Pittsburgh Interdisciplinary Mathematics Review **3** (2025), 82–101

Formal verification of the Euler Sieve via Lean

Isaac (Rucheng) Li

(Communicated by Leonardo Finzi and Raja Krishnaswamy)

Abstract

This paper presents a formal verification of the Euler Sieve algorithm — a linear variant of the classical Sieve of Eratosthenes — using the Lean proof assistant. We begin by discussing the traditional Sieve of Eratosthenes and its inherent redundancy when crossing out composite numbers. We then introduce the Euler Sieve, which overcomes this drawback by ensuring that each composite number is "marked" only once, achieving linear time complexity. Finally, we present a Lean formalization that rigorously verifies the correctness of the Euler Sieve, including definitions, lemmas, and the overall rigorous recursive structure. To the best of our knowledge, this work is the first formal proof of the Euler Sieve by an interactive proof assistant.

Contents

Introduction

A natural number greater than 1 is *composite* if it can be factored as the product of two numbers greater than 1. Otherwise, a number is *prime* if it cannot be so factored. The identification and enumeration of prime numbers constitute a foundational problem in number theory, carrying substantial implications for mathematics and computer science. A common method for identifying prime numbers is known as a *sieve*, an algorithm that begins with the list of numbers from 2 to n and systematically crosses out all composite numbers, leaving only the primes. Among these prime sieves, the Sieve of Eratosthenes, dating back to ancient Greece, is particularly notable for its conceptual simplicity and historical significance. However, this classical algorithm suffers from inherent redundancy, resulting in non-linear time complexity.

In the 20th century, as computers emerged, there was a surge of interest in prime sieving methods. Many scientists tried to address the inefficiency in the Sieve of Eratosthenes, which led to the creation of numerous prime sieves with linear time complexity. However, in 1978, Gries and Misra rediscovered and observed a linear prime sieve which had already been devised in the 18th century — namely, the Euler Sieve. Remarkably, Leonhard Euler formulated this algorithm long before computers existed, and it is widely recognized as the very first linear sieve for generating prime numbers.

In this paper, we present a rigorous formal verification of the Euler Sieve algorithm using the *Lean* interactive proof assistant. Formal proofs involve encoding mathematical statements and their proofs in a formal language that a computer system can rigorously check for logical correctness. *Lean* is a modern, interactive theorem prover that allows mathematicians and computer scientists to build reliable, formally verified proofs with absolutely no errors. Formal verification is particularly significant in algorithmic number theory, as it guarantees the correctness of computational methods.

Our formalization explicitly verifies the correctness of the Euler Sieve by establishing that its output — a complete list of primes and an associated function identifying the smallest prime factor of each integer — is accurate.

1. **Background on the Sieve of Eratosthenes**

The idea of systematically "sieving" out composite numbers from a list of natural numbers can be traced back to ancient Greece. The Sieve of Eratosthenes, attributed to the Greek mathematician Eratosthenes of Cyrene (circa 276–194 BC), is one of the oldest known algorithms for finding all prime numbers up to a given limit n. In his work, Eratosthenes proposed listing all integers from 2 to n and then sequentially crossing out the multiples of each prime, beginning with 2.

1.1. **The naïve version**

To find all prime numbers not exceeding a positive integer n, we begin with the list

$$S = \{2, 3, 4, \ldots, n\}.$$

The method proceeds as follows:

1. For each number p in S, starting from 2, if p has not been crossed out from the set, then p is declared prime.
2. Once p is identified as prime, every multiple of p that lies in S (that is, $2p, 3p \ldots$) is eliminated from S because they must be composite.
3. This procedure is repeated for the next number in the list that has not yet been crossed out.

After all numbers in S have been processed in this manner, the remaining elements of S are exactly the prime numbers up to n.

A very primitive version of the sieve can be described by the following pseudocode:

```
function NaiveSieve(n):
    A[2..n] = true
    for i from 2 to n:
        if A[i] is true:
            for j from 2*i to n with step i:
                A[j] := false
    return all i such that A[i] is true
```

1.2. **The optimized version**

An important observation is that if n is composite, then it must have a factor less than or equal to $\sqrt{n}$. Therefore, it suffices to perform the sieving process only for i up to $\lfloor\sqrt{n}\rfloor$, where $\lfloor\sqrt{n}\rfloor$ is the floor of $\sqrt{n}$ — the greatest integer less than or equal to n.

