



© 2007 SATOCONOR
A collection of journals
for the experimental sciences
submit@satoconor.com
Web portal: <http://www.satoconor.com>
Netherlands Rotterdam
Europe

Scientia Araneae Totius Orbis
Journal of Randomics

Primes, Randomness and the Prime Twin Proof

The article explains randomness and uses recursively self complicating algorithms to prove the Prime Twin conjecture.

By Martin C. Winer

For comments: martin_winer@hotmail.com

Version 1.0 November 7, 2007

Submitted: August 12, 2007

Revisions: November 1, 2007

Accepted: November 7, 2007

Mathematics has long been thought of the study of absolute truths. It is then no wonder then that there have been considerable problems in defining a mathematical notion of randomness. It is one of those notions which we all intrinsically feel we understand yet we consistently fail to define it in absolute terms.

"It is evident that the primes are randomly distributed, we just don't know what 'random' means."

-- R.C. Vaughan

This paper will provide a definition of randomness in mathematical terms. Once we have a working definition we can examine the randomness of a recursively self complicating function called $Pat(2n+1,n)$ which incidentally lays down the primes along the number line. Code examples are provided to assist in understanding the basic concepts presented.

Using notions of randomness and pattern combinatorics it will be shown that many contentious prime constellations such as prime twins, triplets and quadruplets are indeed infinite as commonly suspected.

1. Introduction

1.1. What is random?

Randomness is one of those things that is hard to pin down. As applied to patterns, a random pattern is one which has an infinite supply of complexity. Complexity, in turn, is given by the number of non-reducible, non-recursive (non-reflexive) statements needed to categorize a pattern. As an example, suppose you are asked to categorize yoghurt. You could say that yoghurt is yoghurt and milk. This is a recursive (reflexive) definition which does not categorize yoghurt. Sure, you can make more yoghurt now, but you need to know that yoghurt is milk and bacterial culture to make yoghurt without having yoghurt on hand.

Similarly recursive statements give us (potentially) some information about the pattern at hand, however, to fully categorize the problem, we need non recursive definitions which tell us how to define the pattern out of elemental (deterministic) statements. Each non-reducible, non-recursive statement contributes some measure of complexity. As a corollary, complexity of a pattern can be thought of as the collection of all essential non-reducible information about this pattern.

Each statement about the random pattern is deterministic. That is, it is finite and elemental. However, when you take these deterministic elemental patterns and unboundedly combine them together you get a random pattern.

1.2. Formal definition

Randomness = Recursive self complication cross determinism

1.3. Contrast to Chaitin-Kolmogorov complexity

In the 1960's G. Chaitin and A. N Kolmogorov concurrently developed a notion of algorithmic complexity. Put simply a bit string (ones and zeros) was random if and only if there did not exist a program outputting this string whose size was less than that of the original string. In so doing they examined the notion of compressibility. The Chaitin-Kolmogorov approach determines compressibility based on the size of the program generating the bit string. In contrast, recursive self complication examines the space time complexity of the produced bit string. The produced bit strings must take an infinite amount of time to produce and must be recursively self complicating, hence require an infinite amount of computational space. The Chaitin-Kolmogorov approach regards most small strings as random because the programs that generate them are equal to or larger than the size of the small bit string. For example 100 is considered random. The recursive self complication definition cross determinism definition never considers any finite bit string random. Further only those built via recursive self complication will be considered random.¹

2. How do we get a Random Pattern?

Random patterns as explained above are produced by recursive self complicating using deterministic elementals. We will consider only binary patterns (patterns of ones and zeros). The general form of the algorithm follows

2.1. Pat(f(n))

$Pat(f(n)) = Pat(f(n-1)) \text{ merge } Elemental(f(f_0))$ where.

$Pat(0)$ is the empty pattern,

f_0 is the location (1 based) of the first zero in the lhs pattern (ie $Pat(f(n-1))$), and

$f(n)$ is some function.

