Zeros of the Riemann zeta function

Computation and statistical analysis of the first million zeros of the Riemann zeta function in relation to Montgomery's pair correlation conjecture.

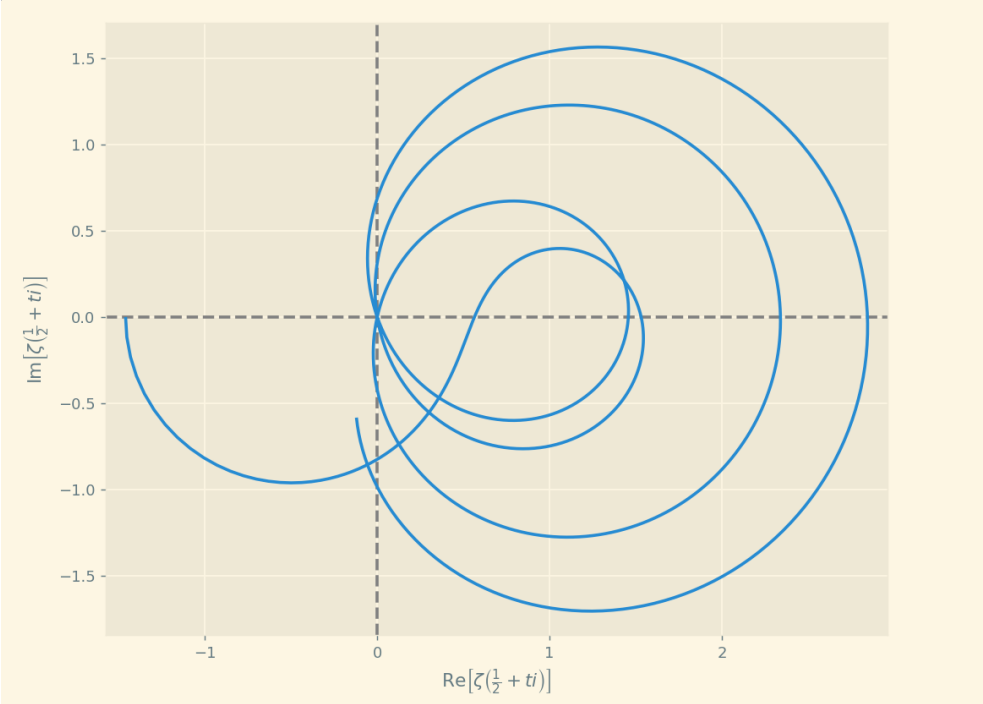


Figure 1: The curve of $\zeta(\frac{1}{2}+ti)$ in $\mathbb{C}$ for $t \in [0, 30]$

By Oisín Davey & Tim Daly

Created as a submission for Maynooth University's MP354 module

15^{th} of May, 2024

All figures used in this document which do not carry citations were created by the authors. For a selection of figures, the code used to produce them is included in the appendix. We have not availed in any way of large language models in either the production of this document, or of any code. Given that all of our references predate 2011, we are confident that none of this document has any such influence. All code used in this project has been written by the authors in Python 3.6.9. All datasets analysed in this project were produced by the authors, although we have viewed public data to corroborate some of our findings. This document was written collaboratively in the overleaf editor. This document is written in such a way as to be approachable to anybody with a moderate understanding of complex analysis, quantum mechanics and computation, and no prior knowledge of the Riemann zeta function has been presumed. Where appropriate, citations have been added.

We've written the python code with a few aesthetic choices in mind. Namely, we've refrained from using any camel-case naming scheme, and have instead opted for variables & functions to both be lowercase and underscore-separated. To indicate float/int type distinctions to the reader and the python interpreter, we've chosen to suffix decimal points to our floating point numerals. The matplotlib style "lightSolarized2" was used throughout. In some cases, we have opted to use Greek UTF-8 characters in the code also, for readability.

Contents

1 The Riemann Zeta function

The Clay mathematics institute has pledged to give one million dollars to anybody who can resolve any of a selection of unsolved problems in maths. Since the original proposal at the turn of the third millennium, only one such problem has been solved. One of these so-called Millennium prizes is offered for the resolution of the infamous **Riemann Hypothesis**.

Central to the Riemann Hypothesis (henceforth RH), is the titular Riemann-Zeta function, hereafter denoted $\zeta(x)$. Firstly, ζ is defined on the complex plane in terms of the following series:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

Under scrutiny, it becomes apparent that, as written, ζ is not defined on the entire real line. For a simple example, consider the attempt to compute $\zeta(-1)$. As defined above, $\zeta(-1) = \sum_{n=1}^{\infty} \frac{1}{n^{-1}} = \sum_{n=1}^{\infty} n = 1 + 2 + 3 + \dots$. This series converges neither in $\mathbb{R}$, nor in $\mathbb{C}$. For a subtler example, one might consider $\zeta(1) = \sum_{n=1}^{\infty} \frac{1}{n}$. This expression is the harmonic series, which also diverges.

Fortunately, ζ does converge for some values. Considering $\zeta(2)$, one obtains the series $\frac{1}{1} + \frac{1}{4} + \frac{1}{9} + \dots$. This series, known as the Basel series, can be shown to converge to $\frac{\pi^2}{6}$. More generally, without alteration of the above definition, $\zeta(s)$ converges precisely when $\mathrm{Re}(s) > 1$. In other words, ζ converges in the open half-plane $\{a + ib : a > 1\}$. Henceforth this will be referred to as $H_{1/2}$.

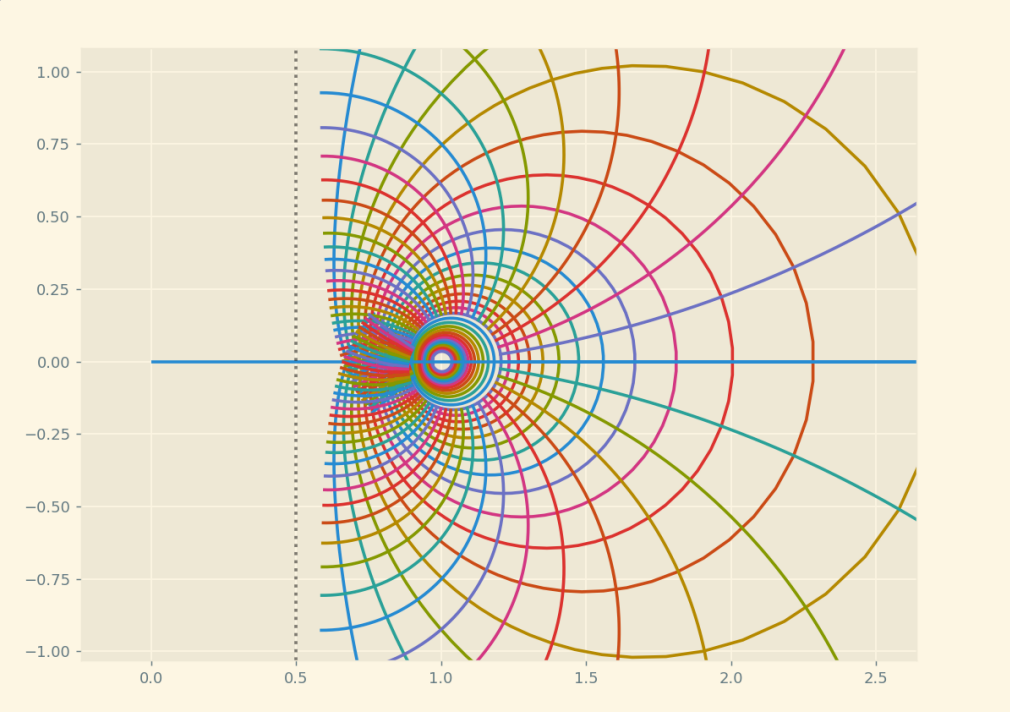


Figure 2: The image of ζ as restricted to the half plane $H_{1/2}$.

In figure 2, the plotted points are of the form $\zeta(s)$, where $s \in H_{1/2}$ is a point on the grid with $\frac{1}{8}$-th spacing. This plot is demonstrative of the continuity of ζ on its domain of definition, as all of the grid lines get mapped to smooth curves. In fact, it turns out that ζ is holomorphic on its domain of definition. Also plotted, rather suggestively, is the vertical line $\{\frac{1}{2} + it : t \in \mathbb{R}\}$. Henceforth this line will be referred to as the **critical line**.

The smooth lines in the image of the open half plane appear to halt in an unsatisfying way as they approach the critical line.[1] Fortunately, owing to its holomorphicity, ζ is open to **analytic continuation**. Through analytic continuation, the domain of definition for ζ may be extended by using its derivatives. In essence, this continues those lines above in the only way which maintains holomorphicity. Incredibly, after applying this continuation, ζ is defined and holomorphic on all of $\mathbb{C}\backslash\{1\}$. From now on, ζ will refer to this continuation.

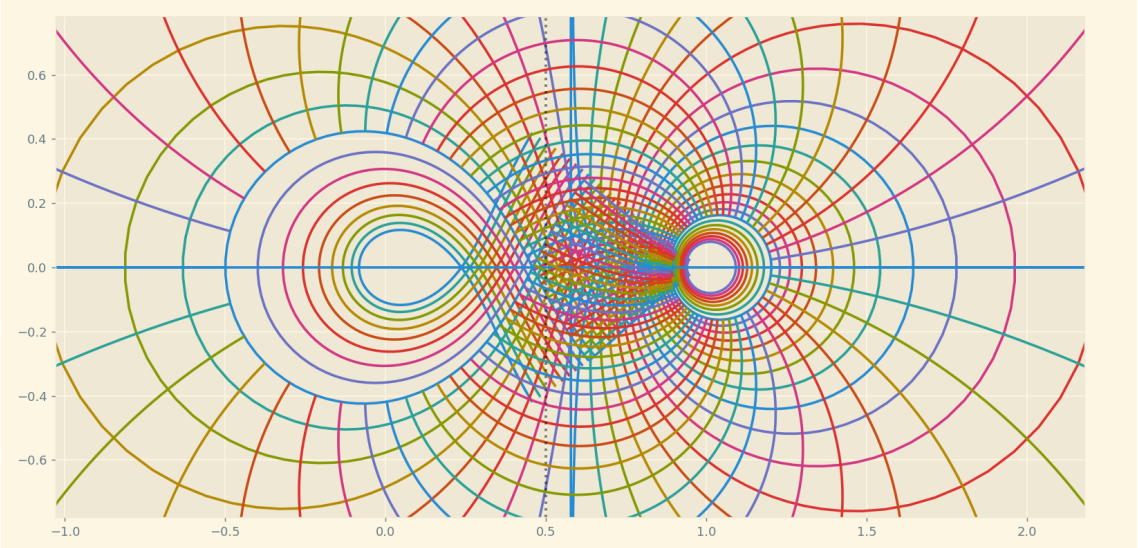


Figure 3: Analytic continuation of figure 2.

RH is a conjecture which states that all non-trivial zeros of ζ lie on the critical line. The Riemann-zeta function satisfies the following functional equation,

$$\zeta(s) = 2^s \pi^{s-1} \sin\left(\frac{s\pi}{2}\right) \Gamma(1-s)\zeta(1-s)$$

Now observe that if s is a negative, even integer, then $\sin\left(\frac{s\pi}{2}\right) = 0 \implies \zeta(s) = 0$. That is to say, $\forall n \in \mathbb{N}, \zeta(-2n) = 0$. These roots are very well behaved, and are so-called "trivial zeros"[2]. All other known zeros lie on the critical line, on which the behavior of ζ is more erratic. For example, the first three roots are given by $0.5 + i14.1347\ldots$, $0.5 + i21.0220\ldots$, and $0.5 + i25.0108\ldots$.

RH can be resolved either by proving that all non-trivial zeros do lie on the critical line,

[1] These lines don't halt at the line $\mathrm{Re}(z) = 0.5$, but rather at the line $\mathrm{Re}(z) \approx 0.57721\ldots$, which is the Euler-Mascheroni constant γ.

[2] This is a slight simplification, as it would imply the same holds for positive even numbers. Essentially Γ has singularities in such a way as to "cancel out" these hypothetical zeros.

or simply by finding a non-trivial zero which doesn't lie on the critical line. One approach to the former resolution is through the Hilbert-Pólya conjecture, which will be given further discussion later. In essence, the conjecture is that the non-trivial zeros correspond to eigenvalues of a certain hermitian operator. The Hilbert-Pólya conjecture implies RH, and is therefore of great importance.

The third conjecture of many in this report, The Montgomery pair correlation conjecture (henceforth PCC) is strongly related with the Hilbert-Pólya conjecture. PCC states that the pair correlation function of the non-trivial zeros is the same as the pair correlation function of random Hermitian matrices in the Gaussian unitary ensemble. This is given detailed discussion in (4).

2 Finding the first million zeros

In this section, having contextualised the problem, the moniker of "non-trivial" will be implicit when referring to zeros of ζ. Owing to the symmetry about the real-axis, the negative portion of the critical line can be ignored in this analysis. Because ζ is holomorphic everywhere except at 1, it is, by definition, meromorphic. A result from complex analysis then implies that ζ is either identically zero, or all of its zeros are isolated. Therefore, for a given zero on the critical line, there must be some neighbourhood around it with no other zeros. So all zeros are discrete and one may meaningfully discuss the "first n zeros" of ζ.

