Evaluate Your FRACTRAN Programs 10x Faster Using this One Weird Trick

Math 490, Math Expositions, VCU

Stuart Geipel

December 8, 2017

Abstract

FRACTRAN is an esoteric programming language designed by John Conway which consists of a sequence of fractions. Existing literature has shown FRACTRAN programs to be equivalent to register machines, however no other techniques have been produced to analyze and optimize the evaluation of arbitrary FRACTRAN programs. Our main result, proving conditions in which a cycle of fractions is guaranteed to occur, which are prevalent in practical FRACTRAN programs, improves existing techniques by a factor of 10 or more for real-world inputs. A method of reducing the number of fractions to multiply is also demonstrated.

1 Introduction

In 1972, the mathematician John Conway[1] introduced FRACTRAN at the annual Number Theory Conference in Boulder, Colorado. During the 70s and 80s several properties were proven, among them Turing completeness, and programs to do a variety of things such as calculating primes were created. Many of these results were from Conway himself.

A FRACTRAN program consists of an ordered sequence of fractions. This is a program[1]

[1]Mike Stay – golem.ph.utexas.edu/category/2006/10/puzzle_4.html#c005735

which computes the Hamming weight of the input (we'll analyze simpler programs later):

$$\frac{3 \cdot 11}{2^2 \cdot 5}, \ \frac{5}{11}, \ \frac{13}{2 \cdot 5}, \ \frac{1}{5}, \ \frac{2}{3}, \ \frac{2 \cdot 5}{7}, \ \frac{7}{2}$$

FRACTRAN can be viewed as a slightly more general Turing machine[2] in that it accepts an integer both as input and output, rather than outputting a boolean – equivalence can be shown by picking two numbers to halt on rather than the range of integers. In fact, the problem of whether a FRACTRAN program halts for all integers is Π_2^0-complete [4], which is equivalent to Turing machines. The integer input to a FRACTRAN program is also its initial state – the result of evaluating a FRACTRAN program is a potentially infinite sequence of integer states (if the program doesn't halt). This is because the evaluation of a program proceeds as follows:

1. The program is given an input integer $n \in \mathbb{N}$. This becomes the initial state.
2. To find the next state, replace $n \coloneqq qn$, where $q \in \mathbb{Q}$ is the first fraction in the program such that $qn \in \mathbb{N}$
3. Repeat step 2 until no such q exists, then halt on n.

In fact, implementing it in Haskell (or any other language) is very simple:

```haskell
naive fs n = maybe [] (next . numerator) match where
  ps = map (*fromIntegral n) fs
  match = find ((==1) . denominator) ps
  next p = p : naive fs p
```

The idea of doing computation under this model may seem confusing, but this is in fact as powerful as an ordinary Turing machine or even your own computer, in theory. Conway devised a prime number generator in which the sequence of states which are powers of two have prime exponents – each prime, in order. He called this program PRIMEGAME, and it consists of only 17 fractions. For programs that have infinite output, filtering the sequence of states for a specific conditon (such as being a power of 2) and looking at the output is common (rather than halting on something).

A simple program to add two numbers is as follows:

$$\frac{5}{2}, \frac{5}{3}$$

Continuing on the theme of powers, this program accepts the input $2^a 3^b$ and outputs 5^{a+b}. It first "moves" powers of 2 over to 5, then powers of 3. The first fraction could be replaced with $\frac{3}{2}$ and be equivalent, but take a more steps. This is because powers of 2 will first move to 3, then to 5 along with the original powers of 3. In a similar but slightly more nuanced way, the second fraction can be replaced with $\frac{2}{3}$ to yield the same output. In this case, first powers of 2 will move to 5, then for powers of 3, it will alternate between using $\frac{2}{3}$ and $\frac{5}{2}$ – this is because $\frac{2}{3}$ "re-enables" $\frac{5}{2}$ to multiply and result in an integer again, which will always occur first due to its precedence in the order of fractions.

