Visualisation of complex multi-valued functions

Alan Ip

BSc in Mathematics and Computing
University of Bath, 2010
Visualisation of complex multi-valued functions

Submitted by: Alan Ip

Copyright

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Mathematical Sciences. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

Signed:
Abstract

In mathematical analysis, the well-definition of a function is commonly seen as a formality, and single-valuedness is usually enforced by adjusting other definitions. For example, the square root function conventionally returns the positive root only, whereas our intuitive view of an inverse of a squared number permits both positive and negative solutions.

This project aims to implement a method for plotting all the possible values of a (complex) multi-valued function simultaneously.
Acknowledgements

My thanks to Dr Russell Bradford, my project supervisor, for his guidance and support throughout.
## Contents

1 Introduction ................................. 1

1.1 Riemann surfaces ......................... 1

1.2 Domain colouring ......................... 2

2 Literature Survey ........................... 3

2.1 Multifunction representation .......... 3

2.2 Multifunctions as inverses .......... 4

2.3 Partitioning \( \mathbb{C} \) .................. 4

2.3.1 Branch cuts ......................... 5

2.3.2 Branch points ....................... 6

2.4 Parameterisation of branch sectors . 6

2.4.1 Solution of an ODE ................ 7

2.4.2 Alternate branch functions .......... 7

2.5 Faithfulness .............................. 8

3 Requirements .............................. 9
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Main objectives</td>
<td>9</td>
</tr>
<tr>
<td>3.2 Limitations</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Functional requirements</td>
<td>10</td>
</tr>
<tr>
<td>3.4 Non-functional requirements</td>
<td>11</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>12</td>
</tr>
<tr>
<td>4.1 Branch cut partitioning</td>
<td>12</td>
</tr>
<tr>
<td>4.1.1 Intervals in $\mathbb{R}$</td>
<td>12</td>
</tr>
<tr>
<td>4.1.2 Naïve branch cuts</td>
<td>13</td>
</tr>
<tr>
<td>4.1.3 Branch cut simplification</td>
<td>15</td>
</tr>
<tr>
<td>4.1.4 Removable branch cuts</td>
<td>17</td>
</tr>
<tr>
<td>4.1.5 Plotting branch cuts</td>
<td>20</td>
</tr>
<tr>
<td>4.1.6 Possible further development</td>
<td>20</td>
</tr>
<tr>
<td>4.2 Branch point partitioning</td>
<td>21</td>
</tr>
<tr>
<td>4.2.1 Algebraic branch points</td>
<td>21</td>
</tr>
<tr>
<td>4.2.2 Logarithmic branch points</td>
<td>23</td>
</tr>
<tr>
<td>4.2.3 The infinity branch point</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4 Partitioning $\mathbb{C}$ by points</td>
<td>27</td>
</tr>
<tr>
<td>4.3 Branch sector parameterisation</td>
<td>29</td>
</tr>
<tr>
<td>4.3.1 ODE construction</td>
<td>29</td>
</tr>
<tr>
<td>4.3.2 Initial conditions</td>
<td>30</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Plots of the principal branch of $\ln(z)$ ........................................... 5

4.1 Examples of plane partitioning by curve extensions .............................. 22

4.2 Arrangement of $P(w, z)$ makes a difference .................................. 24

4.3 Plane partitioning by a given set of points ....................................... 28

4.4 Smart partitioning introduces annuli discontinuities ......................... 29

4.5 Infinite branches are capped by a hard-limit .................................. 30

A.1 $F(w, z) = w^2 - z$ ........................................................................... 37

A.2 $F(w, z) = w^3 - z$ ........................................................................... 37

A.3 $F(w, z) = w^4 - z$ ........................................................................... 38

A.4 $F(w, z) = \sin(w) - z$ .................................................................... 38

A.5 $F(w, z) = \tan(w) - z$ .................................................................... 38

A.6 $F(w, z) = \csc(w) - z$ ..................................................................... 39

A.7 $F(w, z) = \cot(w) - z$ ..................................................................... 39

A.8 $F(w, z) = \cosh(w) - z$ ................................................................... 39
LIST OF FIGURES

A.9 \( F(w, z) = \exp(w) - z \) \hspace{1cm} 40

A.10 \( F(w, z) = \ln(w) - z \) \hspace{1cm} 40

A.11 \( F(w, z) = w^3 - w - z \) \hspace{1cm} 40
Chapter 1

Introduction

Much of elementary mathematical analysis is formed upon the basis of single-valued functions, which can be sufficient for explaining particular concepts, but inappropriate for others. The restrictive nature of single-valued functions deters those without a strong background in specific areas of abstract mathematics (involving advanced topics such as topology and algebraic geometry) from exploring further possibilities that would otherwise follow naturally from basic theory. An early appreciation of multi-valued functions frees us from these bounds, with the immediate bonus of yielding insightful results in standard theories such as complex analysis.

With the general aim of increasing awareness of multi-valued functions, the primary objective of this project is to provide a method for intuitively representing a multi-valued function in a graphical manner — more precisely, to develop a software utility for plotting complex-valued elementary functions. The two most common approaches to visualising complex functions are through Riemann surfaces and domain colouring, of which the former will be chosen for development, but both discussed here for completeness.

1.1 Riemann surfaces

In essence, a Riemann surface is rendered as an object in 3D space that may take more than one $z$ value for particular $(x, y)$ coordinates, i.e. at a multi-valued point. To construct the Riemann surface, the range of the graph is split into smaller ranges so that the function on each sub-range, or branch, is single-valued; each branch is then individually rendered, and finally pieced back together to form the full object.
An advantage of employing Riemann surfaces is that they should be familiar and immediately recognisable by the user, since the input and output points of the function are correlated spatially, somewhat reminiscent of the standard graph of a real-valued function. A disadvantage to this solution is that proof of one-to-one correspondence (see Faithfulness) might be required to ensure that the function is being represented by the correct surface.

1.2 Domain colouring

In contrast with a 3D-projection of the function, domain colouring (5) accepts four dimensional data more naturally by exhibiting the range of the function (a subset of $\mathbb{C}$) in its own complex plane, which entirely eliminates the need to prove the faithfulness of a function and its Riemann surface.

In order to display the map itself, an image is designated (usually a colour wheel of some sort) as a reference on the $z$-plane, and the transformed image (the function applied as a transformation) is painted onto the $w$-plane. Multi-valued functions require a number of $w$-planes for a complete representation of the full range of its values.

Although domain colouring is a technically sound approach, it is less obvious and less intuitive to the layperson than the Riemann surface visualisation method.
Chapter 2

Literature Survey

In order to avoid unnecessary repetition of work, and for a better understanding of the task at hand, a little background research was undertaken on the computational construction of Riemann surfaces, together with relevant supporting theory, as well as an appraisal of current implementations in various computer algebra systems.

2.1 Multifunction representation

The interpretation of a multi-valued function raises several issues, especially when applied in the field of computer algebra. For a human performing integration between two surfaces, for example, we might be able to manually infer which of the multiple values (of a multifunction at a particular point) is more appropriate to use in the calculation, using a little common sense or by applying some of the restrictions from the environment if given. However, for an unaided computer system, this information might not be readily available, and it would be forced to continue with ambiguity or, worse, inaccuracy.

In a similar vein, with the convention of assuming that CAS functions output single values as results (in order to feed them back into other functions as input), we are set with the fundamentally flawed task of deciding which of the multiple values is ‘correct’ for proceeding. Although there have been ways devised to achieve this (most commonly by picking the principal value, the sole qualifying value lying on the principal branch, both of which are defined merely by convention anyway), the point is that all such values are correct because they all satisfy the conditions, and so disregarding any of the values would result in a misrepresentation of the multifunction.
This automatic projection of multiple values down to a single value can be inappropriate at times; for example, in evaluating expressions, familiar simplification rules such as

\[ \ln(z_1 \cdot z_2) = \ln(z_1) + \ln(z_2) \]
\[ (z_1 \cdot z_2)^a = z_1^a \cdot z_2^a \]

are lost when restricted to a particular branch of the logarithm, and so mechanisms such as the **unwinding number** [2], or the **unln** function [11], have previously been proposed to remedy this: the offset(s) between the multiple values are absorbed into an auxiliary function. However, this undoubtably introduces an extra level of complexity as a side effect.

