The Sieve of Zakiya

by
Jabari Zakiya © 2008

Introduction

In [1] I introduced a method to generate all the primes up to N using a specific class of prime generator functions. I showed how I developed these prime generators and how I developed software code to implement a new prime sieve algorithm using them. Benchmarks of this sieve algorithm against the Sieve of Eratosthenes (SoE) showed this method, the Sieve of Zakiya (SoZ), was always faster for any prime generator used, and was faster than the Sieve of Atkin (SoA) for certain generators. In this paper I expand upon those results in [1] and present new findings in a comprehensive and structured manner.

Math Notation and Definitions

SoZ Prime Generators (PG) have form $P_n = mk+(1,r_i)$, k = 0..t. If n is a prime (e.g. P_3) it denotes the order p_n of the generator, i.e. it generates primes > n. An even n (e.g. P_{30}) signifies a PG's modulus, and these PG can generate primes ≥ the first r_i and their order p_n is the first prime less than r_1 in $(1,r_i)$. The value m (a positive even integer) is the modulus. All primes N greater than a PG's order must satisfy (N mod m) ε $\{1,r_i\}$ where the r_i are odd integer residues $p_n < r_i < m$. Alternatively to be prime $(N \pm \varepsilon \{1,r_i\}) \bmod m = 0$. The set $\{1,r_i\}$ has an even number of members and thus an odd number of r_i residues ($1= r_0$), and r_1 is the first prime > p_n and the largest/last $r_i = m-1$.

I classify PGs as being strictly prime (SP) or non-SP. SP PGs moduli have the form P!(n) = 2*3*5...n, i.e. $P!(n) = \prod p_i$ is the factorial of the primes up to n, and their total residues are: $res = \#\{1,r_i\} = \prod(p_i-1)$.

In my original paper [1] (Ultimate Sieve Prime - Sieve of Zakiya) I identified how to generate the PGs. I identified and tested 5 SP PGs: P_3, P_5, P_7, P_{11}, P_{13} and 4 non-SP PGs: P_{60}, P_{110}, P_{150}, P_{180}.

A very interesting property of the set $\{1,r_i\}$ is the members form modular complement (mc) pairs (mcp) e.g. $m = s_i + s_j$ where $s \varepsilon \{1,r_i\}$ and $\sum r_i = m*res/2$, for i=0..(res-1), where $res = \#\{1,r_i\}$ is total residues.

The modular complement pairs come in 3 Forms:
1) a prime r_i is a modular complement to another prime r_j or 1.
2) a non-prime r_i is a modular complement to a prime r_j or 1.
3) a non-prime r_i is a modular complement to another non-prime r_j.

Group 1 PGs have mc of only Form 1, Group 2 PGs have mc of Form 1 and 2, and Group 3 has all 3. From above, Group1 only has P_3 and P_5, Group 2 includes P_7, P_{60}, P_{90}, P_{110}, P_{120}, P_{150}, P_{180}, and Group 3 contains P_{11} and P_{13}. All PGs with moduli > 210 (P_7) will have mcp of Group3.

The Sieve of Zakiya (SoZ)

1) First create an array (list) of prime candidate values.
Initialize an array of length N/2 to false (nil), whose values represent the odd numbers ≤ N (or a range). Then generate a list of the prime candidates (pc) up to N (or over a range) using $P_n = mk+(1,r_i)$, k = 0..t. Set the array to true (or these values) for the index values equal to the pc values that are ≤ N. Then set the array to true at the index values for the primes $p_i < m$.

2) Perform a sieve process to eliminate the non-primes.
Identify all the non-primes in the array using $P'_n = p_j*[mk+(r_i, m+1)] = (p_j*m)k + p_j*(r_i, m+1)$, k = 0..t, where the p_js are the primes $p_n < p_j \le \text{sqrt}(N)$ and $p_1 = r_1$ for each PG.

The following is an illustration of the SoZ using P_5 = 30k+ (1, 7, 11, 13, 17, 19, 23, 29).

First create a list of prime candidates where: n1 = 30k+1, n2 = 30k+7, ... n8 = 30k+29 for k = 1..t

Table 1.

