COMPARING THE COMPUTATIONAL EFFICIENCY OF THREE COMMON PRIME SIEVES.

J. Meyers, J. Zieleman, and R. Folks

Abstract. We explain and use three common algorithms for generating (or sieving) prime numbers. We then time each sieve's performance in generating prime numbers up to *N* for *N* ranging from 1000 to 1,000,000 to see which method is the most efficient at finding an exhaustive list of primes less than *N*. We then present the case that the Sieve of Eratosthenes is the most efficient within this range, and conjecture that as $N \to \infty$, it remains the most efficient.

1. INTRODUCTION

As is common knowledge among mathematicians, there does not yet exist a formula for finding a complete list of prime numbers. There exist formulas that can generate prime or mostly prime numbers, but none yet are comprehensive. This is where prime sieves become of use. Prime sieves, who take their name from the common kitchen instrument, begin with a list of all natural numbers up to a limit *N*, and implement some algorithm for finding which numbers are composite, and letting them pass through the sieve, I.E they mark them as non-prime. Most of the most popular sieves were created before the invention of the modern computer, and since then prime numbers have played a large role in fields of computer science like public key cryptography and hashing. Naturally, the following question arises: which prime sieve is the most efficient?

To solve this problem, we must gather the three most common prime sieving methods: the sieve of Eratosthenes, the sieve of Sundaram, and the sieve of Atkin. Implement each method in Python 3. Then run them to find all primes less than some maximum number *N*, for each N within the testing range 1,000 to 1,000,000, increasing *N* by 1,000 for each run of the loop. Finally we must compare the times each method took using Matlab's graphing libraries.

We hypothesize that the sieve of Eratosthenes will be much faster for smaller values of *N*, but as *N* increases, the sieve of Atkin's computational complexity will grow slower than the other methods, making it faster for larger values of *N*.

2. ANALYSIS OF SIEVING ALGORITHMS

Every method being tested is the work of some mathematician throughout history that has used analysis to exploit some trait of prime numbers to speed up the process of prime finding. In this section, each method being tested is analyzed for how it works, as well as how its computational complexity grows as the amount of numbers to search through grows.

Trial Division. To begin we will demonstrate trial division. The purpose of trial division is to be a benchmark by which to test other prime finding methods. It is the slowest, albeit most intuitive way to determine a number's prime status. It is what a person may come up with when tasked with creating an prime sieve on the spot. Its procedure is as follows:

1. For any number *N*, create a list of all natural numbers less than *N* excluding 1.

2. Divide *N* by every element within this list and if its remainder is an integer, *N* is not prime.

Clearly, trial division makes no effort to reduce the number of computations required to determine if a number is prime. The number of divisions d for this method can be expressed as the equation $d \leq N - 2$, that is, any number requires at most division of every number below it, save the number 1. Trial division is not one of the sieves that we tested and graphed due to the sheer impracticality of the method making it too time consuming to test.

Sieve of Eratosthenes. The sieve of Eratosthenes is a simple and ancient algorithm discovered circa 250BC by Eratosthenes of Cyrene that is used to find prime numbers up to a given limit *N*. It is widely considered to be one of the most efficient ways to find relatively small prime numbers. It begins by defining an upper limit *N*, and declaring that *p* = 2, *p* being the current number the sieve is checking multiples of. Then the algorithm iteratively marks the multiples of *p* as composite. Once $p * q \geq N, q \in \mathbb{Z}$, *p* becomes the next number still in the list. This continues until $p \geq \sqrt{N}$. Consider this example for $N = 25$.

1. List all integers from 2 to $\sqrt{N} = \sqrt{25} = 5$. We will call this list P = $\{2, 3, 4, 5\}$
2. List all integers from 2 to *N*. We will call this list C.
3. C = $\{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25\}$
4. Let $p = 2$.
5. For all $p * q \geq N, q \in \mathbb{Z}$, remove $p * q$ from C.
6. C = $\{2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25\}$

As we can see, once 3 and 5 are done sieving, the list will be composed of completely prime numbers due to no number in the list being greater than 5 * 5, which is the last element cleared from the list. This method improves over trial division by reducing the number of computations it takes to remove a number from the list down to a single computation per number. Another improvement it makes over trial division is how it grows in complexity as $N \to \infty$. As stated, trial division's complexity grows linearly at a rate of $d \leq N - 2$, . The sieve of Eratosthenes' computations are;

$$\sum_{p \in P} \left\lfloor \frac{N}{p} \right\rfloor$$

where the variable p iterates through the list of primes P where $p \in P, p \leq \sqrt{N}$. The resulting summation grows far slower than trial division when you consider repeating each method hundreds of times, making it generally more efficient.

