Data Science
Computer Science

Capstone Report - Spring 2021

Web-App for the Visualization and Calculation of Schwarz-Christoffel Conformal Mapping

Andrew Liu,
Zane Fadul

supervised by
Pedro A. S. Salomao, Oliver Marin & Irith Ben-Arroyo Hartman

Abstract

While the Schwarz-Christoffel conformal mapping method has had computational solutions for almost 30 years, there is only one toolkit that is readily available. Because this toolkit is a MATLAB implementation—there is a considerable barrier for those without a MATLAB license and those with low technical experience to use. Our goal is to create a user-friendly web application to act as a supplemental education tool. This report documents the process and findings towards creating a Web App dedicated to the interactive plotting and visualization of polygons implementing Schwarz-Christoffel Mapping. The project is divided into two parallel stages: Numerical Analysis and App Development.

1 Introduction

Conformal mapping[1] is a core concept in complex analysis—a subject of which Andrew and I were not quite familiar with—and even has many applications outside the realm of pure mathematics. The applications in various fields include: physics, fluid dynamics, minimal surfaces, etc. Specifically, the Schwarz-Christoffel transformation method deals with mapping the upper half-plane onto polygons, whether or not the polygon is closed or not depends on implementation. For our purposes, we will focus on any type of polygon so long as it is closed and not complex, meaning vertices should not lead to the polygon's edges crossing each other.

The only known computational toolkits for computing this mapping method exists as a MATLAB implementation written by Toby Driscoll and an even lower level implementation in FORTRAN. These implementations are robust in features and functionality, but are not as accessible to those who do not have some programming knowledge, and to those who do not have a MATLAB license. We hope to create a web-app implementation that visualizes Schwarz-Christoffel Mapping onto a user-given polygon, and to control the origin of said polygon's interior flow lines based on a unit disk.

Initially, we had no foundations in complex analysis, or even analysis, so our first step towards realizing this project was to research the various approaches and steps to solving conformal mapping, Schwarz-Christoffel mapping in particular, and many meetings with our university's mathematics professor Pedro Salomao. First, we explored the basics of the field of complex analysis to gain an understanding of the field we were working in. From there we were guided to a report documenting the basic procedure of computing the Schwarz-Christoffel transformation dating back to the 1990s. With this detailed exploration of the procedure, we then researched the various terms and methods used in the calculation such as the Gauss-Jacobi Quadrature and Newton-Raphson Method. We then used variations of the equations and algorithmic nuances in our implementation. Finally, we delved into the only existing modern implementation of the Schwarz-Christoffel transformation and met with Pedro Salomao and Toby Driscoll to discuss ways to improve our implementation and our subsequent steps to achieving a working algorithm.

It's important to have a more accessible version of a Schwarz-Christoffel mapping tool, because not only is solving the Schwarz-Christoffel integral a lengthy process by hand, it's not feasible to calculate the problem's accessory parameters by hand, hence why the computational method was created. It's also important that this implementation is also more modern to pave the way for the possible implementation of web APIs for other websites that may need to perform this calculation.

2 Related Work

The following sections detail the concepts that serve as the foundation for this project and some related works that we draw from. While there haven't been accessible web implementations of a Schwarz-Christoffel calculator, there are preexisting calculations of the Schwarz-Christoffel transformation which documents the procedure of calculating the Schwarz-Christoffel Integral [2].

2.1 Conformal Mapping

Conformal mapping [1] is a function that maps points on the upper half plane onto a complex plane to preserve angles and orientation, while not necessarily preserving lengths. Put more rigorously, consider an open subset $\mathbf{U} \subset \mathbb{C}$, then $\mathbf{f} : \mathbf{U} \to \mathbb{C}$ is conformal if and only if it is holomorphic, meaning every point is complex differentiable. The Schwarz-Christoffel Transformation is a specific method of Conformal Mapping. A transformation of the Upper Half-Plane onto the interior of a polygon.

2.2 Accessory Parameters

Given a polygon with N vertices, there are $2N + 2$ accessory parameters needed to solve the Schwarz-Christoffel integral, where A is a set of points $a_1, a_2, ..., a_N$ on the real axis ζ and whose image represents the vertices of a polygon, B is a set of interior angles $\alpha_1\pi, \alpha_2\pi, ..., \alpha_N\pi$ of the polygon transformed into exterior angles β_n through the division of π to each α_n, and C_1 and C_2 are constants. [2]. C_1 and C_2 are important because they are part of a linear transformation in relation to the side-length ratios between vertices of the polygon, given in the general form:

$$\mathrm{F}_i(a_1, a_2, ..., a_{N-1}) = I_i - \lambda_i I_l = 0 \text{ for } i \text{ in } N - 2 \text{ when } i \neq l, l + 1$$

where λ_i represents ratio of the distance of two polygonal points in the set of vertices V indexed at i and $i + 1$ with respect to the distance indexed at l and $l + 1$, and where I represents the integral I_i between a_i and a_{i+1}. C_1 and C_2 prior to this stage are undefined. In our case, we will be utilizing a variation of J.M. Chuang et al.'s method of using a system of nonlinear equations and the set Λ of side-length ratios with respect to a polygon's chosen first side λ to solve for these accessory parameters at each point a on the real axis. [3] It is also important to note that the polygon's vertices should be given in counterclockwise order.

