Arbitrary Precision Computation of Modular Forms

Caelen Feller

Supervised by Prof. Jan Manschot

School of Mathematics, Trinity College Dublin

Abstract

We investigate and develop algorithms for arbitrary precision computation of modular forms using their q-expansions, and extend this to generalisations such as mock forms and higher level forms, implementing these algorithms in C. We develop an efficient plotting interface in C for complex functions, with particular optimisations for modular forms and functions.

1. Introduction

Modular forms, a particular class of holomorphic, complex-valued function on the upper half-plane, present a computationally interesting topic in numerical evaulation. There are useful optimisations to be made due to their self-similar, easily reduced structure and representations as quickly converging power series. A meromorphic extension to modular forms, modular functions, present similar opportunities. The most advanced algorithms for their numerical evaulation are detailed in [3] and [5] and implemented in the C library Arb [4] by Johansson. We detail and implement a more direct but less efficient algorithm. We discuss application of these approaches to generalisations such as "mock modular forms" - E_2, and extension to other modular forms for congruence subgroups.

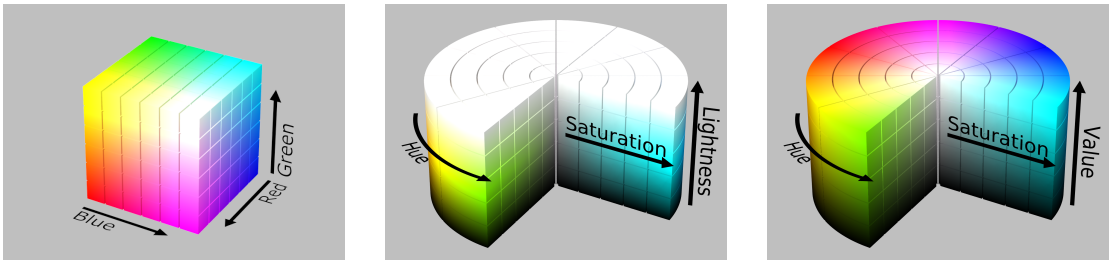
While numerical computation of modular forms has applications in sphere packing [10] and the evaluation of partition functions in physics [8], in this project the primary application is visualisation of their values. Visually representing complex-valued functions shows the ways in which the complex plane is transformed, represents points of interest (zeroes, singularities) and complex-analytic properties such as conformality. Though obvious for simpler, algebraic functions for more complicated, transcendental functions visualisation builds intuition not achievable otherwise. We develop optimised software to produce a broad range of these plots in this project.

2. Domain Colouring

The four dimensional nature of complex valued functions necessitates mapping the complex plane to a *colour space* to visually represent a complex valued function (on the plane or on a 3D surface), or ignore information as is done in "vector-flow" representations of complex valued functions. The map to a colour space is known as a *colour-function* of the complex plane, or "domain colouring" of a complex-valued function.

There are efficient implementations of this functionality in Mathematica nor in Sage, two of the leading computer algebra systems, though non standard implementations in packages like NumPy allow improvements - [2]. As a result CPlot (the complex plotting library developed in this project) was made in C for maximum efficiency, and access from high level systems can be impemented after.

Figure 1: Colour Spaces



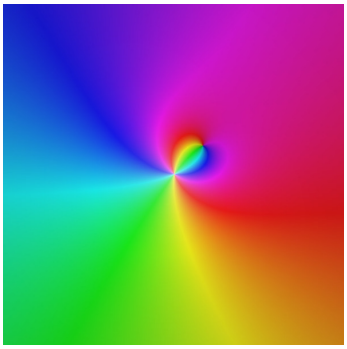
(a) RGB Cube (b) HSL Cylinder (c) HSV Cylinder

The colour space used for a computer screen is RGB colour space, a cartesian cube with red, green and blue light varying in intensity from dim to bright, creating all visible colours. Alternative colour spaces include the cylindrical colour spaces HSV and HSL (Hue, Saturation, Value, Lightness), where explicit white-black gradients are used for the radius an height, while hue is the angle, a wheel going through all colours, as demonstrated in Figure 1.

The standard colour function used in domain colouring is shown in Figure 2. Mathematically, it is defined below, though the constants can change to emphasise the argument or magnitude of values more.