Although the (mis)representation of a multifunction in its computational form is an important issue worth discussing, it primarily concerns the fields of symbolic manipulation and algebraic simplification. For our purposes, emphasis is instead placed more on the practicalities of graph generation rather than the theoretical aspects.

### 2.2 Multifunctions as inverses

One might notice that some of the standard examples of multifunctions, such as the square root function and the logarithm function, have inverses that are firstly non-injective but, more importantly, well-defined: they are single-valued. In such cases, it is actually possible to exploit the existence of this ‘nicer’ inverse to generate Riemann surfaces in any standard numerical plotting package able to generate surfaces of arbitrary expressions [3]. By plotting the inverse of the multifunction with its domain and codomain swapped accordingly, we obtain the correct surface structure representing the multifunction itself.

While this proposal seems all too neat, the most obvious drawback is that not all multifunctions have such friendly inverses. In fact, as any mathematician should know, inverses are in general difficult to calculate; consider the innocent-looking polynomial \( f(z) = z + z^5 \), under which Maple’s `solve` command refuses to continue and simply inserts a `RootOf` placeholder in the solution.

### 2.3 Partitioning

In order to plot branches individually, in a forward-manner, a way of spatially describing the branch is required. Instead of computing a parameterisation for the interior of the
branch, which can be difficult, it is also possible to consider the boundaries of the area too, i.e. the *branch cuts*, for which \[4\] provides a divide-and-conquer algorithm for elementary functions based on roots and logarithms. Unfortunately, the algorithm involves computing the preimages of complex-valued mappings in its recursion step, which is a difficult task akin to calculating inverses of functions.

### 2.3.1 Branch cuts

Assuming the smoothness of Riemann surfaces, every branch cut turns out to be the common edge of two adjacent branches; conceptually, each branch cut can be pictured as a barrier such that traversing a path that crosses this barrier means we move from one branch to another branch on the Riemann surface. When restricted to one single branch however, following the same path results in a jump at the branch cut to the value on the other side. Such discontinuities are easily observed by plotting the multifunction directly, including the branch cut within the interior of the plotting region.

![Figure 2.1: Plots of the principal branch of \(\ln(z)\)](image)

This happens because CAS systems often evaluate functions only to their principal values at any one time (see Multifunction representation), and so we are effectively restrained to working off the principal branch, thereby satisfying our ‘single branch’ condition. The Maple generated output of \(f(z) = \ln(z)\) in the region \(0 < \arg(z) \leq 2\pi\) in Figure 2.1a contains the branch cut of \(\ln(z)\) (defined conventionally as the negative real axis \(1\): the half-line \(\arg(z) = -\pi\) mathematically), and hence contains the discontinuity artefact. No-
tice that if we adjust the plotting region slightly, so that the branch cut is not contained in the interior, then we do not see the artefact: Figure 2.1b shows the plot over the range $-\pi < \arg(z) \leq \pi$, shifting the branch cut to the edge of the range.

Because a multifunction may have more than one branch cut, we need to partition the complex plane (or more specifically, the plotting region concerned) into an arbitrary number of sectors, each of which do not contain any branch cuts in their interiors, and so our objective of plotting branches has developed into plotting branch sectors. This touches upon the topic of binary plane partitions, which is a geometric problem in its own right. \[12\] proposes an algorithm for calculating partitions by straight lines, but branch cuts may be arbitrary curves and so this algorithm is insufficient for the problem at hand.

2.3.2 Branch points

An alternative method of graphing Riemann surfaces, successfully implemented within Mathematica, uses an algorithm for dividing the plotting region into its branch sectors purely through branch points, which in some sense can be viewed as the endpoints of the branch cuts (or even as their mathematical duals).

The algorithm \[13\] argues that the defining elements of such a partition are actually the branch points rather than the branch cuts, of which the latter can be chosen arbitrarily with the only condition being that they connect branch points. With regards to this, the problem of partitioning $\mathbb{C}$ into branch sectors is simplified hugely: all that is essentially required is to compute regions that do not contain a particular set of points (as opposed to lines) within their interiors. The Mathematica implementation chooses to do this by dividing by the polar coordinates (i.e. radially and azimuthally) of the points, although it is theoretically possible to split by their Cartesian components too.

A limitation of this algorithm is that its primary focus lies on algebraic functions, and so does not cover arctrig functions for example, although an idea might be to convert such functions into their logarithmic equivalents (convert/ln in Maple, following \[1\]) and re-express algebraically in terms of exponentials, which can be approximated by series expansions.

2.4 Parameterisation of branch sectors

Once the dimensions of the branch sectors have been determined, the next step is to determine what surface functions to plot within each of the sectors. A couple of approaches
are discussed here together with their applicability.

2.4.1 Solution of an ODE

Continuing on from the algorithm given in [13], one approach involves the construction and solution of first-order ordinary differential equations. The multifunction is first differentiated to give the system of equations, and then the initial conditions for each of the branches are given by evaluating the multifunction for its multiple values at an arbitrary point in the branch sector (Mathematica chooses a point near the edge of the sector).

While Mathematica contains functionality to solve such problems efficiently (NDSolve), such methods may not be widely available in all CAS. Luckily, Maple’s ODE utility contains methods for solving problems numerically too (dsolve/numeric), but the results of preliminary tests showed that computations were unacceptably slow. Optimisations would need to be performed for a functional implementation, which might include returning the interpolated functions per branch sector directly, as opposed to returning sampled coordinates, effectively delegating the interpolation step to render-time for the internal graphics renderer to process.

2.4.2 Alternate branch functions

Although not specifically described as a method of surface parameterisation, the concept of alternate branch functions may be worth consideration. The multiple values at a branch cut is usually expressable in terms of (the principal values of) the function itself [4], and the alternate branch function tries to encapsulate that relationship. Then, given a parameterisation for a branch $b_1$ and the alternate branch function $\tilde{f}$ on a boundary branch cut, the adjacent branch $b_2$ can theoretically be parameterised by the same $\tilde{f}$ without the need for further calculations.

The merits of using alternate branch functions are the broad scope of functions that apply, and its subjectivity to symbolic computations, avoiding potentially expensive numerical methods (cf. Solution of an ODE). For example, the branch surfaces for a large number of algebraic and arctrig functions composed from roots and logarithms can be determined recursively from the simple relations

\[
\begin{align*}
    f : z \mapsto \ln(z) & \implies \tilde{f} : z \mapsto \ln(z) - 2\pi i \\
    f : z \mapsto z^{\frac{a}{b}} & \implies \tilde{f} : z \mapsto z^{\frac{a}{b}} \exp(2\pi i k \frac{a}{b}), \quad k \in \{1, \ldots, b\}
\end{align*}
\]
However, such an implementation would require knowledge of the position of the branch cuts that form the boundary of the branch sector in question, and so this approach suffers from the same issues as the initial determination of the branch cuts (see Branch cuts).

2.5 Faithfulness

As our ultimate goal is to visualise a 4D map in 3D space, we have an inadequate number of axes available to represent the fourth coordinate spatially. Instead, as with other authors on the subject [3], we will opt for manipulating the texture of the surface for this coordinate; more precisely, we will use colour to represent the value of the coordinate.