2.3 Riemann Mapping Theorem

The Riemann Mapping Theorem states that there exists a conformal mapping f from the open unit disk $\mathbb{D} = \{z : |z| < 1\}$ to any simply connected proper sub-domain D of the complex plane [4, 5]. With this in mind, we not only recognize that a conformal mapping exists onto our bounded polygon, but we now need to be able to calculate it. From here, we look at J.M Chuang et al.'s implementation to see that the Riemann's Mapping Theorem motivates the idea that we can arbitrarily fix certain a values in order to find unique solutions to this system of nonlinear equations. The paper suggests three fixed a values, but this is motivated by the assumption that the polygon is open, giving the fixed a values $a_l = 0$, $a_{l+1} = 1$, and $a_N = \infty$ where l represents the index of the fixed a value in the set A [2] In our case, we will only consider fixing a_l and a_{l+1} since our polygon is closed. We also will set these values to be $a_l = -1$ and $a_{l+1} = 1$, for a reason we will explain later.

2.4 Newton-Raphson Method

In order to calculate solutions to our set of nonlinear equations and locate our accessory parameters in A, we are motivated by the Newton-Raphson Method. This method is an iterative method that slowly updates each parameter a value such that its respective solution in our system of equations starts to approach 0, and gets closer to approximating the zeroes of that real function $F_i(a_1, a_2, ..., a_{N-1}) = I_i - \lambda_i I_l = 0$. We use the Newton-Raphson method with several other parameters to calculate the convergence of these accessory parameters, and to obtain a precise estimation for the relevant polygon's Schwarz-Christoffel parameters.

In more detail, the Newton-Raphson method is conducted by starting off with a guessed point that is relatively close to the zero of a function. From that point you find the tangent line of the function at that point, which you then follow to obtain a new point at the zero of that tangent line bringing you closer to the zero of the function.

Finding the tangent line of the zero of the previous iteration's tangent line, you then repeat this step a given number of iterations until you achieve the desired level of precision. This process, illustrated below, may take many iterations to achieve a high level of accuracy to the real Schwarz-Christoffel transformation integral. This method is the backbone of our algorithm.

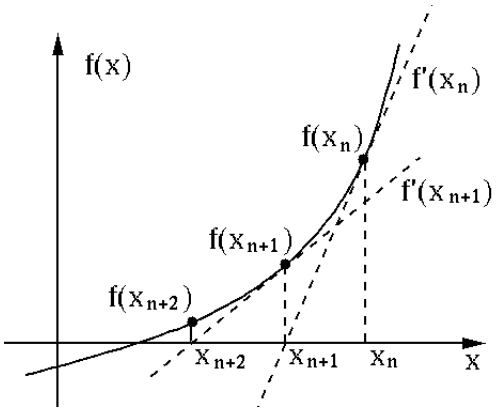

Figure 1: Newton-Raphson Method [6]. Figure is meant to illustrate the process only, variables listed here are not relevant.

2.5 Gauss-Jacobi Quadrature

The Gauss-Jacobi Quadrature is a method of approximating an integral that deals with curves with discontinuities. This is done through the approximation and summation of rectangles under the curve, and require a weight to be applied to the image of a given function in order to determine the integral. We will later use this calculation in the formulation of a Jacobi matrix, its inverse, and then to a column vector representative of the change of a values. Subtracting this from our a values will give us updated values, theoretically closer to our unique solution.

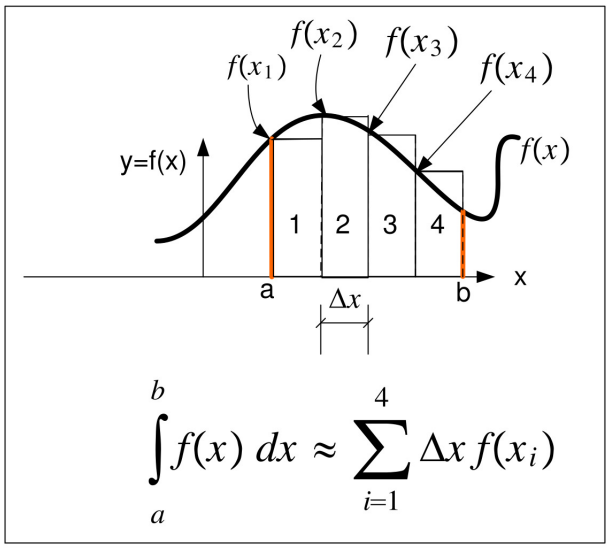

Figure 2: Gaussian Quadrature [7]

2.6 Calculating Constants $C1$ and $C2$

2.6.1 Calculating $C1$

Once we obtain our converged accessory parameters, we can begin to calculate our constants $C1$ and $C2$. After meeting with Pedro Salomao, we were given this formula to calculate $C1$:

$$\text{C1}=\text{r}\cos(\theta) + i\sin(\theta)$$

where,

$$\theta = \eta\pi\beta_i$$

where,

$\eta = \arg(z_i - z_{i+1})$, where z represents the given polygon's vertices represented as complex numbers

where,

$$\text{r= s}_i/I_i, \text{ where } s \text{ is given polygon's side length}$$

In the functions above, i is the index of an arbitrary vertex taken. For our case, we will always choose $i = 1$, so β_i is essentially the first exterior angle of the polygon. Again, $C1$ will be used later in our linear transformation after we've finished applying the Schwarz-Christoffel transformation to our accessory parameters in A.

