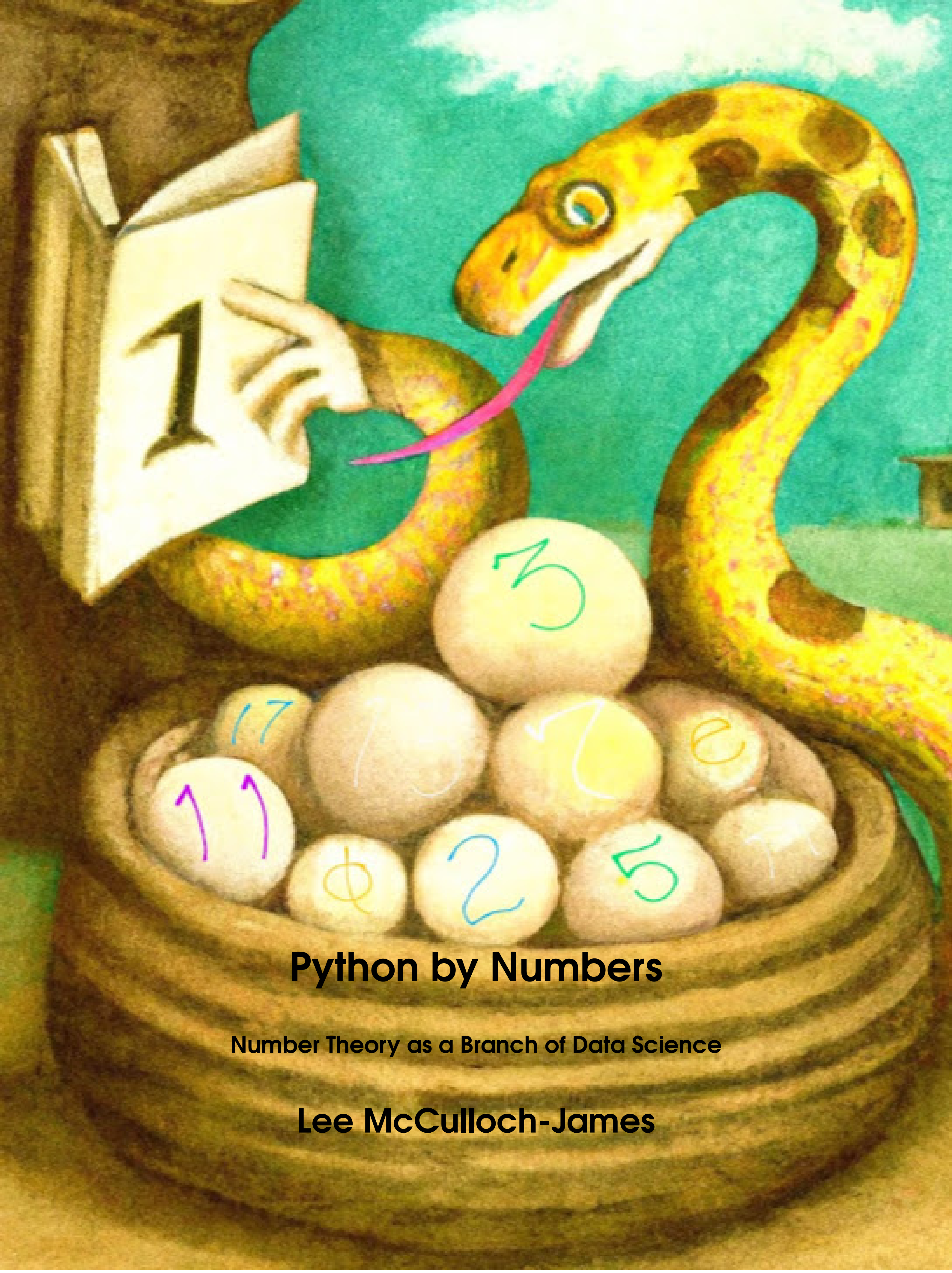


Python by Numbers

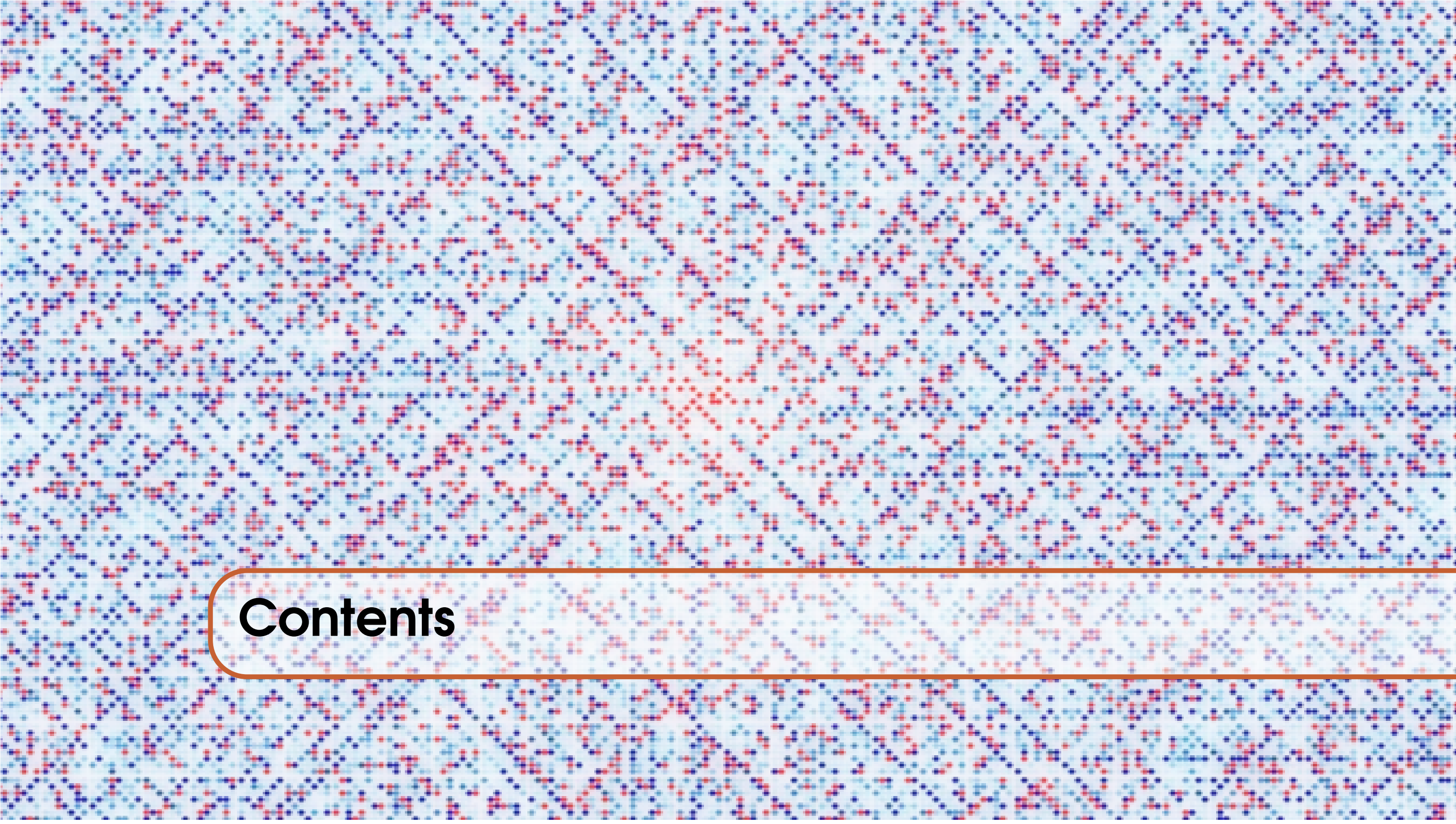
Number Theory as a Branch of Data Science

Lee McCulloch-James

PUBLISHED BY PUBLISHER

BOOK-WEBSITE.COM

First printing, March 2013



Contents

I Part One: Foundations

II Part Two: Foot hill Explorations

Part One: Foundations

1. Foreword

Trying to learn to drive at an older age, after years of being a passive passenger has echoes of my academic journey ambling as I have of late into number theory in a very pedestrian sense. As someone who had never learned to drive during my early years, it was disappointing to discover that the road, previously but a backdrop to my idle thoughts as I gazed out the window, as a driver was a canvas to express the secret garden of my worst fears.

The act of driving—a blend of technical skill, constant vigilance, and decision-making—mirrored my academic journey. Just as I found myself overwhelmed by the enormity of actually having control of a tonnage of roiling metal, my academic endeavors often saw me grappling with a confidence-sapping sense of ill-preparedness. As a PhD student reading the mathematical physics literature I frequently felt like a passenger to a supremely confident Lewis Hamilton effortlessly careering through the formulaic chicanery. My over reliance to take on trust the conceptual leaps made by an author, was pragmatic: the product of an over-accumulation of micro doubts borne of insufficient embedded foundational knowledge of fields such as analysis and Combinatorics. I reflect that I was always merely an appreciative spectator (at best a critic) of the art of the mathematical physicist, as such never a mathematical player, nor a conductor and nigh a composer.

Driving demands moment-to-moment engagement with the present, a constant readiness to respond to the unforeseen. A muscle memory equivalent is required to navigate a challenging paper: without sound foundational knowledge there reaches a breaking point when that thread of vaguely held together reasoning unravels. In both real and abstract realms, the fear of making a mistake due to a lack of foundational understanding is palpable. As a result my recent journeying through the foothills of number theory, are from the perspective of a sympathetic High school teacher, in which I am minded[1] of the importance of foundational knowledge, the value of incremental learning, and the beauty of discovering connections for yourself. The fears and uncertainties that have held me back in past incursions are gradually giving way to a newfound, if not Dunning-Kruger like confidence in my ability to control and navigate another niche realm of abstract thought.

[1] Much to my chagrin I thought that my added value would be the joining of some dots for my students. Not so.

I know now that I will not, ever drive. Instead, rather I will wait for the day when self-driving vehicles become the norm rather than the exception. My risk aversion has led me into the merciless embrace of silicon-based automation. Just as I can anticipate an era of fully self-driving cars, I find myself leaning on technological advancements in my mathematical explorations. Perhaps the better analogy now is as the co-driver to Rally driving's Sébastien Loeb, with whom we might begin to contemplate navigating the murkier terrains of number theory with Python's plotting and symbolic libraries as our map and compass. Together, crafting the pace notes, those algorithms that guide our exploration, anticipating turns and challenges with occasional aplomb. With the heavy lifting of computation and analysis offloaded, we can attend to the conceptual and creative aspects of the discipline.[2]

My leaning on coding in my latter-day studies is not a concession of defeat but an acknowledgment of these tools' potential to enhance our discovery processes. It's a recognition that sometimes, the best way to navigate complexity is not by becoming entangled in the knots of every detail but deploy a scythe to slash our way through the weeds efficiently. As we stand on the threshold of a future in which technology promises to redefine our capabilities, I am mindful that progress often comes from recognizing when to take control and when to let go. In mathematics, as in driving, there is a time for hands-on manipulation and a time for stepping back and allowing the algorithms to take the wheel. The aim of the wander that I wish to share with the reader in the foothills of number theory, is not to acquire the darkest matters of the subject in the spirit of a practitioner but rather about priming the canvas.

In embracing technology we can explore some of those otherwise inaccessible realms of mathematics with some little depth and no little insight. So, while we might not all look forward to the day when we the transport mode of choice that has sealed us off from each other is steered by an actual automaton, the more curious amongst our breed will surely enjoy the support tools that are so adept at unpicking the code within the code, that is mathematics. We will set aside the pursuit of ultimate elegance through definitive proofs, and rather look to reveal structures by heuristically suggesting by dint of depth of computation or by mere plausibility arguments. That is, in a way that welcomes both those taking their first further steps in the spirit of having read [12] to the more seasoned mathematicians who has dipped into [19]. By doing so, we will occasionally uncover some lesser-known secrets by accessible means that additionally happily serves as a focus to hone your python coding development skills.

[2] I rue how statements like "a combinatorial analysis reveals that, due to intrinsic symmetries and antisymmetries, the Riemann tensor possesses 20 independent components, while the Ricci tensor has 10 independent components in a four-dimensional spacetime" would have "then" delivered an undue load on my already stretched knowledge envelope but "now" can be revealed by a casually constructed AI prompt. See Appendix for a detailed exposition on the combinatorial analysis of the Riemann and Ricci tensors' independent components!

1.1 Prologue

An introductory chapter initially sets the stage for how our characters from Number Theory will be unpicked Python coding. How we will ideas from data science to probe Number theoretic concepts in a largely informal manner. As such we will bleed in expositions of technical concepts in a dangerously casual way, some of which will be developed further later on, others of which will serve to merely prime the reader for more formal treatments in their later undergraduate journey. Some high level topics:

1.	Composite Number Partitioning	... how to cut up a number or slice a rectangle
2.	Reptend Prime Orbit Cyclicality	... periodicity of reciprocals
3.	Semi-Pseudo Prime Number Epigraphing	... factorisation beyond the number line
4.	Aperiodic Fibonacci Tiling	... pictorial bunny birth and death
5.	Determining Primitive Triangular Congruency	... determinants of Area and Perimeter
6.	Doubly exaggerated Derrangement	... beyond the factorial!
7.	p-adic number lengths	... binary as a dyadic length metric
8.	Stratification of primes	... slicing primes in spirit of Kleiber
9.	Figurative numbers	... visiting Polygonia
10.	AbSurd Irrationality of Metallica	... golden ratio et al
11.	Giving a Toss by sticking it to Pascal	... applying hockey stick to Bernouilli trials
12.	Ensembles of Balls	... putting balls in boxes
13.	Integer Lattice Point Diophantism	... allusion to elliptic curve grandeur

Part 1: Foundations and Building Blocks

- We begin with foundational concepts in number theory- mostly arithmetic functions that will be utilized explicitly and mostly implicitly throughout the book. Fermat little theorem prime factorisation notion of Least Common Multiple and Greatest Common Divisor.
- We will touch on some nice examples early on to demonstrate the Data science inferencing we can infomally apply to these concepts.

Part 2: Exploring Number Patterns

- We will explore specific number patterns such as composite rectangular numbers, pseudo primes, and figurative numbers finding nice ways to epigraph them using Ulam's Number spiral and golden angle polar plots. Each chapter alludes to either a historical context, slightly prosaic application or a problem that has puzzled mathematicians, to motivate the subsequent data-driven exploration.
- Chapters here are on Fibonacci numbers, stratification of primes, Goldberg conjectures composite number partitioning and golden ratio Metallica.

Part 3: Advanced Topics in Number Theory and Data Science

- This section include more complex topics such as combinatorics, integer lattices, and Diophantine equations.
- here the data analysis is a little more sophisticated even if the number theory is not as we touch on graph theory although we do link Pythagorean triples to Modular functions and congruent numbers to elliptic curves.

Part 4: Special Topics and Applications

Here we discuss theories and methods to solve real-world problems or explore mathematical curiosities, such as the secretary problem, the Enigma machine, and the geometric structures under Klein's Erlangen Program.

1.2 Number Nomenclature

Numbers can be broken down and reconstructed by: *factorization* and *partitioning*:

- Factorization focuses on the **multiplicative structure** of integers.
- Partitioning focuses on their **additive structure**.

Divisors (or factors) of a number include all the integers that can divide that number without leaving a remainder, not just the prime ones. The divisors of 18 are $1, 2, 3, 6, 9, 18$ while its proper set does not include 18 itself. Some of these divisors are composite numbers themselves (such as 6 and 9). When we say a number is "abundant" or "deficient," we are referring to the (proper) sum of its divisors excluding the number itself. We can fully partition the naturals accordingly as (1.1, Amongst the abundants are the "semi-perfects" which posses a subset of divisors equal to the number itself. Perfection occurs when the proper subset of the divisors of the number sum to that number.

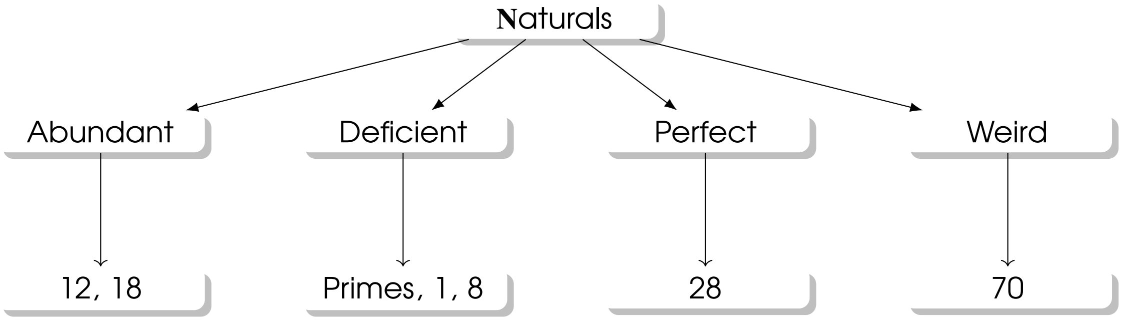


Figure 1.1: Partitions of the Naturals by abundance of divisors.

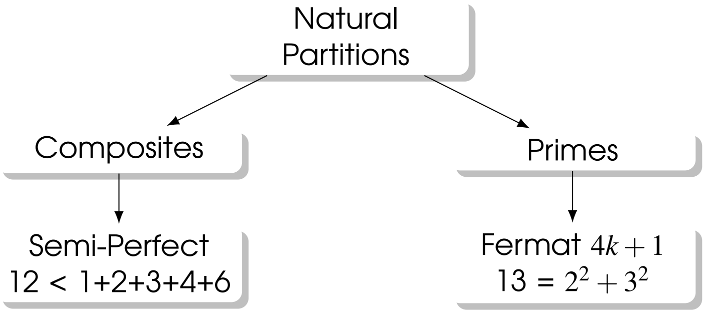


Figure 1.2: Natural Partitions into Composites and Primes, with example types.

Highly Composite Numbers are the antonyms of prime numbers and are presented as such in (1.3). Pseudo primes (only) appear on first blush to be prime, generated as they are by our false friend, Fermat's little theorem.

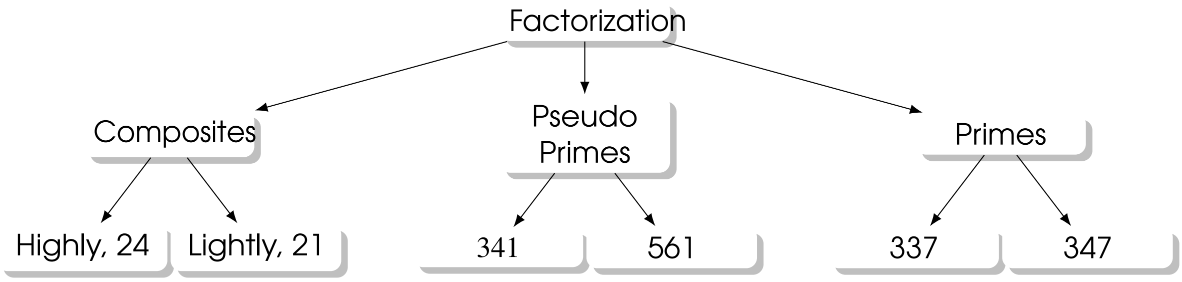


Figure 1.3: Factorization of the Naturals.

When we speak of a number being a “perfect square” or “oblong,” we’re referring to the nature of its divisors in terms of geometric shapes: a perfect square has an even power of prime factors (like 2^2, 3^2), while an oblong number is the product of two consecutive integers (like 2×3, 3×4). The organigram, (1.4) depicts the classification of composite numbers based on their Möbius function (μ) values, thus (for example) distinguishing between perfect square and ”cuboid” numbers which have respectively repeated and unrepeated primes as factor roots:

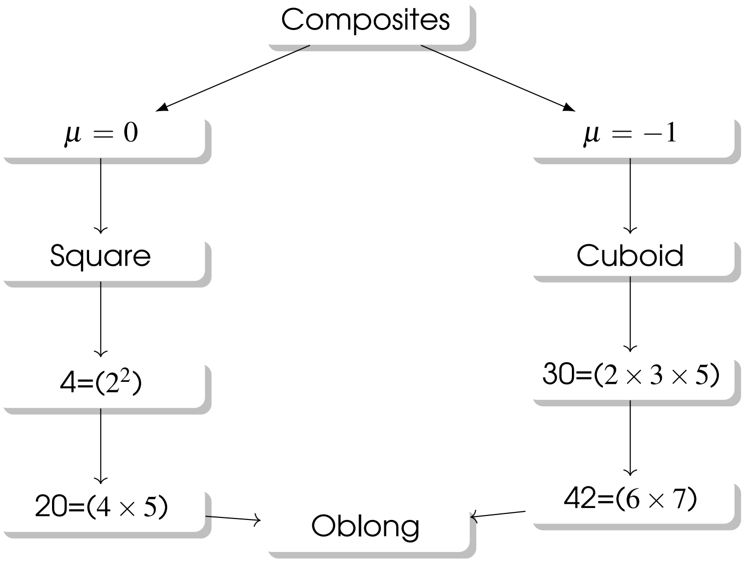


Figure 1.4: Partitions of Composites by their Möbius function.

Natural Prime numbers can be categorised based whether they are expressible as $4k+3$ or $4k+1$, where k is an integer. This subdivision is not merely algebraic but ties deeply into number theory and its applications. The organigram (1.5) illustrates this delineation of prime numbers Gaussian and non-Gaussian primes. *Reptend* primes, which generate periodic continuations in their reciprocal decimal expansions can be of both types.

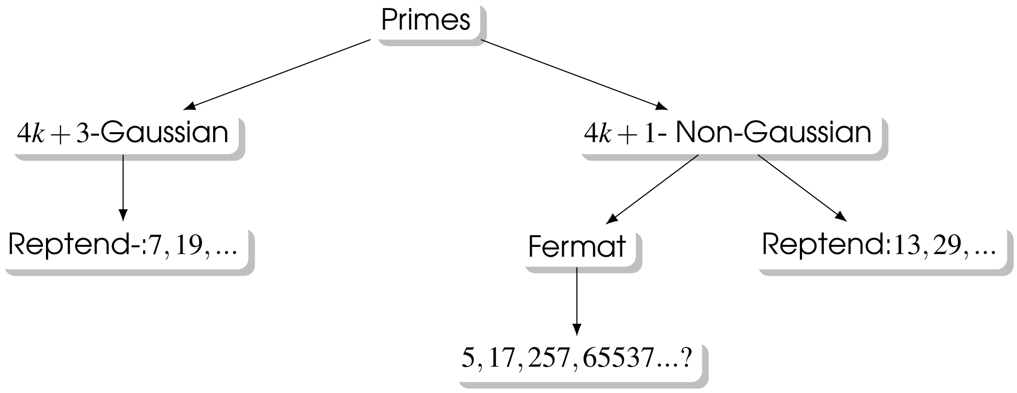


Figure 1.5: Partitions of Primes by their Gaussian Prime status.

Gaussian primes[3] are (also) primes in the set of Gaussian integers, being complex numbers of the form $a+bi$ where a and b are integers. The first few Gaussian prime integers among the natural

[3] The decomposition into factors in the Gaussian integer domain, distinguishes between the multiplicative structures of natural and Gaussian integers.

numbers are:

$$\begin{aligned}3 &= 4\times 0+3\\ 7 &= 4\times 1+3\\ 11 &= 4\times 2+3\\ 19 &= 4\times 4+3\end{aligned}$$

A **N**atural prime is a Gaussian prime only if it is of the form $4k+3$. **N**atural primes that can be factored into two non-real ”conjugate” Gaussian integers are non-Gaussian are of the form $4k+1$:

$$\begin{aligned}2 &= (1-1i)(1+1i)\\ 5 &= (1-2i)(1+2i) = 4\times 1+1 = 2^2+1^2\\ 17 &= (1-4i)(1+4i) = 4\times 4+1 = 4^2+1^2\\ 37 &= (1-6i)(1+6i) = 6\times 6+1 = 6^2+1^2\end{aligned}$$

Non-Gaussian Primes of the $4k+1$ type are to be called Fermat because they can always be expressed as the sum of two square numbers (a^2+b^2). A special subset of primes sit within the $4k+1$ Non-Gaussian primes. These Fermat[4] primes are of the form $F_k = 2^{2^k}+1$ for some non-negative integer k. Pierre de Fermat conjectured (wrongly) that all such numbers of this form are prime. As of now, Fermat primes are known to exist only for $n = 0, 1, 2, 3$, and 4, corresponding to the primes $3, 5, 17, 257$, and 65537 respectively. No other Fermat primes have been found, and it is an open question in mathematics whether any more exist.

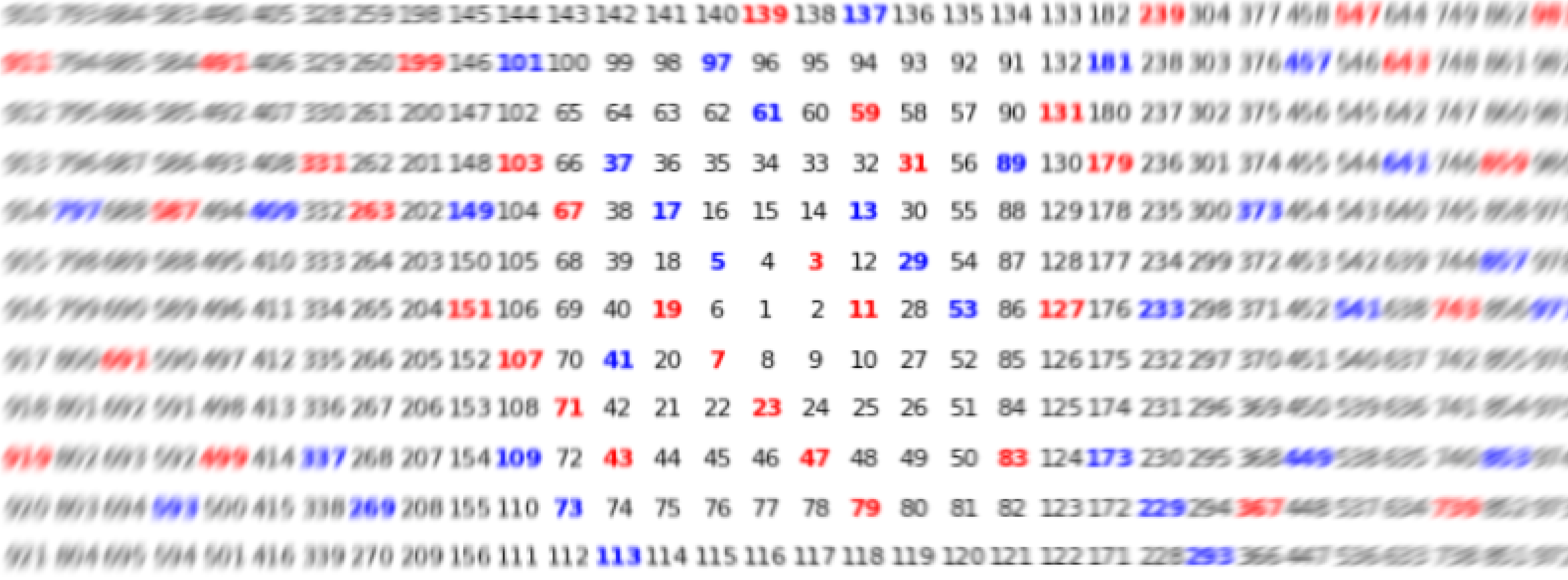
Blind Purposeful Striving

Much of the time you may have the impression you are being led down a dark alley only to face some mathematical embodiment of the abyss. What better way to pop the existential vacuousness of our blind, purposeless striving with the blind purposeful strut that our kensin and myosin proteins must surely be seeking to engender in us `https://elifesciences.org/articles/05413#abstract.`.

> “All willing springs from lack, from deficiency, and thus from suffering. Fulfillment brings this to an end; yet for one wish that is fulfilled there remain at least ten which are denied. Furthermore, the desire lasts long, the demands are infinite; the satisfaction is short and scantily measured out. But even the final satisfaction is itself an illusion; the wished-for end is only a deception. The wish fulfilled at once makes way for a new one; the former is a known delusion, the latter a delusion not as yet known.”
> — *Arthur Schopenhauer, "The World as Will and Representation."*

Fear not the calling of these numeric sirens, for of all the forms to chase or at worst to merely appreciate, these Platonic ones are surely worthy of gaining Nietzschian wilful mastery over.

[4] The allure of Fermat primes lies not only in their scarcity but also in their crucial role in the construction of regular polygons that can be constructed with a compass and straightedge. According to Gauss’s famous result, a regular polygon with p sides can be constructed in this manner if and only if p is a Fermat prime or a product of distinct Fermat primes and a power of 2.



2. Number Theory as a Data Science

> These were spectator manuals. Implicit in every line is the idea that
> 'Here is the machine, isolated in time and in space from everything else in the universe.
> It has no relationship to you, you have no relationship to it,
> other than to turn certain switches, maintain voltage levels,
> check for error conditions....' and so on."
> — *Robert M. Pirsig, Zen and the Art of Motorcycle Maintenance*[13]

This is a book of highly developed characters embroiled in a web of intricate plot and sub plots. It is not a novel novel but a novel journey into arithmetic number theory, treating it as a data science, whose secrets are to be partially unpicked by Python code. Whether you are a budding number arithmetic enthusiast, a high school teacher, that most curious of creatures that has maintained some semblance of youthful curiosity, or a Python novice eager to grasp its syntax, this book offers an opportunity to delve into and represent the pattern and structures of that most ubiquitous of languages - our number system.

Viewing the Number line as a rich data source on which to perform data science, we will explore age-old questions, using computational tools to gain insights and draw out otherwise unseen patterns in the landscape of numbers. *Data Science* is an interdisciplinary field that involves extracting insights and knowledge from data using various techniques, tools, and methodologies encompassing the collecting, processing, analyzing, and interpretation of data to derive insights and support decision-making. Data science combines skills from computer science, statistics, mathematics, domain expertise, and data visualization to address complex problems and uncover patterns, trends, and correlations in large and diverse data sets.

We will undertake coding projects, employing Python[1], sedately and mostly informally reveal some fundamental number theoretic concepts by treating the number line as a large data set, create visualizations, and apply statistical techniques to navigate through some delicacies from discrete

[1] *python* is a beginner-friendly, widely used, high level programming language renowned for its simplicity and versatility that will allow us to implement algorithms to outline conjectures and results with some efficiency and clarity.

Mathematics, unveiling some connections and numerical relationships. This book is aimed at anyone with a preference to take a sejourn through elementary algebraic number theory through the lens of data science rather than tackling its formal structures head on. Whether you are a student about to step into higher education, a high school teacher seeking engaging mathematical resources, or an inquisitive individual eager to rekindle your passion for numbers, this book is supposed to offer a unique blend of theory, computation, and practical applications. The spirit of the book is twofold:

- use pictures to support a conjecture or generate a natural question;
- develop some programming skills along the way to support our enquiries.

Google Colab coding platform

Google Colab provides a versatile and powerful platform for data science and machine learning projects. being an online platform that provides a Jupyter notebook environment for interactive computing and data analysis. As it allows users to write and execute Python code in the browser with no configuration required, it provides access to powerful computing resources, and an easy means for sharing the code. These will be shared as colab notebook .ipynyb files. Colab also allow users to import libraries and dependencies necessary for their code to run:

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

Users write Python code in cells and each cell can be executed independently. The code snippet:

```
def compute_square(x):
    return x * x
square_of_2 = compute_square(2)
print(square_of_2)
```

follows our general construction template, with functions like `compute_square` declared before being called in the main thread of the code with or without local variables.

- `x` is a *local* variable within the `compute_square` function.
- `square_of_2` is a *global* variable, assigned the return value of `compute_square(2)`.

After writing the code, users execute the cell by clicking the play button or pressing Ctrl+Enter and the code will run in the Google Colab servers, and the output including print statements, plots, and other visualizations, will be displayed directly under the code cell.

Let's first prime our starting route with some vague introductory remarks:

Definition 1 *Number theory* is a branch of pure mathematics that primarily deals with the properties, relationships, and patterns of integers (whole numbers) and their various properties. Key areas of focus within number theory that we will look at include:

Prime numbers: as natural numbers greater than 1 that have no divisors other than 1 and themselves, Number theory explores how these are distributed among natural numbers, positing prime number theorems and conjectures.

- **Divisibility**: examines how one integer is said to divide another if the quotient is also an integer.
- **Congruence**: Modular arithmetic deals with remainders when dividing integers and has applications in cryptography and computer science.
- **Diophantine equations**: involves finding integer solutions for polynomial equations with multiple variables. A famous example is that of Fermat's Last Theorem.
- **Continued fractions**: which represent real numbers in an elegant way using a sequence of partial fractions.

I will occasionally highlight a key python coding snippet used in the course of expositions like the following functions that implement fundamental algorithms (inefficiently) related to prime numbers. By utilizing the Sieve of Eratosthenes principle in modified form prime numbers are generated by:

```
def generate_primes(limit):
    primes = []
    for num in range(2, limit):
        is_prime = True
        for div in primes
            if div * div > num:
                break
            if num % div == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(num)
    return primes
```

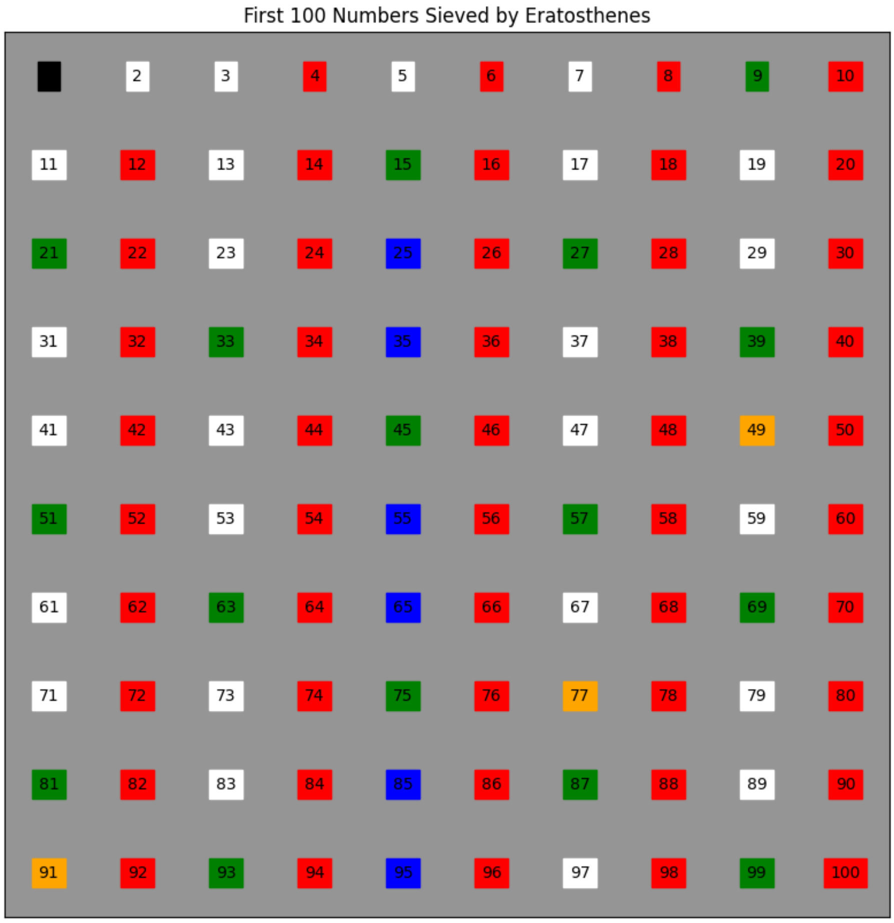


Figure 2.1: Sieve of Eratosthenes.

Key Operations:

- Uses a nested for loop to iterate through numbers up to the given limit to check divisibility by all previously found prime numbers.

- Employs a square root optimization (`div * div > num`) to reduce the number of checks needed to determine if the current number is prime which is crucial because if n is not divisible by any prime number less than or equal to $\sqrt{n}$, then n is prime.
- Appends the number to the list of primes if it is not divisible by any of primes found so far.

Prime factors of a given number n, are calculated and returned in a list by:

```
def prime_factors(n):
    factors = []
    i = 2
    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)
    if n > 1:
        factors.append(n)
    return factors
```

Key Operations:

- Starts with smallest prime, 2, and iteratively checks divisibility of n by successive integers.
- If n is divisible by i, i is added to the list of factors, and n is divided by i. This step is repeated until n is no longer divisible by i, at which point i is incremented. `n % i` is used to check if the value of variable n is divisible by the value of variable i without any remainder. The % operator returns the remainder of the division operation between n and i. If `n % i == 0` evaluates to True, it means that n is evenly divisible by i.
- This process uses a while loop that continues as long as $i^2 \leq n$, leveraging the fact that if n is not divisible by any number less than or equal to its square root, then n must be prime.
- If after this process n is greater than 1, n itself is a prime factor and is added to the list of factors. This condition handles cases where n is a prime number or the remaining n after division is a prime number.

2.1 Modular Arithmetic

Definition 2 *Digital Root* Digital Roots are derived from a number by the process of iteratively summing its digits until a single-digit number is obtained. Formally, the digital root of a non-negative integer is the value obtained after the iterative process of summing its digits. For instance, the digital root of 942 is calculated as $9+4+2=15$, and then $1+5=6$.

The code multiplicationTableDigitalRoots.ipynb delivers the following table.

n/m	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
2	2	4	6	8	10	12	14	16	2	4	6	8	1	3	5	7
3	3	6	9	12	15	18	21	24	3	6	9	3	6	9	3	6
4	4	8	12	16	20	24	28	32	4	8	3	7	2	6	1	5
5	5	10	15	20	25	30	35	40	5	1	6	2	7	3	8	4
6	6	12	18	24	30	36	42	48	6	3	9	6	3	9	6	3
7	7	14	21	28	35	42	49	56	7	5	3	1	8	6	4	2
8	8	16	24	32	40	48	56	64	8	7	6	5	4	3	2	1
9	9	18	27	36	45	54	63	72	9	9	9	9	9	9	9	9
10	10	20	30	40	50	60	70	80	1	2	3	4	5	6	7	8
11	11	22	33	44	55	66	77	88	2	4	6	8	1	3	5	7
12	12	24	36	48	60	72	84	96	3	6	9	3	6	9	3	6
13	13	26	39	52	65	78	91	104	4	8	3	7	2	6	1	5
14	14	28	42	56	70	84	98	112	5	1	6	2	7	3	8	4
15	15	30	45	60	75	90	105	120	6	3	9	6	3	9	6	3
16	16	32	48	64	80	96	112	128	7	5	3	1	8	6	4	2
17	17	34	51	68	85	102	119	136	8	7	6	5	4	3	2	1
18	18	36	54	72	90	108	126	144	9	9	9	9	9	9	9	9

Equivalence with Modular Arithmetic

A digital root is just modular arithmetic, specifically modulo 9. The digital root of a number being congruent to the number modulo 9, with an exception for numbers divisible by 9. Symbolically, for any positive integer n:

$$\text{digital root of } n = (n \mod 9) + (1 - \frac{n \mod 9}{9})$$

We will use this digital signature to investigate by coding the periodicity of various power sequences such as $2^n, 3^n$.

The rhythm of the periods of digital roots and modular results of these powers provide insights into the underlying mathematical properties and behaviors of numbers and sequences.

2.1.1 Clock Arithmetic and Encoding

Modular arithmetic, or clock arithmetic, is a powerful tool for analyzing and interpreting complex sequences, such as power series. By applying this method, we can distill intricate patterns and behaviors within sequences into more manageable forms. The exploration of periodicity and patterns in modular arithmetic provides crucial insights for creating robust encryption algorithms, ensuring secure and reliable data protection mechanisms. In cryptography, modular arithmetic is central to public key cryptography algorithms, including ElGamal and RSA (which we will explore a little more later) encryption systems. For now we will just look at the Digital Roots (DR) of the powers of bases, ranging from 2 to 200 and observe the inherent periodicity in these roots, revealing fundamental patterns that underpin the behavior of numbers in their exponential forms. Given a

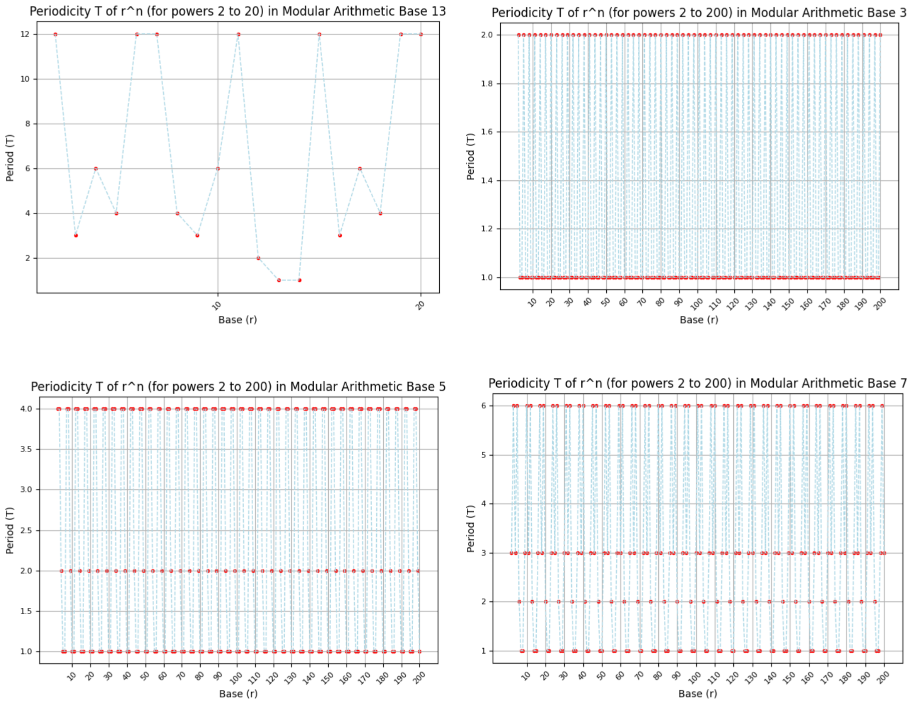


Figure 2.2: Periodicity of power series according to base 3,5 and 7

base r in the range from 2 to 200, we will compute the Digital Root (DR) of its power. For instance, for $r = 3$ and an exponent of 6, we have:

$$\mathrm{DR}[3^6 = 729] \rightarrow 9+2+7 = 18 \rightarrow 1+8 = 9$$

Periodicity of the Periods

Each base r has an associated period T, denoting the interval over which its DR repeats. The periods T for various bases r are enumerated as follows:

$T_{\mathrm{DR}(2^n)} = 6,\ T_{\mathrm{DR}(3^n)} = 1,\ T_{\mathrm{DR}(5^n)} = 6,\ T_{\mathrm{DR}(7^n)} = 3$

The period of these periods, T, also exhibits a periodic nature, fundamentally aligned to the digit 9. The sequence of periods manifests as:

$\{6,1,3,6,1,3,2,1,1\}$

This recurrent sequence encapsulates the inherent rhythm within the digital roots of the powers of bases. The exploration of digital roots and their associated periods reveals intricate patterns . This study sheds light on the hidden rhythmic structures within the universe of numbers, offering novel insights and perspectives for further mathematical exploration and analysis.

What questions do these numbers conjure? The periodicity of all n-power series $2^n, 3^n, \ldots$ mod prime number base always end in 2 ,1, 1 why?[2]

[2] It's great to observe patterns as they can be the stepping stones towards deeper mathematical truths, but they do not constitute a proof on their own.

Number	Values
2	1, 1
3	2, 1, 1
5	4, 4, 2, 1, 1
7	3, 6, 3, 6, 2, 1, 1
11	10, 5, 5, 5, 10, 10, 10, 5, 2, 1, 1
13	12, 3, 6, 4, 12, 12, 4, 3, 6, 12, 2, 1, 1
17	8, 16, 4, 16, 16, 16, 8, 8, 16, 16, 16, 4, 16, 8, 2, 1, 1
19	18, 18, 9, 9, 9, 3, 6, 9, 18, 3, 6, 18, 18, 18, 9, 9, 2, 1, 1

Table 2.1: non repeating periodicity strings for base prime, p

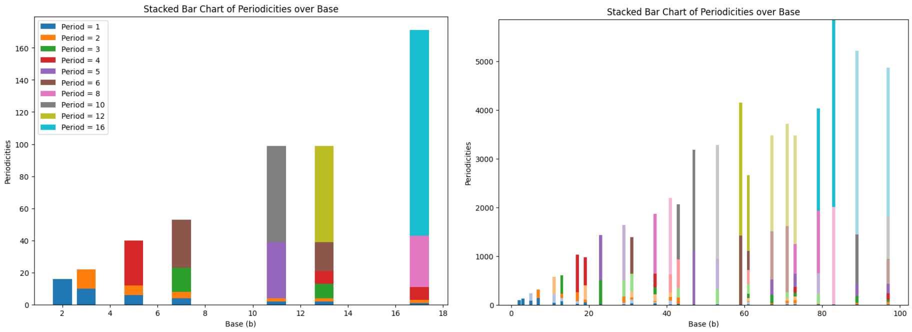


Figure 2.3: Stack charts pictorially representing the rhythms in the table above.

2.2 Arithmetic Functions and Algorithms

In number theory, several important arithmetic functions arise in the study of integers. We introduce four such functions: the number of divisors $\tau(n)$, and the sum of divisors, $\sigma(n)$ functions, Euler's totient, $\phi(n)$ and Möbius, $\mu(n)$ functions.

2.2.1 Divisor, $p|n$

Given two integers n and p, we say that p divides n (denoted as $p|n$) if there exists an integer c such that $n = p \cdot c$. In this case, p is called a divisor of n, and n is called a multiple of p. Consider $p = 5$ and $n = 30$ so that $n = p \cdot c = 5 \cdot 6$ and we say 5 divides 30, denoted as $5|30$, making 5 a divisor of 30, and 30 a multiple of 5.

1. n_p (Number of Prime Factors):is often referred to as the "number of prime divisors" or "number of distinct prime factors" of positive integer, n.
2. r_p (Reduced Number of Prime Factors): represents the count of distinct prime factors of **n** and is sometimes called the "radical" of n.
3. f (Number of Distinct Factors): (or divisors) is typically referred to as the "number of divisors" or "number of factors" of n.

2.2.2 Number and Sum of Divisors Functions $\tau(n)$, $\sigma(n)$

The number of divisors function, denoted $\tau(n)$, gives the number of positive divisors of n, while the sum of divisors function, denoted $\sigma(n)$, calculates the sum of all positive divisors of n:

- $\tau(12) = 6$ since its divisors are 1, 2, 3, 4, 6, 12, and so $\sigma(12) = 1+2+3+4+6+12 = 28$.
- $\tau(14) = 4$ since its divisors are 1, 2, 7, 14 and so $\sigma(14) = 1+2+7+14 = 24$.

The class of numbers, known as *perfect numbers* is succinctly defined in terms $\sigma(n)$. A number n is perfect if the sum of its positive divisors (excluding itself) is equal to n, or equivalently, $\sigma(n) = 2n$. So that the first perfect number is 6 because $1+2+3 = 6$, and $\sigma(6) = 1+2+3+6 = 12 = 2\times 6$.

2.2.3 Euclid's Algorithm for finding the GCD of Multiple Numbers

Euclid's algorithm is a method for finding the greatest common divisor (gcd) of two or more numbers. Let's consider finding the gcd of four numbers: a, b, c, and d. We'll denote the gcd of these numbers as gcd(a, b, c, d). The key idea behind Euclid's algorithm is that gcd(a, b, c, d) is equal to gcd(gcd(a, b), c, d). In other words, we can find the gcd of multiple numbers by iteratively finding the gcd of pairs of numbers. A Geogbra implementation by David Wees is here. To illustrate this, let's find the gcd of the numbers $a = 36$, $b = 24$, $c = 54$, and $d = 27$.

$$\begin{aligned}
\gcd(a,b,c,d) &= \gcd(\gcd(a,b),c,d)\\
&= \gcd(\gcd(36,24),54,27)\\
&= \gcd(12,54,27)\\
&= \gcd(\gcd(12,54),27)\\
&= \gcd(6,27)\\
&= \gcd(\gcd(6,27))\\
&= \gcd(3)\\
&= 3
\end{aligned}$$

Therefore, gcd(36, 24, 54, 27) is equal to 3. Euclid's algorithm can be applied iteratively as in the code GCD-iterate.ipynb until we reach a point where we have only one number left, which is then the gcd of the original set of numbers.

```
def find_gcd(numbers):
    gcd_result = numbers[0]
    for num in numbers[1:]:
        gcd_result = math.gcd(gcd_result, num)
    return gcd_result
```

- `find_gcd` starts by initializing `gcd_result` with the first number in the list.
- It then iterates through the rest of the numbers in the list (starting from the second number).
- For each number, it updates `gcd_result` by computing the GCD of the current `gcd_result` and the number using the `math.gcd` function.
- After processing all the numbers, returns `gcd_result`, (GCD of all numbers in the list).

`find_gcd` is called and user inputs are taken and controlled in the following snippet.

```
    n = int(input("Enter the number of numbers: "))
    numbers = []
    for i in range(n):
        number = int(input(f"Enter number {i+1}: "))
        numbers.append(number)
    gcd_result = find_gcd(numbers)
```

2.2.4 Euler's Totient Function $\phi(n)$

Euler's totient function $\phi(n)$ counts the positive integers up to a given integer n that are relatively prime (*co-prime*) to m. So for example for the number 12, we identify all numbers less than it that do not share any common factors with 12 other than 1. That is, given its prime factorization, $12 = 2^2 \times 3$ and the numbers less than 12 are $1, 2, 3, \ldots, 11$ we find the gcd of each of these numbers with 12:

$$\begin{aligned}
&\gcd(1,12) = 1, \text{ coprime,}\\
&\gcd(2,12) = 2, \text{ not coprime,}\\
&\gcd(3,12) = 3, \text{ not coprime,}\\
&\gcd(4,12) = 4, \text{ not coprime,}\\
&\gcd(5,12) = 1, \text{ coprime,}\\
&\vdots\\
&\gcd(11,12) = 1, \text{ coprime,}
\end{aligned}$$

thus identifying the coprimes to 12 as 1, 5, 7, and 11, and $\phi(12) = 4$, as they are the only numbers less than it whose gcd with 12 is 1. For a prime number p, the value of $\phi(p)$ is $p-1$, because all integers less than p are relatively prime to p. For a composite number n with a prime decomposition $n = p_1^{e_1} p_2^{e_2} p_3^{e_3} \cdots$, where p_i are distinct prime numbers and e_i are their respective exponents, the totient function can be derived using the principle of inclusion-exclusion[3] of combinatorics. The key observation is that the fraction of numbers less than n and not relatively prime to n can be expressed in terms of the fractions of numbers divisible by each of the prime factors of n. Specifically, $\phi(n)$ can be calculated as:

$$\phi(n) = n\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right)\left(1 - \frac{1}{p_3}\right)\cdots$$

which arises because for each prime p_i dividing n, $\frac{1}{p_i}$ of the numbers less than n are divisible by p_i and hence not relatively prime to n. Subtracting these fractions from 1 gives the fraction of numbers that are relatively prime to n, and multiplying by n gives the count of such numbers. Consider again our example, $n = 12$, which has a prime decomposition of $2^2 \times 3$ and thus according to the formula:

$$\phi(12) = 12\left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{3}\right) = 12 \times \frac{1}{2} \times \frac{2}{3} = 4,$$

there are 4 integers up to 12 that are relatively prime to 12. In a little more detail, we write the

[3] This principle corrects for the fact that when we add the sizes of individual sets, elements that are common to multiple sets get counted more than once. So suppose we have two sets, A and B. The principle states that to find the number of elements in the union of A and B (denoted $A \cup B$), we must add the number of elements in A and B and then subtract the number of elements that are in both A and B (denoted $A \cap B$):$|A \cup B| = |A| + |B| - |A \cap B|$

- Let A be a set of people who like apples, and suppose there are 20 people in set A.
- Let B be a set of people who like bananas, and suppose there are 15 people in set B.
- Suppose 5 people like both apples and bananas and so are counted in both set A and set B.

Applying the inclusion-exclusion principle, the number of people who like either apples or bananas or both is:

$$|A \cup B| = |A| + |B| - |A \cap B| = 20 + 15 - 5 = 30$$

.

totient as a product taken over all distinct prime divisors p of n.:

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

Consider the prime factorization of 54 as 2×3^3 so that we have:

$$\phi(54) = 54 \left(1 - \frac{1}{2}\right) \left(1 - \frac{1}{3}\right).$$

Looking at this in terms of the fractions of numbers up to 54 that are not co-prime to it due to each prime factor and their combinations we have:

$$\phi(54) = 54 \left(1 - \left(\frac{1}{2} + \frac{1}{3}\right) + \frac{1}{2 \times 3}\right) = 54 \left(1 - \frac{1}{2} - \frac{1}{3} + \frac{1}{6}\right) = 18$$

where:

- $-\frac{1}{2}$ accounts for numbers divisible by 2,
- $-\frac{1}{3}$ accounts for numbers divisible by 3,
- $+\frac{1}{2\times3}$ corrects for the over-counting of numbers divisible by both 2 and 3 (subtracted twice).

The inclusion-exclusion principle thus gives an alternating sum:

$$\phi(n) = n \left(\sum_r \frac{1}{p_r} - \sum_{r>s} \frac{1}{p_r p_s} + \sum_{r>s>t} \frac{1}{p_r p_s p_t} - \dots \right)$$

To find the totient function $\phi(p^j)$ for a prime power p^j where p is a prime number and j is a positive integer, just now consider that:

1. **Numbers Not Relatively Prime to p^j:** The numbers that are not relatively prime to p^j are the multiples of p within the range from 1 to p^j:

   $$p, 2p, 3p, \dots, p^{j-1}p$$

   So in total, there are p^{j-1} such numbers.
2. **Total Numbers from 1 to p^j:** There are p^j numbers in total.
3. **Numbers Relatively Prime to p^j:** To count the numbers that are relatively prime to p^j, subtract the count of numbers not relatively prime to p^j from the total count of numbers:

   $$\phi(p^j) = p^j - p^{j-1}$$

Hence, $\phi(p^j) = p^j - p^{j-1}$, saying that there are $p^j - p^{j-1}$ numbers that are relatively prime to p^j.

Totient Function Dot Plot

The dot plot (2.4) presents a dot corresponding to a number's coprime and the Totient function $\phi(m)$ as a line graph, counting co-primes for each number m.

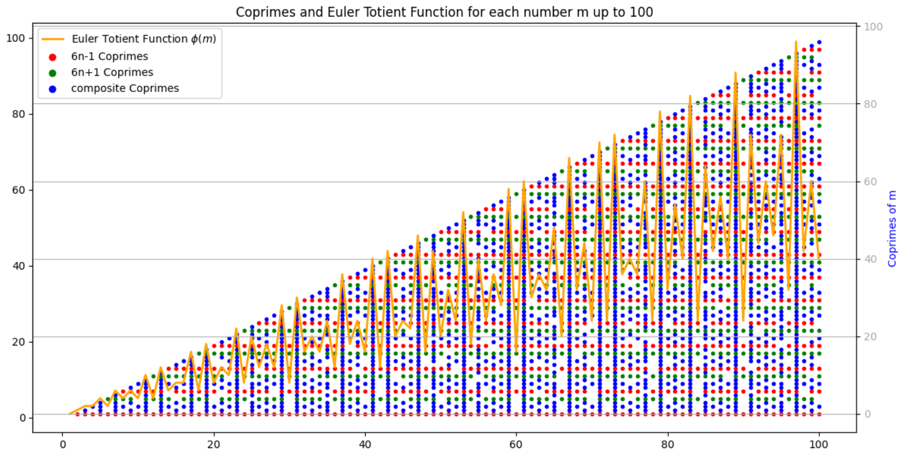


Figure 2.4: Euler totient function with dot plot of coprimes.

`plot_coprimes_above_totient`, below creates a visualization that represents the coprimes of each integer up to a specified limit, `up_to_m` overlaying the coprimes on top of $\phi(m)$, to provide a comparative view between the individual coprimes and the count of coprimes for each integer.

```
def plot_coprimes_above_totient(up_to_m):
    plt.figure(figsize=(14, 7))
    for m in range(1, up_to_m + 1):
        coprimes = find_coprimes(m)
        for coprime in coprimes:
            color = ’blue’
            if (coprime - 1) % 6 == 0:
                color = ’red’   # Color for 6n-1 coprimes
            elif (coprime + 1) % 6 == 0:
                color = ’green’  # Color for 6n+1 coprimes
            plt.plot(m, coprime, ’o’, color=color, markersize=3)
    m_values = np.arange(1, up_to_m + 1)
    phi_values = np.array([len(find_coprimes(m)) for m in m_values])
```

Below is a breakdown of the function’s operations:

1. The function initializes a plot with a specified figure size.
2. For each integer m in the range from 1 to `up_to_m`, it:
   (a) Calls the function `find_coprimes(m)` to retrieve list of integers that are coprime to m.
   (b) Iterate over the list of coprimes and plot each coprime as dot on the plot, where:
      - Red dots represent coprimes of the form $6n-1$.
      - Green dots represent coprimes of the form $6n+1$.
      - Blue dots represent other coprimes.
   (c) Adjust the size of the markers for better visibility.
3. Generate an array of integers, `m_values`, from 1 to `up_to_m`.
4. Calculate $\phi(m)$ for each integer m in `m_values`, creating an array of `phi_values`.
5. Overlay the `phi_values` on the same plot.

The following segment of the `plot_coprimes_above_totient` function overlays the Euler Totient function, $\phi(m)$, on the plot that initially represents coprimes for each integer m. A second y-axis, `ax2`, is created using the `twinx()` method allowing for the representation of the Euler Totient function $\phi(m)$ on a different scale.

```
ax2 = plt.gca().twinx()
ax2.plot(m_values, phi_values, color='orange', linewidth=2, zorder=1)
ax2.set_ylabel('$\phi(m)$', color='darkgrey')
ax2.tick_params(axis='y', labelcolor='darkgrey')
ax2.legend(loc='upper right')
```

The key operations performed in this segment are as follows:

1. The `ax2.plot` method is used to plot the values of $\phi(m)$ against m. The line representing $\phi(m)$ is set to have an orange color, a linewidth of 2, and a z-order of 1, ensuring that it is drawn on top of other elements in the plot.
2. The y-axis label is set to '$\phi(m)$', and its color is set to dark grey using the `set_ylabel` and `tick_params` methods.
3. A legend is added positioned at the upper right of the plot area.

This segment of the function ensures that the Euler Totient function $\phi(m)$ is clearly represented and distinguished on the plot, providing an insightful comparison between the count of coprimes for each integer and the individual coprimes themselves.

Möbius Function $\mu(n)$

The Möbius function, $\mu(n)$ is defined for all positive integers n as follows:

$$\mu(n) = \begin{cases} 1 & \text{if } n = 1, \\ 0 & \text{if } n \text{ has a prime factor } p \text{ with } p^2|n, \\ (-1)^k & \text{if } n \text{ is a product of } k \text{ distinct prime numbers.} \end{cases}$$

$\mu(n)$ plays a crucial role in analytic number theory and combinatorics and we note its key properties with examples as:

- $\mu(n) = 0$ if n has any prime factor raised to a power greater than 1,
- $\mu(n) = (-1)^k$ if n is a product of k non repeating prime factors.
- $\mu(12) = 0$ because $12 = 2^2 \times 3$ has a squared prime factor.
- $\mu(14) = -1$ since 14 is the product of two distinct prime numbers (2 and 7).
- $\mu(30) = -1$ because $30 = 2 \times 3 \times 5$ is a product of $k = 3$ distinct prime numbers.
- $\mu(54) = 0$ because $54 = 3^3 \times 2$ has a prime factor (3) raised to a power greater than 1.

We will come to think of numbers like 30 as a kind of (hyper)-cuboid in the space of its prime factors, given they have no "square" sections, (no repeated prime dimension) and with a Möbius function equal to negative 1. Here, "square" refers to any prime factor raised to a power greater than 1, not just a power of 2 or any even power.

Table 2.2 lists *oblong* numbers of the form $n = m \times (m-1)$, ranging from $n = 30$ to $n = 90$. For each number, the table provides values for Euler's totient function $\phi(n)$, the number of divisors function $\tau(n)$, the Möbius function $\mu(n)$, and lists all possible cuboid representations based on their distinct factors.

n	$\phi(n)$	$\tau(n)$	$\mu(n)$	**Cuboid Possibilities**
20	8	6	0	$20 = 2 \times 2 \times 5$, $20 = 1 \times 4 \times 5$
30	8	8	-1	$30 = 2 \times 3 \times 5$, $30 = 1 \times 6 \times 5$
42	12	8	1	$42 = 2 \times 3 \times 7$, $42 = 1 \times 6 \times 7$
56	24	8	0	$56 = 2 \times 2 \times 14$, $56 = 1 \times 8 \times 7$
72	24	12	0	$72 = 2 \times 2 \times 18$, $72 = 2 \times 3 \times 12$, $72 = 1 \times 8 \times 9$
90	24	12	0	$90 = 2 \times 3 \times 15$, $90 = 2 \times 5 \times 9$, $90 = 1 \times 10 \times 9$

Table 2.2: Arithmetic functions associated to the Oblong Numbers

2.2.5 Least Common Multiple

The Least Common Multiple of the set of integers $\{1, 2, \ldots, n\}$ plays a significant role in number theory and combinatorics.

Definition 3 *Least Common Multiple* ,(LCM) of the integers $a_1, a_2, \ldots, a_n$, denoted $\mathrm{LCM}[a_1, a_2, \ldots, a_n]$, is the smallest positive integer m such that m is a multiple of each a_i, where $1 \leq i \leq n$:

$$\mathrm{LCM}[a_1, a_2, \ldots, a_n] = \min\{m \in \mathbb{N} : m \geq \max(|a_1|, |a_2|, \ldots, |a_n|) \text{ and } \forall i, a_i | m\}$$

where $a_i|m$ denotes that a_i is a divisor of m, and $\mathbb{N}$ represents the set of natural numbers.

The natural logarithm of the LCM of these integers provides insights into their collective divisibility properties and connects to deep results in analytic number theory, including those related to Apéry's work on the irrationality of $\zeta(3)$. While our discussion is not directly related to Apéry's theorem[4], the approach of examining properties of numbers through their prime factorization and logarithmic characteristics is in the spirit of his work.

[4] Apéry's contributions to number theory, especially his proof of the irrationality of $\zeta(3)$, allude to distributional intricacy amongst the integers and primes. For a fuller discussion The Irrationals, [5] is an irrational romp.

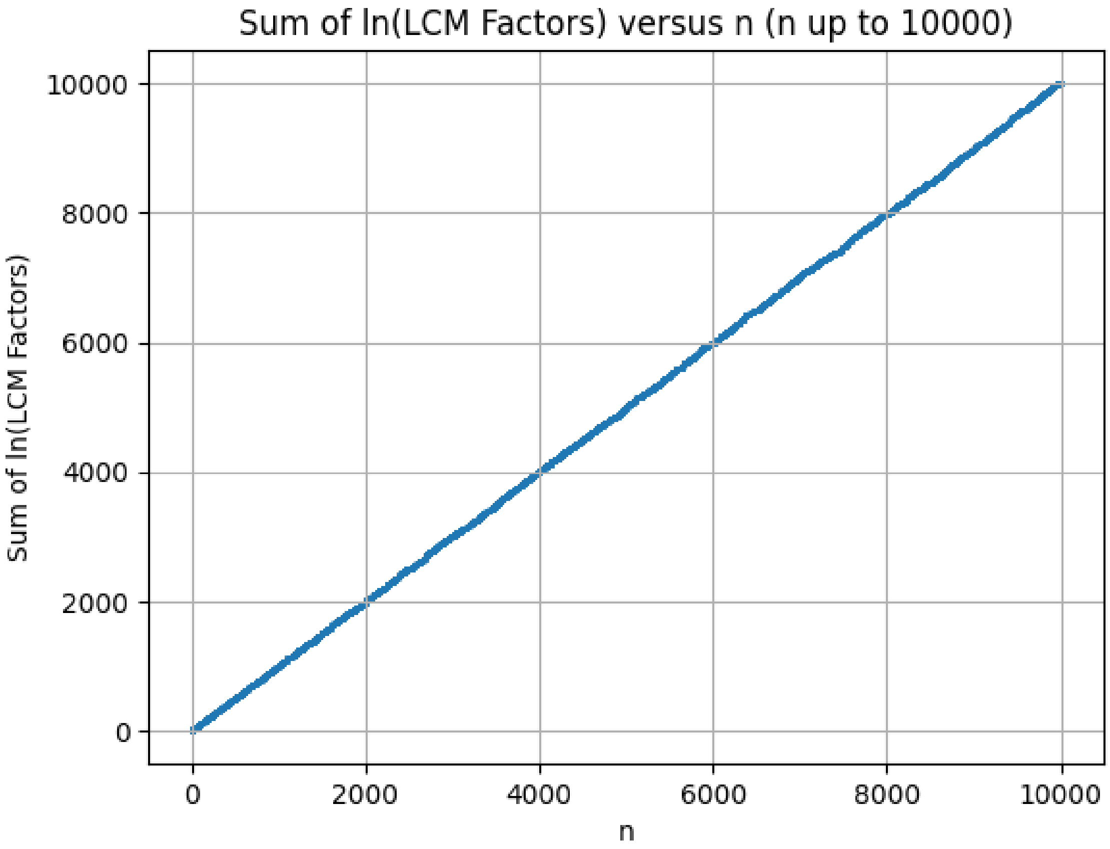


Figure 2.5: Log of LCM of 1 to n versus n

The scatter plot of $\ln(\text{LCM}([1,2,\ldots,n]))$ versus n provides demonstrates, if nothing else, the effectiveness of computational techniques in handling large-scale numerical problems offering a visualization of how the growth of combined divisibility of the first n integers is of an exponential nature.

Calculation of $\ln(\textbf{LCM})$

Direct computation of the LCM for a large set of numbers can quickly lead to very large integers, posing challenges for standard numerical operations.

To manage these large calculations, utilizes the sum of the natural logarithms of prime factors raised to their highest powers not exceeding n, thus avoiding the direct computation of the LCM.:

$$\ln(\text{LCM}([1,2,\ldots,n])) = \sum_{p\leq n} \left\lfloor \frac{\ln n}{\ln p} \right\rfloor \ln p \tag{2.1}$$

where p are prime numbers. While the Fundamental Theorem of Arithmetic states that every integer greater than 1 can be uniquely factorized into prime numbers up to ordering and the logarithmic property that the logarithm of a product equals the sum of the logarithms of the factors ($\log(ab) = \log(a) + \log(b)$) is instrumental in transforming the product of prime factors into a sum, facilitating computation.

The method for calculating $\ln(\text{LCM}([1,2,...,n]))$ involves:

1. Enumerating all primes up to n.
2. For each prime p, determining the highest power m such that $p^m \leq n$.

3. Summing the natural logarithms of each prime raised to its maximum power, utilizing the property $\log(p^m) = m\log(p)$.

This approach yields:

$$\ln(\mathrm{LCM}([1,2,...,n])) = \sum_{p\leq n} m_p \ln(p)$$

where m_p is the maximum power of each prime p not exceeding n. By converting the LCM's prime factorization into a sum of logarithms, this method:

- Avoids numerical overflow, enabling the handling of large n.
- Improves computational efficiency compared to direct LCM calculation.

LCM Code Snippet

```
def ln_lcm_sum(n):
    primes = list(primerange(1, n+1))
    ln_sum = 0
    for p in primes:
        m = 1
        while p**m <= n:
            m += 1
        ln_sum += np.log(p**(m-1))
    return ln_sum

def ln_lcm_sequence_sum(max_n):
    n_values = np.arange(1, max_n + 1)
    ln_lcm_values = [ln_lcm_sum(n) for n in n_values]
    return n_values, ln_lcm_values
```

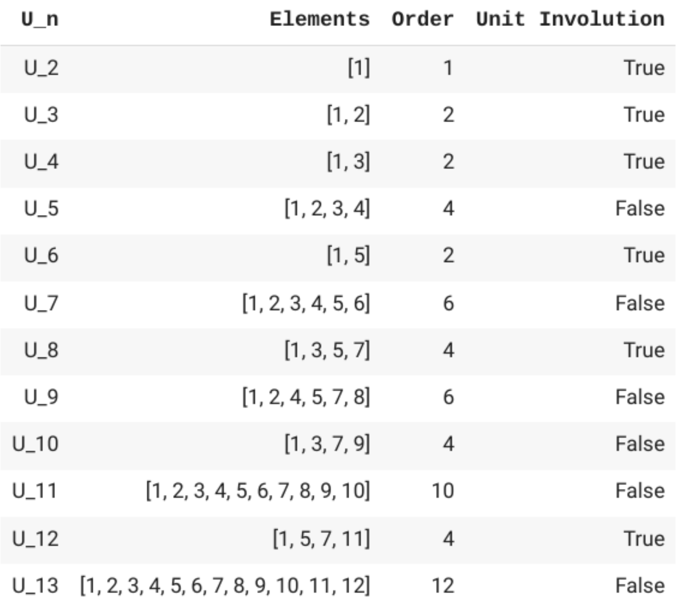
The code employs libraries such as `numpy` for numerical operations and `sympy` for prime number generation with the main steps being:

- Generate prime numbers up to n.
- Calculate the highest power of each prime that is $\leq n$.
- Sum the natural logarithms of these primes raised to their respective highest powers to significantly reduce computational overhead.

2.3 The Group U_n of Invertible Integers Mod n

We have noted that Euler's totient function, $\phi(n)$ counts the positive integers up to a given integer n that are relatively prime to n (that the two numbers share no common factors other than 1). To see it in action consider now the *group*[5] U_n, the set of integers less than n and relatively prime to n. The operation under which U_n forms a group is multiplication modulo n, denoted as $a \cdot b \mod n$. Such an operation takes two elements from the set, multiplies them, and then finds the remainder when this product is divided by n, effectively "wrapping around" when the product exceeds n. Formally, $U_n = \{a \in \mathbb{Z} | 1 \leq a < n, \gcd(a,n) = 1\}$, where $\gcd(a,n)$ denotes the greatest common divisor of a and n.

The *order* of the group U_n is given by $\phi(n)$, the *totient* function of n, because the number of elements in U_n is exactly the count of numbers less than n that are coprime to n. When n is a prime number p, the order of the group U_p is given as $\phi(p) = p - 1$ since for a prime number p, every integer from 1 to $p - 1$ is relatively prime[6] to p.



U_n	Elements	Order	Unit Involution
U_2	[1]	1	True
U_3	[1, 2]	2	True
U_4	[1, 3]	2	True
U_5	[1, 2, 3, 4]	4	False
U_6	[1, 5]	2	True
U_7	[1, 2, 3, 4, 5, 6]	6	False
U_8	[1, 3, 5, 7]	4	True
U_9	[1, 2, 4, 5, 7, 8]	6	False
U_10	[1, 3, 7, 9]	4	False
U_11	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	10	False
U_12	[1, 5, 7, 11]	4	True
U_13	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]	12	False

Figure 2.6: Group of Invertibles

For U_n, a unit *involution* is an element a for which the equation $a^2 \equiv 1 \mod n$ holds true. In other words, each element is its own inverse under the group operation, which in the case of U_n is multiplication modulo n. So for the group U_8, defined as $U_8 = \{a \in \mathbb{Z} : 1 \leq a < 8, \gcd(a,8) = 1\}$, the unit involution property can be demonstrated as follows:

- $1^2 \equiv 1 \mod 8$
- $3^2 \equiv 9 \equiv 1 \mod 8$
- $5^2 \equiv 25 \equiv 1 \mod 8$
- $7^2 \equiv 49 \equiv 1 \mod 8$

Hence, all the elements of U_8 are unit involutions because squaring each element results in an answer equivalent to 1 modulo 8.

[5] Here the concept of a *group* is a framework for describing symmetry and operations within sets. A group consists of a set of elements combined with an operation that joins any two elements to form a third in such a way that closure, associativity, the existence of an identity element, and the existence of inverse elements are satisfied.

[6] since none of those integers share any common factors with p other than 1. For a prime number p, we simply count all numbers from 1 up to $p - 1$, as all of these numbers do not share any divisors with p.

The code invertiblesUn.ipynb uses **SymPy Library**, for its symbolic mathematics that provides tools to work with the totient function and greatest common divisor (GCD) and **Pandas Library**, to organize the elements of U_n, their orders, and other properties into a structured table format and has the following key features:

```
import pandas as pd
from sympy.ntheory import totient, is_primitive_root
from sympy import gcd

def find_U_n_elements_and_order(n):
    elements = [a for a in range(1, n) if gcd(a, n) == 1]
    order = totient(n)
    involution_exists = all(pow(a, 2, n) == 1 for a in elements)
    return elements, order, involution_exists
```

The core steps here are:

1. **Determining Elements**: For each n from 2 to 13, we identified the elements of U_n by finding integers less than n that are coprime to n. This involves calculating the GCD of each integer with n and selecting those where the GCD equals 1.
2. **Calculating Order**: The order of each group U_n is found using the totient function $\phi(n)$, reflecting the number of elements in the group.
3. **Checking Unit Involution**: We examine if each element in U_n satisfies the condition $a^2 \equiv 1 \mod n$, indicating a unit involution within the group.

2.3.1 Role of U_n in RSA

RSA is a public-key cryptosystem widely used for secure data transmission whose security is based on the practical difficulty of factorizing the product of two large prime numbers. Central to its encryption and decryption mechanism is the group of invertible integers mod n, U_n or $(\mathbb{Z}/n\mathbb{Z})^*$. The key generation process is the first step in setting up RSA encryption comprising selecting a public key and a private key by choosing two distinct large prime numbers p and q. These primes are essential for calculating $n = pq$ and Euler's totient function $\phi(n) = (p-1)(q-1)$. The value of n is the modulus for both the public and private keys and its factorization is kept secret.

Selecting the Public Key Exponent

With the totient function, $\phi(n)$, counting the number of integers that are invertible modulo n, the public key exponent e is chosen such that e is in $U_{\phi(n)}$, meaning e is coprime to $\phi(n)$ and therefore has a multiplicative inverse modulo $\phi(n)$.

Calculating the Private Key Exponent

The private key exponent d is the multiplicative inverse of e modulo $\phi(n)$. This means $ed \equiv 1 \mod \phi(n)$, which is possible because e is part of the group $U_{\phi(n)}$.

RSA Encryption Protocol and Key Generation

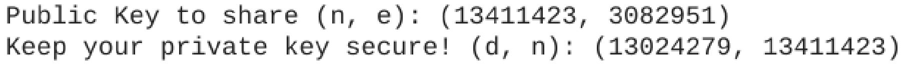
Both the public key exponent e and the private key exponent d are chosen from U_n with the encryption protocol involving three stages:

1. **Key Generation** - recipient generates a public key and a private key. The public key is shared with the sender, while the private key is kept secret.
2. **Encryption** - sender encrypts the message using the recipient's public key and sends the encrypted message.

3. **Decryption** - recipient decrypts the received message using their private key to recover the original plaintext.

Stage 1: Key Generation

In the first stage, the ultimate recipient of the ciphered message uses RSA-senderGenerateKey.ipynb, to generate a pair of keys: a public key and a private key. The public key (n,e) is shared with the sender, while the private key (d,n) is kept secret.



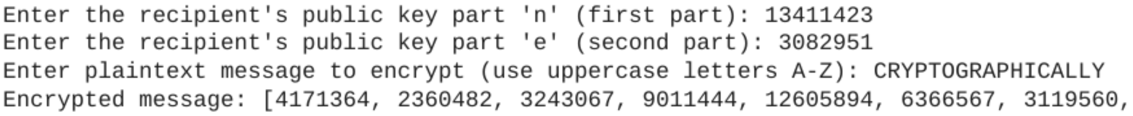
```
Public Key to share (n, e): (13411423, 3082951)
Keep your private key secure! (d, n): (13024279, 13411423)
```

Figure 2.7: Future Recipient performs Key Generation and Public Key Sharing

The public key consists of two numbers where n is the product of two randomly chosen large primes, and e is a number such that $1 < e < \phi(n)$ and $\gcd(e,\phi(n)) = 1$. The private key is a number d such that $ed \equiv 1 \pmod{\phi(n)}$.

Stage 2: Encryption

The second stage is performed by the sender, using the code, encryptPKI.ipynb. The sender uses the recipient's public key to encrypt a message and creates a ciphertext.



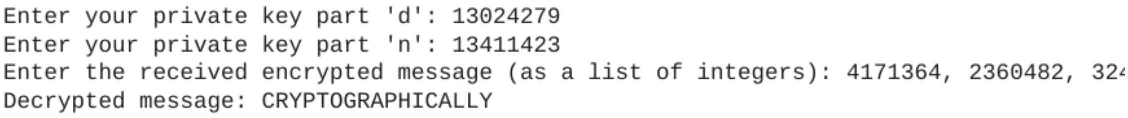
```
Enter the recipient's public key part 'n' (first part): 13411423
Enter the recipient's public key part 'e' (second part): 3082951
Enter plaintext message to encrypt (use uppercase letters A-Z): CRYPTOGRAPHICALLY
Encrypted message: [4171364, 2360482, 3243067, 9011444, 12605894, 6366567, 3119560,
```

Figure 2.8: Sender Encrypts the Message with the Public Key

The encryption process converts the plaintext message into a series of numbers based on an agreed-upon scheme, and then each number is encrypted using the formula $c \equiv m^e \pmod{n}$, where m is the message and c is the ciphertext.

Stage 3: Decryption

In the final stage, the recipient decrypts the received ciphertext using decryptRSA.ipynb their private key recovering the original plaintext from the ciphertext.



```
Enter your private key part 'd': 13024279
Enter your private key part 'n': 13411423
Enter the received encrypted message (as a list of integers): 4171364, 2360482, 32
Decrypted message: CRYPTOGRAPHICALLY
```

Figure 2.9: Recipient Decrypts the Ciphertext with the Private Key

Decryption utilizes the private key with the formula $m \equiv c^d \pmod{n}$.

Key Generation

The key generation process is crucial in RSA encryption and heavily relies on Euler's totient function, $\phi(n)$, where n is the product of two prime numbers p and q, with $\phi(n) = (p-1)(q-1)$ since p and q are prime. The `generate_keys` function generates the keys using the `sympy` and `random` libraries:

```
def generate_keys():
    prime_range_start = 10**3
    prime_range_end = 10**4
    p = randprime(prime_range_start, prime_range_end)
    q = randprime(prime_range_start, prime_range_end)
    while q == p:
        q = randprime(prime_range_start, prime_range_end)
    n = p * q
    phi_n = (p - 1) * (q - 1)
    e = random.randrange(2, phi_n)
    while gcd(e, phi_n) != 1:
        e = random.randrange(2, phi_n)
    d = mod_inverse(e, phi_n)
    return (n, e), (d, n)
```

1. Prime numbers p and q are generated using `randprime` within a specified range until distinct.
2. product $n = p \cdot q$ and Euler’s totient function $\phi(n) = (p-1)(q-1)$ are computed.
3. public key exponent e is selected randomly, $1 < e < \phi(n)$ with e coprime to $\phi(n)$ using `gcd`.
4. private key exponent d is calculated as the `mod_inverse` of e modulo $\phi(n)$.
5. function returns the public key (n,e) and the private key (d,n).

RSA Encryption and Deccryption

encryptPKI.ipynb comprises an encryption function and a user-interactive segment:

```
def encrypt(public_key, plaintext):
    n, e = public_key
    ciphertext = [pow(ord(char) - ord(’A’) + 1, e, n) for char in plaintext.upper() if ’A’
    return ciphertext
```

`encrypt` takes the public key and plaintext as inputs and returns the ciphertext. The public key is a tuple (n,e), where n is the RSA modulus and e is the public exponent. The plaintext is processed to uppercase and mapped from characters to integers with ’A’ starting at 1. Each character is then encrypted using modular exponentiation and prompted to enter the modulus and exponent of public key:

```
n = int(input("Enter the recipient’s public key part ’n’ (first part): "))
e = int(input("Enter the recipient’s public key part ’e’ (second part): "))
public_key = (n, e)
plaintext = input("Enter plaintext message to encrypt (use uppercase letters A-Z): ")
ciphertext = encrypt(public_key, plaintext)
```

The RSA decryption process is implemented by the `decrypt` function,

```
def decrypt(private_key, ciphertext):
    d, n = private_key
    plaintext = [chr((pow(char, d, n) - 1) % 26 + ord(’A’)) for char in ciphertext]
    return ’’.join(plaintext)
```

which iterates over the `ciphertext` list, and for each encrypted numeric value `char`, applies the RSA decryption algorithm, plaintext_char $= \left(\text{char}^d \mod n\right)$. Since the encrypted characters are shifted by 1 during encryption (starting from ’A’ as 1), the function subtracts 1 before applying the modulo operation with 26 and then adds the ASCII value of ’A’ to map the numbers back to letters. The resulting list of characters, `plaintext`, is then concatenated to form the decrypted message, which is returned as a string.

2.3.2 Riemann Zeta Function, $\zeta(s)$

The Riemann Zeta and Euler totient functions exhibit a formal similarity. The Euler product representation of the Riemann Zeta function $\zeta(s)$ for $\mathrm{Re}(s) > 1$:

$$\zeta(s) = \prod_{p \text{ prime}} \frac{1}{1-p^{-s}}$$

We can expand the term $\frac{1}{1-p^{-s}}$ substituting the formula for a geometric series:

$$\frac{1}{1-p^{-s}} = 1 + p^{-s} + p^{-2s} + p^{-3s} + \cdots$$

into the infinite product as:

$$\zeta(s) = \prod_{p \text{ prime}} \left(1 + p^{-s} + p^{-2s} + p^{-3s} + \cdots\right)$$

The formal structure of the Riemann Zeta function and Euler's totient function is notably similar. Both involve products over primes, and both have terms of the form $(1-p^{-\alpha})$ where α is related to the number being factored.

2.3.3 Euler's Product form of Riemann Zeta function

Consider the Riemann zeta function for $\Re(s) > 1$:

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

Euler relates this series to a product over all primes by applying a key idea similar to the sieve of Eratosthenes, where multiples of primes are systematically "sieved out" from a list of integers as:

$$\begin{aligned}\zeta(s) &= \prod_{p \text{ prime}} \frac{1}{1-p^{-s}} = \left(1 + \frac{1}{2^s} + \frac{1}{2^{2s}} + \cdots\right)\left(1 + \frac{1}{3^s} + \frac{1}{3^{2s}} + \cdots\right)\left(1 + \frac{1}{5^s} + \frac{1}{5^{2s}} + \cdots\right)\cdots \\ &= \prod_{p \text{ prime}} \left(1 + \frac{1}{p^s} + \frac{1}{p^{2s}} + \frac{1}{p^{3s}} + \cdots\right)\end{aligned}$$

The terms are the reciprocals of the powers of a prime number. To ensure that each number (and thus each term in the series) is generated exactly once, even though each number has a unique prime factorization the product "sieves" through all possible products of prime powers as Eratosthenes would sieve through multiples of primes to find prime numbers. As a result, the expanded product precisely covers every positive integer exactly once, mirroring the sum in the definition of $\zeta(s)$. Consider step by step the first two prime factors after 2 (i.e., 3 and 5):

1. Start with the series for the prime 2, which includes all powers of 2:

$$1 + \frac{1}{2^s} + \frac{1}{2^{2s}} + \cdots$$

2. Multiply by $\frac{1}{3^s}$ from the series for the prime 3:

$$\left(1 + \frac{1}{2^s} + \frac{1}{2^{2s}} + \cdots\right)\left(1 + \frac{1}{3^s}\right)$$

and so including terms like $\frac{1}{3^s}$, $\frac{1}{2^s \cdot 3^s}$, etc., adding numbers that are products of 2 and 3 to the series.

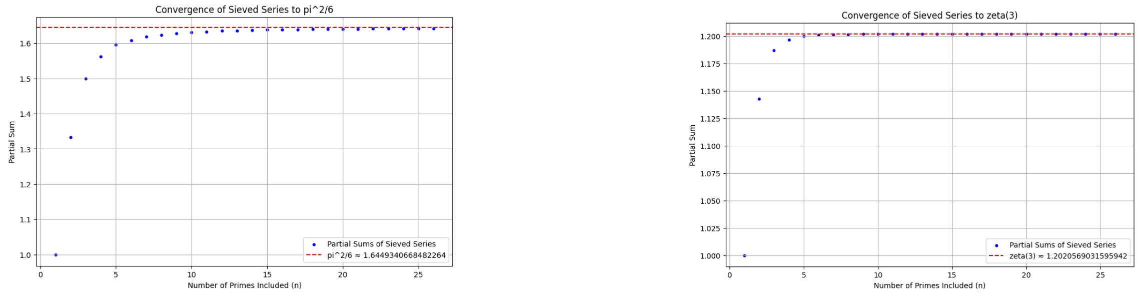


Figure 2.10: Partial sums to Riemmann number $s = 2, 3$

3. Multiply by $\frac{1}{5^s}$ from the series for the prime 5:

$$\left(1+\frac{1}{2^s}+\frac{1}{2^{2s}}+\cdots\right)\left(1+\frac{1}{3^s}\right)\left(1+\frac{1}{5^s}\right)$$

further adding terms like $\frac{1}{5^s}$, $\frac{1}{2^s \cdot 5^s}$, $\frac{1}{3^s \cdot 5^s}$, etc., systematically including numbers that are products of 2, 3, and 5.

The Python code, dynamicEulerProductRiemann.ipynb for this function uses Sympy's `Rational` and the function `illustrate_sieving_rational_partial_sums` implements the sieve,

```
def illustrate_sieving_rational_partial_sums(primes, s, terms):
    product_series = [Rational(1)]
    partial_sums = [Rational(1)]
    for p in primes:
        prime_series = [Rational(1, p ** (s * i)) for i in range(terms)]
        new_product_series = [sum(product_series[k] * prime_series[i - k]
                    for k in range(max(0, i - terms + 1),
                                   min(i + 1, len(product_series))))
                    for i in range(len(product_series) + terms - 1)]
        product_series = new_product_series
        partial_sums.append(sum(product_series))
    return partial_sums
```

It takes a list of prime numbers, a value s, and the number of terms to consider for each prime's series. Here is the functionality of the code described step by step:

1. Initialize the product series, `product_series`, with the rational number 1, representing the starting point of the product before any primes are included. Initialize `partial_sums` similarly to keep track of the sums as primes are added.
2. Iterate over each prime number in the provided list of primes.
3. For each prime p, construct a truncated geometric series, `prime_series`, with terms of the form $1/p^{si}$, where i ranges from 0 to the specified number of terms minus one.
4. Convolve the current `product_series` with the `prime_series` to include the new prime, mimicking the inclusion of a new factor in Euler's product.
5. Update `product_series` with the new series after the convolution.
6. Compute the new partial sum by summing the updated `product_series` and append it to the `partial_sums`.
7. After iterating over all primes, return the list of `partial_sums`, representing the cumulative sums of the series after the inclusion of each prime.

Our Mathematical Language

Let us finish this initial survey with a nod to that more lofty aim as as a mathematician, towards which we will occasionally err, we will need to enunciate some definitions. Indeed a great deal (if not all?) of mathematics is just definitions. The ability to typeset these notes in latex and present mathematical formula in increasingly compact but dense format being that hallmark of mathematics as the archetypal discovered language of the universe. Consider parenthetically to this point the following phrasing (concerned with less rudimentary matters as the sum of the harmonic series[7] -"the reciprocal of squared integers"):

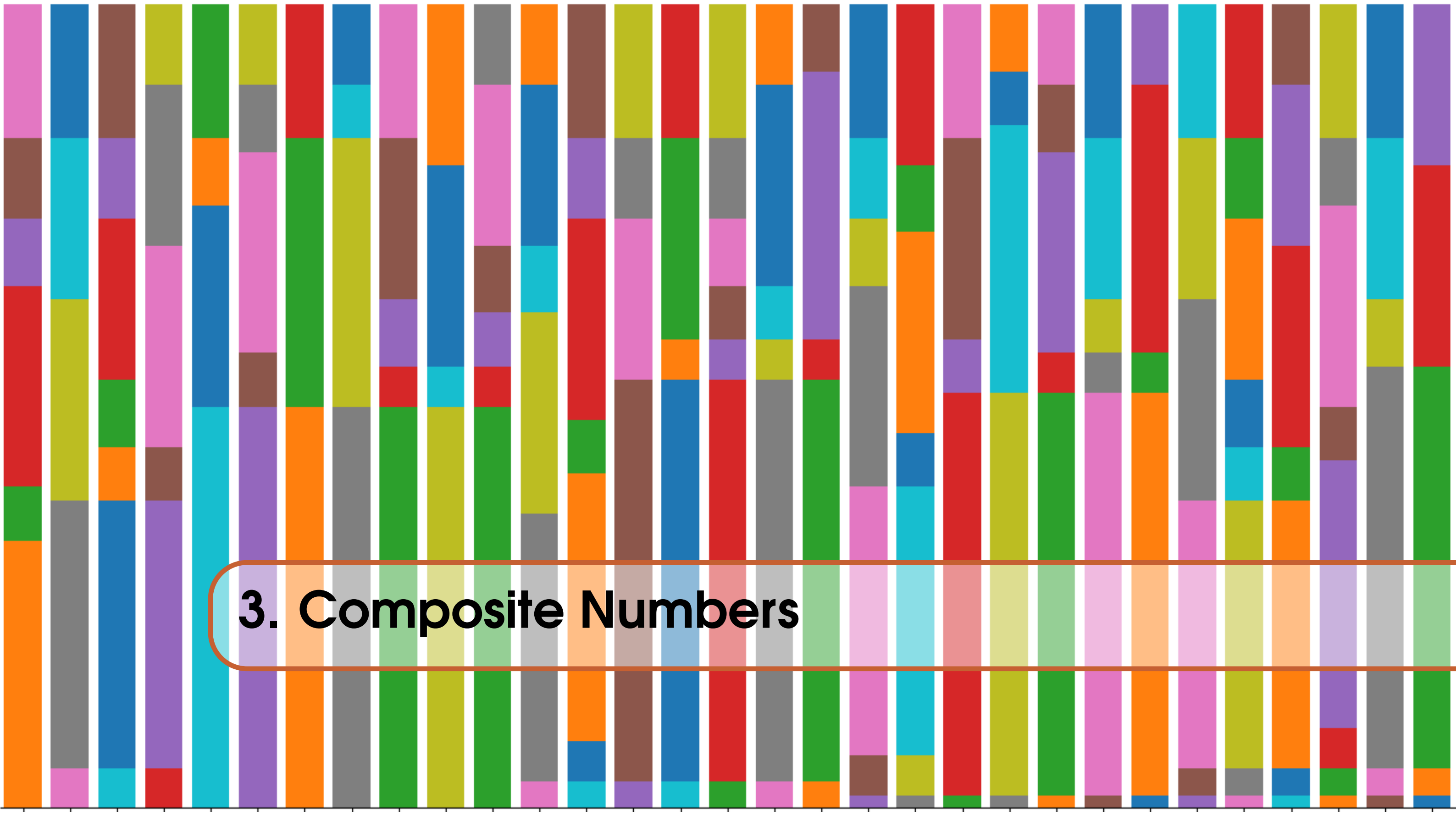
$$\zeta(2) \equiv \frac{\pi^2}{6} = \sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots$$

Note the use of $\equiv$ for "Psi-two" ($\zeta(2)$ which with the symbol $:=$ indicates a definition.[8] In latex the first equation is typeset as:

```
 \[
\zeta(2) \equiv\frac{\pi^2}{6} =& \sum_{n=1}^{\infty} \frac{1}{n^2}=
\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dotsb\]
```

[7] The infinite sum of the harmonic series when the terms are restricted to prime numbers is known as the **Prime Zeta Function**:$P(s) = \sum_{p \text{ prime}} \frac{1}{p^s}$ where s is a complex number with real part greater than 1. This series is a special case of the more general **Riemann Zeta Function**, which was implicitly defined above for $s = 2$ as $\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$. The Prime Zeta Function $P(s)$ converges for values of s with real part greater than 1, similar to the Riemann Zeta Function. However, unlike the Riemann Zeta Function, there isn't a known simple closed-form expression for the Prime Zeta Function in terms of familiar constants. It's a complex function that has important connections to number theory.

[8] Note also the choice to use merely "=" after $\frac{\pi^2}{6}$ which is justified given π's more natural definition amongst it multifarious forms.



3. Composite Numbers

"'Oh the laws of physics and of logic... the number system,...
the principle of algebraic substitution. These are ghosts.
We just believe in them so thoroughly they seem real."
— *Robert M. Pirsig, Zen and the Art of Motorcycle Maintenance*[13]

Integers can be classified as in fig 3.1 into two main subsets: even and odd numbers. Even numbers are divisible by 2 without leaving a remainder, while odd numbers have a remainder of 1 upon division by 2. Prime numbers are positive integers greater than 1 that possess only two distinct positive divisors. The number 2 uniquely fits this criterion among even numbers. The singularity of 2 as the only even prime number carries profound implications.

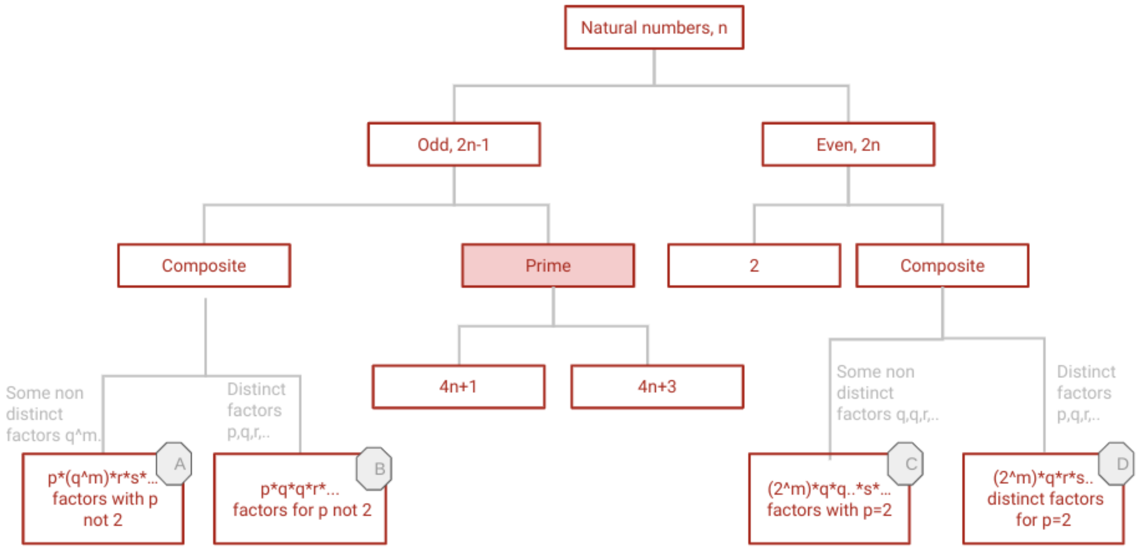

Natural numbers, n
Odd, 2n-1
Even, 2n
Composite
Prime
2
Composite
4n+1
4n+3
Some non distinct factors q^m.
Distinct factors p,q,r,..
Some non distinct factors q,q,r,..
Distinct factors p,q,r,..
A
p*(q^m)*r*s*... factors with p not 2
B
p*q*q*r*... factors for p not 2
C
(2^m)*q*q..*s*... factors with p=2
D
(2^m)*q*r*s.. distinct factors for p=2

Figure 3.1: Organigram of Integers.

I can - as is the spirit of the book - rather casually further classify positive integers according to the labelled boxes A,B,C,D above which highlight some "apparently" arbitrary distinctions that you may draw between odd composites: $45 = 3^2 \cdot 5$, $105 = 3 \cdot 5 \cdot 7$ and even ones: $350 = 2 \cdot (5^2) \cdot 7$, $210 = 2 \cdot 3 \cdot 5 \cdot 7$. We will collectively call these **n-rectangle** numbers and will elaborate soon.

I will be as casual as possible with definitions for as long as possible defining objects in both non standard as well as standard ways. An example from the organigram being the two prime generator classes "Fermat", $F_n = 4n+1$ and "Gauss", $G_n = 4n+3$. We know that these generators, (in covering the odds) **exhaustively** generate primes do not **exclusively** generate them, noting for instance the odd composites $F_2 = 4 \cdot 2 + 1 = 3^2$, $F_5 = 4 \cdot 5 + 1 = 3 \cdot 7$, $F_6 = 5^2$, and $G_8 = 5 \cdot 7$, $G_9 = 3 \cdot 13$ rendering fig 3.1 a little sloppy. A moments pause and you might well ask what is it about those choice of n, if anything, that make the prime generators fail? Why not ask: do either of the subsets of the integers $\{2,5,6,8,11,12,14,...\}$ or $\{3,6,8,9,12,13,15,...\}$ or their intersection $\{8,12,21,23,29,30,...\}$? as generated by, $failedPrimeGenerators$ warrant further investigation?

n	P_{4n+1}	C_{4n+1}	P_{4n+3}	C_{4n+3}
1	5		7	
2		9	11	
3	13			15
4	17		19	
5		21	23	
6		25		27
7	29		31	
8		33		35
9	37			39
10	41		43	
11		45	47	
12		49		51
13	53			55
14		57	59	
15	61			63
16		65	67	
17		69	71	
18	73			75
19		77	79	
20		81	83	
21		85		87
22	89			91
23		93		95
24	97			99
25	101		103	
26		105	107	
27	109			111
28	113			115
29		117		119
30		121		123
31		125	127	
32		129	131	

Table 3.1: Failed Prime Generator Composites.

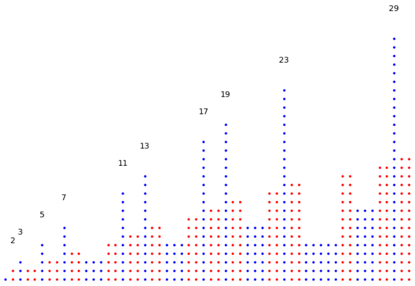
Is it worthwhile to keep asking whether the subset of "failed prime" composites, $F_{fp} \equiv \{9,21,25,27,33,...\}$ or $G_{fp} \equiv \{15,27,35,39,...\}$ are of separate interest? Indeed why not ask whether the $4n+3$ generator will continue to fail - as it is for these lower values of n - more often than the $4n+1$ generator? The `is_prime` function from: *failedPrimeGenerators* generates primes:

```
def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True
    if num % 2 == 0 or num % 3 == 0:
        return False
    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True
```

by using the function[1] `is_prime(num)` to determine whether a given number is prime by checking for divisibility by smaller primes up to the square root of the number.

3.1 Composites as p-rectangles, $p > 1$

Fig 3.1 colors consecutive integers as blue then red from 2 to 30 and represented any composite number as a rectangle of dots in which the shortest side has a width that is its smallest factor divisor. That is, we see $9 = 3 \cdot 3$ and $18 = 2 \cdot 3^2$ and note with this interlaced coloring requirement, that only 2 amongst the primes is red.



The prime factorization of the integers up to 30 as generated by code: primeFactorizationRadicalisation is shown in the p-fac column of table 3.1. The $n_p - r_p$ column highlights the difference between these two counts, which tells us the repetition of prime factors.

[1] It first checks for base cases: if `num` is less than or equal to 1, 2 or 3, returns

- Next, it checks for divisibility by 2 and 3. If `num` is divisible by 2 or 3 (i.e., `num % 2 == 0` or `num % 3 == 0`), it returns `False` because only the even 2 is prime, and numbers divisible by 3 are not prime.
- Then enters a loop starting from `i = 5` and continues until `i * i` is greater than or equal to `num` (as prime factors of a number cannot be greater than its square root) checking whether `num` is divisible by `i` or by `i + 2` (next possible prime candidates after 3). If `num` is divisible by either `i` or `i + 2`, it function returns `False`.
- If the loop completes without finding any divisors of `num`, it means that `num` is not divisible by any integers up to its square root, and therefore it's a prime number. In this case, the function returns `True`.

n	p-fac	n_p	r_p	$n_p - f_p$	f	f-ply	factors
1		0	0	0	1	1	$\{1\}$
2	2	1	1	0	2	1	$\{1, 2\}$
3	3	1	1	0	2	1	$\{1, 3\}$
4	2^2	2	1	1	3	2	$\{1, 2, 4\}$
5	5	1	1	0	2	1	$\{1, 5\}$
6	$2 \cdot 3$	2	2	0	4	2	$\{1, 2, 3, 6\}$
7	7	1	1	0	2	1	$\{1, 7\}$
8	2^3	3	1	2	4	2	$\{1, 2, 4, 8\}$
9	3^2	2	1	1	3	2	$\{1, 3, 9\}$
10	$2 \cdot 5$	2	2	0	4	2	$\{1, 2, 10, 5\}$
11	11	1	1	0	2	1	$\{1, 11\}$
12	$2^2 \cdot 3$	3	2	1	6	3	$\{1, 2, 3, 4, 6, 12\}$
13	13	1	1	0	2	1	$\{1, 13\}$
14	$2 \cdot 7$	2	2	0	4	2	$\{1, 2, 14, 7\}$
15	$3 \cdot 5$	2	2	0	4	2	$\{1, 3, 5, 15\}$
16	2^4	4	1	3	5	3	$\{1, 2, 4, 8, 16\}$
17	17	1	1	0	2	1	$\{1, 17\}$
18	$2 \cdot 3^2$	3	2	1	6	3	$\{1, 2, 3, 6, 9, 18\}$
19	19	1	1	0	2	1	$\{1, 19\}$
20	$2^2 \cdot 5$	3	2	1	6	3	$\{1, 2, 4, 5, 10, 20\}$
21	$3 \cdot 7$	2	2	0	4	2	$\{1, 3, 21, 7\}$
22	$2 \cdot 11$	2	2	0	4	2	$\{1, 2, 11, 22\}$
23	23	1	1	0	2	1	$\{1, 23\}$
24	$2^3 \cdot 3$	4	2	2	8	4	$\{1, 2, 3, 4, 6, 8, 12, 24\}$
25	5^2	2	1	1	3	2	$\{1, 5, 25\}$
26	$2 \cdot 13$	2	2	0	4	2	$\{1, 26, 2, 13\}$
27	3^3	3	1	2	4	2	$\{3, 1, 27, 9\}$
28	$2^2 \cdot 7$	3	2	1	6	3	$\{1, 2, 4, 7, 14, 28\}$
29	29	1	1	0	2	1	$\{1, 29\}$
30	$2 \cdot 3 \cdot 5$	3	3	0	8	4	$\{1, 2, 3, 5, 6, 10, 15, 30\}$

We have introduced the non-standard **ply**[2] terminology here suggested by fig 3.1.

- **0-ply** : a prime number;
- **1-ply** : a composite that has only 1 rectangle representation;
- **2-ply** : a composite, that has 2 distinct representations.

Note that 6 with prime divisors $2,3$, $6 = 2 \cdot 3 \sim 2 \cdot 3 \equiv 3 \cdot 2$ is a **2-ply** as a mere re-orientation of dots through the commutativity of multiplication is not considered a distinct rectangle. Whereas the associative property of multiplication tells us that 12 as the product of three prime divisors $12 = 2(\cdot 2 \cdot 3) = (2 \cdot 2) \cdot 3 =\sim 2 \cdot 6 = 4 \cdot 3$ will be a **3-pl** while $30 = 2 \cdot (3 \cdot 5) = (2 \cdot 3) \cdot 5 = 3 \cdot (2 \cdot 5) \sim 2 \cdot 15 = 6 \cdot 5 = 3 \cdot 10$ with 4 distinct rectangular representations is a **4-ply**. Is that the story? That an **f-ply**f-ply possesses f+1 prime factors? Pause and think of some counter examples and then look at the end of chapter notes on Combinatorics.[3]

The following function is used to calculate the *f-ply* value, which represents the number of ways to form distinct rectangles using the factors of a number. The second value returned by the function (`len(factors)`) is used to determine the *f-ply* value.

```
def count_factors(num):
    factors = set()
    for i in range(1, int(num ** 0.5) + 1):
        if num % i == 0:
            factors.add(i)
            factors.add(num // i)
    return factors, len(factors)
```

The `count_factors` function calculates the factors of a given number `num` and returns both the set of factors and the count of factors. Here's a breakdown of how the function works:

- `factors = set()`: Initialize an empty set to store the factors of the given number.
- `for i in range(1, int(num ** 0.5) + 1)`: Iterate through the numbers from 1 to the square root of `num`. This is done to efficiently find pairs of factors for `num`.
- `if num % i == 0`: Check if the current number `i` evenly divides the given number `num` without a remainder. If it does, then `i` is a factor.
- `factors.add(i)`: Add `i` to the set of factors.
- `factors.add(num // i)`: Since `i` is a factor of `num`, `num // i` is also a factor. For example, if `num` is 12 and `i` is 2, then both 2 and `12 // 2 = 6` are factors of 12.
- After the loop, the set `factors` contains all the factors of `num`, including both the ones smaller than the square root and the corresponding larger factors.
- `return factors, len(factors)`: The function returns a tuple containing two values. The first value is the set of factors calculated in step 6, and the second value is the count of factors, which is the length of the set. This provides information about the number of distinct factors that `num` has.

Here is a code snippet to draw Fig 3.1:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.axis('off')  # Turn off the axis
spacing_x = -40  # Initial horizontal spacing
color_index = 0  # Initial color index
```

[2] A ply refers to the number of layers that a paper consists of. For toilet paper, there's usually between one to four layers of paper pressed together to make a parent roll.

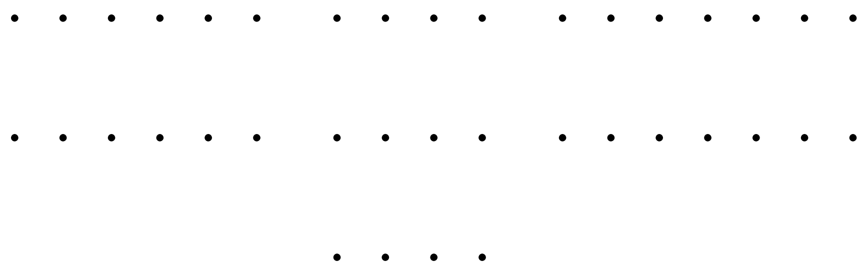
[3] The square number,$16 = 2 \cdot 2 \cdot 2 \cdot 2 \sim 2 \cdot 8 = 4\dot{4}$ is a 3-ply with 4 prime factors, whereas 90 with its four partially distinct prime factors $90 = 2 \cdot 3 \cdot 3 \cdot 5 \sim 1 \cdot 90 = 2 \cdot 45 = 6 \cdot 15 = 9 \cdot 10 = 30 \cdot 3 = 18 \cdot 5$ is a 6-ply.

```
for i in range(1, 31):
    color = "blue" if color_index == 0 else "red"
    if is_prime(i):
        shift_right = 1
        vertical_height = i
    else:
        lowest_factor = lowest_prime_factor(i)
        shift_right = lowest_factor if lowest_factor is not None else 1
        vertical_height = i // (shift_right if shift_right is not None else 1)
```

The line items after the `else` clause determine how the dots should be positioned vertically for both prime and composite numbers in the visualization, considering the shift for composites and the vertical height for both cases.

3.2 The Rectangular Composites

The composites $12 = 2 \times 6 = 4 \times 3$ and $14 = 2 \times 7$ are respectively 2-ply and 1-ply numbers.



We define the following in order to make clear the distinctions between composites:

Definition 1 *Proper Factors of a Composite Number c* are the subset of divisors of c which are not c itself. For 12, whose divisors are 1, 2, 3, 4, 6, and 12 has proper factors 1, 2, 3, 4, and 6.

Definition 2 *Rectangular Number* Is a composite number that can be arranged into a rectangle with integer dimensions greater than 1. Six can be arranged in a rectangle with dimensions 2×3.

Definition 3 *Oblong Number* is a rectangle numbers whose area is the product of two consecutive integers, represented as $n(n+1)$. Twelve is the product of the consecutive integers 3 and 4.

Definition 4 *Semi-Prime Number* is a singular (1-ply) rectangle number that is the product of two prime numbers. The two primes can be identical so $9 = 3 \times 3$, is semi and square.

Definition 5 *Non-square Deficient Number* is rectangle number that is deficient, meaning the sum of its proper divisors (excluding itself) is less than the number itself. 14 is not a perfect square with three proper divisors 1, 2, and 7 summing to 10, which is less than 14.

3.2.1 Bubble chart of Semi-prime Totients

Semi-primes are of particular interest in number theory and cryptography due to their role in prime factorization problems. The plot is of the distribution of semi-primes up to 1000, where each point represents a semi-prime composed of two prime factors p and q. The x-axis and y-axis correspond to the smaller and larger prime factors of the semi-primes, respectively. The size of each point is proportional to the totient of the semi-prime, while the color indicates the nature of its Mobius function value - red for semi-primes with square factors and green for others.

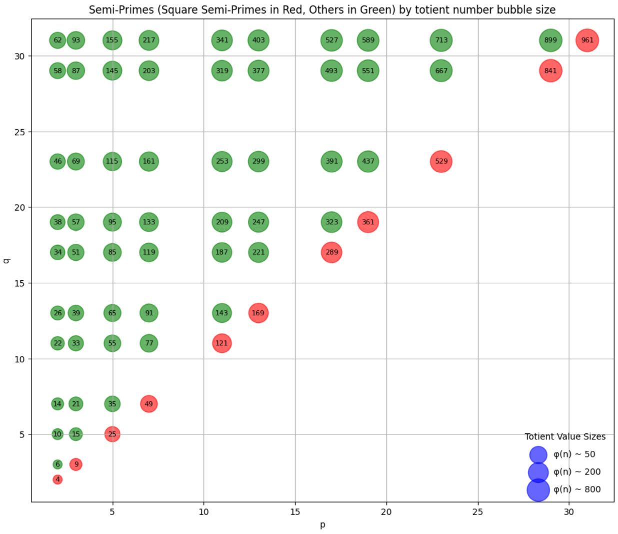


Figure 3.2: Totients of Semi-Primes

The `mobius` function overleaf determines the free-of-square condition of an integer taking a single integer argument n and returning a value based on the prime factorization of n.

```
def mobius(n):
    if n == 1:
        return 1
    factors = set()
    for p in primerange(1, n + 1):
        if p*p > n:
            break
        if n % p == 0:
            n //= p
            if n % p == 0:
                return 0  # Square factor found
            factors.add(p)
    if n > 1:
        factors.add(n)
    return -1 if len(factors) % 2 else 1
```

The function immediately returns 1 if n is 1, and:

- A set called `factors` is initialized to keep track of unique prime factors of n.
- A loop iterates through the prime numbers up to n using the `primerange` function which is sufficient because prime factors of n cannot exceed n itself.
- If the square of the current prime number exceeds n, the loop breaks, optimizing the function by avoiding unnecessary iterations.
- Within the loop, if n is divisible by the current prime p, n is divided by p. If n is still divisible by p, this implies p^2 is a factor, it returns 0, indicating that n has a squared prime factor.
- If n does not have a squared prime factor, the prime p is added to the set of factors.
- After the loop, if n is greater than 1, n is itself a prime and is added to the set of factors.
- returns -1 if the number of unique prime factors is odd, and 1 if even.

Finding Semi-Primes

The semi_primes_and_mobius function identifies semi-prime numbers within a given range and computes their Mobius and totient values:

```
def semi_primes_and_mobius(limit):
    primes = list(primerange(1, int(np.sqrt(limit)) + 1))
    semi_primes = []
    for i in range(len(primes)):
        for j in range(i, len(primes)):
            prod = primes[i] * primes[j]
            if prod <= limit:
                mu_val = mobius(prod)
                phi_val = totient(prod)
                semi_primes.append((primes[i], primes[j], prod, mu_val, phi_val))
    return semi_primes
```

- Generating a list of prime numbers up to the square root of the limit, as any semi-prime less than or equal to the limit will have factors no greater than this value.
- Iterating over pairs of these primes to find their products, which are the semi-primes.
- Calculating the Mobius value (mu_val) to check if the semi-prime has square factors and the Euler's totient value (phi_val), which represents the count of numbers less than the semi-prime that are coprime to it.
- Storing the pair of primes, their product (the semi-prime), and the calculated values in a list.

Visualizing Semi-Primes

The following code snippet focuses on bubble sizes, centering, and labeling scaling the totient values so as to adjust bubble sizes and color encoding the Mobius function value. The labels are carefully centered within each bubble and logarithmic scaling ensures bubble sizes are proportionate when the range of totient values is large.

In order to dynamically generate the necessary attributes for each point in the scatter plot, list comprehensions are used to iterate through a list of tuples, here semi_primes_info contains information about $(p, q, \text{semi-prime}, \mu(n), \phi(n))$, where p and q are the prime factors of the semi-prime, $\mu(n)$ is the Mobius function value, and $\phi(n)$ is the Euler's totient function value for the semi-prime.

