Using prime numbers to create high entropy derivative numbers

by Sylvain Saucier

Abstract

Derivative bit entropic numbers are numbers containing a minimum of set bits. It is meant to be used within a pseudo random number generator using an internal state composed of these derivative numbers and using XOR to generate the sequence. To avoid a situation where a seed set to zero create a sequence that expose the internal state.

This paper explore the evolution of the method used and their entropic quality.

Methods

Constructing a derivative number

The function takes as input a 64 bits unsigned integer. Starting at bit 0 and iterating through each bit, the function multiply the derivative with a prime number located in one of two lists, depending if the bit is set to 0 or 1.

```
uint64_t nz_derivative_64(uint64_t input)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709}; //every other prime, 2 and up
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719}; //every other prime, 3 and up

    for(int x = 0; x < 64; x++)
    {
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        mask <<= 1;
    }

    return output;
}
```

Measuring quality of the derivative function

In order to measure the quality of a derivative function, the function is used on a large sample of inputs and data is collected to measure two aspects.

Set bit count distribution

The sum of all the set bits (1) in a given derivative number.
For example:

- 01101000 have a set bit count of 3
- 10110111 have a set bit count of 6

When represented in a graphic the desired shape is bell curve. A centred bell means the distribution of 0 and 1 are well balanced, this is desired. A bell curve that touches the edges of the graphic means that getting all 0 or all 1 is a possibility. This is not a desired outcome.

Individual bit count distribution

The position of individual set bit in a given derivative number, starting at 0 with least significant bit.
For example:

- 01101000 have individual set bits at positions 3, 5, 6
- 10110111 have a set bit count of 0, 1, 2, 4, 5, 7

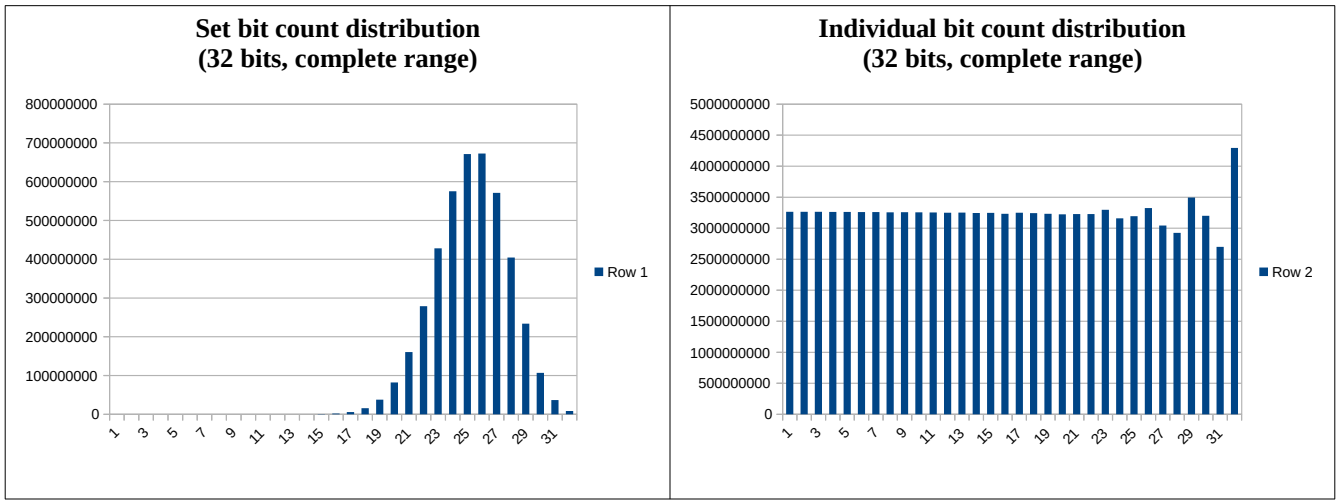
When represented on a graphic the desired shape is a straight line. Otherwise it means a certain bit has more probability of being set, this is not a desired outcome.