We soon arrive at the problem of choosing which coordinates to display spatially and which coordinate to display by colour: the $xy$-plane is covered by the $\Re(z)$, $\Im(z)$ coordinates of the domain of the multifunction $\text{dom}(f)$ by convention, but there is question as to whether we plot the $z$-axis using the values of $\Re(f(C))$ or $\Im(f(C))$ of the codomain (or indeed a function of both, $|\cdot|$ for example). Notice that the decision is not arbitrary, as we cannot plot a single point in $xyz$-space with multiple distinct colours, but we can plot distinct points $(x, y, z_1)$, $(x, y, z_2)$, ... with the same colour. In other words, we require a one-to-one correspondence between $(x, y, z)$ points and the depicted surface [3].

Obviously, the exact correspondence depends on the multifunction, and in the logarithm example we see that plotting $(x, y, \Im(z))$ spatially with $\Re(z)$ as colour gives a faithful representation of its Riemann surface (proof in [3]). However, we know that for elementary functions and algebraic functions [13], plotting one of either real or imaginary part will certainly yield a faithful representation.
Chapter 3

Requirements

Given the previous discussion into the history of how Riemann surfaces have been viewed, it is now appropriate to lay down a set of objectives that the end product should meet.

3.1 Main objectives

The goal of the project is to investigate contemporary ideas and techniques in determining the various branches of Riemann surfaces, and to implement a 3D plotting system that generates faithful representations of Riemann surfaces based upon such ideas.

The primary objectives of this project are to:

- conduct research on existing techniques and algorithms for constructing Riemann surfaces; this might be done by analysing the source code, or pseudo-code, from the authors of existing implementations;
- extend upon algorithms by studying recent advances in related areas; this will require evaluating research texts for their applicability on the subject;
- develop a tool that consolidates the theoretical mathematics supporting the notion;

as well as to, in general, provide some insight and inspiration into where future work on the topic might be directed. In terms of physical results, the project will produce a library of functionality that allows the user to generate 3D visualisations of a Riemann surface of an input function.
CHAPTER 3. REQUIREMENTS

3.2 Limitations

The extent to which any project can reach is always bound by the availability of resources, and this is especially the case for this project. The major limitations involved in the development of this project in particular are:

1. **Time**: the project must be deliverable before the deadline of 7th May 2010. However, given the significant amount of work from other undertaken courses, good time management will be important to ensure that all necessary work is completed to an acceptable schedule. It will also be necessary to limit the scope of the project appropriately, given the limited time available to work on the project, so that expectations of the final outcome are achievable.

2. **Literature**: texts that exist on the topic of Riemann surfaces is highly abstract, and the amount of literature on their computational visualisation is equally sparse. References will need to be made to generic texts on complex analysis, and appropriate extractions will be made from them too.

3. **Software**: the availability of mathematical software is a must for such mathematically-inclined research to commence. Given that I have no previous experience with any mathematical packages at all, the recommendation and availability of the Maple software package on the University of Bath machines suggests that it should be chosen for conducting the investigation. The successful development of the project will then depend on the server status of the University machines maintained by BUCS (gigaterms) upon which Maple can be accessed.

3.3 Functional requirements

The Maple package is a piece of software that enables mathematical experiments to be carried out, and it incorporates its own programming language as an interface for issuing commands to its processing kernel and to evaluate mathematical objects in general. Any development on the Maple platform will primarily involve designing and developing a library of batched functionality that can be re-used by the ‘end-users’ of Maple, who are most likely going to be relatively proficient at the language. As such, one requirement is that the library can be included in other Maple code through the use of its native read function.

In terms of setting the scale of the project, another requirement will be that the library must be able to accept complex elementary functions as input for generating Riemann surfaces. This is a respectably-sized family of functions for which the project can focus on, and for which the task should be feasible within the given time constraints. Possible
extensions to the library might include the set of algebraic functions, involving polynomials in two variables.

Other functional requirements such as the ability to rotate viewpoints, add contour lines and colour to the surface will automatically be satisfied through the use of the native Maple plots package.

3.4 Non-functional requirements

As with most software libraries in general, usage of the library should not require in-depth technical understanding of how the code is written internally. There will not be a graphical interface as such; instead, the ‘interface’ to the library will be through the declarations of the Maple procedures, which will require clear in-place documentation and commenting.

Also, as with most mathematical/numerical software, it will be important that results are obtained in as short computation time as possible. Mathematical calculations can be intensive, especially when applied at scale, and so emphasis will be placed upon improving the efficiency of the code.
Chapter 4

Implementation

4.1 Branch cut partitioning

The algorithm given in [4] served as a starting point for determining branch sectors. It describes a method of calculating the branch cuts of a multifunction \( f \), which are essentially the images \( \gamma(I) \) of complex paths \( \gamma : I \rightarrow \mathbb{C} \) on intervals \( I \subseteq \mathbb{R} \) in their usual mathematical sense, and are hereby modelled as such.

4.1.1 Intervals in \( \mathbb{R} \)

As intervals form an integral part of the definition of a branch cut, a clear (programming) interface to working with intervals was needed. Maple is able to model real intervals using the \texttt{RealRange} data structure, but functionality for performing set operations on these structures was lacking, and it required writing a small library of extra procedures to fill this gap.

The following interval-oriented procedures were implemented:

- intersecting \((A \cap B, \text{IntervalIntersection})\),
- inverting \((\mathbb{R} \setminus A, \text{IntervalInversion})\),
- set differencing \((A \setminus B, \text{IntervalDifference})\),
- maximising \((\max(A), \text{IntervalMaximum})\),
CHAPTER 4. IMPLEMENTATION

- minimising \( \min(A) \), \texttt{IntervalMinimum},
- testing if an object is an interval \( \text{IsInterval} \),
- testing if an object is in an interval \( x \in A \), \texttt{InInterval}, and
- computing the image of a continuous real-valued function on an interval \( f(A), C^0 \ni f : A \to \mathbb{R} \), \texttt{IntervalImage}).

Although fairly straightforward to implement in general, care was necessary when dealing with the edge cases, such as the points \( \pm\infty \), singleton sets (in Maple, \texttt{RealRange(x, x)} is equal to \( x \)) and the sets \( \emptyset \) and \( \mathbb{R} \).

In particular, the \texttt{IntervalImage} procedure was built with various \texttt{if} clauses, due to the extra handling for piecewise-defined functions and also due to the shortcomings of the inbuilt Maple functions \texttt{minimize} and \texttt{maximize}: ranges are restricted to either closed intervals (i.e. \texttt{minimize(f(x), x=0..1)}) or single inequalities (i.e. \texttt{minimize(f(x), x < 1)}). For example, it is not possible to compute the minimum of the function \( f(x) = \frac{1}{x} \) over the interval \((0, 1]\) (requiring two inequalities \( \{0 < x, x \leq 1\} \)), or the minimum of the parabola \( f(x) = (x - \frac{1}{2})^2 \) over \((0, 1)\) (requires two inequalities \textit{and} is an open interval), both of which do indeed have well-defined minima.

4.1.2 Naïve branch cuts

Given the ability to construct branch cut data structures using intervals and procedures, it is now possible to start describing how to computationally generate the branch cuts of a multifunction. The basic idea of the algorithm in [4] is to recurse through each of the arguments of the function \( f \) and to collect branch cuts from whenever a subexpression \( f_i \) that contains branch cuts is passed. This is possible in Maple, as expressions are internally structured as expression trees (more formally, directed acyclic graphs), which are particularly applicable to recursive schemes.