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
n1	1	31	61	91	121	151	181	211	241	271	301	331	361	391	421	451	481	511
n2	7	37	67	97	127	157	187	217	247	277	307	337	367	397	427	457	487	517
n3	11	41	71	101	131	161	191	221	251	281	311	341	371	401	431	461	491	521
n4	13	43	73	103	133	163	193	223	253	283	313	343	373	403	433	463	493	523
n5	17	47	77	107	137	167	197	227	257	287	317	347	377	407	437	467	497	527
n6	19	49	79	109	139	169	199	229	259	289	319	349	379	409	439	469	499	529
n7	23	53	83	113	143	173	203	233	263	293	323	353	383	413	443	473	503	533
n8	29	59	89	119	149	179	209	239	269	299	329	359	389	419	449	479	509	539

Now generate the non-primes by P'_5 = $(p_j*30)k + p_j*(7, 11, 13, 17, 19, 23, 29, 31)$, for k = 0..t, and

$$n1_k = (p_j*30)k+p_j*7, \quad n2_k = (p_j*30)k+p_j*11, \quad \ldots \quad n8_k = (p_j*30)k+p_j*31$$

Shaded values in Table 1 are the non-primes shown in Table 2.

Table 2.

p_j	$n1_0$	$n2_0$	$n3_0$	$n4_0$	$n5_0$	$n6_0$	$n7_0$	$n8_0$	$n1_1$	$n2_1$	$n3_1$	$n4_1$	$n5_1$	$n6_1$	$n7_1$	$n8_1$	$n1_2$
7	49	77	91	119	133	161	203	217	259	287	301	329	343	371	413	427	469
11	77	121	143	187	209	253	319	341	407	451	473	517	539	583	649	671	737
13	91	143	169	221	247	299	377	403	481	533	559	611	637	689	767	793	871
17	119	187	221	289	323	391	493	527	629	697	731	799	833	901	1003	1037	1139
19	133	209	247	323	361	437	551	589	703	779	817	893	931	1007	1121	1159	1273
23	161	253	299	391	437	529	667	713	851	943	989	1081	1127	1219	1357	1403	1541
29	203	319	377	493	551	667	841	899	1073	1189	1247	1363	1421	1537	1711	1769	1943
31	217	341	403	527	589	713	899	961	1147	1271	1333	1457	1519	1643	1829	1891	2077
37	259	407	481	629	703	851	1073	1147	1369	1517	1591	1739	1813	1961	2183	2257	2479
41	287	451	533	697	779	943	1189	1271	1517	1681	1763	1927	2009	2173	2419	2501	2747

The rows are the primes $p_n < p_j \leq \text{sqrt}(N)$, the columns are the residues [mk+(r_i, m+1)], and the table entries are their products. The first product (row1:col1) is always $p_1^2 = r_1^2$. The subsequent ith row/col products along the diagonal equal p_i^2 until the first ni_k column value becomes non-prime. For P'_5 this becomes first true when k=1, but for larger PGs with non-prime r_i this becomes true for k=0. All the products to the left of the diagonal values are previously computed values. This symmetry can be used to perform an optimized sieve process that becomes more efficient as N gets bigger.

Table 3 shows the progression of the diagonal products factors for Table 2, which is similar for all PGs.

Table 3.

ni_k	7	11	13	17	19	23	29	31	37	41	43	47	49	53	59	61	67	71	73	77	79	83
p_i	7	11	13	17	19	23	29	31	37	41	43	47	53	59	61	67	71	73	79	83	89	97

Using N = 500 as an example, the prime candidates (pc) ≤ 500 will be generated for k=1..16 in Table 1. From Table 2, all the values ≤ 500 generated by the primes < sqrt(500) = 22, i.e. (7, 11, 13, 17, 19) are eliminated from the pc list in Table 1 (the shaded values). Thus, the non-shaded Table 1 values ≤ 500 for k = 1..16, plus the primes < m=30, (2, 3, 5, 7, 11, 13, 17, 19, 23, 29), are the primes ≤ 500.

So from Table 1, the number of pc ≤ 500 are (16*8)-2 = 126, minus 41non-primes, leaving 85 primes, plus the 10 primes < 30, gives Pi(500) = 95.

Coding

The SoZ PGs only use odd integers. Thus, the natural odd numbers can be converted into sequential array index values by i = (n-3)/2, where n is a natural odd value, i its converted value and / is integer division, and thus n = 2i+3. So the indices map to integers i→n, as 0→3, 1→5, 2→7, 3→9, etc.

1) Identify the prime candidates.

For languages with static arrays, initialize a sieve array to "false" that is (N+m) elements long. The last index value for the sieve values will be lndx = (N-1)/2, so using the converted values, the sieve array will have length (lndx + (m/2)). Then create variables n1..nk for the PG's residues for their converted values, where n1=-1 (1→-1). Then, while the largest ni < lndx, increment each ni by m/2, then set the array to "true" (or the index value) for these index values. When finished, set the sieve array to "true" for the indices equal to the converted primes < m (i.e p_i →i; 3→0, 5→1, 7→2, 11→4, etc.). This completes the process to identify and load the sieve array with all the prime candidates ≤ N.

2) Perform the SoZ sieve.