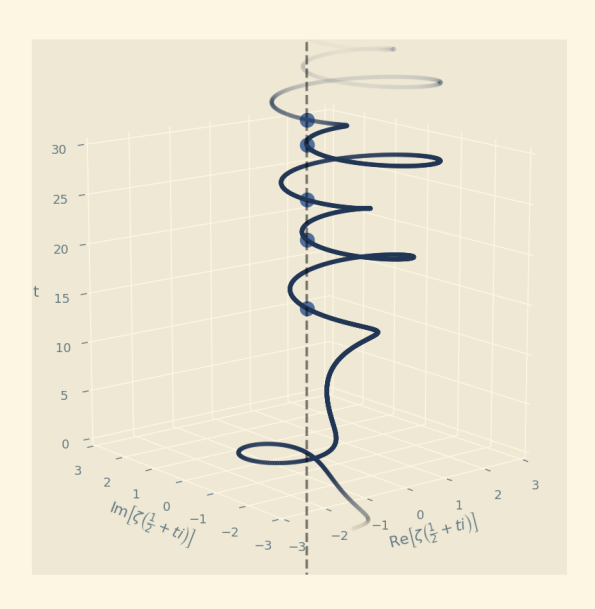


Figure 4: The path of $\zeta(\frac{1}{2}+ti)$ in $\mathbb{C}$ as t increases, displaying the first 5 non-trivial zeros.

2.1 Brief history of computing zeros

We would be remiss not to specifically acknowledge the field-defining work done in this area by Andrew Odlyzko. At present, Odlyzko has computed the first 10^{22}[Odl91] zeros on the critical line, and has computed many of the zeros around zero 10^{23}. In the project brief given, it was suggested that $10,000$ zeros were to be found. In light of Odlyzko's work, $10,000$ seemed pitiful, and so a exploration of his account[OS88] of ζ computation.

Ultimately, before analytic continuation, ζ is computed through an infinite series $\zeta(s) = \sum_{n=1}^{\infty} n^{-s}$. This series is open to the Euler-Maclaurin summation method, because the summand n^{-s} has an immediate antiderivative, namely $\frac{n^{1-s}}{s-1}$. By choosing $n, m \in \mathbb{N}$ to be sufficiently large, one obtains the following expression:

$$\zeta(s) = \sum_{j=1}^{n-1} j^{-s} + \frac{1}{2}n^{-s} + \frac{n^{1-s}}{s-1} + \sum_{k=1}^{m}\left[\frac{B_{2k}}{(2k)!}n^{1-s-2k}\prod_{j=0}^{2k-2}(s+j)\right] + E_{m,n}(s).$$

Here, B_i refers to the i-th Bernoulli number, and $E_{n,m}(s)$ is the error term. This error bound has been shown to satisfy the inequality:

$$|E_{m,n}(s)| < \left|\frac{s+2m+1}{\mathrm{Re}(s)+2m+1}\frac{B_{2m+2}}{(2m+2)!}n^{-1-s-2m}\prod_{j=0}^{2k}(s+j)\right|.$$

The specifics of the Euler-Maclaurin method implies that these formulae hold on the analytic continuation of ζ also.

Up until 1930, all computations of zeta zeros were performed using this method. This changed around 1932, when Carl Ludwig Siegel discovered an integral equation in Riemann's unpublished papers[Sie32]. In particular, he found the following expression:

$$\zeta(s) = \sum_{n=1}^{N}\frac{1}{n^s} + \pi^{s-\frac{1}{2}}\frac{\Gamma\left(\frac{1-s}{2}\right)}{\Gamma\left(\frac{s}{2}\right)}\sum_{n=1}^{M}\frac{1}{n^{1-s}} + R(s).$$

Where N and M are any natural numbers, and $R(s)$ is a certain contour integral whose integrand depends on N, and whose path depends on M. Specifically, the integral is the following:

$$R(s) = \frac{-\Gamma(1-s)}{2\pi i}\int_{C(M)}\frac{(-x)^{s-1}e^{-Nx}}{e^x-1}\,dx$$

The integrand has singularities at all points $\{\ldots, -4\pi i, -2\pi i, 0, +2\pi i, +4\pi i, \ldots\}$. The contour $C(M)$ is path which comes from $+\infty$, encircles the origin and goes back to $+\infty$. It also encircles precisely those singularities $2\pi k$, where $k \in \{-M, \ldots, +M\}$. This formula can be derived without much ado from a standard ζ contour, using the functional equation, and then applying the Residue theorem to the singularities of the form $2\pi k$.

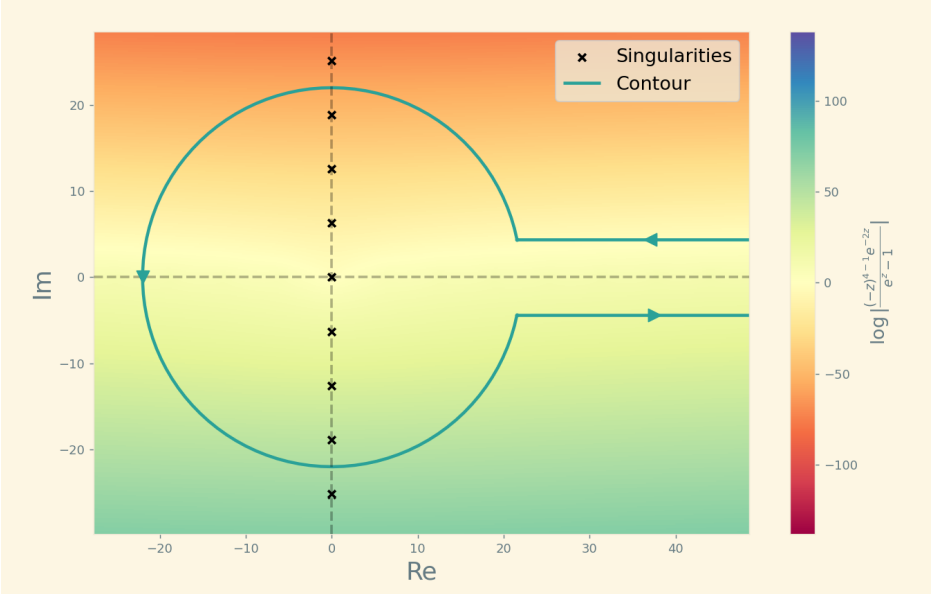


Figure 5: The contour $C(M)$ for $M = 3$, superposed onto a plot of the modulus of integrand for $N = 2, s = 4$.

Siegel used a saddle point approximation of this contour integral, resulting in an asymptotic formula for the modulus of ζ along the critical line. Specifically, as $t \to \infty$, Siegel's function approaches the function $Z(t) := e^{i\vartheta(t)}\zeta\left(\frac{1}{2} + it\right)$, where $\vartheta(t)$ is a phase parameter which cancels the phase of ζ in such a way that $Z(t)$ is a real-differentiable real-valued function of t.[3] This function is hereafter referred to as the Hardy Z function, named for G. H. Hardy. Critical line zeros and zeros of $Z(t)$ immediately have a one-to-one correspondence.

Siegel's approximation yields:

$$Z(t) = 2\sum_{n=1}^{N(t)} \frac{\cos\left[\vartheta(t) - t\log n\right]}{\sqrt{n}} + \frac{(-1)^{N(t)+1}}{(t/2\pi)^{1/4}}\left[\Psi(\varrho(t)) + \frac{1}{12\pi^2\sqrt{t/2\pi}}\Psi^{(3)}(\varrho(t)) + \dots\right].$$

$$\text{Where } \Psi(\varrho) := \frac{\cos\left[\frac{\pi}{2}(\varrho^2 + 3/4)\right]}{\cos(\pi Z)},\ \vartheta(t) := \operatorname{Im}\log\Gamma\left(\frac{1}{4} + i\frac{t}{2}\right) - \frac{t}{2}\log\pi, \text{ and}$$

$$N(t) = \left\lfloor\sqrt{t/2\pi}\right\rfloor,\ \varrho(t) = 1 - 2\left(\sqrt{t/2\pi} - N(t)\right).$$

Today, nearly every attempt to compute the critical line zeros of ζ makes use of Siegel's approximation. The efficiency bottleneck for computation of $Z(t)$ is the calculation of the term $\sum_{n=1}^{N(t)} \frac{\cos[\vartheta(t)-t\log n]}{\sqrt{n}}$. This term makes $\mathcal{O}(\sqrt{t})$ trigonometric calculations, and is computed for each evaluation of $Z(t)$. Algorithmic complexity analysis of this may be found in appendix C(6.3), but sufficed to say that this term is in desperate need of a speed-up.

[3] In truth, this is not really taking the modulus of ζ, as that would produce cusps at all zeros, and would therefore not be differentiable. $Z(t)$ avoids this because it can be negative.

By 1988, Andrew Odlyzko and Arnold Schönhage had produced the eponymous Odlyzko-Schönhage algorithm[OS88]. Their insight was to make use of a Fast-Fourier-Transform (FFT) to reduce the problematic sum term into a shorter sum of rational functions, for which they had already established a speedy algorithm to compute. Although Hardy's Z function was the subject of their paper, their technique generalized to several classes of series, most notably Dirichlet series. Seemingly, all attempts in the literature to produce efficient computations of $Z(t)$ rely on discrete fourier analysis of the problematic sum in Siegel's approximation. See appendix C(6.3) for our attempts in this respect.

2.2 Gram points & Rosser's rule

In search of the first n zeros, one needs to be sure that no zeros have been missed. The standard root-finding techniques make no such assurances. It is not sufficient to find any n zeros, as the statistics of nearby zeros is central to the discussion. It was suggested in our brief that, in order to carry this task out, a chosen interval should be subdivided many times over until no more roots can be detected through bracketing. This seemed unsatisfactory, as it gives only tenuous assurance that all such zeros have been found, and it isn't clear how such a scheme would scale to a larger set of zeros.

On researching zero-finding methods, one quickly comes across the concept of Gram points and Gram's law. In 1903, Gram published a list of the first 15 zeros of ζ[Edw74]. In doing so, he notice that the roots of $Z(t)$ and the roots of $\sin\vartheta(t)$ seemed to alternate. He proposed[Gra03] that it was not unlikely that this pattern should continue, and provided some justification to this effect. The roots of $\sin\vartheta(t)$ are precisely the solutions of the equation $\vartheta(t) = n\pi$ for $n \in \mathbb{N}$. Owing to the monotonicity and unboundedness of $\vartheta(t)$, there is exactly one such solution for each n, and we may therefore speak of t as the n-th **gram point**.

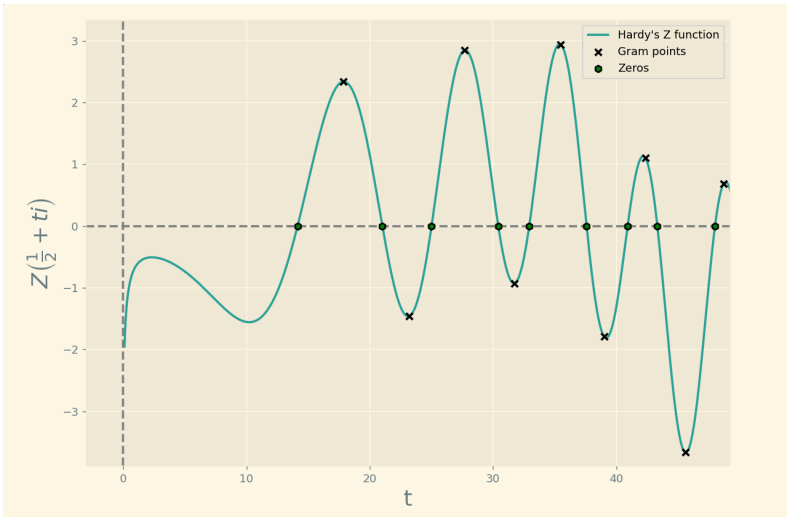


Figure 6: The first few gram points superposed onto $Z(t)$. The gram points appear to align with local extrema of $Z(t)$.

This pattern does not continue forever, unfortunately. Specifically, at the 126-th zero, the pattern breaks. In particular, we have that, between the 125th and 126th gram points, there are no zeros, but between the 126th and 127th gram points, there are two zeros.

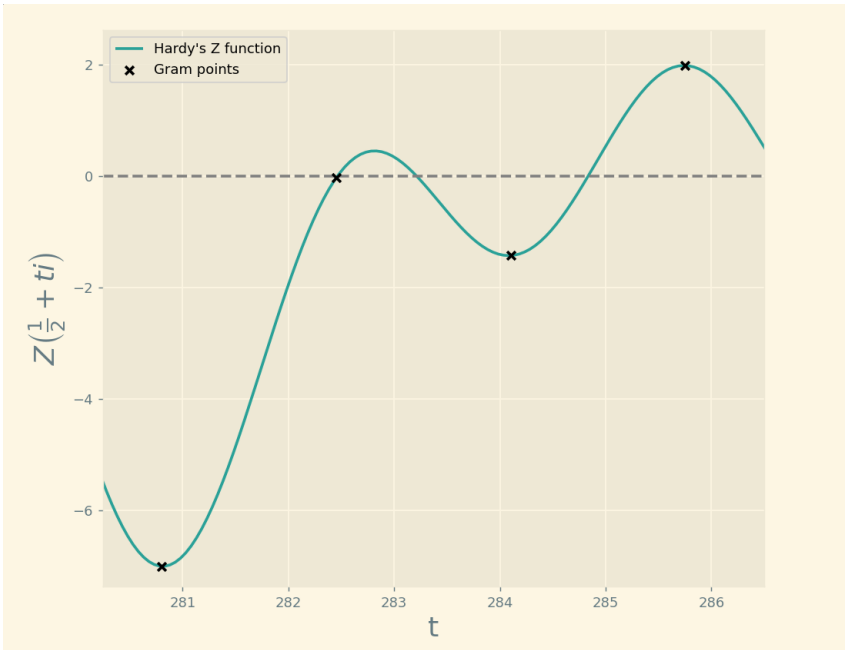


Figure 7: The 125-th to the 128-th gram points.