The ratio between consecutive states will always be one of the fractions in the program. Exponentials are a recurring theme in FRACTRAN for related reasons, thanks to the Fundamental Theorem of Arithmetic. Due to the theorem, to halt on a number n, the prime factors of n have to be either in the original input, or in the numerators it multiplies by in the FRACTRAN program. It actually goes deeper than this – a FRACTRAN program can be written as a register machine where the registers correspond to prime factors and their values to their powers. Most of the time registers and prime factors are interchangeable in language.

Conway writes the registers as r_p, where p is a prime number. Unfortunately, this register machine is only capable of addition and subtraction by constant values based on conditions. Multiplying and dividing by prime factors adds or subtracts from the current state by the power of that prime factor. So the former adding program would take $O(a+b)$ time, because it consists of two steps, both requiring decrementing r_2 or r_3 until they're 0. Furthermore, the simple addition program destroys r_2 and r_3 to create r_5, much like addition in the esoteric programming language Brainfuck. A more complicated example which preserves the input using more registers is as follows:

$$\frac{7}{11}, \quad \frac{5 \cdot 13 \cdot 11}{2 \cdot 7}, \quad \frac{5 \cdot 17 \cdot 11}{3 \cdot 7}, \quad \frac{1}{7}, \quad \frac{2}{13}, \quad \frac{3}{17}$$

This takes in $2^a 3^b 7$ as input, outputting $2^a 3^b 5^{a+b}$. Since destroying r_2 and r_3 is unavoidable, it produces $5^{a+b} 13^a 17^b$ as an intermediate state. Then, it moves $r_{13} \to r_2, r_{17} \to r_3$ to restore the original state and halt on $2^a 3^b 5^{a+b}$. So, it's natural to ask what r_7 and r_{11} are doing, which haven't been mentioned: They serve as a state boundary to separate adding to r_{13} and r_{17} from subtracting to them. Until r_2 and r_3 are depleted, it will oscillate between having a factor of 7 and 11, due to the first fraction. Then the state change happens once it can reach 1/7.

To make matters more complicated, for the same reason FRACTRAN programs can't in general halt on arbitrary integers, finite-length FRACTRAN programs always have a finite number of registers. For the sake of simplicity, this article will only consider finite FRACTRAN programs, but the techniques used here will generally work on infinite ones as well. However, the practical consequence of writing a finite FRACTRAN program is that potentially unbounded state needs to fit in a finite number of registers. So, to simulate a tape, for example, FRACTRAN states must become extremely large per register.

There are two primary results provided – the first is a method of statically reducing the fractions that could be q. The second is a method of detecting certain types of cycles in state which are common in FRACTRAN programs. It effectively turns the innermost loop of many FRACTRAN programs into $O(1)$ operations, where an operation is considered to be the number of steps the state goes through, similar to time spent in the real world. In some cases this reduces algorithms by a linear factor (addition becomes $O(1)$ rather than $O(a + b)$, naive multiplication becomes $O(a)$ rather than $O(ab)$, and so on. (For simplicity's sake we avoid the additional logarithmic factors that are technically present in all of these expressions, considering arithmetic/fraction testing as constant-time operations.) This is possible because these FRACTRAN algorithms are inefficient in the first place – looking at it a different way, these inefficiences can be mended to match real computers because sometimes they're predictable. As an example, we can look at the Hamming weight program, which effectively divides by 2 repeatedly and counts up the remainders. The way it does this is less efficient than real computers – it instead attempts to decrement by 2 repeatedly, counting ones left over. Cycle detection sees and replaces this decrementing loop with actual division operators.

The result of the improved FRACTRAN interpreter can be very significant. There were 3 sequential improvements over the naive implementation: first modifying it to be register-based, then reducing tested fractions, then detecting cycles. The main benchmark used was the well-known PRIMEGAME prime-calculating function, run on the first 100 primes. The interpreter was halted after it found the 100th prime. The only non-naive, factorization-based interpreter the author could find online, written by Nick Johnson[2], is likely effectively the state-of-the-art. Implemented in Python, it was found to complete the benchmark in 4 minutes, 28.5 seconds. Our factorization-based interpreter was significantly more performant, completing in 1 minute 36 seconds. Yet the best (cycle-based) interpreter completed in 8.0 seconds, over 30 times faster than the reference Python implementation. For larger amounts of primes, the difference grows due to better time complexity. Our naive implementation, which is a fairly performant implementation of the naive algorithm which most people write, finished in 7 minutes 18.9 seconds – about 55x slower than the cycle detector.

