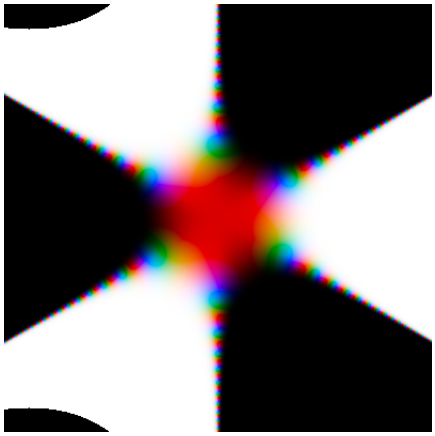
Complex Color Plots: New Methods and Insights

Kioshi Morosin

Summer 2021



1 Introduction

In real analysis, the graphical representation of a function $f : \mathbb{R} \to \mathbb{R}$ is standard: a set of points, each of which represents the value of f at a given value in the domain. However, in complex analysis a point in two-dimensional space is identified with a single complex number. This gives an intuitive graphical interpretation of functions $f : \mathbb{R} \to \mathbb{C}$ - a subset of $\mathbb{C}$ represented as a set of points. The graphical interpretation of a function $f : \mathbb{C} \to \mathbb{C}$ is less intuitive.

1.1 Definition. A *color space* is a finite set C of 3-vectors, each of which corresponds to a distinct perceptual color.

The color space readers will be most familiar with is the Standard Red-Green-Blue (sRGB) color space. It consists of triples (r, g, b) where each of $r, g, b \in [0, 1]$. This corresponds to the three colors of pixel used in modern color LCD and LED displays, with each component of the color representing relative intensity of the light emitted.

The physical parameters corresponding to each vector component of a color are arbitrary. For instance, we will find it useful to consider LCh color spaces, in which color components correspond to luminance, chroma, and hue resepctively. Informally, luminance refers to how "light" a color appears, so in traditional color spaces, a minimal luminance represents black and a maximal luminance represents white. Similarly, chroma represents how "colorful" the color is, which is similar to saturation.

We will also find it useful to discuss *Lab* color spaces, which have parameters L for lightness, and a and b which together encode chroma and hue. In particular, we will make use of the Oklab[1] color space, due to its (both conceptually and computationally) simple relationship with sRGB, and its perceptually uniform qualities.

1.2 Definition. Let C be an *LCh* color space. Then a *color map* is a function $\Psi : \mathbb{C} \to C$ such that the following hold:

1. Given complex numbers y and z, then $\Psi(y)_L = \Psi(z)_L$ if and only if $|y| = |z|$, that is, the lightness of the color depends only on the modulus of the complex number.
2. $\Psi(y)_h = \Psi(z)_h$ if and only if $\arg y = \arg z$, that is, the hue of the color depends only on the argument of the complex number.

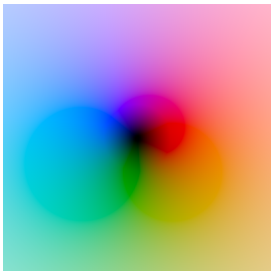
1.3 Remark. There are additional qualities required to make a color map practical. One might naïvely suppose that the sRGB color space is appropriate to render color plots of complex functions because it is the space used by virtually all computer graphics applications, corresponding to the physical pixels of their displays. However, the sRGB space has two major flaws. First, colors are encoded nonlinearly due to the process of gamma correction[2] (that is, the number in the (r, g, b) triple is nonlinearly) related to the intensity of the emitter. Second, the space is not perceptually uniform. This means that a gradient of color with varying hue but constant lightness and chroma will appear to the human eye to have varying lightness and chroma as well.

Given a color map, we can then evaluate a complex function over a two-dimensional lattice of points. At each complex point in the lattice, the value of the function will be mapped via a color map to a color, which can then be transformed into an sRGB color for display as an image.

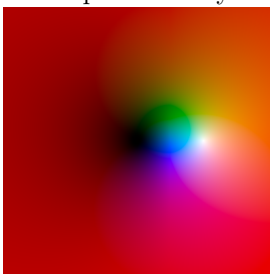
2 Applications of Color Plots

Much of complex analysis is focused on the characteristics of the zeros and singularities of meromorphic functions. Due to the relationship between lightness and modulus, in a color plot, zeros and poles are immediately apparent.

