LMS J. Comput. Math. 18 (1) (2015) 675–683
doi:10.1112/S1461157015000194

Two compact incremental prime sieves

Jonathan P. Sorenson

Abstract

A prime sieve is an algorithm that finds the primes up to a bound n. We say that a prime sieve is *incremental*, if it can quickly determine if $n+1$ is prime after having found all primes up to n. We say a sieve is *compact* if it uses roughly $\sqrt{n}$ space or less. In this paper, we present two new results.

- We describe the *rolling sieve*, a practical, incremental prime sieve that takes $O(n \log\log n)$ time and $O(\sqrt{n}\log n)$ bits of space.
- We also show how to modify the sieve of Atkin and Bernstein from 2004 to obtain a sieve that is simultaneously sublinear, compact, and incremental.

The second result solves an open problem given by Paul Pritchard in 1994.

1. *Introduction and definitions*

A *prime sieve* is an algorithm that finds all prime numbers up to a given bound n. The fastest known algorithms, including Pritchard's wheel sieve [16] and the Atkin–Bernstein sieve [1], can do this using at most $O(n/\log\log n)$ arithmetic operations. The easy-to-code sieve of Eratosthenes requires $O(n \log\log n)$ time, and there are a number of sieves in the literature that require linear time [17, 18].

Normally, running time is the main concern in algorithm design, but in this paper we are also interested in two other properties: *incrementality* and *compactness*.

We say that a sieve is *compact* if it uses at most $n^{1/2+o(1)}$ space. Bays and Hudson [4] showed how to segment the sieve of Eratosthenes so that only $O(\sqrt{n})$ bits of space are needed (see also [6, 20]). Pritchard [17] showed how to apply a fixed wheel to get a compact sieve that runs in linear time. Atkin and Bernstein [1] gave the first compact sieve that runs in sublinear time. For other sieves that address space issues, see, for example, [9, 11–13, 21, 22].

Loosely speaking, a sieve is *incremental* if it can determine the primality of $n+1$ after having found all primes up to n, using a small amount of additional work. Bengelloun [5] presented the first prime sieve that needed only a constant amount of additional work to do this, thereby taking a total of $O(n)$ time to find all primes up to n. Elaborating on comments in Bengelloun's paper, Pritchard [19] showed how to improve Bengelloun's sieve using a dynamic wheel so that in constant time it determines the the primality of all integers from n to $n+\Theta(\log\log n)$ (here, $g = \Theta(f)$ means there exist constants $0 < c_1 \leqslant c_2$ such that $c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)$ for all large n). In other words, it takes $O(1 + (p-n)/\log\log n)$ time to find p, if p is the smallest prime exceeding n. Pritchard's version of Bengelloun's sieve takes $O(n/\log\log n)$ time to find all primes up to n.

To clarify, let us define a sieve as $t(n)$-*incremental* if, in the worst case, it requires $(p-n)\cdot t(n) + O(1)$ operations to find $p = p_{\pi(n)+1}$, the smallest prime exceeding n, after having found all primes up to n. Thus, Bengelloun's sieve is $O(1)$-incremental, and Pritchard's improvements make it $O(1/\log\log n)$-incremental.

Received 22 April 2015; revised 14 July 2015.

2010 Mathematics Subject Classification 11Y16, 68Q25 (primary), 11Y11, 11A51 (secondary).

Supported by a grant from the Butler University Holcomb Awards Committee.

Open Problem 1. Design a prime sieve that is both compact and $o(1)$-incremental [19].

In this paper, we address the problem from two perspectives, both practical and theoretical.

- We present a sieve algorithm, with pseudocode, that finds the primes up to n using $O(n \log\log n)$ arithmetic operations and $O(\sqrt{n} \log n)$ bits of space, and that is $O(\log n/\log\log n)$-incremental. It is based on the segmented sieve of Eratosthenes, adapting some of the ideas in Bennion's sieve [11]. Our sieve uses a circular array of stack datastructures and works well in practice; we present timing results where it outperforms Bengelloun's sieve.
- We also prove the following theorem.

  Theorem 1.1. *There exists a prime sieve that is both compact and $O(1/\log\log n)$-incremental.*

  Our proof relies on modifying the sieve of Atkin and Bernstein [1].

