THE ODD FIBONACCI ZETA FUNCTION

DAVID LOWRY-DUDA

Abstract. We look at the (odd) Fibonacci zeta function and comment briefly on the behavior of its zeros and poles.

1. Introduction

In forthcoming work [AKLDW24a, AKLDW24b] with my collaborators Eran, Chan, and Alex, we describe different ways to understand the Fibonacci zeta function

$$Z_{\mathrm{Fib}}(s) := \sum_{n\geq 1} \frac{1}{F(n)^s} = 1 + 1 + \frac{1}{2^s} + \frac{1}{3^s} + \cdots$$

and certain generalizations associated to real quadratic fields. I gave a talk [LD20b] about this in 2020 and it's sat in various states since then.

The focus of our forthcoming papers is that these objects (and their generalizations) can be understood in several different ways — including one way through modular forms! This is surprising and interesting.

This note is not as surprising nor as deep. In this focused technical note, I consider the question of studying the zeros and plot of the Fibonacci zeta function.

Actually, I restrict my attention to the *odd Fibonacci zeta function*

$$Z(s) := Z_{\mathrm{Fib}}^{\mathrm{odd}}(s) = \sum_{n\geq 1} \frac{1}{F(2n-1)^s},$$

consisting of odd-indexed terms of the Fibonacci zeta function. Although it's possible to study the whole Fibonacci zeta function (or alternately just the even-indexed terms, the *even Fibonacci zeta function*), the odd Fibonacci zeta function has a simple analytic description and behaves in many way like a simplest component.

In our work, we prove the following continuation.

Theorem 1. *Write* $\varepsilon = \frac{1+\sqrt{5}}{2}$. *Then for all* $s \in \mathbb{C}$ *away from poles of the summands,*

$$Z(s) := Z_{\mathrm{Fib}}^{\mathrm{odd}}(s) = \frac{5^{s/2}}{8\Gamma(s)\log\varepsilon} \sum_{m\in\mathbb{Z}} (-1)^m \Gamma\Big(\frac{s}{2} + \frac{\pi i m}{2\log\varepsilon}\Big)\Gamma\Big(\frac{s}{2} - \frac{\pi i m}{2\log\varepsilon}\Big). \tag{1}$$

I take this for granted and defer the proof to our forthcoming work.

This continuation shows many properties. There are trivial zeros (coming from poles of $\Gamma(s)$) that are reminiscent of the trivial zeros of the standard Riemann zeta function, $\zeta(s)$; and there is a half-lattice of poles from the other Gamma functions. Other than that, this is a Dirichlet series with meromorphic continuation.

Date: December 4, 2024.

This work was supported by the Simons Collaboration in Arithmetic Geometry, Number Theory, and Computation via the Simons Foundation grant 546235.

It is easy to write down Dirichlet series, but a random Dirichlet series will almost surely have a natural boundary on the abscissa of convergence. Dirichlet series with continuation are special.

On the other hand, I've almost always studied *automorphic* L-functions, which are the nicest possible family of Dirichlet series.

We examine two guiding questions in this note:

(1) What does $Z(s)$ look like?
(2) Where are its zeros?

Through this exploration, we might build up additional intuition. If we only look at the nicest possible Dirichlet series, we can forget what makes them special or what properties actually distinguish them.

2. Making Plots

From (1) and Stirling's formula, it's clear that the summands decay extremely quickly in m. Using only the first $2|\mathrm{Im}\, t|$ summands will be sufficient. I always use at least 20 to prevent over approximating. Thus making a plot of $Z(s)$ reduces to quickly computing lots of Gamma functions, adding them together, and then visualizing the result.

I typically use `Sagemath` [Sag20] for plots of complex functions (especially as I wrote the complex plotting routines in [LD20a]). But it turns out that sage is *stunningly slow* at making nice plots of $Z(s)$.

So instead, I describe here a simple implementation in C++ (with plotting in python). This takes two steps.

(1) First, compute an approximation of $Z(s)$ on a square grid of points.
(2) Then import this grid of points into python for plotting.

2.1. Computing $Z(s)$ on a mesh.

The only nontrivial computational component is computing the gamma function at complex arguments. There are many implementations, but I tend to use `arblib` (now part of `flintarb`) [Joh17] for all sorts of special functions, unless it's not sufficient for some reason.