2.6.2 Calculating $C2$

After calculating $C1$, $C2$ can now be defined as:

$$\text{C2=z}_i + C1 \int_{a_i}^{0} \frac{1}{\Pi_{j=1}^{N-1}(\zeta - a_j)^{\beta_j}}\, dx$$

In this section, we are also consistently setting $i = 1$ just as we did when we calculated $C1$. We are also getting closer to the actual forward mapping using the Schwarz-Christoffel integral.

2.7 Schwarz-Christoffel Transformation

Once we solve for our accessory parameters in A, the following equation is equivalent to that of a conformal mapping onto a polygonal domain:

$$\text{z=C}_1 \int_0^{\zeta} \Pi_{j=1}^{N-1}(\zeta - a_j)^{a_j - 1}\, d\zeta + C_2$$

To get to this step, our algorithm must iterate until the difference between the current and previous iteration's set of a values is smaller than our accepted error value. In our case, we have chosen the accepted error to be $10^{-}3$. Once the a values are below this accepted error, then we will know that the a values have converged to a point to where they approach the solutions of our nonlinear set of equations.

2.8 Toby Driscoll's MATLAB Toolbox

This MATLAB toolbox is the only readily available testing implementation of the Schwarz-Christoffel Mapping. The toolbox provides users with a MATLAB API to draw polygons and apply Schwarz-Christoffel transformations to applied conformal mapping [8]. The toolbox is based on a FORTTRAN package called SCPACK that was developed in the early 1980's by L.N.Trefethren [9][10]. We met with Toby Driscoll, and he was kind enough to point us towards his book, where he discusses more in depth about his MATLAB tool kit, which has many different facets to solve for accessory parameters, and many variations on mapping, such as mapping from a unit disk, mapping from a strip, from a rectangle, etc. He also talks about the factors that may lead to failure when using this algorithm, such as crowding and branch cuts. We will be discussing crowding later in this paper.

3 Solution

In this section, we'll cover the architecture reasoning, process, and struggles encountered throughout the course of this project.

3.1 Tech Stack

This project is a web app implementation so we needed two main components for our stack, a front end framework or markdown language of choice, and a web server to handle calculations and to render our views on. Because this web app is calculation heavy, we decided to go with a Django Python back end for our web server to make use of Python's extensive computing libraries such as SciPy, NumPy, and etc. And for our front end we decided to use the ReactJS library for its state driven rendering system instead of a web framework such as ExpressJS or AngularJS, as we already decided on Django as our web server.

Now that we had decided upon both the back end and front end, there were difficulties in getting the two systems to work together properly. The main difficulties came from Django's built in view rendering system which directly conflicted with React. In order to get Django to render the React views, we had to override the Django view rendering system with React webpacks.

3.2 Process

3.2.1 Polygon Object

We wanted to allow the user to be able to see many different aspects of the polygon they are plotting, and we also rely on quite a few properties of the polygon to run our algorithm to find accessory parameters, such as side lengths, vertex information, and angle calculations. We decided it made more sense to create a separate custom Polygon object that could calculate all of these things before being inserted into our algorithm. Given a 1x1 square whose first vertex start at the origin, these are the properties our Polygon object calculates upon instantiating:

VERTICES
A: 0j, B: (1+0j), C: (1+1j), D: 1j
LINES
a: A: 0j <-> B: (1+0j) line length: 1.0 line slope: 0.0
b: B: (1+0j) <-> C: (1+1j) line length: 1.0 line slope: nan
c: C: (1+1j) <-> D: 1j line length: 1.0 line slope: -0.0
d: D: 1j <-> A: 0j line length: 1.0 line slope: nan
INTERIOR ANGLES
ABC: 1.5708 BCD: 1.5708 CDA: 1.5708 DAB: 1.5708
EXTERIOR ANGLES
ABC: 1.5708 BCD: 1.5708 CDA: 1.5708 DAB: 1.5708

It's worth noting at this point that Python expresses complex numbers with the imaginary constant j, not i.

3.2.2 Algorithm Initialization

The mapping of the upper half plane onto a bounded polygonal domain uses the Schwarz-Christoffel Integral. The integral itself, as well as the the mapping's linear transformations require the accessory parameters in order to be calculated. Our main algorithm is meant to find these a values, and from there we can calculate the other two parameters $C1$ and $C2$, and then calculate the forward mapping and flow lines determined by a complex point on the open unit disk. Within this explanation, details such as how we index elements will be expressed as they would be in a mathematical explanation. So instead of the first element of a list being indexed

at 0 as it would in code, we will continue to express it as being indexed at 1, as that is consistent with the algorithm mathematically.

