Finding Singularities via Symbolic Manipulation

May 1, 2019

Akash Gaonkar
akash.gaonkar@colorado.edu
Undergraduate Student, University of Colorado at Boulder

Valliappan Chidambaram
vach7169@colorado.edu
Undergraduate Student, University of Colorado at Boulder

Abstract

When working with large or complicated expressions, it can often be challenging to take the derivative by hand. We can compute a numerical approximation of the derivative at a certain point, but this loses the flexibility that an expression provides. This issue is solved in the real case by taking symbolic derivatives, which leave variables unsubstituted and produce a new expression representing the derivative. While similar techniques can be applied to take the derivative of complex functions, it is far more difficult because the derivative only exists where the expression is analytic. We consider a certain class of elementary functions and compositions of these functions. For any such expression, we give an algorithm that attempts to determine where the function is analytic and produces its symbolic derivative. We implement this algorithm and evaluate it over several examples, and compare it to existing methods of complex differentiation.

1 Introduction

A complex function is analytic on some region R if it has a Taylor series that converges to it pointwise on that region. Functions are differentiable on a region R if and only if they are analytic on that region. This means that to take the derivative of a function, you must first know if and where it is analytic. We don't allow expressions that contain functions that are analytic nowhere, like $\overline{z}$, $Re(z)$, and $Im(z)$, so we only need to find singular points of functions that are mostly analytic. Finding these singular points enables various useful things. For example, if you could classify singular points as isolated, branch, and cluster points, and you could calculate the residues at the isolated singular points, then it would be possible to use the Cauchy-Residue theorem and indented contours to calculate the integral of any closed loop in the complex plane on the allowed functions.

Finding the singular points of a symbolic function is difficult because it is difficult to calculate a Taylor series and test its convergence on a symbolic function. Even if that were possible, it would be difficult to find the singular points, the set of points on which a pointwise convergent Taylor series couldn't be found. It is possible to find the singularities of a function in different ways, using our knowledge of the singularities of simpler expressions, like our method described in Section 3. Other computer algebra systems, such as Maple and WolframAlpha use similar methods to calculate singular points in the functions they are given. Maple and WolframAlpha can both find all isolated singular points and branch points of arbitrary functions, although WolframAlpha reports singularities in terms of the inputs to functions (for example, the singularities for $sin(1/z)$ are reported in the $1/z$-plane) and very often fails. Our method allows the user to find all isolated singular points, branch points, and cluster points of functions that we can simplify correctly. When we are unable to solve for the roots of certain equations, we report the answer similar to WolframAlpha. Unlike Maple and WolframAlpha, we are unable to classify our singular points into poles and essential singular points, and we are unable to perform additional computations such as calculating residues at singular points. We are also unable to determine if infinity is a singular point, or the type of singular point at infinity.

Our achievements are:

- An algorithm and a new implementation that is able to find isolated singular points and branch points in terms of roots of functions.
- An algorithm and a new implementation that is able to find cluster points from singular and branch points.
- A new implementation that is able to take the derivative of a complex expression.

2 Expressions

$$\begin{aligned}
\langle \mathit{Expr} \rangle \ &::= \ z \mid \mathbb{C} \mid \sin(\langle \mathit{Expr} \rangle) \mid \cos(\langle \mathit{Expr} \rangle) \mid \exp(\langle \mathit{Expr} \rangle) \\
&\mid \ \log(\langle \mathit{Expr} \rangle) \mid \langle \mathit{Sum} \rangle \mid \langle \mathit{Term} \rangle \\
\langle \mathit{Sum} \rangle \ &::= \ \langle \mathit{Expr} \rangle \times \mathbb{C} \mid \langle \mathit{Expr} \rangle \times \mathbb{C} + \langle \mathit{Sum} \rangle \\
\langle \mathit{Term} \rangle \ &::= \ \langle \mathit{Expr} \rangle^{\mathbb{C}} \mid \langle \mathit{Expr} \rangle^{\mathbb{C}} \times \langle \mathit{Term} \rangle
\end{aligned}$$