However, for essentially best-case scenarios, such as Mike Stay's Hamming weight calculating program, the cycle-based implementation performs thousands of times faster for large inputs. Our register-based implementation would complete in 11.4 seconds for 24 digits, yet

[2]StackOverflow - Reference implementation in Python: `stackoverflow.com/a/1780262`

the cycle-based implementation kept completing essentially instantly until we increased the digits by 3 orders of magnitude.

2 FRACTRAN as a Register Machine

In order to use the earlier reconceptualization of FRACTRAN as a register machine, it must be formalized. Define $[l] = \{1, 2, \ldots, l\} = \{i \in \mathbb{N} \mid i \leq l\}$. This set will be used frequently to define finite sequences. Next, we'll create a function R to extract the registers from a particular value that can be used in both the state and inside fractions. It will be used frequently enough that we'll introduce special syntax to avoid nested parentheses. Let P be the set of primes, and $R : \mathbb{N} \times P \to \mathbb{N}$ retrieve a prime factor's multiplicity such that

$$n = \prod_{p \in P} p^{R(n,p)} .$$

For example, $R(75, 5) = 2$, because $75 = 3 \cdot 5^2$. For any $n \in \mathbb{N}$, $\{p \in P \mid R(n, p) \neq 0\}$ is finite, meaning almost all of our (infinite number of) registers are always 0.

Let $(a_i/b_i)_{i \in [l]}$ be a finite sequence of *reduced* fractions of length l which represent a FRACTRAN program. This means a and b are sequences of numerators and denominators, respectively, which is an important distinction. For the evaluation of the program itself, let $(n_i)_{i \in [k]}$ be a sequence of integers (representing the progression of state) for some $k \in \mathbb{N} \cup \{\aleph_0\}$, meaning this sequence could be infinite or any finite length. Under this representation n_1 is the input to the program and its last element is its output. For the rest of the article, a, b, and n will always refer to these sequences.

For integer sequences such as a, b, and n, we will denote accessing register p of the ith element of the sequence as

$$a_{i,p} = R(a_i, p) ,$$

which will be helpful in quickly using R.

Finding the next state in a program consists of testing against up to l fractions; in other words, multiplying n_i with them and testing if it results in an integer. Since an integer times a rational is always rational, the only way this product could be in non-integral is for the fraction's denominator to have to have some register greater than n_i's, resulting in the result having a denominator above 1. This brings us to the register machine-based definition of FRACTRAN, which lends itself to quick evaluation:

$$n_{i+1} = n_i \frac{a_j}{b_j} \ ,$$

where j is the index of the first fraction such that

$$\forall p \in P : n_{i,p} \geq b_{j,p} \ ,$$

which can ignore a_j because the fraction is reduced, i.e. $\forall p \in P : \neg \left[a_{j,p} > 0 \land b_{j,p} > 0 \right]$. Due to power laws, this recurrence has the property

$$n_{i+1,p} = n_{i,p} + a_{j,p} - b_{j,p}$$

which can be used for efficient evaluation by computers.

From this perspective, we can represent n_i, a_i, and b_i all as maps from primes to integer powers. This change is the key the speed of both Nick Johnson's and our implementation, because it operates with the registers separated rather than rendundantly performing overlapping computations when entire fractions are multiplied.

Putting this all together, we can create a significantly faster evaluation function than naively using Haskell's arbitrary precision arithmetic, which is already fast (and even wins for very small inputs sometimes). In fact, one of the big reasons the difference isn't larger is because of how performant Haskell's arbitrary precision arithmetic is in the first place. When it beats the naive method, it's because it replaces huge multiplications and GCD operations (on numbers which comprise of all the registers at once) with simple addition and comparsions of smaller, separate registers. The data storage is logarithmic to the size of the registers (because they represent powers) instead of linear.