A further optimization is possible by noting that for a prime i, any multiple $k \times i$ with $k < i$ would have been crossed out already when processing the prime factor of k. Hence, we can safely begin the inner loop at i^2.

This gives the final optimized pseudocode:

```
function EratosthenesSieve(n):
    A[2..n] = true
    for i from 2 to floor(sqrt(n)):
        if A[i] is true:
            for j from i*i to n with step i:
                A[j] := false
    return all i such that A[i] is true
```

But still, it maintains the problem of crossing out a composite number several times. This is crucial and causes the non-linear running time of the algorithm.

1.3. **Complexity analysis**

Before we delve into the technical details, we briefly explain what we mean by complexity analysis. In computer science, time complexity analysis measures the efficiency of an algorithm in terms of the number of operations it performs with respect to the input size n. For the purpose of this paper, we use the following definition and notations:

- We say that a sieve algorithm has run time $O(f(n))$ if there exist constants $C > 0$ and $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, the numbers between 2 and n are crossed out at most $C\,f(n)$ times. Hence, $O(f(n))$ gives an asymptotic upper bound on the running time.

- We say that a sieve algorithm has run time $\Omega(f(n))$ if there exist constants $C > 0$ and $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, the numbers between 2 and n are crossed out at least $C\,f(n)$ times. Thus, $\Omega(f(n))$ gives an asymptotic lower bound on the running time.

- If a sieve algorithm has both an asymptotic upper bound $O(f(n))$ and an asymptotic lower bound $\Omega(f(n))$, we say that it has run time $\Theta(f(n))$.

Specifically, if an algorithm crosses out each number once, its running time is $\Theta(n)$.

Theorem. *The Optimized Sieve of Eratosthenes has run time* $\Theta(n \log\log n)$.

Proof. For each prime p with $p \leq \sqrt{n}$, the inner loop (which crosses out multiples of p between p^2 and n) executes exactly

$$L(p) = \left\lfloor \frac{n-p^2}{p} \right\rfloor + 1$$

iterations. Hence, we obtain the following bounds for the number of crossing-out operations for a fixed prime p:

$$\frac{n}{p} - p \leq L(p) \leq \frac{n}{p} - p + 1.$$

Summing over all primes p satisfying $p \leq \sqrt{n}$, the total number of crossing-out operations $T(n)$ satisfies

$$\sum_{p \leq \sqrt{n}} \Big(\frac{n}{p} - p\Big) \leq T(n) \leq \sum_{p \leq \sqrt{n}} \Big(\frac{n}{p} - p + 1\Big).$$

We can rewrite the main term as

$$\sum_{p \leq \sqrt{n}} \frac{n}{p} = n \sum_{p \leq \sqrt{n}} \frac{1}{p}.$$

A classical result in analytic number theory, known as Mertens' second theorem, states that

$$\lim_{x\to\infty} \Big(\sum_{p \leq x} \frac{1}{p} - \log\log x\Big) = M,$$

or equivalently,

$$\sum_{p \leq x} \frac{1}{p} \;=\; \log\log x \;+\; M \;+\; O(1),$$

where M denotes the Meissel–Mertens constant (approximately 0.261497). Setting $x = \sqrt{n}$, we obtain

$$\sum_{p \le \sqrt{n}} \frac{1}{p} = \log\log\sqrt{n} + M + O(1) = \log\log n - \log 2 + M + O(1).$$

Therefore, the upper bound for $T(n)$ becomes

$$T(n) \le n\Big(\log\log n - \log 2 + M + O(1)\Big) - \sum_{p \le \sqrt{n}} (p-1).$$

Thus,

$$T(n) = O(n \log\log n).$$

Similarly, the lower bound for $T(n)$ is

$$T(n) \ge n\Big(\log\log n - \log 2 + M + O(1)\Big) - \sum_{p \le \sqrt{n}} p.$$

For the sum $\sum_{p \le \sqrt{n}} p$, observe that

$$\sum_{p \le \sqrt{n}} p \le \sum_{i=1}^{\lfloor \sqrt{n} \rfloor} i \le \frac{\sqrt{n}(\sqrt{n}+1)}{2} = \frac{n+\sqrt{n}}{2},$$

which does not alter the overall complexity. Thus,

$$T(n) = \Omega(n \log\log n).$$

Therefore, we deduce that

$$T(n) = \Theta(n \log\log n). \qquad \square$$

This theorem implies that even an optimized version of the sieve algorithm is asymptotically larger than a linear algorithm (an algorithm with run time $O(n)$).

