

Developing Software to Find Arithmetic Progressions of 28 Primes

Faculty Adviser: Ravi Fernando

Graduate Mentor: Zishen Qu

IML Scholars: Alex Mok, Amy Bradu, Rafael Rojas,
Shubhankar Desai, Steven Zhen

May 18, 2025

Abstract

Our goal is to find an arithmetic progression of 28 prime numbers (AP28), to hopefully beat the current record of 27. We adapted Jaroslaw Wróblewski's algorithm, which currently works for AP26, where he used a 3-stage approach to finding candidates for AP26 progressions, by first finding AP24s that might also extend to AP26, and then actually checking to see if it is an AP26. We generalized Wróblewski's sieving algorithm to increase our search space in terms of working with different common differences. We use the Bateman-Horn conjecture to prioritize certain regions of our search space for our programs through predicted probabilities of the likeliness to contain an AP28. To find these candidates computationally, instead of using the original way of finding candidates of shorter progressions, we aggressively weeded out candidates so that we were only left with possible AP28s. We altered bounds in candidate generation to ensure that we are looking for an AP28 and used Bryan Little's implementation of Wróblewski's design for operating on GPU and CPU to run this in a multithreaded manner rather than a single-thread, which improved speed considerably. Our current version of searching for an AP28 uses the same primes for sieving as in Wróblewski's code. In the future, we hope to deploy this working version to PrimeGrid so that our search for AP28s can get started through BOINC. Eventually, we also want to implement our generalization on our program of finding AP28s to work on GPUs and multithreaded CPUs.

1 Introduction

Our goal is to find an arithmetic progression of 28 prime numbers. Currently, the world record is 27, so we hope our work can help beat it.

The **Green-Tao Theorem** states that

“There are arbitrarily long arithmetic progressions of primes” [4].

This essentially means that for every natural number k , there exists some arithmetic progression of prime numbers with length k . This can be represented as a pair (n, D) where $n, n + 1D, n + 2D, \dots, n + (k - 1)D$ are all prime numbers. In our project, our goal is to help find a pair (n, D) where $n, n + 1D, n + 2D, \dots, n + 27D$ are all prime numbers.

Thus, we know that such an arithmetic progression exists; now, the challenge comes from actually finding it. For AP27, this pair is

$$(n = 224584605939537911, D = 18135696597948930)$$

There are so many possible candidates for (n, D) , so finding a correct pair computationally requires a level of complexity, in both the mathematics and programming aspect of the project.

Some of the ideas we will address in this paper include how to optimally search for possible AP28 candidates, exploring existing methods for finding arithmetic progressions of primes, how to eliminate bad candidates, and how to make this efficiently run on a computer (through both CPU and GPU).

2 The Algorithm

Our general approach is adapted from a particular algorithm proposed by Jarosław Wróblewski, in which his method attempts to find AP26's. Understanding this algorithm is vital given our modification of code serves to optimize this algorithm to be able to find AP28's. Jarosław Wróblewski's algorithm serves to split the search into three stages, stage one is finding the good candidates to possibly extend to AP26's, stage two is checking whether those certain candidates can be extended to AP26's, and stage three is checking for primality of the sequence. Here is a working example of how the algorithm functions for finding an AP26.

Suppose that we want an AP26. The way Wróblewski attempts to find an AP26 is he sieves for AP24's that may be extended to AP26's. From this, suppose we desire an AP24 of the form:

$$n + 0D, n + 1D, \dots, n + 23D,$$

which may be extended to an AP26. In particular, we are looking for a sequence of numbers of the form $n - 2D, n - 1D, n + 0D, n + 1D, \dots, n + 23D, n + 24D, n + 25D$, where at least 26 of these consecutive terms are prime. This is what we define as the 'window.'

2.1 Stage 1 Candidate Generation

The goal of this stage is to utilize sieving in such a way that creates candidates with relatively high probability of being an AP26. We define $23\#$ as the product of all primes between 2 and 23, also known as 23 primorial. In general, an AP26 cannot exist unless the common difference D is divisible by $23\#$. From this, we define the following:

1. For the common difference D that separates each prime in an AP26, let $23\#|D$ such that $29, 31, 37, 41, 43, 47, 53, 59 \nmid D$. By ensuring D is a multiple of $23\#$, we are in

essence filtering out a large amount of composite numbers from surviving candidate generation by making its divisors strictly out of small primes. This filtering of the composites allows for an easier formulation of the loops that iterate over each sequence.