3 Reducing Fraction Tests

The first new result of this paper offers a method of statically determining conditions when certain fractions will never be evaluated and thus don't require testing for whether its product could result in an integer. This has a memory cost of $O(l^2)$, quadratic to the number of fractions, which is typically dwarfed by the state of the program. We achieve this by representing possible consecutive sequences of 2 fractions as a directed graph with l vertices (representing the fractions) and edges representing transitions from one fraction to the next. If you plot the sequence of fractions used by the program, it will always travel along vertices of this graph.

More formally, we say each state n_i has a fraction a_j/b_j accompanying it, the one which was multiplied by n_{i-1} to result in n_i. If we're looking at the state n_i, we denote the "source vertex" to be this fraction – and completing the analogy, $n'_{i+1}s$ source vertex is n_i's destination vertex. So when we plot pairs of source/destination vertices (the pairs representing directed edges) that it uses as the state progresses, we plot its movement along the graph.

The looped complete graph with l vertices represents all possible combinations. This corresponds to the naive case of checking (up to) all l destination fractions (again, to see if the result is an integer and qualifies to be the next state). Doing this, we effectively are saying we think any state transition is possible. However in practice, a good portion of these edges are never used and cannot by used based off the program definition. In real programs, the possible transitions of state are usually linearly proportional to the states themselves. It takes a human to fully analyze this graph, but some of the intuition used can be captured by an algorithm. If edges can be removed, since these correspond to source vertices/destination vertices, if we already know the source fraction for the state n_i, we only need to test what edges are left as possible destination fractions. Storing this adjacency matrix costs the $O(l^2)$ memory.

So the question becomes, how do we eliminate edges from this graph? The answer lies in the fact that reaching a certain fraction guarantees it already eliminated denominators coming before it in order of iteration. Our result can eliminate up to the $j-1$ fractions preceding it in the program given a_j/b_j as the source vertex – resulting in a modest constant factor of 1/2 as the maximum fraction of removals. However, this can be critical because many real-world cycles come from adjacent fractions, meaning that most or all of the $j-1$ preceding redundant fractions can be eliminated from the testing process. In practice, this can result in a performance gain of 5 times more for large programs, such as Anh H. Trinh's self-interpreting FRACTRAN implementation[3], which uses 84 fractions.

Theorem 1 (Transition Elimination)**.** *Suppose the source fraction whose product resulted in n_i is a_j/b_j. Then the destination fraction will never be a_k/b_k for $k \in K$ where*

$$K = \{k \in [j-1] \mid \forall p \in P : a_{j,p} > 0 \implies b_{k,p} = 0\}$$

This looks like a mouthful, but it essentially says that for every fraction a_k/b_k preceding our source fraction a_j/b_j in the program (i.e. $k < j$), it's possible to eliminate it if a_j never adds to a register which is nonzero in b_k. That's because (from the contrapositive route) if $b_{k,p}$ is nonzero, it's only possible for fraction k to be a destination if our source numerator $a_{j,p}$ adds to $n_{i,p}$ (is greater than zero).

Proof. Suppose we reached n_i through the source fraction a_j/b_j, and our next k is actually in K. Since $k < j$, this fraction was not picked last time, i.e. $\exists p \in P : n_{i-1,p} < b_{k,p}$. Since registers are non-negative, this implies $b_{k,p} > 0$, which, by the contrapositive of our set

[3]FRACTRAN interpreter in FRACTRAN — `stackoverflow.com/a/1802570`

builder condition, implies $a_{j,p} = 0$. Therefore

$$n_{i,p} \leq n_{i-1,p} < b_{k,p} \ ,$$

meaning n_i doesn't satisfy fraction k either, or any other fraction with that denominator – i.e., fraction k is not the destination fraction. □

The consequence of this is that we can substantially reduce the set of fractions tested per step. For fraction j, this can result in an upper bound of $j - 1$ times fewer tests per step, which for some types of programs that exploit this presents an asymptotic gain in speed relative to the length of the program. This can be implemented as an array of fraction lists, in which the keys are source fractions and values the list of connected destinations (in order of testing).