2. **Improvement in the Euler Sieve**

In later developments, efforts were made to address the redundancy inherent in the Sieve of Eratosthenes. In the 18th century, Leonhard Euler introduced a new prime sieve designed to avoid repeatedly crossing out the same composite numbers, achieving a linear time complexity.

The key idea is to ensure that each composite number is "marked" exactly once, using only its smallest prime factor. This refined method is now commonly known as the Euler Sieve. Moreover, the Euler Sieve produces not only a prime list, but also a least factor function. It is particularly useful because the least factor function can accelerate rapid factorization.

In the classical Sieve of Eratosthenes, one begins with the list

$$S = \{2, 3, 4, \ldots, n\},$$

and proceeds by crossing out every composite number as a multiple of some prime. In contrast, the Euler Sieve avoids the simple crossing out of numbers. Instead, it marks each number with its least factor(equivalent to its least prime factor, which is greater than 1) using the function f. Consequently, a number i for which $f(i) \neq i$ is a composite number.

The Euler Sieve starts with a function f which for all x,

$$f(x) = 0,$$

and an empty prime list

$$ps = [].$$

The method proceeds as follows:

1. For each number i in the domain $\{2, \ldots, n\}$, if $f(i)$ is not yet assigned (i.e., $f(i) = 0$), then i is recognized as prime, and we set $f(i) := i$. Then, we add i to the end of the prime list.
2. For each i and for each prime p from the prime list so far, if $p \leq f(i)$ and $(i \times p) \leq n$, the number $i \times p$ is marked by setting $f(i \times p) = p$. This ensures that $i \times p$ is assigned its least factor.
3. This procedure is repeated by increasing i by 1 till it reaches n.

The pseudocode for the Euler Sieve is presented below:

```
function EulerSieve(n):
    f[2..n] = 0
    ps = []
    for i from 2 to n: // outer loop
        if f[i] == 0:
            f[i] = i
            append i to ps
        for each prime p in primes: // inner loop
            if p > f[i] or i * p > n:
                break
            f[i * p] = p
    return (primes, f)
```

We observe that the assignment `f[i * p] = p` uses a prime p which by construction satisfies $p \leq f[i]$ (the least factor of i). So it is indeed the smallest prime factor of $i \times p$. This guarantees that each composite is marked exactly once, just as intended, and yields an overall linear time complexity. All we need to ensure is that each state in the looping process gives a correct prime list and a correct least factor function.

More details will be shown in section §4. Formalization of the Euler Sieve. Note, however, that in our formalization, we only verified the algorithm's correctness, not its linear running time. We will describe current progress made on extending this formalization in section §5. Discussion and Future Work.

3. The Lean prover

In recent years, the pursuit of rigorous, error-free mathematics has led to the development of formal verification tools known as proof assistants. These systems enable mathematicians and computer scientists to construct and verify proofs with machine-checked precision, ensuring that every logical inference adheres strictly to foundational rules. Among these tools, *Lean* has emerged as a prominent platform, blending expressive mathematical language with robust verification capabilities.

3.1. Overview of Lean

Lean is an interactive theorem prover and functional programming language developed to support formal reasoning in mathematics and computer science. It is based on *dependent type theory*, a framework that unifies programming and logic, allowing for the expression of complex mathematical concepts and the construction of proofs within the same language.

Lean's design emphasizes both automation and user interaction. It offers powerful tools for automated reasoning, such as tactics that can simplify proof construction, while also allowing users to guide the proof process interactively. This balance makes Lean suitable for a wide range of formalization tasks, from verifying simple algorithms to formalizing advanced mathematical theories.

3.2. The Mathlib library

A cornerstone of Lean's ecosystem is *mathlib*, an extensive library of formalized mathematics developed collaboratively by the Lean community. Mathlib encompasses a broad spectrum of mathematical domains, including algebra, analysis, topology, and number theory. Its comprehensive collection of definitions, theorems, and proofs serves as a valuable resource for users aiming to build upon existing formalizations or contribute new ones.