J. I. Hutchinson, despite personally finding these points of failure, opted to call the the tendency of zeta zeros to alternate with gram points "Gram's law". This is a law in the chemical sense, insofar as it fails to be true infinitely often. In fact, it has been shown to formally *almost never* hold true.[Kor10] There are "weak forms" of Gram's law which haven't been disproven as of writing, but no useful forms of Gram's law have been proven correct so far.

It follows that we cannot outright prove we have found all the zeros we're looking for without more rigorous analysis. We will therefore treat one of the weak forms of Gram's law as a heuristic for zero-hunting. Specifically, we will make use of **Rosser's rule**. To understand Rosser's rule, we must first define some terms.

Denote the n-th Gram point by g_n. A Gram point is called a "good" if $(-1)^n Z(g_n) > 0$. A good Gram point is awarded this epithet because if this were always true, Gram's law would always hold. Let the n-th good Gram point be denoted G_n. We'll call the interval $[g_i, g_{i+1}]$ between two consecutive Gram points a "Gram interval",[4] as we similarly define $[G_n, G_{n+1}]$ to be a "good Gram interval". Each good Gram interval contains a certain amount of standard Gram intervals. The specific quantity of Gram intervals contained within a good Gram interval is referred to as the "length" of the good Gram interval.

[4]In other treatments, it can matter quite a bit as to whether we discuss $(g_i, g_{i+1}]$, $[g_i, g_{i+1})$, $[g_i, g_{i+1}]$ or (g_i, g_{i+1}). This won't matter for us.

Rosser's rule conjectures that each good Gram interval contains exactly as many zeros as its length[5]. This is a weaker form of Gram's law, as it would be immediately implied by Gram's law. Rosser's rule, as expected, fails infinitely often. However, the first failure of Rosser's rule happens at the $13,999,825$-th Gram point[Kor10], which lies outside of the region we are studying. In the spirit of good sport though, we have made allowances in the code to handle a potential violation of Rosser's rule. This is achieved more or less via the subdivision scheme referenced in the project prompt, but localised to a good Gram interval. In particular, we begin by evenly subdividing the good Gram intervals into $l+1$ regions, where l is the length of the interval. Then, until we have detected l zeros via bracketing, we further subdivide the bracketed regions in half. If a region is subdivided 10 times, the program gives up. We again mention that this never occurs on our interval of study.

All that's left is to compute the Gram points. Fortunately, Edwards[Edw74] provides an excellent asymptotic formula for g_n in the following form.

$$g_n \approx 2\pi \exp\left(1 + W\left(\frac{8n+1}{8e}\right)\right).$$

Above, the function W is the Lambert W function. For our code, because this is not very computationally expensive[6], we used the scipy library's implementation of W.

2.3 Taylor expansion of $\Psi(\varrho)$ & Asymptotic expansion of $\vartheta(t)$

Throughout the script running, several calls will be made to the function $\Psi(\varrho)$ along with it's third derivative $\Psi(\varrho)$. As a reminder:

$$\Psi(\varrho) := \frac{\cos\left[\frac{\pi}{2}(\varrho^2 + 3/4)\right]}{\cos(\pi Z)}.$$

It is not difficult, (although tedious) to get a closed form expression for the third derivative. By defining the terms $\mathrm{spr} := \sec(\pi\varrho)$, $\mathrm{tpr} := \tan(\pi\varrho)$, $\mathrm{chr} := \cos\left[\frac{\pi}{2}(\varrho^2 + \frac{3}{4})\right]$, and lastly $\mathrm{shr} := \sin\left[\frac{\pi}{2}(\varrho^2 + \frac{3}{4})\right]$, we can write:

$$\Psi^{(3)}(\varrho) = \pi^2 \mathrm{spr}\left[\pi \mathrm{tpr}\cdot\mathrm{chr}(5\mathrm{spr}^2 + \mathrm{tpr}^2) - 3\pi\varrho\,\mathrm{shr}(\mathrm{spr}^2 + \mathrm{tpr}^2) - 3\mathrm{tpr}(\mathrm{shr} + \pi\varrho^2\mathrm{cr}) + \pi\varrho^3\mathrm{shr} - 3z\mathrm{chr}\right].$$

Trigonometric functions are computationally expensive, and these functions will be called roughly $\mathcal{O}(M\sqrt{M})$(see 6.1) times, where we want M zeros. There is room for optimisation here, because of the very small domain of $\Psi(\varrho)$. By definition, $\varrho(t) = 1 - 2\{\sqrt{t/2\pi}\}$, where $\{\}$ denotes the fractional part of a number. It follows that, $\forall t > 0$, $\varrho(t) \in (-1, 1]$. It may not be obvious from its definition, but $\Psi(\varrho)$ has no (non-removable) singularities. This is because the roots of the denominator align with, and have the same order as, the roots of the numerator.

[5] In most of the literature, Rosser's rule will actually be the statement that a "Gram block" will have as many zeros as its length, where a "Gram block" is a good Gram interval which contains no other good Gram intervals. These two statements are equivalent, and we believe the above is easier to work with.

[6] Insofar as we only need to compute a million Gram points, which is much smaller than our speed bottleneck.

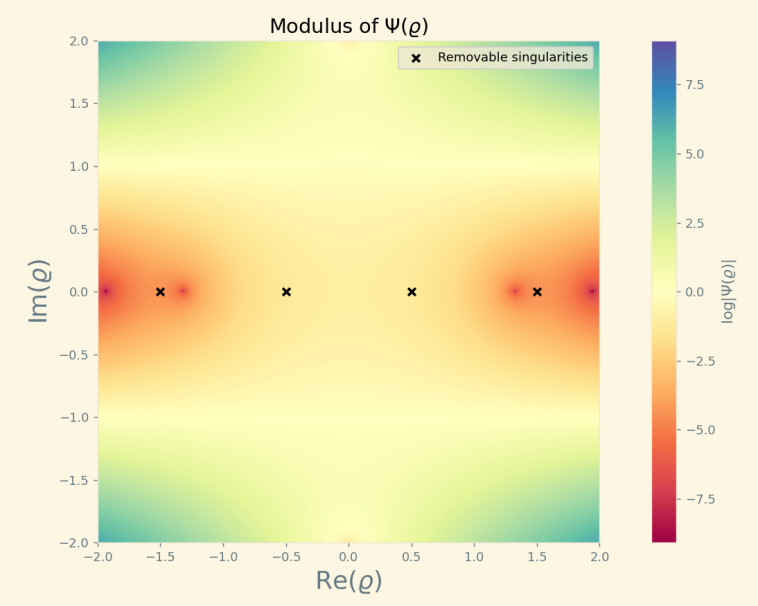


Figure 8: The modulus of $\Psi(\varrho)$ over the region of interest. The function is well-behaved around its removable singularities.

Therefore, to cut down on computation expense, we've opted to use a Taylor expansion of $\Psi(\varrho)$ about 0. The function then only needs to evaluate polynomials, and finding the third derivative of $\Psi(\varrho)$ comes immediately. Somewhat arbitrarily, we choose to do this expansion up to fifteen terms, which seemed to be sufficient, although we did not analyse the error properly.

Moreso than $\psi(\varrho)$, the function $\vartheta(t)$ is called several times throughout the code. Recall that $\vartheta(t)$ is defined to be $\operatorname{Im}\log\Gamma\left(\frac{1}{4}+i\frac{t}{2}\right)-\frac{t}{2}\log\pi$. The details are a bit tedious, and can be found in pages $106-120$ of [Edw74], but in summary, through Sterling's approximation we have, *mutatis mutandis*, a wonderful asymptotic expansion for $\vartheta(t)$ given by:

$$\vartheta(t)\sim\frac{t}{2}\log\frac{t}{2\pi}-\frac{\pi}{8}-\frac{t}{2}+\frac{1}{48t}.$$

While more terms *could* be included, the series doesn't actually converge, which is to be expected from an asymptotic expansion, so this will suffice.

2.4 Pegasus algorithm

Following the subdivision scheme described earlier, one must perform straightforward root-finding to locate the zeros. Fortunately, Hardy's Z function is incredibly well behaved locally, so almost all root-finding methods are open to use. To be transparent, our million zeros were initially computed simply by using binary search[7]. This was done for safety more than anything, as we could be certain it would produce correct answers. We were tentative about using, say, Newton Raphson, as we didn't want to run the million zeros calculation more than once.

[7]Bisection, in MP354.

After having run the major calculation though, we opted to switch to the so-called Illinois algorithm. This is an improvement on regula falsi.[8] Suppose we have a function $f(x)$. Suppose a root of f is bracketed by x_a and x_b. In computation of a root, we concern ourselves with two pairs (x_a, f_a) and (x_b, f_b), where typically $f_a = f(x_a)$ and $f_b = f(x_b)$.

Regula falsi proceeds by considering a new point $x_c := \frac{x_a f_b - x_b f_a}{f_b - f_a}$, and assigning $f_c := f(x_c)$. Then, if the root is bracketed by x_a and x_c, we replace the pair (x_b, f_b) with (x_c, f_c). Likewise, if x_c and x_b brackets the root, we replace the pair (x_a, f_a) with (x_c, f_c). After a while, regula falsi approaches a somewhat static configuration wherein one pair is never again updated, while the other pair approaches the root. The idea of the Illinois algorithm is to give some impetus to the static root to increase the rate of convergence.

The trick is to recall the sign of the most recently chosen false position. Then, if the new false position has the same sign as the previous, we detect the aforementioned staticity. To counteract this, we take the end-point of opposing sign to the false position, say x_s, and replace f_s with $\frac{1}{2} f_s$. This will encourage the next false position to be of opposing sign.

```
def func(x):
    return x**2-x-1

(a, fa) = (0, func(0))
(b, fb) = (2, func(2))

ϵ = 0.000001

while abs(b-a) > ϵ:
    c = (a*fb-b*fa)/(fb-fa)
    fc = func(c)
    (a, fa) = (a, 0.5*fa) if fb*fc > 0 else (b, fb)
    (b, fb) = (c, fc)

print(a)
```

Listing 1: Illinois algorithm example

While researching algorithmic complexity, we happened across a variant of the illinois algorithm called the pegasus algorithm[M D72]. It operates in much the same way, except it reduces the size of fa in the above code not by half, but rather by the term $fb/(fb+fc)$. Detailed analysis can be found in the references, but the brief version is that pegasus is a little faster[9], but sounds a lot cooler, so we opted for it.

[8] Method of false position, in MP354.

[9] A potential downside is a chance for division by zero. However, this is an exceedingly unlikely prospect for us here.

```
def func(x):
    return x**2-x-1

(a, fa) = (0, func(0))
(b, fb) = (2, func(2))

ϵ = 0.000001

while abs(b-a) > ϵ:
    c = (a*fb-b*fa)/(fb-fa)
    fc = func(c)
    (a, fa) = (a, fa*fb/(fb+fc)) if fb*fc > 0 else (b, fb)
    (b, fb) = (c, fc)

print(a)
```

Listing 2: Pegasus algorithm example

2.5 Multiprocessing

An interesting property of the Rosser's rule approach is the possibility for parallelisation[10]. By this we mean that when computing the roots within a good Gram interval, we at no point rely on results from computations of any other blocks. Hence, with multiple computers, it would be possible to process multiple blocks simultaneously, or, as it were, "in parallel".

While we don't have access to a network of computers, we can avail of the multiple cores in a given computer. The computation was performed on a 4-core laptop with intel i5 processors. These processors have hyper-threading[11] enabled, so we have 8 effective threads of computation.

We therefore chose to use the python multithreading module on our Rosser's rule computations. Analysis of speedups from multithreading is a bit of an art, so we will gesture to the empirical improvements in performance. Without multithreading, our program took around 23 seconds to compute $10,000$ zeros. With multithreading, our program took only 10 seconds to do the same.[12] While multithreading does make our computations quicker, it does not actually effect the computational time complexity, because it can only speed up performance to a scalar multiple. This fact notwithstanding, it's better to take 97 minutes to compute a million zeros than 3 hours.

[10] The technical term for the exact situation we have is "embarrassingly parallelisable".
[11] Each core can perform two jobs simultaneously.
[12] These computations were done with all other optimisations made. Without making the improvements above, the code takes several minutes to find $10,000$ zeros.

2.6 Summary

With all of the above improvements on efficiency, our program computes [13] 10,000 zeros in 10 seconds, 100,000 zeros in 4 minutes and 34 seconds, and 1,000,000 zeros[14] in 1 hour and 37 minutes. In terms of efficiency, we are reasonably pleased with these results. We suggest a complete study of the Odlyzko-Schönhage algorithm for further speed-ups, as there are an array of fourier approaches which are applicable to this problem. The code would also port to C++ very easily, which would improve the speed at least tenfold[15].

For the analysis we are doing, high-precision isn't incredibly important. In fact, from our sojourn in the literature, high-precision isn't very well studied in this matter, mainly because the statistics of the distribution is more important. Nevertheless, we should reflect on the precision of our results. We will treat Odlyzko's dataset as the standard here. The asymptotic functions mean that, for the first hundred zeros, we aren't very precise. Specifically, we are accurate to two decimal places from the first zero, up until height "77.144...", whereafter we are accurate to three decimal places. Then it isn't until height "350.408..." where our program is consistently correct to 4 decimal places. Because we've set out root-finding algorithms to have an input precision of 0.00001, this level of precision is as good as it gets hereafter.

It may be interesting to see if higher precision could be obtained. One thing that could be straightforward to implement is to use more and more asymptotic terms for smaller heights. This would involve analysis of error estimates, the theory of which is laid out in great detail in [Edw74].

