This calculated the second derivative of the angle between each x and y point with respect to distance between each point. Each point was equally spaced, so the curvature of each point was found with respect to the same distance. In the last project, we found that curvature varied more if points were further apart of closer together rather than the actual curvature of the river, so this approach seemed like it would better capture the local curvature of each point.

I found the width of the river at each point by finding the closest point on the other side of the bank and taking the distance between the two points.

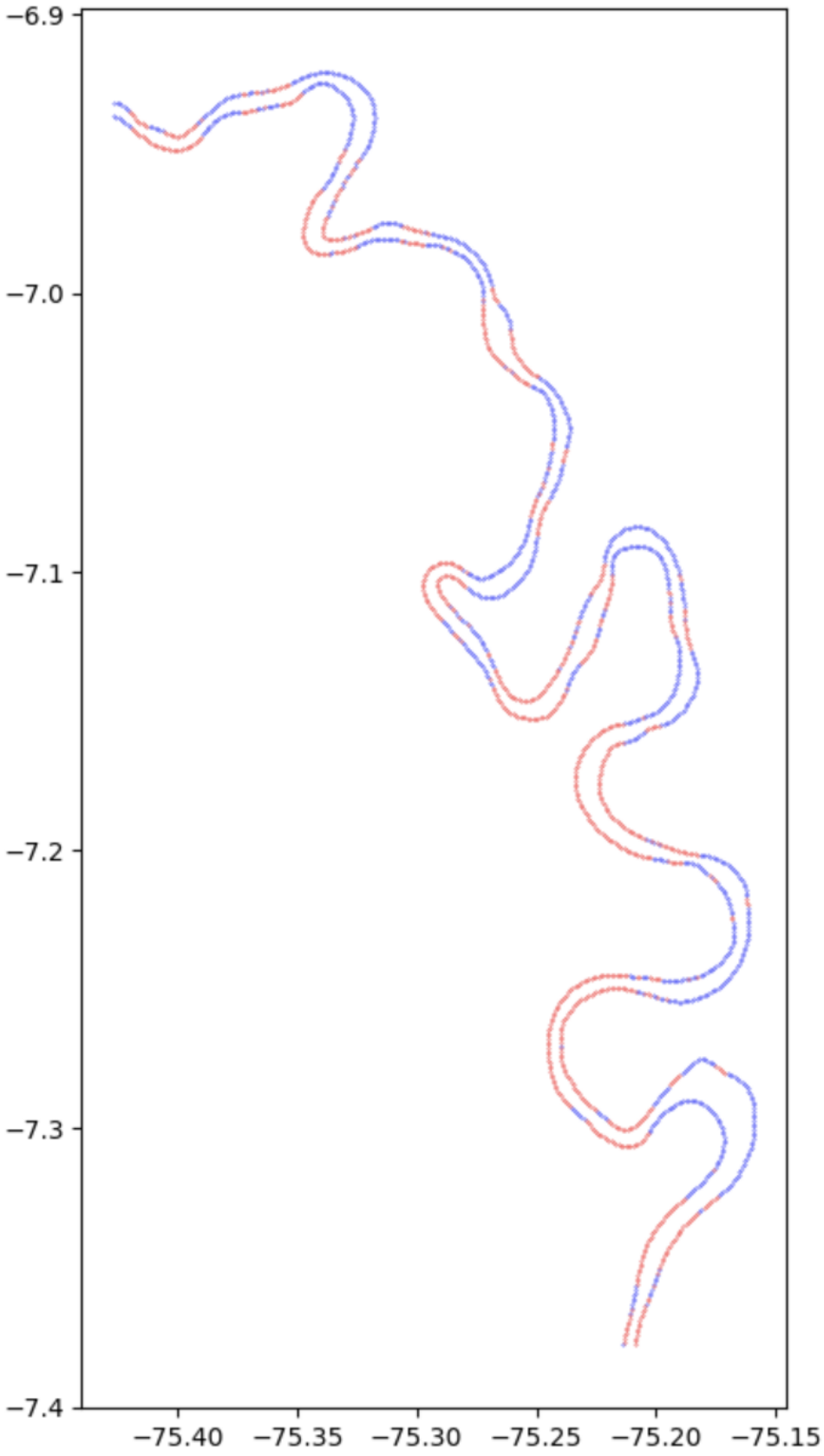
Next, I sifted through each year of data and found the distance each point moved in a year and the curvature of each point. This data was put into a linear model in order to solve for B in the physical model. I re ran the linear regression because I was not sure if the coordinate systems were the same between my data and the data used in the Physical Approach.

3.1.3 Modelling River Movement

To model the river movement, I first calculated the distance each point needed to move based on the width of the river at that point and the curvature. Then I had to find the outward normal direction for each point. To find this, I calculated the slope of the line of the current point and the point before, then the slope between the current point and the point after. I added these slopes up then took the inverse to find the perpendicular slope. I then found the determinant of the following matrix:

$$\begin{vmatrix} dx_1 & dy_1 \\ dx_2 & dy_2 \end{vmatrix}$$

The first row represents changes in each direction between the current point and the point before. The second row represents changes in each direction between the current point and the point after. If the determinant negative then the change was subtracted from the current x and y, and if it were positive, then the change in the x and y direction was added to the current position. This seemed to produce promising results. The following image shows the river with red dots representing the parts of the river with a negative determinant and blue dots representing the parts of the river with a positive determinant.



The final movement was then calculated by the following two algorithm:
if $m == 0$ **then**

$x_{new} = x + sign * (intercept + B * (width * curvature)^{1/2})$
$y_{new} = y$
else if $m == \infty$ **then**
$x_{new} = x$
$y_{new} = y + sign * (intercept + B * (width * curvature)^{1/2})$
else
$x_{new} = x + sign * (intercept + B * (width * curvature)^{1/2})$
$y_{new} = y + sign * (intercept + B * (width * curvature)^{1/2})$
end if
return x_{new},y_{new}

Where sign represents the sign of the determinant explained above. The banks were then filtered using the Savitzky–Golay filter with a window of 50 and a polynomial degree of 4. I also attempted to check for bank line collapses by checking the minimum distance between the newly predicted points and the points on the opposite bank that were closest to that point before it was moved. I did this by storing the closest point on the opposite bank for each point in a Points class. Then I found the distance between the new point and the stored point. If they were under a minimum threshold apart, I found the distance between the x and y coordinates of the predicted point and closest point on the other bank before prediction and subtracted the minimum distance threshold times the distance in each direction divided by the total distance apart.

I then checked for oxbow formations on each bank. This was done by checking if any line segments between each point were intersecting or not. If they were, I kept the indices of the points that were before and after the indices where the intersection was found.

3.2 Approach 2

For this method, we decided to use the Jurua river centered around 37 25.818' N, 122 05.36' W in Brazil that had a section of river where an oxbow lake proceeded to form over the time we observed it. We captured data from between 1984 and 2020.

The approach we used in order to convert the image into actual data points was the same as was used in a previous project this semester: "Our approach for turning the images gathered from Google Earth into actionable data involved creating a color mask to separate the river from the land. Blacking out every pixel that isn't the river, whiting out the river pixels, and subsequently creating a pixel wide outline as the location of our banks. This was done by converting the RGB image to HSV (Hue, Saturation, Value) and cleverly blocking out specific ranges of color based on this specific river...The bank was isolated by looking at the neighbors of each pixel. If the pixel had a neighboring 0 (where the land was) it was considered the bank. Afterward, we cleaned up the images and, using a breadth first search algorithm, generated our first plots." [KSW23] Using this approach, we arrived at the following images:

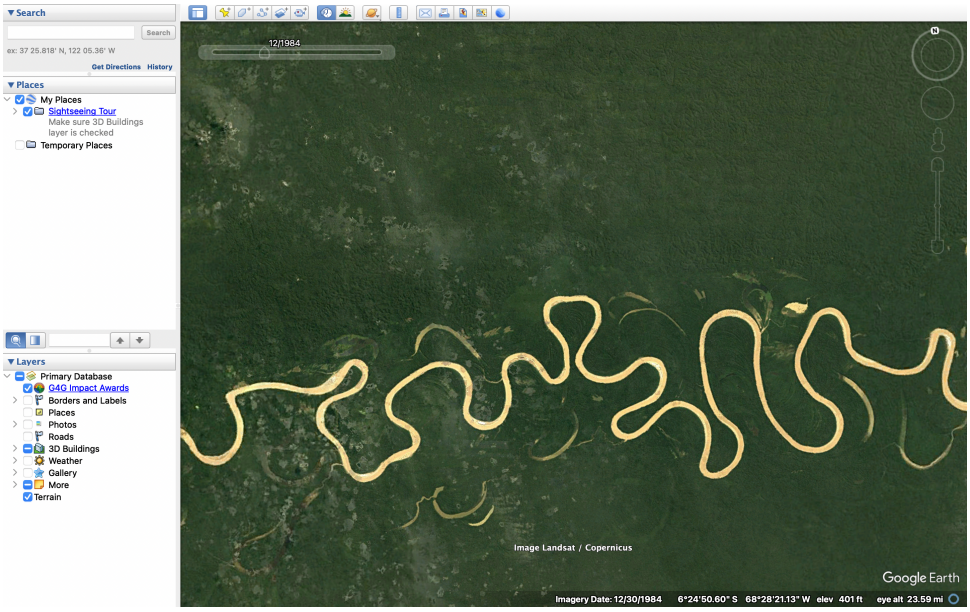


Figure 2: Satellite image of our river section from 1984

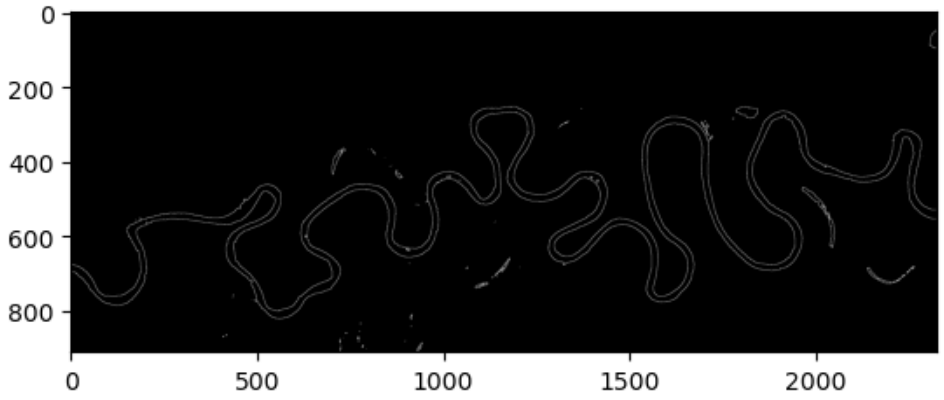


Figure 3: Banks isolated from the Google Earth image

3.2.1 Data Processing

In order to begin modeling the meandering of the river, and to detect any oxbow formation, we needed to accomplish a number of tasks:

- We needed to separate the banks into an upper and lower bank (not to be confused with an inner and outer bank, which has more to do with local curvature).
- We needed to turn the high fidelity image into a slightly grainier image that removed strange cusps and artefacts from the data collection process, instead of a one pixel thick line representing a river bank.
- We also needed to perform some kind of local smoothing on the data points due to the continuous nature of river banks.

Separating the banks proved to be easier than expected by keeping track of a maximum distance traveled in the X coordinate, iterating through the data, and finding the two

points which had the largest change in their X coordinate. This index is where the data could be split into an upper and lower bank.

Once the banks were separated, we took an average of clusters of three data points, in order make the image more grainy and remove some clusters of points. A further smoothing algorithm was used to soften the bank profiles and force them to be less jagged. We used a weighted moving average algorithm, where we treated the upper bank as two different series, an x coordinate series, and a y coordinate series, that were indexed in time, and performed a weighted moving average on them by using the following formula:

$$\hat{X}_i = \sum_{k=-L}^{L} \alpha_k * X_{k+i} \tag{17}$$

Where $\hat{X}_i$ is the new ith smoothed data point, L is the window of the weighted moving average, and α_k is given by this formula:

$$\alpha_k = \frac{1 - k^2 \frac{I_2}{I_4}}{2L + 1 - \frac{(I_2^2)}{I_4}} \tag{18}$$

and I_2 is given by $I_2 = \frac{L(L+1)(2L+1)}{3}$, and I_4 is given by $I_4 = \frac{I_2[3L^2+3L-1]}{5}$. We proceeded to run the series of upper bank points and lower bank points through this algorithm. Now, these are known formulas used for the analysis and smoothing of time series data, which isn't entirely appropriate here, given that our river cannot, strictly speaking, be represented by a function. However, given that the data points are ordered by index, using this algorithm to smooth the series of X data points and Y data points separately both for the upper and lower banks only serves to help reduce some of the jaggedness of our predicted bank profiles.

With the data processed and prepared, we could begin our second approach to modeling the meandering of the river.

3.2.2 Local Outward Normal Movement

From the work done in the **Physics Motivated Approach** section, we determined that the movement of banks occurs on the outer bank in a given curve in the outward normal direction to that point on the bank. The step in this direction is determined by the average speed of the water, which in turn is dependent on some constants times $\sqrt{\kappa}$, where κ is the local curvature of the river.