```
x = [item[0] for item in semi_primes_info]  # p values
y = [item[1] for item in semi_primes_info]  # q values
sizes = [np.log(int(item[4]) + 1) * 100 for item in semi_primes_info]
colors = ['red' if item[3] == 0 else 'green' for item in semi_primes_info]

plt.figure(figsize=(12, 10))
scatter = plt.scatter(x, y, s=sizes, c=colors, alpha=0.6)

labels = [item[2] for item in semi_primes_info]
for i in range(min_length):
    plt.annotate(labels[i], (x[i], y[i]), fontsize=8, ha='center', va='center')
```

The variable item in the list comprehension is a placeholder for the current tuple from the list. Square brackets item[] are used to access specific elements within the tuple by their index where indexing starts at 0 and so item[0] refers to the first element:

- `x = [item[0] for item in semi_primes_info]` extracts the first prime factor p from each tuple, creating a list of x-coordinates for the scatter plot.
- `y = [item[1] for item in semi_primes_info]` extracts the second prime factor q from each tuple, creating a list of y-coordinates for the scatter plot.
- `sizes = [np.log(int(item[4]) + 1) * 100 for item in semi_primes_info]` computes the sizes of the scatter plot bubbles by applying a logarithmic scale to the totient values ($\phi(n)$). The totient value is accessed with `item[4]`, incremented by 1 to avoid taking the logarithm of 0, and then multiplied by 100 to scale the bubble size appropriately for visualization. It also converts totient values from `sympy.Integer` to the standard Python `int` type which is necessary because the `sympy.Integer` type is not compatible with the NumPy `log` function.
- `colors = ['red' if item[3] == 0 else 'green' for item in ...]` determines the color of each bubble based on the Mobius function value $\mu(n)$. If `item[3]` (the Mobius function value) is 0, indicating a square factor in the semi-prime, the bubble is colored red; otherwise, it is green.
- `labels = [item[2] for item in semi_primes_info]` retrieves the semi-prime number itself from each tuple to use as labels for the scatter plot points.

3.2.2 Semi-prime Perimeter to Area ratios

The following table is a product of this code and lists those 1-ply semi-primes that can only be presented as one type of rectangle.

Number	Divisors	P/A
6	(2, 3)	1.67
8	(2, 4)	1.50
10	(2, 5)	1.40
14	(2, 7)	1.29
15	(3, 5)	1.07
21	(3, 7)	0.95
22	(2, 11)	1.18
26	(2, 13)	1.15
27	(3, 9)	0.89
33	(3, 11)	0.85
34	(2, 17)	1.12
35	(5, 7)	0.69
38	(2, 19)	1.11
39	(3, 13)	0.82
46	(2, 23)	1.09
51	(3, 17)	0.78

Table 3.2: List of numbers, their divisors, and corresponding P/A values.

The scatter plot that summaries this table follows

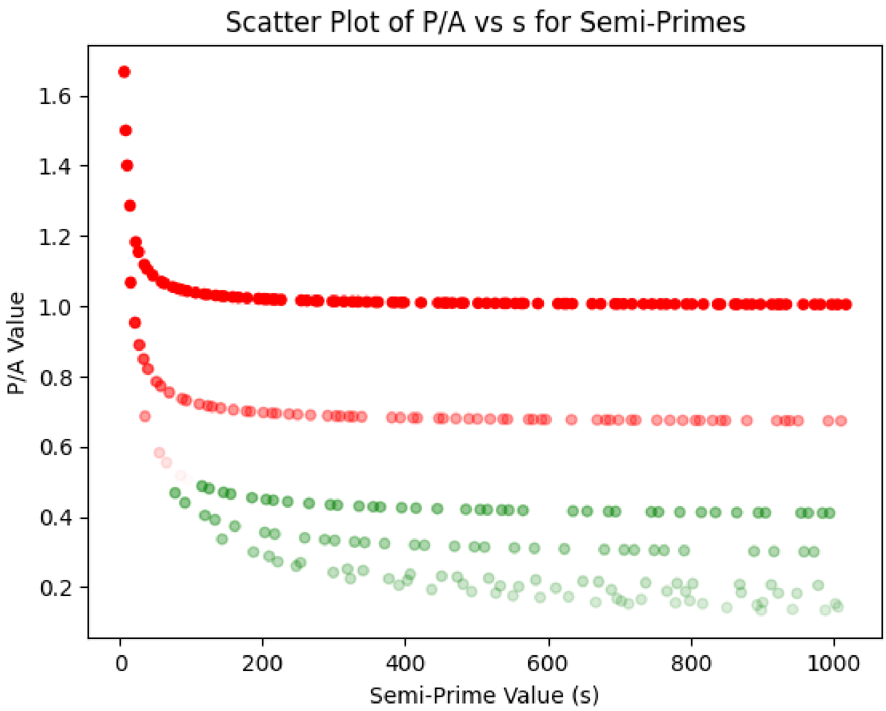


Figure 3.3: Primes and Composites.

3.2.3 Divisor Density Ratio

A non standard metric that might provide additional insight into a number's combinatorial properties is what we term as the number's *Divisor Density Ratio* (DDR).

Definition 6 *Divisor Density Ratio (DDR)* Given a number a, let m represent the multiplicity of the number, or the number of unique ways it can be expressed as the sum of its proper divisors. The DDR is then defined as: $\mathrm{DDR}(a) = \frac{m}{a}$

This ratio measures the combinatorial efficiency of a number. A higher DDR indicates that, relative to its magnitude, the number can be represented in many ways using its divisors. A striking example of a relatively small number with an unusually high DDR is 360. Its prime factorization is given by $360 = 2^3 \times 3^2 \times 5$. This leads to a significant number of divisors, precisely 24 if we include 1 and 360 itself. The formula to determine the total number of divisors of a number based on its prime factorization is: $d(n) = (a_1+1)(a_2+1)\dots(a_k+1)$ For 360, this results in:

$$d(360) = (3+1)(2+1)(1+1) = 24$$

It is the diverse ways in which these divisors can be grouped to sum to 360 that results in its high DDR. 360 boasts a staggering 22,208 unique combinations from its divisors.

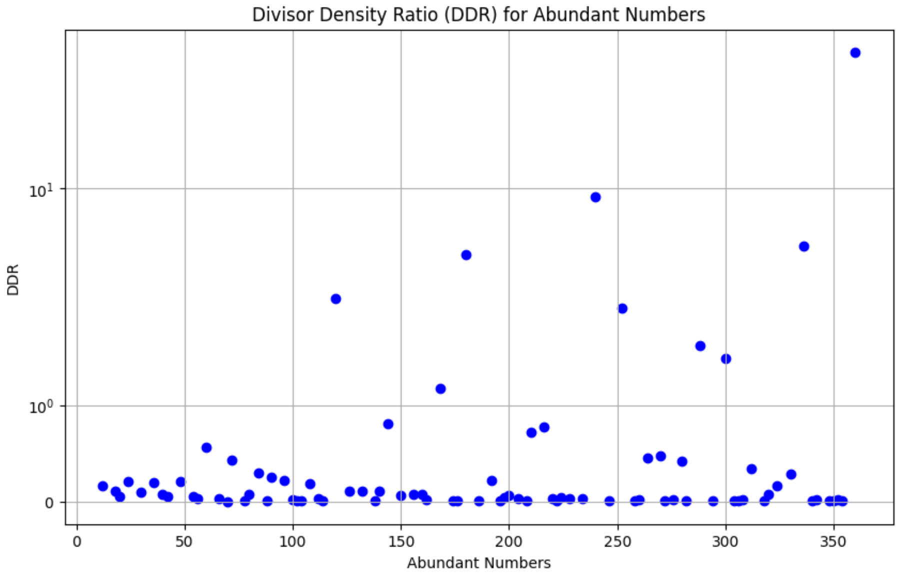


Figure 3.4: Primes and Composites.

The relative anomaly of 360 and its exceptionally high DDR suggests metrics offer fresh ways to assess and appreciate numbers beyond traditional divisibility properties.

DiversityDensityRatioOfAbundantNumbers.ipynb delivers the semi-log plot of Fig 3.4s.

a	m	a/m (DDR)
360	22208	61.69
240	2157	8.99
336	1554	4.62
180	751	4.17
120	278	2.32
252	516	2.05
288	469	1.63
300	446	1.49
168	197	1.17

Table 3.3: Abundant numbers whose DDRs> 1

3.3 Highly Composite and Super abundant Numbers

Definition 4 *Highly Composite number* is defined as a positive integer that has more divisors than any smaller positive integer than itself. The prime factorization of highly composite number greater than 36 appears to follow the form $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$, where p_i are distinct prime numbers and a_i are their corresponding exponents such that $a_1 \geq a_2 \geq \dots \geq a_k$, and typically, the last exponent $a_k = 1$.

As such Highly Composite numbers, *HCN* are antonyms of prime numbers. The stacked chart below illustrates their prime factorization. Each segment of the bar represents the power of a prime factor, and the height of the stack corresponds to the total number of times all such prime factors appear. The different colors in each stack represent the different prime factors and note that the blocks never become longer as we add them to lower ones indicative of a result of Ramanunjan, `https://archive.lib.msu.edu/crcmath/math/math/h/h269.htm`[20].

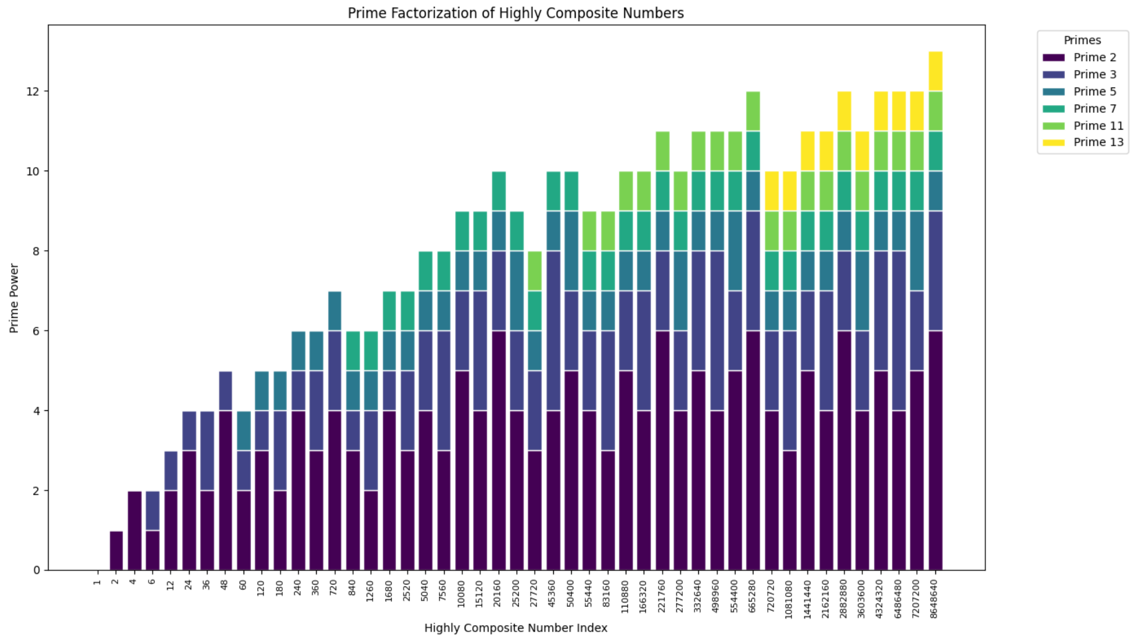


Figure 3.5: Stack Chart of Highly Composite Numbers.

`find_highly_composite_numbers(n)` function in HighlyCompositeNumbers.ipynb calculates all highly composite numbers up to a specified limit *n*. It contains a nested function `count_divisors(num)` that determines the number of divisors for a given integer *num* by iterating up to its square root and counting divisors in pairs to optimize the process.

```
def find_highly_composite_numbers(n):
    def count_divisors(num):
        divisors = 0
        for i in range(1, int(num**0.5) + 1):
            if num % i == 0:
                divisors += 2 if num // i != i else 1
        return divisors
```

The main function maintains a list, `highly_composite_numbers`, to store the highly composite numbers identified and a variable, `max_divisors_so_far`, to keep track of the largest number of divisors found for any number and then iterates over every number from 1 to *n*, calling `count_divisors(i)` for each number *i*.

```
    highly_composite_numbers = []
    max_divisors_so_far = 0
    for i in range(1, n + 1):
        divisors_count = count_divisors(i)
        if divisors_count > max_divisors_so_far:
            highly_composite_numbers.append(i)
            max_divisors_so_far = divisors_count
    return highly_composite_numbers
```

If a number *i* has more divisors than any previous number (i.e., more than `max_divisors_so_far`), it is appended to the list of highly composite numbers, and `max_divisors_so_far` is updated to reflect the new maximum number of divisors encountered.

The code HighlyCompositeNumberOblongs.ipynb delivers the following table:

HCN	Prime Factorization	Is Oblong
1		False
2	2	True
4	2^2	False
6	2×3	2×3
12	$2^2 \times 3$	3×4
24	$2^3 \times 3$	False
36	$2^2 \times 3^2$	False
48	$2^4 \times 3$	False
60	$2^2 \times 3 \times 5$	False
120	$2^3 \times 3 \times 5$	False

Table 3.4: Table of Highly Composite Numbers (HCN), their prime factorizations, and whether they are Oblong.

Problem 3.1 Amongst the beginning of the list of Highly Composite Numbers are two consecutive Oblong numbers $6 = 2 \times 3$ and $12 = 3 \times 4$. Does this happening again?

Definition 5 *Superabundant Number* is a positive integer for which the sum of divisors (inclusive of the number itself) divided by the number is greater than that for any smaller positive integer.

For the Divisor Function, $\sigma(n)$ we have that $\frac{\sigma(n)}{n}$ exceeds $\frac{\sigma(k)}{k}$ for all $k < n$, which means that the number is superabundant if $\sigma(n)/n > \sigma(k)/k$ for every positive

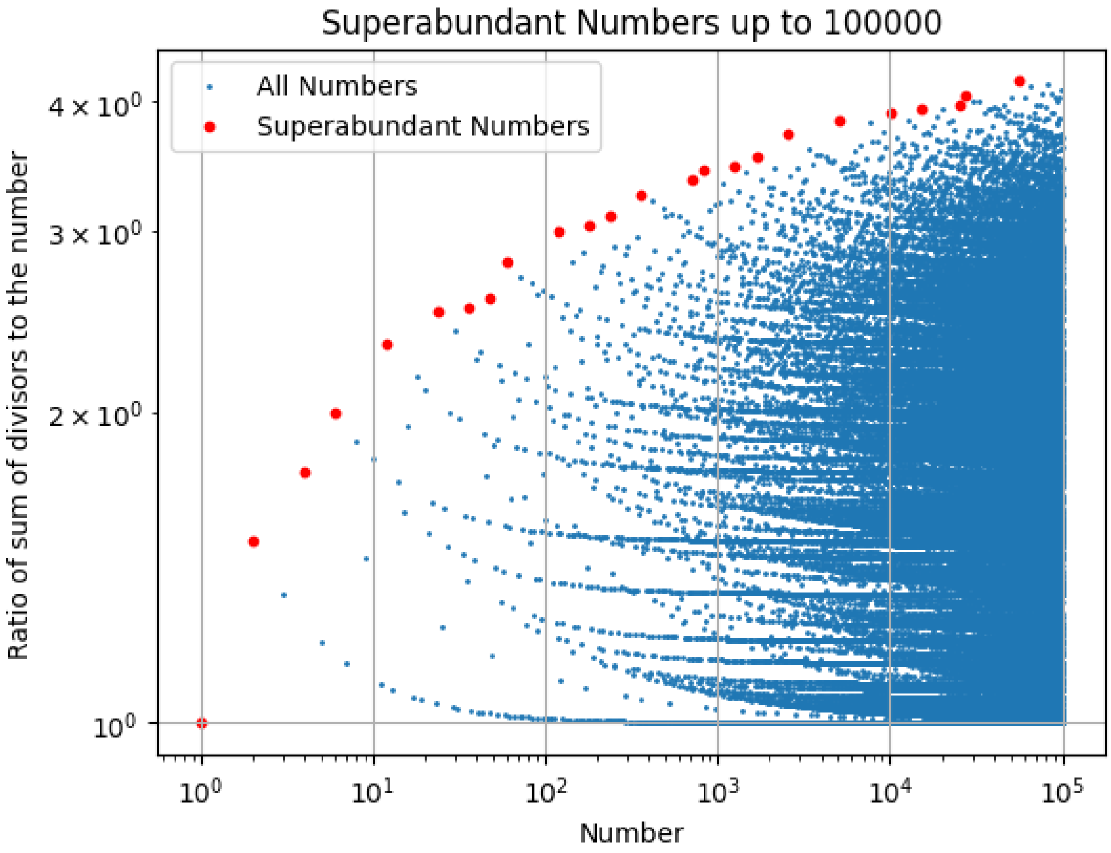


Figure 3.6: Scatterplot of Highlighted Superabundant Numbers.

A superabundant number is not just about having many divisors; it's about having a sum of divisors that is large relative to the number itself, more so than for any smaller number. The code, superAbundant.ipynb draws the scatterplot

The function `sum_of_divisors(n)` is defined to calculate the sum of all positive divisors of an integer n, which includes both 1 and n itself. The function works as follows:

```
def sum_of_divisors(n):
    total = 0
    for i in range(1, n + 1):
        if n % i == 0:
            total += i
    return total
```

For a given input n, the function initializes a variable `total` to 0, which accumulates the sum of divisors. It iterates over all integers from 1 to n and adds i to `total` if i is a divisor of n, which is checked by the condition $n\%i == 0$.

The `plot_superabundant_numbers(limit)` function plots the ratios of the sum of divisors to the number for each integer up to a given `limit`. The essential steps of the function are:

```
def plot_superabundant_numbers(limit):
    ratios = []
    max_ratio = 0
    superabundant_numbers = []
    for n in range(1, limit + 1):
        ratio = sum_of_divisors(n) / n
        ratios.append(ratio)

        if ratio > max_ratio:
            max_ratio = ratio
            superabundant_numbers.append(n)
```

The function maintains a list called `ratios` to store the ratio $\sigma(n)/n$ for each n, where $\sigma(n)$ is the sum of divisors of n. It keeps track of the highest ratio found so far in `max_ratio`. For each number n, if the computed ratio is greater than `max_ratio`, then n is considered superabundant, and n is added to the list `superabundant_numbers`.

3.3.1 Roundness

Definition 6 *Roundness* of a number, n is defined as the sum of the powers of its prime factors expressed as $n = p_1^{a_1} \cdot p_2^{a_2} \cdot \ldots \cdot p_k^{a_k}$, where p_i are prime numbers and a_i are their respective powers, so that the roundness r of the number n is given by $r = a_1 + a_2 + \ldots + a_k$.

Here are the roundness calculations for selected numbers:

- **Number 8:** Prime Factorization: $8 = 2^3$, Roundness: $r = 3$.
- **Number 18:** Prime Factorization: $18 = 2^1 \cdot 3^2$, Roundness: $r = 1 + 2 = 3$.
- **Number 32:** Prime Factorization: $32 = 2^5$, Roundness: $r = 5$.

The code, regressionCompositeRoundness.ipynb delivers the following plot:

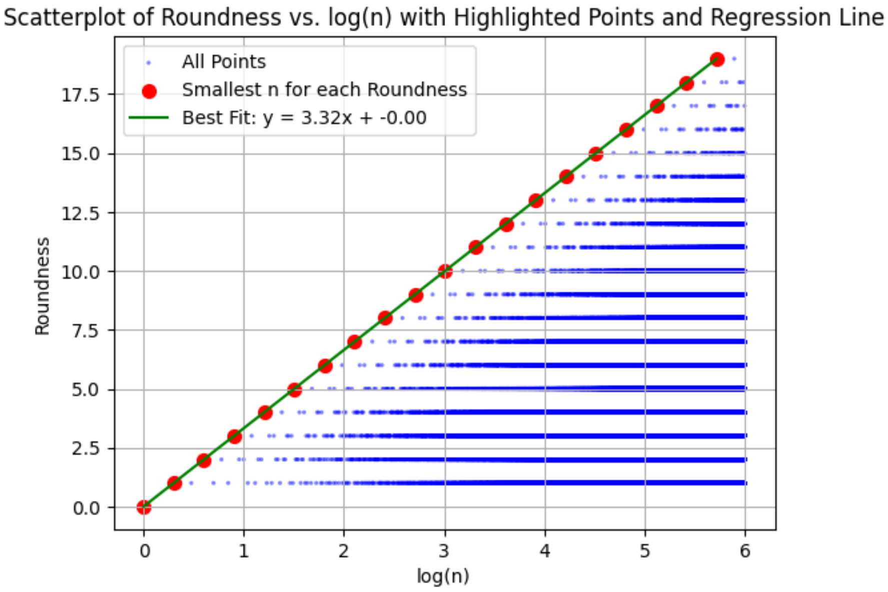


Figure 3.7: roundness of first 1,000,000 numbers.

The regression equation in logarithmic-linear space, $y = 3.32x$, where $x = \log_{10}(n)$, translates to a power law in the original non-logarithmic space:

$$n = 10^{\frac{y}{3.32}}$$

This indicates that the number n scales with the power of 10 to the roundness value y divided by the slope of the linear fit.

We can also plot these on a golden spiral where each point represents a number, and the color of the point corresponds to the number's roundness value. A distinct color is assigned to each roundness value, with a total of 12 colors used to represent roundness values from 0 to 12. The color mapping is as follows:

- Roundness 0 is colored Black.
- Roundness 1 is colored Blue (Code: #1f77b4).
- Roundness 2 is colored Orange (Code: #ff7f0e).
- Roundness 3 is colored Green (Code: #2ca02c).
- ... (additional colors and roundness values would continue similarly) ...
- Roundness 12 is colored Dark Red (Code: #8B0000).

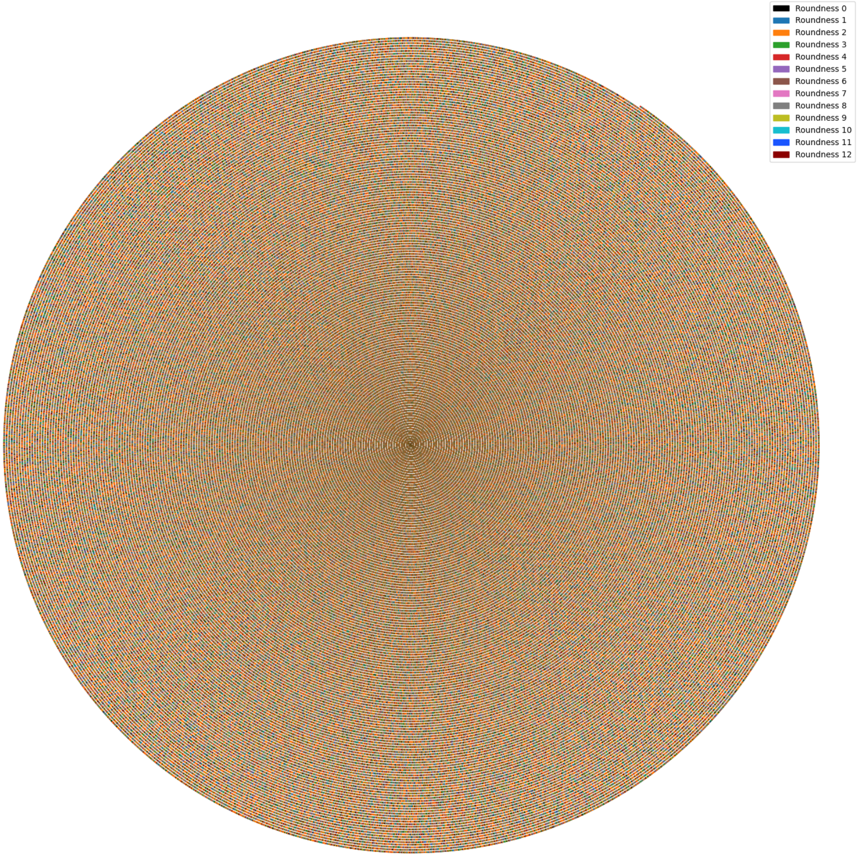


Figure 3.8: roundness of first million numbers.

Roundness is determined in the following function

```
def calculate_roundness_up_to_n(n):
    roundness_values = [0] * (n + 1)

    for i in range(2, n + 1):
        if roundness_values[i] == 0:  # i is prime
            # Set roundness for all multiples of the prime
            for j in range(i, n + 1, i):
                power = 0
                number = j
                while number % i == 0:
                    number //= i
                    power += 1
                roundness_values[j] += power

    return roundness_values
```

The function iterates through the first n numbers, identifying prime numbers and their multiples. For each multiple of a prime, it calculates the power to which the prime divides the number (i.e., the roundness of the number with respect to that prime factor) and sums these powers to obtain the total roundness.

3.4 Fermat's Little Theorem

Fermat's Little Theorem states that for any prime number p and any integer a not divisible by p, the following holds:

$$a^{p-1} \equiv 1 \pmod{p} \tag{3.1}$$

Fermat's Little Theorem provides a way to test if a number is likely prime, but it is not foolproof when used alone due to the existence of Fermat pseudoprimes. For actual prime numbers, however, the theorem holds for all appropriate bases. Consider the prime number $p = 7$. With base $a = 3$, Fermat's Little Theorem states:

$$3^{7-1} \equiv 1 \pmod{7} \tag{3.2}$$

Computing this, we find that $3^6 = 729$, which indeed satisfies:

$$729 \equiv 1 \pmod{7} \tag{3.3}$$

This illustrates that for true prime numbers, Fermat's Little Theorem is always satisfied for any base a coprime to p.

Pseudoprimes

However, there exist composite numbers (non-primes) for which this congruence relation still holds for certain bases a. These are known as *Fermat pseudoprimes* to the base a, abbreviated as psp(a). For instance, consider the composite number $n = 341$. For the base $a = 2$, we find that:

$$2^{340} \equiv 1 \pmod{341} \tag{3.4}$$

Although 341 (which is 11×31) is not a prime, it satisfies Fermat's condition for base 2, making it a Fermat pseudoprime to base 2, or a *Poulet Number*, psp(2)s, named after the mathematician Poulet which are composite numbers n such that:

$$2^{n-1} \equiv 1 \pmod{n} \tag{3.5}$$

Poulet numbers are significant as they show the limitations of Fermat's Little Theorem for primality testing. While a prime number will satisfy the condition $a^{p-1} \equiv 1 \pmod{p}$ for any integer a not divisible by p, Poulet numbers are composite yet they satisfy the condition for $a = 2$, misleadingly indicating they are prime. These numbers are significant in number theory and cryptography because they serve as exceptions to Fermat's Little Theorem, highlighting its limitation for primality testing.

Table of Pseudoprimes Per Base

The table below lists pseudoprimes for prime bases up to 29 and is produced by BasePseudoPrime.ipynb. Notably, the pseudoprimes for base 2 are Poulet numbers, which are composite numbers that satisfy Fermat's Little Theorem for base 2.

Base	**2**	**3**	**5**	**7**	**11**	**13**	**17**	**19**
psp	341	91	4	6	10	4	4	6
	561	121	124	25	15	6	8	9
	645	286	217	325	70	12	9	15
	1105	671	561	561	133	21	16	18
	1387	703	781	703	190	85	45	45
	1729	949	1541	817	259	105	91	49
	1905	1105	1729	1105	305	231	145	153
	2047	1541	1891	1825	481	244	261	169
	2465	1729	2821	2101	645	276	781	343
	2701	1891	4123	2353	703	357	1111	561

Analysis of Pseudoprime Remainders

The spider plot illustrates the remainders of $a^{n-1} \mod n$ for prime bases a up to $n-1$, where n is a chosen pseudoprime. Each 'spoke' on the plot represents a prime base, and the radial distance from the center of the plot corresponds to the logarithm of the remainder for that base.

For a pseudoprime n, certain bases will yield a remainder of 1, suggesting that n exhibits prime-like characteristics for these bases, despite being composite. The plot's logarithmic scale emphasizes the differences in remainders, making it easier to spot the bases where the remainder is 1 (as they align with the center of the plot). The significance of base 2 in the context of pseudoprimes is highlighted by the fact that a pseudoprime to base 2, often referred to as a Poulet number, deceives the base 2 Fermat primality test. Poulet numbers are a specific subset of pseudoprimes for which $2^{n-1} \equiv 1 \mod n$. The code, SpiderPseudoprimes.ipynb generate this plot and to analyze the remainders is provided below:

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sympy import primerange, isprime

# List of the first 10 Fermat pseudoprimes to base 2
pseudoprimes = [...]

def analyze_pseudoprime(pseudoprime):
    ... # (Python code that generates the plot and DataFrame)

selected_pseudoprime = 341
df_remainders = analyze_pseudoprime(selected_pseudoprime)
print(df_remainders.head(20))
```

An example spider plot for the pseudoprime $n = 341$ is shown below.

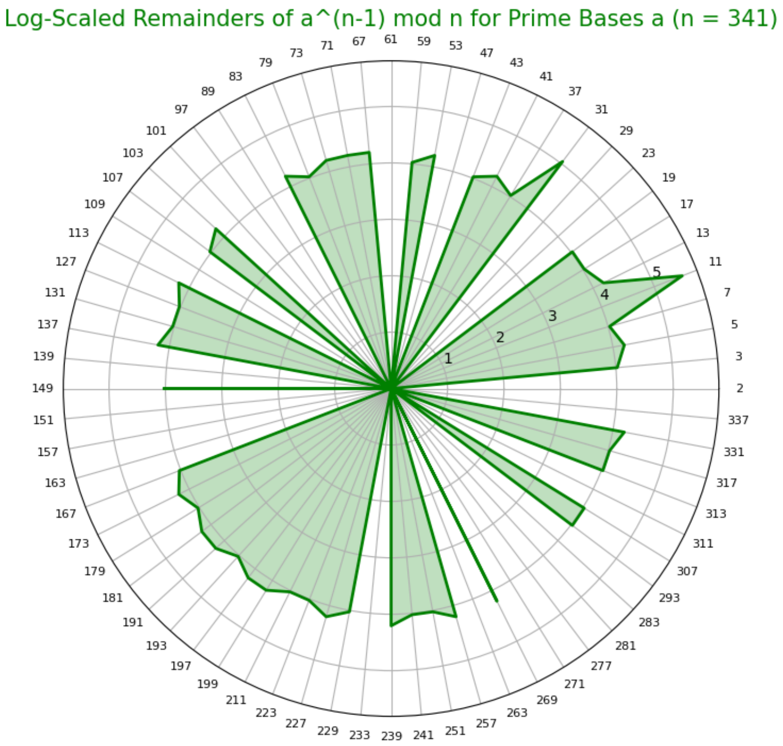


Figure 3.9: Log-Scaled Remainders of $a^{340} \mod 341$ for Prime Bases a

The spider plot illustrates the logarithmically scaled remainders of $a^{342} \mod 343$ for various prime bases a. Notably, for base 19, which is a factor of 342 ($19 \times 18 = 342$), we observe a remainder of 1, in accordance with Fermat's Little Theorem.

For base 7, we may note that since 343 is a perfect cube of 7 (7^3), the theorem would predict a remainder of 1, because 7^{342} is equivalent to $(7^3)^{114}$, and any integer power of 343 will be congruent to 1 modulo 343. The spider plot shows a remainder that is close to 10^{-2} on the logarithmic scale, indicating a computational rounding error or a precision limitation in the calculation, as mathematically, the remainder should indeed be 1.

343 is not prime since $343 = 7^3$, and while it is a composite number it does not satisfy the second criterion due to the lack of coprimality:

$$\mathrm{GCD}(7,343) = 7 \neq 1$$

Because 7 and 343 share a common factor (which is 7 itself), they are not coprime, $7^{342} \mod 343 = 1$. Since one of the key features of pseudoprimes is that they should appear prime for a base to which they are coprime, 343 does not exhibit this characteristic with respect to base 7.

For a number to be a pseudoprime to a given base, it should "fool" Fermat's Little Theorem by satisfying the congruence even though it is composite. 343 fails to "fool" the theorem in base 7 because the theorem's congruence holds trivially for any power of 343 when reduced modulo 343. Therefore, while 343 does satisfy the congruence $a^{n-1} \equiv 1 \mod n$ for base 7, it does not do so as a pseudoprime; rather, it does so as a consequence of its own factorial structure being a cube of 7.

3.4.1 Carmichael Numbers as Odd Pseudoprimes

A Carmichael number is an odd composite number n that satisfies the congruence

$$a^{n-1} \equiv 1 \mod n \tag{3.6}$$

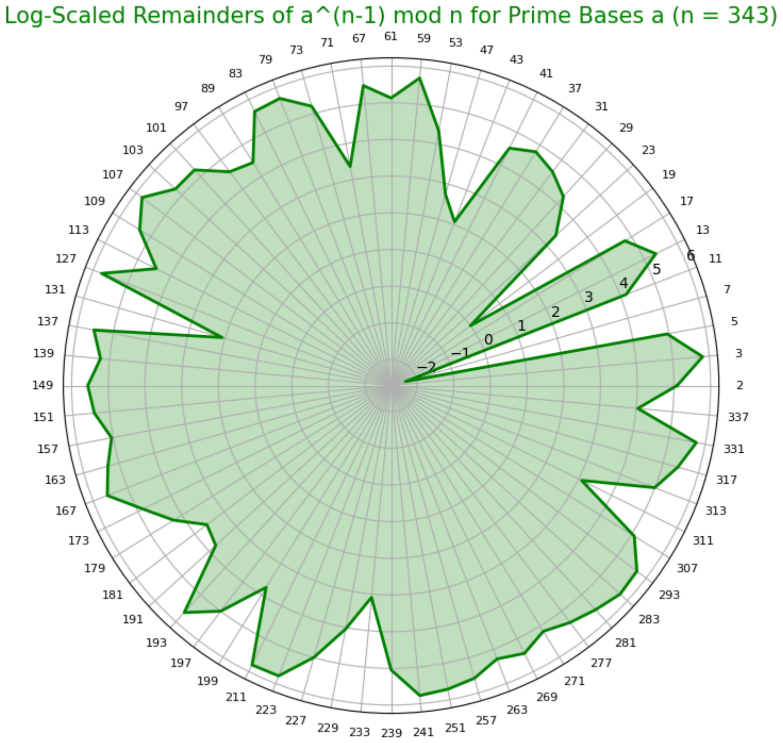


Figure 3.10: Log-Scaled Remainders of $a^{342} \mod 343$ for Prime Bases a

for every integer a that is relatively prime to n, i.e., $\gcd(a,n) = 1$. This property makes them *pseudoprimes* to any base that is relatively prime to them. While all Carmichael numbers are pseudoprimes, not all pseudoprimes are Carmichael numbers. Carmichael numbers are a special class of odd pseudoprimes exhibiting the aforementioned property for all a coprime to n. This is in contrast to general pseudoprimes, which may only satisfy the congruence for certain values of a. It is this universality that distinguishes Carmichael numbers and makes them particularly deceptive in primality testing, as they pass the Fermat primality test for every choice of a coprime to n.

3.4.2 Three-Factor Carmichael Numbers

A subset of Carmichael numbers, *cuboid Carmichael numbers*, being square-free and with an odd number (three) of distinct prime factors has a Möbius function $\mu(n) = -1$, can be visualized as "cuboids" in the space of prime numbers, where each dimension corresponds to one of the prime factors. These three-factor Carmichael numbers are of the form $(6k+1)(12k+1)(18k+1)$, under the condition that each factor is a prime number. This specific form is of interest because, once one prime factor is fixed, the number of possible Carmichael numbers that can be constructed is finite. The following table lists the first five cuboid Carmichael numbers

The code, carmichael.ipynb process of generates the first 25 Carmichael numbers, including the first five of the three-factor form, and involves several key steps:

1. Identify composite numbers that are not prime.
2. For each composite number n, check if it satisfies $a^{n-1} \equiv 1 \mod n$ for all a such that $1 < a < n$ and $\gcd(a,n) = 1$. This identifies n as a Carmichael number.
3. For generating three-factor Carmichael numbers of the form $(6k+1)(12k+1)(18k+1)$, iterate through values of k and check if each of the factors is prime. If all three factors are prime, the product is a Carmichael number of the desired form.

k	Prime Factors	Carmichael Number
1	$(7,13,19)$	1729
6	$(37,73,109)$	294409
35	$(211,421,631)$	56052361
45	$(271,541,811)$	118901521
51	$(307,613,919)$	172947529

Table 3.5: First five Carmichael cuboid numbers

```
def is_carmichael(n):
    if isprime(n):
        return False
    # Check if n satisfies Fermat's Little Theorem for all a < 100 and relatively prime to
    for a in range(2, min(n, 100)):
        if gcd(a, n) == 1 and pow(a, n-1, n) != 1:
            return False
    return True

def find_first_five_carmichael():
    carmichael_numbers = []
    for n in count(start=2):  # Start checking from the first composite number
        if is_carmichael(n):
            carmichael_numbers.append(n)
        if len(carmichael_numbers) == 25:
            break
    return carmichael_numbers
```

3.4.3 Nearly Square number maneuvers

12 by virtue of having a $12 = 3 \times 4$ (nearly square) representation is referred to in the literature as an *oblong* number while 14 by virtue of just having 1 rectangular representation is referred to as a semi-prime number. Consider the difference of two adjacent square numbers, 4 and 5

$$(5^2-4^2) = (25-16) = 9 = (4+5)(72^2-71^2) \qquad = (5184-5041) = 143 = (71+72)$$

which is easily seen and pictured to be the case for all n by noting the difference of the squares of n and $n+1$ is the sum of the two consecutive numbers, n and $n+1$

$$\begin{aligned} n^2-(n-1)^2 &= n^2-n^2+2n-1 \\ &= 2n-1 = n+(n-1), \end{aligned}$$

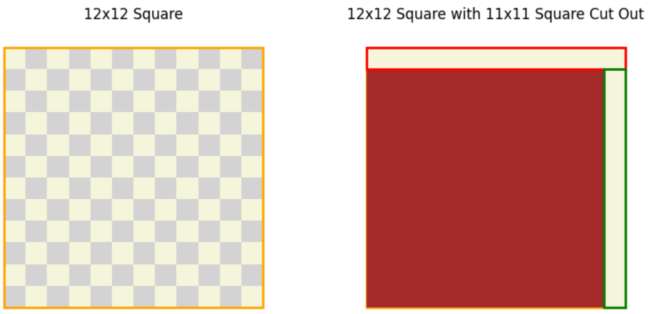


Figure 3.11: Difference of consecutive square numbers.

This is implemented using the patches module in `matplotlib` which provides classes for drawing primarily 2D shapes like rectangles, circles and polygons with the checkerboard being drawn in code[4].

```
for i in range(n):
    for j in range(n):
        if (i+j) % 2 == 0:
            color = "lightgray"
        else:
            color = "beige"
        ax[0].add_patch(patches.Rectangle((i, j), 1, 1, facecolor=color)
```

Then draw an outer rectangle of size $n \times n$ with edgecolor "orange" and no fill

```
ax[0].add_patch(patches.Rectangle((0, 0), n, n,
    facecolor="none", edgecolor="orange", linewidth=2))
```

setting the x/y-axis limits of $ax[0]$ to $[-1, n+1]$ the title of $ax[0]$ to "$n \times n$ Square" and the aspect ratio of $ax[0]$ to equal before Turning off the axis, $ax[0]$:

```
ax[0].set_xlim(-1, n + 1)
ax[0].set_ylim(-1, n + 1)
ax[0].set_title(f"{n}x{n} Square")
ax[0].set_aspect('equal', adjustable='box')
ax[0].axis("off")
```

[4] Difference of Consecutive Squares.ipynb

Difference of Two Squares

Consider $D \in \mathbb{N}$, $D = \{1,3,4,5,7,8,9,\ldots\}$ formed of the differences obtained by subtracting the square of any one integer from the square of another, larger integer,

- The difference between 1^2 and 0^2 is $1-0=1$.
- The difference between 1^2 and 2^2 is $4-1=3$.
- The difference between 2^2 and 3^2 is $9-4=5$.
- The difference between 3^2 and 4^2 is $16-9=7$.
- The difference between 1^2 and 3^2 is $9-1=8$.
- The difference between 3^2 and 5^2 is $25-16=9$.

It is the first difference between two quadratic series, a^2-b^2:
b Series: b^2 where $b=1,2,3,\ldots$ generates the square numbers: 1, 4, 9, 16, 25, …
a Series: a^2 where $a=1+d,2+d,3+d,\ldots$ is set of square numbers, albeit displacement by d. Now every square number is either a multiple of four or one more than a multiple of four, depending on whether it is a square of an even or an odd number, [**asti2001**]. If n is even, n^2 is a multiple of 4 while if n is odd, n^2 is one more than a multiple of 4 so every element of D is one of either three cases: a multiple of four, $4k$ one or three more than its multiple, $4k+1$, $4k+3$:

Case 1: $D(4k)$
n is expressible as $(n/4+1)^2-(n/4-1)^2$, so for example, $n=16$, $k=4$ we have:

$$D(k) = \left(\frac{4k}{4}+1\right)^2 - \left(\frac{4k}{4}-1\right)^2 \text{ and so } D(4) = 5^2-3^2 = 16.$$

Case 2: $D(4k+1)$
n is expressible as $(n+\frac{1}{2})^2-(n-\frac{1}{2})^2$, so for $n=9$, $k=2$:

$$D(k) = \left(4k+1+\frac{1}{2}\right)^2 - \left(4k+1-\frac{1}{2}\right)^2 \text{ and so } D(2) = \left(\frac{19}{2}\right)^2 - \left(\frac{17}{2}\right)^2 = 18$$

Case 3: $D(4k+3)$
n is again expressible as $\left(n+\frac{1}{2}\right)^2-\left(n-\frac{1}{2}\right)^2$ so for $n=15$, $k=3$:

$$D(k) = \left(4k+3+\frac{1}{2}\right)^2 - \left(4k+3-\frac{1}{2}\right)^2 \text{ and so } D(3) = \left(\frac{31}{2}\right)^2 - \left(\frac{29}{2}\right)^2 = 30.$$

For D [5] to represent a difference of squares, a and b must be distinct; hence, $b > a$. Specifically, given two integers a and b where $a > b$, we can expect:

- $4k$ category is the most numerous since the square of an even number is always a multiple of 4, any difference a^2-b^2 where both a and b are even will fall into this category. There are more combinations of even numbers that can produce differences of the form $4k$.
- $4k+1$ category should be less numerous because it requires one of the numbers, either a or b, to be even and the other to be odd as the square of an even number is a multiple of 4, while the square of an odd number is of the form $4m+1$.

[5] The differences D can be either odd or even, depending on the values of b and b. Notably, when b and a are consecutive integers, D as we have just seen is always odd because it represents the difference between consecutive square numbers, which always yields an odd number. Over the long run, one might expect to observe a prevalence of more odd numbers than even numbers among the differences as each time a and b are consecutive, the resulting difference is odd, and as a increases, there are numerous instances where a and b are consecutive. In contrast, for D to be even, a and b need to be further apart, which happens less frequently than them being consecutive.

- $4k+3$ category is expected to be the least numerous since it requires both a and b to be odd. Given that the square of an odd number is $4m+1$, the difference $(4m+1)-(4n+1)$ simplifies to $4(m-n)$, a multiple of 4 we should observe $4k+3$ only when there is an additional 3 involved in the difference.

The code, 4kraceDifferenceOfTwoSquares.ipynb delivers the following stacked histogram chart:

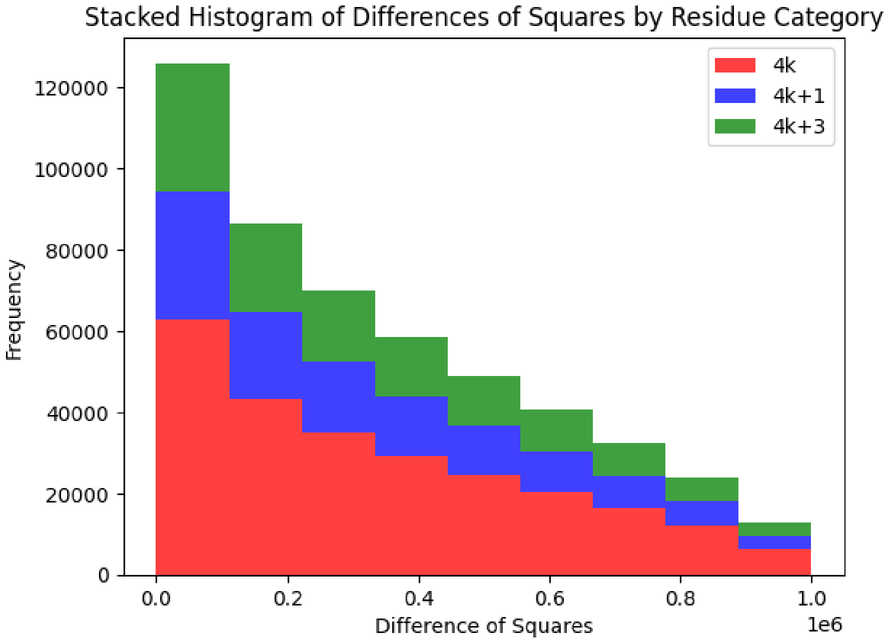


Figure 3.12: Stacked Histogram of Differences of Squares by Residue Category.

`plot_stacked_histogram` plots an absolute frequency stacked histogram:

```
def plot_stacked_histogram(residues, bins=10):
    min_edge = min(min(residues['4k']), min(residues['4k+1']), min(residues['4k+3']))
    max_edge = max(max(residues['4k']), max(residues['4k+1']), max(residues['4k+3']))
    bin_edges = np.linspace(min_edge, max_edge, bins)
    data = [residues['4k'], residues['4k+1'], residues['4k+3']]
    colors = ['red', 'blue', 'green']
    labels = ['4k', '4k+1', '4k+3']
    plt.hist(data, bins=bin_edges, stacked=True, color=colors, label=labels, alpha=0.75)
```

taking the data for each residue category and plotting them in stacks on the same bin, with the height of each colored segment representing the absolute number of occurrences in that bin:

- Determines bin edges spaced between minimum/maximum values across residue categories.
- Prepares the data for stacking by categorizing them into `'4k'`, `'4k+1'`, and `'4k+3'`.
- Plots the histogram using the `plt.hist` function with the `stacked=True` parameter.

`plot_proportional_stacked_histogram` overleaf presents a proportional view in which the stack's height represents the percentage contribution of that category to the bin's total, normalized such that the sum of the stacks in each bin is equal to 100%. The bin's width is implicitly defined by the choice of the number of bins parameter, `bins=10`. The function:

- Flattens the list of all residue categories to determine the bin edges, which are again linearly spaced between the minimum and maximum values.
- Calculates weights for normalization.
- Plots weighted histogram using `plt.hist` with proportional frequency weights.

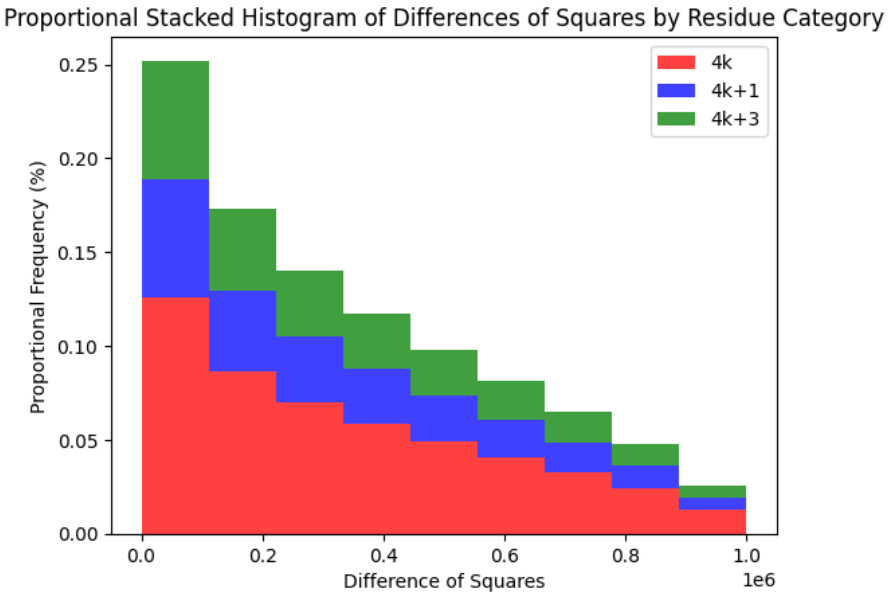


Figure 3.13: Proportional Stacked Histogram of Differences of Squares by Residue Category.

```
def plot_proportional_stacked_histogram(residues, num_bins):
    all_values = [val for sublist in residues.values() for val in sublist]
    min_edge = min(all_values)
    max_edge = max(all_values)
    bin_edges = np.linspace(min_edge, max_edge, num_bins)
    total_counts = len(all_values)
    weights_4k = np.ones_like(residues['4k']) / total_counts
    weights_4k1 = np.ones_like(residues['4k+1']) / total_counts
    weights_4k3 = np.ones_like(residues['4k+3']) / total_counts
    plt.hist([residues['4k'], residues['4k+1'], residues['4k+3']], bins=bin_edges,
             weights=[weights_4k, weights_4k1, weights_4k3],

             stacked=True, color=['red', 'blue','green'],

             label=['4k', '4k+1', '4k+3'], alpha=0.75)
```

While one function counts differences, the other delivers the percentage each category contributes to the total within each bin, thus normalizing the data. The selection of the number of bins affects the width of each bin and, consequently, the distribution of the data points within them. In the proportional stacking approach, a larger number of bins could lead to a finer granularity in the histogram, while a smaller number may group more data points into wider bins. The ‘alpha‘ parameter, set within the call to `plt.hist`, directly influences the visual output by adjusting the opacity level of the colors. An ‘alpha‘ value of 1 would result in fully opaque colors, while an ‘alpha‘ value closer to 0 increases transparency.

Function Parameter Handling in Python

In Python, functions can be defined with default arguments, allowing them to be called with fewer arguments than the number of parameters specified during function definition:

```
def plot_stacked_histogram(residues, bins=10):
```

This is useful for providing common defaults for functions that may need to be configured differently on every call. While the `bins` parameter[6] is a part of the function signature, the actual function call, `plot_stacked_histogram(residues)`, does not explicitly pass the bins parameter. Rather, `bins` is assigned a default value of 10 which acts as a fallback when the function is called without a second argument. When a function is defined with default arguments, Python creates a function object that contains a tuple representing default values for parameters. Upon function invocation, the arguments passed are matched positionally to the parameters. For any missing arguments, Python retrieves the corresponding default value from the tuple and uses it in the function's execution context.[7]

Diophantine connection

The is directly related to finding integer solutions to the Diophantine equation of the form

$$z^2 - y^2 - k = 0.$$

Given a specific k, we can find z and y such that the difference of their squares is equal to k and characterization of D ensures that we can always find such z and y if k is of the form $4k, 4k+1$, or $4k+3$.

Counting Distinct Numerical Categories for a Given a

Given a natural number a, it is of interest to count various distinct numerical categories derived from the differences of two quadratic series. Specifically, we can consider the cumulative number of primes (P), non-prime odds (O), square-tiled compound rectangles (T), and non-tiled compounds (R), excluding duplicates.

- **Prime Numbers (P)**: count of prime numbers appearing in the sequence.
- **Non-Prime Odd Numbers (O)**: count of distinct non-prime odd numbers, including odd composite numbers and odd perfect squares.
- **Square-Tiled Compound Rectangles (T)**: count of numbers that can be represented as a rectangular grid fully filled with squares of varying sizes.
- **Non-Tiled Compound Numbers (R)**: count of numbers that cannot be represented as fully square-tiled rectangles.

Ensures that each unique number is only counted once in each category, providing clarity in the distribution and prevalence of each category within the sequence. The code difference of two squares.ipynb delivers the following histograms.

3.4.4 Oblong Numbers

Oblong numbers (or *pronic numbers*) are a subset of the rectangular numbers being of the form $n(n+1)$. As such they are the product of two consecutive integers and thus apart from square numbers have the maximal perimeter to area ratio of all numbers. We have that the cumulative sum of the even numbers is an *oblong* number

$$\begin{aligned}
n = 2: &\quad 2+4 = (2+1)^2 - (2+1) = 3^2 - 3 = 9 - 3 = 6 = 2 \times 3 \\
n = 3: &\quad 2+4+6 = (3+1)^2 - (3+1) = 4^2 - 4 = 16 - 4 = 12 = 3 \times 4 \\
n = 4: &\quad 2+4+6+8 = (4+1)^2 - (4+1) = 5^2 - 5 = 25 - 5 = 20 = 4 \times 5,
\end{aligned}$$

[6] `bins` is not a dummy variable, but rather an optional parameter with a default value which when called as `plot_stacked_histogram(residues)`, is automatically assigns the value of 10 to the `bins` parameter inside the function.

[7] Therefore, the absence of an argument for a parameter with a default value does not cause an error. Instead, Python uses the default value, allowing the function to operate as if the value were passed explicitly by the caller.

and generally this is because,

$$\begin{aligned}2+4+6+\ldots+2n &= 2(1+2+3+\ldots+n)\\ &= 2\left(\frac{n(n+1)}{2}\right) = n(n+1).\end{aligned}$$

We can see this using the formula for the sum of an arithmetic series $\left(\frac{n}{2}(2a+(n-1)d)\right)$ where $a=2$ and $d=2$:

$$S(n) = \frac{n}{2}(2a+(n-1)d)$$
$$S(2) = \frac{n}{2}(2*2+(n-1)2) = \frac{n}{2}(4+2n-2) = \frac{n}{2}(2n+2) = n(n+1)$$

Consider now the odd numbers series: $1,3,5,7,9,11,13,15,17,19,21,23,25,27,29$ and note how we can express the cubic numbers as the sum of consecutive odd numbers:

$$\begin{aligned}1^3 &= 1\\ 2^3 &= (3+5)\\ 3^3 &= (7+9+11)\\ 4^3 &= (13+15+17+19)\\ &\vdots\end{aligned}$$

The series of the position p to start the series $p(n) = 1,3,7,13,21,\ldots$ where $21 = (5\times4)+1$ and it is in the form of a quadratic sequence $n(n-1)+1$.

n	$p(n)$	Δp_n	$\Delta^2 p_n$
1	1		
2	3	2	
3	7	4	2
4	13	6	2
5	21	8	2

Where,

$$\begin{aligned}p(n) &= n(n-1)+1\\ \Delta p_n &= n+1\\ \Delta^2 p_n &= 2\end{aligned}$$

We note that if two oblong numbers are neighbors, their product will also be oblong. To see why let the two neighboring oblong numbers be $n(n+1)$and $(n+1)(n+2)$ and their product is:

$$n(n+1)\times(n+1)(n+2) = n(n+2)\times(n+1)^2$$

Now, $(n+1)^2$ is a square number, and multiplying a square by an oblong number makes it oblong. Therefore, the product of neighboring oblong numbers will always be oblong. For other cases, let's consider two non consecutive oblong numbers: $a(a+1)$ and $b(b+1)$ whose product is:

$$a(a+1)\times b(b+1) = ab(a+1)(b+1)$$

For this to be oblong, ab and $(a+1)(b+1)$ should differ by 1 (or in some configurations, one should be the square of the other). By way of example, we have that the oblong multiple 2×210:

$$(1\times2)\times(14\times15) \equiv 2\times210 = 420 = 20\times21$$

that is itself an oblong number as a rearrangement of their prime factors reveals:

$$1\times2\times14\times15 = (1\times2\times7)\times(3\times5) = 14\times15$$

To generalize and find more such examples requires careful factorization or brute-forcing for smaller number ranges to explore such products. We shall use python to do the latter. The table below presents the first few such products.

Multiplication	Result	Oblong
$(1\times2)\times(2\times3)$	12	3×4
$(1\times2)\times(14\times15)$	420	20×21
$(1\times2)\times(84\times85)$	14280	119×120
$(2\times3)\times(3\times4)$	72	8×9
$(2\times3)\times(14\times15)$	1260	35×36
$(2\times3)\times(34\times35)$	7140	84×85
$(3\times4)\times(4\times5)$	240	15×16
$(3\times4)\times(34\times35)$	14280	119×120
$(3\times4)\times(62\times63)$	46872	216×217

Table 3.6: Oblong Multiplications and Results

The Python script below comprises several components:

- **is_oblong(n)**: function that checks if a given number n is oblong or not.
- **find_oblong_products(limit)**: function that computes products of oblong numbers up to a given limit and returns a list of pairs that when multiplied produce another oblong number.
- After obtaining the pairs, the script then prints them in a specific format.

```
def is_oblong(n):
    i = 1
    while i * (i + 1) < n:
        i += 1
    return i * (i + 1) == n
def find_oblong_products(limit):
    oblongs = [i * (i + 1) for i in range(1, limit + 1)]
    results = []
    for i in range(len(oblongs)):
        for j in range(i, len(oblongs)):
            product = oblongs[i] * oblongs[j]
            if is_oblong(product):
                k = 1
                while k * (k+1) < product:
                    k += 1
                results.append(((i + 1, i + 2), (j + 1, j + 2), (k, k+1)))
    return results
pairs = find_oblong_products(1000)
```

The latter part of the script uses the `matplotlib` library to visualize the results, both as a 3D scatter plot and as a bubble chart. The output is the following 3-d graph and bubble chart based on the (l,m,n) co-ordinates representing the $l(l+1)*m(m+1) = n(n+1$ We have coloured the dots

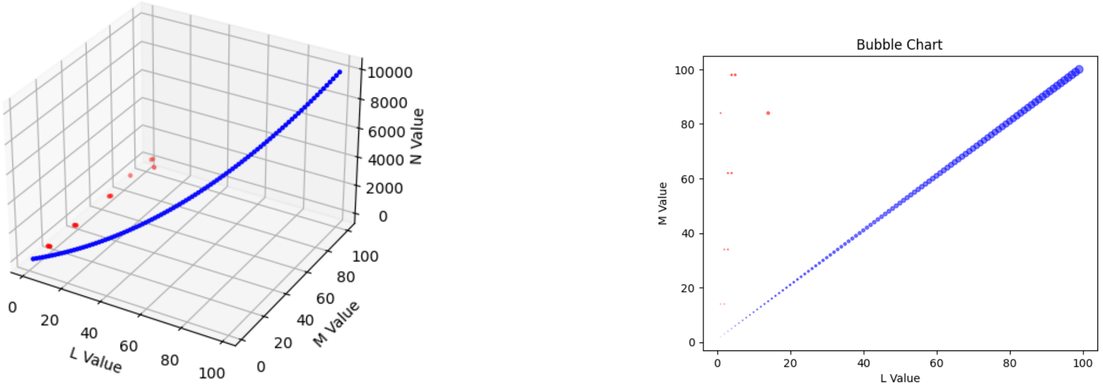


Figure 3.14: Products of Oblong numbers as z-co-ordinates or sized bubbles

representing the product of consecutive oblong numbers as blue while red if otherwise. As such it appears that apart from some initial lower order exceptions only consecutive oblong numbers deliver themselves another oblong number. The reader is invited to investigate further with the code. The charting is performed in the following part of the code:

```
l_values = [p1[0] for p1, _, _ in pairs]
m_values = [p2[0] for _, p2, _ in pairs]
n_values = [p3[0] for _, _, p3 in pairs]
colors = [’blue’ if (abs(l - m) == 1 or abs(m - n) == 1)
else ’red’ for l, m, n in zip(l_values, m_values, n_values)]
fig = plt.figure()
ax = fig.add_subplot(111, projection=’3d’)
ax.scatter(l_values, m_values, n_values, c=colors,
marker=’o’, s=4)  # s=7 sets the point size
ax.set_xlabel(’L Value’)
ax.set_ylabel(’M Value’)
ax.set_zlabel(’N Value’)
plt.show()
largest_n = max(n_values)
bubble_sizes = [50 * (n/largest_n) for n in n_values]
plt.scatter(l_values, m_values, s=bubble_sizes, c=colors, alpha=0.5)
```

1. **Extraction of Values**:
   - `l_values`: This list comprehension extracts the first value of every tuple in `pairs` list.
   - `m_values`: extracts the first value of the second tuple for every entry in `pairs`.
   - `n_values`: extracts the first value of the third tuple from each entry in `pairs`.
2. **Determine colors based on consecutive oblongs**: A list comprehension is used to determine the color of each point. If either of the oblong numbers is consecutive (i.e., their difference is 1), then the color is set to blue; otherwise, it is set to red.
3. **3D Plot**:
   - A new figure is initialized with `plt.figure()` and 3D subplot is added with `add_subplot`.
   - `scatter` method plots the 3D points whose size is set with parameter `s=4`.
   - Labels are set for each of the x, y, and z-axes and the plot is displayed using `plt.show()`.
4. **Bubble Chart**:

- The largest value in `n_values` is determined.
- Bubble sizes are computed based on the ratio of each value in `n_values` to the largest value, scaled by a factor of 50.
- The `scatter` method plots the bubble chart with sizes determined by the previously computed bubble sizes.

3.4.5 Cumulative sum chart of Oblong Numbers

The sum of the reciprocals of the first n oblong numbers is:

$$\sum_{k=1}^{n} \frac{1}{k(k+1)}$$

We can split each fraction using partial fraction decomposition:

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

Using this identity, we can expand and then collapse our sum:

$$\sum_{k=1}^{n} \left(\frac{1}{k} - \frac{1}{k+1}\right) = \left(1 - \frac{1}{2}\right) + \left(\frac{1}{2} - \frac{1}{3}\right) + \left(\frac{1}{3} - \frac{1}{4}\right) + \cdots + \left(\frac{1}{n} - \frac{1}{n+1}\right)$$

as it is a telescoping series, where you can see that after the 1, the negative fraction in each term will cancel out the positive fraction in the next term so that we are left with:

$$\sum_{k=1}^{n} \frac{1}{k(k+1)} = 1 - \frac{1}{n+1} = \frac{n+1}{n+1} - \frac{1}{n+1} = \frac{n}{n+1}$$

That the cumulative partial sums of reciprocals of the oblong numbers to 1 as n tend to infinity can be coded and displayed as in Fig.8.8 as a set of stacked chart versus oblong number, n:

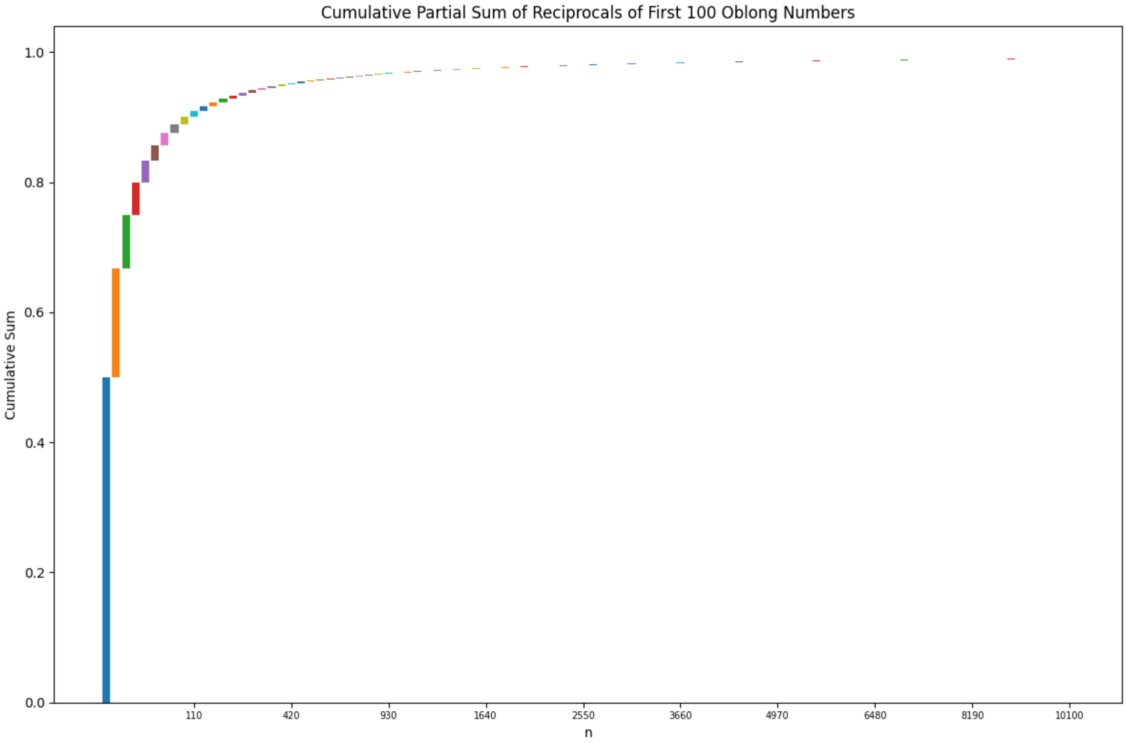


Figure 3.15: Partial sums of first 100 Oblong numbers.

The essential snippet from stackedOblongReciprocals.ipynb follows.

```
def stacked_bar_chart(n, scale='linear'):
    oblong_numbers = generate_oblong_numbers(n)
    reciprocals = [1/num for num in oblong_numbers]
    cumsums = np.cumsum(reciprocals)
    fig, ax = plt.subplots(figsize=(12, 8))
    for i in range(n):
        if i == 0:
            ax.bar(i + 1, reciprocals[i])
        else:
            ax.bar(i + 1, reciprocals[i], bottom=cumsums[i-1])
    if scale == 'log':
        ax.set_xscale('log')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
```

`stacked_bar_chart` provides a choice of scale and readability of axis ticks for large values of n:

- `generate_oblong_numbers(n)` is called to generate the first n oblong numbers.
- a list comprehension calculates the reciprocals of these oblong numbers.
- `np.cumsum` is used to find the cumulative sums of the reciprocals.
- A matplotlib figure and axis are created with `plt.subplots`, specifying a figure size.
- A for loop iterates over the range n:
  - For the first index $i = 0$, `ax.bar` is called with the height of the reciprocal.
  - For subsequent indices, `ax.bar` is called with the additional `bottom` parameter set to the previous cumulative sum.
- The x-axis is optionally set to a logarithmic scale with `ax.set_xscale('log')`.
- Axis labels are rotated 45 degrees for clarity with `plt.xticks(rotation=45)`.

3.5 Tiled Rectangles and Electrical Circuits

Formation of the System of Equations

[11] asks us to consider an electrical circuit derived from the square tiling of a rectangle, with resistors of 1 ohm each. The circuit is driven by a potential difference and can be represented by the following system of equations, based on Kirchhoff's laws and Ohm's law:

1. The current entering the source node is equal to the sum of currents leaving it. For Node 1, this is expressed as:

$$XI_1 = 8 = I_{(1,0)} + I_{(1,2)} + I_{(1,3)} = (V_1 - V_0) + (V_1 - V_2) + (V_1 - V_3) \tag{3.7}$$

2. The sum of currents entering a node is equal to the sum of currents leaving it. For Node 0, this is expressed as:

$$I_{(1,0)} + I_{(2,1)} = I_0 = 8 \tag{3.8}$$

3. The voltage differences around a closed loop sum to zero. For the loop involving Nodes 1, 2, and 3, this is expressed as:

$$(V_1 - V_2) + (V_3 - V_2) = (V_2 - V_0) \tag{3.9}$$

By rearranging and aligning the coefficients of V_1, V_2, and V_3 from the above equations, we can form the matrix A as follows: From Equation (3.7):

$$8 = 3V_1 - V_2 - V_3$$

From Equation (3.8):

$$8 = V_1 + V_2$$

From Equation (3.9):

$$0 = V_1 - 4V_2 + V_3$$

Thus, the matrix A coefficients of the system of three linear equations and the vector b are formed as:

$$A = \begin{pmatrix} 3 & -1 & -1 \\ 1 & 1 & 0 \\ 1 & -3 & 1 \end{pmatrix}, \quad b = \begin{pmatrix} 8 \\ 8 \\ 0 \end{pmatrix}$$

Solving the system $AV = b$ provides the voltages V_1, V_2, and V_3 at each node in the circuit. From the electrical circuit can be conceptualized the square tiling of a rectangle, specifically a 40 = 5x8 rectangle. Modelled as a weighted graph with resistors, each of 1 ohm, driven by a potential difference V the circuit has a current $I = 8A$ and a full potential difference of $V_1 = 5V$. The number of resistors corresponds to the number of squares in the filled rectangle. Using Kirchhoff's law of conservation of charge, which states that the flow of current into a node is equal to the current flowing out, and Ohm's law ($V = IR$) with all resistors being 1 ohm, the voltage differences between nodes are the same as the currents flowing between nodes.

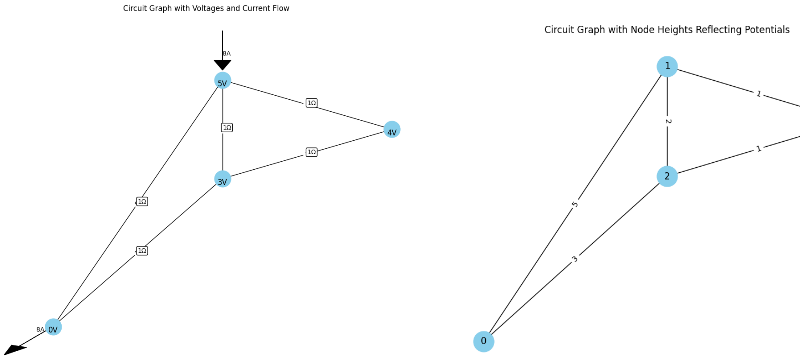

Circuit Graph with Voltages and Current Flow
8A
5V
1Ω
1Ω
4V
1Ω
3V
1Ω
1Ω
8A
0V
Circuit Graph with Node Heights Reflecting Potentials
1
1
2
3
1
2
5
3
0

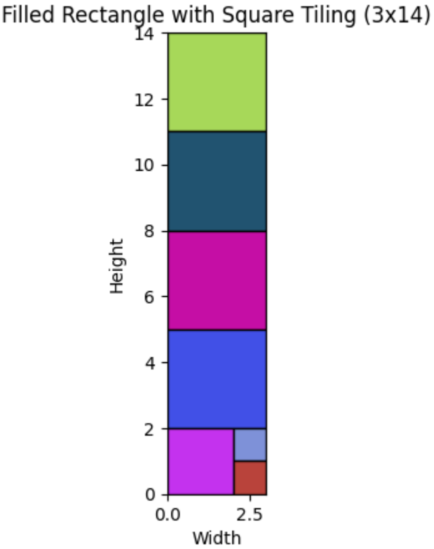
Figure 3.16: Electrical Circuit and Graph of square tiled 40.

The circuit is constructed with nodes and edges representing the connections between them, such as edges (0,1), (0,2), (2,3), etc. The solution of the system $AV = b$ provides the voltages at each node, given the initial conditions and the arrangement of the circuit.

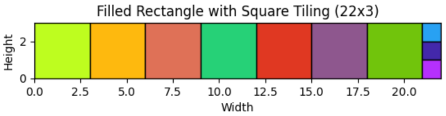
The circuitGraphrectangleFill.ipynb creates the circuit diagram and graphs tha t map to the filling of the fibonacci rectangle

3.5.1 filling rectangles by divisor squares

$$\begin{aligned} 42 &= 3 \times 14 \\ &= 3^2 + 3 \times 11 \\ &= 3^2 + 3 \times (3+8) \\ &= 3^2 + 3^2 + 3 \times 8 \\ &= 2 \cdot 3^2 + 3 \times (3+5) \\ &= 3 \cdot 3^2 + 3 \times 5 \\ &= 3 \cdot 3^2 + 3 \times (3+2) \\ &= 4 \cdot 3^2 + (2+1) \times 2 \\ &= 4 \cdot 3^2 + 2^2 + 2 \cdot 1^2 \end{aligned}$$

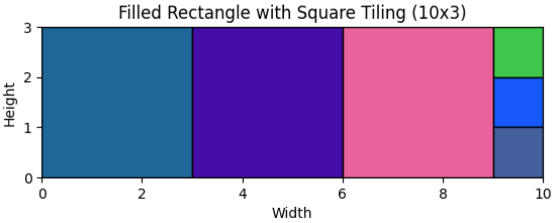


$$\begin{aligned} 66 &= 3 \times 22 \\ &= 3^2 + 3 \times 19 \\ &= 3^2 + 3 \times (3 \times 6 + 1) \\ &= 3^2 + 3^2 \times 6 + 3 \times 1 \\ &= 3^2 + 6 \times 3^2 + 3 \\ &= 7 \times 3^2 + 3 \times 1^2 \end{aligned}$$



But finding those tilings by squares that tile the rectangle completely is not so easy. Consider the (incomplete) set of possible square dessication of 30:

$$\begin{aligned}30 &= 1\cdot 2^2 + 1\cdot 1^2 + 1\cdot 5^2\\ &= 1\cdot 2^2 + 26\cdot 1^2\\ &= 3\cdot 1^2 + 3\cdot 3^2\\ &= 12\cdot 1^2 + 2\cdot 3^2\\ &= 21\cdot 1^2 + 1\cdot 3^2\\ &= 30\cdot 1^2\\ &= 5\cdot 1^2 + 1\cdot 5^2\end{aligned}$$



We can see that the following code dissectingRectangles.ipynb does not pick the last possible sum as the fully tiled rectangle:

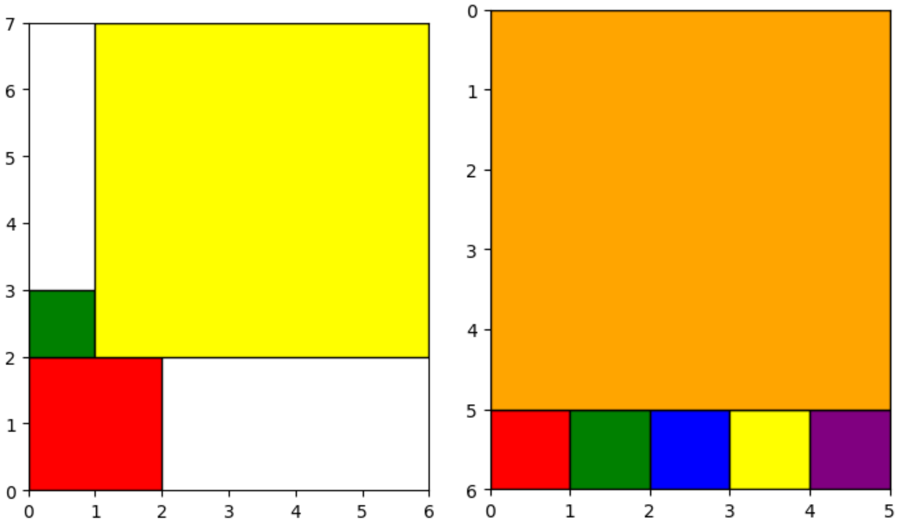


Figure 3.17: Badly dessicated tiled rectangle of 30.

In order to systematically fill the rectangle we need a way of systematically ordering the squares and aborting dead ends and this is achived in backtrackFill.ipynb

Rationale Behind the Ordered Set U in the Tile Filling Algorithm

The tile filling algorithm for a rectangle, such as one with dimensions 399 = 19x21, involves a systematic approach to arranging squares within the rectangle. The ordered set U plays a crucial role in this process. The rationale behind U can be explained as follows:

1. **Defining the Coordinate System:** In the context of the algorithm, each square is identified by its lower-left corner coordinates (X,Y) and its size. The ordered set U is used to define a systematic approach to how these squares are considered for placement within the rectangle.
2. **Ordering of Points:** In U, a point (X,Y) precedes another point (X',Y') if either:
   - The sum of the coordinates of (X,Y) is less than that of (X',Y'), i.e., $X+Y < X'+Y'$. This implies prioritizing points closer to the origin of the coordinate system.
   - If $X+Y = X'+Y'$, then the point with the lower Y value is given precedence, i.e., $Y < Y'$. This rule further orders points that are equidistant from the origin.
3. **Ordering of Squares:** A square $[(X,Y),L]$ precedes another square $[(X',Y'),L']$ in the set U if either:
   - The point (X,Y) precedes (X',Y') in the set of points. This ensures that squares are

  considered in an order that starts from the bottom left of the rectangle and progresses upwards and rightwards.
  - If $(X,Y) = (X',Y')$, then the square with the smaller size L is given precedence. This ensures that, for the same starting point, smaller squares are considered first, allowing for a more granular and flexible filling of the space.

4. **Backtracking Implementation:** In the context of the backtracking algorithm, this ordered set U is essential for systematically exploring potential square placements. The algorithm starts with larger squares and tries to place them in the earliest position available according to the order defined in U. If the placement leads to a dead end where no further squares can be placed, the algorithm backtracks, removes the last placed square, and tries the next possible square in the ordered set.
5. **Efficiency and Coverage:** This ordered approach ensures that the algorithm efficiently covers the rectangle space. By starting from the bottom left and moving upwards and rightwards, and by considering smaller squares for the same starting point, the algorithm can adaptively fill in smaller gaps that larger squares might leave behind.
6. **Goal of the Algorithm:** The ultimate aim is to fill the entire rectangle with squares of varying sizes without overlapping and ensuring all spaces are covered. The ordered set U is instrumental in achieving this by providing a structured way to consider all potential placements of squares.

The following function is an example of a backtracking algorithm, systematically exploring all possible placements of squares and retreating when a dead-end is reached. It is performed in backtrackFill.ipynb

```
def find_solution(board, squares, result, start_row=0, start_col=0):
    for row in range(start_row, len(board)):
        for col in range(start_col if row == start_row else 0, len(board[0])):
            if not board[row][col]:
                for size in squares:
                    if is_valid_placement(board, row, col, size):
                        place_square(board, row, col, size, True)
                        result.append(((col, len(board) - row - size), size))
                        if find_solution(board, squares, result, row, col):
                            return True
                        # Backtrack
                        place_square(board, row, col, size, False)
                        result.pop()
                return False  # No valid square placement found for this cell
    return True  # All cells are filled
```

Description of the `find_solution` Function

The `find_solution` function is a recursive backtracking algorithm designed to fill a rectangular grid with squares of varying sizes. The function's mechanism can be described as follows:

1. The function iterates over each cell in the grid, starting from the specified `start_row` and `start_col`. It searches for the first empty cell in the grid.
2. For each empty cell found, the function tries to place a square of each possible size, starting from the largest. The placement of a square is deemed valid if it fits within the grid boundaries and does not overlap with previously placed squares.
3. If a square can be placed, the function:
   - Marks the cells covered by the square as filled.

   - Adds the square's position and size to the `result` list. The position is adjusted to ensure correct Y-coordinate representation.
   - Recursively calls itself to attempt filling the remaining part of the grid.
4. If the recursive call successfully fills the grid, the function returns `True`, indicating a successful tiling.
5. If placing a square of any size at the current cell does not lead to a solution, the function backtracks:
   - The last placed square is removed, and its cells are marked as empty again.
   - The function returns `False`, triggering further backtracking in previous recursive calls.
6. The process continues until all cells are filled or no valid placement is found for the initial cells, in which case the function returns `False`, indicating no solution.

3.6 Number Classification

Numbers are often studied for their divisibility properties, specifically the count and nature of their divisors. We are in need of some further standard clarifying definitions.

1. **Perfect Numbers:** A number is perfect if the sum of its proper divisors (excluding itself) equals the number itself. Example: 28 (divisors: 1, 2, 4, 7, 14; sum = 28)
2. **Semi-Perfect Numbers:** A number is semi-perfect if it is the sum of some of its proper divisors. Example: 12 (divisors: 1, 2, 3, 4, 6; sum of 1, 2, 3, 6 = 12)
3. **Abundant Numbers:** A number is abundant if the sum of its proper divisors exceeds the number itself. Example: 12 (divisors: 1, 2, 3, 4, 6; sum = 16)
4. **Deficient Numbers:** A number is deficient if the sum of its proper divisors is less than the number itself. Example: 8 (divisors: 1, 2, 4; sum = 7)
5. **Weird Numbers:** A number is weird if it is abundant but not the sum of any combination of its proper divisors. Example: 70 (divisors: 1, 2, 5, 7, 10, 14, 35; sum = 74, but no subset sums to 70)

The headers in table of these properties for the first 20 integers have the following meaning:

1. **Number** n: The integer in consideration.
2. **Prime factors of** n: The prime numbers that divide n.
3. **Average of all divisors,** $P(n)$: The mean of all divisors of n.
4. $\frac{P}{A}$ **where** $A = m \times n$: m is the number of divisor pairs of n.
5. **Sum of proper divisors,** S_k: The sum of all divisors of n excluding n itself.
6. **Classification based on** S_k: Categorization into deficient, semi-perfect-perfect, semi-perfect abundant, or weird based on the sum of proper divisors.

n	Prime Factors	$P(n)$	P/A	S_k	Classification
1	[]	1.00	N/A	0	deficient
2	[2]	1.50	0.75	1	deficient
3	[3]	2.00	0.67	1	deficient
4	[2, 2]	2.33	0.58	3	deficient
5	[5]	3.00	0.60	1	deficient
6	[2, 3]	3.00	0.25	6	semi-perfect-perfect
7	[7]	4.00	0.57	1	deficient
8	[2, 2, 2]	3.75	0.23	7	deficient
9	[3, 3]	4.33	0.48	4	deficient
10	[2, 5]	4.50	0.23	8	deficient
11	[11]	6.00	0.55	1	deficient
12	[2, 2, 3]	4.67	0.13	16	semi-perfect abundant
13	[13]	7.00	0.54	1	deficient
14	[2, 7]	6.00	0.21	10	deficient
15	[3, 5]	6.00	0.20	9	deficient
16	[2, 2, 2, 2]	6.20	0.19	15	deficient
17	[17]	9.00	0.53	1	deficient
18	[2, 3, 3]	6.50	0.12	21	semi-perfect abundant
19	[19]	10.00	0.53	1	deficient
20	[2, 2, 5]	7.00	0.12	22	semi-perfect abundant

Table 3.7: 1-20 classified according to sums and product of the proper divisors

Below are a few numbers that lead up to the weird one that is 70 as coded in UlamNumberClassify.ipynb

n	Prime Factors	$P(n)$	P/A	S_k	Classification
60	[2, 2, 3, 5]	14.00	0.04	108	semi-perfect abundant
61	[61]	31.00	0.51	1	deficient
62	[2, 31]	24.00	0.19	34	deficient
63	[3, 3, 7]	17.33	0.09	41	deficient
64	[2, 2, 2, 2, 2, 2]	18.14	0.09	63	deficient
65	[5, 13]	21.00	0.16	19	deficient
66	[2, 3, 11]	18.00	0.07	78	semi-perfect abundant
67	[67]	34.00	0.51	1	deficient
68	[2, 2, 17]	21.00	0.10	58	deficient
69	[3, 23]	24.00	0.17	27	deficient
70	[2, 5, 7]	18.00	0.06	74	weird (70)

Table 3.8: 60-70 classified according to sums and product of the proper divisors

An *abundant number* is defined as a number for which the sum of its proper divisors (excluding itself) is greater than the number itself. For a number to be abundant, its divisors must sum to a value larger than the number. This typically requires the number to have multiple small prime factors, leading to a sufficiently large number of divisors as borne out by this coding of the following histogram.

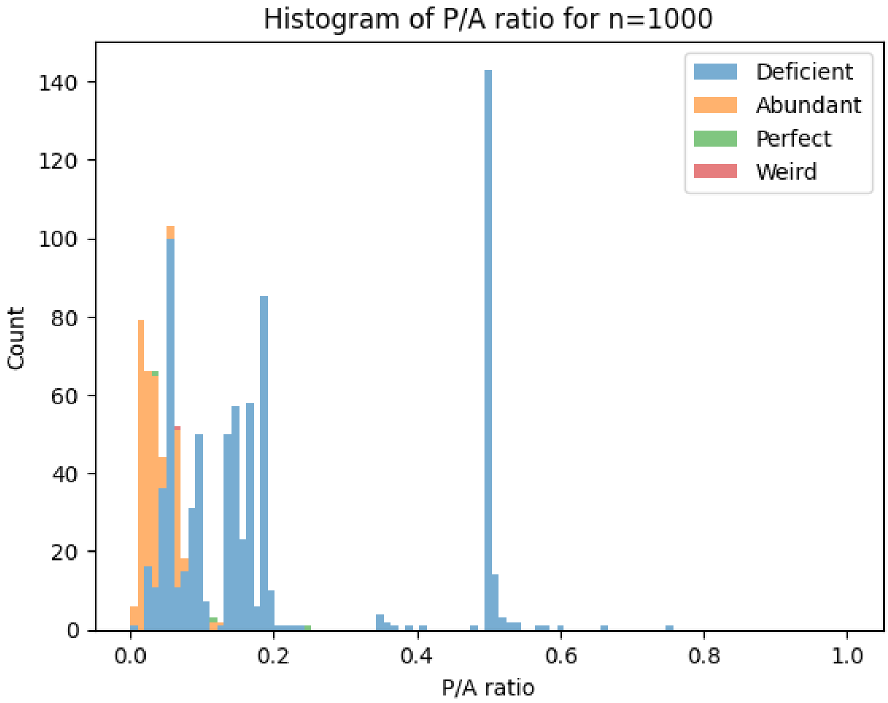


Figure 3.18: Histogram of Deficiency vs Abundant frequencies by Perimeter/Area ratio.

knapsackSemiPerfection.ipynb delivers a list of semi-perfect numbers.

3.6.1 Ranking Metrics

Rank correlation coefficients, proffer a robust comprehension of relationships in ordinal or non-linear data summarising inter-relationships. listingNumber Classification.ipynb delivers the following plot:

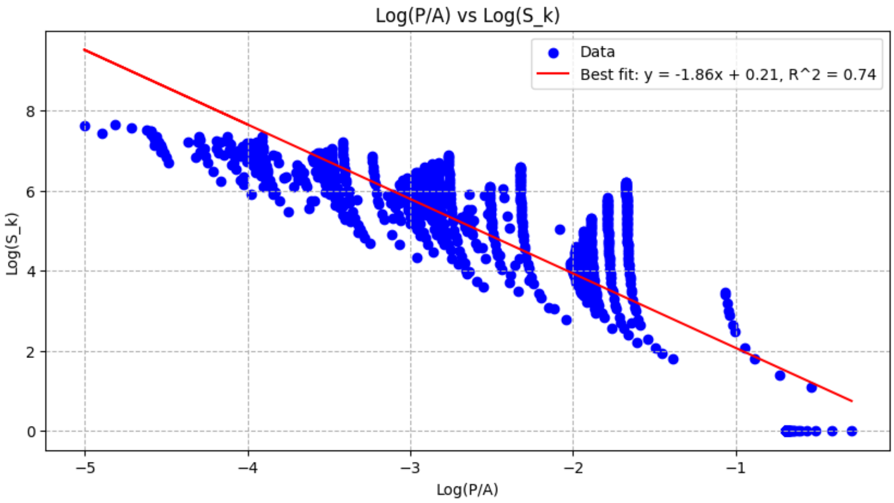


Figure 3.19: Histogram of Deficiency vs Abundant frequencies by Perimeter/Area ratio.

and it also presents the rank correlation coefficients:

```
from scipy.stats import linregress, spearmanr, kendalltau
    spearman_coef, _ = spearmanr(prime_count_values, PA_values)
    kendall_coef, _ = kendalltau(prime_count_values, PA_values)
    print(f"Spearman's rho (P/A vs Count of Prime Factors): {spearman_coef:.3f}")
    print(f"Kendall's tau (P/A vs Count of Prime Factors): {kendall_coef:.3f}")
```

- Spearman's ρ (P/A vs Count of Prime Factors): -0.939
- Kendall's τ (P/A vs Count of Prime Factors): -0.826
- Spearman's ρ (Log(P/A) vs Log(S_k)): -0.836
- Kendall's τ (Log(P/A) vs Log(S_k)): -0.659

These rank correlation coefficients reveal strong inverse relationships between the variables under consideration. Specifically:

- Spearman's ρ and Kendall's τ between the P/A ratio and the count of prime factors ($\rho = -0.939$, $\tau = -0.826$) indicate a very strong negative correlation, implying that a higher P/A ratio is consistently associated with a lower count of prime factors.
- For the log-transformed variables, Spearman's $\rho = -0.836$ and Kendall's $\tau = -0.659$ also suggest a strong inverse relationship, albeit slightly weaker than the non-transformed counterparts.

These correlations underscore the link between the P/A ratio and prime factors' distribution, a cornerstone in number theory. The strong negative correlations suggest that the P/A ratio could serve as a predictive marker for the density of prime factors, offering insights into the structural properties of numbers. Particularly, the robustness of these correlations, even under log transformation, highlights the non-linear dynamics that govern prime distribution and its relationship to other number theoretic properties.

Definition 7 *Rank correlation coefficients* , such as Kendall's τ and Spearman's ρ, provide non-parametric measures to gauge the strength and direction of association between two ranked variables and are especially apt in scenarios where one seeks to fathom the ordinal relationship between two data sets.

Unveiling Relationships: When juxtaposing two disparate metrics like the P/A ratio versus the number of primes, rank correlations can underline if the ordinality in one metric echoes the ranking in its counterpart bringing to light non-linear relationships that might otherwise be overlooked.

- **Gaining Data Insights**: The beauty of rank correlations lies in their ability to provide insights even when metrics operate on disparate scales or dimensions. A salient Spearman's ρ between the P/A ratio and the number of primes could insinuate that numbers with a pronounced P/A ratio usually possess a higher number of prime factors.
- **Probing Deeper into Number Properties**: A significant rank correlation between two ostensibly unrelated number attributes, say the sum of proper factors and the P/A ratio, could hint at structural ties.

Magic Numbers in Atomic theory

Nucleons (protons and neutrons) in atomic nuclei fill in successive shells with the following numbers:

Nucleon shells: $2, 6, 12, 8, 22, 32, 44, \ldots$

Cumulatively, the magic numbers representing total nucleons for completely filled sets of shells are:

Magic numbers: $2, 8, 20, 28, 50, 82, 126, \ldots$

Isotopes with both proton and neutron counts that match the magic numbers, known as *doubly magic isotopes*, exhibit enhanced stability are:

- ^{4}He: 2 protons, 2 neutrons
- ^{16}O: 8 protons, 8 neutrons
- ^{40}Ca: 20 protons, 20 neutrons
- ^{48}Ca: 20 protons, 28 neutrons
- ^{56}Ni: 28 protons, 28 neutrons
- ^{132}Sn: 50 protons, 82 neutrons
- ^{208}Pb: 82 protons, 126 neutrons.

Electrons in atoms, on the other hand[8], fill in shells around the nucleus following:

Electron shells: $2, 8, 18, 32, \ldots$

We can see the appeal to their magic when reflecting that the stability shells are related to the Perfect numbers:

$$6 = 1 + 2 + 3$$
$$28 = 1 + 2 + 4 + 7 + 14$$

as well as less remarkably $12 = 2^2 \times 3$, $18 = 2 \times 3^2$ and $20 = 2^2 \times 5$ the first three square numbers.

[8] Both nucleon and electron shell models show an increasing number of "seats" as we move to higher shells. However, the patterns differ due to the distinct nature of the forces and the particles involved. Electrons fill based on the electromagnetic force and obey the Pauli exclusion principle in a spatial context with spin. Nucleons, on the other hand, interact through the strong nuclear force, which is much stronger and shorter-ranged than the electromagnetic force.

While the shell model concept can be applied to both, the details and the resulting magic/fill numbers are different between the atomic electron configurations and nuclear configurations.

3.7 Semi-Perfect Numbers and the Knapsack Problem

A number's divisors, especially its proper divisors, can lead to various combinations that sum up to the number itself, a principle foundational to problems like the Knapsack problem[9] which is one of the most studied problems in Combinatorial optimization. The Knapsack Problem asks, given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit, and the total value is as large as possible. Formally, for a set of items $i = 1, \dots, n$, each with value v_i and weight w_i, and a maximum weight capacity W, the goal is to maximize $\sum_{i=1}^{n} v_i x_i$, subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0,1\}$. The Knapsack Problem finds relevance in numerous modern contexts, including:

- **Resource Allocation:** In industries, where resources must be optimally allocated within budgetary constraints.
- **Data Compression:** In computer science, especially in methods involving lossless data compression in which the encoding of data allows for the original data to be perfectly reconstructed from the compressed data despite the elimination of redundancy in data.
- **Financial Portfolio Optimization:** In finance, for optimizing the selection of investments under a budget constraint.
- **Cryptographic Algorithms:** Some cryptographic algorithms use knapsack-type problems as a basis for public key cryptography.
- **Logistics and Supply Chain Management:** For maximizing the value of goods transported or stored within capacity limits.

Specifically, given a set of items, each with a weight and a value, the objective is to determine the number of each item to include in a knapsack so that the total weight does not exceed a given limit while maximizing the total value.

Efficient algorithms for partitioning variants of the Knapsack Problem find applications in many discrete optimization contexts. The problem can be described as follows, Let:

- n be the number of items.
- w_i be the weight of the i^{th} item.
- v_i be the value of the i^{th} item.
- W be the maximum weight the knapsack can hold.

Objective: $\max \sum_{i=1}^{n} x_i v_i$

Subject to: $\sum_{i=1}^{n} x_i w_i \leq W \; x_i \in \{0,1\}$ for all $1 \leq i \leq n$

Where x_i is a variable which is 1 if the i^{th} item is included in the knapsack and 0 otherwise. Imagine then a knapsack with a weight capacity of 60 units. You're given a set of weights (or items) corresponding to the proper divisors of 60: $\{1,2,3,4,5,6,10,12,15,20,30\}$ Your challenge is to pick some of these items so that they sum up exactly to 60, the weight capacity of the knapsack. From the divisors of the semi-perfect 60, there is more than one way to fill the knapsack to completely fill the knapsack without exceeding its weight limit, for example:

$$2+4+12+18+24 = 60$$
$$1+2+3+6+12+36 = 60$$

[9] In computer science, the knapsack problem is considered NP-hard, meaning it can get computationally intensive as the number of items (or divisors) increases. The fact that a number like 60 offers a multiplicity, $m = 34$ of solutions to the knapsack problem is a testament to the richness of this computational challenge.

Figure 3.20 presents the 34 ways you can pack such a rucksack as coded in stackingSemiPerfectionsOfAbundants.ipynb and the code for this chart follows.

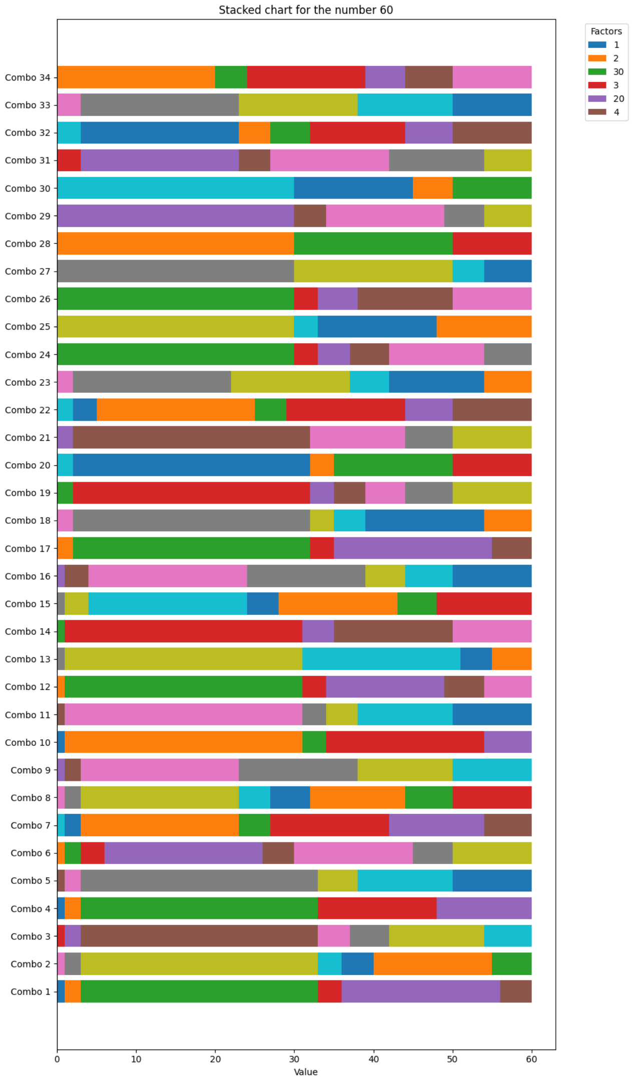


Figure 3.20: Knapsack combinations comprising weights that are the proper factors of 60

```
def stacked_chart_for_abundant(a):
    combinations = semi_perfect_combinations(a)
# Set up the figure and axis
    fig, ax = plt.subplots(figsize=(10, len(combinations) * 0.5))
# We only need to label the number once, so we use this flag
    first_bar = True
    y_pos = range(len(combinations))
        # For each combination, create a stacked bar
    for idx, combo in enumerate(combinations):
        left = 0
        for segment in combo:
            ax.barh(idx, segment, left=left, label=f'{segment}' if first_bar else "")
            left += segment
        first_bar = False
```

4. CuboidNumbers

“Mathematics is the art of giving the same name to different things.’
— *Henri Poincaré*[**Heisenberg**]

The function `is_square(n)` that checks if a given positive integer *n* is a perfect square or not.

```
def is_square(n):
    # Returns True if n is a perfect square, otherwise False
    return int(math.sqrt(n))**2 == n
```

The Python function called `is_non_square(n)` determine whether a given positive integer *n* is a "non-square" number or not.

```
def is_non_square(n):
    if is_square(n):
        return False
    else:
        for i in range(2, int(math.sqrt(n))+1):
            if n % i == 0:
                if is_square(i) or is_square(n // i):
                    return False
        return True
```

A non-square number is a positive integer that is not a perfect square, meaning it cannot be expressed as the square of another integer. For example, 2, 3, 5, 6, 7, 8, 10, and so on are non-square numbers.

Here’s how the code works:

1. The function takes a positive integer *n* as its input.
2. It first checks whether *n* is a perfect square using a function called `is_square(n)`. If *n* is found to be a perfect square, the function returns `False`, indicating that it is not a non-square number.
3. If *n* is not a perfect square, the code enters a loop that iterates through integers starting from 2 up to the square root of *n*.

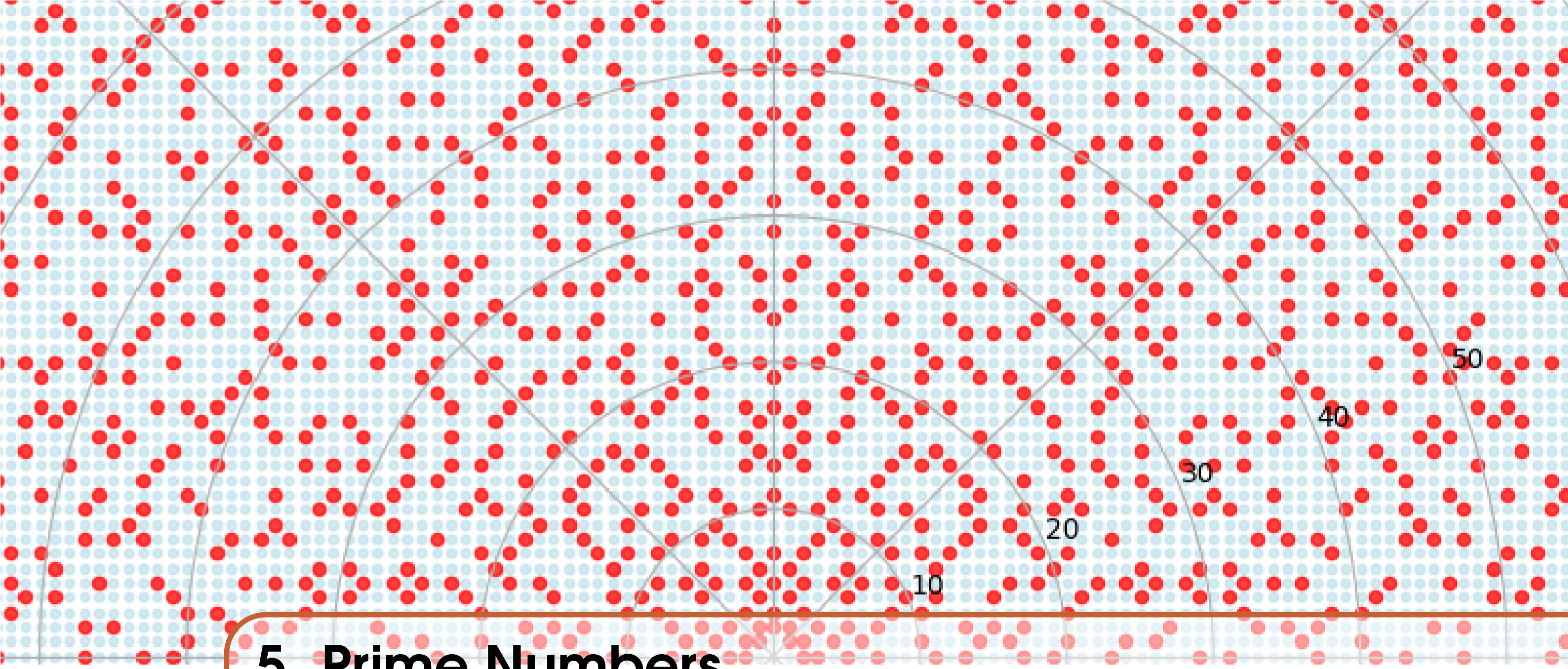
4. For each integer i within this range, it checks if n is divisible by i. If it is divisible, it further checks whether either i or $n//i$ (the integer division result) is a perfect square using the `is_square(n)` function.
5. If either i or $n//i$ is found to be a perfect square, the function returns `False`, indicating that n is not a non-square number.
6. If none of the conditions for being a perfect square are met during the loop, the function returns `True`, indicating that n is indeed a non-square number.
7. the import statement for the `math` module, which provides the `sqrt` function used in the code.

We will say that a cuboid number is any composite number that has a prime factor decomposition that is the same as its radical and in which at maximum has three distinct prime factor divisors.

```
def radical(n):
    factors = set(prime_factors(n))
    return math.prod(factors)

def is_rich_composite(n):
    factors = prime_factors(n)
    return len(set(factors)) < len(factors)
```

Here is the code to determine the surface Area, volume and squared diagonal of a cuboid number:

```
def surface_area(p, q, r):
    return 2 * ((p - 1) * (q - 1) + (p - 1) * (r - 1) + (q - 1) * (r - 1))

def volume(p, q, r):
    return (p - 1) * (q - 1) * (r - 1)

def diagonal_square(p, q, r):
    return (p - 1)**2 + (q - 1)**2 + (r - 1)**2
```



5. Prime Numbers

“The beauty of that is that amongst all the possible forms of understanding, the one form practiced in mathematics is singled out as “true” understanding. Only the employment of a precise, logically consistent language, [the only] language so far capable of formalisation of proofs , can it become possible to lead to true understanding.”
— *Heisenberg in The Debate between Plato and Democritus*[**Heisenberg**]

We start with some definitions:

In order to investigate the various subsets of the integers, that may be of mixed composite and prime form, including the set of polygon, r-gon numbers lets formalize our language a little.

Definition 8 *Modular arithmetic* is clock arithmetic in which you only care about the remainder when dividing numbers. For example, if you have 17 apples and you want to divide them into groups of 5, you’d get 3 groups of 5 apples, and there would be 2 apples left over. So, in modular arithmetic, we only care about the 2 apples left over, not the 3 groups of 5. Prime integers in Python are generated using the modular arithmetic symbol, %

Definition 9 *Modulus (m)*, in modular arithmetic, is like the size of the clock. It tells you where the clock hand wraps around and starts again. For example, in a 12-hour clock, the modulus $m = 12$. For instance, within Python we write 17%5=2, for modulus 5.

Definition 10 *Dividend (d)*, is the number you want to divide. In our example above, the dividend, d is 17.

Definition 11 *Remainder (r)* , in modular arithmetic is is the result of the division. It’s the leftover part that we care about. In 17%5, the remainder is 2.

Table 5.1: Extracted columns from the given table

n	prime factorisation	proper factors
1		$\{1\}$
2	2	$\{1, 2\}$
3	3	$\{1, 3\}$
4	2×2	$\{1, 2, 4\}$
5	5	$\{1, 5\}$
6	2×3	$\{1, 2, 3, 6\}$
7	7	$\{1, 7\}$
8	$2 \times 2 \times 2$	$\{1, 2, 4, 8\}$
9	3×3	$\{1, 3, 9\}$
10	2×5	$\{1, 2, 5, 10\}$
11	11	$\{1, 11\}$
12	$2 \times 2 \times 3$	$\{1, 2, 3, 4, 6, 12\}$
27	$3 \times 3 \times 3$	$\{1, 3, 9, 27\}$
28	$2 \times 2 \times 7$	$\{1, 2, 4, 7, 14, 28\}$
29	29	$\{1, 29\}$
30	$2 \times 3 \times 5$	$\{1, 2, 3, 5, 6, 10, 15, 30\}$
31	31	$\{1, 31\}$
32	$2 \times 2 \times 2 \times 2 \times 2$	$\{1, 2, 4, 8, 16, 32\}$

Back to our rectangular n-ply diagram 3.1, and you may be drawn to the "observation" that in representing consecutive integers as blue-red, the prime numbers, except 2 appear as all blue. Let us expand our `range(1, 31)` from 30 to 100:

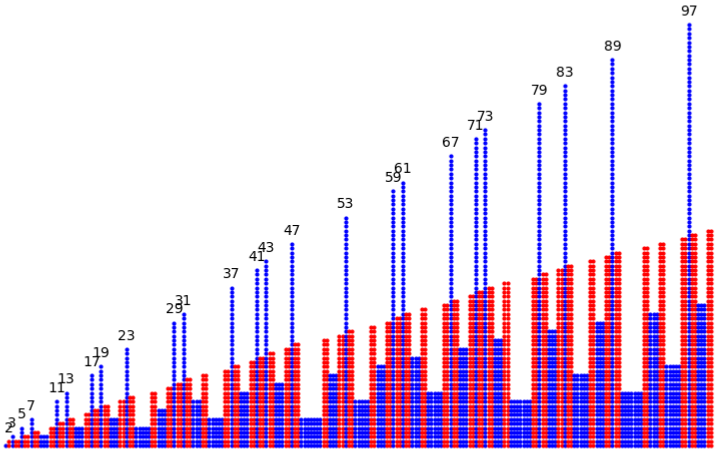


Figure 5.1: Primes and Composites.

That the odd primes remain blue is a result of there being an odd number of intervening composites between consecutive primes and we are led to the natural conjecture:

Theorem 5.0.1 — Prime frequency. is such that there is always an odd number of composites between consecutive primes.

This prompts us to increase the span of numbers over which we code for. Tweak the code to drop labelling the primes, re-sizing the figure, and reducing size of marker:

```
plt.figure(figsize=(15, 3))
..
plt.plot(spacing_x, _ * 2, marker=’o’, markersize=1, color=color)
```

We know that the prime numbers are all odd except 2 so there must always be an odd number of composites between the primes. Indeed we will see that all primes can be generated by either of the pair of generators $6n \pm 1$ and $6n \pm 3$ or $4n+1$ and $4n+3$. We merely ask ourselves at this juncture whether either something further interesting is being revealed here or something rather obvious is being overlooked. That is in the spirit of a:

- **mathematician**, look for a reasoned proof for our idle assertion;
- **data analyst** look for a more efficient way to present the data set;

If the reader is to be the latter I invite you to think or adapt the python code and consider the structure of the composites that interleave the primes.

5.1 Partitioning the primes

There are plenty of ways to skin a cat we can partition the primes according to various grouping schemes. While in themselves nothing deep is revealed such an analysis provides an excuse to perform some nice data analysis. Accordingly various horse races can be set up to determine which delineations holds the more prevalent set of primes in the short and long run

Fermat-4n+1 Type: If a prime number (let’s say "p") leaves a remainder of 1 when divided by 4, it belongs to the "4n+1" type. For example, 5 is a "4n+1" prime because $5\%4 = 1$.

Gauss-4n+3 Type: a prime number of the "4n+3" type is, for example, 7 because $7\%4 = 3$.

All the primes larger than 5 can be grouped by sixes, that is by starting at 7 and 11 we can create two arithmetic series with generators $6n+1$ and $6n+5$ To construct a four horse race consider the split into four arithmetic series of form $p+8k$ for initial primes $p \in \{7, 11, 13, 17\}$.

4n+1
5, 13, 17, 29, 37, 41, 53, 61, 73, 89, 97, 101, 109, 113, 137, 149
4n+3
3, 7, 11, 19, 23, 31, 43, 47, 59, 67, 71, 79, 83, 103, 107, 127, 131, 139, 151

Table 5.2: Categorization of primes up to 150 into 4n+1 and 4n+3 series

6n+1
7, 13, 19, 31, 37, 43, 61, 67, 73, 79, 97, 103, 109, 127, 139, 151
6n+5
5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89, 101, 107, 113, 137, 149

Table 5.3: Categorization of primes up to 150 into 6n+1 and 6n+5 series

Series 7+8k	**Series 11+8k**
7, 31, 71, 103, 127	11, 19, 43, 59, 67, 83, 107, 139
Series 13+8k	**Series 17+8k**
13, 29, 37, 53, 61, 73, 97, 109, 149	17, 41, 73, 89, 97, 113, 137

Table 5.4: Categorization of primes up to 150 into 7+8k, 11+8k, 13+8k, and 17+8k series

5.1.1 Quadratic generators

In fact we need not be so linear with our thinking and consider instead the race between the number of intersections between three quadratic generators $n^2 + c$ and the primes,

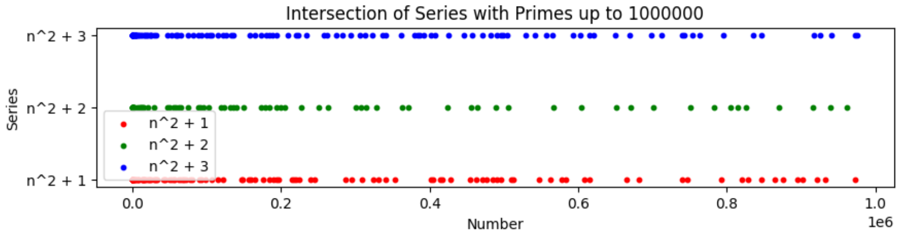


Figure 5.2: Proper Divisors in interval 1-500 with max 500 terms in sequence.

quadraticPrimeRace.ipynb uses the Sieve of Eratosthenes to generate primes

```
def sieve_of_eratosthenes(n):
    primes = []      # Initialize a list to track prime numbers
    # Create a boolean array "prime[0..n]" and initialize all entries as true.
    # A value in prime[i] will finally be false if i is Not a prime, else true.
    prime = [True for i in range(n + 1)]
    p = 2 # Start with the smallest prime number
    # Iterate over each number p starting from 2 to sqrt(n)
    while p * p <= n:
        # If prime[p] is not changed, then it is a prime
        if prime[p] == True:
            # Update all multiples of p
            for i in range(p * p, n + 1, p):
                prime[i] = False
        p += 1
    # Collect all prime numbers
    for p in range(2, n):
        if prime[p]:
            primes.append(p)
    return primes
primes = sieve_of_eratosthenes(10000)
```

The function `find_intersections` is designed to identify the intersection points between a given series of numbers and a list of prime numbers.

```
def find_intersections(series, primes):
    return [n for n in series if n in primes]
max_n = int(np.sqrt(1000000))
series_1 = [n**2 + 1 for n in range(1, max_n)] \ldots
intersections_1 = find_intersections(series_1, primes) \ldots
```

Taking two argument list, `series` and `primes`:

- returning a new list that consists of numbers from `series` that are also found in `primes` achieved through a list comprehension, which iterates over each number `n` in `series` and includes `n` in the output list if `n` is present in `primes`.

The code then generates three different series of numbers (`series_1`, `series_2`, and `series_3`) by applying the formula $n^2 + k$ for $k = 1,2,3$ respectively.

5.1.2 Mersennes Prime Horseplay

Definition 12 *Mersennes Primes* are prime numbers of the form $2^p - 1$ where p is also a prime number. These primes are named after Marin Mersenne, a French monk who studied these numbers in the early 17th century. The Great Internet Mersenne Prime Search (GIMPS), is a distributed computing project focused on finding these rare primes. The largest of the 51 known Mersenne primes is $2^{82,589,933} - 1$, discovered in December 2018. Here is the full list of the first 51 known Mersenne primes:

2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951, 30402457, 32582657, 37156667, 42643801, 43112609, 57885161, 74207281, 77232917, 82589933.

Now consider the split of Mersennes primes into Fermat and Gauss types. The divisibility test for 4 is straightforward: a number is divisible by 4 if its last two digits form a number that is divisible by 4. This is because any number can be expressed as the sum of a multiple of 100 (which is a multiple of 4) and its last two digits and the multiple of 100 doesn't affect the remainder when divided by 4. Accordingly we can see that 521 has remainder 1 given that 4 divides 21 remainder 1. MersennesSplit.ipynb code does the splitting:

```
group_4n1 = []
group_4n3 = []
for prime in mersenne_primes:
    if prime % 4 == 1:
        group_4n1.append(prime)
    elif prime % 4 == 3:
        group_4n3.append(prime)
```

4n+1 Mersenne Primes	4n+3 Mersenne Primes
5, 13, 17, 61, 89, 521, 2281, 3217, 4253, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 132049, 859433, 1398269, 2976221, 3021377, 6972593, 13466917, 30402457, 32582657, 42643801, 43112609, 57885161, 74207281, 77232917, 82589933	3, 7, 19, 31, 107, 127, 607, 1279, 2203, 4423, 86243, 110503, 216091, 756839, 1257787, 20996011, 24036583, 25964951, 37156667

Table 5.5: Categorization of Mersenne primes into 4n+1 and 4n+3

And the Fermat primes, $4n+1$ for the moment at least are winning the Mersenne race.

Mersenne Primes Categorized by Last Digit

Ends in 1:

- 521, 2281, 3217, 9941, 21701, 216091, 2976221, 20996011, 42643801, 57885161, 74207281,

Ends in 3:

- 3, 2203, 4423, 11213, 86243, 110503, 859433, 6972593, 24036583, 82589933

Ends in 7:

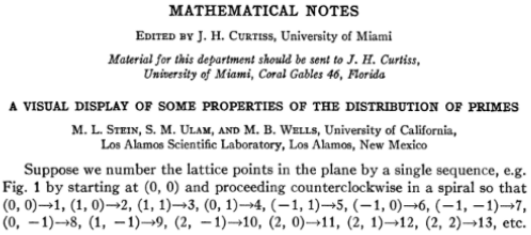
- 7, 107, 127, 607, 19937, 44497, 1257787, 3021377, 13466917, 30402457, 32582657, 37156667, 77232917

Ends in 9:

- 89, 1279, 9689, 23209, 132049, 756839, 1398269, 43112609

5.2 Prime Epigraphing

Epigraphy, the art of interpreting ancient inscriptions, aligns with Euclid's claim that "The laws of nature are but the mathematical thoughts of God." As the discipline uncovers the hidden narratives in stone and metal, so we can look to reveal the patterns of mathematica in constructions such as the Ulam spiral.



MATHEMATICAL NOTES

EDITED BY J. H. CURTISS, University of Miami

Material for this department should be sent to J. H. Curtiss,
University of Miami, Coral Gables 46, Florida

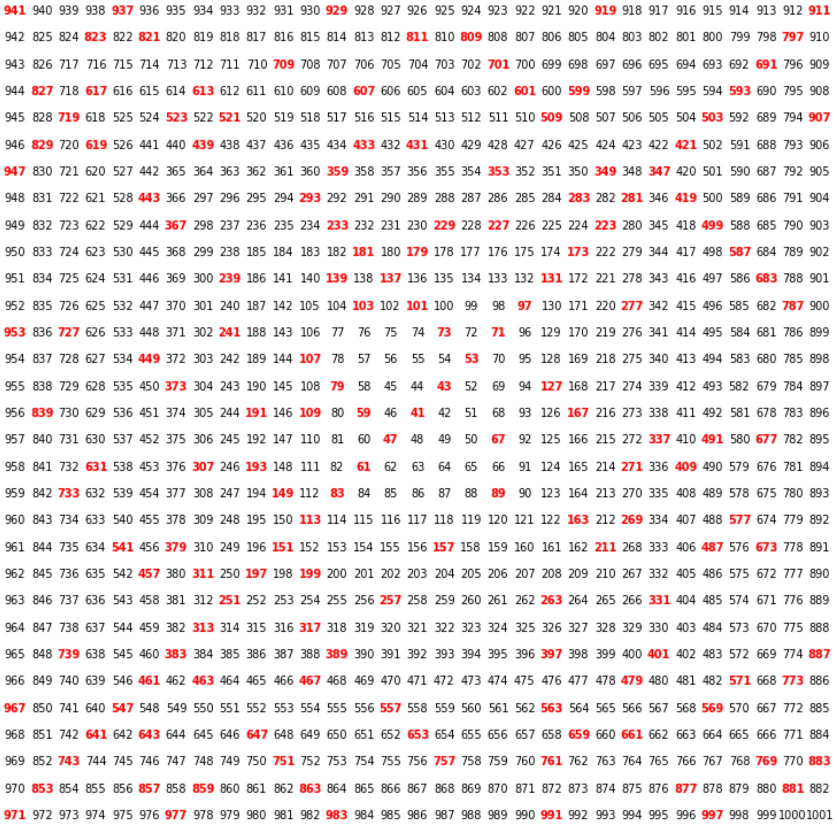
A VISUAL DISPLAY OF SOME PROPERTIES OF THE DISTRIBUTION OF PRIMES

M. L. STEIN, S. M. ULAM, AND M. B. WELLS, University of California,
Los Alamos Scientific Laboratory, Los Alamos, New Mexico

Suppose we number the lattice points in the plane by a single sequence, e.g. Fig. 1 by starting at (0, 0) and proceeding counterclockwise in a spiral so that (0, 0)→1, (1, 0)→2, (1, 1)→3, (0, 1)→4, (−1, 1)→5, (−1, 0)→6, (−1, −1)→7, (0, −1)→8, (1, −1)→9, (2, −1)→10, (2, 0)→11, (2, 1)→12, (2, 2)→13, etc.