Set a variable n =0 (used to count the number of primes ≤ sqrt(N) the sieve will go through). Within an outer loop, starting at the value for r_1 and iterating up to the sqrt(N), for each loop value i, if sieve[i] not true (i.e not prime) get the next outer loop value. If sieve[i] true, create variables that equal all the res number products pj and compute p1= i*r_1 ....pn =i*(m+1) and k = i*m. Then compute x=k*n/res, then increment n. Add x to all the p1....pn products. (This will skip computing the values to the left of the diagonal in Table 2, and begin each res*s, s=0..t primes group at the start of each i*[r_1, (m+1)] group. If this step is omitted, the values to the left of the Table 2 diagonal will be recomputed.) Then, while pn < lndx, set the sieve array at each pj value to "false", increment each pj by k, and repeat. When pn becomes > lndx, starting at p1, test to see if pj < lndx, and set sieve[pj]= false if so. Perform this test for the remaining pj values. This ends the sieve for the outer loop value i. Get next i and repeat till outer loop finished. The sieve array indices now represent the converted primes from 3..N where their values are "true". Converting these indices to natural numbers, and appending to 2, gives Pi(N).

At the website listed in [1] code to perform various SoZ PGs are available in Ruby, Python, and Forth, including benchmarks to compare performances against the Sieve of Eratosthenes (SoE) and Sieve of Atkin (SoA). A generalized routine (sozPn) is included in the Ruby code, which can perform the SoZ for all Group 1,2 PGs and P_{11}, and when supplied an array of Form 3 mcp values, any Group 3 PGs (to the limitations of the system's memory). Below is Ruby code for an implementation of sozP_3.

```
Code 1.
class Integer
   def sozP3
      # all prime candidates > 3 are of form 6*k+(1,5)
      # initialize sieve array with only these candidate values
      # where sieve contains the odd integers representatives
      # convert integers to array indices/vals by  i = (n-3)/2
      n1, n2 = -1, 1;  lndx = (self-1)/2;  sieve = []
      while n2 < lndx
        n1 +=3;  n2 += 3;  sieve[n1] = n1;  sieve[n2] = n2
      end
      #now initialize sieve array with (odd) primes < 6, resize array
      sieve[0]=0;  sieve[1]=1;  sieve = sieve[0..lndx-1]

      n = 0   # primes count
      1.step((Math.sqrt(self).to_i - 3)/2,1) do |i|
        next unless sieve[i]
        # j  i  5i  7i  6i  p1=5i,  p2=7i,  mp=6i
        # 5->1  11  16  15
        j = (2*i)+3;  p1 = (2*j)+i;  p2 = p1+j;  mp = p2-i
        x = mp*(n/2); n += 1   # x = mp*(n/res)  res = 2
        p1 += x;  p2 += x
        while p2 < lndx
          sieve[p1] = nil;  sieve[p2] = nil;  p1 += mp;  p2 += mp
        end
        sieve[p1] = nil if p1 < lndx
      end
      return [2] if self < 3
      [2] + sieve.compact.map {|i| (2*i)+3 }
   end
end
```

Partitioning over a Range

Instead of finding all the primes up to N, the SoZ can be used to find primes over a range as similarly done with SoE partitioning methods. Prichard [3] and Tomas Oliveira e Silva [4] have constructed methods based on the SoE to perform sieving over a range. There is nothing inherent in their methodology that would prohibit the use of SoZ PGs instead of the SoE, in fact, they could make them much faster.

Parallel Processing

Both the prime candidates generation process $P_n = mk+(1,r_i)$ and the sieve $P'_n = p_j*[mk+(r_i, m+1)]$ can inherently be done in parallel. Since the residues act as independent vectors into the number space they can be used simultaneously while acting on the same array. A minimally fully optimized parallel implementation would use as many processing threads/cores as a PG's total residues. More optimized versions could divide the number space up to N into ranges, for some/all residues, for increased speed.

Prime Generators

For SP (strictly prime) PGs the number of total residues are: $res = \#\{1,r_i\} = \prod(p_i-1)$.

Example: for P_{13}, $res = \#\{1,r_i\} = \prod(p_i-1) = (2-1)(3-1)(5-1)(7-1)(11-1)(13-1) = 1*2*4*6*10*12 = 5760$.

SP PGs P_3 and P_5 have Group 1 mcp, P_7 has Group 2 mcp, and SP PGs $\geq P_{11}$ have Group 3 mcp. SP PGs also have increasing orders and increasing modulus/res ratios. Since the release of my paper [1], I have verified P_{17}. My system's memory limitations effectively prohibit verification of larger SP PGs.