Let's consider the points $p_u = (x_u, y_u)$, $p_d = (x_d, y_d)$, where p_u is a point on the upper bank, and p_d is a point on the lower bank, and these points are the closest points to each other on opposite banks. Let the water momentum vector be $\hat{w}$, the normal vectors to u_n and d_n, be $\hat{N}_u$ and $\hat{N}_d$, W_{ud} the current width of river, and the corresponding curvatures κ_u and κ_d

The step forward is given by $u_{n+1} = u_n + (c\sqrt{W_{ud}\kappa_u})\hat{N}_u \cdot \hat{w}$ and $d_{n+1} = d_n + (c\sqrt{W_{ud}\kappa_u})\hat{N}_d \cdot \hat{w}$, where c is some constant set at the beginning of the time step.

if $\kappa_u \neq 0$ and $\kappa_d \neq 0$ **then**
  if $\kappa_u < \kappa_d$ **then**
    $d_{n+1} = d_n + (c\sqrt{W_{ud}\kappa_u})\hat{N}_u \cdot \hat{w}$

else
$u_{n+1} = u_n + (c\sqrt{W_{ud}\kappa_d})\hat{N}_d \cdot \hat{w}$
end if
else if $\kappa_u = 0$ and $\kappa_d \neq 0$ **then**
$u_{n+1} = u_n + (c\sqrt{W_{ud}\kappa_d})\hat{N}_d \cdot \hat{w}$
else if $\kappa_u \neq 0$ and $\kappa_d = 0$ **then**
$d_{n+1} = d_n + (c\sqrt{W_{ud}\kappa_u})\hat{N}_u \cdot \hat{w}$
else
$d_{n+1} = d_n$ and $u_{n+1} = u_n$
end if
return u_{n+1},d_{n+1}

If $\kappa \neq 0$, then generally speaking the bank with a smaller curvature corresponds to the outer bank of a given curve, and the movement of the banks should only be in the direction of the outward normal of the outer bank (which is the reason why the algorithm checks the upper and lower curvature of banks). The algorithm also checks for $\kappa = 0$, which indicates a straight line. In that case, if only one of the banks is straight, then we force that bank to move in the direction of the curved bank. If both banks have a $\kappa = 0$, then we don't adjust the points at all, since the water momentum vector should be traveling orthogonal to the normal vectors at those points. Additionally, due to the discontinuous nature of the data, after every time step the smoothing algorithm was applied to both the X coordinates of the upper bank, and then the Y coordinates of the upper bank, and then the process was repeated for the lower bank, in order to reduce the jaggedness in our predictions.

In order to perform this time stepping process, at each point on the river we need to know the unit vector in the direction of the water momentum vector, $\hat{w}$, the normal vector to a given point on the river, $\hat{N}$, and the curvature, κ, at a given point on the river. We also needed to address something that often resulted from this method which was bank collision (where the banks would close onto themselves or overlap). Lastly, there was the challenge of oxbow detection and when to remove the oxbow lakes.

The first problem we chose to address was the question of curvature and the normal vector to a given point on a river bank. Conveniently, menger curvature provides a way to determine both of these values.

3.2.3 Menger Curvature

Menger curvature is defined as the reciprocal of the radius of the circle that passes through three points:

$$C(a, b, c) = \frac{1}{R} = \frac{4A}{|a - b||b - c||c - a|} \tag{19}$$

where a, b, c represent points on the Euclidean plane and A denotes the area of the triangle with corners at a, b, c. An alternate approach involves solving a 3x3 linear system of equations, and finding the radius from the equation of the circle of best fit.

To find normal vectors along the river, it is simple enough to find the outward normal to the circle generated from Menger Curvature. There were some challenges here, however, when the three points were colinear (which was solved by finding the line of best fit and letting the curvature equal to zero). In order to avoid overestimating local changes in

curvature and outward normal direction, we selected three points that were a minimum distance from each other (for simulations this was set to the local river width divided by two).

With Menger Curvature cleanly addressing the issues with curvature and normal vectors, we shifted our attention to the unit water direction vectors.

3.2.4 Water Momentum Vectors

Since we didn't necessarily know what exactly the water momentum vector was, we figured we could estimate it by looking at the banks of the river. Generally speaking, it tracks that the flow of the water would be directed to some extent by the banks of the river. So, we constructed a vector using the following procedure:

- Choose a window size w, and select a point to consider.
- Examine a number of points (equal in number to the window size w) along the lower bank that are indexed to be before the current point p.
- Determine the average change in X and Y coordinate across these points
- Repeat the process for points on the upper bank
- Average the average changes in X on the upper and lower bank, and Y on the upper and lower bank.
- This vector of change in X and change in Y is the estimate for the water momentum vector

For simplicity's sake, we turned all vectors into unit vectors. Using this vector, we then projected this vector onto the normal vector of each point, and made a step in the direction of the normal vector based on the above conditions.

That led us to consider how to force a minimum distance between the two banks.

3.2.5 Bank Collision

Preventing bank collision is a tricky task, however we achieved some success by considering a grid surrounding any given point on a river bank:

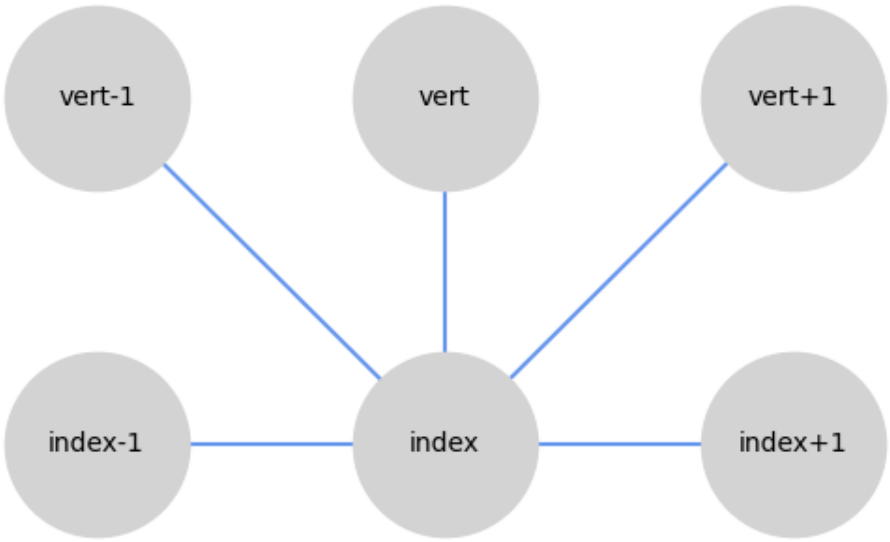


Figure 4: Grid surrounding a point of a given index

It's obvious that the closest points to a given point with index i on the same bank are just the two points at indices $i-1$ and $i+1$ If we can find the closest point on the opposite bank to a given point at the ith index of a bank, then we know the two other closest points on the other bank.

In order to address this problem, as well as to facilitate easier storage of the required information for this time stepping process, we implemented an Object Oriented Programming approach, creating two distinct classes. The first was a generic *point* class with the following attributes and methods:

```
class point:

    def __init__(self,x,y,idx,bank):

        #Inputs
        #X,Y coordinates
        #Index of series
        #Bank Side (Upper or Lower)
        self.x = x
        self.y = y
        self.ind = idx
        self.bank = bank
        self.left = None
        self.right = None
        self.vert = None
        self.diagL = None
        self.diagR = None
        self.width = None
        self.curv = None
        self.WD = None
        self.Norm = None
        self.xf = None
```

```
        self.yf = None
        self.minW = None


    def distance(self,p):

        ...

    def step(self,cof):

        ...

    def adjust(self):

       ...
```

The full code for the methods is omitted for the sake of brevity but included in the appendix.

The second class is a *River* class, with the following methods and attributes:

```
class River:

    def __init__(self):

        self.Upper_Bank = None
        self.Lower_Bank = None

    def add_points(self,points,bank):
        ...

    def indices(self):
      ...

    def NearDB(self):
        ...

    def WaterDirection(self,window):
        ...

    def Smoothing(self,wind,wdsize):
        ...

    def Curves(self,wind):
        ...

    def Update(self,UP,DOWN,w,wc):
        ...

    def Oxbow_Detect(self,window,bank):
        ...

    def Oxbow_Snip(self,wind,wd,wc):
        ...

    def stepping(self,cof):
        ...

    def adjusting(self):
```

```
        ...

    def plot(self,title,past = False):
        ...
```

Again, the full code for the methods is omitted for the sake of brevity, however is included in the appendix attached to this paper.

Using the *adjust* method of the *point* class, we can maintain distance between the grid of points on both the opposite and same bank to the point in question, by "nudging" the points away from each other when a step forward in time would collapse the distance between any pair of points less than some minimum threshold, which for simulations and predictions was set at $\frac{\min W_r}{2}$, where $\min W_r$ is the minimum river width.

3.2.6 Oxbow Detection and Removal

The last consideration for this approach was the methods for detecting and removing oxbows from the bank profile of the river. The detection of the Oxbow lake being formed initially seemed tricky, but in the end, was relatively simple, due to the grid method of maintaining river width, and the classes established in the previous section. The general approach for finding when an oxbow has formed is as follows:

- Find the minimum width of the river by iterating through points on the upper and lower banks.
- Iterate through each point on the upper bank, and from a given point p iterate forward from that point, recording the distance between each subsequent point.
- If the distance between the point p and some point ahead of p but also on the same bank, let's say q, is ever less than some metric (for simulations this was set to $\frac{\min W_r}{2}$), then an Oxbow lake has likely formed, so record the index of point p and the index of point q, and record the indices on the opposite bank that correspond to the points closest to p and q.
- Repeat the process for the lower bank, and record the indices where the above conditions are met.

This method works because an oxbow will be formed by two non-adjacent points on the same bank being less than some minimum distance, so simply scanning along the upper river bank and then the lower river bank one at a time will be sufficient to find any oxbows that form. After finding the indices where the oxbow lake begins and ends, we can now move on to removing the oxbow from the river bank profile. Removing the oxbow was heavily facilitated by the work done in making a *River* and *point* class. When attempting to remove the Oxbow from the bank that it was detected on (which is the inner bank) there is a straightforward process: you cut out the section of bank between the starting index of the oxbow and the ending index of the oxbow. Now, finding the corresponding section of the other bank (the outer bank) to remove such that the river bank profiles do not cross each other is as simple as referencing the `vert.ind` attribute of the *point* class, and making a corresponding cut between the two indices of points on the opposite bank.

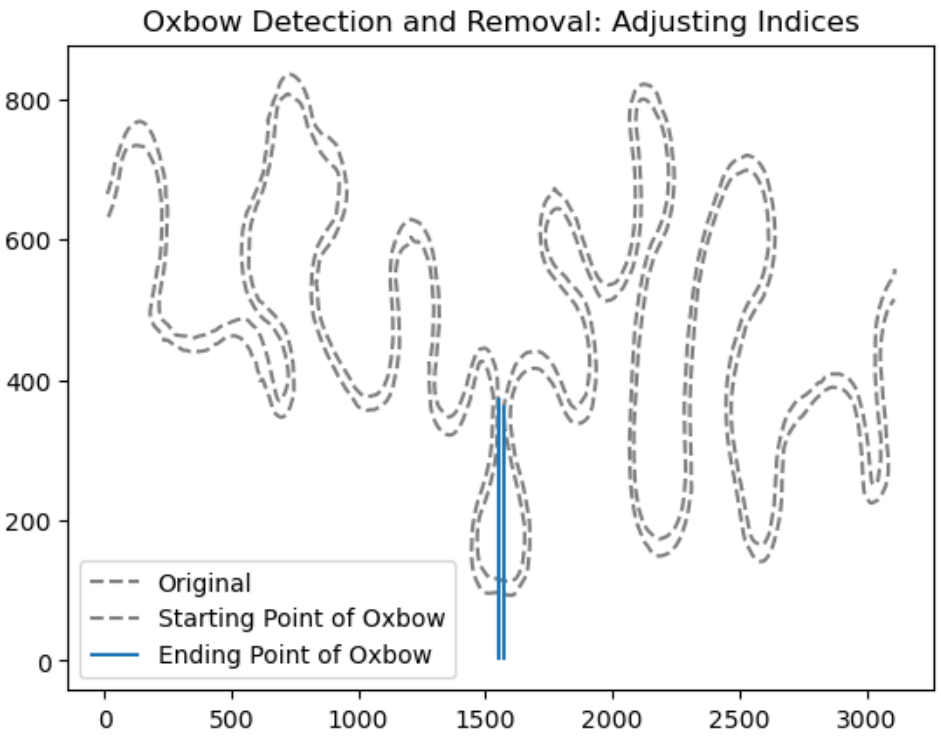


Figure 5: Oxbow Detection and Removal Indices Adjustment

Implementing this in practice, we can achieve the following when we relax the condition around minimum distance between two points on the same bank:

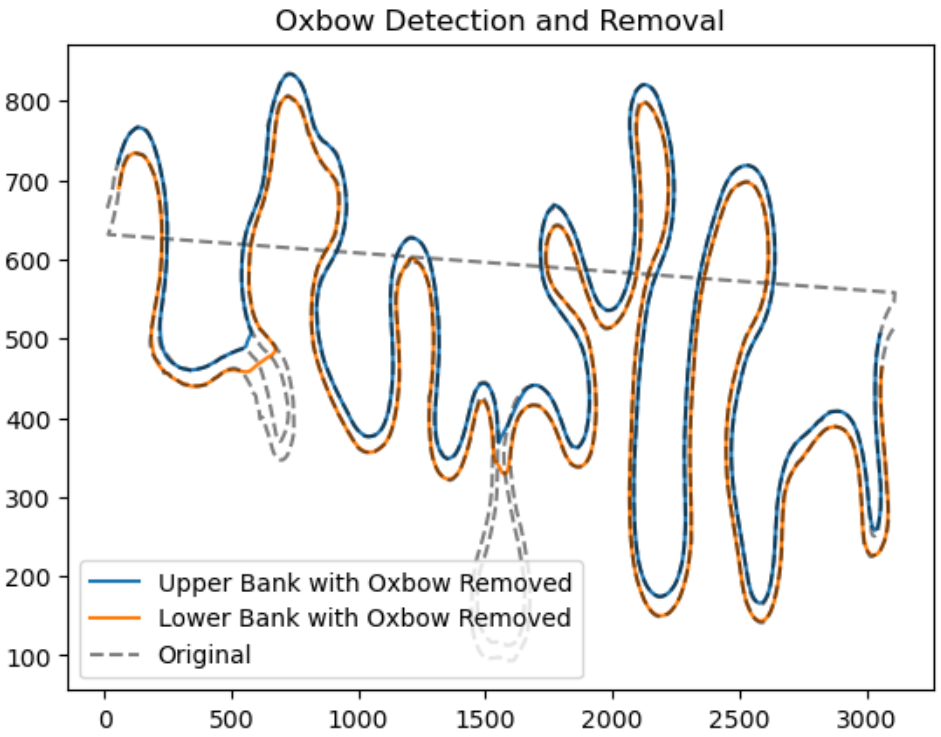


Figure 6: Oxbow Detection and Removal for a Given Bank Profile

With the above algorithms outlined for our time stepping, we can begin to examine

how our algorithm performs compared to real data.