Figure 5.3: Ulam's original paper on spiralling primes.

Stanislaw Ulam, known for his work in the Standard Model of physics, suggested a spiralling arrangement of numbers to reveal hidden structures within the prime numbers. In the image below prime numbers are written in red and their affinity to diagonals is almost a cause for pause.



941	940	939	938	**937**	936	935	934	933	932	931	930	**929**	928	927	926	925	924	923	922	921	920	**919**	918	917	916	915	914	913	912	**911**
942	825	824	**823**	822	**821**	820	819	818	817	816	815	814	813	812	**811**	810	**809**	808	807	806	805	804	803	802	801	800	799	798	**797**	910
943	826	717	716	715	714	713	712	711	710	**709**	708	707	706	705	704	703	702	**701**	700	699	698	697	696	695	694	693	692	**691**	796	909
944	**827**	718	**617**	616	615	614	**613**	612	611	610	609	608	**607**	606	605	604	603	602	**601**	600	**599**	598	597	596	595	594	**593**	690	795	908
945	828	**719**	618	525	524	**523**	522	**521**	520	519	518	517	516	515	514	513	512	511	510	**509**	508	507	506	505	504	**503**	592	689	794	**907**
946	**829**	720	**619**	526	441	440	**439**	438	437	436	435	434	**433**	432	**431**	430	429	428	427	426	425	424	423	422	**421**	502	591	688	793	906
947	830	721	620	527	442	365	364	363	362	361	360	**359**	358	357	356	355	354	**353**	352	351	350	**349**	348	**347**	420	501	590	687	792	905
948	831	722	621	528	**443**	366	297	296	295	294	**293**	292	291	290	289	288	287	286	285	284	**283**	282	**281**	346	**419**	500	589	686	791	904
949	832	723	622	529	444	**367**	298	237	236	235	234	**233**	232	231	230	**229**	228	**227**	226	225	224	**223**	280	345	418	**499**	588	685	790	903
950	833	724	623	530	445	368	299	238	185	184	183	182	**181**	180	**179**	178	177	176	175	174	**173**	222	279	344	417	498	**587**	684	789	902
951	834	725	624	531	446	369	300	**239**	186	141	140	**139**	138	**137**	136	135	134	133	132	**131**	172	221	278	343	416	497	586	**683**	788	901
952	835	726	625	532	447	370	301	240	187	142	105	104	**103**	102	**101**	100	99	98	**97**	130	171	220	**277**	342	415	496	585	682	**787**	900
953	836	**727**	626	533	448	371	302	**241**	188	143	106	77	76	75	74	**73**	72	**71**	96	129	170	219	276	341	414	495	584	681	786	899
954	837	728	627	534	**449**	372	303	242	189	144	**107**	78	57	56	55	54	**53**	70	95	128	169	218	275	340	413	494	583	680	785	898
955	838	729	628	535	450	**373**	304	243	190	145	108	**79**	58	45	44	**43**	52	69	94	**127**	168	217	274	339	412	493	582	679	784	897
956	**839**	730	629	536	451	374	305	244	**191**	146	**109**	80	**59**	46	**41**	42	51	68	93	126	**167**	216	273	338	411	492	581	678	783	896
957	840	731	630	537	452	375	306	245	192	147	110	81	60	**47**	48	49	50	**67**	92	125	166	215	272	**337**	410	**491**	580	**677**	782	895
958	841	732	**631**	538	453	376	**307**	246	**193**	148	111	82	**61**	62	63	64	65	66	91	124	165	214	**271**	336	**409**	490	579	676	781	894
959	842	**733**	632	539	454	377	308	247	194	**149**	112	**83**	84	85	86	87	88	**89**	90	123	164	213	270	335	408	489	578	675	780	893
960	843	734	633	540	455	378	309	248	195	150	**113**	114	115	116	117	118	119	120	121	122	**163**	212	**269**	334	407	488	**577**	674	779	892
961	844	735	634	**541**	456	**379**	310	249	196	**151**	152	153	154	155	156	**157**	158	159	160	161	162	**211**	268	333	406	**487**	576	**673**	778	891
962	845	736	635	542	**457**	380	**311**	250	**197**	198	**199**	200	201	202	203	204	205	206	207	208	209	210	267	332	405	486	575	672	777	890
963	846	737	636	543	458	381	312	**251**	252	253	254	255	256	**257**	258	259	260	261	262	**263**	264	265	266	**331**	404	485	574	671	776	889
964	847	738	637	544	459	382	**313**	314	315	316	**317**	318	319	320	321	322	323	324	325	326	327	328	329	330	403	484	573	670	775	888
965	848	**739**	638	545	460	**383**	384	385	386	387	388	**389**	390	391	392	393	394	395	396	**397**	398	399	400	**401**	402	483	572	669	774	**887**
966	849	740	639	546	**461**	462	**463**	464	465	466	**467**	468	469	470	471	472	473	474	475	476	477	478	**479**	480	481	482	**571**	668	**773**	886
967	850	741	640	**547**	548	549	550	551	552	553	554	555	556	**557**	558	559	560	561	562	**563**	564	565	566	567	568	**569**	570	667	772	885
968	851	742	**641**	642	**643**	644	645	646	**647**	648	649	650	651	652	**653**	654	655	656	657	658	**659**	660	**661**	662	663	664	665	666	771	884
969	852	**743**	744	745	746	747	748	749	750	**751**	752	753	754	755	756	**757**	758	759	760	**761**	762	763	764	765	766	767	768	**769**	770	**883**
970	**853**	854	855	856	**857**	858	**859**	860	861	862	**863**	864	865	866	867	868	869	870	871	872	873	874	875	876	**877**	878	879	880	**881**	882
971	972	973	974	975	976	**977**	978	979	980	981	982	**983**	984	985	986	987	988	989	990	**991**	992	993	994	995	996	**997**	998	999	1000	1001

Figure 5.4: Ulam Spiral with primes highlighted in red.

5.2.1 Fermat-Gauss Prime split

Let us recall as **Fermat primes** that subset of primes $\{2,3,5,7,11,\cdots\}$ that can be generated by $4n+1$. We achieve this partition similarly to previously as per the code snippet:

```
def split_primes(n):
    fermat_primes = set()
    other_primes = set()
    for i in range(2, n):
        if is_prime(i):
            if pow(2, i-1, i) == 1:
                if i \% 4 == 1:
                    fermat_primes.add(i)
                else:
                    other_primes.add(i)
    return fermat_primes, other_primes
```

Some coding notes are:

- ⋆ `fermat_primes = set()` creates (and defines) the empty Fermat prime set.
- ⋆ The `if  pow(2, i-1, i) == 1` clause is checks whether $2^{(i-1)}\ mod\ i$ is equal to 1.
- ⋆ The `pow()` function is used to perform modular exponentiation, which calculates the result of raising 2 to the power of `(i-1)` modulo `i`. The mod operation returns the remainder when $2^{(i-1)}$ is divided by i, i.e. to check if i is a Fermat prime, a prime number of the form $2^{(p-1)}$ where p is also a prime.

We illustrate our delineation of the two prime factor types by color coding them in blue and red.

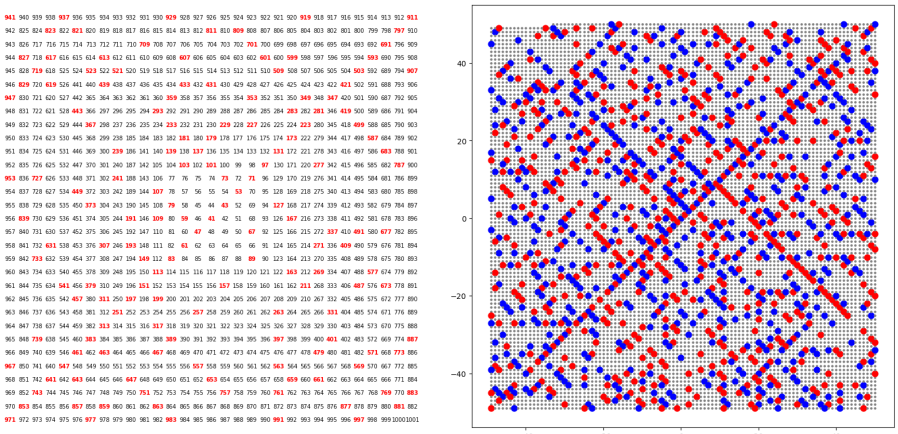


Figure 5.5: Ulam Spiral for Number Persistence and Zap Depth

This is achieved in the code, $smallDotUlamFermat_spiral.ipynb$ by the snippet

```
def ulam_spiral(s, E):
    x, y = 0, 0
    dx, dy = 0, -1
    spiral = {}
    for n in range(s, E+1):
        if is_prime(n):
            spiral[(x,y)] = ’red’
        else:
             spiral[(x,y)] = 'white'
        if x == y or (x < 0 and x == -y) or (x > 0 and x == 1-y):
            dx, dy = -dy, dx
        x, y = x+dx, y+dy
    return spiral
\label{ulamSpiral:code}
```

Some coding points of note here

- `for`-loop structure syntax in which no `next` is required as in visual basic;
- `if-else` clause structures first delivers red or white c-ordinate points depending on whether number is prime and then separately shifts across and up co-ordinate plane.

5.2.2 Twin primes

The distribution and frequency of coupled prime pairs are a subject of much focus. Among these, twin primes , which are pairs of the form $(p, p+2)$, have garnered significant attention. The natural conjecture is that as numbers grow larger, twin primes become less frequent as the likelihood of both p and $p+2$ being prime diminishes with increasing p. Roughly the probability of a randomly chosen odd number p being prime is about $1/\ln(p)$, implying that the joint probability of p and $p+2$ being prime is approximately $1/(\ln(p)\cdot\ln(p+2))$. of the form $p, p+2$. We can just as well

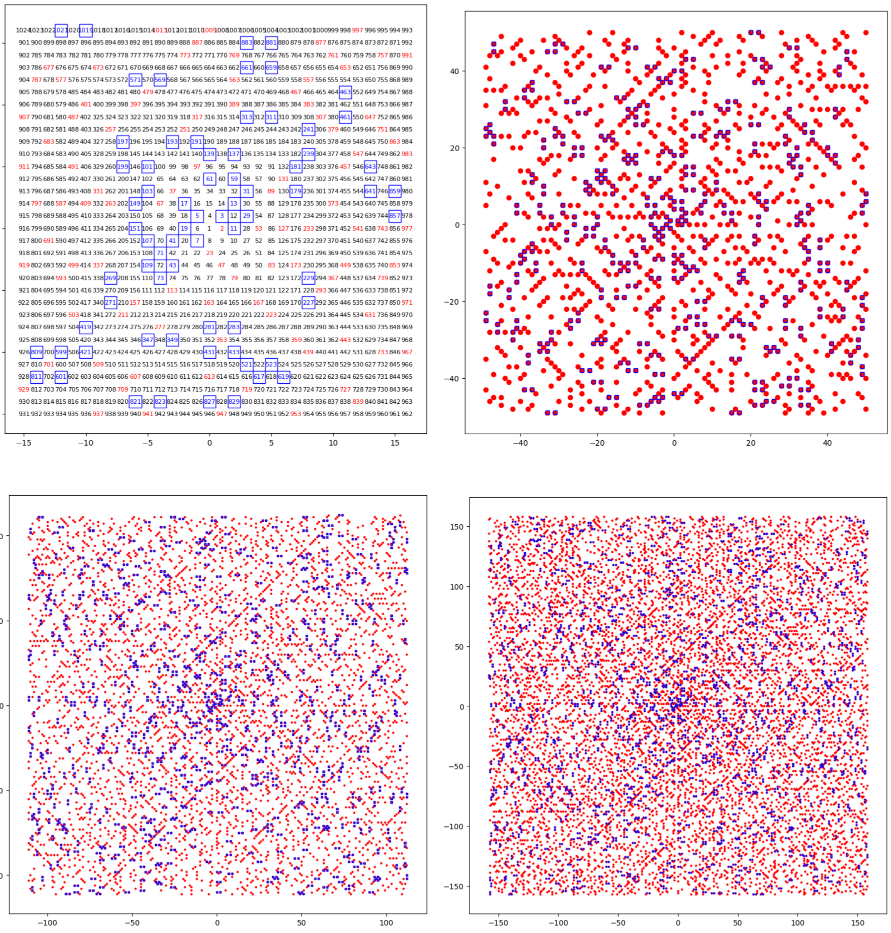


Figure 5.6: Ulam Spiral showing Twin Primes for up to 1000,10000,50000 and 100000.

define coupled primes of the form $p, p+2n$ for any integer n. For larger gaps, $n > 1$ the term "twin primes" is replaced by more general descriptions, such as "cousin primes" for $n = 2$ and "sexy primes" for $n = 3$, or simply "primes with a gap of $2n$" for larger n. We can reasonably expect the frequency of these generalized prime pairs to decrease more rapidly than that of twin primes as n increases. This is because the larger the gap $2n$, the higher the probability that at least one number within the interval $[p, p+2n]$ is composite, a likelihood that escalates with n.

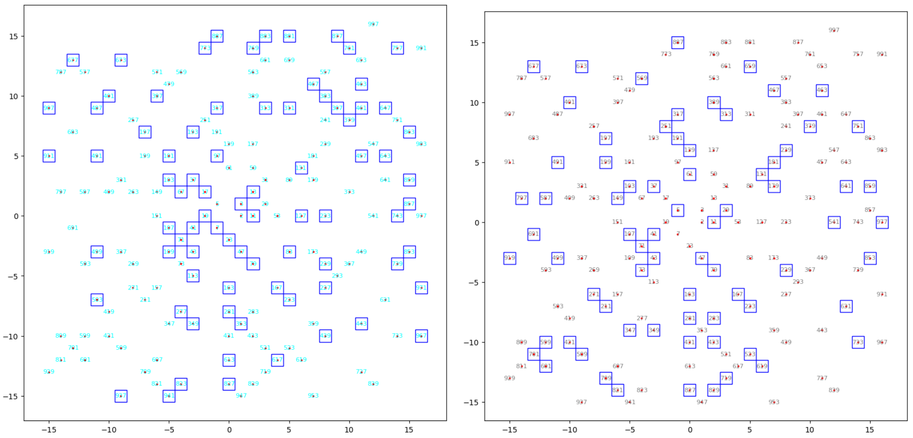


Figure 5.7: Ulam Spiral showing Paired Primes of differences 4 and 32.

A snippet from the code follows.

```
def is_twin_prime(n):
    return is_prime(n) and (is_prime(n-2) or is_prime(n+2))

for coord in spiral:
    if spiral[coord] == 'red':
        ax.scatter(coord[0], coord[1], color='red', s=3)
        if is_twin_prime(s):
            ax.add_patch(plt.Rectangle((coord[0]-0.5, coord[1]-0.5), 1, 1,
                        fill=False, edgecolor='blue', linewidth=1))
        if plot_numbers:
            ax.text(coord[0], coord[1], str(s), color='grey', ha='center',
                    va='center', fontsize=8)
```

5.2.3 Ulam Spiral for Number Persistence

The *persistence* of a number, [10] is defined as the number of steps required to reduce it to a single-digit number by repeated multiplication. For example, for the number 387:

$$3*8*7 = 168 \rightarrow 1*6*8 = 48 \rightarrow 4*8 = 32 \rightarrow 3*2 = 6$$

The persistence of 387 is 4. Below is the Ulam Spiral highlighting the persistence of numbers from 1 to 100.

5.2.4 Ulam Spiral for Zap Depth

The Zap Depth, [10] of a number is the number of steps needed to reach a single-digit number by continuously summing the squares of the digits. For example, the zap depth of 31 is 2, calculated as follows:

$$3^2 + 1^2 = 10 \rightarrow 1^2 + 0^2 = 1$$

Below is the Ulam Spiral highlighting the Zap Depth of numbers from 1 to 100.

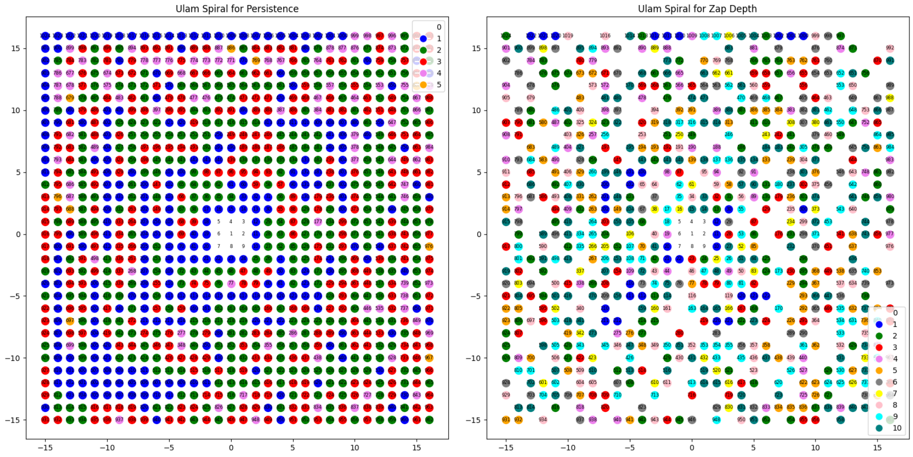


Figure 5.8: Ulam Spiral for Number Persistence and Zap Depth

5.2.5 n-ply Ulam spirals

In light of a our n-ply definition we wish to see how the Ulam spiral appears when we colour numbers according to whether they are 1-ply (prime) as red or a gradient of blue that spans from light to dark with increasing f-ply. When we increase to 50,000 numbers again we can start to see some diagonal lines developing:

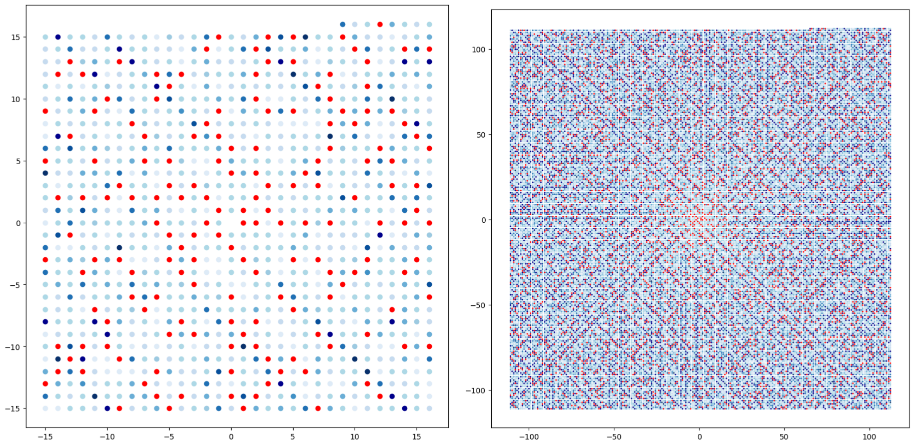


Figure 5.9: Ulam Spiral with graded ply colouring for n=1000 and n=50,000.

Here the variations of colour do not reveal the full variety of f-ply that exist. What seems more natural to describe the variety of rectangles that can be constructed for a number is a simple histogram with bins the f-ply. Let us construct histograms of plies up to a user defined number, n.

As we increase n we see that 2,4 and 8 ply increasingly dominate.

Ranking the top 4 for increasing values of n we have:

1-ply (Prime) numbers numbers are ousted from equal top spot when n=100.

Questions that beg:

- For what value of n do double ply numbers remain the most numerous?

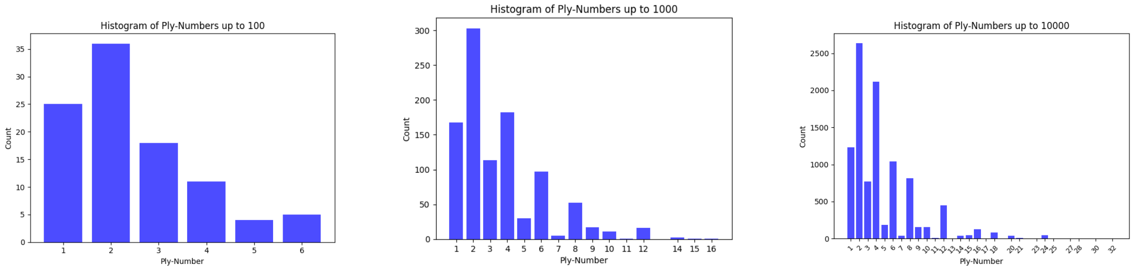


Figure 5.10: Histograms of f-ply of positive integers for up to n=100,1000 and 10,000.

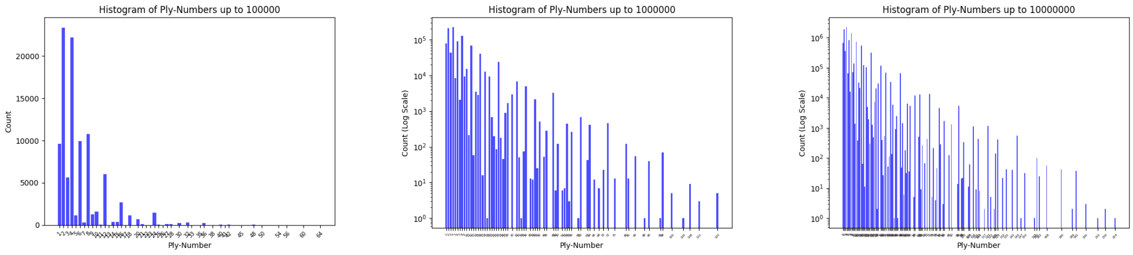


Figure 5.11: f-ply of positive integers for n=100,000 with linear and n=1,000,000 with log axis.

- Why are even -ply numbers apparently more numerous than odd-ply numbers?
- Does the even preference sustain as n tends to infinity?

Here is a snippet from the *failedPrimeGenerators code* used to generate these:

```
E = int(input('Enter ending number: '))
ply_histogram = get_ply_histogram(E)
plt.bar(ply_histogram.keys(), ply_histogram.values(), color='blue', alpha=0.7)
plt.xlabel('Ply-Number')
plt.ylabel('Count')
plt.title(f'Histogram of Ply-Numbers up to {E}')
if len(ply_histogram) > 10:
    plt.xticks(list(ply_histogram.keys()), rotation=45, fontsize=4)
else:
    plt.xticks(list(ply_histogram.keys()))
plt.tight_layout()
```

1. User is prompted to input an ending number, which is stored as E and represents the largest value for which ply-values will be calculated.
2. The function `get_ply_histogram` is called with E as an argument and computes the histogram of ply-values for numbers up to E. The histogram is a dictionary where each key represents a unique ply-value and the associated value represents the frequency of that ply-value.
3. The histogram data is plotted as a bar graph with x-axis the unique ply-values and y-axis the frequencies of these ply-values.
4. The x-axis is labeled as "Ply-Number".
5. A title, "Histogram of Ply-Numbers up to E", is set for the plot.
6. If there are more than 10 unique ply-values in the histogram, the labels are rotated by 45

n	rank 1	rank 2	rank 3	rank 4
10	1	2	-	-
100	2	1	3	4
1,000	2	4	1	3
10,000	2	4	1	3
100,000	2	4	8	6
1,000,000	2	4	8	6
10,000,000	4	2	8	6

Table 5.6: f-ply rankings for increasing values of n

degrees and their font size is reduced to avoid overlap.

7. Finally, the layout of the plot is adjusted to ensure all elements fit well within the figure area.

5.2.6 Perimeter-Area ratio of f-plies

Given our geometric picture of composites it is interesting to investigate the Perimeter/Area ratio of such rectangles. We know that squares of all rectangles have the minimum P/A amongst rectangles. What would Ulam look like if instead of plotting by number of factors we plotted grading by P/A. For n up to 10000 we have the following:

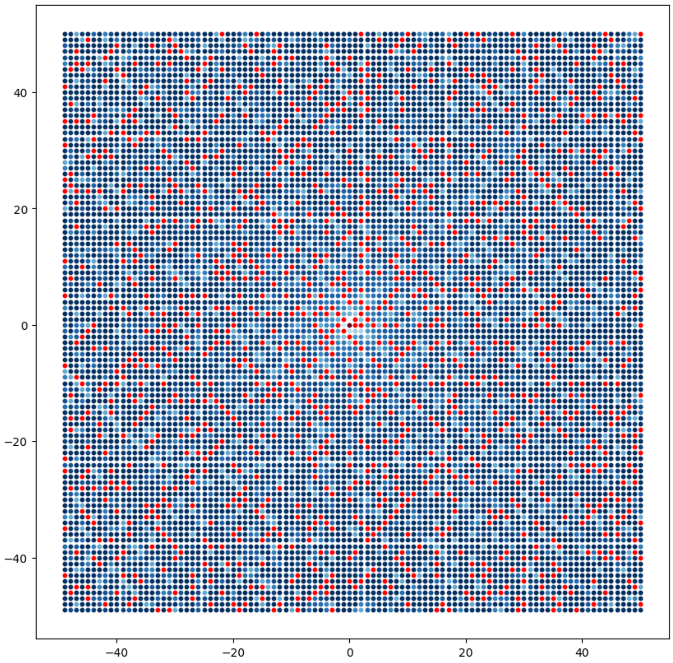


Figure 5.12: Ulam Spiral for n up 10,000 graded by minimum P/A ratio of each rectangle number.

The function that finds the minimum ratio amongst the two factor pairs making the rectangle

```
def min_pa_ratio(n, factor_set):
    min_ratio = float('inf')
    for f1 in factor_set:
        f2 = n // f1
        P = 2 * (f1 + f2)
        A = f1 * f2
        ratio = P / A
        if ratio < min_ratio:
            min_ratio = ratio
    return min_ratio
```

The function `min_pa_ratio` takes two arguments: an integer `n` and a set of factors `factor_set`. Its purpose is to find the minimum perimeter-to-area ratio for rectangles with an area of `n`. The function works as follows:

1. Initialize `min_ratio` to infinity. This variable will store the minimum ratio found.
2. Iterate through each factor `f1` in the `factor_set`.
3. For each `f1`, calculate `f2` as the integer division of `n` by `f1`. This represents the complementary factor such that `f1` × `f2` equals `n`.
4. Calculate the perimeter `P` of the rectangle as $2 \times (\texttt{f1} + \texttt{f2})$.
5. Calculate the area `A` as `f1` × `f2`, which should equal `n`.
6. Compute the perimeter-to-area ratio as `P` divided by `A`.
7. If this ratio is less than the current `min_ratio`, update `min_ratio` with this new ratio.
8. After iterating through all factors, return the `min_ratio`.

5.3 Complex Gaussian Integers and Primes

The Gaussian integers ($\mathbb{Z}[i]$) are a subset of the field of complex numbers of the form $a+bi$, where a and b are integers exemplifying an *integral domain* because they:

- are a *commutative ring* under addition and multiplication.
- maintain the integrity of multiplication, evidenced by the absence of zero divisors.

A ring[1] is an algebraic structure consisting of a set equipped with two binary operations: addition whose archetype are the set of integers $\mathbb{Z}$ with addition and multiplication closed and associative under both operations with an additive identity (0), and every element with an additive inverse.

Definition 13 *Commutative Rings* are sets R which possess operations closed under addition and multiplication beyond the usual integer arithmetic that satisfy the following conditions:

Addition and multiplication are associative and commutative.

2. There exists an additive identity $0 \in R$ such that $a+0=a$ for all $a \in R$.
3. Each element $a \in R$ has an additive inverse $-a \in R$ such that $a+(-a)=0$.
4. Multiplication is distributive over addition: $a \cdot (b+c) = a \cdot b + a \cdot c$ for all $a,b,c \in R$.

The set of Gaussian integers, $\mathbb{Z}[i]$, where i is the imaginary unit, is an example of a commutative ring. The zero divisor condition is broken in the ring $\mathbb{Z}[i]/(5)$ of Gaussian integers modulo 5, as the product of the (non-zero) elements $2+i$ and $3+2i$ is zero $(2+i)(3+2i) \equiv 0 \mod 5$.

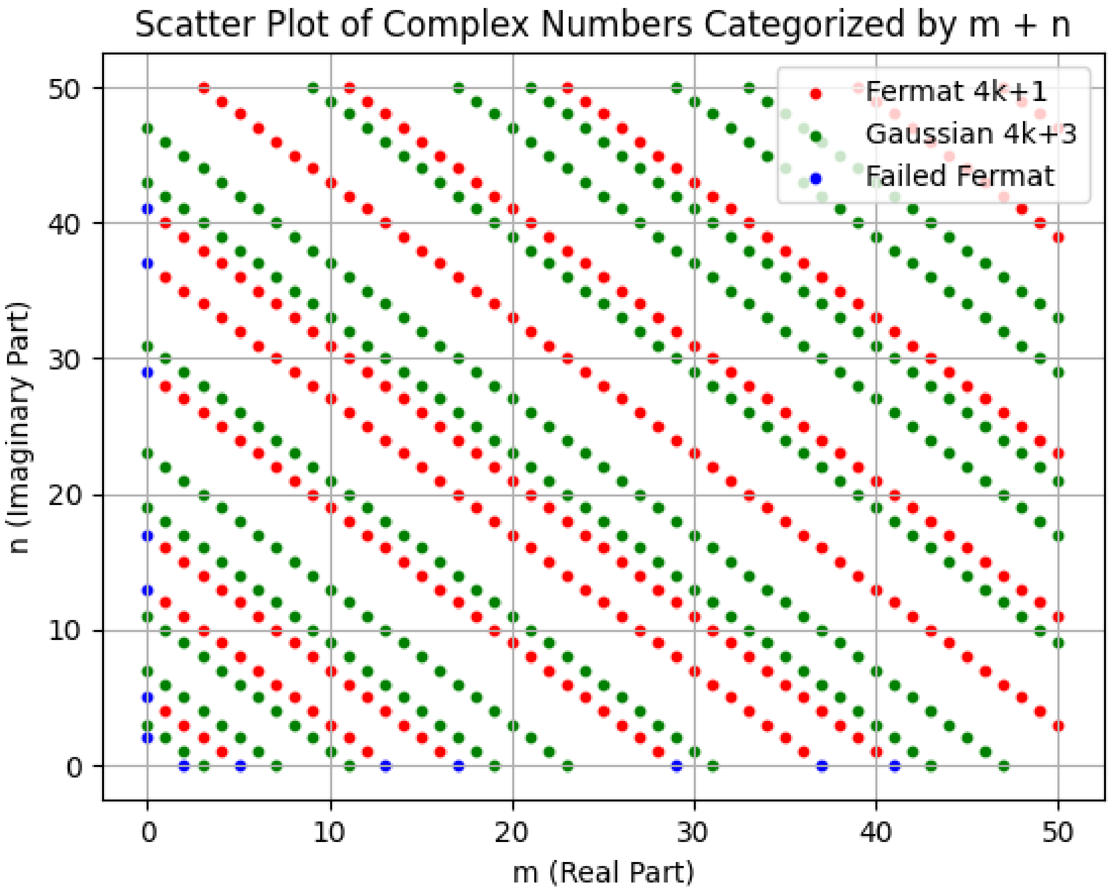


Figure 5.13: Primes of 4n+1 and 4n+3 for of the Gaussian Integers.

[1] The concept of a ring originated from the German term "Zahlring," introduced by David Hilbert meaning a "number ring," suggesting a collection of numbers.

The Python script identifies Gaussian primes, according to whether the sum $a+b$ of its real and imaginary parts is prime and, if so, whether it is of (Fermat),$4k+1$ or (Gaussian) $4k+3$ form. Further those integers whose real or imaginary parts are zero are Gaussian primes if the non-zero part is of $4k+3$ form. The 'Failed Fermat' composite alternatives are dotted in blue on the two axes $(5,0) = (1,2) \times (1,-2) = (1+2i) \times (1-2i)$.

```
color_map = {
    'Fermat 4k+1': 'red',
    'Gaussian 4k+3': 'green',
    'Failed Fermat': 'blue', }
for category, color in color_map.items():
    if category != 'Other':  # Exclude 'Other' from the legend
        subset = df[df['category'] == category]
        plt.scatter(subset['m'], subset['n'], c=color, label=category, s=2)
```

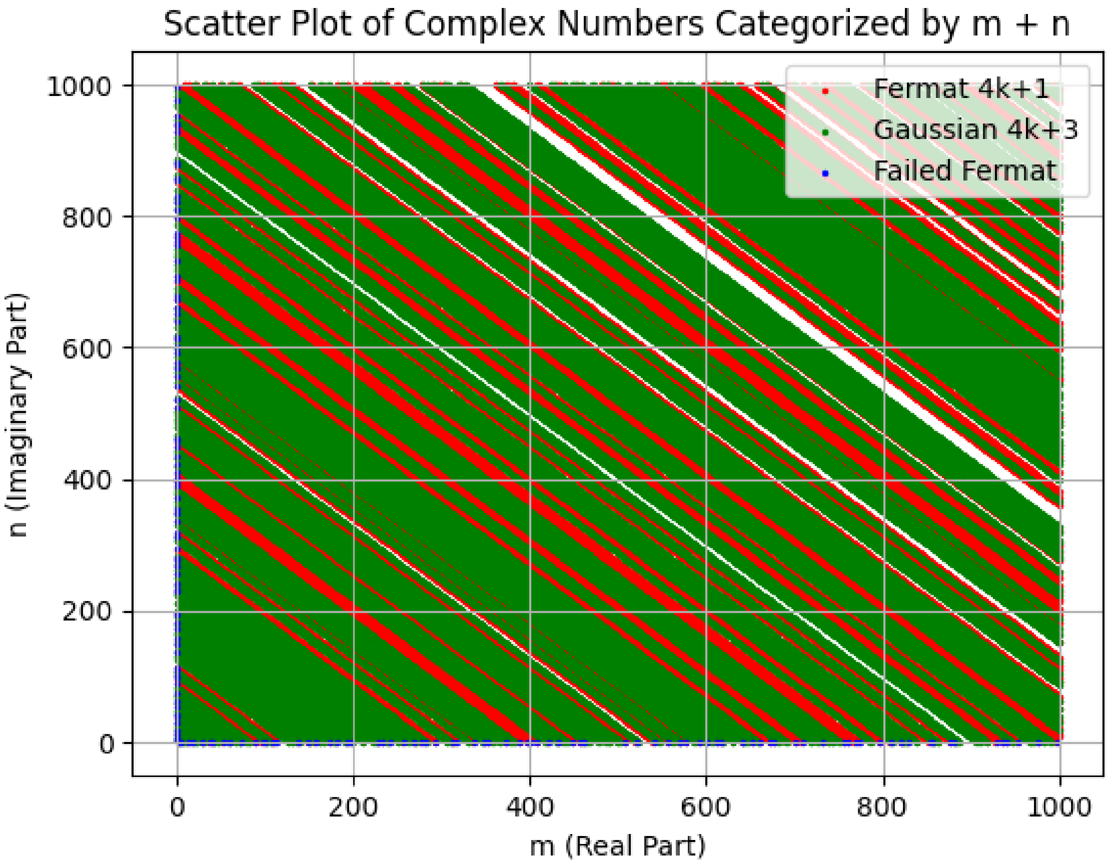


Figure 5.14: First 1000 Primes of 4n+1 and 4n+3 form of the Gaussian Integers.

1. **Color Map Definition:** A dictionary named `color_map` is defined, which assigns a specific color to each category of data points.
2. **Data Plotting:** The code iterates over each category in the `color_map` dictionary and except 'Other', it filters the DataFrame `df` to create a subset containing only the data points belonging to the current category. The size of the scatter points is set to 2 for finer detail.
3. **Legend Handling:** The conditional statement within the loop (`if category != 'Other'`) ensures that the 'Other' category is not included in the legend of the plot.

Integral Domain of Gaussian Integers

An integral domain is a commutative ring with no zero divisors. The term "integral" is used to suggest the wholeness or completeness of the domain. Gaussian integers, a subset of the *field*[2] of complex numbers of the form $a+bi$ where $a,b \in \mathbb{Z}$, form an integral domain. In algebraic number theory foundational structures such as commutative rings, e.g.$\mathbb{Z}[i]$ can be *extended* by operations like *modulo*, that provide a more limited operational closure of the number system. While $(\mathbb{Z}[i])$ form an integral domain (a commutative ring with no zero divisors), the set of numbers, $a+b\sqrt{-5}$ form only a commutative ring due to the presence of zero divisors. As such the Gaussian integers, are characterized by the commutativity of its multiplication and fact that if the product of two Gaussian integers is zero, then at least one of the multiplicands must be zero:

- $(1+2i)\cdot(3+i) = -5+5i = (3+i)\cdot(1+2i)$
- $(a+bi)(c+di) = 0$, then either $a+bi = 0$ or $c+di = 0$.

The Moat of Gaussian Integers:

Gaussian integers are complex numbers of the form $a+bi$, where a and b are integers and they form a lattice on the complex plane punctuated with unique prime elements, known as Gaussian primes. Now Gaussian integers of the form $p+ip$, where p is a prime number, are not typically Gaussian primes themselves because they can be factored into $p(1+i)$, except when p equals 2, since $1+i$ is a unit in the ring of Gaussian integers and also a Gaussian prime. So if we are to plot the domain of Gaussian primes we observe the fractal pattern of 95.15) known as a Gaussian moat.

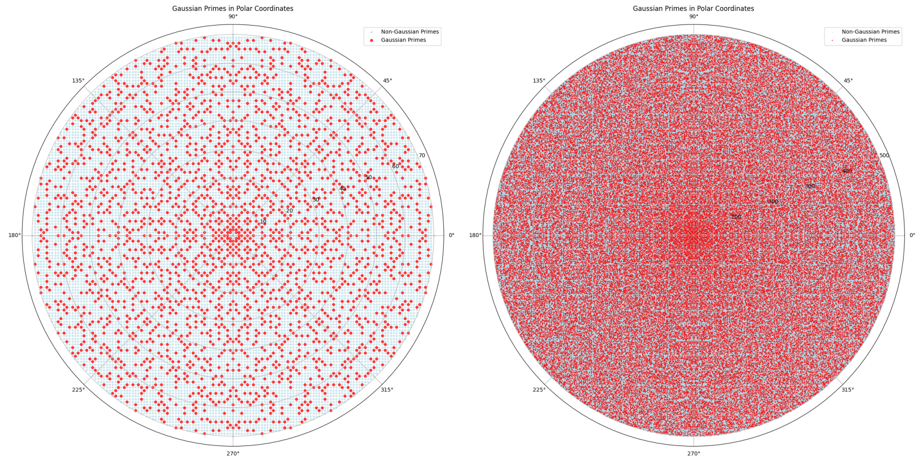


Figure 5.15: Gaussian Prime Moats up to n=70 and 500

The `is_gaussian_prime` function determines whether a Gaussian integer, expressed in the form $a+bi$, is a Gaussian prime. The function works as follows:

```
def is_gaussian_prime(a, b):
if a != 0 and b != 0:
    return isprime(a**2 + b**2)
else:
    return isprime(max(abs(a), abs(b)))
```

[2] A field is a ring, with a complete algebraic structures in which every non-zero element has a multiplicative inverse. The term "field" was first used by Moore in 1893. The field of complex numbers $\mathbb{C}$ has the property that every non-zero element in $\mathbb{C}$ has a multiplicative inverse.

- If both a and b are non-zero, the Gaussian integer is a prime if and only if a^2+b^2 is a prime number in the natural numbers. This is based on the norm of a Gaussian integer, which for any Gaussian integer $z=a+bi$ is given by $N(z)=a^2+b^2$.
- If either a or b is zero, the Gaussian integer is a prime if the non-zero part is a prime number in the natural numbers and the other part is zero. This handles the cases of $a+0i$ and $0+bi$.

The function can be described accordingly as:

$$\text{is\_gaussian\_prime}(a,b)=\begin{cases}\text{isprime}(a^2+b^2) & \text{if } a\neq 0 \text{ and } b\neq 0,\\ \text{isprime}(\max(|a|,|b|)) & \text{if } a=0 \text{ or } b=0.\end{cases}$$

This function takes two integers a and b as input and returns a boolean value indicating whether $a+bi$ is a Gaussian prime.

Definition 14 *Quadratic Fields* both real and imaginary, are fundamental in the study of algebraic number theory. These include numbers of the form $a+b\sqrt{d}$, for any square-free integer d and $a+b\sqrt{-p}$, which are notable *field extensions*, $\mathbb{Q}(\sqrt{d})$ and $\mathbb{Q}(\sqrt{-p})$ of the rational numbers $\mathbb{Q}$. When d is positive, $\mathbb{Q}(\sqrt{d})$ forms a real quadratic field.

Field Extension of Integers

In the case when d is a both negative and a prime $-p$, in which case the resulting field $\mathbb{Q}(\sqrt{-p})$ is an imaginary *quadratic field* that extends the rationals to include the square roots of negative primes, forming a subset of the complex numbers $\mathbb{C}$. Extending the set of integers $\mathbb{Z}$ with a surd of the form $\sqrt{-p}$ where p is a prime number, creates the algebraic structure of a ring, $\mathbb{Z}[\sqrt{-p}]$ which consists of all numbers of the form $a+b\sqrt{-p}$, where a,b are integers with the following properties:

- is closed under addition and multiplication.
- contains the elements of $\mathbb{Z}$ and $\sqrt{-p}$.
- forms an *integral domain*, short of a field, as not all elements have multiplicative inverses within the ring.

The set of numbers of the form $a+b\sqrt{-5}$ where a and b are integers forms a (multiplicative) commutative ring but not an integral domain due to the presence of zero divisors, violating the *integral* aspect of an integral domain:

- $(1+\sqrt{-5})(2-\sqrt{-5})=-3+\sqrt{-5}=(2-\sqrt{-5})(1+\sqrt{-5})$
- $(1+2\sqrt{-5})(1-2\sqrt{-5})=21$ and $(2+\sqrt{-5})(2-\sqrt{-5})=-1$, and yet $21\cdot(-1)=0$,

While the Gaussian integers form a **subring** of the complex numbers and share some properties with the integers, surd-based quadratic field systems do not.

Metallic numbers, such as the golden and silver ratio, like $\frac{b+\sqrt{b^2+4ac}}{2a}$ when extended to the complex plane, can have complex forms with irrational components. The metallic number-based complex number $z=\frac{1+i\sqrt{b^2+4}}{2}$ and its conjugate $\bar{z}=\frac{1-i\sqrt{b^2+4}}{2}$ has a norm of z, given by $|z|^2=z\bar{z}$, which simplifies to $\frac{b^2+5}{4}$. In cases where this norm is a prime number, it is of the Fermat form $(4k+1)$ as evidenced from:

$$\frac{b^2+5}{4}=k+\frac{1}{4}\quad\text{for some integer } k.$$

In $\mathbb{Z}[i]$, a Fermat prime cannot be a Gaussian prime since it can be expressed as the sum of two squares. However, in the extended field that includes irrational numbers, the complex forms of metallic numbers exhibit a connection between their norms and Fermat primes.

5.4 Prime Spiral Polar Rays

Twin primes (marked in red below) are pairs of primes that are two units apart, i.e., $[p, p+2]$ where both p and $p+2$ are prime. Below are polar coordinate scatterplots where now each point, (n,n) instead represents an integer n up to 10^6, and the radial coordinate is proportional to $\log_{10}(n)$.

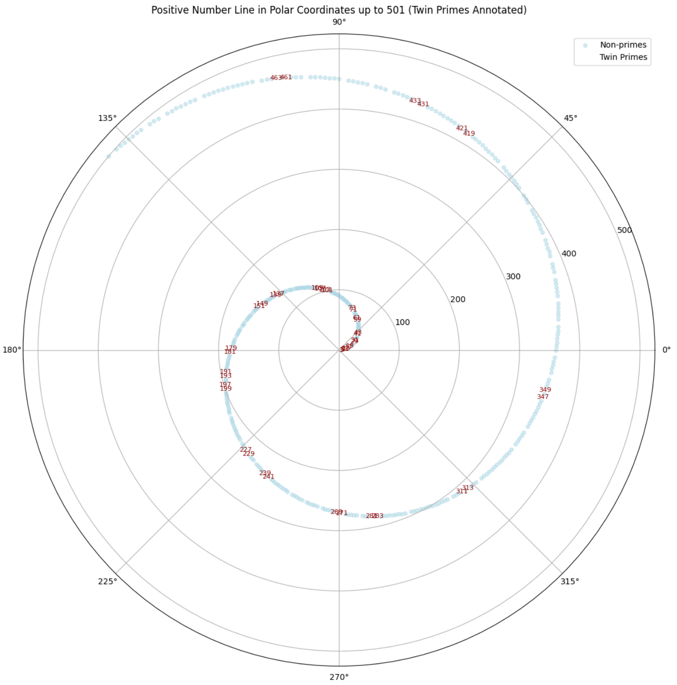


Figure 5.16: Spiral of twin (p,p) up to 500.

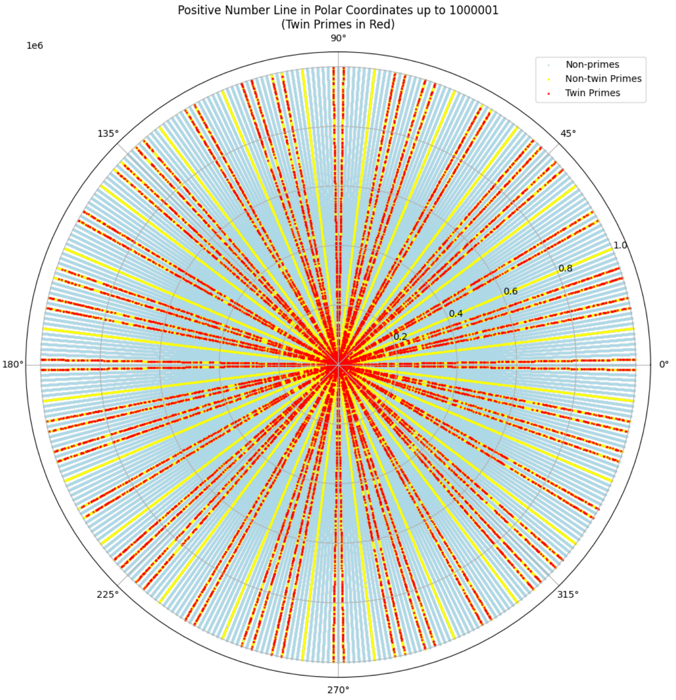


Figure 5.17: Spiral of twin (p,p) up to 1000000.

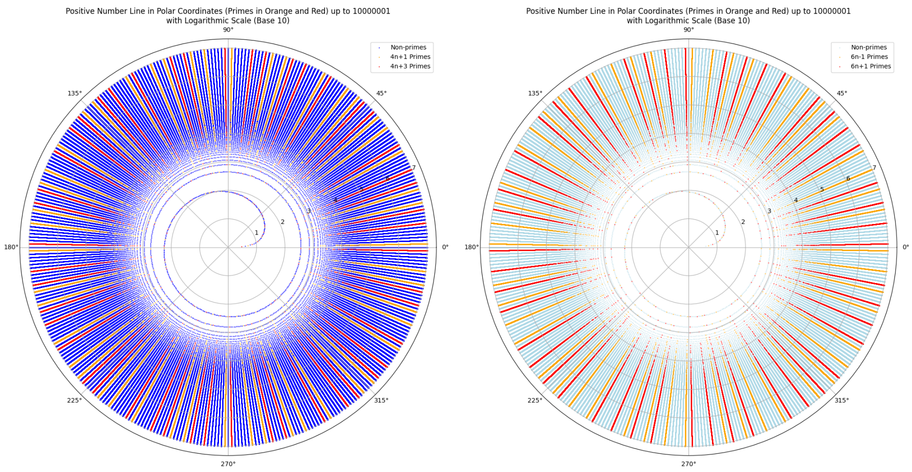


Figure 5.18: Polar Spirals delineating prime types

All integers can be expressed in one of the following *residue classes* of mod 6) forms: $6n$, $6n+1$, $6n+2$, $6n+3$, $6n+4$, or $6n+5$. Prime numbers, except for 2 and 3, conform to the form $6n\pm 1$ to avoid divisibility by 2 or 3 and on a polar plot the primes are more likely to appear at angles corresponding to these forms. Similarly, *residue classes mod 4*, $4n+1$ and $4n+3$, cluster also in twelve batches of four in each quadrant but along a different set of angles. Twin primes appear in 9 pairs across each quadrant.

3Blue1Brown explores the prime spoke behaviour that conjures from the inherent spiral plotting of (n,n) on a non-complex polar plane as a visual consequence of the relationship between Euler's totient function, ϕ and continued fraction approximations of π. As we are mapping primes onto polar coordinates, their angular position is proportional to their magnitude. Since $3/1$ is a crude approximation of π, we find that $6n$ positions map to approximately 2π, meaning that after six "turns", the points have traversed just under a full circle. Improved approximations can be obtained from the continued fraction representation of π:

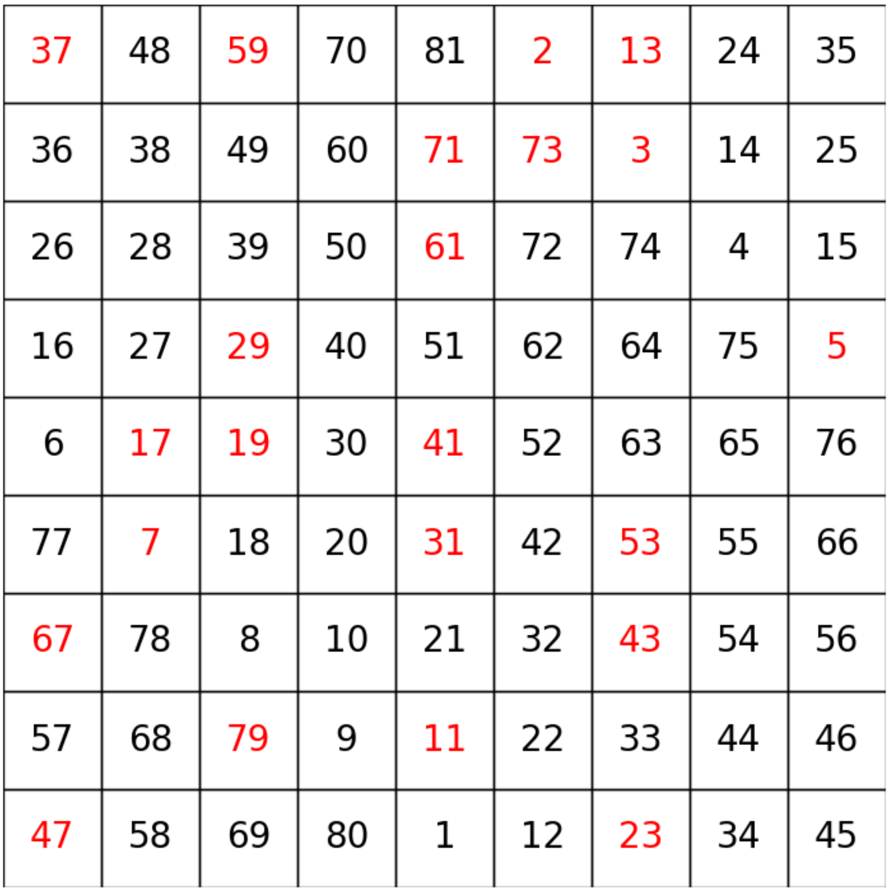
$$\pi = [3;7,15,1,292,\ldots] \approx 3+\cfrac{1}{7+\cfrac{1}{15+\cfrac{1}{1+\cfrac{1}{292+\cdots}}}}$$

$$\approx 3+\frac{1}{7}=\frac{22}{7} \;\; ; \;\; 3+\frac{1}{7+\frac{1}{15}}=\frac{333}{106} \;\; ; \;\; 3+\frac{1}{7+\frac{1}{15+\frac{1}{1}}}=\frac{355}{113}$$

A better approximation, $\pi \approx 22/7$, implies that $44n$ positions correspond to nearly 2π, creating seven almost complete turns and leads to the primes $44n\pm 1$ delineating the circular structure. $\frac{355}{113}$ as the best rational approximation of π with a denominator of three digits results in 113 turns delivering an almost completely divergent ray structure with barely a hint of spiralling. 710 as twice the numerator of the fraction $\frac{355}{113}$ has prime factorization $710 = 2\times 5\times 71$, and is divisible by 5 which we observe in the polar plot as the primes come in batches of 4. Euler's *totient function*, $\phi(n)$ counts the integers up to n that are coprime (relatively prime) to n. We have $\phi(710) = 710\cdot\frac{1}{2}\cdot\frac{4}{5}\cdot\frac{70}{71} = 280$ filtering out multiples of 5 giving the "missing teeth" in the radial distribution. The terms $\frac{1}{2}$, $\frac{4}{5}$, and $\frac{70}{71}$ act as filters to exclude non-coprimes. The reader is invited to explain the 36 twin prime pairs and to adapt the code overleaf to consider semi-primes or semi-perfect number patterns.

5.5 Magic Squares with an Ulam twist

Magic squares, often relegated to the realm of recreational mathematics hold a quaint charm for enthusiasts will not capture our attention for too long save for looking at the implementation of a nice generating routine that produces magic squares with primes identified with a nod to Ulam of the following form:



37	48	59	70	81	2	13	24	35
36	38	49	60	71	73	3	14	25
26	28	39	50	61	72	74	4	15
16	27	29	40	51	62	64	75	5
6	17	19	30	41	52	63	65	76
77	7	18	20	31	42	53	55	66
67	78	8	10	21	32	43	54	56
57	68	79	9	11	22	33	44	46
47	58	69	80	1	12	23	34	45

Figure 5.19: 9 × 9 Magic square generated by Siamese method with central element 41.

A key characteristic of an odd-ordered magic square is its central number, which can be calculated using the formula:

$$\text{Central Number} = \frac{1}{2}(n^2+1)$$

where n is the side length of the square so for a 9x9 magic square the central number is $\frac{1}{2}(9^2+1) = 41$, which suggestively corresponds to Euler's famous quadratic formula. Accordingly we might seek this relationship and investigate pairs (a,b) for the quadratic formula n^2+an+b determining if b can be the central number of a magic square by finding an integer n such that $b = \frac{1}{2}(n^2+1)$ The *Siamese method* as nicely articulated in [9] for generating an odd-ordered magic square follows these steps:

1. Start with the number 1 in the bottom middle cell.
2. Subsequent numbers are placed in a diagonally down-right pattern.
3. If this movement leads outside the square's boundaries, it wraps around to the opposite side.
4. If a cell is already filled or the next diagonal cell is outside the bottom-right corner, the next number is placed directly above the current one. Continue until square is filled.

The algorithm ensures that each row, column, and main diagonal of the resulting $n \times n$ square sums up to the same magical number and is implemented in the following code:

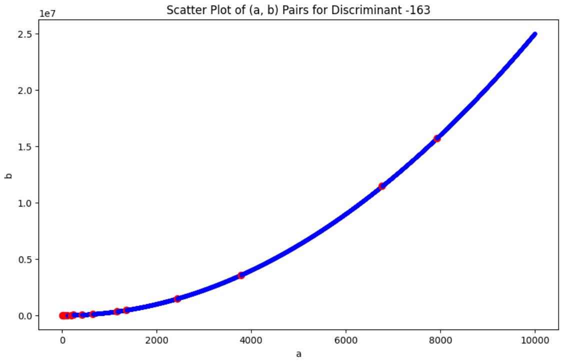
```
def generate_magic_square(n):
    magic_square = [[0]*n for _ in range(n)]# Create zero filled n x n matrix
    num = 1
    i, j = n-1, n//2  # Starting position
    while num <= n**2:
        # Place the current number
        magic_square[i][j] = num
        num += 1
        new_i, new_j = (i + 1) % n, (j + 1) % n
    # Check if the new position is filled or at the bottom right corner
        if magic_square[new_i][new_j] != 0 or (i == n-1 and j == n-1):
            i = (i - 1 + n) % n  # Move one square up
        else:
            i, j = new_i, new_j
    return magic_square
```

Discriminating between useful squares

The code, 163discriminant.ipynb generates a list of (a,b) pairs, and plots them to identify those that can form the central number of a magic square. The zip function is used to combine multiple iterable objects (lists or tuples):

```
for a, b, color in zip(a_values, b_values, colors):
    if color == 'red':
        plt.scatter(a, b, color=color, s=50)  # Larger red dots
    else:
        plt.scatter(a, b, color=color, s=10)  # Smaller blue dots
```

Specifically, `zip(a_values, b_values, colors)` creates an iterator that aggregates elements from three lists: `a_values`, `b_values`, and `colors`. Each element in the iterator is a tuple (a, b, color), where a is from `a_values`, b is from `b_values`, and color is from `colors`. The `for` loop then iterates over these tuples such that the point (a,b) is plotted on the scatter plot with a specified color and contingent size. The code identifies the Euler solution amongst many others that can also entertain a discriminant of 163,



a	b	Magic Square Size (n)
1	41	9
9	61	11
17	113	15
33	313	25
⋮	⋮	⋮
7921	15685601	5601

Figure 5.20: Set of (a,b) that deliver both central Magic number and discriminant of 163.

5.6 Perfect Numbers

The search for new perfect numbers is ongoing. It is unknown whether there are any odd perfect numbers.

Definition 15 *Perfect number* Perfect numbers are positive integers that are equal to the sum of their proper divisors (excluding themselves). The first few perfect numbers [7] are:

6 (the divisors are $1,2,3$)
- 28 (the divisors are $1,2,4,7,14$)
- 496 (the divisors are $1,2,4,8,16,31,62,124,248$)
- 8128 (the divisors are $1,2,4,8,16,32,64,127,254,508,1016,2032,4064$)
- $33,550,336$

All known perfect numbers have a close connection with *Mersenne primes*, such that if 2^p-1 is Mersenne, then $\left(2^{p-1}\right)(2^p-1)$ is perfect. This is implemented in the following code[3],

```
for _ in range(4):  # To find the next 4 perfect numbers
    p = next_p(p)
    while not is_prime(2**p - 1):
        p = next_p(p)
    perfect = (2**(p - 1)) * ((2**p) - 1)
    ln = 2**p - 1  # Mersenne prime, the largest divisor
```

A more brute force way to find a perfect numbers, is to use the recursion relation, [16]

$$l_n = 2l_{n-1} + 1$$

to find the l_n, that is the cumulative sum from 1 to a prime number. Each perfect number is then the sum of the numbers from 1 to l_n, inclusive:

- For $l_n = 2$, the Perfect Number is 6 and the Cumulative Sum is $1+2+3$.
- For $l_n = 3$, the Perfect Number is 28 and the Cumulative Sum is $1+2+3+4+5+6+7$.
- For $l_n = 31$, the Perfect Number is 496 and the Cumulative Sum is $1+2+3+\cdots+30+31$.

The code[4] implements the algorithm in which the recursion continues until l_n is a prime number:

```
def next_ln(ln):
    while True:
        ln = 2 * ln + 1
        if is_prime(ln):
            return ln
```

We see in the code snippet below the use of a loop up to the 6th perfect number[5] achieves the cumulative summation.

```
ln = 3  # as 2 corresponds to the first perfect number 6
for _ in range(6):  # To find the next 6 perfect numbers
    # Calculate perfect number using cumulative sum
    cumulative_sum_list = [i for i in range(1, ln + 1)]
    perfect = sum(cumulative_sum_list)
    cumulative_sum = ’+’.join(map(str, cumulative_sum_list))
    print(f"Perfect Number: {perfect}, Cumulative Sum: {cumulative_sum}")
    ln = next_ln(ln)
```

[3] MersennesPrimeToPerfect.ipynb

[4] perfectRecursion.ipynb

[5] Code crashes beyond that on google colab without paying extra!

5.6.1 Perfect-like Miscelania

Some properties of the Perfects that the reader is encouraged to explore are gathered here All known perfect numbers except 6 have reduced (*digital roots*) of 1:

$$\begin{aligned}
496 &\to 4+9+6=19 \to 1+9 \to 10 \to 1+0=1\\
28 &\to 2+8=10 \to 1+0=1\\
8128 &\to 8+1+2+8=19 \to 1+9 \to 10 \to 1+0=1
\end{aligned}$$

The perfect number 496 is the sum of the series from 1 to 31. As Gauss advises writing the series 1 to 16 in ascending order and then below it in descending order from 31 to 17:

1	2	3	4	5	6	7	89	10	11	12	13	14	15	16
31	30	29	28	27	26	25	2423	22	21	20	19	18	17	–
32	32	32	32	32	32	32	3232	32	32	32	32	32	32	16

delivers the sum of all the terms as $15 \times 32 + 16$, which is also equal to $\frac{31 \times 32}{2}$, the perfect number 496. All known perfect numbers are even and end in 6 or 28 (preceded by an odd number). The sum of the reciprocals of all its divisors is 2. For example,

$$6 : 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{6} = 2$$

Palindromic numbers

Here are some examples of squares of numbers that are palindromic:

$$\begin{aligned}
11^2 &= 121\\
111^2 &= 12321\\
1111^2 &= 1234321\\
11111^2 &= 123454321\\
111111^2 &= 12345654321\\
1111111^2 &= 1234567654321\\
11111111^2 &= 123456787654321\\
111111111^2 &= 12345678987654321\\
1111111111^2 &= 1234567900987654321
\end{aligned}$$

$$\begin{aligned}
232 &= 4 \times 58\\
23432 &= 4 \times 5858\\
2345432 &= 4 \times 586358\\
234565432 &= 4 \times 586414358\\
23456765432 &= 4 \times 5864194358\\
2345678765432 &= 4 \times 586419749358\\
234567898765432 &= 4 \times 58641974691358
\end{aligned}$$

All these numbers are divisible by 4. Consider the palindromic numbers:

$$232, 23432, 2345432, 234565432, \ldots$$

Note: What are the series of digits that we need to insert at each stage?
We observe the following series:

$$85, 63, 41, 19, 96, 74, \ldots$$

Observation: Each subsequent number is 22 less than the previous number, cycling back after adding 99 to 19 to start afresh:

Number	Difference
85	
63	-22
41	-22
19	-22
96	$+77(-22+99)$
74	-22

This pattern is consistently observed in the generation of the palindromic numbers. Consider the multiplication of 123456789 by numbers 2 through 8:

$$\begin{aligned}
123456789 \times 2 &= 246913578\\
123456789 \times 3 &= 370370367\\
123456789 \times 4 &= 493827156\\
123456789 \times 5 &= 617283945\\
123456789 \times 6 &= 740740734\\
123456789 \times 7 &= 864197523\\
123456789 \times 8 &= 987654312
\end{aligned}$$

Observation:
In each result, except where the multiplier is a multiple of 3 (i.e., 3 and 6), all nine digits are repeated again in the product. The products 370370367 and 740740734 include digit repetition but not all nine digits from 1 to 9 are present.

5.6.2 Snaking Divisors

Consider the series in which the sum of the proper divisors of the abundant number, 30, begets the abundant 42 which begets numbers that peak at the divisor deficient 259 from which we descend.

Starting Number	Divisors
30	[1, 2, 3, 5, 6, 10, 15]
42	[1, 2, 3, 6, 7, 14, 21]
54	[1, 2, 3, 6, 9, 18, 27]
66	[1, 2, 3, 6, 11, 22, 33]
78	[1, 2, 3, 6, 13, 26, 39]
90	[1, 2, 3, 5, 6, 9, 10, 15, 18, 30, 45]
144	[1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72]
259	[1, 7, 37]
45	[1, 3, 5, 9, 15]
33	[1, 3, 11]
15	[1, 3, 5]
9	[1, 3]
4	[1, 2]
3	[1]

Fig (5.21) plots such a divisor series from 1 to 40 with blue starting and red ending points. Note the omission of prime numbers with no proper divisors other than 1 while Perfect numbers (6,28) whose sum of proper divisors equal the number itself are identified as unending in their greyness.

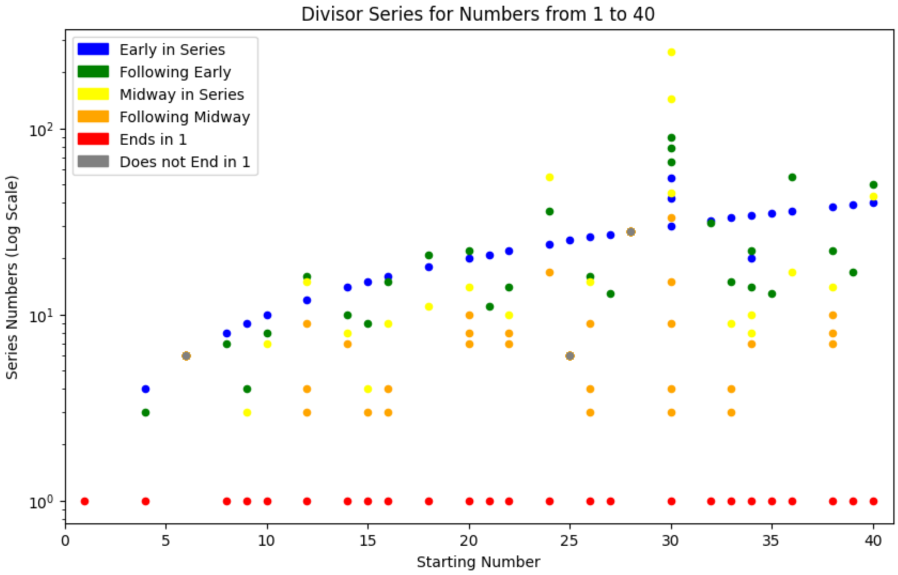


Figure 5.21: Proper Divisor Series starting Blue ending Red.

Early doors, just 138 possesses a complex lineage that only half a century ago, without modern computational tools, would have been a daunting to determine. Its series enters an extended phase of abundance, where each subsequent number is larger than the preceding one spanning 117 terms, reaching a peak at 179,931,895,322 after which a gradual descent of an additional 60 terms takes us to 1.

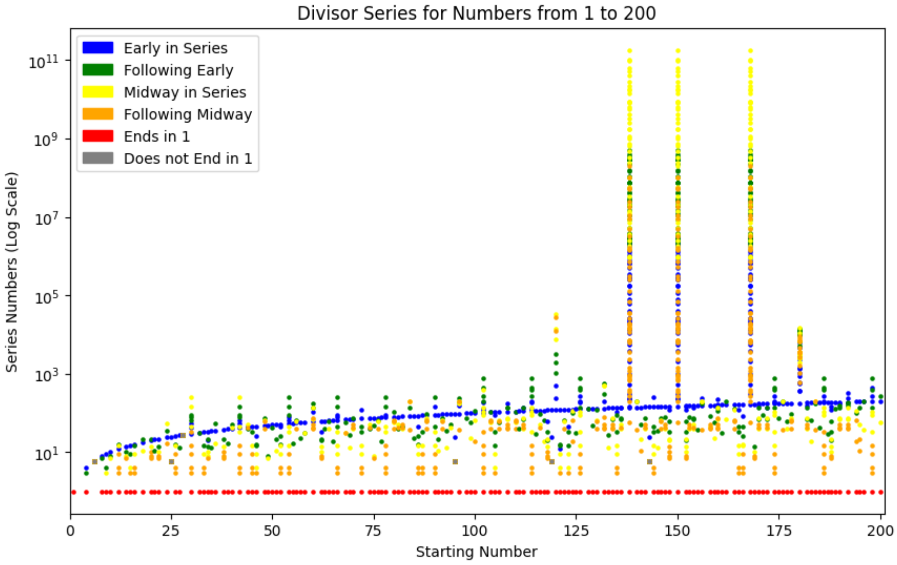


Figure 5.22: Proper Divisors in interval 1-200 for a maximum of 200 terms in any one series.

We can see in fig (5.23) a cyclical behavior with period of two of the amicable[6] pair (220, 284).

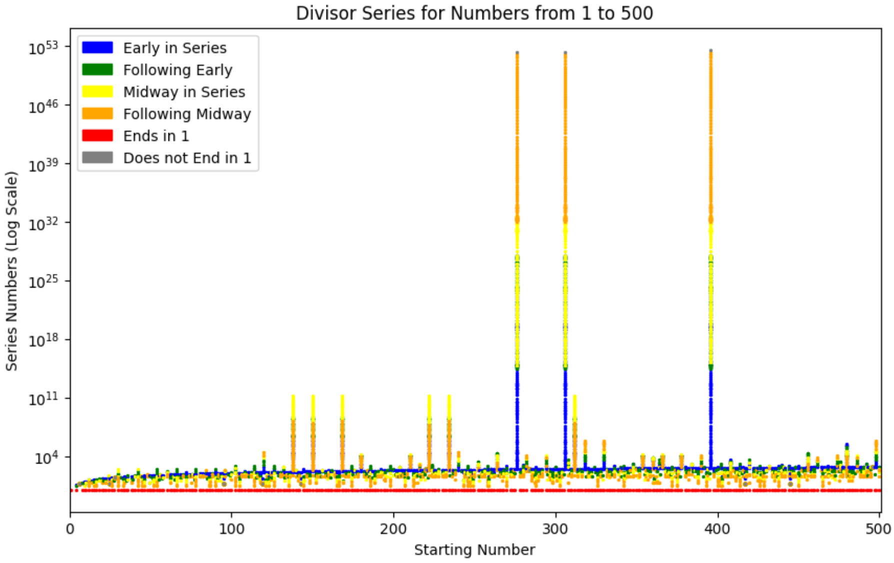


Figure 5.23: Proper Divisors in interval 1-500 with max 500 terms in sequence.

Our implementation allows for setting a maximum length for the series which curtails those series that do not converge to 1 within a tolerable number of steps and are marked with a grey dot at

[6] Sociable groups, as generalizations of amicable pairs to cycles of longer periods will also not converge to 1. Happily the first crowd of such sociable numbers does not occur before 1,264,460.

the last computed value. The relevant snippet from snakeDivisorGreyEndBelowTolerance.ipynb follows:

```
def visualize_all_series(min_number, max_number):
    fig, ax = plt.subplots(figsize=(10, 6))
    colors = [’blue’, ’green’, ’yellow’, ’orange’]
    num_colors = len(colors)
    end_color = ’red’  # Color for series ending in 1
    non_converging_color = ’gray’  # Color for series not converging to 1
```

The sympy library routine is used to work out proper divisors and `generate_series` is repeatedly summing the proper divisors of a number until the number becomes 1 or the series reaches a specified maximum length.

```
from sympy import divisors, isprime
def get_divisor_sum(n):
    return sum(divisors(n)) - n  # Subtract n to exclude the number itself
def generate_series(start, max_length=5000):
    series = [start]
    while start > 1 and len(series) < max_length:
        start = get_divisor_sum(start)
        series.append(start)
    return series
```

- `start`: starting number for the series.
- `max_length` (optional): maximum length of the series. Defaults to 5000 if not specified.
- In a loop, calculate the sum of proper divisors of the current number using `get_divisor_sum`
- Append result to the series and repeat until the current number becomes 1 or the series reaches the maximum allowed length.

```
for start_number in range(min_number, max_number + 1):
        if not isprime(start_number):  # Skip primes
            series = generate_series(start_number)
            step = max(1, len(series) // num_colors)
            for i, number in enumerate(series):
                if i == len(series) - 1:  # Last number in series
                    color = end_color if number == 1 else non_converging_color
                else:
                    color_index = min(i // step, num_colors - 1)
                    color = colors[color_index]
                ax.scatter([start_number], [number if number > 0 else 1], color=color, s=20
```

The loop iterates through a range of numbers (from `min_number` to `max_number`) and visualizes each number’s divisor series on a scatter plot.

- Skip the number if it is prime (`isprime` function checks for primality).
- Generate the divisor series for the number using the `generate_series` function.
- Determine the color of each point in the series based on its position:
  - The last number in the series is colored with `end_color` if it’s 1, indicating successful convergence to 1, or `non_converging_color` if it’s not 1, indicating the series did not converge to 1 within the maximum length.

  - Other numbers in the series are colored based on their position, using a set of predefined colors (`colors`) and dividing the series into segments (`num_colors`).
- The x-coordinate in the scatter plot is the starting number, and the y-coordinate is the number in the series. The color is determined as described above.

5.6.3 Diversity Density Ratio revisited

The Ulam spiral that reveals the pattern of semi-perfect abundance if not quite non semi-perfect abundance (coded with UlamNumberClassify.ipynb

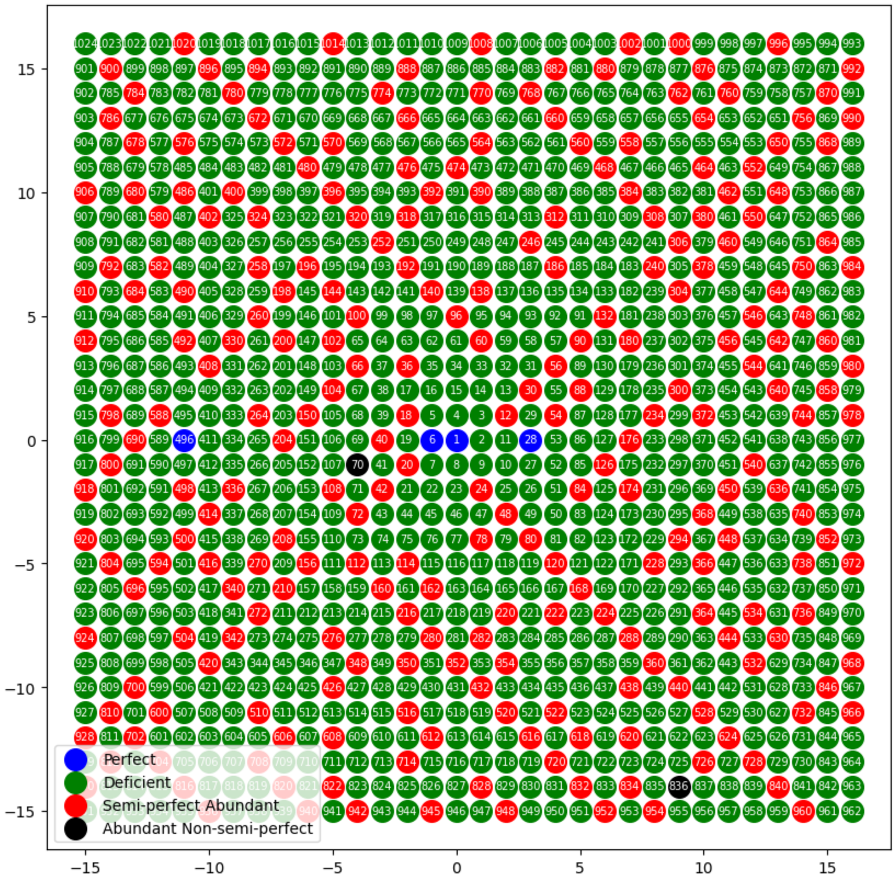


Figure 5.24: Ulam Spiral showing Number imperfections.

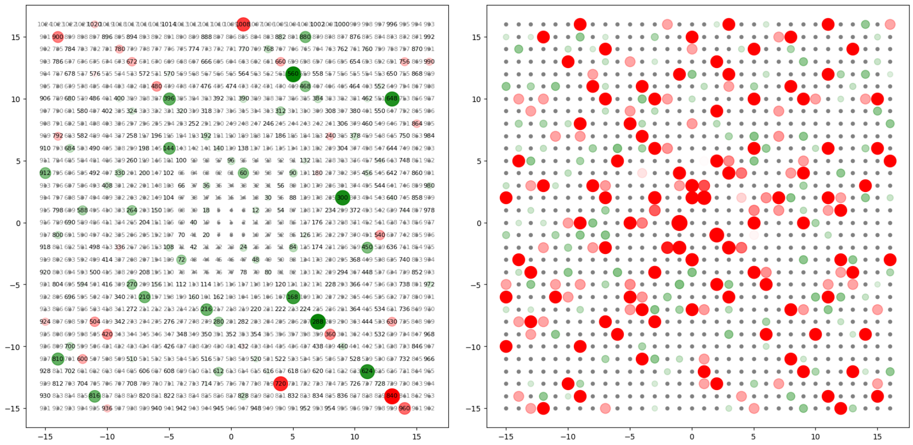


Figure 5.25: Ulam spiral of abundant composites,a with multiple, m proper factor representations (green to red, small to BIG) by their Divisor Density Ratio, m/a and Semi-Perfect numbers colored green to red, small to Big by P/A ratio.

5.7 Density of Primes in Residue Classes

The distinction between the densities of primes of the forms $4n+1$ (or equivalently $6n+5$) and $4n+3$ (or equivalently $6n+1$) is a longstanding observation in number theory.

5.7.1 Prime races

As previously discussed the primes can be partitioned into various grouping schemes. Numbers of the form $6n+1$ and $6n+5$ are always odd and do not usually have a plethora of small prime factors, a characteristic which makes it uncommon for them to have a large set of divisors whose sum would categorize them as abundant, especially in lower ranges.

1. **Numbers of the form $6n+1$:**
   - These are 1 more than a multiple of 6.
   - If they are prime, they clearly aren't abundant.
   - If they are composite, their smallest prime divisor can't be 2 or 3. This implies that their smallest possible prime factor is at least 5 (or even larger if they aren't divisible by 5). Such numbers can't have many small prime factors, making it less likely for their divisors to sum up to a value much larger than the number itself.
2. **Numbers of the form $6n+5$:**
   - These are 1 less than a multiple of 6.
   - Again, if they are prime, they aren't abundant.
   - If they are composite, their smallest prime divisor can't be 2 or 3. Moreover, numbers of the form $6n+5$ aren't divisible by 5. This ensures that their smallest possible prime factor is relatively large, making it less likely for them to be abundant.

For larger numbers, a more in-depth analysis would be needed to determine the abundance of these forms which we will address in the following by gathering first some related observations:

1. **Quadratic Residues and Non-residues**:
   - A prime p is of the form $4n+1$ if and only if -1 is a quadratic residue modulo p, i.e., there exists an integer x such that $x^2 \equiv -1 \pmod{p}$.
   - In contrast, for primes of the form $4n+3$, -1 is a quadratic non-residue modulo p.
2. **Factors of Integers**:
   - When you multiply two numbers of the form $4n+1$, the result is also of the form $4n+1$. For example, $5(4(1)+1)$ and $13(4(3)+1)$ are both primes, and their product, 65, is $4(16)+1$.
   - However, when you multiply a number of the form $4n+3$ by another of the same form, the result is of the form $4n+1$. This means that composite numbers of the form $4n+1$ can have prime factors that are both of the form $4n+1$ and $4n+3$.
   - On the other hand, composite numbers of the form $4n+3$ must have all their prime factors of the form $4n+3$. This restrictiveness implies that numbers of the form $4n+3$ have fewer ways to be factored and thus are more likely to be prime.
3. **Distribution in Arithmetic Progressions**:
   - It's a consequence of the generalized Riemann hypothesis that primes are equally distributed among different residue classes, meaning, in the long run, primes of the form $6n+1$ and $6n+5$ should have similar densities. But for smaller ranges (e.g., below 10^9), the distribution might not be balanced.
4. **Empirical Observations**:
   - In practice, this code seems to suggest that primes of the form $4n+3$ are more frequent in certain ranges. However, as numbers get large, the difference in densities decreases. Furthermore, it's unclear if one form surpasses the other infinitely often or if they eventually balance out.

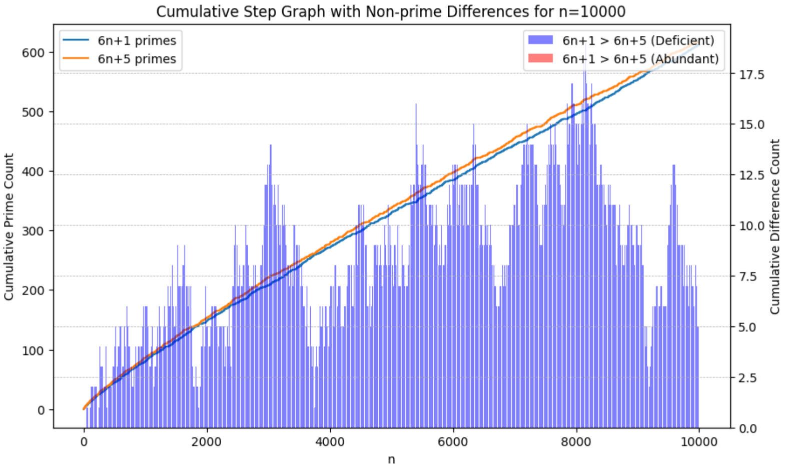


Figure 5.26: Histogram of Deficiency vs Abundant frequencies by Perimeter/Area ratio.

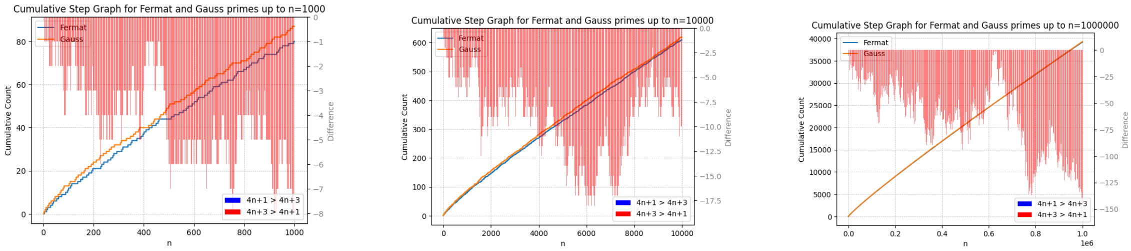


Figure 5.27: cumulative staircase of 4n+1 and 4n+3 prime generators and rolling differences for n=1000 n=1,000,000.

n	Prime Factors	$P(n)$	P/A	S_k	Classification
60	[2, 2, 3, 5]	14.00	0.04	108	semi-perfect abundant
61	[61]	31.00	0.51	1	deficient
62	[2, 31]	24.00	0.19	34	deficient
63	[3, 3, 7]	17.33	0.09	41	deficient
64	[2, 2, 2, 2, 2, 2]	18.14	0.09	63	deficient
65	[5, 13]	21.00	0.16	19	deficient
66	[2, 3, 11]	18.00	0.07	78	semi-perfect abundant
67	[67]	34.00	0.51	1	deficient
68	[2, 2, 17]	21.00	0.10	58	deficient
69	[3, 23]	24.00	0.17	27	deficient
70	[2, 5, 7]	18.00	0.06	74	weird (70)

Table 5.7: Classification based on divisors and their properties (continued)

5.8 Euler's Quadratic Prime Generator

The search for a formula that generates all prime numbers led Euler to his quadratic prime generator formula, n^2+n+a, where a is a constant and n ranges through positive integers, produces primes for many values of n and specific choices of a. The most famous incarnation is given by:

$$n^2+n+41$$

which produces primes for all values of n up to $n=40$. For n=41 we have 41^2+41+a which regardless of the value of a, the resulting expression is composite:

n	137_n	Prime?
36	1373	prime
37	1447	prime
38	1523	prime
39	1601	prime
40	1681	composite
41	1763	composite
42	1847	prime
43	1933	prime
44	2021	composite

The function `euler_generator(n)` taken from code takes a single argument n representing the upper limit of the sequence to be generated and iterates over a range of positive integers i from 1 to n using a `for` loop.

```
def euler_generator(n):
    for i in range(1, n+1):
        p = i**2 + i + 41
        if is_prime(p):
            yield (i, p, "prime")
        else:
            yield (i, p, "composite")
```

Within the loop, Euler's formula $p(n)=n^2+n+41$ is applied to calculate the value of p and the function then checks whether the calculated value p is a prime or a composite number using an external function `is_prime(p)`. If p is prime, a tuple $(i, p, \text{"prime"})$ is yielded, indicating that the generated value is a prime number otherwise, a tuple $(i, p, \text{"composite"})$ is yielded.

5.9 Helgott's Quadratic Prime Generator Formula

Consider the discriminant of $p(n) = n^2 + n + 41 = 0$ using the coefficients of the general quadratic equation $ax^2 + bx + c = 0$, where $a = 1$, $b = +1$, and $c = 41$ calculated using the formula $D = b^2 - 4ac$.

$$D = (-1)^2 - 4 \cdot 1 \cdot 41$$
$$D = 1 - 164 = -163$$

Number theorists have explored alternative quadratic formulas with varying degrees of success. One notable example is the quadratic formula involving the square root of 163, given by $\frac{1}{2}\left((9+\sqrt{163})n^2 + 3n + 1\right)$. Helgott's quadratic prime generator formula, $n^2 - 79n + 1601$, is another. Consider its discriminant using the coefficients of the general quadratic equation $a = 1$, $b = -79$, and $c = 1601$

$$D = (-79)^2 - 4 \cdot 1 \cdot 1601 = 6241 - 6404 = -163$$

Both Helgott's and Euler's quadratic formula have a common discriminant of -163. The relationship between the discriminants and the primality results hints at the intricate connections between quadratic expressions, discriminants, and prime generation. However, there are no other quadratic formulas with a discriminant of -163 that generate prime numbers for consecutive positive integer values of n. While there are other quadratic formulas with the discriminant -163, such as $n^2 - 163n + 6721$, they do not generate prime numbers for consecutive positive integer values of n. The following code snippet:

```python
x_fermat_euler, y_fermat_euler = [], []
x_non_fermat_euler, y_non_fermat_euler = [], []
for n in range(n_range):
    prime = euler(n)
    if is_prime(prime):
        if prime % 4 == 1:
            x_fermat_euler.append(n)
            y_fermat_euler.append(prime)
        else:
            x_non_fermat_euler.append(n)
            y_non_fermat_euler.append(prime)
```

taken from here initializes lists to store the values of n and the corresponding prime numbers for each category. It then runs a loop that iterates through a range of n values up to a predefined limit, denoted by `n_range`. For each n value:

1. The `euler(n)` function is called to generate a prime number using Euler's quadratic prime generator formula.
2. The code checks if the generated number is indeed a prime using the `is_prime()` function.
3. If the generated number is a prime:
   (a) It checks whether the prime number is congruent to 1 modulo 4 (satisfying Fermat's theorem). If it is, the n and prime values are appended to `x_fermat_euler` and `y_fermat_euler`, respectively.
   (b) If the prime number is not congruent to 1 modulo 4, it means it doesn't satisfy Fermat's theorem. In this case, the n and prime values are appended to `x_non_fermat_euler` and `y_non_fermat_euler`, respectively.

The code categorizes the prime numbers generated by Euler’s formula based on whether they satisfy Fermat’s theorem or not. The resulting lists `x_fermat_euler`, `y_fermat_euler`, `x_non_fermat_euler`, and `y_non_fermat_euler` are expected to contain the data points representing these categories, which are then to be used for further analysis or visualization.

5.9.1 Plotting the Quadratics

The following chart uses the plotting library, Matplotlib to plot with different markers and labels for different categories of primes

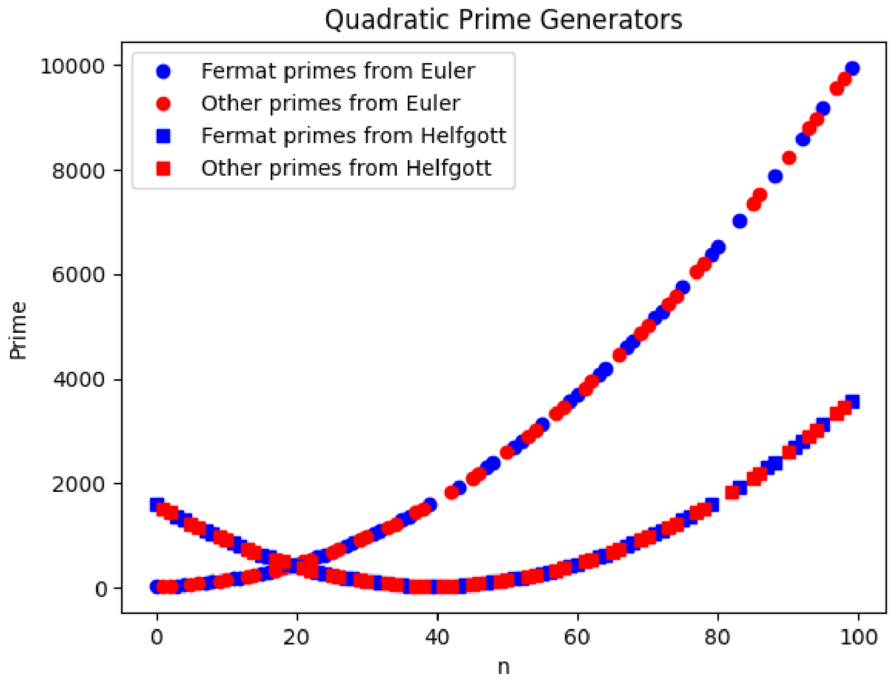


Figure 5.28: Quadratic Generators of Primes.

deploying the `ax.plot()` function to create multiple plots on the same graph.

```
ax.plot(x_fermat_euler, y_fermat_euler, ’bo’, label=’Fermat primes from Euler’)
ax.plot(x_non_fermat_euler, y_non_fermat_euler, ’ro’, label=’Other primes from Euler’)
ax.plot(x_fermat_helfgott, y_fermat_helfgott, ’bs’, label=’Fermat primes from Helfgott’)
ax.plot(x_non_fermat_helfgott, y_non_fermat_helfgott, ’rs’, label=’Other primes from Helfgo
ax.set_xlabel(’n’)
ax.set_ylabel(’Prime’)
ax.set_title(’Quadratic Prime Generators’)
ax.legend()
plt.show()
```

Each `plot()` call takes in four arguments: `x` values, `y` values, marker style, and a label for the legend and `x_fermat_euler`, `y_fermat_euler`, etc., represent lists of data points for the respective categories of primes. The code further customizes the plot by setting labels for the x-axis and y-axis, adding a title, and including a legend to differentiate between different categories, while the `plt.show()` function is called to display the plot.

5.9.2 Significance of $\sqrt{163}$

The appearance of $\sqrt{163}$ in certain quadratic expressions that generate prime numbers hints at a connection between quadratic formulas, irrational numbers, and the primes. Less prosaically, 163 is the largest Heegner number, which plays a role in the theory of modular forms and elliptic curves:

- -1: Field: $\mathbb{Q}(\sqrt{-1})$
- -2: Field: $\mathbb{Q}(\sqrt{-2})$
- -3: Field: $\mathbb{Q}(\sqrt{-3})$
- -7: Field: $\mathbb{Q}(\sqrt{-7})$
- -11: Field: $\mathbb{Q}(\sqrt{-11})$
- -19: Field: $\mathbb{Q}(\sqrt{-19})$
- -43: Field: $\mathbb{Q}(\sqrt{-43})$
- -67: Field: $\mathbb{Q}(\sqrt{-67})$
- -163: Field: $\mathbb{Q}(\sqrt{-163})$ (largest known Heegner number)

The Heegner numbers are important in understanding complex multiplication of elliptic curves and class numbers of *imaginary quadratic fields*. The Heegner number 1 is associated with the **field** of Gaussian integers, which are complex numbers of the form $a+bi$, where a and b are integers and $i = \sqrt{-1}$ is the imaginary unit. The Heegner number 1 represents the simplest case of complex multiplication and is related to the field of Gaussian integers. We will revisit these numbers later. Consider for now the following rather astounding feature of this set of numbers.

n	H_n	$e^{\pi\sqrt{H_n}}$
1	1	23.14069263277926680189
2	2	85.01969522320720784592
3	3	230.76458831914575853261
4	7	4071.93209522526103683049
5	11	33506.14306559242686489597
6	19	885479.77768015523906797171
7	43	884736743.99977517127990722656
8	67	147197952743.99981689453125000000
9	163	262537412640768256.00000000000000000000

Table 5.8: Tabulated list of n, H_n, and $e^{\pi\sqrt{H_n}}$

What we notice is that in particular H_7, H_8, H_9 are essentially integers (despite being the exponential of irrational numbers) to increasingly better approximations. H_9 being all but the integer twenty-six trillion, two hundred fifty-three billion, seven hundred forty-one million, two hundred sixty-four thousand, seventy-eight hundred twenty-five. The code snippet for the table is:

```
import math
heegner_numbers = [1, 2, 3, 7, 11, 19, 43, 67, 163]
print("{:<5} {:<15} {:<30}".format("n", "H_n", "e^(pi*sqrt(H_n))"))
print("-" * 45)
for n, H_n in enumerate(heegner_numbers, start=1):
    index_product = math.exp(math.pi * math.sqrt(H_n))
    print("{:<5} {:<15} {:<30.20f}".format(n, H_n, index_product))
```

which begins by defining a list called heegner_numbers. The print function is used to display the column headers with appropriate formatting string ":<5 :<15 :<30" aligning the columns for n, H_n, and $e^{\pi\sqrt{H_n}}$ to the left with specified widths. A for loop iterates through the heegner_numbers list using the enumerate function, which provides both the index n and the corresponding H_n value. Inside the loop, the code calculates $e^{\pi\sqrt{H_n}}$ using the math.exp and math.sqrt functions. The result is then printed with a precision of 20 $(.20f)$ decimal places.

Quadratic Prime Generator Formula with Discriminant $\sqrt{-67}$

As one might now suspect the quadratic formula associated with the imaginary quadratic field $\mathbb{Q}(\sqrt{-67})$ has a quadratic prime generator formula given by:

$$f(n) = n^2 + n + 17$$

The discriminant of a quadratic field $\mathbb{Q}(\sqrt{d})$ is given by $D = d$ if $d \equiv 1 \pmod 4$, and $D = 4d$ if $d \equiv 2,3 \pmod 4$. In this case, for $\mathbb{Q}(\sqrt{-67})$, the discriminant $D = -67$ since $-67 \equiv 3 \pmod 4$. Only when $n = 16$ do we get a square composite number $17^2 = 289$ as before with Euler's formula:

n	67_n	Prime?
12	173	Prime
13	199	Prime
14	227	Prime
15	257	Prime
16	289	Composite

Table 5.9: Number type from $\mathbb{Q}(\sqrt{-67})$ quadratic generator

Only when reaching $n = 112$ do we generate the first composite, 12673 comprised of product of three **distinct** prime factors, $12673 = 19 \cdot 23 \cdot 29$ which we will call a **Cuboid composite**:

n	67_n	67_n-Factors	163_n	163_n-Factors
112	12673	19, 23, 29	12697	
113	12899		12923	
114	13127		13151	
115	13357	19, 19, 37	13381	

Table 5.10: Prime Factorization Table for Quadratic Prime generators

Here we have tabulated the integers (composite and prime) arising from both $f(n) = n^2 + n + 17$ and $p(n) = n^2 + n + 41$ quadratic generators. We will not call 13357 a cuboid composite as it has a repeated (squared) prime factor. That is, $\text{RADICAL}(13357) = \text{RADICAL}(19^2 \cdot 37) = 19 \cdot 37$ whereas $\text{RADICAL}(12673) = 19 \cdot 23 \cdot 29$. The first such cuboid number generated by Euler's formula is $176861 = 47 \cdot 53 \cdot 71$ for $n = 420$ as can be seen with this code. Its first **rectangular** composite $1763 = 41 \cdot 43$ presented in the table overleaf is also noteworthy.The reader might consider how to investigate the frequency of composite generation between primes for both the series. Do they follow the same pattern?

5.10 End Tables

Factor information of a number, n in which each row includes its prime factorization, the number of prime factors (n_p), the number of distinct prime factors (r_p), the difference between n_p and r_p ($n_p - r_p$), the total number of factors (f), and the value of $f - 2 \cdot n_p$.

Table 1: Prime Factorization Table

n	P_{4n+1}	C_{4n+1}	P_{4n+3}	C_{4n+3}
1	5		7	
2		9	11	
3	13			15
4	17		19	
5		21	23	
6		25		27
7	29		31	
8		33		35
9	37			39
10	41		43	
11		45	47	
12		49		51
13	53			55
14		57	59	
15	61			63
16		65	67	
17		69	71	
18	73			75
19		77	79	
20		81	83	
21		85		87
22	89			91
23		93		95
24	97			99
25	101		103	
26		105	107	
27	109			111

Table 2: $C(n)_{4n+1}$ and $C(n)_{4n+3}$

$C(n)_{4n+1}$	$C(n)_{4n+3}$
2	3
5	6
6	8
8	9
11	12
12	13
14	15
16	18
17	21
19	22
20	23
21	24
23	27
26	28
29	29
30	30
31	33
32	35
33	36
35	38
36	39
38	42
40	43
41	45
42	46
44	48
46	50

n	67_n	67_n-Factors	163_n	163_n-Factors
1	19		43	
2	23		47	
3	29		53	
4	37		61	
5	47		71	
6	59		83	
13	199		223	
14	227		251	
15	257		281	
16	289	17, 17	313	
17	323	17, 19	347	
18	359		383	
19	397		421	
20	437	19, 23	461	
21	479		503	
22	523		547	
23	569		593	
24	617		641	
25	667	23, 29	691	
26	719		743	
27	773		797	
28	829		853	
29	887		911	
30	947		971	
31	1009		1033	
32	1073	29, 37	1097	
33	1139	17, 67	1163	
34	1207	17, 71	1231	
35	1277		1301	
36	1349	19, 71	1373	
37	1423		1447	
38	1499		1523	
39	1577	19, 83	1601	
40	1657		1681	41, 41
41	1739	37, 47	1763	41, 43

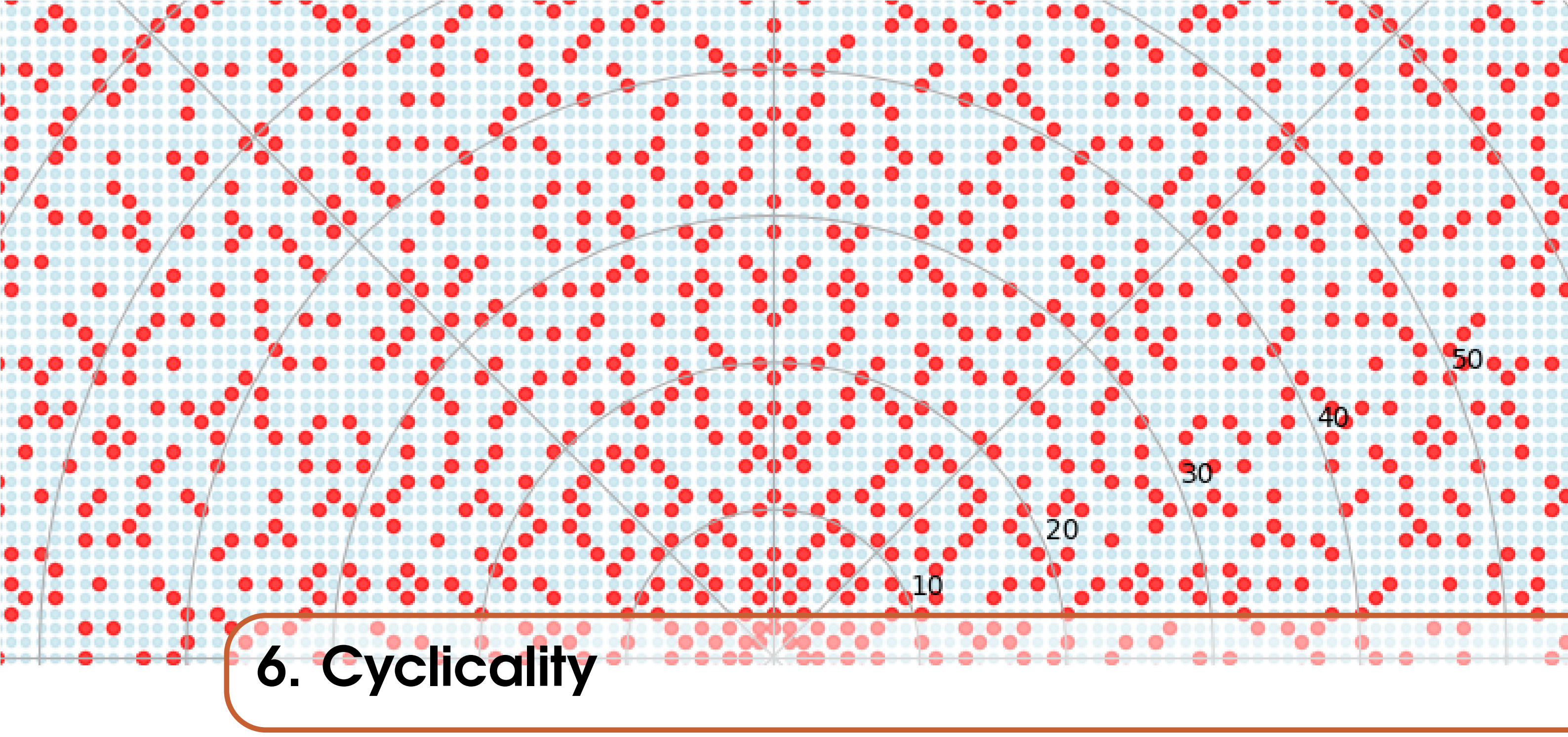
Table 5.11: A Rhythm in the Prime Factorization Table?

Table 5.12: Naturals, n; primes, p; composites, c; oblong, o; semi-prime, s-p; deficient, d; abundant ; triangle t and square-binary, b, written by the code lists of numbers.ipynb

n	p	c	o	s-p	d	a	t	b
1	-	-	-	-	1	-	1	1
2	2	-	2	-	2	-	-	-
3	3	-	-	-	3	-	3	-
4	-	4	-	-	4	-	-	4
5	5	-	-	-	5	-	-	-
6	-	6	6	6	-	-	6	-
7	7	-	-	-	7	-	-	-
8	-	8	-	8	8	-	-	-
9	-	9	-	-	9	-	-	9
10	-	10	-	10	10	-	10	-
11	11	-	-	-	11	-	-	-
12	-	12	12	-	-	12	-	-
13	13	-	-	-	13	-	-	-
14	-	14	-	14	14	-	-	-
15	-	15	-	15	15	-	15	-
16	-	16	-	-	16	-	-	16
17	17	-	-	-	17	-	-	-
18	-	18	-	-	-	18	-	-
19	19	-	-	-	19	-	-	-
20	-	20	20	-	-	20	-	-
21	-	21	-	21	21	-	21	-
22	-	22	-	22	22	-	-	-
23	23	-	-	-	23	-	-	-
24	-	24	-	-	-	24	-	-
25	-	25	-	-	25	-	-	25
26	-	26	-	26	26	-	-	-
27	-	27	-	27	27	-	-	-
28	-	28	-	-	-	-	28	-
29	29	-	-	-	29	-	-	-
30	-	30	30	-	-	30	-	-
31	31	-	-	-	31	-	-	-
32	-	32	-	-	32	-	-	-
33	-	33	-	33	33	-	-	-
34	-	34	-	34	34	-	-	-
35	-	35	-	35	35	-	-	-
36	-	36	-	-	-	36	36	36
37	37	-	-	-	37	-	-	-
38	-	38	-	38	38	-	-	-
39	-	39	-	39	39	-	-	-
40	-	40	-	-	-	40	-	-
41	41	-	-	-	41	-	-	-
42	-	42	42	-	-	42	-	-
43	43	-	-	-	43	-	-	-
44	-	44	-	-	44	-	-	-
45	-	45	-	-	45	-	45	-
46	-	46	-	46	46	-	-	-
47	47	-	-	-	47	-	-	-
48	-	48	-	-	-	48	-	-
49	-	49	-	-	49	-	-	49
50	-	50	-	-	50	-	-	-

Table 5.13: Naturals, n; primes, p; composites, c; oblong, o; semi-prime, s-p; deficient, d; abundant ; triangle t and square-binary, b.

n	p	c	o	s-p	d	a	t	b
51	-	51	-	51	51	-	-	-
52	-	52	-	-	52	-	-	-
53	53	-	-	-	53	-	-	-
54	-	54	-	-	-	54	-	-
55	-	55	-	55	55	-	55	-
56	-	56	56	-	-	56	-	-
57	-	57	-	57	57	-	-	-
58	-	58	-	58	58	-	-	-
59	59	-	-	-	59	-	-	-
60	-	60	-	-	-	60	-	-
61	61	-	-	-	61	-	-	-
62	-	62	-	62	62	-	-	-
63	-	63	-	-	63	-	-	-
64	-	64	-	-	64	-	-	64
65	-	65	-	65	65	-	-	-
66	-	66	-	-	-	66	66	-
67	67	-	-	-	67	-	-	-
68	-	68	-	-	68	-	-	-
69	-	69	-	69	69	-	-	-
70	-	70	-	-	-	70	-	-
71	71	-	-	-	71	-	-	-
72	-	72	72	-	-	72	-	-
73	73	-	-	-	73	-	-	-
74	-	74	-	74	74	-	-	-
75	-	75	-	-	75	-	-	-
76	-	76	-	-	76	-	-	-
77	-	77	-	77	77	-	-	-
78	-	78	-	-	-	78	78	-
79	79	-	-	-	79	-	-	-
80	-	80	-	-	-	80	-	-
81	-	81	-	-	81	-	-	81
82	-	82	-	82	82	-	-	-
83	83	-	-	-	83	-	-	-
84	-	84	-	-	-	84	-	-
85	-	85	-	85	85	-	-	-
86	-	86	-	86	86	-	-	-
87	-	87	-	87	87	-	-	-
88	-	88	-	-	-	88	-	-
89	89	-	-	-	89	-	-	-
90	-	90	90	-	-	90	-	-
91	-	91	-	91	91	-	91	-
92	-	92	-	-	92	-	-	-
93	-	93	-	93	93	-	-	-
94	-	94	-	94	94	-	-	-
95	-	95	-	95	95	-	-	-
96	-	96	-	-	-	96	-	-
97	97	-	-	-	97	-	-	-
98	-	98	-	-	98	-	-	-
99	-	99	-	-	99	-	-	-
100	-	100	-	-	-	100	-	100



6. Cyclicality

6.1 Arithmetic and Metric of $p-$adic Numbers

For a given prime, p, its p-adic numbers offer a a novel way to think about separation in which distances are not measured in terms of absolute differences, but rather the p-adic distance between two numbers is determined by how divisible their difference is by powers of p.

p-adic Metric

The p-adic metric (or distance) between two numbers x and y is defined as:

$$d_p(x,y) = |x-y|_p,$$

where

$$|x-y|_p = p^{-\max\{n: p^n \text{ divides } (x-y)\}}.$$

This metric has the counter-intuitive property that two numbers are considered "close" if their difference is highly divisible by p. For example, consider two 7-adic numbers, $x = 7$ and $y = 28$, in the 7-adic system. Their difference, $28 - 7 = 21$, is divisible by 7 but not by 7^2. Therefore, the 7-adic distance between x and y is $d_7(7,28) = |21|_7 = 7^{-1} = \frac{1}{7}$. This compares to the 7-adic distance $d_7(15,64) = |49|_7 = 7^{-2} = \frac{1}{49}$, indicating that 64 is "closer" to 15 than 28 is to 7 in the 7-adic metric system. For any p-adic numbers x and y, the p-adic metric $d_p(x,y)$ satisfies the following:

- **Non-negativity:** $d_p(x,y) \geq 0$ for all x, y, with equality if and only if $x = y$.
- **Symmetry:** $d_p(x,y) = d_p(y,x)$, reflecting the metric's indifference to the order of x and y.
- **Triangle Inequality:** $d_p(x,z) \leq d_p(x,y) + d_p(y,z)$, for any triple x, y, and z

For the 7-adic system, the triangle inequality, a cornerstone of metric spaces, reads:

$$|x-z|_7 \leq \max(|x-y|_7, |y-z|_7).$$

Consider the 7-adic distances between 35, 28, and 7:

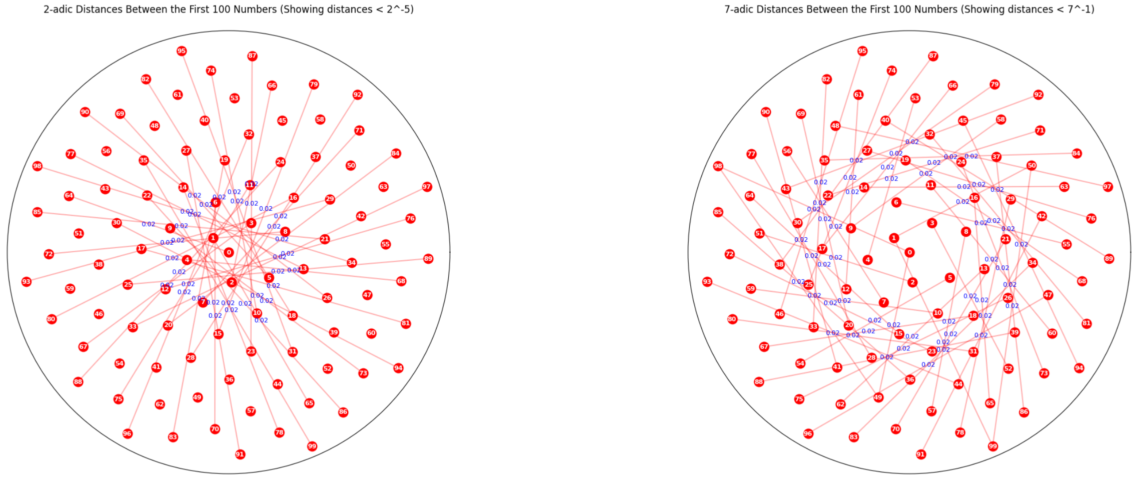
1. between 35 and 28 is $|35-28|_7 = |7|_7 = 7^{-1}$.

2. between 28 and 7 is $|28-7|_7 = |21|_7 = 7^{-1}$.
3. between 35 and 7 is $|35-7|_7 = |28|_7 = 7^{-1}$.

We confirm adherence to the triangle inequality by substituting the calculated 7-adic distances:

$$\begin{aligned} d_7(35,7) &\leq d_7(35,28) + d_7(28,7), \\ 7^{-1} &\leq 7^{-1} + 7^{-1}, \\ &\leq \max(7^{-1}, 7^{-1}) = 7^{-1}. \end{aligned}$$

We can picture equivalent distances on for binary/dyadic and 7-adic space by drawing on a polar spiral: The code snippet provided generates a visualization of p-adic distances between a set of numbers plotted on a polar coordinate system. The essential features of the code are:


2-adic Distances Between the First 100 Numbers (Showing distances < 2^-5)

7-adic Distances Between the First 100 Numbers (Showing distances < 7^-1)

Figure 6.1: 2-adic and 7-adic equal lengths.

- The number of points n is defined, representing the total numbers to visualize.
- The golden angle in radians is used to distribute the points aesthetically around the polar plot.
- The radial coordinates are determined using a square root scale for better visibility and distribution of points.
- A function `padic_distance` is defined to calculate the p-adic distance between any two numbers x and y, based on divisibility by powers of a prime p.
- For accurate placement of labels at the midpoint of the lines connecting points, polar coordinates are converted to Cartesian coordinates, the midpoint is calculated, and then converted back to polar coordinates.
- The plot is initialized using matplotlib's polar projection, with customized aesthetics such as the removal of axis labels and the inclusion of the grid.
- Lines are drawn to connect points representing p-adic distances, with labels indicating the distances placed at their midpoints. Labels are only displayed for distances less than $p^{-\text{min\_power}}$, based on the user-specified minimum power threshold.
- Each point is represented by a red dot, and its decimal number is labeled in white text.
- The plot is titled with information about the p-adic distances being shown, and the threshold for displaying labels.

For example, the code can be used to display the 3-adic distances between the first 81 numbers, showing only those distances less than 3^{-2}.

Arithmetic in p-adic Numbers

The p-adic numbers are constructed by considering all possible infinite series of the form:

$$a = \sum_{n=k}^{\infty} a_n p^n,$$

where k can be any integer (positive, negative, or zero), a_n are digits from 0 to $p-1$, and p is a prime number. This expansion allows for operations such as addition, subtraction, and multiplication to be defined in a manner consistent with the usual arithmetic operations but based on p-adic expansions.

6.2 Cyclicality of the Reptend Primes

Definition 16 *Reptend Prime* is a prime number p for which the decimal expansion of $\frac{1}{p}$ has a repeating cycle of maximum length $p-1$. This means the repeating decimal part of $\frac{1}{p}$ is $p-1$ digits long and under 100 comprise: $7, 17, 19, 23, 29, 47, 59, 61, 97$.

Definition 17 *Cycle Number* is a number in which cyclic permutations of its digits form successive multiples of the number.

The repeating decimal cycle of $\frac{1}{7}$ is of length $6 = 7-1$ since the order of 10 modulo 7 is 6, ($10^6 \equiv 1 \mod 7$), and no smaller power of 10 satisfies this. For $\frac{1}{7}$, the cyclic nature of its decimal expansion ('142857') is such that a permutation of this sequence is a multiple of the original number:

$$\begin{aligned}
1 \times \frac{1}{7} &= 0.142857142857\ldots \\
2 \times \frac{1}{7} &= 0.285714285714\ldots \\
3 \times \frac{1}{7} &= 0.428571428571\ldots \\
&\vdots \\
6 \times \frac{1}{7} &= 0.857142857142\ldots
\end{aligned}$$

$$\begin{array}{l}
140000000000000000000000 \\
002800000000000000000000 \\
000056000000000000000000 \\
000001120000000000000000 \\
000000022400000000000000 \\
000000000448000000000000 \\
000000000008960000000000 \\
000000000000179200000000 \\
000000000000003584000000 \\
000000000000000071680000 \\
000000000000000001433600 \\
000000000000000000028672
\end{array}$$

Each line on the left represents a cyclic permutation of '142857'. On the right we double 2×7 and continue doubling before adding together in a staggered fashion, to arrive at a sequence where each digit is a cyclic permutation of the previous one and whose final is 142857142857142857142272. Below is the code snippet from reptandCycleGenerator.ipynb to generate the staggered sum:

```
def generate_and_sum_numbers_for_cyclic_number_1_over_p(p):
    sequence = [2 * p * 2**i for i in range(12)]
    number_list = []
    cumulative_shift = 0
    for i, num in enumerate(sequence):
        d_current = len(str(num))  # Number of digits in the current number
        d_previous = len(str(sequence[i-1])) if i > 0 else 0
        if i > 0:
            cumulative_shift += 2 - (d_current - d_previous)
        prefill_zeros = '0' * cumulative_shift
        filled_number = prefill_zeros + str(num)
        number_list.append(filled_number)
    max_length = max(len(num) for num in number_list)
    padded_numbers = [num.ljust(max_length, '0') for num in number_list]
```

Generating the cyclic number for $\frac{1}{7}$ by doubling and applying a staggered summation is tied to the modular arithmetic properties of 7. The reader is invited to see if other primes can be similarly generated.

6.2.1 Remainders of Powers of Ten by 7

Consider the remainders of the powers of ten, 10^n when divided by 7 for $n = 1$ to 7:

$$\begin{aligned} 10^0 &\mod 7 = 1 \\ 10^1 &\mod 7 = 3 \\ 10^2 &\mod 7 = 2 \\ 10^3 &\mod 7 = 6 \end{aligned} \qquad \begin{aligned} 10^4 &\mod 7 = 4 \\ 10^5 &\mod 7 = 5 \\ 10^6 &\mod 7 = 1 \\ 10^7 &\mod 7 = 3 \\ &\vdots \end{aligned}$$

Noting that $10^7 \mod 7 = 10^1 \mod 7$, we have that $10^7 - 10^1$ is divisible by 7. Consequently, we can assert that $10(10^6 - 1)$ is divisible by 7. Similarly we can say that $10^6 - 10^0 = 10^6 - 1$ must be divisible by 7 and is in fact:

$$\frac{10^6 - 1}{7} = 142857$$

as:

$$\frac{142857}{10^6}\left[1 + \frac{1}{10^6} + \frac{1}{10^{12}} + \ldots\right] = \frac{142857}{10^6(1 - \frac{1}{10^6})} = \frac{142857}{10^6 - 1} = \frac{142857}{999999} = \frac{1}{7}$$

using the formula for an infinite geometric sum, $S = \frac{a}{1-r}$ for first term a and common ratio r.