P_{17} = 510510k + (1,[92159 r_i residue values: 42324 primes, 23339 Form 2 mc, 26496 Form 3 mc])

Apparently, an infinite number of PGs with moduli that are multiples of 30 exist. Those from 30 thru 180 can be constructed by successively adding 30*k, k=1..t, to the base residues of $P_{5/30}$.

$P_{5/30}$ = 30k + (1, 7, 11, 13, 17, 19, 23, 29)
P_{60} = 60k + (1, 7...29) + (31, 37, 41, 43, 47, 49, 53, 59)
P_{90} = 90k + (1, 7...59) + (61, 67, 71, 73, 77, 79, 83, 89)
P_{120} = 120k + (1, 7...89) + (91, 97, 101, 103, 107, 109, 113, 119)
P_{150} = 150k + (1, 7...119) + (121, 127, 131, 133, 137, 139, 143, 149)
P_{180} = 180k + (1, 7...149) + (151, 157, 161, 163, 167, 169, 173, 179)

Note, all these PGs have order $p_n = 5$ ($r_1 = 7$), and the same modulus/res ratio (15/4), and have mcp of Group 2. Also, their modulus values are all products of only powers of 2, 3, and 5.

$30 = 2^1*3^1*5^1$; $60 = 2^2*3^1*5^1$; $90 = 2^1*3^2*5^1$; $120 = 2^3*3^1*5^1$; $150 = 2^1*3^1*5^2$; $180 = 2^2*3^2*5^1$

Though 210 is a multiple of 30, it's a SP modulus (210 = 2*3*5*7) and thus $p_n = 7$ ($r_1 = 11$). It has mcp of Group 2, and its modulus/res ratio (210/48 = 35/8) is also higher than the lower moduli values.

Valid PGs for multiples of 30 moduli ≥ 240 all have mcp of Group 3, and will have m/res ratios 15/4, except for *special* multiples. For m=2*210 =420 and m=6*210=1260, 420/96=1260/288 = 35/8, the same for m=210. The largest valid multiple of 30 modulus I verified was m/res = 30000/8000 = 15/4.

Of the PGs verified to date, P_{110} is the oddity. With $p_n = 2$ ($r_1 = 3$), it has the lowest modulus/res ratio (110/42 = 55/21) and is the slowest, and has Group 2 mcp.

Efficiency

The efficiency of a prime generator, ef(P_n), attempts to quantify how well a PG can find primes over a span of m numbers using the number res = #{1,r_i} residues. As the ratio m/res becomes bigger a PG will be able to find more primes faster. For SP PGs this becomes $\ln(\prod p_j / \prod(p_j-1))$ for $p_j \geq 3$.

Table 4.

P_n	P_3	P_5	P_{60}	P_{110}	P_{150}	P_{180}	P_7	P_{11}	P_{13}	P_{17}
modulus	6	30	60	110	150	180	210	2310	30030	510510
residues	2	8	16	42	40	48	48	480	5760	92160
ef(P_n) = ln(m/res)	1.099	1.322	1.322	0.963	1.322	1.322	1.476	1.571	1.651	1.712

Complexity Analysis

[2] states the SoA computes primes up to N using O(N/log log N) operations, while the SoE is O(N).

An efficient SoE considers only the odd numbers ≤ N as prime candidates, and iterates over the odd primes ≤ sqrt(N) to eliminate their odd valued multiples starting at p_i^2, which identify $[(N- p_i^2)/2p_i + 1]$ multiples for each prime.

Example N=100: the primes ≤10, are 3, 5, 7; there are 16 multiples for 3→(9, 15, 21, 27, ...99); and 8 multiples for 5→(25, 35, 45, ...95); and 4 multiples for 7→(49, 63, 77, 91). Thus 16+8+4 = 28, minus 2 duplicates, identify 26 odd non-primes ≤100. Thus pi(100) = 50-26 odd + 1 even (2) = 25 primes.

The SoZ identifies at most res*(N/m) prime candidates for each PG. It then iterates through the primes p_i where $p_n < p_i \leq \text{sqrt}(N)$ and $p_1 = r_1$ for each PG. Each p_i spans at most $(N- p_i^2)/mp_i$ groups of residues from p_i^2 to N, which are the p_i multiples to the right of the diagonal in Table 2, which makes the approx max number of multiples for each p_i over this span: $res*(N- p_i^2)/mp_i$. The number of p_i multiples up to p_i^2 in Table 2 is simply: $res*p_i/m$. Here (p_i/m) is the percentage of residue groups spanned by p_i, and then $res*p_i/m$ is the number of residue terms. Thus, the total number of multiples < N for each p_i is:

(1) $$res*(N- p_i^2)/mp_i + res*p_i/m = res*N/mp_i$$

This represents the number of multiples for the non-optimized sieve, starting at column 1 in Table 2. Example N=1000: m=30, res =8, thus for p_i =19 there are 8*1000/(30*19) = 14 multiples < 1000, and from Table 2 for p_i =19 it shows there are, in fact, 13 multiple of 19 < 1000.

