Branch Cuts in Computer Algebra

Adam Dingle
Richard Fateman
Computer Science Division, EECS Dep't
University of California at Berkeley

Abstract

Many standard functions, such as the logarithm and square root functions, cannot be defined continuously on the complex plane. Mistaken assumptions about the properties of these functions lead computer algebra systems into various conundrums. We discuss how they can manipulate such functions in a useful fashion.

1 Introduction

Many standard functions, such as the logarithm and square root functions, cannot be defined continuously on the complex plane. When working with such functions, arbitrary lines of discontinuity, or *branch cuts*, must be chosen. For example, the conventional branch cut for the complex logarithm function lies along the negative real axis, so that $\log(-1) = \pi i$ but when ϵ_1 is small, real, and positive, we require $\log(-1-\epsilon_1 i) = -\pi i + \epsilon_2$ for some small, complex ϵ_2.

Most computer algebra systems provide little assistance in working with expressions involving functions with complex branch cuts. Worse, by their ignorance of the existence of branch cuts, algebra systems sometimes produce incorrect answers. In this paper, we will show how an expression's branch cuts may be mechanically computed, and how an expression with branch cuts may sometimes be simplified within each of its branches. Even when an expression cannot be simplified, its branch cuts may yield useful geometric insights. We will focus our discussion on functions of one complex variable.

We will sometimes be informal about functions in this paper, representing a function such as $\lambda z.z+1$ as the expression $z+1$. The interpretation of an expression as either an expression or function will be contextually clear.

2 Domain of Mathematical Expressions

In the sections that follow, we will assume that there is some set $\mathcal{P}$ of complex functions which we call *primitive functions.* For the sake of discussing common branch cut shapes such as lines and circles, we assume that $\mathcal{P}$ contains at least the addition, multiplication, and exponential functions, as well as the complex constants (which we can consider to be functions of zero arguments). Let $\mathcal{P}^*$ be the closure of $\mathcal{P}$ under functional composition. We assume furthermore that the branch cuts of each primitive function are known, and that we have an equation solving mechanism which can sometimes invert functions in $\mathcal{P}^*$. Below, we will develop a procedure which will be able to find the branch cuts for some functions in $\mathcal{P}^*$. Our results are sufficiently general that they do not for the most part depend on the exact set of functions in $\mathcal{P}$.

The procedures discussed below are suitable for implementation in a computer algebra system. Typically, a computer algebra system will have a set of primitive functions from which complex expressions may be built; these primitive functions will form the set $\mathcal{P}^*$. In our Mathematica [10] implementation, $\mathcal{P}$ contains the `Power` and `Log` functions, which have branch cuts, as well as all Mathematica functions which do not have branch cuts, including the addition and multiplication functions and trigonometric functions. It would not be difficult to add other primitive functions with branch cuts to the Mathematica implementation; adding inverse trigonometric functions such as `ArcTan` to the system would not increase its power, since such functions can be defined in terms of `Log` anyway. (In fact, `Power` can also be defined in terms of `Log`, but we found it necessary to add `Power` since Mathematica sometimes simplifies expressions containing `Log` to equivalent expressions containing `Power`.)

3 Branch Cut Representation

In this section we will develop a mathematical representation for branch cuts; the representation will be useful for storing branch cuts in a computer algebra system.

In later sections we will need to perform the following operations on branch cuts:

1. Find the image of a branch cut under an algebraic transformation.
2. Determine whether two branch cuts overlap, and where they overlap.
3. Determine whether two branch cuts intersect, and where they intersect.

Unless the set $\mathcal{P}$ of primitive functions is trivially small, branch cuts may be arbitrary algebraic curves, and we cannot in general expect to find their overlaps and intersections mechanically. By using algebraic knowledge to convert

Submitted to 1994 International Symposium on Symbolic and Algebraic Computation, Oxford (UK), July, 1994

branch cuts to canonical forms, we may determine the overlapping sections of some simple branch cuts. Similarly, by using the equation-solving knowledge present in a computer algebra system, we may determine the intersection points of some simple branch cuts.

Since branch cuts may take many geometric shapes, such as lines, circles, other conic sections, or wedges, we might choose to represent a branch cut as the name of a known shape, along with parameters for that particular shape. For example, a circle might be represented by the symbol "circle" along with its center and radius. Harlan Seymour describes such a representation in his master's thesis [9]. Seymour's library is able to derive conformal mappings that transform one given shape into another given shape. Our problem is more difficult: given a shape S and a mapping M, we need to be able to recognize the shape to which S is mapped under M. In particular, we may encounter branch cuts which might not be representable in any fixed library of geometric shapes; in such a situation we would like our system to degrade gracefully, representing the unfamiliar branch cuts in some way.

We will adopt a more general approach, representing a branch cut as the image of a real interval under an algebraic transformation in $\mathcal{P}^*$; this will allow us to represent branch cuts which might not be representable in a given library of geometric shapes. A branch cut will be represented as a triple (r_0, r_1, f), where r_0 and r_1 are real numbers or positive or negative infinity, and f is a function mapping each point in the interval $[r_0, r_1]$ to the complex plane. For example, we may represent the standard branch cut of the log function as $(-\infty, 0, \lambda z.z)$; the branch cut of $\log(z+i)$ is $(-\infty, 0, \lambda z.z - i)$. When we map the cut (r_0, r_1, f) under the transformation g, we obtain the cut $(r_0, r_1, g \circ f)$. Notice that branch cuts' representations are not unique; for example, $(-\infty, 0, \lambda z.z{+}1)$ is equivalent to $(-\infty, 1, \lambda z.z)$. As another example, $(0, \infty, \lambda z.z^3)$ is equivalent to $(0, \infty, \lambda z.z)$.

