Fall 2019

Complex Varieties as Minima

Richard Koss
Eastern Illinois University

Complex Varieties as Minima

Richard Koss

December 2019

Abstract

We will explore various numeric methods of finding roots of an analytic function over some open set of the complex plane. We will discuss a method of visually observing the roots, a gradient descent method for finding the roots of an analytic function, a gradient descent method for solving systems of analytic functions, and finally a method of descent that uses osculating circles to find roots of an analytic function. Of particular interest to this thesis are roots of complex polynomials. There will be examples, code snippets, and outputs of programs to illustrate all of these methods.

Acknowledgements

To all who have contributed to my life and education, thank you.

Contents

List of Figures

Introduction

The main purpose of this thesis is to illustrate new numeric methods to find zeros of complex analytic functions. All of these methods can be done in any programming language of choice and rely on iterative algorithms. The idea for these methods originated with attempting to find an improved version of gradient descent. Gradient descent was originally proposed by Cauchy in 1847 [2]. He gives no formal treatment here just an idea. He had promised to revisit this idea in his next memoir and give it a more formal and complete treament but sadly it seems such was never made or was lost. The gradient tells you the direction and magnitude of fastest increase or decrease of a function at a point. Thus, the idea Cauchy had of gradient descent is that if we move in the direction proportional to the negative gradient we should eventually reach a local minimum with respect to the current point. He gave no specifics as to how one should do this specifically but only gave the foundational outline of the idea and an argument that you should eventually approach a local minimum. In this traditional view of descent we always trust the gradient to point us in the right direction and we must try to choose a scaling factor (Cauchy only argued that we can find one that will work) to move proportional to the magnitude of the gradient in the direction of the negative gradient. This form of traditional gradient descent has difficulties in general. Firstly, it is unclear how big each step should be as Cauchy did not posit a suggestion. Any surface with unsuitable geometry such as flat spots, valleys, other similar phenomena that impact the gradient can, as we might expect, be problematic for gradient descent. For problems exhibiting some of these difficult terrain features, gradient descent increasingly zigzags or hemstitches as the gradients point nearly orthogonally to the shortest direction to a minimum point [5].

There are existent methods that rely on information given by the gradient. One such method is due to Nesterov [8, 9] which is commonly called the Nesterov Accelerated Gradient Descent. It is a method which is designed to be implemented on smooth convex problems and whose name stems from a term that acts to accelerate your point

along at each iteration. Another method is the Broyden−Fletcher−Goldfarb−Shanno (BFGS) algorithm [3, 4]. It seeks out stationary points of your function using an approach similar to Newton's method. It is in a class of methods that are denoted quasi-Newton. They are any method where one replaces the true jacobian in Newton's method with an approximate one. Another such method is due to Broyden [1] which also falls within the quasi-Newton class of methods. It is a method that finds roots in k-variables.

One of the appeals of gradient descent is that it extends to higher dimensions and requires only that your function be continuously differentiable. Some of the major general approaches to remedy the slowness or inherent difficulties of gradient descent include preconditioning, calculating a better step size at each iteration, and calculating a better direction at each iteration. Preconditioning is the process by which one changes the geometry of the space to be analyzed. This is to make the objects more suitable to the numeric method you wish to employ. One of the downfalls to this is that we need be careful not to do it in such a way as to lose that which we wish to find. Calculating a better step size is that instead of simply taking one proportional to the gradient, we may take the time to compute a more suitable step size. This allows us to overcome some of the cases where the gradient gets incredibly small as we cross a relatively flat region. Calculating a better direction allows us to avoid some of the aforementioned zigzagging or hemstitching effects. Of course, a collective adverse effect all of these share is that they require additional computation either by us or on the part of the computer and have the potential to increase runtime or total number of operations depending on the implementation. Thus, there may possibly be a harsh tradeoff in that you may get better accuracy or fewer iterations but it may end up taking longer overall to complete. For optimal results, a certain balance must be struck between computational might and efficiency.

The method of gradient descent presented within this thesis uses all three of these methods in an effort to successfully implement Cauchy's idea for gradient descent.

One of the advantages of this method is that it maintains computational efficiency while still delivering improved performance as will be illustrated by example. The alternate method of descent (though not a gradient descent) still uses a similar pre-conditioning technique that will be discussed later in this thesis. First, let us review a couple of key components for the success and implementation of these methods. The first of these is the Maximum Modulus Principle and contributes to both methods. The second of these is the application of osculating circles which applies only to the aptly named osculating descent method.

Maximum Modulus Principle

The maximum modulus principle is key to the success of the algorithms contained herein. The maximum modulus principle may be formally stated in a variety of ways. Here is one such way.

[14, pp. 165-167] **The maximum-modulus theorem.** Let $f(z)$ be an analytic function, regular in a region D and on its boundary C, which we take to be a simple closed contour. If $|f(z)| \leq M$ on C, then $|f(z)| < M$ at all interior points of D, unless $f(z)$ is a constant (when of course $|f(z)| = M$ everywhere).

Proof. We may prove the theorem by contradiction. Suppose there exists an interior point z_0 of D such that $|f(z_0)|$ has a value at least equal to its value anywhere else. Since $f(z)$ is analytic in D we may use the Taylor series expansion of $f(z)$ in powers of $z - z_0$ with some radius of convergence.

$$f(z) = \sum_{n=0}^{\infty} a_n (z - z_0)^n$$

Putting $z - z_0 = re^{i\theta}$, $a_n = A_n e^{i\alpha_n}$, we obtain

$$f(z) = \sum_{n=0}^{\infty} A_n r^n e^{i(\alpha_n + n\theta)}$$

.

$$\text{Therefore} \qquad |f(z)|^2 = \sum_{m=0}^{\infty} \sum_{n=0}^{\infty} A_m A_n r^{m+n} e^{i(\alpha_m + m\theta - \alpha_n - n\theta)}.$$

Suppose first that $a_0 \neq 0$. Since the double series is absolutely convergent, we may rewrite it as a single series in r with some radius of convergence. Let k be the smallest positive value of n for which $a_n \neq 0$. Then

$$|f(z)|^2 = A_0^2 + 2A_0 A_k r^k \cos(\alpha_0 - \alpha_k - k\theta) + \sum_{n=k+1}^{\infty} c_n r^n$$

,

where $|c_n| < c^n$ for some value of c. Hence

$$\left| \sum_{n=k+1}^{\infty} c_n r^n \right| < \sum_{n=k+1}^{\infty} c^n r^n = \frac{c^{k+1} r^{k+1}}{1 - cr}$$

which is less than $A_0 A_k r^k$ if r is small enough. For such a value of r, $|f(z)|^2 - A_0^2$ takes both positive and negative values as θ varies between 0 and 2π. Thus we have A_0 is neither a maximum nor a minimum of $|f(z)|$.

We must have that at least one $a_n (n > 0)$ which is not zero, otherwise, $f(z) = a_0$ for all z.

Finally, if $a_0 = 0$, $|f(z_0)| = 0$, which cannot be a maximum but must be a minimum. □

We have also shown during this proof that $|f(z)|$ cannot have a minimum other than 0 in D, assuming $f(z)$ is nonconstant. Since, $f(z)$ is analytic we also know that roots are isolated points [14, p. 88]. These two observations are key to the success of the algorithms detailed in this thesis.

Osculating Circle

The term osculating circle comes from latin and means "kissing circle", as it kisses a given curve. Given a plane curve, the osculating circle is the tangent circle that best approximates the curvature of that curve at a given point. In other words, it tightly hugs and mimics the curve at that point. The only things we actually need from the osculating circle are the Cartesian coordinates of the center of the osculating circle. If we substitute $t = x$ and $y = f(x)$ for some function f, then the Cartesian coordinates of the center of the osculating circle is given as follows [10] :

$$(x_c, y_c) = \left(x - f'(x)\frac{1 + f'(x)^2}{f''(x)}, \quad f(x) + \frac{1 + f'(x)^2}{f''(x)} \right) \tag{1}$$

This gives us the formula for the algorithm we will use when we look at the osculating descent method. We must now determine how this applies to our specific problem. Later when we talk about the process of preconditioning and other things we will end up turning our functions of z into functions of the real and imaginary parts x, y, respectively. Thus to apply this function with the above substitutions in mind we have to then consider our functions as functions of x and $y = f(x)$. If we start by looking at $f(z) = 0$ then we end up looking at $F(x, f(x)) = 0$ for the osculating descent method. All this means is that we will treat y as function of x for deriving the appropriate formulas for our problem. We need to solve $F(x, f(x)) = 0$, the general implicit function, for $f'(x)$ and $f''(x)$ so that we may substitute them into the formulas above and acquire the coordinates for our center of the osculating circle for our original function of interest. Implicit differentiation yields the following:

$$F'(x, f(x)) = \frac{\partial F}{\partial x}\frac{dx}{dx} + \frac{\partial F}{\partial y}\frac{dy}{dx} = 0$$

Solving for $f'(x)$ we acquire:

$$f'(x) = \frac{-F_x}{F_y} \tag{2}$$

$$F''(x, f(x)) = F_{xx} + F_{yy}\frac{dy}{dx} + F_{yx}\frac{dy}{dx} + F_{yy}\frac{dy}{dx}\frac{dy}{dx} + F_y\frac{dy}{dx^2} = 0$$

Solving for $f''(x)$ we acquire:

$$f''(x) = \frac{-F_{xx}(F_y)^2 + 2F_{xy}F_xF_y - F_{yy}(F_x)^2}{(F_y)^3} \tag{3}$$

Now that we have acquired these we may substitute equations 2 and 3 into 1 for the center of the osculating circle and use this for the osculating descent. Note that when the partial derivative with respect to y is zero this formula will fail. In this case we need to treat x as a function of y instead. All we need to do to accomplish this is substitute x for y wherever it occurs in the formulas.