6.2.2 Reptend Primes in Base-2

Similarly consider now the remainders of 2^n when divided by 7 for increasing values of n:

$$\begin{aligned} 2^1 &\mod 7 = 2 \\ 2^2 &\mod 7 = 4 \\ 2^3 &\mod 7 = 1 \end{aligned} \qquad \begin{aligned} 2^4 &\mod 7 = 2 \\ 2^5 &\mod 7 = 4 \\ 2^6 &\mod 7 = 1 \\ &\vdots \end{aligned}$$

Observing that after $2^3 \mod 7 = 1$, the remainders start repeating every 3 powers, as $2^3 - 2^0 = 7$ is obviously divisible by 7. For powers of 2, the cycle repeats every 3 exponents inferring that:

$$\frac{1}{7} = (0.\overline{001})_2$$

Thus for reptend primes, the binary representation of their reciprocals gives rise to repeating patterns. Each of these binary representations is a repeating cycle with a period that corresponds to the length of the reptend prime's cycle in binary. While the reptend prime concept relates to the cyclical behavior of the reciprocals of prime numbers in a decimal system, (where the length of the repeating cycle is maximized) a base-2 exploration is delivered by the code, polarReptendPrimesGolden.ipynb with function `is_primitive_root` checking whether the number 2 is a primitive root of a given prime number, generating two sets and returning `True` if these sets are equal:

1. `required_set` contains all integers from 1 to `prime` - 1.
2. `actual_set` generated by computing $2^{powers} \mod p$ for each `powers` from 1 to p- 1.

```
def is_primitive_root(prime):
    required_set = set(num for num in range(1, prime))
    actual_set = set(pow(2, powers, prime) for powers in range(1, prime))
    return required_set == actual_set
```

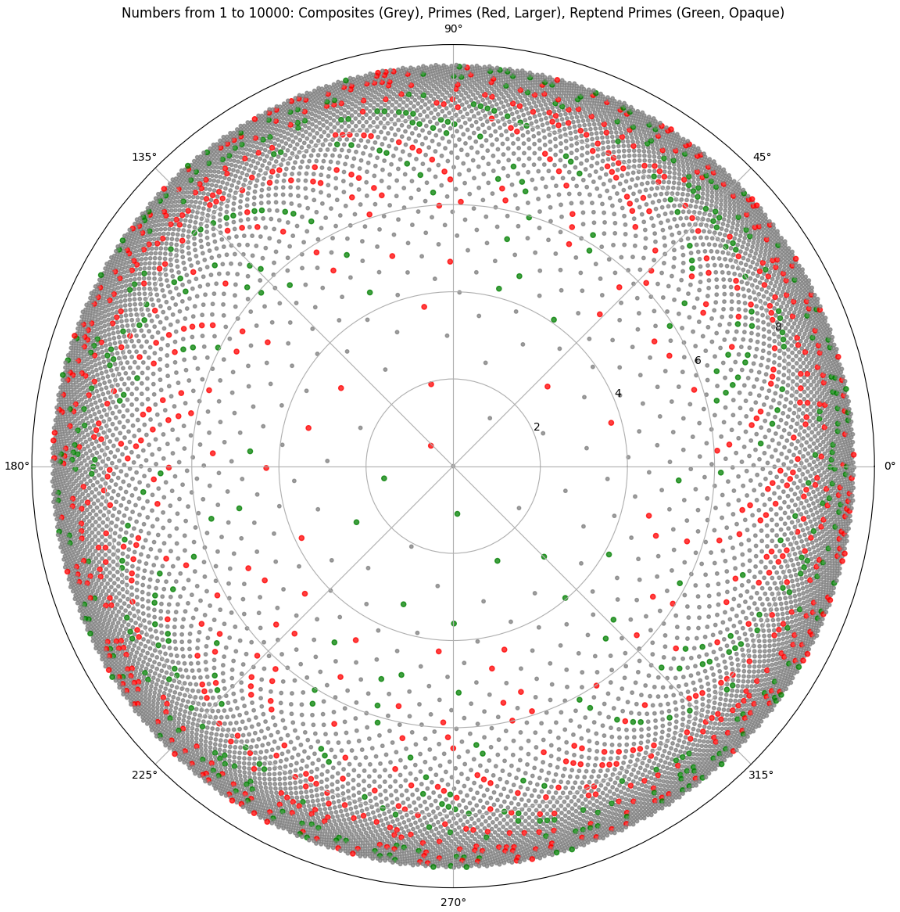


Figure 6.2: Reptend primes amongst their madding throng.

The `full_cycle_period` function checks if a given number is prime using `isprime` and then determines if 2 is its primitive root by calling `is_primitive_root`.

```
def full_cycle_period(prime):
    if not isprime(prime):
        return False
    return is_primitive_root(prime)
n = 10000
primes = list(primerange(1, n+1))
reptend_primes = [p for p in primes if full_cycle_period(p)]

theta = np.arange(n) * golden_angle_rad
radii = np.log(np.arange(1, n + 1))
```

Primes up to $n = 10000$ are generated using `primerange` and reptend primes are identified among these by checking which ones have a full cycle period. The golden angle is used to calculate the θ values, and logarithmic radii are computed to achieve a better distribution of points.

We will see that this binary representation offers a clearer visualization of the cyclical patterns inherent to reptend primes. We will do this in the following by formalising our previous discussion of 'magic' cyclical numbers in the language of deterministic chaos. In particular in the stably chaotic randomness of applying a *shift operator* to the reciprocal of reptend nmubers.

Discovering distribution of Cycle Lengths of Primes

For a given prime p, the length of the repeating cycle is $p-1$ digits if p is a full reptend prime. The sequence of digits that repeats in the decimal representation of $\frac{1}{p}$ is known as the reptend. The cyclicality of a reptend prime is equal to the order of 10 modulo the prime number so for 19, $\frac{1}{19}$ has a decimal representation whose cycle length is 18 and reptend number is 52631578947368421.

The Python logDirectReptandPrimeListingCycles.ipynb, determines cycle type classification from the cycle length as the ratio $\frac{p-1}{\text{cycle length}}$. If the cycle length is 0 (for primes like 2 and 5 that do not have repeating decimals), the cycle type is labeled as 'Other'. A DataFrame is generated to tabulate the primes, their cycle lengths, the inferred cycle types, and the repeating cycles and sample from the table is provided below.

Prime	Cycle Length	Cycle Type	Reptend number
2	0	-	0
3	1	2	3
5	0	-	0
7	6	1	142857
11	2	5	09
13	6	2	076923
17	16	1	0588235294117647
19	18	1	052631578947368421
⋮	⋮	⋮	⋮
9973	554	18	…

Table 6.1: Cycle Lengths and Types of Primes

The code calculates the cycle length of the reciprocal of a prime number by defining two functions: `find_cycle_period` and `get_repeating_cycle`.

- `find_cycle_period` determines the length of the repeating cycle in the decimal representation of $\frac{1}{p}$. It does so by iterating through multiples of 10 and checking when the remainder repeats, which indicates the start of a new cycle.
- `get_repeating_cycle` retrieves the actual digits that make up the repeating cycle of $\frac{1}{p}$.

```
def find_cycle_period(prime):
    remainder = 1
    seen_remainders = {}
    position = 0
    while True:
        remainder = (remainder % prime) * 10
        position += 1
        if remainder == 0:
            return 0  # No repeating cycle for this prime.
        if remainder in seen_remainders:
            return position - seen_remainders[remainder]
        seen_remainders[remainder] = position
```

The function find_cycle_period is designed to determine the period of the repeating decimal cycle (cyclicality) of the reciprocal of a given prime number:

- The function initializes a variable called remainder with the value 1, representing the initial numerator for the division by the prime number.
- An empty dictionary, seen_remainders, is created to keep track of remainders that have already been encountered during the division process.
- The variable position is set to 0 and serves as a counter for the number of decimal places processed.
- The function enters an infinite loop, within which the following steps occur:
  1. The remainder is updated by taking the current remainder modulo the prime number and multiplying the result by 10. This operation mimics long division, shifting the decimal point one place to the right.
  2. The position counter is incremented to indicate that we have moved one decimal place further in the division process.
  3. If the remainder becomes 0, the function returns 0, indicating that there is no repeating decimal cycle for this prime (the reciprocal is a terminating decimal).
  4. If the current remainder has been seen before (it exists in the seen_remainders dictionary), the function calculates the length of the cycle by subtracting the position where this remainder was first seen from the current position. This value is then returned as the cycle period.
  5. If the remainder is new (not in seen_remainders), it is added to the dictionary with its corresponding position.
- This process continues until a repeating remainder is found, or until a remainder of 0 is reached.

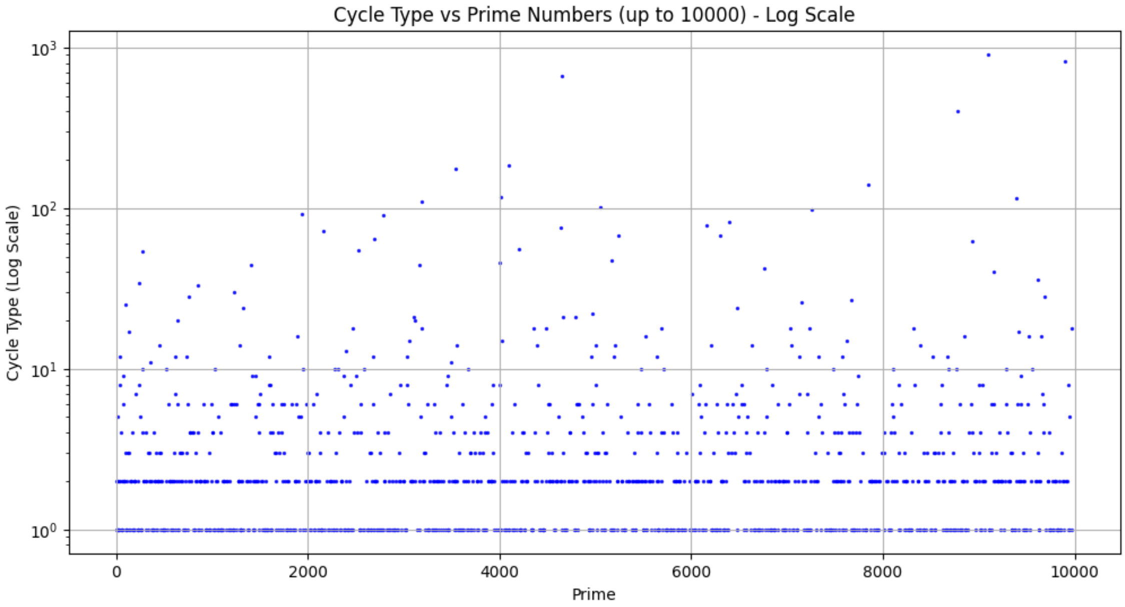


Figure 6.3: Reptend Number cycle length for primes up to 10000.

logAndlinCycleLengthvsPrimeByCycleType.ipynb delivers the scatter plot overleaf, where lines representing cycle 1 are steeper compared to cycles 2, 3, and 4, which relates to the distribution of repeating cycles within the decimal expansions of reciprocals of prime numbers.

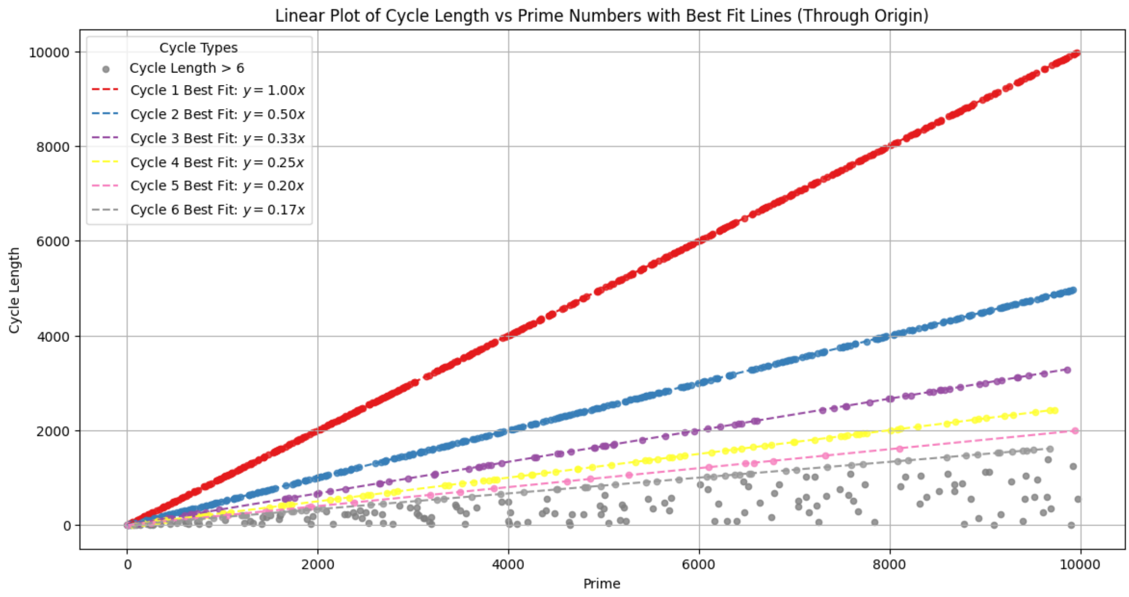


Figure 6.4: Reptend Number cycle length for primes up to 10000.

- **Direct Proportionality:** The cycle length is directly proportional to the prime p for cycle 1 primes, as full reptend primes have a repeating decimal cycle length of $p-1$. This results in a steeper line for cycle 1 as p increases.
- **Cycle Definitions:** The definitions of cycle types are based on the division of $p-1$ by integers 2, 3, 4, etc. Hence, the cycle length for a given prime p decreases as the cycle type number increases, leading to less steep lines for higher cycle types.
- **Prime Number Theorem:** The theorem suggests that primes become less frequent as numbers get larger, affecting the absolute increase in cycle length for each type. Thus, higher cycle types exhibit increasingly shallow slopes.
- **Modular Arithmetic:** The cycle type corresponds to the order of 10 modulo p. For cycle 1, the order is $p-1$, showing the full scope of modular arithmetic. For cycles 2 and beyond, the orders are factors of $p-1$, resulting in a shorter repeating decimal cycle.
- **Number Density:** The number of primes that show these cycle properties decreases for higher cycle types, which is reflected in the sparser distribution of points on the plot for cycles 2, 3, and 4.

```
if not subset.empty:
    X = subset[’Prime’].values.reshape(-1, 1)
    y = subset[’Cycle Length’].values
    reg = LinearRegression(fit_intercept=False)
    reg.fit(X, y)
    X_fit = np.linspace(X.min(), X.max(), 100).reshape(-1, 1)
    y_fit = reg.predict(X_fit)
    ax.plot(X_fit, y_fit, linestyle=’--’, color=colors(cycle_type - 1),
            label=f’Cycle {cycle_type} Best Fit: $y = {reg.coef_[0]:.2f}x$’)
```

The study of reptend primes in base 10, characterized by their maximal-length repeating cycles in the decimal representation of fractions $1/p$ for prime p, serves as a precursor to analyzing lesser known phenomena in any base and in particular binary.

6.3 Binary representations of Rationals

The number 5 can represented as $5_{10} = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, corresponding to the binary string 101 while 3 is represented as $3_{10} = 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 011_2$.

Definition 18 *Binary representation* of every integer k between 0 and $2^n - 1$ is the unique representation in binary form of k as

$$k = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0,$$

where $b_{n-1}, \ldots, b_0$ are bits that can either be 0 or 1.

This is modelled in the bifurcation tree below, in which each level of the tree corresponds to a binary digit (b_i) in the binary representation of a number.

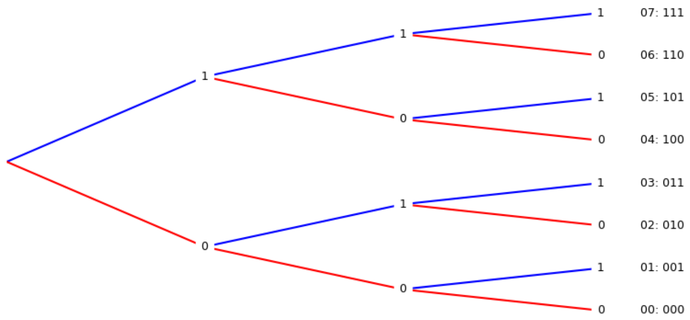


Figure 6.5: Bifuricating probability tree

Starting from the root of the tree, moving left or right represents choosing a 1 (right) or 0 (left) for the corresponding binary digit. As such, each path from the root to a leaf represents a unique binary number, with the leaf nodes representing the binary strings from 0 (bottom) to $2^n - 1$ (top). The code, bifuricatingTree.ipynb generates a bifurcation tree, which is used to visualize binary decision processes. The tree is drawn by calling `draw_path` with the starting coordinates and an empty path.

```
def draw_path(x_start, y_start, level, path=""):
    if level == levels:
        number = f'{int(path, 2):02d}: {path}'  # Padded number if single digit
        ax.text(x_start + 0.2, y_start, number, fontsize=9, ha='left', va='center')
        ax.text(x_start, y_start, path[-1], fontsize=9, ha='center', va='center',
            ↪ backgroundcolor='white')
        return
    ax.plot([x_start, x_start + 1], [y_start, y_start + 2**(levels-level-1)], 'b-')
    ax.plot([x_start, x_start + 1], [y_start, y_start - 2**(levels-level-1)], 'r-')
    if level > 0:  # Ensure we don't place a label at the starting point
        label = '1' if path[-1] == '1' else '0'
        ax.text(x_start, y_start, label, fontsize=9, ha='center', va='center',
            ↪ backgroundcolor='white')
    draw_path(x_start + 1, y_start + 2**(levels-level-1), level + 1, path + "1")
    draw_path(x_start + 1, y_start - 2**(levels-level-1), level + 1, path + "0")
draw_path(0, 0, 0, "")  # Adjusted starting point on the y-axis

ax.axis('off')
```

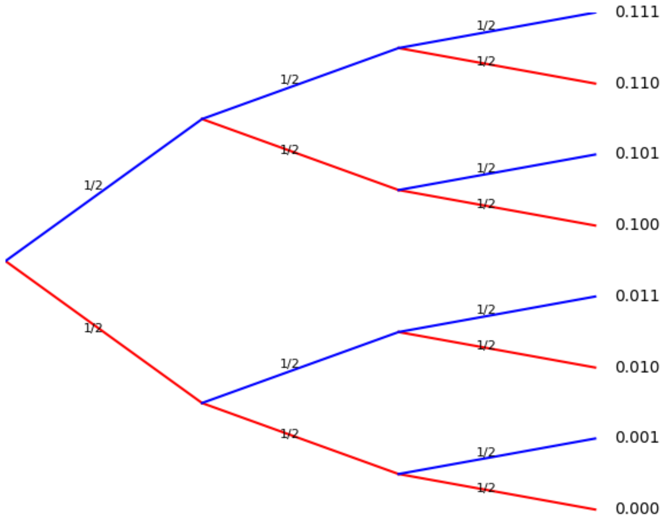
- the recursive function, `draw_path`, is defined within the main `plot_adjusted_bifurcation_tree` function to draw individual paths of the bifurcation tree. At each fork in the path:
  - A blue line represents a ’Heads’ decision, or binary ’1’.
  - A red line represents a ’Tails’ decision, or binary ’0’.

- Labels are added at each fork to indicate the binary decision made at that point. If the path chooses 'Heads', a '1' is displayed; if 'Tails', a '0'.
- At the final level, the binary representation of the path is displayed alongside its decimal equivalent, which is padded to ensure a consistent format.
- The axes are hidden to focus attention on the bifurcation tree itself.
- `plt.tight_layout` function is used to adjust the layout, preventing any potential clipping of the tree's branches.

We touch on this concept by formalizing our previous discussion of 'magical' cyclical numbers, by focusing on the stably chaotic randomness engendered by the application of the shift operator to the reciprocals of reptend numbers. As such we are more interested here in considering the prime reciprocals of k

$$\frac{1}{5} = 0.\overline{0011} = \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^7} + \frac{1}{2^8} + \cdots$$
$$\frac{1}{11} = 0.\overline{000101110100} = \frac{1}{2^4} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \cdots$$
$$\frac{1}{13} = 0.\overline{000111000010} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^{12}} + \cdots$$

The probabilityTree.ipynb below serves as a graphical calculator for the binary decimals of fractions, with each blue upward branch revealing an additional '1' bit in the binary sequence.



Each path from the root of the tree to a leaf can be seen as a sequence of decisions, with each decision corresponding to a digit in the binary fraction. Here's how to interpret the tree:

- The **root node** represents the whole number part of the fraction. For fractions between 0 and 1, this is always 0.
- Each level of the tree after the root represents a binary digit after the decimal point with **first level** corresponding to $\frac{1}{2}$, or 0.5, **second level** to $\frac{1}{4}$, or 0.25, and so forth.
- At each level, taking a **left branch**, colored in red, signifies adding a '0' to the binary fraction, whereas taking a **right branch**, colored in blue, signifies adding a '1'.
- The **probability** associated with each branch is $\frac{1}{2}$ akin to the outcome of a coin toss.
- To represent a fraction like $\frac{1}{3}$ with a binary expansion of $0.0101\ldots$, one would follow a path of alternating left and right branches, starting with a left branch for '0', then a right branch for '0.01', followed by another left for '0.010', and so on, ad infinitum.

6.4 Sum of $\left(\frac{1}{2}\right)^{2n}$

Following Hamkins, [6], consider the sum from n to infinity of $\left(\frac{1}{2}\right)^{2n}$,

$$\sum_{n=0}^{\infty}\left(\frac{1}{2}\right)^{2n}=\left(\frac{1}{2}\right)^{0}+\left(\frac{1}{2}\right)^{2}+\left(\frac{1}{2}\right)^{4}+\left(\frac{1}{2}\right)^{6}+\ldots=1+\frac{1}{4}+\frac{1}{16}+\frac{1}{64}+\ldots=\frac{4}{3}$$

As a geometric series with first term $a = 1$ and common ratio $r = \left(\frac{1}{2}\right)^2 = \frac{1}{4}$ and sum S of an infinite geometric series given by: $S = \frac{a}{1-r}$ we see in this case that:

$S = \frac{1}{1-\frac{1}{4}} = \frac{1}{\frac{3}{4}} = \frac{4}{3}$. The sum of the series as n tends to infinity is $\frac{4}{3}$ is suggested by the convergence of the partial sums.

$$n=0: \quad \left(\frac{1}{2}\right)^{0}=\frac{1}{1}$$

$$n=1: \quad \left(\frac{1}{2}\right)^{0}+\left(\frac{1}{2}\right)^{2}=\frac{1}{1}+\frac{1}{4}=\frac{4+1}{4}=\frac{5}{4}$$

$$n=2: \quad \left(\frac{1}{2}\right)^{0}+\left(\frac{1}{2}\right)^{2}+\left(\frac{1}{2}\right)^{4}=\frac{1}{1}+\frac{1}{4}+\frac{1}{16}=\frac{16+4+1}{16}=\frac{21}{16}$$

$$n=3: \quad \left(\frac{1}{2}\right)^{0}+\left(\frac{1}{2}\right)^{2}+\left(\frac{1}{2}\right)^{4}+\left(\frac{1}{2}\right)^{6}=\frac{1}{1}+\frac{1}{4}+\frac{1}{16}+\frac{1}{64}=\frac{64+16+4+1}{64}=\frac{85}{64}$$

$$n=4: \quad \left(\frac{1}{2}\right)^{0}+\left(\frac{1}{2}\right)^{2}+\left(\frac{1}{2}\right)^{4}+\left(\frac{1}{2}\right)^{6}+\left(\frac{1}{2}\right)^{8}=\frac{1}{1}+\frac{1}{4}+\frac{1}{16}+\frac{1}{64}+\frac{1}{256}=\frac{341}{256}$$

$$n=5: \quad \left(\frac{1}{2}\right)^{0}+\left(\frac{1}{2}\right)^{2}+\left(\frac{1}{2}\right)^{4}+\left(\frac{1}{2}\right)^{6}+\left(\frac{1}{2}\right)^{8}+\left(\frac{1}{2}\right)^{10}$$
$$=\frac{1}{1}+\frac{1}{4}+\frac{1}{16}+\frac{1}{64}+\frac{1}{256}+\frac{1}{1024}=\frac{1365}{1024}$$

This suggests the following rather more satisfying visual proof.

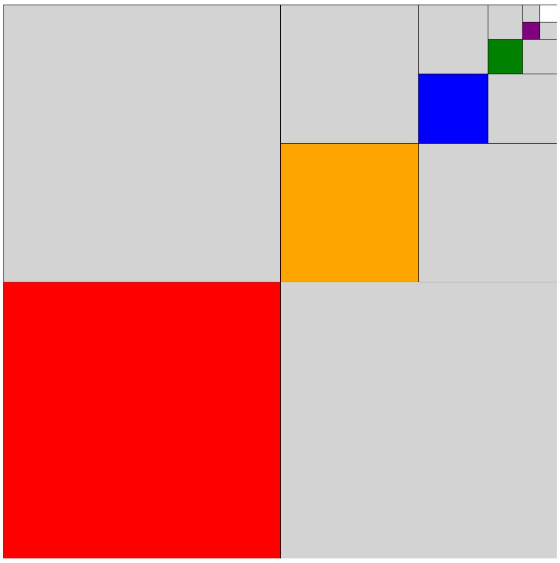


Figure 6.6: A visual suggestion of the infinite sum $\left(\frac{1}{2}\right)^{2n} = 3/4$

The first large square of the image represents 1, subsequent colored smaller squares represent $\frac{1}{4}, \frac{1}{16}, \frac{1}{64}, \ldots$ The total area of the colored squares equals $4/3$, which is the sum of the infinite series.

The code to write it has the following noteworthy features:

```
def draw_squares(ax, x, y, size, depth, colors_iter):
    if depth == 0 or not colors_iter:
        return
    half_size = size / 2
    ax.add_patch(patches.Rectangle((x, y + half_size), half_size, half_size,
... facecolor='lightgrey', edgecolor='black', linewidth=0.5))
    ax.add_patch(patches.Rectangle((x + half_size, y + half_size), half_size, half_size,
... facecolor='white', edgecolor='black', linewidth=0.5))
    color = next(colors_iter, None)
    if color:
        ax.add_patch(patches.Rectangle((x, y), half_size, half_size, facecolor=color,
edgecolor='black', linewidth=0.5))
        ax.add_patch(patches.Rectangle((x + half_size, y),...
    draw_squares(ax, x + half_size, y + half_size, half_size, depth-1, colors_iter)
```

Here the `draw_squares` function is defined with parameters: the axes (`ax`), coordinates (`x, y`), `size`, `depth`, and `colors_iter` with main features:

- **Termination Condition:**
  - The function halts if `depth` equals 0 or if `colors_iter` is exhausted.
- **Half Size Calculation:**
  - Computes half of the `size`, termed as `half_size`.
- **Upper Left/Right Square:**
  - A light/white grey square is drawn in the upper left/right section, framed with a black border.
- **Lower Left Square (Conditional):**
  - A square, colored based on the `colors_iter`, is crafted in the lower left section if a color is available, encircled with a black border.
- **Recursion:**
  - The function recursively invokes itself, focusing on the upper right (white) square for further subdivisions.

6.5 Orbits of n-cycles

As we have seen reptend primes, p_r, are characterised by their reciprocals in base-10 having reptend lengths of $p-1$ giving rise to what we might term "magical numbers" due to their cyclical decimal expansions. So for instance, the decimal representation of $\frac{1}{7}$ is $0.\overline{142857}$, and the multiples of 142857 within the range of 1 to 6 generate cyclic permutations of this sequence, so that $6 \times 142857 = 857142$. These cycles are more elegantly revealed in a number's (*binary*) representation, through the application of a shift map $S(x) = 2x \mod 1$, where the initial value (or "seed") undergoes successive doublings and modular reductions. This process translates the cyclicality of reptend primes into a binary orbit. While a random process by definition cannot be deterministic, the quirk we observe is the counter-intuitive cornerstone of deterministic chaos that a deterministic process can exhibit behavior indistinguishable from randomness.

Formally, a n-cycle is a system defined as a set of n distinct states $\{x_1, x_2, ..., x_n\}$ such that $S(x_i) = x_{(i \mod n)+1}$, with $x_{n+1} \equiv x_1$. It is both deterministic, as the successor of each state is uniquely defined, and chaotic, as the resulting orbit—though finite and repeating—shares properties with sequences generated by random processes, such as a lack of discernible pattern to an uninformed observer. The reciprocals of reptend primes, when expressed in binary, reveal a structured yet complex dynamical behavior through the application of the shift map $S(x) = 2x \mod 1$ underscoring the deterministic yet chaotic behavior of their binary expansions under the shift map.

6.5.1 shift Map

The shift map $S(x) = 2x \mod 1$ acts on a binary fraction, say $x = 0.\overline{010011}$ by doubling x and taking the fractional part of the result.

This action can be visualized as a series of nested squares where each square represents a binary digit (bit) in the binary expansion of x. The unit square is recursively divided into four equal parts (quadrants), and the top right quadrant is subdivided at each step, with the depth of recursion corresponding to the precision of the binary representation. The outer square represents the unit interval $[0,1]$, and each subsequent inner square represents the fractional part after the shift map is applied. The squares get recursively smaller, symbolizing the binary digits shifting to the left.

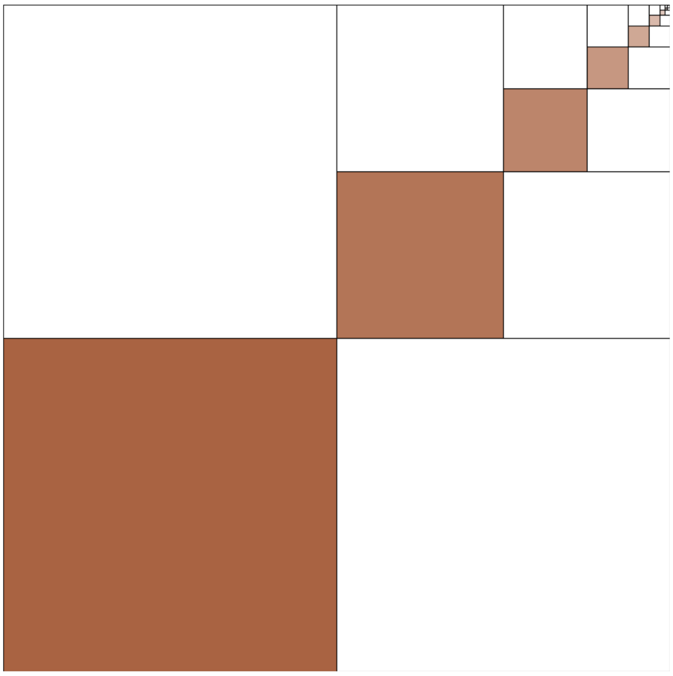


Figure 6.7: Action of the Shift map

The depth of recursion and shading in the code, recursiveSquares.ipynb corresponds to the precision and the binary digits of the fraction being represented:

- The initial seed $x_0 = 0.101$ is represented by a unit square.
- After applying the shift map, the seed becomes $x_1 = S(x_0) = 0.01$ (discarding the overflow), visualized by quartering the unit square and focusing on the top right quadrant.
- This continues recursively, with the depth of recursion indicating the number of binary digits considered and each iteration representing a binary shift operation.

6.5.2 Spider diagrams of n-cycle orbits

The shift map, denoted as $S(x) = 2x \mod 1$, applied to the reciprocal of a reptend prime in binary representation, creates orbits or n-cycle sequences in which the shift map returns to the initial value after n applications:

- **1-cycle** ($x^{[1]}$): The binary representation $0.\overline{1} = 1$ represents a fixed point or a 1-cycle under the shift map, since applying $S(x)$ to 1 gives 0 modulo 1, which effectively maps back to 1, thus forming a 1-cycle orbit denoted as $O^{[1]} = \{1\}$.
- **2-cycle** ($x^{[2]}$): The binary representation $0.\overline{01} = \frac{1}{3}$ alternates between two values under the shift map, representing a 2-cycle orbit. Applying $S(x)$ yields:

$$O^{[2]} = \left\{ \frac{1}{3}, \frac{2}{3} \right\},$$

where each subsequent application alternates between these two values so we have:

$$\begin{aligned} x_0 &= \frac{1}{3} = 0.0101010101... \\ S(x_0) &= \frac{2}{3} = 0.1010101010... = x_1 \\ S(x_1) &= \frac{1}{3} = 0.0101010101... = x_0 \end{aligned}$$

- **3-cycles**: There are two examples of 3-cycles with their respective orbits:
  - The binary representation $0.\overline{001} = \frac{1}{7}$ produces the orbit:

$$O_1^{[3]} = \left\{ \frac{1}{7}, \frac{2}{7}, \frac{4}{7} \right\}.$$

  - The binary representation $0.\overline{011} = \frac{3}{7}$ results in the orbit:

$$O_2^{[3]} = \left\{ \frac{3}{7}, \frac{6}{7}, \frac{5}{7} \right\}.$$

  These 3-cycles show that the shift map produces three distinct values before repeating, illustrating the cyclical nature of the map when applied to these seeds.

In the case of reptend primes, the length of the cycle in the binary representation of their reciprocals is maximal, equating to the prime minus one. This property allows for a rich structure of orbits, with each reptend prime generating an n-cycle that displays its unique characteristics in both its numerical and binary form under the shift map dynamics.

The code, ShiftOperatorOrbitsReptend.ipynb creates the following

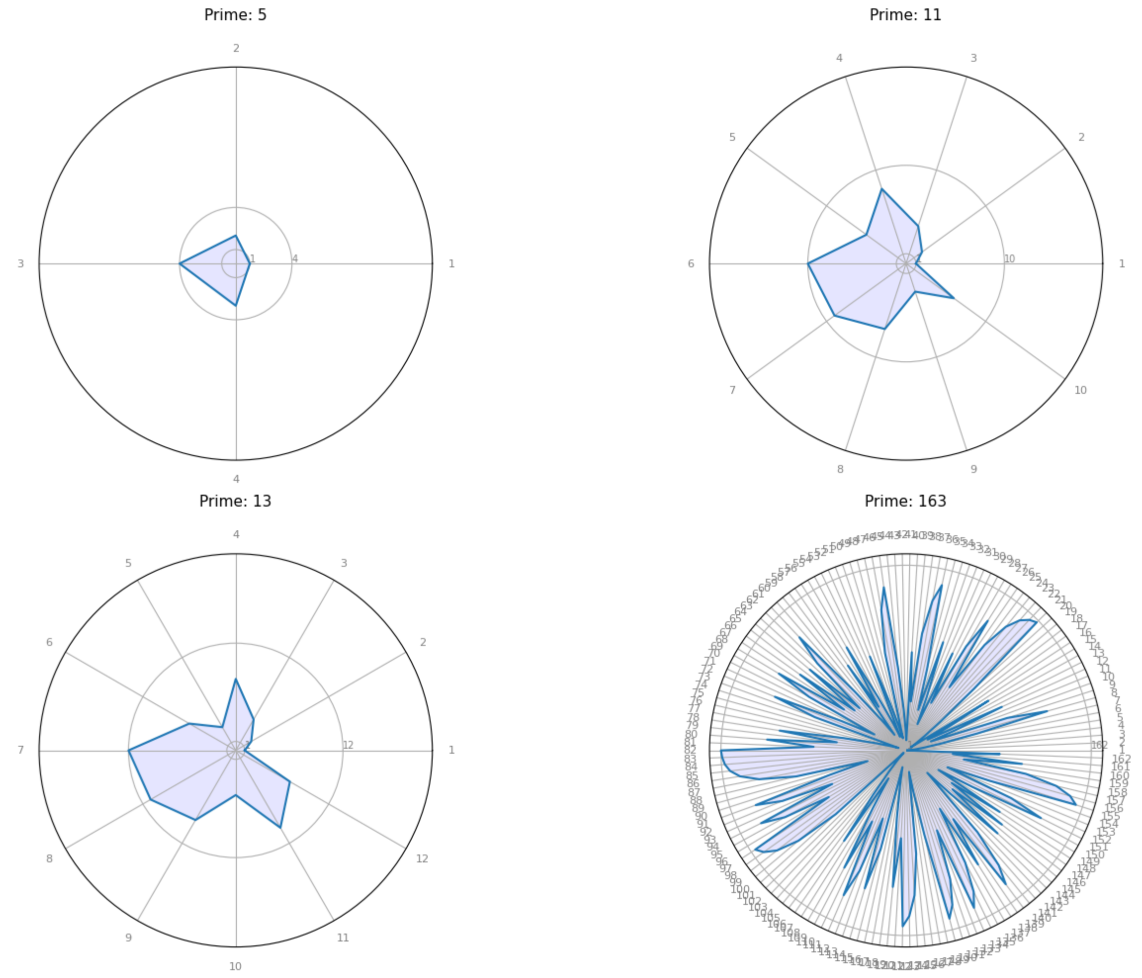


Figure 6.8: Orbits for reptend primes,$p_r = 5, 11, 13, 163$

This is all to say that while a random process cannot be deterministic a entirely deterministic process (as it happens) can (at least) appear be totally random. The code reptendFinder.ipynb identifies those primes giving rising to maximal periods.

R Consider how the notion of reptend is linked to the underlying base (decimal) representation of a number. What does binary representation to a number's cyclicality?

6.6 Primitive Roots and Base-b Reptend Primes

A primitive root modulo n is a number that, when raised to successive powers, generates all the integers from 1 up to $n-1$, inclusive, exactly once when considered modulo n and is a cornerstone concept of cyclic groups in cryptography.

Definition 19 *primitive root modulo n* is a number g which if for every integer a such that $1 \leq a < n$ and a is coprime to n (i.e., $\gcd(a,n) = 1$), there exists an integer k such that $g^k \equiv a \mod n$.

The smallest positive integer k for which $g^k \equiv 1 \mod n$ equals $\phi(n)$, where ϕ denotes Euler's totient function. For a prime p, we have $\phi(p) = p-1$, indicating that a primitive root g modulo a prime p must satisfy $g^k \equiv 1 \mod p$ only when $k = p-1$, covering all integers from 1 to $p-1$ as k ranges from 1 to $p-1$. So When we say "2 is a primitive root modulo 17", it implies that the powers of 2, taken modulo 17, generate all the integers from 1 to 16 in some order, without repetition, before the sequence repeats. The code, primitiveRootsModulo.ipynb checks each number n up to some number $m-1$ to determine if it is a primitive root modulo m.

```
def check_primitive_root(modulus, n):
    residues = set()
    for k in range(1, modulus):
        residue = pow(n, k, modulus)
        print(f"{n}^{k} mod {modulus} = {residue}")
        residues.add(residue)
    return len(residues) == modulus - 1
modulus = 11
primitive_roots = []

for n in range(1, modulus):
    print(f"Checking if {n} is a primitive root modulo {modulus}:")
    if check_primitive_root(modulus, n):
        primitive_roots.append(n)

print(f"Primitive roots modulo {modulus}: {primitive_roots}")
```

For $m = 11$ the code calculates $n^k \mod 11$ for each k from 1 to 10, tracking residues to ensure uniqueness. Upon completing the loop for a given n, it verifies if the collection of residues matches $m-1$, indicating a full cycle and, consequently, that n is a primitive root modulo 11.

$2^1 \mod 11 = 2$
$2^2 \mod 11 = 4$
$2^3 \mod 11 = 8$
$2^4 \mod 11 = 5$
$2^5 \mod 11 = 10$
$2^6 \mod 11 = 9$
$2^7 \mod 11 = 7$
$2^8 \mod 11 = 3$
$2^9 \mod 11 = 6$
$2^{10} \mod 11 = 1$

2 is a primitive root modulo 11.

$3^1 \mod 11 = 3$
$3^2 \mod 11 = 9$
$3^3 \mod 11 = 5$
$3^4 \mod 11 = 4$
$3^5 \mod 11 = 1$
$3^6 \mod 11 = 3$
$3^7 \mod 11 = 9$
$3^8 \mod 11 = 5$
$3^9 \mod 11 = 4$
$3^{10} \mod 11 = 1$

3 does not generate a full cycle modulo 11.

An examination of the primitive roots modulo n elucidates the cyclic properties of numbers within modular arithmetic and facilitates our extension to base-b reptend primes by understanding the maximal repeating cycles of $1/p$ in any chosen base b. .

6.7 Bakers Folding Interleaving Chaotic Map

The Baker's map, [3] is a chaotic map that is an instructive example of a system exhibiting both chaotic and regular behavior. Imagine a square piece of dough which then compressed in one direction to half its height, while simultaneously being stretched in the perpendicular direction to double its length, keeping the area constant. The resulting rectangular shape is then cut in half, and the right half is placed on top of the left half, restoring the original square shape. This process is then repeated. Formally, the Baker's map is defined on the unit square $[0,1]\times[0,1]$ and it transforms a point (x,y) within this square as follows:

$$B(x,y)=\begin{cases}(2x,\frac{y}{2}) & \text{if } 0\leq x\leq 0.5,\\ (2x-1,\frac{y+1}{2}) & \text{if } 0.5<x\leq 1.\end{cases} \tag{6.1}$$

This piece-wise definition ensures the area is preserved and the square domain is maintained after the transformation. This is performed in the snippet of bakerMapOncircleIntersectipynb

```
def baker_map(x, y):
if x <= 0.5:
    return x * 2, y / 2
else:
    return (2 * x - 1), (y + 1) / 2
```

which delivers the following set of plots.

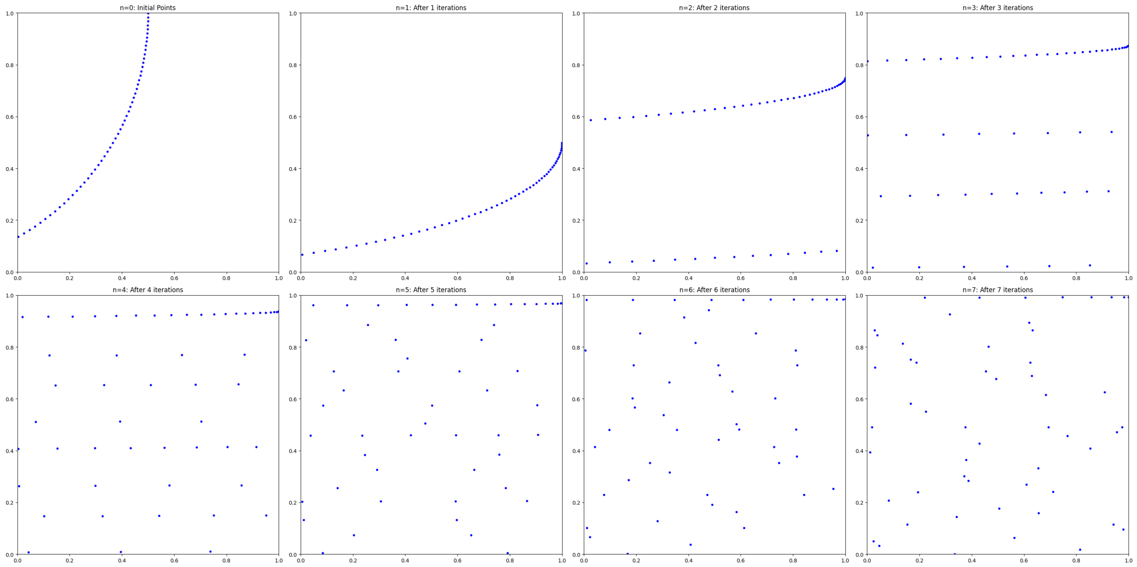


Figure 6.9: Baker stretch-compress-cut-add-Map

Each iteration is visualized in a series of subplots, showing a progression from order to chaos:

- **Intersection Calculation:** The initial set of points is determined by the intersection of lines of the form $y = mx + m$ with a unit circle inscribed in a unit square.

  ```
  intersection_points = [baker_map(x, y) for x, y in intersection_points]
  ```

- **Iterative Application:** The map is applied iteratively, with each iteration representing one folding cycle resulting in points becoming increasingly scattered, demonstrating the map's mixing property.

6.8 Benford's Law

Benford's Law, or the First-Digit Law, predicts that in many naturally occurring collections of numbers, the leading digit is more likely to be small digit. The probability $P(n)$ of the first digit of a number being n is given by the logarithmic distribution:

$$P(n) = \log_{10}\left(\frac{n+1}{n}\right)$$

The table to the right lists the probabilities $P(n)$ for each first digit n from 1 to 9 according to Benford's Law.

First Digit n	Probability $P(n)$
1	30.1%
2	17.6%
3	12.5%
4	9.7%
5	7.9%
6	6.7%
7	5.8%
8	5.1%
9	4.6%

Benford's Law has practical applications in fields such as forensic accounting and fraud detection, where deviations from this distribution can indicate anomalies or fabricated data. We note for instance that an account with a 1% annual interest rate compounded monthly, the time to double, triple etc the account balance over time follows the law.

Multiplier	Time taken to @ 1.0%	Time between @ 1.0%	Probability
Double	70	70	30.17%
Triple	111	41	17.67%
Quadruple	140	29	12.50%
Quintruple	162	22	9.48%
Six x	181	19	8.19%
Seven x	196	15	6.47%
Eight x	209	13	5.60%
Nine x	221	12	5.17%
Ten x	232	11	4.74%

Benford's empirical rule reflects nature counting geometrically rather than arithmetically despite our aversion to logarithms even as our senses are tuned to the environment's power laws. We explore this geometric nature directly by looking at the series $2^0, 2^1, 2^2, ..2^n$ and its scaled versions by deploying benfordGeometricSeries.ipynb which performs the first digit extraction in the function:

```
from collections import defaultdict
    def first_digit_frequencies(multiplier, power_range):
    frequencies = defaultdict(int)
    for power in range(power_range):
        number = multiplier * (2 ** power)
        digit = first_digit(number)
        frequencies[digit] += 1
    return frequencies
```

delivers the set of histograms for the power 2 series law, B as well as those scaled by factors of $2,3,5,7$

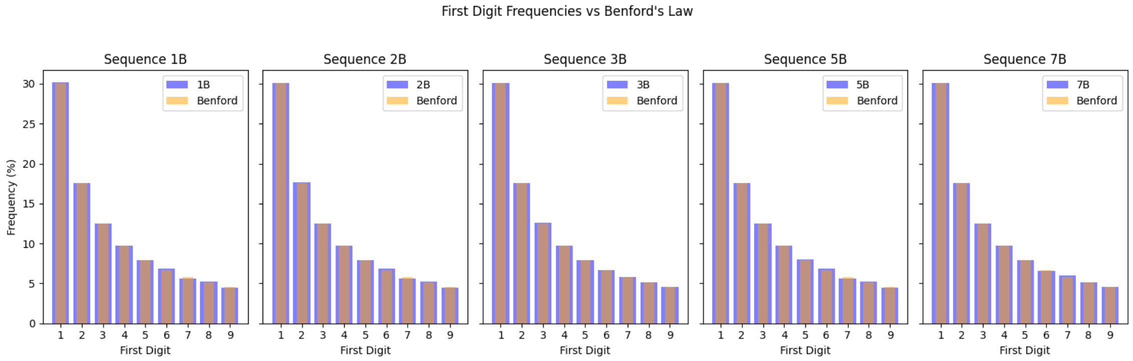


Figure 6.10: Benford behaviour of 2^n and its scaled versions

Mersennes Prime: also follow the Law as the BenfordMersennes.ipynb bears out. The essential elements of the code are a function to retrieve the first digit of a number and a calculation of the observed frequencies

```
def first_digit(n):
    return int(str(n)[0])
observed_frequencies = [0] * 9  # For digits 1 through 9
for prime in mersenne_primes:
    digit = first_digit(prime)
    observed_frequencies[digit - 1] += 1
```

Fibonacci Numbers: despite its deterministic nature, also exhibits leading digits in its numbers that follow the Law. fibonacciBenford.ipynb delivers the sequence to strip down with snippet

```
def fibonacci_sequence(n):
    fib_sequence = [1, 1]
    while len(fib_sequence) < n:
        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])
    return fib_sequence
```

Below are the Histograms of first number frequencies with a comparative Benford distribution.

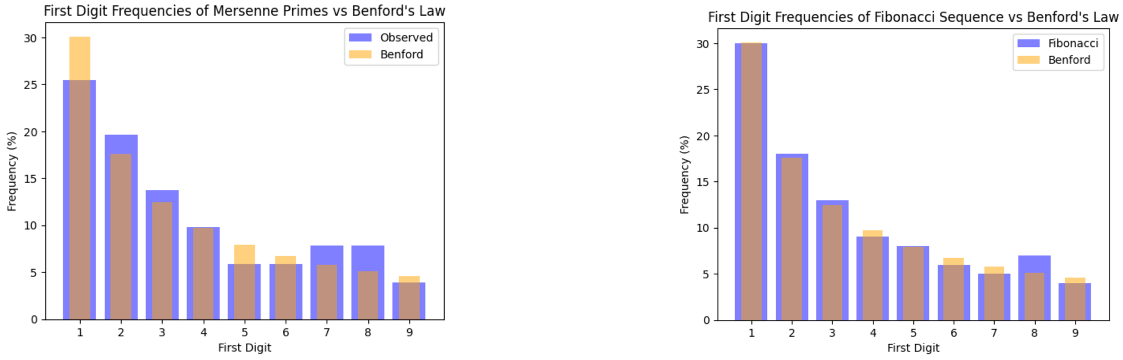


Figure 6.11: Mersenne's Primes and Fibonacci sequences following Benford's Law.

For Fibonacci n=100, Chi-sq = 1.05, P-value = 0.9979, while for n=1000 Chi-sq = 0.20, P-value = 1.0000.

Conclusion: For all the tested figurative number series, the Chi-squared values are below the critical value of 15.51 for 8 degrees of freedom at a significance level of 0.05, and the p-values are above 0.05. Therefore, we do not reject the null hypothesis for either of the sequences, concluding that the distribution of first digits in these number series does not show significant deviation from Benford's Law.

Figurative Number Sequences

To test the geometric nature of abeyance to Benford we explore the quadratic series of various figurative number sequences (Triangle, Square, Pyramid, Cubic, and Pentagon):

```
def triangle_numbers(n):
    return [i * (i + 1) // 2 for i in range(1, n + 1)]
def square_numbers(n):
    return [i ** 2 for i in range(1, n + 1)]
def pyramid_numbers(n):
    return [i * (i + 1) * (2 * i + 1) // 6 for i in range(1, n + 1)]
def cubic_numbers(n):
    return [i ** 3 for i in range(1, n + 1)]
def pentagon_numbers(n):
    return [i * (3 * i - 1) // 2 for i in range(1, n + 1)]
```

In order to explicitly determine deviation from the Benford distribution we apply a Chi-squared test to the distribution of their first digits. Accordingly we assume the null hypothesis (H0) that the distribution follows the law, while the alternative hypothesis (H1) suggests a deviation from it as implemented in the following code snippet:

```
for name, frequencies in all_frequencies.items():
    observed = [frequencies[digit] for digit in range(1, 10)]
    expected = [n * (p / 100) for p in expected_benford]
    chi_stat, p_value = chisquare(f_obs=observed, f_exp=expected)
    print(f"Sequence {name}: Chi-sq = {chi_stat:.2f}, P-value = {p_value:.4f}")
    if chi_stat > critical_value:
        print(f"  Result for {name}: Reject the hypothesis ...")
    else:
        print(f"  Result for {name}: Do not reject the hypothesis ...")
```

When n=100 we note the following suggestive results:

- **Triangle Numbers:** Chi-sq = 11.83, P-value = 0.1589. The Chi-squared value is below the critical value of 15.51, and the p-value is above 0.05, indicating no significant deviation from Benford's Law. Hence, we do not reject H0 for Triangle Numbers.
- **Square Numbers:** Chi-sq = 9.05, P-value = 0.3380. As the Chi-squared value is below the critical threshold, and the p-value is well above 0.05, the Square Numbers do not significantly deviate from Benford's Law so we do not reject H0 for Square Numbers.
- **Pyramid Numbers:** Chi-sq = 4.69, P-value = 0.7898. H0 is thus not rejected for Pyramid Numbers.
- **Cubic Numbers:** Chi-sq = 3.60, P-value = 0.8910. H0 is not rejected for Cubic Numbers.

- **Pentagon Numbers:** Chi-sq = 5.27, P-value = 0.7282. H0 is not rejected.

Upon extending the series to n=1000 we see rejection across the board:

Sequence	Chi-sq	P-value	Result
Triangle	104.70	0.0000	Reject the hypothesis that the distribution follows Benford's Law.
Square	103.42	0.0000	Reject the hypothesis
Pyramid	32.42	0.0001	Reject the hypothesis
Cubic	46.38	0.0000	Reject the hypothesis
Pentagon	39.51	0.0000	Reject the hypothesis

Table 6.2: Chi-squared Test Results for Various Sequences

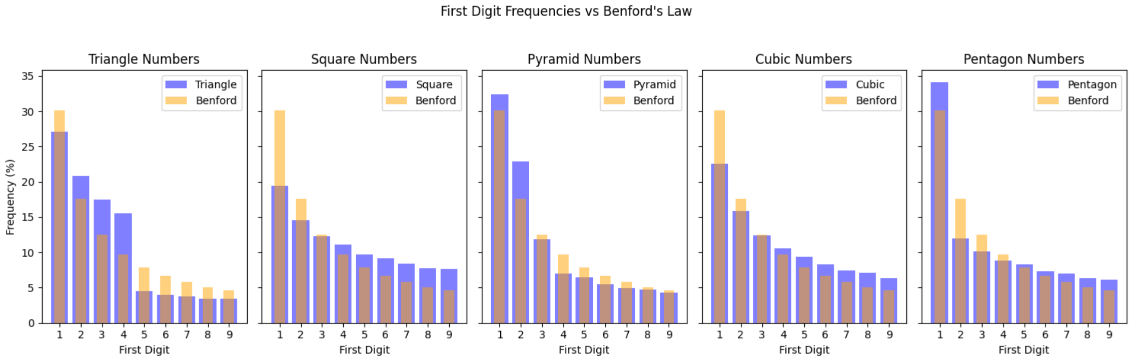
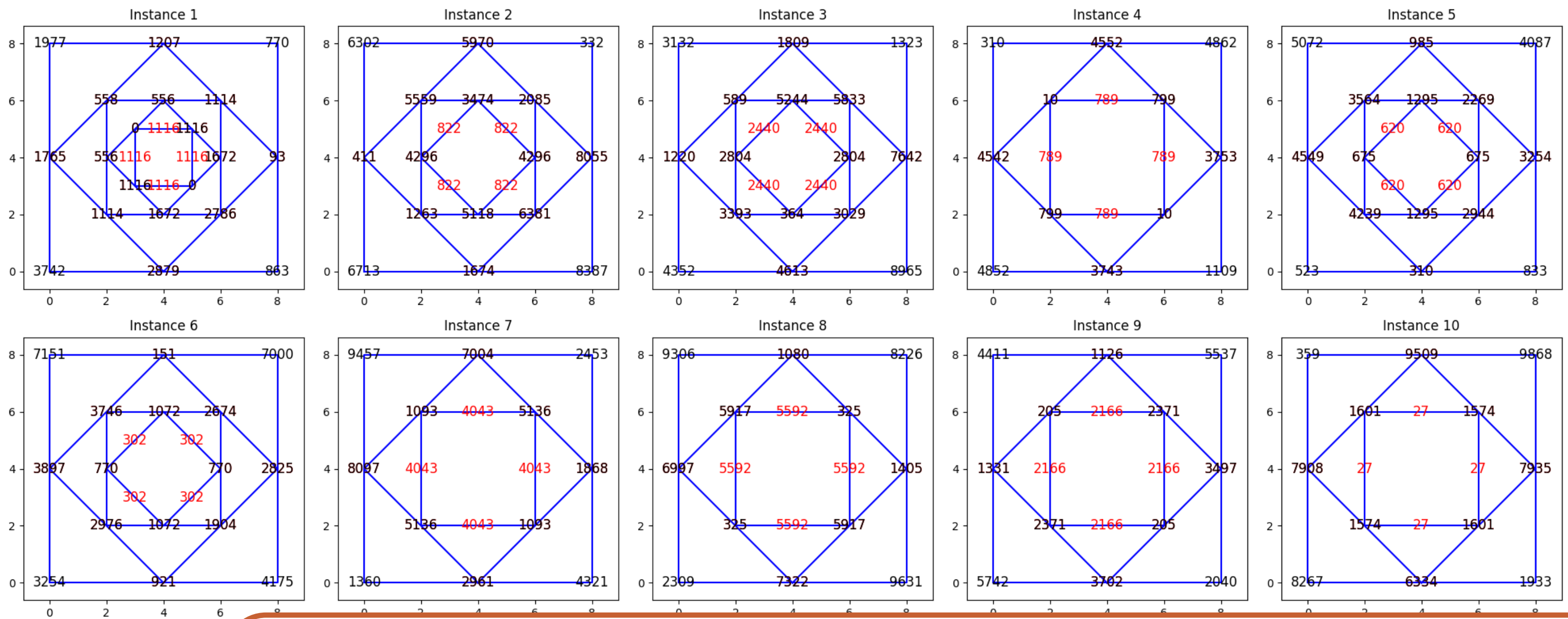


Figure 6.12: Benfordian behaviour of the first n=1000 terms of various figurative series

6.9 Aliquot Sequences

Aliquot Sequence: A sequence that begins with any positive integer where each subsequent term is the sum of the *proper divisors* of the previous term. A *proper divisor* is a positive divisor of a number n other than n itself. This distinction ensures no sequence gets stuck with repeating the number itself.

1. **Terminating Sequence:** Ends when it reaches a prime number or 1, as their only proper divisor is 1.
2. **Cyclical Behavior:** If a sequence revisits a number, it enters a cycle.
   - **Perfect Numbers:** A cycle of length 1, e.g., 28 (sum of divisors: 1 + 2 + 4 + 7 + 14 = 28).
   - **Amicable Numbers:** A cycle of length 2, e.g., 220 and 284.
   - **Sociable Numbers:** A cycle of length greater than 2. The length of the cycle defines the sociability of the numbers.
     - 3-cycle: 12, 16, 15.
     - 4-cycle: 1264460, 1547860, 1727636, 1305184.
     - 5-cycle: Example missing in common knowledge till 2021.
   - Other lengths (e.g., 26 terminates without cycling back).



7. Polynomial Sequences

Monte Carlo Simulation of Square Vertex sequences

Examining differences to explore polynomial sequences is a well-acknowledged mathematical tool. A sequence is deemed to follow (be generated by) a polynomial of order n if the n-th differences are constant.

- **First Differences:** Constant first differences correspond to linear sequences.
- **Second Differences:** Constant second differences align with quadratic sequences, portraying a fixed curve pattern in the sequence of numbers.
- **Third Differences:** Constant third differences denote cubic sequences, highlighting a three-dimensional, curved pattern in the sequence of numbers.

The plots provide a novel depiction of the world of arithmetic differences and random numbers.

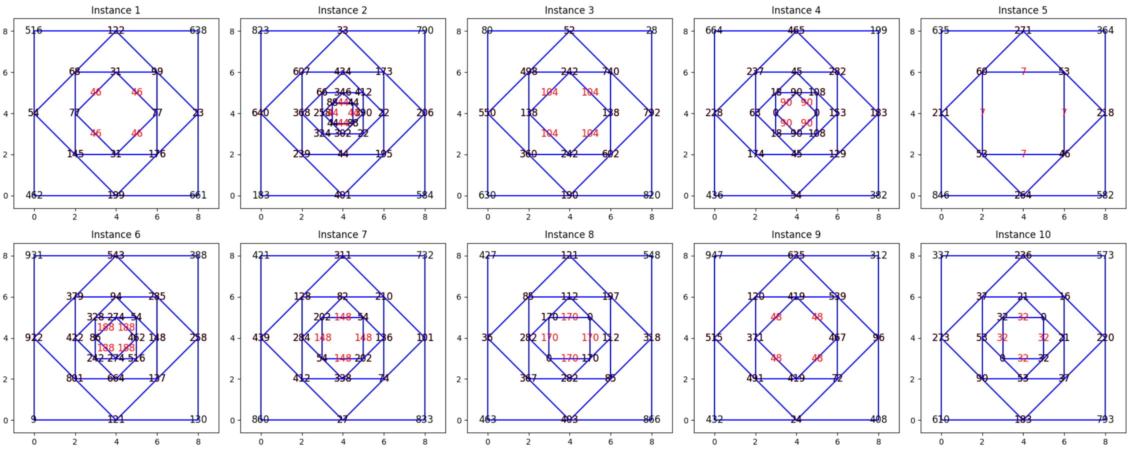


Figure 7.1: Ten Square Sequences for interval $[1, 1000]$.

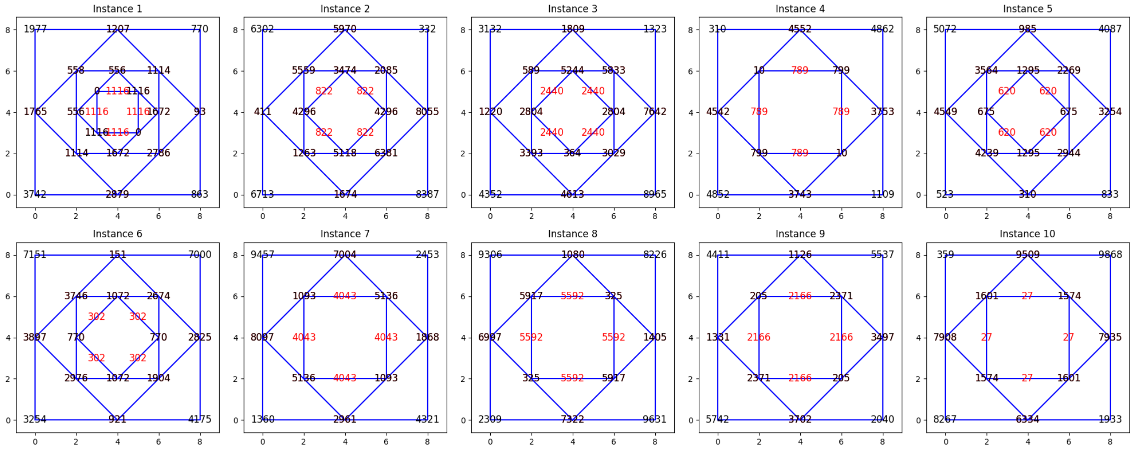
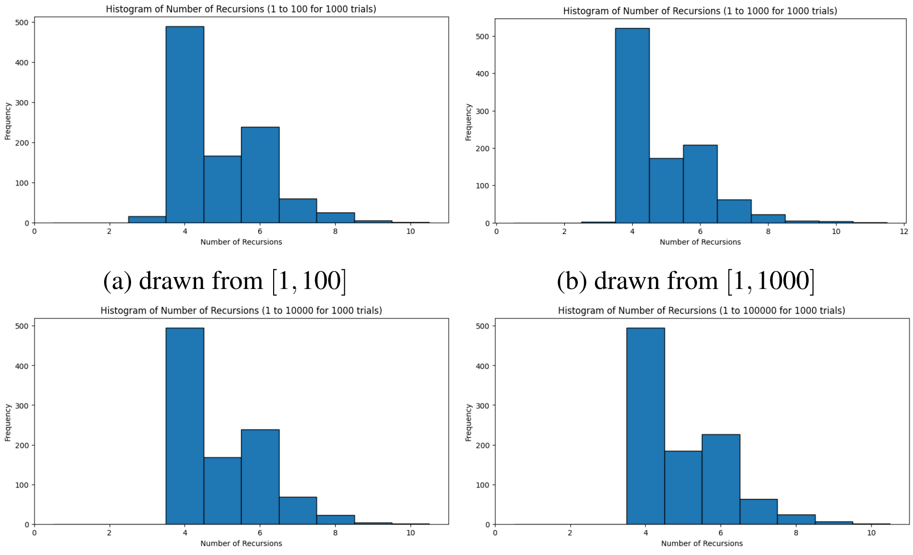


Figure 7.2: Ten Square Sequences for interval $[1,10000]$.

Each square in the plots is labelled with numbers randomly drawn from an interval $[1,l]$, where l is a user-defined limit, ensuring a unique numerical landscape for each square. In the construction[1] [4] of these squares by multipleSquare-differenceDrawVertex.ipynb, every vertex is labeled with a unique number, drawn randomly from the interval $[1,l]$ without replacement. At each side's midpoint of the larger square, the absolute difference between the two adjacent vertices is calculated and displayed. These midpoints subsequently serve as the vertices for an inscribed square, and the process is recursively repeated until all vertices hold the same number. MonteCarloSquareMidpointVertex.ipynb simulates the construction a thousand times and plots a histogram of the depth of recursion necessary for mid point vertex differences to settle on a value. Contrary to expectations, a seemingly systematic procedure does not settle on a fixed number too early. Despite the pattern of calculating differences (which might suggest a quick convergence to a fixed number), the randomness infused at each vertex ensures a diverse and unpredictable pathway to uniformity.



(a) drawn from $[1,100]$

(b) drawn from $[1,1000]$

(c) $drawnfrom[1,10000]$

(d) drawn from $[1,100,000]$

Figure 7.3: Monte Carlo Simulations showing Depth of recursion required given draws widths

[1] This construction is taken from "Even More Mathematical Activities", Brian Bolt Cambridge Educational, 1987".

7.1 Worpitsky triangle

Consider the problem, [15] of counting the total number of squares in a chessboard,

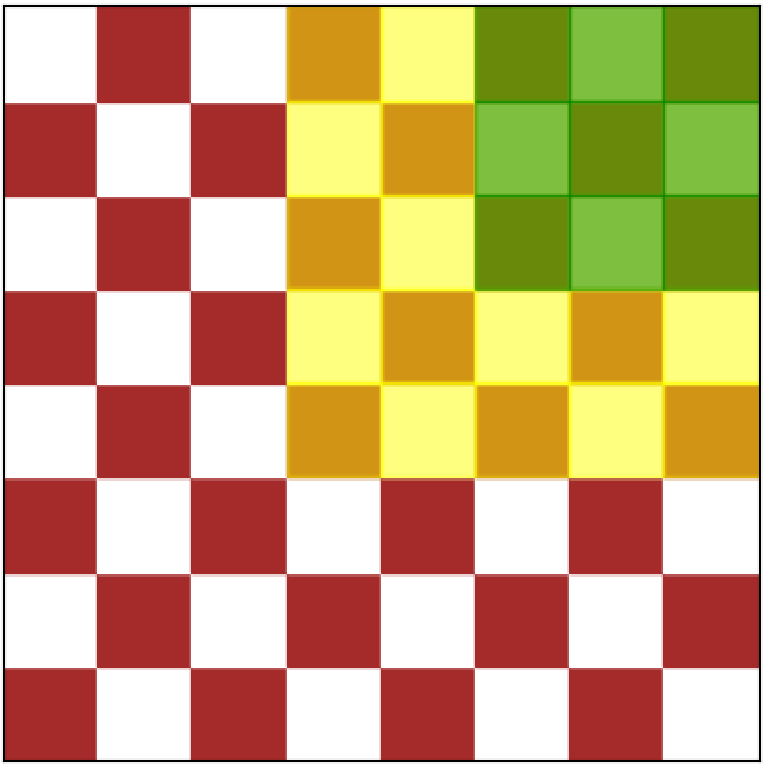


Figure 7.4: How many squares in a 8×8 chessboard.

The correct total number of squares on an 8x8 chessboard, including all sizes of squares from 1x1 to 8x8, is calculated as the sum of the first eight square numbers:

For 1x1 squares: $8 \times 8 = 64$

For 2x2 squares: $7 \times 7 = 49$

For 3x3 squares: $6 \times 6 = 36$

For 4x4 squares: $5 \times 5 = 25$

For 5x5 squares: $4 \times 4 = 16$

For 6x6 squares: $3 \times 3 = 9$

For 7x7 squares: $2 \times 2 = 4$

For 8x8 squares: $1 \times 1 = 1$

Adding these gives: 64 + 49 + 36 + 25 + 16 + 9 + 4 + 1 = 204. Ponder now how you would approach answering the natural extension to this problem, of how many cubes are in a 8×8 Rubik's cube. Is there a smart way to calculate such a sum? We will outline first a relatively non standard way and then certainly a more novel way that will enable us to count the hypercubes in any hyper-Rubik's cube.

7.1.1 Sum of $1^j + \ldots + n^j$

One way to proceed -if one did not know any better -would be to derive formula for the sum of the natural numbers and their higher powers. Let us for completeness do this for $p = 1,2,3,4$ by considering the difference between consecutive square numbers $i^2 - (i-1)^2 = 2i - 1$ and using a bit of telescoping, while summing these differences from $i = 1$ to n gives:

$$\sum_{i=1}^{n}(i^2 - (i-1)^2) = \sum_{i=1}^{n}(2i-1)$$

The left-hand side telescopes, meaning all terms except the first and the last cancel out:

$$n^2 - 0^2 = \sum_{i=1}^{n}(2i-1) \text{ so we are left with: } n^2 = \sum_{i=1}^{n}(2i-1)$$

Now, notice that the sum on the right-hand side can be rewritten as:

$$\sum_{i=1}^{n} 2i - \sum_{i=1}^{n} 1 = 2\sum_{i=1}^{n} i - n \text{ and so, we have: } n^2 = 2\sum_{i=1}^{n} i - n.$$

We rearrange this to solve for the sum of the first n natural numbers as:

$$2\sum_{i=1}^{n} i = n^2 + n \text{ so the first } n \text{ natural numbers sum as the triangle numbers: } \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Sum of the square numbers

To derive the formula for the sum of squares, similarly consider rather perversely the sum of cubes from 1^3 to n^3 and from 0^3 to $(n-1)^3$:

$$\begin{aligned}\sum_{i=1}^{n} i^3 - \sum_{i=0}^{n-1} i^3 &= \sum_{i=1}^{n}\left(i^3 - (i-1)^3\right) \\ &= \sum_{i=1}^{n}\left(3i^2 - 3i + 1\right)\end{aligned}$$

Noticing that the left-hand side telescopes to n^3:

$$\sum_{i=1}^{n} i^3 - \sum_{i=0}^{n-1} i^3 = n^3, \text{ we have: } n^3 = \sum_{i=1}^{n}\left(3i^2 - 3i + 1\right) = 3\sum_{i=1}^{n} i^2 - 3\sum_{i=1}^{n} i + \sum_{i=1}^{n} 1$$

Using our triangle formula for the sum of the first n natural numbers, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, we have:

$$n^3 = 3\sum_{i=1}^{n} i^2 - 3\left(\frac{n(n+1)}{2}\right) + n = 3\sum_{i=1}^{n} i^2 - \frac{3n(n+1)}{2} + n$$

Rearranging as $3\sum_{i=1}^{n} i^2 = n^3 + \frac{3n(n+1)}{2} - n$, means we solve for the sum of squares as:

$$\sum_{i=1}^{n} i^2 = \frac{n^3 + \frac{3n(n+1)}{2} - n}{3} = \frac{1}{6}n(n+1)(2n+1)$$

Sum of higher order numbers

For cubics, we proceed similarly with the sum of quartics from 1^4 to n^4 and from 0^4 to $(n-1)^4$:

$$\begin{aligned}\sum_{i=1}^{n} i^4 - \sum_{i=0}^{n-1} i^4 &= \sum_{i=1}^{n} \left(i^4 - (i-1)^4\right)\\ &= \sum_{i=1}^{n} \left(4i^3 - 6i^2 + 4i - 1.\right)\end{aligned}$$

Noticing that the left-hand side telescopes, $\sum_{i=1}^{n} i^4 - \sum_{i=0}^{n-1} i^4 = n^4$ we have

$$n^4 = \sum_{i=1}^{n} \left(4i^3 - 6i^2 + 4i - 1\right) = 4\sum_{i=1}^{n} i^3 - 6\sum_{i=1}^{n} i^2 + 4\sum_{i=1}^{n} i - \sum_{i=1}^{n} 1.$$

Using the formulas for the sum of the first n natural numbers and the sum of squares, we get:

$$\begin{aligned}n^4 &= 4\sum_{i=1}^{n} i^3 - 6\left(\frac{n(n+1)(2n+1)}{6}\right) + 4\left(\frac{n(n+1)}{2}\right) - n\\ &= 4\sum_{i=1}^{n} i^3 - n(n+1)(2n+1) + 2n(n+1) - n.\end{aligned}$$

Rearranging to solve for the sum of cubes:

$$\begin{aligned}4\sum_{i=1}^{n} i^3 &= n^4 + n(n+1)(2n+1) - 2n(n+1) + n,\\ \sum_{i=1}^{n} i^3 &= \frac{n^4 + n(n+1)(2n+1) - 2n(n+1) + n}{4} = \frac{n^4 + 2n^3 + n^2}{4} = \frac{1}{4}n^2(n+1)^2.\end{aligned}$$

We can keep playing this game deriving the sum of quartics, from the shifted sum of quintics from 1^5 to n^5 and from 0^5 to $(n-1)^5$:

$$\sum_{i=1}^{n} i^5 - \sum_{i=0}^{n-1} i^5 = \sum_{i=1}^{n} \left(i^5 - (i-1)^5\right) = \sum_{i=1}^{n} \left(5i^4 - 10i^3 + 10i^2 - 5i + 1\right),$$

so that the sum of quartics follows from:

$$5\sum_{i=1}^{n} i^4 = n^5 + [\text{terms involving lower powers of } i]$$

For quintics, from the sum of sextics from 1^6 to n^6 and from 0^6 to $(n-1)^6$ we have:

$$\sum_{i=1}^{n} i^6 - \sum_{i=0}^{n-1} i^6 = \sum_{i=1}^{n} \left(i^6 - (i-1)^6\right) = \sum_{i=1}^{n} \left(6i^5 - 15i^4 + 20i^3 - 15i^2 + 6i - 1\right),$$

so the sum of quintics follows from:

$$6\sum_{i=1}^{n} i^5 = n^6 + [\text{terms involving lower powers of } i]$$

7.1.2 The Pyramid number of the Twelve Days of Christmas presents

The classic Christmas song *"The 12 Days of Christmas"* implies a cumulative gift-giving pattern. If each day's gifts are added cumulatively to the total from the previous days, the problem is to find the total number of presents received over the 12 days.

Gauss is famously known for quickly finding the sum of a series of 1 to 100, If we write the series $1,2,3,\ldots,n$ backwards as $n,n-1,n-2,\ldots,1$, and sum each pair, we get for each pair $n+1$. For n numbers, there are n pairs each summing to $n+1$ and the total sum of the series is half the product of the number of pairs or equivalently half the sequence of oblong numbers $1\times 2, 2\times 3, 3\times 4,\ldots,n\times(n+1)$. So our total number of gifts is halve the sum of the first 12 oblong numbers :

$$\frac{1}{2}\sum_{n=1}^{12} n(n+1) = \sum_{n=1}^{12}\frac{n(n+1)}{2}$$

i.e. the first 12 triangular numbers, a sum of a quadratic sequence whose general nth term is given by $a_n = An^2+Bn+C$, where A, B, and C are constants and whose sum of the first n terms, S_n is:

$$S_n = \sum_{i=1}^{n}(Ai^2+Bi+C) \tag{7.1}$$

$$= (A+B+C)+(4A+2B+C)+(9A+3B+C)+\ldots+(An^2+Bn+C) \tag{7.2}$$

For oblong, *pryonic* numbers, we have $A=1$, $B=1$, and $C=0$ so the nth term is $P_n = n^2+n$:

$$P_n = \sum_{n=1}^{n}(n^2+n) = \sum_{n=1}^{n} n^2 + \sum_{n=1}^{n} = \frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}$$

$$= \frac{n(n+1)(2n+1)+3n(n+1)}{6} = \frac{n(n+1)\left[(2n+1)+3\right]}{6} = \frac{n(n+1)(n+2)}{3}$$

$$P_{12} = \frac{12\times 13\times 14}{3} = \frac{2184}{3} = 728,$$

and 364 presents are delivered over the twelve days of Christmas. Noting the number generators

naturals: $\binom{n+0}{1}$, triangles: $\binom{n+1}{2} = \frac{n(n+1)}{2}$, pyramids: $\binom{n+2}{3}$, Pascal's triangle is:

$$\begin{array}{cccc}
\binom{n-1}{0} & \binom{n+0}{1} & \binom{n+1}{2} & \binom{n+2}{3} \\
\hline
\binom{1-1}{0} & & & \\
\binom{2-1}{0} & \binom{1+0}{1} & & \\
\binom{3-1}{0} & \binom{2+0}{1} & \binom{1+1}{2} & \\
\binom{4-1}{0} & \binom{3+0}{1} & \binom{2+1}{2} & \binom{1+2}{3} \\
\vdots & \vdots & \vdots & \vdots
\end{array}
\qquad
\begin{array}{cccc}
\binom{n-1}{0} & \binom{n+0}{1} & \binom{n+1}{2} & \binom{n+2}{3} \\
\hline
\binom{0}{0} & & & \\
\binom{1}{0} & \binom{1}{1} & & \\
\binom{2}{0} & \binom{2}{1} & \binom{2}{2} & \\
\binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \\
\vdots & \vdots & \vdots & \vdots
\end{array}$$

so the sums of the third column (triangle numbers) and fourth column (pyramid numbers) are:

$$\sum_{k=0}^{n}\binom{n+k}{2} \text{ and } \sum_{k=0}^{n}\binom{n+k}{3}.$$

We have then as our number of Christmas presents the twelfth pyramid number:

$$\binom{n+2}{3} = \frac{(n+2)(n+1)n}{3!} \qquad \binom{12+2}{3} = \frac{(12+2)(12+1)12}{6} = 364.$$

Pascal's Triangle of Difference of Powers

We collect the coefficients from the differences of consecutive powers noting how they resemble those of the binomial expansion[2], albeit with alternating signs:

Power Difference	Coefficients
$i^1-(i-1)^1$	$1i^0$
$i^2-(i-1)^2$	$2i^1-1$
$i^3-(i-1)^3$	$3i^2-3i+1$
$i^4-(i-1)^4$	$4i^3-6i^2+4i-1$
$i^5-(i-1)^5$	$5i^4-10i^3+10i^2-5i+1$
$i^6-(i-1)^6$	$6i^5-15i^4+20i^3-15i^2+6i-1$
$i^7-(i-1)^7$	$7i^6-21i^5+35i^4-35i^3+21i^2-7i+1$

Recall that the difference of two consecutive squares, from (**??**) told us $4^2-3^2=2\cdot4-1=7$ so from the above we can see now how to determine the difference of two consecutive cubes $4^3-3^3=3\cdot4^2-3\cdot4+1=37$. Whereas Pascal's Triangle comprises rows of numbers formed of the sums of two numbers above:

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}$$

the coefficients in the differences of consecutive powers are formed of a modified Pascal's triangle which comprises the differences between the numbers directly above reading from right to left so that $-6-4=-10$ and $4-(-6)=10$ according to:

$$\begin{array}{ccccccccccc}
 & & & & & -1 & & & & & \\
 & & & & -1 & & 1 & & & & \\
 & & & -1 & & 2 & & -1 & & & \\
 & & -1 & & 3 & & -3 & & 1 & & \\
 & -1 & & 4 & & -6 & & 4 & & -1 & \\
-1 & & 5 & & -10 & & 10 & & -5 & & 1
\end{array}$$

[2] To be sure, the binomial expansion of $(i-(i-1))^n$ is not quite what we are looking for:

$$(i-(i-1))^n=\sum_{k=0}^{n}\binom{n}{k}i^{n-k}(-i+1)^k$$

since for n=2 we have $(i-(i-1))^2=i^2-2i(i-1)+(i-1)^2=i^2-2i^2+2i+i^2-2i+1=1$.

Powered Pascal Triangle

Consider now the numbers in *Worpitzky*, "power Pascale" Triangle, that are obtained by multiplying each of the two numbers directly above by their respective column position before summing them:

$$\begin{array}{ccccccccccc}
 & & & & & 1_1 & & & & & \\
 & & & & 1_1 & & 1_1 & & & & \\
 & & & 1_1 & & 3_2 & & 2_3 & & & \\
 & & 1_1 & & 7_2 & & 12_3 & & 6_4 & & \\
 & 1_1 & & 15_2 & & 50_3 & & 60_4 & & 24_5 & \\
1_1 & & 31_2 & & 180_3 & & 390_4 & & 360_5 & & 120_6 \\
 & & \vdots & & & \vdots & & & \vdots & &
\end{array}$$

The numbers are thus generated according to $7_2 + 12_3 \equiv 7 \times 2 + 12 \times 3 = 50_3$. The following snippet from WorpitskyPascaleTriangleMersennes generates the triangle for any user n

```
def construct_power_pascal(order):
triangle = [[1]]
for i in range(1, order):
    row = [1]
    for j in range(1, i+1):
        left_above_val = triangle[i-1][j-1] * (j) if j-1 >= 0 else 0
        direct_above_val = triangle[i-1][j] * (j+1) if j < len(triangle[i-1]) else 0
        value = left_above_val + direct_above_val
        row.append(value)
    triangle.append(row)
return triangle
```

We can identify the Mersenne's primes, $2^p - 1$ along the second left to right diagonal, $3, 7, 31, 127, 8191$, $131071, 524287$ on p rows $2, 3, 5, 7, 13, 17, 19$ through the snippet:

```
def compare_to_mersenne_primes(column):
return [val for val in column if is_prime(val)]
```

Akin to Pascal's Triangle, in terms of binomial coefficients we will denote Worpitzky thus:

$$\begin{array}{ccccccccccccccc}
 & & & \binom{0}{0} & & & & \quad & & & & {}_0W_0 & & & \\
 & & \binom{1}{0} & & \binom{1}{1} & & & & & & {}_1W_0 & & {}_1W_1 & & \\
 & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & & & & {}_2W_0 & & {}_2W_1 & & {}_2W_2 & \\
\binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} & & {}_3W_0 & & {}_3W_1 & & {}_3W_2 & & {}_3W_3 \\
\vdots & & & \ddots & & & \vdots & & \vdots & & & \ddots & & & \vdots
\end{array}$$

For any given n, this triangle can be used to represent the sum of the series up to n^k for $k \in \mathbb{N}$. We have thus for the cumulative sum up to 1 squares,

$$1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2 = {}_2W_0 \binom{n}{1} + {}_2W_1 \binom{n}{2} + {}_2W_2 \binom{n}{3}.$$

Given a required value of n = 8 for a chessboard, we can use the Worpitsky triangle to compute the sum of power series up to $n = 8$ as follows:

$$1^0+2^0+3^0+4^0+5^0+6^0+7^0+8^0 = 1\binom{8}{1} = 8$$

$$1^1+2^1+3^1+4^1+5^1+6^1+7^1+8^1 = 1\binom{8}{1}+1\binom{8}{2} = 8+28 = 36$$

$$1^2+2^2+3^2+4^2+5^2+6^2+7^2+8^2 = 1\binom{8}{1}+3\binom{8}{2}+2\binom{8}{3} = 8+84+112 = 204$$

$$1^3+2^3+3^3+4^3+5^3+6^3+7^3+8^3 = 1\binom{8}{1}+7\binom{8}{2}+12\binom{8}{3}+6\binom{8}{4}$$

$$= 8+196+672+420 = 1296$$

So our Power-Pascal Triangle code for $d = 8$ up to 4th row: [[1], [1, 1], [1, 3, 2], [1, 7, 12, 6]] gives us the total number of cubes in an $8 \times 8 \times 8$ Rubik's 3-d cube as 1296. This is achieved in the snippet below from WorpitsktTriangleSeriesSum.ipynb

```
def display_sums(triangle, n):
for i, row in enumerate(triangle):
    terms = [f"{coef}*{n}^C_{j+1}" for j, coef in enumerate(row)]
    sum_terms = [f"{coef * binomial_coeff(n, j+1)}" for j, coef in enumerate(row)]
    actual_sum = sum(coef * binomial_coeff(n, j+1) for j, coef in enumerate(row))

    sequence_terms = " + ".join(f"{k+1}^{i}" for k in range(n))
    sum_string = " + ".join(terms)
    sum_values_string = " + ".join(sum_terms)

    print(f"{sequence_terms} = {sum_string} = {sum_values_string} = {actual_sum}")
print()
```

The Worpitsky triangle thus offers an elegant and efficient means of computing sums of power series using combinations. Its structural similarity to Pascal's Triangle, coupled with its distinctive properties, make it a powerful tool for a variety of mathematical calculations. when we multiply it with Pascal we form the triangle: Multiplication Result:

1

1 1

1 6 2

1 28 36 6

1 60 300 240 24

1 155 1800 3900 1800 120

7.1.3 Polynomial Regression

Polynomial sequences can often be more insightfully analyzed by transforming them into a factorial basis rather than examining them in their standard polynomial form. This approach simplifies the process of identifying the sequence generator and deducing the polynomial coefficients.

Motivation for the Change of Basis

The factorial basis transformation is designed to convert the standard polynomial coefficients to those that correspond to a polynomial expressed in terms of falling factorials. The falling factorial $x^{(n)}$ is defined as:

$$x^{(n)} = x \cdot (x-1) \cdot (x-2) \cdot \ldots \cdot (x-n+1)$$

This basis simplifies the calculation of polynomial coefficients and provides a more direct relationship between the sequence values and the coefficients of the generating polynomial.

Using the Worpitzky Matrix

To apply the Worpitzky matrix, we must first transform our given sequence into a factorial basis. Consider a polynomial sequence $P = \{P_1, P_2, \ldots, P_n\}$ with its associated differences D, S, T, F. Rather than analyzing the sequence P directly, we can investigate the sequence generator by examining the transformed sequence $\{P_1, D_1, S_1, F_1, \ldots\}$.

For instance, if P is a quadratic sequence, it suffices to use three terms $Q = \{Q_1, Q_2, Q_3\}$, known as the standard basis, to form the Vandermonde matrix. However, the extrapolated sequence Q_e can be equally generated from the factorial basis $\{Q_1, D_1, S_1\}$, forming a set of linear simultaneous equations. The coefficients of the quadratic $q(x) = ax^2 + bx + c$ can be deduced from these equations:

$$\begin{aligned} 2a &= S_1, \\ 3a + b &= D_1, \\ a + b + c &= Q_1. \end{aligned}$$

Similarly, for a cubic polynomial, the relationships become:

$$\begin{aligned} 6a &= T_1, \\ 12a + 2b &= S_1, \\ 7a + 3b + c &= D_1, \\ a + b + c + d &= Q_1. \end{aligned}$$

These sets of equations can be represented in matrix form, with coefficients related to the Worpitzky Triangle.

Matrix Representation of Polynomial Coefficients

Given a sequence of terms generated by a polynomial, we can represent the system of equations that determine the polynomial coefficients in matrix form. For quadratic, cubic, and quartic polynomials, these matrices correspond to the coefficients of the terms when the polynomials are expressed in the standard basis.

7.2 Matrix form of linear systems

Quadratic Case

For a quadratic polynomial $ax^2 + bx + c$, the matrix equation is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} Q_1 \\ D_1 \\ S_1 \end{bmatrix}$$

Cubic Case

For a cubic polynomial ax^3+bx^2+cx+d, the matrix equation is:

$$\left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \cline{2-4} 7 & 1 & 0 & 0 \\ 12 & 3 & 1 & 0 \\ 6 & 2 & 1 & 1 \end{array}\right] \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} Q_1 \\ D_1 \\ S_1 \\ T_1 \end{bmatrix}$$

Quartic Case

Given a sequence $P: p_0, p_1, p_2, p_3, \ldots, p_n$, we want to find a quartic polynomial $q(n)$ such that:

$$q(n) = an^4 + bn^3 + cn^2 + dn + e$$

where $p_k = q(k)$ for $k = 0, 1, 2, \ldots, n$.

To find the coefficients a, b, c, d, e, we solve the system of linear equations formed by plugging in the values of n corresponding to the given sequence:

$$\begin{cases} a(0)^4 + b(0)^3 + c(0)^2 + d(0) + e = p_0 \\ a(1)^4 + b(1)^3 + c(1)^2 + d(1) + e = p_1 \\ a(2)^4 + b(2)^3 + c(2)^2 + d(2) + e = p_2 \\ a(3)^4 + b(3)^3 + c(3)^2 + d(3) + e = p_3 \\ \vdots \\ a(n)^4 + b(n)^3 + c(n)^2 + d(n) + e = p_n \end{cases}$$

By solving this system, we obtain the coefficients of the quartic polynomial. This results in a system of equations based on evaluating the polynomial at $n = 1, 2, 3, 4$, giving us the values of the sequence:

$$\begin{aligned} P(1) &= a + b + c + d + e = 21 \\ P(2) &= 16a + 8b + 4c + 2d + e = 136 \\ P(3) &= 81a + 27b + 9c + 3d + e = 465 \\ P(4) &= 256a + 64b + 16c + 4d + e = 1176 \end{aligned}$$

We also have $P(0) = e = 0$ since the sequence starts from 0 when $n = 0$.

Using these values, we can set up our matrix equation $A\mathbf{x} = \mathbf{b}$ to solve for the coefficients a, b, c, d, e.

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \\ 81 & 27 & 9 & 3 & 1 \\ 256 & 64 & 16 & 4 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 21 \\ 136 \\ 465 \\ 1176 \\ 0 \end{bmatrix}$$

Since the sequence is generated by a quartic polynomial, the fourth differences should be constant. This constant is the leading coefficient of the n^4 term times 4! (factorial of 4).

Having determined the leading coefficient, one would proceed backwards through the differences to solve for the remaining coefficients of the polynomial. By evaluating the third, second, and first differences (and the zeroth, which is the sequence itself), the coefficients of the n^3, n^2, n^1, and n^0 terms can be determined, respectively.

For a quartic polynomial $ax^4 + bx^3 + cx^2 + dx + e$, the matrix equation and its solution can be illustrated with the specific sequence (0, 21, 136, 465, 1176). whose zeroth, first, Second, and Third differences respectively: P, D, S, T are laid out as follows:

$$\begin{array}{llllllllll}
P: & 0 & & 21 & & 136 & & 465 & & 1176 \\
D: & & 21 & & 115 & & 329 & & 711 & \\
S: & & & 94 & & 214 & & 382 & & \\
T: & & & & 120 & & 168 & & & \\
F: & & & & & 48 & & & &
\end{array}$$

The corresponding matrix equation using this "factorial basis" given the first four differences of the sequence is:

$$\left[\begin{array}{c|cccc}
\multicolumn{1}{c}{1} & 0 & 0 & 0 & 0 \\ \cline{2-5}
15 & 1 & 0 & 0 & 0 \\
50 & 7 & 1 & 0 & 0 \\
60 & 12 & 3 & 1 & 0 \\
24 & 6 & 2 & 1 & 1
\end{array}\right]
\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}
=
\begin{bmatrix} 48 \\ 120 \\ 94 \\ 21 \\ 0 \end{bmatrix}$$

We can see what is meant by this being a factorial basis now. We have:

$$a = \frac{F}{4!} = \frac{48}{24} = 2$$

Where F represents the fourth difference. Given that the fourth difference is 48 and $4! = 24$, the leading coefficient a is 2.

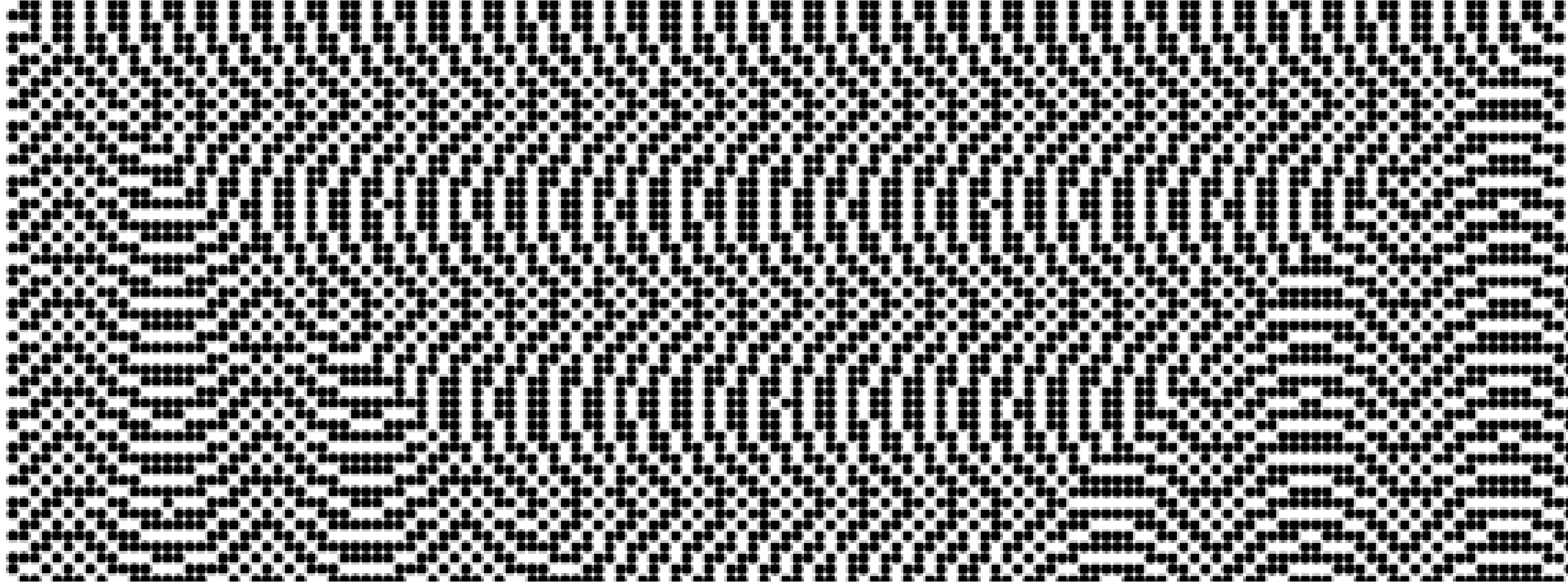
We note that the quartic matrix with the cubic submatrix at the bottom right, has entries of the Worpitzky triangle:

$$\begin{array}{ccccccccc}
 & & & & 1_1 & & & & \\
 & & & 1_1 & & 1_1 & & & \\
 & & 1_1 & & 3_2 & & 2_3 & & \\
 & 1_1 & & 7_2 & & 12_3 & & 6_4 & \\
1_1 & & 15_2 & & 50_3 & & 60_4 & & 24_5
\end{array}
\qquad
\begin{bmatrix}
1_1 & & & & \\
15_2 & 1_1 & & & \\
50_3 & 7_1 & 1_1 & & \\
60_4 & 12_2 & 3_1 & 1_1 & \\
24_5 & 6_3 & 2_2 & 1_1 & 1_1
\end{bmatrix}$$

To solve for the coefficients a, b, c, d, e, we compute the inverse of the 5x5 matrix (if it is invertible) and then multiply it by the right-hand side vector.

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \left[\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 15 & 1 & 0 & 0 & 0 \\ 50 & 7 & 1 & 0 & 0 \\ 60 & 12 & 3 & 1 & 0 \\ 24 & 6 & 2 & 1 & 1 \end{bmatrix} \right]^{-1} \begin{bmatrix} 48 \\ 120 \\ 94 \\ 21 \\ 0 \end{bmatrix}$$

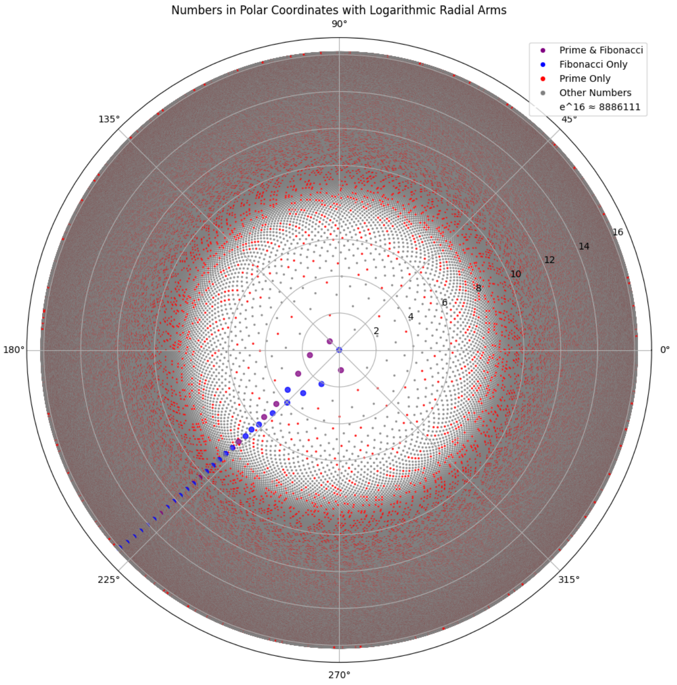
Part Two: Foot hill Explorations



8. Fibonacci Miscellany

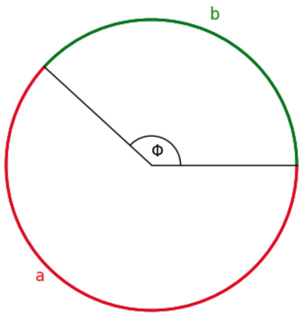
“No man ever steps in the same river twice, for it’s not the same river and he’s not the same man.”
— *Heraclitus*

The Fibonacci sequence starts with $f_1 = 1, f_2 = 2$, and for all $n > 2$, $f_n = f_{n-1} + f_{n-2}$. The plot below represents the first 10,000,000 integers on a polar coordinate system, in which each point’s radial distance from the origin is determined by the natural logarithm of the integer which allows for a more uniform distribution of points that grow exponentially.



(a) Prime-Fibonacci Polar Golden Angle Spiral

The angular position of each integer is incremented by the golden angle, which is defined by the equation $\frac{a+b}{a} = \frac{a}{b}$, where a and b are consecutive terms of the Fibonacci sequence and $a > b$. The golden angle, Φ is the fractional part of a full circle that is not covered when dividing the circumference into a segment proportional to $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$, the ratio of the length of the larger arc, a to the length of the smaller arc, b with the angle subtended by the smaller arc being $\Phi = 360^\circ \times \left(1 - \frac{1}{\phi}\right) \approx 137.507764^\circ$.



We have thus,

$$\begin{aligned}\frac{\text{Golden angle}}{360^\circ} &= \left(1 - \frac{2}{1+\sqrt{5}}\right) = \left(\frac{\sqrt{5}-1}{\sqrt{5}+1}\right) \times \left(\frac{\sqrt{5}-1}{\sqrt{5}-1}\right)\\ \frac{\Phi}{2\pi} &= \left(\frac{(\sqrt{5}-1)^2}{5-1}\right) = \left(\frac{5-2\sqrt{5}+1}{4}\right) = \left(\frac{3-\sqrt{5}}{2}\right),\\ \Phi &= (3-\sqrt{5})\pi \approx 2.39996 \ \text{ rad}\end{aligned}$$

Because this angle is an irrational multiple of 360°, our sequence fills the circle densely, and the points corresponding to the Fibonacci numbers might be expected to eventually distribute across all possible angles. However, due to the nature of the Fibonacci sequence's growth, the points asymptotically align along a specific radial line due to the fact that the ratio of successive Fibonacci numbers converges to the golden ratio. The angle at which this alignment occurs[1] appears for these lowly numbers to be circa 221°. The code, goldenAnglePolarPrimeFibonacci.ipynb achieves this.

Index	Fibonacci Number	Angle (degrees)	Difference from Prior	Ang (radians)	
27	514229	222.46619	-	3.883	$21/17\pi$
28	832040	222.45079	0.01539	3.882	$21/17\pi$
29	1346269	222.42475	0.02604	3.882	$21/17\pi$
30	2178309	222.38331	0.04144	3.881	$21/17\pi$
31	3524578	222.31582	0.06748	3.880	$21/17\pi$
32	5702887	222.20690	0.10892	3.878	$21/17\pi$
33	9227465	222.03049	0.17640	3.875	$37/30\pi$
34	14930352	221.74516	0.28533	3.870	$16/13\pi$

Table 8.1: Fibonacci Numbers and Their Angles

Given that golden angle is irrational, this clustering might not be expected to converge to a single angle. Instead, as you plot more and more Fibonacci numbers, you might expect these

[1] determineAsymptoticAngle.ipynb separately calculates this

clusters to spread out and fill the circle more evenly due to the density property of irrational rotations. Not so.

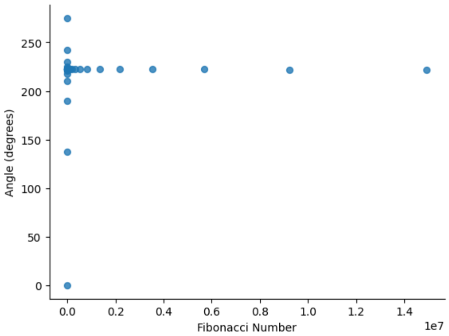


Figure 8.2: Asymptotic Fibonacci Polar Golden Spiral Angle

8.1 Fibonacci primes

The Prime Number Theorem (PNT), as formulated by Gauss, describes the distribution of prime numbers over the natural numbers. It states that the number of primes less than or equal to a given number n can be approximated by the logarithmic integral function $\mathrm{Li}(n)$, but for practical purposes, it is often approximated by the ratio $\frac{n}{\ln(n)}$. This can be expressed mathematically as:

$$\pi(n) \sim \mathrm{Li}(n) \sim \frac{n}{\ln(n)}$$

where $\pi(n)$ is the prime-counting function that counts the number of prime numbers less than or equal to n, and $\ln(n)$ is the natural logarithm of n. In contrast, the occurrence of prime numbers in the Fibonacci sequence, denoted by F_n, does not follow such a well-defined distribution. While the Prime Number Theorem suggests a thinning out of primes as natural numbers grow, the distribution of primes among the exponentially increasing Fibonacci numbers is less predictable and far less dense. The attached image provides a visualization of this phenomenon using polar coordinates, with prime Fibonacci numbers highlighted.

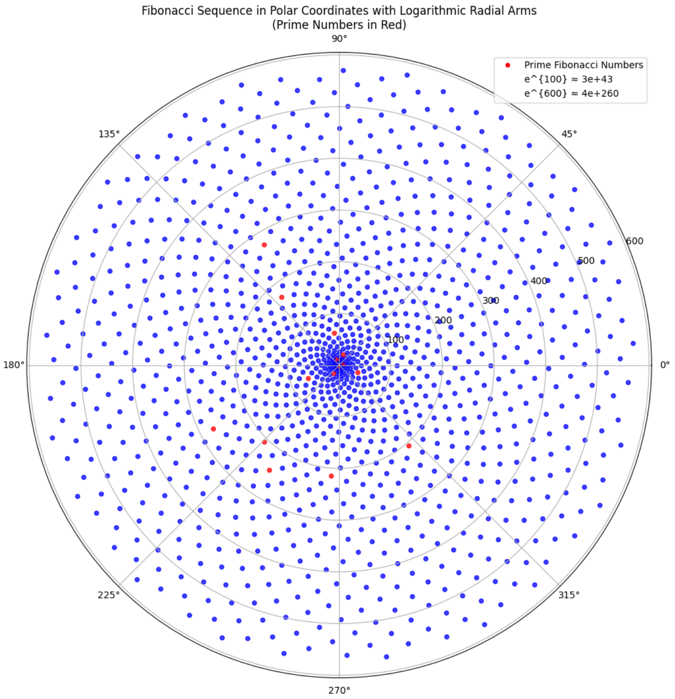


Figure 8.3: Prime Fibonacci Polar Golden Spiral

We note the following:

n	1	2	3	4	5	6	7	8	9
f_n	1	1	2	3	5	8	13	21	34
f_n^2	1	1	4	9	25	64	169	441	1156
$f_n^2 + f_{n+1}^2$	2	5	13	34	89	233	610	1597	4181

and that the product of two Fibonacci terms that bookend the square of another sums to one with alternating parity ,

$$\begin{aligned} f_3^2 - f_2 \cdot f_4 &= 2^2 - 1 \cdot 3 = +1 \\ f_4^2 - f_3 \cdot f_5 &= 3^2 - 2 \cdot 5 = -1 \\ f_5^2 - f_4 \cdot f_6 &= 5^2 - 3 \cdot 8 = +1 \\ &\vdots \end{aligned}$$

When n is even, the result of the expression is -1, and when n is odd (except for $n = 2$), the result is $+1$. That is, given $f_{n+1} = f_n + f_{n-1}$:

$$f_n^2 - f_{n-1} \cdot f_{n+1} = \begin{cases} +1 & \text{if } n \text{ is odd} \\ -1 & \text{if } n \text{ is even} \end{cases}$$

So

$$\begin{aligned} f_n^2 - f_{n-1} \cdot (f_n + f_{n-1}) &= f_n^2 - f_{n-1} \cdot f_n - f_{n-1}^2 \\ &= f_n^2 - 2f_{n-1}^2 - f_{n-1} \cdot f_{n-2}. \end{aligned}$$

As in $f_3^2 - 2f_2^2 - f_2 \cdot f_1 = 2^2 - 2(1^2) - 1(1) = 1$.
Also, for any Fibonacci number f_n, the relation

$$f_n^3 + f_{n+1}^3 - f_{n-1}^3 = f_{n+2}$$

holds true such that

$$\begin{aligned} 2^3 + 3^3 - 1^3 &= 8 + 27 - 1 = 34 \\ 3^3 + 5^3 - 2^3 &= 27 + 125 - 8 = 144 \\ 5^3 + 8^3 - 3^3 &= 125 + 512 - 27 = 610 \end{aligned}$$

8.2 Fibonacci Oblongs

Consecutive partial sums of the squares of Fibonacci numbers, $f_n = \{1, 1, 2, 3, 5, 8, 13, 21, ..\}$ give:

$$\begin{aligned} f_1^2 + f_2^2 &= f_2 \cdot f_3 & \text{where} \quad f_1^2 + f_2^2 &= 1^2 + 1^2 = 1 \cdot 2 \\ f_1^2 + f_2^2 + f_3^2 &= f_3 \cdot f_4 & \text{where} \quad f_1^2 + f_2^2 + f_3^2 &= 1^2 + 1^2 + 2^2 = 2 \cdot 3 \\ f_1^2 + f_2^2 + f_3^2 + f_4^2 &= f_4 \cdot f_5 & \text{where} \quad f_1^2 + f_2^2 + f_3^2 + f_4^2 &= 1^2 + 1^2 + 2^2 + 3^2 = 3 \cdot 5. \end{aligned}$$

Putting numbers to this we see, the following,

$$\begin{aligned} 1^2 + 1^2 &= 1 \cdot 2 \\ 1^2 + 1^2 + 2^2 &= 2 \cdot 3 \\ 1^2 + 1^2 + 2^2 + 3^2 &= 3 \cdot 5 \\ 1^2 + 1^2 + 2^2 + 3^2 + 5^2 &= 5 \cdot 8 \\ 1^2 + 1^2 + 2^2 + 3^2 + 5^2 + 8^2 &= 8 \cdot 13 \end{aligned}$$

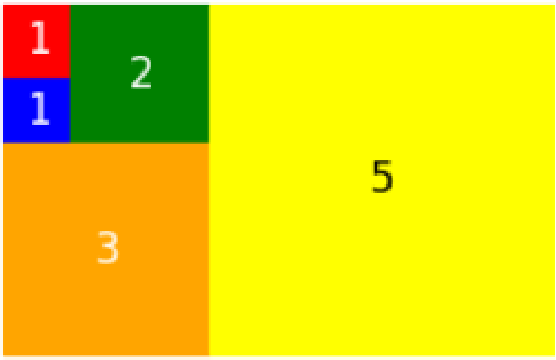


Figure 8.4: Lesser drawn Fibonacci Rectangle.

This is implemented by way of this code[2] snippet in which the `matplotlib.patches` module (aliased as `patches`) plays a central role.

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
x, y = 0, 0
fig, ax = plt.subplots()
ax.set_aspect('equal', 'box')
blue = patches.Rectangle((x, y), 1, 1, edgecolor='blue', facecolor='blue')
ax.add_patch(blue)
ax.text(x + 0.5, y + 0.5, '1', ha='center', va='center', fontsize=12, color='white')
```

1. **Purpose of** `matplotlib.patches`:

[2] FibonacciRectangles.ipynb

- The code uses the `Rectangle` class from the `patches` module to create a blue square.
- `blue = patches.Rectangle((x, y), 1, 1, edgecolor='blue', facecolor='blue')` creates a blue square (or rectangle with equal height and width):

    (a) (x,y) denotes the bottom-left corner of the rectangle.
    (b) The next two arguments `1, 1` define the width and height of the rectangle, respectively. Since they are both `1`, this creates a square.
    (c) `edgecolor='blue'` sets the color of the square's border.
    (d) `facecolor='blue'` sets the fill color of the square.

2. **Integration with** `matplotlib`:

    - After creating the `Rectangle` object, it's not immediately displayed on the figure. To make it visible, we need to add it to the axes (`ax`) using the `add_patch()` method.
    - `ax.add_patch(blue)` integrates the `Rectangle` object with the main figure, enabling it to be displayed when `plt.show()` (not provided in the code snippet) is called.

3. **Labeling the Rectangle**:

    - The code also places a text label `'1'` at the center of the square using the `ax.text()` method. The coordinates `x + 0.5, y + 0.5` ensure the text is centered since the square's width and height are `1`.

The Fibonacci numbers $1,1,2,3,5...$ have rectangles thus $\mathscr{F} = 2,6,15,40,...$that are extended according to the following:

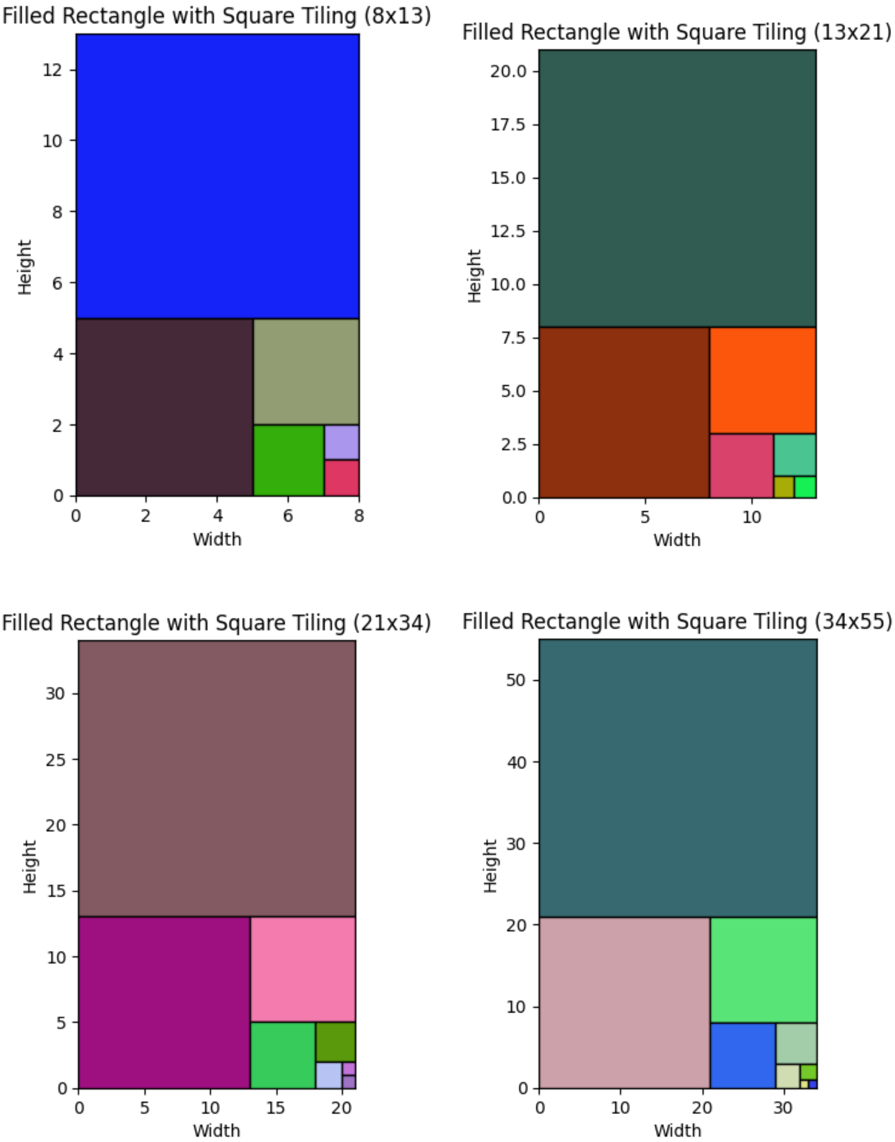


Figure 8.5: Fibonacci Rectangles

The Fibonacci rectangles property relies on the specific starting values of the Fibonacci sequence ($F_0 = 0, F_1 = 1$) and the recursive relation, and it does not generally apply to other sequences[3] with different starting values, even if they use the same recursive relation.

[3] We note that the Lucas numbers $L_n : 1, 3, 4, 7, 11, 18, 29, 47, \ldots$, the property just described for Fibonacci numbers does not hold. The Lucas numbers, like the Fibonacci numbers, are defined recursively as $L_n = L_{n-1} + L_{n-2}$ with initial conditions $L_0 = 2$ and $L_1 = 1$, but the sum of the squares of consecutive Lucas numbers is not equal to the product of two consecutive Lucas numbers. For example:

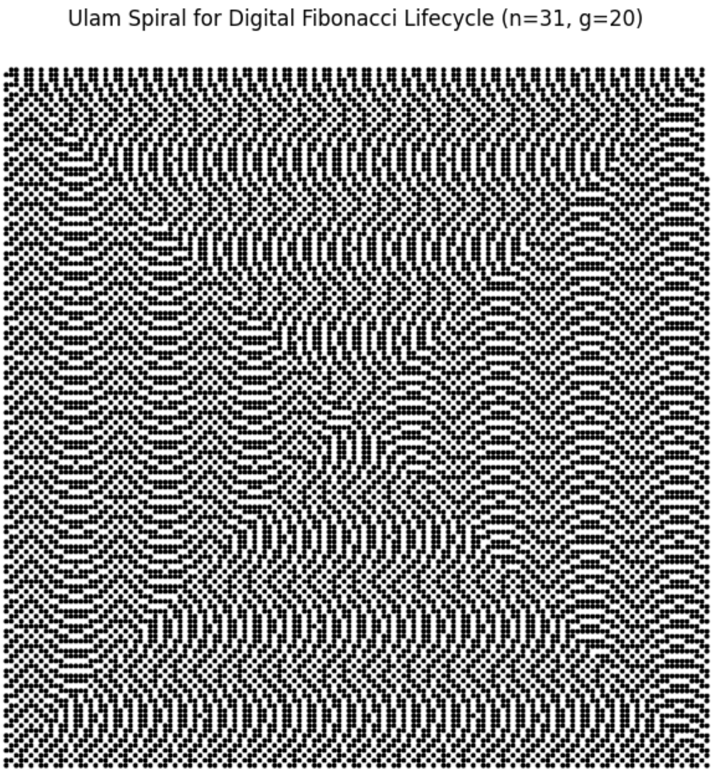
$$1^2 + 3^2 + 4^2 + 7^2 = 75 \neq 11 \times 7$$

8.2.1 Aperiodic Tilings from Digital Fibonacci Sequences and Ulam Spirals

We will now generate aperiodic tilings using a modified Digital Fibonacci sequence wrapped in an Ulam spiral. The Digital Fibonacci sequence is constructed based on three algorithmic conditions applied to binary states representing bunnies (0) and rabbits (1):

1. A bunny (0) matures into a rabbit (1).
2. A rabbit (1) gives birth to a bunny (0) and matures further (becomes 10).
3. After $g = 4$ generations, a rabbit (1) expires and is represented by a space (" "), indicating the end of its lifecycle.

In this context, g represents the generational limit for the survival of a rabbit. For $g = 4$, only great grandparents are present with their bunnies, while $g = 3$ allows only grandparents. The case where $g = \infty$ implies a Fibonacci persistent survivor rabbit with no expiration.

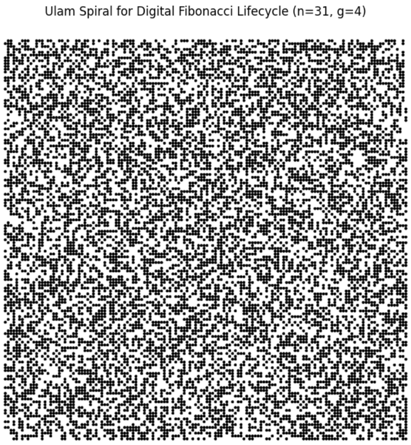


(a) Digital Fibonacci Ulam Spiral Tiling, no death

```
truncated_sequence = ''.join(random.choices(['1', '0'], k=20000))
```

The Ulam spiral is constructed by wrapping the concatenated binary series of the Digital Fibonacci sequence in a clockwise outward path. The sequence is generated in reverse, beginning with the most recent generation and tracing back to the origin.

We may contrast this with an Ulam tiling based on $\{0,1\}$ drawn at random which looks like the tiling opposite. The code, AperiodicUlamofLifeCycle10digitalFibonacci.ipynb generates this tiling. The full concatenated digital series `reversed_full_digital_sequence` are built by reversing the sequence after generation and we limit the scope of our analysis to the first 20000 digits which allows for a sufficiently suggestive display.



Implementation methods of note are:

- **Truncation**: The sequence was truncated to the first $k = 20000$ digits to maintain a manageable spiral size for plotting.
- **Sorting**: The generation and plotting of the spiral required a systematic approach to sorting and arranging the binary digits into their spiral positions.
- **Visualization**: Matplotlib was used for visualization, with adjustments made to the plotting scale to ensure clear representation of the tiling pattern.