3.1 Branch cut simplification

In this section we will define a set of simplification rules for branch cuts. Given two simple branch cuts that overlap but which have different algebraic representations, our simplification rules will sometimes map the branch cuts into similar forms, which will allow us to detect the overlap between the branch cuts. For common shapes such as lines and arcs, we will define *canonical forms*; we will design our simplification rules so that in simple cases a shape will be mapped to its canonical form. (Because branch cuts may have arbitrarily complicated algebraic representations, we cannot hope to map shapes to their canonical forms in general.) We will also show how to construct branch cut simplification rules from conformal mapping rules. Our approach is simple, and does not properly handle algebraic features such as singularities.

First, here are some general rules for simplification. If (r_0, r_1, f) is equivalent to (r_0', r_1', f'), then $(r_0, r_1, g \circ f)$ is equivalent to $(r_0', r_1', g \circ f')$; this allows us to perform simplifications on subexpressions of transformations. When given a branch cut $B = (r_0, r_1, f)$ to simplify, we decompose f into a composition of functions $f = f_1 \circ \ldots \circ f_n$. If any rules can be used to simplify (r_0, r_1, f_n) to (r_0', r_1', f_n'), we let B' $= (r_0', r_1', f_1 \circ \ldots \circ f_n')$, and (recursively) invoke our simplification procedure on B'. If no rules will simplify (r_0, r_1, f_n), we attempt to simplify each $(r_0, r_1, f_i \circ \ldots \circ f_n)$ in turn for smaller and smaller values of i; if each of those simplification attempts fails, we have no way of simplifying B.[1]

To ensure that $r_0 \le r_1$ for every cut (r_0, r_1, f), we map (r_0, r_1, f) to (r_1, r_0, f) if $r_0 > r_1$. If $f(x)$ is a continuous real function whose minimum and maximum values within the interval $[r_0, r_1]$ are m_0 and m_1, respectively, then we may simplify $(r_0, r_1, \lambda x.f(x))$ to $(m_0, m_1, \lambda x.x)$; we call this simplification the *real function rule*. For example, consider the cuts $(0, \infty, \lambda x.x^3)$ and $(0, \infty, \lambda x.x)$; in the previous section we claimed that these two cuts are equivalent. Since the minimum and maximum values of $\lambda z.z^3$ in the real interval $[0, \infty]$ are 0 and ∞, respectively, the real function rule maps $(0, \infty, \lambda z.z^3)$ to $(0, \infty, \lambda z.z)$.

3.1.1 Lines and segments

We define the canonical form for a line segment, ray, or line to be a cut of the form $(r_0, r_1, \lambda z.uz + n)$, where $|u| = 1$, $0 \le \arg u < \pi$ and n is normal to u. It is not difficult to see that any line segment, ray or line has a unique representation in this form.

We may convert any cut of the form $(r_0, r_1, \lambda z.w_1 z + w_2)$, where w_1 and w_2 are arbitrary complex numbers, to its canonical form by decomposing w_2 into the sum $w_2 = p + l$, where p is parallel to w_1 and l is orthogonal to w_1. Let u be the complex number such that $|u| = 1$, $0 \le \arg u < \pi$ and u is parallel to w_1; write $w_1 = w_1' u$ and $p = p' u$, where w_1' and p' are real. The given cut may now be rewritten as $(r_0, r_1, \lambda z.u(w_1' z + p') + l)$, which may be put into its target canonical form using the real function rule. If a line is mapped under a linear transformation, the conversion we have just outlined will always reduce the result to a line in canonical form.

When a line or segment $(r_0, r_1, \lambda z.uz + n)$ is taken to a real power, the result will be a line or segment, or a pair of lines or segments, if $n = 0$; the following rules ensure that the result will reduce to canonical form. We map $(r_0, r_1, \lambda z.(uz)^r)$ to $(r_0, r_1, \lambda z.u^r z^r)$; u^r will be of unit length, and z^r will be further reduced under the following rules. If r is an integer, then z^r is a real function, so (r_0, r_1, z^r) will be reduced by the real function rule. If $r_0 \ge 0$ and $r_1 \ge 0$, then (r_0, r_1, z^r) may be reduced to (r_0^r, r_1^r, z). If $r_0 \le 0$ and $r_1 \le 0$, then (r_0, r_1, z^r) may be reduced to $(-r_0, -r_1, (-1)^r z^r)$, and then the preceding rule will apply. If $r_0 \le 0$ and $r_1 \ge 0$, and r is not an integer, then we break the cut (r_0, r_1, z^r) into the cuts $(r_0, 0, z^r)$ and $(r_1, 0, z^r)$, which will simplify using the rules already given.

3.1.2 Circles and arcs

The canonical form for a circle centered at the origin is defined to be $(-\pi, \pi, \lambda z.re^{iz})$. The canonical form for an arc of a circle centered at the origin is $(r_0, r_1, \lambda z.re^{iz})$, where $0 < r_1 - r_0 < 2\pi$ and $0 \le r_1 < 2\pi$. Therefore, to simplify $(r_0, r_1, \lambda z.e^{iz})$ with $r_1 > r_0$, we first test to see if $r_1 - r_0 \ge 2\pi$; if so, the cut simplifies to $(-\pi, \pi, e^{iz})$, representing the unit circle. Otherwise, we let $r_0' = r_0 \bmod 2\pi$ and $r_1' = r_1 \bmod 2\pi$. If $r_0' > r_1'$, we correct r_0' by letting $r_0'' = r_0' - 2\pi$; otherwise let $r_0'' = r_0'$. Then the cut simplifies to (r_0'', r_1', e^{iz}).

This exponential form is convenient for expressing arcs, but inconvenient for solving equations—both because equations involving angles generally have a multiplicity of solutions and because the built-in equation solvers of systems such as Mathematica often cannot solve such equations.

[1] The current Mathematica implementation, described below, attempts to simplify (r_0, r_1, f_n) only.

