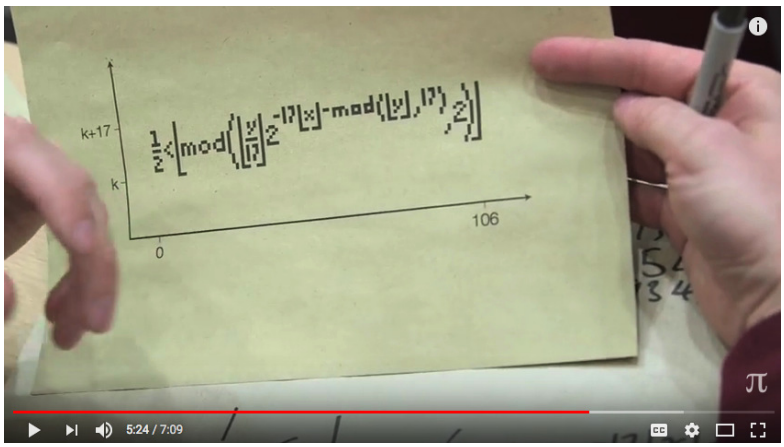
Tupper's Self-Referential Formula

10 October 2017

Jim Stevenson

Recently I viewed a startling video by Matt Parker about the Tupper Self-Referential Formula ([1]).



I found it difficult to fathom, so I looked it up on *Wikipedia* ([2]) and found the following explanation (with references omitted):

(https://en.wikipedia.org/wiki/Tupper%27s_self-referential_formula,)

Tupper's Self-Referential Formula

From Wikipedia, the free encyclopedia, 19 June 2017

Tupper's self-referential formula is a formula that visually represents itself when graphed at a specific location in the (*x*, *y*) plane.

History

The formula was defined by Jeff Tupper and appears as an example in Tupper's 2001 SIGGRAPH paper on reliable two-dimensional computer graphing algorithms. This paper discusses methods related to the GrafE formula-graphing program developed by Tupper.

Although the formula is called "self-referential", Tupper did not name it as such.

Formula

The formula is an inequality defined as:

$$\frac{1}{2} < \left\lfloor \mathrm{mod}\left(\left\lfloor \frac{y}{17} \right\rfloor 2^{-17\lfloor x \rfloor - \mathrm{mod}(\lfloor y \rfloor, 17)}, 2 \right) \right\rfloor$$

or, as plaintext,

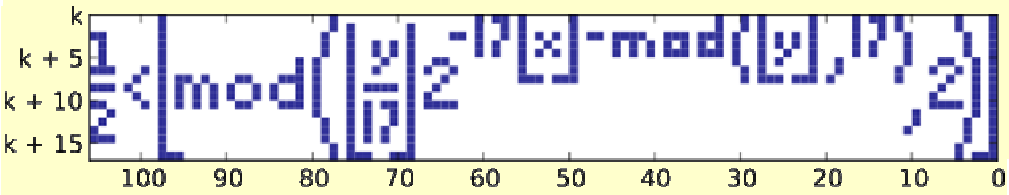
1/2 < floor(mod(floor(y/17)*2^(-17*floor(x)-mod(floor(y), 17)),2))

where ⌊ ⌋ denotes the floor function, and mod is the modulo operation.

Let *k* equal the following 543-digit integer:

960 939 379 918 958 884 971 672 962 127 852 754 715 004 339 660 129 306 651 505 519 271 702
802 395 266 424 689 642 842 174 350 718 121 267 153 782 770 623 355 993 237 280 874 144 307
891 325 963 941 337 723 487 857 735 749 823 926 629 715 517 173 716 995 165 232 890 538 221
612 403 238 855 866 184 013 235 585 136 048 828 693 337 902 491 454 229 288 667 081 096 184
496 091 705 183 454 067 827 731 551 705 405 381 627 380 967 602 565 625 016 981 482 083 418
783 163 849 115 590 225 610 003 652 351 370 343 874 461 848 378 737 238 198 224 849 863 465
033 159 410 054 974 700 593 138 339 226 497 249 461 751 545 728 366 702 369 745 461 014 655
997 933 798 537 483 143 786 841 806 593 422 227 898 388 722 980 000 748 404 719 461 014 655