2. For the primes we wish utilize for sieving, let $A = \{2, 3, 5, 29, 31, 37, 41, 43, 47, 53, 59\}$ where we will sieve modulo $a_i \in A$. This choice of prime numbers is based off of Wróblewski's own testing of the speed regarding which primes to sieve by.

Having defined the preliminaries, we will now begin the sieving process.

Let $p \in A$. We call r , the residue of $n \pmod{p}$, **good** if none of the terms of $n + 0D, n + 1D, \dots, n + 23D$ are divisible by p . And, the residue is labeled **bad** if any of the terms are divisible by p . For $p = 2, 3, 5$, all non-zero residues are labeled as good, $p = 29$ the only good residue is $3D$, and the rest of the p 's have good residues $1D, 2D, 3D, \dots, (p - 24)D$. This is how we sieve for possible AP24's which could possibly be extended to an AP26.

Next, define $MOD = \prod_{i=1}^{11} (a_i)$, for $a_i \in A$ where we will call a residue R of $n \pmod{MOD}$ good if and only if it reduces to a good residue modulo any individual prime divisor of MOD . And since we desire to generate good residues R of $n \pmod{MOD}$ through residues which pass the following congruence classes, for $p = 31, 37, 41, 43, 47, 53, 59$, define residue x_0 and define s_3, s_5, s_p each with the congruences:

$$\begin{aligned} x_0 &\equiv 1 \pmod{2} & x_0 &\equiv D \pmod{p} \\ x_0 &\equiv 1 \pmod{3} & s_3 &\equiv 1 \pmod{3} \\ x_0 &\equiv 1 \pmod{5} & s_5 &\equiv 1 \pmod{5} \\ x_0 &\equiv 3D \pmod{29} & s_p &\equiv D \pmod{p}, \end{aligned}$$

where each s_i are divisible by all $a_i \in A$ except for the aforementioned congruences. For the purpose of ensuring each residue is a multiple of the smaller primes, congruences using mod MOD is important. From this, the Chinese Remainder Theorem shows that this condition for residues is equivalent to being in some modulo class mod MOD , or more simply, being in the same modulo class as the product of all the smaller primes.

From here, we can find the good residues R of $n \pmod{MOD}$ through the following formula: $R = x_0 + j_3s_3 + j_5s_5 + j_{31}s_{31} + \dots + j_{59}s_{59}$, where we define j_i as:

$$\begin{aligned} j_3 &= 0, 1 \\ j_5 &= 0, 1, 2, 3 \\ j_{31} &= 0, 1, 2, 3, 4, 5, 6 \\ j_{37} &= 0, 1, 2, 3, 4, 5, 6, \dots, 12 \\ &\dots\dots\dots \\ j_{59} &= 0, 1, 2, 3, 4, \dots, 34, \end{aligned}$$

where we utilize nested loops to obtain these good residues by strictly using addition, subtraction and comparison. These nested loops serve to 'loop' through the aforementioned

system of congruences one prime at a time. This process of finding good residues R of $n \pmod{MOD}$ is what we define as ‘candidate generation’. Finding the possible ‘candidates’ that can be extended to AP26’s through the aforementioned congruences is extremely important in our search for an AP28. Now, having a method to generate the desired ‘candidates’ R of $n \pmod{MOD}$ efficiently, we move on to the second stage of the algorithm to check whether these candidates can be extended to AP26’s.

2.2 Stage 2 Sieving

For each good residue R of $n \pmod{MOD}$ of the form $R_i = n + iD$ for $i = 0, 1, 2, \dots, 23$, we know each term is not divisible by any $a_i \in A$. But, to continue finding good residues R of $n \pmod{61}, n \pmod{67}, \dots$, the algorithm, instead of individually checking each singular sequence generated by the good residue, lets each R_i represent sixty four sequences. Instead of $R_i = n + iD$, it instead treats each R_i as $R_i = h \times MOD + n + iD$ for $i = 1, 2, 3, \dots, 23$ and $h = 0, 1, 2, \dots, 63$. This ingenuity allows us to recognize that these sequences have the same residues of the small primes. As a result, they behave similarly to if we sieved by the small primes themselves. Thus, since we have already sieved using these smaller primes we are able to do the sieving on all 64 of the sequences simultaneously, helping avoid a lot of extra operations. This type of sieving is known as ‘bit-mask sieving,’ a method where for each $n \pmod{61}, n \pmod{67}, \dots$, we use a 64-bit word from an already computed table which allows for us to iterate over each desired $n \pmod{p}$. These ‘64-bit words’ are what we call shifts, a set of 64 subsequent elements of an admissible modulo class modulo MOD which we then represent as a binary string of relevant length. This method to check for primality of a sequence allows for high speed generation as each application of the bit-mask sieving strictly only checks the aforementioned ‘candidates’ which have successfully passed the $n \pmod{MOD}$ test. And, given both stages hard stop at the encounter of a ‘bad residue,’ this allows for high speed sieving where each new application of the bit-mask sieving is a 1 if the residue is not divisible by any of the primes in $n \pmod{61}, n \pmod{67}, \dots$ and 0 if any are. Effectively, we use binary to represent the successful and unsuccessful sequences with speed.