Accordingly, when our Mathematica implementation (discussed below) needs to solve an equation involving complex exponentials, it converts the equation to a pair of equations in real variables representing the real and imaginary parts of the first equation (this occurs in the functions `Intersect` and `RealSolve`). In this way our implementation can find, for example, the intersection points of a line and a circle.

3.1.3 Adding rules for conformal mappings

In general, if M is a conformal map that maps a shape S_1 into the shape S_2, we would like $M(B_1)$ to simplify to B_2, where B_1 and B_2 are canonical forms for the shapes S_1 and S_2. Kober's dictionary of conformal mappings [6] describes the effect that common mappings have on common shapes. We can add algebraic information from the dictionary to our system in the form of rules that ensure that when shapes are mapped, they will simplify to canonical form.

For example, section 3.3 in Kober's dictionary indicates that the general bilinear transformation $w = az + b/cz + d$ maps the circle $|z - z_0| = r$ (with $|z_0 + d/c| \neq r$) to the circle $|w - w_0| = R$, where

$$w_0 = \frac{(az_0 + b)(\bar{c}\bar{z_0} + \bar{d}) - a\bar{c}r^2}{|cz_0 + d|^2 - |c|^2 r^2}$$

and

$$R = \frac{r|ad - bc|}{||cz_0 + d|^2 - |c|^2 r^2|}$$

We add a cut simplification rule which maps the cut

$$(-\pi, \pi, \lambda z.\frac{a(re^{iz} + z_0) + b}{c(re^{iz} + z_0) + d})$$

to $(-\pi, \pi, \lambda z.Re^{iz} + w_0)$. Adding several more rules will allow the system to simplify any branch cut that involves mapping a line or circle by bilinear transformations.

3.2 Branch cut inversion

As discussed above, we may find the inverse image of a branch cut B under a mapping f by finding the image of B under each inverse function of f. Typically, we can use the equation solving facility of a computer algebra system to invert f: we solve the equation $f(z) = w$ for the variable z. A complication that arises in practice is that f may have an inverse function which is defined only on a portion of the complex plane. For example, in the computation of the branch cuts of $\sqrt{\sqrt{z} - 2}$, we must find the inverse image of the ray $[-\infty, 2]$ under the mapping $f(z) = \sqrt{z}$. As it turns out, $f^{-1}(z) = z^2$ is an inverse for f when $-\pi/2 < \arg(z) <= \pi/2$, but when z is in the left half-plane, $f(w) = z$ has no solution.

If an algebra system's equation solver can generate inequalities which indicate when a solution exists for a particular equation, we can use those inequalities to remove portions of branch cuts where an inverse function is undefined. In the above example, for instance, we find that the the inequality $-\pi/2 < \arg(z) <= \pi/2$ holds only on the subinterval $[0, 2]$ of $[-\infty, 2]$, so the inverse image of $[-\infty, 2]$ under $\sqrt{z}$ is the branch cut $(0, 2, z^2) = (0, 4, z)$.

3.3 Checking for overlap

Once we have placed branch cuts into canonical form, we can often tell if two branch cuts overlap: (r_0, r_1, f) overlaps with (r_0', r_1', f) if the intervals $[r_0, r_1]$ and $[r_0', r_1']$ overlap. This test is necessary for combining branch cuts to eliminate them, as discussed above, and is adequate for combining many simple sorts of branch cuts.

4 Branch Cut Computation

In this section we discuss methods for computing the branch cuts of a function of one complex variable. The branch cuts are interesting in their own right, and will also lead to algebraic simplifications within individual branches. In the discussion that follows, we assume that if f is a primitive function with branch cuts, then either f is a function of a single complex variable, or f has branch cuts only in its first argument (that is, f is continuous in each of its other arguments). This assumption has not proved to be restrictive. In our Mathematica implementation, the only functions in $\mathcal{P}$ that have branch cuts are the `Log` and `Power` functions; `Log` is a function of one variable and `Power` is continuous in its second argument (since $x^y = e^{y\log(x)}$).

4.1 Branch cut enumeration

Here is a procedure which will find all possible branch cuts of a function $E(x) \in \mathcal{P}^*$. First, if $E(x)$ is a primitive function of x, its branch cut may be looked up in a table. In our Mathematica implementation, the branch cut table contains the function `Log` and its branch cut $(-\infty, 0, \lambda x.x)$, and the function `Power` and its identical branch cut $(-\infty, 0, \lambda x.x)$.

If $E(x) = f(g_1(x), g_2(x), \ldots, g_n(x))$ where f is a primitive function, we first (recursively) compute the branch cuts of g_1 through g_n; then the branch cuts of E are, at most, the branch cuts of g_1 through g_n along with those points which g_1 maps onto the branch cuts of f. (Recall that f may have branch cuts only in its first argument.) In general, g_1 may be hard to invert, so it will sometimes be difficult to obtain a constructive representation for the branch cuts of the composite function E; in practice, a computer algebra system may sometimes be able to invert g_1 without human intervention. We will discuss the problem of branch cut inversion in greater detail later in this paper.

4.2 Branch cut elimination

The procedure sketched above will find all possible branch cuts for a complex function, but may return some *removable branch cuts*: branch cuts across which the function is not actually discontinuous. For example, consider the function $f(z) = \log(z + 1) - \log(z - 1)$. The above procedure will compute that the branch cut of $\log(z + 1)$ is the interval $[-\infty, -1]$ and that the branch cut of $\log(z - 1)$ is the interval $[-\infty, 1]$, and conclude that the branch cut of $f(z)$ is $[-\infty, 1]$. As it turns out, $f(z)$ is only discontinous across the interval $[-1, 1]$; $f(z)$ is continuous across $[-\infty, -1]$, which is a removable branch cut. We will modify our procedure so that it will eliminate many removable branch cuts; our modified procedure will return $[-1, 1]$ as the branch cut of $f(z)$.