Remark 2. `arb` gives provably strong results and typically works in interval arithmetic. This takes additional work. Here I immediately throw away the intervals and just use the midpoint approximations. This is wasteful, but `arb` is fast and correct. What more could I ask for?

I use `acb.h` and `acb_gamma` from `arb`.

```
#include <complex>
#include <acb.h>
#include <arb.h>
#include <arf.h>

std::complex<double> gamma(const std::complex<double>& z) {
  const slong prec = 53;
  acb_t zz, res; acb_init(zz); acb_init(res);
  acb_set_d_d(zz, z.real(), z.imag());
  acb_gamma(res, zz, prec);
  double real = arf_get_d(
      arb_midref(acb_realref(res)), ARF_RND_NEAR
  );
  double imag = arf_get_d(
      arb_midref(acb_imagref(res)), ARF_RND_NEAR
```

```
  );
  acb_clear(zz); acb_clear(res);
  return std::complex<double>(real, imag);
}
```

In practice, `gamma` shadows a common map, so I actually put this in my own namespace `dld::gamma`.

Computing $Z(s)$ is now completely routine. Take m up to $\max(20, \lceil |\mathrm{Im}(s)| \rceil)$ in (1) and add them up. Complete code is included in the appendix.

Making the Mesh. I use a slightly nontrivial way to compute the grid itself — nontrivial because I use multithreading to make this a parallel computation.

The overall structure is simple: divide the rows into `number-of-threads` many groups and then assign appropriate rows to each thread. For simplicity I have each thread write its rows to a separate temporary file and then combine them together afterwards.

This is my extremely simple form of multithreading. If some rows are far more computationally intense (which they are), then this won't evenly distribute computational load. But the naive independence means that I don't have to worry about shared memory or mutexes or other concurrency problems. A small possible improvement would be to split into more chunks and have a thread pool, but that added complexity doesn't seem worth it to me.

Regardless, the output is a CSV containing (x, y, value) rows. I use two variants: one where the values are $\arg(Z(x+iy))$ and the other where the values are the pair $\big(\mathrm{Re}(Z(x+iy)), \mathrm{Im}(Z(x+iy))\big)$

2.2. **Plotting the data.** Perhaps the most obvious way to plot the resulting data would be to assign your domain coloring (cf. [LD21], or `https://en.wikipedia.org/wiki/Domain_coloring`, or the beautiful visualizations of Frank Farris) and then color each computed point. Unfortunately, this requires an enormously dense grid of computed pixels to make a good looking plot.

Instead, we interpolate between computed pixels. And by "we", I mean `matplotlib`, as this is implemented with various interpolation algorithms in `imshow`.

Alternately, to plot lines when the real or imaginary parts are 0, we can use `matplotlib`'s `contour`, which implements a marching squares algorithm for contour evaluation.

Both approaches can introduce artifacts. Fortunately, we are plotting holomorphic functions and the local behavior is either tame or too complex for any plotting method to have a chance (in practice). And in practice the viewer can detect when an artifact is introduced because it looks funny.

The broad look of these plotting routines in python look like the following.

```
# x_vals, y_vals, arg_vals, real_vals, imag_vals from CSV
import matplotlib.pyplot as plt

## I stored these in most significant bit order, hence 'F'
arg_grid = arg_vals.reshape((grid_size, grid_size), order='F')
plt.imshow(arg_grid,
           extent=[x_vals.min(), x_vals.max(),
                   y_vals.min(), y_vals.max()],
           origin='lower', interpolation="catmul")

# Or, for contours #
plt.contour(imag_grid, colors="#f9ae54",
```

```
            extent=[x_vals.min(), x_vals.max(),
                    y_vals.min(), y_vals.max()],
            origin="lower")
plt.contour(real_grid, colors="#0482d7",
            extent=[x_vals.min(), x_vals.max(),
                    y_vals.min(), y_vals.max()],
            origin="lower")
```

These produce the images in Figure 1. The argument plot in the Figure uses a colormap that is discontinuous at the boundary, which (in my normalization) amounts to a light-dark discotinuity when the imaginary part is 0.

3. Comments on the Plots and Zeros

I chose to plot in the rectangle with opposite vertices $-35-5i$ and $5+35i$. The aspect ratio is 1 because I almost always want my aspect ratio to be 1. The odd Fibonacci zeta function $Z(s)$ is rather boring for $\operatorname{Re} s > 0$ and antisymmetric over the real axis, so it's not worth including too much in those directions.