The 32 bit bridge

Because the all the possible inputs is composed of 2^{64} possibilities, i ran an experiment using a 32 bit version of the function to allow the processing of all 2^{32} possibilities.

```
uint32_t nz_derivative_32(uint32_t input, uint8_t offset)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709}; //every other prime, 2 and up
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719}; //every other prime, 3 and up

    for(int x = 0; x < 32; x++)
    {
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        mask <<= 1;
    }

    return output;
}
```

Running through the entire 32 bit number space, we can observe the function have a strong bias towards bits set to 1, even allowing for derivative numbers using only bits set to 1. The individual bit count distribution also contains animalies in bits 0 through 9. This algorithm is not fit for its purpose.



32 bit with left rotation

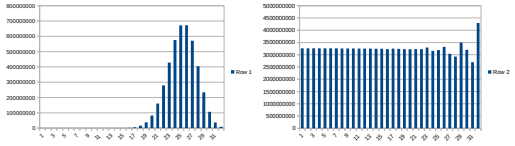
In order to correct the anomalies, I tried doing a circular left shift after the multiply operation. I ran it 32 times changing the offset or rotation from 0 to 31. The left rotation introduced a different anomaly. It mostly corrected the set bit bias for the lowest n bits, where n equal to the offset. At offset 31 we can see only bit 31 have a bias and the bell curve is quite centred now. However this method cannot eliminate the bit 31 bias, making unfit for its purpose. The cause of this effect is unknown to me.

```
uint32_t nz_derivative_32(uint32_t input, uint8_t offset)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709}; //every other prime, 2 and up
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719}; //every other prime, 3 and up

    for(int x = 0; x < 32; x++)
    {
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        output = (output << offset) | (output >> (32 - offset));
```

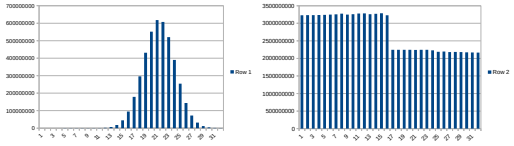
```
        mask <<= 1;
    }

    return output;
}
```

Set bit count distribution and Individual bit count distribution by left rotation offset
(32 bits, complete range, left rotation offset 0 to 31)