Figure 1. The grammar for the allowed expressions

Our system allows the expressions defined by **Expr** in Figure 1. More specifically, we allow the complex variable z, complex numbers, sums, products, powers, sin, cos, log, exp, and compositions of these functions. Each of these expressions are analytic except for a countable number of singular points. The goal of our algorithm is to produce a list of the isolated singular points, branch points, and cluster points. For example, the function $1/log(z-1)$ has an isolated singular point at $z = 2$ and a branch point at $z = 1$. This is because expressions have isolated singular points where their denominators are 0, and $log(z-1) = 0$ when $z = 2$. Expressions also have branch points where the input to a log or non-integer power is zero or infinity, and $z - 1 = 0$ when $z = 1$, so that is a branch point. An example with a cluster point would be $1/cos(1/z)$. Its singularities can be found as follows:

$$cos\left(\frac{1}{z}\right) = 0 \implies \frac{1}{z} = \frac{\pi}{2} + n\pi \implies z = \frac{1}{\frac{\pi}{2} + n\pi},\ n \in \mathbb{Z}.$$

Because $lim_{n\to\infty} z = 0$, it means that there are an infinite number of singular points about $z = 0$, so it's a cluster point. It is also possible for there to be infinitely many cluster points. For example, the singular points of $1/sin(1/sin(z))$ can be found as follows:

$$sin\left(\frac{1}{sin(z)}\right) = 0 \implies \frac{1}{sin(z)} = n\pi \implies sin(z) = \frac{1}{n\pi}$$
$$\implies z = sin^{-1}\left(\frac{1}{n\pi}\right),\ n \in \mathbb{Z}.$$

Because $lim_{n\to\infty} z = sin^{-1}(0) = m\pi,\ m \in \mathbb{Z}$, there are an infinite number of cluster points. We have shown how to find the singularities of specific expressions, we will now show how to find the singularities for arbitrary expressions.

3 Analyticity of Compositions

We now consider how to systematically find where our expressions are analytic. In general, this would require solving the Cauchy Riemann equations for arbitrary functions, but the restrictions of our expression language allow us to use a more tractable approach.

Definition 1 (Countably invertible function)**.** We say a function $f : \mathbb{C} \to \mathbb{C}$ is *countably invertible* if the set of inputs to f that attain a given value is countable, i.e.

$$\forall w \in \mathbb{C}.\, \exists\, m_f(w) : \mathbb{N} \twoheadrightarrow \{z \mid f(z) = w\}.$$

For our purposes, we also include complex infinity, $\infty \in \mathbb{C}$, and refer to $m_f(w)$ as the counting function.

Note first that our atoms, z, sin, cos, log, $\cdots$ are countably invertible, with the exception of a constant expression. We will show that many combinations of expressions are also countably invertible, and use this invertibility to find singularities.

Lemma 1. *Scalings of countably invertible functions are countably invertible, i.e. if f is countably invertible then so is cf, assuming $c \neq 0$.*

Proof. Because f is countably invertible, it has a counting function $m_f(v)$. Then because $-f(w) = f(-w)$, we can construct $m_{-f}(w) = m_f(-w)$. □

Lemma 2. *Compositions $(f \circ g)$ of countably invertible functions are countably invertible.*

Proof. Let f, g be countably invertible. We know there exist $m_f(u) : \mathbb{N} \twoheadrightarrow \{z \mid f(z) = u\}$, and $m_g(v) : \mathbb{N} \twoheadrightarrow \{z \mid g(z) = v\}$. Then if we fix a $w \in \mathbb{C}$, we can define an onto mapping from $\mathbb{N}^2$ to our inverses,

$$m(w) : \mathbb{N} \times \mathbb{N} \twoheadrightarrow \{z \mid f(g(z)) = w\},$$
$$m(w, n_f, n_g) = m_g(m_f(w, n_f), n_g)).$$

Then because $\mathbb{N}^2$ is countably infinite, there is a bijection $p : \mathbb{N}^2 \leftrightarrow \mathbb{N}$, so we complete our proof by defining our counting function $m_{f\circ g}(w, n) = m(w, p(n))$. □

Lemma 3. *Complex powers of countably invertible functions are countably invertible, i.e. if f is countably invertible then f^c is countably invertible for $c \neq 0$.*

Proof. Note first that $f(z)^c = e^{c\log(f(z))} = e^c e^{\log(f(z))}$. Then because $f(z)$ is countably invertible, from Lemma 2 we know that $p = e^{\log(f(z))}$ has a counting function $m_p(v)$. Thus we can construct a counting function $m_{f^c}(w) = m_p(w/e^c)$. □

We now use countable invertibility to compute a result on the inverses of of expressions. Recall that for each atom $f \in (z, \sin(z), \cos(z), \cdots)$, the associated counting function $m_f(w, n)$ provides the inverses for f. For example,

$$m_{e^z}(w, n) = \log(w) + i2n\pi \quad ; \quad m_{cz}(w, n) = w/c.$$

Thus, by leaving n as a free variable, we have an expression that describes all inverses of the atom.