2.1 Figure. A color plot of the identity map. There is a zero at the center of the image.



2.2 Figure. A color plot with a zero in the center of the image and a pole directly to its right.

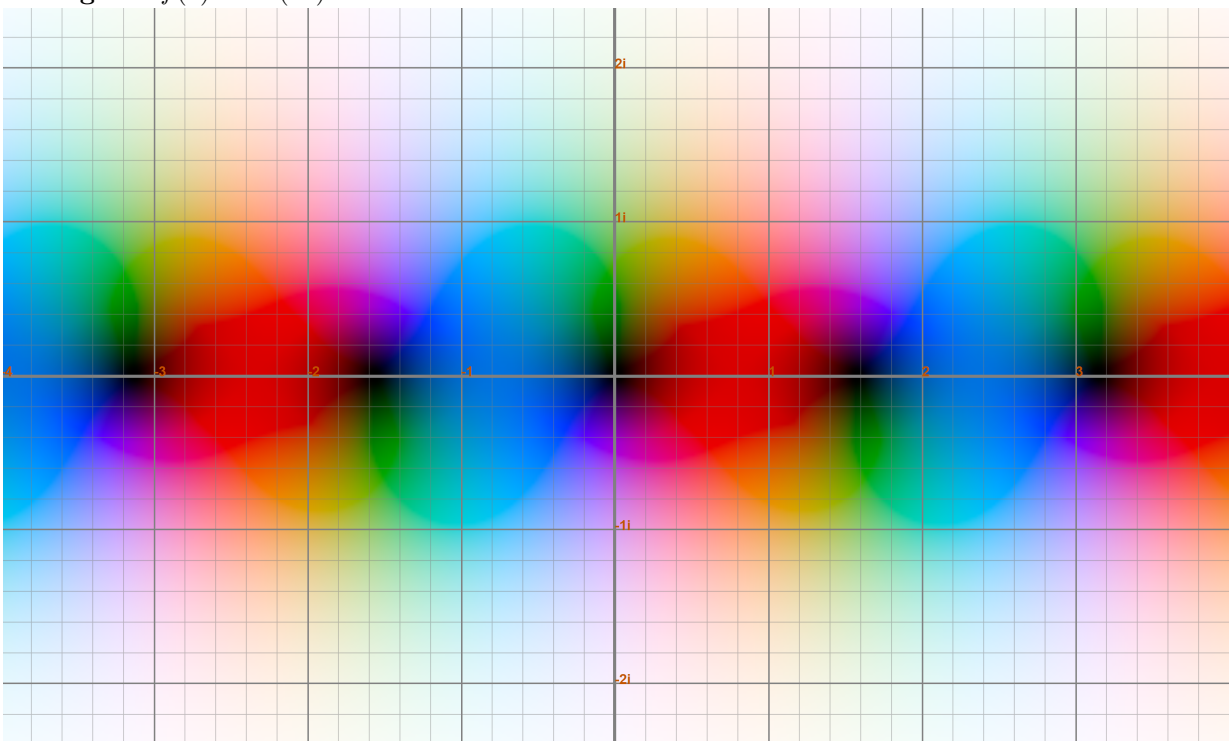


[1] https://bottosson.github.io/posts/oklab/

[2] https://en.wikipedia.org/wiki/SRGB#Specification_of_the_transformation

Extending this, we naturally attain a better understanding through color plots of periodic functions. For example, take the singly-periodic sine function (scaled for convenience):

2.3 Figure. $f(z) = \sin(2z)$



Immediately, we observe zeros at multiples of $\frac{\pi}{2}$, which aligns with our analytic understanding of the function. Not only is the periodicity readily apparent, we also observe that there is a pole at infinity, reached along any path that is not concurrent with the real line.

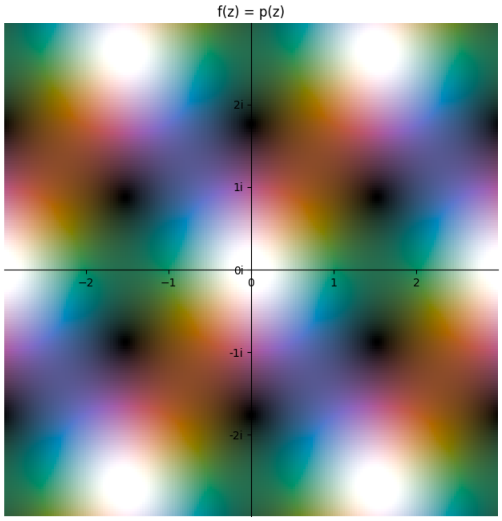
While the concept of a singly-periodic function is familiar to complex analysis students from Calculus, an understanding of doubly-periodic functions might be more elusive. For example, consider the function

$$\wp(z) = \frac{1}{z^2} + \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} \frac{1}{(z - m\omega_1 - n\omega_2)^2} - \frac{1}{(m\omega_1 + n\omega_2)^2}$$

for some complex parameters ω_1, ω_2. For a typical undergraduate student, the periodicity of this is not at all apparent, and even an explanation of the Weierstrass $\wp-$function may not lead to an understanding of doubly-periodic functions. However, the color plot of the $\wp$-function makes this clear:

