Probabilistic Estimation of Prime Number Density

Nick Rui

March 2025

Introduction

Prime numbers are mysterious and nonchalant. One probabilistic question to be asked about prime numbers is, given a random natural number $x \in [1, n]$, what is the probability that x is prime, in terms of n?

From the Prime Number Theorem, we now know that probability is approximately $1/\ln(n)$. More precisely, the number of prime numbers in the range of $[1, n]$, denoted $\pi(n)$ is roughly

$$\pi(n) \approx \frac{n}{\ln(n)}$$

Thus, the probability of picking a random number from $[1, n]$ and it being prime is $1/\ln(n)$. Said differently, the proportion of prime numbers from $[1, n]$ is roughly $1/\ln(n)$.

Gauss was the first to think of this idea. More impressively, he put forward this conjecture (which would eventually lead to the Prime Number Theorem) as a *teenager*. Since I am still 18, I thought tackling this problem was appropriate.

Gauss had a lot of aura because this guy literally brute-forced his way to this conclusion. He would straight-up manually verify — by hand — whether or not numbers on the scale of millions if they were prime or not. By counting how many primes he found, he guessed that $\pi(n)$ had to grow something like the function $n/\ln(n)$.

Unfortunately, I do not have the patience and precision to manually work out prime numbers by hand. Luckily, I have the power of computation...and probability.

In this project, I verify that Gauss' conjecture (and the Prime Number Theorem) is correct through two probabilistic applications:

(1) A primality test using sampling and Bayes' Theorem.

(2) Random (Monte Carlo) sampling to estimate the proportion of primes between $[1, n]$.

1 Bayesian Primality Test

The main idea here is Fermat's little theorem, which says that if p is a prime number, then for all nonzero integers $a < p$ we have

$$a^{p-1} \equiv 1 \mod p.$$

Say we have some candidate number x which we want to determine whether or not it is prime. The naive brute-force approach is to check whether any integer $2 \leq a \leq \lceil \sqrt{n} \rceil$ divides x. If not, then x is prime. However, this requires checking lots of numbers. On the other hand, if we randomly pick some a and $a^{x-1} \not\equiv 1 \mod x$, then we immediately know that x is not prime.

However, it could be by chance that $a^{k-1} \equiv 1 \mod k$ for composite k. In fact, there do exist some rare numbers (called Carmichael numbers) k that satisfy Fermat's little theorem for all a coprime to k. These Carmichael numbers essentially "behave" like primes, but aren't actually prime (a good example is 561). Thus, if our number k satisfies Fermat's little theorem for one a, we cannot claim it is prime, but intuitively we know it is more likely to be prime than we previously had thought. This is where Bayes' theorem and inference can be applied.

Instead of the brute-force approach to test whether or not a number is prime — which takes a lot of work but always gives a concrete, correct answer — I took a randomized approach, which takes much less work but only returns a probability belief that a number is prime. Using randomized sampling to yield a numeric result (a *Monte Carlo* algorithm) is justified here because our end goal is to look at the proportion of primes between $[1, n]$ for some large n, and so occasionally over-counting a prime is (as we will see later on) insignificant for large n.

(In Math 62DM, we learned a rapid (polynomial time) deterministic algorithm to do this (the AKS primality test). Though that algorithm is objectively better, whats the fun without a little uncertainty?)

Let p represent our belief that $x \in [1, n]$ is prime. The algorithm does the following:

- Assume a prior belief of $p = 0.5$ (because we don't know any better).
- Randomly pick an integer "witness" $2 \leq a_1 < x$.
- Check if a_1 divides x. If so, then k must be composite, and so $p = 0$.
- Check if $a_1^{x-1} \equiv 1 \mod x$. (This may seem computationally heavy, but with a repeated-squaring trick with modular arithmetic we can rapidly verify this.) If so, then we update our belief p to reflect that. (If not, then again $p = 0$).
- Repeat for many iterations. Randomly pick more integers $2 \leq a_i < x$ and perform the same Fermat tests with these "witnesses" a_i.