2.3 Stage 3 Primality

Once a candidate survives both stage one and stage two, Wróblewski begins testing for the primality of 26 consecutive terms in a certain window. To do this, he begins by testing the $n + 5D, n + 6D, \dots, n + 23D, n + 24D, n + 25D$ stopping once it encounters a composite. Then, if there are at least 10 consecutive primes within the sequence, he then begins testing of the $n - 2D, n - 1D, \dots, n + 4D$ for primality. Again, the testing for primality stops at encountering a composite number. By eliminating the possibility of a composite number being a part of the middle portion of the sequence, we know that we either found the AP or did not.

3 Modifying Software to Search for AP28

Wróblewski’s work on finding an AP26 is the foundation of our search for an AP28; the main principles and concepts behind his code and sieves are carried over for our purposes. We discuss the changes made to Stage 1 of Wróblewski’s AP26 search; Stages 2 and 3 are the same. We keep the following assumptions made by Wróblewski:

1. Progression difference D is divisible by $23\#$ and is **not** divisible by any of the following primes: 29, 31, 37, 41, 43, 47, 53, 59. D is fixed during the whole search segment.
2. We use the following primes for sieving at this stage: 2, 3, 5, 29, 31, 37, 41, 43, 47, 53, 59.

The third assumption, “We do sieving for sequences of length 24 in a specific way described below,” is not kept, because we are searching for an AP28. The details behind our changes of the sieving algorithm in Stage 1 is described below.

3.1 Switching From Sieving for an AP24 Candidate to an AP28 Candidate

As shown in Section 2.1, the sieving algorithm of Wróblewski’s code tries to find the first term of an AP24 with the condition that it is congruent to $3D \pmod{29}$. Since we are only focusing on AP28s, we altered the sieve to directly search for the first term of an AP28. More specifically, we recategorized the good and bad residues. Let p be a prime number used for sieving. The following is Wróblewski’s definition:

1. For $p = 2, 3, 5$ all non-zero residues are good.
2. For the remaining p ’s all the good residues are $D, 2D, 3D, \dots, (p - 24)D$.
3. We make $p = 29$ an exception, where we consider $3D$ to be the only good residue.

Our change is straightforward; all the good residues are now $D, 2D, 3D, \dots, (p - 28)D$ for the remaining p ’s. We use this definition in our generalized search for AP28, which is described later.

3.2 AP28 Search Program

It is expected that the only one good residue for modulus 29 is $D \pmod{29}$; however, we decided that $3D \pmod{29}$ is to remain a good residue for the sake of backwards compatibility with the ongoing AP search from PrimeGrid and its method of dividing work units. To reconcile, our AP28 Search program actually sieves for the third term of an AP28. Thus, for the remaining p ’s, all the good residues are now $3D, \dots, (p - 26)D$.

Being able to directly sieve for an AP28 instead of shorter APs saves computational time. To further reduce running time, we tightened the search space by skipping sequences that are guaranteed to be not be AP28s. We do this by restricting the sieving with primes 31

to 59, specifically we remove the first two and last two elements of j_{31}, \dots, j_{59} as defined in Section 2.1.

As a result of these changes, we reduced runtime significantly. To somewhat measure the improvement, we observe the number of candidates that Wróblewski's code generates compared to the AP28 Search program, because the number of candidates is the number of iterations the sieving algorithm has. AP28 Search creates 620217000 candidates while Wróblewski's creates 4808136200 candidates. $4808136200/620217000 \approx 7.752$, so AP28 Search is at least 7.752 times faster.