After we discuss some preliminaries in § 2, we present our rolling sieve in § 3 and we prove our theorem in § 4. We conclude with some timing results in § 5.

2. *Preliminaries*

In this section, we discuss our model of computation, review some helpful estimates from elementary number theory, and review the sieve of Eratosthenes.

2.1. *Model of computation*

Our model of computation is a RAM with a potentially infinite, direct access memory.

If n is the input, then all arithmetic operations on integers of $O(\log n)$ bits have unit cost. This includes $+$, $-$, $\times$, and division with remainder. Comparisons, array indexing, assignment, branching, bit operations, and other basic operations are also assigned unit cost. Memory may be addressed at the bit level or at the word level, where each word has $O(\log n)$ bits.

Space is measured in bits. Thus, it is possible for an algorithm to touch n bits in only $O(n/\log n)$ time if memory is accessed at the word level. The space used by the output of a prime sieve, the list of primes up to n, is not counted against the algorithm.

This is the same model used in [9, 21, 22].

2.2. *Some number theory*

We make use of the following estimates. Here the sums over p are over primes only, and $x > 0$.

$$\pi(x) := \sum_{p \leqslant x} 1 = \frac{x}{\log x}(1 + o(1)), \tag{2.1}$$

$$\sum_{p \leqslant x} \frac{1}{p} = \log\log x + O(1), \tag{2.2}$$

$$\sum_{p \leqslant x} \log p = x(1 + o(1)). \tag{2.3}$$

For proofs, see Hardy and Wright [14].

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

2.3. *Sieve of Eratosthenes*

We present algorithms using a C++ style that should be familiar to most readers. We assume that integer variables can accurately hold any integer represented in standard signed binary notation in $O(\log n)$ bits. In practice, this means 64 bits.

Recall that the sieve of Eratosthenes uses a bit vector to represent the primes up to n. All are assumed to be prime (aside from 0 and 1) and, for each prime p found, its multiples q are enumerated and 'crossed off' as not prime, as shown by the following.

```
BitVector S(n+1); // bit vector for 0..n
S.setAll();
S[0]=0; S[1]=0;
for(p=2; p*p<=n; p=p+1)
  if(S[p]==1)  // so that p must be prime
    for(q=p*p; q<=n; q=q+p)
      S[q]=0;
```

This requires $\sum_{p\leqslant\sqrt{n}} n/p = O(n \log\log n)$ arithmetic operations (using (2.2)) and $O(n)$ bits of space.

2.4. *Segmented sieve of Eratosthenes*

The main loop of a segmented sieve is simply a loop over each segment. Here Δ is the size of that segment; we choose Δ to be proportional to $\sqrt{n}$. We assume here that the primes below $\sqrt{n}$ have already been found some other way (perhaps with the unsegmented sieve above).

```
for(left=sqrt(n)+1; left<=n; left=left+delta)
{
  right=min(left+delta-1,n);
  sieve(left,right);
}
```

Each segment is then sieved by crossing off multiples of each prime p below $\sqrt{n}$.

```
Primelist P(sqrt(right)); // list of primes <= sqrt(n)
BitVector S(delta); // bit vector for left..right
S.setAll();
for(i=0; i<P.length; i++)
{
  p=P[i];
  first=left+(p-(left%p))%p; // min. first>=left st. p|first
  for(q=first; q<=right; q=q+p)
    S[q-left]=0;
}
```

Using (2.2), the running time for a single segment is proportional to

$$\sum_{p\leqslant\sqrt{n}} \left(1+\frac{\Delta}{p}\right) = O\left(\Delta \log\log n + \frac{\sqrt{n}}{\log n}\right)$$

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

and summing this cost over n/Δ segments gives $O(n \log\log n)$. The total space used is $O(\sqrt{n})$ bits; this is dominated by space for the the list of primes up to $\sqrt{n}$ (using (2.3)), plus Δ bits for the segment bit vector.

Remarks 1.

- Both sieves can readily be modified to completely factor all integers up to n by replacing the bit vector with an array of lists, and adding p to the list for each q generated by the inner loop.
- Pritchard [17] added a *static wheel* to the segmented sieve to obtain an $O(n)$ running time. For a discussion of the static wheel, see [21, § 2.4], which gives C++-style pseudocode and a running time analysis.
- For a parallel version of the segmented sieve, see [23].