We know that $Z(s)$ has a half-lattice of poles, so we expect the lattice like regularity. We also understand the alternating zeros and poles on the real line.

But it turns out that there seems to be approximately one zero rather close to every pole.

Initially I thought this was surprising. But the zero and polar behavior is largely constrained by Jensen's Formula. I state it in two forms (both of which I learned as Jensen's Formula as a grad student).

Theorem 3 (Jensen's Formula V1)**.** *Let $f \not\equiv 0$ be meromorphic on the closed disk $B_R(0)$. Let $a_1, \ldots, a_p$ denote the zeros of f in $B_R(0)$, counting multiplicities, and let $b_1, \ldots, b_q$ denote the poles in $B_R(0)$, also with multiplicities. Then for any z in $|z| < R$ which is not a zero or a pole,*

$$\log|f(z)| = \int_0^{2\pi} \frac{R^2 - |z|^2}{|Re^{i\theta} - z|^2} \log|f(Re^{i\theta})| \frac{d\theta}{2\pi} - \sum_{i=1}^{p} \log\left|\frac{R^2 - \overline{a_i} z}{R(z - a_i)}\right| + \sum_{j=1}^{q} \log\left|\frac{R^2 - \overline{b_j} z}{R(z - b_j)}\right|.$$

Theorem 4 (Jensen's Formula V2)**.** *With the same notation as above, except omitting zeros or poles at $z = 0$; let $f(z) = c_f z^{\operatorname{ord}(0)} + \ldots$, where c_f is the leading nonzero coefficient of the Laurent expansion around 0. Then*

$$\log|c_f| = \int_0^{2\pi} \log|f(Re^{i\theta})| \frac{d\theta}{2\pi} - \sum_{i=1}^{p} \log\left|\frac{R}{a_i}\right| + \sum_{j=1}^{q} \log\left|\frac{R}{b_j}\right| - (\operatorname{ord}(0)) \log R. \tag{2}$$

The second form comes essentially from expanding at $z = 0$ after modifying and removing the zero/polar behavior there.

The point is that $Z(s)$ grows too slowly to have interesting polar or zero behavior. Rearranging (2) and estimating growth, we observe that

$$-\sum_{i=1}^{p} \log\left|\frac{R}{a_i}\right| + \sum_{j=1}^{q} \log\left|\frac{R}{b_j}\right| \ll \log R.$$

(I use $\ll$ here to mean that the LHS is bounded in magnitude and not merely bounded above). There are approximately R^2 poles within R of the origin, in a half-lattice. In order to counteract this growth, there must be regularly spaced zeros as well.