Method of Viewing Roots

Let $f(z)$ be the analytic function whose roots you wish to observe. Make the substitution $z = x + yi$ and let $g = f(x + yi)\overline{f(x + yi)}$. $g(x, y)$ by definition is greater than or equal to zero and will only be equal to zero when $f(z) = 0$. Thus we have a nonnegative surface that preserves all of the roots. However, if we stop here and look at the graph of the surface it will appear to be quite flat with no ability to visually discern if the area over which we are plotting contains any of the roots (see Figure 1). However, we may remedy this by applying the natural logarithm to our function $g(x, y)$. Let $h(x, y) = \log(g(x, y))$. Then over the same area, the graph of $h(x, y)$ takes on a much different appearance. This is due to the natural log exhibiting a singularity when $g(x, y)$ is equal to zero. This causes a sharp spike to appear at the zero which reaches down toward $-\infty$ (see Figure 2). The (x, y) coordinates of this spike will correspond exactly to the coordinates of the zero of the original function. Figures 3-5 give more examples of these surfaces.

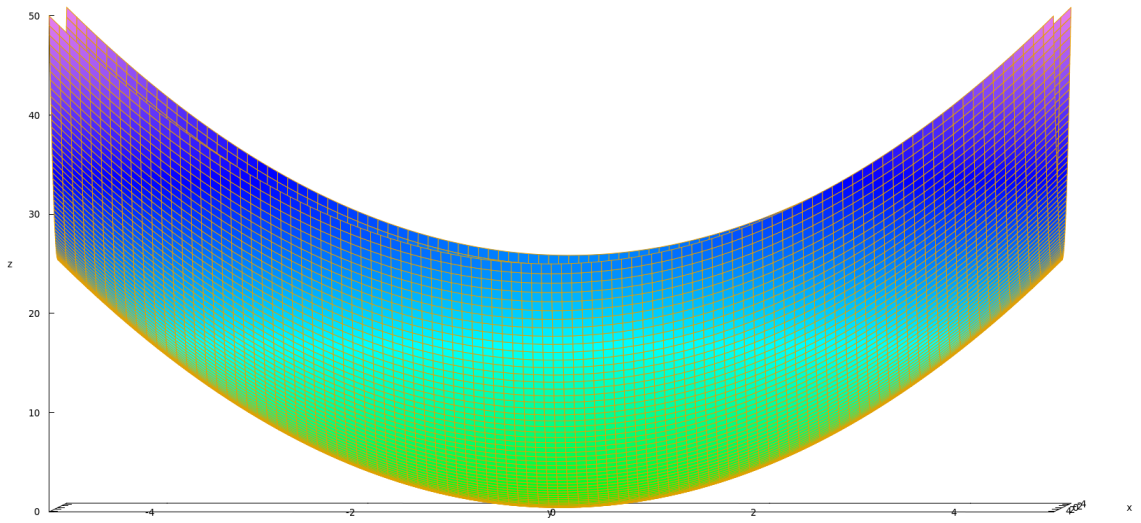

Figure 1: Graph of $f(z) = z$ from -5 to 5 on both x and y axes using the non-spiked method taking $g(x, y) = f(x, y)\overline{f(x, y)}$. Created using Maxima.

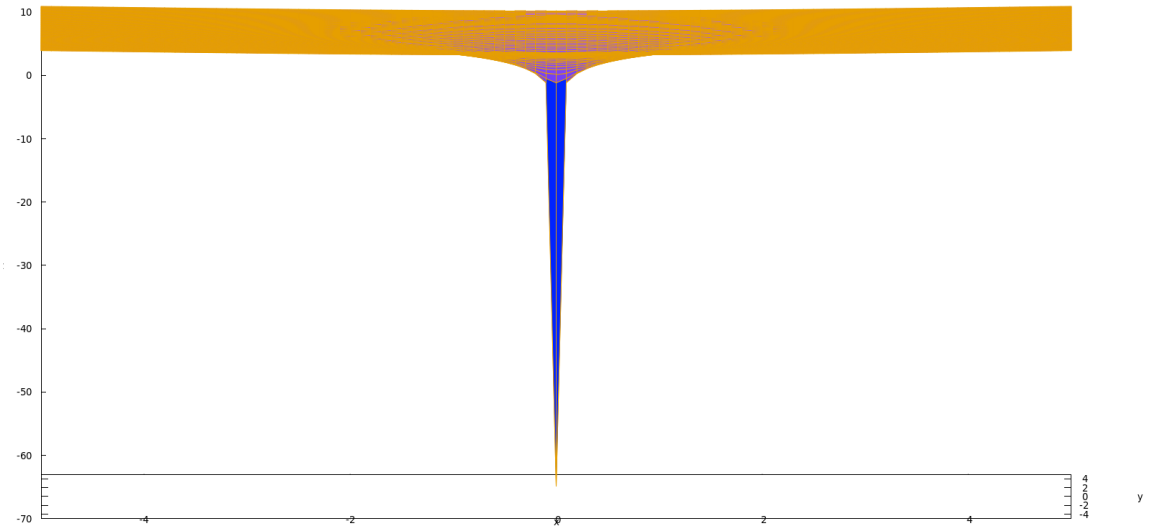

Figure 2: Graph of $f(z) = z$ from -5 to 5 on both x and y axes using the spiked method taking $h(x, y) = \log g(x, y)$, where $g(x, y)$ is the same as in Figure 1. Created using Maxima.