3. *Algorithm description*

In this section we present our rolling sieve. It is in many ways similar to the hopping sieve of Bennion [11].

The primary data structure is a circular array of stacks. Each array location corresponds to an integer n, and when n is reached by the sieve, its stack will contain the prime divisors of n up to $\sqrt{n}$. If the stack is empty, then either n is prime or the square of a prime. In the latter case, we discover a new prime by which to sieve: call this r. So we push r onto $n + r$'s stack.

When moving from n to $n+1$, we pop each prime p from ns stack and push it onto the stack for $n + p$, the next stack position corresponding to an integer divisible by p.

If the position for $n + p$ is larger than Δ, the size of the circular array, we simply wrap around to the front.

We are now ready to present pseudocode.

3.1. *Precomputation*

Let `start` be the value of the first integer we wish to test for primality. To set up, we the find the primes up to $\sqrt{\texttt{start}}$ and push them onto the correct stacks. Note that array position 0 corresponds to `start` as we begin.

```
r=floor(sqrt(start))+1;
s=r*r;
PrimeList P(r-1);
delta=r+2;
StackArray T(delta);
for(i=0; i<P.length; i=i+1)
{
  p=P[i];
  j=(p-(start%p))%p;
  T[j].push(p);
}
pos=0; n=start;
```

We have $\lfloor\sqrt{n}\rfloor = r - 1 = \Delta - 3$ so that $n < s = r^2$ and $r + 1 < \Delta$ hold.

3.2. *Primality of n*

Once the array of stacks is set up, we can check each successive integer n for primality, as shown in the function below, which returns true if n is prime, and false otherwise. It also sets its state to be ready to handle $n + 1$ on the subsequent call.

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

```
bool next()
{
  isPrime=true;
  while(!T[pos].isEmpty()) // process prime divisors
  {
    p=T[pos].pop();
    T[(pos+p)%delta].push(p);
    isPrime=false;
  }
  if(n==s) // then n is a square
  {
    if(isPrime) // then r is in fact prime
    {
      T[(pos+r)%delta].push(r);
      isPrime=false;
    }
    r=r+1; s=r*r;
  }
  n=n+1;
  pos=(pos+1)%delta;
  if(pos==0) { delta=delta+2; }
  return isPrime;
}
```

Remarks 2.

- The list of primes stored in stacks includes all primes up to $\sqrt{n}$, unless n is, in fact, the square of a prime. If this is the case, we detect that r is prime and put it into the appropriate stack. This is how we incrementally grow the list of primes up to $\sqrt{n}$ for sieving.
- We can grow Δ and the array of stacks over time by simply adding two to Δ when `pos` reaches zero. We make sure that no prime stored in the stacks exceeds Δ, or in other words, $r < \Delta$ is invariant. The new stacks added to the end of the array are initialized to empty. Note that if we start $\Delta = \lfloor\sqrt{n}\rfloor + 3$, and then after Δ function calls increment by 2, we end up iterating the mapping $(n, \Delta) \to (n + \Delta, \Delta + 2)$. Iterating k times gives $(n, \Delta) \Rightarrow (n + k\Delta + k(k-1), \Delta + 2k)$. Squaring $\Delta + 2k$ shows that the segment stays larger than $\sqrt{n}$ over time, and the $k(k-1)$ term ensures it will not exceed $O(\sqrt{n})$. Extending an array of stacks in the RAM model discussed in § 2 is straightforward, but in practice it is not so simple. One option is to use a fixed value for Δ, only use primes up to Δ in the stacks, and then numbers that seem prime should be prime tested as an extra step. See, for example, the ideas in [22].
- To find all primes up to a bound n, simply print the primes up to 100, say, then set up using `start` = 100, and repeatedly call the `next()` function, printing when it returns true.

```
int nextprime()
  { while(!next()); return n-1; }
```

- The sieve can readily be modified to generate integers in factored form. Simply make a copy of n, and, as the primes are popped from the stack, divide them into n. When the stack is empty, what remains is either one or prime. We leave the details to the reader.

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

An early version of this algorithm was used for this purpose in generating data published in [3].
- The array of stacks datastructure used here can easily be viewed as a special-case application of a *monotone priority queue*. In this view, each prime up to $\sqrt{n}$ is stored in the queue with its priority set to the next multiple of that prime. See, for example, [7].
- For references on number theoretic algorithms and their analysis, please see [2, 8, 15].