Therefore, the maximum number of all prime multiples for $p_n < p_i \leq \text{sqrt}(N)$ is:

(2) $$\sum res*N/mp_i = res*N/m \sum 1/p_i$$

Then by Mertens Theorem: $\sum 1/p_i \simeq \log \log n$ where the p_i are all the primes up to $n = N^{1/2}$. This will be a bigger upper bound than necessary in this case since (2) only sums the primes $p_n < p_i \leq N^{1/2}$. So the total number of prime multiples (non-primes) eliminated by the sieve will be:

(3) $$res*N/m \sum 1/p_i \simeq (res*N/m)(\log \log N^{1/2}) - E(N)$$

where E(N) is an error term involving the prime sums for $p_i \leq p_n$.

A more optimized sieve can be performed by determining which kth residue group each diagonal p_i starts in and skipping over the preceding k residue group computations for that. So if p_i starts in the diagonal group for k=3, subtract all the k*res = 3*res duplicates for the 3 residue groups for k=0,1,2 from the left of the diagonal values in Table 2. For the residue group p_i starts in, it will compute all those res multiples. For a (res x res) matrix, this will resort in res*(res-1)/2 duplicate multiples being produced (left half matrix values) just once for all the res number p_i that span that residue group.

From Table 2, where res = 8, the first 8 sieve primes will produce 8*(8-1)/2 = 28 duplicates, which are the values to the left of the diagonal for the first 8x8 (p_i*r_i) values. Then starting with p_9=37, the next 8 sieve primes will skip the first 8x8 group k=0 multiples, and produce the multiples for k=1, etc.

Though it may seem algorithmically sub-optimal to generate these duplicate products it is actually computationally optimal for coding. All the ni_k products for p_i for each kth residue group have to be produced anyway, as they form the base residue group successive values of mp_i are added to generate the ni_k for the higher k residue groups. In Code 1 this is the purpose of x = mp*n/res, which calculates the diagonal kth residue group each p_i starts in. However, after the ni_0 products and the kth base group $x=kmp_i$ are computed, all the other products are computed as just the additions of the value mp_i to the preceding kth residue group values. No multiplications are necessary.

k_{max} equals the integer division of the sieve primes $p_n < p_i \leq N^{1/2}$ by res, i.e. $k_{max} = [Pi(N^{1/2})-Pi(p_n)-1]/res$. Each kth group of res number primes will skip over $k*(res * res) = k*res^2$ duplicates. Thus, for k=0, no res^2 duplicates will be skipped; for k=1, 1 res^2 group will be skipped; then 2, 3, etc. Thus, the total res^2 groups skipped are $\sum k = 1+2+...+k_{max} = k_{max}*(k_{max}+1)/2$ and therefore the maximum number of total duplicates skipped is $res^2 *(k_{max}*(k_{max}+1)/2)$.

Thus, an optimized reduced number of sieve multiples shown in Table 2 can be approximated as (4):

(4) $$res*N/m \sum 1/p_i - res^2 *(k_{max}*(k_{max}+1)/2)$$

and thus from (3) we get for (4)

(5) $$(res*N/m)(\log \log N^{1/2}) - res^2 *(k_{max}*(k_{max}+1)/2)$$

To get an expression in N, and to make the math easier, I will use $k_{max} \simeq Pi(N^{1/2})/res$, where here I'll use $Pi(N^{1/2}) \simeq N^{1/2}/(\log N^{1/2})$. After factoring this produces:

(6) $$res^2 *(k_{max}*(k_{max}+1)/2) = (N + res*(\log N^{1/2})*N^{1/2}) / (2*(\log N^{1/2})^2)$$

Replacing this is (5) gives the number of non-primes to be eliminated by the optimized sieve to be:

(7) $$(res*N/m)(\log \log N^{1/2}) - (N + res*(\log N^{1/2})*N^{1/2}) / (2*(\log N^{1/2})^2)$$

Again, the first term in (7) represents all the non-primes shown in Table 2 for a PG. As the PGs order grow, fewer non-primes are computed by the first term, as res/m gets smaller, while the second term gets bigger as res increases, which increasingly reduces the first term value more.