Let $\mathcal{P}$ represent the event that x is prime, and let F be the event that our number x satisfies Fermat's little theorem for one randomly sampled a_i. What is $P(\mathcal{P}|F)$? With Bayes' theorem, we have

$$P(\mathcal{P}|F) = \frac{P(F|\mathcal{P})P(\mathcal{P})}{P(F)}$$

In context, we have

- $P(\mathcal{P}) = p$, the prior belief that x is prime.
- $P(F|\mathcal{P}) = 1$, since if k is prime then it must satisfy Fermat's little theorem.
- $P(F) = P(F|\mathcal{P})P(\mathcal{P}) + P(F|\mathcal{P}^C)P(\mathcal{P}^C) = p + (1-p)P(F|\mathcal{P}^C)$ by the law of total probability. We have one unknown here, namely the probability that a non-prime number will end up satisfying Fermat's little theorem for one randomly sampled a_i. One could suppose that a_i^{x-1} could be any equivalence class mod x $(0, 1, \ldots, x-1 \mod x)$, and so a broad assumption we will make is that in the *worst case* this probability is 1/2 (this is a very broad and bold assumption).

Thus, our updated posterior belief is

$$\begin{aligned} P(\mathcal{P}|F) &= \frac{P(F|\mathcal{P})P(\mathcal{P})}{P(F)} \\ &= \frac{p}{p + (1-p)\frac{1}{2}}. \end{aligned}$$

(See Figure 1 and Figure 2.)

The equation we derived confirms our intuition: as we observe successful Fermat tests, our confidence in the number's primality increases. Starting from a neutral prior of $p = 0.5$, a single passing test raises our confidence to approximately 0.667. With repeated successful tests, this confidence grows even further, as each new piece of evidence compounds upon the previous updates.

For example, if we update our prior to 0.667 after one test, and then perform a second test which also passes, our posterior probability becomes approximately 0.8. A third successful test would increase this to around 0.89. This demonstrates how quickly our certainty grows with multiple passes.

However, we must remember that this approach remains probabilistic — it cannot provide absolute certainty like a deterministic primality test would. For example, the presence of Carmichael numbers, which pass all Fermat tests despite being composite, creates an inherent limitation to this method.

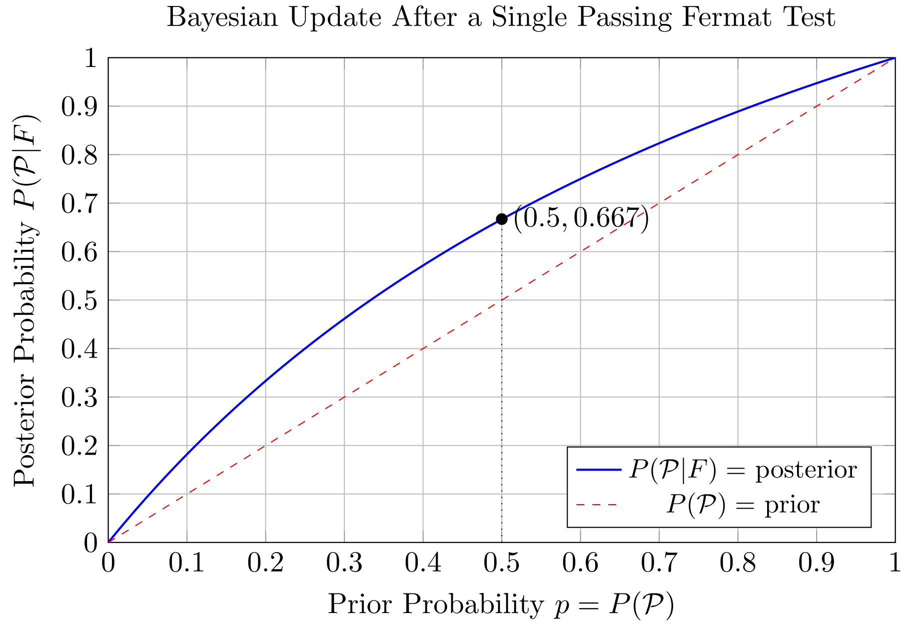

Figure 1: The curve shows how a prior probability p is updated to posterior probability $P(\mathcal{P}|F)$ after observing a successful Fermat test. For example, a prior of $p = 0.5$ is updated to $P(\mathcal{P}|F) \approx 0.667$. The dashed line represents no change in probability.