Even with programs having very short fraction lengths such as PRIMEGAME, this has a slight (measured as about 32.4%) performance benefit.

4 Cycle Detection

The most important aspect of algorithm optimization is improving time complexity. While the register machine interpretation can yield a good speedup on its own, the terms it replaces in the time complexity already have modest real-world coefficients. We won't get the 10 times (or greater) improvement we're looking for on real-world programs with just that alone, and the benefit won't scale with larger input either. A very valuable improvement would be to get rid of the steps themselves – this is what cycle detection does for us.

As a simple example, take the program

$$\frac{5}{2}, \ \frac{2}{3} \ .$$

As we know from earlier, this accepts $2^a 3^b$ and returns 5^{a+b}. If you asked someone who just learned how FRACTRAN works to evaluate this on $2^9 3^9$, they'd start by doing a few steps by hand to understand the pattern, yielding $2^8 3^9 5, 2^7 3^9 5^2$ and so on. They would notice that the first fraction repeats until 2 runs out, and "skip" some steps. Cycle detection performs this automatically by remembering previous steps. Because it remembers a history, even for longer fraction cycles (as we know from earlier, this program moves $r_3 \to r_5$ by alternating between the first and second fraction) it will see the state matches the step before last, detecting a cycle of length 2.

We'll prove that under certain conditions like these, we know exactly what sequence of fractions will repeat being used, for at least a certain lower bound of loops. This allows us to calculate with just arithmetic the state after all those loops complete. The preconditions are common enough in the real world that many FRACTRAN algorithms are sped up by a linear factor. In practice, there will be some "leftover" from the same cycle before and after the skip is done which are manually evaluated: a single initial cycle is manually evaluated while the interpreter builds a history of state, and the trailing part (after the skip) is left over because it sometimes can't complete a full cycle before the registers are depleted.

This is fine though, because as long as the portion of leftovers relative to the size of the skip goes to 0 for large inputs, it doesn't matter. What we really want is the change in time complexity – and since the previous algorithm already works by evaluating manually, it doesn't have a large cost associated with doing so.

Obviously, the states we're forming "cycles" from aren't actually identical. In the first example, r_2 is decreasing while r_5 increases. The main insight here is that for most of the steps while r_2 goes to 0, both r_2 and r_5 (the ones that change) are too large for any of the fractions to actually have conditions based on their contents. This relates back to the fact that conditional logic in FRACTRAN is created by making registers in n_i small enough (comparable to the denominators) that they may or may not pass over a fraction based on how large they are.

So, the state is partitioned into two groups of registers based on whether the registers are too large to for conditional logic, the progression of state, to depend on them. We call these two groups the data and logic registers, corresponding to registers being larger or smaller than any denominator's, respectively. Combining the data and logic registers together again produces the real state. The (large) data registers have the critical property that every denominator has a smaller value for that register, meaning it could never be the reason a fraction isn't chosen as a source.

So, cycle matches are found based on whether the logic registers match for that state. When a cycle is found, the differences in the data registers (which never affect the cycle by definition) is calculated, which is the rate of change per cycle. Then, a "skip" is performed, leaping forward arithmetically based on how many cycles we can perform until some data register with a decreasing difference runs out, or in other words, would become a logic register. This way, we only perform the number of cycles which are guaranteed to safely correspond to our register state.

Our first step towards formalizing this is to put these ideas into symbols. We'll reuse the idea of the maximum value of a prime factor p (or the register) for all denominators, so we'll denote that as

$$m_p = \max_{i \in [l]} b_{i,p} \ .$$

Now let $F_i \subset P$ be the prime factors of n_i. Let $D_i \subseteq F_i$ be the set of "data" factors (the set of registers) – that is, prime factors which are so large no condition can depend on it:

$$D_i = \{p \in F_i \mid n_{i,p} \geq l m_p\}$$

Lastly, define the logical state L_i as what's left – the small prime factors – or $L_i = F_i \setminus D_i$. Lastly, we'll define state retrieval for a set of factors as

$$S(n_i, F_i) = \{(p, n_{i,p}) \mid p \in F_i\}$$

which we'll use to compare state.

