PROGRAMMING IN FRACTRAN: AN EXPERIENCE REPORT

EZRA FURTADO-TIWARI

Abstract.
A summary of the progress made in CMPTGCS 130E, Spring 2025. The project explored the Fractran language, studying its advantages and limitations. The repository can be found at `https://github.com/ezraft/FRACTRAN`.

1. Introduction

The following report will explain the process and analysis conducted in producing the programs in `https://github.com/ezraft/FRACTRAN`. Code samples will be provided in this paper but additional programs are provided in the repository.

Fractran is an esoteric programming language created by John Conway in 1987. The language is made up of lines of instructions $a/b\ c$, with $a, b, c \in \mathbb{N}$, $b, c \neq 0$. The instruction as given previously is interpreted as follows: let x be the input given to this instruction. If $x \equiv 0 \bmod b$, set $x := x \cdot a$ and jump to line c. If there are multiple instructions $(a_i, b_i, c_i), i \in \mathbb{N}$ in a line, execute the first one (rigorously, the instruction with minimum i) that satisfies $x \equiv 0 \bmod b_i$. If no instruction in the current line can be executed then the program halts.

The following is a valid program in Fractran:

```
15/1 2
7/7 2,2/3 2,2/5 2,1/7 3
15/2 3,1/1 2
```

When given the input $n = 7^k$, this will produce the output 2^{2^k}. Note in particular that $\gcd(a, b) \neq 1$ in the first instruction; in other words, we do not require that each fraction is in lowest terms.

This program can be shortened to a 1 line program, which we refer to as a FRACTRAN-1 program; the following is such a simplification:

```
77/77 1,22/33 1,22/55 1,17/77 1,255/34 1,11/17 1
```

When run with the input $n = 3 \cdot 5 \cdot 11 \cdot 17 \cdot 7^k$, this will produce the output 2^{2^k}. A theorem of Conway (given in [Con87]) in his original paper states that any n-line Fractran program may be simplified to a FRACTRAN-1 program by multiplying the input by a constant C specific to the program; this example illustrates why this is true. In particular, we use a "flag" k_j for each line j, given by some prime number not used in the original n-line program. To transition from line ℓ_1 to line ℓ_2, we add the instruction k_{ℓ_2}/k_{ℓ_1}. For all instructions corresponding to line j, we multiply by k_{ℓ_j}/k_{ℓ_j}. Because of this we only execute instructions corresponding to one line at a time.

In the case of FRACTRAN-1 programs we can simplify our syntax; for instance we can denote the program above as

$$\frac{77}{77}\frac{22}{33}\frac{22}{55}\frac{17}{77}\frac{255}{34}\frac{11}{17}.$$

Date: June 6, 2025.

The flags corresponding to line 1 and 2 are 11 and 17, respectively; we see that the instruction $\frac{17}{77}$ facilitates the transition from line 1 to line 2, decrementing k in the process.

An interesting question we can ask about FRACTRAN-1 programs is the minimum number of instructions required to write an equivalent program; this is in general a difficult problem, even for fixed programs. For the computation above, for instance, we can reduce the program by 1 instruction as follows:

$$\frac{77}{77}\frac{44}{33}\frac{17}{77}\frac{51}{34}\frac{11}{17}.$$

In this case we would run our program with the input $n = 3 \cdot 11 \cdot 17 \cdot 7^k$. In this case, instead of adding the values in registers 3 and 5 to register 2 at each stage, we add the value in register 3 twice.

2. A Basic Fractran Interpreter

The first software artifact created over the course of this project was the initial Fractran interpreter. This naive interpreter uses python integers for computations, and executes Fractran programs one instruction at a time. The code is provided at `https://github.com/ezraft/FRACTRAN/blob/main/interpreter.py`.

3. Writing Basic Fractran Programs

In this section we describe some of the building blocks and insights that led to the construction of more complicated Fractran programs developed in the next sections. We also describe a FRACTRAN-1 program to compute the n-th Fibonacci number, in a very similar style to the one used to compute powers of 2 referenced in the introduction to this report.