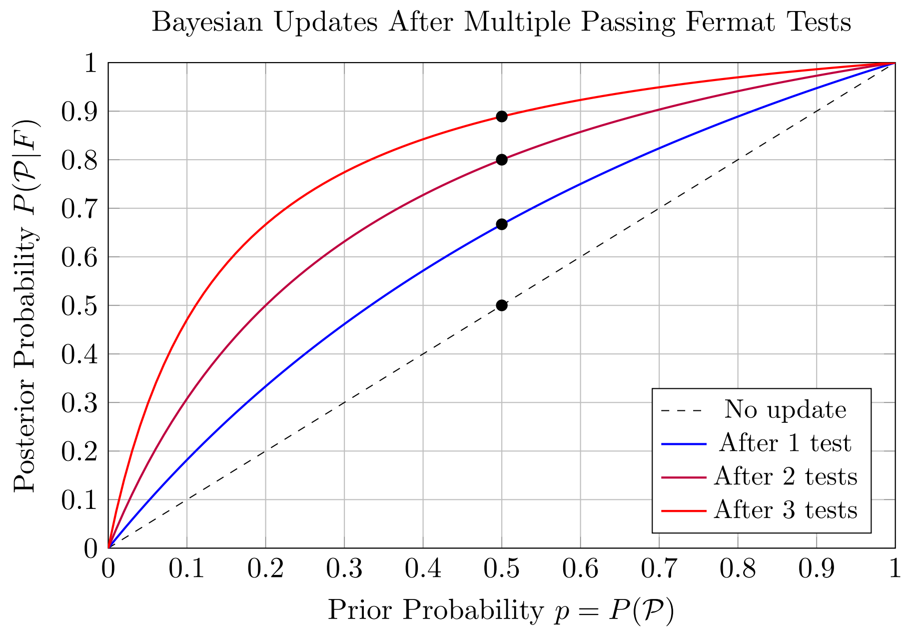

Figure 2: This graph shows how the posterior probability evolves with multiple sequential Fermat tests, starting from different prior values. Each curve represents the update after an additional test passes. Notice how a prior of $p = 0.5$ is updated to approximately 0.67, 0.8, and 0.89 after one, two, and three passing tests respectively.

2 Verification of the Prime Number Theorem

With our probabilistic primality test established, we can now use it to estimate the proportion of prime numbers in different ranges, testing Gauss's conjecture.

The approach is as followed straightforward:

- For a range of values of $n = e^j$ for $j = 2, 3, \ldots, 20$.
- Randomly sample numbers between 1 and n. These are our "prime candidates".
- For each sampled number, use our Bayesian primality test to determine if it's likely prime.
- Calculate the proportion of probable primes among our samples
- Compare this proportion to the theoretical $1/\ln(n) = 1/j$ predicted by the Prime Number Theorem

This Monte Carlo approach gives us an empirical verification of the Prime Number Theorem without requiring us to identify all primes in a range. Instead, we're using probability theory twice: once in our primality test, and again in our sampling approach.

3 Results

The actual Python implementation of this can be found in a Jupyter notebook. There, you will find the necessary functions for the Bayesian primality test and prime number theorem verification, along with the example data that I show below.

When running this experiment for values of j from 2 to 20 (corresponding to numbers up to approximately $e^{20} \approx 485$ million), I observed a remarkable alignment between the sampled proportions and the theoretical predictions from the Prime Number Theorem.

As j increases and we examine larger ranges, the proportions continue to follow the $1/\ln(n)$ curve predicted by the Prime Number Theorem. This provides compelling empirical evidence for Gauss's original conjecture.