```
full_digital_sequence = ''.join(digital_sequence_lifecycle_generations)
reversed_full_digital_sequence = full_digital_sequence[::-1]
truncated_sequence = reversed_full_digital_sequence[:20000]
```

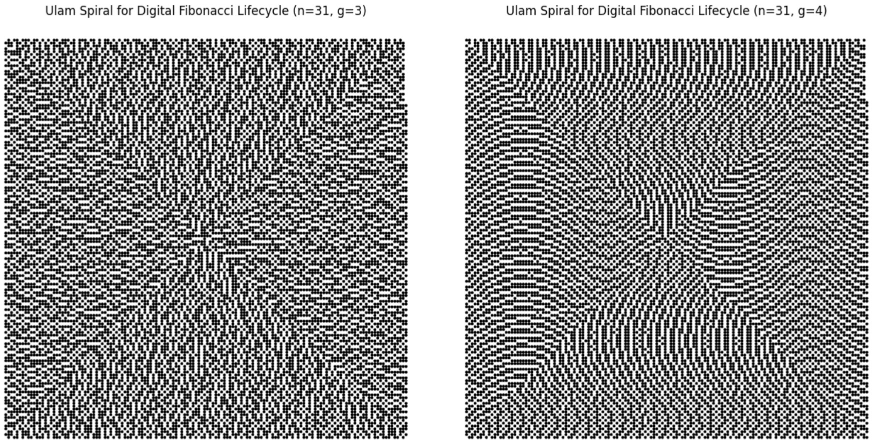


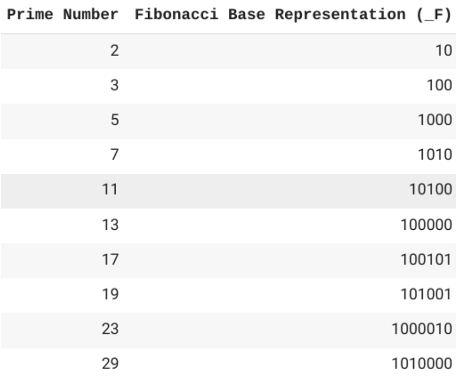
Figure 8.7: Digital Fibonacci Ulam Spiral Tiling, g=3,4

Theorem 8.2.1 — Zeckendorf's Theorem. Every positive integer can be represented uniquely as the sum of one or more distinct nonconsecutive Fibonacci numbers. More formally, given any positive integer N, there exists a unique set of Fibonacci numbers $\{F_i\}$ such that

$$N = \sum_i F_i$$

where no two F_i are consecutive in the usual Fibonacci sequence.

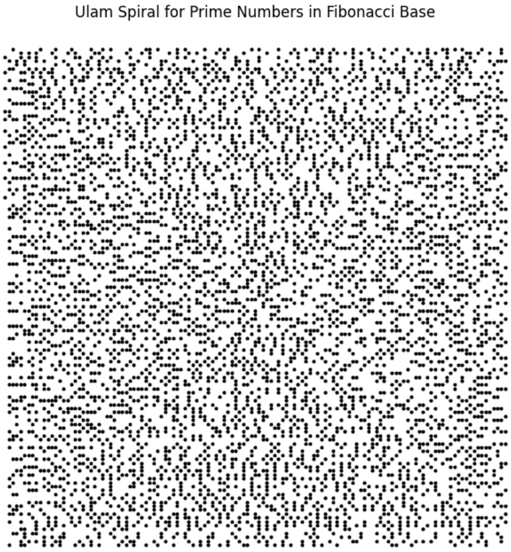
For example, the Zeckendorf representation of the decimal number 100 in terms of Fibonacci numbers $(1,2,3,5,8,13,21,\ldots)$ would be $100 = 89+8+3$, which corresponds to the binary representation of $100_{10} = 101001000_F$, where the subscript F indicates the Fibonacci base.



Prime Number	Fibonacci Base Representation (_F)
2	10
3	100
5	1000
7	1010
11	10100
13	100000
17	100101
19	101001
23	1000010
29	1010000

Figure 8.8: Zeckendorf representation of Primes.

The primes represented as an Ulam spiral in Zeckendorf representation appear to chime rather than rhyme:



Ulam Spiral for Prime Numbers in Fibonacci Base

(a) Ulam Spiral of concatenated Zeckendorf representations of first 5000 Primes

The function `generate_zeckendorf_representation` from UlamPrimeFibonacciBaseEfficient.ipynb implements the algorithm to find the Zeckendorf representation of a given integer and operates as follows:

1. The `fib_sequence` array is prepared with Fibonacci numbers in ascending order.
2. Starting from the end of this array, the function iterates backwards, checking if the Fibonacci number can be part of the representation for the given `number`.
3. If the Fibonacci number is less than or equal to the `number`, it is included in the representation, and we move back two indices to ensure non-consecutiveness.
4. This process repeats, decrementing the `number` accordingly, until all possible Fibonacci numbers have been checked.
5. The result is a binary string where '1's represent the inclusion of a Fibonacci number in the sum, and '0's represent its absence.

Below is the Python code for the function:

```
def generate_zeckendorf_representation(number, fib_sequence):
    representation = ['0'] * len(fib_sequence)  # Start with a list of zeros
    index = len(fib_sequence) - 1  # Begin from the highest Fibonacci number
    while number > 0 and index >= 0:
        if fib_sequence[index] <= number:
            number -= fib_sequence[index]
            representation[index] = '1'
            index -= 2
        else:
            index -= 1
    # Since the representation is built backwards, reverse and join it to form a string
    return ''.join(representation[::-1]).lstrip('0') or '0'
```

Fibonacci as a Divisibility Sequence

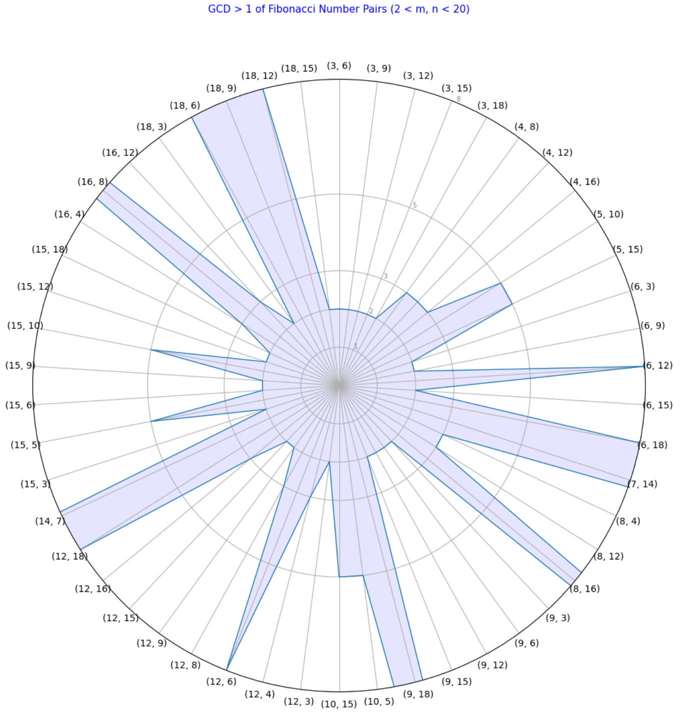
A sequence of integers $\{a_n\}$, is termed a divisibility sequence if for any positive integers m and n, the condition $n|m$ (i.e., n divides m) implies that $a_n|a_m$ (i.e., a_n divides a_m). The Fibonacci sequence $\{f_n\}$ is the archetype with its greatest common divisor (GCD) given by the identity

$$\gcd(f_n, f_m) = f_{\gcd(m,n)},$$

for f_n and f_m and $f_{\gcd(m,n)}$, all Fibonacci numbers the later of which corresponds to the GCD of the indices m and n. Every nth Fibonacci number divides evenly into every nth number after it in the sequence. If n divides m (meaning m is a multiple of n), then gcd(n, m) = n. So, F(gcd(n, m)) = F(n), which is essentially saying that the nth Fibonacci number divides the mth Fibonacci number (where m is a multiple of n). For instance, 3 is the fourth Fibonacci number, and it divides into the eighth term (21) but not the seventh (13). We have thus for $n = 8$ and $m = 12$, $f_8 = 21$ and $f_{12} = 144$ and the GCD of n and m is 4, and $f_{\gcd(8,12)} = f_4 = 3$, which is also the GCD of F_8 and F_{12}. Similarly, for $n = 21$ and $m = 34$, $f_{21} = 10946$ and $f_{34} = 5702887$ and the GCD of n and m is 1, and $f_{\gcd(21,34)} = f_1 = 1$, which matches the GCD of F_{21} and F_{34}. Other examples of divisibility sequences include:

- The powers of a prime number p, where p^n clearly divides p^m for any integers $n \leq m$.
- The sequence of factorials $\{n!\}$, since $n!|m!$ whenever $n \leq m$.
- The sequence of Lucas numbers, which follow the recurrence relation $L_n = L_{n-1} + L_{n-2}$ with initial conditions $L_0 = 2$ and $L_1 = 1$, is also a divisibility sequence.

Divisibility sequences often exhibit a strong connection to the structure of number fields and algebraic integers and may also possess homomorphic mappings to other mathematical structures and can have rich properties in the context of modular arithmetic and Diophantine equations. The study of divisibility sequences intersects with other mathematical areas such as elliptic curves, group theory, and the theory of algebraic numbers, which often do not apply to non-divisibility sequences. The spider plot below has been generated for the first 72 combinations of (m,n) such that $j > (m,n) > i$ and for which the pairs' GCD is greater than 1.



(a) GCD spider of First 20 Fibonacci pairs

Each axis of the spider plot represents a unique pair (m,n), distributed evenly by angle around the circle. The radial distance from the center of the plot to the point on the axis represents the GCD value, (greater than 1 in this selection), of the Fibonacci numbers at those indices. The plot uses a user-defined function that generates pairs within a specified range and filters them based on the GCD condition. The code, spiderFibGCD.ipynb includes a function that creates a spider plot to illustrate the GCDs of pairs of Fibonacci numbers:

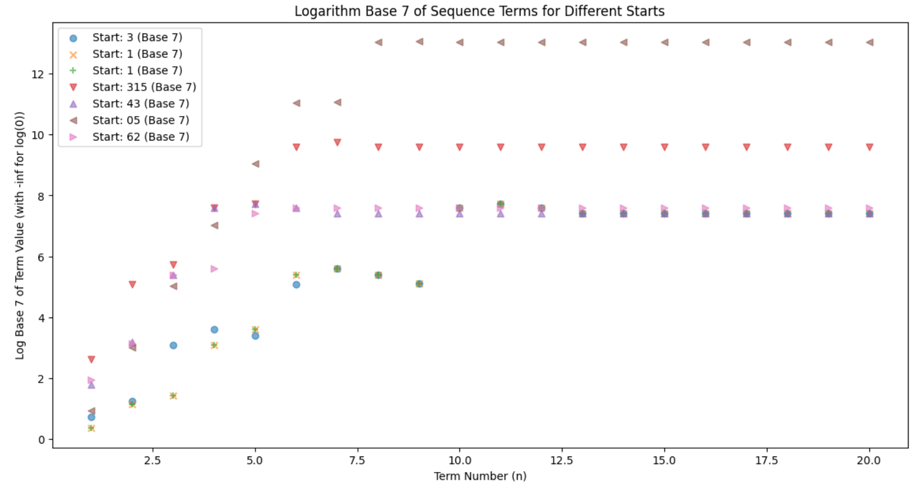
```
def create_spider_plot(pairs, title):
    num_vars = len(pairs)
    angles = [n / float(num_vars) * 2 * pi for n in range(num_vars)]
    angles += angles[:1]  # to close the plot (make it circular)
    fig, ax = plt.subplots(figsize=(12, 12), subplot_kw=dict(polar=True))
    ax.set_theta_offset(pi / 2)
    ax.set_theta_direction(-1)
    labels = [f'({m}, {n})' for (m, n) in pairs]
    plt.xticks(angles[:-1], labels)
    plt.yticks([1, 2, 3, 5, 8], ["1", "2", "3", "5", "8"], color="grey", size=7)
    plt.ylim(0, 8)
    values = [gcd(fibonacci(m), fibonacci(n)) for (m, n) in pairs]
    values += values[:1]
    ax.plot(angles, values, linewidth=1, linestyle='solid')
    ax.fill(angles, values, 'b', alpha=0.1)
    plt.title(title, size=11, color='blue', y=1.1)
    plt.show()
```

- Calculating angles to evenly distribute each Fibonacci pair around a circle using the formula $\frac{n}{\text{num\_vars}} \times 2\pi$, where n is the index of the pair and num_vars is the total number of pairs.
- The plot is circular,as achieved by appending the first angle to the end of the list of angles.
- A polar subplot is initialized using Matplotlib's `plt.subplots` with a specified figure size.
- The plot is oriented to start from the top by setting a theta offset of $\frac{\pi}{2}$ and is made to proceed in a clockwise direction.
- Labels for each axis represent the Fibonacci pairs and are set using Matplotlib's `plt.xticks`.
- Custom y-ticks are set to represent possible GCD values, with the y-axis limit accommodating the maximum expected GCD value.
- The GCD values for the Fibonacci pairs are plotted as points on the spider plot, connected by lines, and the area under the curve is filled with color.

However, not all related sequences that follow a recurrence relation are divisibility sequences. For instance, the Pell sequence defined by $S_n = 2S_{n-1} + S_{n-2}$ with $S_0 = 0$ and $S_1 = 1$ does not necessarily exhibit the same divisibility property as the Fibonacci sequence. It requires specific structural features of a sequence for this property to hold.

8.3 Self-Enumerating Attractor Sequence

In the study of complex dynamical systems, the behavior of recursive sequences can be particularly illuminating. Such sequences often exhibit characteristics akin to attractors—a set of states towards which a system converges. The scatter plot provided visualizes a sequence known for its dynamical properties illustrtaing two distinct behaviors reminiscent of fixed-point attractors and limit cycles.



(a) 20 instances of Count Concatenate Series

Fixed-point attractors are single values that the sequence settles into, regardless of the initial state, as evidenced by the clustering of points at specific heights. In contrast, limit cycles which represent a periodic oscillation of the sequence through a set of states.

The sequence S emerges from a recursive numerical process that is reflective of complex dynamical systems. Beginning with an initial term s_0, the sequence progresses through discrete iterations. Each subsequent term s_n, for n ranging from 1 to 100, is constructed by tallying the digits in s_{n-1} and collating these counts into a new term that omits any count of zero. This process guarantees a digit order descending from 9 to 0.

For instance, starting with $s_1 = "10"$, the sequence evolves as follows:

- From "10", we deduce $s_2 = "1011"$, indicating 1 occurrence of 0 and 1 occurrence of 1.
- From "1011", s_3 is deduced to be "1031", reflecting 1 occurrence of 0 and 3 occurrences of 1.
- Consequently, given "1031", s_4 is deduced to be "102113", signifying:
  - "10" from the single occurrence of 0,
  - "11" from the single occurrence of 1,
  - "13" from the single occurrence of 3.

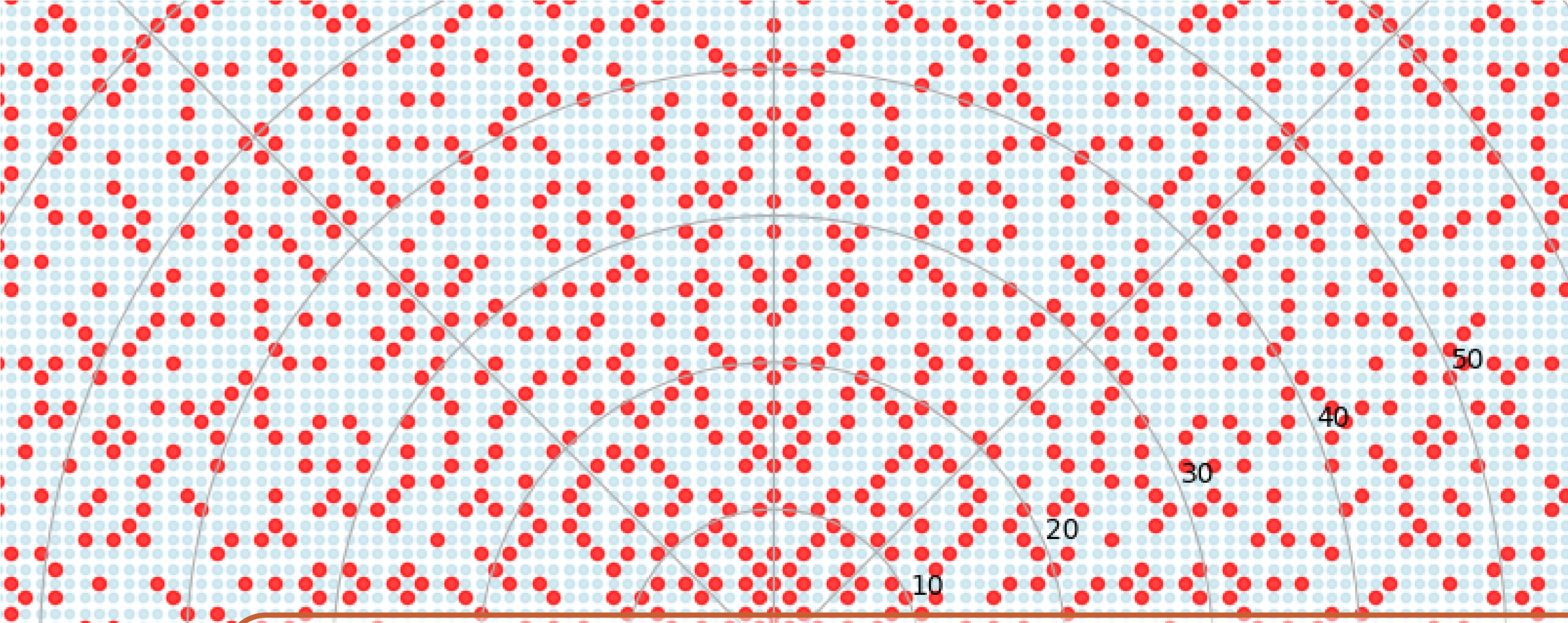
This algorithmically generated sequence thus exhibits the underlying structure and behavior akin to attractors found in dynamical systems.

The code p-adConcatenate.ipynb and the code logPlotLimitConcatenation.ipynb

8.4 head throwing

$A_n(3) = [1,2,4,8,15,29,56,108,208] forn = 0to8; forouroriginaltablearray([[1,0,0,0,0,0,0,0,0,0],[1,0,0,0,0,0,($

where i have added an * for the hockey stick calculation such that $C_7^5 = 12* = 2* + 3* + 4* + 3*$



9. Perspective Compositions

9.1 Lemoine’s conjecture

Lemoine’s Conjecture posits that every odd integer greater than 5 can be expressed as the sum of a prime number and twice another prime.

To count how many ways an odd number n can be expressed in the form $n = p + 2q$, where both p and q are prime numbers, we can follow these steps:

1. Iterate over all prime numbers p up to n.
2. For each p, check if $\frac{n-p}{2}$ is also a prime.
3. If so, increment a counter.
4. Return the final count.

Here is Python code to calculate C_n for any odd n and display it against n:

```
import sympy
def ways_to_express_n(n):
    count = 0
    for p in sympy.primerange(2, n):
        if sympy.isprime((n - p) // 2):
            count += 1
    return count
def calculate_cn_for_odd_n(limit):
    return {n: ways_to_express_n(n) for n in range(7, limit + 1, 2)}
```

The code delivers the following scatter plot:

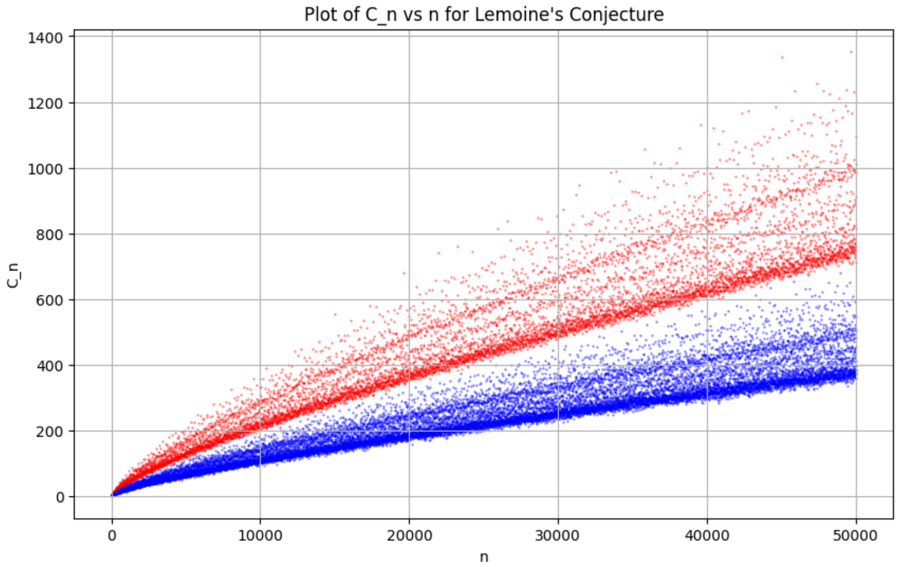


Figure 9.1: Lemoine's Prime Conjecture for n=50000 with red $n \equiv 3 \pmod 6$

The distinction in the scatter plot between red points satisfying $n \equiv 3 \pmod 6$ and the other blue points is rooted in the properties and distribution of prime numbers. Understanding the significance of this distinction requires a brief foray back into the modular arithmetic of primes. Recalling that all prime numbers greater than 3 can be expressed in one of two forms: either 1 more than a multiple of 6 or 1 less, which is expressed as primes p having a residue of either 1 or 5 modulo 6. This is because:

- Numbers that are 0 (mod 6) are divisible by 6.
- Numbers that are 2 (mod 6) or 4 (mod 6) are even.
- Numbers that are 3 (mod 6) are divisible by 3.

As a consequence, the only possible residues for primes greater than 3 are 1 and 5 when taken modulo 6.

Given this understanding, the split in the scatter plot becomes a tool for observing any potential patterns or anomalies in the distribution of numbers based on their modular congruence. By highlighting numbers that are $n \equiv 3 \pmod 6$ in red, we can discern if this subset behaves differently or similarly to the rest in the context of Lemoine's Conjecture or any other number theoretic conjecture under consideration.

9.2 Goldberg Variations

The Goldberg Conjecture, proposed by Michael Goldberg, states that every even integer greater than 2 can be expressed as the sum of two prime numbers, denoted as $m = p+q$. It is an intriguing conjecture that has captured the attention of mathematicians for many years. At present, it remains an open question whether there exists a counterexample that disproves the conjecture.

One approach to probe the Goldberg Conjecture is to analyze the frequency at which individual primes contribute to the sum of even numbers, m. By examining this frequency distribution, we can gain insights into the behavior of prime numbers in relation to the conjecture. However, it is essential to impose certain conditions to ensure meaningful observations.

Let us consider several specific values for m: 100, 1000, 10000, 100000, and a million. As we investigate these cases, we must take into account the dominance of the prime number 3 in the frequency chart if no condition is placed on the $p+q$ sum.

To filter out contributions from lower numbers as m grows larger, we can limit our consideration to prime numbers p and q such that $p+q = m$ and the absolute difference between p and q, denoted as $|p-q|$, is minimized. This additional condition ensures that we focus on pairs of primes that are closest to each other, rather than arbitrary combinations.

For example, let us examine the case of $m = 16$. The possible pairs of primes satisfying the condition $p+q = 16$ are $(11,5)$ and $(13,3)$. However, we will only consider the pair $(11,5)$ since it has the minimum absolute difference, $|11-5| = 6$, which is smaller than $|13-3| = 10$.

By applying this approach to each value of m, we can obtain a refined frequency chart that highlights the contributions of prime numbers with the minimum absolute difference. This analysis allows us to explore the distribution of prime pairs that fulfill the Goldberg Conjecture for larger even numbers, shedding light on the potential patterns and properties of such pairs.

While the Goldberg Conjecture remains unproven, investigating the frequency of prime contributions and considering the conditions mentioned above offers a promising avenue for exploring the conjecture and deepening our understanding of prime numbers and their relationships.

Fermat's Determination and Lagrange's Theorems

Fermat's determination, also known as Fermat's theorem on sums of two squares, states that every prime number of the form $4k+1$ can be expressed as the sum of two squares. By investigating the properties of quadratic residues, Fermat demonstrated that this is always possible for primes of the given form.

Joseph-Louis Lagrange, an Italian-French mathematician, studied the representation of non-negative integers as sums of perfect squares. Lagrange's three-square theorem states that every non-negative integer can be expressed as the sum of three perfect squares. In other words, for any positive integer n, we can find integers a, b, and c such that $n = a^2+b^2+c^2$. Similarly, Lagrange's four-square theorem states that every non-negative integer can be expressed as the sum of four perfect squares. For any positive integer n, we can find integers a, b, c, and d such that $n = a^2+b^2+c^2+d^2$.

It is interesting to note the connection between Fermat's theorem and Lagrange's theorems. Fermat's theorem can be seen as a special case of Lagrange's theorems, where we restrict our attention to prime numbers of the form $4k+1$. In this case, we are expressing a prime as the sum of two squares. However, Lagrange's theorems encompass all non-negative integers, allowing us to express them as the sum of three or four squares.

Primes and sum of squares

"Fermat" Primes of the form $4k+1$: $5,13,17,29,37,41,....$ be expressed as the sum of two squares, a^2+b^2, where a and b are integers. Non-square "Gauss" Primes of the form $4k+3$:

$7, 11, 19, 23, 29, 31, ..$ Although these primes cannot be expressed as the sum of two squares, they can be written as the sum of three squares, $13 = 2^2 + 2^2 + 1^2$. Additionally, some of such **non-square primes** can be expressed as the sum of two cubes, four squares, or other combinations of powers.

9.3 Compound Numbers written as sum of two Squares

The diagonal matrix below shows the first 12 compound numbers formed as sum of two squares

	1	9	25	49
1	2	10	26	50
9			34	58
25		34	50	74
49				98

Figure (is a scatter-plot of square numbers up to $n = 30$, focusing on the distinct patterns formed by odd and even sums. The plot contrasts $\sqrt{s_1}$ and $\sqrt{s_2}$, with each point labeled by the sum $c = s_1 + s_2$.

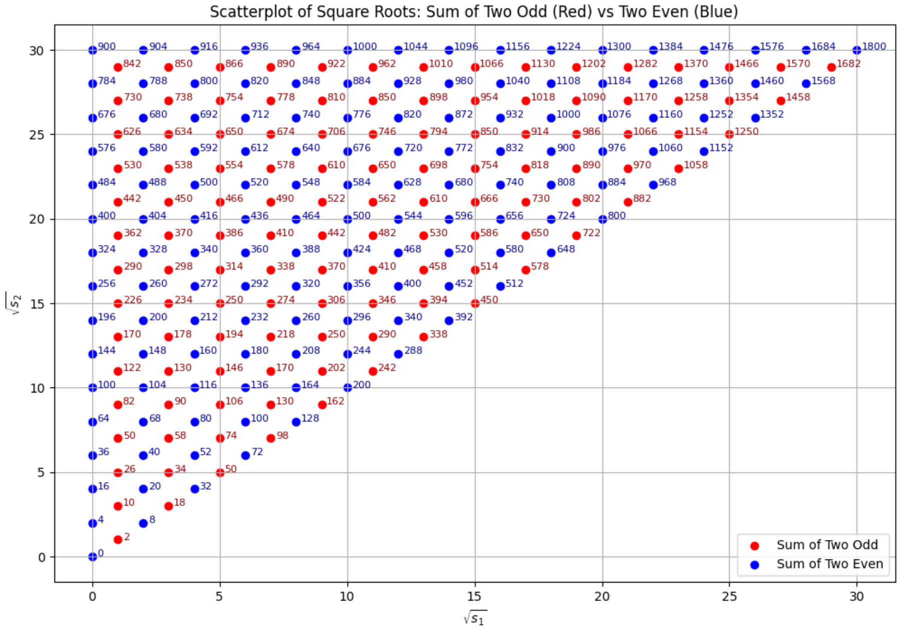


Figure 9.2: Sum of Squares scatterplot

The Square numbers were generated using Python's list comprehension for efficiency. Odd and even squares were separated using conditional statements so as to be distinguished in the visualization. Pairs of squares were combined using nested loops, offering control over the pair selection and emphasizing pairs where $s_2 \geq s_1$. This approach was chosen for its simplicity and effectiveness over using permutation packages. The following matrix represents odd square numbers up to $n = 11$ and their sums:

Consider in light of this the following product of sums of squares,

$$\begin{aligned}
&(1^2+0^2)(2^2+1^2) = 2^2+1^2\\
&(2^2+1^2)(3^2+2^2) = 4^2+7^2\\
&(3^2+2^2)(1^2+4^2) = 5^2+14^2\\
&(4^2+3^2)(5^2+2^2) = (4\cdot 5-3\cdot 2)^2+(4\cdot 2+3\cdot 5)^2 = 14^2+23^2\\
&(5^2+1^2)(2^2+5^2) = (5\cdot 2-1\cdot 5)^2+(5\cdot 5+1\cdot 2)^2 = 5^2+27^2
\end{aligned}$$

which suggests the following identity could be true:

$$(x^2+y^2)(z^2+w^2) = (xz-yw)^2+(xw+yz)^2. \tag{9.1}$$

Let us confirm that indeed it is:

$$\text{Left-Hand Side (LHS)} = (x^2+y^2)(z^2+w^2) = x^2z^2+x^2w^2+y^2z^2+y^2w^2.$$

$$\begin{aligned}
\text{Right-Hand Side (RHS)} &= (xz-yw)^2+(xw+yz)^2\\
&= x^2z^2-2xyzw+y^2w^2+x^2w^2+2xyzw+y^2z^2\\
&= x^2z^2+x^2w^2+y^2z^2+y^2w^2.
\end{aligned}$$

Consider the following restrictions ([1]) on the prime factorisation of the divisors by which a number, n can be constructed so as to be written the sum of squares:

- **Fermat's Theorem on the Sum of Two Squares:** A prime number p of the form $4n+1$ can be expressed as the sum of two squares.
- **Multiplicity of Gauss Type of primes for Sum of Two Squares to hold:** A positive integer n can be expressed as a sum of two squares if and only if every prime factor of the form $4n+3$ in its prime factorization occurs with an even exponent.

Consider, the rectangular number 5490 the Fermat prime $f = 61 = 6^2+5^2$ and Gauss compliant rectangle number $n = 90 = 2\times 3^2\times 5 = 3^2+9^2$. Applying the identity (9.1) we have :

$$\begin{aligned}
5490 &= 61\times 90\\
&= (6^2+5^2)(3^2+9^2)\\
&= (6\cdot 3-5\cdot 9)^2+(6\cdot 9+5\cdot 3)^2\\
&= (18-45)^2+(54+15)^2 = 27^2+69^2.
\end{aligned}$$

We can play a similar game with triangle numbers. The code provided generates a comprehensive visualization of the sums of all possible combinations of triangle numbers up to a specified limit, n. The triangle numbers are generated by the function $T_n = \frac{n(n+1)}{2}$, which is a polynomial representation of the dot patterns forming equilateral triangles. The triangle numbers are foundational in number theory and often appear in various conjectures and theorems, such as those postulated by Goldberg.

Let us determine

Goldberg's conjectures relate to the partitioning of numbers into sums of triangular numbers, akin to the problem of representing numbers as sums of squares, which has been a subject of extensive study in number theory. The code systematically explores all permutations of sums of three triangle numbers, echoing the exhaustive nature of Goldberg's investigations into number partitions.

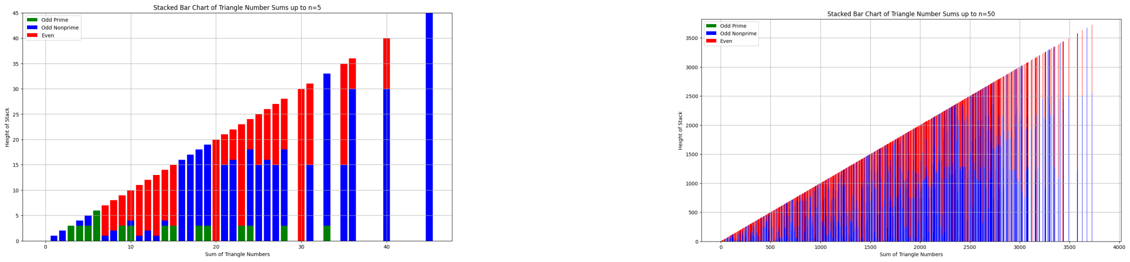


Figure 9.3: The integers written as sum of three triangles.

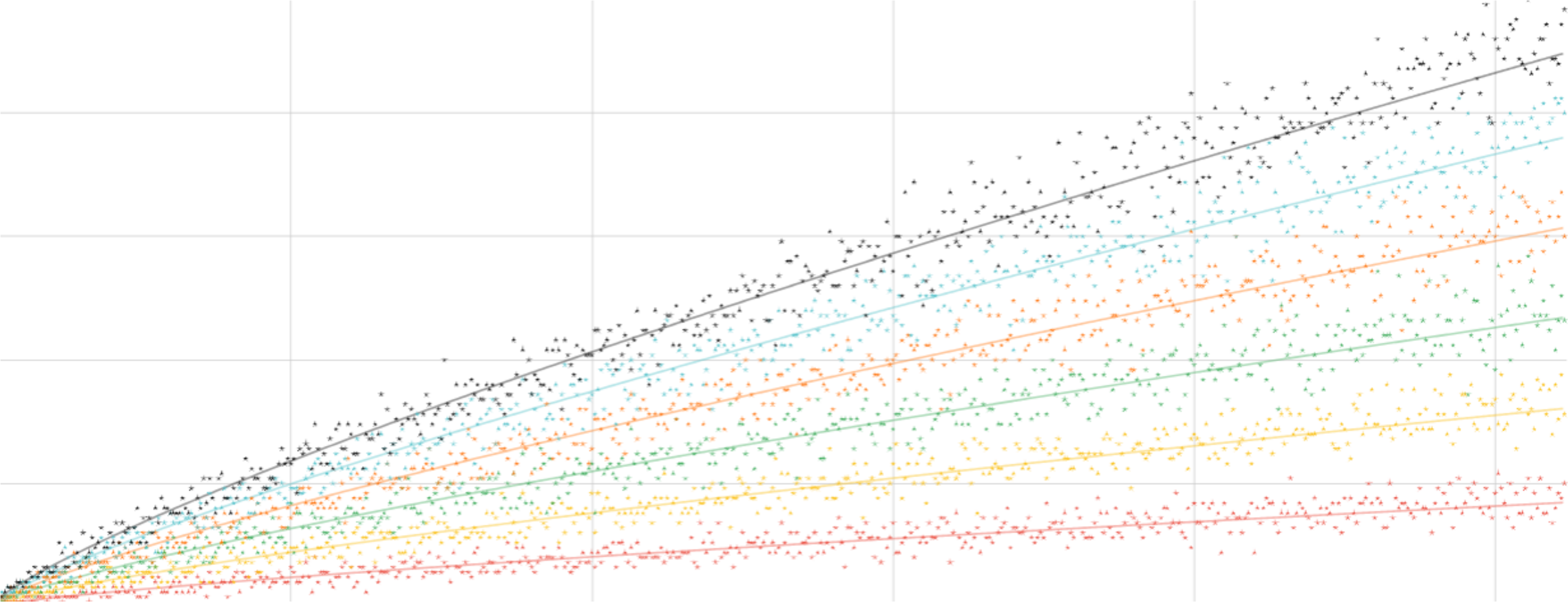
```
triangle_numbers = [triangle_number(n) for n in range(0, n_max + 1)]
# Split the triangle numbers into three sets: odd primes, odd non-primes, and evens (i
t_odd_prime = [t for t in triangle_numbers if is_odd_prime(t)]
t_odd_nonprime = [t for t in triangle_numbers if t % 2 != 0 and not is_odd_prime(t)]
t_even = [0] + [t for t in triangle_numbers if t % 2 == 0]

# Generating unique combinations and their sums
unique_combinations = set()
for t1 in triangle_numbers:
    for t2 in triangle_numbers:
        for t3 in triangle_numbers:
            # Create a sorted tuple of the combination to ensure uniqueness
            combination = tuple(sorted((t1, t2, t3)))
            unique_combinations.add(combination)

# Extract unique sums and their corresponding combinations
unique_sums = {}
for combination in unique_combinations:
    sum_comb = sum(combination)
    if sum_comb not in unique_sums:
        unique_sums[sum_comb] = combination
```

The legendary mathematician Carl Friedrich Gauss laid the foundation for the representation of numbers as sums of polygonal numbers, a concept in the provided code.

9.3.1 Goldberg Conjecture

Here is a list of the first 10 non-square primes that are better expressed as the sum of two cubes and four squares:

- 7 (Sum of two cubes: $1^3 + 2^3$; Sum of four squares: $1^2 + 2^2 + 1^2 + 1^2$)
- 19 (Sum of two cubes: $2^3 + 3^3$; Sum of four squares: $2^2 + 3^2 + 1^2 + 1^2$)
- 31 (Sum of two cubes: $1^3 + 3^3$; Sum of four squares: $3^2 + 3^2 + 1^2 + 0^2$)
- 43 (Sum of two cubes: $3^3 + 4^3$; Sum of four squares: $3^2 + 4^2 + 2^2 + 0^2$)
- 67 (Sum of two cubes: $3^3 + 4^3$; Sum of four squares: $4^2 + 5^2 + 2^2 + 0^2$)
- 79 (Sum of two cubes: $2^3 + 5^3$; Sum of four squares: $4^2 + 5^2 + 2^2 + 2^2$)
- 97 (Sum of two cubes: $2^3 + 3^3$; Sum of four squares: $5^2 + 4^2 + 2^2 + 2^2$)
- 103 (Sum of two cubes: $1^3 + 2^3$; Sum of four squares: $5^2 + 4^2 + 2^2 + 2^2$)
- 109 (Sum of two cubes: $1^3 + 3^3$; Sum of four squares: $6^2 + 2^2 + 1^2 + 0^2$)
- 127 (Sum of two cubes: $1^3 + 2^3$; Sum of four squares: $7^2 + 2^2 + 1^2 + 1^2$)



10. Triangulation

Just now, when I was on the point of coming into my room,
I stopped short because I felt in my hand a cold object which attracted my attention by means of a sort of personality.
I opened my hand and looked: I was simply holding the doorknob.
— *Jean-Paul Sartre, Nausea*[17]

10.1 Prime Stratification with Polygonal Numbers

10.2 Geometric Structures under Klein's Erlangen Program

Felix Klein's Erlangen Program, introduced in 1872, is a framework for the classification of geometries based on their invariance under groups of transformations. This approach provides a unified perspective for understanding different geometric structures by examining the properties that remain unchanged under specific sets of transformations.

Affine Geometry

Affine geometry is concerned with the study of figures invariant under affine transformations, which include translations, linear transformations, and their combinations. These transformations preserve points, straight lines, parallelism, and ratios of segments on parallel lines, but not distances or angles. The affine group consists of all nonsingular linear transformations combined with translations.

Projective Geometry

Projective geometry focuses on properties that are invariant under projective transformations. Such transformations preserve collinearity and the concept of a cross ratio, a key invariant in projective geometry[1]. Projective transformations include all linear transformations of the projective space, considered as linear transformations of a vector space of one higher dimension, modulo scalars.

Metric Geometry

Metric geometry examines the properties invariant under isometries, the transformations that preserve distances between points. This includes translations, rotations, reflections, and their combinations in Euclidean space. Metric geometry studies angles, distances, and the congruence of figures, focusing on the properties preserved under the isometry group.

We will humbly explore geometric spaces—affine, projective, and metric—revealing the different structures each imposes on a generic space gaining a little insight into the what Bohm considers to be the inherent order and transformations within geometric constructs such as polygons, spirals, and structures with varying angles.

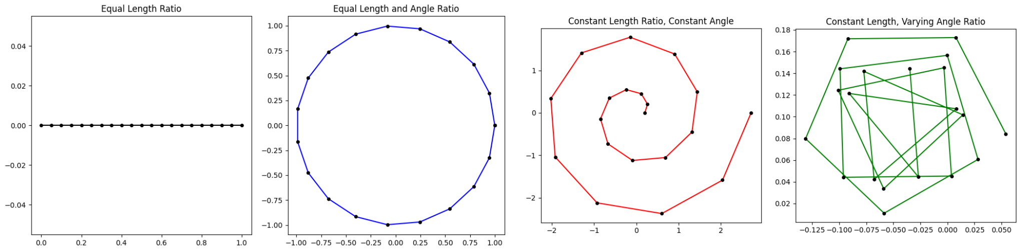


Figure 10.1: Polygon Numbers.

- **Regular Polygon Approximated as a Circle:** The significance lies in the equal ratio of subsequent line segments ($\frac{\text{length } a}{\text{length } b} = 1$) and equal angles between them. This construct illustrates affine geometry's emphasis on maintaining ratios and parallelism.
- **Spiral:** Characterized by a varying length of segments where the ratio between the lengths of successive segments remains constant ($\frac{\text{length } a}{\text{length } b} = k$), and a constant angle between segments. The spiral embodies projective geometry's principles by showcasing how perspective transformations can influence the perception of space and distance.
- **Structure with Constant Length and Varying Angles (Angular Spiral):** Maintains a constant length for each segment while varying the angle between successive segments in a constant ratio. This structure highlights the interplay between metric and projective geometries, emphasizing how angular relationships can transform space while preserving certain geometric properties.

Through these constructs, the distinctive features of affine, projective, and metric geometries are partially evident. We may stage the scalings according to Fibonacci according to the code,BohmiamOrderFibonacci.ipynb does this producing the following set of figures. Each construct embodies the progressive nature of the sequence, translating numerical ratios into spatial dimensions. The constructs are rendered in a Cartesian coordinate system, utilizing a polar transformation where necessary.

Line Segmented by Fibonacci Ratios

The line is constructed by determining segment lengths as the reciprocals of the Fibonacci numbers. These lengths are then cumulatively summed to position the segments along a single axis. The visualization exhibits a linear growth pattern, where each segment diminishes in size.

[1] The cross ratio of four collinear points A, B, C, and D is defined as $\frac{AC}{BC} : \frac{AD}{BD}$, where AC, BC, AD, and BD are distances between the points. For example, if the points are on the real line with coordinates a, b, c, and d, the cross ratio is calculated as $\frac{(a-c)(b-d)}{(a-b)(c-d)}$.

Fibonacci Polygon

Angles for the polygon are generated through a recursive addition, each influenced by the Fibonacci sequence. These angles are then used to compute the vertices of the polygon, ensuring the closure condition is satisfied. The polygon unfolds in a convex form with vertices positioned at intervals dictated by the sum of the angles of the preceding two steps. An angular progression, driven by the additive property of the Fibonacci sequence, culminates in a closed convex polygon.

Fibonacci Spiral

The spiral is crafted by plotting points whose radial separation aligns with the inverse Fibonacci ratios. Each turn of the spiral draws further from the center, illustrating the expansive nature of the sequence in a two-dimensional plane.

Angular Fibonacci Spiral

Commencing from a base angle, each subsequent angle is incremented by an amount derived from the Fibonacci sequence. The vertices thus created are connected, forming a path that spirals outward.

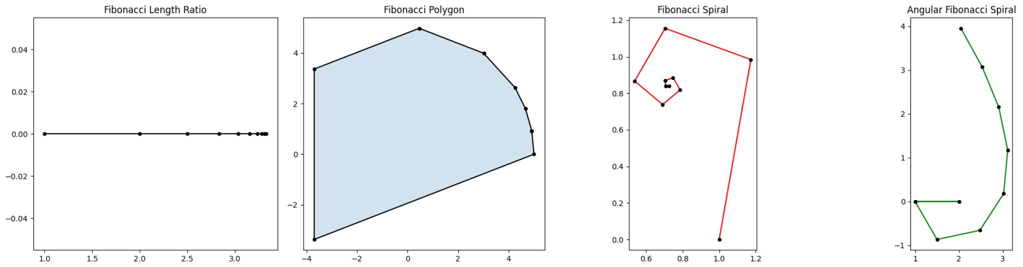


Figure 10.2: Polygon Numbers.

Problem 10.1 How would varying both ratio of length of line segments and ratio of angles?

10.3 Figurative Polygon Numbers

The polygonal numbers are a sequence of numbers that can be represented as dots arranged in the shape of regular polygons. We are interested here in stratifying the prime numbers by these sequences in order to pin down some further structure in the primes. We will explore here how the pattern of polygonal numbers extends from triangles to other polygons.

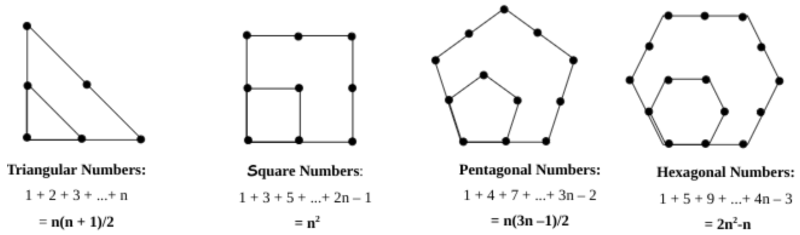


Figure 10.3: Polygon Numbers.

Triangular numbers, T_n are 3-gon numbers whose dot representation form a triangular pattern. The n-th triangular number is the number of dots in the triangular arrangement, formed as a result of stacking dots to form an equilateral triangle with formula :

$$P_4(n) \equiv T(n) = \frac{n\cdot(n+1)}{2}.$$

A pattern emerges when adapting the formula for square numbers, $S_n = n^2 + 0$. to higher r-gon forms from which we infer without full induction the formula for n-gon numbers.

- Square (r=4):

$$P_4(n) = \frac{n(2n+0)}{2} = n^2$$

- Pentagonal (r=5):

$$P_5(n) = \frac{n(3n-1)}{2}$$

- Hexagonal (r=6):

$$P_6(n) = \frac{n(4n-2)}{2} = n(2n-1)$$

- Octagonal (r=8):

$$P(n) = \frac{n(6n-4)}{2} = n(3n-2)$$

Theorem 10.3.1 — *r*-sided polygonal numbers. have an nth generator function of the form:

$$p_n(r) = \frac{n}{2}\cdot((r-2)n+(4-r)) = \frac{1}{2}[(r-2)n^2-(r-4)n] \tag{10.1}$$

Problem 10.2 How might one utilise this formula to determine the sum of the recipricol of hexagonal numbers?

10.3.1 Triangle Number Sum and Difference Quartets

Our data analysis starts from the initial observation that within the subset of triangular numbers: $T(3) = 6, T(5) = 15, T(6) = 21, T(8) = 36$ we note that the sum and difference for the pair $T(5)$ and $T(6)$ are also triangle numbers:

$$T(5)+T(6) = 15+21 = 36 = T(8)$$
$$T(6)-T(5) = 21-15 = 6 = T(3)$$

We therefore ask within the the full set of T_n which pairs of triangular numbers T_x and T_y combine such that both their sum and difference are also triangular numbers. That is, for a pair T_x and T_y, both $T_x + T_y$ and $|T_x - T_y|$ should also be represented as T_z for some integer z.

The code, addingTriangles.ipynb generates the scatter plot presenting all pairs of triangular numbers that satisfy such conditions. Each co-ordinate point (T_x, T_y) indicates that the numbers T_x and T_y satisfy the property that both their sum and difference are triangular numbers.

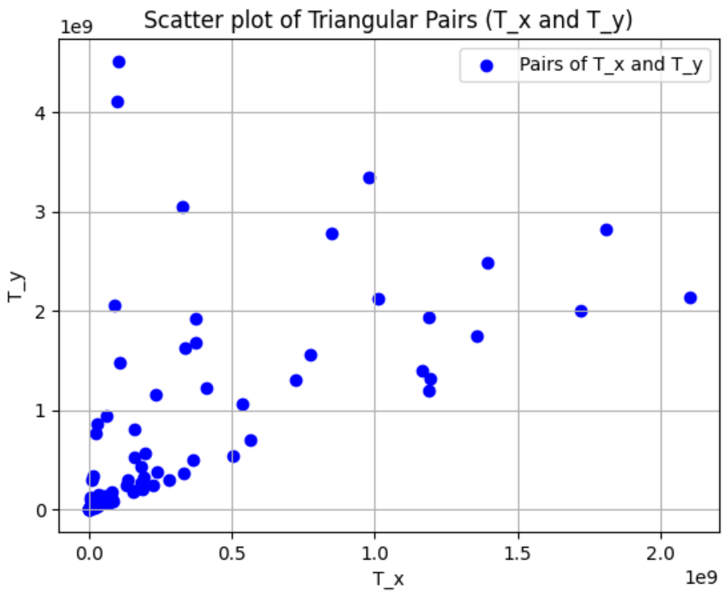


Figure 10.4: Sum and Difference Triangle Numbers up to n=100000

The code also delivers a table that presents the set of such sum difference quartets.

T(5) = 15	T(6) = 21	T(8) = 36	T(3) = 6
T(14) = 105	T(18) = 171	T(23) = 276	T(11) = 66
T(27) = 378	T(37) = 703	T(46) = 1081	T(25) = 325
T(39) = 780	T(44) = 990	T(59) = 1770	T(20) = 210
T(54) = 1485	T(91) = 4186	T(106) = 5671	T(73) = 2701

Table 10.1: Sum Difference Quartet of Triangle numbers

The function `triangular(n)` from addingTriangles.ipynb performs four main tasks:

1. Compute the list of triangular numbers up to a user-defined n.
2. Iterate over pairs of these triangular numbers, calculating their sum and difference.
3. Check if both the sum and difference are also triangular numbers.
4. If both conditions are satisfied, the pair is intended for visualization on a scatter plot.

```
def triangular(n):
    return n * (n + 1) // 2
try:
    max_n = int(input("Enter the value of n to sum  triangular numbers: "))
```

```
except ValueError:
    print("Please enter a valid integer.")
    exit()
triangular_numbers = [triangular(n) for n in range(1, max_n + 1)]
triangular_set = set(triangular_numbers)
max_width = len(str(triangular_numbers[-1]))
Tx = []
Ty = []
for i in range(len(triangular_numbers)):
    for j in range(i + 1, len(triangular_numbers)):
        sum_ = triangular_numbers[i] + triangular_numbers[j]
        diff = triangular_numbers[j] - triangular_numbers[i]
        if sum_ in triangular_set and diff in triangular_set:
            sum_index = triangular_numbers.index(sum_) + 1
            diff_index = triangular_numbers.index(diff) + 1
```

- **Error Handling:** A `try-except` block captures invalid input, ensuring the program requires a valid integer for *max_n*.
- **Optimization:** A set, `triangular_set`, is used to improve the efficiency of membership tests for determining if a sum or difference is a triangular number, leveraging the faster performance of set operations over list searches.

Noteworthy Aspects of the implementation are:

- The program efficiently generates triangular numbers using a list comprehension, encapsulating the generation logic within the `triangular` function.
- Iteration over triangular number pairs is systematically performed, with consideration for the unique properties of triangular numbers, especially focusing on their sum and difference.
- The use of lists `Tx` and `Ty` suggests a preparation for data visualization, likely through scatter plot, though the actual plotting implementation is not included in the snippet.

10.3.2 Primitive Triangle Square pairs

Our dataset of focus here is of products of triangular numbers, $T(i)$ and $T(j)$, that form perfect squares excluding those "non-trivial" cases of *congruent squared pair* products where $T(i) = T(j)$, or those products involving $T(1)$. Rather, we are interested in the following couples.

> **Definition 20** *Primitive Triangular Square pair* $(T(i), T(j))$, as a pair of Triangle numbers that satisfies the following conditions:
> - $i \neq j$ to exclude congruent squared pairs where $T(i) = T(j)$,
> - $i \neq 1$ and $j \neq 1$ to exclude products involving $T(1)$.

Some instances of such couples for $T(2) = 3$ are tabulated in Table 10.2 such as $(T(2), T(24))$ whose product gives a square of 30 side length. The code productTriangles.ipynb produces the log-log plot of Figure 10.5 of a sample of all such *primitive square-paired triangles* $(T(i), T(j))$.

Expression	Result	Square Representation
$T(2) \times T(24)$	900	30^2
$T(2) \times T(242)$	88209	297^2
$T(2) \times T(2400)$	8643600	2940^2
$T(2) \times T(23762)$	846984609	29103^2

.

Table 10.2: Sample of primitive square triangle pairs for $T(2)$.

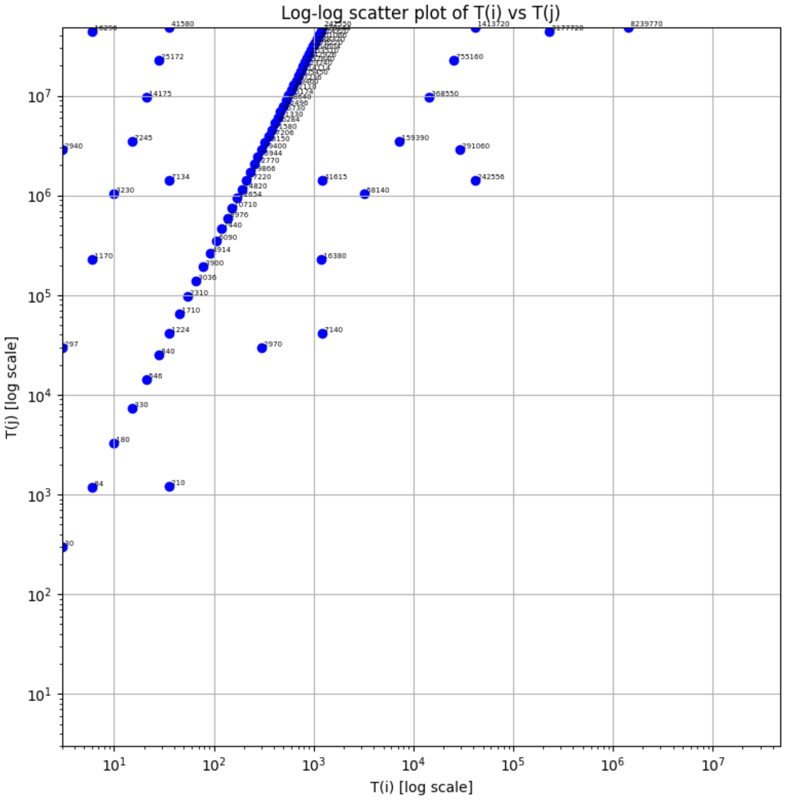


Figure 10.5: Non congruent triangle-pair squares for up to i=10000

The code includes a nested loop structure to find pairs of triangular numbers, $T(i)$ and $T(j)$, whose product is a perfect square. The outer loop iterates over i, and the inner loop iterates over j, to consider the product $T(i) \times T(j)$:

```
def triangular(n):
    return n * (n + 1) // 2
def is_square(n):
    root = int(n**0.5)
    return root * root == n
def find_square_triangles(max_range):
    data = []  # Collecting data for all T(i) and T(j)
    for i in range(2, max_range):  # Start from 2 to avoid T(1)
        for j in range(i+1, max_range):
            product = triangular(i) * triangular(j)
            if is_square(product):
                square_side_length = int(product**0.5)
                data.append((i, j, triangular(i), triangular(j), square_side_length))
    return data
```

To ensure we find only the set of primitive pairs that result in squares, we nest the loop according to $i \neq j$, $i \neq 1$, and $j \neq 1$:

- **Exclude Congruent Squared Pairs:** avoid cases where the same triangular number is squared, i.e., $T(n) \times T(n)$, since this will always result in a perfect square and is not of particular interest. This is achieved by ensuring that j starts from $i+1$, avoiding the case when $i = j$.
- **Exclude $T(1)$ Instances:** Since $T(1) = 1$, to avoid this trivial multiplication, we start the outer loop from $i = 2$.
- **Exclude $T(i) = T(j)$:** To ensure we do not include 'perfect triangle squares' where $T(i) = T(j)$, we again start the inner loop from $j = i+1$.

10.4 Figurative speaking Recipricols

Following "Beyond the Basel Problem: Sums of Reciprocals of Figurate Numbers," We aim here to summarise the derivation of a formula for the sum of the reciprocals of figurate numbers, starting from the infinite series for a given polygonal number side a.

Consider the sum of the reciprocals of the figurate numbers defined by:

$$S_a := \sum_{n=1}^{\infty} \frac{2}{(a-2)n^2-(a-4)n}.$$

For even values of $a \geq 6$, this can be represented as: Given the sum of reciprocals of figurate numbers:

$$S_a := \sum_{n=1}^{\infty} \frac{2}{(a-2)n^2-(a-4)n},$$

for even values of $a \geq 6$, we consider the modified series:

$$\sum_{n=1}^{\infty} \frac{1}{n((a-2)n-(a-4))} x^{(a-2)n-(a-4)} \Bigg|_{x=1}.$$

Utilizing the Maclaurin series for $\ln(1-t) = -\sum_{n=1}^{\infty} \frac{1}{n} t^n$, the series above can be identified as an antiderivative of the form:

$$-\frac{1}{x^{a-3}} \ln(1-x^{a-2}),$$

after setting x to 1 post-integration.

Using the Maclaurin series expansion for $\ln(1-t)$ and performing integration by parts, we find the antiderivative of the form:

$$\int \frac{1}{x^{a-3}} \ln(1-x^{a-2})\, dx.$$

Applying integration by parts with $u = \ln(1-x^{a-2})$ and $dv = x^{-a+3}\, dx$, we get:

$$\int x^{-a+3} \ln(1-x^{a-2})\, dx = \dots$$

We replace a by $2k+2$ to match our desired S_{2k+2}, and after simplifying, we use l'Hôpital's rule to evaluate the limit of the integral as x approaches 1 from the left, resulting in:

$$\lim_{x\to 1^-} F_{2k+2}(x) = 0.$$

Thus, the sum S_{2k+2} is obtained by evaluating the negative of the limit of $F_{2k+2}(x)$ as x approaches 0 from the right and as x approaches 1 from the left. The application of l'Hôpital's rule simplifies the expression to:

$$S_{2k+2} = -2\left[\lim_{x\to 0^+} F_{2k+2}(x) - \lim_{x\to 1^-} F_{2k+2}(x)\right].$$

The resulting expression for S_{2k+2} is the sum of the reciprocals of the figurate numbers for the specific side $2k+2$.

We demonstrate how setting $k=2$ yields the sum of reciprocals of hexagonal numbers:

$$S_{2k+2} = \frac{\ln(k)}{k-1} - \frac{1}{k-1}\sum_{j=1}^{k-1} \cos\left(\frac{2j\pi}{k}\right) \ln\left(2-2\cos\left(\frac{j\pi}{k}\right)\right) + \dots$$

For $k=2$, the formula simplifies to:

$$S_6 = \ln(2) - \frac{1}{1}\left[\cos(\pi)\ln(2-2\cos(\frac{\pi}{2}))\right] + \dots$$

Since $\cos(\pi) = -1$ and $\cos(\frac{\pi}{2}) = 0$, this further simplifies to:

$$S_6 = \ln(2) - (-\ln(2)) + \dots$$

The dots indicate terms that will be evaluated next. Observing that for $k=2$, the sums involving sine terms will vanish because $\sin(\frac{2j\pi}{2}) = \sin(j\pi)$ is zero for integer j. Thus, the only non-vanishing term for $k=2$ is the logarithmic term, which simplifies to $2\ln(2)$, matching the value provided in the table for hexagonal numbers:

$$S_6 = 2\ln(2)$$

For even values of a, this sum can be expressed using a formula involving logarithms and trigonometric functions, which in the case of $a = 2n$ for $n \geq 2$, is given by:

$$
\begin{aligned}
S_{2k+2} = {} & \frac{\ln(k)}{k-1} - \frac{1}{k-1}\sum_{j=1}^{k-1} \cos\left(\frac{2j\pi}{k}\right) \ln\left(2 - 2\cos\left(\frac{j\pi}{k}\right)\right) \\
& + \frac{2}{k-1}\sum_{j=1}^{k-1} \sin\left(\frac{2j\pi}{k}\right) \tan^{-1}\left(\frac{1-\cos\left(\frac{j\pi}{k}\right)}{\sin\left(\frac{j\pi}{k}\right)}\right) \\
& - \frac{2}{k-1}\sum_{j=1}^{k-1} \sin\left(\frac{2j\pi}{k}\right) \tan^{-1}\left(\frac{-\cos\left(\frac{j\pi}{k}\right)}{\sin\left(\frac{j\pi}{k}\right)}\right)
\end{aligned}
$$

Applying this formula for specific values of k delivers a table of values for S_{2k+2}, correlating to the sums of reciprocals of various figurate numbers. The table derived from this formula is as follows:

Number of Sides	Name of Polygons	Sum of Series
3	Triangular	2
4	Square (Basel Problem)	$\frac{\pi^2}{6}$
6	Hexagonal	$2\ln(2)$
8	Octagonal	$3\ln(3) - \frac{\sqrt{3}\pi}{4} + \frac{\sqrt{3}\pi}{12}$
10	Decagonal	$\ln(2) + \frac{\pi}{6}$
14	Tetraikaidecagonal	$\frac{2}{5}\ln(2) + \frac{3}{10}\ln(3) + \frac{\sqrt{3}\pi}{10}$

Table 10.3: Values of Sums of Reciprocals of Figurate Numbers

For example, the sum of the reciprocals of hexagonal numbers, which are a specific type of figurate numbers corresponding to $k = 2$, is given by $2\ln(2)$. This result and others in the table are derived by evaluating the series S_{2k+2} using the above formula for the corresponding k. For further reading on this topic, the reader is referred to the original paper available at ResearchGate.

Conclusion

Through the application of the Maclaurin series, integration by parts, and l'Hôpital's rule, we derive a comprehensive formula for the sum of reciprocals of figurate numbers, denoted by S_{2k+2}. The proof demonstrates a remarkable connection between the figurate numbers, the natural logarithm, and trigonometric identities.

10.4.1 recurrence relations

1. $T(n) = n + T(n-1)$ with $T(1) = 1$
2. $S(n) = (2n-1) + S(n-1)$ with $S(1) = 1$
3. $O(n) = n + n + 1 + O(n-1)$ with $O(1) = 2$
4. $P(n) = 3n - 2 + P(n-1)$ with $P(1) = 1$

10.4.2 Gauss's three triangle Theorem

Gauss at the tender age of $19 = 1+3+15$ proved that the Natural numbers can all be represented as the sum of at most three triangle numbers. Here are the first and last twenty representations for a mere n=1000:

n	T_1	T_2	T_3	s
1	1	-	-	1
2	2	-	-	1
3	1	1	1	3
4	4	-	-	1
5	1	1	3	3
6	6	-	-	1
7	1	3	3	3
8	1	1	6	3
9	3	3	3	3
10	1	3	6	3
11	11	-	-	1
12	1	1	10	3
13	1	6	6	3
14	1	3	10	3
15	3	6	6	3
16	3	3	10	3
17	1	1	15	3
18	6	6	6	3
19	1	3	15	3
20	20	-	-	1
980	1	276	703	3
981	15	105	861	3
982	1	78	903	3
983	1	36	946	3
984	3	78	903	3
985	3	36	946	3
986	28	55	903	3
987	6	78	903	3
988	6	36	946	3
989	10	276	703	3
990	21	66	903	3
991	1	210	780	3
992	1	1	990	3
993	1	496	496	3
994	1	3	990	3
995	1	91	903	3
996	3	3	990	3
997	1	6	990	3
998	1	136	861	3
999	3	6	990	3
1000	3	136	861	3

This code produces the following scatterplot:

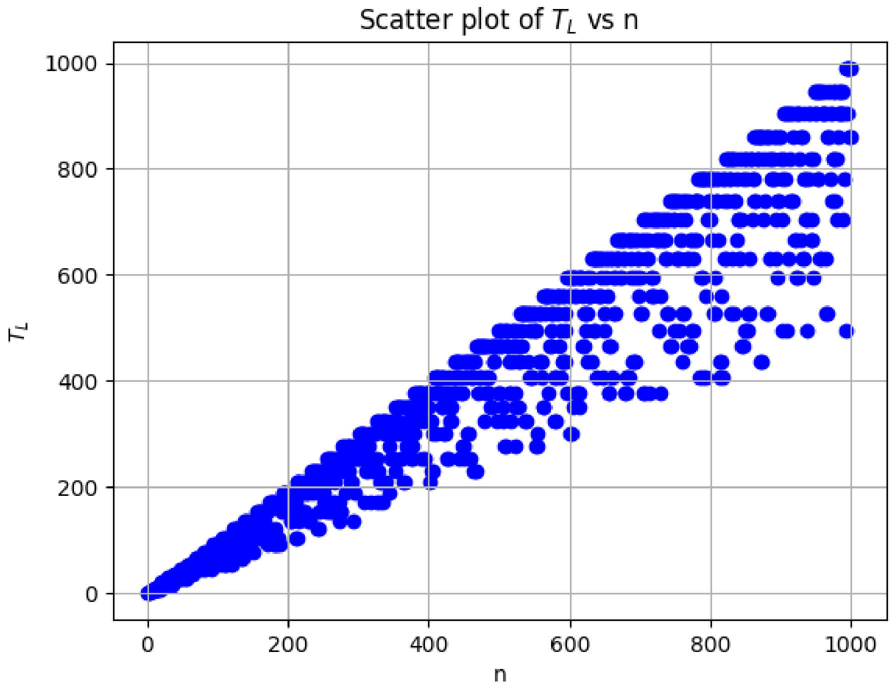


Figure 10.6: Gauss's three triangles: plot of Highest Triangle number versus n for up to n=1000.

The key features of the code are:

```
for n in range(1, max_n+1):
    t1, t2, t3 = represent_as_triangles(n, triangular_numbers)
    s = sum([1 for t in [t1, t2, t3] if t != "-"])
    table.append([n, t1, t2, t3, s])

print(f"{'n':<5}{'T1':<5}{'T2':<5}{'T3':<5}{'s'}")
print('-' * 25)
for row in table:
    print(f"{row[0]:<5}{row[1]:<5}{row[2]:<5}{row[3]:<5}{row[4]}")
```

negative n Series

The table generated presents the coefficients of the quadratic generators for the polygon number series for negative n values, ranging from $n = 0$ to $n = -10$, across various polygons with sides from 3 (triangles) to 8 (octagons).

r	a	b	c
3	$\frac{1}{2}$	$-\frac{3}{2}$	1
4	1	-2	1
5	$\frac{3}{2}$	$-\frac{5}{2}$	1
6	2	-3	1
7	$\frac{5}{2}$	$-\frac{7}{2}$	1
8	3	-4	1

These coefficients were derived by analyzing the first and second differences of the polygon number series for each r-gon. The general quadratic generator for the negative n series is given by:

$$P_r^-(n) = an^2 + bn + c$$

where the coefficients a, b, and c depend on the number of sides r of the polygon. and can be inferrred as $a = \frac{r-2}{2}$, $b = -\frac{r}{2}$, and $c = 1$, giving:

$$P_r^{-n} = \frac{r-2}{2}n^2 - \frac{r}{2}n + 1 = \frac{1}{2}[(r-2)n^2 - rn + 2]$$

We confirm this by substituting $r = 6$ (hexagon) into formula which gives:

$$P_6^{-n} = \frac{6-2}{2}n^2 - \frac{6}{2}n + 1$$

$$P_6^{-n} = 2n^2 - 3n + 1.$$

10.4.3 Polygon number Ulam spirals

The Ulam spiral for the first, second and fourth polygon numbers look like the following.

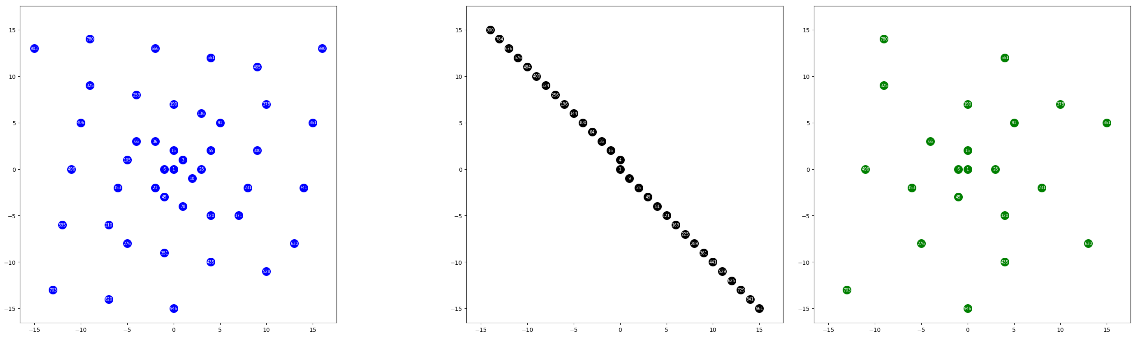
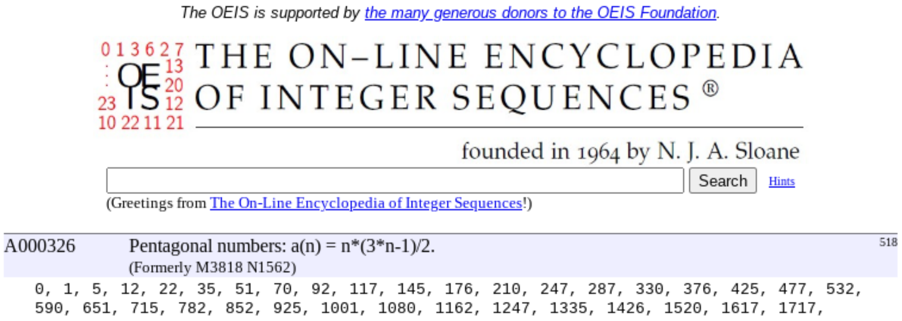


Figure 10.7: Ulam Spiral for triangle, square and hexagon numbers.

The website OEIS run by N Sloane is a remarkable website that tracks any such sequence that you may consider. We can see the pentagonal number sequence in fig 17.18.



The OEIS is supported by the many generous donors to the OEIS Foundation.

0 1 3 6 2 7 13 20 12 21 11 22 10 23 OEIS THE ON-LINE ENCYCLOPEDIA OF INTEGER SEQUENCES ®

founded in 1964 by N. J. A. Sloane

Search Hints

(Greetings from The On-Line Encyclopedia of Integer Sequences!)

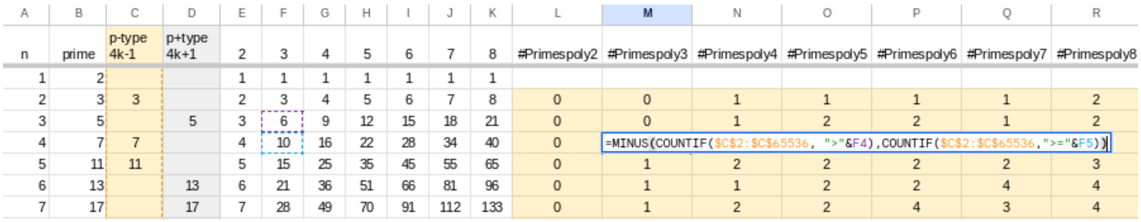
A000326 Pentagonal numbers: a(n) = n*(3*n-1)/2. 518
(Formerly M3818 N1562)

```
0, 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, 176, 210, 247, 287, 330, 376, 425, 477, 532,
590, 651, 715, 782, 852, 925, 1001, 1080, 1162, 1247, 1335, 1426, 1520, 1617, 1717,
```

Figure 10.8: OEIS sequence website.

10.5 Google Sheets Implementation of Polygon-Prime stratification

Now one way to investigate how many prime numbers are in between each of the polygon number sequences is to set up a COUNTIF clause in a spreadsheet as in fig 10.9 below.



A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
n	prime	p-type 4k-1	p+type 4k+1	2	3	4	5	6	7	8	#Primespoly2	#Primespoly3	#Primespoly4	#Primespoly5	#Primespoly6	#Primespoly7	#Primespoly8
1	2			1	1	1	1	1	1	1							
2	3	3		2	3	4	5	6	7	8	0	0	1	1	1	1	2
3	5		5	3	6	9	12	15	18	21	0	0	1	2	2	1	2
4	7	7		4	10	16	22	28	34	40	0	=MINUS(COUNTIF($C$2:$C$65536, ">"&F4),COUNTIF($C$2:$C$65536,">="&F5))					
5	11	11		5	15	25	35	45	55	65	0	1	2	2	2	2	3
6	13		13	6	21	36	51	66	81	96	0	1	1	2	2	4	4
7	17		17	7	28	49	70	91	112	133	0	1	2	2	4	3	4

Figure 10.9: Google Sheets count of Fermat Numbers stratified by polygon numbers.

Google sheets enables you to plot the chart of each of the frequencies of intervening primes between polygon numbers as in fig 10.10 below.

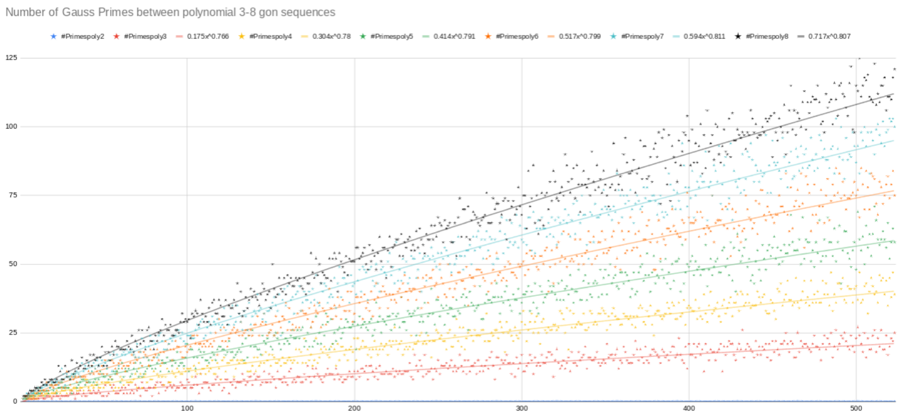


Figure 10.10: Google Sheets chart of Gauss Primes stratified by polygon numbers.

10.6 Python Implementation of Polygon-Prime stratification

This is more easily implemented in Python using the code Here for n=1000 we note the power law best fit of the order $4/5$

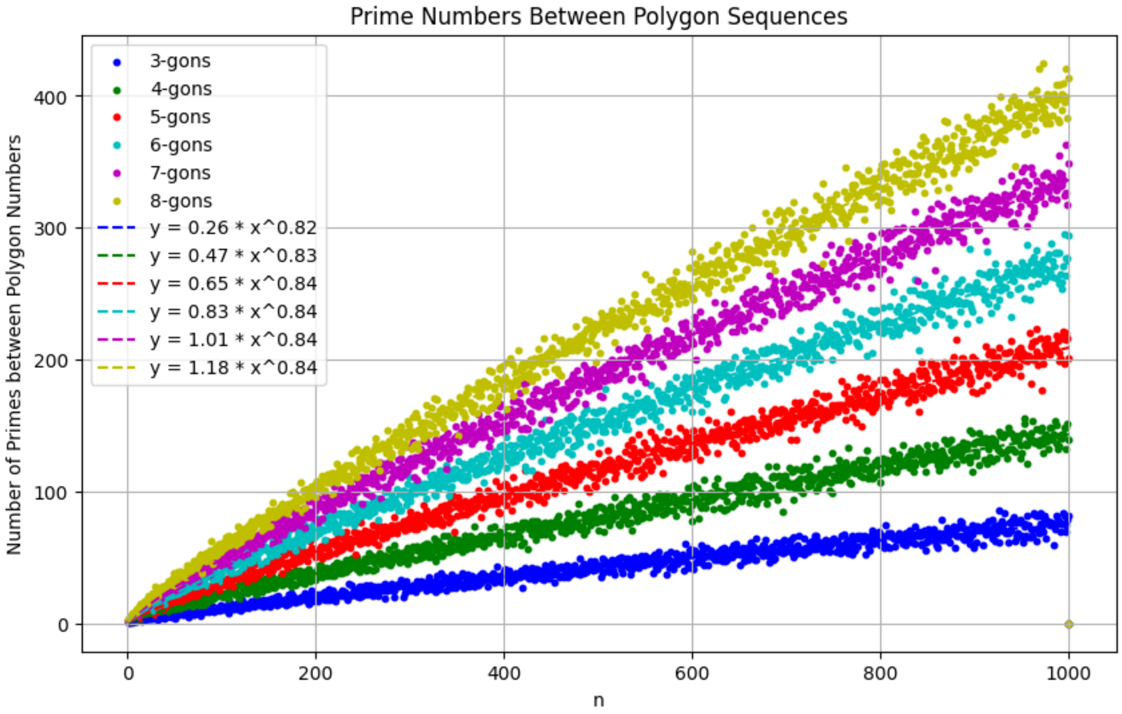


Figure 10.11: Python chart of Primes stratified by polygon numbers.

That is to say that the primes appear between respective polygon numbers of respective r-gon sequences in increasing frequency according to a $4/5$ power law reminiscent of Kleiber's Law[2]

[2] This law has important implications in ecology, physiology, and evolutionary biology. It suggests that larger animals are more energy-efficient per unit of body mass, which may provide advantages in terms of resource utilization and survival. However, the exact value of the exponent b can vary across different taxonomic groups and physiological conditions. While Kleiber's Law is widely observed, there are exceptions and complexities that researchers continue to investigate.

or the Metabolic Scaling Law, describing the relationship between the surface area and volume of animals and their metabolic rate. The law states that as an animal's size increases, its metabolic rate increases at a slower rate than its volume. In other words, larger animals have a relatively lower metabolic rate per unit of body mass compared to smaller animals expressed as $B = a \cdot M^b$, where:

B : Animal's basal metabolic rate (energy consumption at rest)

M : Animal's body mass

a : Proportionality constant

b : Exponent, generally within the range of 0.67 to 0.75

The code has functions to generate polygon numbers and the Power law function of best fit while generating lists of r-polygon running from 1 to n and a list of prime numbers using the primerange function:

```
def polygon_number(n, r):
    return n * ((r - 2) * n + (4 - r)) // 2
def power_law(x, a, b):
    return a * np.power(x, b)
n_values = list(range(1, 10001))
r_values = list(range(3, 9))
polygon_sequences = [[] for _ in r_values]
for n in n_values:
    for i, r in enumerate(r_values):
        polygon_sequences[i].append(polygon_number(n, r))

prime_numbers = list(primerange(1, max(polygon_sequences[-1])))
```

The following code snippet is responsible for calculating the number of prime numbers that fall between consecutive polygon numbers in the generated sequences.

```
prime_counts = [[0] * len(polygon_sequences[0]) for _ in r_values]

for i, sequence in enumerate(polygon_sequences):
    for j in range(len(sequence) - 1):
        prime_counts[i][j] = len([p for p in prime_numbers if sequence[j]
        < p < sequence[j + 1]])
```

This first line initializes a list of lists named `prime_counts`. The outer list has a length equal to the number of different values of r (polygon sides). Each inner list has a length equal to the number of polygon numbers generated for the specific value of r. All inner lists are initially filled with zeros. The expression `[0] * len(polygon_sequences[0])` creates a list of zeros with a length equal to the number of polygon numbers in the sequence for the first value of r.

The outer loop variable _ is used as a placeholder since its value is not used in the loop body. The outer loop iterates over each polygon sequence in the `polygon_sequences` list. i represents the index of the current sequence and `sequence` represents the current list of polygon numbers. The inner loop iterates over each element in the `sequence` list except the last one. j represents the index of the current element. Inside the inner loop, a list comprehension is used to count the number of prime numbers that fall between the current and the next polygon numbers. `[p for p in prime_numbers if sequence[j] < p < sequence[j + 1]]` creates a list of prime numbers (p) from `prime_numbers` that satisfy the condition that they are greater than the current polygon number (`sequence[j]`) and less than the next polygon number (`sequence[j + 1]`). The `len()` function is then used to count the number of prime numbers in this list. The count is stored in the `prime_counts` list at the corresponding position for the given r value (i) and the current polygon number index (j). The plot itself is generated by the following snippet:

```
for i, color in enumerate(colors):
    plt.scatter(n_values, prime_counts[i], color=color, label=labels[i], s=10)
    popt, _ = curve_fit(power_law, n_values, prime_counts[i])
    best_fit_equation = f'y = {popt[0]:.2f} * x^{popt[1]:.2f}'
    best_fit_equations.append(best_fit_equation)

for i, best_fit_equation in enumerate(best_fit_equations):
    plt.plot(n_values, power_law(n_values, *curve_fit(power_law, n_values,  prime_counts[i]
```

For each value of i and for every color in the list of colors:

Use the scatter plot function to display the data points: - n_values on the x-axis - `prime_counts`[i] on the y-axis - Color the points with the corresponding color - Label the data points with the associated label from `labels` - Set the size of the points to 10

Use the `curve_fit` function to fit a power law model to the data: - Input n_values as the independent variable - Input `prime_counts`[i] as the dependent variable - Capture the fitted parameters in $popt$ - Disregard the second output

Construct the equation of the best-fit line: - Calculate the coefficient a from $popt[0]$ - Calculate the exponent b from $popt[1]$ - Create the equation $y = a \cdot x^b$

Add the constructed equation to the list of best-fit equations.

10.6.1 Transformation to Log-Log Plot

To analyze data that seems to follow a fractional power law $y = c \cdot x^m$ in a linear-linear plot, we transform the data into a log-log plot. The equation becomes:

$$\log(y) = \log(c) + m \cdot \log(x)$$

Linear Regression:

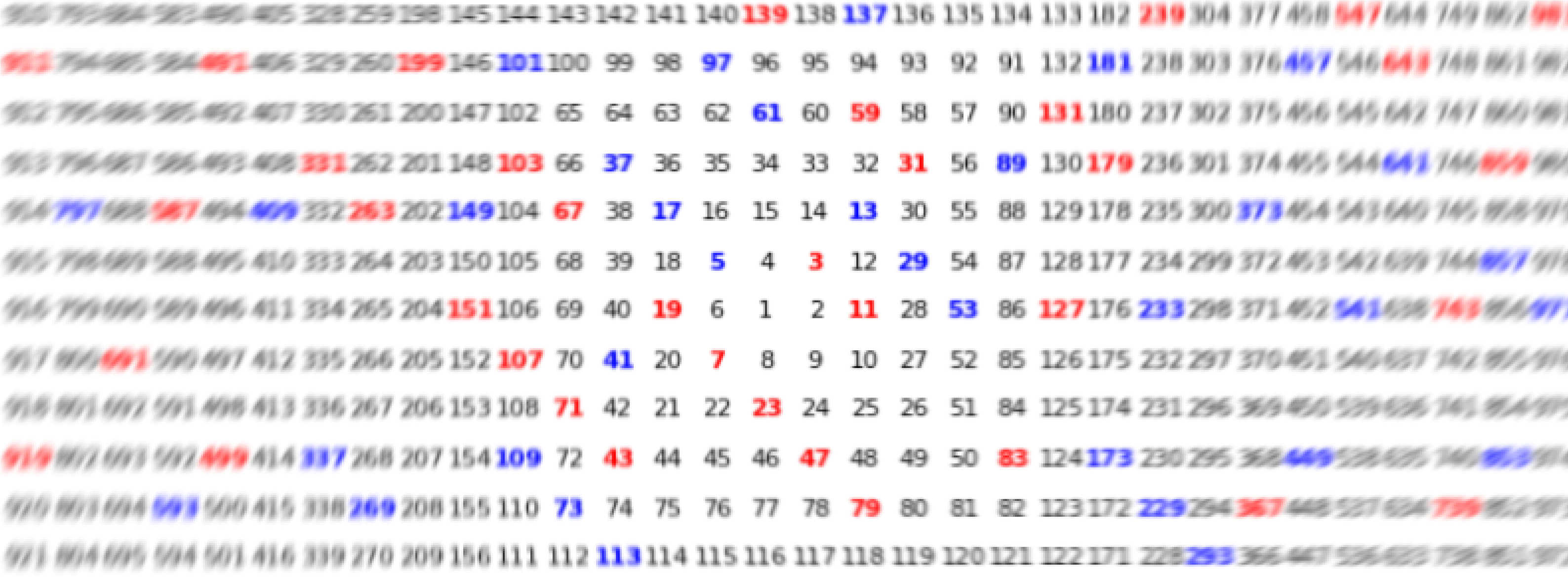
In the log-log plot, the power law relationship appears as a straight line. Performing a linear regression on the log-transformed data helps us determine the values of m and $\log(c)$.

Interpreting the Results:

The slope m of the best fit line corresponds to the power exponent. It indicates the type and strength of the power law relationship. Positive m implies a direct relationship, while negative m implies an inverse relationship.

The intercept $\log(c)$ provides information about the coefficient c. Exponentiating $\log(c)$ gives the actual coefficient value. c controls the scaling of the power law and relates to the normalization or amplitude.

In summary, transforming data and performing linear regression on a log-log plot helps determine the power exponent (m) and coefficient (c) of the power law relationship. The exponent signifies the nature of the power law, while the coefficient controls its scaling.



11. Exageration

> "The beauty of mathematics only shows itself to more patient followers."
> — *Maryam Mirzakhani*

11.1 Double Factorial and Prime Divisibility

The lesser known variant of the factorial operation on the Naturals, $n!$ is the double factorial, $n!!$, the product of all integers up to n that have the same parity as n. Thus for even $n = 2k$, $n!! = 2^k \cdot k!$ represents the product of all even numbers and for odd $n = 2k+1$, it's the product of all odd numbers up to n. Now, for the subset of the prime number naturals, a double factorial operation can be defined based either on the *cardinal* value of the prime (the prime number itself) or the *ordinal* (position) of the prime in the sequence of all primes.

Cardinal-Based Double Factorial: $p_i!!$

Cardinal prime numbers are the primes as considered by their magnitude: the prime number 5 is the cardinal prime representing the quantity of five. The standard double factorial $p_i!!$ as a cardinal-based operation, treats the prime p_i like any other integer so that for all the (odd except 2) primes, $p_i!!$ is the product of all odd numbers up to p_i. This means for instance that$11!! = 11 \times 9 \times 7 \times 5 \times 3 \times 1$

primorial function$p(n)\#$

The primorial function can be defined in two contexts:

1. Primorial of an Integer n, denoted as $n\#$:

$$n\# = \prod_{\substack{p \leq n \\ p \text{ is prime}}} p$$

where the product is taken over all prime numbers p that are less than or equal to n.

2. Primorial of the nth Prime, denoted as $p(n)\#$:

$$p(n)\# = \prod_{i=1}^{n} p(i)$$

where $p(i)$ represents the ith prime number and we have thus for $p(5)\#$, the product of the first 5 prime numbers: $p(5)\# = 2 \times 3 \times 5 \times 7 \times 11 = 23101 = 11\#$

Ordinal-Based Double Factorial: $p_i\#\#$

Ordinal prime numbers are the primes considered by their position in the sequence of prime numbers: 5 is the 3rd ordinal prime number given the sequence $2, 3, 5, 7, \ldots$. For $[p_1, p_2, ..p_6, \cdots] = [2, 3, 5, 7, 11, 13, \cdots]$, the ordinal fifth prime 11 has single ordinal factorial $p_5\# = 11 \times 7 \times 5 \times 3 \times 2$ with a double ordinal factorial of $p_5\#\# = 11 \times 5 \times 2$ where we have introduces the notation $p_i\#$ to represent a product based on the ordinal position of the prime number in the sequence.:

- For even ordinal positions $i = 2k$: $p_i\# = p_{2k} \cdot p_{2k-2} \cdot p_{2k-4} \cdot \ldots$ down to lowest prime.
- For odd ordinal positions $i = 2k+1$: $p_i\# = p_{2k+1} \cdot p_{2k-1} \cdot \ldots$ down to lowest prime.

Regression Modelling of prime factorial ratios

The process of fitting data to a model is a fundamental task in statistical analysis. In the context of analyzing the ratios of prime factorials, we will deploy both exponential and power-law regression models. The ratio of cardinal double, $p_i!!$ to single, $p_i!$ is the reciprocal of $p_{i-1}!!$:

$$\frac{p_i!!}{p_i!} = \frac{1}{p_{i-1}!!}.$$

Consideration of $i = 4$ and $p_7 = 7$, suggests an analogous relationship for ordinal prime factorials:

$$\frac{p_4\#\#}{p_4\#} = \frac{7 \times 3}{7 \times 5 \times 3 \times 2} = \frac{1}{5 \times 2} = \frac{1}{\mathrm{p}_3\#\#}$$

generalising to: $p_i\# = p_i\#\# \times p_{i-1}\#\#$ for $i \geq 2$. Our first model of the cardinal double prime assumes an exponential relationship between the adjusted ratios of the prime factorials and the index of the prime numbers. As such the regression is formulated as $y = e^{ax+b}$ where y represents the ratio, x is the index of the prime number, and a, b are the parameters of the model. The regression is visualized in a scatter plot with a logarithmic scale using Python's `curve_fit` function from the `scipy.optimize` library.

In contrast for the ordinal primes we deploy a power law that describes a variety of phenomena in the natural and social sciences, and is characterized by a polynomial relationship of the form, $y = bx^a$ where a is typically a fractional power.

One challenge faced during such an implementation is the optimization algorithm's tendency to converge to a local minimum that might not capture the required (as it happens) negative power relationship. To address this, we need to constrain the parameters using the `bounds` argument in the `curve_fit` function. Setting bounds ensures the algorithm searches for a solution within a specified range and is useful when the expected behavior of the model is known a priori. We set the bounds for the power-law model exponent a to be negative, in anticipation of a decaying power-law relationship while the coefficient b was allowed to vary freely within positive values as: `bounds` $= ([-\infty, 0], [0, \infty])$

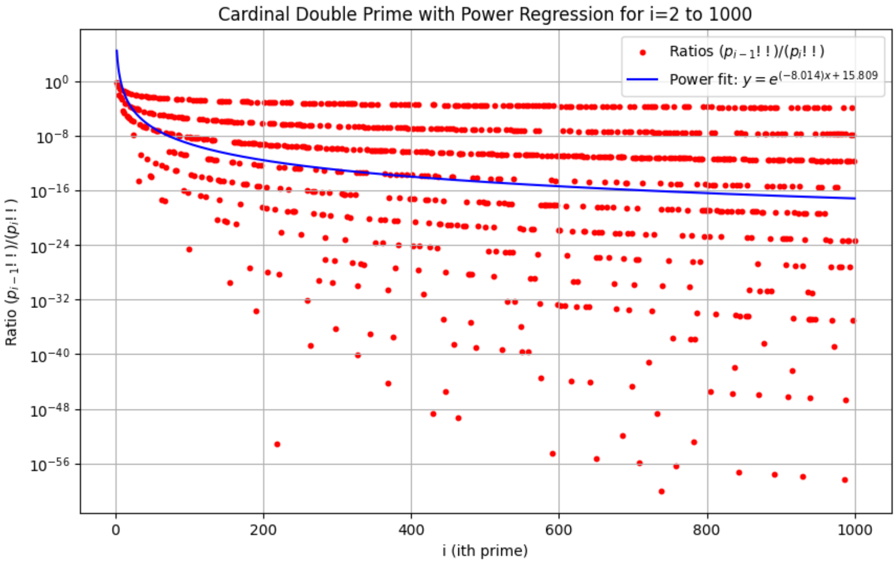


Figure 11.1: Exponential Law Fit for Cardinal Prime Double Factorial Ratio

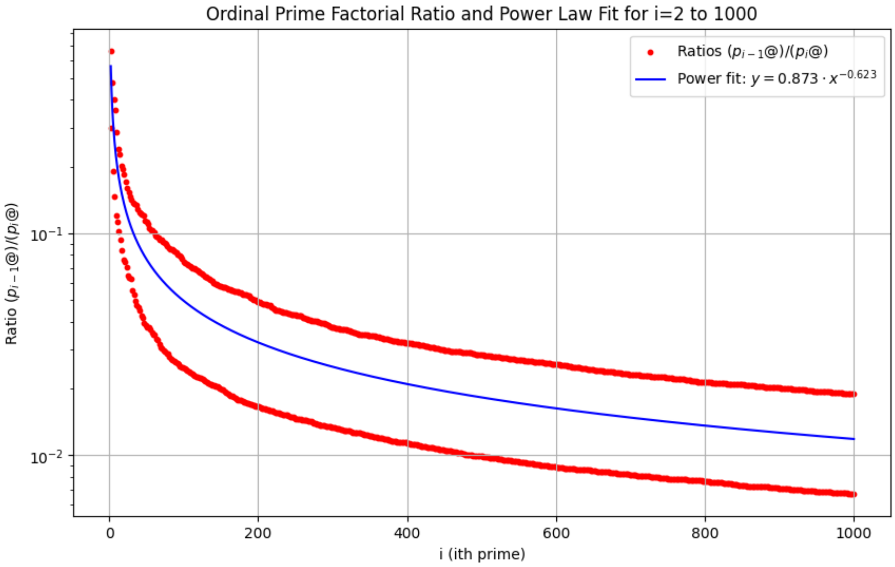


Figure 11.2: Power Law Fit for Ordinal Prime Double Factorial Ratio

Nearest Integer Function

The concept of the nearest integer function denoted by $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively is crucial for evaluating the divisibility of factorials by prime powers. The floor function, $\lfloor x \rfloor$, represents the greatest integer less than or equal to x, while the ceiling function, $\lceil x \rceil$, represents the smallest integer greater than or equal to x. The motivation for introducing the nearest integer function, especially the floor function, arises from the need to systematically count factors in factorial expressions. The floor function is used in Legendre's formula to count the number of occurrences of a prime p in the prime factorization of $n!$, thereby aiding in the evaluation of divisibility by prime powers.

Legendre's Formula and Divisibility

The divisibility of factorials by prime powers is systematically evaluated using the nearest integer function, via the floor function in Legendre's formula which efficiently counts the occurrences of a prime p in the prime factorization of $n!$, and is given by:

$$v_p(n!) = \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + \left\lfloor \frac{n}{p^3} \right\rfloor + \dots$$

Example: For $n = 10$ and $p = 2$, we see that 2^8 is the highest power of 2 dividing 10!:

$$v_2(10!) = \left\lfloor \frac{10}{2} \right\rfloor + \left\lfloor \frac{10}{2^2} \right\rfloor + \left\lfloor \frac{10}{2^3} \right\rfloor = 5 + 2 + 1 = 8$$

For even $n!!$, we first express it as $n!! = 2^k \cdot k!$. For example, for $n = 10$ (hence $k = 5$), $10!! = 2 \cdot 4 \cdot 6 \cdot 8 \cdot 10 = 2^5 \cdot 5!$. Applying Legendre's formula to 5!, the highest power of a prime p that divides 5! is calculated as follows:

$$v_p(5!) = \left\lfloor \frac{5}{p} \right\rfloor + \left\lfloor \frac{5}{p^2} \right\rfloor + \left\lfloor \frac{5}{p^3} \right\rfloor + \dots$$

In this case, for $p = 2$, we get:

$$v_2(5!) = \left\lfloor \frac{5}{2} \right\rfloor + \left\lfloor \frac{5}{2^2} \right\rfloor = 2 + 1 = 3$$

Adding the five 2's from 2^5 (since $10!! = 2^5 \cdot 5!$), the total count of 2's in 10!! is $5 + 3 = 8$. Thus, 2^8 is also the highest power of 2 dividing 10!!. And indeed for $p = 2$, the highest power of 2 that divides any $n!$ and $n!!$ is the same for even n. This is due to the fact that:

- For $n!!$, the term 2^k accounts for the factors of 2 contributed by the even numbers.
- The remaining product, $k!$, and the full factorial, $n!$, share the sequence of numbers $1, 2, \dots, k$, meaning the factors of 2 in this part of the sequence are counted in both $k!$ and $n!$.
- For even n, the structure of $n!$ ensures that the additional even numbers beyond k contribute exactly the factors of 2 that are accounted for by 2^k in $n!!$.

The relationship can be visualized by plotting the highest power of p that divides $n!$ and $n!!$ for a range of n. For even n and $p = 2$ the plots for $n!$ and $n!!$ coincide.

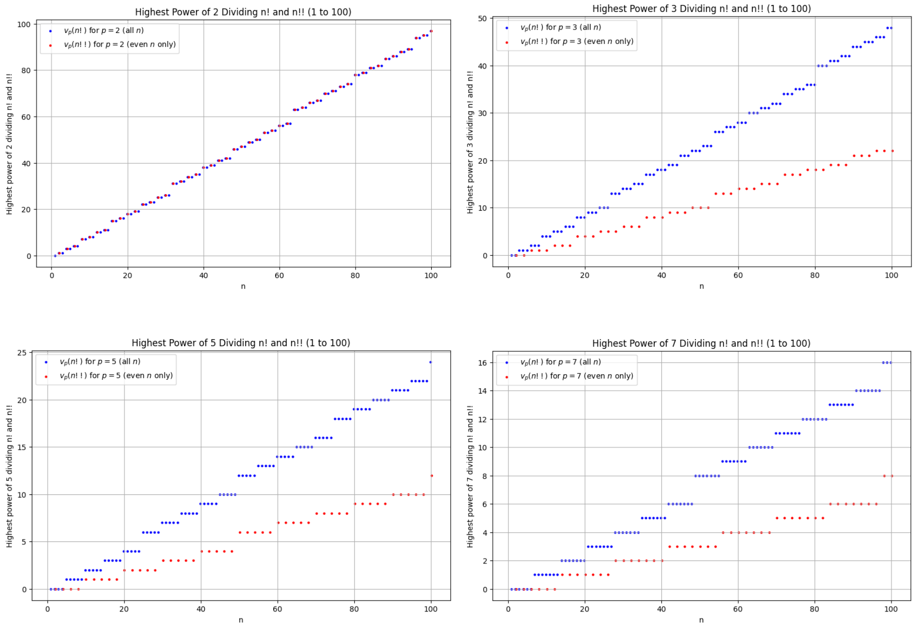


Figure 11.3: Legendre values for divisibility of $n!$ and $n!!$ by p=2,3,5,7

Calculating the highest power of 7 dividing $40!$:
Using Legendre's formula, we have:

$$\nu_7(40!) = \left\lfloor \frac{40}{7} \right\rfloor + \left\lfloor \frac{40}{7^2} \right\rfloor = 5 + 0 = 5$$

So, the highest power of 7 dividing 40! is 7^5. **Calculating the highest power of 7 dividing $40!!$:**
Now, since $40!! = 2^{20} \cdot 20!$, we find the highest power of 7 dividing $20!$:

$$\nu_7(20!) = \left\lfloor \frac{20}{7} \right\rfloor + \left\lfloor \frac{20}{7^2} \right\rfloor = 2 + 0 = 2$$

So, the highest power of 7 dividing $40!!$ is 7^2. Now for odd $n = 2k+1$, the double factorial $n!!$ represents the product of all odd numbers up to n, which does not have a straightforward reduction to a simpler factorial expression. Unlike the even case, there is no common factor we can extract from each term, making the application of Legendre's formula or similar methods less direct.

11.1.1 Euler, e as geometric mean of primes

e, as the base of the natural logarithm can also be represented as the very slowly converging limit of a form of a geometric mean of i primes:

$$e \approx \lim_{n\to\infty}\left(\prod_{i=1}^{n} p_i\right)^{1/p_n} = \lim_{i\to\infty} \sqrt[p_i]{p_i\#} \equiv \lim_{i\to\infty} \sqrt[p_i]{p_i\#\# \cdot p_{i-1}\#\#}.$$

In this formulation we make sure to distinguish the cardinal-powering of the geometric" average over an ordinal prime factorial $p_i\#$. To investigate this slowly converging limit, we will compute this geometric mean for increasingly large i using the Python code, PlotEulerGeometricPrime.ipynb.

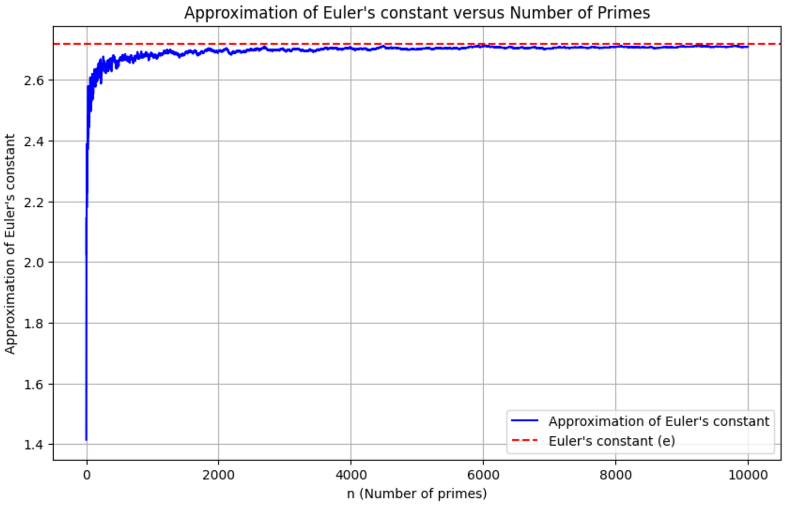


Figure 11.4: Euler’s constant as limit of Cardinal geometric mean of Prime Double Factorial

Our initial brute force approach might involve directly calculating the product of primes up to the i-th prime and then raising this product to the power of $1/p_i$.

```
def approximate_euler_constant(n):
    primes = [prime(i) for i in range(1, n + 1)]
    e_approximations = []
    for i in range(1, n + 1):
        product_primes = np.prod(primes[:i])
        e_approx = product_primes ** (1.0 / primes[i-1])
        e_approximations.append(e_approx)
    return e_approximations[-1]
```

While straightforward, this is not computationally stable as it involves the computation of very large numbers. To optimize the computation, we use in PlotEulerGeometricPrime.ipynb the distributive property of exponentiation $(a \cdot b)^{1/n} = a^{1/n} \cdot b^{1/n}$ taking the logarithm of each prime number, summing these logarithms, and then exponentiating the result to avoid the avoid the overflow problems[1] associated with large products:

[1] In Python, floating point numbers are subject to limitations that arise from their digital representation, which includes a sign, an exponent, and a fraction (significand). Overflow refers to a computation yielding a result beyond the largest value that can be represented, $\approx 1.8 \times 10^{308}$, leading to the result being marked as infinity ("inf"). Conversely, underflow occurs with results too close to zero, under $\approx 5.0 \times 10^{-324}$, causing them to be rounded to zero. Further calculations with these values are unreliable. Additionally, the finite length of the significand means precision loss is inevitable during operations, especially when aligning numbers for addition or subtraction, or when the product or quotient of numbers contains more significant digits than can be accurately represented.