Here, we can see the fundamental parallelogram of the $\wp$-function as an actual parallelogram, with vertices at each of the poles in the plot. Similarly, we see that the interior of each parallelogram is identical. Additionally, it is well-accepted that the zeros of the $\wp$-function are nontrivial to determine analytically. Here, a numeric method shows an approximate value of each of the two zeros inside a parallelogram and illustrates their symmetry.

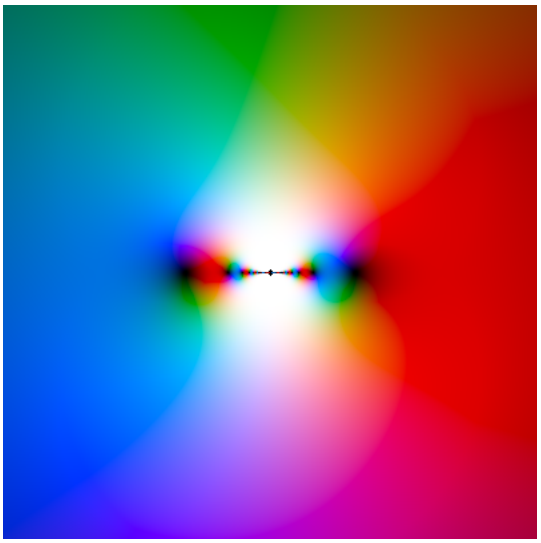
2.4 Figure.



Another concept notably enhanced by visualization is that of the essential singularity. Notably, the common definition that an essential singularity is a singularity which is neither removable nor a pole reveals nothing about the behavior of the function around it. Their behavior is instead specified by Picard's Great Theorem, but how specifically might a function satisfy the theorem's requirement that the image of any neighborhood of the essential singularity must contain every complex number? In short, what does an essential singularity *look* like?

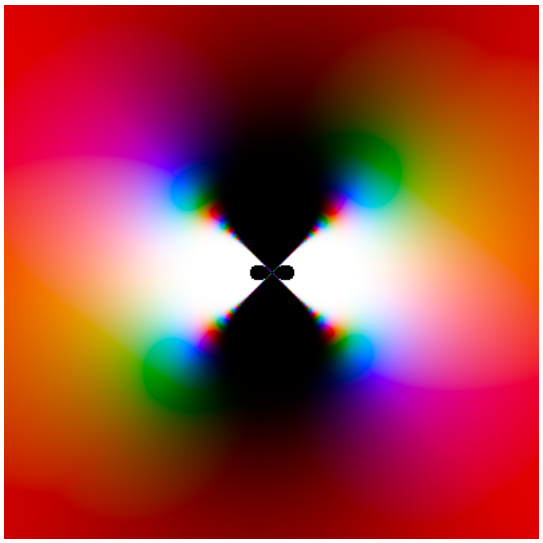
2.5 Figure.

$$f(z) = \sin\left(\frac{4}{z}\right)$$



2.6 Figure.

$$f(z) = e^{\left(\frac{10}{z^2}\right)}$$



This yields an analogue to the concept of order for a removable singularity: we can define the "order" of an essential singularity as the greatest order term of the negative part of the function's Laurent expansion. Graphically, this pseudo-order corresponds to the number of paths along which every complex value is attained.