4.2.1 Terminology

First we introduce some informal terminology. When we define a primitive function that has a branch cut, we must choose one side of the branch cut with which the value of the function on the branch cut is to agree; this choice is arbitrary and is made subject to certain conventions and heuristics.

For example, $\log(-1)$ is arbitrarily chosen to be πi, which is continuous with the values of the log function immediately above the negative real axis on the complex plane; by a different convention $\log(-1)$ could be chosen to be $-\pi i$, which would agree with the values below the branch cut. We say that the *alternate function* g for a function f relative to a branch cut B is identical to f except on the branch B, where f and g take on values agreeing with different sides of the branch cut. The *alternate branch function* for a function f relative to a branch cut B is like the alternate function, but is defined only on the branch cut itself. The alternate branch function for a function f can often be defined in terms of f itself. For example, if $f(z) = \log(z) - 2\pi i$ and is defined only when z on the negative real axis, then f is the alternate branch function for log.

4.2.2 The procedure

Here is a sketch of a modified branch cut computation procedure, which will eliminate removable branch cuts. We recursively compute, for the given expression $E(z)$ and for each of its subexpressions, a list of possible branch cuts and also, for each branch cut, an alternate branch function for the cut. If $E(z)$ is a primitive function, we can look up its branch cuts and alternate branch functions in a table as before.

If $E(z) = f(g_1(z), \ldots, g_n(z))$ where f is a primitive function, then we invoke the procedure recursively on each g_i, returning a set G_i of pairs (B, b) for each subfunction g_i, where B is a branch cut and b is an alternate branch function for B. Then, for each branch cut and corresponding alternate branch function (B, b) of f we compute the set B' of cuts which g_1 maps to B; for each cut C in B' we construct the pair (C, b); let F be the set of all such pairs.

We now construct a set S of branch cuts from the cuts in the sets G_i and F by combining branch cuts wherever they overlap; we may need to break some of the branch cuts into smaller pieces in order to do this. For example, if G_1 contains the branch cut $[-4, 4]$, an interval on the real axis, and G_2 contains the branch cut $[-2, 6]$, then S will contain the branch cuts $[-4, -2]$, $[-2, 4]$ and $[4, 6]$.

Now we construct an alternate branch function for each branch cut B in S. Each branch cut B has an alternate branch function $f' \circ (h_1, \ldots, h_n)$, where f' and the h_i are defined as follows. If B derives from a cut C that appears in a pair (C, b) in F, then $f' = b$; otherwise $f' = f$. (We say that a branch cut derives from each of the branch cuts that overlapped to form it.) If B derives from a cut C that appears in a pair (C, b) in some G_i, then $h_i = b$ (which is the alternate branch function of B in g_i); otherwise $h_i = g_i$.

Finally, we test each branch cut B in S to see if its alternate branch function is algebraically equivalent to f; if so, B is removable. Eliminating the removable branch cuts in S, we are left with the set of branch cuts of E.

The procedure above will necessarily fail to remove some removable branch cuts, because it depends on being able to tell when branch cuts overlap and when functions are algebraically equivalent. Nevertheless, the algorithm will compute an exact set of branch cuts for many simple functions; and, like the preceding procedure, will never miss actual branch cuts.

4.3 An example

For example, consider the computation of the branch cuts of the function $E(z) = \log(z+1) - \log(z-1)$; E is recursively expressed as $h(g_1(z), g_2(z))$ where h is the subtraction function, $g_1(z) = \log(z+1)$ and $g_2(z) = \log(z-1)$. The procedure is recursively invoked on the function g_1, which is of the form $f(g(z))$ where $f = \log$ is primitive. $g = \lambda z.z+1$ has no branch cuts; f has the single branch cut $B = [-\infty, 0]$ with corresponding alternate branch function $b(z) = \log(z) - 2\pi i$. Mapping the cut B under the g^{-1}, we obtain the cut $B' = [-\infty, -1]$ with corresponding alternate branch function $b \circ g = \log(z+1) - 2\pi i$. Similarly, the algorithm is invoked on $\log(z-1)$, returning the cut $[-\infty, 1]$ with alternate branch function $\log(z-1) - 2\pi i$. We determine that the intervals $[-\infty, -1]$ and $[-\infty, 1]$ overlap, and break them into the two intervals $[-\infty, -1]$ and $[-1, 1]$. Since $[-\infty, -1]$ derives from both g_1 and g_2, its alternate branch function is $f \circ (\lambda z.\log(z+1) - 2\pi i, \lambda z.\log(z-1) - 2\pi i) = (\log(z+1) - 2\pi i) - (\log(z-1) - 2\pi i) = \log(z+1) - \log(z-1)$. Since $[-1, 1]$ derives only from g_2, its alternate branch function is $f \circ (\lambda z.\log(z+1), \lambda z.\log(z-1) - 2\pi i) = \log(z+1) - log(z-1) + 2\pi i$. When we trim the set of branch cuts, $[-\infty, -1]$ vanishes because its alternate branch cut is identical to the function E itself; we are left with the single branch cut $[-1, 1]$.

5 Implementation in Mathematica

We have implemented the branch cut computation algorithm in Mathematica; the code is available from the authors. The Mathematica function `BranchCuts` is used to find the branch cuts of a function of a single complex variable. Given such a function, `BranchCuts` returns a list of branch cuts, each of which has the form `Cut[`r_0, r_1, f`]`, representing the range of the complex function f over the real interval (r_0, r_1).