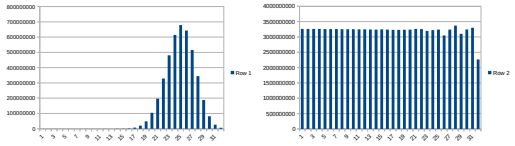
0



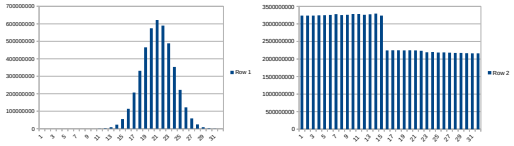
16



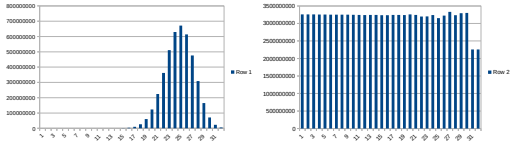
1



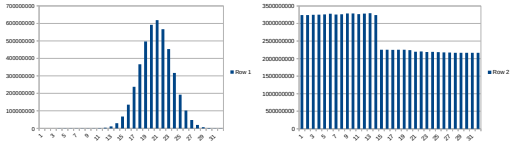
17



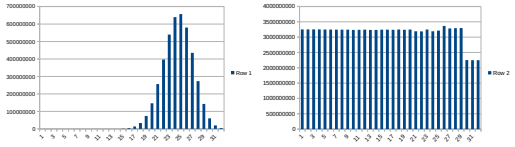
2



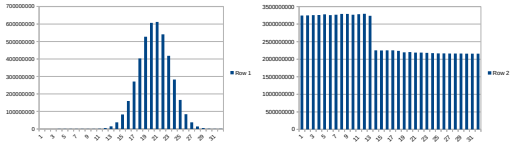
18



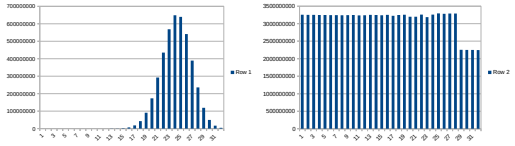
3



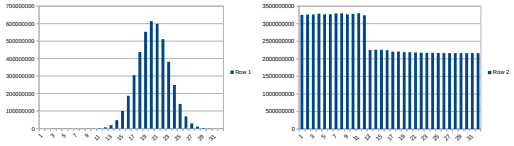
19



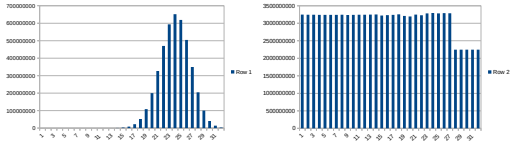
4



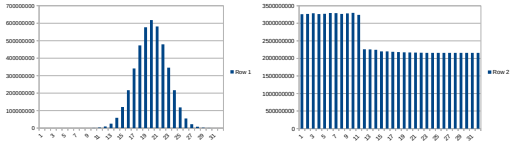
20



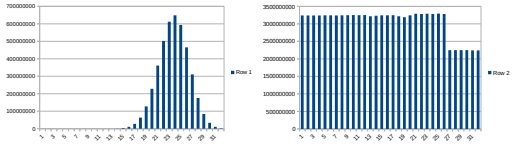
5



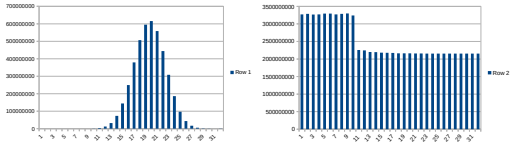
21



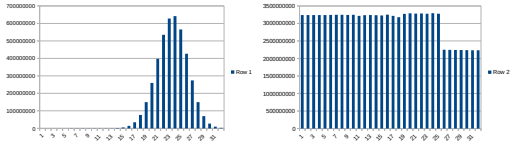
6



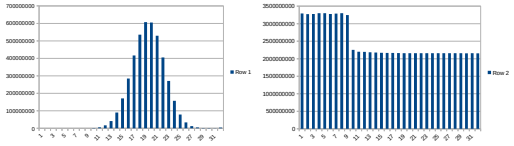
22



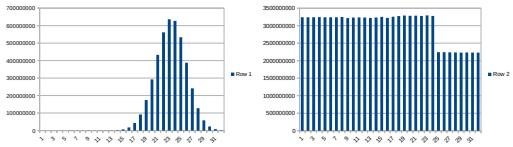
7



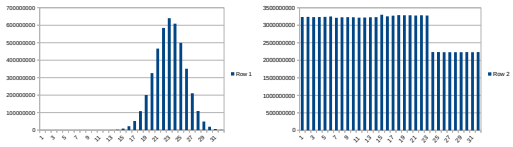
23



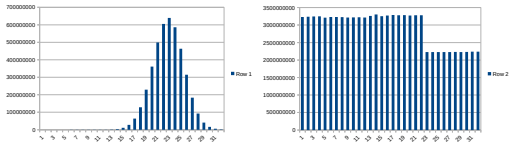
8



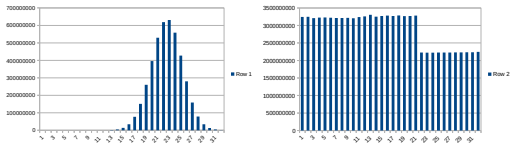
9



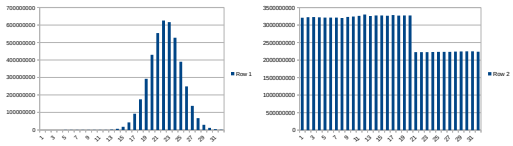