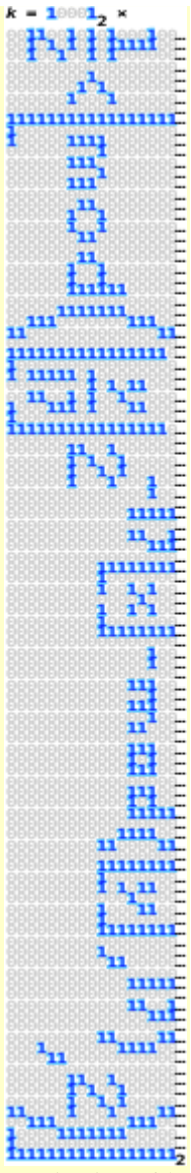
If one graphs the set of points (x, y) in $0 \le x < 106$ and $k \le y < k + 17$ satisfying the inequality given above, the resulting graph looks like this (the axes in this plot have been reversed, otherwise the picture would be upside-down and mirrored):



The formula is a general-purpose method of decoding a bitmap stored in the constant k, and it could actually be used to draw any other image. When applied to the unbounded positive range $0 \le y$, the formula tiles a vertical swath of the plane with a pattern that contains all possible 17-pixel-tall[1] bitmaps. One horizontal slice of that infinite bitmap depicts the drawing formula itself, but this is not remarkable, since other slices depict all other possible formulae that might fit in a 17-pixel-tall [and 106-pixel-wide] bitmap.[2] Tupper has created extended versions of his original formula that rule out all but one slice.

The constant k is a simple monochrome bitmap image of the formula treated as a binary number and multiplied by 17. If k is divided by 17, the least significant bit encodes the upper-right corner $(k, 0)$; the 17 least significant bits encode the rightmost column of pixels; the next 17 least significant bits encode the 2nd-rightmost column, and so on.

A video explaining this formula can be found here ([1]).



Derivation of k

I then checked all the references in the *Wikipedia* posting, as well as Google search results. The ones I reviewed in depth are listed in the References below.

After reading all the explanations in these postings, I think I have sorted out all the confusing aspects of the problem, which mainly derive from imbedding the desired bitmap into the base number k, which in turn is used as part of the indexing scheme into the plot. Also confusing was the various commentators' struggles with index limits. Perhaps that is due to the way bits are numbered (from 0) and to various coding indexing methods (beginning with 0 in C, 1 in Matlab, and implicitly 1 in Python). Finally, remaining discrepancies, in particular the number k, arose from different ways of assigning the bit values to the bitmap number that was multiplied by 17 to give k. (In fact, I use a different way from the *Wikipedia* article.)

Anyway, we have the following situation. We have a bitmap for a 106 x 17 section of the x,y plane written out as a number M. Following the matrix convention we have rows 1, 2, …, 106 and

[1] JOS: "and 106-pixel-wide"

[2] JOS: See my discussion below (p.5) in the Library of Babel Idea for when the bitmaps will repeat.

columns 1, …, 17. We start assigning a bit value of 1 for a pixel plotted and 0 for a pixel not plotted, beginning at the first row and first column. We progress across the 17 column positions of the first row and then move down to the next row and its first position, and so on. (See Figure 1)