3.3. *Analysis*

THEOREM 3.1. *The rolling sieve will find all primes up to a bound n using $O(n \log\log n)$ arithmetic operations and $O(\sqrt{n} \log n)$ bits of space.*

Proof. The running time is bounded by the number of times each prime p up to $\sqrt{n}$ is popped and pushed. But that happens exactly once for each integer up to n that is divisible by p, or

$$\sum_{p \leqslant \sqrt{n}} \frac{n}{p} = O(n \log\log n).$$

Assuming a linked list style stack data structure, the total space used is one machine word for the number of stacks, $\Delta = O(\sqrt{n})$, plus the number of nodes, $\pi(\sqrt{n})$. At $O(\log n)$ bits per word, we have our result. □

THEOREM 3.2. *The rolling sieve is $O(\log n / \log\log n)$-incremental.*

Proof. It should be clear that we need to count the total number of prime divisors of the integers from n to p, where p is the smallest prime larger than n.

Let $\ell = p - n$. First, we consider primes $q \leqslant \ell$. Each such prime q divides $O(\ell/q)$ integers in this interval, for a total of $O(\ell \log\log \ell)$.

Next we consider primes $q > \ell$. Each such prime can divide at most one integer between n and p. Summing up the logarithms of such primes,

$$\sum_{m=n}^{p} \sum_{q \mid m, q > \ell} \log q = O(\ell \log n).$$

The number of such primes q is maximized when the primes are as small as possible. Again, each prime can only appear once in the sum above, so we solve

$$\sum_{\ell < q \leqslant x} \log q = O(\ell \log n)$$

for x. We have $x = O(\ell \log n)$ by (2.3), and so the number of primes is bounded by $\pi(x) = O(\ell \log n / (\log \ell + \log\log n))$.

Dividing through by ℓ completes the proof. □

REMARKS 3.
- For the purposes of analysis, and to limit the space used, our stacks are linked-list based. For speed, array-based stacks are better. For 64-bit integers, an array of length 16 is sufficient.
- Although we can only prove the rolling sieve is $O(\log n / \log\log n)$-incremental, by the Erdös–Kac theorem, in practice it will behave as if it is $O(\log\log n)$-incremental. See, for example, [10].

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

– One could add a wheel to reduce the overall running time by a factor proportional to $\log\log n$, and similarly improve incremental behavior. We have not tried to code this, and growing the wheel by adding even a single prime, in an incremental way, seems messy at best. From a theoretical perspective, the question is moot as we will see in the next section.

4. *The theoretical solution*

In this section we prove Theorem 1.1.

Our proof makes use of the sieve of Atkin and Bernstein [**1**]. In particular, we make use of the following properties of this algorithm.

– The sieve works on segments of size Δ, where $\Delta \approx \sqrt{n}$, and has a main loop similar to the segmented sieve of Eratosthenes discussed in § 2.4.
– The time to determine the primality of all integers in the interval from n to $n + \Delta$ is $O(\Delta/\log\log n)$ arithmetic operations.
– Each interval can be handled separately, with at most $O(\Delta)$ extra space of data passed from one interval to the next.

Any sieve that had these properties can be used as a basis for our proof.

Proof. We maintain two consecutive intervals of information.

The first interval is a simple bit vector and list of primes in the interval n to $n + \Delta$. Calls to the `next()` function are answered using data from this interval in constant time. We can also handle a `nextprime()` function that jumps ahead to the next prime in this interval in constant time using the list.

The second interval, for $n + \Delta$ to $n + 2\Delta$, is in process; it is being worked on by the Atkin–Bernstein sieve. We maintain information that allows us to start and stop the sieve computation after any fixed number of instructions. When this interval is finished, it creates a bit vector and list of primes ready for use as the new first interval.

So, when a call to the `next()` function occurs, after computing its answer from the first interval, a constant amount of additional work time is invested sieving the second interval; enough that after Δ calls to `next()`, the second interval will be completed.

When a call to the `nextprime()` function occurs, we compute the distance to that next prime, ℓ, and invest $O(1 + \ell/\log\log n)$ time sieving the second interval after computing the function value.