10



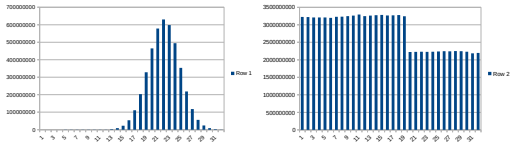
11



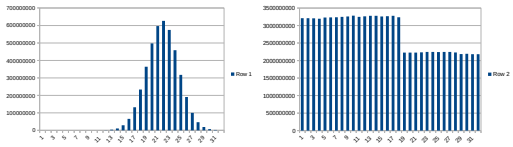
12



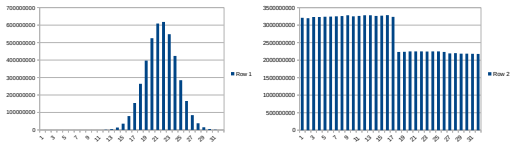
13



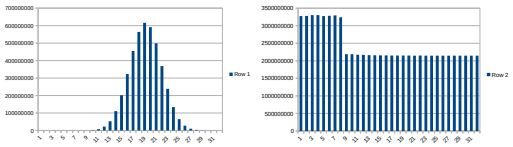
14



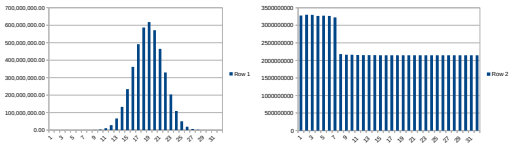
15



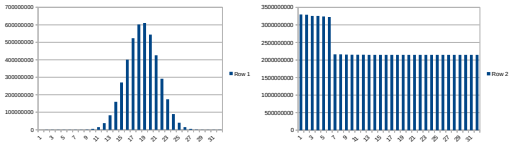
24



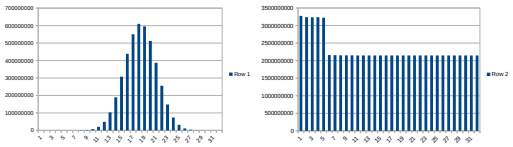
25



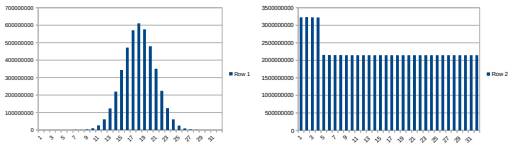
26



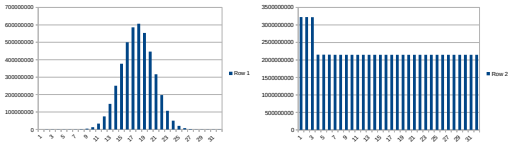
27



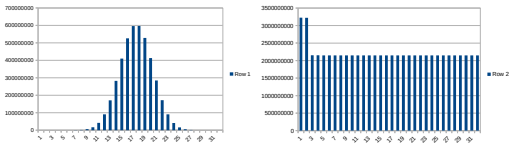
28



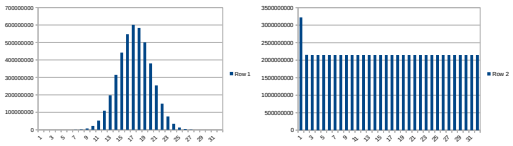
29



30



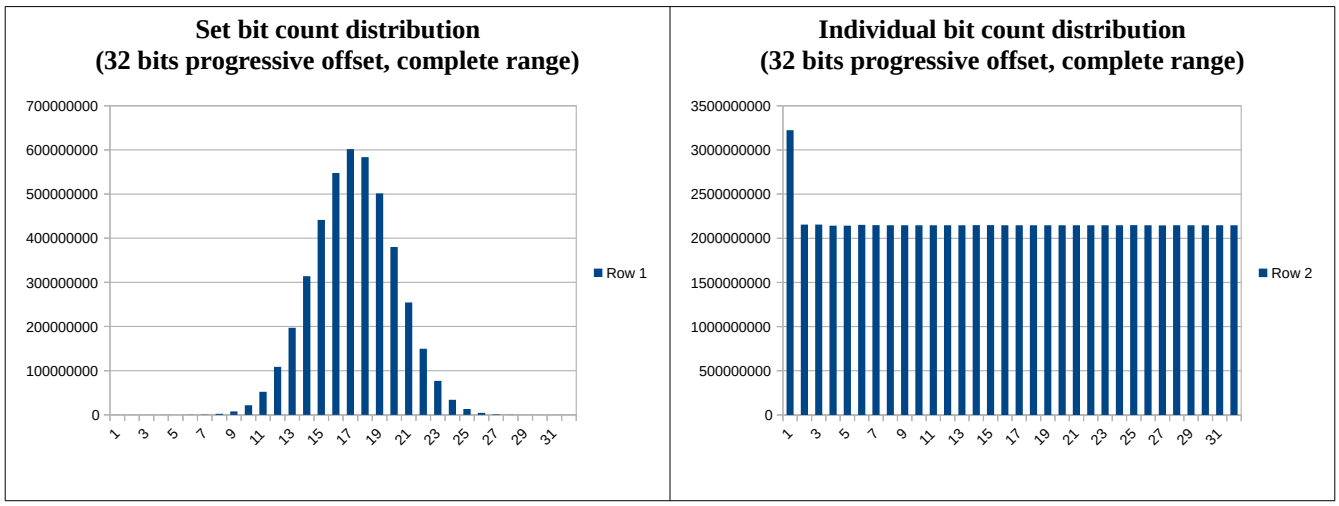
31



32 bit with progressive offset

The next step has been to try to have a non-uniform left rotation offset, using the current bit position as the offset. Results are comparable to a constant offset of 31. This is not suitable for the purpose of the function, something else must be done.