Figure 2: Standard Colour Function

$$H = \frac{\arg(z)}{2\pi}$$
$$S = 1$$
$$L = 1 - 2^{-|z|}$$



Identity $(z - 0.5(1+i))/z^2$

Further colour functions are possible, see Appendix A for further functions and their plots.

3. Modular Forms & Functions

The structure of and computation of modular forms hinges on their transformation behaviour under $\mathrm{SL}_2(\mathbb{Z})$, a finitely generated integer matrix group. The following definitions and properties are standard, for proofs see [7], [9].

3.1. Special Linear Group

Definition 3.1 (Special Linear Group)**.**

$$\mathrm{SL}_2(\mathbb{Z}) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \;\middle|\; a, b, c, d \in \mathbb{Z}, ad - bc = 1 \right\}$$

$$\mathrm{SL}_2(\mathbb{Z}) = \langle S, T \rangle, \quad S = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \quad T = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

We define the group action of $\mathrm{SL}_2(\mathbb{Z})$ as a fractional linear transformation on $\mathbb{H}$

Definition 3.2 (Group Action - Möbius Transformation)**.**

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot z = \frac{za + b}{zc + d}, \quad z \in \hat{\mathbb{C}} \tag{1}$$

Definition 3.3 (Fundamental Domain)**.** A *fundamental domain* F for a subgroup Γ of $\mathrm{SL}_2(\mathbb{Z})$ is a closed subset of $\mathbb{H}$ such that:

1. Every $z \in \mathbb{H}$ is Γ-equivalent to a point in the closure of F.
2. No two distinct points in $\mathbb{H}$ are Γ-equivalent.

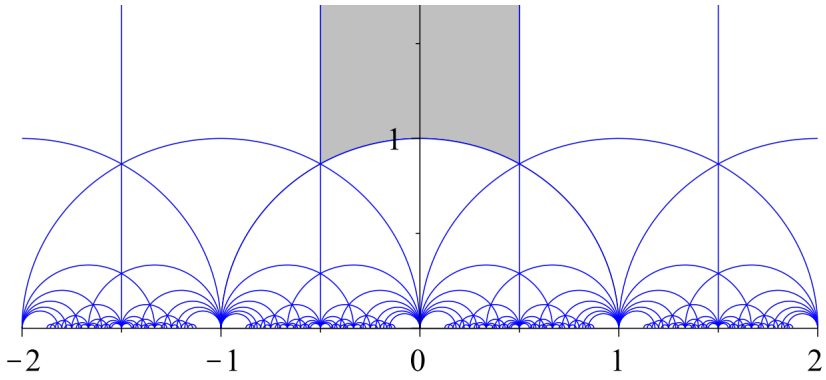


Figure 3: Principle Fundamental Domain for $\mathrm{SL}_2(\mathbb{Z})$: $F = \{z \in \mathbb{H} |\ |\Re(z)| \leq 1/2,\ |z| \geq 1\}$

3.2. Modular Forms

A function $f : \mathbb{H} \to \mathbb{C}$ transforms as a modular form of weight k if it satisfies

$$f(\tau + 1) = f(\tau), \quad f(-1/\tau) = (\tau)^k f(\tau) \tag{2}$$

Consequently, such an f is determined by its values on F (shaded region in fig 3).

Definition 3.4 (Modular Form)**.** A function $f : \mathbb{H} \to \mathbb{C}$ is a modular form of weight k if

1. f transforms as a modular form of weight k
2. f is holomorphic on $\mathbb{H}$.
3. f is holomorphic at $i\infty$

Lemma 3.5. *As $f(\tau+1) = f(\tau)$ every modular form will have a fourier series, expressible as*

$$f(\tau) = \sum_{n\in\mathbb{Z}} a_n q^n,\ q = e^{2\pi\tau i},\ \textit{the form's ``q-series''}$$

This is the form in which typically evaluate modularly transoforming functions, as it is a convergent power series in terms of q, a point on the complex unit disk.