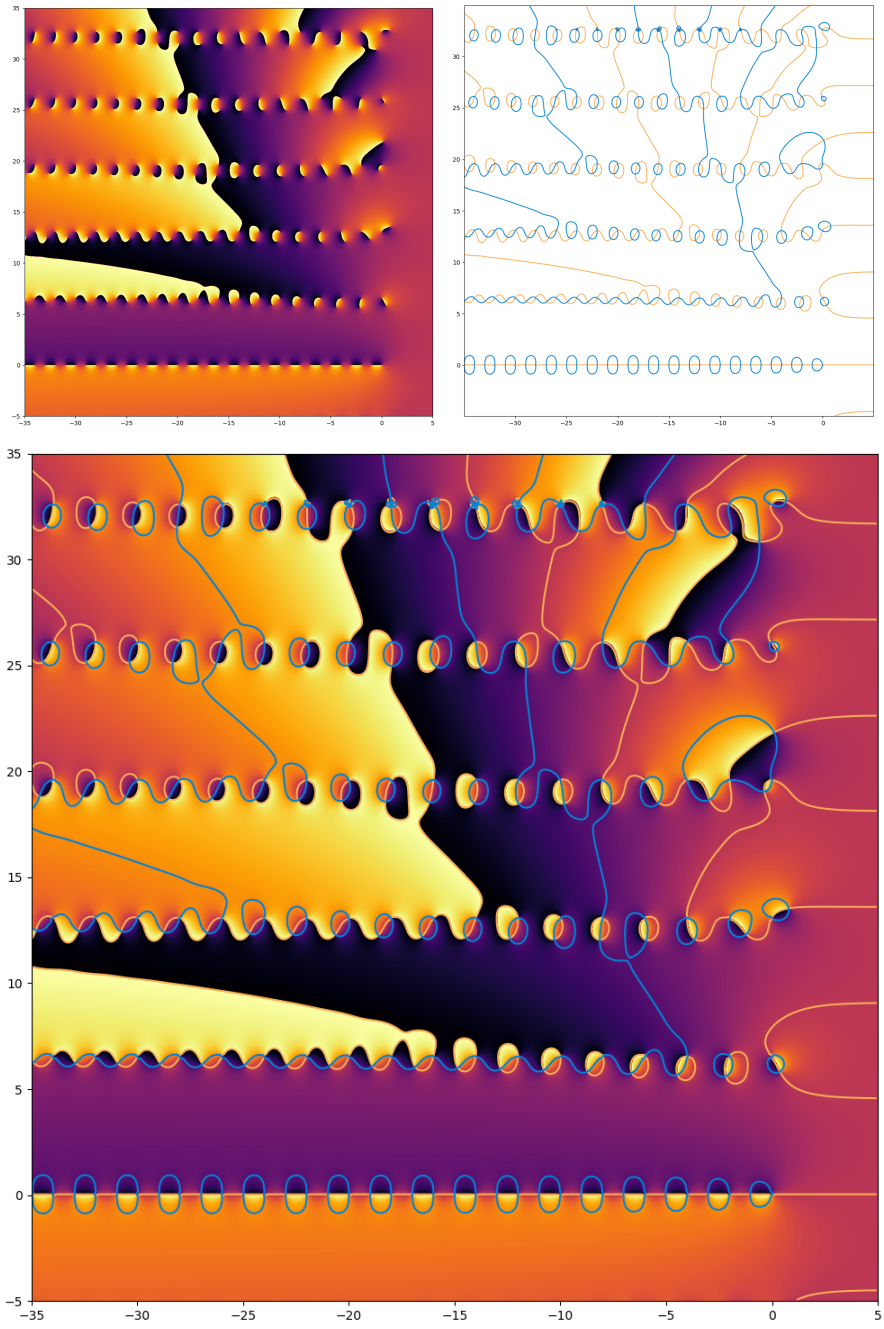

FIGURE 1. Interpolated `imshow` and `contour` plots of $Z(s)$. The last plot is the combined plot.

It's probably possible to say more about the distribution. I suspect that performing argument-principle type evaluation would show that the difference in the number of zeros and poles in a circle is bounded by $O(\log R)$, but I don't carry out this analysis.

3.1. **Specific Zeros.** Do the zeros have any meaning? Should we be able to predict them individually, as opposed to in distribution?

As a first step, we compute a couple of zeros using Newton's method in Figure 2.

0.41054841	+	5.80204742i
0.82209560	+	13.5268604i
-0.32303194	+	19.0607822i
0.13195381	+	21.8573826i
0.23103157	+	25.7125161i

FIGURE 2. The first five zeros corresponding to the first five poles of $Z(s)$.

We make several remarks.

First, these seem to be where we expect from the plots.

Second, there aren't obvious patterns or regularities in the real or imaginary parts (aside from being spaced about as far apart as the poles). Conceivably, these could be algebraic over $\mathbb{Q}[\sqrt{5}, \log(\varphi), \pi]$. Small tests don't support this. (It would not be hard to compute these and many more to much higher precision, and then do a more sophisticated analysis. I don't do this).

Third, several zeros occur in the region of absolute convergence of the underlying Dirichlet series. As Dirichlet series in the region of absolute convergence behave like almost periodic functions, we know there are at least $\asymp T$ zeros of height up to T also in the region of absolute convergence, distributed approximately periodically (cf. [Bes54]).

That's the most I know how to say about their regularity.

Appendix A. Code

A.1. **Gamma Computation.** The header `gamma.hpp`:

```cpp
// gamma.hpp
#ifndef DLD_GAMMA_HPP
#define DLD_GAMMA_HPP

#include <complex>

namespace dld {
  std::complex<double> gamma(const std::complex<double>& z);
}
#endif
```

And the code, largely as shown above.

```cpp
// gamma.cpp
#include "gamma.hpp"
#include <acb.h>
#include <arb.h>
#include <arf.h>

namespace dld {
  std::complex<double> gamma(const std::complex<double>& z) {
    const slong prec = 53;
    acb_t zz, res;
    acb_init(zz); acb_init(res);

    acb_set_d_d(zz, z.real(), z.imag());
    acb_gamma(res, zz, prec);

    double real = arf_get_d(
        arb_midref(acb_realref(res)),
        ARF_RND_NEAR
    );
    double imag = arf_get_d(
        arb_midref(acb_imagref(res)),
        ARF_RND_NEAR
    );
    acb_clear(zz); acb_clear(res);
    return std::complex<double>(real, imag);
  }
}
```