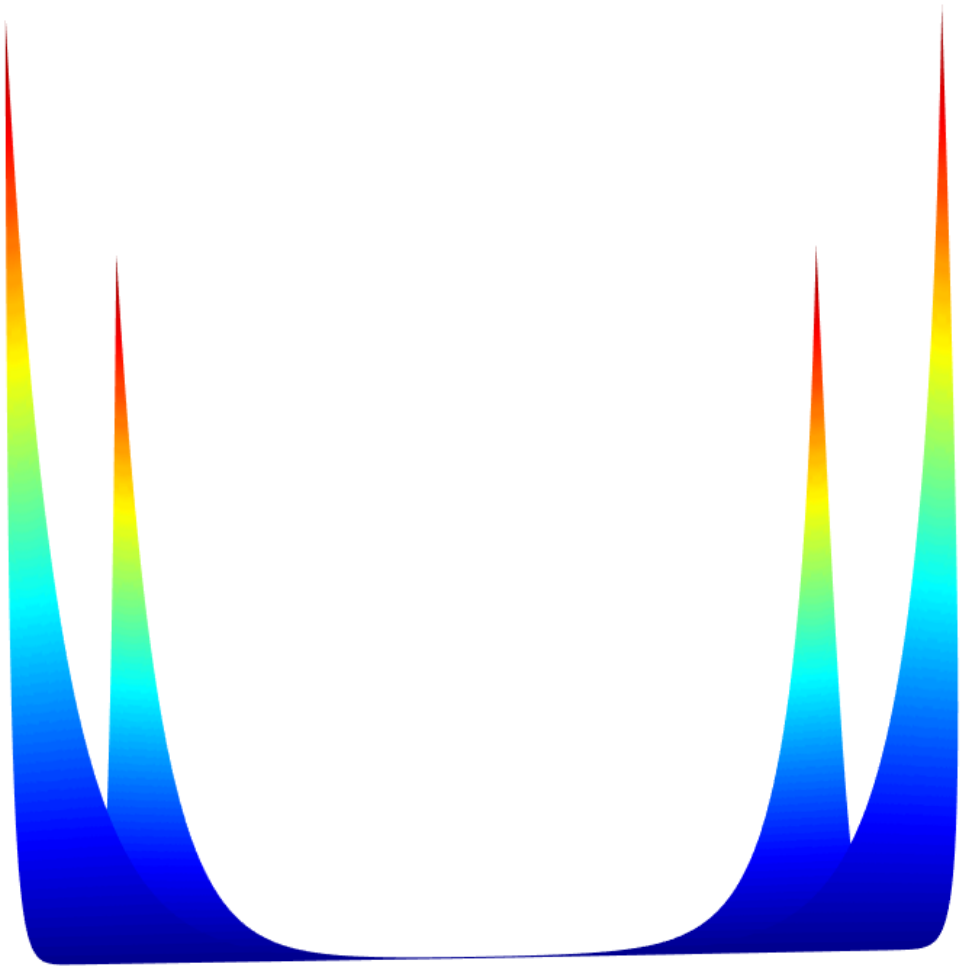

Figure 3: Graph of $f(z) = z(z-3)(z+3)(z-3i)(z+3i)(z+7i)(z-7i)(z-7)(z+7)$ from -8 to 8 on both x and y axes using the non-spiked method taking $g(x,y) = f(x,y)\overline{f(x,y)}$. Created using R.

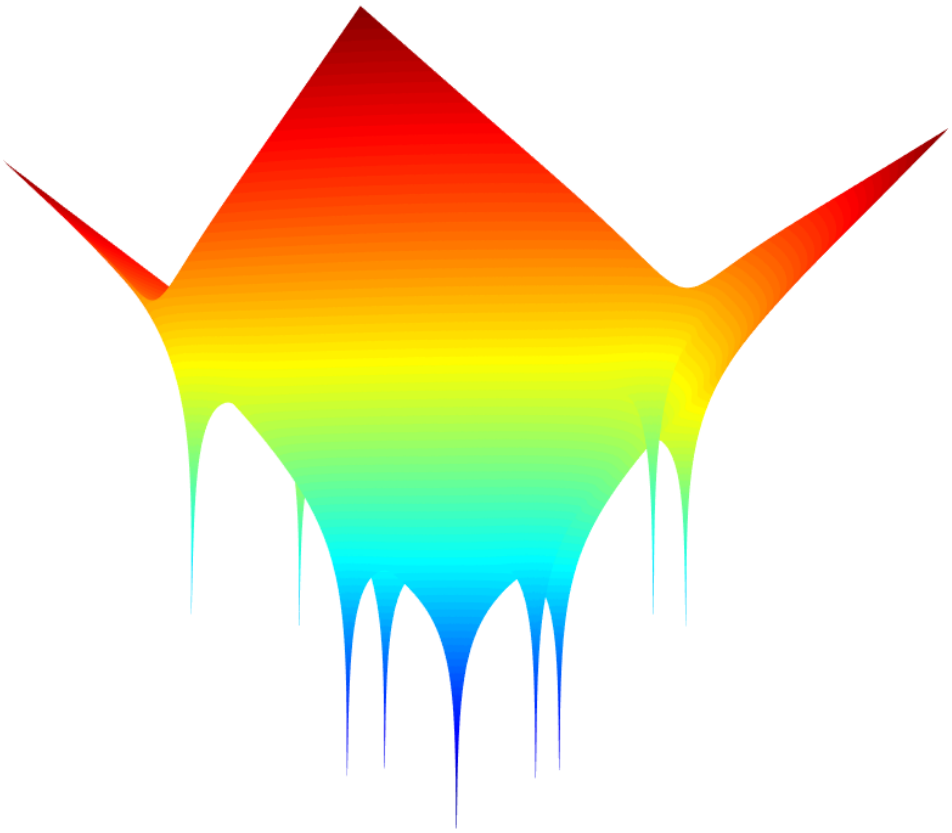

Figure 4: Graph of $f(z) = z(z-3)(z+3)(z-3i)(z+3i)(z+7i)(z-7i)(z-7)(z+7)$ from -8 to 8 on both x and y axes taking $h(x,y) = \log g(x,y)$, where $g(x,y)$ is the same as in Figure 3. Created using R.

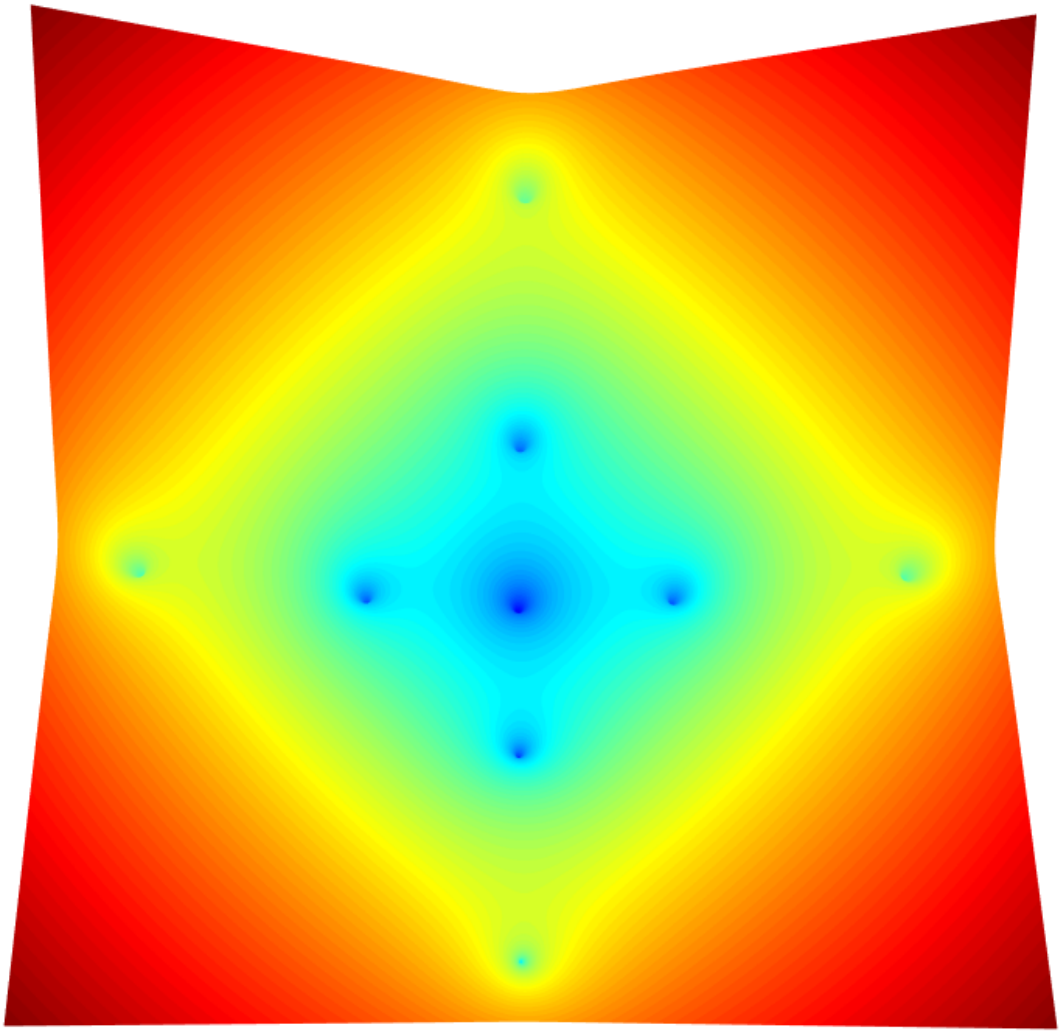

Figure 5: Graph is the same as in Figure 4 but viewed from a top-down perspective. Created using R.

Gradient Descent Method

The gradient descent method illustrated in this thesis can be used for one or several complex variables. We have here that any mention of an analytic function is assumed to be nonconstant. There are some things that are common amongst both of them regardless of dimension (number of variables). They require a function to which the maximum modulus principle may be applied. So we need an analytic function on a specified open region. The maximum modulus principle ensures that we have no local maxima and any local minima are zero. Since the function will be analytic we have that roots are isolated points and thus we can find them uniquely with this algorithm. We also need that repeated applications of the algorithm do not yield points outside of our specified region.