3 Hilbert-Pólya conjecture

The basis for the Hilbert-Pólya conjecture first arose when George Pólya (Pólya György) was asked for a physical reason why the RH should be true, and he suggested that it would be the case if the imaginary parts t of the zeros $\frac{1}{2} + it$ of the Riemann-zeta functions corresponded to eigenvalues of a Hermitian operator. The possible connection of the Hilbert-Pólya operator with quantum mechanics is outlined below.

The Hibert-Pólya operator is of the form $\frac{1}{2} + iH$ where H is the Hamiltonian of a particle of mass m moving in a certain potential $V(x)$. Regarding H as such a Hamiltonian means we make the following assertions,

(i) $\hat{H}$ has a classical limit.

(ii) The classical orbits are all chaotic.

(iii) The classical orbits don't possess time-reversal symmetry.

[13] With a 4 core intel i5 laptop.

[14] In case you don't want to devote 90 minutes of computing time, our results are in this file https://blog.xn–oisn-xpa.ie/zeros/1000000.out. If you don't feel comfortable downloading our file, the millionth zero is found at height "600269.6770..."

[15] This is, naturally, speculation. Anecdotally however, a more than tenfold speedup from python for C++ in applications of this kind is the accepted figure in the community.

3.1 Berry Keating Conjecture

Berry and Keating built on the Hilbert-Pólya conjecture have speculated in their paper [M V99] that one possible operator that satisfies this conjecture might be some quantization of the simple classical Hamiltonian $H_{CL}(X, P)$ would be a function of a single coordinate and its conjugate momentum,

$$H_{cl}(XP) = XP$$

The reason for the association of this Hamiltonian and the Riemann-zeta function is not readily apparent at this point, but we will endeavour to outline the reasons below.

At the quantum level the simplest Hermitian operator associated with xp is

$$H = \frac{1}{2}(XP + PX) = -i\left(X\frac{d}{dX} + \frac{1}{2}\right)$$

The eigenvectors of this Hamiltonian with eigenvalues E corresponding to functions $\psi(X)$ that satisfy

$$H\psi(X) = E\psi(X)$$

are given by

$$\psi(X) = \frac{A}{X^{1/2-iE}}$$

Where A is just some constant. Here one might notice that XP is a complex version of the harmonic oscillator. We know wish to find the corresponding eigenfunction in momentum space, making use of the Fourier transform. We insist on a continuation across $X = 0$ and for simplicity we let $X = |X|$

$$\phi(P) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} dX\psi(X)e^{-iPX}$$

$$= \frac{A}{|P|^{1/2+iE}}2^{iE}\frac{\Gamma\left(\frac{1}{4}+\frac{1}{2}iE\right)}{\frac{1}{4}-\frac{1}{2}iE} = \frac{A}{\sqrt{2}\pi|P/2\pi|^{1/2+iE}}e^{2i\vartheta(E)}$$

This yields a physical interpretation for the Riemann-siegel theta function $\vartheta(t)$, which was central to the locating of the zeros. This argument gives a nice physical interpretation of some of the functions pertaining to the Riemann-zeta function.

There are however a few problems with the Hilbert-Pólya and Berry-Keating conjectures that remain unsolved to this day. One of the principle issues with the Berry-Keating conjecture is they are yet to find suitable boundary conditions that convert XP into a well defined hermitian operator with discrete eigenvalues and it is not clear on which space this operator should act on.

4 Gaussian unitary ensemble

The most common random matrix distributions are the Gaussian ensembles, which model the Hamiltonians of random dynamical systems. The one we will focus on is the Gaussian Unitary Ensemble (GUE). The GUE models random Hamiltonians without time reversal symmetry. Since this is the statistical model which is relevant to the Riemann Zeta Function, this is what we are interested in.

A matrix G is called a GUE if there exists a random matrix A with independent and identically distributed random variables, where,

$$A_{nm} = X_{nm} + iY_{nm}$$

Where X_{nm}, Y_{nm} are real numbers taken from a standard normal distribution whose mean is 0 and variance is 1. Our matrix G is thus given by

$$G = \frac{A + A^*}{2}$$

Taking the eigenvalues of some random Hermitian matrices, one would expect that the distribution of these would be similarly random. Recall the definition of the Poisson distribution. Given a series of points distributed randomly along a line, with average density 1, the probability of finding m points in a given interval of length r is

$$\frac{r^m}{m!}e^{-r}$$

One might naïvely expect the eigenvalues of these random matrices to follow a similar distribution, but this is not the case. The eigenvalues of a random Hermitian matrix don't behave like independent variables, and as such the probability of finding more than one eigenvalue in a short interval is much less than the Poisson distribution would suggest. In essence, the eigenvalues repel each other. The distribution of such eigenvalues is derived below.

It is well established in the field of random matrices[Bog07]that the joint distribution of the eigenvalues, E_j, for the GUE has the following form:

$$P(E_1, E_2, \ldots, E_n) \sim \prod_{i<j} |E_i - E_j|^2 \exp\left(-\sum_{k=1}^{N} E_k^2\right)$$

Using this joint distribution of eigenvalues means we can calculate the n-point correlation functions, which we define as the probability density of having n eigenvalues at given positions R_n. For GUE these have determinantal form

$$R_n(E_1, E_2, \ldots, E_n) = \det[K(E_i, E_j)]|_{i,j=1,\ldots,n}$$

Where as we approach the universal limit of mean density, referring to the behaviour of the average density of the eigenvalues as the size N of the matrix tends to towards infinity.

The kernel $K(E_i, E_j)$ is

$$K(E_i, E_j) = \frac{\sin(\pi(E_i - E_j))}{\pi(E_i - E_j)}$$

In particular we are interested in the unfolded[16]two point correlation function, $n = 2$,

$$R_2(\epsilon) = 1 - \left(\frac{\sin(\pi\epsilon)}{\pi\epsilon}\right)^2$$

where $\epsilon = E_1 - E_2$

Montgomery's pair correlation conjecture states that the zeros of the Riemann-zeta function, normalised in such a way that they on average have unit spacing, have the pair correlation function,

$$g(r) = 1 - \left(\frac{\sin(\pi r)}{\pi r}\right)$$

This conjecture gives further support to the idea that the Hilbert-Pólya conjecture offers a viable approach to proving the RH. The PCC has been partially proven by Montgomery in his paper introducing it.[Mon73] As we can see this is the same formula derived above in the two point correlation function for eigenvalues of the GUE. So there is a clear parallel here between random matrices and the zeros of the Riemann-zeta function.

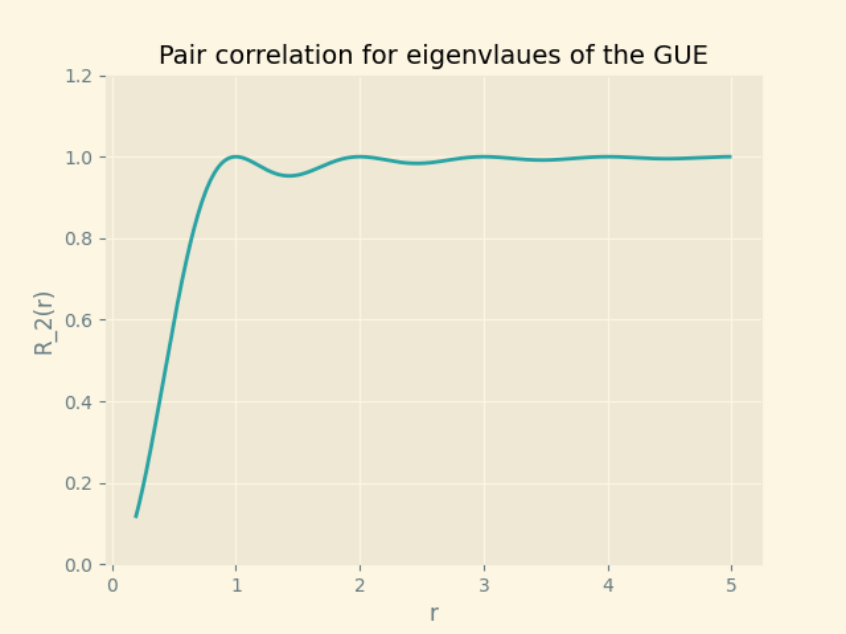


Figure 9: Plot of the pair correlation function for the GUE, using function found above.

[16]Unfolding refers to the normalisation procedure applied to the eigenvalues of the ensemble. The goal of this is to remove the global density fluctuations and rescales the eigenvalues so that the average spacing becomes unitary.

4.1 Computing the pair correlation function for roots of ζ.

We want to compare this result to the actual numerical calculations found. Given that the higher up the critical strip we go the denser the zeros become, and the average vertical spacing between consecutive zeros at some height t is $\frac{2\pi}{\log(\frac{t}{2\pi})}$.[Odl87] Since we want to study the spacings between the zeros, we want to deal with quantities that are mostly invariant with respect to height. To that end we define the formula for the normalised spacing between consecutive non-trivial zeros $\frac{1}{2} + it_n$ and $\frac{1}{2} + it_{n+1}$,

$$\delta_n = \frac{t_{n+1} - t_n}{2\pi} \log\left(\frac{t_n}{2\pi}\right)$$

With this we can reformulate Montgomery's conjecture into something we can apply to the calculated zeros. Expounding on definitions, PCC says that, given $N, M \gg 1$,

$$\frac{1}{M}|\{(n,k) : n \in [N, N+M], , k \geq 0, \delta_n + \cdots + \delta \in [r-\Delta, r+\Delta]\}| \approx \int_{r-\Delta}^{r+\Delta} 1 - \left(\frac{\sin(\pi x)}{\pi x}\right)^2 \, dx.$$

Here, Δ and r can be arbitrary, so long as $r - \Delta > 0$. We then make the following identification, by considering average values of integrands.

$$1 \gg \Delta \implies \frac{1}{2\Delta} \int_{r-\Delta}^{r+\Delta} 1 - \left(\frac{\sin(\pi x)}{\pi x}\right)^2 \, dx \approx 1 - \left(\frac{\sin(\pi r)}{\pi r}\right)^2.$$

We then may rearrange the PCC expression to find something we may computationally test. That will be, for $N, M \gg 1 \gg \Delta$,

$$\frac{1}{2\Delta} \cdot \frac{1}{M}|\{(n,k) : n \in [N, N+M], , k \geq 0, \delta_n + \cdots + \delta \in [r-\Delta, r+\Delta]\}| \approx 1 - \left(\frac{\sin(\pi r)}{\pi r}\right)^2$$

What we're essentially doing here is counting the number of pairs (n, k) where you can fit k zeros near the n-th zero within the interval $[r - \Delta, r + \Delta]$ for some r. Then multiplying this count by a factor of $\frac{1}{2\Delta M}$. Keeping in mind the dataset we've produced, we chose to use $N = 8 \times 10^5, M = 4 \times 10^4, \Delta = 0.095$[17], and we obtained the following plot.

Comparing the numerical calculations to the GUE prediction we can see that it is an excellent prediction. PCC appears to hold fairly well for the first million zeros.

When computing the correlation function from the dataset, we didn't come across any more efficient method other than the naïve method, by which we mean "For each $n \in [N, N+M]$, iterate k from zero and find the minium $k_{\min}$ such that $\delta_n + \cdots + \delta_{n+k_{\min}} \in [r-\Delta, r+\Delta]$, and stop once you've found the maximum such $k_{\max}$." This is not for lack of trying, see appendix D.(6.4) for our best attempt. It turns out that, as defined above, $k_{\min}$ and $k_{\max}$ are always very small, never exceeding 10. So this algorithm is perfectly fine.

[17]Delta would ideally be smaller, but we would need a lot more data to make that happen. Even Odlyzko only used 0.05[Odl87]

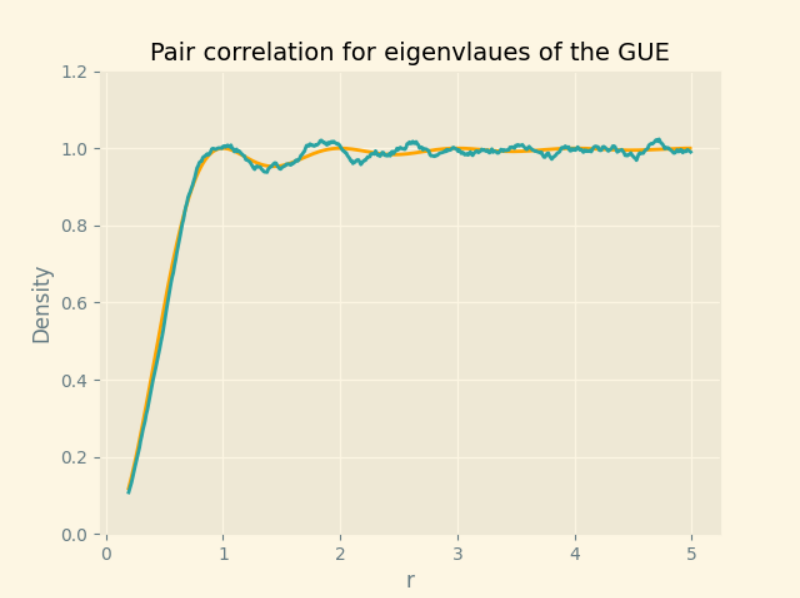


Figure 10: GUE prediction overlaid with the empirical data.