3.3 Approach 3

3.3.1 River and Data Collection

We gathered data on section of the amazon river (Fig 6).To gather our data we used google earth pro to take the actual points of the river at different times. We used the path feature and the historical imagery feature to get the path of the section of our river for different time intervals. The different time intervals we took were 2020,2015,2010,2005 and 2000. Each year we took data points for the outside and the inside of our river. We didn't keep the number of points or the separation between the points constant. On average we had around 70 data points for the inside curve of the river and the outside curve of the river.

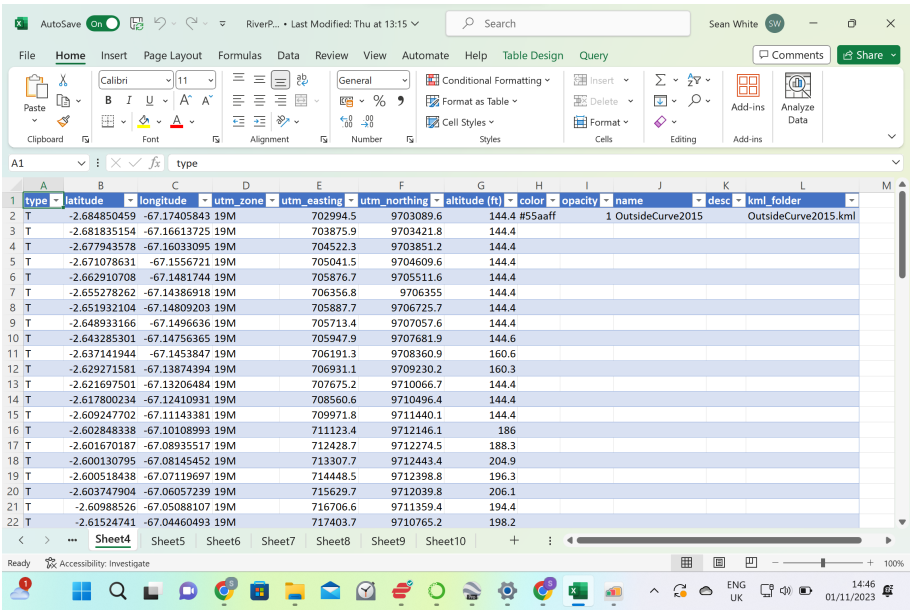
Figure 7: River Path from Google Earth

For us to be able to use our data to model the river we had to be able to transfer it into python. First we saved the data points from the river as an excel file. Then we imported this excel file to python. We then made 10 arrays in python to store all of this data.(Fig 7)

We now had 5 arrays of data points of the outside of the river for different years and 5 arrays with the data for the inside of the river.

3.3.2 Finding the Outward Normal

To find the outward normal we had to first use menger curvature to find the corresponding centre of the a circle mapped for every three points. We used the function provided to us to find the menger curvature of any three points. Once we had the centre of each point we subtracted the point from the centre to find the direction of the outward normal. In the graph below this is shown.Each point is connected to its centre by a straight line in this graph. Once we had the direction of the outward normal I wanted to find the unit



type	latitude	longitude	utm_zone	utm_easting	utm_northing	altitude (ft)	color	opacity	name	desc	kml_folder
T	-2.684850459	-67.17405843	19M	702994.5	9703089.6	144.4	#55aaff	1	OutsideCurve2015		OutsideCurve2015.kml
T	-2.681835154	-67.16613725	19M	703875.9	9703421.8	144.4					
T	-2.677943578	-67.16033095	19M	704522.3	9703851.2	144.4					
T	-2.671078631	-67.1556721	19M	705041.5	9704609.6	144.4					
T	-2.662910708	-67.1481744	19M	705876.7	9705511.6	144.4					
T	-2.655278262	-67.14386918	19M	706356.8	9706355	144.4					
T	-2.651932104	-67.14809203	19M	705887.7	9706725.7	144.4					
T	-2.648933166	-67.1496636	19M	705713.4	9707057.6	144.4					
T	-2.643285301	-67.14756365	19M	705947.9	9707681.9	144.6					
T	-2.637141944	-67.1453847	19M	706191.3	9708360.9	160.6					
T	-2.629271581	-67.13874394	19M	706931.1	9709230.2	160.3					
T	-2.621697501	-67.13206484	19M	707675.2	9710066.7	144.4					
T	-2.617800234	-67.12410931	19M	708560.6	9710496.4	144.4					
T	-2.609247702	-67.11143381	19M	709971.8	9711440.1	144.4					
T	-2.602848338	-67.10108993	19M	711123.4	9712146.1	186					
T	-2.601670187	-67.08935517	19M	712428.7	9712274.5	188.3					
T	-2.600130795	-67.08145452	19M	713307.7	9712443.4	204.9					
T	-2.600518438	-67.07119697	19M	714448.5	9712398.8	196.3					
T	-2.603747904	-67.06057239	19M	715629.7	9712039.8	206.1					
T	-2.60988526	-67.05088107	19M	716706.6	9711359.4	194.4					
T	-2.61524741	-67.04460493	19M	717403.7	9710765.2	198.2					

Figure 8: Excel File of Data

vector of the outward normal. To do this I found the magnitude of the outward normal vector which was equal to.

$$OutwardNormal = (x_i, y_i)$$

$$|OutwardNormal| = (x_i^2 + y_i^2)^{1/2}$$

then the Unit Outward Normal was found by

$$\hat{OutwardNormal} = \frac{(x_i, y_i)}{(x_i^2 + y_i^2)^{1/2}}$$

Now from this calculation we had the direction that each point moved out in over time. We completed this calculation for both the inside bank of the river and the outside bank of the river. Now we had two arrays for the direction of each outward normal of each point on the inside and outside bank.(Fig 8)

3.3.3 Finding the Width

A major breakdown in our previous model was the collapse of our river banks over time. To stop the river banks from collapsing our idea was to find the width at each point of the river and then try to manipulate the point if it's corresponding width was under a certain value. To find the width of the river we first had to define what the width of a river was. We define the width of the river as the distance from a point of the inside bank of the river to the point of intersection of the orthonormal projection on the outside bank of the river. (Fig 9) Now since we had the corresponding centre of each point from our menger curvature we plotted a straight line from the centre of each point to the actual point on the bank. We found the point at which this line intersected the opposite bank.We then calculated the distance between the intersection point and the actual point. Occasionally the normal projection line would intersect the opposite bank more than once. (Fig 10) When this happened we found the distance between each of the multiple points to the

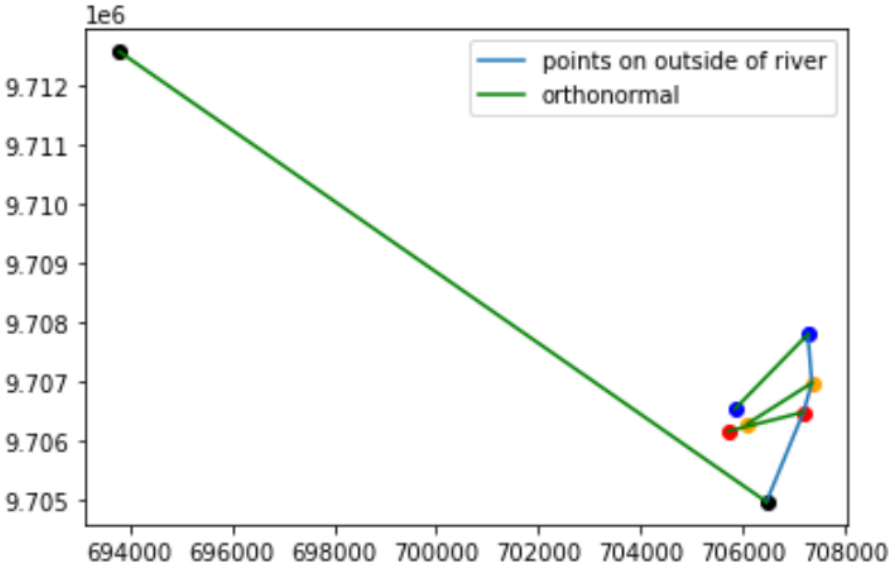


Figure 9: Finding Outward Norm

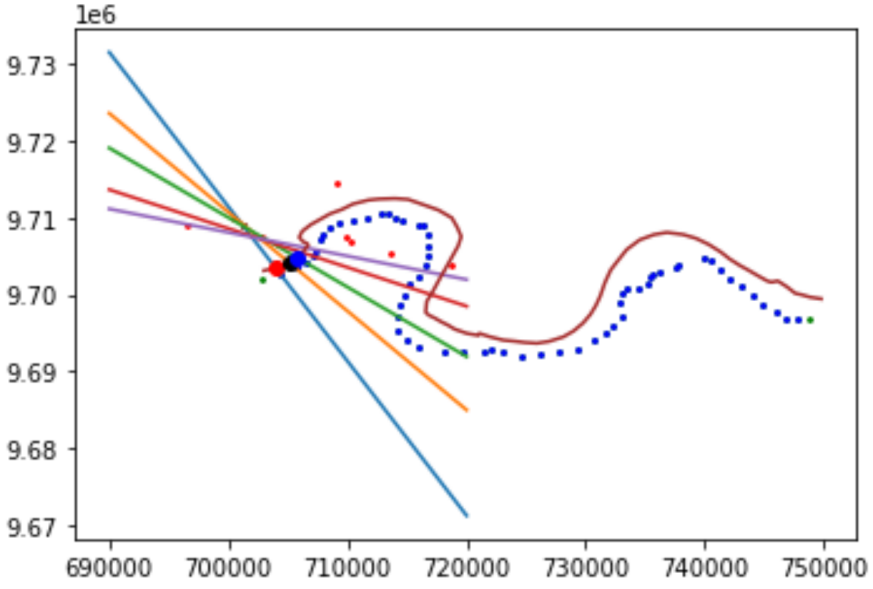


Figure 10: Approach to find Width

```
POINT (703954.3275407022 9703467.158203453)
POINT (705132.2513516181 9704095.580740033)
POINT (705786.0664783607 9704767.712630956)
0  0    POINT (706197.220 9705442.045)
   1    POINT (717003.518 9699990.775)
   2    POINT (728002.067 9694442.523)
dtype: geometry
0  0    POINT (706529.570 9706058.171)
   1    POINT (717934.375 9702571.721)
   2    POINT (730861.453 9698619.911)
dtype: geometry
0  0    POINT (705988.708 9707558.949)
   1    POINT (718374.540 9703774.282)
   2    POINT (731228.105 9699846.693)
dtype: geometry
0  0    POINT (706425.272 9708487.553)
   1    POINT (718263.251 9703466.516)
   2    POINT (730699.103 9698191.894)
dtype: geometry
0  0    POINT (707623.894 9709685.829)
   1    POINT (717151.253 9700364.134)
   2    POINT (723665.702 9693990.310)
dtype: geometry
POINT (708761.1503018468 9710606.297617495)
POINT (709976.4948393347 9711384.217372088)
POINT (710915.7870010338 9711930.681714578)
POINT (712795.166097572 9712401.476139091)
POINT (713856.112300845 9712457.461490724)
POINT (715523.6085673639 9712110.963067468)
POINT (715890.2197490183 9711912.635647289)
POINT (716752.2465874626 9711444.939559801)
POINT (717635.2417662193 9710959.938712442)
0  0    POINT (706446.555 9705904.273)
   1    POINT (719377.303 9708128.384)
```

Figure 11: The Intersecting Points

point we're finding the width for. Then the minimum distance was equal to the width at that point. We now had again two arrays for the width of the river at the points on the inside river bank and the outside river bank.

3.3.4 Finding the Proportionality Constant

Now since we have the river width,curvature and the direction of the normal at every point along the river we had a solid basis to model our river. Using the physical properties of the river that Daniel had found from his work we had a base to model the river. We know that the river moved out in the normal direction in a step proportional to a constant times the square root of the curvature by the width.

$$Outwardstep = B(curvature \times width)^{1/2}$$

To find the Constant in this model we plotted our model for 2015 against the actual plotted data to see which fitted best.

From these graphs it is clear B=25 (Fig 12) is much more accurate at replicating the actual movement of the river than B=50 (Fig 11).For our model we take B=25

3.3.5 Comparing to our previous model

In our previous model we only accounted for movement in one dimension and didn't move the river in the outward normal direction. This caused our model to get very jagged over long periods of time. (Fig 13) In our new model we used the physical properties of the river as our motivation. This resulted in moving our points in the orthonormal direction and moving them in a step proportional to a constant times the square root of the curvature and the width. (Fig 14)