Depending on how the algorithm is implemented there are some things of which we should be aware. If you hit the root exactly (or within machine epsilon) then the algorithm will fail. For any function with multiple unique solutions, there will be a collection of stationary points that will never descend to a zero. They may move but they will continue to shift along to another stationary point. We may think of each zero as exerting a certain amount of pull on our current point in the algorithm. The stationary points are those points in which the pull is balanced in such a way that they cannot escape toward any one zero. Each zero has a region, commonly referred to as a basin of attraction, which is the set of all points which will descend to it. The stationary points reside on the boundaries between these basins. Points sufficiently close to these boundaries exhibit erratic behavior when the algorithm is applied to them in that they descend to a seemingly random zero.

Gradient Descent Method: One Complex Variable

The gradient descent method for a single complex variable may be applied as follows. Let $f(w)$ be an analytic function in some open set of the complex plane to which the maximum modulus principle applies. Making the substitution $w = x + yi$ and multiplying f with its conjugate then calling this new function g, we have $g(x, y) = f(x, y)\overline{f(x, y)} = |f(x, y)|^2$. This new surface S generated by the graph of g is the one upon which we will descend. Now that we have properly preconditioned our problem we are ready for the description of the descent algorithm. Let

$$P_0 = (x_0, y_0) \tag{4}$$

be a point in the domain of $g(w)$. Then

$$P = (x_0, y_0, g(P_0))$$

be the point of S above P_0. Let

$$G(x, y, z) = g(x, y) - z.$$

Then S is given by $G(x, y, z) = 0$. Consider the gradient vector

$$\nabla G(P_0) = \langle g_x(P_0), g_y(P_0), -1\rangle$$

and its projection to the xy-plane $\langle g_x(P_0), g_y(P_0), 0\rangle$. Let L be the line passing through P_0 in the direction of this projection $\langle g_x(P_0), g_y(P_0), 0\rangle$. Let t be a real number; consider the point $P_0(t)$ on L given by

$$P_0(t) = (x_0 + tg_x(P_0), y_0 + tg_y(P_0), 0). \tag{5}$$

We look for the value t_{int} of t such that the vectors $\nabla G(P_0)$ and $\overrightarrow{PP_0(t_{int})} = \langle tg_x(P_0), tg_y(P_0), -g(P_0)\rangle$ are orthogonal. This leads to the equation

$$g_x(P_0)^2 t_{int} + g_y(P_0)^2 t_{int} + g(P_0) = 0. \tag{6}$$

Solving the above equation (3) for t_{int} yields

$$t_{int} = -\frac{g(P_0)}{g_x(P_0)^2 + g_y(P_0)^2} = -\frac{g(P_0)}{\|\nabla g(P_0)\|^2} \tag{7}$$

The corresponding point on L is $P(t_{int})$. Omit the last zero z-coordinate in it, and call a new point $P_{proj}(t_{int})$, that is

$$P_{proj}(t_{int}) = \left(x_0 - \frac{g(P_0)g_x(P_0)}{\|\nabla g(P_0)\|^2}, y_0 - \frac{g(P_0)g_y(P_0)}{\|\nabla g(P_0)\|^2}\right) \tag{8}$$

Finally, replace in (3) P_0 with P_{proj} and repeat the process until the desired precision has been achieved. We propose here the following two gauges of accuracy to determine if and when we should stop iterating in this fashion.

1. $|f(P_0)|$ is small.

2. $\nabla G(P_0) = \langle g_x(P_0), g_y(P_0), -1\rangle$ is almost parallel to the xy-plane. This exactly means that the $\min\{|g_x(P_0)|, |g_y(P_0)|\}$ is large.

The algorithm may be summarized as follows where $g(x, y)$ is the properly preconditioned function (surface):

$$(x_{n+1}, y_{n+1}) = \left(x_n - \frac{g(x_n, y_n)g_x(x_n, y_n)}{\|\nabla g(x_n, y_n)\|^2}, y_n - \frac{g(x_n, y_n)g_y(x_n, y_n)}{\|\nabla g(x_n, y_n)\|^2}\right) \tag{9}$$

Let's look at an example of how this may be applied in practice.

Example 1. Let $f(z) = (z-1)^5(z-(1+i))^4(z-7)^3$ be our function of interest. We can easily identify the roots by inspection of this complex polynomial but this is an illustrative example and in general we would not know the roots. Then we let

$$g(x, y) = f(x, y)\overline{f(x, y)} = (y^2+x^2-14x+49)^3(y^2+x^2-2x+1)^5(y^2-2y+x^2-2x+2)^4$$

, where x, y are the real and imaginary parts, respectively. Now we calculate the partial derivatives of $g(x, y)$ with respect to x and y using Maxima.

$$\begin{aligned} g_x(x, y) = {} & 3(2x-14)(y^2+x^2-14x+49)^2(y^2+x^2-2x+1)^5(y^2-2y+x^2-2x+2)^4 \\ & + 5(2x-2)(y^2+x^2-14x+49)^3(y^2+x^2-2x+1)^4(y^2-2y+x^2-2x+2)^4 + \\ & 4(2x-2)(y^2+x^2-14x+49)^3(y^2+x^2-2x+1)^5(y^2-2y+x^2-2x+2)^3 \end{aligned}$$

$$g_y(x,y) = 6y(y^2+x^2-14x+49)^2(y^2+x^2-2x+1)^5(y^2-2y+x^2-2x+2)^4+$$
$$10y(y^2+x^2-14x+49)^3(y^2+x^2-2x+1)^4(y^2-2y+x^2-2x+2)^4+$$
$$4(2y-2)(y^2+x^2-14x+49)^3(y^2+x^2-2x+1)^5(y^2-2y+x^2-2x+2)^3$$

Now we have obtained everything required to apply the method using equation (8). Starting with any point, let's say $(x_0, y_0) = (-330, 917)$ and performing 300 iterations we arrive at the following output in our Python program (See Appendix for full listing of code) for this method:

```
After 300 iterations we find the following:
Guess for x is 1.0000000000844652500
Guess for y is 1.0000000024292978490
Initial z value (height): 5.4611041058136187810E+71
Final z value (height): 6.137807018349120965 5E-65
x-component of the gradient is 7.029578303082330055 9E-57
y-component of the gradient is 2.021771041797016716 9E-55
The min of the magnitude of x,y-slopes of the gradient is:
4.946158488407109893 1E+54
The elapsed time for the method in seconds is 0.025024577
```

We observe that we are certainly descending toward the zero $(1, 1)$ which corresponds to the complex number $1+i$. We also have met both criteria to feel confident in how close we are to the zero. That is we have a small final height and the magnitude of the minimum of the $xy-$slopes of the gradient is fairly large. Note that although we performed 300 iterations and started relatively far away from a zero, the runtime we experienced was roughly 0.03 seconds. In this example, if we did 19 more iterations we would exceed the set precision for this program. This means we would hit a point where the calculations break down as we get sufficiently close to the zero such that the computer can no longer distinguish the difference between what we have and the true zero with our chosen precision. If we increase the desired precision from Python then

we will be able to execute more iterations, however, this will increase the runtime of each iteration.

Rosenbrock Function

Although what has come to be called the Rosenbrock function does not fall under the purview for which our methods were designed, we have that our gradient descent method does work on it. The Rosenbrock function is defined as follows [13]:

$$f(x, y) = 100(y - x)^2 + (1 - x)^2$$

This function has a minimum value of $f(x, y) = 0$ when $x = y = 1$, with a curved valley along the parabola $y = x^2$ which may be viewed as the dark blue band in Figure 6. This valley may cause difficulties for gradient descent and similar methods and that is why Rosenbrock used it. He wanted to test a proposed improved method of descent. Thus it has seemingly become somewhat of a litmus test for descent methods. As a curiosity, this gradient descent method was applied to the Rosenbrock function so let us see the results starting with $x = -124$ and $y = 431$. (For a full code listing see Appendix.)

```
After 13000 iterations we find the following:
Guess for x is 1.000000000000000000000000328
Guess for y is 1.000000000000000000000000682
Initial z value (height): 22335318125
Final z value (height): 1.75184E-53
x-component of the gradient is
-9.744000000000000000000000341E-26
y-component of the gradient is 5.200E-26
The minimum of the absolute value of x,y-slopes of
the gradient is 10262725779967159277504105.0544
The elapsed time for the method in seconds is 0.121599121
```

So we have started fairly far away and used a considerable number of iterations but the runtime is only about an eighth of a second. We have also acquired both of our measures of accuracy in that the final height is small and the $x, y-$slopes of the gradient are large.