The implementation can find the branch cuts of many simple functions. For example, exercise 13a on page 24 of Carrier, Krook, and Pearson's complex analysis text [1] asks the reader to find the branch cuts of the complex function $\sqrt{1+\sqrt{z}}$. That function is represented in Mathematica as `Fn[z, Sqrt[1 + Sqrt[z]]]`, so we can pose the branch cut problem to the `BranchCuts` function as follows:

```
In[6]:= BranchCuts[Fn[z, Sqrt[1 + Sqrt[z]]]]

Out[6]= {Cut[-Infinity, 0, Identity]}
```

Mathematica reports that the function has the single branch cut $(-\infty, 0, \lambda z.z)$, which is the negative real axis. A bit of thought reveals that this is indeed the only branch cut, since the function $1 + \sqrt{z}$ does not map any values z to the branch cut of $\sqrt{z}$.

The corresponding exercise 13b asks for the branch cuts of the function $\log 1 + \sqrt{z^2+1}$; our implementation can solve this problem as well:

```
In[7]:= BranchCuts[Fn[z, Log[1 + Sqrt[z^2 + 1]]]]

Out[7]= {Cut[-Infinity, -1, I #1 & ],

>  Cut[1, Infinity, I #1 & ]}
```

The branch cuts reported are the imaginary intervals $[i, i\infty]$ and $[-i, -i\infty]$. (The notation `I #1 &` is a Mathematica shorthand for $\lambda z.iz$.)

Our implementation will eliminate many removable branch cuts, such as in this example:

```
In[8]:= BranchCuts[Fn[z, Log[z + 1] - Log[z - 1]]]

Out[8]= {Cut[-1, 1, Identity]}
```

5.1 Difficulties in implementation

Several aspects of Mathematica posed difficulties for our implementation. Perhaps the most severe was the lack of support for the manipulation of inequalities: neither the built-in facilities of Mathematica nor any of the supplied packages are able to simplify expressions involving inequalities (for example, simplifying `4 <= x <= 8 && 5 <= x <= 10` to `5 <= x <= 8`), or to find ranges of real numbers for which a given inequality holds; these operations are necessary for branch cut inversion. Worse, Mathematica's equation solver does not generate inequalities that represent when an equation has a solution. For example:

```
In[13]:= Solve[Sqrt[x] == a, x]

                    2
Out[13]= {{x -> a }}

In[14]:= Reduce[Sqrt[x] == a, x]

                 2
Out[14]= x == a
```

Neither the `Solve` nor `Reduce` commands report that their solution is valid only when a is in the right half-plane.

Other basic mathematical facilities are absent as well. Mathematica does not seem to have a mechanism for finding the maximum and minimum of a polynomial or other real function within a given interval, which is useful for branch cut simplification. The `Solve` program cannot find real roots of complex functions involving the argument function; for example, it cannot find the real root $x = 2$ of the simple equation $\arg(x + 2i) = \pi/4$, which is essential for finding the portion of a branch cut which can be inverted under a transformation such as $\sqrt{z}$.

We were able to circumvent the above difficulties by implementing the missing features; we feel that most features we needed are basic enough that they should be present in packages in the standard distribution, if not in the core language itself.

Another difficulty we encountered was that Mathematica's evaluation mechanism cannot always be easily modified for its built-in functions, so that it is difficult to change Mathematica's notion of a canonical form. For example, the expressions `Sqrt[x]`, `x^(1/2)` and `Exp[1/2 Log[x]]` are mathematically equivalent; Mathematica reduces each of them to `x^(1/2)`. Since exponentiation is the only built-in function of two complex arguments that has a branch cut, we hoped to rewrite each exponentiation operation in terms of the exponential and logarithm functions, which would simplify the branch-cut computation code. Since we could not convince Mathematica to reduce `x^(1/2)` to `Exp[1/2 Log[x]]`, however, we had to write our branch-cut code to be able to handle functions of more than one argument.

6 Computation of Branch Regions

The procedures presented thus far can find the branch cuts of a given complex expression. We now turn to the problem of finding the regions into which an expression's branch cuts divides the complex plane. In particular, we would like to be able to compute the number of regions; we would also like to compute some representation of the boundaries of each region. Finally, we would like to be able to find a point in each region, which will be useful later, when we will show how the expression may sometimes be simplified in each of its branch regions. The procedure described in this section is complicated; an example will follow and may be enlightening.

6.1 Eliminating intersecting branch cuts

To determine the set of regions into which a set C of branch cuts divides the complex plane, we find all pairs of branch cuts that intersect. Intersection can be determined algebraically or perhaps numerically. If two branch cuts B_1, B_2 intersect at a point P, we split each B_i into two cuts at P if P is not an endpoint of B_i. We now have a set S of pairwise non-intersecting cuts, each of which is either a closed curve or a path between two points (either or both of which may be at infinity). We now discard any cuts which have a non-infinite endpoint to which no other cut is connected, since such cuts do not separate branches of the plane.

For each endpoint e, at least two cuts have e as an endpoint; we compute the angle at which each of those cuts leaves e. In the branch cut representation which we have chosen, each cut is represented as the range of a function f on a real segment (t_0, t_1); the angle at which the cut leaves the endpoint $f(t_0)$ is $\arg(f'(t_0))$. For each finite endpoint e, we sort the cuts which have e as an endpoint by the angle at which they leave e. It is possible that two or more cuts may leave e tangentially and thus at the same angle a; in such a case we sort those cuts by computing $r_i = \frac{f_i''(e) \cdot f_i'(e)}{|f_i'(e)|^2}$ for each cut i and comparing the r_i; r_i measures the "twist" of f_i at e. (In this context, $(a+bi)\cdot(c+di)$ for real a, b, c, d means $ac + bdi$; this is like the dot product of two vectors.)

We would also like to sort the segments which have endpoints at infinity around the point at infinity. To do so we consider the image of the segments under the mapping $f(w) = 1/w$; the segments may now be sorted according to the angle at which their projections under f touch the origin, which is the image of the point at infinity under the mapping f.