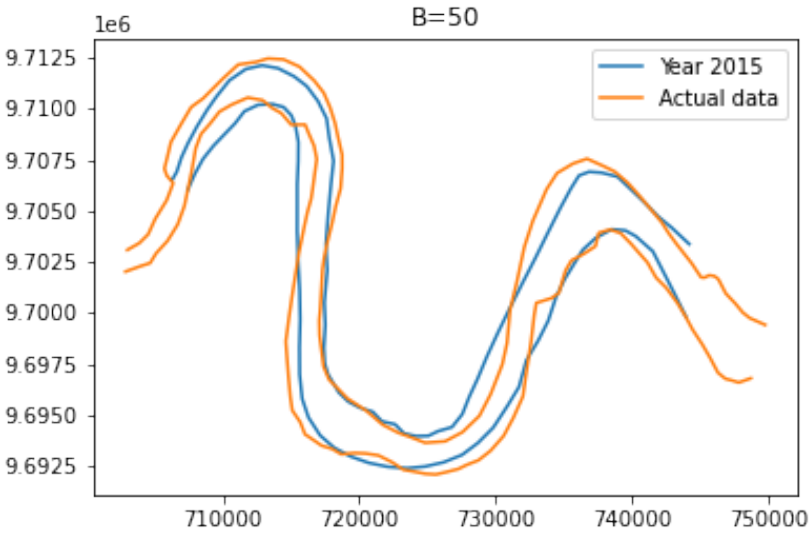


Figure 12: Finding the constant B=50

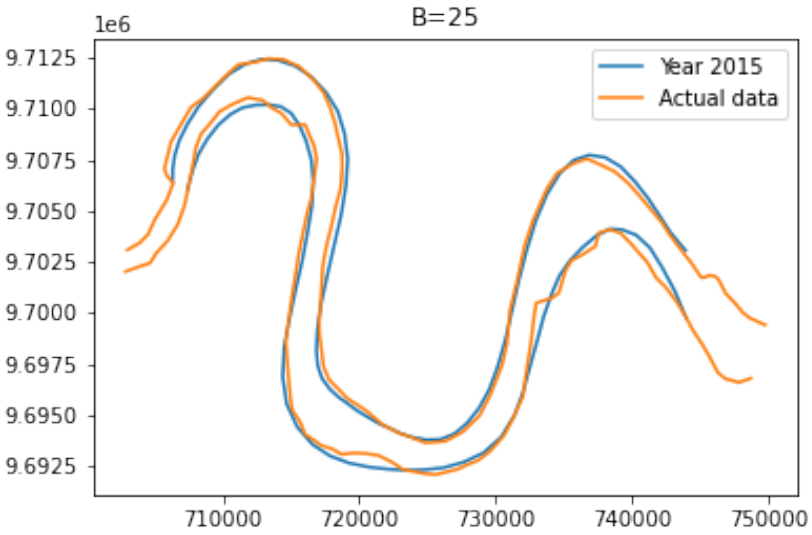


Figure 13: Finding the constant B=25

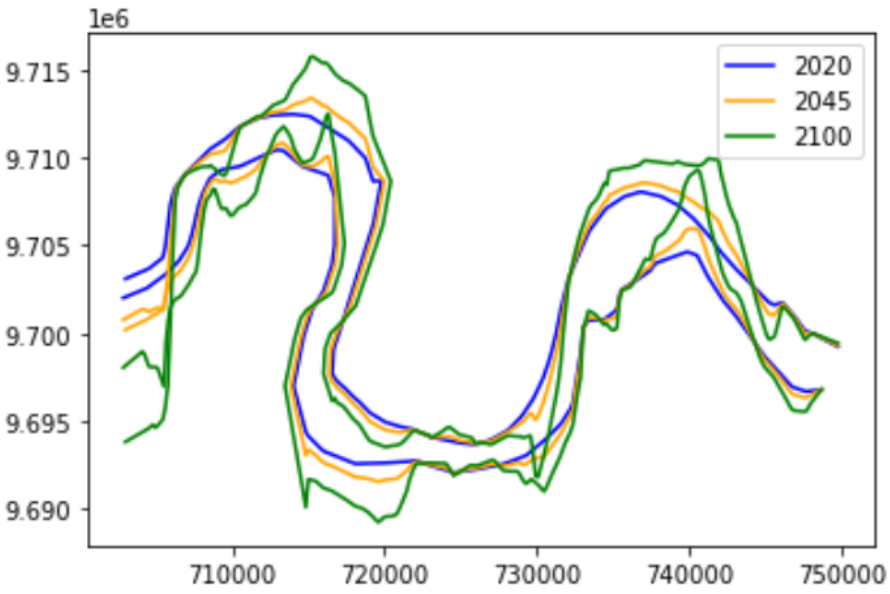


Figure 14: Finding the constant B=25

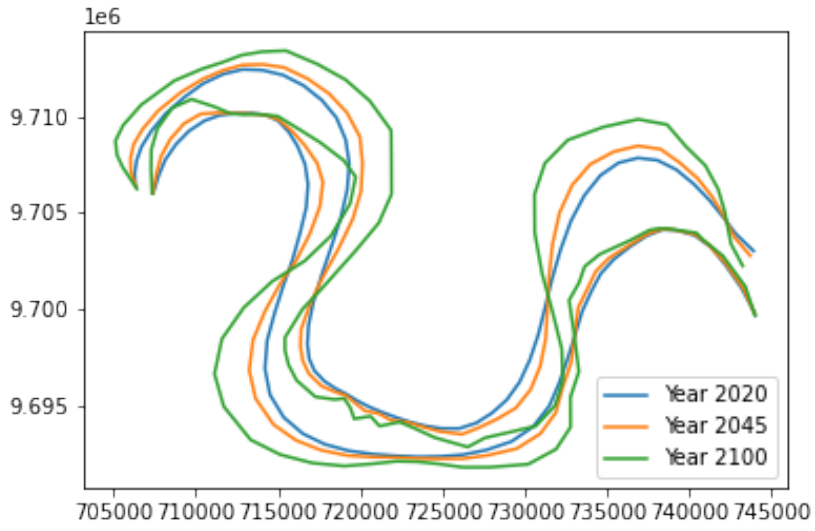


Figure 15: New River

4 Results

4.1 Approach 2

Generally speaking, this approach showed a good degree of promise. When making a projection 20 years into the future, from 1984-2004, using the approach outlined as **Approach 2**, with the constant in front of the step in the outward normal set to the constant found from the linear regression in the **Physics Motivated Approach** section ($B = 350.15$), and without incorporating new data, we get the following plot:

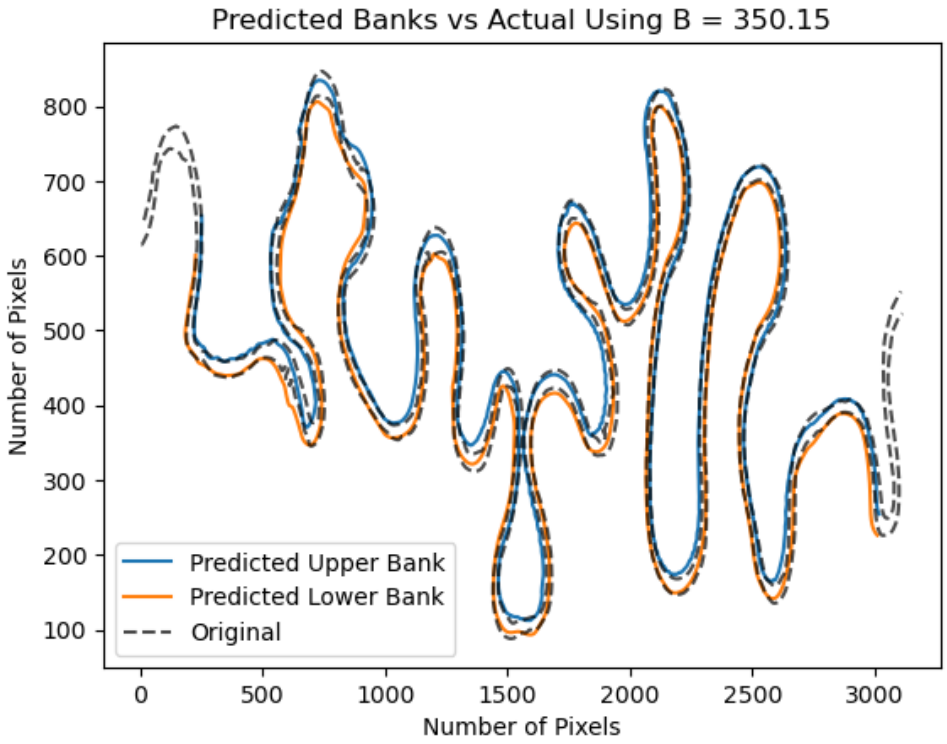


Figure 16: Time step 22 Years Ahead Using **Approach 2**

Based on that time step forward, our second approach was not bad, and appears to have some promise (at least for this section of the Jurua river): the river bank profile of the actual river compared to our predicted bank profile is relatively close to the actual profile of the bank. Interestingly, some of the loops in the true river have outer banks that are further away from the inner banks (when compared to the predicted profile of the river banks). By no means does this prove that our method is guaranteed to be accurate, but it does indicate some promise in this method.

Now the other goal of this work is to be able to detect when an oxbow forms and to be able to remove that oxbow lake from the bank profile. To examine the effectiveness of **Approach 2** on this task, let's consider a time step of 5 years, from 2002 to 2007. The end date was picked because an oxbow has formed in the river section we were examining by that time.

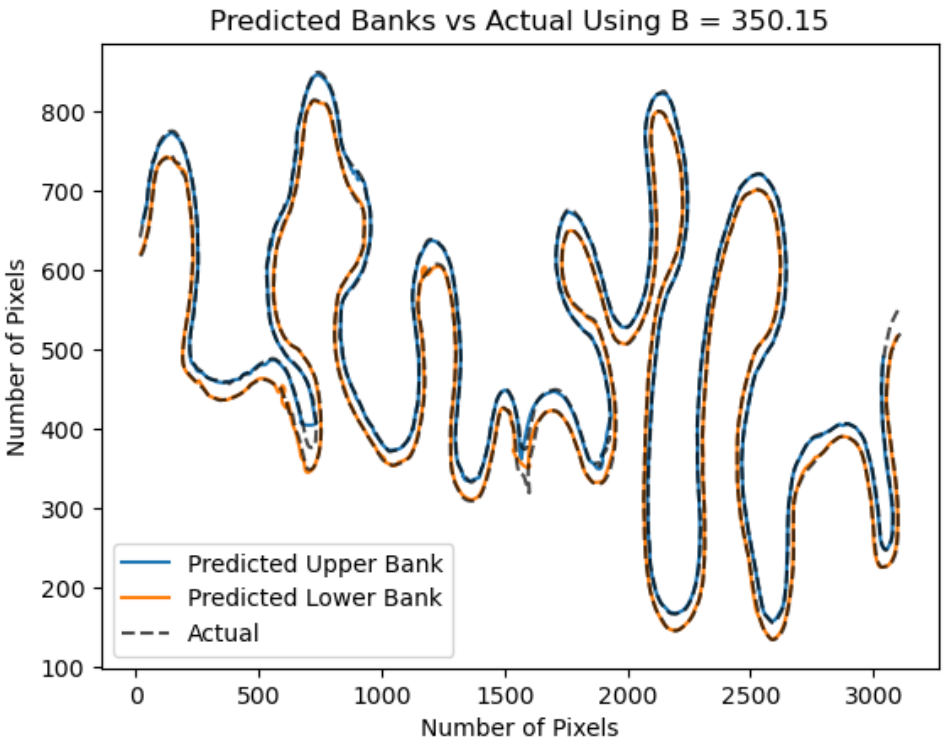


Figure 17: Time step 5 Years Ahead Using **Approach 2**: Detecting Oxbows

Notably, while some of the banks don't match, this method is incredibly accurate on a short term scale, and was able to effectively predict, detect, and remove an oxbow that formed over time, and alter the bank profiles to fairly accurately match the true bank profiles of the river once the oxbow lake formed and was no longer part of the river bank profile.

For fun, let's examine a further prediction, 50 time steps forward, from 1984 to 2034 with a larger constant $c = 600$ multiplying the step in the outward normal direction, and see what the model predicts.

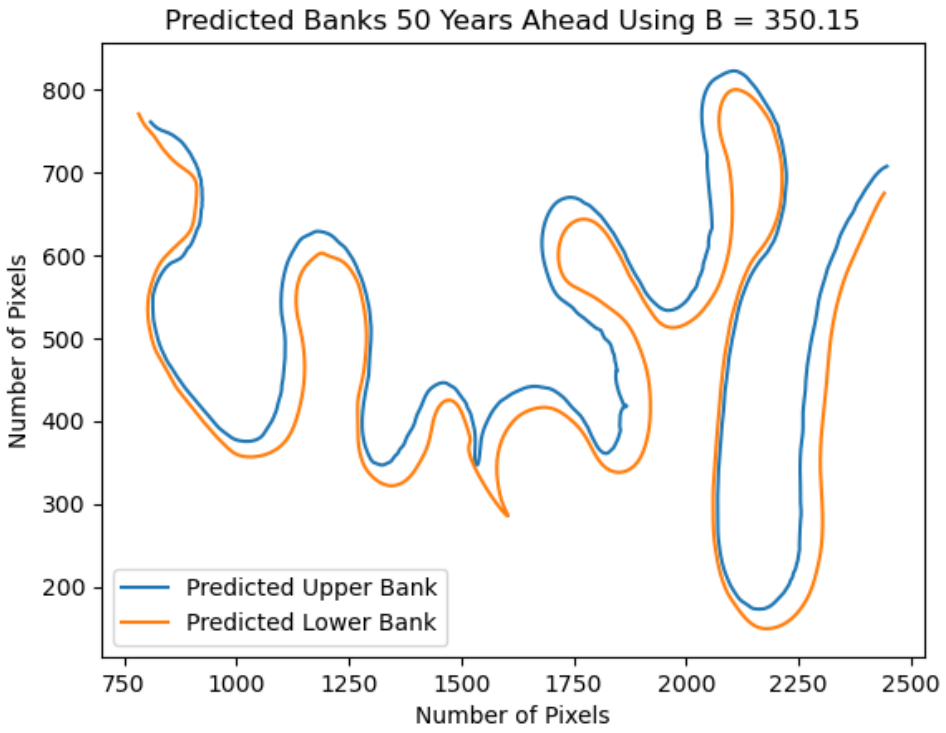


Figure 18: Time step 50 Years Ahead Using **Approach 2**

It's hard to empirically see whether this prediction has any validity, since this is a projection into the future. Maybe in 11 years, we can look back and verify if this predicted profile is anywhere close to the reality. What we can do is compare the predicted bank profile to the bank profiles we do have, and it's clear that if this method is tasked to predict too far into the future, it will struggle: the banks the model predicted get rather close together: they don't quite cross, but the river does get rather narrow. Furthermore, the profile of the river does not look like that of a real river, so it's fair to say this method will struggle when tasked to predict far enough ahead.

4.2 Approach 3

This approach was a huge improvement from our previous interpolation model. It modelled the shape of a river much more effectively than the interpolation approach and I think this is down to the fact that we used the physics based model as our main motivation.

However after around 80 years our model does start to break down and the river banks begin to collapse. From our intuition on how ox-bows form we decided that when the river banks collapse it indicates that a ox-bow is near formation. The problem with this was that ideally as the banks close in the ox-bow formation should be gradually forming not form in an instant. Also since we took data on a small section of a river we didn't have enough data to see where the river connects to after ox-bow formation.