The real numbers of prime multiples (non-primes) can be empirically determined by counting the times Code 1 is inside the inner sieve loop (the while ... end loop) and multiplying that by res. The variable n in Code 1 counts the actual number of sieve primes $p_n < p_i \leq N^{1/2}$ from which k_{max} can be determined. Plotting these values against values of N for the PGs produces a revealing family of curves which graphically depict the comparative performance of the PGs.

Thus, the total operational complexity of the SoZ includes the prime candidates generation process, which creates res*(N/m) pc, and the sieve process, which eliminates (7) number of non-primes. This also represents the time complexity, as the generation/elimination of a prime candidate is merely an array write and variable increment with a constant.

The sieve array only needs N/2 1-bit elements, and the index values contain the numbers information. For coding purposes [N/2 + res] (byte) elements are used. Memory also is used to store and update the res residues values and res sieve product values. Only one set of res variables is needed to hold these values as they are used in separate operations. Thus, the larger PGs will use slightly more memory for larger res values, in addition to the sieve array.

The total SoZ operational complexity is, thus, minimal. It uses mostly array/mem writes and variable additions, and a handful of multiplications and divisions (shifts) and arithmetic/boolean comparisons. A sqrt operation is optional if the sqrt(n) is passed in as a parameter. This is similar to the operational complexity of the SoE. The SoZ is also simple and easy to implement as a parallel algorithm due to its algorithmic and operational simplicity. In comparison, the SoA, and quadratic residue sieves, are more algorithmically and operationally complex (as well as inherently harder to understand and code). They also need to perform many modulo operations (divisions) on increasingly bigger numbers requiring comparatively more computational time.

Therefore, the SoZ can be implemented to generate the primes up to a number N faster than any other method known to presently exist.

Primality testing

The SoZ can be performed backwards to create a deterministic primality tester. The easiest PG to use for this is P_3 = 6k+(1,5). The basic process involves the following steps:

1) return true if N is 2, 3, or 5.
2) return false if N is 1 or an even number.
3) return false if for N > 5 (N mod 6) is not equal to 1 or 5 OR N is a multiple of 5.
4) If n passes these tests, use the prime candidates produced by P_3, starting at k=1, as candidates for the primes ≤ sqrt(N), and perform the reverse of the sieve process $P'_n = p_j*[mk+(r_i, m+1)]$, which in this case is $p_j*[6k+(5, 7)]$. If N is a multiple of p_j > m it will be computed by the sieve. Thus, for every pc n, subtract the products p = 5n and q = 7n from N, and if any of these results is divisible by 6n ($m*p_j$) then N is non-prime. If N can pass this test for all the pc ≤ sqrt(N) it is a prime.

Example for N =119:
N=119 passes the first three tests (119 mod 6 = 5), and the sqrt(119) = 10. So we only need to test it for n = 7, which gives p = 7*5 = 35, q = 7*7 = 49, and k = 7*6 = 42, and then do: (119-49)= 70 and (119-35)=84; with (70 mod 42)=38, but (84 mod 42)=0 (84 is divisible by 42), thus 119 is non-prime. In other words, starting with 5*7, adding successively (42+42) = 7*6*2 (pmk), creates 35+84 = 119.

Example for N =113:
The sqrt(113) = 10, so again, n = 7, and p =35, q=49, and k=42. Now (113-49)=64 and (113-35)=78, and because neither (64 mod 42) or (78 mod 42) is zero, then 113 is prime.

Code 2 (in Ruby) uses P_3 to generate the prime candidates, and is faster than the primality tester in [1].

```
Code 2.
class Integer
   def primz?
      n = self.abs
      return true  if [2, 3, 5].include? n
      return false if n == 1 || n & 1 == 0
      return false if n > 5 && (! [1, 5].include?(n%6) || n%5 == 0)

      sqrtN = Math.sqrt(n).to_i
      n1, n2 = 7, 11
      while n1 <= sqrtN
        # pi= 5*i,  ki = 6*i,    qi = 7*i
        p1 = 5*n1;  k1 = p1+n1;  q1 = k1+n1
        return false if (n-p1)%k1 == 0 || (n-q1)%k1 == 0
        p2 = 7*n2;  k2 = p2+n2;  q2 = k2+n2
        return false if (n-p2)%k2 == 0 || (n-q2)%k2 == 0
        n1 += 6; n2 += 6
      end
      return true
   end
end
```