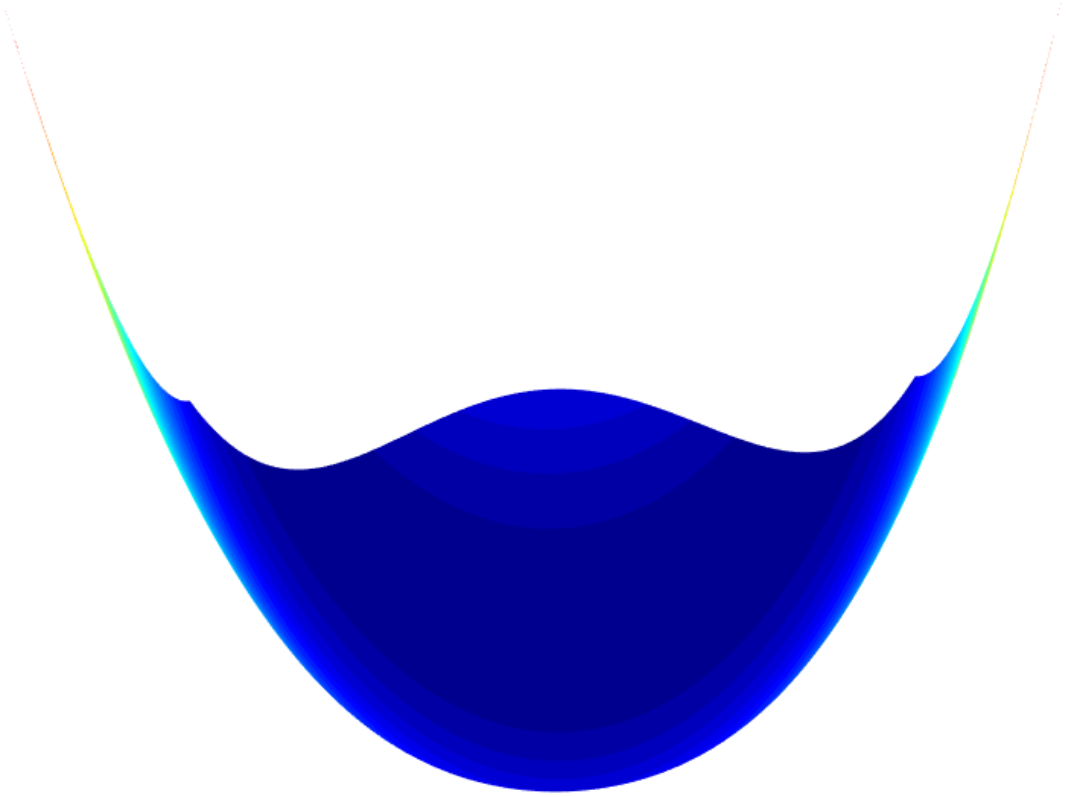

Figure 6: Graph of $f(x, y) = 100(y - x)^2 + (1 - x)^2$ from -2 to 2 on both x and y axes. Created using R.

Now we will see what happens if we attempt to spike the Rosenbrock function to view the roots. We must keep in mind that the Rosenbrock function is not the type of function for which the descent methods in this thesis were designed and we will see a clear consequence of this in Figure 7. This Figure 7 is what the computer thinks the logarithmically spiked version of the Rosenbrock function looks like when viewed from roughly the same perspective as in Figure 6. Notice the jaggedness and uniformity of color. This uniformity of color indicates the computer considers all the

values as roughly equal in height.

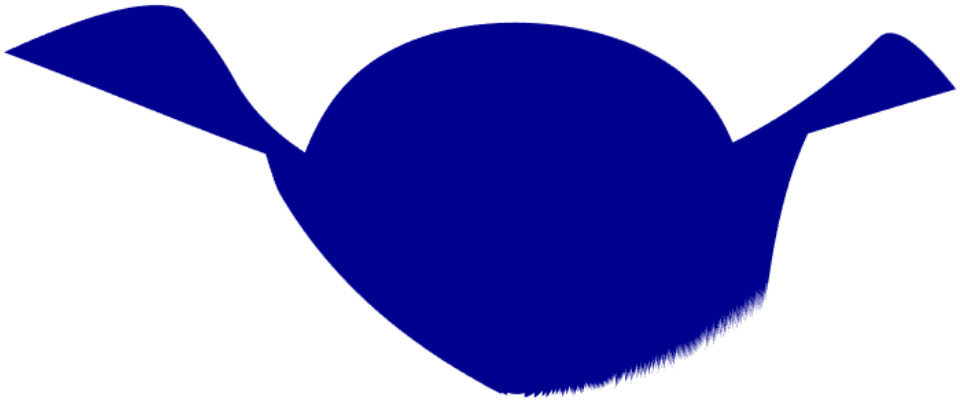

Figure 7: Graph of $\log(f(x, y))$ from -2 to 2 on both x and y axes where $f(x, y)$ is the Rosenbrock function. Created using R.

Gradient Descent Method: Several Complex Variables

The method for several complex variables shall proceed in a similar fashion. We will look at the case of two complex variables and then it will be easy to see how we would extend it to even higher dimensions. Let $f_1(w_1, w_2)$ and $f_2(w_1, w_2)$ be a system of analytic functions of 2 complex variables in an open set of $\mathbb{C}^2$. Then we create the surface

$$g(w_1, w_2) = f_1(w_1, w_2)\overline{f_1(w_1, w_2)} + f_2(w_1, w_2)\overline{f_2(w_1, w_2)}$$

This surface exhibits similar behavior as the one variable case in that the only roots of this surface are the common roots between the two functions and the Maximum Modulus Principle still applies, however, the Minimum Modulus Principle is not known to exist for several complex variables. Thus, there is the possibility that we will descend to nonzero minima.

Starting with the point $P_0 = (x_{1,0}, y_{1,0}, x_{2,0}, y_{2,0})$ we may proceed as before and our algorithm takes the following form on the j^{th} iteration:

$$P_j = \left(x_{1,j-1} - \frac{g(P_{j-1})g_{x_1}(P_{j-1})}{\|\nabla g(P_{j-1})\|^2}, \quad y_{1,j-1} - \frac{g(P_{j-1})g_{y_1}(P_{j-1})}{\|\nabla g(P_{j-1})\|^2}, \right.$$

$$\left. x_{2,j-1} - \frac{g(P_{j-1})g_{x_2}(P_{j-1})}{\|\nabla g(P_{j-1})\|^2}, \quad y_{2,j-1} - \frac{g(P_{j-1})g_{y_2}(P_{j-1})}{\|\nabla g(P_{j-1})\|^2} \right)$$

We simply continue on in this fashion for any number of variables upon which we wish to descend.

Example 2. Let us look at an example of how this is done in practice with a system of 3 complex variables and functions (See Appendix for a full code listing). Let our

system be the following:

$$f_1 = z_1^2 + z_2 + z_3^2$$
$$f_2 = z_1(z_1 - 1) + z_2 + z_3(z_3 - i)$$
$$f_3 = z_1(z_1 - 1)(z_1 - i) + z_2 + z_3(z_3 - 1)(z_3 - i)$$

Then we define

$$g(a, b, c, d, e, f) = f_1(a, b)\overline{f_1(a, b)} + f_2(c, d)\overline{f_2(a, b)} + f_3(a, b)\overline{f_3(a, b)}$$

The solutions to this system are

$$\{(z_1 = 1, z_2 = 0, z_3 = i), (z_1 = z_2 = z_3 = 0), (z_1 = -1, z_2 = 0, z_3 = -i)\}$$

Using $(-3.1, 10.1, 14, -7, 30, -123)$ as a starting point and applying the method 1000 times we arrive at the following

```
After 1000 iterations we find the following:
Guess for a is  2.389413422531040065 2E-11
Guess for b is  2.951089518633617989 7E-11
Guess for c is  3.728078225439400134 9E-11
Guess for d is  -8.582642092213628927 0E-12
Guess for e is  -4.591019079898450861 8E-11
Guess for f is  6.161602192319306800 3E-12
Initial z value (height): 4162535261376.1947284
Final z value (height): 2.845619570403882186 5E-21
a-component of the gradient is  -1.002938977714358514 7E-10
b-component of the gradient is  -1.884987679931026390 1E-11
c-component of the gradient is  1.168746347082995088 2E-10
d-component of the gradient is  -6.272937447718393820 6E-11
e-component of the gradient is  -7.683070439454760402 6E-11
```

f−component of the gradient is 3.587993068888300363 64E−11
The min of the magnitude of the slopes of the gradient is 8556176474.8684851965
The elapsed time for the method in seconds is 1.450609399

We see here that it appears to be descending to the zero $z_1 = z_2 = z_3 = 0$. We also have acquired the conditions that allow us to be confident in our results.