Definition 3.6 (The Eisenstein Series of Weight k)**.**

$$G_k(\tau) = \sum_{\substack{m,n\in\mathbb{Z}\\(m,n)\neq 0}} \frac{1}{(m\tau+n)^k} = 2\zeta(k)\left(1 - \frac{2k}{B_k}\sum_{n=1}^{\infty}\frac{n^{k-1}q^n}{1-q^n}\right) \tag{3}$$

Where $\zeta(k)$ is the Riemann Zeta Function and B_k is the k-th Bernoulli Number.

Proposition 3.7. *G_k is a modular form of weight k for $SL_2(\mathbb{Z})$.*

Theorem 3.8. *The modular forms of weight k, $M_k(SL_2(\mathbb{Z}))$, form a finite dimensional, complex vector space, with basis a combination of G_4 and G_6.*

An algorithm to find these bases is discussed in [9], and implemented in Sage, though it's not relevant to my computatations which take forms in this form already, or as combinations of higher weight Eisenstein Series.

3.3. Modular Functions

Definition 3.9 (Modular Function)**.** A function $f\colon \mathbb{H}\to\mathbb{C}$ is a modular function of weight k if

1. f transforms as a modular form of weight k
2. f is meromorphic on $\mathbb{H}$.
3. f is meromorphic at $i\infty$

The primary example of this is the Klein j-invariant, which can be defined as

$$j(z) = 1728\frac{E_4(z)^3}{E_4(z)^3 - E_6(z)^2} \tag{4}$$

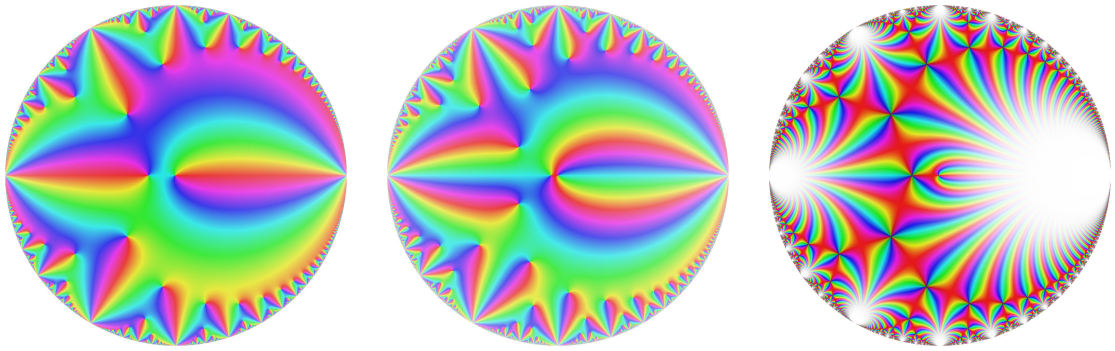
$$E_k = \frac{1}{2\zeta(k)}G_k = 1 - \frac{2k}{B_k}\sum_{n=1}^{\infty}\frac{n^{k-1}q^n}{1-q^n} \tag{5}$$

Its name stems from the fact that is invariant under action by $\mathrm{SL}_2(\mathbb{Z})$, and is thus a weight 0 modular function, as proved in [7].

4. Computation

To compute a transcendental function to arbitrary precision, one typically applies an argument reduction technique, and uses a rapidly convergent series exapansion of the function to evaluate it, computing any error due to its truncation. [5]. For example, to evaluate $\sin(x)$ using its Taylor series, we reduce the argument by the period, decreasing the magnitude and improving convergence by avoiding catastrophic cancellation. [5] However, we first must establish a method for dealing with arbitrarily large and precise numbers.

Figure 4: Plots of modular forms/functions



(a) $E_4(q)$ (b) $E_6(q)$ (c) $j(q)$

(For more plots, see Appendix A)

4.1. Arbitrary Precision and Interval Arithmetic

Numbers are represented programmatically to a specified level of precision as linked collections of word sized *limbs* for countable rings, or as $x \cdot 2^y$ for $\mathbb{R}$, $\mathbb{C}$, where x is the "mantissa" and y the "exponent", both arbitrary precision integers. These representations suit binary formats and optimised arithmetic. These standards are implemented in the MPFR and GMP libraries, both of which are used here.