A.2. **Fibonacci Zeta.** I have a minimal header.

```cpp
// fibo.hpp
#ifndef DLD_FIBO_HPP
#define DLD_FIBO_HPP

#include <complex>

std::complex<double> zodd_smart(const std::complex<double> & s);

#endif
```

The code looks annoying only because C++ can be verbose. It's precisely (1), using standard library computations and the gamma function computation above.

```cpp
// fibo.cpp
#include "fibo.hpp"
#include "gamma.hpp"
#include <cmath>
#include <complex>

typedef std::complex<double> complex;
const long double PI = 3.141592653589793;

// log( (1 + sqrt(5))/2 )
const double logeps = 0.481211825059603;
const double sqrt5 = 2.23606797749979;

complex compute_argument(const complex & s, int m) {
  return s/2.0 + complex(0.0, PI * m / (2*logeps) );
}

complex summand(const complex & s, int m) {
  complex ret;
  if (m % 2 == 0) { ret = 1.0; }
  else { ret = -1.0; }
  ret *= dld::gamma(compute_argument(s, m));
  ret *= dld::gamma(compute_argument(s, -m));
  return ret;
}

complex zodd(const complex & s, int limit=1000) {
  const double logsqrt5 = std::log(sqrt5);
  complex ret = std::exp(s * logsqrt5);
  ret /= (8.0 * dld::gamma(s) * logeps);
  complex sum = summand(s, 0);
  for (int m = 1; m < limit; m++) {
    sum += 2.0 * summand(s, m);
  }
  return ret * sum;
}

complex zodd_smart(const complex & s) {
  int limit = static_cast<int>(std::ceil(std::abs(s.imag())));
  if (limit < 20) { limit  = 20; }
  return zodd(s, 2*limit);
}
```

A.3. **Making the grid.** I use two versions of this, depending on whether I'm making an argument plot or a contour plot. The commented out lines indicate the other version.

```cpp
#include "fibo.hpp"
#include <iostream>
#include <fstream>
#include <complex>
```

```cpp
#include <thread>
#include <vector>
#include <string>

void write_grid_chunk(int start_row, int end_row, int grid_size,
      double min_xval, double max_xval,
      double min_yval, double max_yval,
      const std::string& chunk_filename) {
  std::ofstream file(chunk_filename);

  // file << "real,imag,arg\n";
  file << "real,imag,rpart,ipart\n";
  // Iterate over the assigned grid chunk
  for (int i = start_row; i < end_row; ++i) {
    for (int j = 0; j < grid_size; ++j) {
      double real_part = min_xval + (max_xval - min_xval) * i / (grid_size - 1);
      double imag_part = min_yval + (max_yval - min_yval) * j / (grid_size - 1);
      std::complex<double> s(real_part, imag_part);
      std::complex<double> result = zodd_smart(s);
      // double arg_normalized = (std::arg(result) + PI) / (2 * PI);
      // file << real_part << "," << imag_part << "," << arg << "\n";
      double rpart = std::real(result);
      double ipart = std::imag(result);
      file << real_part << "," << imag_part << "," << rpart << "," << ipart << "\n";
    }
  }
}

// Function to combine all chunk files into the final output file
void combine_files(const std::string& output_filename, int num_chunks) {
      std::ofstream outfile(output_filename);
      std::string chunk_filename;
  // Write the header
  //outfile << "real,imag,arg\n";
  outfile << "real,imag,rpart,ipart\n";

  // Combine the chunk files
  for (int i = 0; i < num_chunks; ++i) {
    chunk_filename = "args_chunk_" + std::to_string(i) + ".csv";
    std::ifstream infile(chunk_filename);
    std::string line;
    // Skip the header line of the chunk file
    std::getline(infile, line);
    // Copy the rest of the chunk file to the final output file
    while (std::getline(infile, line)) {
      outfile << line << "\n";
    }
    infile.close();
  }
  outfile.close();
}