(See the 'Code Samples' section for examples and code output.)

2.2. Merge operation

The merge operation takes the left hand side (lhs) pattern and finds the first zero. It then composes the $Elemental(f(f_0))$ pattern and matches (aligns) the leading one of the elemental with the first zero of the lhs pattern. It then repeats both patterns until $length(lhs \text{ pattern}) * length(Elemental(f(n)))$ bits are produced of each. It then OR's the result producing the new pattern. Further, the merge of any pattern with the empty pattern produces that same pattern. The first zero of the empty pattern is defined to exist at position 1. (The empty pattern can be thought of as an infinite string of only zeros or 0...)

2.3. Elemental(f(f0))

The Elemental(f(f0)) function produces a pattern of one followed by f(f0)-1 zeros. (Recall: f0 is the location of the first zero in the lhs pattern.) An example of an elemental would be 10... = 10101010 and so on. This is an irreducible pattern. It cannot be described any more simply than 10...

2.4. When will Pat(f(n)) produce random patterns?

Pat(f(n)) will produce random patterns IFF Pat(f(n), n=1) has at least 1 one and 1 zero and if f(n+1)>f(n) for all n. This is true due to the pattern combinatorics of the 'merge' function. Examine the merge function (use the code examples and output to assist you) and note that any pattern that exists in the left hand side pattern, is guaranteed to exist in the merged pattern. If Pat(f(n), n=1) has ones and zeros, and Elemental(n+1) has more zeros (and at least one '1') than Elemental(n), we're always guaranteed to get more zeros as we iterate which are later matched with the ones of the Elemental in subsequent merge operations.

2.5. Example Pat(2n+1)

Let's work an example to see how this works. Let's consider Pat(2n+1). n=(0,∞): and observe the recursion at work. (See also Appendix A.)

```
Pat(2n+1,n=0) = emptyPattern by definition
Pat(2n+1,n=1) = Pat(2n+1,n=0) merge Elemental(2f0+1,f0=1)
Pat(2n+1,n=1) = emptyPattern merge Elemental(2f0+1,f0=1); f0 of the empty pattern is defined
to be 1
Pat(2n+1,n=1) = emptyPattern merge Elemental(3) [Elemental(2f0+1,f0=1)]
Pat(2n+1,n=1) = emptyPattern merge 100... (the '...' means repeat everything to the left over and
over again, not just the last entry over and over, but everything to the left, over and over)
Pat(2n+1,n=1) = 100...
Pat(2n+1,n=2) = 100... merge Elemental(2f0 + 1,f0=2); f0 of the lhs pattern occurs at position 2
Pat(2n+1,n=2) = 100... merge Elemental(5)

Pat(2n+1,n=2) = 100... merge 10000...
100... merge 10000...
<=> 100100100100100 (lhs)
    010000100001000 (rhs)
    -----
    110100100101100... = Pat(2n+1,n=2)
Note length(Pat(2n+1,n=2)) = length(Pat(2n+1,n=1)) x length(Elemental(5)) = 3x5 = 15

Pat(2n+1,n=3) = 110100100101100... merge Elemental(2f0 + 1,f0=3); f0 of the lhs pattern
110100100101100... merge 1000000...
<=> 1101001001011001101001001011001101001001011001101001<<truncated>>(lhs)
    0010000001000000100000010000001000000100000010000001<<truncated>>(rhs)
    OR
    -----
=11110010010110011010010110110011010011010110110110100100101110110101100101100110110100101100110
10010011
```

2.6. Examining the descriptions of Pat(2n+1)

Describe Pat(2n+1,n=1) = 100...

This is the pattern of one followed by 2 zeros repeated over and over again

Describe: Pat(2n+1,n=2) = 110100100101100

This is the pattern of one followed by 2 zeros repeated over and over again AND the pattern of 1 followed by 4 zeros, with the first 1 of the pattern at position 2, repeated over and over again.