5 Connection with Quantum Chaos

Quantum chaos is a branch of physics which examines how chaotic classical systems can be described in terms of quantum physics. There are many possible approaches to tackling this problem, one option is to develop methods for solving quantum systems where we can't use perturbation theory, another would be using a semi classical approach such as the periodic orbit theory (see Ouroborology in appendix A(6.1)), another option and one that interests us is correlating statistical descriptions of eigenvalues with the classical behaviour of the same Hamiltonian.

5.1 Bohigas-Giannoni-Schmit conjecture

In their 1984 paper[O B84] Bohigas,Giannoni and Schmit introduced the BGS conjecture stating that the Hamiltonian of a classical chaotic system could be modelled by a random matrix from a Gasussian Ensemble.Though still not proven this conjecture has a myriad of numerical findings backing up it's claim. This offers further implications of the connection between the zeros and random matrices, allowing us to consider the extension of this to include quantum chaos.

Spectral statistics within the context of random matrices refers to the statistical properties and distributions of the eigenvalues of random matrices, encompassing variations, densities and correlations. Most of the results and applications are beyond the scope of this paper, but we employ the use of this definition in the following section. It was first proposed in 1973 by Percival[Per73] that spectral statistics could be one way to distinguish the quantum mechanics of classically chaotic system from that of regular classical systems. Various other attempts were made to build on this but to no avail. The *coup de génie* of Bohigas, Giannoni and Schmit was to realise that the difference between the spectral

statistics of classical and chaotic systems was not just that the former could be modeled by a set of independent numbers when the latter showed some correlation but that the spectral statistics of these chaotic systems could be fully described by the random matrix ensembles.

5.2 Wigner Surmise

Wigner's surmise is a formula for the density of level spacings of normalised eigenvalues. The density of distances for an unfolded set of eigenvalues is approximated by[A Y99]

$$p_\beta(s) = a_\beta s^\beta e^{-b_\beta s^2}$$

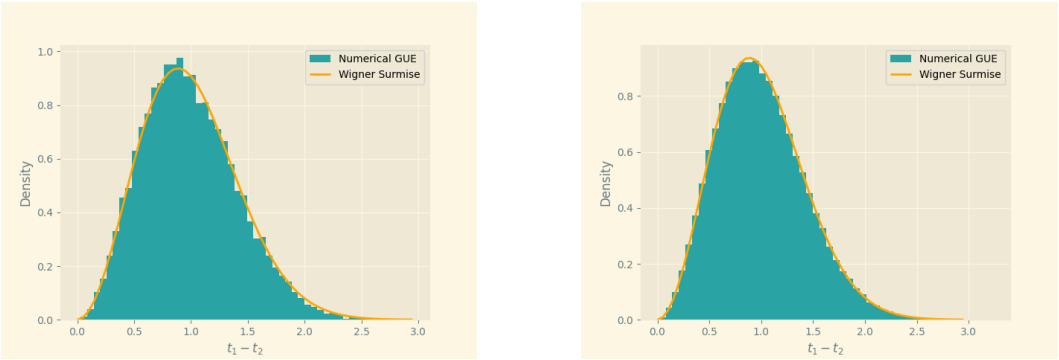
Where

$$a_\beta = 2\frac{\left(\Gamma\left(\frac{\beta+2}{2}\right)\right)^{\beta+1}}{\left(\Gamma\left(\frac{\beta+1}{2}\right)\right)^{\beta+2}} \qquad b_\beta = \frac{\left(\Gamma\left(\frac{\beta+2}{2}\right)\right)^2}{\left(\Gamma\left(\frac{\beta+1}{2}\right)\right)^2}$$

Here β is the Dyson index which measures the level of repulsion of the eigenvalues. For the Gaussian ensembles we have a Dyson index of $1, 2, 4$ for GOE, GUE and GSE respectively. Now we want to test the validity of Wigner's surmise, for this we use a set of random matrices from the GUE ($\beta = 2$) and then caluclate the distribution of the gaps between the eigenvalues using we follow the same method laid out by Edelman and Persson[Ala05]. Given n eigenvalues λ_n, we normalise the level spacings[Odl87] u_n so they have a mean of 1 using

$$u_i = (\lambda_{i+1} - \lambda_i)\frac{\sqrt{4n - \lambda_i^2}}{2\pi}$$

We have omitted some details here since they are beyond the scope of this project, but have included more information in appendix B(6.2) for those interested.



(a) 20×20 GUE with 2000 samples taken

(b) 200×200 GUE with 2000 samples taken

Figure 11: Comparison of Wigner Surmise with GUE eigenvalue distributions

Here we can see that the numerical results get closer and closer to the Wigner surmise as we increase the size of the matrices. Wigner's surmise was a result of his introduction of random matrices into the field of nuclear physics, and is a great demonstration of the use of random matrices to model chaotic quantum systems.

6 Appendix

6.1 A. Ouroborology

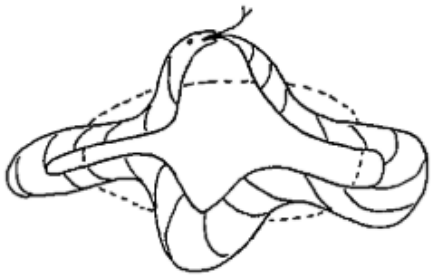


Figure 12: A quantum wave (Ouroborous) interfering constructively around a classical closed orbit.[Ber86]

This is a slight aside but something worth noting nonetheless. Ouroborous was the mythical snake that swallowed it's own tail and surprisingly serves as quite a useful illustration for the interference of quantum waves. A technique developed by Gutzwiller, Balian and Bloch uses such interference forms as the basis for generating the energy spectrum. This technique is used to explain why classically integrable systems have Poisson distributed energy levels and importantly why chaotic systems have GUE distributed energy levels.

6.2 B. Wigner's Surmise

Below we have provided the code used for Wigner's Surmise, here we provide a bit more context for some of the methods used. Here we make use of Painlevé's as another method to verify Wigner's surmise. Painlevé's equations are solutions to certain nonlinear second-order ODE'S in the complex plane that possess Painlevé's quality which is that the only removable singularities are the poles. There are 6 equations and they can all be represented as Hamiltonian systems [Dav15]. Painlevé's V ODE is given by ,

$$(t\sigma)^2 - 4(\sigma - t\sigma')(t\sigma' - \sigma + (\sigma')^2) = 0$$

We impose the boundary conditions on σ as $t \to 0$

$$\sigma(t) \approx -\frac{t}{\pi} - \left(\frac{t}{\pi}\right)^2$$

We also define

$$I(t) = \int_0^t \frac{\sigma(s)}{s} ds$$

We then make use of the *solve ivp* package to solve the initial value problem. We create random GUE's using the method defined in the GUE section. We make use of a Rayleigh distribution which is a continuous distribution for non-negative random variables(Similar to a Chi-square distribution). The rest is fairly straightforward, and we feel the comments on the code suffice for an explanation.

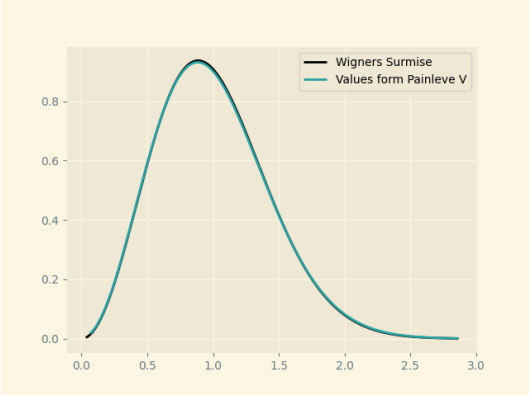


Figure 13: Numerical calculation of Painlevé V function overlaid on Wigner's surmise

Here we see how remarkably accurate Wigner's surmise is and combining this with our results using the GUE corroborates his claim and is definitely an avenue that could be pursued further, perhaps including the other Gaussian ensembles and investigating the connection between the surmise and the nearest neighbour distribution of the zeros of the Riemann-zeta function, with which the surmise bears a remarkable resemblance.

```
import matplotlib.pyplot as plt
import scipy.special
import numpy as np
from scipy.integrate import solve_ivp
from matplotlib import style

plt.style.use('Solarize_Light2')

# This first part here is used to solve the Painleve V differential equation
def f(t, y):
    s = y[0]
    ds = y[1]
    dy0 = ds
    dy1 = -2/t * np.sqrt(max((s-t*ds)*(t*ds - s + (ds)**2), 0))
    dy2 = s/t
    return np.array([dy0, dy1, dy2])

#Define initial and final values
t0 = 1e-15
tf = 9

# Boundary conditions
y0_0 = -t0/np.pi-(t0/np.pi)**2
y0_1 = -1/np.pi-2*t0/np.pi
y0_2 = -t0/np.pi-t0**2/(2*np.pi**2)
y0 = np.array([y0_0, y0_1, y0_2])

# Here we use the solve_ivp to integrate the initial differential equation
sol = solve_ivp(f, (t0, tf) ,y0, max_step=.001)
s = sol.y[0][100:]
ds = sol.y[1][100:]
I = sol.y[2][100:]
```

```
t = sol.t[100:]
x = t/np.pi
E = np.exp(I)
p = E/x**2 * (s**2 + x*np.pi*ds - s)

gamma = lambda x: scipy.special.gamma(x)
def wigner_surmise(s, beta):
    a = 2*(gamma((beta+2)/2))**(beta+1)/(gamma((beta+1)/2))**(beta + 2)
    b = (gamma((beta+2)/2))**2/(gamma((beta+1)/2))**2
    return a * (s**beta) * np.exp(-b*s**2)


wigner_curve = wigner_surmise(x, 2)
plt.plot(x, wigner_curve, color='black',label='Wigners Surmise')
plt.plot(x[100:], p[100:],label='Values form Painleve V',color='#29a3a3')
plt.legend()
plt.show()
```

Listing 3: Wigner versus Painlevé code

6.3 C. Complex power sum algorithm

Below is an informal/imprecise description of a potential algorithm for fast computation of the Hardy Z function on multiple points. As far as we are aware, this approach is novel, insofar as we have not stumbled across it in research, and it is produced independently. We believe that further research and more accurate analysis of this concept would be beneficial.

Specifically, the proposed algorithm computes sums of the form $\sum_{i=1}^{\tau} n^s$. How does this apply to computation of Hardy's Z function? Consider the following manipulations.

$$\begin{aligned}
\frac{\cos(\vartheta(t) - t\ln n)}{\sqrt{n}} &= \frac{1}{\sqrt{n}}\left[\cos(\vartheta(t))\cos(t\ln n) + \sin(\vartheta(t))\sin(t\ln n)\right] \\
&= \frac{1}{\sqrt{n}}\left[\frac{1}{2}\cos(\vartheta(t))\left[e^{it\ln n} + e^{-it\ln n}\right] + \frac{1}{2i}\sin(\vartheta(t))\left[e^{it\ln n} - e^{-it\ln n}\right]\right] \\
&= \frac{1}{\sqrt{n}}\left[\frac{1}{2}\cos(\vartheta(t))\left[n^{it} + n^{-it}\right] + \frac{1}{2i}\sin(\vartheta(t))\left[n^{it} - n^{-it}\right]\right] \\
&= \frac{1}{2}\cos(\vartheta(t))\left[n^{it-1/2} + n^{-it-1/2}\right] + \frac{1}{2i}\sin(\vartheta(t))\left[n^{it-1/2} - n^{-it-1/2}\right] \\
&= \frac{1}{2}n^{it-1/2}\left[\cos(\vartheta(t)) - i\sin(\vartheta(t))\right] + \frac{1}{2}n^{-it-1/2}\left[\cos(\vartheta(t)) + i\sin(\vartheta(t))\right]
\end{aligned}$$

All this to say, when considering the term $\sum_{n=1}^{N(t)} \frac{\cos(\vartheta(t) - t\ln n)}{\sqrt{n}}$, we can instead consider

$$\frac{1}{2}\left[\cos(\vartheta(t)) - i\sin(\vartheta(t))\right]\sum_{n=0}^{N(t)} n^{it-1/2} + \frac{1}{2}\left[\cos(\vartheta(t)) + i\sin(\vartheta(t))\right]\sum_{n=0}^{N(t)} n^{-it-1/2}.$$

Therefore it stands to reason that if we can hastily compute the terms $\sum_{n=0}^{N(t)} n^{it-1/2}$ and

$\sum_{n=0}^{N(t)} n^{it-1/2}$, we'll have a more efficient program. We thence concern ourselves with sums of the form $\sum_{i=1}^{\tau} n^s$, where $\mathcal{O}(\tau) = \mathcal{O}(N(t)) = \mathcal{O}(\sqrt{t})$. Naïvely, this sum takes $O(\tau)$ steps to compute, and it is computed at least as many times as there are zeros we care about (Of course in truth we actually need many more computations, as we also must perform root-finding, but this isn't a relevant detail here). So our idea for a speed-up will revolve around precomputation, which may be availed of to help compute each sum.

It needs be remarked, therefore, that both τ and s are specific to the instance of a given sum, and we do not assume anything about them, other than noting that $\mathcal{O}(\tau) \approx \mathcal{O}(\sqrt{T})$, where T is the height of the millionth zero. More generally, T is the height of the M-th zero, where we want M zeros. We know, from the average vertical spacing in section 4, that T and M grow, more or less, in proportion with one another. That is to say, $\mathcal{O}(T) \approx \mathcal{O}(M)$. Therefore, in the very worst case, $\mathcal{O}(\tau) \approx \mathcal{O}(\sqrt{M})$.[18]

Thus, roughly speaking, to compute M zeros takes about $\mathcal{O}(M\sqrt{M})$ steps. Indeed, although they do not show workings, this is the same term used in Odlyzko's 1988 paper. This is all to say, if precomputation takes fewer than $\mathcal{O}(M\sqrt{M})$ steps, and the sums are quicker than $\mathcal{O}(\sqrt{M})$, we may be onto something.