4 Generalizing AP28 Search

Since primes 29, 31, 37, 41, 43, 47, 53, 59 are used for sieving, D cannot be divisible by those numbers. This limits our choices in common differences; to remove this restriction, we made a modified version of the AP28 Search program that generalizes the sieving algorithm with minor changes to Stage 2. It is important to note that we still maintain $23\# \mid D$.

4.1 Stage 1 Changes

Let P be a set of prime numbers used for sieving, Q be the set of prime factors of D , and E_k be the set of prime numbers from 2 to k . In the AP28 Search program,

$$P = \{2, 3, 5, 29, 31, 37, 41, 43, 47, 53, 59\}, Q = \{2, 3, 5, 11, 13, 17, 19, 23\}.$$

Notice that $P = (E_{59} \setminus Q) \cup \{2, 3, 5\}$. Sieving with primes not in Q is useful, since there are more bad residues to partition \mathbb{N} , reducing our search space. Sieving with 2, 3, 5 are clearly useful, despite being factors of D , due to their low number of congruence classes, so we include them in P .

For the sake of simplicity in the code, we want to sieve with 8 primes. Suppose we want to search for an AP28 where $D \mid \ell$ with $\ell \in \{29, 31, 37, 41, 43, 47, 53, 59\}$. Then,

$$Q = \{2, 3, 5, 11, 13, 17, 19, 23, \ell\}.$$

Thus, $P = (E_{61} \setminus Q) \cup \{2, 3, 5\}$. We include the prime 61 in P so that $|P| = 8$. After determining P and Q , we determine the good and bad residues as described in the Section 3.1 and sieve for candidates as usual. Our generalized AP28 Search implements this case.

Another case it implements is searching for an AP28 where $\ell, \alpha \in \{29, 31, 37, 41, 43, 47, 53, 59\}$ with $\ell \mid D$, $\alpha \mid D$, and $\ell \neq \alpha$. Then,

$$Q = \{2, 3, 5, 11, 13, 17, 19, 23, \ell, \alpha\}.$$

Thus, $P = (E_{67} \setminus Q) \cup \{2, 3, 5\}$. Again, we include the prime 67 in P so that $|P| = 8$, and we determine the good and bad residues as described in the Section 3.1 and sieve for candidates as usual.

4.2 Stage 2 Changes

Since our generalized AP28 Search potentially uses 61, 67, or both for the sieve, we may not need those primes to be in the bitmask sieving. Therefore, we exclude them when these situations arise.

4.3 CONST_Generator

Each time the factors of D change, the parameters of the program also change, namely s_3, s_5, s_p for all $p \in P$ with $p > 2$ and MOD as described in Section 2.1, $n0, n30$, and the primes used for sieving that are not factors of D (i.e. PRIME1, PRIME2, etc.). The constants $n0, n30$ are used for helping to calculate the candidates. Luckily, these inputs are directly derived from P and Q with the Chinese Remainder Theorem. We made a program in C++, CONST_Generator, that does all of the computations of the various constants and writes to the appropriate header file, which is CONST.H for the AP28 Search program and CONST_kp.q.H for the generalized AP28 Search. Moreover, CONST_Generator also declares the OK and OKOK arrays in the header file.

CONST_Generator takes in seven inputs:

- filename: a string that specifies the path to the header file that CONST_Generator writes to
- commonDifferenceDivisors: an array that contains the same elements as the set Q
- primesForCandidates: an array that contains the same elements as the set P
- specialPrime: a prime number where we want to specify a single good residue of its modulus; this is analogous to only letting $3D \bmod 29$ to be a good residue in Wróblewski's program. In this case, specialPrime would be 29. If we do not want to give a special condition, then specialPrime should be set to 0.
- multiply: an integer that specifies the good residue of modulo specialPrime; for example, to specify that $3D \bmod 29$ is the only good residue of modulo 29, multiply is set to 3. If we do not want to give a special condition (i.e. specialPrime = 0), then set multiply to be 0.
- bitmaskPrime: the greatest prime number that we use for bitmask sieving as described in Section 2.2
- sievePrime: the greatest prime number that we use for sieving beyond bitmaskPrime; in the AP28 Search program, sievePrime is 541