2.7 Example. *To find the Laurent expansion for*

$$f(z) = \sin\left(\frac{1}{z}\right)$$

at $z = 0$, we observe that $\sin(z)$ is an entire function with Laurent expansion

$$\sin(z) = \sum_{k=1}^{\infty} (-1)^{n-1} \frac{z^{(2n-1)}}{(2n-1)!}$$

It is well-known that this series converges everywhere, so the Laurent expansion for f is

$$f(z) = \sum_{k=1}^{\infty} (-1)^{n-1} \frac{z^{-(2n-1)}}{(2n-1)!} = z^{-1} - \frac{z^{-3}}{3!} + \frac{z^{-5}}{5!} - \dots$$

*Thus the essential singularity of f at $z = 0$ has pseudo-order 1, indicating that there is one path along which all complex values are attained. This is confirmed by **2.5**, showing that this path is concurrent with the real axis.*

3 Glam

Motivation

Currently, the most accessible methods of creating complex color plots are using MATLAB or the Python libraries `matplotlib` and `cplot`. We chose the Python approach for our initial investigation because the MATLAB approach was less well-documented. Although the extensive ecosystem of mathematical Python libraries made plotting nontrivial functions (such as in **2.4**) easier, two flaws were apparent. First, for a function such as the $\wp-$function, it is slow. The plot in **2.4** was generated in approximately two minutes, which renders it impractical for the sort of quick experimentation one would use plots for in real analysis. Second, it requires programming knowledge, and often extensive boilerplate code to view a simple plot. This makes it inaccessible to the mathematics undergraduate.

To address this, we created Glam, a graphing calculator for complex-valued functions, designed to be comparable to the Desmos[3] graphing calculator for real-valued functions. By its nature, a graphing calculator should not require any knowledge to use beyond the ability to enter an equation, so our first concern was speed. While Javascript in browsers has been heavily optimized as of late, it is still unsuitable for large calculations (a small color plot such as **2.1** requires $160,000$ function evaluations). Thus, we chose to build the core analytic engine of Glam using C++ compiled to WebAssembly using Emscripten. This posed some difficulties, being a relatively new technology, but the performance, at least in theory, is unparalleled.

Design

The simple approach, regardless of language, would be to parse a mathematical expression into a list of computational operations using something like Dijkstra's shunting-yard algorithm. While this is acceptible for a traditional calculator, the volume of calculations required render this degree of abstraction away from the computer's processor suboptimal. Our solution was to perform "just-in-time" (JIT) compilation of a list of operators and operands directly to WebAssembly.

[3] `https://www.desmos.com/calculator`

JIT compilation to WebAssembly is achieved using `binaryen`[4], which also forms the backend for the Emscripten compiler toolchain. In practice, the compiler accepts a Javascript object holding a stack of operators and operands with the function

```
fxn<std::complex<double>, compiled_fxn<std::complex<double>>>
    math_compiler_dp::compile(const emscripten::val &stack) { /* ... */}
```

3.1 Remark. For machine-code targets, the standard way of loading JIT-compiled code using C/C++ is to map a region of memory (`mmap(2)`) with the executable flag set (`PROT_EXEC`), then copy the compiled binary code into the region. The start of the region can then be reinterpreted as a function pointer and the code can be executed. However, WebAssembly virtual machines isolate executable memory (the *code segment*) from readable/writable memory (the *data segment*). Thus, the traditional approach will not work. Instead, we create a new WebAssembly module to contain the JIT-compiled code.

`binaryen` uses an intermediate representation (IR) to programatically encode WebAssembly instructions that does not have a one-to-one correspondence with the outputted bytecode. Fundamentally, the IR is a tree of nodes, while WebAssembly bytecode is a set of stack instructions. Because our mathematical input takes the form of a stack, it is more logical to write stack-based instructions rather than tree-based ones. For each stack object, we process it in the following way:

```
auto type = stackObj["type"].as<int32_t>();
auto value = stackObj["value"].as<std::string>();
switch (type) {
    case 0: // NUMBER
        // we take advantage of boost's parsing even if we don't want
        // a multiprecision complex number
        fv->visit_complex(mp_complex(value).convert_to<std::complex<double>>());
        break;
    case 1: // IDENTIFIER
        if (!fv->visit_variable_dp(value)) {
            mv.abort();
            abort();
        }
        break;
    case 2: // OPERATOR
        visit_operator(fv, value);
        break;
    default:
        GLAM_COMPILER_TRACE("unrecognized stack object " << type);
        mv.abort();
        abort();
}
```

Observe the "dp" suffix on some of the above functions and classes. This denotes the precision of the JIT function. Our original idea was to use fixed-precision arithmetic via the Boost `multiprecision` library. This library could be used to guarantee that all calculations are performed to an arbitrary decimal precision. However, for nontrivial functions, we observed that computation of sufficient function values to generate a

[4] `https://github.com/WebAssembly/binaryen`

color plot was unacceptably slow, and noted that the additional precision would be discarded due to the limited gamut of colors available to display. While it is likely that fixed-precision arithmetic will be necessary for certain functions (especially those defined by slowly-convergent series, for example), we decided that double-precision arithmetic, which is intrinsically supported by WebAssembly, would be preferable.