The collaborative nature of mathlib fosters a dynamic environment where contributors can share their work, receive feedback, and collectively advance the formalization of mathematics. This communal effort not only accelerates the development of the library but also promotes best practices in formal proof construction.

3.3. **Applications and impact**

Lean has been employed in various significant formalization projects, demonstrating its versatility and robustness. Notably, it has been used to formalize complex mathematical results, such as the proof of the *Liquid Tensor Experiment* led by Peter Scholze and the formalization of the *Polynomial Freiman–Ruzsa (PFR) conjecture* by Terence Tao and collaborators. These endeavors highlight Lean's capacity to handle intricate mathematical arguments and its growing role in contemporary mathematical research.

Beyond pure mathematics, Lean's formal verification capabilities have applications in computer science, particularly in verifying the correctness of software and algorithms.

3.4. **Getting started with Lean4 for beginners**

For readers new to formal verification and theorem proving, learning Lean4 might seem daunting initially. Here we list a recommended sequence of resources:

- **The Natural Number Game**[1]: This interactive web-based tutorial introduces the basics of formal proofs in Lean through a gamified approach. Users solve progressively challenging puzzles involving natural numbers, arithmetic, and logic.

- **Mathematics in Lean**[2]: This free online textbook provides a comprehensive introduction to using Lean for formal mathematics. It covers foundational concepts, essential techniques, and practical examples in different mathematical branches.

- **Theorem Proving in Lean 4**[3]: This is also a free online textbook. It is an authoritative resource that serves as both a reference manual and a textbook,

[1]https://adam.math.hhu.de/#/g/leanprover-community/nng4
[2]https://leanprover-community.github.io/mathematics_in_lean/
[3]https://leanprover.github.io/theorem_proving_in_lean4/

describing Lean's core features, advanced concepts, and programming strategies in detail.

- **Lean 4 Web**[4]: This online platform enables immediate experimentation with Lean4 and Mathlib without any installation. Beginners can quickly test examples and familiarize themselves with Lean's syntax and proof style interactively.

In the subsequent sections, we will delve into the formalization of the Euler Sieve algorithm within Lean, illustrating how the prover's features facilitate the rigorous verification of algorithmic correctness.

4. Formalization of the Euler Sieve

In this section, we present our *Lean* formalization of the Euler Sieve(the Linear Sieve). Our approach is organized into several layers, each building systematically upon the previous one to ensure the correctness of the algorithm by construction. First, we use functions to represent arrays and rewrite the algorithm in a recursive style. Then, we introduce two `final` types that explicitly represent the state at each stage of computation. After establishing several auxiliary lemmas to support the subsequent correctness proofs, we provide a detailed, step-by-step explanation of the three `complete` functions. These functions extend the original recursive definition by embedding correctness proofs within the recursion itself, explicitly passing verified states as parameters. Ultimately, this structured approach culminates in a fully verified construction of the Euler Sieve.

4.1. Simulating the update of functions

In *Lean*, we simulate an update of a function by returning a new function. To achieve this, we define the following *Lean* function:

```
def updateFunction (f : Nat → Nat) (k v : Nat) : Nat → Nat :=
  fun x => if x = k then v else f x
```

This `updateFunction` works as follows: it takes an existing function f, an index k, and a new value v. It returns a new function which, when applied to an index x, checks if x is equal to k. If so, it returns v. Otherwise, it returns the original value $f(x)$. This provides a purely functional way to simulate the update operation.

[4]https://live.lean-lang.org/

4.2. Rewriting the algorithm recursively

Since Lean is a functional programming language, we want to rewrite the for-loop version of the Euler Sieve in Section §2 into a recursive version. The outer loop in the pseudocode over i becomes `EulerSieveAux`, and the inner loop over the current prime list becomes `processPrimes`. This approach preserves the behavior of the original Euler Sieve while ensuring termination can be verified in *Lean*.