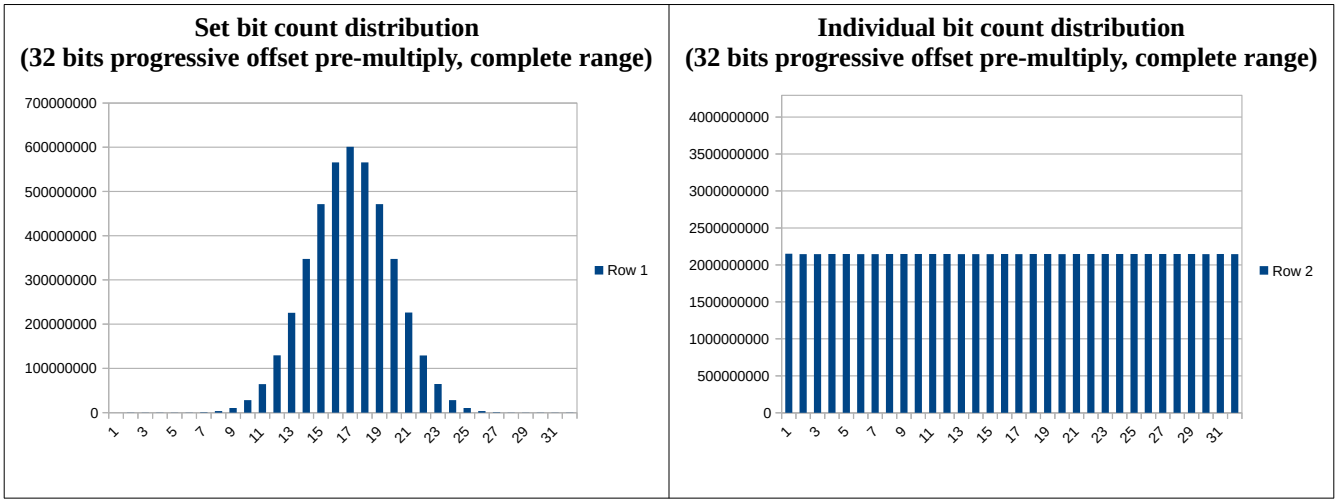
```
uint32_t nz_derivative_32b(uint32_t input)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709}; //every other prime, 2 and up
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719}; //every other prime, 3 and up

    for(int x = 0; x < 32; x++)
    {
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        output = (output << x) | (output >> (32 - x));
        mask <<= 1;
    }

    return output;
}
```



32 bit with progressive offset before the prime multiplication

To mix things up I decided to keep the progressive offset but move the operation before the multiplication. This small change made all the difference. My intuition was that the order was not so significant. Maybe the “moving wall” observed on the first offset test would disappear, I did not tested it and don't know. It could be an interesting thing to study in the future. I am focused on getting my function to perform adequately for its purpose despite a slight bias for set bits to 1 in general. Next step, moving back to 64 bits space.

```
uint32_t nz_derivative_32c(uint32_t input)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709};
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719};

    for(int x = 0; x < 32; x++)
    {
        output = (output << x) | (output >> (32 - x));
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        mask <<= 1;
    }

    return output;
}
```



The bridge from 32 bit to 64 bit

To move to 64 bit space, I will need to use a significant sample of the 64 set. To do so, I created an algorithm to create sample numbers. I used it first on the 32 bit set and then on the 64 bit set. This algorithm use base bit patterns and apply XOR transformation of all possible 1 bit to 6 bit permutation, and then use a NOT transformation on each sample. This ensure my sample set will be set bit neutral (as much 1 as 0 in the whole set). By using an all 0 pattern as one of the base, the edge cases are being explored. Please see the code below as reference.