Note: as we iterate, we get more and more irreducible, non-recursive statements.

3. Stateless Entities versus Stateful Algorithms

The number 1 is stateless. It is 1, it will always be 1 and will never change. Algorithms however, describe things which can exist in certain states. Suppose I wanted to construct an algorithm that defined the number 1. First of all it's important to realize that 1 is actually 0.999999999999 and so on.

Proof: $1/3 = 0.33333333$ and so on

Multiply both side by 3

$1 = 0.99999999$ and so on.

Now lets construct an algorithm to make 0.99999 and so on. $One(1) = 9/10 = 0.9$, $One(n) = 9/10^n + One(n-1)$.

If we allow this algorithm to run indefinitely, it will produce 0.999999... and so on and hence produce 1. However, at any stage $One(n)$ for any finite n does NOT EVER equal 1. $One(n)$ for any finite n , represents a state of the algorithm, but with unbounded n , this describes the number 1.

Examining $Pat(f(n))$, we see that provided $f(n)$ meets the requirements of $Pat(1)$ having at least one 1 and one 0 and that $f(n+1) > f(n)$ for all n , that $Pat(f(n))$ will produce more and more complicated patterns as it iterates. For no n will $Pat(f(n))$ produce a fully random pattern, there will only be finite (yet possibly great – certainly increasing -- amounts of) complexity. However, if we allow $Pat(f(n))$ to iterate unboundedly, we get a truly random pattern.

4. Importance of the Set of First Zeros.

Note that as we iterate, the description of $Pat(f(n))$ goes something like, the pattern of 1 followed by x zeros starting at position y repeated over and over AND the pattern of 1 followed by p zeros starting at position q repeated over and over again ... AND ... and so on. So the position of the first zero in the lhs pattern becomes the position of the first 1 in the rhs pattern. As such the set of first zeros is an important one because it contains necessary information we need to categorize $Pat(f(n))$. Let's call this set $firstZeros$. The $firstZeros$ set for $Pat(2n+1)$ is $\{1,2,3,5,6,8,9...\}$

4.1. Allowable constellations in firstZeros

Notice in the $firstZeros$ set for $Pat(2n+1)$ the entries 1,2,3. Note that in the rest of the entries you never get 3 numbers 1 apart ever again. This marks an unallowable constellation. Allowable constellations are ones where candidates for their creation occur infinitely. When allowable constellations first appear, the elemental patterns which currently exist can be arranged such that the zeros of these patterns can align to produce a candidate for this pattern. A candidate for a pattern is the same pattern XOR'ed (that is with the bits reversed). For example, a candidate for the constellation $c,c+1,c+2$ is 000 in $Pat(2n+1)$.

Consider 1,2,3 ($c,c+1,c+2$): The elemental patterns here are:

100...

10000...

1000000...

Try as you may, you can't shift these patterns such you can get 3 zeros to align in all 3 patterns. (Simply put, the pattern 100... blocks any such attempt because there is always a 1 in any 3 considered positions). Thus all constellation candidates for $c,c+1,c+2$ are destroyed

at once, by the first instance of the constellation no less. Thus the constellation $c, c+1, c+2$ occurs only once, but isn't allowed.

How about just 1,2? Sure, that's allowable. The elementals are:

100...

10000... where two zeros can be made to align in all the elementals

^^

How about the simplest constellation, just 1? Ie, just one entry. This first occurs with the elementals 100... and sure there are two zeros (we need only one) which can be aligned to produce this constellation.

4.2. All allowable constellations in firstZeros occur infinitely

Any allowable constellation is a building block upon which future complexity of $\text{Pat}(f(n))$ is built. Recall that $\text{Pat}(f(n))$ recursively takes the complexity of the lhs pattern and merges in the complexity of some elemental. The key here is that any complexity (details) that occurred in the lhs are preserved in the new pattern. Simply put this means that if you had two consecutive zeros in the lhs pattern, the new pattern will have two consecutive zeros somewhere in the new pattern.