While these representations are useful, and the current best solution for specialised algorithms, it is necessary to be constantly aware of rounding behaviour and error propogation, which does not suit more general number theoretic and mathematical application. A more mathematically stable method allowing easier error tracking and more predictable behaviour is *interval arithmetic* - representing intervals around a number as a ball with arbitrary precision center and radius, rather than the number itself. Arb, a C library, is the standard for ball arithmetic in numerical computing [4], providing automatic error propogation for simple arithmetic and implementations of many complex functions. As defined by Johansson, an "interval implementation" of a function preserves set inclusion, and thus guarantees containment of the function's correct value. This allows for integrated error tracking in the numerical data structure. While there can be issues with error overestimation, this is typically solvable by cautious analysis, as discussed in [4].

4.2. Error Bounding

As metioned in Section 4.1 simple arithmetic is handled by Arb. Thus, power series computation is easily handled. The task of bounding and computing the error caused by truncating the power series remains. The restrictions on the bounds given give us motivation for the level of argument reduction neceassary to implement a function. For the example of evaluation of Sine through its Taylor series, a bound for the truncation error, given in the Arb documentation, is the current term of the series. This follows from the fact that $\sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$ has strictly decreasing, alternating terms.

Theorem 4.1. *For $|q| < 1,\ q \in \mathbb{C}$,*

$$\sum_{n=k}^{\infty} \frac{n^K q^n}{1-q^n} \leq \frac{|q|^k}{|1-q|} \left(\sum_{i=1}^{K} \binom{K}{i} k^i \left(|q| \frac{d}{d|q|} \right)^{K-i} \frac{1}{1-|q|} \right) \tag{6}$$

Proof.

$$\begin{aligned}
&\sum_{n=k}^{\infty} \frac{n^K q^n}{1-q^n} \leq \sum_{n=k}^{\infty} \frac{n^K |q|^n}{|1-q|} \leq \frac{|q|^k}{|1-q|} \sum_{n=k}^{\infty} n^K |q|^{n-k}. \\
&= \frac{|q|^k}{|1-q|} (k^K + (k+1)^K |q| + \dots) \\
&= \frac{|q|^k}{|1-q|} \left(\sum_{i=1}^{K} \binom{K}{i} k^i \sum_{n=0}^{\infty} n^{K-i} |q|^n \right) \\
&= \frac{|q|^k}{|1-q|} \left(\sum_{i=1}^{K} \binom{K}{i} k^i \left(|q| \frac{d}{d|q|} \right)^{K-i} \frac{1}{1-|q|} \right) \text{ (by Lemma 4.2)}
\end{aligned}$$

□

Lemma 4.2.

$$\sum_{n=0}^{\infty} n^k x^n = \left(x \frac{d}{dx} \right)^k \frac{1}{1-x} \tag{7}$$

Note: This is a well known property of the polylogarithm function, $\mathrm{Li}_k(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^n}$

Proof. It is clear that

$$x \frac{d}{dx} \sum_{n=0}^{\infty} n^k x^n = \sum_{n=0}^{\infty} n^{k+1} x^n$$

The property desired follows from induction on k with the base case of $k = 0$ □

Corollary 4.3. *For $|q| < 1,\ q \in \mathbb{C}$, we can bound the error of E_4 using:*

$$\sum_{n=k}^{\infty} \frac{n^3 q^n}{1-q^n} \leq \frac{|q|^k}{|1-q|} \left(\frac{k^3}{1-|q|} + \frac{3k^2|q|}{(1-|q|)^2} + \frac{3k|q|(1+|q|)}{(1-|q|)^3} + \frac{|q|(|q|^2+4|q|+1)}{(1-|q|)^4} \right) \tag{8}$$

This follows from a simple application of 4.1, as does a similar bound for E_6.

4.3. Modular Forms

With an error bound found, we find a way to reduce the argument of a modular form. Fortunately, the transformation rule of a modular form readily suggests a method, taking the modular form to the fundamental domain of $\mathrm{SL}_2(\mathbb{Z})$. As any $z \in \mathbb{H}$ can be taken to F using the action of some $\gamma \in \mathrm{SL}_2(\mathbb{Z})$, and $\mathrm{SL}_2(\mathbb{Z}) = \langle S, T \rangle$, we can find some finite composition of S, T taking z to F (by definition of F and logic in [1]).