32 bit pattern generator

```
uint32_t patterns[] = {0x00000000, 0x55555555, 0x33333333, 0x11111111};
uint64_t count = 0;
uint32_t pattern = 0;
FILE* output = fopen("patterns32.bin", "wb");

for(int p = 0; p < 4; p++)
{
	pattern = patterns[p];
	fwrite(&pattern, sizeof(uint32_t), 1, output);
	count++;

	pattern = ~pattern;
	fwrite(&pattern, sizeof(uint32_t), 1, output);
	count++;

	for(int b1 = 0; b1 < 64; b1++)
	{
		pattern = (patterns[p] ^ ((uint32_t)1 << b1));
		fwrite(&pattern, sizeof(uint32_t), 1, output);
		count++;

		pattern = ~pattern;
		fwrite(&pattern, sizeof(uint32_t), 1, output);
		count++;
	}

	for(int b1 = 0;    b1 < 31; b1++)
	for(int b2 = b1+1; b2 < 32; b2++)
	{
		pattern = (patterns[p] ^ ((uint32_t)1 << b1 | (uint32_t)1 << b2));
		fwrite(&pattern, sizeof(uint32_t), 1, output);
		count++;

		pattern = ~pattern;
		fwrite(&pattern, sizeof(uint32_t), 1, output);
		count++;
	}

	for(int b1 = 0;    b1 < 30; b1++)
	for(int b2 = b1+1; b2 < 31; b2++)
	for(int b3 = b2+1; b3 < 32; b3++)
	{
		pattern = (patterns[p] ^ ((uint32_t)1 << b1 | (uint32_t)1 << b2 | (uint32_t)1 <<
b3));
		fwrite(&pattern, sizeof(uint32_t), 1, output);
		count++;

		pattern = ~pattern;
		fwrite(&pattern, sizeof(uint32_t), 1, output);
		count++;
	}

	for(int b1 = 0;    b1 < 29; b1++)
	for(int b2 = b1+1; b2 < 30; b2++)
	for(int b3 = b2+1; b3 < 31; b3++)
	for(int b4 = b3+1; b4 < 32; b4++)
	{
		pattern = (patterns[p] ^ ((uint32_t)1 << b1 | (uint32_t)1 << b2 | (uint32_t)1 <<
b3 | (uint32_t)1 << b4));
		fwrite(&pattern, sizeof(uint32_t), 1, output);
```

```
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint32_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 28; b1++)
        for(int b2 = b1+1; b2 < 29; b2++)
        for(int b3 = b2+1; b3 < 30; b3++)
        for(int b4 = b3+1; b4 < 31; b4++)
        for(int b5 = b4+1; b5 < 32; b5++)
        {
                pattern = (patterns[p] ^ ((uint32_t)1 << b1 | (uint32_t)1 << b2 | (uint32_t)1 <<
b3 | (uint32_t)1 << b4 | (uint32_t)1 << b5));
                fwrite(&pattern, sizeof(uint32_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint32_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 27; b1++)
        for(int b2 = b1+1; b2 < 28; b2++)
        for(int b3 = b2+1; b3 < 29; b3++)
        for(int b4 = b3+1; b4 < 30; b4++)
        for(int b5 = b4+1; b5 < 31; b5++)
        for(int b6 = b5+1; b6 < 32; b6++)
        {
                pattern = (patterns[p] ^ ((uint32_t)1 << b1 | (uint32_t)1 << b2 | (uint32_t)1 <<
b3 | (uint32_t)1 << b4 | (uint32_t)1 << b5 | (uint32_t)1 << b6));
                fwrite(&pattern, sizeof(uint32_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint32_t), 1, output);
                count++;
        }
}
printf("%llu\n", count);
```