The inner loop of the Euler Sieve is defined as follows:

```
def processPrimes (i : Nat) (f : Nat → Nat) (ps : List Nat)
(n : Nat): Nat → Nat :=
  match ps with
  -- if no primes in prime list, return f
  | []          => f
  -- else, split the prime list ps to an element p and the following
  ↪  sublist ps'(process the next prime after p)
  | p :: ps'    =>
    if p > f i ∨ i * p > n then f
    else
      let f' := updateFunction f (i * p) p
      processPrimes i f' ps' n
```

The outer loop of the Euler Sieve is defined as follows:

```
def EulerSieveAux (n i : Nat) (ps : List Nat) (f : Nat → Nat) :
(List Nat × (Nat → Nat)) :=
  if i > n then (ps, f)
  else
    let (ps', f') :=
      if f i = 0 then
        -- i is prime, update f
        let fNew := updateFunction f i i
        -- enter inner loop
        (ps ++ [i], processPrimes i fNew (ps ++ [i]) n)
      else
        -- enter inner loop
        (ps, processPrimes i f ps n)
    EulerSieveAux n (i + 1) ps' f'
-- ensures the recursion will terminate
termination_by n + 1 - i
```

Finally, call `EulerSieveAux` with `EulerSieve` to pack everything together:

```
def EulerSieve (n : Nat) : (List Nat × (Nat → Nat)) :=
  -- start sieving at 2, set initial list to empty and fuction to
  ↪  zero
  EulerSieveAux n 2 [] (fun _ => 0)
```

4.3. Necessary types

To maintain the correctness invariants throughout the recursion, we define several *dependent types* that bundle both data and their correctness properties. This approach ensures that once a value of these types is constructed, it automatically carries all necessary information, so subsequent functions can rely on it without an additional proof burden. The idea is to use these types within the recursive function, allowing us to construct an induction proof during recursion.

4.3.1 `final_PS`

We begin with a predicate to ensure that a list of natural numbers is strictly sorted in ascending order:

```
def list_sorted : List Nat → Prop
| []       => True
| [x]     => True
| x::y::l => x < y ∧ list_sorted (y::l)
```

Next, we package this property together with a bounding condition, creating the `sorted_bounded_list` type:

```
def sorted_bounded_list (i j: Nat) : Type :=
  { ps : List Nat // list_sorted ps ∧ (∀ x ∈ ps, j ≤ x ∧ x ≤ i) }
```

An object of type `sorted_bounded_list` i j is thus a sub-type of containing:

1. A list `ps` of natural numbers.
2. A proposition that `ps` is sorted in ascending order.
3. A proposition that every element of `ps` lies between `j` and `i`.

While `sorted_bounded_list` i j guarantees sorting and bounding, we still need a proposition that this list exactly consists of the primes in the specified range. We capture this condition with `correct_PS`:

```
def correct_PS : sorted_bounded_list i j → Prop :=
  --split data and proposition from sorted_bounded_list i j
  fun ⟨ps, _h⟩ =>
    ∀ x, j ≤ x ∧ x ≤ i → (x ∈ ps ↔ Nat.Prime x)
```

Thus, `correct_PS` states that for every x between `j` and `i`, membership in `ps` is equivalent to x being a prime.

Finally, we combine these notions into `final_PS`:

```
def final_PS (i: Nat) : Type :=
  { ps : sorted_bounded_list i 2 // correct_PS ps }
```

4.3.2 `final_FS`

Similarly, we need to formalize the least factor function with another final type:

```
def final_FS (i n : Nat) : Type :=
  { f : Nat → Nat //
      (∀ m, (f m = 0) ∨ (f m = Nat.minFac m)) ∧
      (∀ x, 2 ≤ x ∧ x ≤ i →
          f x = Nat.minFac x ∧
          ∀ y, (Nat.Prime y ∧ 2 ≤ y ∧ y ≤ f x) →
            x * y ≤ n → f (x * y) ≠ 0) }
```

An element of type `final_FS` i n therefore ensures:

1. For any natural number m, `f(m)` is either 0 or the least factor of m.
2. For each x in the range $[2, i]$, `f(x)` has already been correctly set to the least factor of x.
3. Crucially, whenever $x \times y \leq n$ and y is a prime that less or equal to the least factor of x, we must not leave `f(x × y)` at 0 (which should be the least factor of $x \times y$). This condition enforces that every composite discovered in `processPrimes` is indeed marked correctly.

4.4. **Lemmas for proofs**

Several auxiliary lemmas are used to support the proofs of correctness in our formalization. We only show one of the key lemmas in this paper.