Computationally, we can take z to within some error ϵ of F with an iterated algorithm, similar to that used in the more geometrical proof that F is the fundamental domain of $\mathrm{SL}_2(\mathbb{Z})$ presented in [1]. An implementation of this algorithm is

in the Arb library, which is used internally and by this project. At a given working precision it does not guarrantee a transformation which will take the point to within ϵ of F, but working precision doing so is guaranteed to exist.

As $z \in F \implies |e^{2\pi i z}| < 1$, we can now apply the bounds from 4.2 to compute E_k to arbitrary precision using its q-series. While faster converging series exist, this method is the basis of generalisations to higher level forms in Section 6, and thus is of interest and easier to understand.

In particular, more efficient methods discussed in [3] are implemented in Arb for E_k. These use representations of E_4 and E_6 in terms of Jacobi theta functions, which transform as modular forms of weight 1/2, and are forms for a congruence subgroup. Their q-series converge more quickly as the exponent of q is quadratic in n. This method also benefits from the use of rectangular splitting, which reorders multiplication of the q-series in order to prioritise muliplications of similar precision values. This allows for low level improvements at high-precision in muliplication.

To recap, the direct algorithm for computing E_4, E_6 can be summarised as

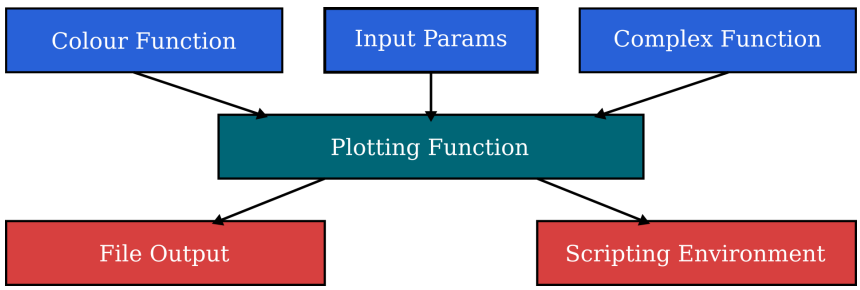
1. Reduce argument to F.
2. Calculate truncated q-series for reduced argument term by term.
3. Compute error for each value of k, finish when sufficiently small.
4. Apply tranformation rule, getting final value.

5. Implementation

5.1. CPlot - Arb based Plotting Library

The abstract structure of the plotting library is shown in Figure 5. The implementation provides the facility to plot any Arb-based interval implementation of a function, in such a way that any zoom level or resolution is possible, memory allowing. Due to "chunk-based" image processing methods (aka pipeline processing), the only memory overhead is that incurred by parallel evaluation of values, which is mostly a CPU intensive task unless memory caching is necessary - i.e. in repeated evaluation of a reccurence relation. The chunks are cached to disk immediately, and post processed into one image, allowing any other compositing to take place.

Figure 5: Library Structure for CPlot



In the colour function computation arbitrary precision can be discarded for machine size numbers once we are manipulating only RGB values, as a screen/image format can only display a certain number of colours.

5.2. CForm - Modular Form Extension Library for Arb

The various algorithms for computing E_4 and E_6 are discussed in 4.3. The direct methods are implemented in CForm, while the theta algorithms are in Arb. The Eisenstein series of higher weight can be computed by reccurrence relation.

$$\sum_{k=0}^{n} \binom{n}{k} d_k d_{n-k} = \frac{2n+9}{3n+6} d_{n+2},\ d_k = (k+3/2)k!G_{k+2}$$

Another feature implemented in CForm, excepting the extensions discussed in Section 6, is evaluation of polynomials and rational functions of E_4 and E_6, a common task when evaluating arbitrary modular forms and functions. This is optimised by decomposing into the theta series neceassary to compute the function. Simplification is a futher optimisisation that could be implemented, requiring additional computer algebra, the overhead of which is potentially not worth the gain at low precisions.

6. Further Research

6.1. Higher Level Modular Forms, Mock Modular Forms

There are two directions of generalisation explored by this project. The first are modular forms for congruence subgroups of $\mathrm{SL}_2(\mathbb{Z})$. The second generalisation is that of relaxing the transformation rule, to allow what are called "mock modular forms", which have an error term when transformed. A notable example of this is E_2, defined in the same way as E_k where $k = 2$. For mock modular forms on $\mathrm{SL}_2(\mathbb{Z})$, the computation method is the same as detailed in Section 4, with a different error bound depending on the q-series of the form, and computation of the transformation error when brought to the fundamental domain.