Following [4], the only elementary functions that contain pre-defined branch cuts in my implementation are the logarithm \( \ln(z) \) with branch cut \( i(\{z \in \mathbb{R} \mid z \leq 0\}) \), where \( i \) denotes the identity map, and the \( n \)th root \( \sqrt[n]{z} \) with branch cut \( i(\{z \in \mathbb{R} \mid z < 0\}) \). As the requirements specify that focus should be placed on complex elementary functions, it is enough to cater just for the inverse trigonometric and inverse hyperbolic trigonometric functions, which can be expressed entirely as combinations of \( n \)th roots and logarithms by pre-converting them to their logarithmic forms. This is done through Maple’s \texttt{convert/ln}, and is executed at the time of branch cut construction. However, the code can easily be extended to include more definitions if needed in future.
An extra consideration implemented in the code is the openness of the endpoints, which is not considered in [4]. This was possible given the availability of the new interval manipulation utilities written, and is put into effect using the same Open structure that is used internally for the definition of open-ended RealRange intervals, so that consistency can be achieved (from a programmer’s point of view). Examples of this are found in the definition of the $n$th root branch cut (around zero), and also in the function pre-image calculations. The aim of allowing the possibility of open endpoints is to add an extra level of precision to the calculations; however, it is not technically necessary because in our primary goal of graphing a surface patch, the openness of the boundary of the plotting area makes no difference to the graphed output because the plot function only accepts closed intervals anyway. Furthermore, testing was incomplete as the general classes of functions containing singularities were not thoroughly examined, and so may need re-assessment in future.

Listing 4.1: Openness of the zero endpoint of the root branch cut

One of the more difficult aspects of implementing the algorithm was indeed finding the function pre-image, which the paper suggested to find the subtly-different image of the inverse function instead, assuming that this can be done using the native solving utility of the CAS. However, this assumption is quite significant, and even when implemented (here in CutPreimage) its execution frequently requires a lot of low-level manipulations involving piecewise-defined functions and their conditions. For example, considering the relatively simple modulus function:

```maple
> solve(y = abs(x), x) assuming 0 < y, y < 1;
y, -y
> solve(y = abs(x), x) assuming y < 1;
\{  [0]   y = 0
\{  [y, -y]  0 < y
\{  []      otherwise
```

immediately introduced a piecewise structure, which needed extra attention. To deal with piecewise structures, the structure is decomposed into expression parts and condition parts, corresponding to the left and right hand columns of the displayed output respectively, which is easily done as piecewise objects are already internally defined in this way within Maple.
For a condition part, each of the contents of the buffer $\text{previousConditions}$ are individually negated and then collectively ‘AND’-ed along with the current condition to generate a resultant condition. The buffer $\text{previousConditions}$ itself is accumulated by adding each condition part to it after it is processed.

The expression part takes the last generated resultant condition and passes it along with the expression itself to the procedure $\text{generateBranchCutsFromCondition}$ for it to build a suitable domain for the expression, and finally to generate a set of branch cuts.

```latex
> \text{CutPreimage}(x \rightarrow \text{abs}(x), \text{Cut}(x \rightarrow x, \text{RealRange}(-1, 1))); \\
\{[x \rightarrow x, \text{RealRange}(\text{Open}(0), 1)], [x \rightarrow x, \text{RealRange}(-1, \text{Open}(0))]\}
```

Listing 4.2: Manipulating piecewise defined functions to return multiple Cuts

The reason for the need for this extra complication is that, in written mathematics, piecewise functions generally bear the inherent assumption that the condition parts are mutually exclusive, i.e. each ‘piece’ of a piecewise function can only be satisfied if the preceding ‘pieces’ cannot, and so $\text{CutPreimage}$ tries to imitate this assumption using boolean logic.

### 4.1.3 Branch cut simplification

One of the key points mentioned in [4] is the idea of the canonical form for a branch cut. Because a branch cut is the image of a path, the parameterisation of the path is not important, and this is why branch cut simplifications are possible. Canonical forms are useful firstly from the point of view of simpler symbolic manipulations, but secondly and more importantly, they are crucial for comparisons of branch cuts (see Removable branch cuts), which is why simplifications are immediately performed on initial construction of the branch cut data structure (in Cut).

Some of the branch cut reductions suggested in the paper were implemented together with other similar rules hereby suggested as they stem from the same principles.

- One of the simplification rules mentioned in [4] was the intermediate value rule, which applies to continuous real-valued paths $\gamma : I \rightarrow \mathbb{R}$ only. As Maple does not contain functionality for determining whether a function is real-valued or not, this required writing a separate procedure $\text{isReal}$, testing for the condition $\Im(\gamma(x)) = 0, \forall x \in I$.

An implementation of a continuity test was also required as the Maple procedure $\text{iscont}$ proved insufficient; discontinuities at the endpoints of closed intervals were not recognised, for example
> iscont(1/x, x=-1..1);
false

> iscont(1/x, x=0..1);
true

shows that the infinite discontinuity \( \lim_{x \to 0^+} \frac{1}{x} = \infty \) was completely ignored by \texttt{iscont}. The new improved \texttt{IsContinuous} function written for this implementation extends \texttt{iscont} by checking if either of the endpoints of \( I \) are both closed and finite, and if so tests the limit of \( \gamma \) at that endpoint according to the usual mathematical definition of continuity.

With \( \gamma \) satisfying the above conditions (and \( \gamma \neq i \), which would otherwise result in infinite recursion), the branch cut \( L = \gamma(I) \) is replaced with \( L' = i(\min(L), \max(L)) \).

- A similar reduction rule was deduced for continuous, purely imaginary-valued paths \( \gamma : I \to \mathbb{R}i \), and follows the same structure as the previous rule with the difference of the test condition \( \Re(\gamma(x)) = 0, \forall x \in I \) (\texttt{isImaginary}).

- For paths that are real-valued but not continuous at a finite number of points, it will still be possible to obtain a simplification by dividing the branch cut into several branch cuts disjoint at the discontinuities. For example, for \( f : [-1, 1] \to \mathbb{R}, x \mapsto \frac{1}{x} \), there is a pole at \( x = 0 \), which is dealt with by dividing the domain into two intervals \([-1, 0) \) and \((0, 1]\), and then recursively applying simplification on each sub-interval.

> Cut(x -> 1/x, RealRange(-1, 1));
[x -> x, RealRange(-infinity, -1)], [x -> x, RealRange(1, infinity)]

This technique is accomplished in the procedure \texttt{SplitDiscontinuousCut}.

- More generally, straight line branch cuts can always be reduced to a canonical form. The precise definitions and method are given in [4], and were implemented as a combination of procedures:

  - \texttt{IsLine}: tests whether a path is a line. This is done by differentiating the expression and testing if the derivative is constant.

  - \texttt{CanonicalLine}: tests whether a line \( \gamma : x \mapsto ax + b \) is already in its canonical form. More precisely, it tests whether the gradient \( a \in \mathbb{C} \) satisfies the conditions of

    * unit length, \( |a| = 1 \),
    * direction such that \( \arg(a) \in [0, \pi) \), and
    * orthogonal to the offset, \( a \cdot b = 0 \).

  - \texttt{CanonicaliseLine}: converts a line to canonical form. This follows [4] with the \textit{added extra} of swapping the openness (of which [4] does not take into account) of the endpoints of the domain.
• Further geometric shapes such as circles and arcs of circles were not implemented, but have their canonical forms defined in [4] should someone wish to implement them in future.

4.1.4 Removable branch cuts

A potential issue with a composite multifunction $f$ is that branch cuts generated from its individual constituent functions $f_i$ may be cancelled out when considered collectively in $f$. Such erroneously blacklisted areas where the function is not actually discontinuous are called removable branch cuts following the terminology in [4], which goes further to propose a method for deciding the correct branch cuts.

As a brief outline of what is needed, given the branch cuts of each of the individual $f_i$ (our naïve branch cuts), the union of these cuts is reformulated as a pairwise disjoint union of smaller branch cuts dependent on the pairwise intersections between the $f_i$ cuts. Each is then re-evaluated for branch cut status within the final composite $f$.

The several steps in this operation require supporting sub-operations, hereby explained in finer detail.