Lemma 4.1 (mul_of_least)**.** *Let p be a prime number and let $x \geq 2$ be an integer. If p is less than or equal to the least factor of x, then the smallest factor of $p \times x$ is p.*

```
lemma mul_of_least {p x : Nat} (hp : Nat.Prime p) (hx : 2 ≤ x)
  (h : p ≤ Nat.minFac x) : Nat.minFac (p * x) = p := by
  have hl: Nat.minFac (p * x) ≤ p := by
    apply Nat.minFac_le_of_dvd
    apply Nat.Prime.two_le hp
    apply Nat.dvd_mul_right
  have hg: p ≤ Nat.minFac (p * x) := by
    let q := Nat.minFac (p * x)
    have hq_dvd : q ∣ p * x := by apply Nat.minFac_dvd
    have hq_nz : p * x ≠ 1 := by aesop
    have hq_prime : Nat.Prime q := Nat.minFac_prime hq_nz
       have hq_cases : (q  ∣ p)  ∨ (q  ∣ x) := (Nat.Prime.dvd_mul
    ↪  hq_prime).mp hq_dvd
    match hq_cases with
    | Or.inl hqp =>
      have q_eq_p : q = p := (Nat.prime_dvd_prime_iff_eq hq_prime
      ↪  hp).mp hqp
      linarith
    | Or.inr hqx =>
      apply le_trans h (Nat.minFac_le_of_dvd (Nat.Prime.two_le
      ↪  hq_prime) hqx)
  linarith
```

Explanation:

`hl` : First, it proves that the least factor of $p \times x$ is less than or equal to p by using the property that the least factor must be less than or equal to other factors.

`hg` : Second, it shows that p is less than or equal to the least factor of $p \times x$ by letting q be the least factor of $p \times x$ and using the fact that q divides $p \times x$. Because q is also a prime, we can analyze the two possible cases — either q divides p or q divides x — and apply the given hypothesis that p is less than or equal to the least factor of x.

Then, we combine them together to get p is equal to the least factor of $p \times x$.

4.5. **Complete functions**

The final piece of our formalization consists of three complete functions that integrate all previous components to yield a fully verified Euler Sieve. It expands the previous recursive functions and passes the correct propositions as parameters with proofs. Thus, the code is organized in such a way that it automatically includes the required proofs during recursion. In this approach, the induction proof step is directly embedded within the recursive construction, streamlining the verification process.

4.5.1 `processPrimes_complete`

This function is the expanded version of `processPrimes`. It integrates all the necessary proof obligations with the algorithmic update of the least prime function f. In every recursive call, the function carries along five key propositions (named $hf1$ through $hf5$) that ensure the correctness of the sieve. In the end, `processPrimes_complete` returns a type `final_FS (i+1) n` with its range upper bound.

Here is the simplified code fragment:

```
def processPrimes_complete (n i j : Nat) (hi : 2 ≤ i)
  (ps : sorted_bounded_list (i+1) (j+1)) (hps : correct_PS ps)
  (f : Nat → Nat)
  (hf1 : ∀ m, (f m = 0) ∨ (f m = Nat.minFac m))
  (hf2 : ∀ x, 2 ≤ x ∧ x ≤ i →
    f x = Nat.minFac x ∧
    ∀ y, (Nat.Prime y ∧ 2 ≤ y ∧ y ≤ f x) → x*y ≤ n → f(x*y) ≠ 0)
  (hf3 : f (i+1) = Nat.minFac (i+1))
  (hf4 : ∀ a, (Nat.Prime a ∧ a ≤ j)
    → f ((i+1)*a) = Nat.minFac ((i+1)*a))
  (hf5 : ∀ m, f m ≤ (i+1))
  : {F : final_FS (i+1) n // ∀ m, F.1 m ≤ (i+1)} :=
match ps with
| ⟨[], ps_props⟩ => -- Base case
    ⟨ ⟨ f, hf1, ... ⟩, hf5 ⟩
| ⟨[p], ps_props⟩ =>
  if hcond : p > f (i+1) ∨ (i+1)*p > n then
    ⟨ ⟨ f, hf1, ... ⟩, hf5 ⟩
  else
    let f' := updateFunction f (p * (i+1)) p
    -- create an empty sorted_bounded_list
    let ps' := empty_sorted_bounded_list (i+1) (p+1)
    -- construct necessary propositions
    ......
    processPrimes_complete n i p hi ps' hps' f' hf1' hf2' hf3' hf4'
     ↪  hf5'
| ⟨p :: ps', ps_props⟩ =>
  if hcond : p > f (i+1) ∨ (i+1)*p > n then
    ⟨ ⟨ f, hf1, ... ⟩, hf5 ⟩
  else
    let f' := updateFunction f (p * (i+1)) p
    let ps_new : sorted_bounded_list (i+1) (p+1)
    := ⟨ q :: ps', ... , ... ⟩
    -- construct necessary propositions
    ......
    processPrimes_complete n i p hi ps' hps' f' hf1' hf2' hf3' hf4'
     ↪  hf5'
```