Our initial algorithm lives in a python class called SchwarzChristoffel, which takes a list of vertices. On initialization, a Polygon object is created from the vertices, and useful properties from that object are extracted such as our set of external angles B and set of length ratios with respect to the distance between our two fixed z points, which we refer to as Λ. At this point we also default $C1 = 1$ and $C2 = 0$. Initializing also causes the algorithms initial approximation of a values. Throughout the entire duration of the algorithm, $l = 1$, and $a_l = -1, a_{l+1} = 1$. The rest of the a values are defaulted to increment by ones starting from a_{l+2} until a_N.

The reason we set our fixed a values to -1 and 1 is due to the Gauss-Jacobi roots method that we are using from SciPy, which assumes an integral boundary from -1 to 1, this differs a bit from the original algorithm which is from 0 to 1, but the result is essentially the same, since we are just using this roots method to get weights and points before approximating the integral ourselves.

Before going into our main algorithm loop, we also initialize a list of our nonlinear equations $I_i - \lambda_{i-1} I_l$, we call this list F.

3.2.3 Algorithm Main Loop

The main loop is broken by two conditions, we'll first talk about our standard case: when the acceptable error is met. We place this condition in a function called paramsValidated() which takes the current a values, and the previous calculated a values. The loop continues while the parameters have not been validated. Because our algorithm does not start with previously calculated a values, the function still returns false.

We start our loop by first defining our I values given our current a values. Our next step is to calculate our list I. using the following formula:

$$\mathrm{I}_i = \int_{a_{i-1}}^{a_i} (\tfrac{a_i - a_{i-1}}{2})^{1-\beta_{i-1}-\beta_i} \prod_{j=1}^{N-1, j\neq i-1, j\neq i} \frac{1}{(\frac{a_i - a_{i-1}}{2} x + \frac{a_i + a_{i-1}}{2} - a_j)^{\beta_j}} dx$$

This expression would then be passed into our gaussJabobiQuad() function, which takes a function of x, and the two exterior angles β_{i-1} and β_i, so as to solve for jacobi roots using the weight function $(1-x)^{-\beta_i}(1+x)^{-\beta_{i-1}}$. Because the function being passed into gaussJacobiQuad() can only be a function of one variable x, we had to take a functional programming approach to create a function that given x, could get the product of the inner expressions $\frac{1}{(\frac{a_i - a_{i-1}}{2} x + \frac{a_i + a_{i-1}}{2} - a_j)^{\beta_j}}$, which also depend on the conditions $j \neq i-1, j \neq i$ One cannot multiply expressions in Python, so we took a recursive approach to have each term enter a list of terms, and when our auxiliary function is called, all the terms would recursively be multiplied as they are passed in x from our most shallow function, gaussJacobiQuad().

Once we have our newly calculated list I, we can solve the equations in F, and receive a local version $[F]$ that again represents the difference of side length ratios with respect to this local calculation of I. We represent $[F]$ as a column vector.

From here, we can calculate our Jacobi Matrix $[J]$, which is equivalent to:

$$[\mathrm{J}] = [\delta S] - [\Lambda][\delta L]^T$$

Here, $[\delta S]$ represents a matrix of Side Length first order derivatives $\frac{\delta I_i}{\delta a_k}$ for i ranging from $[1, l) \cup (l, N-1)$, and k ranging from $[1, l) \cup (l+1, N)$. $[\Lambda]$ simply represents our λ values represented as a column vector, and $[\delta L]^T$ represents the transposition of a column vector of Side Length first order derivatives, this time in the form $\frac{\delta I_l}{\delta a_k}$.

The first order derivatives can be approximated by the five-point Lagrange differential formula [2]:

$$\delta I_i / \delta a_k \approx [I_i(a_k - 2h) - 8I_i(a_k - h) + 8I_i(a_k + 2h)]/12h$$

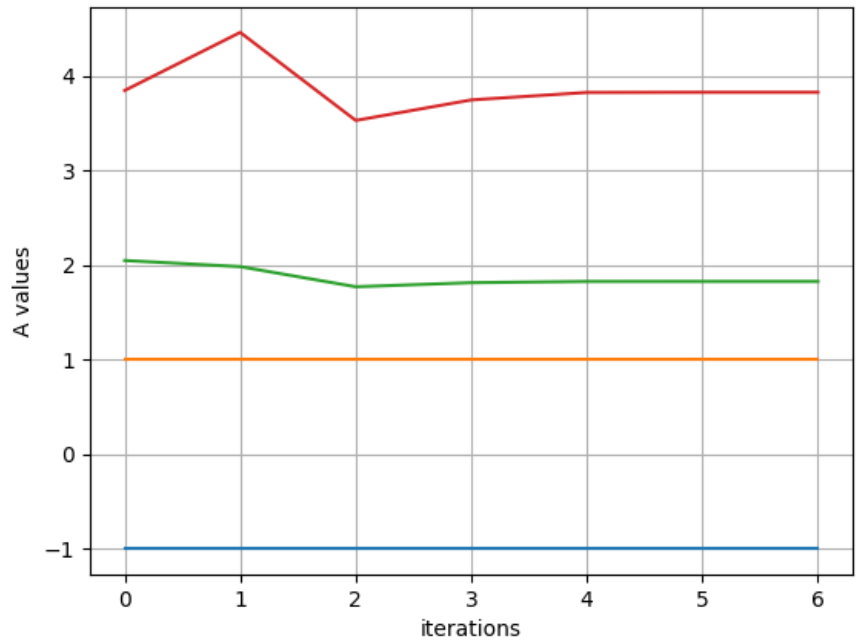

Figure 3: We can see that the a values converge after 6 iterations.