Hybrid configurations can be created by using bigger PGs to generate the primes for the reverse sieve. Since the SP PGs are the most efficient, they will generate relatively fewer non-primes, thus making verifying primes (which goes thru all the residues ≤ sqrt(N)) potentially faster. Coding wise, this means there will be res number of tuples (p,q,k) tests, with the residues iterated by m. The ultimate hybrid, however, computes the primes ≤ sqrt(N), and feeds this solely primes list to the reverse sieve. For testing really large numbers this is the fastest method. Code 3 shows this hybrid method.

```
Code 3.
class Integer
   def primz?(input=nil) # accepts array of primes: 7...sqrt(n)
      n = self.abs
      return true  if [2, 3, 5].include? n
      return false if n == 1 || n & 1 == 0
      return false if n > 5 && (! [1, 5].include?(n%6) || n%5 == 0)

      # if primes array is provided, use it, otherwise compute it
      primes = input || (Math.sqrt(n).to_i.sozP11 - [2,3,5])
      primes.each do |i|
        # p= 5*i,   k = 6*i,  q = 7*i
        p = 5*i;    k = p+i;  q = k+i
        return false if (n-p)%k == 0 || (n-q)%k == 0
      end
      return true
   end
end
```

In Code 3, the primes $7 \le p_i \le \text{sqrt}(N)$ can be input as a sorted array or computed internally if no array is input. Here sozP11 generates the primes, but other appropriate or available generators can be used. This method is perfectly suited for being implemented by cluster farms, which can divide the primes into small lists to be processed in parallel by as many cores/threads there are available to apply to it.

Thoughts and Conclusion

Because the PGs residues include the primes < m they really don't start to shine until N is much larger than m^2. This characteristic can mask the utility of the larger PGs. So for P_{13} with m = 30030, N should be much greater than 9 billion, and for P_{17}, N should to be much greater than 260 billion to get the optimum benefit from using it. This poses a problem for assessing the true performance of these generators on 32-bit PCs, which may only have a few GBs of memory, but needs lots of memory to represent and process these mega sized numbers.

A satisfying spinoff from the prime generators work is the identification of a simple, short, and elegant deterministic primality test algorithm. This should have wide use for many applications.

Also from empirical analysis, I found 2 good approximations of $\sum p_i$ for the primes up to N:

(8) $$(1/2)*N^2/(\log N) \; < \; (5/\pi^2)*N^2/(\log N) \; \simeq \sum p_i$$

One area of research I see a need for is characterizing where the Form 3 mcp come from analytically. I know how to computationally generate them, but not analytically. In fact, a general area of research should be to assess if the PGs can be modeled into a family of curves that would create them, much the same way I took the residues from the curves identified in [2] and combined them into the P_{60} PG.

Then there's the continuing work in finding/verifying all the possible PGs of form $P_n = mk+(1,r_i)$. This includes the SP and non-SP generators, and determining if all the multiples of 30 moduli, except for the *special* ones (multiples of SP moduli, et al?) have a m/res ratio of 15/4. There's a lot of interesting math going on here, which I hope spurs lots of academic, et al, interest and insights.

Lastly, with the increasing proliferation of multicore systems, cluster farms, and mega GBs of usable memory, the SoZ PGs can be used to greatly increase the search, and testing, of large primes.

References

[1] Ultimate Prime Sieve -- Sieve of Zakiya (SoZ), Jabari Zakiya, June 2008,
http://www.4shared.com/dir/7467736/97bd7b71/sharing.html

[2] Prime Sieves Using Binary Quadratic Forms, A.O.L. Atkin & Daniel J. Bernstein
http://cr.yp.to/ntheory.html#primesieves

[3] Fast Compact Prime Number Sieves (among others), Paul Pritchard
http://historical.ncstrl.org/tr/temp/ocr/cs-tr.cs.cornell.edu/TR81-473

[4] Segmented Sieve of Eratosthenes, Tomas Oliveira e Silva
http://www.ieeta.pt/~tos/software/prime_sieve.html

This paper also available at: http://www.4shared.com/dir/7467736/97bd7b71/sharing.html

Contact: jzakiya (at) yahoo (dot) com

Sieve of Zakiya (SoZ) Prime Generators

P_3 = 6k+(1,5)

P_5 = 30k+ (1, 7, 11, 13, 17, 19, 23, 29)

P_{60} = 60k+(1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59)

P_{110} = 110k+(1, 3, 7, 9, 11, 13, 17, 19, 21, 23, 27, 29, 31, 37, 39, 41, 43, 47, 49, 51, 53, 57, 59, 61, 63, 67, 69, 71, 73, 79, 81, 83, 87, 89, 91, 93, 97, 99, 101, 103, 107, 109)

P_{150} = 150k+(1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, 67, 71, 73, 77, 79, 83, 89, 91, 97, 101, 103, 107, 109, 113, 119, 121, 127, 131, 133, 137, 139, 143, 149)

P_{180} = 180k+(1,7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59, 61, 67, 71, 73, 77, 79, 83, 89, 91, 97, 101, 103, 107, 109, 113, 119, 121, 127, 131, 133, 137, 139, 143, 149, 151, 157, 161, 163, 167, 169, 173, 179)