```
def approximate_euler_constant_using_logs(n):
    primes = [prime(i) for i in range(1, n + 1)]
    log_sum_primes = 0
    e_approximations = []
    for i in range(1, n + 1):
        log_sum_primes += np.log(primes[i-1])
        # Compute the exponentiated average of the log values
        e_approx = np.exp(log_sum_primes / primes[i-1])
        e_approximations.append(e_approx)
    # Get the last approximation as the result
    return e_approximations[-1]
```

11.1.2 Wilson's Formula

Plotting $(p-1)! \mod p$ versus p gives the straight line graph, Figure **??** as implemented by the following snippet

```
from sympy import primerange, factorial
primes = list(primerange(2, 10000))
# Compute (p-1)! % p for each prime p
factorial_mod_p = [(p, factorial(p-1) % p) for p in primes]
# Separate the list into two lists: x (primes) and y ((p-1)! % p)
x, y = zip(*factorial_mod_p)
```

plotWilsonModular.ipynb generates a list of prime numbers within a specific range using the `primerange` function from the `sympy` library:

- Generate primes: $P = \{p \mid p \text{ is prime}, 2 \leq p < 10000\}$.
- Compute $(p-1)! \mod p$ for each prime $p \in P$.
- Store the results in two lists: $X = \{p\}$ and $Y = \{(p-1)! \mod p\}$.

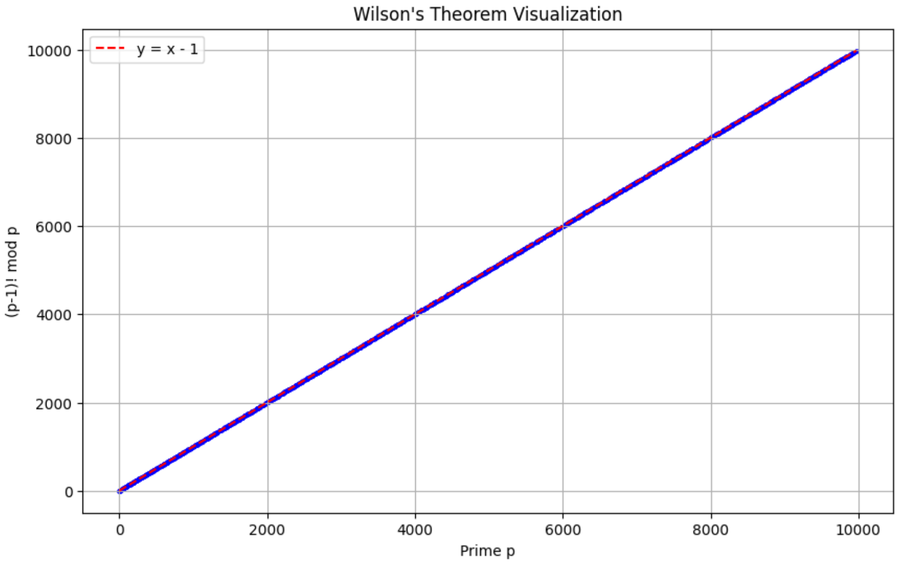


Figure 11.5: Wilson theorem

This is a picture of a result known as Wilson's formula, which for an odd prime p states that the residue of $(p-1)!$ mod p is -1 :$(p-1)! = -1 \mod p$ For $p = 13$:

$$\begin{aligned}(p-1)! &= (13-1)! = 12!\\ &= 12\cdot 11\cdot 10\cdot 9\cdot 8\cdot 7\cdot 6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1 = 479001600\\ 479001600 &\equiv -1 \mod 13\end{aligned}$$

It is though more illuminating and efficient computationally to deal with the congruences on the fly. So again for $p = 13$, we compute $(p-1)! \mod p$: $(p-1)! = 1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12$. and calculate the product modulo p step by step:

$$\begin{aligned}&\equiv(2\cdot 3\cdot 4\cdot 5\cdot 6\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(6\cdot 4\cdot 5\cdot 6\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(11\cdot 5\cdot 6\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(3\cdot 6\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(5\cdot 7\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(9\cdot 8\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(7\cdot 9\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(11\cdot 10\cdot 11\cdot 12) \mod 13\\ &\equiv(6\cdot 11\cdot 12) \mod 13\\ &\equiv(1\cdot 12) \mod 13 = -1 \mod 13\end{aligned}$$

There is no such Wilson formula for a cardinal double factorial, $p(n)\#\#$:

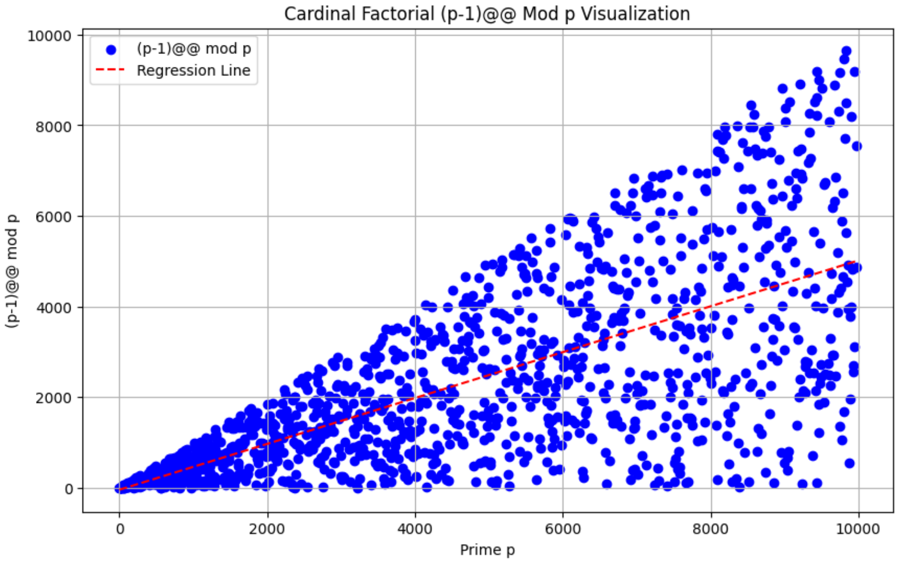


Figure 11.6: Lack of Wilson for $p(n)\#\#$

For instance, to compute $(11-1)\#\# \mod 11$, we identify the prime numbers leading up to and including the prime immediately preceding 11 (which is 7, the 4th prime). Now $(11-1)\#\# = 7\#\#$, which is given by multiplying every second prime number in reverse order: $7\#\# = 7\cdot 3 = 21$. Now, compute $21 \mod 11$ to show that $(11-1)\#\# \equiv 10 \mod 11$: $21 \mod 11 = 10$. The code snippet delivering this is *plotWilsonModular@@.ipynb*

```
def cardinal_factorial_with_lag(primes, p, lag):
    if len(primes) < lag or p not in primes or primes.index(p) < lag:
        return None
    lagged_index = primes.index(p) - lag
    return cardinal_factorial(primes, lagged_index)
```

The function `cardinal_factorial_with_lag` is designed to compute a variant of the factorial for a list of ordered prime numbers, incorporating a user-specified lag:

- **Input Parameters:**
  - `primes`: A list of prime numbers in ascending order.
  - `p`: A prime number whose position in `primes` is of interest.
  - `lag`: An integer indicating the number of positions before `p` to start the product calculation.
- **Functionality:**
  1. Validates that the `primes` list is sufficiently long, `p` is in `primes`, and `p`'s position is greater than `lag`. Returns `None` or handles errors if conditions are not met.
  2. Finds the index of the prime that is `lag` positions before `p` (denoted as `lagged_index`).
  3. Calls a function `cardinal_factorial`, passing `primes` and `lagged_index`, to compute the product of primes up to the prime at `lagged_index`.
  4. Returns the result from `cardinal_factorial`.

Gamma function

The Gamma function extends the concept of factorials to include non-integer values and is defined by the integral:

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt, \quad \text{Re}(z) > 0$$

For positive integers n, the Gamma function is related to the factorial function, satisfying:

$$\Gamma(n) = (n-1)!$$

The value of $\Gamma\left(\frac{3}{2}\right)$ is $\frac{1}{2}\sqrt{\pi}$. For $\left(\frac{1}{2}\right)!$:

$$\left(\frac{1}{2}\right)! = \Gamma\left(\frac{1}{2}+1\right) = \Gamma\left(\frac{3}{2}\right)$$

For $(-1/2)!$, we have:

$$(-1/2)! = \Gamma\left(-\frac{1}{2}+1\right) = \Gamma\left(\frac{1}{2}\right)$$

The value of $\Gamma\left(\frac{1}{2}\right)$ is known to be $\sqrt{\pi}$, therefore:

$$(-1/2)! = \sqrt{\pi}$$

We collate the relationships:

$$\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$$

$$\left(\frac{1}{2}\right)! = \Gamma\left(\frac{3}{2}\right)$$

$$\Gamma\left(\frac{3}{2}\right) = \left(\frac{1}{2}\right)\Gamma\left(\frac{1}{2}\right) = \frac{1}{2}\sqrt{\pi}$$

We relate $\Gamma(2)$, $\left(\frac{1}{2}\right)!$, and $\zeta(2)$ through π as follows:
1. Relation between $\left(\frac{1}{2}\right)!$ and π:

$$\left(\frac{1}{2}\right)! = \frac{1}{2}\sqrt{\pi}$$

2. Relation between $\Gamma(2)$ and π:

$$\Gamma(2) = \frac{3}{2}\Gamma\left(\frac{3}{2}\right) = \frac{3}{2}\cdot\frac{1}{2}\sqrt{\pi} = \frac{3\sqrt{\pi}}{4}$$

3. For the Basel problem solution $\zeta(2)$:

$$\zeta(2) = \frac{\pi^2}{6}$$

For $\left(\frac{1}{2}\right)!$:

$$\left(\frac{1}{2}\right)! = \Gamma\left(\frac{1}{2}+1\right) = \Gamma\left(\frac{3}{2}\right)$$

$$\Gamma\left(\frac{3}{2}\right) = \frac{1}{2}\Gamma\left(\frac{1}{2}\right) = \frac{\sqrt{\pi}}{2}$$

Interpolating between $\left(\frac{1}{2}\right)!$ and $\zeta(2)$, we observe the different ways in which π manifests in these two expressions:

$$\left(\frac{1}{2}\right)! = \frac{\sqrt{\pi}}{2} \quad \text{and} \quad \zeta(2) = \frac{\pi^2}{6}$$

11.2 Historical Significance of Harmonic Series and Logarithms

The harmonic series, $\sum_{n=1}^{\infty}\frac{1}{n}$ and the natural logarithm function, $\ln(n)$, have been central to mathematical inquiry since the dawn of calculus and number theory. Their interrelationship is perhaps most famously encapsulated in the study of the integral $\int \frac{1}{x}dx$, which directly leads to the logarithm function, bridging the gap between discrete and continuous mathematics:

$$\int \frac{1}{x}dx = \ln|x| + C$$

where C is the constant of integration defines the logarithm function but also highlights the deep connections between algebraic structures and geometric areas, laying the groundwork for much of modern analysis. The Euler-Mascheroni constant γ is defined as the limiting difference between the harmonic series and the natural logarithm of n:

$$\gamma = \lim_{n\to\infty}\left(-\ln(n) + \sum_{k=1}^{n}\frac{1}{k}\right) \approx 0.57721.$$

Discovered in the context of Euler's work on the Basel problem and the Riemann zeta function, γ intertwining discrete sums and continuous integrals. Similarly, the Meissel-Mertens constant M is defined as the limit of the difference between the sum of the reciprocals of primes and $\ln(\ln(n))$, M underscoring the nuanced distribution of prime numbers:

$$M = \lim_{n\to\infty}\left(-\ln(\ln(n)) + \sum_{p\leq n}\frac{1}{p}\right) \approx 0.261497$$

where p represents the prime numbers. The code, eulerMascerino.ipynb delivers both the limiting sums of the irrationals

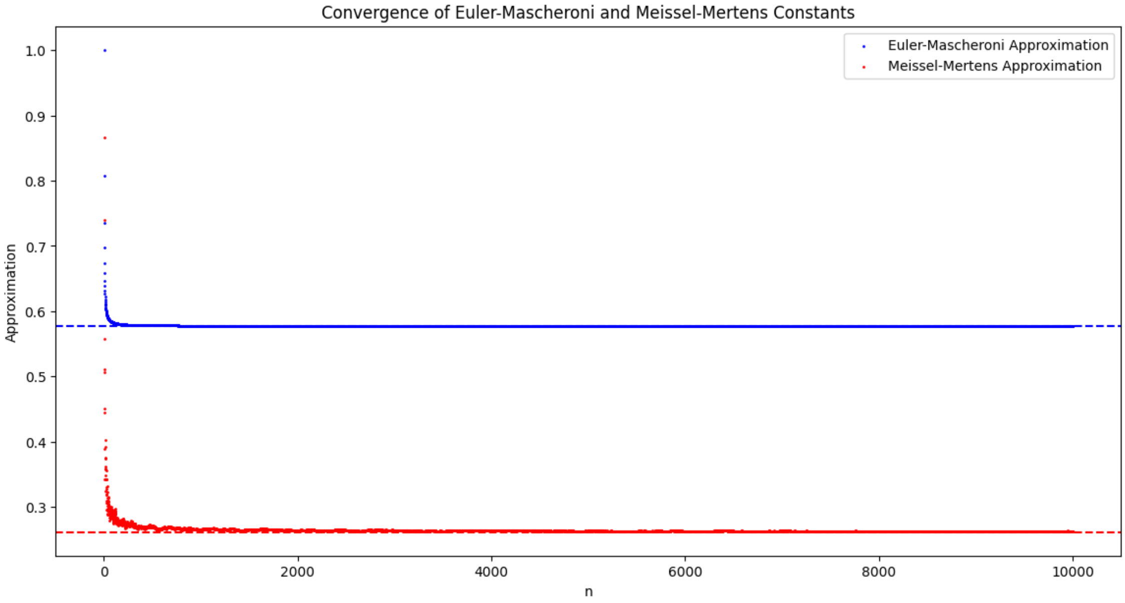


Figure 11.7: euler-Mascerino

11.2.1 Euler constant, e

The *reflection* formula for the Gamma function reads:

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(\pi z)}$$

Setting $z = \frac{1}{2}$, you get:

$$\Gamma\left(\frac{1}{2}\right)^2 = \pi$$

The harmonic numbers are defined as:

$$h(q) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{q}$$

The Euler-Mascheroni constant γ is defined as the limiting difference between the harmonic numbers and the natural logarithm:

$$\gamma = \lim_{q\to\infty}(h(q) - \ln(q))$$

The Riemann Zeta function is defined for $s > 1$ as:

$$\zeta(s) = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \ldots$$

There is a relationship between $\zeta(s)$ and γ involving the alternating zeta function, also known as the Dirichlet eta function $\eta(s)$, where:

$$\eta(s) = 1 - \frac{1}{2^s} + \frac{1}{3^s} - \dots$$

and $\eta(s)$ is related to $\zeta(s)$ by:

$$\eta(s) = (1 - 2^{1-s})\zeta(s)$$

The Euler-Mascheroni constant γ can be expressed in terms of an infinite series involving $\zeta(s)$ for integer values of s.

11.2.2 Double Factorial

For odd n:

$$n!! = 2^{\frac{n-1}{2}} \left(\frac{n}{2}\right)!$$

$$n!! = 2^{\frac{n-1}{2}} \Gamma\left(\frac{n}{2} + 1\right)$$

For even n:

$$n!! = \frac{n!}{\left(\frac{n}{2}\right)! 2^{n/2}}$$

$$n!! = \frac{\Gamma(n+1)}{\Gamma\left(\frac{n}{2} + 1\right) 2^{n/2}}$$

12. Irrational Analysis

While the familiar rationals and the enigmatic irrationals may seem to inhabit distinct realms, there exist profound connections that bridge them. One of the most significant of these connections is embodied in the Riemann Zeta function, a function that not only links rational numbers to irrationals but also intertwines them with the distribution of prime numbers which is defined for complex numbers s with a real part greater than 1 by the series

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

The Zeta function's deep connection with prime numbers is revealed by the Euler product formula, which expresses $\zeta(s)$ as an infinite product over primes p:

$$\zeta(s) = \prod_{p\,\text{prime}} \frac{1}{1-p^{-s}}.$$

12.0.1 Bernoulli

The values of $\zeta(s)$ at positive even integers are rational multiples of π to a power, demonstrating how a rational input leads to an output intimately tied to the irrational number π:

$$\zeta(2n) = (-1)^{n+1} \frac{B_{2n}(2\pi)^{2n}}{2(2n)!},$$

where B_{2n} represents the Bernoulli numbers.

The finite sum of even powers $1^k + 2^k + \ldots + n^k$ for an even integer k can be expressed using Bernoulli numbers B_j as follows:

$$\sum_{i=1}^{n} i^k = \frac{1}{k+1} \sum_{j=0}^{k} \binom{k+1}{j} B_j n^{k+1-j}$$

Meanwhile, the infinite sum of reciprocals of even powers is related to the Bernoulli numbers through the Riemann zeta function for even positive integers s:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s} = \frac{(-1)^{\frac{s}{2}-1}(2\pi)^s B_s}{2 \cdot s!}, \quad \text{for even } s > 1$$

These equations relate the finite sums of powers of integers with the infinite series of their reciprocals for even powers, using the Bernoulli numbers as interpolators. The Bernoulli numbers B_n are calculated using the following recursive formula, which is implemented in the provided Python code:

$$B_n = -\frac{1}{n+1} \sum_{k=0}^{n-1} \binom{n+1}{k} B_k, \quad \text{for } n \geq 1$$

where: - $B_0 = 1$ is initialized at the start of the list B. - The sum $\sum_{k=0}^{n-1} \binom{n+1}{k} B_k$ is computed within the nested `for` loop, iterating through all previously computed Bernoulli numbers ($B[k]$), multiplying each by the binomial coefficient $\binom{n+1}{k}$ calculated using `binomial(m + 1, k)`. - The variable Bm accumulates the sum of these terms. Initially, $Bm = 0$. - After computing the sum, Bm is updated to represent the n-th Bernoulli number by applying the formula $Bm = -Bm/(m+1)$, effectively isolating B_n on the left side of the equation. - This newly computed Bernoulli number B_n is then appended to the list B.

This process repeats for each n from 1 up to the specified limit, thereby generating a list of Bernoulli numbers.

The Python code snippet to compute Bernoulli numbers is as follows:

```
def compute_bernoulli_numbers(n):
    B = [Rational(1)]  # Initialize with B0 = 1
    for m in range(1, n):
        Bm = 0  # Initialize the sum
        for k in range(m):
            Bm += binomial(m + 1, k) * B[k]  # Calculate the sum based on previous Bernoull
        Bm = -Bm / (m + 1)  # Adjust for the Bernoulli number formula
        B.append(Bm)
    return B
```

12.0.2 polynomials

k=1: n^2 $\frac{}{2-\frac{n}{2}k=2:\frac{n^3}{3}-\frac{n^2}{2}+\frac{n}{6}k=3:\frac{n^4}{4}-\frac{n^3}{2}+\frac{n^2}{4}k=4:\frac{n^5}{5}-\frac{n^4}{2}+\frac{n^3}{3}-\frac{n}{30}k=5:\frac{n^6}{6}-\frac{n^5}{2}+\frac{5n^4}{12}-\frac{n^2}{12}k=6:\frac{n^7}{7}-\frac{n^6}{2}+\frac{n^5}{2}-\frac{n^3}{6}+\frac{n}{42}k=7:\frac{n^8}{8}-\frac{n^7}{2}+\frac{7n^6}{12}-\frac{7n^4}{24}+\frac{n^2}{12}k=8:\frac{n^9}{9}-}$

12.1 Euler, e as a continued fraction

Euler's irrational constant, $e \approx 2.71828\ldots$, is well-known for its natural occurrence in various fields such as complex analysis, and number theory. Its most elegant representation is surely its continued fraction form, which allows for successive approximations to any desired level of accuracy. The partial quotients of the continued fraction are separated from its integer part, $[e] = 2$, as: $e = [2;1,2,1,1,4,1,1,6,\ldots]$ where the sequence after the initial term of 2 follows a pattern in which every third term, starting from the second term, is an even integer that can be represented as $2k$, where k is a positive integer. All other terms in the sequence are 1.

$$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \cfrac{1}{1 + \ddots}}}}}}$$

In other words, for $e = [2;a_1,a_2,a_3,\ldots]$ the sequence a_i in the continued fraction are not a sequence of denominators but rather a sequence of operations leading to the next fraction.

We use the function `calculate_e(n_terms)` from continuedConvergente.ipynb to initialize the first two terms of the continued fraction and then iteratively append terms:

```
def calculate_e(n_terms):
    terms = [2, 1]
    # Calculate the terms of the continued fraction up to n_terms
    for n in range(2, n_terms):
        if n % 3 == 2:  # Every third term is 2k where k = 1,2,3,...
            terms.append(2 * (n // 3 + 1))
        else:  # All other terms are 1
            terms.append(1)
    # Start with the last term of the continued fraction
    e_approx = Decimal(terms.pop())
    # Calculate the convergents by iteratively updating e_approx
    while terms:
        e_approx = terms.pop() + Decimal(1)/e_approx
    return +e_approx
```

The construction of the convergents is from the innermost part of the fraction outwards. Starting from the end of the computed continued fraction formula sequence, the function nests each term within the fraction of the previous one and so for the sixth convergent it considers, the convergent sum opposite.

`calculate_e` computes the continued fraction representation of e taking one argument, `n_terms`, which specifies the number of terms to use and operates as follows:

1. The first two terms of the continued fraction are initialized in a list called `terms`.
2. A `for` loop calculates additional terms of the continued fraction based on a pattern and appends them to the `terms` list.
3. The `while` loop by checking for the non-emptiness of the `terms` list then iteratively computes the convergents from the last term to the first.

As long as there are terms left in the list, the loop continues to execute according to the pseudocode:

```
while terms is not empty:
    remove the last term from terms
    update the current approximation of e
```

Inside the loop, the last term is em popped from the list and added to the reciprocal of the current approximation, `e_approx`. Some more technical aspects in the code are the following:

- The unary plus operator, $+$ in the final return statement applies the current context precision to `e_approx` ensuring that the value of e is returned with the desired number of decimal places as specified by the precision setting in the Decimal context.
- `e_approx = Decimal(terms.pop())` initializes the approximation with the last term of the continued fraction. This is because the continued fraction is built backwards, starting from the last term and adding the reciprocal of the approximation to each preceding term. The method `pop()` is used to remove and return the last element of the list `terms` and is a standard operation in Python that modifies the list by removing the last item.

The code then generates a scatter plot that illustrates the convergence to e with an increasing number of partial quotients.

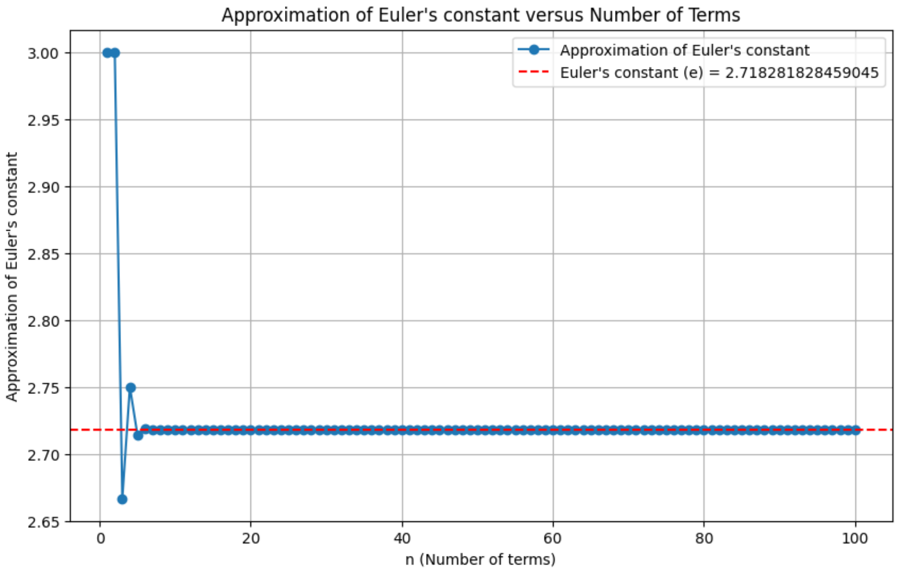


Figure 12.1: Euler's constant to increasing precision with

12.2 Pi

Below is an Ulam spiral of the 10,000 digits of pi starting in middle with 1. Highlighted is 999999, first run of six same consecutive numbers at 762nd position. $\pi \times 10^{762}$ is thus almost an integer being $\pi\times$ circumference Universe $\approx \pi^2 \times 10^{26}$m in 21 times subdivisions of Planck length, $(10^{-35}m$

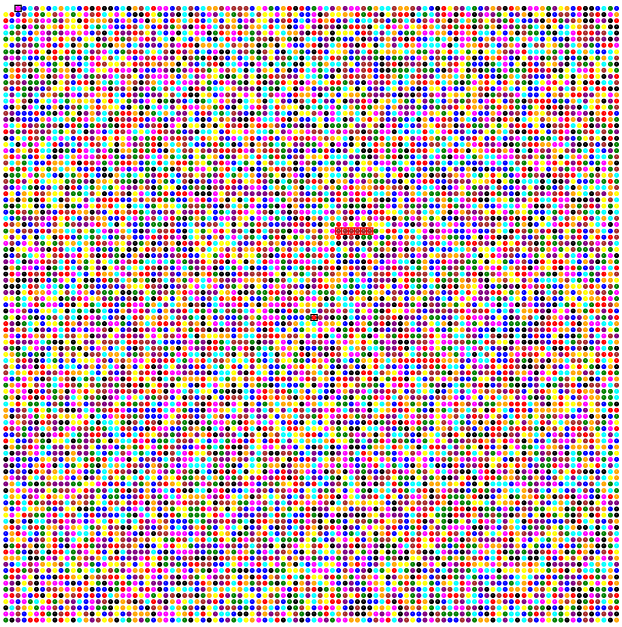


Figure 12.2: UlamSpiralPi

12.3 Continued fractions form of surds

Surds are reducible into integer multiples of prime surd parts allowing us to represent a composite surd as a product of its prime surds, $\sqrt{24} = \sqrt{4} \times \sqrt{6} = 2\sqrt{2}\sqrt{3}$. Given that any prime p may be decomposed into a square and residual part, $p = a^2 + b$ its continued fractions (written in the compact form, $\sqrt{p} = [a; b, b, b, \ldots])$ provide an alternative representation for surds:

$$\sqrt{p} = \sqrt{a^2+b} = a + \cfrac{b}{2a + \cfrac{b}{2a + \cfrac{b}{2a + \cdots}}}. \tag{12.1}$$

So for $p = 2 = 1^2 + 1$ we have a continued fraction representation with five terms of:

$$\sqrt{2} = 1 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}} = 1.4142135623730951$$

Similarly for $s = \sqrt{164} = 2\sqrt{41}$ with ,$p = 41 = 6^2 + 5$ we have

$$\sqrt{164} = 2 \times \{6 + \frac{5}{12 + \frac{5}{12 + \frac{5}{12 + \frac{5}{12}}}}\} = 2 \times 6.4031242374328485.$$

The code, continuedFractions-Latex.ipynb lists the primes with squares removed and their complements as per the table below.

n	$4n+1$	a	b	$b \mod a$	$4n+3$	a	b	$b \mod a$
1	5	2	1	1	7	2	3	1
2	-	-	-	-	11	3	2	2
3	13	3	4	1	-	-	-	-
4	17	4	1	1	19	4	3	3
5	-	-	-	-	23	4	7	3

Table 12.1: The remainder of the first seven primes mod their largest square component

This is produced in 4n+14n+3a2+b.ipynb with the following essential snippet

```
def find_ab_for_prime(p):
        # Start with the largest integer less than sqrt(p)
    a = int(math.sqrt(p))
    while a > 0:
        b = p - a**2
        if b > 0:
            return a, b
        a -= 1
    return None, None
```

The latex code for the continued fractions are produced in the following snippet

```
def continued_fraction_representation(a, b, max_terms):
    terms = [a] + [2*a] * (max_terms-1)
    # Convert the terms to LaTeX format
    latex_terms = str(terms[0]) + "+\\frac{" + str(b)
    for term in terms[1:]:
        latex_terms += "}{" + str(term) + "+\\frac{" + str(b)
    latex_terms = latex_terms.rstrip("+\\frac{" + str(b))
    latex_terms += "}" * (max_terms - 1)
    return "\[\\sqrt{" + str(a**2 + b) + "} = " + latex_terms + "\]"
```

scatterplotbmodaPrimes.ipynb delivers the following scatterplot of b mod a versus p

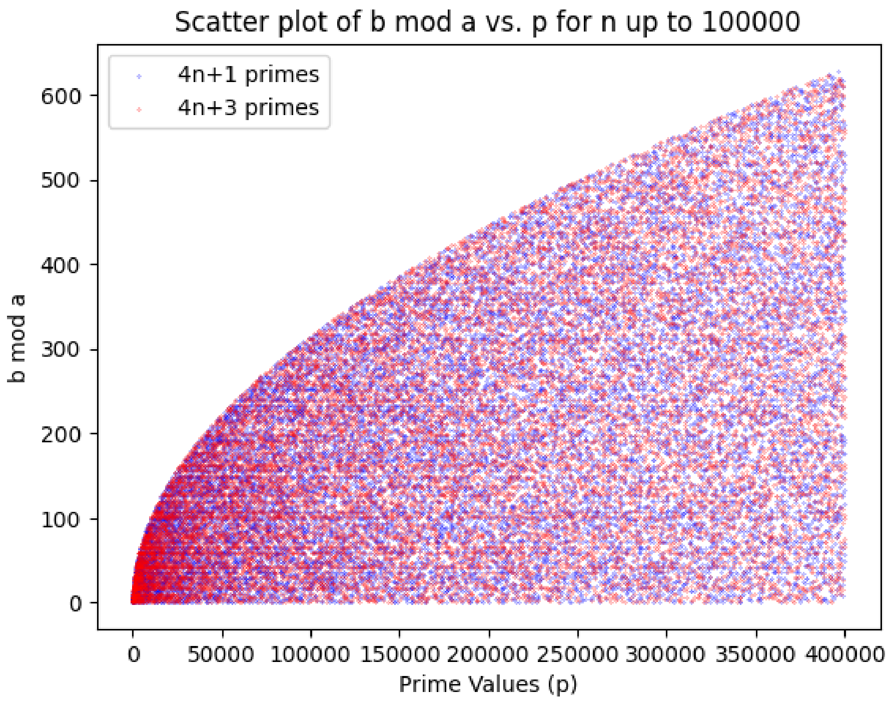


Figure 12.3: b mod a versus $p = a^2 + b$.

```
def scatter_plot_data(limit_n):
rows = prime_table_upto_n(limit_n)
p_values_4n1 = []
remainders_4n1 = []
p_values_4n3 = []
remainders_4n3 = []
for row in rows:
    if row[1] != "-":
        p_values_4n1.append(row[1])
        remainders_4n1.append(row[4])
    if row[5] != "-":
        p_values_4n3.append(row[5])
        remainders_4n3.append(row[8])
return p_values_4n1, remainders_4n1, p_values_4n3, remainders_4n3
```

12.3.1 e and π as Continued Fractions

$$e = 2 + \cfrac{1}{2\times 1 + \cfrac{1}{2\times 2 + \cfrac{1}{2\times 3 + \cfrac{1}{2\times 4 + \cfrac{1}{2\times 5 + \cfrac{1}{2\times 6 + \cfrac{1}{2\times 7 + \cfrac{1}{2\times 8 + 1}}}}}}}}$$

The receding sum representation in LaTeX is:

$$e = [2; 1, 2\times 1, 1, 2\times 2, 1, 2\times 3, 1, 2\times 4, 1, 2\times 5, 1, 2\times 6, 1, 2\times 7, 1, 2\times 8, 1]$$

$$\pi = 3 + \cfrac{1}{6 + \cfrac{1}{6 + \cfrac{1}{6 + \cfrac{1}{6 + \cfrac{1}{6 + \cfrac{1}{6 + \frac{1}{6}6}6}6}6}6}6}$$

The receding sum representation in LaTeX is:

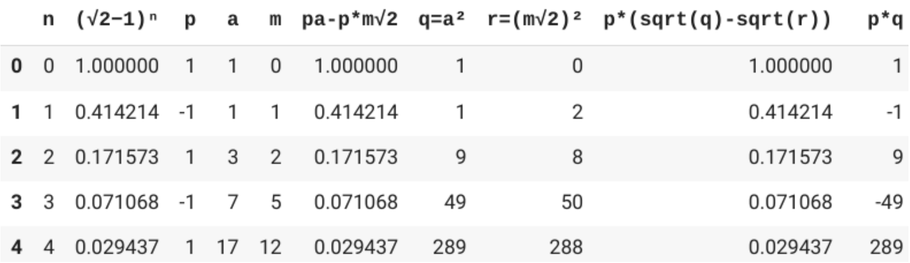
$$\pi = [3; 6, 6, 6, 6, 6, 6, 6, 6, 6]$$

12.4 Square root difference of squares

Our focus here is to consider the following sequence,

$$\begin{aligned}(\sqrt{2}-1)^2 &= \sqrt{9}-\sqrt{8} = \delta_S'^2, \\ (\sqrt{2}-1)^3 &= \sqrt{50}-\sqrt{49} = -\delta_S'^3, \\ (\sqrt{2}-1)^4 &= \sqrt{289}-\sqrt{288} = \delta_S'^4,\end{aligned} \tag{12.2}$$

in which we have defined the silver ratio $\delta_S \equiv 1+\sqrt{2}$ and its conjugate, $\delta_S' \equiv 1-\sqrt{2}$ and ask how we might go about determining the recursion relations for such a sequence. Our panda database is initialized in DiophantineSquareSilverRatio.ipynb with calculated values for $(\sqrt{2}-1)^n$ for $n=0$ through $n=4$, based on using the coefficients of Pascal triangle according to the binomial theorem: and looks like the



	n	(√2−1)ⁿ	p	a	m	pa-p*m√2	q=a²	r=(m√2)²	p*(sqrt(q)-sqrt(r))	p*q
0	0	1.000000	1	1	0	1.000000	1	0	1.000000	1
1	1	0.414214	-1	1	1	0.414214	1	2	0.414214	-1
2	2	0.171573	1	3	2	0.171573	9	8	0.171573	9
3	3	0.071068	-1	7	5	0.071068	49	50	0.071068	-49
4	4	0.029437	1	17	12	0.029437	289	288	0.029437	289

Figure 12.4: panda data frame with breakdown of powers of conjugate silver ratio

For $n=2$, the binomial expansion is:

$$(\sqrt{2}-1)^2 = (\sqrt{2})^2 + 2(\sqrt{2})(-1) + (-1)^2 = 2 - 2\sqrt{2} + 1 = 3 - 2\sqrt{2}$$

For $n=3$, the binomial expansion is:

$$\begin{aligned}(\sqrt{2}-1)^3 &= (\sqrt{2})^3 - 3(\sqrt{2})^2(1) + 3(\sqrt{2})(-1)^2 - (-1)^3 \\ &= 2\sqrt{2} - 6 + 3\sqrt{2} - 1 = -7 + 5\sqrt{2}\end{aligned}$$

For $n=4$, the binomial expansion is:

$$\begin{aligned}(\sqrt{2}-1)^4 &= (\sqrt{2})^4 - 4(\sqrt{2})^3(1) + 6(\sqrt{2})^2(-1)^2 - 4(\sqrt{2})(-1)^3 + (-1)^4 \\ &= 4 - 8\sqrt{2} + 12 - 4\sqrt{2} + 1 = 17 - 12\sqrt{2}\end{aligned}$$

12.4.1 Solving Recursive Relations via Linear Algebra

DiophantineSquareSilverRatio.ipynb gathers these initial calculations in a panda dataframe and extends them using implied intertwining recursive relations recast from its contemporaneous form:

$$\begin{aligned} a_n &= m_n + a_{n-1} & a_n &= a_{n-1} + 2m_{n-1}, \\ m_n &= m_{n-1} + a_{n-1} & m_n &= m_{n-1} + a_{n-1}, \end{aligned} \tag{12.3}$$

so as to be written in matrix form whose solutions for λ in the determinant equation are:

$$\begin{pmatrix} a_n \\ m_n \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a_{n-1} \\ m_{n-1} \end{pmatrix}. \qquad \det\left[\begin{pmatrix} 1-\lambda & 2 \\ 1 & 1-\lambda \end{pmatrix}\right] = 0,$$

This determinant and its simplification read as:

$$(1-\lambda)(1-\lambda)-2(1)=0, \qquad \lambda^2-2\lambda-1=0,$$

The eigenvalues are determined by solving this characteristic equation using symbolic computation:

```
from sympy import symbols, Eq, solve, sqrt, N
lambda_ = symbols('lambda')
char_eq = Eq(lambda_**2 - 2*lambda_ - 1, 0)
eigenvalues = solve(char_eq, lambda_)
```

or simply by completing the square

$$(\lambda-1)^2=2,$$
$$\lambda_{1,2}=1\pm\sqrt{2},$$

corresponding to the silver ratio $\delta_S \equiv \sqrt{2}+1$ and negative its conjugate, $-\delta_S' \equiv -(\sqrt{2}-1)$.

General Solution for a_n and m_n

Given any sequence $\{a_n\}$ that follows specific recursive relations, we can express its general solution in terms of the sequence's eigenvalues. For eigenvalues of the matrix representation of the recursive relations, λ_1 and λ_2, the general solution for the sequence a can be written as:

$$a_n = c_1 \cdot \lambda_1^n + c_2 \cdot \lambda_2^n,$$

where c_1 and c_2 are constants determined by the initial conditions of the sequence. Given our initial conditions $a_0 = 1$ and $a_1 = 1$, we establish a system of equations to solve for c_1 and c_2:

$$\begin{aligned} a_0 = 1: &\quad 1 = c_1\cdot(1+\sqrt{2})^0 + c_2\cdot(1-\sqrt{2})^0 = c_1+c_2, \\ a_1 = 1: &\quad 1 = c_1\cdot(1+\sqrt{2})^1 + c_2\cdot(1-\sqrt{2})^1 = (c_1+c_2)+\sqrt{2}(c_1-c_2) \end{aligned}$$

Similarly for the sequence m_n, given the initial conditions $m_0 = 0$ and $m_1 = 1$, we solve for constants, d_1 and d_2, that fit a similar general solution, $m_n = d_1\cdot\lambda_1^n + d_2\cdot\lambda_2^n$, form:

$$\begin{aligned} m_0 = 0: &\quad 0 = d_1\cdot(1+\sqrt{2})^0 + d_2\cdot(1-\sqrt{2})^0 = d_1+d_2, \\ m_1 = 1: &\quad 1 = d_1\cdot(1+\sqrt{2})^1 + d_2\cdot(1-\sqrt{2})^1 = (d_1+d_2)+\sqrt{2}(d_1-d_2). \end{aligned}$$

These can be solved by equating rational and irrational parts or if that algebra is a stretch you can compute these c_1, c_2, d_1, d_2 with generatingFunctionCharacteristic.ipynb:

```
c1, c2, d1, d2 = symbols('c1 c2 d1 d2')
eq1_a = Eq(c1 + c2, 1)
eq2_a = Eq(c1 * (1+sqrt(2)) + c2 * (1-sqrt(2)), 1)
eq1_m = Eq(d1 + d2, 0)
eq2_m = Eq(d1 * (1+sqrt(2)) + d2 * (1-sqrt(2)), 1)
solutions_a = solve((eq1_a, eq2_a), (c1, c2))
solutions_m = solve((eq1_m, eq2_m), (d1, d2))
```

which yields: ((c1: 1/2, c2: 1/2, d1: sqrt(2)/4, d2: -sqrt(2)/4)). That is:

$$c_1 = \frac{1}{2}, \quad c_2 = \frac{1}{2}, \quad d_1 = \frac{\sqrt{2}}{4}, \quad d_2 = -\frac{\sqrt{2}}{4}.$$

So that the general solutions for the sequences $\{a_n\}$ and $\{m_n\}$ are:

$$a_n = \frac{1}{2}\left(1+\sqrt{2}\right)^n + \left(\frac{1}{2}\right)\cdot\left(1-\sqrt{2}\right)^n,$$

$$m_n = \frac{\sqrt{2}}{4}\cdot(1+\sqrt{2})^n - \frac{\sqrt{2}}{4}\cdot(1-\sqrt{2})^n.$$

For which to verify we note for $n = 2$:

$$a_2 = \frac{1}{2}\left(1+\sqrt{2}\right)^2 + \left(\frac{1}{2}\right)\cdot\left(1-\sqrt{2}\right)^2 = 3,$$

$$m_2 = \frac{\sqrt{2}}{4}\cdot(1+\sqrt{2})^2 - \frac{\sqrt{2}}{4}\cdot(1-\sqrt{2})^2 = 2,$$

which the snippet from generatingFunctionCharacteristic.ipynb with n = 2 confirms:

```
# Define the symbol for lambda and the characteristic equation
lambda_ = symbols(’lambda’)
char_eq = Eq(lambda_**2 - 2*lambda_ - 1, 0)

# Solve the characteristic equation for eigenvalues
eigenvalues = solve(char_eq, lambda_)
# Define symbols for constants c1, c2, d1, d2
c1, c2, d1, d2 = symbols(’c1 c2 d1 d2’)

# Equations for a_n and m_n based on given conditions
eq1_a = Eq(c1 + c2, 1)
eq2_a = Eq(c1 * eigenvalues[0] + c2 * eigenvalues[1], 1)
eq1_m = Eq(d1 + d2, 0)
eq2_m = Eq(d1 * eigenvalues[0] + d2 * eigenvalues[1], 1)

# Solve the systems for a_n and m_n
solutions_a = solve((eq1_a, eq2_a), (c1, c2))
solutions_m = solve((eq1_m, eq2_m), (d1, d2))

def calculate_sequences(n, solutions_a, solutions_m, eigenvalues):
    # Extract solved values for c1, c2, d1, d2
    c1_val, c2_val = solutions_a[c1], solutions_a[c2]
    d1_val, d2_val = solutions_m[d1], solutions_m[d2]
    lambda_1, lambda_2 = eigenvalues
    # Calculating numeric values using a_n and m_n  constants, eigenvalues
    a_n_formula = c1_val * lambda_1**n + c2_val * lambda_2**n
    m_n_formula = d1_val * lambda_1**n + d2_val * lambda_2**n
    a_n_value_numeric = N(a_n_formula)
    m_n_value_numeric = N(m_n_formula)

    return a_n_value_numeric, m_n_value_numeric
```

Our matrix representation of the recursive relation highlights the interconnectedness of algebra, geometry, and number theory.

Noting now instead our original assertion, eq. **??**, consider for$P_{n+1} = P_n + 1$ that

$$(\sqrt{2}-1)^n = \sqrt{P_{n+1}} - \sqrt{P_n},$$

we have that

$$
\begin{aligned}
(\sqrt{2}-1)^n(\sqrt{2}-1)^n &= (\sqrt{P_{n+1}}-\sqrt{P_n})(\sqrt{P_{n+1}}-\sqrt{P_n}),\\
&= (\sqrt{P_{n+1}}\sqrt{P_{n+1}}) + \sqrt{P_n}\sqrt{P_n} - 2\sqrt{P_n}\sqrt{P_{n+1}},\\
\text{so that } (\sqrt{2}-1)^{2n} &= 2P_n + 1 - 2\sqrt{P_n}\sqrt{P_{n+1}},
\end{aligned}
$$

Rearranging we have then that,

$$\sqrt{P_n}\sqrt{P_{n+1}} = \frac{1}{2}\left[(1+2P_n)+(1-\sqrt{2})^{2n}\right], \tag{12.4}$$

Multiplying through by 2 and squaring this reads as

$$4P_n(P_n+1) = (1+2P_n)^2 + (1-\sqrt{2})^{4n} + 2(1+2P_n)(1-\sqrt{2})^{2n}, \tag{12.5}$$

Expanding that is,

$$4P_n^2 + 4P_n = 1 + 4P_n^2 + 4P_n + (1-\sqrt{2})^{4n} + 2(1+2P_n)(1-\sqrt{2})^{2n}, \tag{12.6}$$

Cancelling terms we have then that

$$1 = (\sqrt{2}-1)^{4n} + 2(1+2P_n)(\sqrt{2}-1)^{2n}, \tag{12.7}$$

Defining $\delta_S' \equiv (\sqrt{2}-1)$ means our formula reads:

$$
\begin{aligned}
1 &= \delta_S'^{2n} + 2\delta_S'^{n} + 4P_n\delta_S'^{n},\\
P_n &= \frac{1}{4\delta_S'^{n}}\left(1-\delta_S'^{2n}-2\delta_S'^{n}\right).
\end{aligned}
$$

The series discussed here is identified in OEIS (Online Encyclopedia of Integer Sequences) under the entry A001108. Our code delivers the following scatterplot:

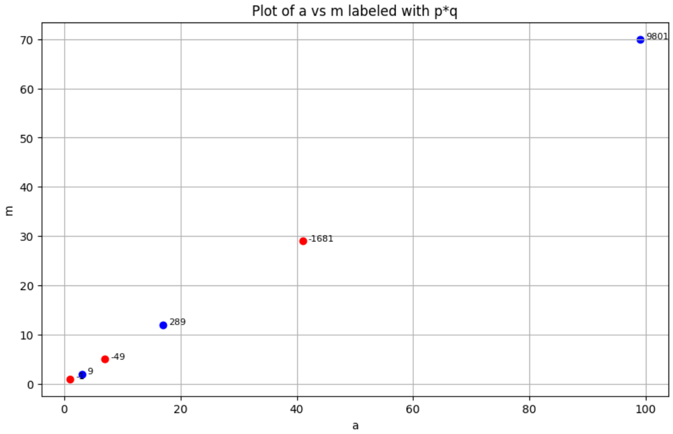


Figure 12.5: Intertwining coefficient sequence of difference of sequential square roots

Each point in the plot is labeled by p × q and color-coded based on the sign of p × q (red for negative, blue for positive) reflecting the oscillating parity that arises from the recursion. To ensure integer representation for p × q explicit formatting of the labels as integers is required in code.

12.5 Quadratic generator of silver ratio powers

The metallic numbers are a set of irrationals that are represented by the general quadratic equation

$$\lambda^2 - q\lambda - 1 = 0,$$

where q is a positive integer whose solutions, the metallic ratios, are given by

$$\frac{q+\sqrt{q^2+4}}{2} \quad \text{and} \quad \frac{q-\sqrt{q^2+4}}{2}.$$

The most well-known metallic numbers include:

- The **Golden Ratio** (ϕ): $q = 1$, yielding $\lambda^2 - \lambda - 1 = 0$ with positive solution

  $$\phi = \frac{1+\sqrt{5}}{2}.$$

- The **Silver Ratio** (δ_s): $q = 2$, yielding $\lambda^2 - 2\lambda - 1 = 0$ with positive solution

  $$\delta_s = 1+\sqrt{2}.$$

- The **Bronze Ratio** (δ_b): $q = 3$, yielding $\lambda^2 - 3\lambda - 1 = 0$ with positive solution

  $$\delta_b = \frac{3+\sqrt{13}}{2}.$$

The quadratic equations that describe δ_s^n and $\delta_s'^n$. We note for example:

- δ_s^1 arises from $\lambda^2 - 2\lambda - 1 = 0$
- $\delta_s^2 = (\sqrt{2}+1)^2$, we find $\delta_s^2 = \sqrt{2}+2\sqrt{2}+1 = 3+2\sqrt{2}$ with corresponding quadratic equation derived from $[\lambda - (3+2\sqrt{2})][\lambda - (3-2\sqrt{2})] = \lambda^2 - 6\lambda + 1$.
- δ_s^3, $\delta_s^3 = (\sqrt{2}+1)^3$ simplifies to $7+5\sqrt{2}$ to yield the quadratic equation $\lambda^2 - 14\lambda - 1$,
- δ_s^4 results in $\lambda^2 - 34\lambda + 1 = 0$.

We note the constant term c alternating between $+1$ and -1, and the coefficient $b = 2a$.

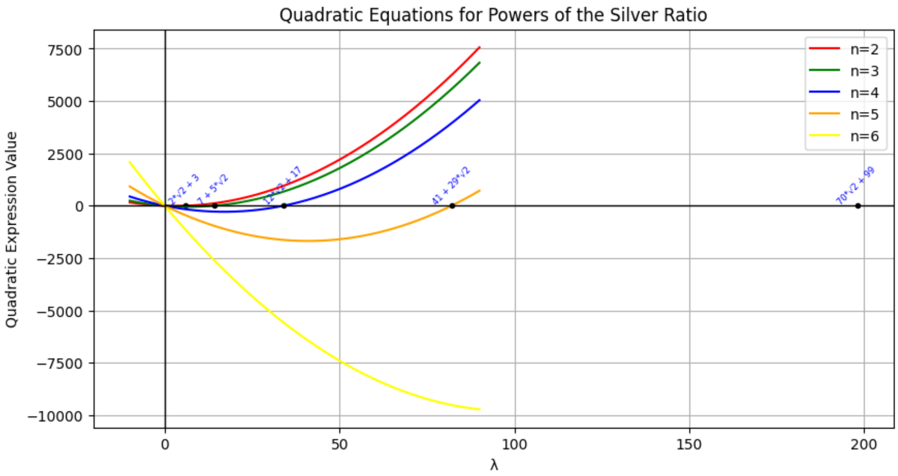


Figure 12.6: Quadratics that generate powers of silver ratio

Our goal is to generalize this process for higher powers of δ_s, identifying the coefficients of the resulting quadratic equations using symbolic computation to validate the hypothesis that $b = 2a$.

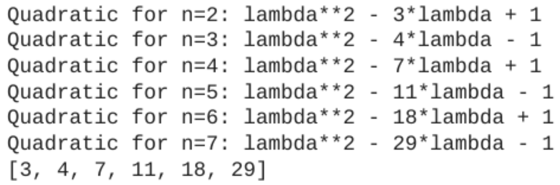
12.6 Golden Ratio Quadratic Coefficients as Lucas Numbers

The powers of the golden ratio and their conjugates can similarly form a basis for generating a sequence of quadratic expressions with intriguing numerical properties. By using the symbolic computation library SymPy, we define the symbolic variable λ and employ the `symbols`, `expand`, `sqrt`, and `Poly` functions to construct and analyze the quadratic equations.

```
from sympy import symbols, expand, sqrt, Poly
lambd = symbols('lambda')

def generate_golden_quadratic(n):
    phi = (1 + sqrt(5)) / 2
    phi_conjugate = (1 - sqrt(5)) / 2
    phi_power = phi**n
    conjugate_power = phi_conjugate**n
    quadratic = expand((lambd - phi_power) * (lambd - conjugate_power))
    return quadratic

# Generate quadratic equations and extract the coefficients
coefficients_list = []
for n in range(2, 12):
    quadratic = generate_golden_quadratic(n)
    poly = Poly(quadratic, lambd)
    coefficients = -poly.all_coeffs()[1]  # Coefficient of lambd
    coefficients_list.append(coefficients)
    print(f"Quadratic for n={n}: {quadratic}")
```

The core of our procedure is the function `generate_golden_quadratic`, which computes the n-th power of the golden ratio and its conjugate. These powers are then used to form quadratic equations by expanding the product $(\lambda - \phi^n)(\lambda - \phi'^n)$, for ϕ' the conjugate of the golden ratio.

Upon generating the quadratic equations for powers $n = 2$ to $n = 11$, our code extracts the coefficients of the λ term which are $[3, 4, 7, 11, 18, 29, 47, 76, 123, 199]$:



```
Quadratic for n=2: lambda**2 - 3*lambda + 1
Quadratic for n=3: lambda**2 - 4*lambda - 1
Quadratic for n=4: lambda**2 - 7*lambda + 1
Quadratic for n=5: lambda**2 - 11*lambda - 1
Quadratic for n=6: lambda**2 - 18*lambda + 1
Quadratic for n=7: lambda**2 - 29*lambda - 1
[3, 4, 7, 11, 18, 29]
```

Figure 12.7: Quadratics that generate powers of golden ratio

The Lucas sequence (L_n) is defined by the recurrence relation $L_n = L_{n-1} + L_{n-2}$ with initial terms $L_0 = 2$ and $L_1 = 1$ and then $2, 1, 3, 4, 7, 11, 18, 29, \ldots$. Binet provides an explicit expression for the n-th term of the Lucas sequence $L_n = \phi^n + (1-\phi)^n$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, and $1 - \phi$ is its conjugate which comparing to our derived quadratic expressions means we can assert:

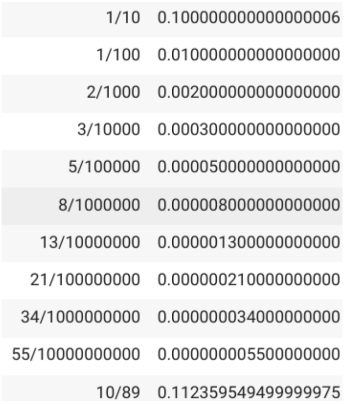
$$\lambda^2 - (2a_n)\lambda - 1 = 0,$$

in which the coefficients a_n of the linear follow the Lucas sequence with $a_n = \frac{L_n}{2}$.

12.7 Metallic Rationals

R When you are next idly asked by an astro-planing believer for your favourite number, request for clarification: rational or irrational. For rational why would you not offer up 10/89?

Our objective here is to deconstruct a tweet by @potetoichiro that suggests a bit of spooky numerology in which the Fibonacci series form the lead numbers at each placeholder for the partial sums of the rational 10/89 suggesting other metal reside near by:



1/10	0.100000000000000006
1/100	0.010000000000000000
2/1000	0.002000000000000000
3/10000	0.000300000000000000
5/100000	0.000050000000000000
8/1000000	0.000008000000000000
13/10000000	0.000001300000000000
21/100000000	0.000000210000000000
34/1000000000	0.000000034000000000
55/10000000000	0.000000005500000000
10/89	0.112359549499999975

Figure 12.8: The golden rational of 10/89

The snippet from generic10m9FibonacciGeneration.ipynb generates the partial sum sequences for metal ratios and the following violin graph

```
def generate_metallic_sequence(n, q, initial_values):
    sequence = initial_values.copy()
    while len(sequence) < n:
        sequence.append(sequence[-2] + q * sequence[-1])
    return sequence
n_terms = 10  # Number of terms to generate
golden_sequence = generate_metallic_sequence(n_terms, 1, [1, 1])
```

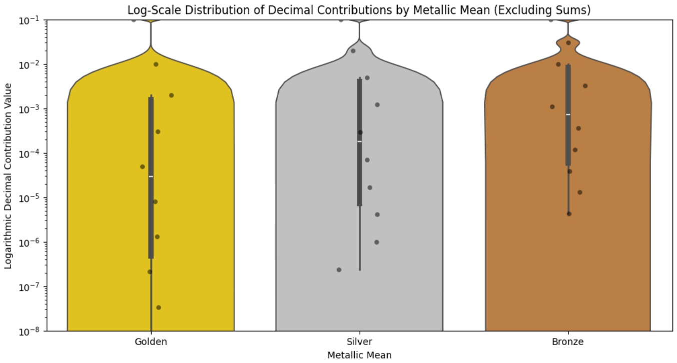


Figure 12.9: The place contributions of the first three metallic rationals

The violin plot provides a composite visualization, combining aspects of a box plot with a kernel density estimation. The width of each violin is proportional to the density of data points at each level of the "Decimal Equivalent", which allows for the immediate assessment of the distribution's modality and variance. The overlay of individual data points, plotted as a strip plot, gives insight into the actual distribution of values within each metallic mean category, exposing the concentration of data and potential outliers. Notably, their absences of data points in certain regions of the violins.

This implementation is initiated by defining a color palette within the Python environment, which maps the categorical data of metallic means to a corresponding color as the palette is instantiated as a dictionary with keys 'Golden', 'Silver', and 'Bronze', associated with the colors 'gold', 'silver', and a hexadecimal color code representing bronze, respectively. `Seaborn`, a statistical data visualization library built on top of `matplotlib`, is utilized to construct the violin plot. The `violinplot` function is invoked with parameters to plot the 'Metal' category on the x-axis and the 'Decimal Equivalent' on the y-axis. The data source is specified as the filtered `DataFrame` `df_all_filtered`, with the plot's scale and bandwidth fine-tuned through the 'scale' and 'bw' arguments, respectively. In concert with the violin plot, a `stripplot` is generated to overlay the individual data points, providing granular visibility into the dataset's distribution. The color 'k' (black) is selected for the points, with a semi-transparent alpha value and jitter applied to enhance clarity and avoid overlapping. Subsequent to the plot's creation, `matplotlib`'s `yscale` function is employed to transmute the y-axis into a logarithmic scale, effectively compressing the extensive range of the data into a more interpretable form. The axis limits are explicitly set to span from 10^{-8} to 10^{-1}, encompassing the full breadth of the dataset's magnitude.

Connection between Fibonacci Generating Function and Binet's Formula

A generating function is a power series in which the coefficients of the terms encode information about a sequence of numbers, $\{\alpha_n\}$ as:

$$G(\lambda) = \sum_{n=0}^{\infty} \alpha_n \lambda^n$$

The recurrence relation $F_n = F_{n-1} + F_{n-2}$, with initial conditions $F_0 = 0$ and $F_1 = 1$ has a generating function encapsulating its Fibonacci sequence within a finite expression:

$$G(\lambda) = \frac{\lambda}{1-\lambda-\lambda^2} \tag{12.8}$$

whose denominator can (as we have seen) be factored using the roots, ϕ and its conjugate $\psi = \frac{1-\sqrt{5}}{2}$ of the characteristic equation associated with the Fibonacci sequence: those solutions to the equation $\lambda^2 - \lambda - 1 = 0$. We can express $G(\lambda)$ in terms of partial fractions explicitly involving ϕ and ψ:

$$G(\lambda) = \frac{\lambda}{(1-\phi\lambda)(1-\psi\lambda)} = \frac{A}{1-\phi\lambda} + \frac{B}{1-\psi\lambda},$$

in which A and B are found by multiplying both sides by the denominator on the left side to give:

$$\lambda = A(1-\psi\lambda) + B(1-\phi\lambda).$$

Upon setting $\lambda = \frac{1}{\phi}$ this reads (since $\phi - \psi = \sqrt{5}$) as $\frac{1}{\phi} = A\left(1 - \frac{\psi}{\phi}\right) \Rightarrow A = \frac{1}{\sqrt{5}}$.
Similarly, setting $\lambda = \frac{1}{\psi}$ yields $B = -\frac{1}{\sqrt{5}}$ gives our partial fraction as:

$$G(\lambda) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi\lambda} - \frac{1}{1-\psi\lambda}\right). \tag{12.9}$$

Expanding each term into a geometric series using the expansion formula:

$$\frac{1}{1-r\lambda}=\sum_{n=0}^{\infty}(r\lambda)^n. \quad \text{we have:} \quad \frac{1}{1-\phi\lambda}=\sum_{n=0}^{\infty}(\phi\lambda)^n, \quad \frac{1}{1-\psi\lambda}=\sum_{n=0}^{\infty}(\psi\lambda)^n.$$

Substituting these expansions back into the expression, (12.9) gives us:

$$G(\lambda)=\frac{1}{\sqrt{5}}\left(\sum_{n=0}^{\infty}(\phi\lambda)^n-\sum_{n=0}^{\infty}(\psi\lambda)^n\right).$$

Now since $\phi\lambda$ and $\psi\lambda$ are both raised to the power of n in their respective series, we combine them in a power series :

$$G(\lambda)=\frac{1}{\sqrt{5}}\sum_{n=0}^{\infty}(\phi^n-\psi^n)\,\lambda^n\equiv\sum_{n=0}^{\infty}F_n\lambda^n.$$

in which the coefficient of λ^n is the nth Fibonacci number, represented by *Binet's* formula:

$$F_n=\frac{\phi^n-\psi^n}{\sqrt{5}}.$$

Substituting $\lambda=\frac{1}{10}$ into (12.8), we obtain:

$$G\left(\frac{1}{10}\right)=\frac{\frac{1}{10}}{1-\frac{1}{10}-\left(\frac{1}{10}\right)^2}=\frac{\frac{1}{10}}{1-\frac{1}{10}-\frac{1}{100}}=\frac{\frac{1}{10}}{\frac{100}{100}-\frac{10}{100}-\frac{1}{100}}=\frac{\frac{1}{10}}{\frac{89}{100}}=\frac{10}{89}.$$

Exercise 12.1 The reader is invited to discover the rational number that embodies the Tribonacci series, by adpating the solver below. ■

```
from sympy import symbols, solve, Eq
x = symbols('x')
equation = Eq(x / (1 - x - 1*x**2), 10/89)
solution = solve(equation, x)
solution
```

Sequence Types and Their Generating Functions

For convenience we summarize the recurrence relations and generating functions for the Silver and Bronze Ratios, as well as for the Tribonacci and Lucas Numbers, in the following table:

For the silver ratio, substituting $\lambda=\frac{1}{10}$ into its generating function:

$$G_S\left(\frac{1}{10}\right)=\frac{1/10}{1-1/10-2(1/10)^2}$$

yields a decimal expansion that aligns with the silver ratio sequence. Although not as straightforward as the Fibonacci sequence, it similarly results in a patterned expansion reflective of its sequence.

Similarly, for the bronze ratio, substituting $\lambda=\frac{1}{10}$ into its generating function:

$$G_B\left(\frac{1}{10}\right)=\frac{1/10}{1-1/10-3(1/10)^2}$$

Sequence Type	**Recurrence Relation**	**Generating Function**
Golden Ratio	$F_n = F_{n-1} + F_{n-2}; F_0 = 0, F_1 = 1$	$G_F(\lambda) = \frac{\lambda}{1-\lambda-\lambda^2}$
Silver Ratio	$a_n = a_{n-1} + 2a_{n-2}$	$G_S(\lambda) = \frac{\lambda}{1-\lambda-2\lambda^2}$
Bronze Ratio	$a_n = a_{n-1} + 3a_{n-2}$	$G_B(\lambda) = \frac{\lambda}{1-\lambda-3\lambda^2}$
Tribonacci	$T_n = T_{n-1} + T_{n-2} + T_{n-3}$	$G_T(\lambda) = \frac{\lambda}{1-\lambda-\lambda^2-\lambda^3}$
Lucas Numbers	$L_n = L_{n-1} + L_{n-2}; L_0 = 2, L_1 = 1$	$G_L(\lambda) = \frac{2-\lambda}{1-\lambda-\lambda^2}$

Table 12.2: Sequence Types by Recurrence Relations and Generating Functions

results in a decimal expansion that reflects the bronze ratio sequence. Like the silver ratio, the bronze ratio sequence creates a unique pattern in its decimal expansion.

Note: The actual numeric decimal expansions for the silver and bronze ratios would require specific calculations based on their generating functions. The sequences generated by these ratios, while not directly translating to simple decimal representations as the Fibonacci sequence does in $\frac{10}{89}$, still exhibit interesting mathematical properties worth exploring.

12.8 Metallic Unit Area Right Triangles

In the proper spirit of recreational mathematics we will consider the set of right triangles with a unit area formed of suitably scaled metallic numbers and their conjugates. Given a right triangle with sides scaled by the metallic number $(\sqrt{p}+1)^n$ and its conjugate $(\sqrt{p}-1)^n$, where p is a prime number and n is an integer, we define the sides as:

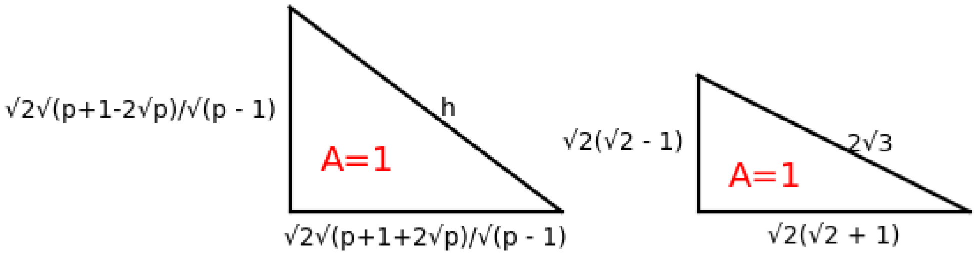


Figure 12.10: Metallic Triangles of Unit Area with shining Silver example

$$side\_a = r \cdot (\sqrt{p}+1)^n, \quad side\_b = r \cdot (\sqrt{p}-1)^n.$$

The area $A = 1$ of the triangle is given by:

$$A = \frac{1}{2} \cdot side\_a \cdot side\_b = \frac{1}{2} \cdot r^2 \cdot (\sqrt{p}+1)^n \cdot (\sqrt{p}-1)^n = 1.$$

which simplifies to:

$$r = \sqrt{2} \cdot \sqrt{\frac{1}{(p-1)^n}}.$$

For $n = 1, 2$, and 3, the solutions for r are respectively:

$$r_1 = \sqrt{2} \cdot \sqrt{\frac{1}{p-1}}, \quad r_2 = \sqrt{2} \cdot \sqrt{\frac{1}{p^2-2p+1}}, \quad r_3 = \sqrt{2} \cdot \sqrt{\frac{1}{p^3-3p^2+3p-1}}.$$

The scaling factor r maintains the unit area for all prime numbers p and integer values of n, highlighting an intrinsic property of metallic numbers in the construction of such triangles.

Given a prime number p, we define the sides of the triangle using scaled metallic numbers and their conjugates, whose short-sided lengths are determined as follows:

$$side\_a = \sqrt{2\left(\frac{p+1-2\sqrt{p}}{p-1}\right)}, \quad side\_b = \sqrt{2\left(\frac{p+1+2\sqrt{p}}{p-1}\right)},$$

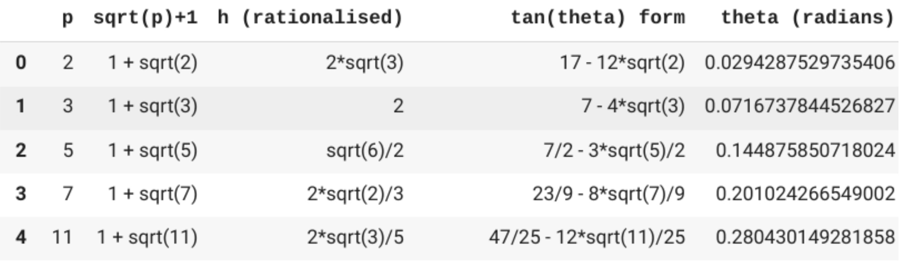
and whose hypotenuse h is calculated as follows:

$$h = \sqrt{2\left(\frac{p+1-2\sqrt{p}}{p-1}\right) + 2\left(\frac{p+1+2\sqrt{p}}{p-1}\right)}.$$

Their smallest angle θ is derived from the tangent ratio:

$$tan(\theta) = \frac{p+1-2\sqrt{p}}{p+1+2\sqrt{p}},$$

with θ in radians obtained via $\theta = \operatorname{atan}(tan(\theta))$, $\quad \theta_{\text{rad}} = N(\theta)$, $\quad \theta_{\text{in }\pi} = \frac{\theta}{\pi}$.



	p	sqrt(p)+1	h (rationalised)	tan(theta) form	theta (radians)
0	2	1 + sqrt(2)	2*sqrt(3)	17 - 12*sqrt(2)	0.0294287529735406
1	3	1 + sqrt(3)	2	7 - 4*sqrt(3)	0.0716737844526827
2	5	1 + sqrt(5)	sqrt(6)/2	7/2 - 3*sqrt(5)/2	0.144875850718024
3	7	1 + sqrt(7)	2*sqrt(2)/3	23/9 - 8*sqrt(7)/9	0.201024266549002
4	11	1 + sqrt(11)	2*sqrt(3)/5	47/25 - 12*sqrt(11)/25	0.280430149281858

Figure 12.11: Metallic Triangles of Unit Area with shining Silver example

We use symbolic mathematics libraries with key operations including:

- The `simplify` function to streamline expressions,
- The `atan` function to calculate the arctangent of the angle,
- The `N` function to normalize the angle in radians.

Figure 12.12 is a scatterplot in which the small angle θ in radians is plotted against prime number p. Each point is labeled with the value of the hypotenuse of the corresponding right triangle.

From the scatterplot, we observe:

- The small angle θ increases with the prime number p.
- The rate of increase in θ seems to diminish as p gets larger, suggesting a logarithmic-like relationship between θ and p.

Each point on the scatter plot is labelled by converting the hypotenuse expression to a string and formatting by replacing "sqrt" with the Unicode symbol and removing unnecessary characters.

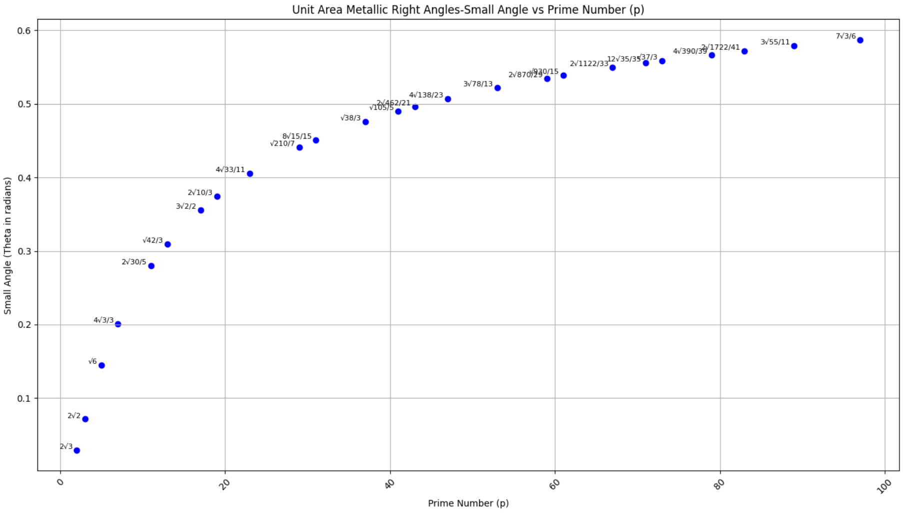


Figure 12.12: Scatterplot of the Small Angle θ vs Prime Number p

13. Unlikely Unreality of the Ramadunjan being Nu

The relationship between the Ramanujan constant $\mathscr{R}$ and the expressions involving the limit of prime roots and the basic Bernoulli number of the Basel problem are discussed. The Basel problem as first posed by Pietro Mengoli in 1644 was to find the exact value of the sum

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \cdots \tag{13.1}$$

and later was solved by Leonhard Euler in 1735, who showed that

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}. \tag{13.2}$$

Euler's solution involves the use of the zeta function $\zeta(s)$, which is defined as

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}. \tag{13.3}$$

Using the zeta function, we can express the sum in terms of $\zeta(2)$:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \zeta(2). \tag{13.4}$$

The Ramanujan constant $\mathscr{R}$ is defined as

$$\mathscr{R} = e^{\pi\sqrt{163}}. \tag{13.5}$$

We have

$$\sqrt{163}\pi = \sqrt{163 \times 6\zeta(2)} \tag{13.6}$$

so the Ramanujan constant $\mathscr{R}$ is:

$$\mathscr{R} = e^{\sqrt{163\times 6\zeta(2)}} \quad = \lim_{n\to\infty} \sqrt[n]{\prod_{p\leq n} p}^{\sqrt{163\times 6\zeta(2)}} \tag{13.7}$$

The second expression uses the fact that $e = \lim_{n\to\infty} \sqrt[n]{\prod_{p\leq n} p}$, which relates the limit of prime roots to the constant e.

The Basel problem is closely related to the Bernoulli numbers through the zeta function via the formula

$$\zeta(-n) = -\frac{B_{n+1}}{n+1}, \tag{13.8}$$

for $n \geq 1$, where B_n is the nth Bernoulli number. This formula relates the values of the Riemann zeta function at negative integers to the Bernoulli numbers. In particular, we have

$$\zeta(-1) = -\frac{B_2}{2} = -\frac{1}{12}. \tag{13.9}$$

where B_n is the nth Bernoulli number. This formula can be used to express $\zeta(s)$ as a sum over the Bernoulli numbers:

$$\zeta(s) = \frac{1}{1-2^{1-s}} \sum_{n=0}^{\infty} \frac{B_n}{n!}(2\pi)^s, \tag{13.10}$$

Using this formula, we can express $\zeta(2)$ in terms of the Bernoulli numbers as

$$\zeta(2) = \frac{\pi^2}{6} = \frac{1}{1-2^{-1}} \cdot \frac{B_2}{2!}(2\pi)^2 = \frac{B_2}{6}. \tag{13.11}$$

The solution to the Basel problem involves the second Bernoulli number. The Riemann zeta function is related to the Bernoulli numbers through the following formula:

$$\zeta(n) = \sum_{k=1}^{\infty} \frac{1}{k^n} = \frac{(-1)^{n-1} B_n}{2n}, \tag{13.12}$$

where B_n is the nth Bernoulli number. So, for example, we have:

$$\zeta(2) = \frac{\pi^2}{6} = \frac{(-1)^{2-1} B_2}{2\times 2}, \tag{13.13}$$

which gives $B_2 = \frac{1}{6}(\pi^2)$. Similarly, we have:

$$\zeta(-2) = -\frac{1}{12} = \frac{(-1)^{(-2)-1} B_{-2}}{2\times(-2)}, \tag{13.14}$$

which gives $B_{-2} = -\frac{1}{12}$. Note that the formula for B_n only holds for even n, so there is no such formula for odd-indexed Bernoulli numbers.

The Bernoulli numbers up to B_8 expressed in terms of the Riemann zeta function are:

$$\begin{aligned}
B_0 &= 1 \\
B_1 &= -\frac{1}{2} \\
B_2 &= \frac{1}{6}\left(\zeta(-2) - 1\right) \\
B_3 &= 0 \\
B_4 &= -\frac{1}{30}\left(\zeta(-4) + \zeta(-2) - \frac{5}{2}\right) \\
B_5 &= 0 \\
B_6 &= \frac{1}{42}\left(\zeta(-6) - \zeta(-4) + \zeta(-2) - \frac{1}{2}\right) \\
B_7 &= 0 \\
B_8 &= -\frac{1}{30}\left(\zeta(-8) + \zeta(-6) - \zeta(-4) + \zeta(-2) - 1\right)
\end{aligned}$$

Since $\zeta(2) = \frac{\pi^2}{6}$ and $B_0 = 1, B_1 = 0, B_2 = \frac{1}{6}$ the Ramanujan constant is related to the Bernoulli numbers as

$$\pi = \sqrt{6\zeta(2)} = \sqrt{6}\frac{\pi}{\sqrt{2}\cdot 2}\cdot\frac{B_2}{2!}, \tag{13.15}$$

Then, using this formula, we can rewrite the Ramanujan constant as

$$\mathscr{R} = e^{\sqrt{163}\pi} = e^{\sqrt{6\cdot 163\zeta(2)}} = e^{\sqrt{6}\frac{\pi}{\sqrt{2}\cdot 2}\cdot\frac{B_2}{2!}(\pi/\sqrt{2})^2}. \tag{13.16}$$

14. The ABC Conjecture

We will look here into the key concepts behind this conjecture, exploring the rationale behind the notion of the radical of a prime factorization and the significance of co-prime triplets involved in the $a+b=c$ conjecture.

> **Definition 21** *Radical Factorization* Consider a composite number n and its prime factorization into distinct primes $p_1, p_2, \ldots, p_k$. The radical of n, denoted as $\text{rad}(n)$, is the product of these distinct prime factors, i.e., $\text{rad}(n) = p_1 \cdot p_2 \cdots p_k$.
>
> The significance of the radical lies in its ability to measure the "magnitude" of numbers in a unique way. It serves as an essential tool in our quest to unravel the mysteries of the ABC conjecture.

14.1 The co-prime ABC Triplet

Now, picture three positive whole numbers a, b, and c that satisfy the equation $a+b=c$. These numbers form what we call an ABC triplet. However, the triplet is more than just a random assortment of digits; it possesses a remarkable property - coprimality.

The co-prime numbers are those that share no common factors other than 1. In other words, the greatest common divisor (GCD) of any two co-prime numbers is equal to 1. For an ABC triplet (a,b,c), we have $\gcd(a,b) = \gcd(a,c) = \gcd(b,c) = 1$. This property makes co-prime triplets particularly intriguing and is a crucial aspect of the ABC conjecture.

14.2 The ABC Conjecture

The ABC conjecture, formulated by Joseph Oesterlé and David Masser in 1985, posits a relationship between the three components of an ABC triplet. It states that for any $\varepsilon > 0$, there exists a constant K_ε such that for every ABC triplet (a,b,c) with $a+b=c$, the following inequality holds:

$$c < K_\varepsilon \cdot \text{rad}(abc)^{1+\varepsilon}$$

In simpler terms, the conjecture suggests that the sum c of the two co-prime numbers a and b cannot be much larger than the radical of their product abc. If proven true, the ABC conjecture would have profound implications for number theory, with consequences reaching far beyond its original formulation.

14.2.1 Radical of a Number

To unpack this a little more let's talk about the radical of a number. Imagine we have a composite number, and we prime factorize it into its basic building blocks (prime numbers). Now, the radical of this composite number is the product of all its distinct prime factors. For example, the radical of 20 is $2 \times 5 = 10$, as it is the product of the distinct prime factors 2 and 5.

Why do we care about the radical of a number, you ask? Well, it turns out that the radical plays a crucial role in understanding the ABC conjecture. It helps us measure the "magnitude" of numbers and their relationships in a unique way, and you'll see why this matters soon!

14.2.2 The co-prime ABC Triplet

Now, picture this: three numbers - 'a,' 'b,' and 'c' - hanging out together in an equation of the form 'a + b = c.' These numbers are not just any random digits; they have some special properties. For starters, they are positive whole numbers, and even more interestingly, they are co-prime!//

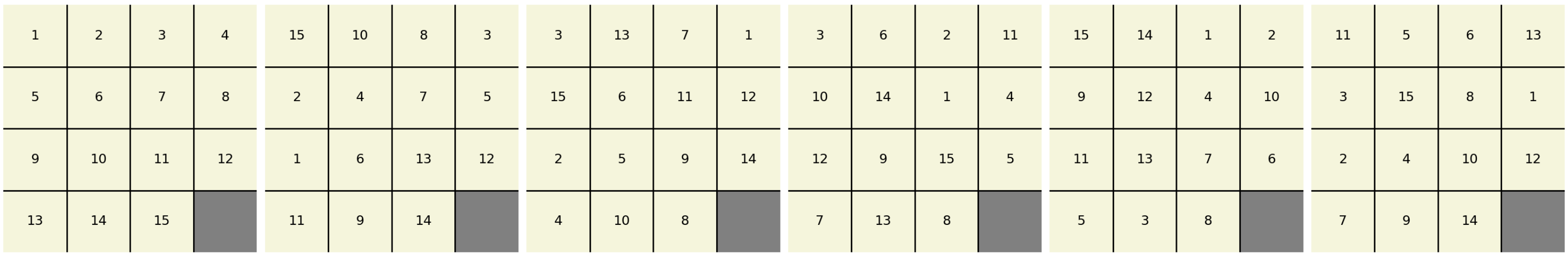
Hold up, co-prime? Don't worry; it's not as complicated as it sounds. co-prime simply means that 'a,' 'b,' and 'c' share no common factors other than 1. In other words, their greatest common divisor (GCD) is 1. For example, the triplet (2, 3, 5) is co-prime because the GCD of 2, 3, and 5 is 1. They have no other common divisors. But the triplet (4, 6, 10)? Not co-prime! The GCD of 4, 6, and 10 is 2.

So, why do we focus on co-prime triplets for the ABC conjecture? Well, that's where the real magic lies! co-prime triplets are like hidden gems in the vast landscape of numbers. They give us insight into the mysterious connections between numbers, and the ABC conjecture takes us on a quest to explore these connections further.

The ABC conjecture has spurred a multitude of research in number theory and sparked various related conjectures and results. It remains an open problem, and mathematicians continue to make significant strides in their pursuit of understanding the mysteries of numbers. Be prepared to encounter a treasure trove of elegant proofs, intricate theorems, and surprising connections between seemingly unrelated mathematical concepts. Happy exploring.

Part Three: Higher Altitude Explorations

12-winLossFrequency



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

15	10	8	3
2	4	7	5
1	6	13	12
11	9	14	

3	13	7	1
15	6	11	12
2	5	9	14
4	10	8	

3	6	2	11
10	14	1	4
12	9	15	5
7	13	8	

15	14	1	2
9	12	4	10
11	13	7	6
5	3	8	

11	5	6	13
3	15	8	1
2	4	10	12
7	9	14	

15. Combinatorics

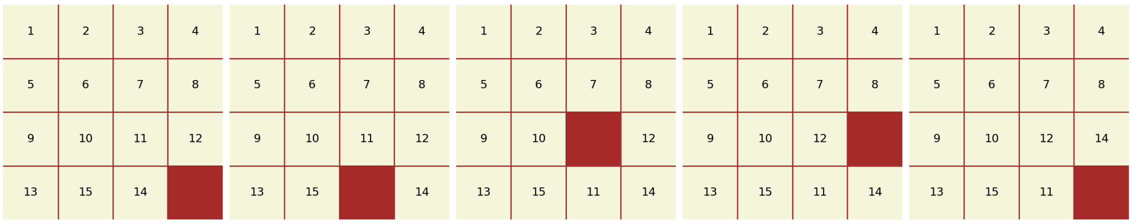
> If you cannot solve the proposed problem
> do not let this failure afflict you too much
> but try to find consolation with some easier success,
> try to solve first some related problem;
> then you may find courage to attack your original problem again.
> — *G. Poyla , How to Solve it* [14]

15.1 Sam Lloyd Problem

The Lloyd problem, often referred to as the 15-puzzle, presents a fascinating challenge in combinatorial mathematics and the world of recreational puzzles. At first glance, the problem might seem simple: you're given a square board partitioned into 16 smaller squares, where 15 of these squares have numbered tiles from 1 to 15 and one remains empty. The goal is to rearrange these tiles to get them in ascending order by only sliding tiles into the empty space.

Naive Implementation

A legitimate shuffle 'round[1] is presented below given the initial configuration presented to the user with the 14th and 15th tiles interchanged.



1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

1	2	3	4
5	6	7	8
9	10	11	12
13	15		14

1	2	3	4
5	6	7	8
9	10		12
13	15	11	14

1	2	3	4
5	6	7	8
9	10	12	
13	15	11	14

1	2	3	4
5	6	7	8
9	10	12	14
13	15	11	

Figure 15.1: Sam Loyd 15-Shuffle boards illustrating a shuffle round.

[1] SamLloydOneShuffle.ipynb delivers this picture.

You might at first naively try to replicate the challenge by simply drawing numbers without replacement and observing the configurations that arise. However, such an approach does not truly reflect the nature of the puzzle as borne out by the mixed *parity* (odd and even) nature of the *Disorder* numbers associated with these two random configurations.

```
Square 9:
15  2  1  9
 8 11 13  3
 6 10  4 12
 7 14  5
Order: [15, 2, 1, 9, 8, 11, 13, 3, 6, 10, 4, 12, 7, 14, 5]
Disorder: 48

Square 10:
 5  4 13  1
14  9 10  6
 7  8 11  2
12  3 15
Order: [5, 4, 13, 1, 14, 9, 10, 6, 7, 8, 11, 2, 12, 3, 15]
Disorder: 45
```

Figure 15.2: Naive Loyd 15-Shuffle based on random 1-15 draw.

So the Disorder of a number sequence measures how many pairs of elements are out of their expected ascending order. More formally,

Definition 22 *Disorder* Given a sequence S of n distinct numbers, the “Disorder” $D(S)$ of the sequence is defined as the number of pairs $(S[i], S[j])$ such that $i < j$ but $S[i] > S[j]$.

Mathematically, this can be represented as:

$$D(S) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} I(S[i] > S[j])$$

Where:

- I is an indicator function such that:

$$I(\text{true}) = 1$$

$$I(\text{false}) = 0$$

- n is the length of the sequence.
- $S[i]$ is the i^{th} element of the sequence.

For a sequence sorted in ascending order, $D(S)$ will be 0. The greater the value of $D(S)$, the further the sequence is from being sorted. For example, consider the list $[4, 3, 2, 1]$ in which we have a total disorder of 6 given by:

- $(4, 3)$, $(4, 2)$, and $(4, 1)$ are out of order.
- $(3, 2)$ and $(3, 1)$ are out of order.
- $(2, 1)$ is out of order.

For a sorted list (either ascending or descending), the disorder will be 0. The larger the returned value, the more pairs in the list are out of their expected order if the list were to be sorted in ascending order.//

The function calculate_disorder(numbers) within the code[2]delivers disorder:

```
def calculate_disorder(numbers):
    disorder = 0
    for i in range(len(numbers)):
        for j in range(i + 1, len(numbers)):
            if numbers[i] > numbers[j]:
                disorder += 1
    return disorder
```

that calculates and returns the "disorder" of a list of numbers which is the count of how many pairs of numbers in the list are out of order, i.e., how many pairs (i, j) exist such that $i < j$ and numbers[i] > numbers[j] by:

1. Initialize the disorder count to 0.
2. Iterate through each number in the list using an index i.
3. For each i, iterate through the subsequent numbers in the list using index j.
4. For every pair (numbers[i],numbers[j]) where numbers[i] > numbers[j], increment the disorder count.
5. Return the final count of disorder.

The function draw_puzzle(n), create and prints the series of random 4x4 number puzzles:

```
def draw_puzzle(n):
    summary = []
    disorders = []
    for count in range(n):
        numbers = list(range(1, 16))
        random.shuffle(numbers)
        print(f"Square {count + 1}:")
        for i in range(4):
            for j in range(4):
                index = 4 * i + j
                if index == 15:
                    print("  ", end=" ")
                else:
                    print(f"{numbers[index]:2d}", end=" ")
            print()
```

1. It initializes two empty lists: summary and disorders, although these lists are not used within the provided code.
2. For each iteration from 0 to $n-1$, the function:
   (a) Creates a list, numbers, consisting of integers from 1 to 15.
   (b) Shuffles the numbers list randomly.
   (c) Prints the header Square count + 1.
   (d) Iterates through a 4x4 grid and:
      - For the last grid cell (bottom-right), it prints two spaces.
      - For all other cells, it prints the corresponding number from the shuffled list.
   (e) After every row of 4 numbers, a new line is printed.

The result is a series of n shuffled 4x4 number squares where the bottom-right cell is always empty, naively simulating a sliding puzzle.

[2] naiveSamlLloyd-wrong.ipynb

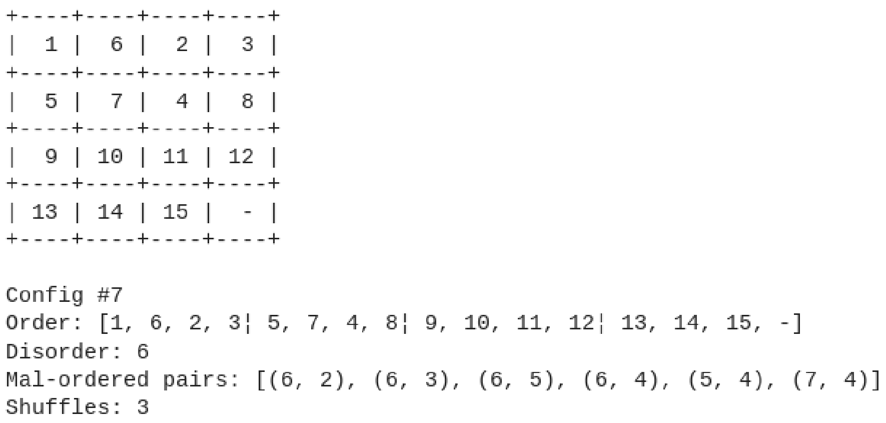
In the genuine Lloyd problem, the empty space, located at the bottom right, has a unique role. During the course of the game, this space is allowed to move around, facilitating the rearrangement of the numbered tiles. However, at the end of each shuffle, the empty space must return to the bottom right position, thus constraining the possible configurations one can achieve. Loyd's Fifteen puzzle using Hamilton's notion of a network circuit is explored in Benson's, 'Moment of Proof' [2]. In it he lays out on full the proof of Loyd's money being safe. The natural order of the integers 1-15 denotes the normal increasing order. Arrangements of the numbered tiles contain each integer 1-15 exactly once but not in the natural order.

A list is said to be odd or even depending on whether the total number of inversions of the list is odd or even. An inversion is a pair of number in the list (not necessarily adjacent) that are not in the natural order. The extent of the disorder of the list with respect to the natural order is called the "disorder" counting the number of inversions in the list. Arrangements of the 15 puzzle are said to be odd or even depending on whether the associated list is odd or even. Disorder serves as an *invariant property* of the configuration. Thus if we were to physically swap tiles 14 and 15, no amount of allowed shuffles would get us to a configuration where the tiles are arranged in an ordered sequence from 1 to 15. This is due to a shift from an odd parity disorder to an even parity disorder. In other words, the total number of moves required to get from a given configuration back to the ordered sequence is either always even or always odd. This peculiarity ensures that not all configurations are solvable and why Loyd was able to offer a $1000 prize reward to anyone doing so.

Config	Linear Configuration	Disorder Number
5002	$3,6,4,8 \mid 5,1,2,7 \mid 14,13,10,12 \mid 9,11,15,-$	26
5003	$1,3,4,7 \mid 5,10,2,8 \mid 9,14,6,11 \mid 13,15,12,-$	18
5004	$1,2,8,3 \mid 6,7,4,11 \mid 5,9,15,10 \mid 13,14,12,-$	18
5005	$5,2,12,3 \mid 6,1,4,8 \mid 11,7,14,15 \mid 10,9,13,-$	28
5006	$9,3,5,6 \mid 10,4,11,7 \mid 2,1,8,12 \mid 13,14,15,-$	30

Table 15.1: Even parity configurations of the Loyd puzzle

An allowable genuine shuffle with disorder number looks like the following.



```
+----+----+----+----+
|  1 |  6 |  2 |  3 |
+----+----+----+----+
|  5 |  7 |  4 |  8 |
+----+----+----+----+
|  9 | 10 | 11 | 12 |
+----+----+----+----+
| 13 | 14 | 15 |  - |
+----+----+----+----+

Config #7
Order: [1, 6, 2, 3¦ 5, 7, 4, 8¦ 9, 10, 11, 12¦ 13, 14, 15, -]
Disorder: 6
Mal-ordered pairs: [(6, 2), (6, 3), (6, 5), (6, 4), (5, 4), (7, 4)]
Shuffles: 3
```

Figure 15.3: Loyd 15 shuffle using shuffle function.

The following functions work together to allow for a randomized, solvable configuration of the 15 puzzle, starting from a solved state with the empty cell in the bottom-right corner implementing allowable shuffling mechanisms for the 15 puzzle:

```
    def valid_moves(empty_index):
    moves = []
    row, col = divmod(empty_index, 4)
    if row > 0:
        moves.append(empty_index - 4)
    if row < 3:
        moves.append(empty_index + 4)
    if col > 0:
        moves.append(empty_index - 1)
    if col < 3:
        moves.append(empty_index + 1)
    return moves
def shuffle_board(board, moves_count):
    empty_index = 15
    for _ in range(moves_count):
        move = random.choice(valid_moves(empty_index))
        board[empty_index], board[move] = board[move], board[empty_index]
        empty_index = move
    while empty_index != 15:
        move = random.choice(valid_moves(empty_index))
        board[empty_index], board[move] = board[move], board[empty_index]
        empty_index = move
    return board
```

`valid_moves(empty_index):` returns a list of indices corresponding to the tiles that can be moved into the empty space. It determines which tiles can be moved to the empty space with the 4x4 grid visualized as a linear list of length 16, where the index of the empty cell is given by `empty_index`. Using the row and column information derived from this index:

- If the empty cell is not in the top row, the tile above can slide down.
- If the empty cell is not in the bottom row, the tile below can slide up.
- If the empty cell is not in the leftmost column, the tile on the left can slide right.
- If the empty cell is not in the rightmost column, the tile on the right can slide left.

`shuffle_board(board, moves_count):` shuffles the puzzle by performing a series of valid moves.

1. The empty cell's index is initialized to 15 (the bottom-right corner).
2. For a specified number of moves (`moves_count`), a valid move is randomly selected, and the chosen tile is swapped with the empty space.
3. An additional loop ensures that the empty space is moved back to its initial position (index 15). This ensures that the empty space remains in the bottom-right corner after shuffling.

Measures of Rank coefficients

Both Spearman Rank and Kendall's Tau provide a measure of rank correlation and are designed to measure ordinal association. They provide insight into the ordinal structure of the data, rather than precise positional differences.

Definition 23 The Spearman Rank correlation coefficient, ρ, assesses the strength and direction of the monotonic relationship between two ranked variables. It is computed as:

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)}$$

where d_i is the difference between the two ranks of each observation and n is the number of observations.

Motivation: Spearman Rank is motivated by the need to determine the degree of association between two ordinal or ranked variables. It can be seen as a non-parametric version of the Pearson correlation.

Definition 24 Kendall's Tau, denoted as τ, is a measure of rank correlation. It calculates the difference between the number of concordant and discordant pairs divided by the total number of pairs. It is given by:

$$\tau = \frac{n_c - n_d}{n(n-1)/2}$$

where n_c and n_d are the number of concordant and discordant pairs, respectively, and n is the number of observations.

Motivation: Kendall's Tau is motivated by the need for a measure that can handle ties in data and does not assume a linear relationship between the variables. It is useful when the data is ordinal.

Women's Vault Competition Judging with Multiple Judges

Consider three gymnasts: A, B, and C with scores based on the evaluation of three different judges combined from Difficulty (D-Score) and Execution (E-Score) components:

Gymnast	Judge 1 (Total)	Judge 2 (Total)	Judge 3 (Total)
A	14.4	14.5	14.3
B	14.5	14.4	14.4
C	14.3	14.3	14.2

$$\begin{aligned} &\text{Judge 1's Ranking:} \quad B > A > C \\ &\text{Judge 2's Ranking:} \quad A > B > C \\ &\text{Judge 3's Ranking:} \quad B = A > C \end{aligned}$$

We can calculate the Spearman, Kendall Tau, and Disorder measures for each pairwise comparison between judges (Judge 1 vs. Judge 2, Judge 1 vs. Judge 3, and Judge 2 vs. Judge 3).

Judge 1 vs. Judge 2

The rank differences d_i for gymnasts A, B, and C are respectively 1, -1, and 0. Using *Spearman's rank correlation* coefficient formula, we have a correlation between Judge 1 and Judge 2 of 0.5:

$$\rho = 1 - \frac{6\sum d_i^2}{n(n^2-1)} = 1 - \frac{6(2)}{3(8)} = 0.5$$

For each possible pair of gymnasts, we have:

- For the pair (A, B): Judge 1 ranks B before A, while Judge 2 ranks A before B (discordant).
- For the pair (A, C): Both judges rank A before C (concordant).
- For the pair (B, C): Both judges rank B before C (concordant).

From this, we find *Kendall's Tau* between Judge 1 and Judge 2 is $\frac{1}{3}$ as:

$$\text{Concordant pairs (C)} = 2 \quad \text{Discordant pairs (D)} = 1 \quad \tau = \frac{C-D}{C+D} = \frac{1}{3}$$

Counting the pairs that are not in the same order for both judges, we have a total *disorder* between Judge 1 and Judge 2 of 1 as (A, B) are in a different order.

Judge 1 vs. Judge 3

Given the rank differences for gymnasts A, B, and C, which are 0.5, -0.5, and 0 respectively, the Spearman's rank correlation coefficient is $\rho = 1 - \frac{3}{24} = 0.875$

- A and B are ranked equally or with the same difference by both judges.
- A-C and B-C pairs are concordant between Judge 1 and Judge 3.

leads Kendall's Tau correlation between Judge 1 and Judge 3 of $\tau = \frac{C-D}{C+D} = \frac{2-0}{2} = 1$ There's no disorder between Judge 1 and Judge 3's rankings as both rank C last and A, B as tied or with the same difference. Total disorder = 0.

Judge 2 vs. Judge 3

Given the rank differences for gymnasts A, B, and C, which are -0.5, 0.5, and 0 respectively, the Spearman's rank correlation coefficient is again $\rho = 1 - \frac{3}{24} = 0.875$

- A and B have no specific order between Judge 2 and Judge 3.
- A-C and B-C pairs are concordant between Judge 2 and Judge 3,

for a Kendall's Tau between Judge 2 and Judge 3 of $\tau = \frac{C-D}{C+D} = \frac{2-0}{2} = 1$ There's no disorder between Judge 2 and Judge 3's rankings as both rank C last and A, B as tied or with the same difference.

So from these statistical measures we may conclude that, Judges 1 and 2 have notable differences in their rankings, suggesting a potential divergence in their evaluation criteria. On the other hand, Judges 1 and 3, and especially Judges 2 and 3, have a high degree of agreement, suggesting a common understanding or criteria for evaluation.

Discriminating Puzzle Configurations

In the context of the 15 puzzle, it has been observed that for given disorder numbers, there can be equivalent values for the Spearman Rank and even more so for Kendall's Tau correlation coefficients. This brings forth the question of whether these ranking coefficients can comprehensively discriminate between distinct positions of the puzzle. While both coefficients can indicate whether the puzzle configuration is closer or farther from the solution in terms of ordinal structure, they might not capture the nuanced differences between two distinct, yet similarly structured, configurations. Two different configurations might have similar or even identical rank structures, leading to high correlation values, even if the actual tile positions differ.

While statistical measures such Spearman Rank and Kendall's Tau can provide further insights into the ordinal nature of the 15 puzzle configurations, they are by design not granular enough to discriminate between every unique configuration. The code subplotsHistScatterDisorderSpearman.ipynb delivers both a histogram of Disorder numbers and correlation analysis

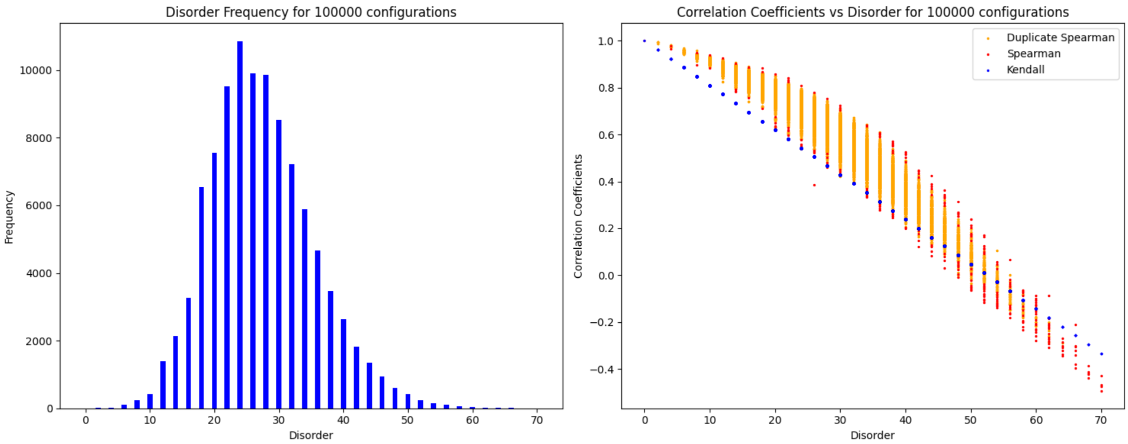


Figure 15.4: Spearman rank and Kendall tau ranking coefficients as possible unique Disorder measures.

```
def draw_puzzle(n, moves_count):
    disorders = []
    kendalls = []
    spearmans = []
    summaries = PrettyTable(['Config #', 'Linear Configuration', 'Disorder', 'Kendall\'s Ta
    seen_configs = set()
    i = 0
    while i < n:
        board = list(range(1, 16)) + ['-']
        shuffle_board(board, moves_count)
        config_tuple = tuple(board)
        if config_tuple in seen_configs:
            continue
        seen_configs.add(config_tuple)
        disorder = calculate_disorder(board[:-1])
        disorders.append(disorder)
        kendall_corr, _ = kendalltau(board[:-1], range(1, 16))
        spearman_corr, _ = spearmanr(board[:-1], range(1, 16))
        kendalls.append(kendall_corr)
        spearmans.append(spearman_corr)
        formatted_order = ', '.join(map(str, board))
        summaries.add_row([i + 1, formatted_order, disorder, round(kendall_corr, 2), round(
        i += 1
```

The function `draw_puzzle(n, moves_count)` constructs a table summarizing the disorder and correlation metrics of up to n unique shuffled configurations of the 15 puzzle, ensuring no repeated configurations. Here's a detailed breakdown of its functionality:

1. Three lists, `disorders`, `kendalls`, and `spearmans`, are initialized to store metrics about each generated configuration.
2. A table `summaries` is initialized with the columns 'Config #', 'Linear Configuration', 'Disorder', 'Kendall's Tau', and 'Spearman Rank' using the `PrettyTable` module.
3. A set, `seen_configs`, is used to keep track of previously generated configurations, ensuring that duplicate configurations are not considered.
4. Using a `while` loop, up to n unique configurations are generated. For each configuration:

(a) A solved board configuration is created as a list with numbers 1 to 15, followed by an empty space represented by '`-`'.

(b) The board is then shuffled using the previously mentioned `shuffle_board` function.

(c) The configuration is checked against the `seen_configs` set. If it has been seen before, the loop continues to the next iteration without incrementing the counter.

(d) Metrics are then calculated for this unique configuration:

- The `disorder` (as described in previous explanations) of the configuration.
- The `Kendall's Tau` correlation coefficient compared to a solved configuration.
- The `Spearman Rank` correlation coefficient, again compared to a solved configuration.

(e) These metrics, along with the linear representation of the board, are then added as a new row to the `summaries` table.

(f) The counter *i* is incremented.

15.2 Placing Distinct Balls In distinguishable Boxes

We aim to calculate the number of arrangements $A(r,n)$ where we have more distinguishable balls r than distinguishable boxes such that $r \geq n$. The recursive relation for these arrangements is given by:

$$A(r,n) = \sum_{k=1}^{r} \binom{r}{k} A(r-k, n-1)$$

Where the $\binom{r}{k}$ represents the combination of choosing k elements from r. To get a sense of where this comes from imagine we have r distinct books that we want to arrange on n distinct shelves, represented by $A(r,n)$.

- **Combinations:** The term $\binom{r}{k}$ represents choosing k books from r. By choosing k books, we decide which subset of our books will be placed on the first shelf.
- **Recursive Arrangement:** The term $A(r-k, n-1)$ represents the ways to arrange the remaining $r-k$ books on the $n-1$ remaining shelves.
- **Summation:** Summing over all k values considers all possible numbers of books for the first shelf, then recursively calculates the arrangements for the remaining books.

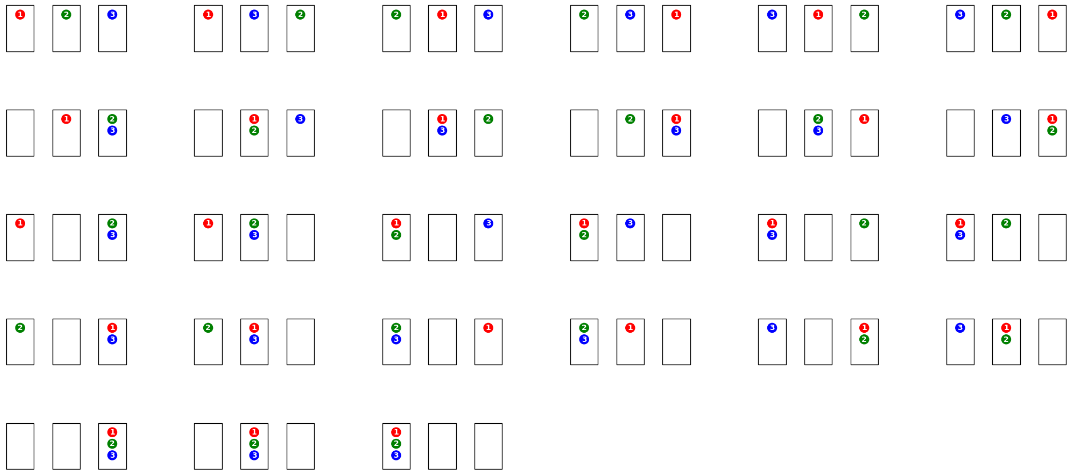


Figure 15.5: 27 ways of placing 3 distinguishable balls in 3 distinguishable boxes.

This visualization 15.5 displays the 24 distinct ways that distinguishable balls can be arranged in distinct boxes, revealing at first the $n!/n^n$ probability of delivering only one ball in one box (in adherence to a Pauli-exclusion principle that forbids a box from accommodating more than one ball. Our code visualizes the different arrangements of distinguishable balls in distinguishable boxes. An arrangement in this context means the different ways the balls can be placed in the boxes.

- **Combinations**: Refers to the selection of items without considering the order.
- **Permutations**: Refers to the arrangement of items where the order is essential.
- **Arrangement**: A general term used here to describe both the above scenarios, including cases where multiple balls can be placed in a single box.

15.2.1 Code Workflow

The Python code from snippet is as follows:

```
def generate_combinations(balls, boxes_count):
    # generate permutations of balls for
cases where there is exactly one ball in each box
    permutations = list(itertools.permutations(balls))

    # Convert to format used by draw_combination
    perms_converted = []
    for perm in permutations:
        converted = [[] for _ in range(boxes_count)]
        for idx, ball in enumerate(perm):
            converted[idx].append(ball)
        perms_converted.append(converted)

    # generate all other possible distributions
    distributions = list(itertools.product(range(1, boxes_count+1), repeat=len(balls)))
    dist_converted = [[] for _ in range(len(distributions))]
    for dist_idx, dist in enumerate(distributions):
        for ball_idx, box in enumerate(dist, 1):  # Start enumeration at 1
            dist_converted[dist_idx].append((str(ball_idx), box))
```

1. First, the code calculates the **permutations** of balls for the specific case where there's precisely one ball in each box. This ensures every box has a single, distinguishable ball.
2. After obtaining the permutations, the code generates the different distributions (or arrangements) of the balls among the boxes.
3. The distributions include scenarios where multiple balls can be present in a single box. This is achieved using the Cartesian product, effectively mapping balls to box indices.
4. The code then filters out previously computed permutations from these distributions to avoid redundancy.
5. Once the arrangements are generated, they are visualized using 'matplotlib' with each box and ball represented graphically.

15.2.2 Generator formulae for r balls in n boxes

The generator formula, $A(r,n) = \sum_{k=1}^{r} \binom{r}{k} A(r-k, n-1)$, encapsulates selecting subsets of books (combinations) and then arranging the remaining books across multiple shelves.

Consider the interplay of combinations and permutations in the formula which considers all ways to distribute r books across n shelves, accounting for empty shelves.

$$\text{constituents} = \sum_{k=0}^{n} (-1)^k \binom{n}{k} (n-k)^r$$

- $(-1)^k$: Introduces alternating signs, a theme in combinatorial arguments.
- $\binom{n}{k}$: Represents choosing k shelves from n to be left empty.
- $(n-k)^r$: Expresses the permutations of arranging r books on $n-k$ shelves.

The code recurciveArrangementsOfrinn.ipynb has the following features

- **Memoization**: An optimization technique used to speed up recursive algorithms. It involves storing solutions to overlapping subproblems to avoid redundant computations.

- **Python's comb module**: The 'comb' function from the 'math' module is used to compute combinations.
- **List comprehensions**: Used extensively in Python to generate lists without the need for appending to a list in a loop. It provides a concise way to create lists.

```
from math import comb, factorial
memo = {}
# Recursive formula for A(r, n)
def A_recursive(r, n, display=False):
    if (r, n) in memo:
        return memo[(r, n)]
    if r < n:
        return 0
    if n == 1:
        return 1
    if r == n:
        return factorial(r)
    # Recursive calculation
    constituents = [(comb(r, k), A_recursive(r - k, n - 1)) for k in range(1, r + 1)]
    total = sum([c[0] * c[1] for c in constituents])

    # Displaying each calculation (if display is True)
    if display:
        for k, c in enumerate(constituents, start=1):
            print(f"{c[0]} x A({r - k}, {n - 1}) = {c[0] * c[1]}")
    memo[(r, n)] = total
    return total
```

The function '$A_recursive$' calculates $A(r,n)$ using recursion and memoization. If the solution for a specific (r, n) pair is already computed, it fetches it from the memoization dictionary 'memo' to avoid redundant computations.

The output from the code for r=6, n=4, $A(6,4)$ is:

$$C(6,1) \times A(5,3) = 6 \times 150 = 900$$
$$C(6,2) \times A(4,3) = 15 \times 36 = 540$$
$$C(6,3) \times A(3,3) = 20 \times 6 = 120$$
$$C(6,4) \times A(2,3) = 15 \times 0 = 0$$
$$C(6,5) \times A(1,3) = 6 \times 0 = 0$$
$$C(6,6) \times A(0,3) = 1 \times 0 = 0$$

Using the fixed generator formula, $A(6,4) = \sum_{k=0}^{r}(-1)^k \times C(4,k) \times (4-k)^6$ we have

$$(-1)^0 \times C(4,0) \times (4-0)^6 = 4096$$
$$(-1)^1 \times C(4,1) \times (4-1)^6 = -2916$$
$$(-1)^2 \times C(4,2) \times (4-2)^6 = 384$$
$$(-1)^3 \times C(4,3) \times (4-3)^6 = -4$$
$$(-1)^4 \times C(4,4) \times (4-4)^6 = 0$$

Both return $A(6,4) = 1560$.

15.3 Statistical Ensembles

Between the Thermodynamic *M*acrostate of a system described in rough terms by Molar amount, Internal energy, U or notions of Pressure (phenotypes) and the innumerable *microstates* comprising N distinguishable particles all of which could characterise such a rough description there exists the Thermal Equilibrium *Distribution*. For a simple assembly,**??**GuenaltStat] consider the following Macrostate comprising N=4 distinguishable particles labelled A,B,C and D. with total energy, $U = 4\varepsilon$. The question is what are the possible states of any one particle, the solution being states of (non degenerate) energies $0, \varepsilon, 2\varepsilon, 3\varepsilon, \ldots$ labelled $j = 0, 1, 2, ..$ with $\varepsilon_j = j\varepsilon$ We define the distributions as $\{n_j\}$, with $j = 0, 1, 2, 3..$, and noting that any allowable distributions must satisfy

$$\sum_j n_j = 4; \sum_j \varepsilon_j = 4\varepsilon$$

Consider the five possible *distributions*

Distribution	n_0	n_1	n_2	n_3	n_4	n_5	...
1	3	0	0	0	1	0	...
2	2	1	0	1	0	0	...
3	2	0	2	0	0	0	...
4	1	2	1	0	0	0	...
5	0	4	0	0	0	0	...

Table 15.2: A Distribution of A,B,C and D

A *microstates* specifies the state of each of the four particles. In order to count their number consider the possibilities:

1. A is in state $j = 4$; while B, C, and D are in state $j = 0$.
2. B is in state $j = 4$; while A, C, and D are in state $j = 0$.
3. C is in state $j = 4$; while A, B, and D are in state $j = 0$.
4. D is in state $j = 4$; while A, B, and C are in state $j = 0$.

Distribution 1 ($t^1 = 4$) microstates:

1. A in state $j = 4$; B, C, D in state $j = 0$
2. B in state $j = 4$; A, C, D in state $j = 0$
3. C in state $j = 4$; A, B, D in state $j = 0$
4. D in state $j = 4$; A, B, C in state $j = 0$

Distribution 2 ($t^2 = 12$) microstates:

$n_1 = 1$ and $n_4 = 1$:

1. A in state $j = 1$; B in state $j = 4$; C, D in state $j = 0$
2. B in state $j = 1$; C in state $j = 4$; A, D in state $j = 0$
3. ...

$n_0 = 2$ (the rest of the particles are in state $j = 0$):

1. A, B in state $j = 0$; C, D in other states
2. ...

Distribution 3 ($t^3 = 6$) microstates:

$n_2 = 2$:

1. A in state $j = 2$; B in state $j = 2$; C, D in state $j = 0$
2. ...

$n_0 = 2$ (the rest of the particles are in state $j = 0$):

1. A, B in state $j = 0$; C, D in other states
2. ...

Distribution 4 ($t^4 = 12$) microstates:

$n_1 = 2$ and $n_3 = 1$:

1. A, B in state $j = 1$; C in state $j = 3$; D in state $j = 0$

$n_0 = 1$ (the rest of the particles are in other states):

1. A in state $j = 0$; B, C, D in other states

Distribution 5 ($t^5 = 1$) microstates:

1. A, B, C, D in state $j = 4$

Hence $t^1 = 4$ for

The code, microstatesEquivalentOfDistributions.ipynb for Distribution 3 returns: $t^3 = 6$ microstates for Distribution 3:

1. $('0','2','2','0')$
2. $('2','0','2','0')$
3. $('0','2','0','2')$
4. $('2','0','0','2')$
5. $('2','2','0','0')$
6. $('0','0','2','2')$

The crucial function is the following

```
from itertools import permutations
# For each distribution, generate all unique permutations of states (microstates)
for idx, distribution in enumerate(distributions):
    microstates = set(permutations(distribution))
    total_microstates = len(microstates)
    # Add the total number of microstates for the current distribution to Omega.
    Omega += total_microstates
    print(f"t{to_superscript(idx+1)}={total_microstates} microstates for Distribution {idx+
    for microstate_idx, microstate in enumerate(microstates):
        print(f"  Microstate {microstate_idx+1}: {microstate}")
    print()
```

1. Iterate over each *distribution* in the list `distributions`.
2. For the current *distribution*, generate all unique permutations of its states. These permutations are termed *microstates*.
3. Compute the total number of *microstates* for the current *distribution*.
4. Increment the variable Ω by the total number of *microstates* for the current *distribution*.
5. Print the total number of *microstates* for the current *distribution*, labeling it with a superscript notation:

   $$t^{(\text{idx}+1)} = \text{total\_microstates}$$

   where `idx` is the current index of the distribution.
6. For each *microstate* in the set of *microstates*, print the *microstate*.

15.4 Optimisation models

Optimization problems often require effective strategies to find the best solution within a vast search space. Key concepts in this area include evaluation functions, hill climbing, and simulated annealing, each playing a critical role in navigating towards optimal solutions.

Evaluation Function: At the heart of many optimization algorithms is an evaluation function, a mathematical tool used to assess the quality or fitness of a given solution. In the context of spin systems, for instance, this function might calculate the number of runs of consecutive arrows with the same orientation. The higher the number of such runs, the better the configuration is considered, thereby guiding the search process.

Hill Climbing: Hill climbing is an iterative optimization algorithm that starts with an arbitrary solution and makes small changes to it. If the change results in a better solution as per the evaluation function, the algorithm moves to this new solution. This process continues until no further improvements can be made. However, hill climbing can get trapped in local maxima – points that are better than their immediate neighbors but not the best overall. These algorithms use an *evaluation function* to decide whether to move to a neighboring state or not, based on its relative quality.

Simulated Annealing: To address the limitations of hill climbing, simulated annealing introduces a probabilistic approach. Inspired by the metallurgical process of annealing, this method uses a "temperature" parameter to determine the likelihood of accepting worse solutions. Initially, when the temperature is high, the algorithm is more likely to accept sub-optimal solutions, allowing it to explore more of the solution space and potentially escape local maxima. As the temperature decreases, the algorithm becomes more conservative, honing in on the best solution found. This temperature-dependent approach provides a balance between exploration and exploitation, increasing the chances of finding a global maximum.

Toy Ising Spin Model

In statistical physics, spin models are used to understand magnetic properties of materials. One of the most famous models is the Ising model, which describes the interaction of spins on a lattice where each spin interacts with its neighbors and has been pivotal in studying phase transitions and understanding critical phenomena. The Monte Carlo method, is a powerful tool for studying the Ising model, especially near critical points where analytical solutions are not feasible. Although our simulation does not include an external field, the concept of spins changing direction is fundamental to understanding paramagnetic materials.

Our Monte Carlo simulation focuses on a simple system of spins, represented by arrows pointing either up or down capturing the essence of spin models, in which each spin can be in one of two states: up (+) or down (-). The simulation randomly reverses the spins of adjacent arrows, mimicking the thermal fluctuations in real magnetic systems.

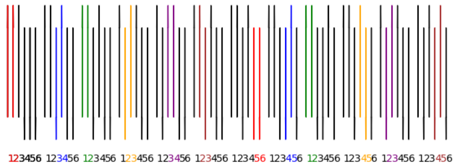


Figure 15.6: Spin UP and Spin Down manoeuvres.

The sinippet from the codemonteArrowIsing.ipynb draws the arrows

```
def draw_arrows(spins, ax, offset=0, swap_indices=None):
    for i, spin in enumerate(spins):
        pos_x = i + offset  # Apply offset for horizontal positioning
        color = 'black'  # Default color
        # Change color if this arrow is part of the swapped pair
        if swap_indices and i in swap_indices:
            color = swap_indices[2]  # third element in swap_indices tuple is color
        # Draw the arrow
        if spin == '+':
            ax.arrow(pos_x, 0.1, 0, 0.8, head_width=0.2, head_length=0.2, fc=color, ec=colo
        else:
            ax.arrow(pos_x, 0.9, 0, -0.8, head_width=0.2, head_length=0.2, fc=color, ec=col
        # Text is always below the arrow
        ax.text(pos_x-0.05, -0.3, str(i+1), color=color)
```

Spin Representation

We represent spins as a series of arrows, initially set in a specific order. The spins are denoted as '+' for up and '-' for down. The core of our simulation is the Monte Carlo method, which randomly selects and reverses adjacent spins. This method reflects the random nature of thermal fluctuations in magnetic systems.The Python script uses matplotlib to draw arrows corresponding to each spin state. The arrows are colored differently at each step to track the changes over time. The script generates a histogram of the number of turns required to reach a specific end configuration.

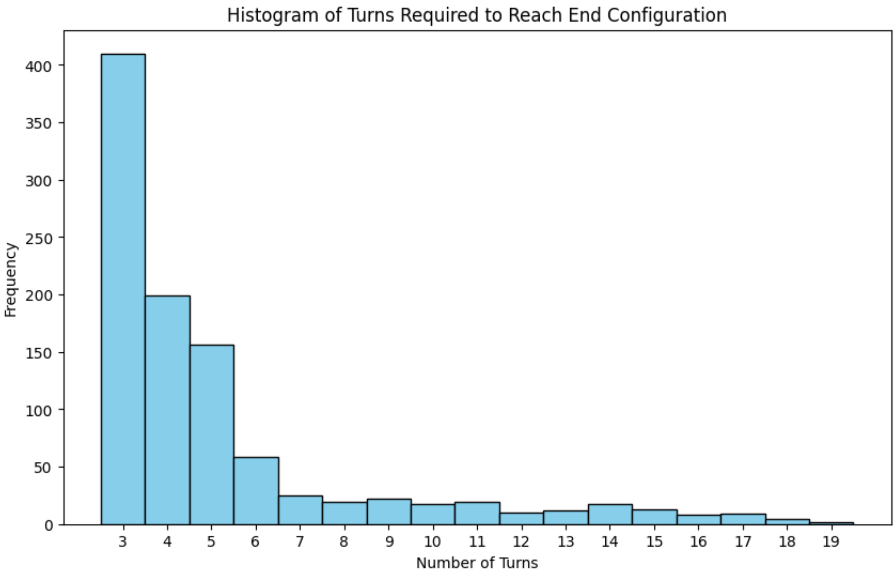


Figure 15.7: Number of manoeuvres of arrows to have them aligned in parallel and anti-parallel fashioned .

15.5 The Secretary Problem

The secretary problem, also known as the marriage problem or the sultan's dowry problem, demonstrates a scenario of optimal stopping theory. The problem was originally formulated during the 1950s and 1960s and involves a decision-making process where an interviewer aims to hire the best secretary out of n applicants, who are interviewed one by one in a random order. The interviewer must decide immediately after each interview whether to hire the applicant or not, and the decision is irrevocable. The goal is to maximize the probability of selecting the best applicant. With more modern sensibilities, it can be seen as applicable to a variety of situations like real estate, online dating, and any scenario where one must choose the best option without the ability to return to previous choices.

In the secretary problem, the objective is to maximize the probability of selecting the best candidate from a sequentially ordered set of candidates. The optimal stopping rule suggests that the best time to make a decision is after evaluating approximately $\frac{1}{e}$ of the candidates, where e is the base of the natural logarithm. The $\frac{1}{e}$ threshold, or the optimal stopping rule strategy maximizes the probability of choosing the best option and is grounded in the balance between having enough information to make an informed decision and the risk of passing up the best choice and offers the best chance of making the optimal choice in a scenario where options are presented sequentially and decisions are irrevocable.

There is a similarity between the optimal stopping rule in the secretary problem and the evaluation function used in hill-climbing optimization algorithms. As such the evaluation function is a comparison process that assesses whether a current candidate is better than all previously evaluated ones.

15.5.1 Exploration and Exploitation

The phase before reaching the $\frac{1}{e}$ threshold in the secretary problem can be likened to the exploration phase in hill-climbing, where one gathers information about the environment. After this threshold, the decision phase begins, similar to the exploitation phase in hill-climbing, using the information gathered earlier to make the best decision. The code, chooseSecretary.ipynb delivers for an interview panel of 9 an initial exploration pool of 3 from which happily the best candidate presents 2 candidates after the observation pool: **Enter the size of the pool of secretaries (p):** 9
Shuffled list of ranks: [3, 6, 2, 8, 1, 5, 7, 9, 4]
Revealed so far: [3, 6, 2]
Revealed so far: [3, 6, 2, 8, 1]
Remaining list: [7, 9, 4]

15.5.2 Automated Secretary Problem with Optimal Stopping

Monte Carlo simulations are used to show the surprising optimality of the stopping routine. The automated code version implements the optimal stopping rule, stopping after evaluating the first $\frac{n}{e}$ candidates, where e is the base of the natural logarithm. This rule is based on the theory that after seeing about 36.8% of the applicants, the interviewer has enough information to make an informed decision about the rest. Consider the secretary problem with a pool of 28 candidates.

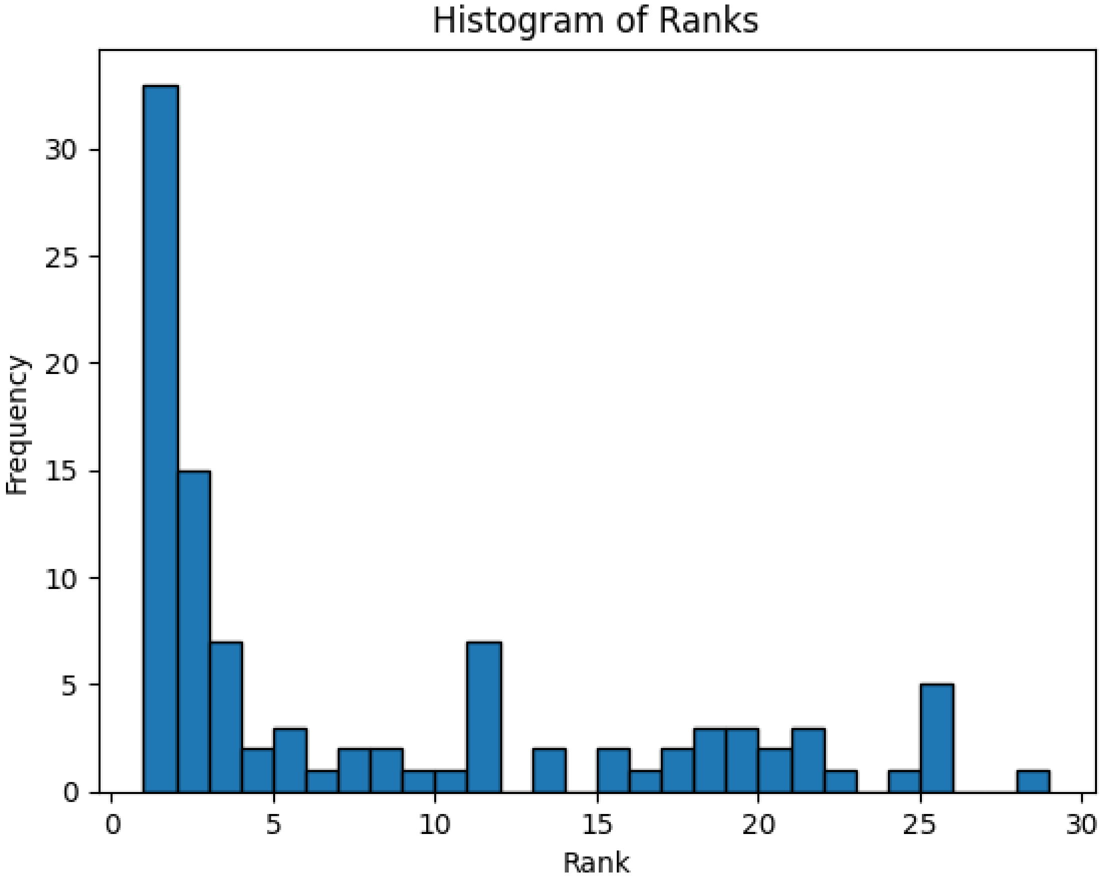


Figure 15.8: Histogram of rankings of final choices in Secretary Problem

Analysis of Iterations

Iteration 2

In the second iteration, the best candidate in the initial observation phase was ranked 8. During the subsequent decision phase, the algorithm evaluated candidates with ranks [17, 27, 22, 23, 2]. It stopped at the fifth candidate in this phase (overall 15th candidate), who had a rank of 2, better than the best candidate from the initial phase.

Iteration 3

The third iteration saw the best candidate in the initial phase with a rank of 1. After this phase, the algorithm reviewed candidates with ranks [2, 10, 27, 4, 24, 11, 5, 23, 12, 28, 3, 25, 16, 8, 17, 22, 21, 20]. The decision to stop was made at the 28th candidate, who had a rank of 20. This decision was made at the end of the candidate list as no candidate surpassed the best rank of 1 from the initial phase.

Iteration 4

In the fourth iteration, the best-ranked candidate during the initial phase was 7. The subsequent phase included candidates with ranks [24, 1]. The algorithm stopped at the second candidate in this phase (overall 12th candidate), who was ranked 1, surpassing the initial phase's best rank. These iterations demonstrate the variability in the stopping decisions based on the ranks of candidates encountered. The algorithm adapts its decision based on the best rank found in the initial observation phase and the ranks of subsequent candidates.

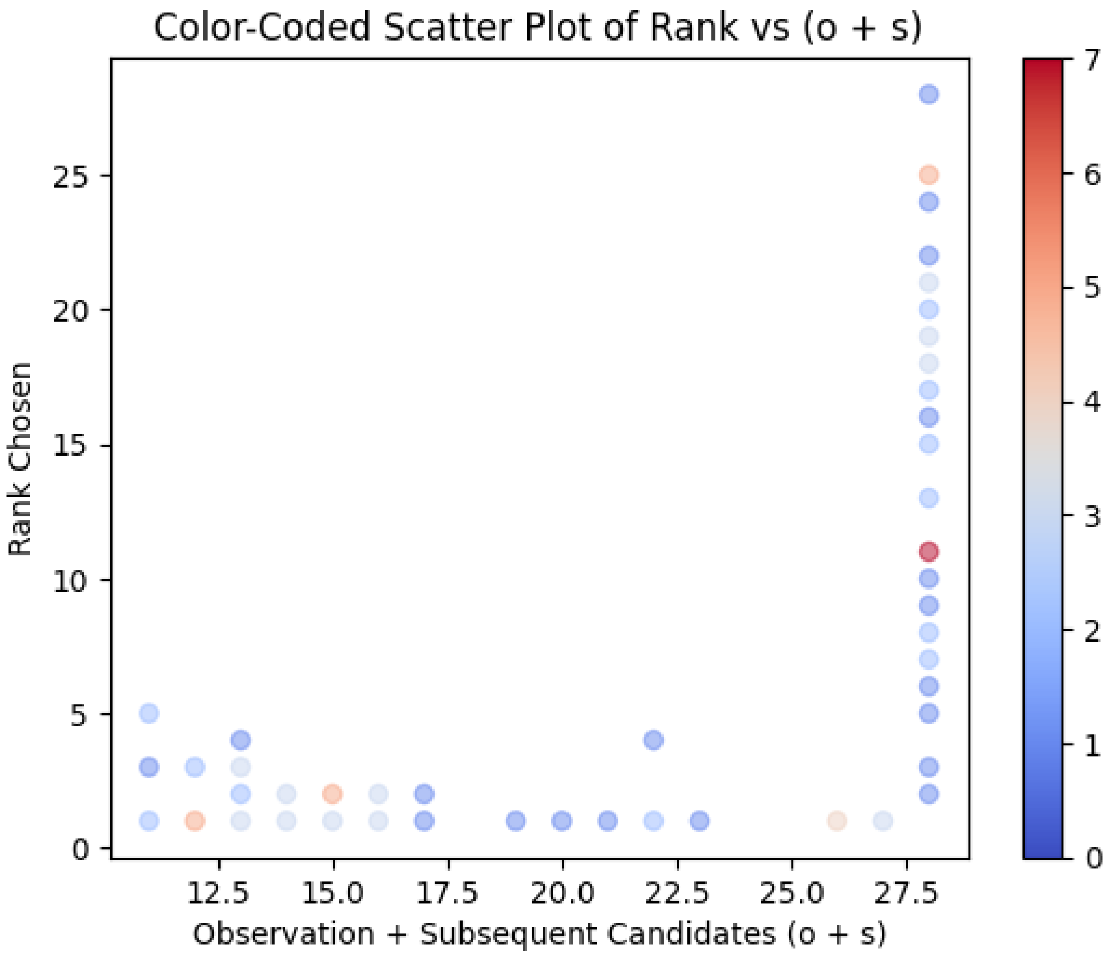


Figure 15.9: Heat gradient Scatter plot of rankings of final choices versus number of interviews in Secretary Problem

15.5.3 Balancing Exploration and Exploitation

The $\frac{1}{e}$ threshold represents an optimal balance between exploration and exploitation, akin to determining the right moment in hill-climbing when one should stop exploring and start exploiting. It's important to note the inherent differences between the two problems. The secretary problem involves a fixed sequence and irrevocable decisions, whereas hill-climbing allows for more flexible exploration and the possibility of backtracking.

We highlight here the importance of balancing information gathering with decision making, a principle applicable in various fields of problem-solving and optimization.

15.6 The Enigma Machine and its Plugboard Feature

The Enigma machine, a cryptographic device employed extensively by Nazi Germany during World War II, is one of the most iconic cipher machines in the history of cryptography. Designed as a rotor cipher machine, the Enigma was capable of transposing each letter inputted into another letter via a series of rotors and reflectors. Each rotor had 26 possible positions corresponding to the 26 letters of the alphabet, and these rotors would step forward with each keystroke, providing a changing encryption with every letter typed.

One of the defining features of the Enigma machine, and the one we focus on here, was its *plugboard* (known as the *Steckerbrett* in German). The plugboard acted as a preliminary layer of transposition before the electrical current of the keyed-in letter proceeded to the rotors. It consisted of a series of sockets, each corresponding to a letter of the alphabet. These sockets could be pairwise connected using cables. If, for instance, the letters A and B were connected via the plugboard, then pressing A would send a current that, at this initial stage, was rerouted as if B had been pressed, and vice versa.

Now, [21] if one were to consider a scenario where only a subset of the available plug holes were used, the question arises: How many different configurations are possible given a certain number of cables? Consider such a task of selecting m pairs from n objects. The following logic can be employed to derive the formula for this:

1. **Choosing the first pair:** For the first object in the pair, we have n choices. Once the first object has been selected, for the second object in the pair we have $n-1$ choices. Therefore, the total number of ways to choose the first pair is $n(n-1)$.
2. **Choosing subsequent pairs:** For the first object in the second pair (after two objects have already been used for the first pair), there are $n-2$ choices left. For its partner in the pair, $n-3$ choices remain. This gives $(n-2)(n-3)$ ways to choose the second pair after the first. This pattern continues for subsequent pairs.

Given this approach, the total number of ways to choose m pairs is:

$$n(n-1)(n-2)(n-3)\ldots(n-2m+2)(n-2m+1)$$

However, since the order in which we select the pairs doesn't matter (i.e., selecting the pairs AB and CD is the same as CD and AB), we must divide by $m!$. Additionally, because the order within each pair also doesn't matter (i.e., AB is the same as BA), we must further divide by 2^m to account for the 2 possible orders in each of the m pairs. Thus, the formula to compute the number of ways to select m pairs from n objects is:

$$\frac{n!}{(n-2m)!m!2^m}$$

For the Enigma's plugboard, with $n = 6$ holes and $m = 3$ cables, the formula we have

$$\frac{6!}{(6-6)!3!2^3} = \frac{6!}{0!3!8} = \frac{720}{1\times 6\times 8} = \frac{720}{48} = 15$$

For the Enigma's plugboard, with $n = 6$ holes and $m = 2$ cables, this formula gives:

$$\frac{6!}{(6-4)!2!2^2} = 45$$

indicating that there are 45 distinct ways to connect 2 cables in 6 plug holes.

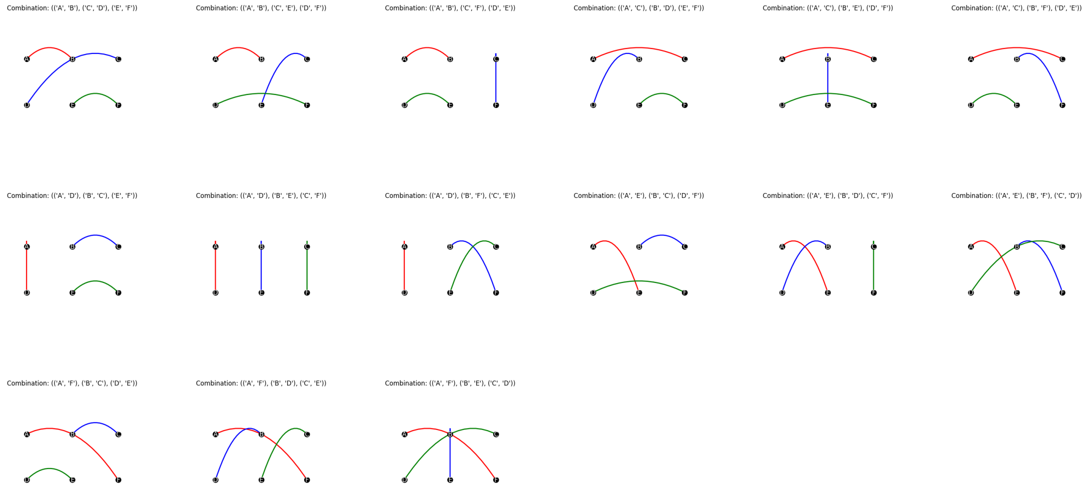

Combination: (('A', 'B'), ('C', 'D'), ('E', 'F'))
Combination: (('A', 'B'), ('C', 'E'), ('D', 'F'))
Combination: (('A', 'B'), ('C', 'F'), ('D', 'E'))
Combination: (('A', 'C'), ('B', 'D'), ('E', 'F'))
Combination: (('A', 'C'), ('B', 'E'), ('D', 'F'))
Combination: (('A', 'C'), ('B', 'F'), ('D', 'E'))
Combination: (('A', 'D'), ('B', 'C'), ('E', 'F'))
Combination: (('A', 'D'), ('B', 'E'), ('C', 'F'))
Combination: (('A', 'D'), ('B', 'F'), ('C', 'E'))
Combination: (('A', 'E'), ('B', 'C'), ('D', 'F'))
Combination: (('A', 'E'), ('B', 'D'), ('C', 'F'))
Combination: (('A', 'E'), ('B', 'F'), ('C', 'D'))
Combination: (('A', 'F'), ('B', 'C'), ('D', 'E'))
Combination: (('A', 'F'), ('B', 'D'), ('C', 'E'))
Combination: (('A', 'F'), ('B', 'E'), ('C', 'D'))

Figure 15.10: 15 Enigma Variations of 3 cables and 6 holes.

These combinations result from the code[3] snippet below drawing Bezier curves between pairs of points that symbolize the plugboard holes that are connected by cables in the Enigma machine.