This can be generalized to any countably invertible expression.

Theorem 1. *Inverses of countably invertible expressions can be described as expressions with a finite number of free integer variables.*

Proof. We proceed by strong induction over the nesting depth of expressions.

For our base case, an expression f with nesting depth 0 has no subexpressions, i.e. it is one of our atoms. Then as we noted previously, its counting function is the inverse described as an expression, and the expression has at most 1 free integer variables.

Now consider an expression f with nesting dept n. Then it has subexpressions $f_1, f_2, \cdots, f_k$, each of which can have nesting depth up to $n - 1$. From the proofs of our earlier lemmas, we know that we can construct a counting function for f using only the counting functions of the subexpressions. Since $f_1, \cdots, f_k$ have nesting depth at most $n - 1$. By our inductive assumption, we know that their counting functions $m_{f_1}, \cdots, m_{f_k}$ are expressions with finite free integer variables $(n_{1,1}, \cdots, n_{1,j_1}), (n_{2,1}, \cdots n_{2,j_2}), \cdots, (n_{k,1}, \cdots, n_{k,j_k})$ respectively. Thus m is just an expression, and has $\sum_{i=1}^{k} j_i$ free variables. □

This result means we can invert many complicated functions, which then allows us to solve for singularities.

Isolated Singularities Recall that at an isolated singularity, an expression f is either complex infinity (∞) or undefined. In our system, complex infinity is part of $\mathbb{C}$, and our functions are always defined. Thus, f has a singularity where it is ∞. By Theorem 1, we construct f^{-1}, and we know there is a singularity at $f^{-1}(\infty)$.

Branch Points Branch points in our expressions are introduced by log and non-integer powers of expressions. Thus, we can search through our expression for instances of $\log(f : Expr)$, or $(f : Expr)^c$. By Theorem 1 we know f has an inverse, so we can compute $f^{-1}(0)$ and $f^{-1}(\infty)$ to produce expressions describing the branch points of the individual log or power. Then by listing all those expressions, we can describe all branch points of our initial $Expr$.

Cluster Points Having found expressions for isolated singularities and branch points, we may wonder whether there are cluster points among or created by our singularities.

Lemma 4. *If the expressions $f_1, \cdots, f_k$ describe isolated singularities and branch points of an expression, then any cluster point that can be derived using all of $f_1, \cdots, f_k$ can also be derived by only using some particular f_j.*

Proof. Suppose there was a sequence of $\{z_n\}_{n=1}^{\infty}$ of points converging to a different point z such that each z_i was described by at least one f_j. Then z would be a cluster point. Note that because $f_1, \cdots, f_k$ are a finite number of expressions, there must be at least one f_j such that an infinite number of points in our sequence $\{z_n\}$ are described by f_j (otherwise the sequence would be finite). This means that there is some sequence $\{z_{jn}\}_{n=1}^{\infty} \to z$ such that each z_{ji} is described by f_j. □

Lemma 5. *If the expression f containing free variables $n_1, \cdots, n_k$ describes a cluster point, then at least one of the free variables is ∞ at the cluster point.*

Proof. Suppose f described a cluster point c where no n_i was ∞. Since f describes a cluster point, there must be some sequence of $\{z_j\}$ that converges to that cluster point, and all z_j used $n_1, \cdots, n_k \leq M$. Then because $n_1, \cdots, n_k \in \mathbb{N}$, we know that there can be no more than kM elements in our infinite sequence. This is a contradiction, so there must be some n_i that grows to ∞. □

Combining these two lemmas we can describe all cluster points of our expression by taking each of our singularity expressions and setting each of their free variables to ∞, one at a time.

What about non-countably invertible expressions? It is easy to produce non-countably invertible expressions by using sums or products of expressions. We cannot use the above techniques on said expressions, i.e we cannot invert them. However, if we treat these sums or products as a single variable (effectively a u-substitution), then we can still solve for the sum/product rather than solving for z. We then leave that last bit of work to the user. As an optimization, for the specific case of finding where a product $e_1 e_2 \cdots e_n$ is zero or infinity, we can simply combine the results for each subexpression e_i.

4 Symbolic Root Finding

Our expressions are represented using something known as an abstract syntax tree (AST). ASTs are useful for representing things that can be written as grammars, such as the expressions we allow.