Table 1: Comparison of Estimated vs. Theoretical Prime Density (Selected Values)

j	$n \approx e^j$	Theoretical $1/\ln(n)$	Monte Carlo Estimation
5	1.48×10^2	0.2000	0.2433
10	2.20×10^4	0.1000	0.1103
15	3.27×10^6	0.0667	0.0725
20	4.85×10^8	0.0500	0.0507
25	7.20×10^{10}	0.0400	0.0410
30	1.07×10^{13}	0.0333	0.0364
35	1.59×10^{15}	0.0286	0.0280

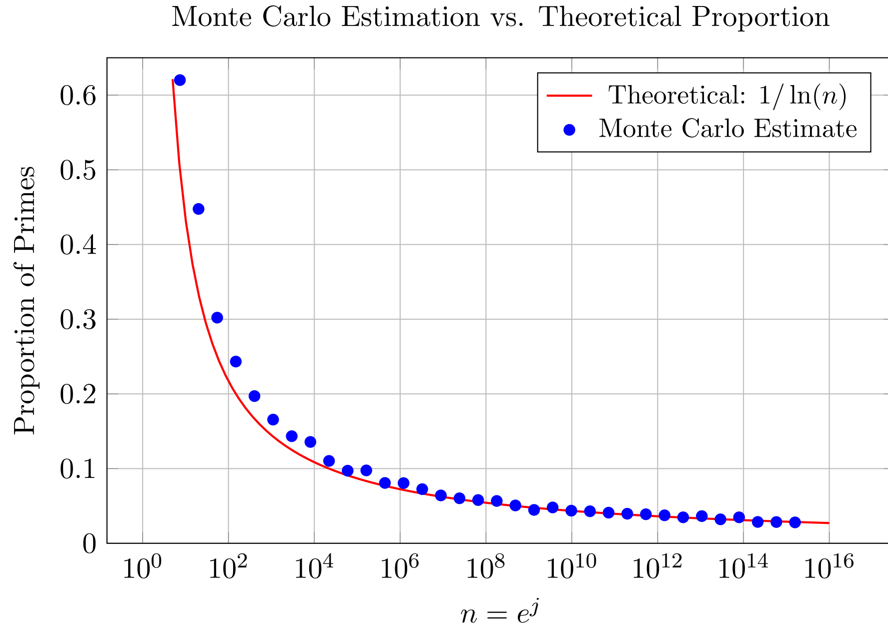

Figure 3: Comparison between Monte Carlo estimates of prime density and the theoretical prediction from the Prime Number Theorem. The alignment demonstrates the validity of both our probabilistic approach and Gauss's original conjecture.

3.1 Error Analysis

Our approach introduces some sources of error, including

(i) False positive detection of primes (the presence of Carmichael numbers).

(ii) Statistical sampling error from randomly selecting numbers

(iii) Probabilistic error from our Bayesian primality test

With enough samples, errors (ii) and (iii) become negligible. What is obvious, however, is the existence of Carmichael numbers causing false positives in our Bayesian primality test. This is obvious when we realize that our estimates seem to always be a bit higher than the actual proportion of primes. The gap between our estimate and the actual curve can be thought of as the density of Carmichael numbers.

However, from the data it seems that the density of Carmichael numbers gets smaller and smaller as n increases. Thus, our algorithm gives a good enough estimate for large values of n.

4 Conclusion

Through a probabilistic approach, I've demonstrated an alternative perspective to verifying the Prime Number Theorem using Bayesian inference and Monte Carlo sampling. Unlike Gauss's manual counting, this method leverages computational power and probability theory to achieve the same insight with less effort.

The Bayesian primality test offers an interesting trade-off between computational efficiency and certainty, allowing us to quickly classify numbers as probably prime or definitely composite.

A probabilistic primality test could have potential application in fields like cryptography, where verifying if a number is prime or not is a problem that computers have to solve. However, we must be wary and ethical of the way such algorithms are used, since if we develop a primality test that could determine prime numbers in a very fast amount of time (computer scientists fear that quantum computers could do this — if implemented correctly), then much of the current cybersecurity space could be threatened.

Future work might include refining the estimate of $P(F|\mathcal{P}^C)$ beyond our simplified 1/2 assumption, incorporating additional primality criteria to better identify Carmichael numbers, or using more sophisticated sampling strategies to reduce the variance in our proportion estimates.