In this way, by the time the first interval has been used up, the second interval has been completely processed, and we can replace the first interval with the second, and start up the next one. □

Remarks 4.

– The proof is a bit artificial and of theoretical interest only. Coding this in practice seems daunting.
– Any segmented sieve, more or less, can be used to make this proof work. In particular, if a compact sieve that is faster, or uses less space, can be found, such a sieve immediately implies a version that is also incremental.
– Pritchard has also pointed out that *additive* sieves (ones that avoid the use of multiplication and division operations) are desirable due to the fact that hardware addition is much faster than multiplication or division. From a theoretical perspective, this is moot since a multiplication table together with finite-precision multiplication and division routines can reduce all arithmetic operations on $O(\log n)$-bit integers to additions.
  That said, see [**16**, **19**].

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

5. *Timing results*

We implemented the Sieve of Eratosthenes, Bengelloun's sieve, the segmented Sieve of Eratosthenes, and our new rolling sieve. Each algorithm found all primes up to n for $n = 2^{20}, 2^{22}, \ldots, 2^{30}$, and the timing results, in seconds, are shown in Table 1.

The code used to obtain these timings is available from the journal website, and matches pseudocode presented here and in [5] reasonably well.

We used the GNU `g++` compiler with `-O2` optimization. The STL `vector` class was used for dynamically growing arrays for all four algorithms. The platform was a Linux server with Intel Xeon E5540 cpus rated at 2.53 GHz with 8192 kb cache.

Remarks 5.

- We also ran two other, highly optimized prime sieve implementations that are freely available online.
  The first was Dan Bernstein's code for the Atkin–Bernstein sieve [1]. Using the `primespeed` program from his `primegen-0.97` package (available from `cr.yp.to`), his code found all primes up to 2^{30} in about a second. This is roughly 10 times faster than our segmented sieve of Eratosthenes code. Dan's code is written in C instead of C++, does not use the `vector` class for arrays, and uses a wheel.
  The second was Kim Walisch's implementation of the segmented sieve of Eratosthenes [24]. It also uses a wheel, is highly optimized, and seems to be a fair bit faster than Bernstein's code. Kim's code, `primesieve-5.4.2`, found all primes up to 2^{30} in under a half second on our platform.
  If the goal is to find all primes up to n in a non-incremental way, using one of these implementations or something similar is surely the way to go.
- The incremental sieves are significantly slower than the two versions of the sieve of Eratosthenes. This can largely be blamed on the overhead to call a function for each number up to n.
- The referee noted that the rolling sieve seems to be about four or five times slower than the segmented sieve of Eratosthenes, and this relationship may be platform independent. Observe that the innermost loop of the sieve of Eratosthenes is basically an array indexing operation, a bit flip, an addition, and a comparison (one can optimize out the subtraction). The equivalent work in the rolling sieve requires, in addition, both push and pop stack operations, with their associated additions and array indexing operations, and possible function call overhead. This probably is the major cause of the four or five factor increase in running time.
- We chose not to implement fixed wheels for these four algorithms, but it should be fairly straightfoward to do so if desired. See, for example, [21, Algorithms 2.3 and 2.5].
- The segmented sieve of Eratosthenes and the rolling sieve can both find all primes in an interval $[n, n + \Delta]$ for n as large as 2^{50}, say. We are quite confident that the rolling sieve would not win a timing contest on a problem of this type.

Table 1. *Prime sieve running times, in seconds.*

$n =$	2^{20}	2^{22}	2^{24}	2^{26}	2^{28}	2^{30}
Eratosthenes	0.01	0.02	0.09	0.46	2.65	12.02
Bengelloun	0.04	0.21	0.93	3.96	15.28	63.91
Segmented Eratosthenes	0.01	0.06	0.20	0.70	2.70	10.18
Rolling	0.04	0.16	0.64	2.67	10.96	44.73

https://doi.org/10.1112/S1461157015000194 Published online by Cambridge University Press

Acknowledgement. Thanks to the referee who provided helpful comments that improved this paper.