Sieve of Sundaram. The sieve of Sundaram is a prime sieve whose discovery is credited to the Indian mathematician S. P. Sundaram during the 1930s. This sieve works closely to the sieve of Eratosthenes, but instead of checking every number, only odd numbers are checked. The sieve achieves this using the following algorithm.

1. Begin with a list of the natural numbers from 1 to *N*.
2. Remove all numbers from the list that are of form i + j + 2ij where I and j are beholden to the following conditions.
   - $i, j \in \mathbb{N}, 1 \leq i \leq j$ and
   - $i + j + 2ij \leq N$
3. All remaining numbers in the list are put into the equation $2n + 2$ and added to the final list of primes

This algorithm doesn't just skip even numbers, it doesn't even consider them. All numbers in the remaining list can be proven to be prime using the following proof.

1. All odd integers can be expressed as $2k + 1$ where $k \in \mathbb{Z}$. Let o be an odd integer.
2. Let k be expressed as $k = i + j + 2ij$, i and j being constrained by the conditions above.
3. The follow equations are equivalent:
   - $o = 2(i + j + 2ij) + 1$
   - $o = 2i + 2j + 4ij + 1$
   - $o = (2i + 1)(2j + 1)$

This proves that o has two non-trivial odd factors, meaning it cannot be prime! The observant will note that this method does not just generate primes up to *N* however. This is due to the fact that all numbers in the list are doubled and have two added to them. In order to have this method compete on equal ground with the other two sieves, we have adjusted the maximum number N we are passing into each of our methods by the following equation. Let M be the new max number to check primes up to and let N be the number that this sieve creates primes up to.

$$M = \left\lfloor \frac{N-2}{2} \right\rfloor$$

All that is left to model the number of computations this sieve takes to produce its final list. Due to this sieve closely resembling the sieve of Eratosthenes, its growth will be similar. We again will use a summation, but this time the size of certain sets will need to be known. Let L be a set of all combinations of i + j + 2ij that satisfy the conditions of the sieve. Let Q be the list containing $\mathbb{N} \leq N$, N defined as above, and not marked off by the set L. Thus, we get the following;

$$4 \sum_{l \in L} l + 2(|Q| - |L|)$$

To explain, each element of the list L requires four computations to achieve, so each element is multiplied by four to get the total number of computations required. Then, once the list has been sieved, each element of that list must be multiplied by two and have two added to it, which is two separate computations, thus yielding the constants two and four. Subtracting the cardinality of Q from L yields the amount of numbers to be computed to reach the final list of primes.

Sieve of Atkin. The sieve of Atkin, invented in 2003 by A. O. L Atkin and D. J. Bernstien is the most complex sieve that we researched. It operates by creating a partition of the natural numbers up to some maximum number *N*. Each number is placed into one of four blocks of the partition by the following rules.

1. For each $x \leq N, x \in \mathbb{N}, b = x(mod 60)$
2. x is sorted into the following blocks of the partition based on the value of b
   - $b \in \{1, 13, 17, 29, 37, 41, 49, 53\} = P_1$
   - $b \in \{7, 19, 31, 43\} = P_2$
   - $b \in \{11, 23, 47, 59\} = P_3$
   - $b \notin P_1 \cup P_2 \cup P_3$

Depending on which block x was sorted into, the number of solutions x has with its respective block's quadratic equation determines its prime status. Each block's quadratic equation is listed as follows.

- $P_1 : 4x^2 + y^2 = n$
- $P_2 : 3x^2 + y^2 = n$
- $P_3 : 3x^2y^2 = n$

- P_4 : no equation, all $x \in P_4$ are not considered.

At this point, a list is created that has cardinality equal to the size of *N*. Each element of this list is simply a true or false value (true meaning prime and false meaning composite) that is initially set to false. For each positive integer solution to the number's respective equation, its position on the list is flipped. This means that only number that have an odd number of integer solutions are deemed to be prime candidates. All that is left at this point is to sieve the remaining list in a manner similar to the sieve of Eratosthenes, except instead of natural number multiples being marked off the list, each number's squares, up to $\sqrt{N}$ are marked off the list.

If the algorithm seems puzzling, that's because it is. There is a six page paper by its creators about how it operates, its correctness, and how it grows in computational complexity that is cited under our acknowledgments. Because their paper does a much more thorough job of explaining the algorithm, it will not be detailed further here.