First, commonDifferenceDivisors and primesForCandidates are sorted by ascending order. CONST_Generator then uses the Sieve of Eratosthenes to obtain an array of primes from 2 to sievePrime; this is for declaring the OK and OKOK arrays. After this, the program begins writing to filename. It computes the common difference, or D , by multiplying the elements of commonDifferenceDivisors and writes to filename. Then, it finds the values of PRIME1, PRIME2, etc. by set subtracting primesForCandidates and commonDifferenceDivisors and

write them to filename. Next, it computes the value of MOD by simply multiplying the elements of `primesForCandidates` and writes to filename. For the $n0$ constant, the code solves the following system of congruences:

$$\begin{aligned} n0 &\equiv \text{multiply} \cdot D \pmod{\text{specialPrime}} \\ n0 &\equiv D \pmod{p} \end{aligned}$$

where $p \in P = \text{primesForCandidates}$ with a Chinese Remainder Theorem function by Geeks-forGeeks. If there are no special conditions, then $n0 = D$. For the $n30$ constant, the code solves the following system of congruences:

$$\begin{aligned} n30 &\equiv 1 \pmod{2} \\ n30 &\equiv 1 \pmod{3} \\ n30 &\equiv 1 \pmod{5} \\ n30 &\equiv 0 \pmod{p} \end{aligned}$$

where $p \neq 2, 3, 5$ with the same Chinese Remainder Theorem function. After writing $n0, n30$, it starts computing for s_p for each $p \in \text{primesForCandidates}$ by solving systems of congruences as described in Section 2.1. Finally, `CONST_Generator` writes the OK and OKOK arrays to filename.

5 A Heuristic Approach to Structuring the Search

That every arithmetic progression of 28 values, be they prime or not, is uniquely identified by its first element and common difference gives our search space a two dimensional character. As a consequence, it is not immediately obvious what the most efficient way to iterate through this space is if our fundamental goal is to check those arithmetic progressions which are most likely to consist solely of prime numbers first. The Bateman-Horn conjecture, first postulated at the University of Illinois during a summer research project in 1962, seeks to help quantify this likelihood in this and many other broad applications by generalizing the Prime Number Theorem's predictions regarding the distribution of prime numbers to the distribution of inputs which can make each of a system of polynomials output a prime number. Unlike the prime number theorem itself, the Bateman-Horn conjecture has a built in proportionality constant which takes into account the extent to which modular constraints on the inputs into these polynomials can affect the likelihood with which they all output prime values. As such, the Bateman-Horn Conjecture provides a stronger predictive basis with which to structure our search, because it allows us to account for the fact that that the variations in sieving across each of our programs make (n, D) pairs of comparable magnitude have different probabilities of representing an AP28. Given a system of k polynomials f_1, f_2, \dots, f_k , we can let $\pi_f(x)$ represent the number of natural numbers below x for which each f_i outputs a prime value. Then the Bateman-Horn Conjecture claims that $\pi_f(x)$ is asymptotically equivalent to $C \int_2^x \frac{dn}{\prod_{i=1}^k \ln(f_i(n))}$ as x tends to infinity, and where C is a constant whose derivation will

be discussed shortly [1]. It is worth noting that often in the literature this result will be

phrased slightly differently, because as x tends to infinity, the leading term begins to dominate each polynomial, further simplifying the denominator of the logarithmic integral to the form $\left(\prod_{i=1}^k \deg(f_i)\right) \cdot \ln(n)^k$, but for our purposes that additional simplification is not well justified, as we are considering the polynomials $f_1(n) = n, f_2(n) = n + D, \dots, f_{28}(n) = n + 27D$, and there are areas in the search space where it is not necessarily the case that n dominates the growth of, for example, $n + 27D$. While this conjecture remains unproven, it is not unreasonable to ascribe meaningful predictive power to it, as, for instance, it has been observed that its prediction of how many pairs of twin primes (defined by the polynomials $f_1(n) = n, f_2(n) = n + 2$) below some particular n do appear to approach the true value to a meaningful extent, at least up to the input $n = 10^9$ [1]. We can interpret this result to mean that the integrand, $f(n, D) = \frac{C}{\prod_{i=0}^{27} \ln(n_i D)}$, provides a heuristic approximation for the probability that an (n, D) pair of a given magnitude represent an AP28.

The constant C is meant to quantify the fact that depending on the structure of each individual polynomial, for any given prime p , the conditions that $p \mid f_1(n), p \mid f_2(n), \dots, p \mid f_k(n)$ are not necessarily independent events, for instance, in the twin prime case where $f_1(n) = n, f_2(n) = n + 2$, both numbers will have the same parity, and thus they are twice as likely to both be odd than a random choice of two numbers. In general this constant is given by the formula $C = \prod_p (1 - \frac{1}{p})^{-k} (\frac{1 - \omega_f(p)}{p})$, where $\omega_f(p)$ is defined as the number of modulo classes with respect to p will make any of the f_1, \dots, f_k output a multiple of p .