64 bit pattern generator

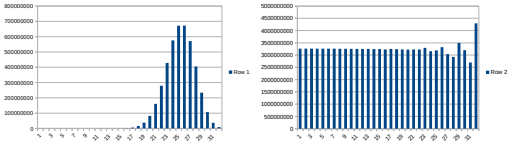
```
uint64_t patterns[] =
        {0x0000000000000000, 0x5555555555555555, 0x3333333333333333, 0x1111111111111111};
uint64_t count = 0;
uint64_t pattern = 0;
FILE* output = fopen("patterns64.bin", "wb");

for(int p = 0; p < 4; p++)
{
        pattern = patterns[p];
        fwrite(&pattern, sizeof(uint64_t), 1, output);
        count++;

        pattern = ~pattern;
        fwrite(&pattern, sizeof(uint64_t), 1, output);
        count++;

        for(int b1 = 0; b1 < 64; b1++)
        {
                pattern = (patterns[p] ^ ((uint64_t)1 << b1));
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 63; b1++)
        for(int b2 = b1+1; b2 < 64; b2++)
        {
```

```
                pattern = (patterns[p] ^ ((uint64_t)1 << b1 | (uint64_t)1 << b2));
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 62; b1++)
        for(int b2 = b1+1; b2 < 63; b2++)
        for(int b3 = b2+1; b3 < 64; b3++)
        {
                pattern = (patterns[p] ^ ((uint64_t)1 << b1 | (uint64_t)1 << b2 | (uint64_t)1 <<
b3));
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 61; b1++)
        for(int b2 = b1+1; b2 < 62; b2++)
        for(int b3 = b2+1; b3 < 63; b3++)
        for(int b4 = b3+1; b4 < 64; b4++)
        {
                pattern = (patterns[p] ^ ((uint64_t)1 << b1 | (uint64_t)1 << b2 | (uint64_t)1 <<
b3 | (uint64_t)1 << b4));
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 60; b1++)
        for(int b2 = b1+1; b2 < 61; b2++)
        for(int b3 = b2+1; b3 < 62; b3++)
        for(int b4 = b3+1; b4 < 63; b4++)
        for(int b5 = b4+1; b5 < 64; b5++)
        {
                pattern = (patterns[p] ^ ((uint64_t)1 << b1 | (uint64_t)1 << b2 | (uint64_t)1 <<
b3 | (uint64_t)1 << b4 | (uint64_t)1 << b5));
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;
        }

        for(int b1 = 0;    b1 < 59; b1++)
        for(int b2 = b1+1; b2 < 60; b2++)
        for(int b3 = b2+1; b3 < 61; b3++)
        for(int b4 = b3+1; b4 < 62; b4++)
        for(int b5 = b4+1; b5 < 63; b5++)
        for(int b6 = b5+1; b6 < 64; b6++)
        {
                pattern = (patterns[p] ^ ((uint64_t)1 << b1 | (uint64_t)1 << b2 | (uint64_t)1 <<
b3 | (uint64_t)1 << b4 | (uint64_t)1 << b5 | (uint64_t)1 << b6));
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;

                pattern = ~pattern;
                fwrite(&pattern, sizeof(uint64_t), 1, output);
                count++;
        }
}
printf("%llu\n", count);
```

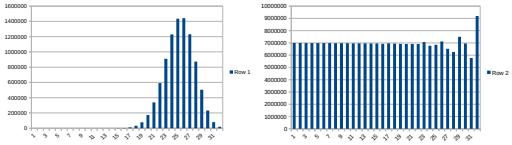
The above code generate a list of numbers in file. The 32 bit version contains 9 192 392 samples. The 64 bit version contains 666 220 258 samples. To validate the sample set is representative of the complete set, I ran the 32 bit set with left rotation using the sample set and compared it against the complete set. Here are some examples of the complete set against the sample set. As both end up plotting the same profiles on the graphics, we can assume the method to generate the sample set is representative of the complete set. We are now ready to move to the 64 bit set.

Set bit count distribution and Individual bit count distribution by left rotation offset
Left: complete set, right: sample set