So what's the big idea then? Consider an example sum for now. For some $s \in \mathbb{C}$:

$$S = 1^s + 2^s + 3^s + 4^s + 5^s + 6^s + 7^s + 8^s + 9^s.$$

This sum can be re-arranged into the following form:

$$1^s + 2^s(1^s + 2^s + 3^s + 4^s) + 3^s(1^s + 3^s) + 5^s + 7^s.$$

Notice that each term above is a multiple of some prime power p^s. In general this is always possible. Consider a sum of googol length. (Note, still finite).

$$S = 1^s + 2^s + 3^s + 4^s + 5^s + 6^s + 7^s + 8^s + 9^s + \dots$$

We can perform a similar re-arrangement, resulting in the following.

$$\begin{aligned}
&1^s \\
&+2^s(1^s + 2^s + 3^s + 4^s + 5^s + 6^s + 7^s + 8^s + 9^s + \dots) \\
&+3^s(1^s + 3^s + 5^s + 7^s + 9^s + 11^s + 13^s + 15^s + 17^s + \dots) \\
&+5^s(1^s + 5^s + 7^s + 11^s + 13^s + 17^s + 19^s + 23^s + 25^s + \dots) \\
&+7^s(1^s + 7^s + 11^s + 13^s + 17^s + 19^s + 23^s + 29^s + 31^s + \dots) + \dots
\end{aligned}$$

How do we determine the parentheticals corresponding to each prime power? For a prime p, the terms that appear in the sum $(1^s + p^s + \dots)$ are precisely those terms which

[18] We are aware that this usage of big O notation is abusive, and that $\mathcal{O}(f)$ refers to a certain equivalence class, so there is not a strong way in which one can say $\mathcal{O}(A) \approx \mathcal{O}(B)$. This is only for a ballpark estimate.

do are coprime to every prime less than p. So for example, looking at the line associated with 5^s, we can see that every term in $\{1, 5, 7, 11, 13, 17, 19, 23, 25, \dots\}$ is coprime to 2 and 3. This isn't hard to see if you try constructing one of these re-arrangements yourself.

Some nomenclature and notation: For a prime p_i, let the n-th number which is coprime to all primes $2, 3, \dots, p_{i-1}$ be denoted $E^n_{p_i}$. These will be referred to as the **eratosthenes numbers**[19] of p_i. Let the sum of powers of the first N eratosthenes numbers of p be denoted S^N_p. That is,

$$S^N_p = \sum_{n=1}^{N} (E^n_p)^s.$$

It should be noted that S^N_p is abusive notation, as we should really be specifying that the power to be taken is s. However, it should be sufficiently contextualised hereafter. In light of these new definitions, let's re-write the googol example.

$$\begin{aligned}
&1^s \\
&+2^s((E^1_2)^s + (E^2_2)^s + (E^3_2)^s + (E^4_2)^s + (E^5_2)^s + (E^6_2)^s + (E^7_2)^s + (E^8_2)^s + (E^9_2)^s + \dots) \\
&+3^s((E^1_3)^s + (E^2_3)^s + (E^3_3)^s + (E^4_3)^s + (E^5_3)^s + (E^6_3)^s + (E^7_3)^s + (E^8_3)^s + (E^9_3)^s + \dots) \\
&+5^s((E^1_5)^s + (E^2_5)^s + (E^3_5)^s + (E^4_5)^s + (E^5_5)^s + (E^6_5)^s + (E^7_5)^s + (E^8_5)^s + (E^9_5)^s + \dots) \\
&+7^s((E^1_7)^s + (E^2_7)^s + (E^3_7)^s + (E^4_7)^s + (E^5_7)^s + (E^6_7)^s + (E^7_7)^s + (E^8_7)^s + (E^9_7)^s + \dots) + \dots \\
&= 1^s + 2^s S^{\text{googol}/2}_2 + 3^s S^{\text{something}}_3 + 5^s S^{\text{something else}}_5 + \dots
\end{aligned}$$

In general, to evaluate $\sum_{i=1}^{\tau} n^s$ is to evaluate the series

$$\sum_{p \le \tau} p^s S^{\eta(p,\tau)}_p.$$

Here $\eta(p, \tau)$ denotes the largest m such that $p \cdot E^m_p \le \tau$. So far, all we've accomplished the re-arrangement of the series, which will ultimately still have τ terms in it. So, here's where we bring in pre-computation. At the beginning of the code, we can compute all relevant eratosthenes numbers E^n_p, of which there are exactly $\lfloor \sqrt{T/2\pi} \rfloor - 1$ (This fact follows from the construction of the eratosthenes numbers as we've done it). We omit the exact details of how to compute these efficiently, but it should be possible in $\mathcal{O}(\sqrt{M} \log M)$ time using a technique similar to the sieve of eratosthenes on a set data structure. (set as implemented in c++, specifically). It could also be computed more naïvely and still be fine.

The above will be our precomputation step. Because we have a time complexity of $\mathcal{O}(\sqrt{M} \log M)$, and $\sqrt{M} \log M << M\sqrt{M}$, this performs how we would like it to. Having computed all needed eratosthenes numbers, we can compute $\eta(p, \tau)$ as described above by using binary search on m to find the largest m such that $p \cdot E^m_p \le \tau$. This step takes $\mathcal{O}(\log M)$ and happens as many times as there are primes less than τ, which is $\mathcal{O}(\tau / \log \tau) = \mathcal{O}(\sqrt{M} / \log M)$, following from the prime counting function. All in all then,

[19] Named as such because they are precisely the unmarked numbers on the p-th step when computing the sieve of eratosthenes.

the η calculations take $\mathcal{O}(\sqrt{M})$ time.

Finally then, we come to the crucial question, how to calculate $S_p^{\eta(p,r)}$ once we know $\eta(p,r)$? More generally, how do we crack $S_{p_i}^N$? Fortunately, by construction, S satisfies the following recursive equation.

$$S_{p_i}^N = (p_i)^s \cdot S_{p_i}^C + S_{p_{i+1}}^{N-C}.$$

Here, C is defined as the largest c such that $p_i \cdot E_{p_i}^c \leq E_{p_i}^N$. For example, $S_5^9 = 5^s \cdot S_5^2 + S_7^7$ and $S_5^{11} = 5^s \cdot S_5^2 + S_7^9$. Such a C always exists, and we have $C, N-C < N$, so this recurrence always terminates with a term of the form $S_q^1 = 1$.

We can, however, terminate the recurrence even earlier in many cases. For any p such that $p^2 > N$, the recurrence relation above simply produces a sum of consecutive prime powers and a 1. So for example, $S_5^{10} = 1^s + 5^s + 7^s + 11^s + 13^s + 17^s + 19^s + 23^s$. In fact, we conjecture that most recurrences will go this way. So it will be worth our time to speed this up. When computing the sum for a specific power s, let's first compute the sums 2^s, $2^s + 3^s$, $2^s + 3^s + 5^s$, $2^s + 3^s + 5^s + 7^s$ and so forth, until we reach the sum $2^s + 3^s + 5^s + \cdots + P^s$ for the largest possible $P \leq N$. These are called prefix sums, and can be computed iteratively, whereby $\text{prefix}[i+1] = \text{prefix}[i] + (p_i)^s$. This will only take as many steps as there are primes less than N, which as mentioned earlier is $\mathcal{O}(N/\log N) = \mathcal{O}(\sqrt{M}/\log M)$. Hence to compute these for all M zeros will take $(M\sqrt{M}/\log M)$. This term is the slowest analysed term thus far, but it is still faster than the original program. Perhaps further study could fix this bottleneck.

The only question left is "how many calls will be made to S_p^N throughout the recursive process?" We have not satisfactorily analysed this as of right now. Certainly the depth of the recursion is very reasonably bounded, owing to the prefix sum trick. However, this says nothing of the breadth of the recursion. We believe the best thing to do would be to simply implement this program, and see how it behaves. Owing to time pressures, however, we save this for future research.

6.4 D. Binary search for pair correlation.

This is a discussion of an algorithm for computing the pair correlation function which has a better computational time complexity than the naïve approach, but which ultimately runs slower on the region we are studying. If the region were made larger, this algorithm ought to be considered.

The idea is to do a bit of precomputation so that we may quickly evaluate sums of the form $\Delta_k(n) := \delta_n + \cdots + \delta_{n+k}$, and then to make use of discrete binary search. By this we mean that if we want to find the smallest k such that $\delta_n + \cdots + \delta_{n+k} > x$ for some x, we first guess a lower bound and an upper bound. Let $L := 0$ and let $R := 10000$, so $\Delta_L(n) = \delta_n$, and $\Delta_R(n) = \delta_n + \cdots + \delta_{n+10000}$. Certainly, we will have $\Delta_L(n) < x < \Delta_R(n)$. In this way, L and R "bracket" the answer, just as in the bisection root finding method. So we

continue just as we would for that. Consider halfway between L and R, i.e. consider 5000. If $\Delta_{5000}(n) < x$, we set L equal to 5000. If $\Delta_{5000}(n) > x$, we set R equal to 5000, and so forth.

The details are actually a lot more subtle than this, mainly because binary search over discrete sets have several edge cases, but this is the basic principle. In this way, provided we can compute $\Delta_k(n)$ in $\mathcal{O}(1)$ time, we can compute $k_{\min}$ in $\mathcal{O}(\log P)$ time, where P is the number of zeros we are after. Comparing this to the naïve method, which has a time complexity of $\mathcal{O}(P)$, we can see an improvement.

So how do we compute $\Delta_k(n)$ on the fly? We first compute $\Delta_k(0)$ for all $k < P$. This will take $\mathcal{O}(P)$ time but will only have to be done once. Then, to compute $\Delta_k(n)$, you just take $\Delta_{k+n}(0) - \Delta_{\{}n-1\}(0)$. Why does this work? Because $\Delta_{k+n}(0) - \Delta_{n-1}(0) = (\delta_0 + \cdots + \delta_{n-1} + \delta_n + \cdots + \delta_{n+k}) - (\delta_0 + \cdots + \delta_{n-1}) = \delta_n + \cdots + \delta_{n+k} = \Delta_k(n)$. This is the same concept of "Prefix sums" as was mentioned in appendix B.

Just because the time complexity is better, however, doesn't mean that the performance is better. While this is usually true, it turns out that, in our interval of study, the naïve method has an early termination.

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

f = open('1000000.txt', encoding="utf-8")

Delt = 0.095
M = 40000
N = 800000

n = int(f.readline())

zeros = [float(f.readline()) for i in range(n)]

deltas = [np.log(zeros[i]/(2.0*np.pi))*(zeros[i+1]-zeros[i])/(2.0*np.pi) for
    i in range(n-1)]

for i in range(1,num_zeros):
    prefix_deltas[i] = prefix_deltas[i-1] + deltas[i-1]

def g(x):
    print(x)
    ans = 0
    for n in range(N, N+M):
        L = n
        R = 99999
        # Find maximum k such that delta[n]+...delta[n+k] < x+Delt
        while L < R:
            k = (R+L+1)//2
            if prefix_deltas[k]-prefix_deltas[n-1] < x+Delt:
                L = k
            else:
```

```
                R = k-1
        kmax = L
        L = n
        R = n + 4
        # Find minimum k such that delta[n]+...delta[n+k] > x-Delt
        while L < R:
            k = (R+L)//2
            if prefix_deltas[k]-prefix_deltas[n-1] > x-Delt:
                R = k
            else:
                L = k+1
        kmin = L
        if prefix_deltas[kmax]-prefix_deltas[n-1] > x+Delt or prefix_deltas[
    kmin]-prefix_deltas[n-1] < x-Delt:
            continue
        if prefix_deltas[kmax]-prefix_deltas[n-1] < x-Delt or prefix_deltas[
    kmin]-prefix_deltas[n-1] > x+Delt:
            continue
        ans += (kmax-kmin+1)/(2.0*Delt*M)
    return ans

def gue(x):
    return 1.0 - (np.sin(np.pi*x)/(np.pi*x))**2

domain = np.linspace(2*Delt, 5, 500)
codomain = [g(x) for x in domain]

plt.plot(domain, codomain)
plt.plot(domain, gue(domain))
plt.show()
```

Listing 4: Binary search pair correlation code

6.5 E. Python code for computing zeros.