Because the only values in the (n, D) space the program considers are those which are generated as candidates, we can compute $\omega_f(p)$ for each prime in a way which reflects the sieving they have experienced. Since each of the programs we have created sieve by different primes for the candidate generation phase, they each have different particular values of C . In general, there are three cases for how we treat a given prime p when calculating this constant for various programs. If p is directly sieved for in candidate generation then it is guaranteed that none of $f_1(n) = n, f_2(n) = n + D, \dots, f_{28}(n) = n + 27D$ are divisible by p , and thus that $\omega_f(p) = 0$. If p is not sieved for, then there are two cases, either p is one of the small primes $7, 11, \dots, 23$ which D is necessarily divisible by, or one of the primes which the program makes no assurances as to D 's modulo class with respect to p . In the first case, since $D \equiv 0 \pmod p$, it holds that all of $n, n + D, \dots, n + 27D$ are equivalent mod p , and thus, the only case in which any of these values are $0 \pmod p$ is the case where $n \equiv 0 \pmod p$, hence, in this case $\omega_f(p) = 1$. Lastly, where D is not necessarily divisible by p , we cannot take one value of $\omega_f(p)$ which will hold for all (n, D) pairs within the search space, since pairs where $p \mid D$ will have a different $\omega_f(p)$ value than those where $p \nmid D$. As such, we instead take a probabilistic approach to get a weighted average for the $\omega_f(p)$ across all elements of the search space. $p \mid D$ with a probability of $\frac{1}{p}$, and in this case we have already demonstrated that $\omega_f(p) = 1$. On the other hand, $p \nmid D$ with probability $\frac{p-1}{p}$, and in this case, all of $0, D, 2D, \dots, 27D$ will be in different modulo classes with respect to p , and so there are 28 potential modulo classes n could fall in which would make one of the elements of the progression divisible by p , specifically $[0], [-D], [-2D], \dots, [-27D]$. Thus in this case $\omega_f(p) = 28$. This means the weighted average of the values of $\omega_f(p)$ for these sorts of primes is $(\frac{1}{p})(1) + (\frac{p-1}{p})(28) = \frac{28p-27}{p}$.

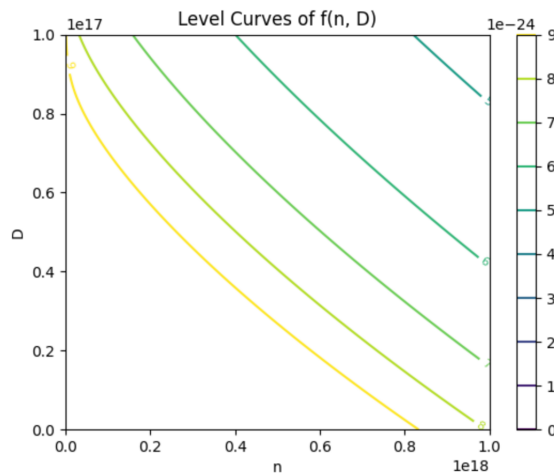
Applying these various values of $\omega_f(p)$ to the full formula for the value of C , we see that primes which are sieved in candidate generation contribute a factor of $(1 - \frac{1}{p})^{-28}$, those which are not sieved in candidate generation but by which D is necessarily divisible contribute a factor of $(1 - \frac{1}{p})^{-28}(1 - \frac{1}{p}) = (1 - \frac{1}{p})^{-27}$, and those by which the program imposes no condition on the divisibility of D by p and which are not sieved for in candidate generation contribute a factor of $(1 - \frac{1}{p})^{-28}(1 - \frac{28p-27}{p^2}) = (1 - \frac{1}{p})^{-28}(\frac{p^2-28p+27}{p^2})$. For the standard program which considers in its candidate generation phase the primes 2, 3, 5, 29, 31, \dots , 59, computing out the first 100,000 partial products of C gives us an approximate value of $4.9313 \cdot 10^{21}$. It is worth considering the extent to which this value quantifies the value of our sieving, for an arbitrary collection of 28 numbers, the probability that all of those numbers are odd is $(\frac{1}{2})^{28}$, but simply by sieving by 2, we force this probability to be 1, meaning the probability that our particular selection of 28 values are all prime increases by a factor of 2^{28} which is reflected in the fact that because $\omega_f(2) = 0$, the factor by which the 2 contributes to our Bateman Horn constant is $(1 - \frac{1}{2})^{-28} = 2^{28}$. We can see more generally that when we sieve by a given prime, it contributes to C by a factor of $(1 - \frac{1}{p})^{-28} = (\frac{p}{p-1})^{28}$, a significant improvement in the probability that our particular arithmetic progression contains only prime numbers over that of an arbitrary selection of 28 numbers. Thus, the vast magnitude of the ultimate value of C can be seen as a measure of how effectively our sieving culls the pool of candidates.