where

$$\mathrm{I}_i(a_k \pm nh) = I_i(a_1, a_2, ..., a_{N-1})|_{a_i = a_i \pm nh}, n = 1, 2$$

h is defaulted to 0.01 here. This is to slightly alter each a value during calculation. From there, we create a column vector $[A]$, which will represent our a next iteration's a values excluding a_l and a_{l+1}. We can calculate our new a values by with the following [2]:

$$[\mathrm{A}]^{(t+1)} = [A]^{(t)} - [J]^{-1(t)}[F]^{(t)}$$

Here, t simply refers to the number of iterations. At this point, $[A]^{(t)}$ will be labeled as our previous a values, and once we add back a_l and a_{l+1} to their respective indices in $[A]^{(t+1)}$, $[A]^{(t+1)}$ will now represent our current a values as we prepare for our next iteration.

By now, we have a previous set of a values, and a current set. As the next loop starts, we use paramsValidated() to check whether or not each a in $[A]^{(t+1)} - [A]^{(t)}$ has a difference less than the accepted error. We continue the algorithm.

We can see a sample case for the same square from earlier in Figure 3. After convergence, we perform the aforementioned steps above to calculate $C1$, $C2$, and our Schwarz-Christoffel integral to produce our mapping in Figure 4.

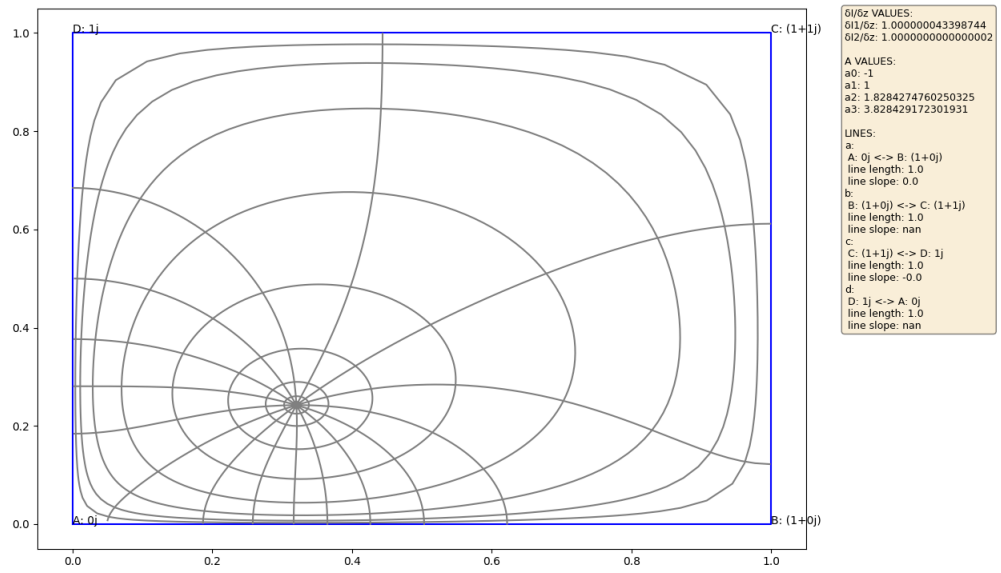

Figure 4: Simple 1x1 square's flow lines after the algorithm

3.3 API calls

In order to render new data to the front end and add user interaction in calculations such as calculating flow lines, we need to make backend API calls from the User Interactions. For example, we have the following main APIs:

- data_transfer
  Creates a polygon object from user input vertices, returns polygon data such as: Line Lengths, Line Slopes, Interior Angles, and Exterior Angles.

- get_sc
  Takes vertices from user input polygon, creates an instance of the SchwarzChristoffel class which calculates the accessory parameters and returns the following data: lambda, Is, and I ratios.

- get_flows
  References the instance of SchwarzChristoffel created in get_sc to calculate the conformal mapping flow lines on the polygon.

While both the get_sc and get_flows APIs both reference the same class of SwarzChristoffel, theyre separated to remove unnecessary work. This allows us to calculate lamda, Is, and I ratios once, and generate flow lines from different alphas as a result from user input.

To call back end APIs from the front end, we use Axios and Django built in http requests. Each of the APIs are POST requests that send the relevant data to the API domain and will return the API calculation.