```
for i, pair in enumerate(combo):
    x1, y1 = positions[pair[0]]
    x2, y2 = positions[pair[1]]
    # Calculate control point for Bezier curve
    ctrl_x = (x1 + x2) / 2
    ctrl_y = max(y1, y2) + 0.5  # 0.5 offset to give it a nice curve
```

This loop navigates through each pair of plugboard holes in the given combination (‘combo‘). The ‘enumerate‘ function provides an index ‘i‘ for each pair, which is used later to select a color for the curve. The positions of the two plugboard holes in the current pair are extracted and their positions are stored as (x, y) coordinates. For the Bezier curve, a control point is calculated. This control point influences the curve’s shape. The x-coordinate of the control point is the midpoint between ‘x1‘ and ‘x2‘. The y-coordinate is slightly above the highest of ‘y1‘ and ‘y2‘, ensured by the added offset of ‘0.5‘. The final line in the loop establishes the Bezier curve,

```
path = Path([(x1, y1), (ctrl_x, ctrl_y), (x2, y2)],
[Path.MOVETO, Path.CURVE3, Path.CURVE3])
patch = patches.PathPatch(path, facecolor=’none’, lw=2, edgecolor=colors[i])
ax.add_patch(patch)
```

in which a path is defined with a starting point, control point, and ending point. ‘Path.MOVETO‘ indicates the start, while ‘Path.CURVE3‘ denotes the use of a quadratic Bezier curve. A patch is then created from this path and added to the plotting area (‘ax‘).

[3] enigmaPlugboard.ipynb.

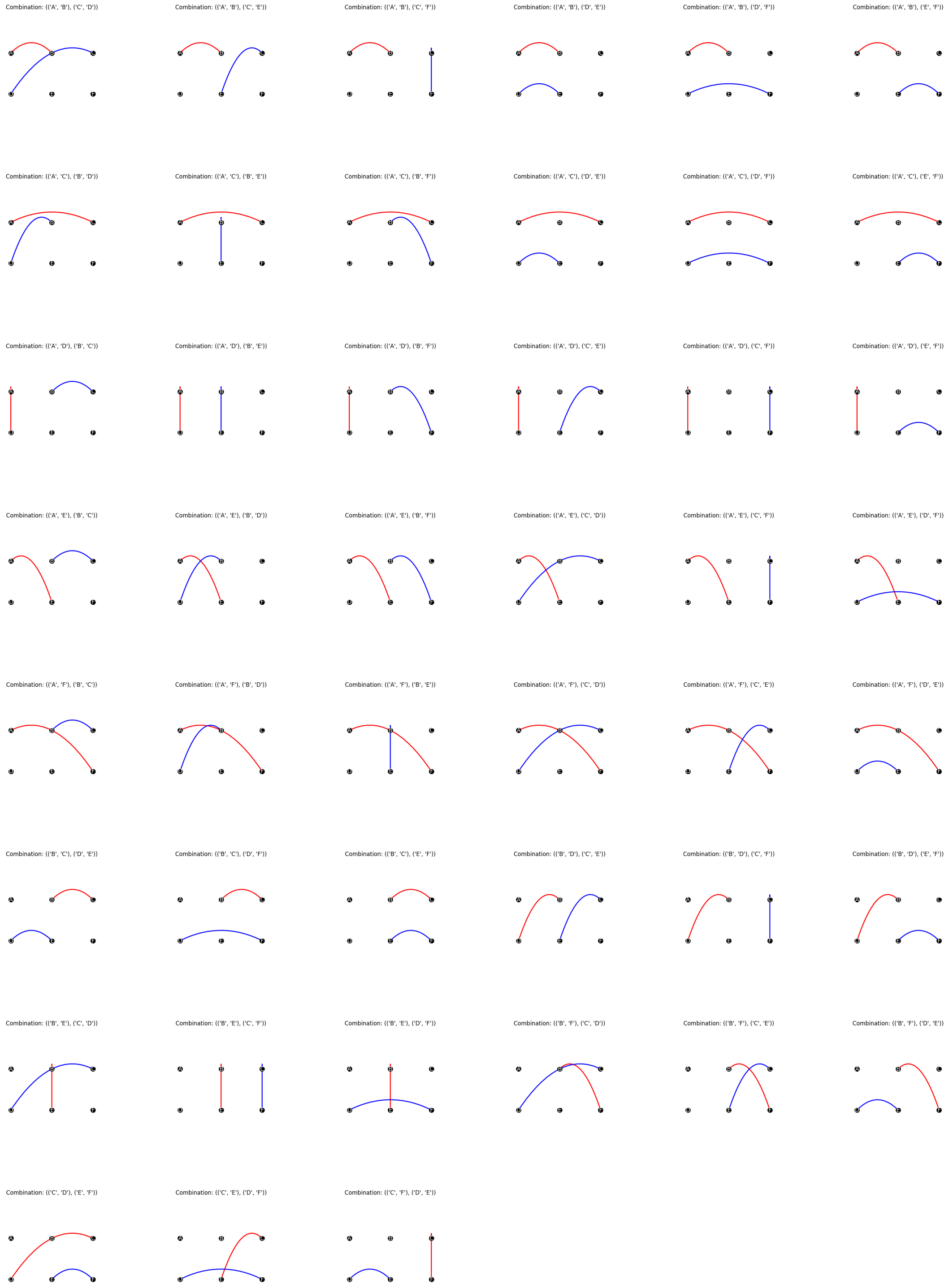

Combination: (('A', 'B'), ('C', 'D'))
Combination: (('A', 'B'), ('C', 'E'))
Combination: (('A', 'B'), ('C', 'F'))
Combination: (('A', 'B'), ('D', 'E'))
Combination: (('A', 'B'), ('D', 'F'))
Combination: (('A', 'B'), ('E', 'F'))
Combination: (('A', 'C'), ('B', 'D'))
Combination: (('A', 'C'), ('B', 'E'))
Combination: (('A', 'C'), ('B', 'F'))
Combination: (('A', 'C'), ('D', 'E'))
Combination: (('A', 'C'), ('D', 'F'))
Combination: (('A', 'C'), ('E', 'F'))
Combination: (('A', 'D'), ('B', 'C'))
Combination: (('A', 'D'), ('B', 'E'))
Combination: (('A', 'D'), ('B', 'F'))
Combination: (('A', 'D'), ('C', 'E'))
Combination: (('A', 'D'), ('C', 'F'))
Combination: (('A', 'D'), ('E', 'F'))
Combination: (('A', 'E'), ('B', 'C'))
Combination: (('A', 'E'), ('B', 'D'))
Combination: (('A', 'E'), ('B', 'F'))
Combination: (('A', 'E'), ('C', 'D'))
Combination: (('A', 'E'), ('C', 'F'))
Combination: (('A', 'E'), ('D', 'F'))
Combination: (('A', 'F'), ('B', 'C'))
Combination: (('A', 'F'), ('B', 'D'))
Combination: (('A', 'F'), ('B', 'E'))
Combination: (('A', 'F'), ('C', 'D'))
Combination: (('A', 'F'), ('C', 'E'))
Combination: (('A', 'F'), ('D', 'E'))
Combination: (('B', 'C'), ('D', 'E'))
Combination: (('B', 'C'), ('D', 'F'))
Combination: (('B', 'C'), ('E', 'F'))
Combination: (('B', 'D'), ('C', 'E'))
Combination: (('B', 'D'), ('C', 'F'))
Combination: (('B', 'D'), ('E', 'F'))
Combination: (('B', 'E'), ('C', 'D'))
Combination: (('B', 'E'), ('C', 'F'))
Combination: (('B', 'E'), ('D', 'F'))
Combination: (('B', 'F'), ('C', 'D'))
Combination: (('B', 'F'), ('C', 'E'))
Combination: (('B', 'F'), ('D', 'E'))
Combination: (('C', 'D'), ('E', 'F'))
Combination: (('C', 'E'), ('D', 'F'))
Combination: (('C', 'F'), ('D', 'E'))

Figure 15.11: 45 Enigma Variations of 2 cables and 6 holes.

15.7 Number of Triangles on an $n \times n$ Grid

The interplay between Combinatorics and geometry has long been a focal point of mathematical research, often producing insights that are both profound and elegant in their implications. A classic instance of this interplay is in the study of triangles formed on an $n \times n$ grid of dots, a problem not far removed from the more practical concerns of World War II cryptanalysis.

Imagine now our Enigma plugboard as an $n \times n$ grid. We adapt the problem of fitting cables to one of forming triangles on this grid, where each vertex of the triangle corresponds to a plug hole. The question becomes: In how many ways can we draw non-collinear triangles on this grid if two cables can attach to a common plug?

Each such triangle formed can be categorised its geometric properties, specifically its area and perimeter. We distinguish between two types of areas: the *Geometric Area*, which uses the classical formula for the area of a triangle, and the *Pick's Theorem Area*, which is a fascinating result that relates the area of a simple lattice polygon to the number of lattice points on its boundary and interior. Ultimately, our aim is to verify Pick's theorem for both right and scalene triangles. Validating this theorem for our grid-based triangles can lead us closer to a general formula, $T(n)$, which dictates the number of available triangles for a given $n \times n$ set of dots.

Purpose of the Code

The PickTriangleCombos.ipynb code visually investigate our problem. For a user-specified n, it:

- Calculates all possible non-collinear triangles on the grid.
- Computes the Geometric and Pick's Theorem areas for each triangle.
- Determines the perimeter for each triangle.
- Presents a visual representation of each triangle, annotated with its key properties.

This detailed combinatorial and geometric analysis will propel us towards understanding and, hopefully, proving our general formula $T(n)$.

In an $n \times n$ grid, there are n^2 points. Any 3 distinct points on this grid can either form a triangle or be colinear. A co-linear line is formed by three points in a straight line. On the other hand, a triangle is formed by any three non-colinear points.

Triangles Formed by Choosing Any 3 Points

- Each point on the grid can act as a vertex of a triangle.
- There are n^2 points on an $n \times n$ grid.
- The number of triangles you can form by choosing any 3 out of the n^2 points is found by the following considerations.

Given the n^2 total points:

1. First Point: n^2 choices.
2. Second Point: $n^2 - 1$ choices.
3. Third Point: $n^2 - 2$ choices.

The total ways to select these 3 points in sequence is:

$$n^2 \times (n^2 - 1) \times (n^2 - 2).$$

However, the order of selection does not matter, so we divide by 3! (since there are 3! ways to arrange 3 items):

$$\binom{n^2}{3} = \frac{n^2 \times (n^2 - 1) \times (n^2 - 2)}{3!}$$

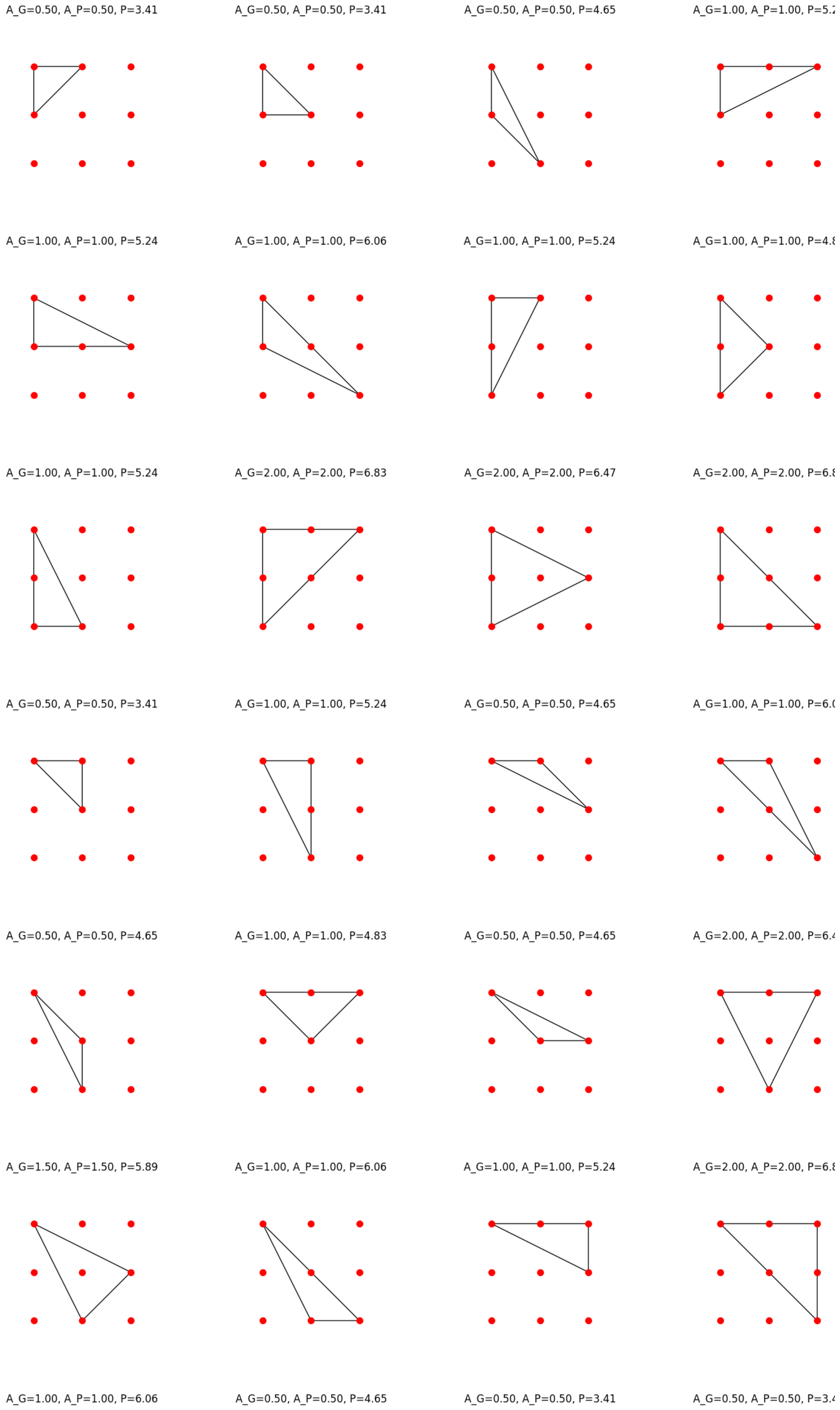

A_G=0.50, A_P=0.50, P=3.41
A_G=0.50, A_P=0.50, P=3.41
A_G=0.50, A_P=0.50, P=4.65
A_G=1.00, A_P=1.00, P=5.2
A_G=1.00, A_P=1.00, P=5.24
A_G=1.00, A_P=1.00, P=6.06
A_G=1.00, A_P=1.00, P=5.24
A_G=1.00, A_P=1.00, P=4.8
A_G=1.00, A_P=1.00, P=5.24
A_G=2.00, A_P=2.00, P=6.83
A_G=2.00, A_P=2.00, P=6.47
A_G=2.00, A_P=2.00, P=6.8
A_G=0.50, A_P=0.50, P=3.41
A_G=1.00, A_P=1.00, P=5.24
A_G=0.50, A_P=0.50, P=4.65
A_G=1.00, A_P=1.00, P=6.0
A_G=0.50, A_P=0.50, P=4.65
A_G=1.00, A_P=1.00, P=4.83
A_G=0.50, A_P=0.50, P=4.65
A_G=2.00, A_P=2.00, P=6.4
A_G=1.50, A_P=1.50, P=5.89
A_G=1.00, A_P=1.00, P=6.06
A_G=1.00, A_P=1.00, P=5.24
A_G=2.00, A_P=2.00, P=6.8
A_G=1.00, A_P=1.00, P=6.06
A_G=0.50, A_P=0.50, P=4.65
A_G=0.50, A_P=0.50, P=3.41
A_G=0.50, A_P=0.50, P=3.4

Figure 15.12: Some of the 76 distinct ways to draw triangles on 3x3 grid of dots.

Number of non Triangles Combinations for $n \times n$ Grid

- For random joined lines many of these combinations will not form triangles. Some will be three points lying on a straight line (col-linear points).
- For each row in the grid, the number of combinations of 3 col-linear points is $\binom{n}{3}$. Since there are n rows, the total from all rows is $n \times \binom{n}{3}$.
- Similarly, for each column, the number of combinations of 3 col-linear points is $\binom{n}{3}$. So the total from all columns is also $n \times \binom{n}{3}$.

Let's denote:

$$T(n) = \text{Number of triangles for } n \times n \text{ grid},$$
$$C(n) = \text{Number of co-linear lines for } n \times n \text{ grid}$$

The formula to calculate the number of co-linear lines is:

$$C(n) = \binom{n^2}{3} - \frac{n^2(n^2-1)(n^2-2)}{6}.$$

The code, loopingPickTriangleCombosHistogram.ipynb delivers stacked dot plots that collate similar and indeed congruent triangles according to their Perimeter to Area ratios.

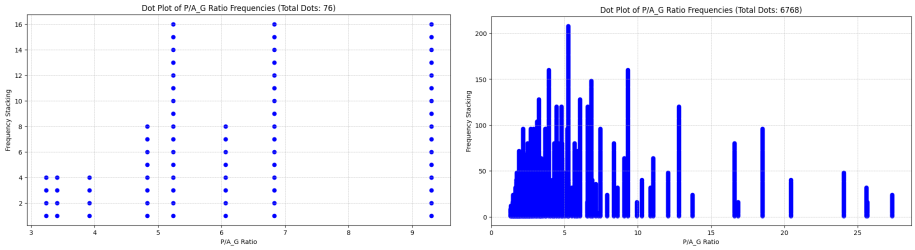


Figure 15.13: Dot plots of the 76 and 6768 ways of drawing triangles on 3x3 and 6x6 dotted grids.

Given our data for n and number of triangles, we can confirm our relationship:

n	Number of Co-linear lines,$C(n)$	Number of Triangles, $C(n)$
3	$\frac{3^2(3^2-1)(3^2-2)}{3!} - 76 = 8$	76
4	$\frac{4^2(4^2-1)(4^2-2)}{3!} - 516 = 44$	516
5	$\frac{5^2(5^2-1)(5^2-2)}{3!} - 2148 = 152$	2148
6	$\frac{6^2(6^2-1)(6^2-2)}{3!} - 6768 = 372$	6768
7	$\frac{7^2(7^2-1)(7^2-2)}{3!} - 17600 = 824$	17600
8	$\frac{8^2(8^2-1)(8^2-2)}{3!} - 40120 = 1544$	40120
9	$\frac{9^2(9^2-1)(9^2-2)}{3!} - 82608 = 2712$	82608
10	$\frac{10^2(10^2-1)(10^2-2)}{3!} - 157252 = 4448$	157252

Table 15.3: Summary Table for n and Number of Co-linear Lines

15.7.1 Perimeter to Area ratio Weighted-least squares

The code stackbubbleRegressionPickTriangle.ipynb delivers a bubble chart

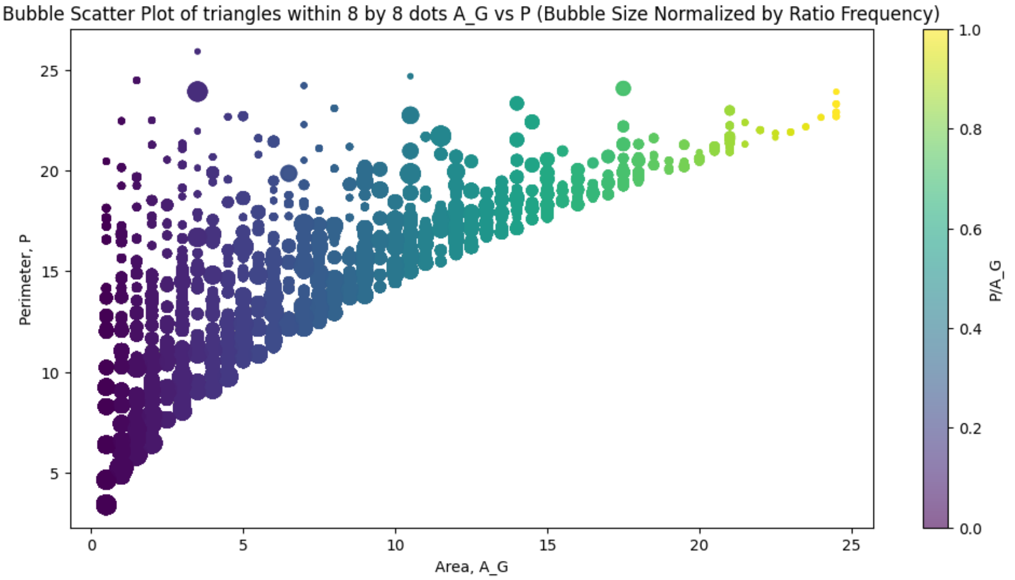


Figure 15.14: Bubble chart of the 40120 ways of drawing triangles on an 8x8 dotted grid.

as well as a weighted least regression chart

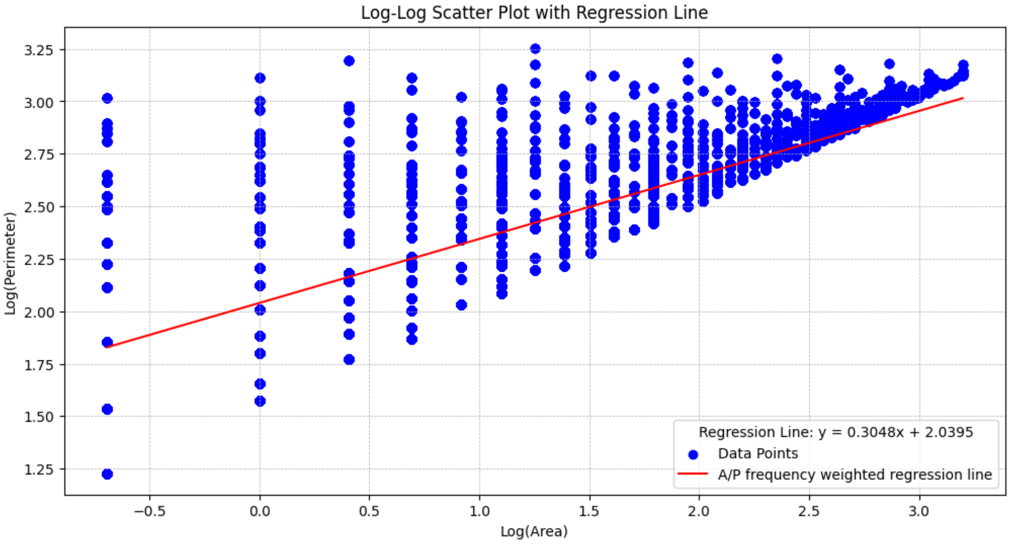


Figure 15.15: Dot plot with P/A weighted regression line of best fit.

15.7.2 log-plot of frequency of triangle types up to n

The code metaLoopedCumulativeDotPlotloopingPickTriangleCombos.ipynb delivers the following two plots

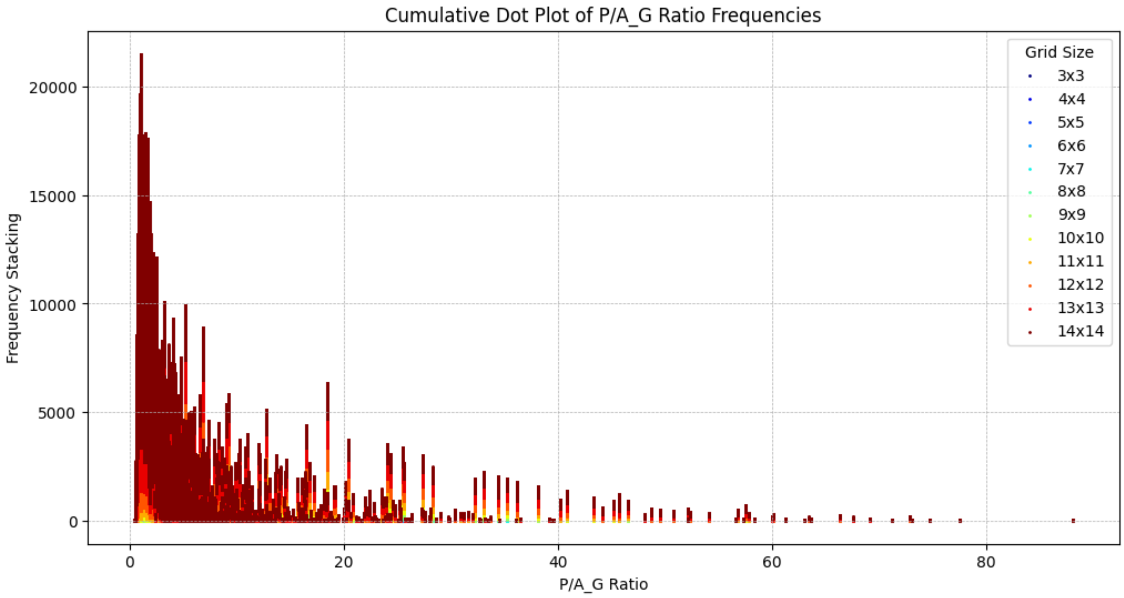


Figure 15.16: Stacked Dot plot of Cumulative frequency of triangle types as delineated by their P/A ratio.

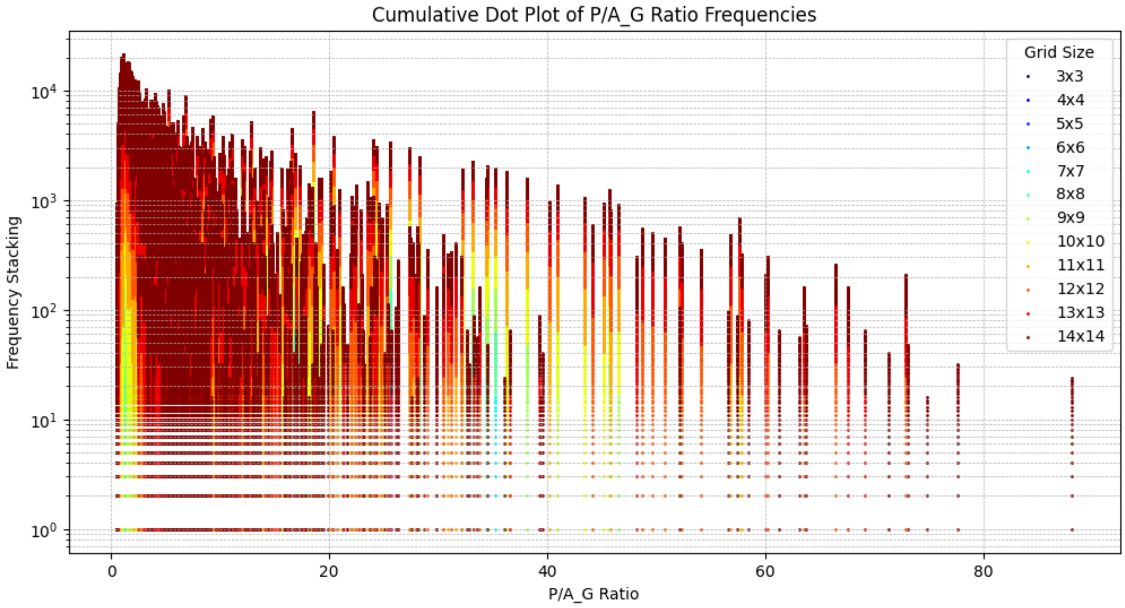


Figure 15.17: Stacked Dot plot of Cumulative Log-frequency of triangles types by their P/A ratio.

15.7.3 Factor Divisor Combinatorics

To determine the number of distinct combinations we might use the number of distinct combinations as given by the binomial coefficient for k items being chosen from n distinct items,

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Consider the set of numbers $\{a,b,c,d\}$ for which $a \neq b \neq c \neq d$. Our task is to determine the number of distinct combinations from either multiplying triplets with a singlet $\{(a \cdot b \cdot c)\}, \{d\}$ or two pairs with two pairs $\{(a \cdot b)\}, \{(c \cdot d)\}$ in which within () we have commutation so that $(a \cdot b) = (b \cdot a)$. For the triplet-singlet case, we have four distinct combinations $\{a \cdot b \cdot c\}$ and $\{d\}$, $\{a \cdot b \cdot d\}$ and $\{c\}$, $\{a \cdot d \cdot c\}$ and $\{b\}$, $\{d \cdot b \cdot c\}$ and $\{a\}$:

$$\begin{aligned}
&(a \cdot b \cdot c) \cdot d \\
&(a \cdot b \cdot d) \cdot c \\
&(a \cdot c \cdot d) \cdot b \\
&(b \cdot c \cdot d) \cdot a
\end{aligned}$$

For the pair-pair case, we have the three distinct combinations $\{a \cdot b\}$ and $\{c \cdot d\}$, $\{a \cdot c\}$ and $\{b \cdot d\}$, $\{a \cdot d\}$ and $\{c \cdot b\}$:

$$\begin{aligned}
&(a \cdot b) \cdot (c \cdot d) \\
&(a \cdot c) \cdot (b \cdot d) \\
&(a \cdot d) \cdot (c \cdot b)
\end{aligned}$$

Now, let's consider the "degenerate" case in which not all elements are distinct, such as $\{a,b,b,d\}$ [4] so that we will have to consider different sub-cases based on the multiplicity of elements. Let's denote the repeated elements as b_1 and b_2: For the triplet-singlet case we have:

$$\begin{aligned}
&(a \cdot b_1 \cdot b_2) \cdot d \\
&(a \cdot b_1 \cdot d) \cdot b_2 \\
&(a \cdot b_2 \cdot d) \cdot b_1 \\
&(b_1 \cdot b_2 \cdot d) \cdot a
\end{aligned}$$

only three of which are distinct for $b_1 = b_2$:

$$\begin{aligned}
&(a \cdot b_1 \cdot b_2) \cdot d \\
&(a \cdot b_1 \cdot d) \cdot b_2 \equiv (a \cdot b_2 \cdot d) \cdot b_1 \\
&(b_1 \cdot b_2 \cdot d) \cdot a
\end{aligned}$$

For the pair-pair case we have only two distinct cases:

$$\begin{aligned}
&(a \cdot b_1) \cdot (b_2 \cdot d) \equiv (a \cdot b_2) \cdot (b_1 \cdot d) \\
&(a \cdot d) \cdot (b_2 \cdot b_1)
\end{aligned}$$

That is, for the non-distinct case $\{a,b,b,d\}$, we have a total of $\binom{3}{2} + \binom{2}{1} = 3+2 = 5$ distinct combinations for binomial coefficient $\binom{n}{k}$.

[4] For the specific triplet-singlet case $\{2,3,3,5\}$ we have the equivalences: $(2 \cdot 3 \cdot 3) \cdot 5 = (3 \cdot 2 \cdot 3) \cdot 5 = (3 \cdot 3 \cdot 2) \cdot 5 = 5 \cdot (2 \cdot 3 \cdot 3) = 5 \cdot (3 \cdot 2 \cdot 3) = 5 \cdot (3 \cdot 3 \cdot 2)$, and for the pair-pair case we have the equivalences:$(2 \cdot 3) \cdot (3 \cdot 5) = (3 \cdot 2) \cdot (3 \cdot 5) = (2 \cdot 3) \cdot (5 \cdot 3) = (3 \cdot 2) \cdot (5 \cdot 3)$

Consider the set of numbers $\{a, b, c, d, e\}$. Our task is to determine the number of distinct combinations of either multiplying quad-singles $\{(a \cdot b \cdot c \cdot d)(e)\}$ or triplet-pairs $\{(a \cdot b \cdot c)(d \cdot e)\}$ in which within () we have commutation so that $a \cdot b = b \cdot a$. For the quad-singles case, there are 24 distinct combinations and for the triplet-pairs case, there are also 24 distinct combinations so in total number of distinct combinations is $24 + 24 = 48$:

$$\begin{array}{cccccc}
 & & & (a \cdot b \cdot c) \cdot (d \cdot e) & (a \cdot c \cdot b) \cdot (d \cdot e) & (a \cdot d \cdot b) \cdot (c \cdot e) \\
(a \cdot b \cdot c \cdot d) \cdot e & (a \cdot b \cdot d \cdot c) \cdot e & (a \cdot c \cdot b \cdot d) \cdot e & (a \cdot b \cdot d) \cdot (c \cdot e) & (a \cdot c \cdot d) \cdot (b \cdot e) & (a \cdot d \cdot c) \cdot (b \cdot e) \\
(a \cdot c \cdot d \cdot b) \cdot e & (a \cdot d \cdot b \cdot c) \cdot e & (a \cdot d \cdot c \cdot b) \cdot e & (b \cdot a \cdot c) \cdot (d \cdot e) & (b \cdot c \cdot a) \cdot (d \cdot e) & (b \cdot d \cdot a) \cdot (c \cdot e) \\
(b \cdot a \cdot c \cdot d) \cdot e & (b \cdot a \cdot d \cdot c) \cdot e & (b \cdot c \cdot a \cdot d) \cdot e & (b \cdot a \cdot d) \cdot (c \cdot e) & (b \cdot c \cdot d) \cdot (a \cdot e) & (b \cdot d \cdot c) \cdot (a \cdot e) \\
(b \cdot d \cdot a \cdot c) \cdot e & (b \cdot d \cdot c \cdot a) \cdot e & (c \cdot a \cdot b \cdot d) \cdot e & (c \cdot a \cdot b) \cdot (d \cdot e) & (c \cdot b \cdot a) \cdot (d \cdot e) & (c \cdot d \cdot a) \cdot (b \cdot e) \\
(c \cdot b \cdot d \cdot a) \cdot e & (c \cdot d \cdot a \cdot b) \cdot e & (c \cdot d \cdot b \cdot a) \cdot e & (c \cdot a \cdot d) \cdot (b \cdot e) & (c \cdot b \cdot d) \cdot (a \cdot e) & (c \cdot d \cdot b) \cdot (a \cdot e) \\
(d \cdot a \cdot b \cdot c) \cdot e & (d \cdot c \cdot b \cdot a) \cdot e & (d \cdot b \cdot c \cdot a) \cdot e & (d \cdot a \cdot b) \cdot (c \cdot e) & (d \cdot c \cdot a) \cdot (b \cdot e) & (d \cdot b \cdot c) \cdot (a \cdot e) \\
 & & & (d \cdot a \cdot c) \cdot (b \cdot e) & (d \cdot c \cdot b) \cdot (a \cdot e) & (d \cdot b \cdot a) \cdot (c \cdot e)
\end{array}$$

For the degenerate case $\{a, a, c, c, e\}$, we have the 15 distinct combinations:

$$\begin{array}{ccc}
(a \cdot a \cdot c \cdot c) \cdot e & (a \cdot c \cdot a \cdot c) \cdot e & (a \cdot c \cdot c \cdot a) \cdot e \\
(c \cdot a \cdot a \cdot c) \cdot e & (c \cdot a \cdot c \cdot a) \cdot e & (c \cdot c \cdot a \cdot a) \cdot e \\
(a \cdot a \cdot c) \cdot (c \cdot e) & (a \cdot c \cdot a) \cdot (c \cdot e) & (c \cdot a \cdot a) \cdot (c \cdot e) \\
(a \cdot a \cdot c) \cdot (e \cdot c) & (a \cdot c \cdot a) \cdot (e \cdot c) & (c \cdot a \cdot a) \cdot (e \cdot c) \\
(a \cdot c \cdot c) \cdot (a \cdot e) & (c \cdot a \cdot c) \cdot (a \cdot e) & (c \cdot c \cdot a) \cdot (a \cdot e)
\end{array}$$

Stirling Numbers

Stirling numbers are combinatorial numbers that arise in various counting problems. They are categorized into two kinds:

1. **Stirling Numbers of the First Kind $S(n, k)$:** Counts the number of permutations of n elements with exactly k permutation cycles.
2. **Stirling Numbers of the Second Kind $\left\{ {n \atop k} \right\}$:** Counts the number of ways to partition a set of n elements into k non-empty subsets.

Connection to Composite Numbers and Their Divisors

If we are considering the number of distinct rectangles that can be formed using the divisors of a composite number (including the number itself), the relevant concept is partitions of a set. This is because, given a set of divisors, we are essentially partitioning them into subsets representing different rectangles.

Thus, for this problem, the **Stirling Numbers of the Second Kind** are applicable, as they deal with partitioning sets into non-empty subsets.

For example, given a composite number with n divisors (including the number itself), the total number of ways to form distinct rectangles by partitioning these divisors is represented by the sum:

$$\sum_{k=1}^{n} \left\{ {n \atop k} \right\}$$

where the summation runs through all possible partitions (or rectangle configurations) using the divisors.

When forming rectangles from divisors of numbers with squared primes, neither Stirling numbers of the second kind nor partition numbers directly address the issue due to divisor redundancies. A customized approach considering prime multiplicities is essential.

15.8 Linear Algebra and Dimensional Analysis

Dimensional Analysis of Fundamental Constants

Our micro and macro universe is governed by four fundamental constants:

- c (speed of light) has units $[L/T]$
- G (gravitational constant) has units $[L^3/M/T^2]$
- H (Hubble constant) has units $[1/T]$
- h (Planck constant) has units $[M \times L^2/T]$

We are interested in forming combinations of these constants to construct unit-full measures of mass [M], length [L], and time [T]. Our approach is to use dimensional analysis to express the mass scale m as:

$$m \sim c^{\alpha} h^{\beta} H^{\gamma} G^{\delta}$$

In order to determine the exponents α, β, γ, and δ by equating the units on both sides for each of the fundamental quantities [M], [L], and [T]. This results in a system of equations that can be solved for the unknowns in terms of one of them, say α. Solving for these exponents, allows us to understand the relations and combinations of these constants to produce different mass scales. This is of particular interest in cosmology and high-energy physics, where the interplay between the scales determined by these constants provides insights into the structure and evolution of the universe.

15.8.1 Mass of Universe

By first assuming that $\beta = 0$ (so that we have a well determined system of three unknowns in three equations) we rewrite our constants in terms of their units to an unknown power:

$$c^{\alpha} = L^{\alpha} T^{-\alpha}$$
$$H^{\gamma} = T^{-\gamma}$$
$$G^{\delta} = M^{-\delta} L^{3\delta} T^{-2\delta}$$

The dimensions of a product of the quantities raised to powers is the product of the dimensions raised to the same powers. So, the system of equations for the dimensions is:

For $[M]$:

$$0 \times \alpha - 0 + 0 - \delta = 0 \implies -\delta = 0 \tag{15.1}$$

For $[L]$:

$$1 \times \alpha - 0 + 0 + 3\delta = 0 \implies \alpha + 3\delta = 0 \tag{15.2}$$

For $[T]$:

$$-1 \times \alpha - \gamma - 2\delta = 0 \implies -\alpha - \gamma - 2\delta = 0 \tag{15.3}$$

From these equations, you can extract the coefficients to form the matrix and proceed to solve for the values of α, γ, and δ using Gaussian elimination:

$$A \begin{bmatrix} \alpha \\ \beta \\ \delta \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{15.4}$$

in which A, the coefficient matrix is given by:

$$A = \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & 3 \\ -1 & -1 & -2, \end{bmatrix} \tag{15.5}$$

and find the solution when A is non-singular, by determining the inverse of A:

$$\begin{bmatrix} \alpha \\ \beta \\ \delta \end{bmatrix} = A^{-1} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{15.6}$$

A's determinant determines the nature of the solutions of our system:

- If $\det(A) = 0$, the matrix A is singular, implying our system of equations either has no solution or an infinite number of solutions.
- If $\det(A) \neq 0$, the matrix A is non-singular and there is a unique solution to the system which can be found using linear algebra methods.

We use python to construct our system of three equations in three unknowns as a matrix equation, :

```
import numpy as np
A = np.array([
    [0, 0, -1],    # coefficients from equation for [M]
    [1, 0, 3],     # coefficients from equation for [L]
    [-1, -1, -2]   # coefficients from equation for [T]
])
B = np.array([1, 0, 0])  # right-hand sides
# Solve using Gaussian elimination
alpha, gamma, delta = np.linalg.solve(A, B)
```

and we use the `Fraction`module to write our powers as fractional indices:

```
from fractions import Fraction
alpha_frac = Fraction(alpha).limit_denominator()
gamma_frac = Fraction(gamma).limit_denominator()
delta_frac = Fraction(delta).limit_denominator()
```

The output is

$$m \sim c^3 H^{-1} G^{-1} = \frac{c^3}{HG} \equiv m_1$$

By implementing a symbolic computation tool, we can efficiently determine the exponents for various combinations, aiding in our understanding of the universe's fundamental scales. This is a whole lot simpler than developing a whole Gaussian elimination routine yourself as per python

```
def Gaussian_elimination_with_pivoting(A, b):
    n = len(b)
    # Forward elimination with pivoting
    for k in range(n):
        # Pivot for largest value in column
        max_row = max(range(k, n), key=lambda i: abs(A[i,k]))
        A[[k, max_row]] = A[[max_row, k]]
        b[k], b[max_row] = b[max_row], b[k]
        # Check for zero pivot
        if np.abs(A[k,k]) < 1.0e-12:
            return None  # Matrix is singular, no unique solution

        for i in range(k+1, n):
            factor = A[i,k] / A[k,k]
            for j in range(k, n):
                A[i,j] -= factor * A[k,j]
            b[i] -= factor * b[k]
    # Back substitution
    x = np.zeros(n)
    x[n-1] = b[n-1] / A[n-1, n-1]
    for i in range(n-2, -1, -1):
        sum_ = b[i]
        for j in range(i+1, n):
            sum_ -= A[i,j] * x[j]
        x[i] = sum_ / A[i,i]
    return x
```

15.8.2 Planck Mass

We can insist insist that a priori $\gamma = 0$, so that our coefficient matrix, A is given by:

$$A = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 2 & 3 \\ -1 & -1 & -2 \end{bmatrix} \tag{15.7}$$

We have our code calculate the determinant then check for singularities:

```
det_A = np.linalg.det(A)
if det_A == 0:
    print("\nMatrix A is singular and our system of equations either has no solutions or ar
else:
    print("\nMatrix A is non-singular so can proceed to solve the system of equations.")
print("\nDeterminant of A:", det_A)
# Step 4: Solve if non-singular
if det_A != 0:
    solution = np.linalg.solve(A, B)
    alpha, beta, delta = solution
```

The output is

$$m_4 = c^{1/2} h^{1/2} G^{-1/2} = \left(\frac{hc}{G}\right)^{1/2} \equiv m_P$$

The mass, m_4 is referred to as the Planck mass, $m_4 \equiv m_P$ while m_1 is the amount of Baryonic matter in a Hubble sphere universe (mass of the universe $m_U = 1.5 \times 10^{53}$ kg.) and m_2 by virtue of

$m_2c^2 = hH$ is the self-gravitational potential mass-energy or the graviton mass. These masses span 121 orders of magnitude (comparable to the discrepancy between QFT's estimation of the vacuum zero point energy and the observed dark energy density value). Such coincidences are likely to abound - as we will shortly see - there are many more masses from where these came from if we entertain forming their geometric means $m_{12..n} = \sqrt[n]{m_1 \times m_2 \times ... \times m_n}$.

Undetermined Systems

To illustrate the under-determined system, let's first determine the equations for the units:

- c (speed of light) has units $[L/T]$, which leads to the equation:

$$\alpha - \gamma = 0 \tag{15.8}$$

- G (gravitational constant) has units $[L^3/M/T^2]$, which leads to:

$$\alpha + 3\delta - \beta - 2\gamma = 0 \tag{15.9}$$

- H (Hubble constant) has units $[1/T]$, which results in:

$$-\gamma = 0 \tag{15.10}$$

- h (Planck constant) has units $[M \times L^2/T]$, which leads to:

$$2\alpha + \beta - \gamma = 0 \tag{15.11}$$

For the sake of demonstration, we'll pick only three of the equations (to match the number of fundamental units: $[M]$, $[L]$, $[T]$). We'll form a 3×4 matrix and solve it, expecting to get a free variable.

the code

Let's find the values of α that satisfy the given expressions for the masses m_1, m_2, m_3, and m_4 using the equations we derived earlier.

Given the general relations from our dimensional analysis:

1. $\beta - \delta = 1$ (from [M] units)
2. $\alpha + 2\beta + 3\delta = 0$ (from [L] units)
3. $-\alpha - \beta - \gamma - 2\delta = 0$ (from [T] units)

We find the general solutions:

$$\beta = \frac{3}{5} - \frac{\alpha}{5}$$
$$\delta = -\frac{\alpha}{5} - \frac{2}{5}$$
$$\gamma = \frac{1}{5} - \frac{2\alpha}{5}$$

Four combinations of each of three fundamental units $[M][L][T]$ out of four unknowns indices, $\alpha, \beta, \gamma, \delta$ are:

$$m_1 = \frac{c^3}{GH} \sim 1.8 \times 10^{53}\,\mathrm{kg}$$

$$m_2 = \frac{hH}{c^2} \sim 1.6 \times 10^{-68}\,\mathrm{kg}$$

$$m_3 = \left(\frac{h^3 H}{G^2}\right)^{1/5} \sim 4.3 \times 10^{-20}\,\text{kg}$$

$$m_4 = \left(\frac{hc}{G}\right)^{1/2} \sim 5.456 \times 10^{-8}\,\text{kg}$$

To derive the values of α that generate these masses, we equate the powers of c, h, H, and G in each mass expression to α, β, γ, and δ respectively.

By substituting the known values of α into our derived expressions, we can confirm the consistency of our equations and validate the given mass scales.

For the given masses, the values of α and corresponding β, γ, δ are:

$$\begin{array}{llll}
\alpha = 3 & \beta = 0 & \gamma = -1 & \delta = -1 \\
\alpha = -2 & \beta = 1 & \gamma = 1 & \delta = 0 \\
\alpha = 0 & \beta = \frac{3}{5} & \gamma = \frac{1}{5} & \delta = -\frac{2}{5} \\
\alpha = 1 & \beta = \frac{2}{5} & \gamma = -\frac{1}{5} & \delta = -\frac{3}{5}
\end{array}$$

There are $\binom{4}{3} = 4$ masses that can be simply formed as combinations of three of the four fundamental constants of nature, c, G, H, h determined by a dimensional analysis of the system of three linear equations, $m \sim c^{\alpha} h^{\beta} H^{\gamma} G^{\delta}$.

There are ${}^4_2C = 6$ ways to form the geometric mean of the first four masses, m_1, m_2, m_3, m_4 comprising 3 of the four fundamental constants ($m_{12} = \sqrt{m_1 \times m_2}$) etc:

$$m_{12} = m_4 = \left(\frac{hc}{G}\right)^{1/2} \equiv m_P$$

$$m_{13} = \left(\frac{c^{15} h^3}{G^7 H^4}\right)^{1/10} \sim 8.9 \times 10^{16}\,\text{kg}$$

$$m_{14} = \left(\frac{c^7 h}{G^3 H^2}\right)^{1/4} \sim 1 \times 10^{23}\,\text{kg}$$

$$m_{23} = \left(\frac{H^6 h^8}{G^2 c^{10}}\right)^{1/10} \sim 2.6 \times 10^{-44}\,\text{kg}$$

$$m_{24} = \left(\frac{H^2 h^3}{G c^3}\right)^{1/4} \sim 3 \times 10^{-38}\,\text{kg}$$

$$m_{34} = \left(\frac{H^2 h^{11} c^5}{G^9}\right)^{1/20} \sim 4.83 \times 10^{-14}\,\text{kg}$$

For $m12$: $\alpha = \frac{3}{2}, \beta = \frac{1}{2}, \gamma = 0, \delta = \frac{-3}{2}$
For $m13$: $\alpha = \frac{9}{5}, \beta = \frac{3}{10}, \gamma = \frac{-2}{5}, \delta = \frac{-13}{10}$
For $m14$: $\alpha = \frac{7}{4}, \beta = \frac{1}{4}, \gamma = \frac{-1}{2}, \delta = -1$
For $m23$: $\alpha = \frac{3}{10}, \beta = \frac{4}{5}, \gamma = \frac{3}{5}, \delta = \frac{-9}{5}$
For $m24$: $\alpha = \frac{1}{4}, \beta = \frac{3}{4}, \gamma = \frac{1}{2}, \delta = \frac{-3}{2}$
For $m34$: $\alpha = \frac{11}{20}, \beta = \frac{11}{20}, \gamma = \frac{1}{10}, \delta = \frac{-13}{10}$

The Planck mass, m_P is the geometric mean of graviton, m_2 and baryonic universe mass, m_1. Given its relative simplicity one might entertain $m_{24} = m_\nu$ with the maximum possible mass of the lightest neutrino at 0.086 eV, which is equivalent to 1.5×10^{-37} kg. There are a further ${}^4_3C = 4$ ways to form the geometric mean as combinations of three from m_1, m_2, m_3, m_4:

$$m_{123} = \left(\frac{Hh^8c^5}{G^7}\right)^{1/15} \sim 5.03 \times 10^{-12}\,\text{kg}$$

$$m_{134} = \left(\frac{h^{11}c^{35}}{G^{19}H^8}\right)^{1/30} \sim 7.5 \times 10^{8}\,\text{kg}$$

$$m_{234} = \left(\frac{h^{21}H^{12}}{c^{15}G^9}\right)^{1/30} \sim 3.4 \times 10^{-32}\,\text{kg}$$

$$m_{124} = \left(\frac{hc}{G}\right)^{1/2} \equiv m_4$$

We see that m_4 circuitously is just the geometric mean of the three masses m_1, m_2, m_4 while m_{234} is of the order of the electron mass, $m_e = 9.11 \times 10^{-31}$ kg. All the other means involve all four constants but are a little contrived in structure.

15.8.3 Weinberg's Mass

Additionally, the literature often quotes a fifth *fundamental* mass as espoused by Weinberg, of the order of a meson mass $\pi_0 = 2.4 \times 10^{-28}$ kg, $K_0 = 8.9 \times 10^{-28}$ kg, $\eta = 9.8 \times 10^{-28}$ kg or a muon $m_\mu = 106\,\text{MeV/c}^2 = 1.9 \times 10^{-28}$ kg) comprised more simply of all four constants (the $2/3$ refers to a choice of α explained below):

$m_5^{2/3} \equiv m_W = \left(\frac{h^2H}{Gc}\right)^{1/3} \sim 3.6 \times 10^{-28}\,\text{kg}.$

Weinberg is solving an undetermined system of three linear equations in the three units of $[M][L][T]$ for the four unknowns of $m \sim h^\alpha H^\beta G^\gamma c^\delta$. As such, there are manifold solutions defined by the constraint $\alpha = \frac{\beta+1}{2} = \frac{3-\delta}{5} = \gamma + 1$ for $\alpha \neq 0, \beta \neq 0, \gamma \neq 0, \delta \neq 0$. Weinberg's appealing choice is to take $\alpha = \frac{2}{3}, \beta = \frac{1}{3}, \gamma = -\frac{1}{3}, \delta = -\frac{1}{3}$. Choosing rather the solution plane defined by $\alpha = \frac{3}{4}$, we have then $\beta = \frac{1}{2}, \gamma = -\frac{1}{4}, \delta = -\frac{3}{4}$, according to the second line in the table captured below:

So giving something more of the order 10^{-38} of a neutrino mass,

$m_5^{3/4} \equiv m_{24} = \left(\frac{h^3H^2}{Gc^3}\right)^{1/4} \sim 3 \times 10^{-38}\,\text{kg}.$

As you can see, you just need to choose your rational exponent α for h to conjure an interesting mass value. That is, if you attribute some geometry to justify using a nice-looking rational. If we vary the exponent α (A in the graph below), we can see that $\alpha = \frac{2}{3}$ delivers near the muon value:

$$m_{135} = \left(\frac{c^{40}h^{19}}{G^{26}H^{7}}\right)^{1/45} \sim 142\,\mathrm{kg}$$

$$m_{145} = \left(\frac{c^{19}h^{7}}{G^{11}H^{4}}\right)^{1/18} \sim 1.54 \times 10^{6}\,\mathrm{kg}$$

$$m_{235} = \left(\frac{H^{23}h^{34}}{G^{11}c^{35}}\right)^{1/45} \sim 6.3 \times 10^{-39}\,\mathrm{kg}$$

$$m_{245} = \left(\frac{H^{8}h^{13}}{c^{11}G^{5}}\right)^{1/18} \sim 6.8 \times 10^{-35}\,\mathrm{kg}$$

$$m_{125} = \left(\frac{Hh^{5}c^{2}}{G^{4}}\right)^{1/9} \sim 1.03 \times 10^{-14}\,\mathrm{kg}$$

$$m_{345} = \left(\frac{H^{16}h^{53}c^{5}}{G^{37}}\right)^{1/90} \sim 9.5 \times 10^{-19}\,\mathrm{kg}$$

An unseemly bunch. There is one combination that is the geometric mean of m_1, m_2, m_3, m_4:

$$m_{1234} = \left(\frac{H^{2}h^{21}c^{15}}{G^{19}}\right)^{1/40} \sim 5.13 \times 10^{-11}\,\mathrm{kg}$$

There are ${}^{5}_{4}C = 5$ ways to form the geometric mean as combinations of four from m_1, m_2, m_3, m_4, m_5, so there are a further four masses:

$$m_{1345} = \left(\frac{h^{53}c^{95}}{G^{67}H^{1}4}\right)^{1/120} \sim 0.629\,\mathrm{kg}$$

$$m_{1235} = \left(\frac{H^{8}h^{34}c^{10}}{G^{26}}\right)^{1/60} \sim 4.64 \times 10^{-16}\,\mathrm{kg}$$

$$m_{2345} = \left(\frac{H^{46}h^{83}}{c^{55}G^{37}}\right)^{1/120} \sim 3.4 \times 10^{-31}\,\mathrm{kg}$$

$$m_{1245} = \left(\frac{H^{2}h^{13}c^{7}}{G^{11}}\right)^{1/24} \sim 4.93 \times 10^{-13}\,\mathrm{kg}$$

There is one combination that is the geometric mean of m_1, m_2, m_3, m_4, m_5:

$$m_{12345} = \left(\frac{H^{16}h^{83}c^{35}}{G^{67}}\right)^{1/150} \sim 1.91 \times 10^{-14}\,\mathrm{kg}$$

Introduction to the Hat Check Problem

The Hat check problem is a classic problem in combinatorial probability theory:

Problem Statement: n people attend a party and check their hats. Due to a gust of wind, the hats are blown away and then randomly redistributed. What is the probability that no one receives their own hat back?

One way to solve the problem is by considering the concept of "derangements".

Definition 25 *Derangement* is a permutation of a set where no element appears in its original position. It's essentially a complete reshuffling of the elements. Also known as "complete permutations"

The probability that no one receives their own hat back can be expressed in terms of derangements as follows. We count the number of permutations where no element is in its original position, then divide it by the total number of permutations. Denoting the number of derangements of n hats by $!n$, or D_n for large values of n, this looks like:

$$\lim_{n\to\infty} \frac{D_n}{n!} = \frac{1}{e}$$

This implies that as n becomes large, the number of derangements D_n approaches $\frac{n!}{e}$. Now, let's relate this to the probability P_n that no one gets their own hat back when there are n people. The probability P_n can be expressed as the ratio of the number of derangements to the total number of permutations, $P_n = \frac{D_n}{n!}$. Substituting the approximate expression for D_n, we get that as the number of hats, (n approaches infinity, the probability that no one gets their own hat back is:

$$P_n \approx \frac{\frac{n!}{e}}{n!} = \frac{1}{e}$$

hatCheckProbelm-derangments.ipynb delivers this limiting result in the form of a scatterplot,

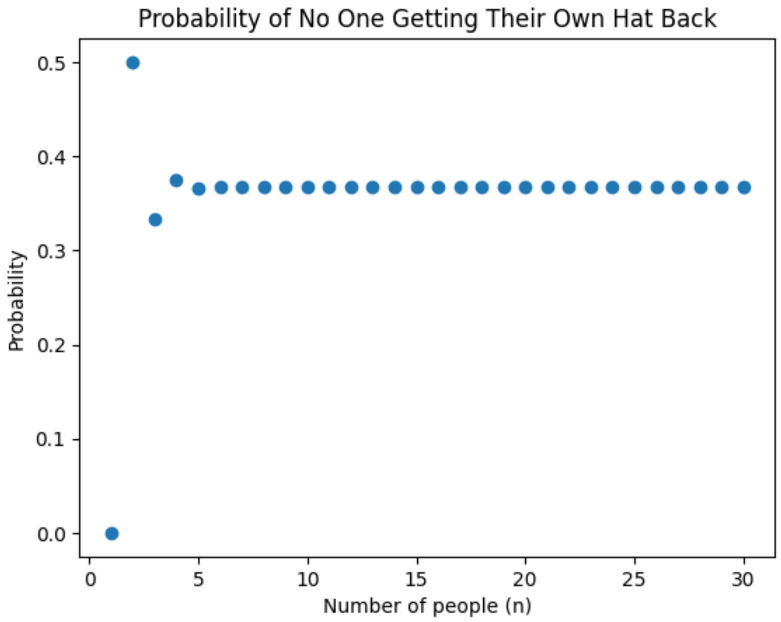


Figure 15.18: Hat check probabilities for n hatted people.

15.8.4 Recursive formula

We will now derive a recursive formula for the probability P_n by first establishing some notation:

- P_n is the probability that no one gets their own hat back when there are n people.
- p_n is the probability that the nth person does not get their own hat back for n people.

Given these definitions, $p_1 = 0$ because if there is only one person, they are guaranteed to get their own hat back, and $p_2 = \frac{1}{2}$ because when there are two people, there are only two possible arrangements, one in which they both get their own hats back and one in which they don't. Thus, the probability is $\frac{1}{2}$ in this case. Now, let's consider P_n, the probability that no one gets their own hat back when there are n people. We can express P_n in terms of p_n as $P_n = p_n \cdot P_{n-1}$.

Because for each valid permutation of n people where the nth person does not get their own hat back, the remaining $n-1$ people must also not get their own hats back. Hence, the probability of n people not getting their own hats back is the probability of the nth person not getting their hat back multiplied by the probability that the first $n-1$ people do not get their own hats back. Now, we need to find the recursive formula for p_n.

Consider the nth person:

1. The probability that the nth person does not get their own hat back is the probability that they get one of the other $n-1$ hats out of the total n hats. This probability is $\frac{n-1}{n}$.
2. Or, the nth person gets their own hat back, but in that case, the $(n-1)$th person must not get their hat back. The probability of the $(n-1)$th person not getting their hat back is p_{n-1}, and the probability that the nth person gets their own hat back is $\frac{1}{n}$.

Thus, the recursive formula for p_n is:

$$p_n = \frac{n-1}{n} \cdot p_{n-1} + \frac{1}{n} \cdot P_{n-2}$$

Combining the expressions for P_n and p_n, we have:

$$P_n = \frac{n-1}{n} \cdot p_{n-1} \cdot P_{n-1} + \frac{1}{n} \cdot P_{n-2}$$

Given the base cases $p_1 = 0$ and $p_2 = \frac{1}{2}$, we can use these recursive formulas to compute the probability P_n for any n where recall that p_n is the probability that the nth person does not get their own hat back, and P_n is the overall probability that no one gets their own hat back when there are n people. For large values of n, we can assume that p_{n-1} and P_{n-1} are approximately equal to P_{n-2}, as the probability does not change significantly from one step to the next due to the large number of permutations. Therefore, we can simplify our recursive formula as:

$$P_n \approx \frac{n-1}{n} \cdot P_{n-2} + \frac{1}{n} \cdot P_{n-2} \approx \left(1 - \frac{1}{n}\right) \cdot P_{n-2}$$

Now, let's examine the behavior of P_n as n approaches infinity.

$$\lim_{n\to\infty} P_n \approx \lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^{n/2} \cdot P_0 \approx \left(\lim_{n\to\infty} \left(1 - \frac{1}{n}\right)^n\right)^{1/2} \cdot P_0 \approx \left(\frac{1}{e}\right)^{1/2} \cdot P_0 \approx \frac{1}{\sqrt{e}},$$

as $P_0 = 1$ (since there is only one possible permutation when there are no people). Hence, for a large number of hats (n approaching infinity), the probability that no one gets their own hat back is approximately $\frac{1}{\sqrt{e}}$, which is approximately $\frac{1}{e}$ when squared. This result shows that as the number of hats becomes large, the probability approaches $\frac{1}{e}$.

15.8.5 interpolating approaches

To derive the formula for "complete permutations" $w(n)$, which when divided by $n!$ gives us the probability $P_n = \frac{w(n)}{n!}$, let's relate it to derangements. Consider the set of all permutations of n elements, denoted by $n!$. From this, we subtract the number of permutations with fixed points to gives us the number of complete permutations:

$$w(n) = n! - !n$$

Now, dividing both sides by $n!$, we get the probability P_n that no one gets their own hat back:

$$P_n = \frac{w(n)}{n!} = 1 - \frac{!n}{n!}$$

in which$!n$ satisfies the recursive formula $!n = (n-1)(!(n-1) + !(n-2))$ so

$$P_n = 1 - \frac{(n-1)(!(n-1) + !(n-2))}{n!}$$

Simplifying this expression gives us the desired recursive formula for P_n in terms of derangements.

```
def derangement(n):
    if n == 0:
        return 1
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return (n - 1) * (derangement(n - 1) + derangement(n - 2))

def probability_no_one_gets_own_hat_back(n):
    total_permutations = math.factorial(n)
    derangements = derangement(n)
    probability = derangements / total_permutations
    return probability
```

To calculate the number of derangements $!n$ and the probability P_n that no one gets their own hat back, we define two functions in Python.

Derangement Function: The function `derangement(n)` calculates the number of derangements for n elements. A derangement is a permutation of a set where no element appears in its original position. The function is defined as follows:

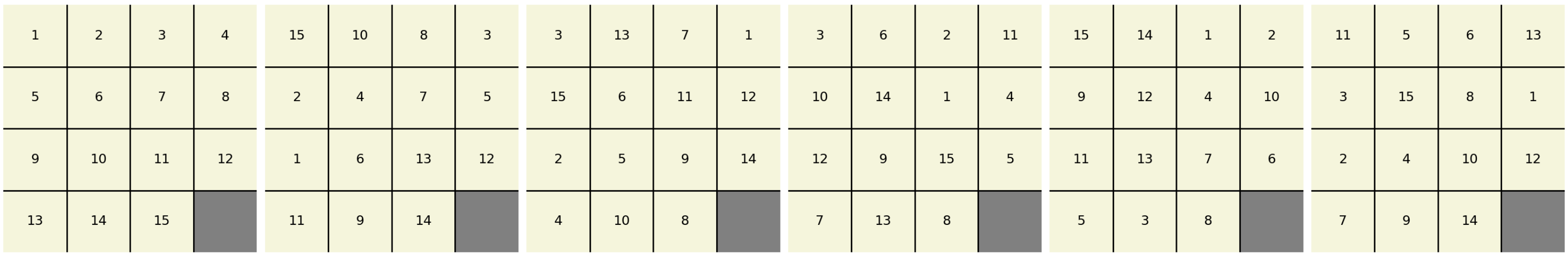
$$\text{def derangement}(n): \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ (n-1) \times (\text{derangement}(n-1) + \text{derangement}(n-2)) & \text{otherwise} \end{cases}$$

Probability Function: The function `probability_no_one_gets_own_hat_back(n)` computes the probability P_n that no one gets their own hat back among n people. It is defined as follows:

$$\text{def probability\_no\_one\_gets\_own\_hat\_back}(n):$$

```
    $\text{total\_permutations} = n!$    (total number of permutations)
    $\text{derangements} = \text{derangement}(n)$    (number of derangements)
    $\text{probability} = \frac{\text{derangements}}{\text{total\_permutations}}$    (probability of no one getting their own hat back)
    return probability
```

These functions provide a way to compute the probability of the hat check problem using derangements, which is a fundamental concept in combinatorial mathematics.



16. Giving a Toss

16.1 Win Loss Frequency

Step 1: Define the Randomness

To begin, we need to precisely define what we mean by a "truly random" win-loss profile. In a genuinely random scenario, the outcome of each game should be independent of previous outcomes. This signifies that the result of one game should not influence the result of the next. In simpler terms, a win or loss today should have no bearing on whether the team wins or loses tomorrow.

Step 2: Identify Winning Streaks

One approach to assess randomness is by considering winning streaks. If the win-loss profile is genuinely random, we should anticipate encountering various lengths of winning streaks throughout the season.

Imagine you're flipping a fair coin. If it's genuinely random, you'll obtain a mixture of heads and tails, and occasionally, you might experience a sequence of consecutive heads or tails. Similarly, if the baseball team's wins and losses are truly random, you'd anticipate observing short, medium, and long winning streaks during the season.

Step 3: Compare to Dice Throwing

Now, let's draw an analogy with throwing two dice. When you throw two dice, you have 36 possible outcomes (6 sides on the first die times 6 sides on the second die). Some outcomes are more likely than others. For instance, there's only one way to achieve a sum of 2 (both dice showing 1), but there are multiple ways to attain a sum of 7 ($1+6$, $2+5$, $3+4$, $4+3$, $5+2$, $6+1$).

In the context of baseball, shorter winning streaks are more probable than longer ones. Just as rolling a sum of 7 is more likely than rolling a sum of 2 with two dice. Longer winning streaks resemble rarer occurrences like rolling a sum of 2. Thus, if the team experiences an unusually extended winning streak, it might raise queries about the randomness of the outcomes.

Step 4: Analyze the Likelihood of Streaks

To ascertain the authenticity of the win-loss profile's randomness, we can compute the likelihood of observing specific streaks. This involves evaluating the probabilities of different lengths of streaks occurring.

For instance, the probability of flipping heads three times in a row with a fair coin is $(0.5)^3 =$

0.125, which is 12.5%. Similarly, we can compute the likelihood of a baseball team having a three-game winning streak, given the assumed win probability per game. If the actual streaks correspond to these probabilities, it suggests randomness.

Step 5: Compare Observations

Lastly, compare the actual win-loss streaks of the LA Dodgers' 1993 season with the calculated probabilities. If you notice streaks that are substantially rarer than what random chance would anticipate, it might imply that the outcomes are not genuinely random. Conversely, if the observed streaks reasonably align with the anticipated probabilities, it supports the notion of randomness.

Remember, randomness doesn't imply that every outcome is equally probable. In a random sequence of coin flips, you might still encounter extended sequences of heads or tails sporadically. Similarly, in baseball, a truly random sequence of wins and losses can still yield streaks of varying lengths.

Keep in mind that assessing randomness is a statistical endeavor. To make a definitive determination about the LA Dodgers' 1993 season, you'd need to conduct an in-depth statistical analysis, including calculating probabilities and comparing them to the observed streaks.

16.2 Win-Loss distribution of Coin-Tossing

In a series of n coin tosses, we consider a win for Alice when the outcome is a head (denoted as 1) and a loss when the outcome is a tail (denoted as 0). Let Z_n denote Alice's fortune after n coin tosses, defined as twice the number of heads (successes) minus the number of tosses:

$$Z_n = 2S_n - n$$

where S_n is the number of successful coin tosses (heads).

The number of successful coin tosses, S_n, follows a binomial distribution $B(n,h;\frac{1}{2})$ given by the binomial coefficient:

$$B(n,h;\frac{1}{2}) = \binom{n}{h}\left(\frac{1}{2}\right)^h\left(1-\frac{1}{2}\right)^{n-h}$$

The variable X represents the number of times Alice is in the lead throughout the series of tosses. To find the probability distribution of X, we follow [18] and generate all possible sequences of heads and tails, keeping track of the number of times Alice is ahead in the sequence.

The Python code implementation generates all possible sequences of 0s (tails) and 1s (heads) for a given length n using binary representation. For each sequence, it counts the number of times Alice is in the lead (when the cumulative sum of heads exceeds the cumulative sum of tails). The function '$calculate_{l}ead_{d}istribution$'accumulates these counts, and '$calculate_{f}inal_{f}ortune$' computes Z_n for each sequence. Finally, 'plot$_{l}ead_{d}istribution$'$visualizesthedistributionof$X$forAlicebeingintheleadanevennumberoft$

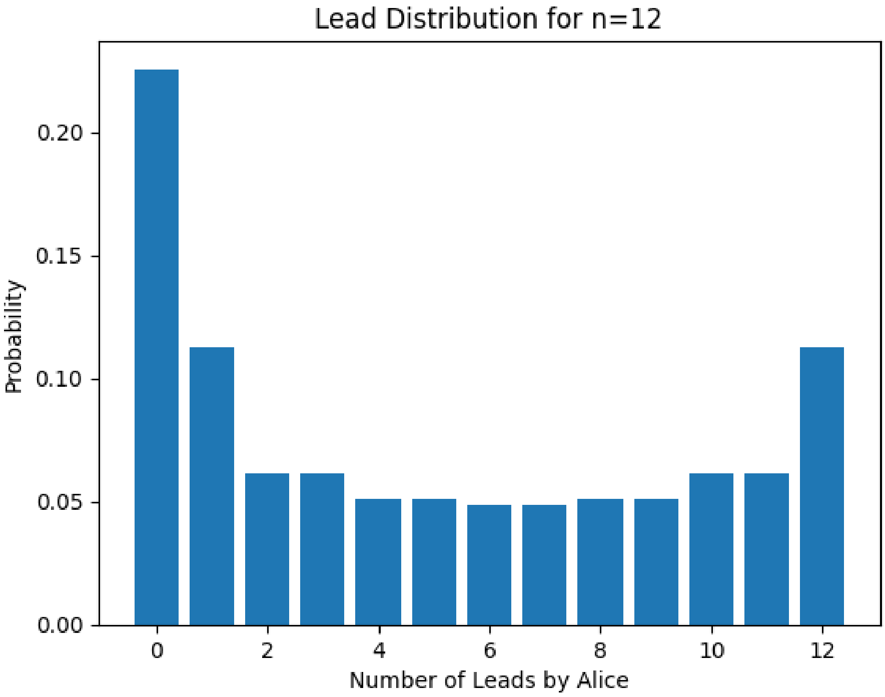


Figure 16.1: Distribution for number times Alice leads Bob.

The function ‘generate$_toss_sequences$‘$producesallpossiblesequencesofcointossesoflength$n$.Foreachsequence,the$

The function ‘calculate$_final_fortune$‘$takesasequenceofcointossesandcalculatesAlice'sfinalfortune$Z$_n$ by evaluating $2S_n - n$, with S_n being the sum of 1s in the sequence. The function ‘plot$_lead_distribution$‘$usesmatplotlibto$

16.3 Sticking it to Pascal

In a series of coin tosses, each individual toss can be considered a Bernoulli trial, a random experiment with exactly two possible outcomes: success (Head) or failure (Tail). If we denote a Head by H and a Tail by T, and assume that the coin is fair, the probability of heads (success) is p and the probability of tails (failure) is $q = 1 - p$.

16.3.1 Bernoulli Trials and the Distribution of Runs of Heads

The probability mass function (pmf) of a Bernoulli distribution, where X is a random variable representing the outcome of a single trial, is given by:

$$P(X = k) = \begin{cases} p & \text{if } k = 1 \\ 1 - p & \text{if } k = 0 \end{cases}$$

16.3.2 Augmented Pascal triangle

When considering a run of Heads, we are interested in the number of consecutive successes in a sequence of Bernoulli trials.

The distribution of the length of the longest run of Heads in a sequence of n coin tosses can be described using an augmented version of Pascal's triangle, [22].

We define the cumulative distribution function $F_n(x)$ for a sequence of n coin tosses as the probability that the longest run of consecutive heads is at most x. It is given by the formula:

$$F_n(x) = \sum_{h=0}^{n} C_n^h(x) p^h q^{n-h}$$

where:

- $C_n^h(x)$ is the number of sequences of length n with exactly h heads and no more than x consecutive heads.
- p is the probability of getting heads on a single coin toss.
- $q = 1 - p$ is the probability of getting tails on a single coin toss.

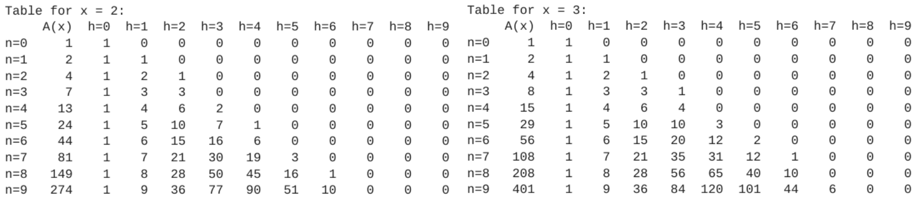
The recursive formula for $C_n^h(x)$, which augments Pascal's triangle, is defined as:

$$C_n^h(x) = \sum_{j=0}^{x} C_{n-1-j}^{h-j}(x)$$

with the initial conditions:

$$C_n^h(3) = \begin{cases} 0 & \text{for } 3 < h < n \\ \binom{n}{h} & \text{for } h \leq 3 \end{cases}$$

pandaX=3HT.ipynb achieves this



```
Table for x = 2:                                              Table for x = 3:
      A(x)  h=0  h=1  h=2  h=3  h=4  h=5  h=6  h=7  h=8  h=9        A(x)  h=0  h=1  h=2  h=3  h=4  h=5  h=6  h=7  h=8  h=9
n=0      1    1    0    0    0    0    0    0    0    0    0  n=0      1    1    0    0    0    0    0    0    0    0    0
n=1      2    1    1    0    0    0    0    0    0    0    0  n=1      2    1    1    0    0    0    0    0    0    0    0
n=2      4    1    2    1    0    0    0    0    0    0    0  n=2      4    1    2    1    0    0    0    0    0    0    0
n=3      7    1    3    3    0    0    0    0    0    0    0  n=3      8    1    3    3    1    0    0    0    0    0    0
n=4     13    1    4    6    2    0    0    0    0    0    0  n=4     15    1    4    6    4    0    0    0    0    0    0
n=5     24    1    5   10    7    1    0    0    0    0    0  n=5     29    1    5   10   10    3    0    0    0    0    0
n=6     44    1    6   15   16    6    0    0    0    0    0  n=6     56    1    6   15   20   12    2    0    0    0    0
n=7     81    1    7   21   30   19    3    0    0    0    0  n=7    108    1    7   21   35   31   12    1    0    0    0
n=8    149    1    8   28   50   45   16    1    0    0    0  n=8    208    1    8   28   56   65   40   10    0    0    0
n=9    274    1    9   36   77   90   51   10    0    0    0  n=9    401    1    9   36   84  120  101   44    6    0    0
```

Figure 16.2: Hockey stick adapted Pascal for X=2 and X=3 cases of consecutive Heads

We can construct the frequency distribution for the longest run L_n by calculating the differences in consecutive values of the cumulative distribution, i.e., $R_n(x) = F_n(x) - F_n(x-1)$.

flexibleFixedRunOfXFrequencyDistribution.ipynb achieves this

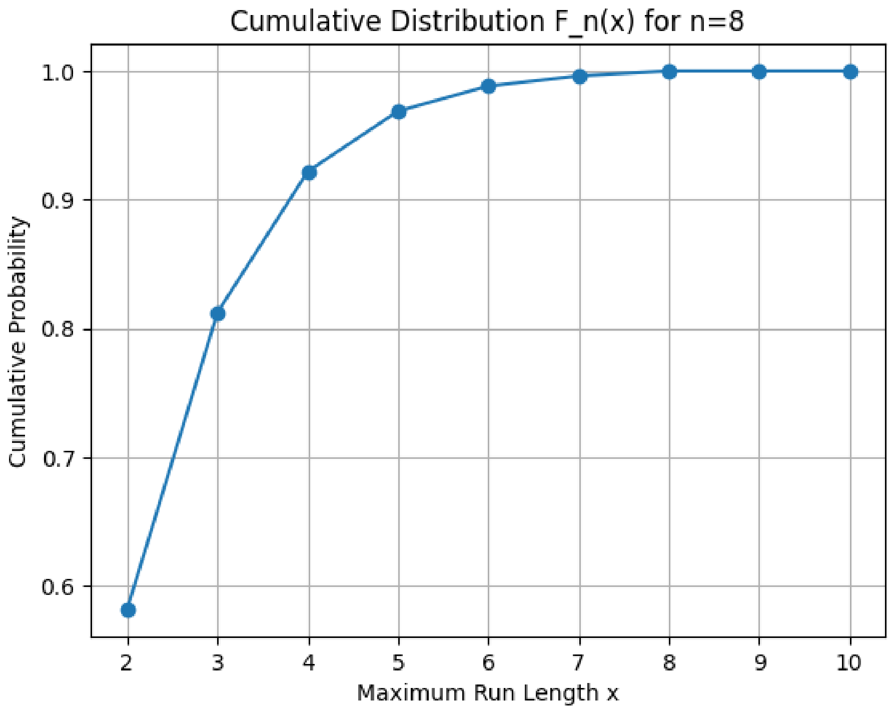


Figure 16.3: Cumulative Frequency Distribution for X=3.

16.4 Coin Toss clustering on a table

When considering a cluster of circles, such as coins on a plane, an exact calculation of the overlapping area can be complex due to the potential for intricate interlacing boundaries. To approximate the total area covered by these circles, we can employ the concept of a convex hull. A convex hull is the smallest convex shape that completely encloses a set of points. In two dimensions, it can be visualized as the shape formed by a rubber band stretched around the exterior points.

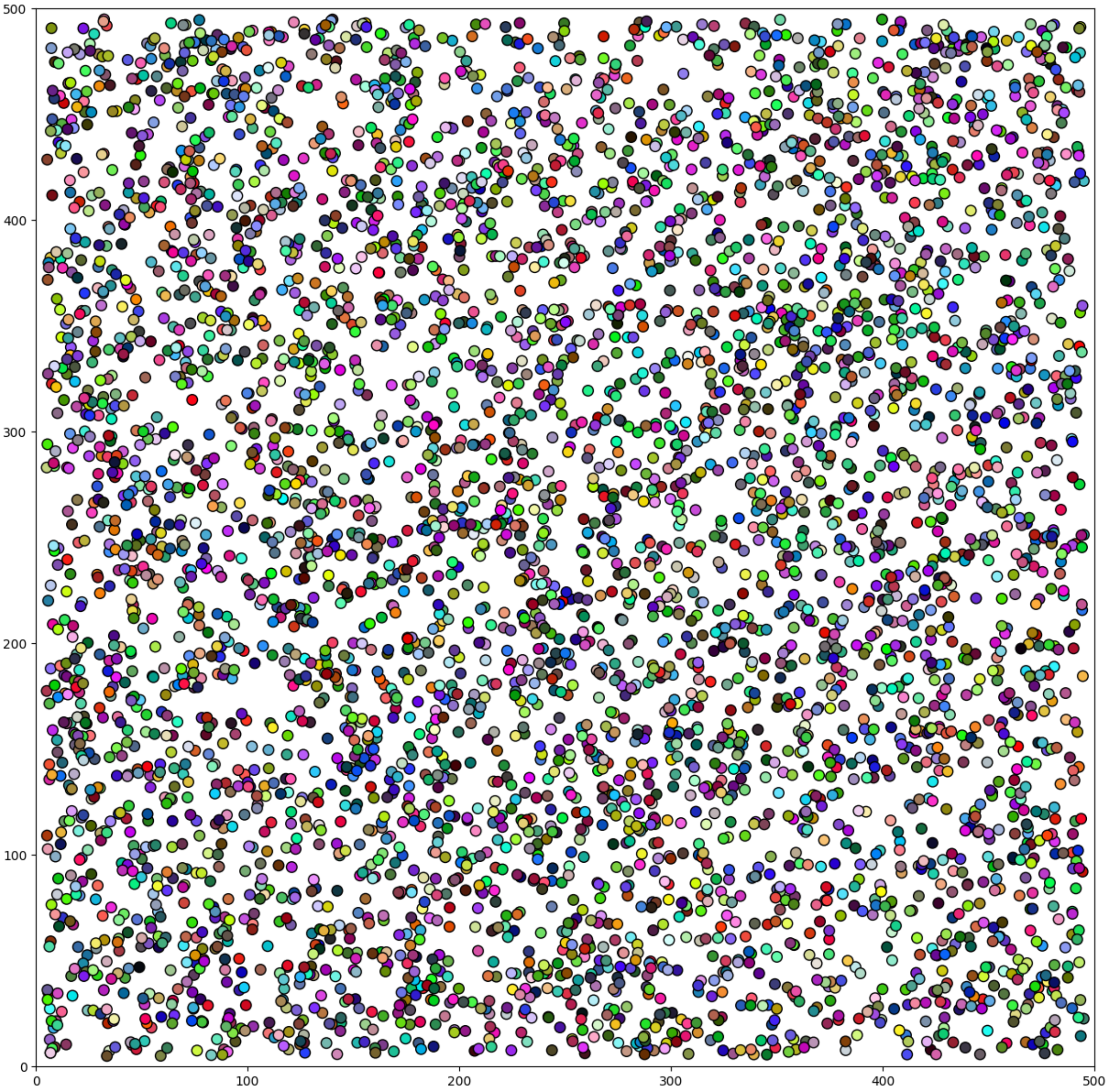


Figure 16.4: Coins Random Tossed on a Table.

For a set of points that represent the centers of the circles, the convex hull does not correspond to the exact area because it does not account for the curvatures of the circles. However, it provides a useful approximation in the following way:

1. If the number of circles is less than three, they cannot form a polygon, and thus the area is considered to be zero since there is no enclosed space by lines.
2. For three or more circles, the convex hull can be computed. The vertices of the convex hull polygon are a subset of the circle centers. This polygon does not include the actual area of the circles themselves, but rather the area of the space between the centers of the outermost circles.

3. The area of the convex hull polygon is considered to be an approximation of the total area covered by the circles. This approximation is more accurate when the circles are closely packed, as the convex hull then tends to align closely with the perimeter of the interlaced area.

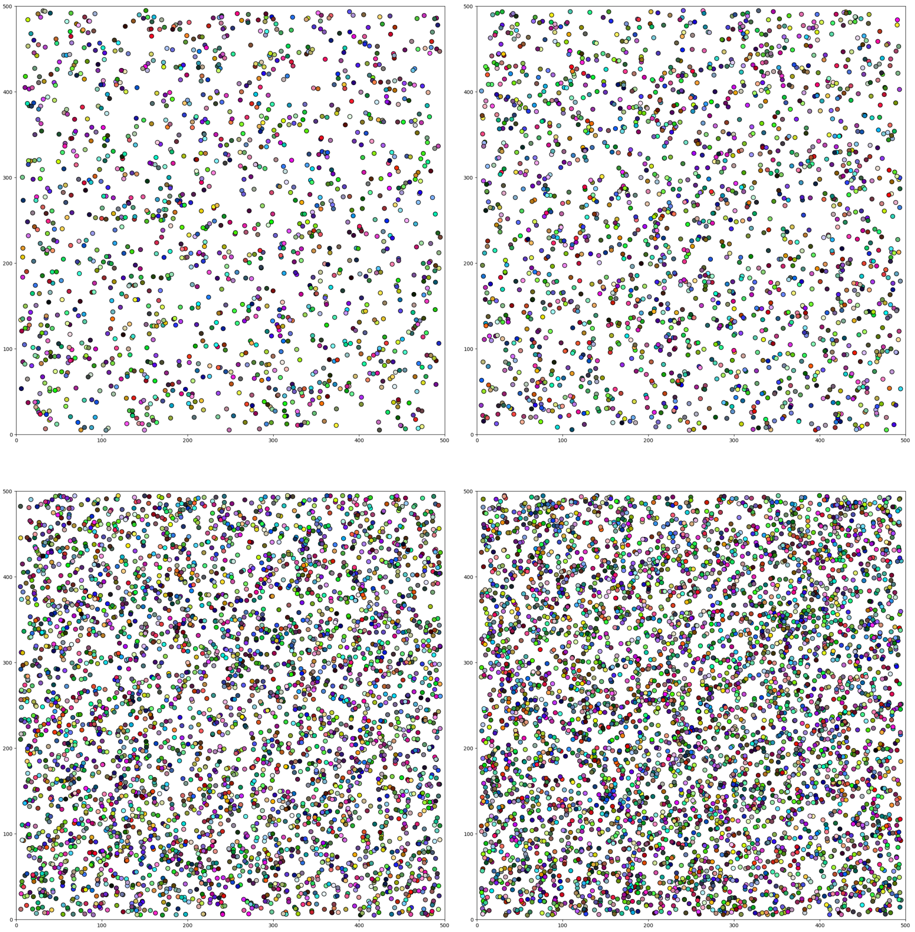


Figure 16.5: 1500, 2000, 4000 and 5000 tosses

Thus, the `ConvexHull` object provides an estimated area (referred to as `volume` in 2D) which we use as an approximation for the complex shape formed by overlapping circles.

$$A_{\text{approx}} = \text{ConvexHull}(\text{coins}).\text{volume} \tag{16.1}$$

Where A_{approx} is the approximate area of the interlaced circles, and `coins` is the set of points representing the centers of the circles.

The Python code presented is part of an algorithm used to identify and analyze clusters of points, which in this context represent the centers of coins on a plane. The purpose of this algorithm is to group coins that form connected 'islands' and then calculate certain properties of these islands, such as their area and the maximum distance between any two coins within an island.

The main functions involved are:

- `find`: Implements the path compression technique, which is part of the Union-Find algorithm used to efficiently find the representative of an element's set.

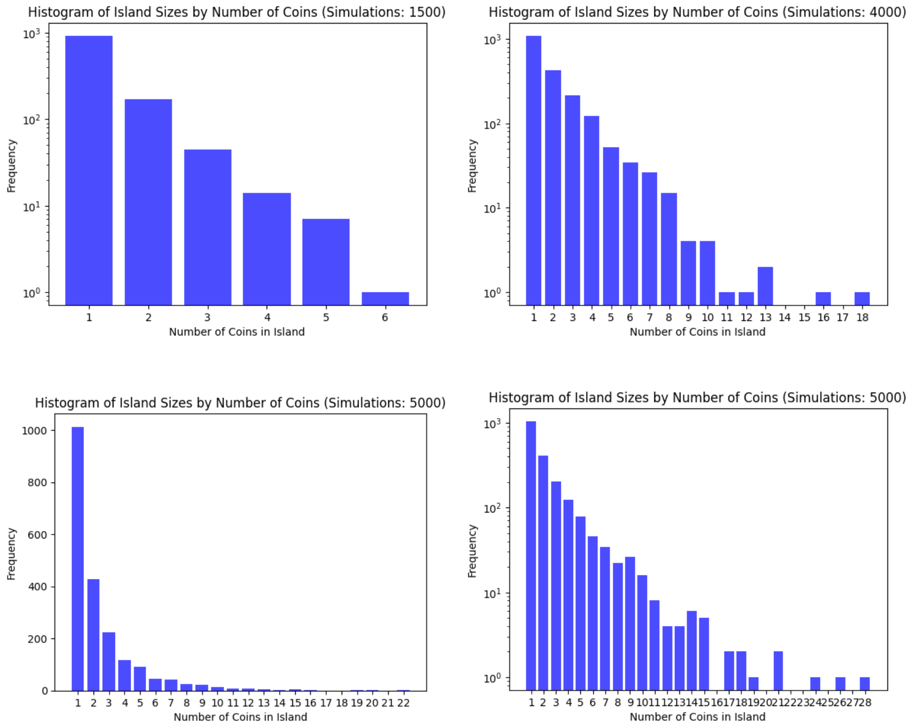


Figure 16.6: By Number Size Coin Island Distributions

- `union`: Joins two sets if they are not already joined, by linking one representative to another. It uses union by rank to keep the tree shallow.
- `get_island_info`: Analyzes all pairs of coins and uses the Union-Find algorithm to group overlapping coins into islands. It also calculates the 'center of mass', maximum length, and approximate area for each island.

The Python packages used in this algorithm include:

- **NumPy**: A fundamental package for scientific computing with Python providing a high-performance multidimensional array object and tools for working with these arrays. It is used here to calculate the mean position of coins in an island (center of mass) and to handle array operations efficiently.
- **SciPy**: This library is used for scientific and technical computing. The function `cdist` from the `scipy.spatial.distance` module calculates the distance between each pair of the two collections of inputs (coin centers in this case). It is crucial for determining whether two coins overlap and hence belong to the same island.

FixedCoinTosser.ipynb

Comparison of Functionality Changes and Improvements

Improvement in Island Detection

- Old Code: Overlapping coins are checked using a nested loop with pairwise distance checks.
- New Code: Implements Union-Find algorithm for more efficient island detection.

Plot Range Extension

- Old Code: `plt.xlim(0, 50*d)` and `plt.ylim(0, 50*d)`
- New Code: `plt.xlim(0, 100*d)` and `plt.ylim(0, 100*d)`

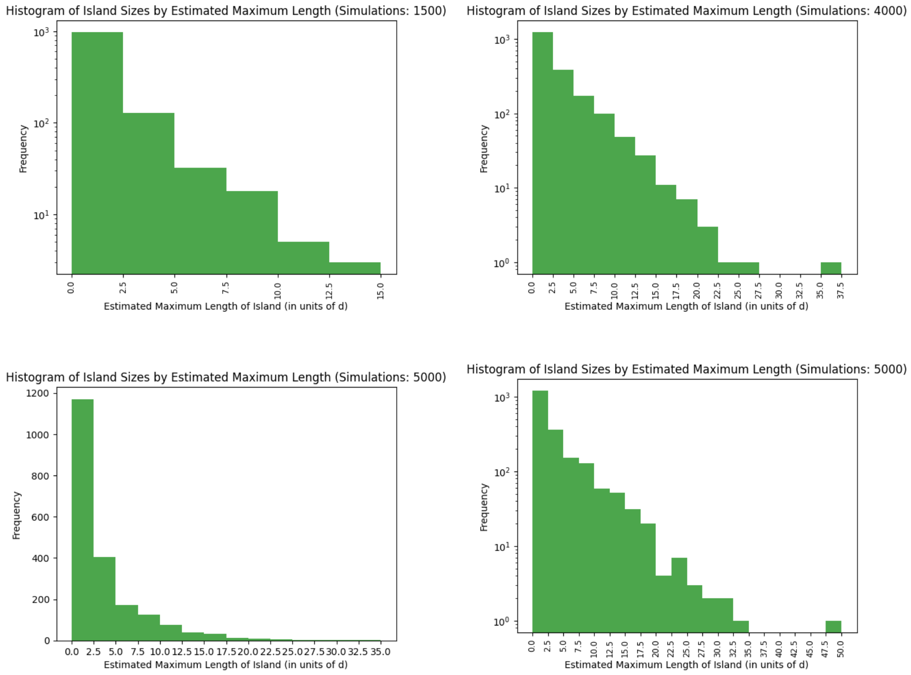


Figure 16.7: By Length Size Coin Island Distributions

Area Calculation

- Old Code: Does not calculate area.
- New Code: Calculates the area of islands using the Convex Hull method.

Histogram Presentation Enhancements

- Old Code: Basic histogram with default scaling.
- New Code: Histograms with log scaling, improved binning, and fitted lines for better data interpretation.

Scalability Improvement

- Old Code: Single simulation with user input for the number of tosses.
- New Code: Allows running multiple simulations over a range of tosses with a function `run_simulations(LT, UT, steps)`.

logLoopedCoinTosserArea.pynb

Rolling Stacked Charts

We have this code MetalogLoopedCoinTosserNumber.pynb delivers the following stacked chart

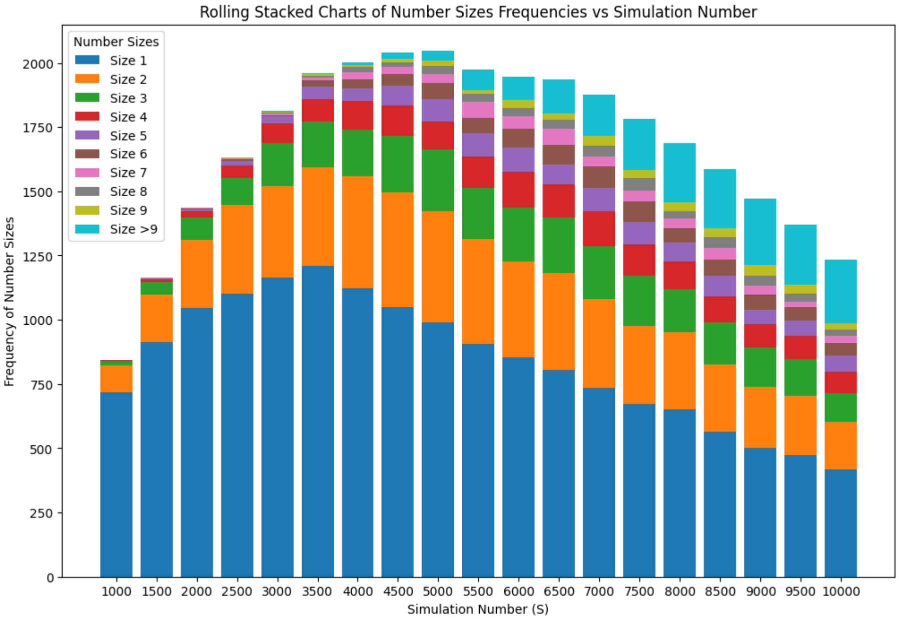


Figure 16.8: Coin Toss Island Number frequency by Number of tossed coins.

whereas this PercentWeightedMetaLoopedCoinTosserNumber.pynb delivers the following normalized bin weighted stacked chart

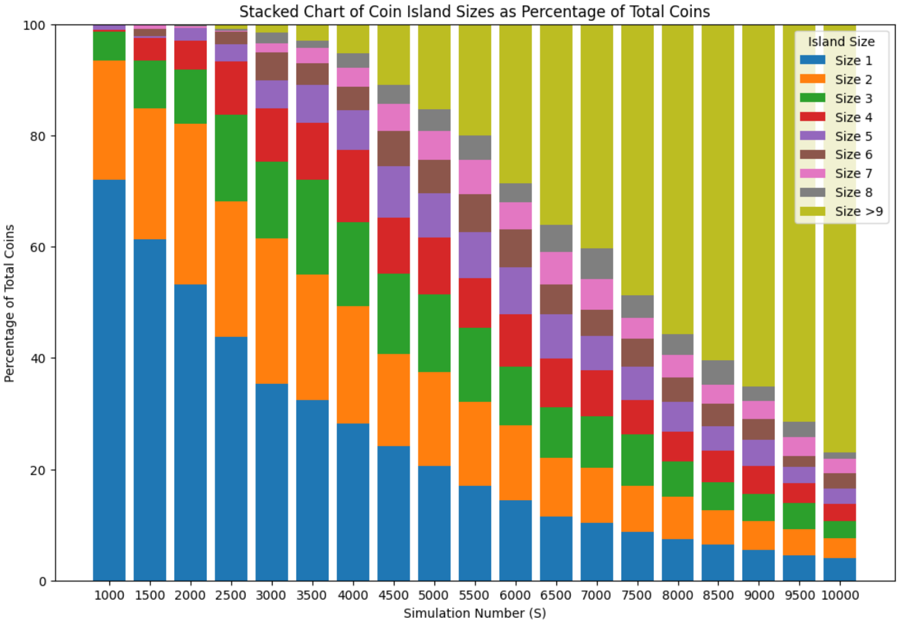
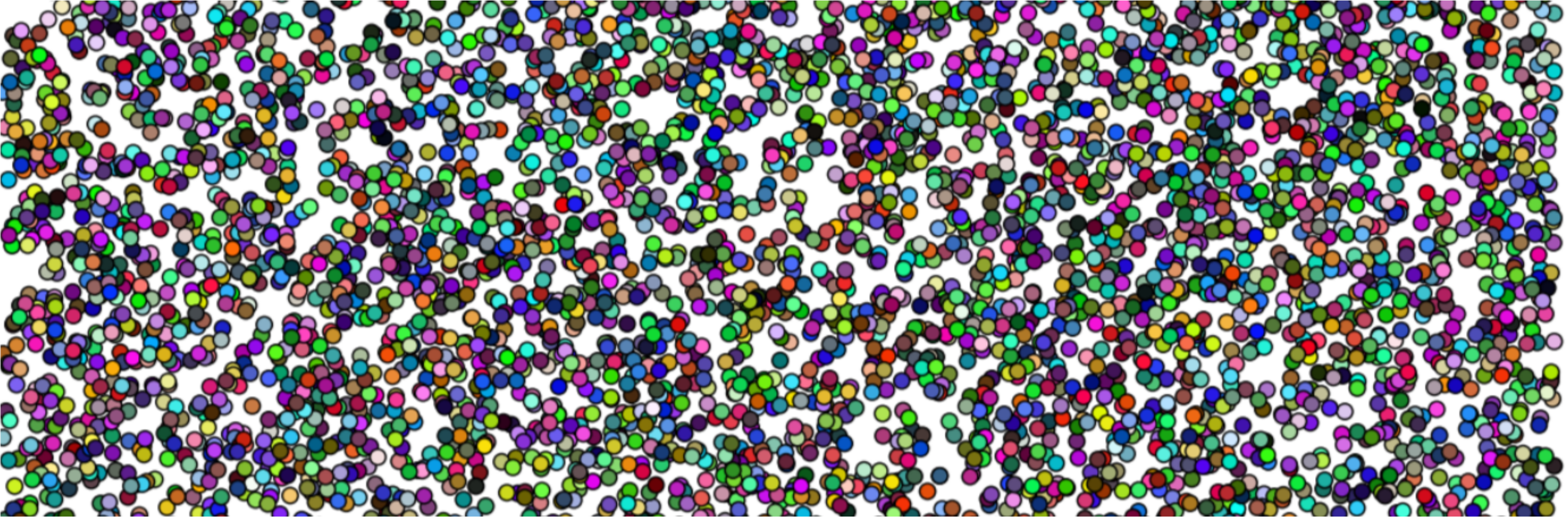


Figure 16.9: Coin Toss Island Number as (bin size weighted) % of Number of tossed coins.