6.2. Computational Techniques for Congruence Forms

The primary congruence subgroups of $\mathrm{SL}_2(\mathbb{Z})$ are:

$$\Gamma_0(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} a & b \\ 0 & d \end{pmatrix} \bmod N \right\}$$

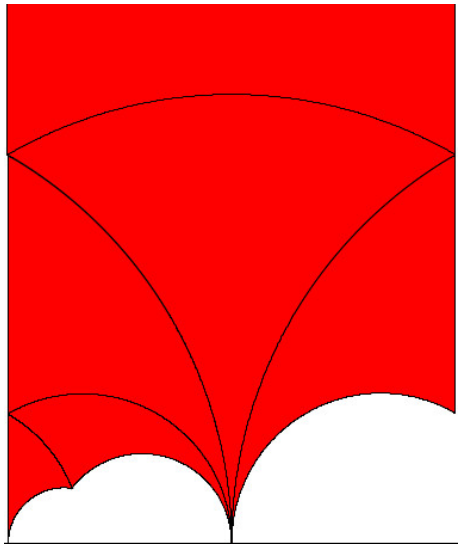
$$\Gamma_1(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \bmod N \right\}$$

$$\Gamma(N) = \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \bmod N \right\}$$

Futher congruence subgroups exist, and many analogues of properties present for forms on $\mathrm{SL}_2(\mathbb{Z})$ apply. See [7] for a more complete treatment.

Modular forms on a congruence subgroup must be holomorphic at all points in the fundamental domain of that subgroup, including at all "cusps" of that subgroup.

For $\Gamma_0(N)$ and $\Gamma_1(N)$, we can still expand forms as a fourier series in terms of q. For $\Gamma(N)$, we can only expand as a fourier series in terms of $q^{1/N}$ [6]. The complication is while there exists a q-series, we cannot transform points to F in the same way, as the fundamental domain of a congruence subgroup Γ can have cusps - points where we get arbitrarily close to the real line, and thus the q-series does



Fundamental Domain for $\Gamma_1(4)$ - Cusps on $\mathbb{R}$ Visible

not quickly converge. Also, the same iterated algorithm to transform points to the fundamental domain is more complicated, as we cannot only use the generators S and T as in Section 4.3.

A potential solution to this is the use the abstraction of vector valued modular forms, but this adds many computational complications we have not addresed yet. As a result, a general solution for all congruence subgroups is difficult to achieve.

6.3. Planned or Incomplete Features

A major extension to the project that will be completed after the final release of the libraries is the creation of a wrapper for the libraries in Sage and Mathematica, using their C interfacing methods, WSTP and Cython/the Python C API. As a wrapper for Arb itself is already present in Sage, deeper integration is more easily achievable.

Potential improvements to the plotting interface rest in how it interfaces with hardware, such as computing the graphs more simultaneously on clusters of machines or graphics cards using CUDA or similar, and internal efficiency improvements. Tuning the internal working precisions to allow maximum efficiency with minimum visual impact is possible as certain areas of a graph need less/more detail. The way to implement this under Arb's philosophy of error handling would be to use an adaptive subsampling method, giving greater weight to areas with high input sensitivity, but this would require greater knowledge of function behaviour.

7. Conclusion

This project has achieved all goals in computing basic modular forms and functions, and broached new ground in beginning to approach computation of more general modular forms, but further research is needed to advance this. CPlot accomplishes all goals satisfactorily in terms of domain coloured graphs, and while further features and optimisations are possible, they are not urgently needed, outside of the more interactive wrapper being developed.

The libraries developed will released online under an open source license, and ideally any new advances in more general modular form computation can be contributed to the Arb library as this smaller project progresses, in order to be subjected to more rigorous testing, futher public use, and attention from major contributors in the field such as Johansson.

References

[1] K. Conrad. Sl2 (z). *Expository note available at http://www. math. uconn. edu/˜ kconrad/blurbs*.