When using double-precision arithmetic, we represent a complex number through the C++ type `std::complex<double>` and through two WebAssembly `f64`s. Thus, we implement a complex constant on the stack as

```
void function_visitor::visit_float(double d) {
    GLAM_COMPILER_TRACE("visit_float " << d);
    auto c = parent->module->allocator.alloc<wasm::Const>();
    c->type = wasm::Type::f64;
    c->value = wasm::Literal(d);
    visit_basic(c);
}

void function_visitor::visit_complex(std::complex<double> z) {
    visit_float(z.real());
    visit_float(z.imag());
}
```

In order to perform arithmetic on these numbers, we found ourselves limited by three design shortcomings of WebAssembly. First, the establishment of WebAssembly as a stack-oriented virtual machine rather than a register-oriented one was relatively recent. As a result, the specification lacks many instructions one would expect for a stack language: `dup`, `swap`, etc. Secondly, the WebAssembly single-instruction/multiple-data (SIMD) proposal[5] is still very new and not entirely adopted by web browsers. Therefore, much of what should be trivial in a stack machine is complicated by requiring the use of temporary local variables. For example, the function $f(z) = z + 1$ compiles as

3.2 Figure.

```
(func $__jit_a47d3cdb16f8caf7 (;1;) (export "__jit_a47d3cdb16f8caf7")
    (param $var0 f64) (param $var1 f64) (result i32)
    (local $var2 f64) (local $var3 f64) (local $var4 f64)
    local.get $var0
    local.get $var1
    f64.const 1
    f64.const 0.0
    local.set $var2
    local.set $var3
    local.set $var4
    local.get $var3
    f64.add
    local.get $var4
    local.get $var2
    f64.add
    global.get $env._arena
    i32.const 167
```

[5] https://github.com/WebAssembly/simd

```
    call_indirect (param f64 f64 i32) (result i32)
    return
)
```

This requires the use of three local variables to perform the addition. In order to simplify the JIT compiler code, we do not attempt to optimize the number of local variables used, and instead create a new one for each usage. However, WebAssembly VMs perform local variable optimization when the JIT module is loaded - the generated code for the above disassembly originally had six local variables, but three were optimized out.

Thirdly, WebAssembly (as it exists in most browsers at the time of writing) does not support functions with multiple return values. This means that the complex value returned from the compiled function must be a pointer to an address in memory. Storing the result of the function in memory is achieved using two design concepts: a fixed arena and a morpheme.

Because each call to the compiled function will require a deterministic number of allocations in memory, we allocate a fixed-size memory arena at compile time for the function. Each call to the arena's allocator will return the next cell of linear memory, and each call to the compiled function will reset the pointer to the next cell to the beginning of the arena. This has the effect of never requiring that memory used by a compiled function be freed, and never requiring new allocations while the function is being evaluated. This is in place of the direct control over the stack given to us by x86 assembly language - we can modify the stack like any other region of memory by means of a stack pointer, which is available to us. Because WebAssembly hides these implementation details from the module, we must use the fixed arena method.

3.3 Definition. We define a *morpheme* to be a small segment of code that performs a mathematical operation that cannot be conveniently written directly as WebAssembly.

For example, in order to exponentiate a complex number, we would like to use the C++ standard library function `std::pow`. However, the signature of this function is complicated due to the use of templates within the standard library. Instead, we define a morpheme function to wrap calls to `std::pow`. All morphemes of a given type have the same call signature. For instance, we define a morpheme that takes one complex number in the form of two double-precision floats as such:

```
#define DEFINE_f64x2(name) std::complex<double> * \
_fmorpheme_ ## name(double a, double b, fixed_arena<std::complex<double>> *arena)
```

For double-precision numbers, we pass the real and imaginary components of a complex number as separate function parameters to reduce memory allocations. In **3.1**, the `call_indirect` instruction is a morpheme invocation - the morpheme is `wrap`, stored at function index 167, and takes one complex number and produces a pointer to the `std::complex<double>` representing that number, which is allocated inside the function's arena. We then return this pointer from the function as a substitute for returning both parts together.

3.4 Remark. Morphemes are called from JIT code indirectly (as in **3.1**). This is because WebAssembly does not support dynamic linking between different modules. Because the functions being called are part of the main Glam module (which is always present), the idiomatic way to do this would be to add function imports to the module. However, these imports would require a layer of indirection through a Javascript function passed to the JIT module at load time, and the context switch between native code and Javascript is slower than an indirect function call.