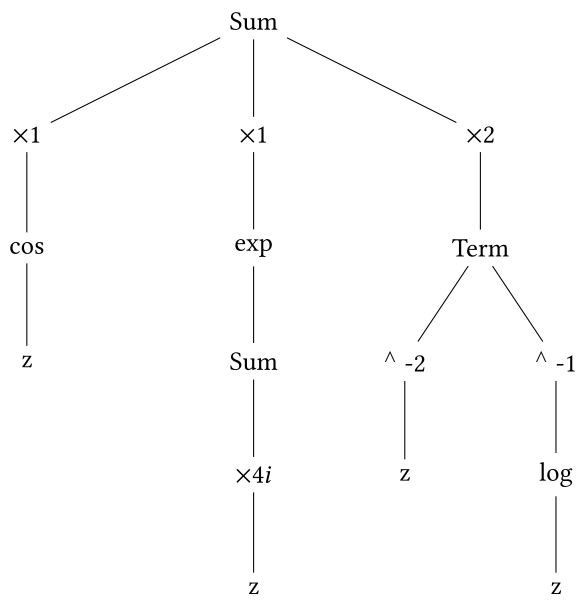


Figure 2. Example AST for $cos(z) + e^{4iz} + 2/(z^2 log(z))$. Note that Sum takes the sum of all of its children, and that Term takes the product of all of its children. Also note that multiplication by a constant takes place in Sums and not Terms.

As you can see in Figure 2, ASTs can show expressions in a surprisingly intuitive way. To find roots on an AST, the first thing we do is simplify it. We perform simplification on ASTs using rules. These rules say that if a subtree of the AST has some property, then part of it can be replaced with a simplified AST. Some of the rules we use are constant evaluation, removal of things with a coefficient of zero in a Sum, and removal of things with a power of one in a Term. We can't use all of the rules from the real numbers. For example, we allow the simplification $e^{log(z)} \to z$, but not $log(e^z) \to z$, because log is a multivalued function (we could simplify $log(e^z) \to z + 2n\pi i$, but this rule was never implemented). We don't have any simplification rules that implement factoring, so we can't successfully simplify the expression $z/(z^2 - z) \to 1/(z-1)$, so our algorithms produce incorrect results on such expressions.

We implemented a routine that tried to find where two expressions were equal. To find roots, we ran this routine to find where an expression equaled zero. Our solving routine worked by trying to solve the outermost expression on the right hand side, and calling itself recursively on the input to the outermost expression. For example, when trying to solve $sin(1/z) = 0$, we find that $sin(z) = 0$ when $z = n\pi$, and then we try to solve $1/z = n\pi$. This doesn't work when trying to solve Sums of more than one expression, or when trying to solve Terms for anything other than zero. We were unable to come up with an algorithm for doing this, so our algorithm simply reports the equations it was unable to solve. WolframAlpha and Maple have similar problems, but they are able to solve more equations with sums and products, and often report numerical solutions when they are unable to solve.

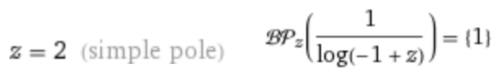
5 Results

Our program produced the correct results on everything we tested. Note that we don't keep track of everything in terms of πs, so they appear as floating point numbers in our solutions.

Our output:

```
Input:    log(-1.000+Z)^-1.000
Deriv:
-1.000*(log(-1.000+Z)^-2.000*(-1.000+Z)^-1.000)
Zeros:    List()
Singular: List(-2.000+Z)
Branch:   List(-1.000+Z)
Cluster:  List()
```

WolframAlpha output:



$z = 2$ (simple pole) $\quad \mathcal{BP}_z\left(\frac{1}{\log(-1+z)}\right) = \{1\}$

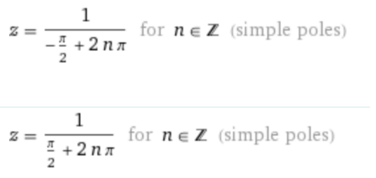
Figure 3. Our output and WolframAlpha's for $1/log(z-1)$.

Our program outputs zeros, singular points, branch points, and cluster points in terms of equations that should equal zero. This means that our program found a singular point at $z = 2$ and a branch point at $z = 1$, which is the same as the results from when we did it manually in Section 2, and the same as WolframAlpha's results.