4.3 Further Work

The biggest challenge with **Approach 1**, was the movement of the points. We were unable to make the points move toward the outward normal and after one time step, the points became scrambled and it did not work. If we had more time, we would re do the

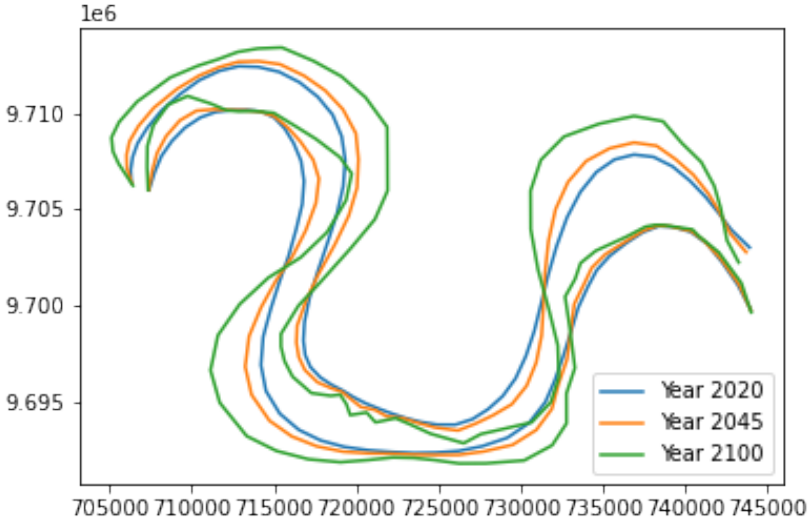


Figure 19: New River

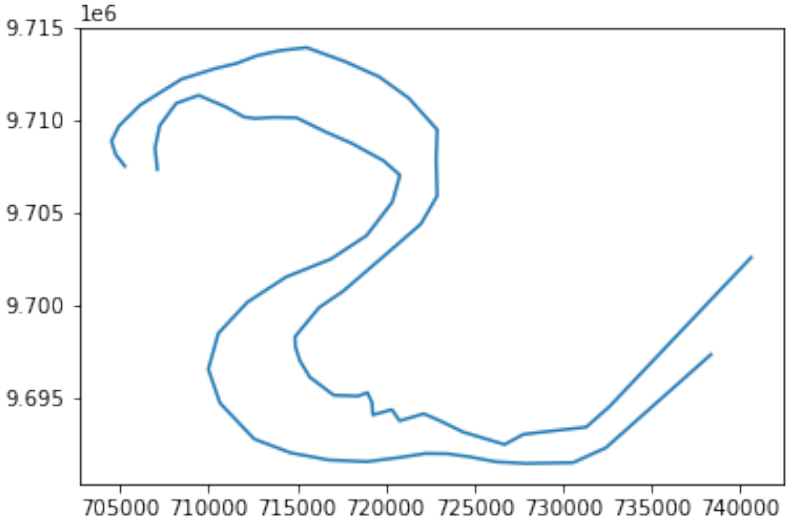


Figure 20: Ox Bow Formation

portion of the physical model we were working with. I think using the curvature method that was proposed was too different to the curvature outlined in the physical approach. We think this smoothing method would be a good way to smooth the river as it does not cut out points at the end and is able to fit polynomial lines to small sections of the river. Also, using the RivMat as a data pre processing tool, we are able to get center lines which would give us the distance to the outer bank. The outer and inner banks kind of switch sides of the river and using the determinant process could help categorize which bank serves as the outer or inner bank in each portion rather than the direction of the outward normal. With the addition of the center line, we could find how the center line moves based on the distance it is from the outer bank where the outer bank isn't necessarily which side of the river it is on, but which side of the river is on the outside of the curve of the river. The inner bank movement could be calculated the same way, but the width would increase to be the full width of the river rather than the half width. The outer bank movement is zero (since the width is taken to be distance from the outer bank), but using the distances the center line and inner bank move, we could find the changes in the outer bank so that the river does not collapse upon itself.

One challenge with **Approach 2**, and generally when using smoothing algorithms, is dealing with an issue of vanishing data points from the ends of the river. The smoothing algorithm looks at a window of points along the series of data, and so the resulting data set is smaller by that window size. When making relatively small predictions forward (say time steps of a year or so), this doesn't significantly impact the river banks. However, when constructing a prediction over a much larger time period, or performing the prediction more frequently (say over a week to week basis), this causes quite the issue, where for a window size of 12 points, if we take 300 steps, the entire data set has vanished. One solution is to restrict the window for smoothing at the ends of the data set, effectively shrinking the window down eventually to half the size at the endpoints of the data set. Another approach would be to slowly feed points into the data set over time at regular intervals, which could in fact boost the accuracy of the approach as you "correct" the algorithm by providing true reference points over time. Lastly, an inelegant solution would be to just expand the length of banks captured in the original screenshot of the river on Google Earth (basically padding the original data set so that the smoothing algorithm never cuts off points of interest).

Additionally, it would be interesting to explore a more sophisticated approach for preventing the banks from clashing instead of enforcing a minimum distance. We suspect this depends on water flow through the river: in dry seasons the minimum width of the river with generally decrease, whereas in high flow years, the river will greatly exceed the banks. Incorporation of water flow information would also be important to calibrate how fast banks move or erode: as an example again, in higher flow years, there would be a much higher water force pushing and dragging material from the bottom and sides of banks.

References

[Eng74] Frank Engelund. "Flow and Bed Topography in Channel Bends". In: *Journal of the Hydraulics Division* 100.11 (1974), pages 1631–1648. https://doi.org/10.1061/JYCEAJ.0004109.

[GB02] W. Graf and K. Blanckaert. "Flow around bends in rivers". In: (Jan. 2002).

[Jon23] Jon. “RivMAP - River Morphodynamics from Analysis of Planforms”. In: *MATLAB Central File Exchange*. (2023). https://www.mathworks.com/matlabcentral/fileexchange/58264-rivmap-river-morphodynamics-from-analysis-of-planforms.

[KSW23] Collin Kadlecek, Alden Soto, and Harrison Wolf. “Modeling Rivers”. Unpublished Manuscript. 2023.

5 Appendix

5.1 Appendix - Python Code for Approach 2

```
import random
import numpy as np
import operator
import math
import matplotlib.pyplot as plt
import pandas as pd

import matplotlib.animation as anim
from IPython.display import HTML
plt.rcParams["animation.html"] = "jshtml"


text = '10.50.13AM.txt'
f = open(text,'r')
Xa = f.readline()[1:-2]
Ya = f.readline()[1:-2]
if(Xa != Ya):
    print(len(Xa), ' ', len(Ya))

print(Xa[0], ' ', Xa[-1])

Xa=  Xa.split(',')
Ya = Ya.split(',')
for i in range(0,len(Xa)):
    Xa[i] = float(Xa[i])
    Ya[i] = float(Ya[i])
Xz = []
Yz = []
Xz.append(Xa)
Yz.append(Ya)

def curv_normal_menger(pts):
    x,y,z = pts
    A = np.array([[2*x[0],2*x[1],1],[2*y[0],2*y[1],1],[2*z[0],2*z[1],1]
                                                    ])
    b = np.array([-1*(x[0]**2+x[1]**2),-1*(y[0]**2+y[1]**2),-1*(z[0]**2
                                                    +z[1]**2)])

    try:
        coeffs = np.linalg.solve(A,b)
        a,b,c = coeffs
        center = [-a,-b]
        if 1/math.sqrt(a**2+b**2-c) < 1e-12:
            k = 0
```

```
43          else:
44              k = 1/(math.sqrt(a**2+b**2-c))
45          mag = math.sqrt((2*(y[0]-center[0]))**2+(2*(y[1]-center[1]))**2
                                                    )
46          norm = [2*(y[0]-center[0])/mag,2*(y[1]-center[1])/mag]
47
48          return k,norm
49      except:
50
51          dy1 = y[1]-x[1]
52          dx1 = y[0]-x[0]
53          dx2 = z[0]-y[0]
54          dy2 = z[1]-y[1]
55          dx3 = z[0]-x[0]
56          dy3 = z[1]-x[1]
57
58          if dx1 == 0:
59              return 0, [1,0]
60          else:
61              mag = math.sqrt((dy1/dx1*y[0])**2+1)
62              return 0,[dy1/dx1*y[0]/mag,-1/mag]
63
64  def avg(rivX,rivY,index,a):
65      testptsd = list(zip(rivX[:index+1],rivY[:index+1]))
66      pd = []
67      xd = []
68      yd = []
69      avg = a
70      for i in range(0,len(testptsd)-avg,avg):
71          avgptxd = sum(rivX[i:i+avg+1])/(avg)
72          avgptyd = sum(rivY[i:i+avg+1])/(avg)
73          pd += [[avgptxd,avgptyd]]
74          xd += [avgptxd]
75          yd += [avgptyd]
76      testptsu = list(zip(rivX[index+1:],rivY[index+1:]))
77      pu = []
78      xu = []
79      yu = []
80
81      for i in range(0,len(testptsu)-avg,avg):
82          avgptxu = sum(rivX[i+1+index:i+index+1+avg+1])/(avg)
83          avgptyu = sum(rivY[i+index+1:i+index+1+avg+1])/(avg)
84          pu += [[avgptxu,avgptyu]]
85          xu += [avgptxu]
86          yu += [avgptyu]
87
88      return pu,pd,[xu,yu],[xd,yd]
89
90  def avgcurvs(ser,average):
91      nser = []
92      indexes = []
93      for i in range(average,len(ser)-average,average):
94          s = sum(ser[i-average:i+average])/(2*average)
95          nser += [s]
96          indexes += [i]
97      return nser,indexes
98
99  def pad(series,indices,maxlen):
```

```
    nseries = []
    pastind = 0
    for i in range(len(indices)):
        for _ in range(pastind,indices[i]):
            nseries += [series[i]]
        pastind = indices[i]

    nseries += (maxlen-indices[-1])*[series[-1]]
    return nseries

def sep(serx):
    mx = 0
    ind = 0
    for i in range(len(serx)-1):
        diff = serx[i]-serx[i+1]
        if mx < diff:
            mx = diff
            ind = i
    return ind

def smooth(ser,L):
    I_2 = L*(L+1)*(2*L+1)/3
    I_4 = I_2*(3*(L**2)+3*L-1)/5

    denom = (2*L+1-(I_2**2)/I_4)

    def a_i(L,i):
        i_sqr = i**2
        return (1-i_sqr*I_2/I_4)/denom

    def f_hat(a,X,i,L):
        res = 0
        for k in range(-L,L+1):
            r = a(L,k)*ser[k+i]
            res += r
        return res
    weighted_average = [f_hat(a_i,ser,i,L) for i in range(L,len(ser)-L)
                                                                    ]
    return weighted_average

class point:

    def __init__(self,x,y,idx,bank):

        #Inputs
        #X,Y coordinates
        #Index of series
        #Bank Side (Upper or Lower)
        self.series = None
        self.x = x
        self.y = y
        self.ind = idx
        self.bank = bank
        self.left = None
        self.right = None
        self.vert = None
        self.diagL = None
        self.diagR = None
```

```
157             self.width = None
158             self.curv = None
159             self.WD = None
160             self.Norm = None
161             self.xf = None
162             self.yf = None
163             self.minW = None
164 
165 
166         def distance(self,p):
167 
168             return math.sqrt((self.x-p.x)**2+(self.y-p.y)**2)
169 
170         def step(self,cof):
171 
172             xvp = self.vert.x
173             yvp = self.vert.y
174             xcp = self.x
175             ycp = self.y
176 
177             wd = np.array(self.WD)
178             NV = np.array(self.vert.Norm)
179             NC = np.array(self.Norm)
180 
181             compC_WD = np.dot(wd,NC)
182             compV_WD = np.dot(wd,NV)
183 
184             xnp,ynp = xcp+cof*math.sqrt(self.curv)*compC_WD*NC[0],ycp+cof*
                                                    math.sqrt(self.curv)*
                                                    compC_WD*NC[1]
185 
186             if self.curv == 0 or (self.curv > self.vert.curv and self.curv
                                                    != 0):
187 
188                 xnp,ynp = xcp+cof*math.sqrt(self.vert.curv)*compV_WD*NV[0],
                                                        ycp+cof*math.sqrt(self.
                                                        vert.curv)*compV_WD*NV[
                                                        1]
189 
190             elif self.curv == 0 and self.vert.curv == 0:
191                 xnp,ynp = xcp,ycp
192 
193             self.xf = xnp
194             self.yf = ynp
195 
196         def adjust(self):
197 
198             grid = [self.vert, self.diagR, self.diagL, self.left, self.
                                                    right]
199 
200             for i in grid:
201                 if i:
202                     dn = math.sqrt((self.xf-i.xf)**2+(self.yf-i.yf)**2)
203 
204                     if dn < self.minW/2:
205 
206                         dx = i.x-self.x
207                         dy = i.y-self.y
```

```
                    mag = math.sqrt(dx**2+dy**2)
                    cx = self.minW*dx/mag
                    cy = self.minW*dy/mag

                    self.xf = self.xf - cx/2
                    self.yf = self.yf - cy/2
                    i.xf = i.xf + cx/2
                    i.yf = i.yf + cy/2


                self.x = self.xf
                self.y  = self.yf
                i.x = i.xf
                i.y = i.yf

class River:

    def __init__(self):

        self.Upper_Bank = None
        self.Lower_Bank = None

    def add_points(self,points,bank):

        if bank == 'upper':
            self.Upper_Bank = points
        elif bank == 'lower':
            self.Lower_Bank = points

    def indices(self):

        for i in range(len(self.Upper_Bank)):
            pt = self.Upper_Bank[i]
            pt.ind = i

        for j in range(len(self.Lower_Bank)):
            pt = self.Lower_Bank[j]
            pt.ind = j

    def mindist(self):

        for ptUP in self.Upper_Bank:
            md = 1e256

            for i in self.Lower_Bank:
                if not i.width:
                    print(i.vert)
                md = min(i.width,md)

            ptUP.minW = md

        for ptDOWN in self.Lower_Bank:
            md = 1e256

            for i in self.Upper_Bank:
                if not i.width:
                    print(i.vert)
                md = min(i.width,md)
```

```
266
267             ptDOWN.minW = md
268
269     def NearLR(self):
270
271         for i in self.Upper_Bank:
272             if i.ind:
273                 if i.ind == 0:
274                     i.left == None
275                 elif i.ind != 0:
276                     if i.bank == 'upper':
277                         i.left = self.Upper_Bank[i.ind-1]
278                     else:
279                         i.left = self.Lower_Bank[i.ind-1]
280
281                 elif i.ind == len(self.Upper_Bank)-1:
282                     i.right = None
283                 else:
284                     if bank == 'upper':
285                         i.right = self.Upper_Bank[i.ind+1]
286                     else:
287                         i.right = self.Lower_Bank[i.ind+1]
288
289     def NearDB(self):
290
291         for ptUp in self.Upper_Bank:
292             mn = 1e256
293             for i in self.Lower_Bank[max(0,ptUp.ind-10):min(ptUp.ind+11
                                                 ,len(self.Lower_Bank))]
                                                 :
294                 d = ptUp.distance(i)
295                 if mn > d:
296                     mn = d
297                     idex = i.ind
298             ptUp.vert = self.Lower_Bank[idex]
299             ptUp.width = mn
300             if not ptUp.vert:
301                 print('Failure to Find Vertical Point')
302                 ptUp.vert = self.Lower_Bank[min(ptUp.ind,len(self.
                                                     Lower_Bank))]
303                 ptUp.width = ptUp.distance(ptUp.vert)
304             if idex == 0:
305                 ptUp.diagL = None
306                 ptUp.diagR = self.Lower_Bank[idex+1]
307             elif idex != 0 and idex < len(self.Lower_Bank)-1:
308                 ptUp.diagL = self.Lower_Bank[idex-1]
309                 ptUp.diagR = self.Lower_Bank[idex+1]
310             elif idex == len(self.Lower_Bank)-1:
311                 ptUp.diagL = self.Lower_Bank[idex-1]
312                 ptUp.diagR = None
313
314         for ptDn in self.Lower_Bank:
315             mn = 1e256
316
317             for i in self.Upper_Bank[max(0,ptDn.ind-10):min(ptDn.ind+11
                                                 ,len(self.Upper_Bank))]
                                                 :
318                 d = ptDn.distance(i)
```

```
                if mn > d:
                    mn = d
                    idex = i.ind

            ptDn.width = mn
            ptDn.vert = self.Upper_Bank[idex]

            if not ptDn.vert:
                print('Failure to Find Vertical Point')
                ptDn.vert = self.Upper_Bank[min(ptDn.ind,len(self.
                                                    Upper_Bank))]
                ptDn.width = ptDn.distance(ptDn.vert)

            if idex == 0:
                ptDn.diagL = None
                ptDn.diagR = self.Upper_Bank[idex+1]
            elif idex != 0 and idex < len(self.Upper_Bank)-1:
                ptDn.diagL = self.Upper_Bank[idex-1]
                ptDn.diagR = self.Upper_Bank[idex+1]
            elif idex == len(self.Upper_Bank)-1:
                ptDn.diagL = self.Upper_Bank[idex-1]
                ptDn.diagR = None

    def WaterDirection(self,window):

        for ptsu in self.Upper_Bank:
            if not ptsu.vert:
                self.NearDB()
                print(f'Attempt to fix the no vertical index for point
                                                    {ptsu.ind}')
            if ptsu.ind-window < 0 or ptsu.vert.ind-window < 0:
                dx1 = self.Upper_Bank[ptsu.ind+1].x-self.Upper_Bank[
                                                    ptsu.ind].x
                dy1 = self.Upper_Bank[ptsu.ind+1].y-self.Upper_Bank[
                                                    ptsu.ind].y
                dx2 = self.Lower_Bank[ptsu.diagR.ind].y-self.Lower_Bank
                                                    [ptsu.vert.ind].x
                dy2 = self.Lower_Bank[ptsu.diagR.ind].y-self.Lower_Bank
                                                    [ptsu.vert.ind].y

            else:

                dx1 = self.Upper_Bank[ptsu.ind].x-self.Upper_Bank[ptsu.
                                                    ind-window].x
                dy1 = self.Upper_Bank[ptsu.ind].y-self.Upper_Bank[ptsu.
                                                    ind-window].y
                dx2 = self.Lower_Bank[ptsu.vert.ind].y-self.Lower_Bank[
                                                    ptsu.vert.ind-
                                                    window].x
                dy2 = self.Lower_Bank[ptsu.vert.ind].y-self.Lower_Bank[
                                                    ptsu.vert.ind-
                                                    window].y

            ax = (dx1+dx2)/2
            ay = (dy1+dy2)/2
            mag = math.sqrt(ax**2+ay**2)
            ptsu.WD = [ax/mag,ay/mag]