```
\Function{RunSimulationsAndCollectData}{$LT$, $UT$, $stepSize$}
    \State $frequencyData \gets \{i: [] \text{ for } i \text{ in range}(1, 10)\}$
    \State $frequencyData[">9"] \gets []$
    \State $totalCoinsData \gets []$
    \State $simulationSteps \gets \text{arange}(LT, UT + stepSize, stepSize)$

    \ForAll{$T$ in $simulationSteps$}
        \State $nSizes \gets \text{simulateCoinToss}(T)$
        \State $totalCoins \gets \text{sum}(nSizes)$
        \State $totalCoinsData \text{ append } totalCoins$

        \State $sizeCounts \gets \{size: nSizes.count(size) \text{ for } size \text{ in ran
        \State $weightedCount \gets \text{sum}(count \times size \text{ for } size, count \

        \For{$size$ in range(1, 10)}
            \State $frequencyData[size] \text{ append } ((sizeCounts[size] \times size) / t
        \EndFor
        \State $frequencyData[">9"] \text{ append } (weightedCount / totalCoins) \times 100
    \EndFor

    \State \Return $frequencyData$, $totalCoinsData$, $simulationSteps$
\EndFunction
```



17. Integer Lattice problems

17.1 Surd diagonals drawn on a lattice

The lattice of points in a 2D plane is the arena for exploring integer number properties. Each point is defined by integer coordinates (a,b). Diagonals between lattice vertices are of surd length. Consider, the question of what surd lengths are possible for all such diagonals of the lattice. What restricts the lengths of such diagonals? The code integerLatticeSurdDiagonals.ipynb delivers our picture to consider:

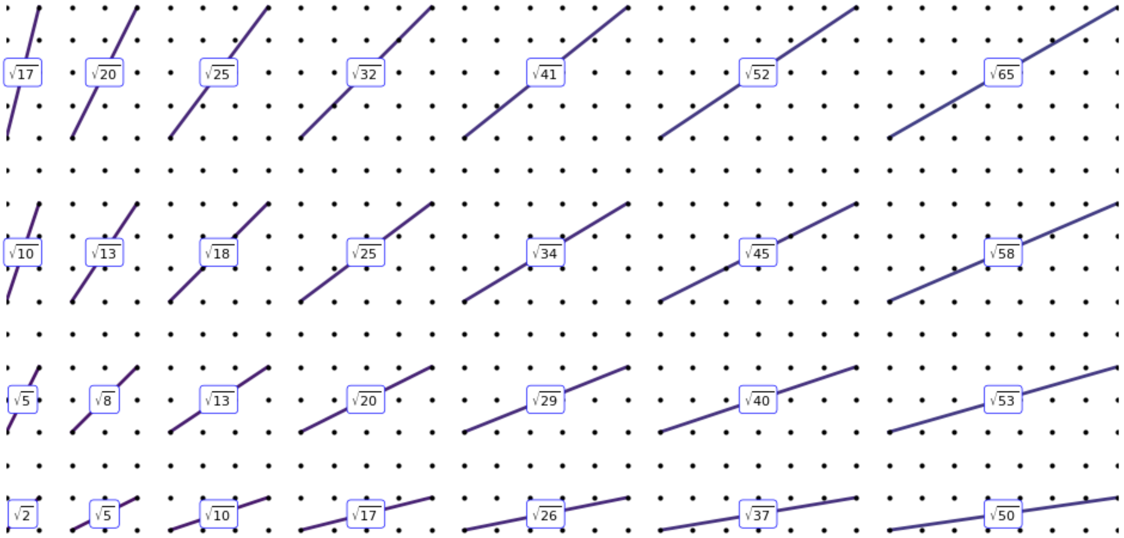


Figure 17.1: Diagonal Lengths of integer lattice

We note that the expression inside the root (the *radicand*) naturally form a quadratic sequence. The symmetry lends itself to a representation of the radicand lengths within a matrix, in which each cell point (a,b) has a value of $c = a^2 + b^2$:

1. **Fermat Primes:** A Fermat prime is a prime number that can be expressed as the sum of two squares, $p = m^2 + n^2$. In our matrix, cells where c is a Fermat prime are highlighted. These primes are directly visualized in our lattice as the length of diagonals that are square roots of prime numbers.
2. **Semi-Primes:** Semi-primes are products of two primes, often represented as $q = p_1 \cdot p_2$. In the lattice, these correspond to surds where $a^2 + b^2$ results in a semi-prime.

By translating the geometric concept of lattice diagonals into a multiplication table we again bridge geometry and number theory. Representing diagonals in the $a \times b$ matrix such that $a =< b$ we have as the only possible lengths, the Fermat primes (blue), semi-(Fermat) primes (green) and those compound numbers (encoded by increasing numbers of factors yellow to orange) that correspond to compound versions of those (semi)-Fermat radicands.

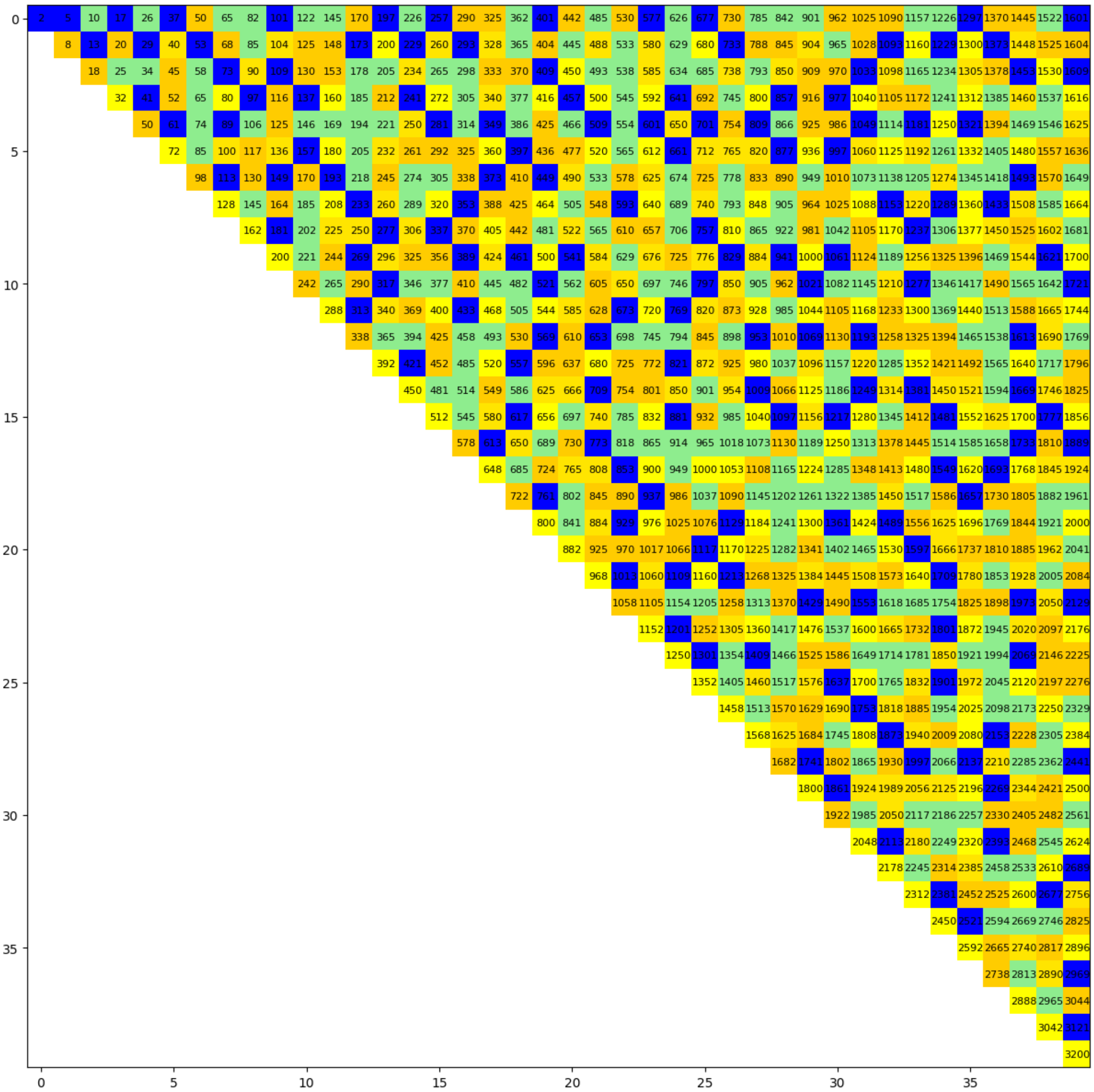


Figure 17.2: Diagonal Lengths of integer lattice as Multiplication table $a^2 + b^2$

numberedMatrixOfLatticeSurdsjustColors.ipynb implements this to produce the matrix (17.2).

We can drop the numbers and just look at the colours for greater representation. or de-emphasise the primes, treat semi-primes as red hot special and compound surds in receding hues of orange as their factor composition multiply

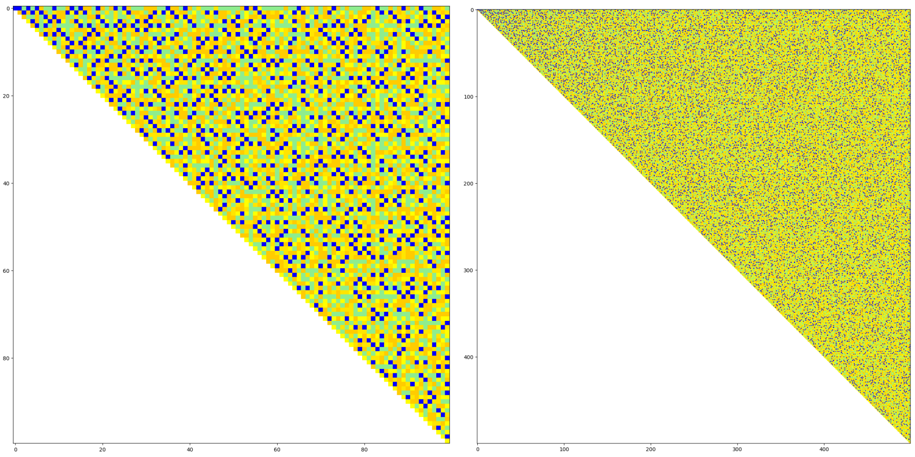


Figure 17.3: 100x100 and 500x500 upper diagonal Matrices of surd length with blue Fermat primes

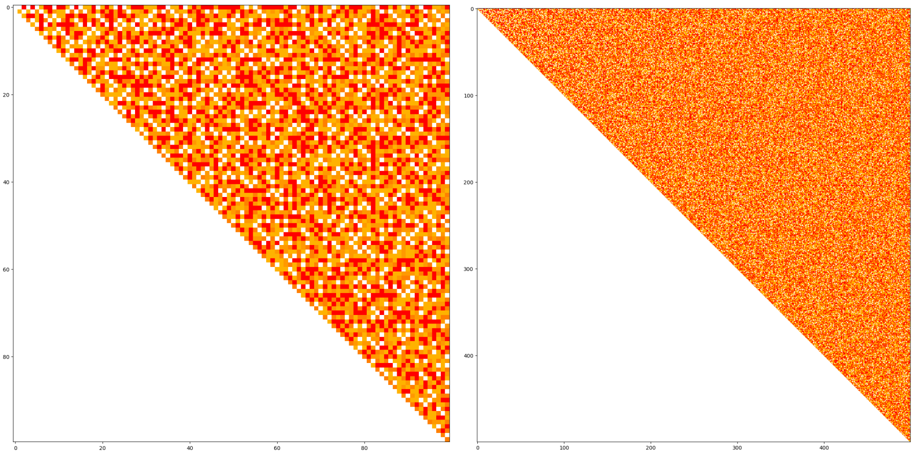


Figure 17.4: Matrices of surd length emphasising factor multiplicity from low-red to high-yellow

The code snippet below highlights the matrix colour presentation:

```
def create_matrix_plot_with_options(matrix_size, print_numbers, condition):
   matrix = np.ones((matrix_size, matrix_size, 3))  # Default white color
   # Calculate the font size dynamically based on the matrix size
   font_size = max(1, int(28 - matrix_size / 2))
   for a in range(1, matrix_size + 1):
       for b in range(1, matrix_size + 1):
           # Check the condition based on user input
           if (condition == ’a<=b’ and a <= b) or (condition == ’a>b’ and a > b):
               c = a**2 + b**2
               # Assign color based on the type of number
               if is_fermat_prime(c):
```

```
        color = [0, 0, 1]  # blue
    elif is_prime(c):
        color = [1, 0, 0]  # Red
    elif is_semi_prime(c):
        color = [0.56, 0.93, 0.56]  # Light green
    else:
        factor_count = count_prime_factors(c)
        #yellow- to orange
        color_intensity = min(factor_count / 5, 1)  # Normalize intensity
        # color = [1, 0.5 + 0.5 * color_intensity, 0]
    matrix[a-1, b-1] = color
```

17.2 Circumscribing polygons

Circumscribing and inscribing polygons provide a method for approximating the circumference of a circle while investigating the relationship between the perimeter and area of polygons. This relationship leads to the conclusion that a circle is the limiting polygon with the minimum perimeter-to-area ratio.

17.3 Circumscribing and Inscribing Polygons

Given a circle with radius r, circumscribing polygons are constructed by placing n vertices equidistantly along the circle's circumference and connecting them. As n increases, the polygon's perimeter approaches that of the circle. The perimeter P_c of the circumscribing polygon with n sides can be expressed as:

$$P_c = n \cdot 2r \cdot \sin\left(\frac{\pi}{n}\right).$$

Inscribing polygons involve placing n vertices on the circle such that the sides of the polygon are tangent to the circle. As n increases, the polygon's perimeter approaches that of the circle. The perimeter P_i of the inscribing polygon with n sides can be expressed as:

$$P_i = n \cdot 2r \cdot \tan\left(\frac{\pi}{n}\right).$$

17.4 Perimeter-to-Area Ratio

Comparing the perimeters and areas of circumscribing and inscribing polygons reveals an interesting relationship. Let A_c be the area of the circumscribing polygon, and A_i be the area of the inscribing polygon. The perimeter-to-area ratios R_c and R_i for circumscribing and inscribing polygons respectively are given by:

$$R_c = \frac{P_c}{A_c}, \quad R_i = \frac{P_i}{A_i}.$$

As n tends to infinity, both R_c and R_i converge to the same value, which is the perimeter-to-area ratio of a circle. This establishes that a circle is the limiting polygon with the minimum perimeter-to-area ratio among all polygons. In mathematical terms:

$$\lim_{n\to\infty} R_c = \lim_{n\to\infty} R_i = \frac{2\pi r}{\pi r^2} = \frac{2}{r}.$$

The code categorizes

```
def inscribe_polygon(n):
    side_length = 2 * math.sin(math.pi / n)
    perimeter = n * side_length
    fraction_of_circumference = perimeter / (2 * math.pi)
    area = 0.5 * n * side_length**2 / math.tan(math.pi / n)
    fraction_of_area = area / math.pi
    return side_length, n, perimeter, fraction_of_circumference, area, fraction_of_area

def calculate_areas_perimeters(n, r):
    side_length_inscribed = 2 * r * math.sin(math.pi / n)
    circumscribed_perimeter = n * side_length_circumscribed
    inscribed_perimeter = n * side_length_inscribed

    circumscribed_area = (circumscribed_perimeter**2) / (4 * math.pi)
    inscribed_area = (inscribed_perimeter**2) / (4 * math.pi)

    return circumscribed_area, inscribed_area, circumscribed_perimeter, inscribed_perimeter
```

17.5 Tournaments in Directed Graphs

A **tournament** in graph theory is a complete directed graph, meaning there is exactly one directed edge between each pair of vertices. In the context of a sporting tournament, vertices can be thought of as players or teams, and edges represent the outcome of a match between them.

Transitive and Cyclic Tournaments

Within tournaments, two important structures are *transitive* and *cyclic* tournaments:

- A **transitive tournament** is one where if player A beats player B, and player B beats player C, then player A also beats player C. In graph terms, this implies no directed cycles.
- A **cyclic tournament** contains cycles, meaning there is no overall winner. If player A beats player B, and player B beats player C, it does not necessarily imply that A beats C.

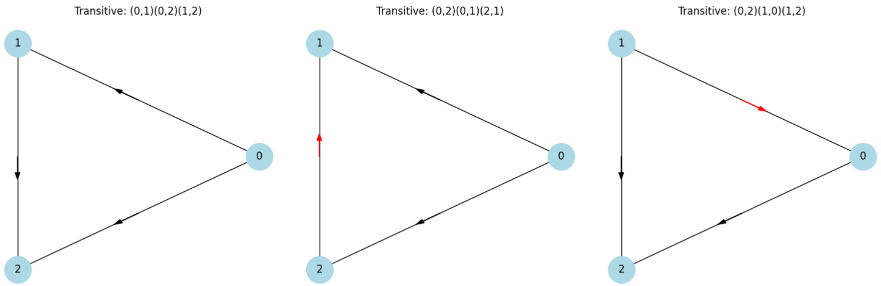

Transitive: (0,1)(0,2)(1,2)
1
0
2
Transitive: (0,2)(0,1)(2,1)
1
0
2
Transitive: (0,2)(1,0)(1,2)
1
0
2

Figure 17.5: Transitive tournaments for n=3 players.

Example for $n = 4$

In the case of $n = 4$ (four players), the tournaments can exhibit both transitive and cyclic properties:

1. **Transitive Example**: Consider players 0, 1, 2, and 3. If 0 beats 1, 1 beats 2, and 2 beats 3, then a transitive structure is formed with edges $(0,1),(1,2),(2,3)$.
2. **Cyclic Example**: If 0 beats 1, 1 beats 2, 2 beats 3, and 3 beats 0, we have a cycle, forming a cyclic tournament.

Finding Subtournaments in a 4-Player Tournament

In a tournament with $n = 4$ players, we can look for smaller subtournaments that are either transitive or cyclic. A subtournament is a tournament formed by a subset of the players of the original tournament. A transitive subtournament is one where the relationships are strictly ordered. In other words, if player A beats player B, and player B beats player C, then player A must also beat player C. In tournaments with $n = 4$, subgraphs exhibiting transitivity can exist even within larger non-transitive structures. These subgraphs highlight partial orderings where some players consistently beat others, forming a hierarchy within the subset of players.

- To find a transitive subtournament for $n = 3$ within an $n = 4$ tournament, we look for a sequence of players where each player defeats the next in the sequence.
- For example, in a tournament with players labeled $0,1,2$, and 3, if the edges are $(0,1),(1,2),(2,3)$, then the subtournament formed by players $0,1$, and 2 is transitive.

Tournamentstransivity.ipynb produces these two figures

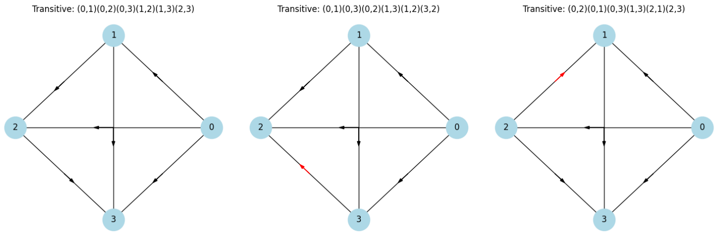

Transitive: (0,1)(0,2)(0,3)(1,2)(1,3)(2,3)
Transitive: (0,1)(0,3)(0,2)(1,3)(1,2)(3,2)
Transitive: (0,2)(0,1)(0,3)(1,3)(2,1)(2,3)
1
2
0
3
1
2
0
3
1
2
0
3

Figure 17.6: Transitive Tournaments for n=4 players

Permutation	**Edges**
0123	(0,1) (0,2) (0,3) (1,2) (1,3) (2,3)
0132	(0,1) (0,3) (0,2) (1,3) (1,2) (3,2)
0213	(0,2) (0,1) (0,3) (2,1) (2,3) (1,3)
0231	(0,2) (0,3) (0,1) (2,3) (2,1) (3,1)
0312	(0,3) (0,1) (0,2) (3,1) (3,2) (1,2)
0321	(0,3) (0,2) (0,1) (3,2) (3,1) (2,1)
1023	(1,0) (1,2) (1,3) (0,2) (0,3) (2,3)
1032	(1,0) (1,3) (1,2) (0,3) (0,2) (3,2)
1203	(1,2) (1,0) (1,3) (2,0) (2,3) (0,3)
1230	(1,2) (1,3) (1,0) (2,3) (2,0) (3,0)
1302	(1,3) (1,0) (1,2) (3,0) (3,2) (0,2)
1320	(1,3) (1,2) (1,0) (3,2) (3,0) (2,0)
2013	(2,0) (2,1) (2,3) (0,1) (0,3) (1,3)
2031	(2,0) (2,3) (2,1) (0,3) (0,1) (3,1)
2103	(2,1) (2,0) (2,3) (1,0) (1,3) (0,3)
2130	(2,1) (2,3) (2,0) (1,3) (1,0) (3,0)
2301	(2,3) (2,0) (2,1) (3,0) (3,1) (0,1)
2310	(2,3) (2,1) (2,0) (3,1) (3,0) (1,0)
3012	(3,0) (3,1) (3,2) (0,1) (0,2) (1,2)
3021	(3,0) (3,2) (3,1) (0,2) (0,1) (2,1)
3102	(3,1) (3,0) (3,2) (1,0) (1,2) (0,2)
3120	(3,1) (3,2) (3,0) (1,2) (1,0) (2,0)
3201	(3,2) (3,0) (3,1) (2,0) (2,1) (0,1)
3210	(3,2) (3,1) (3,0) (2,1) (2,0) (1,0)

Table 17.1: List of Permutations with Edge Labels for $n = 4$

Cyclic Subtournaments

A cyclic subtournament contains cycles, meaning there is no player who defeats all the others in the subset.

- To identify a cyclic subtournament, we look for a sequence of players where the last player in the sequence defeats the first, forming a cycle.
- For instance, if players $0, 1$, and 2 have edges $(0,1),(1,2),(2,0)$, they form a cyclic subtournament.

Given a tournament with 4 players, we can systematically check all combinations of 3 players to determine if they form a transitive or cyclic subtournament:

1. Examine each subset of 3 players and the directed edges between them.
2. Apply the criteria for transitivity and cyclicality to each subset to categorize the subtournament.

The approach would proceed something like the following:

1. First Permutation: 0 1 2 3
   - Edges: $(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)$
   - Meaning: Directed edges from player 0 to players 1, 2, and 3; from player 1 to players 2 and 3; and from player 2 to player 3.
2. Second Permutation: 0 1 3 2
   - Edges: $(0,1),(0,3),(0,2),(1,3),(1,2),(3,2)$
   - Meaning: Directed edges from player 0 to players 1, 3, and 2; from player 1 to players 3 and 2; and from player 3 to player 2.
3. Third Permutation: 0 2 1 3
   - Edges: $(0,2),(0,1),(0,3),(1,3),(2,1),(2,3)$
   - Meaning: Directed edges from player 0 to players 2, 1, and 3; from player 1 to player 3; and from player 2 to players 1 and 3.

17.6 Triangles inscribed in Circles

Our objective in this section is to explore the relationship between the number of distinct triangles that can be formed by connecting the nodes of regular n-gons (polygons). This answers the question posed in ([15] in which we are invited to determine the number of distinct *primitive* triangles that can be inscribed inside a circle dotted with evenly spaced n pins We can see below those twelve triangles that can be inscribed inside a 12-sided dodecagon.

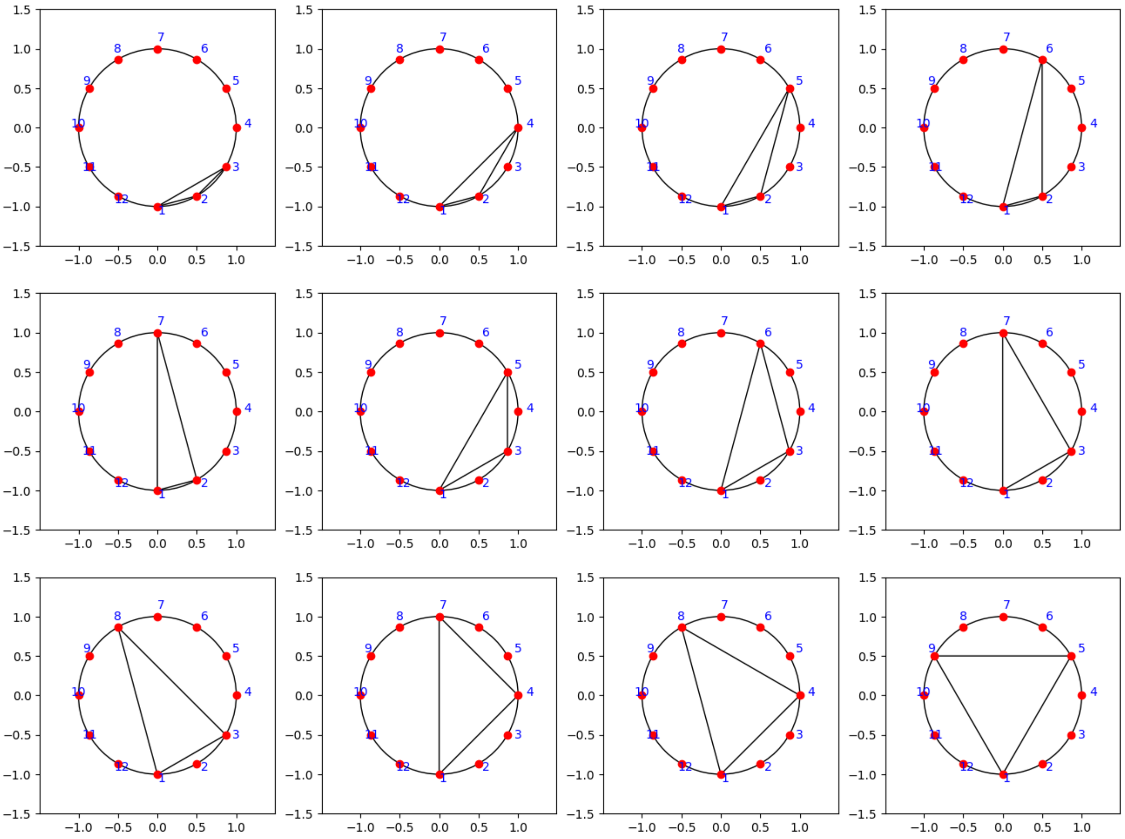


Figure 17.7: Distinct primitive triangles that can be inscribed in an n=12 circular pin board.

For a given number of points, n, on the circle, the task is to identify all possible triangles such that no two triangles are the same under rotations and reflections. The approach we take involves writing Python code, triangleInCircleWithnPins.ipynb a whose key steps are:

1. Calculating all combinations of three points from the n points on the circle.
2. Filtering out triangles that are equivalent under rotation or reflection.
3. Further refining the set of triangles by ensuring each triangle has a unique perimeter, to exclude remaining duplicates.
4. Visualizing these triangles and compiling the data into tables for further analysis.

The code, loopingTriangleInCircles.ipynb yields two key results:

1. A set of distinct triangles for each value of n from n=3 to n=100, along with their perimeters.
2. A log-log plot illustrating the relationship between n and the number of distinct triangles, T, indicating a power law behavior.

T	Triangle	Edges	Perimeter	Area	P/A
12	(1, 5, 9)	[(1, 5), (5, 9), (9, 1)]	5.20	1.30	4.00
11	(1, 4, 8)	[(1, 4), (4, 8), (8, 1)]	5.08	1.18	4.29
10	(1, 4, 7)	[(1, 4), (4, 7), (7, 1)]	4.83	1.00	4.83
9	(1, 3, 8)	[(1, 3), (3, 8), (8, 1)]	4.86	0.93	5.21
8	(1, 3, 7)	[(1, 3), (3, 7), (7, 1)]	4.73	0.87	5.46
7	(1, 3, 6)	[(1, 3), (3, 6), (6, 1)]	4.35	0.68	6.36
6	(1, 3, 5)	[(1, 3), (3, 5), (5, 1)]	3.73	0.43	8.62
5	(1, 2, 7)	[(1, 2), (2, 7), (7, 1)]	4.45	0.50	8.90
4	(1, 2, 6)	[(1, 2), (2, 6), (6, 1)]	4.18	0.43	9.66
3	(1, 2, 5)	[(1, 2), (2, 5), (5, 1)]	3.66	0.32	11.56
2	(1, 2, 4)	[(1, 2), (2, 4), (4, 1)]	2.93	0.18	16.02
1	(1, 2, 3)	[(1, 2), (2, 3), (3, 1)]	2.04	0.07	30.38

Table 17.2: Summary of 12 Inscribed Triangle primitives of Dodecagon 12-gon

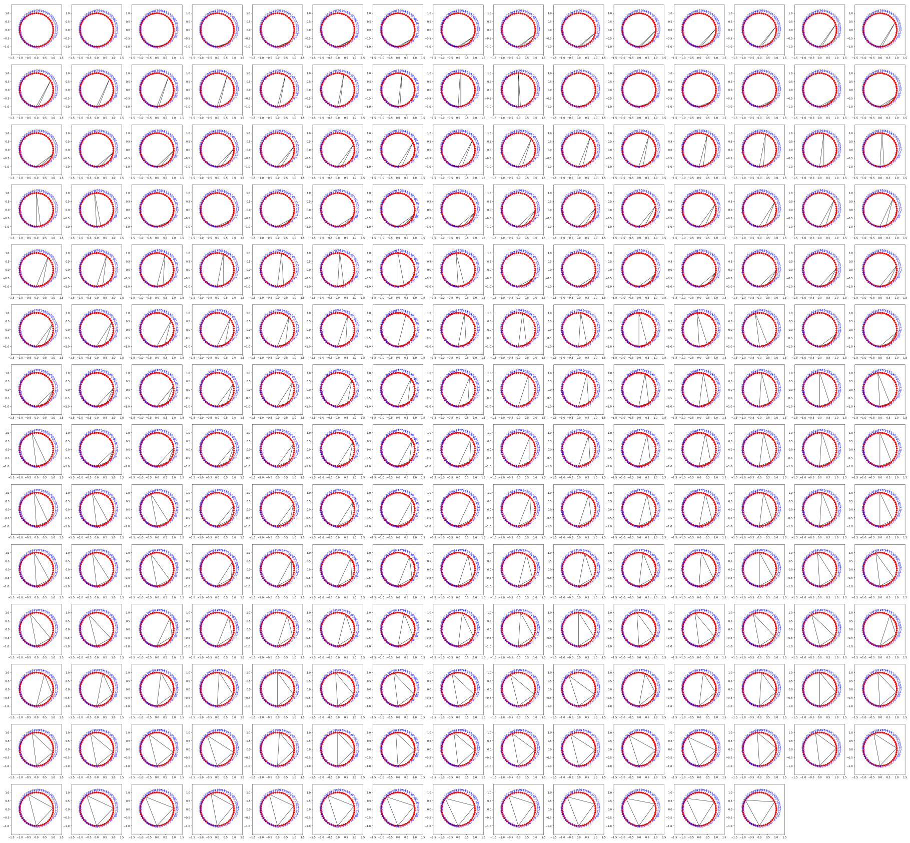


Figure 17.8: 208 distinct primitive triangles that can be inscribed on an n=50 circular pin board.

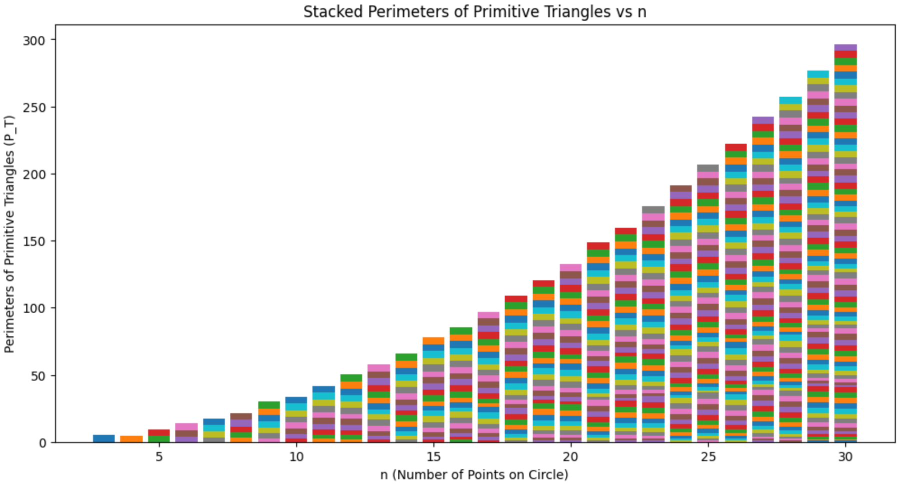


Figure 17.9: Cumulative Perimeter of T primitive triangles that can be inscribed in an n-pin circular board.

From the log-log plot we see a bet fit that is of a quadratic $ax^2 + bx + c$ with a coefficient, a=0.08, b=0,c=0:

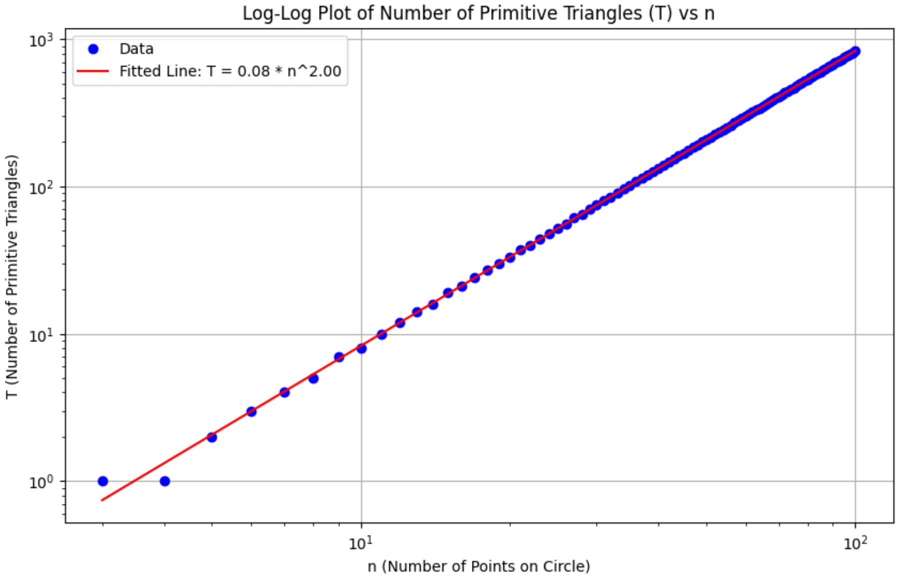


Figure 17.10: Distinct number, T of primitive triangles that can be inscribed in an n circular pin board.

The power law relationship was modeled as $T = a \cdot n^b$, where the fitted parameters were found to be approximately $a \approx 0.083$ and $b \approx 2.00$. This suggests a quadratic relationship between the number of points and the number of distinct triangles, implying that as the number of points on the circle increases, the number of distinct triangles grows quadratically.

17.6.1 Further analysis

The quadratic relationship between the number of points and distinct triangles highlights the rapidly increasing complexity of geometrical configurations as more points are added to the circle.

A further analytical exploration of the relationship between the sum of perimeters of primitive triangles formed by points on a circle (S_T) and the perimeter of the inscribed n-gon (C_n) is now suggested

Perimeter of the n-gon (C_n)

The perimeter of an n-gon inscribed in a circle of radius r is given by:

$$C_n = 2nr\sin\left(\frac{\pi}{n}\right)$$

As n approaches infinity, C_n converges to the circumference of the circle, $2\pi r$.

Sum of Perimeters of Primitive Triangles (S_T)

The sum S_T is influenced by two factors as n increases:

1. The combinatorial increase in the number of distinct triangles.
2. The diversity in the sizes of these triangles.

Unlike C_n, there is no straightforward formula for S_T, making its growth complex and rapid.

Given the combinatorial growth in the number of triangles and their varying sizes, S_T is expected to grow significantly faster than C_n. This rapid growth leads to the observed behavior in the log-log plot where S_T increases more steeply than C_n. While C_n exhibits predictable growth, S_T's growth is more complex and unbounded. This complexity is likely the cause of the rapid increase in S_T relative to C_n.

Suggested Further Investigation

A more detailed mathematical investigation or computational modeling is suggested to precisely characterize the growth of S_T. Potential approaches could include:

- Developing a mathematical model to estimate the growth rate of S_T.
- Computational simulations to observe the behavior of S_T for large values of n.
- Investigating specific geometric configurations and their contributions to S_T.

Centroids and Minimal Spanning Lengths in Triangles

The four "centroids" of a triangle are its geometric centroid, orthocenter, circumcenter, and incenter:

- **geometric centroid** (often simply called the centroid) is the point where the three medians of the triangle intersect where the median of a triangle is a line segment joining a vertex to the midpoint of the opposing side. It is used to determine the *minimal spanning length*, also known as the *Steiner tree length*, which connects the three vertices of a triangle.
- **orthocenter** is the point where the three altitudes of the triangle intersect. An altitude being the perpendicular line from a vertex to the opposite side (or extension of the side).
- **circumcenter** is the center of the circumscribed circle of the triangle being equidistant from all three vertices of the triangle and is the point from which the radii to the vertices are equal.
- **incenter** is the center of the inscribed circle (incircle) of the triangle, tangent to each side and is equidistant from all sides of the triangle.

Because an equilateral triangle is equi-angular, the perpendicular bisectors, the medians, the angle bisectors, and the altitudes are all the same lines so the centroid, circumcenter, incenter, and orthocenter all coincide.

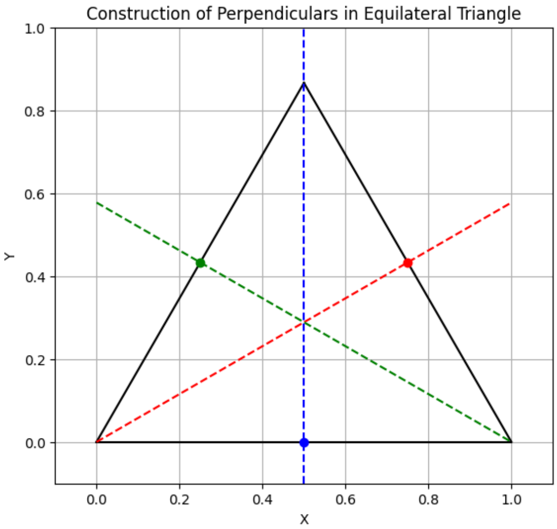


Figure 17.11: Equilateral triangle with perpendicular bisectors

The code, equilateralTriangle-generalForm.ipynb calculates the perpendicular bisectors of the sides of an equilateral triangle using the general form of a straight line equation, $ax+by=c$ to handle the special case of a vertical line, in which $b=0$, so that $x=\frac{c}{a}$. For non-vertical lines, it computes the slope as the negative reciprocal of the slope of the side of the triangle, $m_1 m_2 = -1$.

```
def line_equation(point, midpoint):
    # The perpendicular slope is -1/slope of the side
    if point[0] - midpoint[0] != 0:
        slope = (point[1] - midpoint[1]) / (point[0] - midpoint[0])
        a = -slope
        b = 1
    else:  # If the side is vertical, the perpendicular is horizontal
        a = 1
        b = 0
    c = a * point[0] + b * point[1]
    return a, b, c
```

When plotting the lines, the code checks if $b \neq 0$, which would indicate a non-vertical line, and then calculates y for a range of x values and plots the line. If $b = 0$, indicating a vertical line, the code uses the `plt.axvline` function to plot a vertical line at $x = \frac{c}{a}$. The intersection of the three perpendicular bisectors is also the center of mass of the triangle, assuming it is made of a uniform material. The centroid divides each median in a ratio of 2:1, where the longer segment is between the vertex and the centroid.

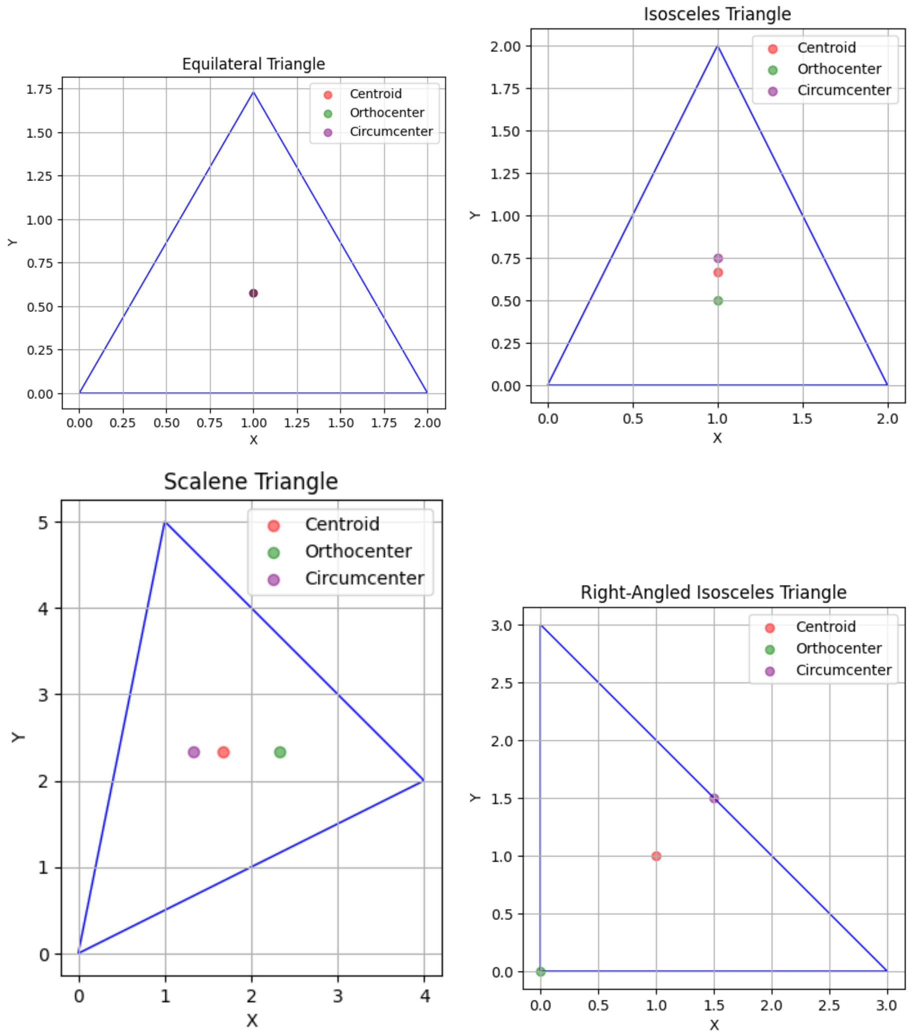


Figure 17.12: Centroid types of some triangles

We wish to lay fibre optic cables, connecting these nodes of commerce, culture, and governance with the least amount of digging and disruption. The geometric centroid (known as the Fermat point of the triangle for non-obtuse triangles) provides the optimal connection point, where the total length of the cable is minimized. The *minimal spanning tree length* provided by the centroid has a length that is $\frac{1}{\sqrt{3}}$ times two thirds of the perimeter (spanning length) of the triangle given that the centroid divides each median in a ratio of $2:1$, and the sum of the distances from the centroid to each vertex is equal to the sum of the medians' lengths divided by 3. Said another way, given side lengths of the triangle as d, the median, m, connecting a vertex to the midpoint of the opposite side, is $m = \frac{\sqrt{3}}{2}d$.

Consider then connecting the three UK cities of Bristol, Birmingham, and London as the vertices of an equilateral triangle.

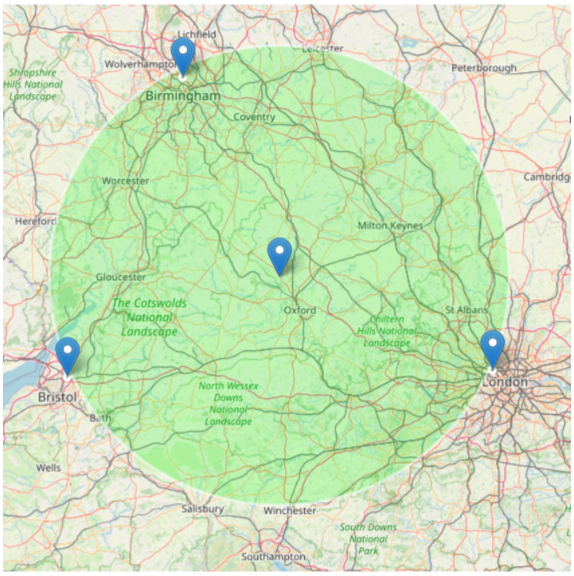


Figure 17.13: UK hubs drawn with mapTool.

The point where medians converge—the centroid—is $\frac{2}{3}m$ from each vertex, or $\frac{d}{2}$ given our equilateral assumption. Thus, the total fibre length required is $3 \times \frac{d}{2}$, a savings of $\frac{d}{2}$ compared to direct inter-city connections. Placing a network hub near the centroid at Oxford ensures equitable service delivery and reduced infrastructure costs.

The distance from the centroid to any vertex is $\frac{2}{3}$ of the median, which is:

$$\text{Distance from Centroid to Vertex} = \frac{2}{3} \times \frac{\sqrt{3}}{2}d = \frac{d}{2}$$

Therefore, the total length of cable required to connect the towns via the centroid is:

$$\text{Total Cable Length} = 3 \times \frac{d}{2} = \frac{3d}{2}$$

This is shorter than the total length of cable that would be required to connect the towns directly to each other, which would result in a total length of ‘2d‘.

For scalene we see the following

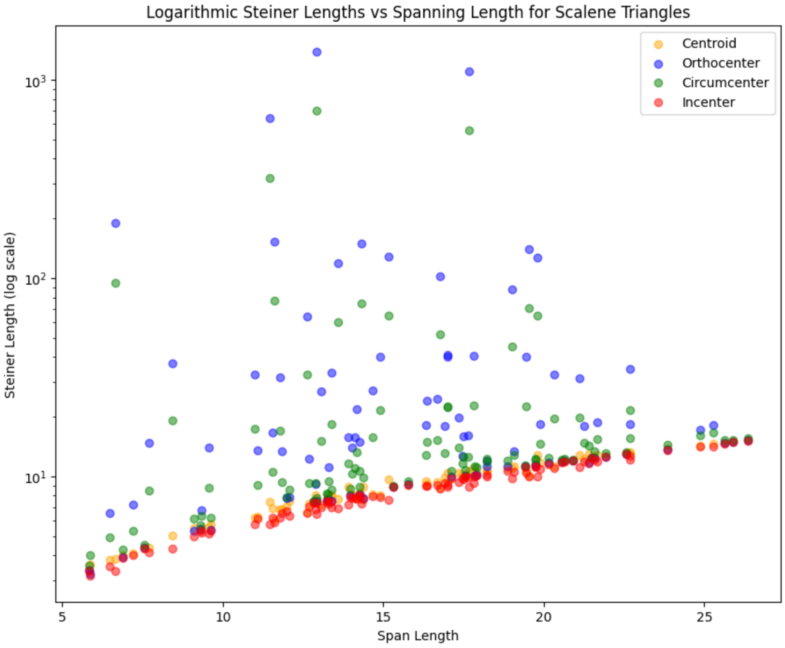


Figure 17.14: Centroid span tree lengths versus spanning Lengths of scalenes

Given that a power law is apparent. Let us focus on the laws that determine Incenter and geog=center

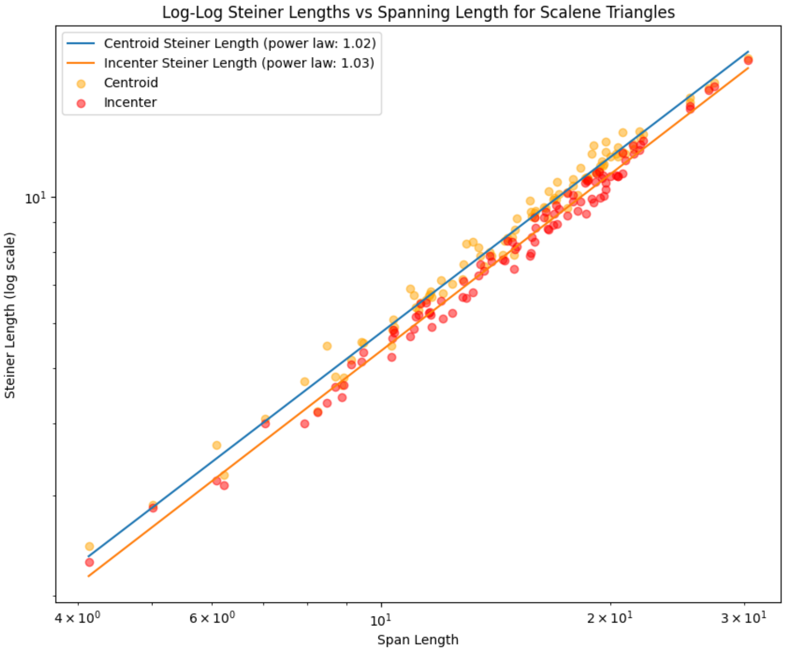


Figure 17.15: Regression of Centroids versus spanning Lengths of scalenes

17.7 spiralling

PrimaryColoredBohmianQuadraticPolygon.ipynb delivers

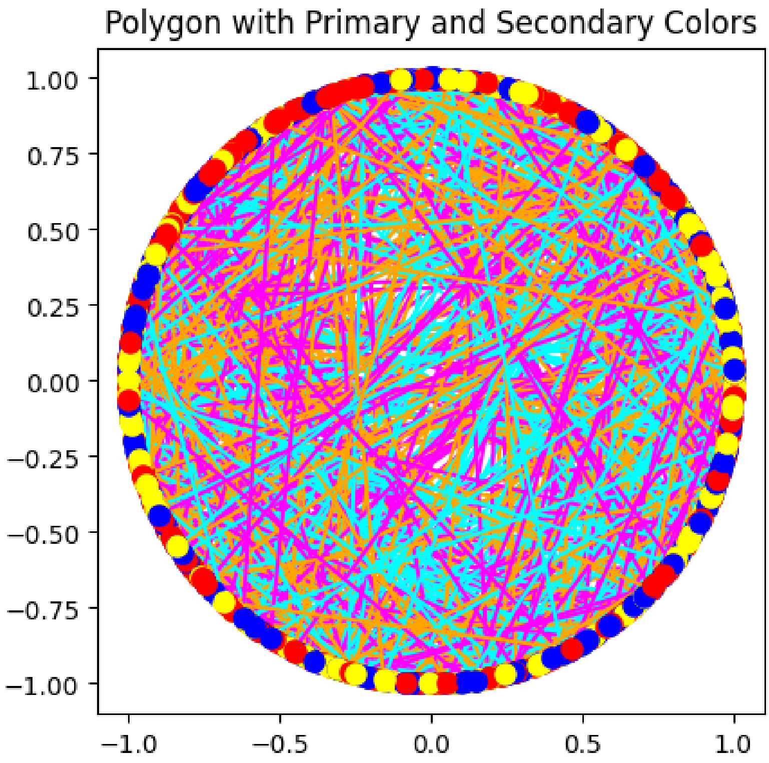


Figure 17.16: Quadratic Scaling

17.8 chapter end notes

Island Area Implementation

By area we have:

Power-Law Fit Implementation

- Old Code: No fitting applied to the histogram data.
- New Code: Applies power-law fitting to the histogram data and plots the fit line.

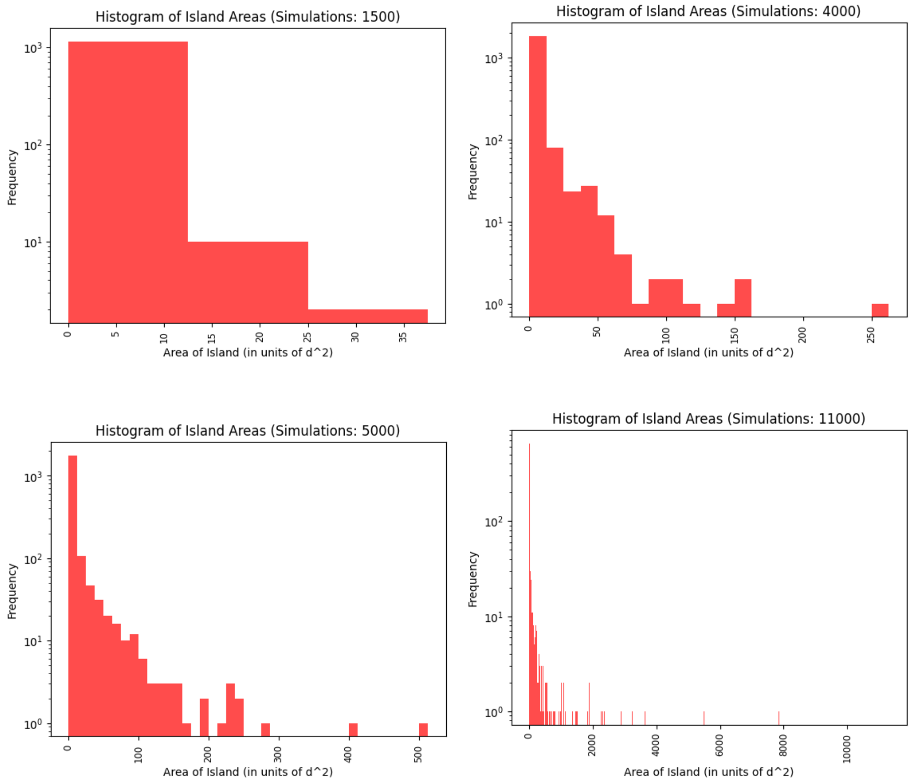


Figure 17.17: By Area Size Coin Island Distributions

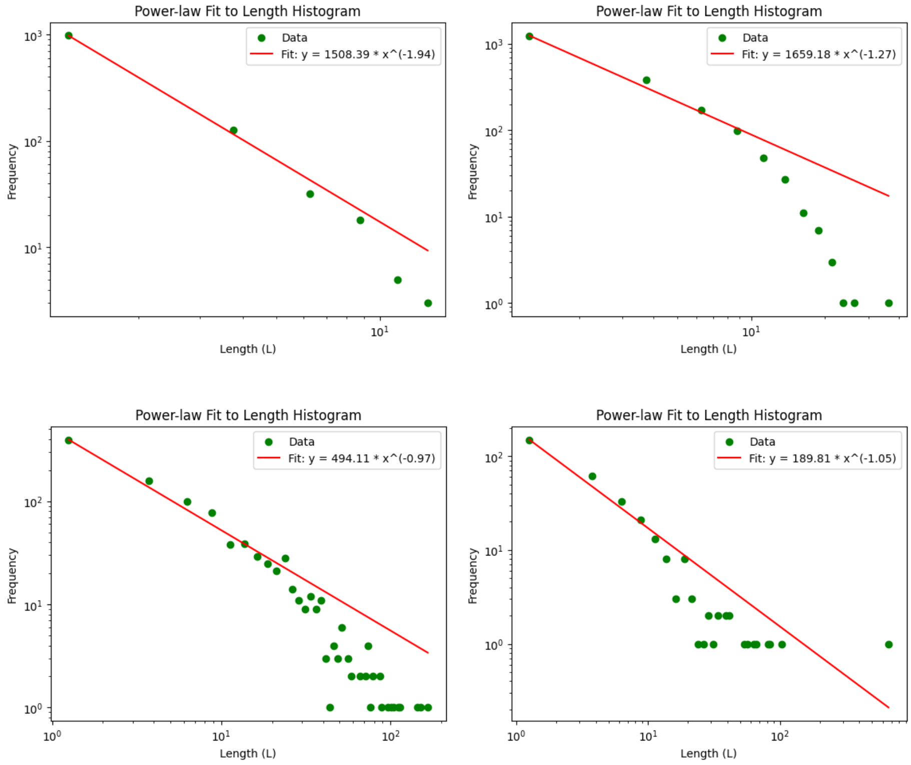
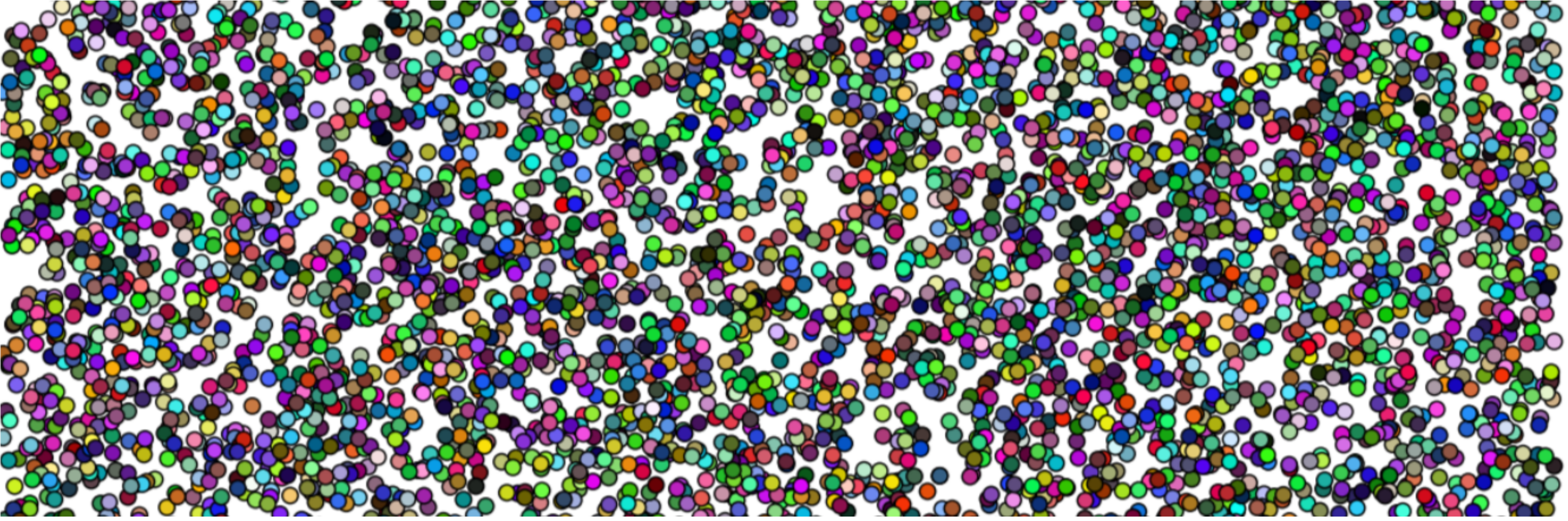


Figure 17.18: By Length Size Coin Island Power Law fit



18. Diophantine Equations

“’‘Geometry is knowledge of the eternally existent.’.”
— *Pythagoras*

1. **Linear Diophantine Equations:** Form: $ax + by = c$.
   Geometric Connection: Represents a line in a plane.
2. **Quadratic Diophantine Equations:** Examples and Geometric Connections:
   - Circle: $x^2 + y^2 = r^2$.
   - Ellipse: $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$.
   - Parabola: $y^2 = 4ax$.
   - Hyperbola: $xy = c$ or $x^2 - y^2 = c$.
3. **Cubic Diophantine Equations:** Example: Mordell’s Equation, $y^2 = x^3 + k$. Special case for the fractional power law equation $y = x^{2/3}$ is $y^2 = x^3$ (Mordell’s Equation with $k = 0$). DiophantineRationalpower.ipynb delivers the following graph

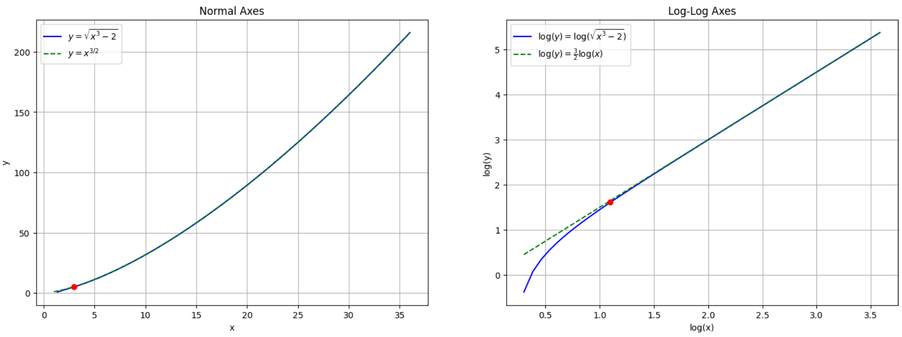


Figure 18.1: The fractional 2/3 power equation as a Diophantine Mordel equation with k=0

4. **igher Degree Equations:** Example: Elliptic curves, $y^2 = ax^4 + bx^3 + cx^2 + dx + e$.
5. **Other Complex Forms:** Diophantine equations can also be in more complex forms, like $a^n + b^n = c^n$ for $n > 2$, which is famously known from Fermat’s Last Theorem.

whereas ellipticCurveStraightLineIntersection.ipynb delivers the elliptical curves

Given a Diophantine equation $ax+by=c$, where a, b, and c are integers, we can use Bezout's identity to find the general solution. Bezout's identity states that if $\gcd(a,b)=d$, then there exist integers u and v such that $au+bv=d$.

18.1 Using Bezout's Identity to Solve Diophantine Equations

To find the general solution of the Diophantine equation, we need to check if the $\gcd(a,b)$ divides c, i.e., $d \mid c$. If d divides c, then we can multiply both sides of Bezout's identity by $\frac{c}{d}$ to get:

$$a\left(\frac{cu}{d}\right)+b\left(\frac{cv}{d}\right)=c$$

Thus, a particular solution of the Diophantine equation is $x_0=\frac{cu}{d}$ and $y_0=\frac{cv}{d}$. The general solution can then be expressed as:

$$x=x_0+\frac{b}{d}t, \quad y=y_0-\frac{a}{d}t$$

where t is an integer.

Let's find the general solution for the Diophantine equation $1485x+1745y=15$:

First, we calculate $\gcd(1485,1745)$:

$$\begin{aligned}
1745 &= 1\cdot 1485+260\\
1485 &= 5\cdot 260+185\\
260 &= 1\cdot 185+75\\
185 &= 2\cdot 75+35\\
75 &= 2\cdot 35+5\\
35 &= 7\cdot 5+0
\end{aligned}$$

Hence, $\gcd(1485,1745)=5$. Since 5 divides 15, we can proceed to find the particular solution. Using Bezout's identity, we have:

$$\begin{aligned}
1485u+1745v &= 5\\
1485(38)+1745(-32) &= 5
\end{aligned}$$

So, $x_0=15\cdot 38=570$ and $y_0=15\cdot -32=-480$.

Therefore, the general solution of the Diophantine equation $1485x+1745y=15$ is:

$$x=570+\frac{1745}{5}t, \quad y=-480-\frac{1485}{5}t$$

where t is an integer.

Analysis of Diophantine Set for Square and Non-Square Numbers

We have that a number a is a *quadratic residue* modulo n if there exists an integer x such that:

$$x^2 \equiv a \pmod{n}. \tag{18.1}$$

For every integer x, there exists an a such that a is a perfect square:

$$a=x^2. \tag{18.2}$$

so that a is always a quadratic residue modulo any n. To identify non-square numbers, we consider the Diophantine set determined by the equation:

$$(a-z^2-x-1)^2+((z+1)^2-a-y-1)^2=0. \tag{18.3}$$

This equation implies that for the sum of two squares to be zero, each term must individually equal zero, leading to the system:

$$(a-z^2-x-1)^2=0, \tag{18.4}$$

$$((z+1)^2-a-y-1)^2=0. \tag{18.5}$$

Solving this system gives:

$$a=z^2+x+1, \tag{18.6}$$

$$a=(z+1)^2+y+1. \tag{18.7}$$

While it is a terribly inefficient way to generate a set excluding square numbers it is useful to observe how easy it to implement such an algorithm with the following snippet from nonSquare.ipynb:

```
def generate_non_squares(limit):
    non_squares = set()
    for x in range(limit):
        for z in range(limit):
            a1 = z**2 + x + 1
            a2 = (z + 1)**2 + 1
            if not is_square(a1) and a1 <= limit:
                non_squares.add(a1)
            for y in range(limit):
                a2y = a2 + y
                if not is_square(a2y) and a2y <= limit:
                    non_squares.add(a2y)
    return sorted(list(non_squares))
```

1. Initialize an empty set `non_squares` and iterate over two variables, x and z, each ranging from 0 up to specified `limit`.
2. Calculate a_1 and a_2 based on the given Diophantine equations, noting that the value of a_2 is initially calculated without the y term, as it will be adjusted in the subsequent loop.
3. Check if a_1 is a non-square integer and within limit. If so, add a_1 to the `non_squares` set.
4. Iterate over a variable y, ranging from 0 up to specified `limit`, to incrementally update a_2:

   $$a_{2y}=a_2+y.$$

5. For each value of a_{2y}, check if it is a non-square integer and within the specified limit. If so, add a_{2y} to the `non_squares` set.

Quadratic Residues and Gauss's Law of Reciprocity

Quadratic residues are pivotal in the study of modular arithmetic, particularly concerning the solvability of quadratic equations and the residue of a square number with respect to a divisor n is key to this understanding: if a number a yields a (*residue* remainder r when divided by n, and if some square number also results in the remainder r, then a is termed a quadratic residue modulo n.

Quadratic Residue Example with $n = 5$ and $a = 3$:

Consider $a = 3$. We aim to determine if an integer x exists such that $x^2 \equiv 3 \mod 5$.

- Testing integers reveals $x = 2$ as a potential solution; however, $2^2 = 4$, and $4 \mod 5 = 4$.
- No integer squared gives a remainder of 3 when divided by 5.
- Hence, 3 is a *non-residue* modulo 5.

In Legendre symbol notation, this is expressed as:

$$\left(\frac{3}{5}\right) = -1 \quad \text{(3 is a non-residue modulo 5)}$$
$$\left(\frac{5}{3}\right) = -1 \quad \text{(5 is a non-residue modulo 3)}$$

Quadratic Residue Example with $n = 7$ and $a = 2$:

Consider $a = 2$. We seek an integer x such that $x^2 \equiv 2 \mod 7$.

- $x = 3$ satisfies this, as $3^2 = 9$ and $9 \mod 7 = 2$.
- Therefore, 2 is a quadratic residue modulo 7, with 3 as its square root modulo 7.

In Legendre symbol notation, this is expressed as:

$$\left(\frac{2}{7}\right) = 1 \quad \text{(2 is a residue modulo 7)}$$
$$\left(\frac{7}{2}\right) = -1 \quad \text{(7 is a non-residue modulo 2)}$$

These hint that there is an underlying law, inevitably formally captured by Gauss: his Law of Quadratic Reciprocity which states:

$$\left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) = (-1)^{\frac{(p-1)(q-1)}{4}}$$

for two odd primes p and q.
For the primes $p = 3$ and $q = 5$:

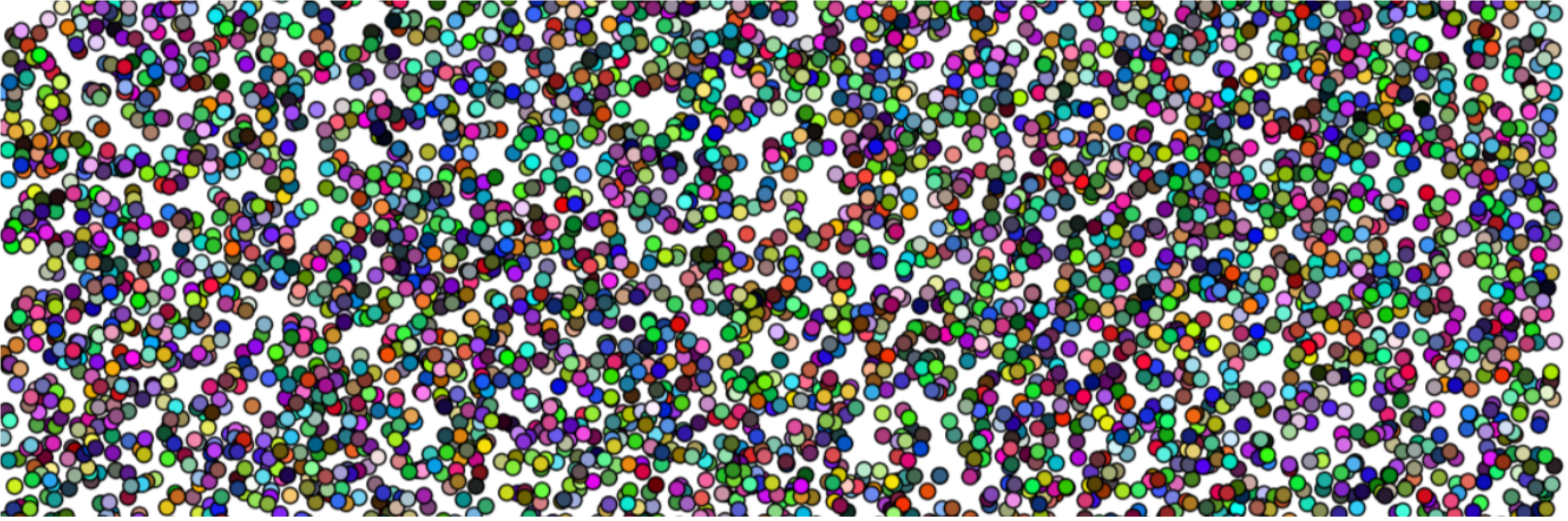
$$(-1) \cdot (-1) = 1 \quad \text{matches} \quad (-1)^{\frac{(3-1)(5-1)}{4}} = 1$$

For the primes $p = 2$ and $q = 7$ we have:

$$(1) \cdot (-1) = -1 \quad \text{matches} \quad (-1)^{\frac{(2-1)(7-1)}{4}} = -1$$

18.1.1 Lattice Point Diophantinism

The Diophantine equation $ax+by=c$ has integer solutions when $\gcd(a,b)$ is a divisor of c. Consider the family of lines L_n that traverse all integer lattice points. The equation of a line in the form $y=mx+y_0$ can be used to describe the lines L_n.



19. Determination

Consider a triangle with vertices at $(0,0)$, $(4,0)$, and $(0,3)$. The area A can be calculated using the determinant formula for vertices (x_1,y_1), (x_2,y_2), and (x_3,y_3):

$$A = \frac{1}{2}\left|\begin{vmatrix} 0 & 0 & 1 \\ 4 & 0 & 1 \\ 0 & 3 & 1 \end{vmatrix}\right| = \frac{1}{2}\left|0\cdot 0\cdot 1 + 4\cdot 3\cdot 1 + 0\cdot 0\cdot 1 - 0\cdot 3\cdot 1 - 4\cdot 0\cdot 1 - 0\cdot 0\cdot 1\right| = 6$$

To confirm this result with the $\frac{1}{2}ab\sin(C)$ formula, observe:
- Side lengths a and b are 4 and 3, respectively, forming the right angle $C = 90°$. - Thus, $\sin(C) = \sin(90°) = 1$.

The area calculation is therefore:

$$A = \frac{1}{2}\cdot 4\cdot 3\cdot 1 = 6$$

Both methods yield an area of 6 square units for the triangle, illustrating that the determinant approach and the $\frac{1}{2}ab\sin(C)$ formula are consistent and provide the same result for the area of a triangle.

19.1 Pythagorean Triples

Pythagorean triples are the set of right angled triangles whose three sides have Natural numbers as lengths. All Pythagorean triple such as $(3,4,5)$, has both integer Perimeters and Areas. $A_{345} = 6$ and $P_{345} = 12$. Another well-known triple, $(5,12,13)$, also possesses an integer area $A_{(5,12,13)}$ of 30.

- **Pythagorean Triples**: These are sets of three positive integers a, b, and c such that $a^2 + b^2 = c^2$. They represent the sides of right triangles with all integer side lengths.

- **Right Triangles with Integral Areas**: Right triangles that have an area that can be expressed as an integer. Notably, these triangles don't necessarily have rational hypotenuse lengths.

A typical formula to generate Pythagorean triples is:

$$\begin{aligned}
a &= m^2 - n^2, \\
b &= 2mn, \\
c &= m^2 + n^2,
\end{aligned}$$

where m and n are coprime (i.e., their greatest common divisor is 1), $m > n$, and they are not both odd (i.e., one of them is even). From this formula, we observe that:

1. a will always be even, because it's the difference of two squares.
2. b will always be even, due to the presence of a factor of 2.
3. c can be odd if both m and n are even.

Therefore, it's impossible for both a and b to be odd simultaneously. This implies there are no Pythagorean triples where both of the two shorter sides (legs) are odd integers.

One can generate scaled versions of triples like $(6, 8, 10)$ that are just multiples of $(3, 4, 5)$ but we are interested only in those like $(7, 24, 25)$ and $(8, 15, 17)$ that are not a multiple of another triple and as such are called *primitive*. Integral triangles are triangle whose areas also take on integer values. The natural question is the set of integral non pythagorean triple triangles as big as the triple set itself.

The n^{th} triangle number, T_n, is given by:

$$T_n = \frac{n(n+1)}{2}$$

Given the standard form of a Pythagorean triple $(m^2 - n^2, 2mn, m^2 + n^2)$, if we set $m = n + 1$:

1. First side (difference of squares):

   $$m^2 - n^2 = 2n + 1$$

   This is always odd.
2. Second side (product of m and n):

   $$2mn = 2n(n+1)$$

   This is twice the n^{th} triangular number and is always even.
3. Third side (sum of squares):

   $$m^2 + n^2 = 2n^2 + 2n + 1$$

   This is always odd.

For a right triangle with legs a and b, the area A is:

$$A = \frac{1}{2}ab$$

For the area to be an integer, at least one of a or b must be even. In our generated triple where $m = n + 1$, one side (the one involving the triangular number) is even, while the other is odd. This ensures the triangle's area is integral.

19.1.1 Generating Pythagorean Primitives

Generating the ordered set of primitive Pythagorean triples from the set of straight lines with rational gradients that intersect the unit circle, reveals some interesting structure as detailed in Kaplan's book, 'Hidden Harmonies', [8]. Now, we will find the lines of the form $y = mx + m$ that emanate from the point $P = (-1, 0)$ and intersect the unit circle centered at $(0, 0)$ at a point $Q = (q, r)$ with rational coordinates q and r. The numerator and denominator of q and r will reveal the Pythagorean triples. Let's substitute $y = mx + m$ into the equation of the unit circle:

$$\begin{aligned} x^2 + y^2 &= 1, \\ x^2 + (mx+m)^2 &= 1, \\ x^2 + m^2x^2 + 2m^2x + m^2 &= 1, \\ (1+m^2)x^2 + 2m^2x + (m^2-1) &= 0. \end{aligned}$$

Using the quadratic formula, we can solve for x:

$$\begin{aligned} x &= \frac{-2m^2 \pm \sqrt{(2m^2)^2 - 4(1+m^2)(m^2-1)}}{2(1+m^2)}, \\ x &= -m^2 \pm \frac{\sqrt{4m^4 - 4(1+m^2)(m^2-1)}}{2(1+m^2)} = -m^2 \pm \frac{\sqrt{4m^4 - 4(m^4-1)}}{2(1+m^2)}, \\ &= -m^2 \pm \frac{\sqrt{4}}{2(1+m^2)} = \frac{1-m^2}{1+m^2} \end{aligned}$$

Now, substitute x back into the equation $y = mx + m$ to get y:

$$\begin{aligned} y &= m\left(\frac{1-m^2}{1+m^2}\right) + m = \frac{m - m^3 + m(1+m^2)}{1+m^2}, \\ &= \frac{-m^3 + 2m^2 + m + m^3}{1+m^2} = \frac{2m}{1+m^2} \end{aligned}$$

The points of intersection (q, r) with the unit circle and the set of straight lines with rational gradient values of m, emanating from $(-1, 0)$ deliver the Pythagorean triangle side lengths as co-ordinates. The three natural numbers a, b, c making up the fractions of those coordinate pairs $latex(\frac{a}{c}, \frac{b}{c})$ are our triples. Each straight line from $latex(-1, 0)$ has the form

$$y = \frac{m}{n}x + \frac{m}{n}.$$

For example, the $(3, 4, 5)$ triple arises from the intersecting yellow line above,

$$y = \frac{\frac{4}{5} - 0}{\frac{3}{5} - (-1)}x + y(0) = \frac{4}{5}\frac{8}{5}x + \frac{1}{2} = \frac{1}{2}x + \frac{1}{2}.$$

Coding of `intersection_point(m)`

The function that calculates the intersection point of a line, with gradient m, emanating from the point (-1, 0) and intersecting a unit circle centered at (0, 0). It returns the coordinates (x, y) of the intersection point with the unit circle defined using numpy arrays:

- `circle_x`: Represents 400 linearly spaced values between -1 and 1.
- `circle_y_positive`: Computes the positive y-values corresponding to `circle_x` based on the equation of the unit circle $y = \sqrt{1-x^2}$.

- `circle_y_negative`: Represents the negative counterpart of `circle_y_positive`.

The figure overleaf with size 8x8 is initialized. The positive and negative y-values of the unit circle are plotted against `circle_x` using a red color. A list of specific gradients (given as fractions) is defined in `highlighted_m`. These represent the slopes of specific lines of interest. Corresponding colors for these lines are defined in `highlighted_colors`.

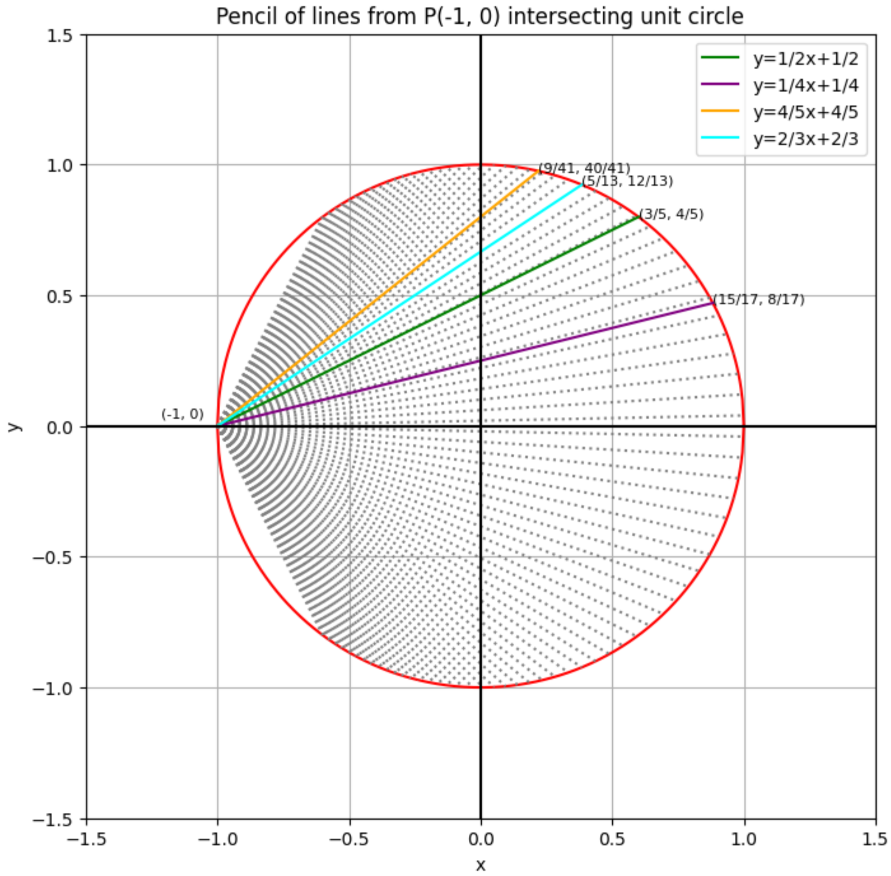


Figure 19.1: Pythagorean Triples as co-ordinate intersections of unit circle with rational gradient straight lines emanating from (-1,0)

The code to draw this is UnitCirclePythagTriples.ipynb with crucial snippet:

```
def intersection_point(m):
    x = (1 - m**2) / (1 + m**2)
    y = (2 * m) / (1 + m**2)
    return x, y
# Define the unit circle
circle_x = np.linspace(-1, 1, 400)
circle_y_positive = np.sqrt(1 - circle_x**2)
circle_y_negative = -np.sqrt(1 - circle_x**2)
# Plotting
fig, ax = plt.subplots(figsize=(8,8))
ax.plot(circle_x, circle_y_positive, ’r’)
ax.plot(circle_x, circle_y_negative, ’r’)
highlighted_m = [Fraction(1, 2), Fraction(1, 4), Fraction(4, 5), Fraction(2, 3)]
highlighted_colors = [’green’, ’purple’, ’orange’, ’cyan’]
```

19.1.2 Pythagorean scatterplots

Plotting primitive triples' hypotenuse against their respective largest acute angle reveals a beaded curtain surely worthy of some more investigation if not in the league of Illustrating Mathematics: hypoteneusevsAcutsPPT.ipynb generates a scatterplot of Acute angles versus hypoteneuse length of primitives.

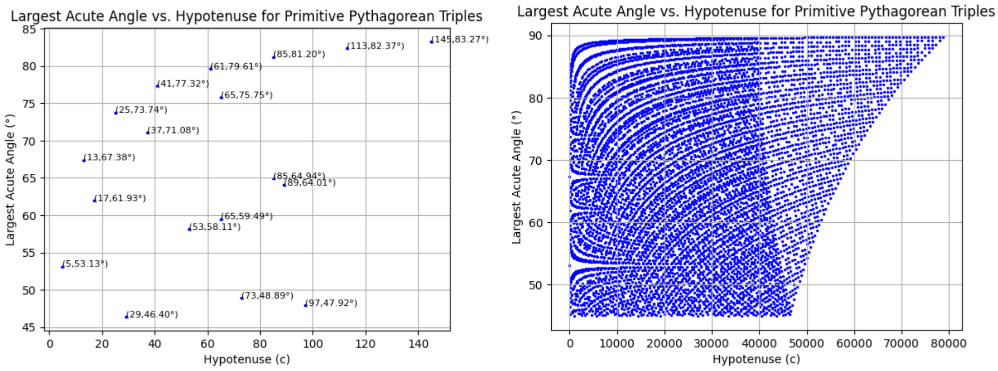


Figure 19.2: Largest Acute angle versus Hypotenuse of Pythagorean Triples

```
    def generate_triples(limit):
    triples = []
    for m in range(2, limit):
        for n in range(1, m):
            if (m - n) % 2 == 1 and np.gcd(m, n) == 1:
                a = m**2 - n**2
                b = 2*m*n
                c = m**2 + n**2
                triples.append((a, b, c))
    return triples
triples = generate_triples(10)
```

The function `generate_triples(limit)` aims to generate primitive Pythagorean triples.

- `limit` (integer): An upper boundary for the values of m. This parameter determines the range within which the Pythagorean triples will be generated.
- A list of tuples, where each tuple (a,b,c) represents a primitive Pythagorean triple.

1. Initialize an empty list called `triples` to store the resulting triples.
2. For each integer m from 2 up to (but not including) `limit`:
    - For each integer n from 1 up to (but not including) m:
        - Check if $m-n$ is odd and the greatest common divisor (GCD) of m and n is 1 (ensuring the generated triple is primitive).
        - If both conditions are met:
        - Append the tuple (a,b,c) to the `triples` list.
3. Return the list `triples`.

The line `triples = generate_triples(10)` outside the function calls `generate_triples` with a limit of 10 and stores the resulting list of triples in the variable `triples`.

logAvsAcutePPT.ipynb delivers a log(area) versus the largest acute angle of the triangle.

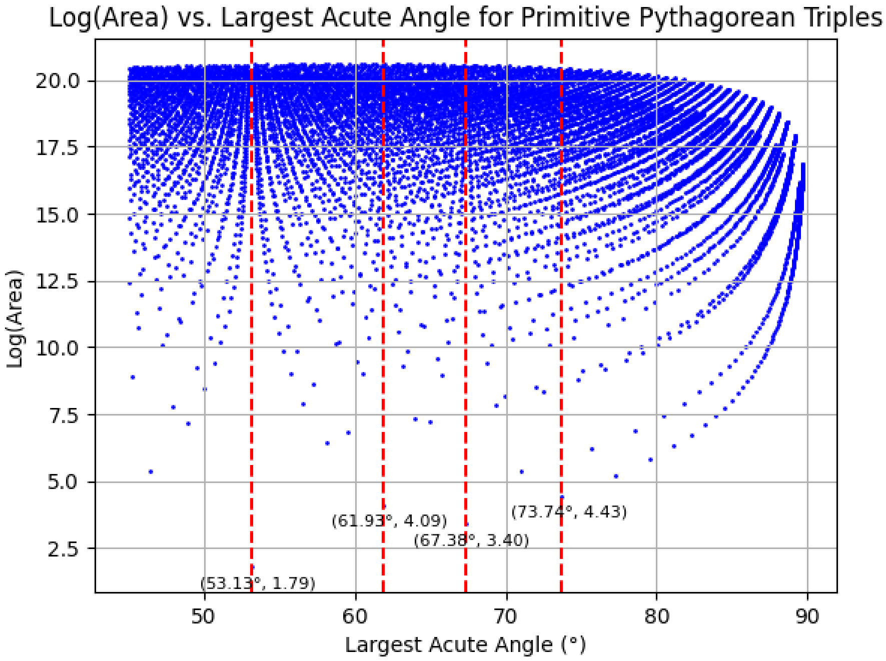


Figure 19.3: The bead curtain of 6000 triples with detractor primitives highlighted.

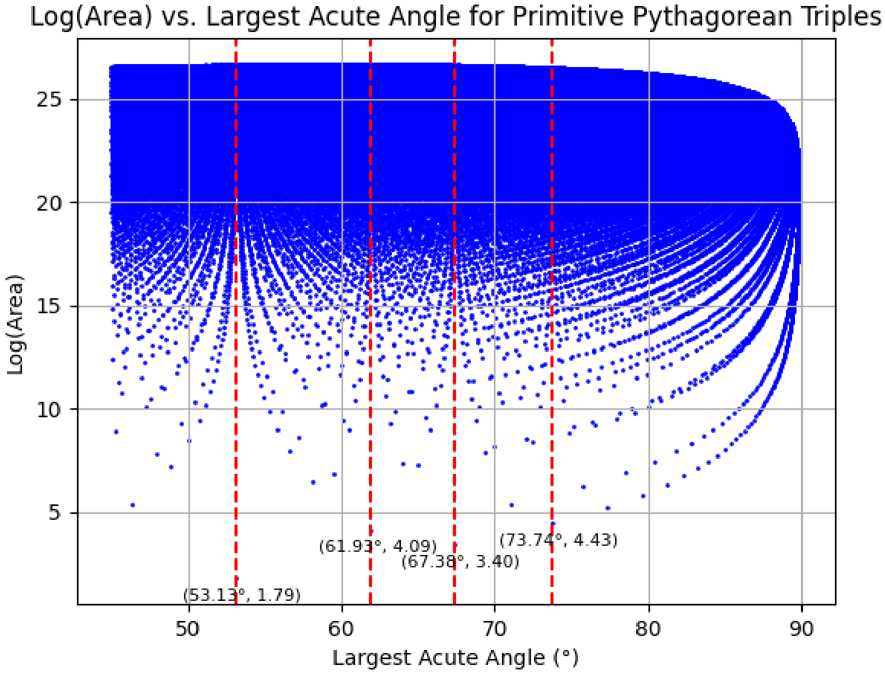


Figure 19.4: A fuller detractor curtain of 30,000 triples.

Those triples with the smallest areas are easily identified being the early stage small triples that seem to be strange detractors.

19.1.3 Annotated log-log plots

Plotting the two shorter sides of the triple in a scatter gram and labelling with respective hypotenuse length is delivered by the following code loglogm-nsquaredPPT.ipynb additionally draws the following scatterplot revealing some limiting structures within the triples.

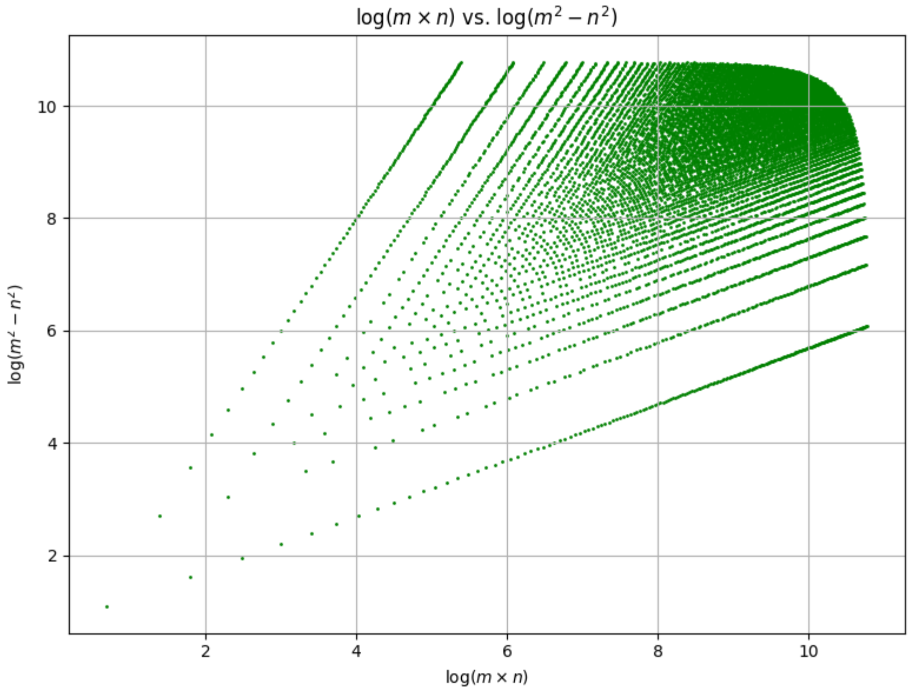


Figure 19.5: Plot of the two shorter sides of the triple in a scattergram.

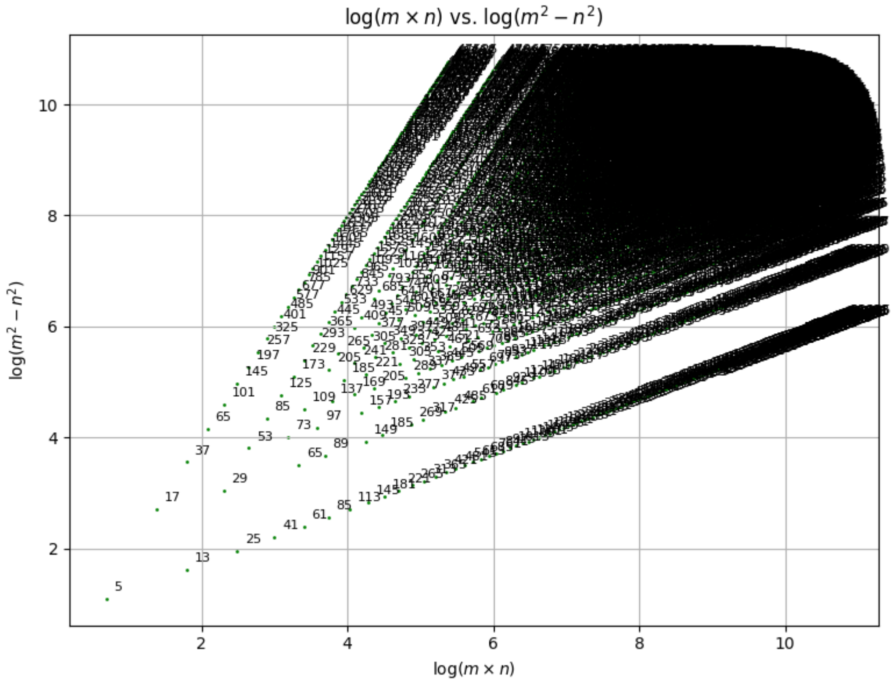


Figure 19.6: Plot of the two shorter sides of the triple in a scattergram.

19.2 Hyperbolic Construction of Pythagorean Triples

Our classic construction of Pythagorean triples using the intersection of a line $y = mx + m$ with a unit circle, defined by $x^2 + y^2 = 1$, works due to the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$ being of Pythagorean triple form, $a^2 + b^2 = c^2$. Consider instead of a circle, a square hyperbola, defined by

$$x^2 - y^2 = a^2, \tag{19.1}$$

for $a = 1$ and similarly invoke the hyperbolic functions sinh and cosh which are to the hyperbola as their trigonometric function cousins are to the circle. The identity for hyperbolic functions that parallels the Pythagorean identity for trigonometric functions is:

$$\cosh^2\theta - \sinh^2\theta = 1. \tag{19.2}$$

Consider a hyperbola defined by (19.1) and a line defined by the equation $y = mx$, where $m = \tan(\theta)$ is the slope of the line. To find the intersection points, we substitute the equation of the line into the hyperbola equation:

$$x^2 - (mx)^2 = a^2 \text{ so, } x^2(1-m^2) = a^2.$$

Solving for x and considering only the positive square root for simplicity, we have the intersection point of the line with the hyperbola in the first quadrant given by

$$x^2 = \frac{a^2}{1-m^2}, \text{ so that } x = \pm\sqrt{\frac{a^2}{1-m^2}} \text{ and } \left(\sqrt{\frac{a^2}{1-m^2}}, m\cdot\sqrt{\frac{a^2}{1-m^2}}\right).$$

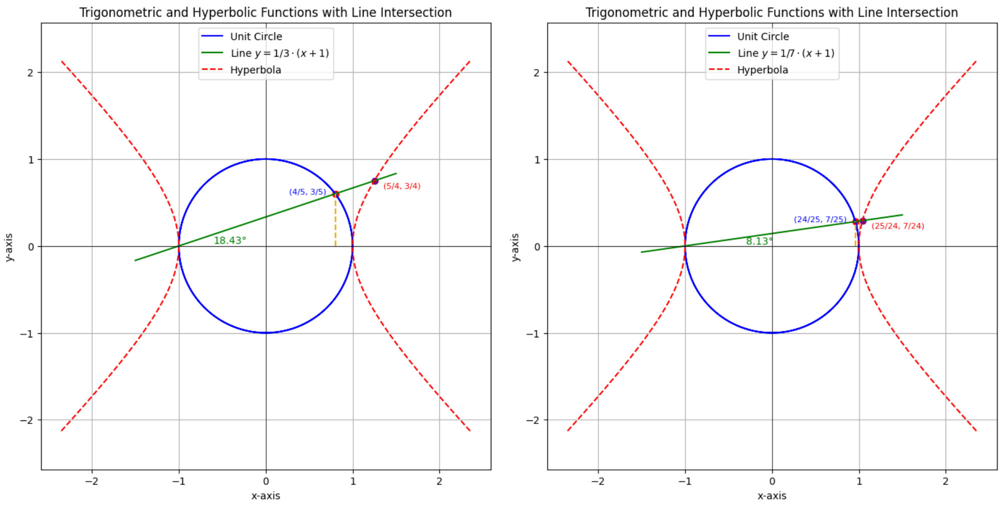


Figure 19.7: Pythagorean Triples from Unit CIrcle and Hyperbola

A line $y = m(x+1)$ with rational slope $m = \frac{p}{q}$ intersecting the unit circle and hyperbola curves yield points with rational coordinates. It is on the unit circle, that these points are usually related to the Pythagorean triples. But as you can see from figure () this can equally an arguably more elegantly be revealed on the square hyperbola.

The code UnitCircleHyperbola.ipynb

19.2.1 Diophantine Equations and Modular Functions:

Exploring variations of the circle's equation, like elliptic curves or general Diophantine equations, often requires the use of modular functions. These functions provide a powerful framework for analyzing properties of elliptic curves, including their rational points and connections to modular forms.

The equation of a square hyperbola (19.1) can be parametriseed by the hyperbolic functions as,

$$x = a\cosh\theta, \quad y = a\sinh\theta.$$

which when substituted into (19.1) we see the hyperbolic identity (19.2) emerge:

$$(a\cosh\theta)^2 - (a\sinh\theta)^2 = a^2 \tag{19.3}$$

Given the hyperbola equation $x^2 - y^2 = a^2$ and a line equation $y = mx + c$, we find the intersection points by substituting the line equation into the hyperbola equation: $x^2 - (mx+c)^2 = a^2$, expanding and rearranging, to get a quadratic equation in terms of x:

$$x^2 - (m^2x^2 + 2mcx + c^2) = a^2.$$

Simplifying further this reads as:

$$(1-m^2)x^2 - 2mcx - (c^2 + a^2) = 0,$$

which is a quadratic equation of the form $Ax^2 + Bx + C = 0$, where $A = 1 - m^2$, $B = -2mc$, and $C = -(c^2 + a^2)$ and is implemented in the following code, cycleHyperbolaPythagTriples.ipynb. The above intersection is achieved with the following snippet:

```
def find_hyperbola_intersection(m, c, a):
    A = 1 - m**2
    B = -2 * m * c
    C = -(c**2 + a**2)

    discriminant = B**2 - 4 * A * C
    if discriminant < 0:
        return []  # No real intersections
    else:
        x1 = (-B + np.sqrt(discriminant)) / (2 * A)
        x2 = (-B - np.sqrt(discriminant)) / (2 * A)
        y1 = m * x1 + c
        y2 = m * x2 + c
        return [(x1, y1), (x2, y2)]
```

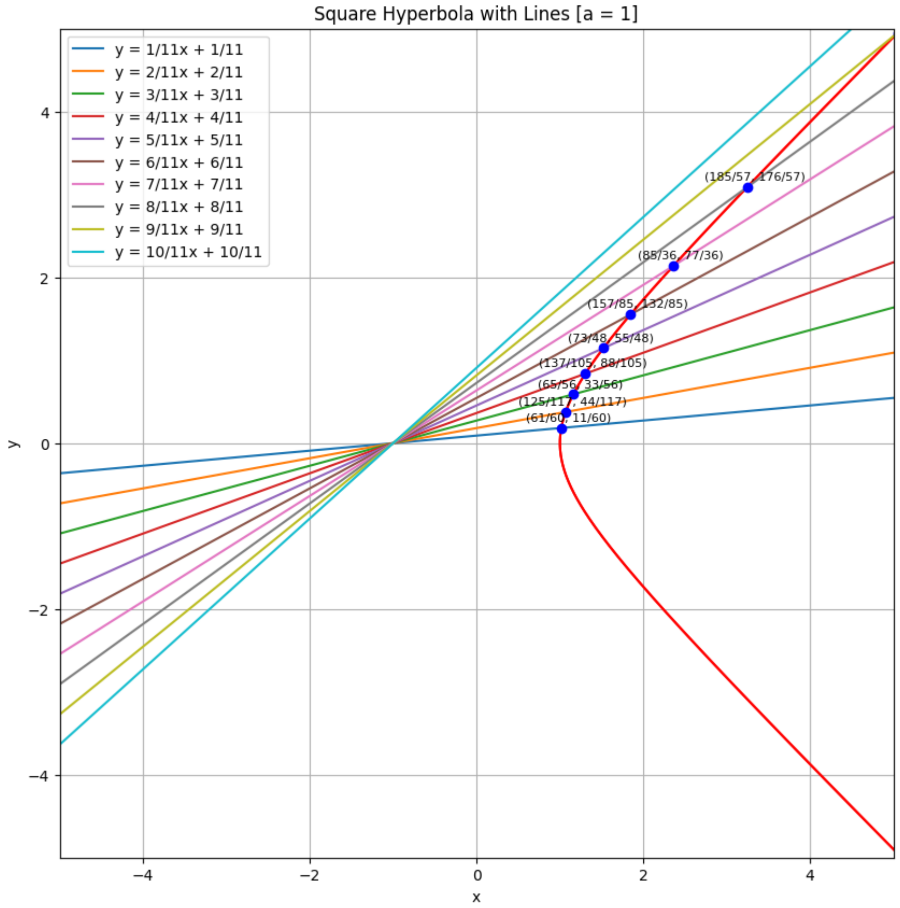


Figure 19.8: Pythagorean Triples from the elevenths.

The snippet sorts the final list

```
for m, c in lines:
    frac_m = Fraction.from_float(m).limit_denominator()
    frac_c = Fraction.from_float(c).limit_denominator()
    line_x = np.linspace(-5*a, 5*a, 400)
    line_y = m * line_x + c
    ax.plot(line_x, line_y, label=f’y = {frac_m}x + {frac_c}’)
    intersections = find_hyperbola_intersection(m, c, a)
    for x_int, y_int in intersections:
        if not (np.isclose(x_int, -1) and np.isclose(y_int, 0)):  # Exclude (-1, 0)
            frac_x = Fraction.from_float(x_int).limit_denominator()
            frac_y = Fraction.from_float(y_int).limit_denominator()
            pythagorean_result = check_pythagorean(frac_x, frac_y)
            intersection_data = (f’{frac_m}’, f’({frac_x}, {frac_y})’, pythagorean_result)
                if intersection_data not in all_intersections:
                    all_intersections.add(intersection_data)
sorted_intersections = sorted(list(all_intersections), key=lambda x: float(Fraction(x[0])))
```

A sorted table of triples constructed from hyperbola intersections follows.

Line Gradient, m	Intersection Coordinate	Pythagorean Triple
1/8	(65/63, 16/63)	$63^2+16^2=65^2$
1/7	(25/24, 7/24)	$24^2+7^2=25^2$
1/6	(37/35, 12/35)	$35^2+12^2=37^2$
1/5	(13/12, 5/12)	$12^2+5^2=13^2$
1/4	(17/15, 8/15)	$15^2+8^2=17^2$
2/7	(53/45, 28/45)	$45^2+28^2=53^2$
1/3	(5/4, 3/4)	$4^2+3^2=5^2$
3/8	(73/55, 48/55)	$55^2+48^2=73^2$
2/5	(29/21, 20/21)	$21^2+20^2=29^2$
3/7	(29/20, 21/20)	$20^2+21^2=29^2$
1/2	(5/3, 4/3)	$3^2+4^2=5^2$
4/7	(65/33, 56/33)	$33^2+56^2=65^2$
3/5	(17/8, 15/8)	$8^2+15^2=17^2$
5/8	(89/39, 80/39)	$39^2+80^2=89^2$
2/3	(13/5, 12/5)	$5^2+12^2=13^2$
5/7	(37/12, 35/12)	$12^2+35^2=37^2$
3/4	(25/7, 24/7)	$7^2+24^2=25^2$
4/5	(41/9, 40/9)	$9^2+40^2=41^2$
5/6	(61/11, 60/11)	$11^2+60^2=61^2$
6/7	(85/13, 84/13)	$13^2+84^2=85^2$

19.3 Congruent Numbers and Square-Free Conditions

A congruent number, N, is defined as a positive integer that can be represented as the area, A, of a right-angled triangle with rational side lengths. We note that if N is a congruent number and s is a square integer such that the Möbius function $\mu(s) = 0$ (indicating that s is not square-free), then the product $N \cdot s$ is also a congruent number. To be sure then a square-free integer is just an integer that is not divisible by any perfect square other than 1. Consider the congruent number $N = 5$, associated with the right-angled triangle with sides $\left(\frac{40}{6}, \frac{9}{6}, \frac{41}{6}\right)$. Multiplying this triangle by a square integer, say $s = 4$, yields a new triangle with sides $\left(\frac{40}{6} \cdot 2, \frac{9}{6} \cdot 2, \frac{41}{6} \cdot 2\right)$, and the area of this new triangle is $5 \cdot 4 = 20$, which is also a congruent number. However, it is the square-free part of the number, the "primitive", $N = 5$, that is of primary interest.

Congruent numbers can be characterized based on their congruence class modulo 8. Specifically, numbers that are congruent to 5, 6, or 7 modulo 8 are always congruent. The following table lists congruent numbers less than 100, categorized by their residue modulo 8.

Congruent Numbers by Residue Classes Modulo 8

Congruent numbers have an interesting distribution when considered modulo 8.

n	$8n+5$	$8n+6$	$8n+7$
0	5	6	7
1	13	14	15
2	-	-	23
3	29	30	-
4	-	38	39
5	45	46	47
6	53	54	55
7	-	62	63
8	69	70	71
9	77	78	79
10	85	86	87
11	-	94	95
12	101	102	103
13	109	110	111
14	-	118	119
15	125	126	127
16	133	134	135
17	-	142	143
18	149	150	151
19	157	158	159
20	165	166	167
21	-	174	175
22	181	182	183
23	189	190	191
24	197	198	199

19.3.1 Congruent Numbers and Elliptic Curves

The congruent number problem can be translated into the problem of finding rational points on the elliptic curve defined by $y^2 = x^3 - N^2x$. Considering the sides of the $N = 7$ congruent triangle as $a = \frac{288}{60}, b = \frac{175}{60}, c = \frac{337}{60}$. That the area, $N = \frac{1}{2}ab$, of the triangle is a congruent number follows as $ab = 2N$. The associated elliptic curve $y^2 = x^3 - 49x$. Any point $P(x,y)$ on the elliptic curve corresponds to the sides of a triangle. A particular solution (x,y) dictates the rational lengths of the sides of the triangle, embedding the geometric property of the triangle into the algebraic structure of the elliptic curve.

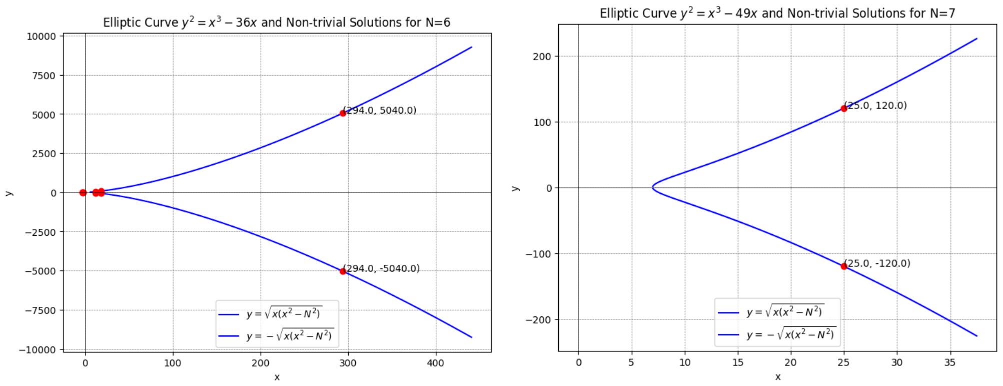


Figure 19.9: Elliptic curve with non primitive Congruent Number 6 and 7 identified

- N=5 the point $(45,300)$ corresponds to triangle side$(3/4,100/27,410/9)$
- N=6 the point $(294,5040)$ corresponds to triangle side $(7/20,720/343,14412/49)$
- N=7 the point $(25,120)$ corresponds to triangle side $(35/24,1176/125,674/25)$

For a given congruent number N, and a point (x,y) on the elliptic curve $y^2 = x^3 - N^2x$, there exists a method utilizes the rational point (x,y) to express the triangle's sides in terms of N, as follows:

- The side a as one of the perpendicular sides of the triangle is found using the formula $a = \frac{y}{x} \cdot N$.
- The side b as the other perpendicular side is found using the formula $b = \frac{y}{x^2} \cdot N^2$.
- The hypotenuse c is found using the formula $c = \frac{x^2+N^2}{x}$.

These translation of the abstract notion of points on an elliptic curve into the concrete geometric terms of a right-angled triangle is particularly important in the proof of the Birch and Swinnerton-Dyer conjecture for a given N, which is one of the Millennium Prize Problems. The conjecture relates the number of rational points on an elliptic curve to the behavior of an associated L-function at $s = 1$, and congruent numbers play a key role in its formulation.

Finding Rational Points on Elliptic Curves

The Python code aims to find non-trivial rational points on the elliptic curve defined by $y^2 = x^3 - N^2x$, where N is a given positive integer known as a congruent number. A non-trivial rational point on this curve corresponds to a right-angled triangle with rational side lengths and an area equal to N.

Significance of the Coordinates

For a given congruent number N, any non-trivial rational point (x,y) on the elliptic curve implies the existence of a right-angled triangle with rational sides a,b,c (where c is the hypotenuse) and area N. Specifically, the x-coordinate can be associated with the leg of such a triangle, while the y-coordinate is related to the difference in the areas of two squares that describe the Pythagorean relationship. The Python code identifies these points through an algorithmic search and visually represents them on the curve, offering a graphical and numerical demonstration of the congruent number problem.

Python Code Functionality

The code performs the following functions:

- Defines the elliptic curve function for a given N.

- Searches for rational points on the curve within a specified range.
- Filters out trivial solutions where $y = 0$, as these do not correspond to triangles.
- Plots the elliptic curve and annotates the found non-trivial rational points.

Connection between Elliptic Curve and a System of Equations

Consider the elliptic curve given by the equation:

$$y^2 = x^3 - N^2x$$

We aim to show a connection between this elliptic curve and a system of two equations of the form:

$$\begin{aligned} x^2 - ay^2 &= z^2 \\ x^2 + ay^2 &= t^2 \end{aligned}$$

We proceed by homogenizing the elliptic curve with a new variable w, assuming $w \neq 0$, and then de-homogenizing by setting $w = 1$ at the end:

$$\begin{aligned} wy^2 &= x^3 - N^2xw^2 \\ y^2 &= \frac{x^3}{w} - N^2xw \end{aligned}$$

Now let's introduce new variables z and t such that:

$$\begin{aligned} z &= \frac{x^2}{w} - Nyw \\ t &= \frac{x^2}{w} + Nyw \end{aligned}$$

Squaring both z and t, we obtain:

$$\begin{aligned} z^2 &= \left(\frac{x^2}{w} - Nyw\right)^2 = \frac{x^4}{w^2} - 2Nxy^2 + N^2y^2w^2 \\ t^2 &= \left(\frac{x^2}{w} + Nyw\right)^2 = \frac{x^4}{w^2} + 2Nxy^2 + N^2y^2w^2 \end{aligned}$$

Adding and subtracting these equations gives us:

$$\begin{aligned} z^2 + t^2 &= 2\frac{x^4}{w^2} + 2N^2y^2w^2 \\ z^2 - t^2 &= -4Nxy^2 \end{aligned}$$

By setting $w = 1$, we can simplify to:

$$\begin{aligned} z^2 + t^2 &= 2x^4 + 2N^2y^2 \\ z^2 - t^2 &= -4Nxy^2 \end{aligned}$$

Consider the elliptic curve given by the equation:

$$y^2 = x^3 - N^2x$$

We aim to show a connection between this elliptic curve and a system of two equations of quadratic forms:

$$\begin{aligned} x^2 - ay^2 &= z^2 \\ x^2 + ay^2 &= t^2 \end{aligned}$$

Let us introduce a new variable u such that $x = u^2$. The elliptic curve equation becomes:

$$y^2 = u^6 - N^2u^2$$

Now, let us express y^2 as a difference of two squares (**??**), by adding and subtracting the same term:

$$y^2 + (\frac{N^2}{2})^2 - (\frac{N^2}{2})^2 = u^6 - N^2u^2$$

This can be rearranged to form:

$$(y + \frac{N^2}{2})(y - \frac{N^2}{2}) = u^2(u^4 - N^2)$$

Let us define two new variables z and t as follows:

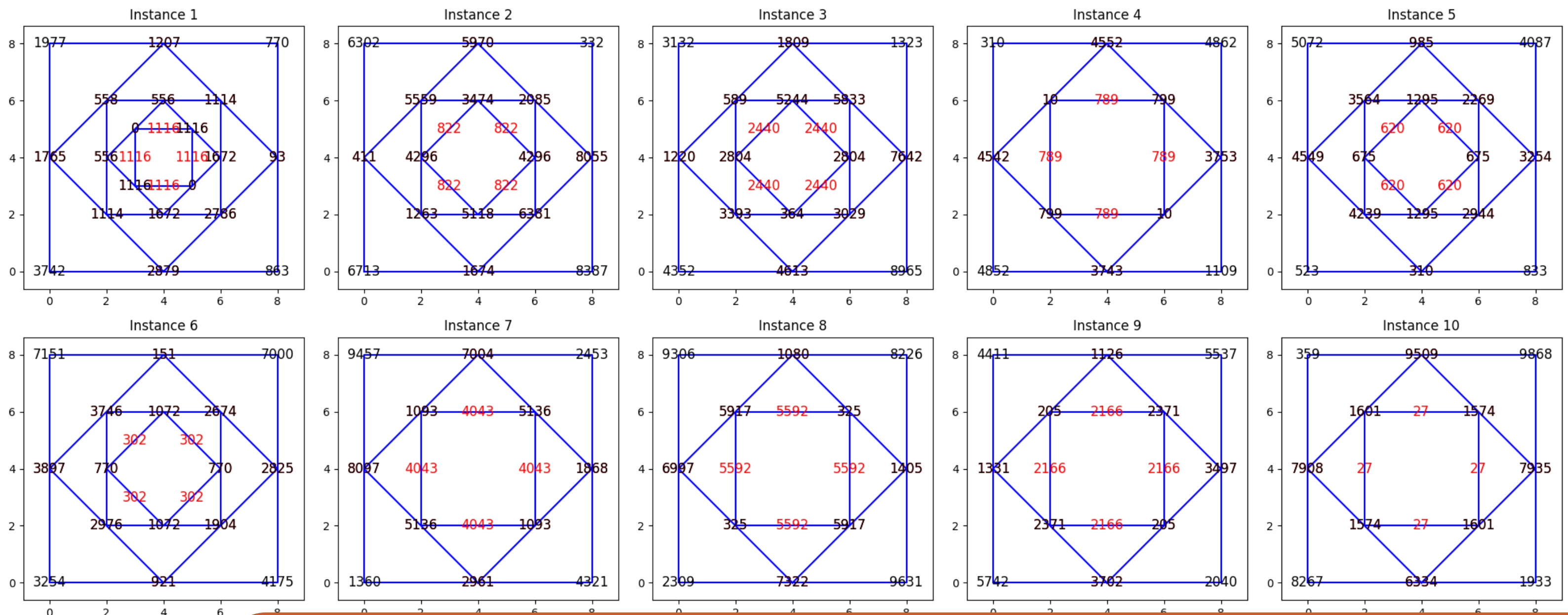
$$\begin{aligned} z &= y + \frac{N^2}{2} \\ t &= y - \frac{N^2}{2} \end{aligned}$$

We then obtain two equations:

$$\begin{aligned} z - t &= N^2 \\ zt &= u^2(u^4 - N^2) \end{aligned}$$

The second equation can be written as:

$$u^2(u^2 - N)(u^2 + N) = zt$$



20. Combinatorial Differential geoemetry

An Epi-genetic prologue

To be inappropriately concrete but as a reflection of the spirit of my own appreciation of the subject consider that, in differential geometry, the Riemann curvature tensor $R^a{}_{b\mu\nu}$ and the Ricci tensor play crucial roles in describing the curvature of spacetime.

Combinatorial Analysis of Tensor Components in 4D Spacetime Analysis[1]

The Riemann tensor is endowed with a set of symmetries and antisymmetries that can be elegantly described using the language of differential forms as $\Omega_{ab} = \frac{1}{4!}R_{ab\mu\nu}dx^\mu \wedge dx^\nu$. The tensor exhibits several key symmetries:

- **Antisymmetry in the last two indices:** $R^a{}_{b\mu\nu} = -R^a{}_{b\nu\mu}$.
- **Symmetry in exchanging the pair of indices:** $R_{ab\mu\nu} = R_{\mu\nu ab}$.
- **Cyclic identity:** $R^a{}_{b\mu\nu} + R^a{}_{\nu b\mu} + R^a{}_{\mu\nu b} = 0$.
- **Bianchi identity:** $\nabla_\lambda R^a{}_{b\mu\nu} + \nabla_\mu R^a{}_{b\nu\lambda} + \nabla_\nu R^a{}_{b\lambda\mu} = 0$.

The inherent asymmetry introduced by the differential form $dx^\mu \wedge dx^\nu$ and the relation of the *ab* indices to the Lorentz group $SO(3,1)$ play a significant role in the tensor's properties. A combinatorial approach reveals the number of independent components and the constraints imposed by their symmetries. The Riemann tensor $R^a{}_{b\mu\nu}$ in four-dimensional spacetime though the interplay of symmetries reduces the naive count of $4^4 = 256$ possible components to 20 independent components:

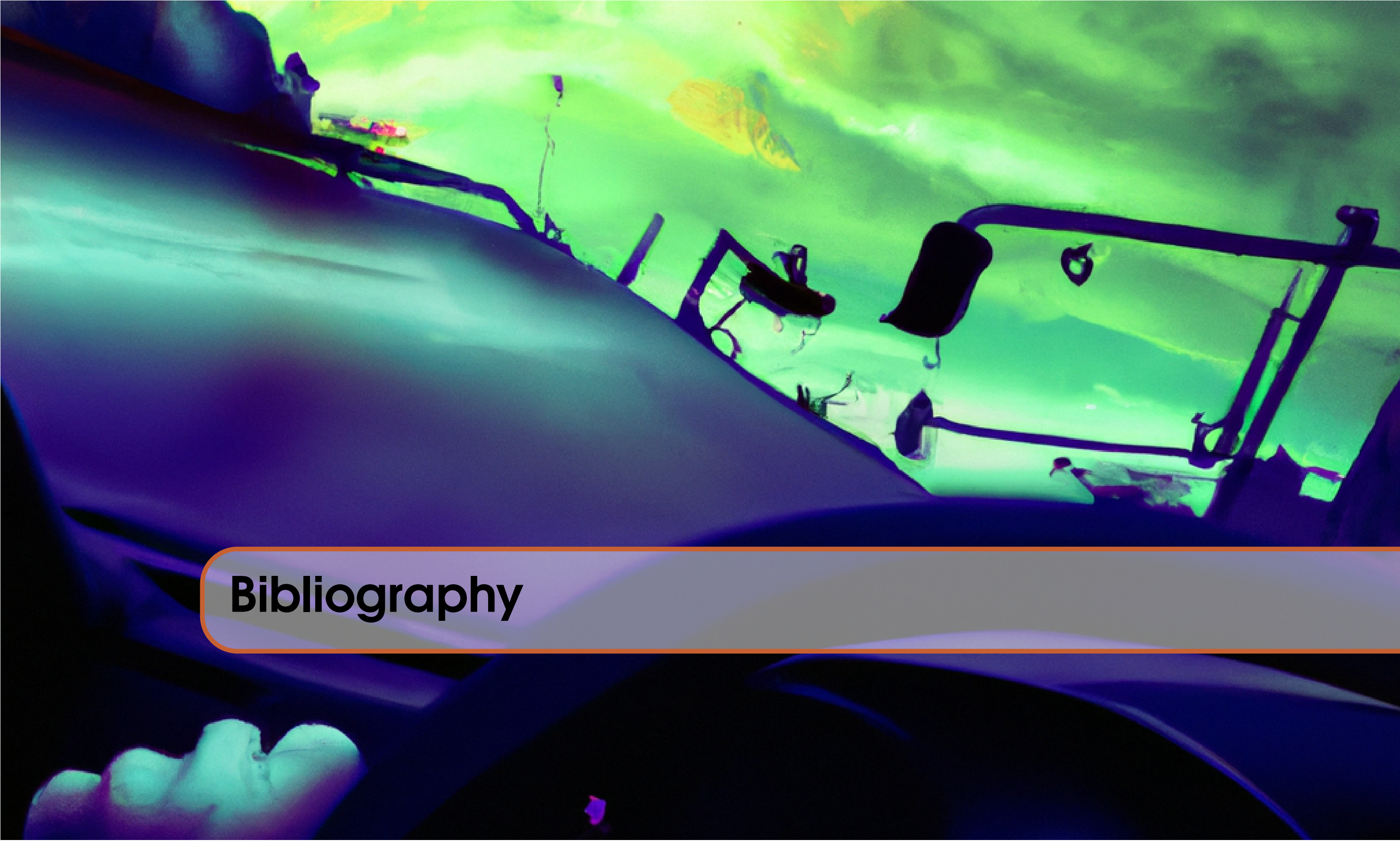
- Antisymmetry in the last two indices ($R_{abcd} = -R_{abdc}$).
- Symmetry between the first and last pair of indices ($R_{abcd} = R_{cdab}$).
- The cyclic identity and Bianchi identity further constrain the components.

The Ricci tensor R_{ab}, derived from the Riemann tensor, exhibits symmetry ($R_{ab} = R_{ba}$). In four-dimensional spacetime, the number of its independent components is:

$$\text{Independent components} = 4^C_2 + 4 = 10$$

This reflects the 10 independent ways to choose two indices from four, considering symmetry, where $4C2$ counts the distinct pairs and the additional 4 accounts for the diagonal components $(a = b)$.

[1] The detailed exposition includes the mathematical derivation of the independent components of the Riemann and Ricci tensors, leveraging their symmetries and antisymmetries. For the Riemann tensor, the combination of antisymmetry in the last two indices, symmetry between index pairs, and the cyclic and Bianchi identities reduces the naive count of $4^4 = 256$ potential components to 20 independent ones. For the Ricci tensor, its symmetric nature in a four-dimensional spacetime results in 10 independent components, calculated as $4C2 + 4$.



Bibliography

Books

[Ash16] Anver Ash. *Summing it up*. Princeton University Press, 2016 (cited on page 187).

[Ben99] Donald C Benson. *The Moment of Proof*. Oxford University Press, 1999 (cited on page 252).

[Blu97] W Blumel R; Reinhardt. *Chaos in Atomic Physics*. Cambridge University Press, 1997 (cited on page 148).

[Bol87] Brian Bolt. *Even More Mathematical Activities*. Cambridge University Press, 1987 (cited on page 154).

[Hav12] Julian Havil. *The Irrationals*. Princeton University Press, 2012 (cited on page 29).

[Haw20] Joel David Hawkins. *Proof and the Art of Mathematics*. The MIT Press, 2020 (cited on page 142).

[Hig11] Peter M Higgins. *Numbers: A Very Short Introduction*. Oxford University Press, 2011 (cited on page 111).

[Kap11] Ellen Robert Kaplan. *Hidden Harmonies*. Bloomsbury Publishing USA, 2011 (cited on page 325).

[Lin86] Malcolm E Lines. *A Number for your thoughts*. Adam Hilger Ltd, 1986 (cited on page 109).

[Mot85] Lorraine Mottershead. *Investigations in Mathematics*. Basil Blackwell Limited, 1985 (cited on page 98).

[Nie74] Jurg et al. Nievergelt. *Compuer Approaches to Mathematical Problems*. Prientice-Hall Inc, 1974 (cited on page 74).

[Pau80] John Allen Paulos. *Innumeracy*. Penguin Books, 1980 (cited on page 12).

[Pir74] Robert M Pirsig. *Zen and the art of motorcycle maintenance: An inquiry into values*. William Morrow and Company, 1974 (cited on pages 17, 39).

[Poy57] George Poyla. *How to Solve it*. Princeton University Press, 1957 (cited on page 249).

[Pri22] Chris Prichard. *Experiencing Mathematics through Investigations*. The Mathematical Association, 2022 (cited on pages 155, 307).

[Rei65] W. J. Reichmann. *The Fascination of Numbers*. Methuen Ci Ltd, 1965 (cited on page 111).

[Sar63] Jean-Paul Sartre. *Nausea*. Penguin Modern Classics, 1963 (cited on page 189).

[Sne75] L. S. Snell. *Introduction to Probability Theory with Computing*. prentice Hall Inc, 1975 (cited on page 290).

[Ste89] Ian Stewart. *Galois Theory*. Chapman and Hall, 1989 (cited on page 12).

[Wei99] E. W. Weisstein. *Concise Encyclopedia of Mathematics*. Chapman Hall/CRCnetBASE, 1999 (cited on page 52).

Articles

[] In: () (cited on page 268).

[Smi90] James Smith. “The College of Mathematics Journal”. In: 21.3 (May 1990), pages 196–207 (cited on page 291).

Index