To determine the value of C for an arbitrary program, we recognize that for each additional prime we force D to be divisible by, we sieve by an additional prime in candidate generation. Letting p_1, p_2, \dots, p_k , represent the primes we incorporate into the common difference, and q_1, q_2, \dots, q_k represent the primes which receive no special treatment by the original program but by which we now sieve, we can then apply the formulae derived above to see the factor by which each program's C value varies from that of the standard version. For all p_i , where that prime originally contributed a factor of $(\frac{p_i}{p_i-1})^{28}$ to C , the new program does not sieve by it but does force D to be a multiple of it, so it now contributes a factor of $(1 - \frac{1}{p_i})^{-27} = (\frac{p_i}{p_i-1})^{27}$. Thus, for each p_i , the value of C will differ by a factor of $\frac{p_i-1}{p_i}$. For each q_i , in the standard program it contributed to the value of C by a factor of $(1 - \frac{1}{q_i})^{-28}(\frac{q_i^2-28q_i+27}{q_i^2})$ and in the adjusted program, is now sieved by, meaning it affects C by a factor of $(1 - \frac{1}{q_i})^{-28}$, thus the new program's value of C is changed by a factor of $\frac{q_i^2}{q_i^2-28q_i+27}$. Thus in general if C_0 is taken to be the standard program's value of C , then for a program which forces p_1, \dots, p_k to be divisible by D and additionally sieves by q_1, \dots, q_k , the new program will have $C = \prod_{i=1}^k \left((\frac{p_i-1}{p_i}) (\frac{q_i^2}{q_i^2-28q_i+27}) \right) C_0$. In practice, this means that the program where we force D to be a multiple of 29, and thus additionally sieve by 61, has a C value of $\frac{26047}{14790}C_0 \approx 1.76C_0$. The table below lists this value for all of the programs we have implemented in practice:

Additional Prime(s) Dividing D	Exact Factor By Which C Increases	Decimal Approximation
29	$\frac{26047}{14790}$	1.761
31	$\frac{3721}{2108}$	1.765
37	$\frac{11163}{6290}$	1.775
41	$\frac{3721}{2091}$	1.780
43	$\frac{26047}{14620}$	1.782
47	$\frac{85583}{47940}$	1.785
53	$\frac{48373}{27030}$	1.790
59	$\frac{107909}{60180}$	1.793
29, 31	$\frac{116924983}{40347120}$	2.898
31, 37	$\frac{50110707}{17159120}$	2.920

Where in each case our q_i 's are simply as many of the prime numbers after 59 as we have taken additional primes to be divisors of D , so in all of these programs $q_1 = 61$ and in the latter two $q_2 = 67$.

Recall that the motivation for the use of the Bateman-Horn conjecture was a desire to understand which regions of the (n, D) plane had greater likelihood of containing an arithmetic progression of 28 primes and thus should be considered first. With the values of C calculated, these regions can be visualized for each program by graphing the level sets of our approximate probability function, $f(n, D) = \frac{C}{\prod_{i=0}^{27} \ln(n_i D)}$. This visualization applied to the standard program can be seen below.



As these level sets represent the set of all (n, D) pairs corresponding to some particular probability, the region between the curves associated with some p_1, p_2 can be thought of as all of the (n, D) pairs whose probability of representing an arithmetic progression of 28 primes lies between those bounds. Using a nested for loop, we can then iterate through all of the candidates within a given region of high probability before moving on to the next,

slightly lower probability region, giving us a rough rule for how to navigate the search space so that we consider those (n, D) pairs which are most likely to describe an AP28 first. We can also check every (n, D) pair with probability above a certain bound for each individual program before moving on to the next lower probability region of the search space of each program collectively, allowing us to efficiently iterate through the full breadth of our search space across all of our programs.