6.2 Traversing the branch regions

Let R be a set containing an arbitrary endpoint e_1 (the endpoint at infinity, if there is one, is convenient). Choose any cut s_1 in S that has e_1 as an endpoint. Let e_2 be the other endpoint of s_1. Moving counterclockwise around e_2 from the cut s_1, let s_2 be the next cut that we encounter. Let e_3 be the other endpoint of s_2. We proceed similarly until we return to e_1, at which point we have traversed the perimeter of a branch region in a clockwise direction. If any of the e_i we encountered are not in R, we now add those e_i to R. Then we choose any endpoint e_1' in R, and any cut s_1' in S that has e_1' and another endpoint e_2' as endpoints, such that we have not already traversed s_1' from e_1' to e_2', and repeat the process, traversing another region and possibly adding more endpoints to R. We continue until we have exhausted the process, namely that until we find that in our traversals we have left every endpoint e in R from each cut that has e as an endpoint.

In many cases a single traversal will determine the set of regions into which the original set C of branch cuts divide the complex plane; the traversal process will have exhausted the set S. In some cases, however, some branch regions may be contained entirely within other regions, and so the traversal will not find all regions. For example, if C consists of two concentric circles C_1 and C_2, with C_1 containing C_2, the first traversal will discover only one of the two circles. In such cases, the problem of discovering which branch regions are contained in other regions is beyond the scope of this paper.

6.3 Choosing points in each region

To choose points that are in a region R, we select any endpoint e on the boundary of R; let s and s' be the cuts on the boundary of R that have e as an endpoint. We compute the angle at which s and s' leave e, and draw a line L which bisects the angle formed as s and s' leave e. We compute all intersections of L with the boundary of R; let e' be the intersection nearest to e, moving from e in the direction of the interior of R. Now we can choose any point on the line segment (e, e') as a point in R.

6.4 A simple example

We will illustrate how the above procedure computes the branch regions of the function $f(z) = \ln((z+i)(z-i)) - (\ln(z+i) + \ln(z-i))$. Our branch cut procedure tells us that the branch cuts of $f(z)$ are the lines $[-\infty, i]$ (using interval notation a bit loosely - this indicates a line from $-\infty$ to i or, more precisely, the set $\{x + i| -\infty < x \leq 0\}$), $[-\infty, -i]$, $[i, i\infty]$ and $[-i, -i\infty]$. The first two branch cuts are the branch cuts of $\log(z-i)$ and $\log(z+i)$, respectively; the last two are obtained by mapping the negative real axis under the inverses of $(z+i)(z-i)$. None of the branch cuts intersect at points other than their endpoints; the set of branch cut endpoints is $\{i, -i, \infty\}$. The branch cuts around ∞ are sorted in the order $[i, i\infty]$, $[-\infty, i]$, $[-\infty, -i]$, $[-i, -i\infty]$ (we must use the second derivative to properly order $[-\infty, -i]$ and $[-\infty, i]$).

We now traverse the branch cuts in S. Starting with the point at infinity, we traverse the regions R_1, bordered by $[i\infty, i]$ and $[i, -\infty]$; R_2, bordered by $[-\infty, i]$, $[i, i\infty]$, $[-i\infty, -i]$, and $[-i, -\infty]$; and R_3, bordered by $[-\infty, -i]$ and $[-i, -i\infty]$.

6.5 Implementation

We have implemented the region-finding procedure in Mathematica. The Mathematica function `Regions` maps a function to a list of its branch regions; each region is represented by a list of its boundaries. Each boundary is represented by a *segment*: a directed cut which is represented by a triple (r_0, r_1, f), where the cut is the range of f over the real interval $[r_0, r_1]$ and is directed toward the endpoint $f(r_1)$. The canonical form for a segment is the same as the canonical form for a cut, described above in the section "Branch cut simplification", except that r_0 may be less than r_1 in a canonical segment.

Our implementation can find the branch regions of the example given in the previous section:

```
In[6]:= Regions[Fn[z, Log[(z + I)(z - I)] -
                       Log[z + I] - Log[z - I]]]

Out[6]= {{Cut[-Infinity, 0, I + #1 & ],

>  Cut[1, Infinity, I #1 & ],

>  Cut[-Infinity, -1, I #1 & ],

>  Cut[0, -Infinity, -I + #1 & ]},

> {Cut[-1, -Infinity, I #1 & ],

>  Cut[-Infinity, 0, -I + #1 & ]},

> {Cut[0, -Infinity, I + #1 & ],

>  Cut[Infinity, 1, I #1 & ]}}
```

Again, `I + #1 &` is a Mathematica shorthand for $\lambda z.z+i$. The regions reported by Mathematica are, in order, R_2, R_3, and R_1.

The Mathematica implementation does not sort the endpoints at infinity; rather, it uses an earlier idea involving a large circle enclosing all of the finite endpoints.

7 Expressing Ambiguity of Branch Choice

We now turn to the problem of simplifying functions that have branch cuts. In general, simplifications which are sound for real expressions will not always work for similar complex expressions, since in the complex case the functions generally have branch cuts. For example, when a and b are real and $b > 0$ we have $\log b^a = a \log b$, which is not always valid on the complex plane.

7.1 `zeroOf` and `rootOf`, `Unln` and `Unbranch`

The simultaneous expression of several choices in a single symbol has been introduced in general computer algebra systems in several ways. Macsyma [7] allows certain computations to be done over algebraic number and algebraic function fields via the command `tellrat`. For example `tellrat(a^2=3)` specifies that $a^2 = 3$ without expressing an opinion as to whether a is positive or negative, or in general, real or complex. Nevertheless, expressions where only a^2 occur are unambiguous. A command `tellrat(y^2=x^2+1)` indicates an algebraic relationship between y and x. Such relationships are used only within Macsyma's rational function subsystem (`rat(..)`), and only then when a variable `algebraic` is set to `true`. This is not applied routinely throughout the system, and thus the system will not know for example, that $a^2 > 0$. Furthermore, Macsyma's built-in commands do not routinely *generate* forms for algebraic numbers for their own use or for the display of answers. If the solution of an algebraic equation cannot be expressed in terms of radicals, the `solve` program declines to solve it. (Adopting a variation of the solutions of Mathematica or Maple, described below, would probably be a quick but partial fix.)