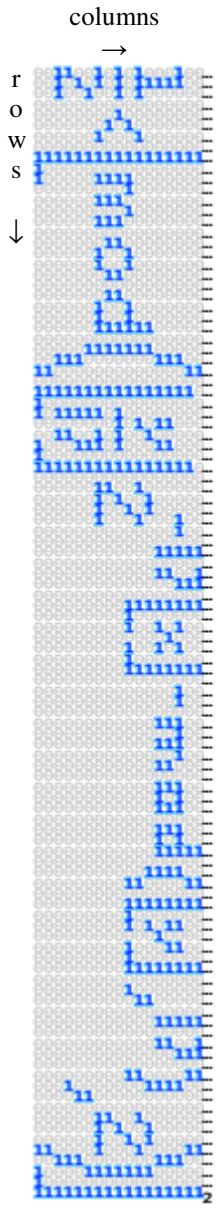


Figure 1

Now the binary number M is written generally as

$$a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_1 2 + a_0$$

where the $a_k = 0$ or 1, and n = deg M if $a_n \neq 0$. The least significant bit (LSB) is a_0. That is where we assign the pixel value from the first row and first column. We assign the pixel value from the first row and second column into a_1, and so on. So, for example, the 38th bit is at index 37 in M and satisfies the relationship 37 = 17*(row-1) + column = 17*2 + 3. Therefore, a_{37} represents the pixel value at row 3 and column 3. (Of course, any other indexing scheme, such as starting at the bottom right pixel and working from right to left, bottom to top (as in the *Wikipedia* article), will give a different number M and a differently oriented ultimate plot.)

Consider the Tupper Formula:

$$\frac{1}{2} < \left\lfloor \mathrm{mod}\left(\left\lfloor \frac{y}{17} \right\rfloor 2^{-17\lfloor x \rfloor - \mathrm{mod}(\lfloor y \rfloor, 17)}, 2 \right) \right\rfloor, \qquad (*)$$

As the discussions in the References have explained, it is really a scheme for retrieving the bitmap xy-plot as x varies from 0 to 106-1 = 105 (over rows 1 to 106) and y "essentially" varies from 0 to 17-1 = 16 (over columns 1 to 17). What "essentially" means is that y actually varies from N to N + 16 where N contains the bitmap M via N = 17*M. The term $\lfloor y/17 \rfloor$ equals M and remains constant while y varies across its range of 17 integers. By assuming integer values for x and y, we can eliminate the floor functions $\lfloor x \rfloor$ and $\lfloor y \rfloor$ in (*).

The negative of the exponent of 2, 17x + mod(y,17), effectively is providing an index k into the coefficients of M, that is

$$k = 17x + \mathrm{mod}(y, 17) = 17x + r$$

where r = 0, 1, …, 16 and x = 0, 1, …, 105. So k = 0, 1, …, 1801. Then $M*2^{-k}$ is just a right shift of k bits, k = 0, 1, …, deg M, and $\mathrm{mod}(M*2^{-k}, 2)$ produces the value of the LSB (a_k), namely 0 or 1. So asking if a_k is > ½, is equivalent to asking if $a_k = 1$. Notice that if k > deg M, then $2^k > M \Rightarrow 0 \leq \lfloor \mathrm{mod}(M*2^{-k}, 2) \rfloor < \lfloor \mathrm{mod}(1, 2) \rfloor = 1$. And so the subsequent bits are 0.[3] This procedure yields Figure 2.[4]

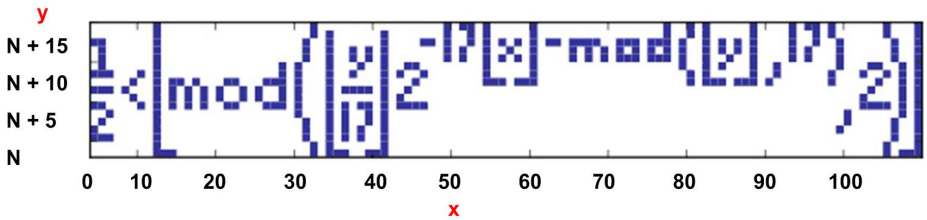


Figure 2 Bitmap Image in Stack of Plots

---

[3] Some of the commentators treated division by 2^k as a right shift and used the C operator >>. But for k > deg M, this is undefined according to *Wikipedia* (https://en.wikipedia.org/wiki/Bitwise_operation): "The result of shifting by a bit count greater than or equal to the word's size is undefined behavior in C and C++."

[4] Figure 1 and Figure 2 are modified from the ones in the *Wikipedia* article with k now set to N.

Simple Example

Suppose we consider a smaller dimensioned plot, say H x W = 5 x 10. Let's see what a bitmap number of M = 57 means. Consider Figure 3.