Look at the constellation 1,2,3 or $c, c+1, c+2$: Once created it destroys itself in $\text{Pat}(2n+1)$ forever. Suppose an allowable constellation such as $c, c+1$ occurred finitely in firstZeros. This means that the constellation candidates were destroyed all at once or they were destroyed iteratively. If the constellation candidates were destroyed all at once, this must mean that it was not an allowable constellation to begin with. This would mean that the elementals forbade the construction of such a constellation candidate anywhere in $\text{Pat}(2n+1)$ and hence such a constellation couldn't have been an allowable constellation in the first place.

Next, suppose the constellation is destroyed iteratively during construction of $\text{Pat}(2n+1)$. This means that the constellation candidates (in this case two adjacent zeros) are not destroyed all at once but simply, magically, never manage to make it to be the head of a new $\text{Pat}(2n+1)$, and hence another entry of this constellation into firstZeros. The more you avoid allowing a certain constellation to be at the head of $\text{Pat}(2n+1)$, the less normally, or randomly, distributed $\text{Pat}(2n+1)$ is. However, this constellation is itself a building block of complexity which is iteratively being used to make $\text{Pat}(2n+1)$ more and more complicated and more and more random. Now we can't suck and blow from the same pipe. If we're using a constellation candidate to build complexity, and complexity in turn produces randomness and normal distribution, then it is impossible to ad infinitum disallow any constellation candidate access to the head of $\text{Pat}(2n+1)$.

Hence it has been shown that immediate or iterative prevention of allowable constellation candidates reaching the head of $\text{Pat}(2n+1)$ is impossible, making all allowable constellations infinite in firstZeros.

4.3. How does all of this apply to prime constellations? The twin primes etc. proof

The latest prime constellations research is by Terrence Tao who incorporates the work of Goldston and Yildirim. Currently, Ben Green and Terrence Tao have been able to show that there are "arbitrarily long arithmetic progressions of primes."² This is on the way to proving that prime constellations such as prime twins are infinite, however, it is not quite there yet. However if we examine $\text{Pat}(2n+1, n)$ more closely with respect to the firstZeros perhaps we can do better.

Examine the firstZeros set for $\text{Pat}(2n+1) = \{1, 2, 3, 5, 6, 8, 9, \dots\}$. Note the magic that occurs when you multiply every element by $2n+1$ yielding: $\{3, 5, 7, 11, 13, 17, 19, \dots\}$ why you get the (odd) primes? Proof? Intuitively this makes sense as $\text{Pat}(2n+1, n)$ is nothing other than a

Sieve of Eratosthenes over all the odd numbers. Examining $\text{Pat}(2n+1, n)$ in more detail we see that all the elementals of $\text{Pat}(2n+1)$ form the pattern of multiples for all the primes. The constellation $c, c+1$ is allowable in $\text{Pat}(2n+1)$, and all allowable constellations occur infinitely. If this is true, then primes that are two apart (prime twins) must occur infinitely because this corresponds to an allowable constellation in $\text{P}(2n+1)$. Moreover ALL constellations of primes which corresponds to allowable constellations in $\text{P}(2n+1)$ are infinite. In case that last part missed your attention that means prime triplets quadruplets etc etc of allowable form are all infinite too.

5. Conclusions

The recursive self complicating program will give the same random string every time you run it. The key is that the longer you run it, the more randomness you get.

Many have commented that the digits of the irrational numbers appear to be random. Take for example the digits of π^* and the digits of e (base of the natural logarithm). The digits of these numbers appear to be random upon first inspection. However, are they 'truly random'? If they were 'truly random' as we all intuitively suspect they are, then wouldn't they include one another? That starting at digit n of π , wouldn't we start to see the digits of e ? If they were 'truly random' given a large enough n , eventually, and randomly, we should start to see all the digits of e nested in the digits of π .