6 Implementation Details

Both the non-divisible fixed difference and divisible fixed difference versions of our program were initially implemented in C++, building on a program originally implemented by Jarosław Wróblewski which implemented a version of sieving and candidate generation for AP26. In our initial versions, we modified the candidate generation and sieving processes to improve runtime by tailoring specifically it to the AP28 search, more aggressively weeding out all candidates that could not be part of an arithmetic progression of 28 primes. The main changes we made were to our initial version were to the candidate generation and sieving process, specifically altering the bounds of the loops in both candidate generation and sieving to start all loops 2 iterations later and end 2 iterations earlier. The rest of the specific implementation details are documented on [5]. This first series of changes alone accounted for a nearly 10 fold improvement in runtime compared to Wróblewski's original version. Even still, these initial versions were only capable of utilizing a single thread on the CPU to perform computations and as such were fairly slow.

Previous searches for APs had been organized by PrimeGrid, a group dedicated to finding large prime numbers. For these previous searches, PrimeGrid had used a version of Wróblewski's program that implemented support for both multi-threaded CPU and GPU as well as integrate with BOINC (Berkeley Open Infrastructure for Network Computing) by Bryan Little allowing for much faster computation and distribution of work across many contributors. To take advantage of this pool of compute resources, we ported the changes we made to the candidate generation algorithm for the non-divisible fixed difference version of our program to Little's program, but we did not make changes to the sieving algorithm. We opted to exclude this change from the final version of our program as changing sieving would result in a fairly marginal runtime improvement of roughly 30%, but come at the the cost of excluding around 80% of good candidates that could survive the sieving process. This would eliminate the vast majority of shorter AP's. Future work can be done to port the divisible fixed difference versions of the program to multi-threaded CPU and GPU.

7 Testing

Testing of our newfound programs has revealed generally positive results. Early in the project we created a program which can take some given arithmetic progression of less than 28 primes and tell us whether it is appropriate that it should survive our sieving and thus be found by our proper AP28 search functions. Testing every AP our programs have output, we have been able to debug each program to ensure that they never output or consider

AP's which do not pass the relevant sieving constraints, and using the wide body of AP's whose common differences match the divisibility constraints of the standard AP28 program which have already been found by PrimeGrid users, we were further able to confirm that we do not find any of the arithmetic progressions which we know to be out there but which should not be found by our program with stricter sieving, but that we do find every AP we know of within the search space which should be considered by the program. We have also run several tests to determine how quickly each program runs, and while the absolute times are a function of the particular CPU on which the code was run, the test results did indicate that all of the programs with additional divisibility criteria spend a comparable amount of time evaluating each candidate. With what little computational power we have put toward running the programs ourselves for testing purposes, we have already found several new arithmetic progressions of primes outside of PrimeGrid's previous search space, a small collection of the longest can be found in the table below.

First Element	Common Difference	Length
26911702440121027	$60 \cdot 29\#$	20
11276493292151	$70 \cdot 29\#$	20
330265802656381	$31\#$	18
31806309530495473	$64 \cdot 29\#$	18
69673227224784361	$31\#$	16
15828827085570437	$1113 \cdot 23\#$	16
4887250706118461	$1032 \cdot 23\#$	16
5792937732540241	$14 \cdot 29\#$	16
19630198732925293	$42 \cdot 29\#$	16
435194648963383	$44 \cdot 29\#$	16
265926834915143	$64 \cdot 29\#$	16

References

- [1] Soren Aletheia-Zomlefer, Lenny Fukshansky, Stephan Garcia. The Bateman-Horn Conjecture: Heuristics, History, and Applications. *Expositiones Mathematicae*, Vol. 38, Issue 4, pg 430–479, 2020.
- [2] Paul T. Bateman, Roger Horn. A Heuristic Asymptotic Formula Concerning the Distribution of Prime Numbers. *Mathematics of Computation*, Vol. 16, No. 79, pg 363–367, 1962.
- [3] Jarosław Wróblewski, How to search for 26 primes in arithmetic progression? 2008
- [4] Ben Green, Terence Tao. The Primes Contain Arbitrarily Long Arithmetic Progressions. *Annals of Mathematics*, Vol. 167, Issue 2, pg 481–547, 2008.
- [5] Ravi Fernando. “Ravifernando/AP28_IML: Source Code for the IML Project on Primes in Arithmetic Progression.” GitHub, GitHub, 2025, github.com/ravifernando/AP28_IML#.