Osculating Descent Method

As we saw in the next introductory subsection, **Method of Viewing Roots** that the spikes produced in locating the roots appear to be rather tubular in nature. Thus, an idea arose for a method as to how one might descend down this tunnel and swiftly converge to a zero. This method came into form by utilizing the formulas for the osculating circle. We will apply the same preconditioning techniques as we did in the introductory section for viewing the roots. In other words, we will multiply our function with its conjugate to create the same surface we have been using and then we spike it at the roots using the natural logarithm. Then starting with any point in our region we like, we iterate by repeatedly applying the formula we derived in the introductory section to each output. Let us apply this method to the function from example 1 from the earlier discussed **Gradient Descent Method: One Complex Variable** section.

Example 3 (See Appendix for full code listing) Let $g(x, y)$ be the same as in Example 1 and let

$$h(x, y) = \log(g(x, y)).$$

Let's use the same starting point an compare our results between the two methods. So using $(x_0, y_0) = (-330, 917)$ and 7 iterations we have the following results

After 7 iterations we find the following:

```
Guess for x is 1.0000000000251796003
Guess for y is 2.01443013543844E-10
The elapsed time for the method in seconds is
0.0008972829999999987
```

Notice here the drastic decrease in the number of iteration to approximate a zero. We also descended to a different zero using this method as we went to $1 + i$ before but now we have 1. If we increase the number of iterations by one then the computer will not be able to distinguish between the approximate zero and the true zero and the method will fail computationally. To resolve this we need only increase the precision we ask from our program. This method tends to very rapidly overwhelm the standard precision for modern programming languages. The downside to this method is that there is no direct way to extend it to higher dimensions as it is due to the nature of osculating circles/spheres.

Conclusion

There are many questions still to be considered on this topic. An interesting topic of further study would be to emulate the idea and success of the osculating descent method in higher dimensions. Another topic that merits further study is to observe how the roots and their multiplicities shape the set of stationary points. Equivalently, one may look at the shape of the basins of attraction. This would allow us in practice to devise a method for selecting smart test points to initiate the algorithms that would ensure or maximize the probability they descend to different roots. This would greatly increase the efficiency of our search in general so that we do not repeatedly descend to a zero we have already found. Yet further study should be made into floating point representation and ways to increase the precision and accuracy we acquire from our programs. There are times when standard floating point representations fail to provide adequate results.

Appendix

Here are the complete Python codes associated with each example provided in this thesis. For example 1, we have:

```
import time
import decimal as dec
dec.setcontext(dec.ExtendedContext)
dec.getcontext().prec = 20



# This is our descent method applied to the function
# f(z)=(z-1)^5 * (z-(1+i))^4 * (z-7)^3
def g(x, y):
    return (y**2+x**2-14*x+49)**3
    *(y**2+x**2-2*x+1)**5*(y**2-2*y+x**2-2*x+2)**4


def gx(x, y):
    return 3*(2*x-14)*(y**2+x**2-14*x+49)**2*
    (y**2+x**2-2*x+1)**5*(y**2-2*y+x**2-2*x+2)**4
    +5*(2*x-2)*(y**2+x**2-14*x+49)**3*(y**2+x**
    2-2*x+1)**4*(y**2-2*y+x**2-2*x+2)**4+4*(2*x-2)*
    (y**2+x**2-14*x+49)**3*(y**2+x**2-2*x+1)**5*
    (y**2-2*y+x**2-2*x+2)**3


def gy(x,y):
    return 6*y*(y**2+x**2-14*x+49)**2*(y**2+x**2
    -2*x+1)**5*(y**2-2*y+x**2-2*x+2)**4+10*y*
    (y**2+x**2-14*x+49)**3*(y**2+x**2-2*x+1)**4*
```

```
    (y**2-2*y+x**2-2*x+2)**4+4*(2*y-2)*
    (y**2+x**2-14*x+49)**3*(y**2+x**2-2*x+1)**5*
    (y**2-2*y+x**2-2*x+2)**3

def gradmag(x, y):
    return gx(x, y)**2 + gy(x, y)**2

def groundx(x, y):
    intPx = g(x, y) * gx(x, y) / gradmag(x, y)
    return x - intPx

def groundy(x, y):
    intPy = g(x, y) * gy(x, y) / gradmag(x, y)
    return y - intPy

x0 = nextX = dec.Decimal(-330)
y0 = nextY = dec.Decimal(917)

iterations = 300
ptm = time.process_time()
for i in range(iterations):
    prevX = nextX
    prevY = nextY
    nextX = groundx(prevX, prevY)
    nextY = groundy(prevX, prevY)
elapsed = time.process_time() - ptm
gradeX = gx(nextX, nextY)
gradeY = gy(nextX, nextY)
```

```
xSlope = 1/abs(gradeX)
ySlope = 1/abs(gradeY)
minSlope = min(xSlope,ySlope)

print('After ', iterations, '
iterations we find the following:')
print('Guess for x is ', nextX)
print('Guess for y is ', nextY)

print('Initial z value (height):',g(x0, y0))
print('Final z value (height):', g(nextX,nextY))
print('x-component of the gradient is ', gradeX)
print('y-component of the gradient is ', gradeY)
print('The min of the magnitude of
x,y-slopes of the gradient is ', minSlope)
print('The elapsed time for the
 method in seconds is', elapsed)
```

Here is the code for the Rosenbrock function example:

```
import time
import decimal as dec
dec.setcontext(dec.ExtendedContext)
dec.getcontext().prec = 30



# This is our descent method applied to the function
# Rosenbrock function f(x,y)=(1-x)^2+100(y-x^2)^2
def g(x, y):
```

```
    return (1-x)**2 + 100*(y-x**2)**2


def gx(x, y):
    return -400*x*(y-x**2)-2*(1-x)


def gy(x,y):
    return 200*(y-x**2)


def gradmag(x, y):
    return gx(x, y)**2 + gy(x, y)**2


def groundx(x, y, speed):
    intPx = g(x, y) * gx(x, y) / gradmag(x, y)
    if speed == 1:
        return x - intPx
    elif speed == 2:
        return x - 2*intPx


def groundy(x, y, speed):
    intPy = g(x, y) * gy(x, y) / gradmag(x, y)
    if speed == 1:
        return y - intPy
    elif speed == 2:
```

```
        return y - 2 * intPy

x0 = nextX = dec.Decimal(-124)
y0 = nextY = dec.Decimal(431)

iterations = 13000
ptm = time.process_time()
for i in range(iterations):
    prevX = nextX
    prevY = nextY
    nextX = groundx(prevX, prevY, factor)
    nextY = groundy(prevX, prevY, factor)
elapsed = time.process_time() - ptm
gradeX = gx(nextX, nextY)
gradeY = gy(nextX, nextY)
xSlope = 1/abs(gradeX)
ySlope = 1/abs(gradeY)
minSlope = min(xSlope, ySlope)

print('After ', iterations, '
iterations we find the following:')
print('Guess for x is ', nextX)
print('Guess for y is ', nextY)

print('Initial z value (height):',g(x0, y0))
print('Final z value (height):', g(nextX,nextY))
print('x-component of the gradient is ', gradeX)
print('y-component of the gradient is ', gradeY)
```

```
print('The minimum of the absolute value
of x,y-slopes of the gradient is ', minSlope)
print('The elapsed time for the method in seconds is',
 elapsed)
```