Partitioning curves in C

It is important to reiterate that the objective here is to divide a set of branch cuts into smaller sub-cuts, such that both the unions describe the same total area on the complex plane. Although a simple concept, there was no pre-existing procedure in Maple to compute such a partition. Hence, this required writing a procedure to do this (hereby dubbed CutsPartition), namely to test a set of branch cuts for any non-empty pairwise intersections. An outline of the algorithm implemented is given in Algorithm 1.

An extreme restriction is made in step 1 of the algorithm by which two cuts can intersect only if the paths $\gamma_i, \gamma_j$ are identical (as suggested by [4]). This relies on the branch cut representation being in its canonical form (see Branch cut simplification), and is one of the major hurdles in terms of extensibility of the algorithm, because canonical forms – although they can be defined for common geometric shapes ([6] gives a few) – are perceivably hard to define for arbitrary curves on a plane in general. An alternative solution to this issue is to rely on the native solve capabilities within Maple to determine the intersection (if it exists), and an experimental attempt was made in CutsIntersection.

# Direct solve? Theoretically (and apparently) works (but hard to manipulate
\section*{Algorithm 1} Partitioning a set of branch cuts

Given a set of branch cuts $C$, consider the cuts $\gamma_i(I_i), \gamma_j(I_j) \in C$ for each $i \neq j$:

1. If $\gamma_i \neq \gamma_j$, or if $I_i \cap I_j = \emptyset$, then these two cuts are disjoint, so skip to the next pair of cuts, or return $C$ if there are no more pairs. Otherwise, denote $I = I_i \cap I_j \neq \emptyset$ and continue.

2. Calculate the two interval (set) differences $D_1 = I_i \setminus I$ and $D_2 = I_j \setminus I$.

3. Replace the original two cuts with the three new cuts (or more if $D_1, D_2$ are not intervals):

$$C' = \left( \bigcup_{k \neq i, k \neq j} \gamma_k(I_k) \right) \cup \gamma_i(D_1) \cup \gamma_i(I) \cup \gamma_i(D_2).$$

4. Re-run algorithm on $C'$.

The main problem with using the \texttt{solve} approach was the difficulty in extracting the useful data from the result, and was therefore not investigated any further due to time constraints. On the other hand, while the algorithmic approach gives a severely restricted result set, it is \textit{guaranteed} to work as long as expressions are put into their canonical form, although it is not always possible to confirm that an arbitrary expression can be formulated as such.

\section*{Removability}

The second step is to decide whether a branch cut is either removable or required. This involves computing an alternate branch function $f'$ for each branch cut $C$, and making comparisons between $f'$ and the original function $f$. Equality signifies that the discrepancy of values at the discontinuities on the branch cut are cancelled, giving a resultant continuity over the thereby nullified cut.

The alternate branch function with respect to a branch cut $C$ is calculated by traversing the expression tree and replacing every subexpression $f_i$ that contributes to $C$ (more explicitly, every $f_i$ that introduces a set of branch cuts $C_i$, at least one of which intersects $C$) with its alternate branch function $f'_i$ (\textit{[3]}).
The replacement of subexpressions with their ‘alternates’ requires low-level operations on both expression trees and operator type variables in Maple, mostly involving \texttt{op}, \texttt{apply}, \texttt{unapply} and \texttt{subsop} commands, in order to be able to separate out the ‘structure’ of an expression with the actual operators used in the expression. A separate mini-library of procedures \texttt{Decompose} and \texttt{Recompose} were written solely for this purpose.

The job of the \texttt{Decompose} procedure is to make the distinction between functions/procedures and the constants/variables in an expression \( f \). The purpose of the abstraction is so that a procedure call situated at one ‘node’ of the expression tree can be ‘swapped out’ for another procedure, effectively performing a syntactic replacement in the expression. This is done by recursing through each node of the expression tree and pulling out the \texttt{operator} \( \text{op}(0, \text{expr}) \), noting its associated number of \texttt{operands} \( \text{nops}(<\text{expr}>\))\), and inserting a dummy placeholder in its place. A restriction on the form of the dummy placeholder is that it needs to be recognisable within the scope of the library only, and so that the placeholder name cannot be easily used mistakenly as a normal variable, in order to avoid out-of-scope issues when the library is put to use by the mathematician. Hence, the form \( _F \) was chosen for each \( i \)th operator substituted, following the Maple convention of prefixing a capital letter with an underscore to denote a variable name reserved for internal use ([8]). The implications of this restriction were that the \texttt{Decompose} procedure involved various string operations such as \texttt{evaln}, \texttt{cat} and \texttt{substring}, potentially hampering the performance of the code, as Maple is primarily a mathematics-oriented language and is optimised as such.

\begin{verbatim}
22  # Replace function with a dummy function within the expression.
23  expr := cat(evaln(_F), index)(op(expr));
\end{verbatim}

Listing 4.3: Syntactic replacement of an operator with a dummy placeholder

Given the output replaced expression \( \tilde{f} \) and an (ordered) list of replacement mappings \( \{f_i(x_1, x_2, \ldots, x_n)\} \), an evaluable expression can be ‘recompiled’ again through the \texttt{Recompose} procedure, effectively undoing the previous extraction by replacing in the opposite direction.

With the ability to syntactically swap operators at any node within an expression tree, it is then possible to run through any expression \( f \) using the replacement algorithm (given by [4], implemented in \texttt{BranchCuts}) and obtain an alternate branch function at a branch cut \( C \), in order to determine its removability.

[4] explains the example \( f : z \mapsto \ln(z+1) - \ln(z - 1) \), which results in the final set of branch cuts just containing the real interval \( \nu((-1, 1]) \) after having the branch cut \( \nu((-\infty,-1]) \) removed from the full \( \nu((-\infty,1]) \). This example also tested positive in my implementation:

\begin{verbatim}
> BranchCuts(z -> ln(z + 1) - ln(z - 1));
\end{verbatim}
\begin{verbatim}
  \{[x \rightarrow x, \text{RealRange}(\text{Open}(-1), 1)]\}
\end{verbatim}
> BranchCuts(z -> sqrt(1 + sqrt(z)));
  {x -> x, RealRange(-infinity, Open(0))}

4.1.5 Plotting branch cuts

Functionality was also written for displaying branch cuts on the complex plane (through
\texttt{PlotBranchCuts}), primarily as a visual aid for testing the correctness of a branch cut
computation.

To make plotting (and working with branch cuts in general) simpler, a set of branch cuts
is always simplified at the end of the computation; for example, two branch cuts that are
connected and have the same paths (i.e. $\gamma_1 = \gamma_2$ and $I_1 \cap I_2 \neq \emptyset$) naturally extend each other,
and so are converted into one single branch cut $\gamma_1(I_1 \cup I_2)$. Again, this simplification is only
valid because the parameterisation of the path is not important: we are only concerned
with the image of the path in $\mathbb{C}$.

4.1.6 Possible further development

Partitioning $\mathbb{C}$ by curves

As discussed in \textbf{Branch cuts}, the problem of binary plane partitioning (not to be mistaken
for binary space partitioning) is a significant problem that deserves to be explored as a
separate project. Although not implemented here, a sketch of a geometric approach is
outlined in Algorithm 2.

The algorithm is supplemented with examples (Figure 4.1) to help clarify the steps.

Sector parameterisation

Even with a successful geometric partitioning of the plane, the partition sectors are going
to be difficult to symbolically parameterise in general. A potential solution might involve
making use of \textit{conformal mappings}, to help map from the shape of the sector to an easily
parameterisable shape, such as a rectangle (for Cartesian coordinates) or a circle (for polar
coordinates) – effectively transforming to another coordinate system.
Algorithm 2 An experimental approach to binary plane partitioning

Given a set of smooth connected curves \( C \), and a plane \( \Pi \) centred at the origin,

1. Divide all non-simple curves at their ‘intersection’s and replace the corresponding simple counterparts into \( C \).

2. Sort all non-closed curves \( C' \subseteq C \) by the furthest end-point of (the closure of) each curve \( C \in C' \), distance descending.