```
"""
zeros.py - Authored by Oisín Davey and Tim Daly - 15/08/2024

This code is to be understood alongside our
project report for the module MP354 - Computational
Physics at Maynooth University. The purpose of zeros.py
is to efficiently compute the first n roots of the Riemann
zeta function along the critical line. Here n can be any number
below 13999824, and it should succesfully report the zeros.
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.special import loggamma, lambertw
from multiprocessing import Pool

"""
Siegel's theta function.
Here, the function either returns the exact expression for theta,
or it returns the asymptotic expansion given in the report.
t=13 was chosen as the cut-off point because this is the first
```

```
integer point at which the asymptotic expansion is correct to 6 decimal
    places.
"""
def theta(t):
    if t<13:
        return np.imag(loggamma(complex(1/4, t/2))) - (t/2)*np.log(np.pi)
    else:
        return t*np.log(t/(2.0*np.pi))/2.0-np.pi/8.0-t/2.0+1/(48*t)

"""
Implementation of Ψ(ϱ) using fifteenth order Taylor polynomial.
Coefficients are stored as a global tuple for speed.
Coefficients are also used to compute the third derivative.
Returns a 2-tuple containing Ψ(ϱ) and it's third derivative at z.
"""

S = np.sin(np.pi/8.0)
C = np.cos(np.pi/8.0)

(pi1, pi2, pi3, pi4, pi5, pi6, pi7) = (np.pi, np.pi**2, np.pi**3, np.pi**4,
    np.pi**5, np.pi**6, np.pi**7)

coeffs = (S, pi1*(pi1*S - C)/2, pi2*(-3*S + 5*(pi2)*S - 6*pi1*C)/24, pi3
    *(-45*pi1*S + 61*pi3*S + 15*C - 75*pi2*C)/720, pi4*(105*S - 1050*pi2*S +
    1385*pi4*S + 420*pi1*C - 1708*pi3*C)/40320, pi5*(4725*pi1*S - 38430*pi3*S
    + 50521*pi5*S - 945*C + 15750*pi2*C - 62325*pi4*C)/3628800, pi6*(-10395*S
    + 259875*pi2*S - 2056725*pi4*S + 2702765*pi6*S - 62370*pi1*C + 845460*pi3*
    C - 3334386*pi5*C)/479001600, pi7*(-945945*pi1*S + 19234215*pi3*S -
    151714563*pi5*S + 199360981*pi7*S + 135135*C - 4729725*pi2*C + 62387325*
    pi4*C - 245951615*pi6*C)/87178291200)

def psi(z):
    return sum((z**(2*n))*coeffs[n] for n in range(0, 8)), sum(2*n*(2*n-1)
    *(2*n-2)*(z**(2*n-3))*coeffs[n] for n in range(2, 8))

"""
Hardy's Z function.
The variables m, and z represent N(t) and ϱ(t) in the report, respectively.
tbar, stbar and thetaval are stored because they are used multiple times.
siegel_sum represents the sum term in the siegel expression for Z(t).
"""

def hardy_z_function(t):
    tbar = t/(2.0*np.pi)
    stbar = np.sqrt(tbar)
    m = np.floor(stbar)
    z = 1.0-2.0*(stbar-m)
    thetaval = theta(t)
    psivals = psi(z)
    siegel_sum = 2.0*sum([np.cos(thetaval-t*np.log(float(n)))/np.sqrt(float(n
    )) for n in range(1, int(m)+1)])
    return np.real(siegel_sum) + ((-1.0)**(m+1)/((tbar)**0.25))*(psivals[0]+
    psivals[1]/(12.0*(np.pi**2)*stbar))

"""
Finds the first n Gram points.
That is, finding all g_m such that ϑ(g_m) = mπ for m in {0,1,2,3, ..., n-1}.
```

```
Using the asymptotic formula as described.
"""
def gram(n):
    return [np.real(2.0*np.pi*np.exp(1+lambertw((8.0*m+1.0)/(8.0*np.e)))) for
     m in range(0, n)]

"""
From a list of Gram points, returns only the indices of the good Gram points.
"""
def good_gram_indices(gram_points):
    good_indices = []
    for n in range(0, len(gram_points)):
        if ((-1)**n)*hardy_z_function(gram_points[n])>=0.0:
            good_indices.append(n)
    return good_indices

"""
Bisection, Illinois, and pegasus methods respectively
L and R are the left and right endpoints of the bracketing interval
    respectively.
δ specifies the input precision sought after.
This is to say, bisection will return a value within a distance δ of a root.
func is the function to find roots of. In practice this will always be
    hardy_z_function,
but it makes the code cleaner.
"""
def bisection(a, b, δ, func):
    while b-a > δ:
        c = 0.5*(b+a)
        (a, b) = (a, c) if func(c)*func(a)<0 else (c, b)
    return a

def illinois(a, b, δ, func):
    (fa, fb) = (func(a), func(b))
    while abs(b-a) > δ:
        c = (a*fb-b*fa)/(fb-fa)
        fc = func(c)
        (a, fa) = (a, 0.5*fa) if fb*fc > 0 else (b, fb)
        (b, fb) = (c, fc)
    return a

def pegasus(a, b, δ, func):
    (fa, fb) = (func(a), func(b))
    while abs(b-a) > δ:
        c = (a*fb-b*fa)/(fb-fa)
        fc = func(c)
        (a, fa) = (a, fa*fb/(fb+fc)) if fb*fc > 0 else (b, fb)
        (b, fb) = (c, fc)
    return a

"""
Finds roots of ζ, assuming Gram's law.
This function is never called, because Gram's law is false,
but it may be called in order to investigate the failure thereof.
"""
def zeros_grams_rule(gram_points):
    return [(pegasus(gram_points[i-1], gram_points[i])) for i in range(1, len
```

```
    (gram_points))]

"""
Finds zeros obeying Rosser's Rule.

In the interval splitting, the argument good_interval is a 3-tuple (L, R, N), where:
    L is the lower gram point of the good interval
    R is the upper gram point of the good interval
    N is the number of gram intervals in the good interval.

slices is a measure of how many times the intervals have been split
in half. If n roots are not found after interval has been cleft in twain
10 times (producing 1024 segments), we assume a violation of rosser's
rule has been found. In this case, we take whichever roots are found after 10 slices.
As mentioned in the report, this never does happen in our million zeros.

In each slice, we count how many roots are detected through bracketing.
If fewer roots are detected than Rosser's rule predicts, we proceed
in clefting the intervals in twain. We must then attend to the new intervals
thereby created, and discard the old intervals.

Then, whether a violation has been detected or not, we perform root-finding
with the pegasus algorithm on all detected roots.
"""
def rossers_interval_splitting(good_interval):
    interval_zeros = []
    (L, R, num_intervals) = good_interval
    sample_inputs = [(L + (float(n)*(R-L))/float(num_intervals)) for n in range(0, num_intervals+1)]
    sample_outputs = [hardy_z_function(x) for x in sample_inputs]
    slices = 1
    while slices < 10:
        root_count = sum((sample_outputs[i]*sample_outputs[i+1] <= 0.0) for i in range(0, len(sample_inputs)-1))
        if root_count < num_intervals:
            slices += 1
            new_sample_inputs = []
            new_sample_outputs = []
            for i in range(0, len(sample_inputs)-1):
                input_midpoint = 0.5*(sample_inputs[i]+sample_inputs[i+1])
                output_midpoint = hardy_z_function(input_midpoint)
                new_sample_inputs.append(sample_inputs[i])
                new_sample_inputs.append(input_midpoint)
                new_sample_inputs.append(sample_inputs[i+1])
                new_sample_outputs.append(sample_outputs[i])
                new_sample_outputs.append(output_midpoint)
                new_sample_outputs.append(sample_outputs[i+1])
            sample_inputs = new_sample_inputs
            sample_outputs = new_sample_outputs
        else:
            break
    if slices == 10:
        print("Rosser's rule violation detected. Moving forward having possibly missed a zero.")
    for i in range(0, len(sample_inputs)-1):
```

```
        L=sample_inputs[i]
        R=sample_inputs[i+1]
        if hardy_z_function(R)*hardy_z_function(L)>0:
            continue
        interval_zeros.append(pegasus(L, R, 0.00001, hardy_z_function))
    return interval_zeros

"""
Helper function for rossers_interval_splitting, making use of parallel
    processing.
If your computer has N cores, you should say p = Pool(processes = N), unless
    your
computer has hyperthreading, you should say p = Pool(processes = 2N).
"""
def zeros_rossers_rule(gram_points, good_indices):
    zeros=[]
    grouped_zeros = []
    p = Pool(processes = 8)
    grouped_zeros = p.map(rossers_interval_splitting, ((gram_points[
    good_indices[i-1]], gram_points[good_indices[i]], good_indices[i]-
    good_indices[i-1]) for i in range(1, len(good_indices))))
    p.close()
    p.join()
    for interval_zeros in grouped_zeros:
            for zed in interval_zeros:
                zeros.append(zed)
    return zeros

"""
It's important that we have a main wrapper for the parallel processing to
    work.
Depending on the operating system, this isn't strictly necessary, but
it definitely won't work for standard windows python interpretter.
"""
def main():
    # Number of zeros
    n=1000

    print("Finding Gram points.")
    gram_points = gram(n)
    print("Gram points found.")
    good_indices = good_gram_indices(gram_points)
    print("Good Gram points found.")
    # The first zero is exceptional insofar as it occurs before the first
    gram point. Hence it is sought independently.
    zeros = [bisection(1.0, gram_points[0], 0.00001, hardy_z_function)] +
    zeros_rossers_rule(gram_points, good_indices)
    """
    Writes the zeros to a file locally.
    If you don't have write priviledges, you'll have to change this.
    """
    outp = open(str(n)+".out", "w")
    outp.writelines([str(zed)+"\n" for zed in zeros])
    outp.close()

    """
    The following plots Hardy's Z function and is to be uncommented only for
```

```
    small n.
    """
    # Arrays of values
    #tdomain = np.linspace(0.1, 100, 1000)
    #codomain = np.array([hardy_z_function(t) for t in tdomain])
    #res=np.array([hardy_z_function(t) for t in gram_points])
    #plt.plot(tdomain, codomain, label='Zeta Function')
    #plt.scatter(gram_points, res, color='red', label='Gram Points')
    #plt.ylabel('Zeta')
    #plt.grid()
    #plt.axhline(y=0,color='black')
    #plt.legend()
    #plt.show()

if __name__ == "__main__":
    main()
```

Listing 5: Full script for computing the first n zeros of ζ

6.6 F. Python code for computing pair correlation of zeta zeros.

```
"""
pair_correlation.py - Authored by Oisín Davey and Tim Daly - 15/08/2024

This code is to be understood alongside our
project report for the module MP354 - Computational
Physics at Maynooth University. The purpose of pair_correlation.py
is to compute and plot the pair correlation function of
the zeros of the Riemann zeta function. The zeros should
be supplied in a text document, where each zero occupies one
line and are ordered consecutively. Also plotted is the pair
correlation function for the gaussian unitary ensemble.
"""

import numpy as np
import matplotlib.pyplot as plt
from multiprocessing import Pool

inp = open('1000000.txt', encoding="utf-8")
num_zeros = 1000000
zeros = [float(inp.readline()) for i in range(num_zeros)]


"""
Figures, as chosen in the report.
"""

Δ = 0.095
M = 40000
N = 800000

"""
δ is a list of the normalised spacings between non-trivial
zeros, as described in the report.
"""

```

```
δ = [np.log(zeros[i]/(2.0*np.pi))*(zeros[i+1]-zeros[i])/(2.0*np.pi) for i in range(num_zeros-1)]

"""
Here we compute the pair correlation using the naive method
described in the report. We stress, however, that
although this method is naive, it is correct and fast.

kmin and kmax are initialised in such a way that we
might immediately detect whether any valid values for
k in the interval [x-Δ, x+Δ] were found. kmax starts
out negative, so if the if-statement ever runs, kmax
will be updated. Likewise kmin starts out impossibly
big, so it too would be updated.
"""
def zeta_g(x):
    ans = 0
    for n in range(N, N+M):
        running_delta_sum = 0
        k = 0
        kmin = +num_zeros+1
        kmax = -num_zeros-1
        while running_delta_sum <= x+Δ:
            if (x-Δ <= running_delta_sum) and (running_delta_sum <= x+Δ):
                kmin = min(kmin, running_delta_sum)
                kmax = max(kmax, running_delta_sum)
            running_delta_sum += deltas[n+k]
            k += 1
        if kmin == +num_zeros+1 or kmax == -num_zeros-1:
            continue
        ans += (kmax-kmin+1)/(2.0*Δ*M)
    return ans

"""
Pair correlation function of the Gaussian unitary
ensemble, to be plotted against g(r).
"""
def gue(x):
    return 1.0 - (np.sin(np.pi*x)/(np.pi*x))**2

"""
It's important that we have a main wrapper for the parallel processing to work.
Depending on the operating system, this isn't strictly necessary, but
it definitely won't work for standard windows python interpretter.
"""
def main():
    domain = np.linspace(2*Δ, 5, 500)
    codomain = [zeta_g(x) for x in domain]
    p = Pool(processes = 8)
    codomain = p.map(g, domain)
    p.close()
    p.join()

    plt.plot(domain, codomain)
    plt.plot(domain, gue(domain))
    plt.show()
```

```
if __name__ == "__main__":
    main()
```

Listing 6: Pair correlation code

6.7 G. Python code used to generate the plots in this document.

The code which follows is not all immensely commented. We feel this is acceptable as this code is ancillary to our main two scripts. The code to plot the Gram points and pair correlation are omitted, as these are just minor modifications of zeros.py and pair_correlation.py respectively. The code for figure 8 is also omitted, as it is just a less complicated version of the code for figure 5.

Figure 1. Title page spiral.

```
"""
zeta_loops.py - Authored by Oisín Davey & Tim Daly - 15/05/2024
This script plots a curve paramaterised by the zeta function along
the critical line. When this curve intersects the origin, the
real and imaginary points are both zero, and this indicates that a
zero has been found.
"""

import mpmath as mp
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import style

plt.style.use('Solarize_Light2')

"""
When using mpmath, one must specify the decimal precision.
Here we are, needlessly I might add, using 25 decimal places.
"""
mp.dps = 25

# Evaluate ζ(0.5+it) for 0<=t<=30.
ζ_domain = np.linspace(0, 30, 1000)
ζ_range = [mp.zeta(complex(0.5, t)) for t in ζ_domain]

ζ_real = [np.real(z) for z in ζ_range]
ζ_imag = [np.imag(z) for z in ζ_range]

# Plotting
plt.xlabel('Re$\left[\zeta\left(\\frac{1}{2}+ti\\right)\\right]$')
plt.ylabel('Im$\left[\zeta\left(\\frac{1}{2}+ti\\right)\\right]$')
plt.axhline(y=0, linestyle='dashed', c='grey')
plt.axvline(x=0, linestyle='dashed', c='grey')

plt.axis('equal')
plt.plot(ζ_real, ζ_imag)
plt.show()
```

Listing 7: Title page spiral code.

Figures 2 & 3. Image of ζ on $H_{1/2}$.