Theorem 2 (Cycle evaluations)**.** *Suppose $S(n_i, L_i) = S(n_j, L_j)$, $i > j$. Then the next $(i-j)c$ states will use c cycles of the same sequence of program fractions, and the logical states $S(n_j, L_j)$ through $S(n_i, L_i)$ – where* [4]

$$c = \min \{\text{life}_p \mid p \in D_i\} \cup \{\aleph_0\} ,$$

$$\text{life}_p = \begin{cases} \aleph_0 & \text{if } n_{i,d} = n_{j,d} \\ 0 & \text{if margin}_p < 0 \\ \aleph_0 & \text{if } n_{i,d} > n_{j,d} \\ \left\lfloor \frac{\text{margin}_p}{n_{j,d} - n_{i,d}} \right\rfloor & \text{if } n_{i,d} < n_{j,d} \end{cases} , \textit{ and}$$

$$\text{margin}_p = \min \{n_{j+k,p} \mid k \in [i-j]\} - m_p .$$

The set we're taking the minimum of consists of the "cycles left" for each register before it becomes a logical register. The first prime factor to run out is the one that limits our skipping behavior, so the minimum comes from this. The numerator of the fraction is the margin before it becomes a logic register, and the denominator is the rate of change per cycle. If a data register stays constant or increases, it doesn't limit us, so we don't use them in the set ($n_{j,d} - n_{i,d} > 0$). If no registers decrease, our set will be empty, resulting in nontermination, or $c = \aleph_0$.

It's useful to note before we begin that's it's possible for 0-length "cycles" to be found. These aren't useful but are discovered along with the rest. In order to avoid an infinite loop in this situation, the history of states is cleared every skip, which naturally resolves it.

First, we'll prove a lemma that shows if the data registers don't become logic registers (by becoming smaller) that the destination fraction will stay the same Next, we'll prove Theorem 2 by showing that the data registers *won't* become logic registers for at least c cycles, or that if they do, that it's an identically periodic behavior each cycle which doesn't require the lemma, which we'll elaborate on later.

[4]To match the cases we created for life_p, the first one to be true in the list is always picked first. (It's possible for both $\text{margin}_p < 0$ and say, $n_{i,d} > n_{j,d}$)

Lemma 3. *The state n_i's destination fraction is the same regardless of what D_i's register values are, so long as each register r_p is not low enough to leave D_i (so long as $n_{i,p} \geq m_p$).*

This implies data registers have no effect on the destination fraction. It's clear that the logical registers (L_i) must be responsible, since if they weren't there would be no registers left that could be changed to change the destination fraction.

Proof. Suppose the destination fraction index is j, and (for contradiction) changing registers in D_i results in the index changing to k. WLOG, set $j < k$. Then because of this change, fraction j no longer met the conditions/resulted in an integer product and it moved forward to k. In other words,

$$\exists p \in P : b_{j,p} > n_{i,p} \wedge b_{k,p} \leq n_{i,p}$$

which is the reason it was rejected. However, by definition $m_p \geq b_{j,p}$, so

$$m_p \geq b_{j,p} > n_{i,p} \ .$$

Therefore, $n_{i,p}$ isn't a data register in the first place ($p \in L_i$), which contradicts our assumption that $p \in D_i$. $\square$

Now, we have the main tool we will use to prove Theorem 2.

Proof of Theorem 2. By Lemma 3, we know logic registers must be responsible for any deviations in the cycle. Since $S(n_i, L_i) = S(n_j, L_j)$, the only change is in the data registers, which we need to prove that these never become logic registers (or that even if they do it wouldn't change the outcome). We will prove inductively that this continues to be the case. There are two ways a data register can become a logic register during the cycle – the first is that they could "dip" temporarily and become a logical registers despite being data registers at the bounds of the cycle (n_i and n_j). The other is that a data register can simply deplete over time and eventually become a logic register.