[2] Empet. Visualizing complex-valued functions with Matplotlib and Mayavi.

[3] A. Enge, W. Hart, and F. Johansson. Short Addition Sequences for Theta Functions. 2018.

[4] F. Johansson. Arb: Efficient Arbitrary-Precision Midpoint-Radius Interval Arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, aug 2017.

[5] F. Johansson. Numerical Evaluation of Elliptic Functions, Elliptic Integrals and Modular Forms. 2018.

[6] L. J. P. Kilford. *Modular Forms: A Classical and Computational Introduction Second Edition*. World Scientific Publishing Company, 2015.

[7] N. Koblitz and N. Koblitz. *Introduction to Elliptic Curves and Modular Forms*. Graduate Texts in Mathematics. Springer New York, 1993.

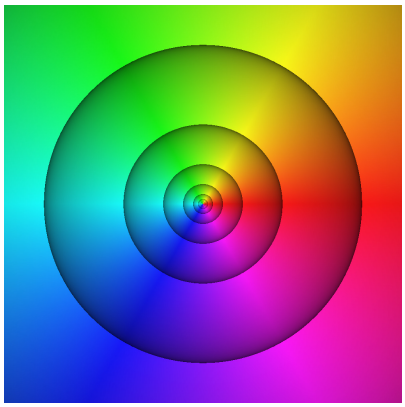
[8] J. Polchinski. String theory, volume 1: An introduction to the bosonic string theory, 2005.

[9] W. A. Stein and P. E. Gunnells. *Modular Forms: A Computational Approach*.

[10] M. S. Viazovska. The sphere packing problem in dimension 8. *Annals of Mathematics*, pages 991–1015, 2017.

Appendix A. Plots

Figure A.6: Radial Logarithmic Colour Function

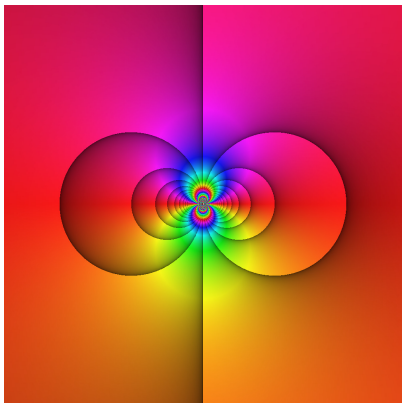


Identity

$$H = \frac{\arg(z)}{2\pi}$$

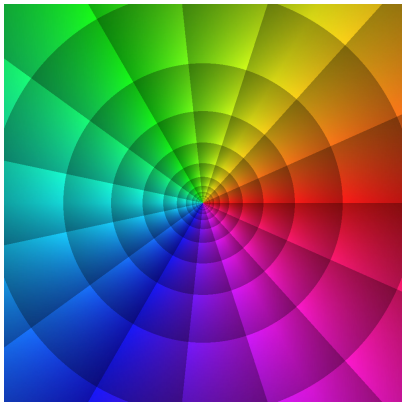
$$S = .9$$

$$V = \lceil \log_2(|z|) \rceil - \log_2(|z|)$$



$e^{1/z}$

Figure A.7: Conformal Colour Function



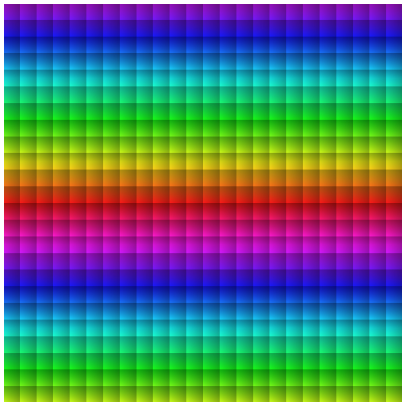
Identity

$$H = \frac{\arg(z)}{2\pi}$$

$$S = .9$$

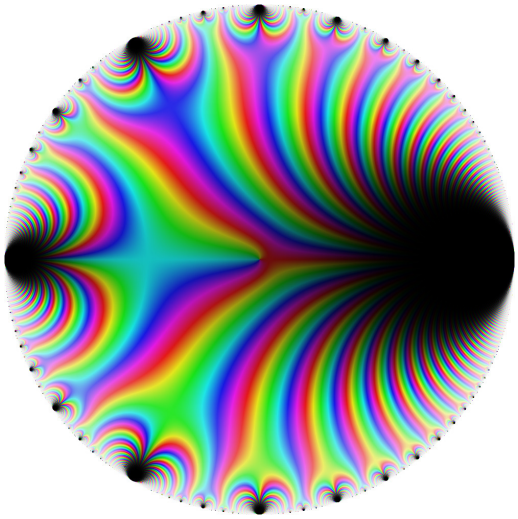
$$f(x) = (\lceil x \rceil - x)(M - m) + m$$