```
"""
half_plane.py - Authored by Oisín Davey & Tim Daly - 15/05/2024
This script takes a gride in the complex plane, and transforms
it under the map zeta. Then plots the transformed points in a
smooth manner.
"""

import mpmath as mp
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import style

plt.style.use('Solarize_Light2')

# Helper function for mp.zeta which avoids the singularity.
def zet(z):
    return mp.zeta(z) if z != 1 else 0

"""
When using mpmath, one must specify the decimal precision.
Here we are, needlessly I might add, using 25 decimal places.
"""
mp.dps = 25

# Points separated by 1/8 intervals to separate gridlines.
x_points = np.arange(1.01, 4, 0.125)
y_points = np.arange(-5, 5, 0.125)

# Closely packed points to compose gridlines.
x_dense = np.linspace(1.0, 3, 200)
y_dense = np.linspace(-5, 5, 200)

for x in x_points:
    plt.plot([np.real(zet(complex(x,y))) for y in y_dense], [np.imag(zet(
    complex(x,y))) for y in y_dense])
for y in y_points:
    plt.plot([np.real(zet(complex(x,y))) for x in x_dense], [np.imag(zet(
    complex(x,y))) for x in x_dense])

plt.axvline(x=0.5, linestyle='dotted', c=(0, 0, 0, 0.5))

plt.axis('equal')
plt.show()
```

Listing 8: Code for the image of zeta on the half-plane.

Figure 4. 3D plot of spiralling zeta function along critical line.

```
"""
zeta_path.py - Authored by Oisín Davey & Tim Daly - 15/05/2024
This script plots a curve paramaterised by the zeta function along
the critical line. Unlike zeta_loops.py however, the parameter t
is given its own axis, so that we might see the path through space.
```

```
"""

import mpmath as mp
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import style

plt.style.use('Solarize_Light2')

"""
When using mpmath, one must specify the decimal precision.
Here we are, needlessly I might add, using 25 decimal places.
"""
mp.dps = 25

# Evaluating ζ along the critical line.
ζ_domain = np.linspace(-20, 50, 5000)
ζ_range = [mp.zeta(complex(0.5, t)) for t in ζ_domain]

"""
Here, col is an anonymous function which alters the opacity
of the points to be plotted depending on the value of t.
This produces the fade-out effect seen in the final image.
"""
col = lambda t : 1.0 if (t <= 15) else (np.exp(-0.5*(t-15.0)))
ζ_colors = [(38/255, 139/255, 210/255, col(abs(t-15))) for t in ζ_domain]

# Telling plt that we want to do things in 3d.
ax = plt.figure().add_subplot(projection='3d')

"""
Plotting the found points, along with their colours.
zorder is very important when plotting in 3d, as it
dictates which points are "in front" of others.

These points represent the path here, so we will want
to superpose the zeros over them, hence they get
zorder=0.
"""
ax.scatter([np.real(z) for z in ζ_range], [np.imag(z) for z in ζ_range], ζ
    _domain, label="$\zeta\left(\\frac{1}{2}+ti\\right)$", c=ζ_colors,
    linewidth=0.5, s=10, zorder=0)

# Plotting the central axis.
ax.plot(np.zeros(len(ζ_domain)), np.zeros(len(ζ_domain)), ζ_domain, c=(0.0,
    0.0, 0.0, 0.5), linestyle='dashed', zorder=10)

# Plotting the zeros. Exact figures from Odlyzko.
ax.scatter([0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [14.134725142, 21.022039639,
    25.010857580, 30.424876126, 32.935061588], c='black', edgecolors='black',
    marker='o', s=120, depthshade=False, zorder=25000)

ax.set_xlim3d(-3, +3)
ax.set_ylim3d(-3, +3)
ax.set_zlim3d(0, 30)

```

```
ax.w_xaxis.set_pane_color((0.0, 0.0, 0.0, 0.0))
ax.w_yaxis.set_pane_color((0.0, 0.0, 0.0, 0.0))
ax.w_zaxis.set_pane_color((0.0, 0.0, 0.0, 0.0))
ax.set_box_aspect([10, 10, 10])

ax.set_xlabel('Re$\left[\zeta\left(\\frac{1}{2}+ti\\right)\\right]$', fontsize
    =12, rotation=30)
ax.set_ylabel('Im$\left[\zeta\left(\\frac{1}{2}+ti\\right)\\right]$', fontsize
    =12, rotation=30)
ax.set_zlabel('t', fontsize=12, rotation=0)

plt.tight_layout()

plt.show()
```

Listing 9: Code for the spiraling 3D zeta plot.

Figure 5. Contour of Siegel's rediscovered integral.

```
"""
contour.py - Authored by Oisín Davey & Tim Daly - 15/05/2024
This script creates a heatmap of the modulus of the integrand
of one of Sigel's integrals, and superposes a contour over
this map. This contour then computes a certain remainder
term which Siegel expands to get his eponymous formula.
"""

import mpmath as mp
import numpy as np
import matplotlib.pyplot as plt

from matplotlib import style

plt.style.use('Solarize_Light2')

fig, ax = plt.subplots()

# The integrand on which the heatmap is based.
def integrand(z):
    return np.log(np.abs(((-z)**3*np.exp(-2*z))/(np.exp(z)-1)))

"""
Dimensions of the heatmap we're looking for, centered
at zero, extending L units down & left, and R units
up & right. N squared samples of this grid are then taken.
"""
L = -50
R = +50
N = 900

# Produces a dynamically shaped grid. Not strictly needed.
X, Y = np.meshgrid(np.linspace(L, R, N), np.linspace(L, R, N))

# All points, interpreted as complex numbers
z = np.array([[integrand(complex(x, y)) for y in np.linspace(L, R, N)] for x
    in np.linspace(L, R, N)])

```

```
# Fences & Fenceposts off-by-one correction.
z = z[:-1, :-1]
z_min, z_max = -np.abs(z).max(), np.abs(z).max()

# Creating heatmap and colour bar.
c = ax.pcolormesh(X, Y, z, cmap='Spectral', vmin=z_min, vmax=z_max)
ax.axis([X.min(), X.max(), Y.min(), Y.max()])
cbar = fig.colorbar(c, ax=ax)
cbar.set_label(label='$\log\left|\\frac{(-z)^{4-1}e^{-2z}}{e^z-1}\\right|$',
    size=15)

ax.axhline(y=0, c=(0, 0, 0, 0.3), linestyle='dashed', zorder=5)
ax.axvline(x=0, c=(0, 0, 0, 0.3), linestyle='dashed', zorder=5)

# Creating and plotting the locations of a few sample singularities.
singularities = np.arange(-20.0*np.pi, 20.0*np.pi, 2.0*np.pi)
ax.scatter(np.zeros(len(singularities)), singularities, c='black', zorder =
    10, label='Singularities', marker='x')

# Plotting the contour
circle = np.array([[7.0*np.pi*np.cos(θ), 7.0*np.pi*np.sin(θ)] for θ in np.
    linspace(0.2, 2.0*np.pi-0.2, 100)])
topline = np.array([[7.0*np.pi*np.cos(0.2)+t, 7.0*np.pi*np.sin(0.2)] for t in
     np.linspace(0, 100, 5)])

ax.plot(circle[:,0], circle[:,1], linewidth=2.5, c='#2aa198')
ax.plot(topline[:,0], topline[:,1], linewidth=2.5, c='#2aa198', label='
    Contour')
ax.plot(topline[:,0], -topline[:,1], linewidth=2.5, c='#2aa198')
ax.scatter([7.0*np.pi*np.cos(0.2), 7.0*np.pi*np.cos(0.2)], [7.0*np.pi*np.sin
    (0.2), -7.0*np.pi*np.sin(0.2)], s=5, c='#2aa198')
ax.scatter([7.0*np.pi*np.cos(0.2)+15.5], [7.0*np.pi*np.sin(0.2)], marker='<',
     c='#2aa198', s=100)
ax.scatter([7.0*np.pi*np.cos(0.2)+16.0], [-7.0*np.pi*np.sin(0.2)], marker='>'
    , c='#2aa198', s=100)
ax.scatter([-7.0*np.pi], [0.0], marker='v', c='#2aa198', s=100, zorder=30)
ax.set_xlabel('Re', fontsize=20)
ax.set_ylabel('Im', fontsize=20)

ax.axis('equal')
ax.legend(prop={'size': 15})

plt.show()
```

Listing 10: Code for the contour integral heatmap.

Figure 11. Wigner Surmise and GUE eigenvalue distribution.

```
import matplotlib.pyplot as plt
import scipy.special
import numpy as np
from scipy.integrate import solve_ivp
from matplotlib import style

plt.style.use('Solarize_Light2')

def random_gue_matrix(n):
    # Here we generate some random real and imaginary parts with mean 0 and
```

```
    variance 1
    real = np.random.normal(loc=0, scale=1/np.sqrt(2), size=(n, n))
    imag = np.random.normal(loc=0, scale=1/np.sqrt(2), size=(n, n))

    # Combine these to make the final matrix
    mat = real + 1j * imag

    return mat

def random_goe_matrix(n):
    # Same as before except with GOE
    a = np.random.normal(loc=0.0, scale=1.0, size=(n, n))
    mat = (a + a.T) / 2.0
    return mat


gamma = lambda x: scipy.special.gamma(x)
def wigner_surmise(s, beta):
    a = 2*(gamma((beta+2)/2))**(beta+1)/(gamma((beta+1)/2))**(beta + 2)
    b = (gamma((beta+2)/2))**2/(gamma((beta+1)/2))**2
    return a * (s**beta) * np.exp(-b*s**2)

# Normalised according to Odlyzko
def normalise(t1, t2):
    return (t1 - t2)*(np.sqrt(4*N-t2**2)/(2*np.pi))

runs = 2000 #Num of rand matrices
delta = []
N = 20 #Size of matrix

#Putting everything together
for j in range(runs):
    mat = random_gue_matrix(N)
    t_s, V = np.linalg.eigh(mat)#Appromiate using Rayleigh(\sqrt2)
    t_s.sort()
    t_smid = t_s[round(N/4):round(3*N/4)]
    for i in range(len(t_smid)-1):
        eig_spaces = [normalise(t_smid[i+1], t_smid[i])]
    delta.extend(eig_spaces)

plt.ylabel("Density")
plt.xlabel("$t_1 - t_2$")
plt.hist(delta, bins=50, density=True, color='#29a3a3', label='Numerical GUE'
    )

t = np.arange(0., 3, 0.05)
wigner_curve = wigner_surmise(t, 2)
plt.plot(t, wigner_curve, label='Wigner Surmise', color='orange')
plt.show()
```

Listing 11: Wigner Surmise and GUE code.

References

[Gra03] J. P. Gram. "Sur les ares de la Fonction ζ de Riemann". In: *Actu Math* 27 (1903), pp. 289–304.

[Sie32] C. L. Siegel. "Über Riemanns Nachlaß zur analytischen Zahlentheorie". In: *Quellen Studien zur Geschichte der Math. Astron. Und Phys* 1 (1932).

[M D72] P. Jarratt M. Dowell. "The "Pegasus" method for computing the root of an equation." In: *BIT* 12 (1972), pp. 503–508.

[Mon73] H. L. Montgomery. "THE PAIR CORRELATION OF ZEROS OF THE ZETA FUNCTION". In: *Analytic number theory* 24 (1973), pp. 181–193.

[Per73] I. C. Percival. "Regular and irregular spectra". In: *J. Physics Conference* 6 (1973).

[Edw74] H. M. Edwards. *Riemann's zeta function.* Academic Press (Republished by Dover Publications in 2001), 1974.

[O B84] C. Schmit O. Bohigas M. J. Giannoni. "Characterization of chaotic quantum spectra and universality of level fluctuation laws". In: *Physics Review Letter* 52 (1984), pp. 1–4.

[Ber86] M. V. Berry. "Riemann's zeta function: A model for quantum chaos?" In: *Quantum Chaos and Statistical Nuclear Physics* (1986).

[Odl87] A. M. Odlyzko. "On the Distribution of Spacings Between Zeros of the Zeta Function". In: *Mathematics of computation* 48 (1987).

[OS88] A. M. Odlyzko and A. Schönhage. "Fast algorithms for multiple evalutations of the Riemann zeta function". In: *Trans. Amer. Math. Soc.* 309 (1988).

[Odl91] A. M. Odlyzko. "The 10^{22}nd zero of the Riemann zeta function." 1991.

[A Y99] M. H. Simbel A. Y. Abdul-Magd. "Wigner surmise for high-order level spacing distributions of chaotic systems". In: *Statistical Physics Review* 60 (1999).

[M V99] J. P. Keating M. V. Berry. "The Riemann Zeros and Eigenvalue Asymptotics". In: *SIAM Review* 41 (1999), pp. 236–266.

[Ala05] Per-Olof Persson Alan Edelman. "Numerical Methods for Eigenvalue Distributions of Random Matrices". In: (2005).

[Bog07] Eugéne Bogomolny. "Riemann Zeta Function and Quantum Chaos". In: *Progress of Theoretical Physics Supplement* 166 (2007), pp. 19–36.

[Kor10] M. A. Korolev. "The Gram law and Selberg's conjecture on the distribution of zeros of the Riemann zeta -function". In: *Izv. Math* (2010).

[Dav15] David J. Fernandex C. Javier Negro David Bermudez. "Solutions to the Painlevé V equation through supersymmetric quantum mechanics". In: *J. Physics Conference* 1 (2015).