AN ALGEBRAIC EXTENSION OF THE ZETA FUNCTION

DAN NORMAND

Abstract. In the 1850's, Riemann set out to find an analytic continuation of Dirichlet's famous series

$$\sum_{n=1}^{\infty} \frac{1}{n^s}$$

and study its behaviour. He succeeded and was able to further link of the zeta function of the complex numbers to the distribution of the primes, a famous result in classical analytic number theory. With this discovery in mind, we ask a similar question: what kinds of *algebraic objects* can we develop an analogue of Dirichlet's series for? How does this new series behave over these algebraic objects? It turns out the answer is that we can define an analogous function over and ring with unity, henceforth simply a ring, one which reduces to exactly the Riemann zeta function when the ring is taken to be $\mathbb{R}$. We aim to characterize the behavior of this generalized zeta function over the modular integers, $\mathbb{Z}_n$, to understand its behavior over discrete rings for which answers can easily be computed.

1. Definition

Let $(R, +, \cdot)$ be a ring. By definition, $(R, +)$ is an abelian group. Consider the group generated by 1,

$$\langle 1, + \rangle$$

for shorthand, we call this set the *stack* and denote it by $\mathcal{S}$. We define the *invertible stack*, $\mathcal{S}^{-1}$ as the set of all r in $\mathcal{S}$ such that r has a multiplicative inverse in R. Formally, we write

$$\mathcal{S}^{-1} = \{r \in \mathcal{S} | r^{-1} \in R\}$$

Finally, we define the *inverted stack* as the set of all r in R such that r^{-1} is in $\mathcal{S}^{-1}$. Formally, we write

$$\mathcal{I} = \{r \in R | r^{-1} \in \mathcal{S}^{-1}\}$$

Here are a few examples:

(1) Let our ring be $\mathbb{Z}$. Then

$$\mathcal{S} = \{1, 2, 3, \ldots\}$$

$$\mathcal{S}^{-1} = \{1\}$$

$$\mathcal{I} = \{1\}$$

(2) Let our ring be $\mathbb{Q}$. Then

$$\mathcal{S} = \{1, 2, 3, \ldots\}$$
$$\mathcal{S}^{-1} = \{1, 2, 3, \ldots\}$$
$$\mathcal{I} = \left\{1, \frac{1}{2}, \frac{1}{3}, \ldots\right\}$$

(3) Let our ring be $\mathbb{R}$. Then

$$\mathcal{S} = \{1, 2, 3, \ldots\}$$
$$\mathcal{S}^{-1} = \{1, 2, 3, \ldots\}$$
$$\mathcal{I} = \left\{1, \frac{1}{2}, \frac{1}{3}, \ldots\right\}$$

(4) Let our ring be $\mathbb{Z}_6$. Then

$$\mathcal{S} = \mathbb{Z}_6$$
$$\mathcal{S}^{-1} = \{1, 5\}$$
$$\mathcal{I} = \{1, 5\}$$

We can now define our zeta function. Given a ring R, the *zeta function over* R, $\zeta_R(s)$ is defined for natural arguments as

$$\zeta_R(s) = \sum_{r \in \mathcal{S}^{-1}} \frac{1}{r^s} = \sum_{r \in \mathcal{I}} r^s$$

Where the summation above is understood to be under the addition operation on the ring. This follows as

$$\frac{1}{(r)^s} = \left(\frac{1}{r}\right)^s$$

that is to say, the multiplicative inverse of r to the s^{th} power is the inverse of r itself to the s^{th} power.

Using the above examples, we have that

(1)
$$\zeta_{\mathbb{Z}}(1) = 1$$

(2)
$$\zeta_{\mathbb{Q}}(1) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots$$

(3)
$$\zeta_{\mathbb{R}}(1) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots$$

(4)
$$\zeta_{\mathbb{Z}_6}(1) = 1 + 5 = 0$$

2. A Simple Primality Test

We need to be able to determine the primality of a given number fairly quickly. However, for our purposes we do not need it to be incredibly efficient, we only consider numbers $\leq 100,000,000,000$. For that reason we use the simple square root test.

Lemma: Let n be a positive integer. If $m \nmid n$ for all integers $m \leq \sqrt{n}$, then n is prime.
Proof: Clearly, if $m \nmid n$ for all integers $m \leq \sqrt{n}$ we need only worry about $m \geq \sqrt{n}$ dividing n to determine if n is prime. However, $m \geq \sqrt{n}$ divides n if and only if there is some $k \leq \sqrt{n}$ which divides n. This follows since if $m \geq \sqrt{n}$ divides n then $n = mk$ where $k \leq \sqrt{n}$ (if $k > \sqrt{n}$ then $mk > m\sqrt{n} \geq \sqrt{n}\sqrt{n} = n$, a contradiction). Hence if there is no m for all integers $m \leq \sqrt{n}$ which divides n, there is no m for all integers $m \geq \sqrt{n}$ which divides n, therefore there is no positive integer $m < n$ which divides n, hence n must be prime.

This test is useful since for $n \approx 10^{11}$ we need only perform $\approx \sqrt{10}(10^5)$ trial divisions, a significant improvement over the brute force method.