5 Acknowledgments

Generative AI was used to

(i) Help write scripts to display data into figures and graph with Matplotlib.

(ii) Help create tables and graphs in LaTeX.

This project was initially created for the CS 109 challenge. I drew inspiration from my experiences learning about prime numbers in Math 62DM (abstract algebra and number theory) and randomized algorithms in CS 161 (algorithms).

6 Appendix (Python Script)

```
# -*- coding: utf-8 -*-
"""109 project.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1WoUsP7_p2zBmc9QRB3VE2CokQTrEtpAn

# Scripts

## Imports
"""

import random
import math
import numpy as np
import matplotlib.pyplot as plt
import time
from tqdm.notebook import tqdm
import pandas as pd
from IPython.display import display, Markdown

random.seed(42)
np.random.seed(42)

"""## Bayesian Primality Test"""

def gcd(a, b):
    """
    Compute the greatest common divisor of a and b using Euclidean algorithm.
    """
    while b:
        a, b = b, a % b
    return a

def power_mod(base, exponent, modulus):
    """
    Compute (base^exponent) % modulus efficiently using repeated squaring algorithm.
    This is much faster than the naive approach for large exponents.
    """
    if modulus == 1:
        return 0

    result = 1
    base = base % modulus
```

```
    while exponent > 0:
        # If exponent is odd, multiply result with base
        if exponent % 2 == 1:
            result = (result * base) % modulus

        # Divide the exponent by 2
        exponent = exponent >> 1  # Same as exponent // 2

        # Square the base
        base = (base * base) % modulus

    return result

def fermat_test(n, a):
    """
    Perform Fermat's primality test for a single witness 'a'.
    Returns:
    - True if n passes the test (might be prime)
    - False if n is definitely composite
    """
    # Edge cases
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Check if a divides n
    if gcd(a, n) > 1:
        return False

    # Check Fermat's Little Theorem: a^(n-1)  1 (mod n)
    if power_mod(a, n-1, n) != 1:
        return False

    return True

def bayesian_primality_test(n, k=10, prior_probability=0.5):
    """
    Perform Fermat's primality test with k random witnesses,
    updating probability using Bayes' theorem.

    Parameters:
    - n: number to test for primality
    - k: number of random witnesses to test
    - prior_probability: prior belief that n is prime
```

```
    Returns:
    - final_probability: posterior probability that n is prime
    """
    # Handle edge cases
    if n <= 1:
        return 0.0

    if n <= 3:
        return 1.0

    if n % 2 == 0:
        return 0.0

    # Initialize current probability
    current_probability = prior_probability

    # Estimate probability of a composite number passing Fermat's test (using 1/2 as in the pap
    p_pass_if_composite = 0.5

    for i in range(k):
        # Choose a random witness between 2 and n-2
        a = random.randint(2, n-2)

        # Perform the test
        test_passed = fermat_test(n, a)

        if test_passed:
            # Update using Bayes' theorem
            # P(prime|pass) = P(pass|prime) * P(prime) / P(pass)
            p_pass_given_prime = 1.0
            p_prime = current_probability
            p_composite = 1.0 - p_prime

            # P(pass) using law of total probability
            p_pass = (p_pass_given_prime * p_prime) + (p_pass_if_composite * p_composite)

            # Apply Bayes' theorem
            current_probability = (p_pass_given_prime * p_prime) / p_pass
        else:
            # If test fails, the number is definitely composite
            current_probability = 0.0
            break

    return current_probability

"""## Monte Carlo Verification of Prime Number Theorem"""

def monte_carlo_prime_theorem(j_values=range(2, 21), num_samples=10000, k=10, threshold=0.95):
```

```
"""
Verify the Prime Number Theorem using Monte Carlo sampling for values x = e^j.

Parameters:
- j_values: range of j values for testing (x = e^j)
- num_samples: number of random samples to test for each j
- k: number of witnesses for the Bayesian primality test
- threshold: probability threshold for declaring a number prime

Returns:
- results: list of dictionaries with experiment results