```

```
365             for pts in self.Lower_Bank:
366                 if not pts.vert:
367                     self.NearDB()
368                     print(f'Attempt to fix the no vertical index for point
                                                        {pts.ind}')
369                 if pts.ind-window <= 0 or pts.vert.ind-window <= 0:
370                     dx1 = self.Lower_Bank[pts.ind+1].x-self.Lower_Bank[pts.
                                                        ind].x
371                     dy1 = self.Lower_Bank[pts.ind+1].y-self.Lower_Bank[pts.
                                                        ind].y
372                     dx2 = self.Upper_Bank[pts.diagR.ind].y-self.Upper_Bank[
                                                        pts.vert.ind].x
373                     dy2 = self.Upper_Bank[pts.diagR.ind].y-self.Upper_Bank[
                                                        pts.vert.ind].y
374
375                 else:
376
377                     dx1 = self.Lower_Bank[pts.ind].x-self.Lower_Bank[pts.
                                                        ind-window].x
378                     dy1 = self.Lower_Bank[pts.ind].y-self.Lower_Bank[pts.
                                                        ind-window].y
379                     dx2 = self.Upper_Bank[pts.vert.ind].y-self.Upper_Bank[
                                                        pts.vert.ind-window
                                                        ].x
380                     dy2 = self.Upper_Bank[pts.vert.ind].y-self.Upper_Bank[
                                                        pts.vert.ind-window
                                                        ].y
381
382                 ax = (dx1+dx2)/2
383                 ay = (dy1+dy2)/2
384                 mag = math.sqrt(ax**2+ay**2)
385                 pts.WD = [ax/mag,ay/mag]
386
387
388     def Smoothing(self,wind,wdsize):
389
390         xupper = [i.x for i in self.Upper_Bank]
391         yupper = [i.y for i in self.Upper_Bank]
392         xlower = [i.x for i in self.Lower_Bank]
393         ylower = [i.y for i in self.Lower_Bank]
394         xupper = smooth(xupper,wind)
395         yupper = smooth(yupper,wind)
396         xlower = smooth(xlower,wind)
397         ylower = smooth(ylower,wind)
398
399         ups = [point(i,j,0,'upper') for i,j in zip(xupper,yupper)]
400         lows = [point(i,j,0,'lower') for i,j in zip(xlower,ylower)]
401         self.Upper_Bank = None
402         self.Lower_Bank = None
403         self.add_points(np.array(ups),'upper')
404         self.add_points(np.array(lows),'lower')
405         self.indices()
406         self.NearLR()
407         self.NearDB()
408         self.mindist()
409         self.WaterDirection(wdsize)
410
411     def Curves(self,wind):
```

```
412
413         psu = [[i.x,i.y] for i in self.Upper_Bank]
414         psd = [[i.x,i.y] for i in self.Lower_Bank]
415         curvsu = []
416         normsu = []
417
418         for i in range(1,len(psu)-1):
419             curv,norm = curv_normal_menger(psu[i-1:i+2])
420             curvsu += [curv]
421             normsu += [norm]
422
423         normsu = [normsu[0]]+normsu+[normsu[-1]]
424         curvsu,indsu = avgcurvs(curvsu,wind)
425
426         curvsd = []
427         normsd = []
428
429         for i in range(1,len(psd)-1):
430             curv,norm = curv_normal_menger(psd[i-1:i+2])
431             curvsd += [curv]
432             normsd += [norm]
433
434         normsd = [normsd[0]]+normsd+[normsd[-1]]
435         curvsd,indsd = avgcurvs(curvsd,wind)
436
437         cu = pad(curvsu,indsu,len(psu))
438         cd = pad(curvsd,indsd,len(psd))
439
440         assert len(cu) == len(normsu)
441
442         for c in range(len(cu)):
443
444             self.Upper_Bank[c].Norm = normsu[c]
445             self.Upper_Bank[c].curv = cu[c]
446
447         assert len(cd) == len(normsd)
448
449         for e in range(len(cd)):
450
451             self.Lower_Bank[e].Norm = normsd[e]
452             self.Lower_Bank[e].curv = cd[e]
453
454     def Update(self,UP,DOWN,w,wc):
455         self.Upper_Bank = None
456         self.Lower_Bank = None
457         self.add_points(np.array(UP),'upper')
458         self.add_points(np.array(DOWN),'lower')
459         self.indices()
460         self.NearLR()
461         self.NearDB()
462         self.mindist()
463         self.WaterDirection(w)
464         self.Curves(wc)
465
466     def Oxbow_Detect(self,window,bank):
467
468         indexes = []
469         begin = 0
```

```
        if bank == 'upper':
            series = self.Upper_Bank
        else:
            series = self.Lower_Bank

        while begin < len(series):
            ptr = begin + window

            while ptr < len(series):
                d = series[begin].distance(series[ptr])
                if d < series[begin].minW/2 and ptr >= begin + window:
                    if series[begin].vert.ind < series[ptr].vert.ind:
                        indexes += [[begin,ptr,series[begin].vert.ind,
                                                                series[ptr]
                                                                .vert.ind]]
                    elif series[begin].vert.ind > series[ptr].vert.ind:
                        indexes += [[begin,ptr,series[ptr].vert.ind,
                                                                series[
                                                                begin].vert
                                                                .ind]]
                    else:
                        indexes += [[begin,ptr,series[begin].vert.ind,
                                                                series[ptr]
                                                                .vert.ind+1
                                                                ]]
                    begin = ptr+1
                    ptr += 1

                ptr += 1

            begin += 1

        return indexes

    def Oxbow_Snip(self,wind,wd,wc):

        trackingu = len(self.Upper_Bank)
        trackingd = len(self.Lower_Bank)

        diffU = 0
        diffL = 0

        up = [[i.x,i.y] for i in self.Upper_Bank]
        low = [[i.x,i.y] for i in self.Lower_Bank]

        indu = self.Oxbow_Detect(wind,'upper')
        print(indu,'upper')
        for ind in indu:

            if ind:
                #print(f'Oxbow Detected at {ind}')
                init,end,init_L,end_L = ind

                up = up[:init-diffU]+up[end-diffU:]

                if end_L-init_L <= 1:
                    low = low[:init_L-diffL]+low[end_L-diffL:]
                    diffL += end_L - init_L
```

```
520
521                 else:
522                     low = low[:init_L-diffL+1]+low[end_L-diffL-1:]
523                     diffL += end_L - init_L -2
524
525                 diffU += end-init
526                 #if len(up) < tracking:
527                  #   print("Snip Snip")
528                 self.Upper_Bank = None
529                 self.Lower_Bank = None
530                 self.add_points(np.array([point(p[0],p[1],None,'upper')
                                                for p in up]),'
                                               upper')
531                 self.add_points(np.array([point(p[0],p[1],None,'lower')
                                                for p in low]),'
                                               lower')
532                 self.indices()
533                 self.NearLR()
534                 self.NearDB()
535                 self.mindist()
536                 self.WaterDirection(wd)
537                 up = [[i.x,i.y] for i in self.Upper_Bank]
538                 low = [[i.x,i.y] for i in self.Lower_Bank]
539
540
541         indd = self.Oxbow_Detect(wind,'lower')
542         diffU = 0
543         diffL = 0
544         print(indd,'lower')
545         for ind in indd:
546
547             if ind:
548                 init,end,init_U,end_U = ind
549                 if end_U-init_U <= 1:
550                     up = up[:init_U-diffU]+up[end_U-diffU:]
551                     diffU += end_U - init_U
552
553                 else:
554                     up = up[:init_U-diffU+1]+up[end_U-diffU-1:]
555                     diffU += end_U - init_U - 2
556
557                 low = low[:init-diffL]+low[end-diffL:]
558
559                 diffL += end-init
560                 self.Upper_Bank = None
561                 self.Lower_Bank = None
562                 self.add_points(np.array([point(p[0],p[1],None,'upper')
                                                for p in up]),'
                                               upper')
563                 self.add_points(np.array([point(p[0],p[1],None,'lower')
                                                for p in low]),'
                                               lower')
564                 self.indices()
565                 self.NearLR()
566                 self.NearDB()
567                 self.mindist()
568                 self.WaterDirection(wd)
569                 up = [[i.x,i.y] for i in self.Upper_Bank]
```

```
                low = [[i.x,i.y] for i in self.Lower_Bank]

        if len(up) < trackingu and len(indu) > 0:
            print('Succesfull Cut on Upper Bank')
        if len(low) < trackingd and len(indd) > 0:
            print('Succesfull Cut on Lower Bank')

    def stepping(self,cof):

        for i_up in self.Upper_Bank:
            i_up.step(cof)
        for i_down in self.Lower_Bank:
            i_down.step(cof)

    def adjusting(self):
        for i_up in self.Upper_Bank:
            i_up.adjust()
        for i_down in self.Lower_Bank:
            i_down.adjust()
    def plot(self,title,past = False):

        plt.plot([i.x for i in self.Upper_Bank],[i.y for i in self.
                                                 Upper_Bank],label = '
                                                 Modeled Upper Bank')
        plt.plot([i.x for i in self.Lower_Bank],[i.y for i in self.
                                                 Lower_Bank],label = '
                                                 Modeled Lower Bank')
        plt.xlabel('Number of Pixels')
        plt.ylabel('Number of Pixels')
        plt.title(title)
        if past:
            plt.plot(past[0][0],past[0][1],'.-',color = 'black',alpha =
                                                 0.7,label = past[2])
            plt.plot(past[1][0],past[1][1],'.-',color = 'black',alpha =
                                                 0.7,label = past[2])
            plt.legend()


#Set-Up for Actual Running of Model
def Set_Up(X,Y,av,win):
    index = sep(X)
    pu,pd,[xu,yu],[xd,yd] = avg(X,Y,index,av)
    xnu,ynu = smooth(xu,win),smooth(yu,win)
    xnd,ynd = smooth(xd,win),smooth(yd,win)
    pu = [[i,j] for i,j in zip(xnu,ynu)]
    pd = [[i,j] for i,j in zip(xnd,ynd)]
    PTU = []
    PTD = []

    for p in range(len(pu)):
        PTU += [point(pu[p][0],pu[p][1],None,'upper')]
    for pl in range(len(pd)):
        PTD += [point(pd[pl][0],pd[pl][1],None,'lower')]

    R = River()
    R.add_points(np.array(PTU),'upper')
    R.add_points(np.array(PTD),'lower')
    R.indices()
```

```
    R.NearLR()
    R.NearDB()
    R.mindist()
    R.WaterDirection(2)

    return R,pu,pd


Jurua,ptu,ptd = Set_Up(Xa,Ya,3,6)
Jurua.Curves(10)
#Jur = np.array(PU),np.array(PD)


Jurua.plot('Jurua River')


fig,ax = plt.subplots()
time = 100
def change(river,timesteps):

    xu = []
    yu = []
    xd = []
    yd = []
    xu += [[i.x for i in river.Upper_Bank]]
    yu += [[i.y for i in river.Upper_Bank]]
    xd += [[i.x for i in river.Lower_Bank]]
    yd += [[i.y for i in river.Lower_Bank]]

    for t in range(1,timesteps):

        river.stepping(0.5)
        river.adjusting()

        river.Smoothing(2,6)

        river.Oxbow_Snip(20,2,10)

        river.Smoothing(2,6)
        river.Curves(10)
        print(f'{len(river.Upper_Bank)},{len(river.Lower_Bank)}')
        xu += [[pt.x for pt in river.Upper_Bank]]
        yu += [[pt.y for pt in river.Upper_Bank]]
        xd += [[pt.x for pt in river.Lower_Bank]]
        yd += [[pt.y for pt in river.Lower_Bank]]
        river.stepping(5)
    return xu,yu,xd,yd

def animate(stuff): #Animation function
    xnu,ynu,xnd,ynd,n = stuff
    ax.cla() #Clear the previous plot to avoid the last river bank
                                        profile from sticking around
    ax.set_xlim(0,3100) #Make it look nice
    ax.set_ylim(0, 1000)
    ax.plot(xnu,ynu) #Plot the banks
    ax.plot(xnd,ynd) #Plot the banks
    ax.set_xlabel('Number of Pixels')
    ax.set_ylabel('Number of Pixels')
```

```
    print(f'Sim {n} years into the future')

    return ax

Xu,Yu,Xd,Yd = change(Jurua,time)

anim.FuncAnimation(fig, animate, frames=[[Xu[i],Yu[i],Xd[i],Yd[i],i]
                                         for i in range(time)], interval=50)

fig, ax = plt.subplots()
labels = ['vert-1','vert','vert+1','index-1','index','index+1']
coords = [[0,2],[3,2],[6,2],[0,0],[3,0],[6,0]]
ax.plot([3,3],[0,2],linestyle='-',c = 'cornflowerblue',zorder=0)
ax.plot([3,0],[0,2],linestyle='-',c = 'cornflowerblue',zorder=0)
ax.plot([3,6],[0,2],linestyle='-',c = 'cornflowerblue',zorder=0)
ax.plot([3,6],[0,0],linestyle='-',c = 'cornflowerblue',zorder=0)
ax.plot([3,0],[0,0],linestyle='-',c = 'cornflowerblue',zorder=0)
for i in range(6):
    ax.scatter(coords[i][0],coords[i][1],s=5000,c = 'lightgrey',zorder=
                                                1)
    ax.text(coords[i][0],coords[i][1],labels[i],va='center', ha='center
                                                ',zorder=1)
plt.xlim([-1, 7])
plt.ylim([-1, 3])
plt.axis('off')
plt.show()
```