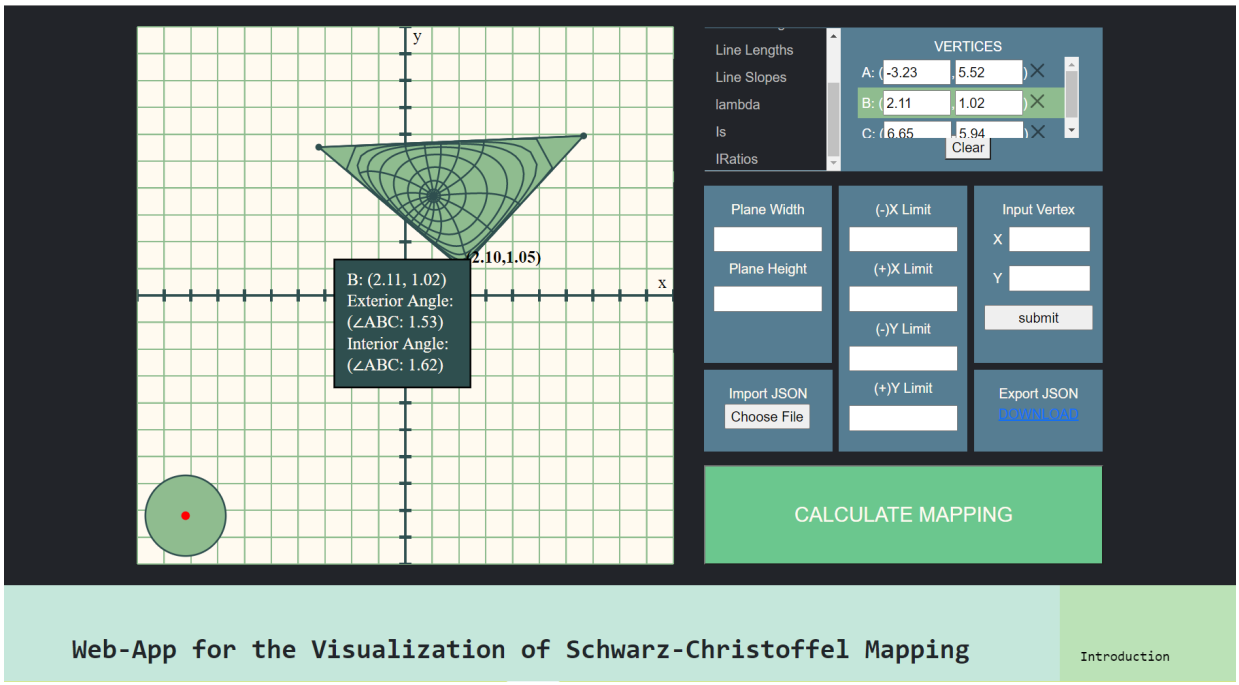

Figure 5: Web App user interface and basic use case

3.4 Web App

While the Web App's main function is to plot and visualize the mapping resulting from the aforementioned APIs, there are several additional features made to serve as the foundation in making the user's experience with the app more dynamic and versatile.

The following summarize the functionalities of the main components making up the Web App:

3.4.1 Coordinate Plane

To start off, the Web App revolves around a coordinate plane on which the user can plot a custom polygon from which to calculate the visualization. This coordinate plane was initially written in p5.js, as it is a versatile tool for graphics rendering in javascript. However, p5.js doesn't work well with ReactJS elements so we decided to transition into the HTML SVG elements for increased customization options and dynamic interactions.

The coordinate plane is setup in a dynamic fashion, meaning that various aspects of the coordinate plane can be modified by the user to their specific preferences. These dynamic options are:

- Coordinate Plane Dimensions
  Modify the specific pixel dimensions (with a default of 630x630 pixels) of the coordinate plane SVG, while maintaining the limits of the coordinate plane.

- Coordinate Plane Limits
  Modify the X and Y limits of the coordinate plane by adjusting the -X Limit (-10), X Limit (10), -Y Limit (-10), Y Limit (10).

3.4.2 Vertices

After the coordinate plane, the next step was plotting and recording vertices. Each of the vertices are recorded in an info/ data log from which the user can manipulate vertex coordinates and

remove specific vertices as they see fit. Upon plotting three or more vertices, we connect the vertices into a polygon and then call the data_transfer API to calculate polygon data such as line slopes, lengths and its angles and display them on the data log.

As plotting the vertices is complete, we next implement a tool tip which renders upon hovering a vertex which displayed the vertex's name and relevant data. This tool tip is also unidirectional linked with the data log to highlight the hovered element.

When hovering a vertex, the user is provided with a more versatile way of adjusting vertex coordinates by dragging and dropping the vertex (as opposed to manual coordinate adjustments in the data log).

Now that the vertices can be easily plotted and modified, we needed to implement a way to insert vertices into the already existing polygon. That way the user does not need to create a new polygon from scratch to insert vertices. To achieve this, we plot an additional interactive line which the user can hover over and plot a vertex to insert. Because the vertex must be pixel perfect plotting to be faithful to the existing polygon, we do a simple calculation that plots the new vertex at the x of the interactive line's y following its slope.
If the inaccuracy of plotting the vertices by hand is a concern, there is an input form on the User Interface that allows the user to plot vertices with specific coordinates without having to rely on the manual plotting.

Finally the user is able to both import and export a JSON file which contains a list of vertices. Upon importing a JSON file, the Web App will automatically plot the vertices, create a polygon, and make a hit to the data_transfer API.

3.4.3 Flow Lines

All the other mentioned components of the Web App served as a foundation for the main function of visualizing the Schwarz-Christoffel Mapping.

To calculate the visualization of the mapping upon the plotted polygon, the user simply needs to hit the 'CALCULATE MAPPING' button. Upon hitting the calculate button, the Web App renders the user interface unit disk and makes a hit to the get_sc API which calculates and returns the lambda, Is, and I ratios necessary for getting the flow lines.