3. PROCEDURE

The testing procedure used in this numerical experiment was simple. Each sieve was programmed into a computer using Python3 as separate functions in one file. Each function was programmed to take one parameter, the maximum number to find primes to. In another file, a loop was constructed such that those functions were called and timed before the loop would adjust the maximum number to find primes up to by one-thousand for each run of the loop. These times were output into a text file in a vector format for use in Matlab. These vectors, being an unwieldy nine-hundred ninety nine elements long, would not simply copy-paste into Matlab's desktop application on Linux without causing error. Rather than work with Matlab's data handling features, our workaround was to use Mathworks' online version of Matlab which had no problem dealing with a vector of this size. Each line was then colored and a legend was added for readability, and the graph was produced.

4. DATA AND CONCLUSIONS

Our hypothesis that any sieve would grow at different rates or be better for a specific range of numbers was false as shown by the graph (see appendix). The time each sieve took to generate a list of prime numbers was exactly proportional to the number of computations required to perform the sieves operations. While the data perplexed our group in the beginning, not in the least part due to the erratic behavior of the lines, the reason that the sieve of Eratosthenes is so much more efficient than the other two methods lies in the ratio of numbers it determines to be composite per computation. The sieve of Eratosthenes makes one computation, multiplying the current prime number by some integer value, and uses that computation to mark off a single number, meaning that the sieve of Eratosthenes has a one to one ratio of computations to removed numbers. Looking to the sieve of Sundaram we can find another ratio. For every six computations made, a single number is marked off, but the numbers are traversed twice as quickly due to even numbers not being considered. The ratio is hard to pin down exactly, but it is somewhere in the range of two or three computations per single number marked off the list, which explains why this sieve is twice as slow as the sieve of Eratosthenes. The sieve of Atkin is even more difficult to find a ratio to. First a computation is made to find its remainder when divided by sixty, then the sieve must find solutions to a quadratic equation which all take time, not to mention that some values will be squared which adds more computations to remove an certain amount of composite numbers from the list. If an exact ratio cannot be found, we may approximated it by looking to the graph. If the sieve of Eratosthenes is a one to one ratio, the sieve of Atkin looks to be around a three to one ratio.

Another question that has already been brought up is why the graphs behave erratically. The short answer is that we don't know. However, there are several observations that can be made about the nature of these spikes that may clue us in to a possible reason for their occurrence. Comparing the frequency of spikes in time between the sieve of Eratosthenes and the sieve of Sundaram show that neither fluctuates more often than the other, however, the sieve of Atkin tends to fluctuate a lot more often than either of the other two. Another observation is that there is some positive relationship between the occurrence of spikes and the maximum number of primes to generate. Finally, we can see that the spikes are relatively the same size, even between sieves. Each peak looks to be about a one tenth of a second off from the data points surrounding it. This would lead us to believe that these spikes have something to do with there being some kind of "snag" happening randomly within the computer when it tries to perform a computation. The evidence for this is that the number of computations required to generate larger and larger sets of primes would produce more snags naturally as a result of there being more computations to catch on. Another related piece of evidence to back this up is the occurrence of spikes in the sieve of Atkin. The sieve of Atkin requires the highest number of computations to perform, so it would make sense that this high number of computations produces more snags in the times. Outside of this explanation, we have no clue as to the nature of these random increases in the time it takes for sieves to run.

5. POSSIBLE TESTING ERRORS

It is entirely possible that our results are due to some error committed either in our code, or within the way we tested the sieves, or perhaps in our methodology entirely. After the collection of data was complete it was discovered that within our code for the sieve of Sundaram that the expression i + j + 2ij was being evaluated twice per loop, rather than being evaluated once and compared twice, which may save time greatly over our initial results. Another possible error is that we are using a programming language that is not best suited for this application. Once each algorithm is implemented such that it only takes a fraction of a second to produce any single list of prime numbers, the element of the computations that slows down the computations the most ceases to be the algorithm, and rather lies in the dealing with lists. Beginning with a list of boolean values and dealing with that works quite well in Python3, but if another language has a more efficient way of dealing with such lists it could dramatically change the results, and perhaps could even make another sieve be faster than the sieve of Eratosthenes. This is all conjecture however, and until further research is done our results remain as they are.

6. FUTURE WORK AND APPLICATIONS

The data we collected for this numerical experiment was only a tiny fraction on the scale of primes that are used in many computational tasks. Many numbers used in RSA's and Cock's algorithm's for public key encryption use a product of two prime number that are themselves larger than one duodecillion, or 10^{39}, which according to the sources cited would currently take more than one quintillion years to check. Needless to say if an algorithm existed that could generate prime numbers far more quickly than the algorithms that currently exist, then computer functions that require large prime numbers explicitly for the fact that they are hard to come up with would be weakened. Therefore, future work would rely on having access to computers outfitted with modern GPUs that are capable of doing these kinds of computations easily as well as the time to let them run and collect data. From this data we could see which algorithms are more or less efficient, which may lead to the creation of new, faster methods of generating prime numbers.