This of course cannot occur. The reason is that if the irrational numbers generally include one another, then, eventually you would get a repeat in the digit sequence at which time the number would no longer be irrational. (recall that any number that has a repeat in its digit expansion is necessarily rational). So suppose at digit 10,000 of π , we start to see the digits of e and at digit 20,000 of e we start to see the digits of π . Then we have a repeat of the digits at π repeating at 20,000 and π is no longer irrational.

So what is the problem here? Possibilities include 1) irrationals are NOT random, and there is a notion of 'truly random' out there or 2) irrationals are random, and our notion of random needs to be adjusted or 3) there is no such thing as random at all. Let us examine the first option. So there is some 'truly random' concept out there which the irrational numbers don't conform to. Great! Let's make a number using this 'truly random' generator. What we'll do is we'll buy one that spits out a truly random 1 or a 0. We'll ask it for a one or a zero 10 times and add up the number of 1's and output the result. We'll keep repeating to build up a string of digits and then prepend "0." to the beginning of it. So we'll have something like 0.2849294729187489290 and so on. This number should be irrational because it has no repeat because it's based on something 'truly random'. There is a problem however: Shouldn't this number also contain all the digits of π ? Sure it should, it's based on something 'truly random' so eventually it should spit out all the digits of π , and e and other irrationals too while you're at it. The problem is that the irrationals are infinite length, so we must have general inclusion, where one includes the other and the other includes the first. Whenever that happens, we get a repeat and the number is no longer irrational. So we've failed to establish a concept of 'truly random'.

This paper takes the second option, there is a notion of random and it just needs to be adjusted. This paper adjusts the notion of random to be the result of a recursively self complicating algorithm. Taking this definition of randomness as a lemma, a proof by construction of the long standing Prime Twin problem can be constructed as shown above.

* π is the result of a recursively self complicating process (a continued fraction) crossed with determinism.

<http://mathworld.wolfram.com/images/equations/PiContinuedFraction/equation3.gif>

<http://mathworld.wolfram.com/images/equations/PiContinuedFraction/equation4.gif>

In both examples we can see patterns emerge, yet, there are no clear patterns in π itself. This is because the recursively self complicating process (the continued fraction) creates randomness out of this elemental determinism.

As for the final option, the option that there is no such thing as randomness, this may be an argument of semantics. There will be some that say that recursive self complication is not random and others that will. However, on an intuitive level we can all speak of some phenomenon with infinite complexity. This to me belies the notion that there is no such thing as random.

Notes & References:

- 1) G. Chaitin 'Meta math: The quest for omega' *First Vintage Books*, 168-172 (2006)
- 2) B. Green, T. Tao 'The primes contain arbitrarily long arithmetic progressions' (Version 6 September 23, 2007)
<http://front.math.ucdavis.edu/math.NT/0404188>
- 3) Further Reading: M. Winer 'Primes: randomness and prime twin proof' (2004-2006)
<http://www.rankyouragent.com/primes/primes.htm>
- 4) Further Reading: M. Winer 'Primes: randomness and prime twin proof (Abridged)' (2004-2007)
http://www.rankyouragent.com/primes/primes_simple.htm

Appendix A

Code Samples:

Some Java and 'C' code has been provided to illustrate some of the concepts on this website. It can be found here:

<http://www.rankyouragent.com/primes/patn.java.htm>

output:

<http://www.rankyouragent.com/primes/patn.java.output.txt>

A 'C' version

<http://www.rankyouragent.com/primes/patn.c.htm>

output:

<http://www.rankyouragent.com/primes/patn.c.output.txt>

Sample output: ... visit the link for full output

```
-----
Elemental(2f0+1,f0=1) = 100 -- shifted to align with f0 --> 100
Pat(2n+1,n=1) = 100
Size = 3
Number of 1's = 1; 1/3 = 0.3333333333333333
Number of 0's = 2; 2/3 = 0.6666666666666666
Total Candidates= 1
Singleton Candidates = 0; 0/1 = 0.0
Twin Candidates = 1; 1/1 = 1.0
-----
Elemental(2f0+1,f0=2) = 10000 -- shifted to align with f0 --> 01000
Pat(2n+1,n=2) = 110100100101100
Size = 15
Number of 1's = 7; 7/15 = 0.4666666666666667
Number of 0's = 8; 8/15 = 0.5333333333333333
Total Candidates= 5
Singleton Candidates = 2; 2/5 = 0.4
Twin Candidates = 3; 3/5 = 0.6
-----
Elemental(2f0+1,f0=3) = 1000000 -- shifted to align with f0 --> 0010000
Pat(2n+1,n=3) =
111100100101100110100101101100110100110101101101001001011101101011001011001011001011001011001101
00100111100
Size = 105
Number of 1's = 57; 57/105 = 0.5428571428571428
Number of 0's = 48; 48/105 = 0.45714285714285713
Total Candidates= 33
Singleton Candidates = 18; 18/33 = 0.5454545454545454
Twin Candidates = 15; 15/33 = 0.45454545454545453
-----
Elemental(2f0+1,f0=5) = 10000000000 -- shifted to align with f0 --> 00001000000
Pat(2n+1,n=4) = TOO BIG!
Size = 1155
```

```
Number of 1's = 675; 675/1155 = 0.5844155844155844
Number of 0's = 480; 480/1155 = 0.4155844155844156
Total Candidates= 345
Singleton Candidates = 210; 210/345 = 0.6086956521739131
Twin Candidates = 135; 135/345 = 0.391304347826087
-----
Elemental(2f0+1,f0=6) = 1000000000000 -- shifted to align with f0 --> 0000010000000
Pat(2n+1,n=5) = TOO BIG!
Size = 15015
Number of 1's = 9255; 9255/15015 = 0.6163836163836164
Number of 0's = 5760; 5760/15015 = 0.3836163836163836
Total Candidates= 4275
Singleton Candidates = 2790; 2790/4275 = 0.6526315789473685
Twin Candidates = 1485; 1485/4275 = 0.3473684210526316
-----
Elemental(2f0+1,f0=8) = 10000000000000000 -- shifted to align with f0 --> 00000001000000000
Pat(2n+1,n=6) = TOO BIG!
Size = 255255
Number of 1's = 163095; 163095/255255 = 0.6389492860081095
Number of 0's = 92160; 92160/255255 = 0.3610507139918905
Total Candidates= 69885
Singleton Candidates = 47610; 47610/69885 = 0.6812620734063104
Twin Candidates = 22275; 22275/69885 = 0.31873792659368966
-----
Elemental(2f0+1,f0=9) = 1000000000000000000 -- shifted to align with f0 -->
00000000100000000000
Pat(2n+1,n=7) = TOO BIG!
Size = 4849845
Number of 1's = 3190965; 3190965/4849845 = 0.6579519551655775
Number of 0's = 1658880; 1658880/4849845 = 0.3420480448344225
Total Candidates= 1280205
Singleton Candidates = 901530; 901530/1280205 = 0.7042075292628915
Twin Candidates = 378675; 378675/1280205 = 0.2957924707371085
-----
Elemental(2f0+1,f0=11) = 1000000000000000000000 -- shifted to align with f0 -->
00000000001000000000000
Pat(2n+1,n=8) = TOO BIG!
Size = 111546435
Number of 1's = 75051075; 75051075/111546435 = 0.6728236092888132
Number of 0's = 36495360; 36495360/111546435 = 0.32717639071118676
Total Candidates= 28543185
Singleton Candidates = 20591010; 20591010/28543185 = 0.7213984704229749
Twin Candidates = 7952175; 7952175/28543185 = 0.2786015295770251
-----
```