After hitting get_sc and getting the necessary parameters, the Web App automatically hits the get_flows API taking the given unit disk origin (default 0+0j). The get_flows API then generates the flow line coordinates in the form of a nested array of strings of coordinates with each outer array being an array flow line coordinates. From this array of coordinates, we process them in the generateFlow function to create a path element to visualize the flow line.

The reason why the get_sc and get_flows APIs are separated is to improve performance when accepting user interactions. The accessory parameters from the get_sc API only need to be calculated and stored once per polygon. By separating the APIs, when adjusting the origin of the unit disk only the get_flows API needs to be hit, significantly reducing the computation time of modifying the flow lines.

3.4.4 Description and Summaries

Following the main visualization component, we include a section of the Web App dedicated to a short summary and description of our project, this report, and special credits.

4 Results and Discussion

4.1 Crowding

Our main issues begin during the initial programming of the algorithm. There are many steps for just the convergence part of the algorithm, and this quickly made the code base quite dense. Bugs

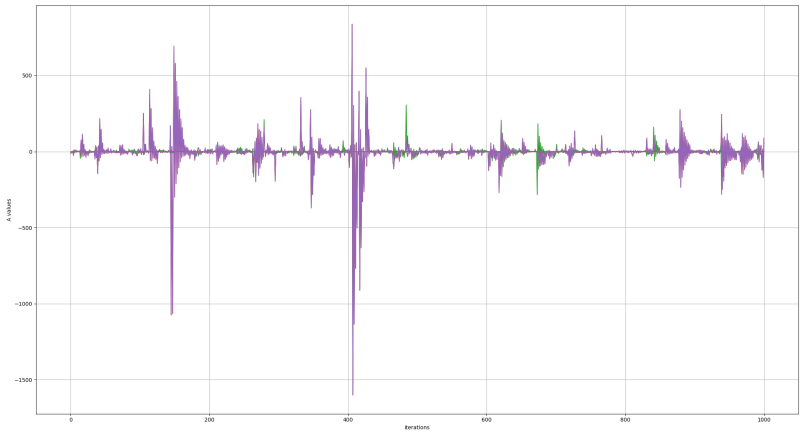

Figure 6: Here is an example of a values not converging, the max number of iterations cap at 1000.

would occur and we would see that our a values just would not converge. After countless meetings with Professor Salomao, we would find smaller bugs to tweak such as index mistakes, recursive logic mistakes, etc. It took us collectively a few weeks before we were set on our code working and our a values converging. The issue though, was that given the shape, the a values would not always converge. You can see in Figure 4 how over time, the a values seemed to oscillate and have huge spikes. Even after these spikes, while the values continue converging again, a small handful of a values would continue to spike again as they converged. We weren't quite sure how to move forward with this. Throughout the algorithm, the following condition must always be satisfied [2]:

$$\mathrm{a}_1 < a_2 < ... < a_{N-1} < a_N$$

However, there were some cases where a values would spike, and throughout the calculation the condition was not maintained. We thought that this could potentially be because we were altering the a values too much during each iteration, so instead we would only subtract 10^{-1} of $[J]^{-1(t)}[F]^{(t)}$ before subtracting it from $[A]^t$. This alleviated the issue for some cases, but we still ran into the same issue for others. We also tried increasing our accepted error from 10^{-5} to 10^{-3}, which seemed to help, but certain cases still persisted.

We also examined that in cases where the algorithm was slowly converging, that the distance between a values had a very high variance. This wasn't necessarily incorrect though. We also tested the validity of our converged a values by testing what we've called our I ratios and their equivalence to the polygon's λ values. I ratios simply are defined as a set of I values $\frac{I_i}{I_l}$ for i ranging from $[1, l) \cup (l, N-1)$. Everything checked out, so while a values occasionally had these issues, we came to a soft conclusion that this variance was potentially causing issues during calculation time.

Later, we met with Toby Driscoll, author of the MATLAB tool kit and told him the problem we were having. He said that this distribution of a values was known as crowding, and it led to issues because of how far away a values really would converge towards. As said in his book about Schwarz-Christoffel mapping, long and thin aspect ratios tend to succumb to crowding, as the angles between certain pairs of curves on the unit disk become exponentially small. This paired with numerical round off mistakes poses an issue that cannot be so easily solved [11].

We attempted to explore different possibilities in order to more precisely calculate exponents to avoid rounding mistakes, such as calculating certain expressions in pure C, but due to time

constraints, we were not able to fully solve this issue.

Another quick fix we found was that if the code ran into any errors in terms of violating the inequality of a values, or exceeding a certain number of iterations, the algorithm would reset with the list of vertices rotated counterclockwise by 1. This is not to say that the shape itself would rotate, but that origin vertex would change to the vertex right before it. This was because our assigned $l, l+1$ indices never change, and we thought that the distribution of a points might not run into the same issue. In some cases it did work, but we should have instead allowed the $l.l+1$ indices to move instead.

Again, because of time constraints, this fix was not attainable.