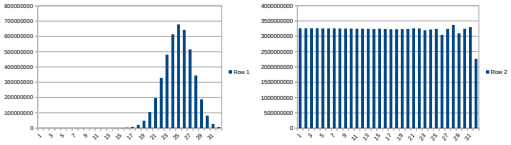
0c



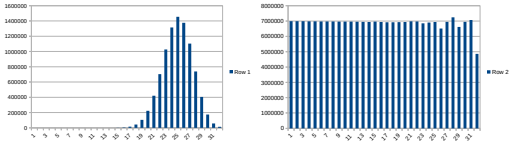
0s



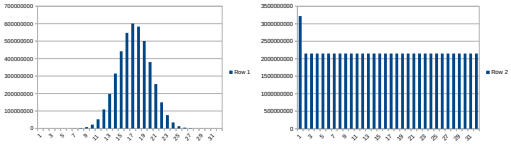
1c



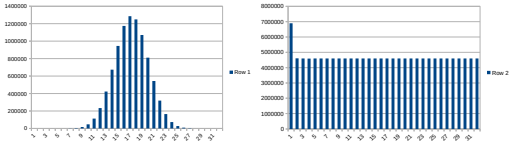
1s



31c



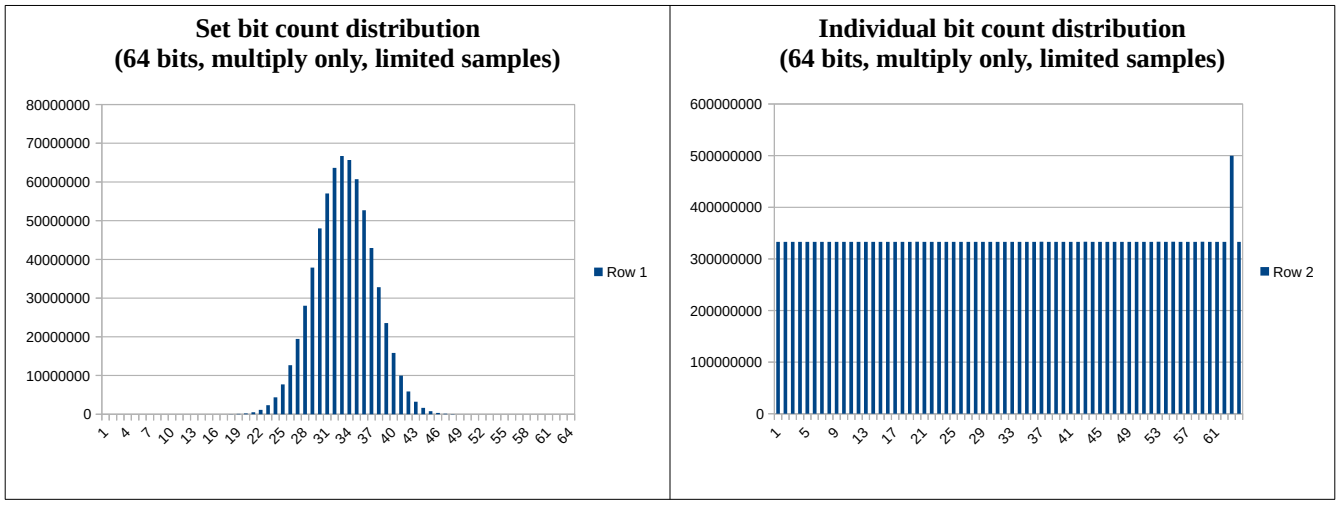
31s



Testing 64 bit vanilla algorithm

The first test I performed using the 64 bit samples is on the “vanilla” algorithm. This uses only the multiplication of prime numbers to generate the derivative. As we can see in the graphs, the bell curve is well centred and fall off quite nicely, however there is a bias on bit 1 to be more often set than not. Compared with the 32 bit version of the vanilla algorithm there are much less anomalies and of different nature. My hypothesis is the larger set of multiplications (64 vs 32) help to smooth it.

```
uint64_t nz_derivative_64_vanilla(uint64_t input)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709};
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719};

    for(int x = 0; x < 64; x++)
    {
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        mask <<= 1;
    }

    return output;
}
```



64 bit algorithm with progressive offset before the prime multiplication

To remove the bias of bit 1, I applied the method used on the 32 bits experiments consisting of doing a left rotation of the bits by n positions, where n is equal to the bit being processed. The bell curve is well centred, falls off nicely and not bits have a significant bias towards being set or not. This is the version of the algorithm I will keep and use.

```
uint64_t nz_derivative_64c(uint64_t input)
{
    uint64_t output = 1;
    uint64_t mask = 1;
    uint16_t primes_even[64] = {2, 5, 11, ..., 677, 691, 709};
    uint16_t primes_odd[64] = {3, 7, 13, ..., 683, 701, 719};

    for(int x = 0; x < 64; x++)
    {
        output = (output << x) | (output >> (64 - x));
        output *= ((input & mask) == 0) ? primes_even[x] : primes_odd[x];
        mask <<= 1;
    }

    return output;
}
```