   Suppose \( \gamma : I \rightarrow C \) is a parameterisation of \( C \), with \( a = \inf(I), b = \sup(I) \). Then such ‘end-points’ are given by

   \[
   \{ e_1, e_2 \} = \{ \lim_{x \rightarrow a^+} \gamma(x), \lim_{x \rightarrow b^-} \gamma(x) \}
   \]

   and the furthest end-point \( e \in \{ e_1, e_2 \} \) is such that \( |e| = \max\{|e_1|, |e_2|\} \).

3. For each of the non-closed curves \( C \in C' \), extend the curve up to either the boundary of \( \Pi \) or an existing curve in \( C \), whichever is first. If the two end-points of (the closure of) the curve \( C \) are \( e_1, e_2 \), then the line is described by

   (a) calculating the direction of the curve at the end-point, \( d = \lim_{x \rightarrow e} \arg(\gamma'(x)) \),

   (b) constructing a straight line from \( e \) in the direction \( d \).

4.2 Branch point partitioning

To avoid the computational complexities of partitioning planes by curves, the alternative method of partitioning by points was considered and implemented alongside as well. In addition to reducing the partitioning criteria to only avoiding a finite number of points, another major argument for this method is that it also allows for a much greater degree of freedom in sector parameterisation later.

4.2.1 Algebraic branch points

A mathematical criterion is given in [13] describing how Mathematica proceeds to calculate the branch points of an algebraic function. It states that for an algebraic polynomial \( P(w, z) \), where \( w = w(z) \) is a multifunction of \( z \), if both the conditions

\[
P(w, z) = 0, \quad \frac{\partial}{\partial w} P(w, z) = 0
\]
are satisfied for some \( z \in \mathbb{C} \), then \( z \) is an \textit{algebraic} branch point, i.e. the number of revolutions needed in traversing a small closed path around the point in order to return to the same branch is finite \([10]\).

For solving systems of multivariate polynomial equations such as in Equation \([4.1]\) Mathematica has the utility \texttt{NSolve} which is adequate for polynomials, but Maple’s standard generic \texttt{solve} command frequently introduces \texttt{RootOf} placeholders, which are difficult to manipulate. An alternative built-in procedure dedicated to solving algebraic systems, \texttt{RootFinding[BivariatePolynomial]}, was considered and tested (which included falling back to floating-point approximations when exact solutions are not available), but this severely restricted the scope of acceptable functions to polynomials only – in violation of our complex elementary functions requirement, and so it was more favourable to compromise on expanding the \texttt{RootOf} objects instead (in \texttt{AlgebraicBranchPoints}).

\begin{verbatim}
# Solve for branch points (z-values)
solutions := {solve({expr(w, z), diff(expr(w, z), w)}, {w, z},
   AllSolutions=true)};
solutions := map(allvalues, solutions);
solutions := map(rhs, map2(op, 2, solutions));

Listing 4.4: Expanding all RootOf structures returned by solve
\end{verbatim}
In fact, using the generic `solve` utility effectively extends the Mathematica implementation, because of its ability to cater for a wider range of functions, such as the inverse trigonometric functions and their hyperbolic counterparts, on top of the polynomials supported in [13].

```mathematica
> AlgebraicBranchPoints((w, z) -> w^2 - z, 1..3);
{0}

> AlgebraicBranchPoints((w, z) -> w^2 - 1 + z^4, 1..3);
{-1, 1, -I, I}

> AlgebraicBranchPoints((w, z) -> w^4 - w - z, 1..3);
\[
\frac{1}{3} \frac{1}{3} \frac{1}{3} \\
\frac{3}{2} \frac{3}{2} \frac{1}{2} \frac{1}{3} \frac{3}{2} \\
\{- \frac{3}{16} - 3/16 I \frac{3}{2}, \frac{3}{16} + 3/16 I \frac{3}{2}\}
\]

> AlgebraicBranchPoints((w, z) -> sin(w) - z, 1..3);
{-1, 1}

> AlgebraicBranchPoints((w, z) -> csch(w) - z, 1..3);
{-I, I}
```

Listing 4.5: Examples of `AlgebraicBranchPoints` output

It is interesting to note the assumptions in the representation of the multifunction used by this approach. The \( n \)th roots \( \sqrt[n]{z} \) embedded within the multifunction \( w(z) \) must be algebraically rearranged in \( P(w, z) \) in such a way that they get replaced with their corresponding inverse (the \( n \)th power \( w^n \)). This is important so that the (non-linear) expression \( P(w, z) \) becomes single-valued and hence avoids the principal-branch ‘restrictions’ automatically carried out by the CAS (see [Multifunction representation]), and so that all the branches of \( w(z) \) are available for computation. For example, in Figure 4.2 setting \( P(w, z) = w^2 - z \) generates the full surface for the complex square root, whereas \( P(w, z) = w - \sqrt{z} \) only gives its principal branch.

### 4.2.2 Logarithmic branch points

As Equation 4.1 is only a sufficient condition for a branch point, solving the equations will not produce all branch points of the multifunction; for example, the branch points that occur at essential singularities will not be returned. Furthermore, these equations necessarily fail to produce logarithmic branch points: branch points with an infinite order. For example, at the branch point \( z = 0 \) for the function \( z \rightarrow \ln(z) \), we have that \( P(w, 0) \neq 0 \) as it cannot “lie on the surface”. Instead, extra functionality is needed to account for this type of branch point.
One possible method for locating the logarithmic branch points of a multifunction is through considering its complex derivative. It is known that if the derivative $w'(z)$ of a function $w(z)$ has a simple pole at a point $z_0$, then $w(z)$ has a logarithmic branch point at $z_0$. Again, this is only a sufficient condition and so it will not calculate all possible logarithmic branch points. However, it does provide us with a method of finding at least a subset of them: experimentally, the logarithmic branch points of the arctrig(h) functions were all successfully recovered in this way (compared against \[14\]).

Firstly, all potential poles of the derivative $w'$ are gathered. For calculating Algebraic branch points the input expression is entered in the form of $F(w, z)$ and so it requires an extra preliminary step to rearrange the expression back into the form $w = w(z)$ before poles can be solved for.

**Theorem 4.1.** Let $f$ be a function analytic around an isolated singularity $z_0 \in \mathbb{C}$ such that $\lim_{z \to z_0} f(z) = \infty$. Then, the following are equivalent:

(a) $z_0$ is a simple pole of $f$ (i.e. a pole of order 1)

(b) $z_0$ is a simple zero of $\frac{1}{f}$

(c) $\text{Res}(f, z_0)$ exists, is non-zero and is given by $\text{Res}(f, z_0) = \lim_{z \to z_0} \frac{1}{z - z_0} f(z)$. 