Our output:

```
Input:    cos(Z^-1.000)^-1.000
Deriv:    -1*(cos(Z^-1)^-2*Z^-2*sin(Z^-1))
Zeros:    List()
Singular:
List(-1*((1.571+3.142*n1)^-1*(e^(-6.283i*n2)))+Z)
Branch:   List()
Cluster:  List(Z)
```

WolframAlpha output:



$z = \frac{1}{-\frac{\pi}{2} + 2n\pi}$ for $n \in \mathbb{Z}$ (simple poles)

$z = \frac{1}{\frac{\pi}{2} + 2n\pi}$ for $n \in \mathbb{Z}$ (simple poles)

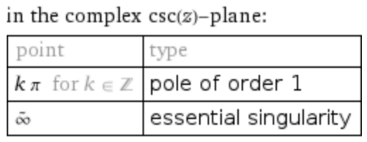
Figure 4. Our output and WolframAlpha's for $1/cos(1/z)$, some decimals have been truncated for brevity.

While it may not be obvious from the start, our output, WolframAlpha's, and what we worked out in Section 2 are equivalent, with the exception that WolframAlpha doesn't report cluster points. The $e^{-6.283in_2}$ exists because we don't have a rule to simplify it to 1.

Our output:

```
Input:    sin(sin(Z)^-1.000)^-1.000
Deriv:
sin(sin(Z)^-1)^-2*sin(Z)^-2*cos(sin(Z)^-1)*cos(Z)
Zeros:    List()
Singular:
List(-1*asin(0.318*((e^(-6.283i*n2))*n1^-1))+
-6.283*n3+Z)
Branch:   List()
Cluster:  List(-3.142*n5+-6.283*n3+Z)
```

WolframAlpha output:



in the complex csc(z)-plane:

point	type
$k\pi$ for $k \in \mathbb{Z}$	pole of order 1
$\tilde{\infty}$	essential singularity

Figure 5. Our output and WolframAlpha's for $1/sin(1/sin(z))$, some decimals have been truncated for brevity. Note that $1/\pi = 0.318$.

Our output is once again equivalent to what we derived in Section 2, with the exception that there are many more quantifiers (the n_i's), that haven't been simplified out. WolframAlpha technically outputs the same thing, but it fails to solve the equations and instead outputs the answer in terms of the $csc(1/z)$-plane, which makes its results harder to interpret and use. It also doesn't identify cluster points.

6 Conclusion and Future Work

We have shown that our method is not only able to find and detect isolated singular points and branch points, but also cluster points, which WolframAlpha doesn't do. Furthermore, our method solves certain cases of singularities that WolframAlpha doesn't, like $1/sin(1/sin(z))$. We aren't able to solve some cases that WolframAlpha can, like $1/f(z)$ where f is a polynomial, but this would be easily fixed with future work. We also don't classify isolated singular points into poles and essential singularities, but this could also be solved with future work. Our algorithm is easily extensible, and it is possible for us to achieve performance much better than WolframAlpha's singularity finding.

There are many improvements that can be made to our algorithm and its implementation. The main issue with our current algorithm is our inability to completely simplify expressions, because this leads to incorrect answers on inputs like $z/(z^2 - z)$. These kinds of problems could be mostly solved by introducing factoring and expanding into our simplify function, which would allow us to cancel common terms, like the z in the earlier expression. Another change we could make to our simplification algorithm would be to allow branching. Currently, the algorithm greedily applies whatever rules match the current expression, but it might simplify something that would remove the possibility of other simplification steps, which is suboptimal. Branching would allow us to try more possibilities and get more simplified expressions. We could also simply add more rules.

The next problem in our algorithm is our inability to solve most equalities consisting of Sums and Terms. We would be able to solve this to some extent by figuring out a way to factor/expand some expressions to produce compositions of polynomials and other functions, which we could then attempt to solve. However, polynomials of degree 5 and above wouldn't be solvable without numerical techniques. Some expressions, like $z + sin(z)$, couldn't be simplified into polynomials, so we would also need numerical root-finding for these expressions. This still wouldn't work if an expression in the equality contained quantifiers. While solving the problem in general is impossible, even for WolframAlpha and Maple, these techniques would greatly increase the number of functions our algorithm would be able to provide answers for.

The other big problems in our system are our inability to simplify subexpressions containing quantifiers, and the fact that we use floating point numbers instead of π and e. Quantifier simplification is an extremely hard problem because quantifiers are elements of $\mathbb{Z}$. An example of this difficulty is $c_1n_1 + c_2n_2$, where $Re(c_1)/Re(c_2)$ or $Im(c_1)/Im(c_2)$ is irrational, because this would simplify to some subset of $\mathbb{C}$, and it would likely be difficult or impossible to represent this subset in terms of quantifiers. The problem of π and e, is solvable by introducing new elements into the AST, and changing how constant expressions are evaluated, but this would take lots of time to implement correctly.

All code in our project is available on:
https://github.com/Anonymous-Stranger/complex-differentiation/tree/exp
rewrite