P_7 = 210k+(1, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 121, 127, 131, 137, 139, 143, 149, 151, 157, 163, 167, 169, 173, 179, 181, 187, 191, 193, 197, 199, 209)

P_{11} = 2310k+(1, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 169, 173, 179, 181, 191, 193, 197, 199, 211, 221, 223, 227, 229, 233, 239, 241, 247, 251, 257, 263, 269, 271, 277, 281, 283, 289, 293, 299, 307, 311, 313, 317, 323, 331, 337, 347, 349, 353, 359, 361, 367, 373, 377, 379, 383, 389, 391, 397, 401, 403, 409, 419, 421, 431, 433, 437, 439, 443, 449, 457, 461, 463, 467, 479, 481, 487, 491, 493, 499, 503, 509, 521, 523, 527, 529, 533, 541, 547, 551, 557, 559, 563, 569, 571, 577, 587, 589, 593, 599, 601, 607, 611, 613, 617, 619, 629, 631, 641, 643, 647, 653, 659, 661, 667, 673, 677, 683, 689, 691, 697, 701, 703, 709, 713, 719, 727, 731, 733, 739, 743, 751, 757, 761, 767, 769, 773, 779, 787, 793, 797, 799, 809, 811, 817, 821, 823, 827, 829, 839, 841, 851, 853, 857, 859, 863, 871, 877, 881, 883, 887, 893, 899, 901, 907, 911, 919, 923, 929, 937, 941, 943, 947, 949, 953, 961, 967, 971, 977, 983, 989, 991, 997, 1003, 1007, 1009, 1013, 1019, 1021, 1027, 1031, 1033, 1037, 1039, 1049, 1051, 1061, 1063, 1069, 1073, 1079, 1081, 1087, 1091, 1093, 1097, 1103, 1109, 1117, 1121, 1123, 1129, 1139, 1147, 1151, 1153, 1157, 1159, 1163, 1171, 1181, 1187, 1189, 1193, 1201, 1207, 1213, 1217, 1219, 1223, 1229, 1231, 1237, 1241, 1247, 1249, 1259, 1261, 1271, 1273, 1277, 1279, 1283, 1289, 1291, 1297, 1301, 1303, 1307, 1313, 1319, 1321, 1327, 1333, 1339, 1343, 1349, 1357, 1361, 1363, 1367, 1369, 1373, 1381, 1387, 1391, 1399, 1403, 1409, 1411, 1417, 1423, 1427, 1429, 1433, 1439, 1447, 1451, 1453, 1457, 1459, 1469, 1471, 1481, 1483, 1487, 1489, 1493, 1499, 1501, 1511, 1513, 1517, 1523, 1531, 1537, 1541, 1543, 1549, 1553, 1559, 1567, 1571, 1577, 1579, 1583, 1591, 1597, 1601, 1607, 1609, 1613, 1619, 1621, 1627, 1633, 1637, 1643, 1649, 1651, 1657, 1663, 1667, 1669, 1679, 1681, 1691, 1693, 1697, 1699, 1703, 1709, 1711, 1717, 1721, 1723, 1733, 1739, 1741, 1747, 1751, 1753, 1759, 1763, 1769, 1777, 1781, 1783, 1787, 1789, 1801, 1807, 1811, 1817, 1819, 1823, 1829, 1831, 1843, 1847, 1849, 1853, 1861, 1867, 1871, 1873, 1877, 1879, 1889, 1891, 1901, 1907, 1909, 1913, 1919, 1921, 1927, 1931, 1933, 1937, 1943, 1949, 1951, 1957, 1961, 1963, 1973, 1979, 1987, 1993, 1997, 1999, 2003, 2011, 2017, 2021, 2027, 2029, 2033, 2039, 2041, 2047, 2053, 2059, 2063, 2069, 2071, 2077, 2081, 2083, 2087, 2089, 2099, 2111, 2113, 2117, 2119, 2129, 2131, 2137, 2141, 2143, 2147, 2153, 2159, 2161, 2171, 2173, 2179, 2183, 2197, 2201, 2203, 2207, 2209, 2213, 2221, 2227, 2231, 2237, 2239, 2243, 2249, 2251, 2257, 2263, 2267, 2269, 2273, 2279, 2281, 2287, 2291, 2293, 2297, 2309)

P_{13} = 30030k+(1, [5759 r_i residue values: 3242 primes, 1431 Form 2 mc, 1086 Form 3 mc])
P_{17} = 510510k+(1, [92159 r_i residue values: 42324 primes, 23339 Form 2 mc, 26496 Form 3 mc])