The AXIOM system [3] provides `zeroOf` and `rootOf` to specify algebraic numbers within some arbitrary choice. AXIOM allows a command of the form `a := rootOf(a**4+1,a)`. The operation `zeroOf` is similar to `rootOf`, but will use radicals when possible. AXIOM differs from Macsyma in two important respects: the system will generate such expressions in the solution of systems of algebraic equations, and will (as a consequence of incorporating this information more fundamentally in its "kernel" operations) provide simplifications when appropriate, not just when explicitly requested. Nesting of algebraic extensions is supported, and no automatic attempt is made to reduce successive extensions to a primitive element. Note that solution of algebraic systems is used internally in AXIOM for other purposes, and therefore results from (in particular) the integration program may include `rootOf` expressions.

The Maple system [2] uses the operator `RootOf` to express the solution of algebraic equations in a similar manner to that of AXIOM. Maple allows the notation to be used for transcendental equations as in `RootOf(cos(x)=x,x)`, but appears to have no routines to manipulate such transcendental forms. In a manner similar to Macsyma, Maple requires special effort on the part of the user to specify simplification of such quantities, requiring the user to apply `evala` or `simplify/RootOf` to effect simplification of expressions with `RootOf`s. Maple does use `RootOf` for its own purposes, in a manner similar to that of AXIOM, and so new expressions may be generated. A finite sum over different roots is supported.

The Mathematica system [10] uses a convention where `Solve` can provide a set of algebraic rules for simplification (through rule application) or the user can define, with the assistance of `AlgebraicRules` a sequence of substitutions intended to simplify expressions. The rules must be explicitly applied to expressions for them to take effect. It appears that, except for `Solve` and `Reduce`, no programs produce `RuleSets`, and that (for example) integrations requiring algebraic expressions which are not expressible in terms of radicals are returned unintegrated.

With the exception of the slight concession of Maple to the need for expression of worse-than-algebraic roots, there are no facilities for dealing with multiple-valued non-algebraic expressions in any system, and with the possible exception of Axiom, even handling expression with algebraic extensions seems "tacked on". G

In unpublished work (but described in `sci.math.symbolic` Internet netnews (April 19, 1991), Charles Patton, Sam Dooley, and others have attempted to derive minimum "add-on" concepts that would permit computer algebra systems to deal with multiple values.

7.2 Simplification using Integer Rounding

Kahan has noted [5] that expressions involving functions with branch cuts can sometimes be written in a simpler form by using the nearest-integer function. For example,

$$\ln(e^z) = z - 2\pi i[\Im(z)/2\pi]$$

where $[a]$ denotes the integer nearest to a, rounding half-integers down, and where $\Im(z)$ denotes the imaginary part of z. Although the form on the right looks more complex, it involves no transcendental functions and may be much easier to compute.

When an expression involves terms v and w such that $e^v = e^w$, the expression may sometimes be simplified by applying to each term the substitution

$$z = \ln(e^z) + 2\pi i[\Im(z)/2\pi]$$

which is a simple variant of the identity above. For example, consider the expression $\ln(w^2) - 2\ln(w)$; notice that $e^{\ln(w^2)} = e^{2\ln(w)} = w^2$. Then we simplify as follows:

$$\begin{aligned}
\ln(w^2) - 2\ln(w) &= \ln(e^{\ln(w^2)}) + 2\pi i[\Im(\ln(w^2))/2\pi] && (1)\\
&\quad - \ln(e^{2\ln(w)}) - 2\pi i[\Im(2\ln(w))/2\pi] && (2)\\
&= 2\pi i[\Im(\ln(w^2))/2\pi] && (3)\\
&\quad -2\pi i[\Im(2\ln(w))/2\pi] && (4)\\
&= 2\pi i[\arg(w^2)/2\pi] - 2\pi i[\arg(w)/\pi] && (5)\\
&= -2\pi i[\arg(w)/\pi] && (6)
\end{aligned}$$

The last step can be performed in a computer algebra system by means of a simple rule that reduces $[\arg(z)/2\pi]$ to 0.

We have implemented the above substitution in Mathematica. The example above can be simplified automatically:

```
In[2]:= ExpTrans[Log[w^2] - 2 Log[w]]

                          Arg[w]
Out[2]= -2 I Pi Round[------]
                           Pi
```

7.3 Multiple Values

The expressions derived using the simplification of the previous section will often contain complicated instances of the nearest-integer function, which are difficult to analyze algebraically. In this section, we will see that each instance of the nearest-integer function is constant in each branch of the simplified expression. We may then ignore the argument of the nearest-integer function, and solve for its value in each branch.

Notice that $\Im(z)/2\pi$ is a half-integer exactly when e^z falls on the negative real axis, which is the branch cut of the ln function. It follows that within any particular branch of the function $\ln(e^z)$, the value of $[\Im(z)/2\pi]$ is constant. Hence, if we simplify an expression using the above identity, the value of the expression in a given branch can be obtained by replacing each occurrence of the greatest-integer function with a particular integer. Thus, we introduce a new (weaker) identity: $z = \ln(e^z) + 2\pi ik$ for some integer k. Simplifying an expression using this weaker identity will yield a set of expressions which represent possible values of the expression in each of its branches. We can derive similar "weak" identites for functions derived from the logarithm: for example, $z = (-1)^k\sqrt{z^2} = \pm\sqrt{z^2}$ for some integer k, and $\sqrt{ab} = \pm\sqrt{a}\sqrt{b}$ for some integer k.

To determine the value of k which is appropriate for an expression E in a given branch, we choose a point in the branch (as previously described) and solve numerically for k. If we are unlucky, the point we choose may not give us a unique value for k, in which case we choose another point and try again. If k has the same value v in all branches, then we may simplify E by replacing k with v.