5.2 Appendix - Python Code for Approach 3

```

from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
import geopandas as gpd
from shapely.geometry import LineString
import math

# 2020 Points

# Outside Of River
import openpyxl
wb = openpyxl.load_workbook("RiverPoints.xlsx")
wb.get_sheet_names()
sheet=wb.get_sheet_by_name('Sheet1')
east=np.zeros(64)
north=np.zeros(64)

x=2
i=0
while x<=65:
    east[i] = sheet.cell(row=x,column=5).value
    x+=1
    i+=1
x=2
i=0
while x<=65:
```

```
    north[i] = sheet.cell(row=x,column=6).value
    x+=1
    i+=1
#print(east)
#print(north)

# Inside of River
import openpyxl
wb = openpyxl.load_workbook("RiverPoints.xlsx")
wb.get_sheet_names()
sheet2=wb.get_sheet_by_name('Sheet2')
east2=np.zeros(72)
north2=np.zeros(72)

x=2
i=0
while x<=73:
    east2[i]=(sheet2.cell(row=x,column=5).value)
    x+=1
    i+=1
x=2
i=0
while x<=73:
    north2[i]=(sheet2.cell(row=x,column=6).value)
    x+=1
    i+=1
#plt.scatter(east,north)
#plt.plot(east2,north2,color='black')
east2=np.concatenate((east2[0:65],east2[68:71]))
north2=np.concatenate((north2[0:65],north2[68:71]))
#plt.scatter(east2,north2)
east=np.concatenate((east[0:46],east[48:64]))
north=np.concatenate((north[0:46],north[48:64]))
#plt.scatter(east2,north2)

# Outside Of River
import openpyxl
wb = openpyxl.load_workbook("RiverPoints.xlsx")
wb.get_sheet_names()
sheet=wb.get_sheet_by_name('Sheet3')
east15=np.zeros(85)
north15=np.zeros(85)

x=2
i=0
while x<=86:
    east15[i] = sheet.cell(row=x,column=5).value
    x+=1
    i+=1
x=2
i=0
while x<=86:
    north15[i] = sheet.cell(row=x,column=6).value
    x+=1
    i+=1
#print(east)
#print(north)