4.2 Leaking

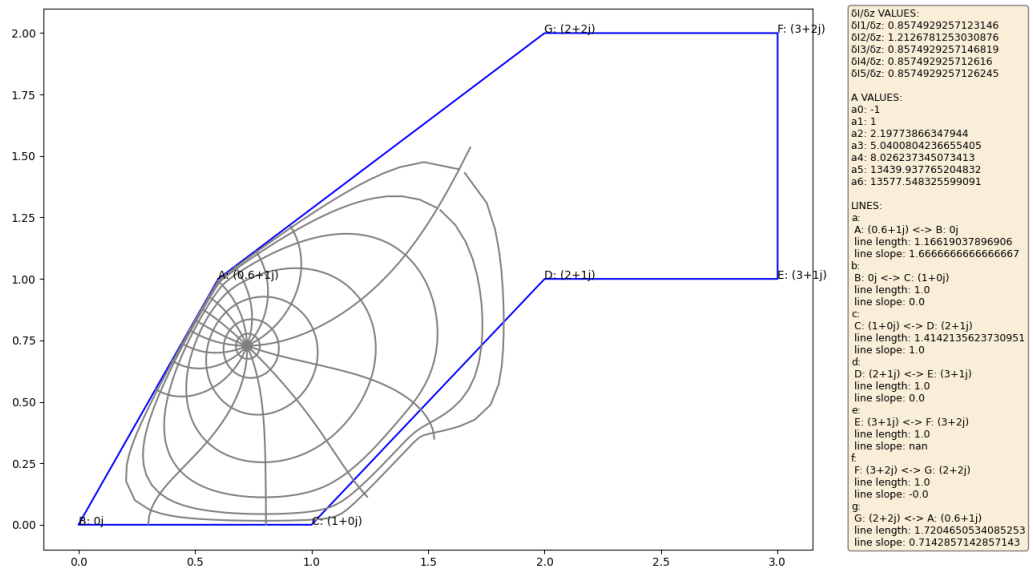

Figure 7:

This phenomenon that we called leaking occurred when graphing flow lines for certain polygons. Even by running a debugger, there were so many numbers that it was nearly impossible to tell where this error was coming from. After combing through the algorithm again, we realized it was a silly recursive logic mistake.

4.3 Web Hosting

Professor Salomao and we all wanted to have this tool hosted online so that it truly could be accessible. As we write this paper, we still have not received a response from the school's IT department in terms of helping us host this web application. We are planning to look towards alternative options for web hosting such as Amazon Web Services or Go Daddy to host our sites on our own.

5 Conclusion

Our web application serves as both a calculator and visualization tool for the Schwarz Christoffel Conformal Mapping method. The user can plot a polygon, and our python backend will calculate the accessory parameters and flow lines to be graphed. There are still quite a few cases where

accessory parameters do not converge, or crowding occurs. If we had more time, we would go back and alter the algorithm to be a bit more flexible in the way it handles corner cases. We would also try to make our web interface a bit more sleek, but we feel that as it stands right now, it gets the job done, and is certainly a more accessible option for this visualization.

We will be pushing our code publicly, so that other users can improve our code base and improve the web tool for everyone's benefit.

6 Acknowledgements

We the authors would like to extend their greatest appreciations to Professor Pedro A. S. Salomao. Without his willingness to meet with us and teach us even when we thought we were hopeless speaks volumes about his teaching ability and desire to truly help his students learn. He is a remarkable professor and mathematician. We also would like to thank Toby Driscoll for taking time out of his schedule to meet with undergraduates who go to a completely different university, and for pointing us to his book for further research. Last but not least, we would like to thank Yinghao Tang, and our peers in room 710 for your emotional support throughout this crazy semester.

References

[1] L. Bieberbach, "Conformal mapping," 1964.

[2] C. H. J.M. Chuang, Q.Y. Gui, "Numerical computation of schwarz-christoffel transformation for simply connected unbounded domain," 1993.

[3] L. Kantorovich and V. Kryiov, "Approximate methods of higher analysis (interscience, noordhoff) 521-542." 1958.

[4] R. M. Porter, "History and recent developments in techniques for numerical conformal mapping. proceedings of the international workshop on quasiconformal mappings and their applications," 2005.

[5] A. Waugh, "The riemann mapping theorem." [Online]. Available: https://sites.math.washington.edu/~morrow/336_18/papers18/alex.pdf

[6] R. Wang, 2015. [Online]. Available: http://fourier.eng.hmc.edu/e176/lectures/NM/node20.html

[7] N. M. Abassi, "Review of gaussian quadrature method," 2006. [Online]. Available: http://www.12000.org/my_courses/UCI_COURSES/CREDIT_COURSES/spring_2006/spring_MAE_207/other_class_documents/Nasser_Abbasi/numerical_integration/numIntegration.htm

[8] T. Driscoll, "Schwarz–christoffel toolbox user's guide." [Online]. Available: https://tobydriscoll.net/project/sc-toolbox/guide.pdf

[9] L. N. Trefethen, "Numerical computation of the schwarz–christoffel transformation siam j. sci. stat. comput., 1, pp. 82–102." 1980.

[10] "Scpack user's guide. mit numerical analysis report 89-2," 1989.

[11] T. Driscoll and L. N. Trefethen, "Schwarz-christoffel mapping," 2002.