Below is a breakdown of each hypothesis:

- `hf1` This proposition ensures that `f(m)` never contains invalid data: it is always equal to zero or the least factor of m.
- `hf2` This is the proposition part of type "`final_FS` i n". It ensures that every number before the current call of `processPrimes_complete` is correct. This proposition is invariant during the recursion, we only need to show that the update function will not change anything in the previous domain.
- `hf3` This proposition indicates `f(i + 1)` is also set to its least factor. Same as `hf2`, it can also be considered invariant during the recursion.
- `hf4` This proposition is the core of `processPrimes_complete`. It ensures that for every prime `a` not exceeding `j`, `f((i + 1) × a)` is directly assigned to the correct least factor. During the recursion of `processPrimes_complete`, after prime `p` is processed, `j` is updated to `p` for the subsequent recursive call. As `j` increases during the recursion, this proposition will ultimately be equivalent to

  $$\forall a,\ \Big(Nat.Prime\ a \land a \le (i+1)\Big) \to f\Big((i+1)\cdot a\Big) = Nat.minFac\Big((i+1)\cdot a\Big).$$

  We can then combine it with `hf2` and `hf3` to get the desired proposition part of `final_FS (i+1) n`.

`hf5` This proposition states an upper bound, guaranteeing that the range of $\texttt{f}(\texttt{m})$ never exceeds $\texttt{i}+1$. It is crucial for `EulerSieveAux_complete`. We need it to tell that a marked number is indeed a composite number.

The definition inside `processPrimes_complete` follows a pattern match on the prime list:

- **Base Case:** The prime list is empty. In this branch, the function simply packages the current least factor function `f` (together with the input hypotheses) into an element of type `final_FS (i+1) n`.
- **Recursive Case:** The prime list contains a singleton $[p]$ or more elements $(p :: q :: ps')$. In this branch, the function first checks a condition:
  - If $\texttt{p} > \texttt{f}(\texttt{i}+1)$ or $(\texttt{i}+1) * \texttt{p} > \texttt{n}$, then no further update is necessary and the current state is packaged accordingly.
  - Otherwise, the least prime function is updated at index $(\texttt{i}+1) * \texttt{p}$ via `updateFunction`, and `processPrimes_complete` increases j to p, and recurses on the tail of the prime list.

Hence, we build `processPrimes_complete` and ensure that it will output a type of `final_FS (i+1) n` with upper bound as desired.

4.5.2 `EulerSieveAux_complete`

The function `EulerSieveAux_complete` is the Expansion of `EulerSieveAux`. It recursively extends the sieve from the current index i up to n and updates the current prime list and least factor function. The function calls `processPrimes_complete` to mark the multiples of $i+1$ and then enters recursion. In the end, it returns a type `final_PS n` and a type `final_FS n n`

A simplified code fragment is provided below:

```
def EulerSieveAux_complete (n i: Nat)(hi: 2 ≤ i)(hin: i ≤ n)
  (PS : final_PS i) (F : final_FS i n) (hF: ∀ m, F m ≤ i)
  : (final_PS n) × (final_FS n n) :=
if h : i = n then
  (PS, F)  -- Base case: i reached the upper bound n.
else
  have hi' : 2 ≤ i+1 := by linarith
  have hi'n : i+1 ≤ n := lt_iff_le_and_ne.mpr ⟨hin, h⟩
  let ⟨⟨ps, psProp1⟩, psProp2⟩ := PS
  let ⟨f, fProp⟩ := F
  if hval : f (i+1) = 0 then  -- Case: i+1 is prime.
    let newps := ps ++ [i+1]
    let newF  := updateFunction f (i+1) (i+1)
    -- construct necessary propositions
    ......
    let F' := processPrimes_complete n i 1 hi newPS ... newF ...
 EulerSieveAux_complete n (i+1) hi' hi'n ⟨⟨newps,...⟩, ...⟩ F'.1 F'.2
  else  -- Case: i+1 is composite.
    -- we only change the bound of PS
    let newPS : sorted_bounded_list (i+1) 2 := ...
    -- construct necessary propositions
    ......
    let F' := processPrimes_complete n i 1 hi newPS ... f ...
    EulerSieveAux_complete n (i+1) hi' hi'n ⟨newPS,...⟩ F'.1 F'.2
```

The code is very straightforward. The function mainly reflects the original `EulerSieveAux`. It does not need to construct many extra propositions to make the recursive calls. We only need to make sure that the updated prime list and the least factor function still belong to the `final` types.