```

```cpp
void save_arggrid_to_file(int grid_size,
      double min_xval, double max_xval,
      double min_yval, double max_yval,
      const std::string& filename) {
  int num_threads = 6;
  // Divide the grid rows by the number of threads
  int chunk_size = grid_size / num_threads;
  std::vector<std::thread> threads;
  for (int t = 0; t < num_threads; ++t) {
    int start_row = t * chunk_size;
    int end_row = (t == num_threads - 1) ? grid_size : (t + 1) * chunk_size;
    std::string chunk_filename = "args_chunk_" + std::to_string(t) + ".csv";

    // Start a new thread to write a portion of the grid
    threads.push_back(
      std::thread(write_grid_chunk, start_row, end_row,
      grid_size, min_xval, max_xval, min_yval, max_yval,
      chunk_filename)
    );
  }

  for (auto& t : threads) {
    t.join();
  }

  combine_files(filename, num_threads);

  // Clean up chunk files
  for (int i = 0; i < num_threads; ++i) {
    std::string chunk_filename = "args_chunk_" + std::to_string(i) + ".csv";
    std::remove(chunk_filename.c_str());
  }

  std::cout << "Data saved to " << filename << std::endl;
}

int main() {
  int grid_size = 400;
  double min_xval = -35.0;
  double max_xval = 5.0;
  double min_yval = -5.0;
  double max_yval = 35.0;
  std::string filename = "sizes.csv";
  save_arggrid_to_file(grid_size, min_xval, max_xval, min_yval, max_yval, filename);
  return 0;
}
```

A.4. **Plotting Code.**

```python
import numpy as np
import matplotlib.pyplot as plt
import csv

```

```
def plot_contours(filename):
    x_vals = []
    y_vals = []
    r_vals = []
    i_vals = []
    with open(filename, 'r') as csvfile:
        reader = csv.DictReader(csvfile)
        for row in reader:
            x_vals.append(float(row['real']))
            y_vals.append(float(row['imag']))
            r_vals.append(float(row['rpart']))
            i_vals.append(float(row['ipart']))
    # Convert to numpy arrays
    x_vals = np.array(x_vals)
    y_vals = np.array(y_vals)
    r_vals = np.array(r_vals)
    i_vals = np.array(i_vals)

    # Reshape arg_vals into grid
    grid_size = int(len(r_vals)**.5 + 0.5)
    r_grid = r_vals.reshape((grid_size, grid_size), order='F')
    i_grid = i_vals.reshape((grid_size, grid_size), order='F')

    plt.figure(figsize=[10, 10])
    plt.contour(i_grid, colors="#f9ae54",
                extent=[x_vals.min(), x_vals.max(), y_vals.min(), y_vals.max()],
                origin="lower")
    plt.contour(r_grid, colors="#0482d7",
                extent=[x_vals.min(), x_vals.max(), y_vals.min(), y_vals.max()],
                origin="lower")
    plt.tight_layout()
    plt.savefig("plot.png")

plot_contours('sizes.csv')
```

References

[AKLDW24a] Eran Assaf, Chan Ieong Kuan, David Lowry-Duda, and Alexander Walker. The Fibonacci zeta function and continuation, 2024. Forthcoming. (Cited on page 1)

[AKLDW24b] Eran Assaf, Chan Ieong Kuan, David Lowry-Duda, and Alexander Walker. The Fibonacci zeta function and modular forms, 2024. Forthcoming. (Cited on page 1)

[Bes54] Abram Samoilovitch Besicovitch. *Almost periodic functions*. Dover New York, 1954. (Cited on page 6)

[Joh17] F. Johansson. Arb: efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66:1281–1292, 2017. (Cited on page 2)

[LD20a] David Lowry-Duda. phase_mag_plot. https://github.com/davidlowryduda/phase_mag_plot/, September 2020. [Online; Reference version at https://doi.org/10.5281/zenodo.4035117]. (Cited on page 2)

[LD20b] David Lowry-Duda. The Fibonacci zeta function and modular forms. Talk at Darmouth, available at https://davidlowryduda.com/notes-from-a-talk-at-dartmouth-on-the-fibonacci-zeta-function/, May 2020. (Cited on page 1)

[LD21] David Lowry-Duda. Visualizing modular forms. Talk at Oregon, available at https://davidlowryduda.com/wp-content/uploads/2021/05/visualizing_modular_forms-compressed.pdf, May 2021. (Cited on page 3)

[Sag20] Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.8)*, 2020. https://www.sagemath.org. (Cited on page 2)