For Example 2 (system of complex variables) we have:

```
import time
import decimal as dec


dec.setcontext(dec.ExtendedContext)
dec.getcontext().prec = 20



# This is our descent method applied to the system
f1 = z1^2 + z2 + z3^2,
f2 = z1(z1-1) + z2 + z3(z3-i),
f3 = z1(z1-1)(z1-i) + z2 + z3(z3-1)(z3-i)
def g(a, b, c, d, e, f):
    return f**6-2*f**5+3*e**2*f**4-2*e*f**4+4*f**4
    -4*e**2*f**3+4*e*f**3-2*d*f**3+2*b**3*f**3-2*
    b**2*f**3-6*a**2*b*f**3+4*a*b*f**3+2*a**2*f**3
    -2*a*f**3-4*f**3+3*e**4*f**24*e**3*f**2+8*e**2
    *f**2-6*c*e*f**2+18*a*b**2*e*f**2-6*b**2*e*f**2
    -12*a*b*e*f**2+6*b*e*f**2-6*a**3*e*f**2+6*a**2*
    e*f**2-2*e*f**2+2*d*f**2-2*c*f**2-2*b**3*f**2-
    6*a*b**2*f**2+8*b**2*f**2+6*a**2*b*f**2-2*b*f**2
    +2*a**3*f**2-8*a**2*f**2+4*a*f**2+2*f**2-2*e**4*f
    +4*e**3*f+6*d*e**2*f-6*b**3*e**2*f+6*b**2*e**2*f+
```

```
    18*a**2*b*e**2*f-12*a*b*e**2*f-6*a**2*e**2*f+6*a*e
    **2*f-4*e**2*f+4*d*e*f+4*c*e*f+4*b**3*e*f-12*a*b**2
    *e*f-12*a**2*b*e*f+32*a*b*e*f-8*b*e*f+4*a**3*e*f-4*
    a*e*f+6*a*b**2*f-4*b**2*f-4*a*b*f+2*b*f-2*a**3*f+4*
    a**2*f-2*a*f +e**6-2*e**5+4*e**4+2*c*e**3-6*a*b**2*
    e**3+2*b**2*e**3+4*a*b*e**3-2*b*e**3+2*a**3*e**3-2*a
    **2*e**3-2*e**3-2*d*e**2+2*c*e**2+2*b**3*e**2+6*a*b**2
    *e**2-8*b**2*e**2-6*a**2*b*e**2+2*b*e**2-2*a**3*e**2+8
    *a**2*e**2-4*a*e**2+2*e**2-2*b**3*e+2*b**2*e+6*a**2*b*
    e-8*a*b*e+2*b*e-2*a**2*e+2*a*e+3*d**2-2*b**3*d+2*b**2*d+
    6*a**2*b*d+4*a*b*d-2*b*d-2*a**2*d+2*a*d+3*c**2-6*a*
    b**2*c-2*b**2*c+4*a*b*c-2*b*c+2*a**3*c+2*a**2*c-2*a*c
    +b**6-2*b**5+3*a**2*b**4-2*a*b**4+4*b**4-4*a**2*b**3+4*a
    *b**3-2*b**3+3*a**4*b**2-4*a**3*b**2+8*a**2*b**2-4*a*b**2
    +2*b**2-2*a**4*b+4*a**3*b-2*a**2*b+
    a**6-2*a**5+4*a**4-4*a**3+2*a**2

def ga(a, b, c, d, e, f):
    return -12*a*b*f**3+4*b*f**3+4*a*f**3-2*f**3+18*b**2*e*
    f**2-12*b*e*f**2-18*a**2*e*f**2+12*a*e*f**2-6*b**2*f**2
    +12*a*b*f**2+6*a**2*f**2-16*a*f**2+4*f**2+36*a*b*e**2*f
    -12*b*e**2*f-12*a*e**2*f+6*e**2*f-12*b**2*e*f-24*a*b*e*f
    +32*b*e*f+12*a**2*e*f-4*e*f+6*b**2*f-4*b*f-6*a**2*f+8*
    a*f-2*f-6*b**2*e**3+4*b*e**3+6*a**2*e**3-4*a*e**3+6*
    b**2*e**2-12*a*b*e**2-6*a**2*e**2+16*a*e**2-4*e**2+12*a
    *b*e-8*b*e-4*a*e+2*e+12*a*b*d+4*b*d-4*a*d+2*d-6*b**2*c+
    4*b*c+6*a**2*c+4*a*c-2*c+6*a*b**4-2*b**4-8*a*b**3+4*
    b**3+12*a**3*b**2-12*a**2*b**2+16*a*b**2-4*b**2-8*
```

```
    a**3*b+12*a**2*b-4*a*b+6*a**5-10*a**4+16*a**3-12*a**2+4*a

def gb(a, b, c, d, e, f):
    return 6*b**2*f**3-4*b*f**3-6*a**2*f**3+4*a*f**3+
    36*a*b*e*f**2-12*b*e*f**2-12*a*e*f**2+6*e*f**2-6*
    b**2*f**2-12*a*b*f**2+16*b*f**2+6*a**2*f**2-2*f**2-
    18*b**2*e**2*f+12*b*e**2*f+18*a**2*e**2*f-12*a*e**2*f+
    12*b**2*e*f-24*a*b*e*f-12*a**2*e*f+32*a*e*f-8*e*f+
    12*a*b*f-8*b*f-4*a*f+2*f-12*a*b*e**3+4*b*e**3+4*a*e**3-
    2*e**3+6*b**2*e**2+12*a*b*e**2-16*b*e**2-6*a**2*e**2+
    2*e**2-6*b**2*e+4*b*e+6*a**2*e-8*a*e+2*e-6*b**2*d+
    4*b*d+6*a**2*d+4*a*d-2*d-12*a*b*c-4*b*c+4*a*c-
    2*c+6*b**5-10*b**4+12*a**2*b**3-8*a*b**3+16*b**3-
    12*a**2*b**2+12*a*b**2-6*b**2+6*a**4*b-8*a**3*b+
    16*a**2*b-8*a*b+4*b-2*a**4+4*a**3-2*a**2

def gc(a, b, c, d, e, f):
    return -6*e*f**2-2*f**2+4*e*f+2*e**3+2*e**2+6*c-
    6*a*b**2-2*b**2+4*a*b-2*b+2*a**3+2*a**2-2*a

def gd(a, b, c, d, e, f):
    return -2*f**3+2*f**2+6*e**2*f+4*e*f-2*e**2+6*d-
    2*b**3+2*b**2+6*a**2*b+4*a*b-2*b-2*a**2+2*a

def ge(a, b, c, d, e, f):
    return6*e*f**4-2*f**4-8*e*f**3+4*f**3+12*e**3*f**2-
    12*e**2*f**2+16*e*f**2-6*c*f**2+18*a*b**2*f**2-
    6*b**2*f**2-12*a*b*f**2+6*b*f**2-6*a**3*f**2+6*a**2
```

```
    *f**2-2*f**2-8*e**3*f+12*e**2*f+12*d*e*f-12*b**3*e*f
    +12*b**2*e*f+36*a**2*b*e*f-24*a*b*e*f-12*a**2*e*f+
    12*a*e*f-8*e*f+4*d*f+4*c*f+4*b**3*f-12*a*b**2*f-
    12*a**2*b*f+32*a*b*f-8*b*f+4*a**3*f-4*a*f+6*e**5-
    10*e**4+16*e**3+6*c*e**2-18*a*b**2*e**2+6*b**2*
    e**2+12*a*b*e**2-6*b*e**2+6*a**3*e**2-6*a**2*
    e**2-6*e**2-4*d*e+4*c*e+4*b**3*e+12*a*b**2*e-
    16*b**2*e-12*a**2*b*e+4*b*e-4*a**3*e+16*a**2*e
    -8*a*e+4*e-2*b**3+2*b**2+6*a**2*b-8*a*b+2*b-2*a**2+2*a

def gf(a, b, c, d, e, f):
    return6*f**5-10*f**4+12*e**2*f**3-8*e*f**3+16*f**3-
    12*e**2*f**2+12*e*f**2-6*d*f**2+6*b**3*f**2-6*b**2
    *f**2-18*a**2*b*f**2+12*a*b*f**2+6*a**2*f**2-6*a*f**2
    -12*f**2+6*e**4*f-8*e**3*f+16*e**2*f-12*c*e*f+
    36*a*b**2*e*f-12*b**2*e*f-24*a*b*e*f+12*b*e*f-
    12*a**3*e*f+12*a**2*e*f-4*e*f+4*d*f-4*c*f-4*b**3*f-
    12*a*b**2*f+16*b**2*f+12*a**2*b*f-4*b*f+4*a**3*f-
    16*a**2*f+8*a*f+4*f-2*e**4+4*e**3+6*d*e**2-6*b**3*
    e**2+6*b**2*e**2+18*a**2*b*e**2-12*a*b*e**2-
    6*a**2*e**2+6*a*e**2-4*e**2+4*d*e+4*c*e+4*b**3*e-
    12*a*b**2*e-12*a**2*b*e+32*a*b*e-8*b*e+4*a**3*e-
    4*a*e+6*a*b**2-4*b**2-4*a*b+2*b-2*a**3+4*a**2-2*a

def gradmag(a, b, c, d, e, f):
    return ga(a,b,c,d,e,f)**2 + gb(a,b,c,d,e,f)**2 +
    gc(a,b,c,d,e,f)**2 + gd(a,b,c,d,e,f)**2 +
    ge(a,b,c,d,e,f)**2 + gf(a,b,c,d,e,f)**2
```

```
def grounda(a, b, c, d, e, f):
    intPa =  g(a,b,c,d,e,f)*ga(a,b,c,d,e,f)/
    gradmag(a,b,c,d,e,f)
    return a - intPa

def groundb(a, b, c, d, e, f):
    intPb =  g(a,b,c,d,e,f)*gb(a,b,c,d,e,f)/
    gradmag(a,b,c,d,e,f)
    return b - intPb

def groundc(a, b, c, d, e, f):
    intPc =  g(a,b,c,d,e,f)*gc(a,b,c,d,e,f)/
    gradmag(a,b,c,d,e,f)
    return c - intPc

def groundd(a, b, c, d, e, f):
    intPd =  g(a,b,c,d,e,f)*gd(a,b,c,d,e,f)/
    gradmag(a,b,c,d,e,f)
    return d - intPd

def grounde(a, b, c, d, e, f):
    intPe =  g(a,b,c,d,e,f)*ge(a,b,c,d,e,f)/
    gradmag(a,b,c,d,e,f)
    return e - intPe

def groundf(a, b, c, d, e, f):
    intPf =  g(a,b,c,d,e,f)*gf(a,b,c,d,e,f)/
```

```
    gradmag(a,b,c,d,e,f)
    return f - intPf

a0 = nextA = dec.Decimal(-3.1)
b0 = nextB = dec.Decimal(10.1)
c0 = nextC = dec.Decimal(14)
d0 = nextD = dec.Decimal(-7)
e0 = nextE = dec.Decimal(30)
f0 = nextF = dec.Decimal(-123)

iterations = 1000


ptm = time.process_time()
for i in range(iterations):
    prevA = nextA
    prevB = nextB
    prevC = nextC
    prevD = nextD
    prevE = nextE
    prevF = nextF
    nextA = grounda(prevA, prevB, prevC, prevD, prevE, prevF)
    nextB = groundb(prevA, prevB, prevC, prevD, prevE, prevF)
    nextC = groundc(prevA, prevB, prevC, prevD, prevE, prevF)
    nextD = groundd(prevA, prevB, prevC, prevD, prevE, prevF)
    nextE = grounde(prevA, prevB, prevC, prevD, prevE, prevF)
    nextF = groundf(prevA, prevB, prevC, prevD, prevE, prevF)
elapsed = time.process_time() - ptm
```

```
gradeA = ga(nextA, nextB, nextC, nextD, nextE, nextF)
gradeB = gb(nextA, nextB, nextC, nextD, nextE, nextF)
gradeC = gc(nextA, nextB, nextC, nextD, nextE, nextF)
gradeD = gd(nextA, nextB, nextC, nextD, nextE, nextF)
gradeE = ge(nextA, nextB, nextC, nextD, nextE, nextF)
gradeF = gf(nextA, nextB, nextC, nextD, nextE, nextF)
aSlope = 1 / abs(gradeA)
bSlope = 1 / abs(gradeB)
cSlope = 1 / abs(gradeC)
dSlope = 1 / abs(gradeD)
eSlope = 1 / abs(gradeE)
fSlope = 1 / abs(gradeF)
minSlope=min(aSlope, bSlope, cSlope, dSlope, eSlope, fSlope)

print('After ', iterations,
' iterations we find the following:')
print('Guess for a is ', nextA)
print('Guess for b is ', nextB)
print('Guess for c is ', nextC)
print('Guess for d is ', nextD)
print('Guess for e is ', nextE)
print('Guess for f is ', nextF)

print('Initial z value (height):', g(a0, b0, c0, d0, e0, f0))
print('Final z value (height):',
g(nextA, nextB, nextC, nextD, nextE, nextF))
print('a-component of the gradient is ', gradeA)
```

```
print('b-component of the gradient is ', gradeB)
print('c-component of the gradient is ', gradeC)
print('d-component of the gradient is ', gradeD)
print('e-component of the gradient is ', gradeE)
print('f-component of the gradient is ', gradeF)
print('The min of the magnitude of the slopes of the
gradient is ', minSlope)
print('The elapsed time for the method in seconds is ',
elapsed)
```