Figure 4.2: Arrangement of $P(w, z)$ makes a difference
The \[a\] \iff \[b\] equivalence in Theorem 4.1 is an elementary mathematical result familiar to complex analysts, and in particular, our interest lies in the equivalence of a point \(z_0\) being a simple pole of \(w'\) if and only if both \(\frac{1}{w'}(z_0) = 0\) and \(\left(\frac{1}{w'}\right)'(z_0) \neq 0\), of which the first condition was chosen for implementation as a (very) loose criterion for gathering a set of potential poles of \(w'\). Although Maple was able to solve for this condition for a range of expressions without problems, there were issues when experimenting with \(F(w, z) = \tanh(w) - z\) and \(F(w, z) = \coth(w) - z\), which sometimes returned the points \{−1, 1\} (correct for both expressions) but at other times fell short with an infinite recursion error.

\[\text{LogarithmicBranchPoints(unapply(tanh(w) - z, w, z), 1..3);}\]
\{-1, 1\}

\[\text{LogarithmicBranchPoints(unapply(tanh(w) - z, w, z), 1..3);}\]
\text{Error, (in ithprime/small) too many levels of recursion}

Listing 4.6: Haphazard error messages

Instead, a workaround for this non-deterministic issue was to solve for poles from an alternative definition of a pole. Using the definition of a singularity from Theorem 4.1, a command is issued to solve the equation \(|w'(z)| = \infty\), for which Maple correctly interpreted the “equality” in the limit-wise sense, and even managed to give the correct superset of solutions as generated from solving through the previous method in Theorem 4.1\((b)\), but in an evidently more stable manner.

\[\text{#potentialPoles := \{solve(1/w_dash(z), z)\}; \# Doesn't work all the time}\]
\[\text{potentialPoles := \{solve(abs(w_dash(z)) = infinity, z)\};}\]

Listing 4.7: Comparison of unstable/stable solutions

The last step of computing the simple poles is to ‘filter’ the gathered points and to test for whether each pole is simple, and this is done by calculating the residue of the function at the point. As discussed in [9], a simple pole can be determined by considering the equivalence \[a\] \iff \[c\] in Theorem 4.1 and so the conditions in \[c\] are tested in the procedure \text{IsComplexPole}. This was done by considering the limit directly, and not through the internal Maple procedure \text{residue} because the latter tries to return an expression for the value of the residue itself, for which it is unable to evaluate at particular points, whereas all that is required here is a boolean value for whether it is equal to zero or not.

\[\text{# Logarithmic branch point for ln at z = 0}\]
\[\text{ln_dash := diff(ln(z), z)};\]
\[\text{ln_dash := unapply(ln_dash, z)};\]
\[\text{residue(ln_dash(z), z = 0)};\]
\[1\]

\[\text{IsComplexPole(ln_dash, 0, 1)};\]
true

> # No logarithmic branch point for sqrt at z = 0, but fails to evaluate
> sqrt_dash := diff(sqrt(z), z):
> sqrt_dash := unapply(sqrt_dash, z):
> residue(sqrt_dash(z), z = 0);
> 1
> residue(------, z = 0)
> 1/2
> 2 z

> IsComplexPole(sqrt_dash, 0, 1);
false

> # No logarithmic branch point for sqrt at z = 1
> residue(sqrt_dash(z), z = 1);
> 0

> IsComplexPole(sqrt_dash, 1, 1);
false

Listing 4.8: Comparison of complex residue calculations

Although the residue criterion is useful for testing the validity of simple poles, it is inconvenient for finding possible poles, for which Theorem 4.1 (b) (and its workaround) are more applicable, and so that is why both these criteria are used in the implementation of LogarithmicBranchPoints.

> LogarithmicBranchPoints((w, z) -> w^3 - z, 1..3);
{}  
> LogarithmicBranchPoints((w, z) -> sin(w) - z, 1..3);
{}  
> LogarithmicBranchPoints((w, z) -> exp(w) - z, 1..3);
{0}  
> LogarithmicBranchPoints((w, z) -> tan(w) - z, 1..3);
{-I, I}  
> LogarithmicBranchPoints((w, z) -> tanh(w) - z, 1..3);
{-1, 1}

Listing 4.9: Examples of LogarithmicBranchPoints output
4.2.3 The infinity branch point

The above methods are able to compute finite branch points, but the issue with a possible branch point at infinity still remains unaccounted for. However, as will be explained later, it will not pose a problem if this possibility is ignored altogether.

Mathematically, infinity branch points are unearthed by applying the bilinear transformation $z \mapsto \frac{1}{z}$ prior to solving for a branch point at zero (a decent explanation is given in [7] in terms of Laurent series expansions). Inevitably, this results in a rational function — which the RootFinding[BivariatePolynomial] approach would have been unable to cope with anyway — which can then be checked for the zero branch point.

Such a task should only involve a couple of trivial substitutions, and so is left as a future exercise (perhaps in BranchPoints) to be solved if only for completeness’ sake.

4.2.4 Partitioning $\mathbb{C}$ by points

Once we have obtained the correct set of non-infinite branch points $P$ by which to partition our plotting region, attention is turned to the geometric aspects of the task.

Regardless of the coordinate system in place, a standard method of dividing the region would be to take the individual coordinates of all the points $P$ in question and calculate their Cartesian product (or product set) $P' \supseteq P$ to use as the vertices of the sectors. For example, a representation in Cartesian coordinates would be $P' = \{ x + yi \mid x \in \mathbb{R}(P), y \in \mathbb{I}(P) \}$, and in polar coordinates $P' = \{ r \exp(i\theta) \mid r \in |P|, \theta \in \arg(P) \}$. In principle, this introduces extra points that do nothing more than simplify computational aspects, but which do not complicate matters for a Maple implementation because the plots[display] command is able to ‘connect’ continuous regions in a visually coherent manner.

However, it is possible to be smarter about this and to partition the region generating a fewer number of sectors. In terms of performance, a little more time is spent on calculating a more optimal partition, but a lot less time is wasted at the surface parameterisation stage later (which is much more intensive) as we have fewer sectors to compute.

The first step in calculating a more efficient partition is to apply some sort of ordering to $P$. Although $\mathbb{C}$ cannot theoretically be ordered in the usual mathematical sense, we need an ordering of branch points from which to start building sectors. The $\mathbb{C} \rightarrow \mathbb{R}$ projections $|z|$ (abs) and $\arg(z)$ (argument) are used in the procedure SortComplex to order complex numbers by first comparing the distances of the points from the origin, and then by comparing their (principal) arguments.
The next step is to iterate expanding outwards through each possible annuli in turn, whose boundaries are the circles with radii defined by the distances of the branch points, and then to further divide each annulus by the angles given by only the branch points lying on the outer boundary. This can be compared with the standard partition method, which divides every annulus by the angles of all branch points, thereby making unnecessary divisions.

![Figure 4.3: Plane partitioning by a given set of points](image)

Although theoretically sound, the partition from the refined approach (implemented experimentally in BranchSectors) resulted in discontinuity artefacts occurring at the boundaries of the annuli being displayed when generating the full Riemann surface (Figure 4.4). This is possibly due to the algorithm providing an insufficient number of initial conditions (when solving for the surface equation, see Initial conditions) to allow for the surface of one sector to remain connected to the surface of its (radially) adjacent sector. Nevertheless, due to the time constraints given for the project, the standard finer-granularity partition was used, which proved satisfactory.

An interesting point to note here is how infinite branch points are finally dealt with. Because the size of the plotting region will always remain finite (in practical terms), it is enough to simply replace any infinite branch point with an artificial branch point at the edge of the plotting region (in BranchSectors, this is done on the negative real axis, keeping in line with the convention of the logarithmic branch cut being extended along the negative real axis). An additional nicety is that, together with an artificial branch point inserted at the origin too, sectors can then be generated for multifunctions containing less than two branch points without the need for extra exception handling within the code (i.e. for technical reasons). This is why the implementation forces the ‘zero branch point’ and the ‘infinity branch point’ to be included in the set of partitioning points \( P \), regardless of whether they are actual branch points of the multifunction.
4.3 Branch sector parameterisation

The current implementation builds on the [13] method of constructing an ODE system, and then solving it for a surface representation for each branch of the multifunction $w(z)$ (in its rearranged form $F(w, z)$). More specifically, the solutions are parameterisations along the two coordinate dimensions (i.e. in this case, the distance $r$ and the angle $\theta$) for various areas of the sector as defined by some given grid. These parameterisations allow sample points to be interpolated and collectively compiled as a final plotting mesh for that sector.