3. The Euler $\varphi(n)$ function, Euler's Theorem

As it would turn out, the Euler phi function, $\varphi(n)$, ends up playing an important role in our study of the zeta function and hence it would make sense to be sure we are familiar with both it and Euler's Theorem. The phi function counts the number of positive divisors of an integer n. Explicitly, if n has the prime factorization

$$n = p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m}$$

then

$$\varphi(n) = (p_1 - 1)p_2^{k_1 - 1}(p_2 - 1)p_2^{k_2 - 1} \cdots (p_m - 1)p_m^{k_m - 1}$$

In particular,

$$\varphi(p) = p - 1$$

for a prime p. It should be immediately apparent why we needed a primality test if we hope to use the phi function.

Alongside the phi function, Euler developed a theorem which we also make much use of.

Euler's Theorem: Given the ring $\mathbb{Z}_n$, if $a \in \mathbb{Z}_n$ has a multiplicative inverse, then

$$a^{\varphi(n)} = 1$$

Euler's Theorem is not actually stated in this form, however, this is a perfectly acceptable interpretation of it. It is not hard to see that $\mathbb{Z}_n$ is a ring, if a is a unit in $\mathbb{Z}_n$, then it has a multiplicative inverse, and if $a^{\varphi(n)} \equiv 1 \pmod{n}$, then $a^{\varphi(n)} = 1$ in $\mathbb{Z}_n$.

4. ζ over $\mathbb{Z}_n$

We finally turn our attention to how ζ behaves over $\mathbb{Z}_n$. We start out with an incredibly important results.

Theorem: $\zeta_{\mathbb{Z}_n}(s)$ is periodic with period $\varphi(n)$.
Proof: For all $r \in \mathcal{I}$, we know that r has a multiplicative inverse. Since we are taking a finite sum, we have that

$$\zeta_{\mathbb{Z}_n}(\varphi(n)+m) = \sum_{r\in\mathcal{I}} r^{\varphi(n)+m} = \sum_{r\in\mathcal{I}} r^{\varphi(n)} r^m \equiv \sum_{r\in\mathcal{I}} r^m \pmod{n} = \zeta_{\mathbb{Z}_n}(m)$$

This is a wonderful result since it tells us that we only need to check $\varphi(n)$ cases for each $\mathbb{Z}_n$; a great improvement over countably many cases.

I have developed many conjectures about how ζ behaves, but have not fully developed the proofs of these conjectures just yet. In order to better support my conjectures, naturally I must collect more data hence where this programming comes in. The general structure of the programs are as such:

First, I developed header files which define three functions, `zeta(),isPrime()`, and `phi()`. `isPrime()` takes in an integer argument and returns 0 if the argument is composite, 1 if the argument is prime. It determines this using trial devisions via the square root method defined above. `phi()` takes in an integer argument and returns $\varphi(n)$ by using `isPrime()` to determine via trial division which primes divide the argument and how many times. This returns the prime factorization of the argument in a 2-dimensional array from which φ can easily be calculated.

`zeta()` behaves a bit differently. It takes two integer arguments: the first is the *order* (so n in $\mathbb{Z}_n$) and the second is the *argument* (s in $\zeta_{\mathbb{Z}_n}(s)$). It then returns $\zeta_{\mathbb{Z}_n}(s)$. It computes this differently if n is prime or composite. If n is prime, then it simply raises each i for $1 \leq i \leq n-1$ to the s^{th} power, reducing them modulo n if they exceed $n-1$ for each exponentiation, sums them one by one and reduces modulo n. This follows since all nonzero elements of $\mathbb{Z}_n$ have inverses if n is prime.

If n is composite, then `zeta()` first must determine which elements of $\mathbb{Z}_n$ have multiplicative inverses. It does this by trial multiplication: if $a \in \mathbb{Z}_n$, then a has a multiplicative inverse if and only if there is some $b \in \mathbb{Z}_n$ such that $ab \equiv 1 \pmod{n}$. Once it has determined which elements have inverses, it then proceeds exactly as in the case where n is prime, except it only exponentiates and sums the elements which have multiplicative inverses.

There are a few problems with the `zeta()` function. First is that it does seem to have an upperbound to its arguments. It seems that it becomes unreliable past order 50,000 (namely that it returns negative values, something which *should* be

impossible). I would think this is likely due to the fact that trying to exponentiate at these sizes cause an overflow which ruins the result. This may be remedied by using prime factorizations and reducing modulo n for each squared factor, but I do not think that this would be incredibly fast for exceedingly large arguments. In view of this, it is also worth mentioning that `zeta()` slows down around order 5,000 - it takes around a second to return the calculation. By order 10,000, the slowdown is far more significant - about 6 seconds for it return the calculation. If you push the boundary and are willing to accept possible incorrect answers, it takes roughly a minute to return the calculation by order 50,000.

With these functions developed, the idea was then to collect some data to stare at. It was fairly simple to program something which would write the values to `.csv` files which I could later use. Similarly, it should not be difficult to write a program which takes in these `.cvs` files to analyze the data; looking for particular patterns I expect to see. A nicer language such as R may then be used in place of C to visual this data.

The last piece worth mentioning is that it will indeed take some time to create these `.cvs` files. As the order increases, on average the arguments zeta can take also increases. Compound this with the fact that `zeta()` takes longer to compute at higher orders, it means that the higher order I wish to examine, the longer it will take to collect the data for it. So far I have only been able to create three such files: one which showed the values of zeta for orders 3-250. This took only a minute or two to create, fairly quick. I then tried order 3-500. This took several minutes, but no more than fifteen. Then I attempted orders 3-1000 which took about 45 minutes to complete. As of this moment, I have been letting the program run for orders 1001-2000. It has been running continuously for nine hours and I do not expect it to complete for at least another day.