References

1. A. O. L. Atkin and D. J. Bernstein, 'Prime sieves using binary quadratic forms', *Math. Comp.* 73 (2004) 1023–1030.
2. E. Bach and J. O. Shallit, *Algorithmic number theory*, vol. 1 (MIT Press, 1996).
3. E. Bach and J. P. Sorenson, 'Approximately counting semismooth integers', *Proceedings of the 38th International Symposium on Symbolic and Algebraic Computation*, ISSAC '13 (ACM, New York, NY, 2013) 23–30.
4. C. Bays and R. Hudson, 'The segmented sieve of Eratosthenes and primes in arithmetic progressions to 10^{12}', *BIT* 17 (1977) 121–127.
5. S. Bengelloun, 'An incremental primal sieve', *Acta Inform.* 23 (1986) no. 2, 119–125.
6. R. P. Brent, 'The first occurrence of large gaps between successive primes', *Math. Comp.* 27 (1973) no. 124, 959–963.
7. B. V. Cherkassky, A. V. Goldberg and C. Silverstein, 'Buckets, heaps, lists, and monotone priority queues', *SIAM J. Comput.* 28 (1999) no. 4, 1326–1346, doi:10.1137/S0097539796313490.
8. R. Crandall and C. Pomerance, *Prime numbers, a computational perspective* (Springer, New York, 2001).
9. B. Dunten, J. Jones and J. P. Sorenson, 'A space-efficient fast prime number sieve', *Inform. Process. Lett.* 59 (1996) 79–84.
10. P. Erdös and M. Kac, 'The Gaussian law of errors in the theory of additive number theoretic functions', *Amer. J. Math.* 62 (1940) no. 1, 738–742.
11. W. F. Galway, 'Robert Bennion's 'hopping sieve'', *Proceedings of the Third International Symposium on Algorithmic Number Theory*, ANTS-III (Springer, Berlin, Heidelberg, 1998) 169–178.
12. W. F. Galway, 'Dissecting a sieve to cut its need for space', *Algorithmic number theory*, Leiden, 2000, Lecture Notes in Computer Science 1838 (Springer, Berlin, 2000) 297–312.
13. W. F. Galway, 'Analytic computation of the prime-counting function'. PhD Thesis, University of Illinois at Urbana-Champaign, 2004, available at http://www.math.uiuc.edu/∼galway/PhD_Thesis/.
14. G. H. Hardy and E. M. Wright, *An introduction to the theory of numbers*, 5th edn (Oxford University Press, 1979).
15. D. E. Knuth, *The art of computer programming: seminumerical algorithms*, vol. 2, 3rd edn (Addison-Wesley, Reading, MA, 1998).
16. P. Pritchard, 'A sublinear additive sieve for finding prime numbers', *Commun. ACM* 24 (1981) no. 1, 18–23; 772.
17. P. Pritchard, 'Fast compact prime number sieves (among others)', *J. Algorithms* 4 (1983) 332–344.
18. P. Pritchard, 'Linear prime-number sieves: A family tree', *Sci. Comput. Program.* 9 (1987) 17–35.
19. P. Pritchard, 'Improved incremental prime number sieves', *First International Algorithmic Number Theory Symposium* (ANTS-1), Lecture Notes in Computer Sciences 877 (eds L. M. Adleman and M.-D. Huang; Springer, Berlin, Heidelberg, 1994) 280–288.
20. R. C. Singleton, 'Algorithm 357: An efficient prime number generator [a1]', *Commun. ACM* 12 (1969) no. 10, 563–564.
21. J. P. Sorenson, 'Trading time for space in prime number sieves', *Proceedings of the Third International Algorithmic Number Theory Symposium* (ANTS III), Lecture Notes in Computer Sciences 1423 (ed. J. Buhler; Springer, Berlin, Heidelberg, 1998) 179–195.
22. J. P. Sorenson, 'The pseudosquares prime sieve', *Proceedings of the 7th International Symposium on Algorithmic Number Theory* (ANTS-VII), Lecture Notes in Computer Sciences 4076 (eds F. Hess, S. Pauli and M. Pohst; Springer, Berlin, Germany, 2006) 193–207; ISBN 3-540-36075-1.
23. J. P. Sorenson and I. Parberry, 'Two fast parallel prime number sieves', *Inf. Comput.* 144 (1994) no. 1, 115–130.
24. K. Walisch, 'primesieve: Fast c/c++ prime number generator', http://primesieve.org, 2015.

Jonathan P. Sorenson
Computer Science and Software Engineering
Butler University
Indianapolis, IN 46208
USA
sorenson@butler.edu