"""
results = []

for j in tqdm(j_values, desc="Testing j values from 2 to 20"):
    x = math.exp(j)
    num_primes = 0

    print(f"\nTesting j = {j}, x = e^{j}  {x:.2f}")
    start_time = time.time()

    for i in range(num_samples):
        # Generate a random number from 2 to x (avoid 0, 1)
        sample = max(2, int(random.uniform(2, x)))

        # Use the Bayesian primality test with confidence threshold
        prob = bayesian_primality_test(sample, k=k)
        if prob >= threshold:
            num_primes += 1

    # Calculate proportions
    elapsed_time = time.time() - start_time
    sampled_proportion = num_primes / num_samples
    theoretical_proportion = 1 / math.log(x)

    # Calculate ratio and error
    ratio = sampled_proportion / theoretical_proportion
    error = abs(sampled_proportion - theoretical_proportion) / theoretical_proportion * 100

    result = {
        ’j’: j,
        ’x’: x,
        ’sampled_proportion’: sampled_proportion,
        ’theoretical_proportion’: theoretical_proportion,
        ’ratio’: ratio,
        ’error’: error,
        ’time’: elapsed_time
    }
```

```
        results.append(result)

        print(f"  Samples tested: {num_samples}")
        print(f"  Primes found: {num_primes}")
        print(f"  Sampled proportion: {sampled_proportion:.6f}")
        print(f"  Theoretical (1/ln(x)): {theoretical_proportion:.6f}")
        print(f"  Ratio: {ratio:.4f}")
        print(f"  Error: {error:.2f}%")
        print(f"  Time elapsed: {elapsed_time:.2f} seconds")

    return results

"""## Visualization"""

def plot_monte_carlo_results(results):
    """
    Plot the Monte Carlo results compared to theoretical predictions.

    Parameters:
    - results: List of dictionaries with experiment results

    Returns:
    - plt: Matplotlib figure object
    """
    plt.figure(figsize=(12, 8))

    # Extract data
    j_values = [result['j'] for result in results]
    x_values = [result['x'] for result in results]
    sampled_props = [result['sampled_proportion'] for result in results]
    theoretical_props = [result['theoretical_proportion'] for result in results]

    # Create comparison plot with log scale for x-axis
    plt.semilogx(x_values, theoretical_props, 'r-', linewidth=2, label='Theoretical: 1/ln(x)')
    plt.semilogx(x_values, sampled_props, 'bo', markersize=6, label='Monte Carlo Estimate')

    # Format plot
    plt.grid(True, which="both", ls="-")
    plt.xlabel('x = e^j')
    plt.ylabel('Proportion of Primes')
    plt.title('Monte Carlo Estimation vs. Theoretical Proportion')
    plt.legend()

    # Set y-axis limits with some margin
    plt.ylim(0, max(max(sampled_props), max(theoretical_props)) * 1.2)

    return plt
```

```
"""# Testing"""

def main():
    """Main execution function to verify the Prime Number Theorem."""
    print("Verifying the Prime Number Theorem using Monte Carlo sampling")
    print("============================================================")
    print("This will estimate the proportion of primes for x = e^j where j ranges from 2 to 20
    print("For each range, we'll sample random numbers and test primality using the Bayesian a
    print("Then we'll compare the results to the theoretical prediction from the Prime Number 
    print("\nParameters:")
    print("- j values: 2 to 20")
    print("- Samples per range: 10,000")
    print("- Fermat test witnesses: 10")
    print("- Primality threshold: 0.95")
    print("\nStarting verification...\n")

    # Run Monte Carlo verification for j from 2 to 20
    results = monte_carlo_prime_theorem(
        j_values=range(2, 36),
        num_samples=10000,
        k=10,
        threshold=0.95
    )

    # Display results as a table
    print("\nSummary of Results:")
    df_results = pd.DataFrame(results)
    display(df_results[['j', 'x', 'theoretical_proportion', 'sampled_proportion', 'error', 'ti

    # Create comparison plots
    print("\nGenerating visualization of results...")
    plot_monte_carlo_results(results).show()

    # Save results to CSV
    df_results.to_csv('prime_number_theorem_verification.csv', index=False)
    print("Results saved to prime_number_theorem_verification.csv")

    print("\nVerification completed!")

main()
```