$$V = \lceil \log_2(|z|) \rceil - \log_2(|z|)$$

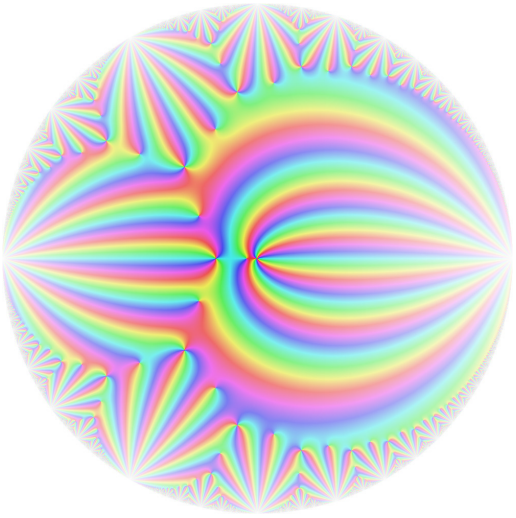


e^{z}

Figure A.8: Further Plots

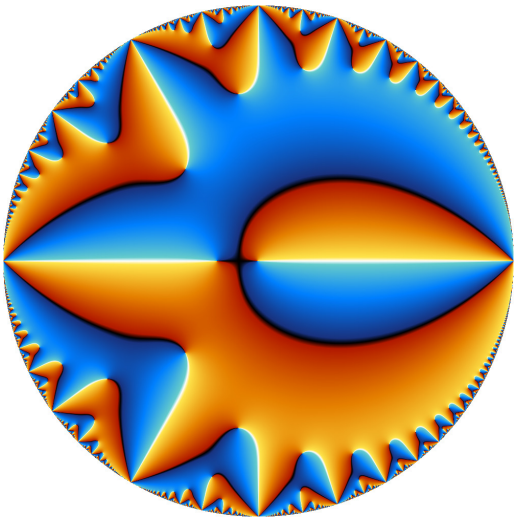


(a) $\Delta(q) \in M_{12}(\mathrm{SL}_2(\mathbb{Z}))$

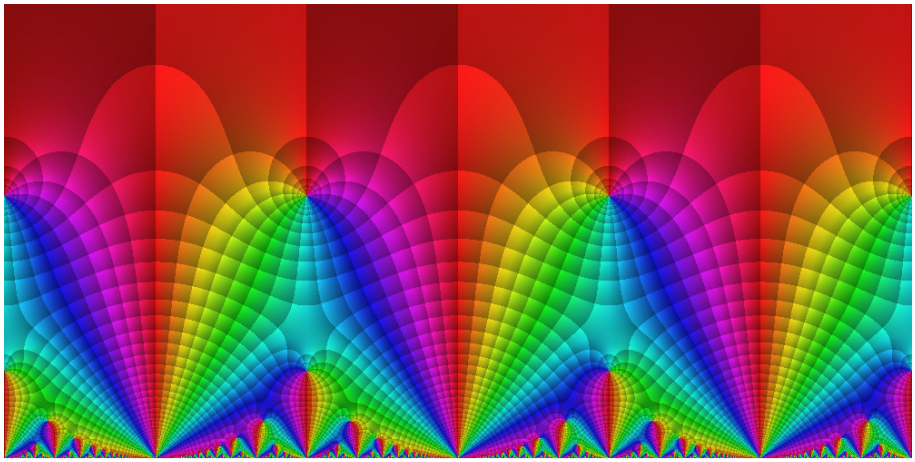


(b) $E_{12}(q)$

(c) Quantitative Colour Func



(d) $E_4(q)$



(e) $E_4(z)$