However, note that $f(i+1) \neq 0$ does not necessarily indicate that $i+1$ is composite. If $f(i+1) = i+1$, then $i+1$ is prime, and this is entirely permissible in the `final_FS` type. To correctly infer the status of $i+1$, we must combine the information provided by the hypothesis $hF : \forall m, Fm \leq i$, which can be inferred by the proposition part of the `processPrimes_complete`'s return value(where `hf5` is used for).

4.5.3 `EulerSieve_complete`

The top-level function `EulerSieve_complete` provides the fully verified Euler Sieve for any natural number n (with $2 \leq n$). It sets up the initial prime list and least factor function, then calls `EulerSieveAux_complete` to recursively build the complete sieve. It returns a type `final_PS n` and a type `final_FS n n`, which guarantees the correctness of the sieve's outputs.

A simplified code fragment is given below:

```
def EulerSieve_complete (n : Nat) (hn : 2 ≤ n)
  : final_PS n × final_FS n n :=
let PS_init : final_PS 2 :=
  ⟨ [2], ...⟩ -- initial prime list with proof
let f := updateFunction (updateFunction (fun _ => 0) 2 2) 4 2
let F_init : final_FS 2 n :=
  ⟨ f, ...⟩ -- initial least factor function with proof
-- Start the auxiliary recursion
EulerSieveAux_complete n 2 (by norm_num) hn PS_init F_init (...)
```

Thus, `EulerSieve_complete` encapsulates the full verified construction of the Euler Sieve.

4.6. Results

Our formalization of the Euler Sieve in *Lean* has been successfully completed and verified in Lean4 v4.19.0 build for Linux Ubuntu. The final code is over 900 lines. For full details, please visit the GitHub repository: https://github.com/IsaacLi74/Euler_Sieve.

5. Discussion and future work

In our current formalization, we have rigorously established the correctness of the Euler Sieve algorithm using the Lean proof assistant. While our approach ensures precise verification, it does not yet address the proof of linear running time, which remains an important topic for future development.

Beyond verifying correctness, our methodology demonstrates a systematic framework. The structured design, including explicit states, correctness conditions, and recursive constructions, provides a general blueprint that can guide the verification of other sequence-generating algorithms.

Overall, future work may include:

- Formally proving the linear time complexity of the Euler Sieve.
- Extending the framework to verify other sequence-generating algorithms.

Acknowledgements

I would like to thank Professor Thomas Callister Hales for his invaluable guidance throughout my work in MATH 1900. His insights and mentorship have greatly contributed to the development of this paper. What began as a simple learning project quickly grew into more substantial research, thanks to his encouragement at every stage — from selecting the problem and verifying the algorithm to creating the poster

and submitting the final paper. It is hard to believe that all of this has taken place in just two months.

References

[1] D. Gries and J. Misra, "*A linear sieve algorithm for finding prime numbers*," 1978. [Online]. Available: https://doi.org/10.1145/359657.359660

[2] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "*The Lean Theorem Prover*," 2015. [Online]. Available: https://lean-lang.org/papers/system.pdf

[3] Lean Community, "*mathlib4*," 2022. [Online]. Available: https://github.com/leanprover-community/mathlib4

[4] J. Avigad, L. de Moura, and S. Kong, "*Theorem Proving in Lean*," 2015. [Online]. Available: https://leanprover.github.io/theorem_proving_in_lean4/

[5] J. Avigad and P. Massot, "*Mathematics in Lean*," 2020. [Online]. Available: https://github.com/leanprover-community/mathematics_in_lean

Isaac (Rucheng) Li
Department of Mathematics, University of Pittsburgh, Pittsburgh, PA
E-mail: ISL74@pitt.edu