After compilation, the JIT module is serialized to a binary format, loaded through the Javascript WebAssembly API, and wrapped in a generic `fxn` object. To produce a color plot, a lattice of points is derived from a rectangle in the complex plane and a resolution representing the number of samples to take per unit interval:

```
template <>
void multipoint<std::complex<double>, std::complex<double>>
        ::inner_init(const _domain_t &from,
                     const _domain_t &to,
                     uint32_t res) {
    GLAM_TRACE("inner init dp (C->C)");
    auto dim = to - from;
    samples = std::vector<_domain_t>(static_cast<size_t>(
            ceil(dim.real() * dim.imag() * res * res)
            ));
    latspace(from, to, _domain_t(1. / res, 1. / res), samples);
    this->colors = color_buffer(samples.size());
    GLAM_TRACE("initialized with " << samples.size() << " samples.");
}
```

As we iterate over the sample points to evaluate the function, we also convert the function value to an Oklab color:

```
struct Lab {
    float L;
    float a;
    float b;

    Lab() { }

    /**
    * computes a and b from chroma and hue
    * @param _L luminance
    * @param _C chroma
    * @param _h hue
    */
    Lab(float _L, float _C, float _h): L(_L), a(_C * std::cos(_h)), b(_C * std::sin(_h)) { }

    template <typename T> static Lab from_complex(T complex) {
        std::complex<double> z;
        if constexpr (is_mp<T>()) {
            // we convert to double-precision first because multiprecision
            // atan2 is *much* slower
            z = complex.template convert_to<std::complex<double>>();
        } else {
            z = std::complex<double>(complex);
        }
        const auto argument = static_cast<float>(arg(z) + M_PI_4);
        const auto modulus = static_cast<float>(abs(z));
```

```
            const float mod_scaled = modulus / (modulus + 1);
            const float x = 0.3f;

            return Lab(mod_scaled, x - x * 2.f * std::abs(mod_scaled - 0.5f), argument);
        }
    };
```

Observe that the modulus is scaled to be always less than 1, and the argument is translated such that a red color will correspond to the complex number 1. We also do our arithmetic as single-precision floating point numbers, noting that even double-precision numbers will not give significant benefit for the operations performed. Then, we convert the *Lab* color to sRGB, noting the piecewise-defined gamma correction function `from_linear`:

```
    struct rgba {
        uint8_t r;
        uint8_t g;
        uint8_t b;
        uint8_t a;

        rgba() { }

        static float from_linear(float x) {
            if (x <= 0.0031308f) {
                return 12.92f * x;
            }
            return 1.055f * std::pow(x, 1.f / 2.4f) - 0.055f;
        }

        explicit rgba(const Lab &lab, uint8_t alpha) {
            float l = std::pow(lab.L + 0.3963377774f * lab.a + 0.2158037573f * lab.b, 3);
            float m = std::pow(lab.L - 0.1055613458f * lab.a - 0.0638541728f * lab.b, 3);
            float s = std::pow(lab.L - 0.0894841775f * lab.a - 1.2914855480f * lab.b, 3);

            this->r = std::clamp(static_cast<int>(std::round(from_linear(4.0767416621 * l
                    - 3.3077115913 * m + 0.2309699292 * s) * 255)),
                                 0,
                                 255);
            this->g = std::clamp(static_cast<int>(std::round(from_linear(-1.2684380046 * l
                    + 2.6097574011 * m - 0.3413193965 * s) * 255)),
                                 0,
                                 255);
            this->b = std::clamp(static_cast<int>(std::round(from_linear(-0.0041960863 * l
                    - 0.7034186147 * m + 1.7076147010 * s) * 255)),
                                 0,
                                 255);
            this->a = alpha;
        }
    };
```

We include the `a` parameter (representing alpha, or transparency) such that an array `rgba[]` is binary-equivalent to the memory buffer in which Javascript represents an image. By doing this, we can return a Javascript `Float64Array` as a view of the color data without copying it and increasing computation time. Finally, we draw the color plot to a `canvas`, embedding it in an SVG `<image>` tag.

Future Work

Currently, Glam is still in-development, but a version usable for most elementary functions is available at `https://glam.hex.lc` with source code available under the Apache 2.0 License at `https://github.com/glammath/glam`. Development effort is presently focused on the user interface, which is currently limited by a fixed viewing window and an ad-hoc mathematical expression parser pending replacement. Bug reports and code contributions are welcome at the Github link.