7.4 An example

Consider the function $f(z) = \ln((z+i)(z-i)) - (\ln(z+i) + \ln(z-i))$, used as an example in the previous section. Notice that

$$e^{\ln((z+i)(z-i))} = e^{\ln(z+i)+\ln(z-i)} = (z+i)(z-i)$$

so we may use the identity $z = \ln(e^z) + 2\pi ik$ to simplify $f(z)$ to $2\pi ik$ for some integer k. Thus for each branch of the function $f(z)$ there is some integer k such that $f(z)$ is the constant $2\pi ik$ within that branch.

In the previous section we showed how to find the branch regions R_1, R_2, and R_3 of f, and showed that we can find a point in each region. We find a point p in R_1—say, $p = -2+2i$. We have $f(p) = 2\pi ik_1$; then $k_1 = f(p)/2\pi i$, and we can evaluate k_1 numerically:

```
In[13]:= f[z_] := Log[(z + I)(z - I)] -
                   (Log[z + I] + Log[z - I])

In[14]:= N[f[-2 + 2 I] / (2 Pi I)]

                           -17
Out[14]= -1. + 7.0679 10      I
```

We have found that $k_1 = -1$ (since we know that k_1 is an integer, we can discard the small error introduced by the numerical computation), and so $f(z) = -2\pi i$ when z is in R_1. Similarly, we find that $f(z) = 0$ when z is in R_2, and that $f(z) = 2\pi i$ when z is in R_3.

7.5 A challenge problem

Consider a problem posed by W. Kahan in [5] as a challenge for computer algebra systems: if $R(z) = (z + 1/z)/2$ and $S(w) = w + \sqrt{w+1}\sqrt{w-1}$, simplify $S(R(z))$. (Notice that S is a "weak inverse function" for R as $R(S(z)) = z$ for all z.) Expansion of $S(R(z))$ yields

$$S(R(z)) = \frac{z^2+1}{2z} + \sqrt{\frac{(z+1)^2}{2z}}\sqrt{\frac{(z-1)^2}{2z}} \quad (7)$$

$$= \frac{z^2+1}{2z} \pm \frac{(z+1)(z-1)}{2z} \quad (8)$$

This last expression simplifies by case analysis to z or $1/z$. We may conclude that $S(R(z))$ is equivalent either to z or to $1/z$ in each of its branches.

The procedures described in previous sections can determine (if an appropriate conformal-mapping rule is present) that the branch cuts of the function $S(R(z))$ are the unit circle and the negative real axis. These branch cuts partition the plane into two regions, inside and outside the unit circle testing a point in each region reveals that $S(R(z)) = z$ outside the circle and that $S(R(z)) = 1/z$ inside the circle. In this example there are no "unlucky" test points, since $z = 1/z$ only at $z = 1$ and $z = -1$, which are on the circle itself.

We may similarly determine how $S(R(z))$ may be simplified on each branch cut by testing a point on that cut, but we must be careful to keep individual branch cuts separate. Strictly speaking, the unit circle is not a single branch cut of $S(R(z))$ but actually two branch cuts, namely the lower and upper halves of the circle (respectively, the arcs $(\pi, 2\pi, e^{iz})$ and $(0, \pi, e^{iz})$), each generated from a separate inverse function of $R(z)$. It turns out that on the lower half of the unit circle, $S(R(z)) = 1/z$; on the upper half of the circle, $S(R(z)) = z$.

8 Conclusion

We have described and implemented procedures which can find the branch cuts of some simple functions, and which can sometimes simplify algebraic expressions within individual branch regions.

It seems a bit *ad hoc* to solve for the simplified expression that is appropriate in each branch of an expression. Might it be possible to derive the simplification algebraically for each branch?

The Mathematica implementation is at best a working prototype or a proof of concept: it may not work satisfactorily on problems other than simple examples. In particular, its knowledge of conformal maps could be greatly expanded, and many of the basic tools we must use are not adequate as given by the system. It would be nice to have an implementation of the algebraic simplification within branches.

In our implementation we do not distinguish between closed and open intervals. More careful attention to the endpoints of intervals would allow us to represent and manipulate singularities, which we can consider to be degenerate branch cuts.

A Acknowledgments

Large portions of this paper are based on Adam Dingle's Master's project [4] written under the direction of Richard Fateman, University of California at Berkeley. Prof. W. Kahan, the second reader for this report, also contributed helpful suggestions.

Adam Dingle was supported in part by a National Science Foundation Graduate Fellowship. This research was sponsored in part by the National Science Foundation Grant No. CCR-9214963 and NSF Infrastructure Grant number CDA-8722788.

References

[1] G. F. Carrier, M. Krook, and C. E. Pearson. *Functions of a Complex Variable: Theory and Techniques*, McGraw-Hill, 1966.

[2] B. W. Char, K. 0¿ Geddes, et al. *Maple V Library Reference Manual*, (and other volumes) Springer-Verlag, 1991.

[3] Richard D. Jenks and Robert S. Sutor. *AXIOM the Scientific Computation System.* NAG and Springer Verlag, NY, 1992.

[4] A. Dingle. "Branch Cuts in Computer Algebra," Master's Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1991.

[5] W. Kahan. "Instead of UNLN", unpublished paper, April 1991.

[6] H. Kober. *Dictionary of Conformal Representations*, Dover, 1957.

[7] Macsyma Inc. *Macsyma Reference Manual*, Version 14, 1991.

[8] Z. Nehari. *Conformal Mapping*, McGraw-Hill, 1952.

[9] H. Seymour. "Conform: A Conformal Mapping System", Master's Thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1985.

[10] S. Wolfram. *Mathematica, A System for Doing Mathematics by Computer*, Second edition, Addison-Wesley, 1991.