So, we split into cases for each individual register r_p for $p \in D_i$. Case 1 is if $n_{i,p} = n_{j,p}$, in which case, since the register doesn't change between cycles, this register would satisfy the same conditions indefinitely, a length of $\aleph_0$. This ensures that even if there's a dip, which would violate Lemma 3's conditions, it would trigger the same logic each cycle.

Case 2 is if the register is changing: $n_{i,p} \neq n_{j,p}$. Since we can't guarantee logical state wouldn't be affected if there was a temporary dip, we set $c = 0$ if during the cycle it drops into logical registers temporarily. (We will cover how this is calculated and makes its way into the theorem later.) Setting $c = 0$ makes the statement vacuously true and is effectively a way of saying our theorem can't make any guarantees in this situation. If this situation doesn't occur (if it's data registers throughout), then there are two final subcases.

Subcase 1 is if $n_{i,p} > n_{j,p}$. Since r_p already satisfies the lower bound for being in D_i, increasing each cycle results in it never leaving D_i. r_p will always satisfy Lemma 3, so we have a minimum of $\aleph_0$.

Subcase 2 is (predictably) if $n_{i,p} < n_{j,p}$. This case is of concern because outside the various conditions we've previously covered which invalidate cycle detection and set $c = 0$, this will be the limiting factor that actually determines how long the cycle is. Essentially, the job here is to calculate the "burn rate" each cycle and divide what margin we have by that – because we can only guarantee Lemma 3 will be satisfied for that long.

It happens that a "dip" into logical state corresponds directly to having a negative margin. We define the margin as the minimum value the r_p reached that cycle minus m_p:

$$\text{margin}_p = \min \{n_{j+k,p} \mid k \in [i - j]\} - m_p \ .$$

As a result, if the cycle started with that register diminished by $1 + \text{margin}_p$ (having a margin of -1), it would become a logic register for at least one step in the cycle, which has the possibility of changing conditional logic.

So, we can safely set c to however many cycles would avoid producing a negative margin, which is

$$\left\lfloor \frac{\text{margin}_p}{n_{j,d} - n_{i,d}} \right\rfloor .$$

Combining these conditions together we can construct the expression for the "life" of a data register r_p, a lower bound for how many cycles it can be guaranteed to last:

$$\text{life}_p = \begin{cases} \aleph_0 & \text{if } n_{i,d} = n_{j,d} \\ 0 & \text{if margin}_p < 0 \\ \aleph_0 & \text{if } n_{i,d} > n_{j,d} \\ \left\lfloor \frac{\text{margin}_p}{n_{j,d} - n_{i,d}} \right\rfloor & \text{if } n_{i,d} < n_{j,d} \end{cases}$$

The final cycle length will be the quickest register to lose our guarantees, via a minimum:

$$c = \min \{\text{life}_p \mid p \in D_i\} \cup \{\aleph_0\} .$$

The final union with $\aleph_0$ is because if $D_i = \varnothing$ then the state is identical by definition and determinism forces an indefinite cycle. Thus, we've shown the consequent of Theorem 2. $\square$

It seems like the conditions for cycle detection are very specific. So, knowing this, why is the precondition still so common in real-world FRACTRAN programs? It's because real programs have a finite amount of registers and a finite amount of fractions, but an unbounded amount of data and computation steps. Pathological programs could have no short/useful cycles of L but they're harder to construct in the first place. PRIMEGAME for example consists of basic loops to check divisibility that could be optimized.[3]

An algorithm to exploit this property can be readily constructed. The first problem is limiting the memory storage which is linear to the number of steps – instead, we use a circular buffer coupled with a set of logic registers which stores the C most recent steps. So then searching for a match takes $O(log(C))$ (at least for our functional programming implementation).