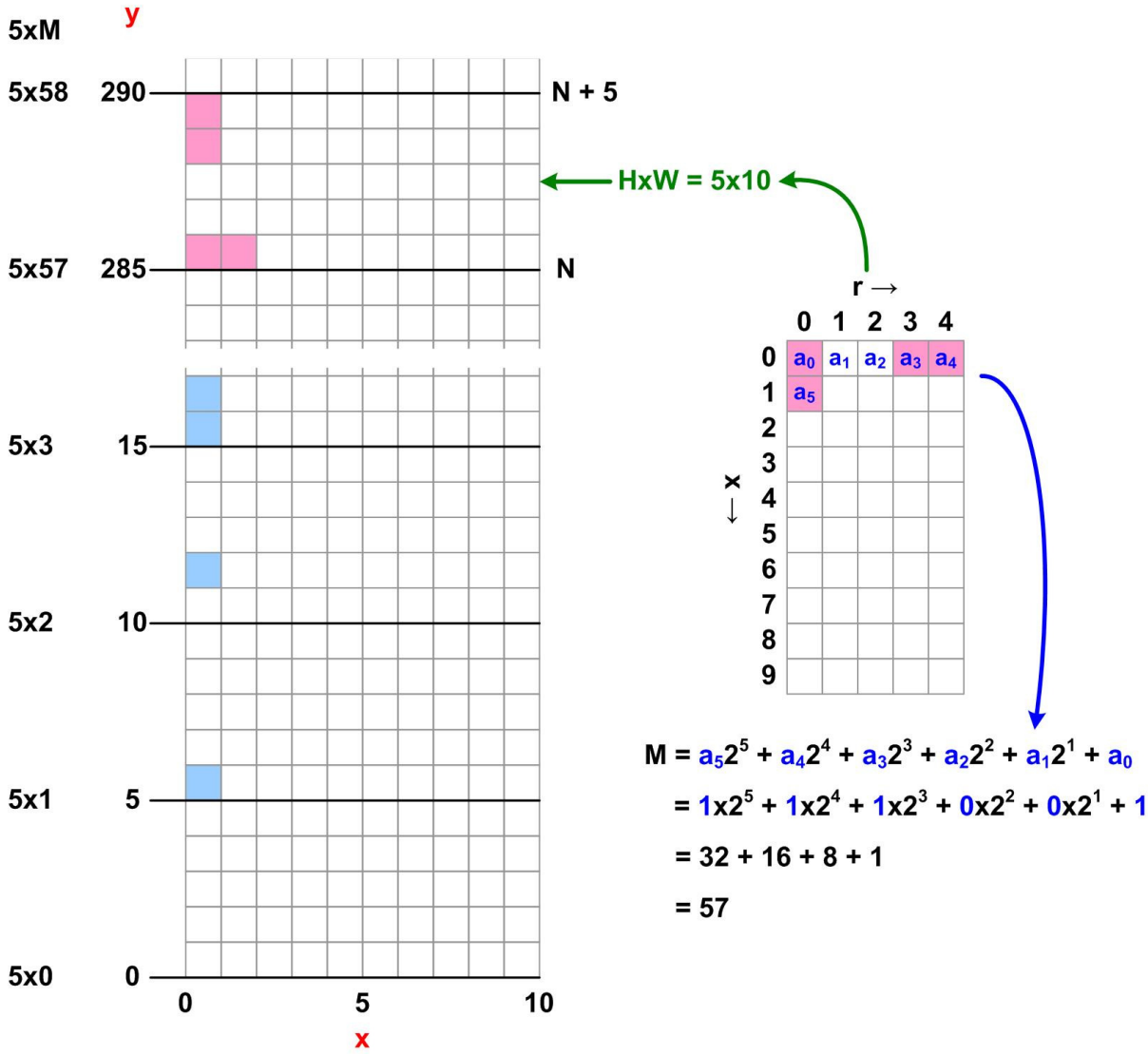


Figure 3 Simple Example with H = 5 and W = 10

We see that the bitmap corresponding to 57 (shown in the right-hand grid) is mapped into the stack of bitmap plots at the y = 5 * 57 = 285 level. Notice that the first, bottom plot in the stack is blank because all the coefficients of M = 0 are 0. The next higher plot has only one pixel turned on, since M = 1 means only $a_0 = 1$. The next plot still has only one pixel turned on, since M = 2 means only $a_1 = 1$. But the next plot has two pixels turned on, since M = 3 means $a_0 = 1$ and $a_1 = 1$. And so on.