4.3.1 ODE construction

As explained in [13], the generation of the two ODEs is performed only once because it depends only on the multifunction and not on the sector region. Denoting $F'(w, z) = \frac{\partial}{\partial w} F(w, z)$, then the radial and azimuthal interpolations, $\Psi_\rho$ and $\Psi_\varphi$, can be calculated from the first-order differential equations:

$$\Psi'_\rho(\rho) = F'(\Psi_\rho, \rho d) \cdot \frac{\partial}{\partial \rho} (\rho d)$$

(4.2)
\[ \Psi'_\varphi(\varphi) = F'(\Psi_\varphi, r \exp(i\varphi)) \cdot \frac{\partial}{\partial \varphi} (r \exp(i\varphi)) \] (4.3)

which is implemented in MakeOdes and returns a pair of inert expressions, which are necessary for delaying the evaluation of the derivatives \( \Psi'_\rho \) and \( \Psi'_\varphi \) until later.

### 4.3.2 Initial conditions

In general, initial conditions are required in order to be able to solve ODEs, and so the next step is to determine these conditions by finding a point for every possible branch surface; in other words, we provide an arbitrary point \( z \in \mathbb{C} \) that lies in the sector (the implementation just uses the position of the initial vertex of the sector plus some small error), and then we collect the multiple correspondences \( w(z) \in \mathbb{C} \) that lie on the various branches, such that \( F(w, z) = 0 \).

This is done in InitialConditions by issuing a simple solve command, and works with relative ease for polynomials as they only have a finite number of branches (i.e. the cube root, Figure 4.5a) and so we obtain a finite number of \( w \) solutions. However, extra handling is needed for when the input function spans infinitely-many branches, giving rise to an infinite number of \( w \) values (i.e. the complex logarithm, Figure 4.5b); in this case, a hard-limit is put into place, which controls the number of values to return (the ‘first’ three, by default) after which SampleValues goes through to evaluate each numerically.

![Figure 4.5: Infinite branches are capped by a hard-limit](image)

(a) \( w^3 - z \) has 3 branches  
(b) \( \exp(w) - z \) should have \( \infty \) branches
4.3.3 Interpolation

The next step is to produce the mesh of \( z-w \) correspondences per branch per sector. This involves solving the previously constructed ODEs for each branch, for each sector using a given grid for dividing the sector (Algorithm 3).

The solution of the ODE uses the internal Maple function \texttt{dsolve}, which is a generic utility capable of solving ODEs involving polynomials, circular and hyperbolic functions in two variables, which satisfies our requirements. For extensibility reasons, however, the \texttt{dsolve/numeric} version of the procedure was used in order to guarantee a solution in the case of non-elementary complex functions; the option of obtaining a numerical solution to the ODE is available to us because the primary goal of the implementation is only to produce a \textit{visualisation} of the Riemann surface rather than to capture a \textit{rigorous} mathematical model of the surface – in that case, an exact symbolic solution \textit{would} indeed be preferable.

\begin{algorithm}
\textbf{Algorithm 3} Sampling for a mesh for a sector-branch

Let \( \varepsilon \) be some small error. Given a sector \( S = [r_0 \exp(i\theta_0), r_1 \exp(i\theta_1)] \subset \mathbb{C} \), its grid \( G = [\delta_r, \delta_\theta] \), and one of its branches with initial surface point \( w_0 \in \mathbb{C} \),

1. Compute the radial interpolator \( \Psi_\rho \) by solving Equation 4.2 with the initial condition \( \Psi_\rho(r_0 + \varepsilon) = w_0 \).

2. Letting \( r \) step through from \( r_0 \) to \( r_1 \) in intervals of \( \delta_r \):
   
   (a) Interpolate at \( r \) to give \( w_0' = \Psi_\rho(r) \).
   
   (b) Compute the azimuthal interpolator \( \Psi_\phi \) by solving Equation 4.3 with the initial condition \( \Psi_\phi(\theta_0 + \varepsilon) = w_0' \).
   
   (c) Letting \( \theta \) step through from \( \theta_0 \) to \( \theta_1 \) in intervals of \( \delta_\theta \):
      
      i. Interpolate at \( \theta \) to give \( w = \Psi_\phi(\theta) \).
      
      ii. Add the pair \([z, w]\) to some structure \( B \), where \( z = r \exp(i\theta) \).

3. Return \( B \).
\end{algorithm}

Thus, accumulating the results of running Algorithm 3 on every branch in every sector gives a final data-structure of \( z-w \) correspondences \textit{representative of} a given multifunction expression \( F(w, z) \) plus a granularity specification \([\delta_r, \delta_\theta]\) (implemented in \texttt{RawRiemannSurface}). These correspondences can then be evaluated before being passed to the native \texttt{plot} functions in order to display the surface to screen.
4.4 Plotting

A difference between this implementation and the Mathematica implementation is that at
the interpolation stage, Mathematica decides to apply the passed ‘display’ function (i.e.
$\Re$ or $\Im$, or any other $\mathbb{C} \to \mathbb{R}$ mapping we care to submit to the algorithm) directly to
the interpolated value before returning the mesh in $[x, y, \text{display}(w)]$ terms, whereas the
current implementation returns the correspondences in their entirety as $[z, w]$ pairs.

The branching off of ideas is for both performance and flexibility reasons. Considering
that the ODE solving is almost certainly the most intensive step, we can save on CPU
cycles if we wish to plot the same multifunction using multiple display functions: we can
generate the fully representative correspondences once and save them to memory (or even
some non-volatile storage), before using them to plot multiple (not necessarily all faithful)
Riemann surfaces with various display functions ($\text{RiemannSurface}$).

The full representation of the $z$-$w$ correspondences would also give greater freedom in
setting the colour function of the surfaces in each plot. However, this was not attempted
due to time constraints.
Chapter 5

Conclusions

Through this investigation, the successful implementation of a functional library was produced with design decisions heavily backed up by various mathematical theories surrounding multi-valued functions and complex analysis in general. Given the lack of available direction from the public domain, many approaches had to be considered including both geometric approaches and analytic approaches, of which the latter seemed to prevail due to its applicability to algorithmic computations; nevertheless, many new insights were gained through the undertaking of this project, and it is intended that the fruits of this project can give something back to the scientific community also.

While the scope of the library is evidently limited in certain areas, given the time constraints of the project, the satisfaction of the requirements of accepting both complex elementary functions and (in part) algebraic functions as input forms a strong basis for future work to build upon. In terms of software development, the code was written with extensibility and efficiency in mind throughout the whole process, and there are many obvious 'entry points' deliberately left for extension and/or improvements for those with an interest in the subject.

In terms of language development, it may be worthwhile paying some attention towards particular methods employed in the implementation, as many of these techniques are not only applicable to complex analysis but reach more generally into the field of geometry as a whole. Some parts of the functionality were written in light of deficiencies in the native language, and so it is suggested that these points be considered for inclusion as improvements to the language itself.

It is clear that development on Riemann surfaces is still at an early stage, with even basic literature being sparse and incoherent. Given the overall simplicity of the library,
it is sincerely hoped that this can be used as an educational tool for giving an insightful introduction to the concept of multi-valued functions to students yet to appreciate the beauty of its geometry.
Bibliography


35

Appendix A

Output images

Figure A.1: $F(w, z) = w^2 - z$

Figure A.2: $F(w, z) = w^3 - z$
Figure A.3: $F(w, z) = w^4 - z$

Figure A.4: $F(w, z) = \sin(w) - z$

Figure A.5: $F(w, z) = \tan(w) - z$
Figure A.6: $F(w, z) = \csc(w) - z$

Figure A.7: $F(w, z) = \cot(w) - z$

Figure A.8: $F(w, z) = \cosh(w) - z$
Figure A.9: \( F(w, z) = \exp(w) - z \)

Figure A.10: \( F(w, z) = \ln(w) - z \)

Figure A.11: \( F(w, z) = w^3 - w - z \)