The algorithm starts by checking for matches inside the circular buffer. If no match is found, it simply continues evaluating the normal way, inserting states as it goes into the buffer. If a match is found, it uses Theorem 2's results to calculate the cycle length c, performs the skip and clears the buffer. The skip is performed by setting

$$n_{i+c(i-j)} = n_i \left(\frac{n_i}{n_j}\right)^c ,$$

which effectively evaluates the cycle c times. It can be seen that if $c = 0$ it doesn't change any values; however, 0-length skips don't result in an infinite loop because it has to rebuild the buffer.

One useful aspect of this is that if the $c = \aleph_0$ then we can detect at runtime the infinite loop and terminate rather than sit without user output. This may be useful for avoiding bugs in FRACTRAN programs.

5 Results

The code can be found at `github.com/pimlu/fractran`. (Or at least, it will be once I clean it up and document it...)

These benchmarks were performed on a Pentium G4600. There were no repeat runs, but the results were nonetheless distinct. Some benchmarks were skipped on the slower performing algorithms because they would take too long.

First, we test on PRIMEGAME with varied amounts of fractions. We included Nick Johnson's Python reference implementation as one of the options, versus our similar register-based implementation, the "fraction optimized" implementation which is the same as register-based but with the fraction adjacency matrix reducing the set of checked fractions, and finally the cycle-detecting implementation, which also happens to include the fraction optimizations. Our cycle implementation only outputs intermediate states that aren't skipped, but since it only skips arithmetic loops, the moments where it outputs primes themselves aren't skipped (since they're between loops).

Benchmark	Naive	Python Ref.	Reg. based	Fract. Opt.	Cycle Det.
PRIMEGAME 50	25.4s	22.7s	7.2s	5.7s	**1.4s**
PRIMEGAME 100	7m18.9s	4m28.5s	1m36.0s	1m12.5s	**8.0s**
PRIMEGAME 200	—	—	—	—	**43.0s**

Next we move to Mike Stay's Hamming weight calculator, which is arguably the best-case scenario for the cycle detector. HAMMING k consists of a benchmark with the input 2^{2^k-1}. So that the non-naive implementations don't waste time with inefficient (and irrelevant) factoring implementations, the register is input directly with the value $(2, 2^k - 1)$.

Benchmark	Naive	Python Ref.	Reg. based	Fract. Opt.	Cycle Det.
HAMMING 18	17.6s	0.8s	0.2s	0.2s	**<0.001s**
HAMMING 20	4m30.9s	3.0s	0.7s	0.7s	**<0.001s**
HAMMING 24	—	45.9s	11.4s	11.4s	**<0.001s**
HAMMING $24 \cdot 10^1$	—	—	—	—	**0.003s**
HAMMING $24 \cdot 10^2$	—	—	—	—	**0.044s**
HAMMING $24 \cdot 10^3$	—	—	—	—	**0.586s**
HAMMING $24 \cdot 10^4$	—	—	—	—	**13.709s**

6 Possible Improvements

Our results find conditions in which fraction testing can be eliminated and cycles can be detected and skipped forward. However, both conditions are relatively convservative right now. Reducing the fraction graph further based on more in-deph analysis is certainly possible, since humans do it every time they write programs; however, new results are needed to automate the process. In addition, it may be possible in the future to skip forward higher order cycles (such as the nested loops you would find in multiplication) if the skip itself can be saved into the buffer of previous states.

7 Bibliography

1. Conway, J. H. "Unpredictable Iterations." *In Proceedings of the 1972 Number Theory Conference Held at the University of Colorado, Boulder, Colo., Aug. 14-18, 1972.* Boulder, CO: University of Colorado, pp. 49-52, 1972.

2. Conway, J. H. (1987). Fractran: A simple universal programming language for arithmetic. In *Open problems in Communication and Computation* (pp. 4-26). Springer New York.

3. Guy, R. (1983). Conway's Prime Producing Machine. *Mathematics Magazine, 56*(1), 26-33. doi:10.2307/2690263

4. Endrullis, J., Grabmayer, C., & Hendriks, D. (2009, August). Complexity of Fractran and Productivity. In *CADE* (pp. 371-387).