For example 3 we have the code used as follows:

```
import numpy as np
import decimal as dec
import time
dec.setcontext(dec.ExtendedContext)
dec.getcontext().prec = 20


# This is our tunnel method applied to the function
f(z)=(z-1)^5 * (z-(1+i))^4 * (z-7)^3
def glog(x, y):
    return 3*np.log(y**2+x**2-14*x+49)+
    5*np.log(y**2+x**2-2*x+1)+4*np.log(y**2-2*y+x**2-2*x+2)


def glogx(x, y):
    return (4*(2*x-2))/(y**2-2*y+x**2+(-2)*x+2)+(5*(2*x-2))
    /(y**2+x**2+(-2)*x+1)+(3*(2*x-14))/(y**2+x**2+(-14)*x+49)
```

```
def glogy(x, y):
    return (4*(2*y-2))/(y**2-2*y+x**2+(-2)*x+2)+(10*y)/
    (y**2+x**2+(-2)*x+1)+(6*y)/(y**2+x**2+(-14)*x+49)


def glogxy(x, y):
    return -(4*(2*x-2)*(2*y-2))/(y**2-2*y+x**2-2*x+2)**2
    -(10*(2*x-2)*y)/(y**2+x**2-2*x+1)**2-(6*(2*x-14)*y)/
    (y**2+x**2-14*x+49)**2


def glogxx(x, y):
    return 8/(y**2-2*y+x**2-2*x+2)-(4*(2*x-2)**2)/
    (y**2-2*y+x**2-2*x+2)**2+10/(y**2+x**2-2*x+1)-
    (5*(2*x-2)**2)/(y**2+x**2-2*x+1)**2+6/
    (y**2+x**2-14*x+49)-(3*(2*x-14)**2)/
    (y**2+x**2-14*x+49)**2


def glogyy(x, y):
    return 8/(y**2-2*y+x**2-2*x+2)-(4*(2*y-2)**2)/
    (y**2-2*y+x**2-2*x+2)**2+10/(y**2+x**2-2*x+1)-(20*y**2)/
    (y**2+x**2-2*x+1)**2+6/(y**2+x**2-14*x+49)-
    (12*y**2)/(y**2+x**2-14*x+49)**2
```

```
def yx(x, y):
    return -glogx(x,y)/glogy(x,y)


def yxx(x, y):
    return (-glogxx(x,y)*glogy(x,y)**2+
    2*glogxy(x,y)*glogx(x,y)*glogy(x,y)-
    glogyy(x,y)*glogx(x,y)**2)/glogy(x,y)**3


def xFinder(x, y):
    return x-yx(x,y)*(1+yx(x,y)**2)/yxx(x,y)


def yFinder(x,y):
    return y + (1 + yx(x, y) ** 2) / yxx(x, y)


xNot = -330
yNot = 917


nextX = xNot
nextY = yNot

iterations = 7

nextXprec = dec.Decimal(xNot)
```

```
nextYprec = dec.Decimal(yNot)
ptm = time.process_time()
for i in range(iterations):
    prevXprec = nextXprec
    prevYprec = nextYprec
    nextXprec = xFinder(prevXprec, prevYprec)
    nextYprec = yFinder(prevXprec, prevYprec)
elapsed = time.process_time() - ptm
print('After ' + str(iterations) +
' iterations we find the following:')
print('Guess for x is ' + str(nextXprec))
print('Guess for y is ' + str(nextYprec))
print('The elapsed time for the method in seconds is'
, elapsed)
```

References

[1] Broyden, C. G. (1965. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation.* American Mathematical Society. 19 (92), 577–593.

[2] Cauchy, A. (1847). Méthode générale pour la résolution des systemes d'équations simultanées. *C. R. Acad. Sci. Paris*, 25, 536-538.

[3] Fletcher, R. (1970). A new approach to variable metric algorithms. *Computer Journal*, 13(3), 317-322.

[4] Goldfarb, D. (1970). A family of variable metric updates derived by variational means. *Mathematics of Computation*, 24 (109): 23–26

[5] Gradient descent. (2019, November 20). Retrieved from https://en.wikipedia.org/wiki/Gradient_descent

[6] Maxima. More information available from maxima.sourceforge.net

[7] Maximum Modulus Principle. (2019, November 15). Retrieved from https://en.wikipedia.org/wiki/Maximum_modulus_principle

[8] Nesterov, Yu. (1998). Introductory lectures on convex programming.

[9] Nesterov, Yu. (1983). A method for unconstrained convex minimization problem iwth the rate of convergence. $O(1/k^2)$. *Doklady AN USSR.* 269, 543-547

[10] Osculating circle. (2019, October 9). Retrieved from https://en.wikipedia.org/wiki/Osculating_circle

[11] Python. More information available from https://www.python.org/

[12] R. More information available from https://www.r-project.org/

[13] Rosenbrock, H. H. (1960). An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3), 175–184. https://doi.org/10.1093/comjnl/3.3.175

[14] Titchmarsh, E. C. (1939). *The theory of functions* (Second edition.). London: Oxford University Press.

[15] Weatherburn, C. E. (1930). *Differential geometry of three dimensions*. Cambridge [England]: University Press.