Notice that the largest value of y for this grid size is $y = H * (2^{\max \deg M+1} - 1) + H = H * (2^{H * W}) = 5 * 2^{50}$. (A fuller explanation of this is provided in the next section.) Now $2^{10} \approx 10^3$ means $5 * 2^{50} \approx 5 \text{ x } (10^3)^5 = 5 \text{ x } 10^{15} = 5$ million billion. So there are a million billion 5 x 10 plot patches needed to cover all possible bitmap cases. This leads to the following computations for the original Tupper

Formula plot.

Library of Babel Idea

If we return to the original Tupper formula and dimensions, since there are H x W = 17 x 106 = 1802 pixels in the plot, we have that the maximum possible degree of M is n = 1802-1 = 1801. Therefore, M is any number from 0 to 2^{1802}-1. Each of these numbers produces a 17 x 106 bitmap of a plot. So M can be thought of as an index into a stack successive 17 x 106 plots. For each increment of M, we get a new plot, starting at y = 17*M. That is, if y = N = 17*M, then the next plot starts at y = 17*(M+1) = N + 17. Therefore the maximum value for y before it repeats the plots is $17*(2^{1802}-1) + 17 = 17*2^{1802}$.

Now $2^n = 10^m$ means $m = n \log_{10}2$. So $m = 1802* \log_{10}2 = 542.456$ and $\log_{10}17 = 1.230$. Therefore $17*2^{1802} = 10^{543.8} \approx 10^{544}$, which is huge — effectively infinite. For example, one trillion = 10^{12} and 10^{544} = 10,000 * $(10^{12})^{45}$ = 10 thousand * (1 trillion)^{45}. These numbers are unimaginable.

So yes, Tupper's Formula will produce any and all possible 17 x 106 plots, like Borges's "Library of Babel." This means every word in every book, written or to be written, in any language, will appear in some 17 x 106 plot patch (if it will fit), though not in consecutive order, that is, all the words of a given book will be scattered "randomly" throughout the patches, possibly miles apart.

If we assume 85 dpi, then 5 patches will fit in one inch of y values. So one mile has 5280*12 = 63,360 inches or 316,800 patches in it. Then 3 miles has about a million patches. But 1 trillion = 1 million * 1 million. So 3 million miles has a trillion patches in it. One lightyear is about 6 million miles or about 2 trillion patches. According to *Wikipedia*,[5] the farthest we can see in the observable universe is about 47 billion lightyears. Let's call that "one universe". So 1 universe = 47 x10^9 lightyears = 47 x 10^9 * 2 trillion patches ≈ 100 billion trillion patches = 10^{23} patches. Therefore, the total number of patches, 10^{544}, would take 10^{544}patches/(10^{23} patches/universe) = 10^{521} universes = 100 thousand * (1 trillion)^{43} universes. So the percentage of patches that could span our universe is the proverbial "drop in the bucket".

References

1. Matt Parker, "The 'Everything' Formula – Numberphile"15 April 2015 (https://www.youtube.com/watch?v=_s5RFgd59ao) and "The 'Everything' Formula (extra bits) – Numberphile" (https://www.youtube.com/watch?v=wx22jdwn5zQ)
2. "Tupper's Self-Referential Formula," *Wikipedia* 19 June 2017 (https://en.wikipedia.org/wiki/Tupper%27s_self-referential_formula)
3. Arvind Narayanan, "Tupper's Self-Referential Formula Debunked," 22 June 2011 (http://arvindn.livejournal.com/132943.html)
4. R. Shreevatsa, "How Does Tupper's Self-Referential Formula Work?" 12 April 2011 (https://shreevatsa.wordpress.com/2011/04/12/how-does-tuppers-self-referential-formula-work/)
5. Ron Avitzur, "The Library of Babel function," January 23, 2007 (http://avitzur.hax.com/2007/01/the_library_of_babel_function.html)
6. K.V. Mohan, "Quines of the World Unite!" January 19, 2007 (http://mohankv.blogspot.com/2007/01/quines-of-world-unite.html)

---

[5] https://en.wikipedia.org/wiki/Observable_universe