```

```
# Inside of River
import openpyxl
wb = openpyxl.load_workbook("RiverPoints.xlsx")
wb.get_sheet_names()
sheet2=wb.get_sheet_by_name('Sheet4')
east215=np.zeros(80)
north215=np.zeros(80)

x=2
i=0
while x<=81:
    east215[i]=(sheet2.cell(row=x,column=5).value)
    x+=1
    i+=1
x=2
i=0
while x<=81:
    north215[i]=(sheet2.cell(row=x,column=6).value)
    x+=1
    i+=1
#print(east2)
#print(north2)

def findCircleh(x1, y1, x2, y2, x3, y3) :
    x12 = x1 - x2;
    x13 = x1 - x3;

    y12 = y1 - y2;
    y13 = y1 - y3;

    y31 = y3 - y1;
    y21 = y2 - y1;

    x31 = x3 - x1;
    x21 = x2 - x1;

    # x1^2 - x3^2
    sx13 = pow(x1, 2) - pow(x3, 2);

    # y1^2 - y3^2
    sy13 = pow(y1, 2) - pow(y3, 2);

    sx21 = pow(x2, 2) - pow(x1, 2);
    sy21 = pow(y2, 2) - pow(y1, 2);

    f = (((sx13) * (x12) + (sy13) *
        (x12) + (sx21) * (x13) +
        (sy21) * (x13)) // (2 *
        ((y31) * (x12) - (y21) * (x13))));

    g = (((sx13) * (y12) + (sy13) * (y12) +
        (sx21) * (y13) + (sy21) * (y13)) //
        (2 * ((x31) * (y12) - (x21) * (y13))));

    c = (-pow(x1, 2) - pow(y1, 2) -
        2 * g * x1 - 2 * f * y1);

    # eqn of circle be x^2 + y^2 + 2*g*x + 2*f*y + c = 0
```

```
    # where centre is (h = -g, k = -f) and
    # radius r as r^2 = h^2 + k^2 - c
    h = -g;
    k = -f;
    sqr_of_r = h * h + k * k - c;

    # r is the radius
    r = round(np.sqrt(sqr_of_r), 5);
    return h

def findCirclek(x1, y1, x2, y2, x3, y3) :
    x12 = x1 - x2;
    x13 = x1 - x3;

    y12 = y1 - y2;
    y13 = y1 - y3;

    y31 = y3 - y1;
    y21 = y2 - y1;

    x31 = x3 - x1;
    x21 = x2 - x1;

    # x1^2 - x3^2
    sx13 = pow(x1, 2) - pow(x3, 2);

    # y1^2 - y3^2
    sy13 = pow(y1, 2) - pow(y3, 2);

    sx21 = pow(x2, 2) - pow(x1, 2);
    sy21 = pow(y2, 2) - pow(y1, 2);

    f = (((sx13) * (x12) + (sy13) *
        (x12) + (sx21) * (x13) +
        (sy21) * (x13)) // (2 *
        ((y31) * (x12) - (y21) * (x13))));

    g = (((sx13) * (y12) + (sy13) * (y12) +
        (sx21) * (y13) + (sy21) * (y13)) //
        (2 * ((x31) * (y12) - (x21) * (y13))));

    c = (-pow(x1, 2) - pow(y1, 2) -
        2 * g * x1 - 2 * f * y1);

    # eqn of circle be x^2 + y^2 + 2*g*x + 2*f*y + c = 0
    # where centre is (h = -g, k = -f) and
    # radius r as r^2 = h^2 + k^2 - c
    h = -g;
    k = -f;
    sqr_of_r = h * h + k * k - c;

    # r is the radius
    r = round(np.sqrt(sqr_of_r), 5);
    return k

def centre(x,y):
    i=0
    centre=np.empty((len(x)-2,2))
```

```
    while(i<len(x)-2):
        x1=x[i]
        x2=x[i+1]
        x3=x[i+2]
        y1=y[i]
        y2=y[i+1]
        y3=y[i+2]
        centre[i,1]=findCirclek(x1, y1, x2, y2, x3, y3)
        centre[i,0]=findCircleh(x1, y1, x2, y2, x3, y3)
        i=i+1
    return centre
def findCircle(x1, y1, x2, y2, x3, y3) :
    x12 = x1 - x2;
    x13 = x1 - x3;

    y12 = y1 - y2;
    y13 = y1 - y3;

    y31 = y3 - y1;
    y21 = y2 - y1;

    x31 = x3 - x1;
    x21 = x2 - x1;

    # x1^2 - x3^2
    sx13 = pow(x1, 2) - pow(x3, 2);

    # y1^2 - y3^2
    sy13 = pow(y1, 2) - pow(y3, 2);

    sx21 = pow(x2, 2) - pow(x1, 2);
    sy21 = pow(y2, 2) - pow(y1, 2);

    f = (((sx13) * (x12) + (sy13) *
        (x12) + (sx21) * (x13) +
        (sy21) * (x13)) // (2 *
        ((y31) * (x12) - (y21) * (x13))));

    g = (((sx13) * (y12) + (sy13) * (y12) +
        (sx21) * (y13) + (sy21) * (y13)) //
        (2 * ((x31) * (y12) - (x21) * (y13))));

    c = (-pow(x1, 2) - pow(y1, 2) -
        2 * g * x1 - 2 * f * y1);

    # eqn of circle be x^2 + y^2 + 2*g*x + 2*f*y + c = 0
    # where centre is (h = -g, k = -f) and
    # radius r as r^2 = h^2 + k^2 - c
    h = -g;
    k = -f;
    sqr_of_r = h * h + k * k - c;

    # r is the radius
    r = round(np.sqrt(sqr_of_r), 5);
    return r


def curve(x,y):
```

```
    i=0
    curve=np.empty(len(x)-2)
    while(i<len(x)-2):
        x1=x[i]
        x2=x[i+1]
        x3=x[i+2]
        y1=y[i]
        y2=y[i+1]
        y3=y[i+1]
        radius=findCircle(x1, y1, x2, y2, x3, y3)
        curve[i]=1/radius
        i+=1
    return curve

def findwidth(x,y,xout,yout):
    xnew=x[1:(len(x)-1)]
    ynew=y[1:(len(y)-1)]
    i=0
    width=np.empty(len(xnew))
    xlin=np.linspace(590000,750000,100)
    while(i<len(xnew)):
        rise=(centre(x,y)[i,1]-ynew[i])/(centre(x,y)[i,0]-xnew[i])
        def f(x):
            return rise*(xlin-xnew[i])+ynew[i]
        line_1=LineString(np.column_stack((xlin,f(xlin))))
        line_2=LineString(np.column_stack((xout,yout)))
        intersection=line_1.intersection(line_2)
        if intersection.geom_type == 'MultiPoint':
            s = gpd.GeoSeries( [intersection])
            exploded_s = s.explode(index_parts=True)
            listarray = []
            for exp in exploded_s:
                listarray.append([exp.x, exp.y])
                nparray = np.array(listarray)
            point=np.array((xnew[i],ynew[i]))
            distances = np.linalg.norm(nparray-point, axis=1)
            min_index = np.argmin(distances)
            width[i]=np.abs(distances[min_index])
        elif intersection.geom_type == 'Point':
            s = gpd.GeoSeries( [intersection])
            exploded_s = s.explode(index_parts=True)
            listarray2 = []
            for exp in exploded_s:
                listarray2.append([exp.x, exp.y])
                nparray2 = np.array(listarray2)
            point=np.array((xnew[i],ynew[i]))
            distance = np.linalg.norm(nparray2-point, axis=1)
            width[i]=np.abs(distance[0])
        i+=1
    i=0
    while(i<len(width)):
        if(math.isnan(width[i]) or width[i]<0):
            width[i]=(width[i-1]+width[i-2])/2
        i+=1
    return width

def smooth(ser,L):
    I_2 = L*(L+1)*(2*L+1)/3
```

```
318     I_4 = I_2*(3*(L**2)+3*L-1)/5
319
320     denom = (2*L+1-(I_2**2)/I_4)
321
322     def a_i(L,i):
323         i_sqr = i**2
324         return (1-i_sqr*I_2/I_4)/denom
325
326     def f_hat(a,X,i,L):
327         res = 0
328         for k in range(-L,L+1):
329             r = a(L,k)*ser[k+i]
330             res += r
331         return res
332
333     weighted_average = [f_hat(a_i,ser,i,L) for i in range(L,len(ser)-L)
                                                ]
334     return weighted_average
335 #smooth(eastmoved,5)
336
337 def River(t):
338     t=t-2020
339     widthinside=findwidth(east,north,east2,north2)
340     widthoutside=findwidth(east2,north2,east,north)
341     eastnew=east[1:len(east)-1]
342     northnew=north[1:len(north)-1]
343     xnorm=eastnew-centre(east,north)[:,0]
344     ynorm=northnew-centre(east,north)[:,1]
345     magnitude=np.sqrt(xnorm**2+ynorm**2)
346     xnormunit=xnorm/magnitude
347     ynormunit=ynorm/magnitude
348     #eastnew=eastnew+t*4388*(np.sqrt(curve(east,north)))*xnormunit
349     #northnew=northnew+t*4388*(np.sqrt(curve(east,north)))*ynormunit
350     eastnew=eastnew+t*20*(np.sqrt(curve(east,north)*widthinside))*
                                                xnormunit
351     northnew=northnew+t*20*(np.sqrt(curve(east,north)*widthinside))*
                                                ynormunit
352     eastnew2=east2[1:len(east2)-1]
353     northnew2=north2[1:len(north2)-1]
354     xnorm2=eastnew2-centre(east2,north2)[:,0]
355     ynorm2=northnew2-centre(east2,north2)[:,1]
356     magnitude2=np.sqrt(xnorm2**2+ynorm2**2)
357     xnormunit2=xnorm2/magnitude2
358     ynormunit2=ynorm2/magnitude2
359     #eastnew2=eastnew2+t*4383*(np.sqrt(curve(east2,north2)))*xnormunit2
360     #northnew2=northnew2+t*4383*(np.sqrt(curve(east2,north2)))*
                                                ynormunit2
361     eastnew2=eastnew2+t*20*(np.sqrt(curve(east2,north2)*widthoutside))*
                                                xnormunit2
362     northnew2=northnew2+t*20*(np.sqrt(curve(east2,north2)*widthoutside)
                                                )*ynormunit2
363     eastmoved=np.empty(len(east))
364     i=0
365     while(i<len(east)):
366         if(i==0):
367             eastmoved[i]=east[0]
368             j=0
369         elif (i==len(east)-1):
```

```
            eastmoved[i]=east[len(east)-1]
        else:
            eastmoved[i]=eastnew[j]
            j+=1
        i+=1
    northmoved=np.empty(len(north))
    i=0
    while(i<len(north)):
        if(i==0):
            northmoved[i]=north[0]
            j=0
        elif (i==len(north)-1):
            northmoved[i]=north[len(north)-1]
        else:
            northmoved[i]=northnew[j]
            j+=1
        i+=1
    eastmoved2=np.empty(len(east2))
    i=0
    while(i<len(east2)):
        if(i==0):
            eastmoved2[i]=east2[0]
            j=0
        elif (i==len(east2)-2):
            eastmoved2[i]=east2[len(east2)-1]
        else:
            eastmoved2[i]=eastnew2[j]
            j+=1
        i+=1
    northmoved2=np.empty(len(north2))
    i=0
    while(i<len(north2)):
        if(i==0):
            northmoved2[i]=north2[0]
            j=0
        elif (i==len(north2)-1):
            northmoved2[i]=north2[len(north2)-1]
            j+=1
        else:
            northmoved2[i]=northnew2[j]
            j+=1
        i+=1
    eastmoved2=smooth(eastmoved2,5)
    northmoved2=smooth(northmoved2,5)
    eastmoved=smooth(eastmoved,5)
    northmoved=smooth(northmoved,5)
    widthnew=findwidth(eastmoved,northmoved,eastmoved2,northmoved2)
    widthnewout=findwidth(eastmoved2,northmoved2,eastmoved,northmoved)
    i=0
    j=1
    count=0
    while(i<len(widthnew)):
        if(widthnew[i]<250):
            eastmovednew=(eastmoved[0:j-1])
            eastmovednew=np.append(eastmovednew,(eastmoved[len(
                                                    eastmoved)-2]+eastmoved
                                                    [j-1])/2)
```

```
425                 eastmovednew=smooth(np.append(eastmovednew,eastmoved[len(
                                                    eastmoved)-2]),1)
426                 eastoxbow=eastmoved[j+3:len(eastmoved)-2]
427                 northmovednew=(northmoved[0:j-1])
428                 northmovednew=np.append(northmovednew,(northmoved[len(
                                                    northmoved)-2]+
                                                    northmoved[j-1])/2)
429                 northmovednew=smooth(np.append(northmovednew,northmoved[len
                                                    (northmoved)-2]),1)
430                 northoxbow=northmoved[j+3:len(northmoved)-2]
431                 plt.figure()
432                 #plt.scatter(eastoxbow,northoxbow)
433                 p=plt.plot(eastmovednew,northmovednew,label='Year %d'%(t+
                                                    2020))
434                 color=p[0].get_color()
435                 count+=1
436                 break
437             i+=1
438             j+=1
439         i=0
440         j=1
441         while(i<len(widthnewout)):
442             if(widthnewout[i]<250):
443                 eastmovednew2=(eastmoved2[0:j])
444                 eastmovednew2=np.append(eastmovednew2,(eastmoved2[len(
                                                    eastmoved2)-2]+
                                                    eastmoved2[len(
                                                    eastmoved)-1])/2)
445                 eastmovednew2=smooth(np.append(eastmovednew2,eastmoved2[len
                                                    (eastmoved2)-2]),1)
446                 eastoxbow2=eastmoved2[j+2:len(eastmoved2)-2]
447                 northmovednew2=northmoved2[0:j]
448                 northmovednew2=np.append(northmovednew2,(northmoved2[len(
                                                    northmoved2)-2]+
                                                    northmoved2[len(
                                                    northmoved2)-1])/2)
449                 northmovednew2=smooth(np.append(northmovednew2,northmoved2[
                                                    len(northmoved2)-2]),1)
450                 northoxbow2=northmoved2[j+2:len(northmoved2)-2]
451                # plt.scatter(eastoxbow2,northoxbow2)
452                 plt.plot(eastmovednew2,northmovednew2,color=color)
453                 count+=1
454                 break
455             i+=1
456             j+=1
457         if (count==0):
458             p = plt.plot(eastmoved2,northmoved2,label='Year %d' %(t+2020))
459             color = p[0].get_color()
460             plt.plot(eastmoved,northmoved,color=color)
461             plt.legend()
462 River(2030)
463 #plt.savefig('NewRiverOx-bow')
464
465 def vals(t,index):
466     widthinside=findwidth(east,north,east2,north2)
467     widthoutside=findwidth(east2,north2,east,north)
468     eastnew=east[1:len(east)-1]
469     northnew=north[1:len(north)-1]
```

```
    xnorm=eastnew-centre(east,north)[:,0]
    ynorm=northnew-centre(east,north)[:,1]
    magnitude=np.sqrt(xnorm**2+ynorm**2)
    xnormunit=xnorm/magnitude
    ynormunit=ynorm/magnitude
    #eastnew=eastnew+t*4388*(np.sqrt(curve(east,north)))*xnormunit
    #northnew=northnew+t*4388*(np.sqrt(curve(east,north)))*ynormunit
    eastnew=eastnew+t*20*(np.sqrt(curve(east,north)*widthinside))*
                                            xnormunit
    northnew=northnew+t*20*(np.sqrt(curve(east,north)*widthinside))*
                                            ynormunit
    eastnew2=east2[1:len(east2)-1]
    northnew2=north2[1:len(north2)-1]
    xnorm2=eastnew2-centre(east2,north2)[:,0]
    ynorm2=northnew2-centre(east2,north2)[:,1]
    magnitude2=np.sqrt(xnorm2**2+ynorm2**2)
    xnormunit2=xnorm2/magnitude2
    ynormunit2=ynorm2/magnitude2
    #eastnew2=eastnew2+t*4383*(np.sqrt(curve(east2,north2)))*xnormunit2
    #northnew2=northnew2+t*4383*(np.sqrt(curve(east2,north2)))*
                                            ynormunit2
    eastnew2=eastnew2+t*20*(np.sqrt(curve(east2,north2)*widthoutside))*
                                            xnormunit2
    northnew2=northnew2+t*20*(np.sqrt(curve(east2,north2)*widthoutside)
                                            )*ynormunit2
    eastmoved=np.empty(len(east))
    i=0
    while(i<len(east)):
        if(i==0):
            eastmoved[i]=east[0]
            j=0
        elif (i==len(east)-1):
            eastmoved[i]=east[len(east)-1]
        else:
            eastmoved[i]=eastnew[j]
            j+=1
        i+=1
    northmoved=np.empty(len(north))
    i=0
    while(i<len(north)):
        if(i==0):
            northmoved[i]=north[0]
            j=0
        elif (i==len(north)-1):
            northmoved[i]=north[len(north)-1]
        else:
            northmoved[i]=northnew[j]
            j+=1
        i+=1
    eastmoved2=np.empty(len(east2))
    i=0
    while(i<len(east2)):
        if(i==0):
            eastmoved2[i]=east2[0]
            j=0
        elif (i==len(east2)-2):
            eastmoved2[i]=east2[len(east2)-1]
        else:
```

```
            eastmoved2[i]=eastnew2[j]
            j+=1
        i+=1
    northmoved2=np.empty(len(north2))
    i=0
    while(i<len(north2)):
        if(i==0):
            northmoved2[i]=north2[0]
            j=0
        elif (i==len(north2)-1):
            northmoved2[i]=north2[len(north2)-1]
            j+=1
        else:
            northmoved2[i]=northnew2[j]
            j+=1
        i+=1
    eastmoved2=smooth(eastmoved2,5)
    northmoved2=smooth(northmoved2,5)
    eastmoved=smooth(eastmoved,5)
    northmoved=smooth(northmoved,5)
    widthnew=findwidth(eastmoved,northmoved,eastmoved2,northmoved2)
    widthnewout=findwidth(eastmoved2,northmoved2,eastmoved,northmoved)
    i=0
    j=1
    count=0
    while(i<len(widthnew)):
        if(widthnew[i]<250):
            eastmovednew=(eastmoved[0:j-1])
            eastmovednew=np.append(eastmovednew,(eastmoved[len(
                                                    eastmoved)-2]+eastmoved
                                                    [j-1])/2)
            eastmovednew=smooth(np.append(eastmovednew,eastmoved[len(
                                                    eastmoved)-2]),1)
            northmovednew=(northmoved[0:j-1])
            northmovednew=np.append(northmovednew,(northmoved[len(
                                                    northmoved)-2]+
                                                    northmoved[j-1])/2)
            northmovednew=smooth(np.append(northmovednew,northmoved[len
                                                    (northmoved)-2]),1)
            count+=1
            if (index==2):
                return eastmovednew
            elif (index==3):
                return northmovednew
            break
        i+=1
        j+=1
    i=0
    j=1
    while(i<len(widthnewout)):
        if(widthnewout[i]<250):
            eastmovednew2=(eastmoved2[0:j])
            eastmovednew2=np.append(eastmovednew2,(eastmoved2[len(
                                                    eastmoved2)-2]+
                                                    eastmoved2[len(
                                                    eastmoved)-1])/2)
            eastmovednew2=smooth(np.append(eastmovednew2,eastmoved2[len
                                                    (eastmoved2)-2]),1)
```

```
            northmovednew2=northmoved2[0:j]
            northmovednew2=np.append(northmovednew2,(northmoved2[len(
                                                    northmoved2)-2]+
                                                    northmoved2[len(
                                                    northmoved2)-1])/2)
            northmovednew2=smooth(np.append(northmovednew2,northmoved2[
                                                    len(northmoved2)-2]),1)
            count+=1
            if(index==0):
                return eastmovednew2
            elif(index==1):
                return northmovednew2
            break
        i+=1
        j+=1
    if(count==0):
        if(index==0):
            return eastmoved2
        elif(index==1):
            return northmoved2
        elif(index==2):
            return eastmoved
        elif (index==3):
            return northmoved

print(north[0])

import random
import numpy as np
import operator
import math
import matplotlib.pyplot as plt
import pandas as pd
from IPython import display

import matplotlib.animation as anim
from IPython.display import HTML
plt.rcParams["animation.html"] = "jshtml"

def change(timesteps):

    fig, ax = plt.subplots(figsize = (6,6))

    ax.set_aspect('equal')

    def animate(n): #Animation function
        while(n<len(timesteps)):
            t=timesteps[n]
            ax.cla() #Clear the previous plot to avoid the last river
                                                    bank profile from
                                                    sticking around
            ax.set_xlim(707000,741000) #Make it look nice
            ax.set_ylim(9690000,9720000)
            p=ax.plot(vals(t,0),vals(t,1),label='Year %d'%(t+2020)) #
                                                    Plot the banks
            color = p[0].get_color()
            ax.plot(vals(t,2),vals(t,3),color=color) #Plot the banks
            plt.legend()
```

```
            plt.title('River Animation')
            return ax
    return anim.FuncAnimation(fig, animate,frames=list(range(0,8)),
                                        interval=500)


timesteps=np.array([0,10,20,30,40,50,60,70])
#animate=change(timesteps)
#animate.save('myanimation.gif')

import random
import numpy as np
import operator
import math
import matplotlib.pyplot as plt
import pandas as pd

import matplotlib.animation as anim
from IPython.display import HTML
plt.rcParams["animation.html"] = "jshtml"

def change(timesteps):

    fig, ax = plt.subplots(figsize = (6,6))

    ax.set_aspect('equal')

    def animate(n): #Animation function
        while(n<len(timesteps)):
            t=timesteps[n]
            ax.cla() #Clear the previous plot to avoid the last river
                                                bank profile from
                                                sticking around
            ax.set_xlim(707000,739500) #Make it look nice
            ax.set_ylim(9690000,9720000)
            p=ax.plot(vals(t,0),vals(t,1),label='Year %d'%(t+2020)) #
                                                Plot the banks
            color = p[0].get_color()
            ax.plot(vals(t,2),vals(t,3),color=color) #Plot the banks
            plt.legend()
            plt.title('River Animation')
            return ax
    return anim.FuncAnimation(fig, animate,frames=list(range(0,7)),
                                        interval=500)


timesteps=np.array([50,60,70,80,90,100,110])
#animate=change(timesteps)
#animate.save('myanimation2.gif')
```