The first program is an adder, which can also be used to swap two numbers:

```
2/3 1
```

The following program adds the number in register 3 to register 2, setting register 3 to 0 in the process. This can be extended to a copy function as follows:

```
15/2 1,1/1 2
2/5 2
```

The program above copies register 2 into register 3. This concept can be easily extended to multiplication by a constant, and adding constants; for instance, to add 5 to register 2, one can simply write the following program:

```
32/1 2
```

Note that since line 2 is empty moving to this line will halt the program. Generally the program $2^k/1\ 2$ will add k to register 2.

Multiplication by a constant k is very similar to our addition function, except we use a k-th power in the numerator. For instance to multiply register 2 by 2 (assuming register 3 is empty) we use the following program:

```
3/2 1,1/1 2
4/3 2
```

Finally, this naturally produces a program to compute division by k and the corresponding remainder. The following program computes register 2 divided by 2 (placing the result in register 2) and the remainder placing the result in register 3).

```
3/2 1,1/1 2
2/9 2
```

In this stage of exploration we also developed a FRACTRAN-1 program to compute the n-th Fibonacci number. This program is structured very similarly to the program described in the introduction. It can be stated as follows:

$$\frac{455}{39}\frac{1}{13}\frac{51}{34}\frac{51}{85}\frac{1}{17}\frac{2}{7}\frac{221}{23}.$$

4. A 1-Bit Stack

In the previous section, we developed a way to get the lowest set bit (dividing by 2 and taking the remainder) and to shift left by 1 position (multiplying by 2). What if we could extend this to simulate a stack in one of the registers?

The program in `https://github.com/ezraft/FRACTRAN/blob/main/stack.sl` uses these tools to do exactly that. The program takes a stack in as input in register 2, a stack of instructions (push/pop) in register 5, and a stack of inputs to push in register 3.

5. A Fractran ALU

Our stack operations turn out to be very convenient for implementing bitwise operations. For instance, how can we implement bitwise AND? Why don't we just pop each bit from both arguments, take the AND, and then push them to a new stack?

This is sufficient, but we need one final step; we need to reverse the answer stack. This is quite easy; we just swap it bit-by-bit into another position.

Similarly, we can implement bitwise OR, bitwise XOR and the bitwise complement (NOT) of a single argument.

Combined with addition, subtraction, multiplication, and division operations, we have all the components required for an ALU.

The relevant programs to this section are provided in the ALU folder of the repository. A general outline of the strategy for implementing bitwise operators is as follows:

(1) Pop a bit from both arguments. Call these b_1 and b_2.
(2) Compute $b := b_1$ op b_2.
(3) Push b onto a new stack r.
(4) Once complete, push all the elements from r onto a stack s, and return.
(5) The output will be provided in register s.

6. Efficient Fractran Interpreter

Naturally one may hope to extend the stack framework developed in the previous section to multi-bit arrays, by choosing a fixed bitwidth and performing multiple single bit pops to pop an element from the stack. Implementations of these functions are provided in the Array folder of the repository. These programs are implemented very similarly to the stack programs, and additionally make use of the power of 2 computation that we introduced in the introduction.

However, these programs illustrate an issue with Fractran; suppose we wanted to evaluate the following very simple program:

```
3/2 1
```

How many operations does this require? Suppose we gave the input 2^{100}. In this case we require 100 executions of the first and only instruction. What about $2^{2^{10}}$? This requires $2^{10} = 1024$ executions. Generally, if we want to add two k-bit numbers using this program (giving the program the input $n = 2^a 3^b$ where $a, b < 2^k$), we need up to $2^k - 1$ operations. This is in contrast to more popular programming languages like Python, which computes addition bitwise, requiring only $O(k)$ steps for addition of k-bit integers.

We can try to simulate our addition program bit-by-bit by adding more instructions; for instance, consider the following program:

```
81/16 2,1/1 2
9/4 4,1/1 4
3/2 4
```

Tracing this program with the input 2^7, we see that we have the following steps:

$$2^7 \to 2^3 3^4 \to 2^1 3^6 \to 3^7.$$

We first subtract $4 = 2^2$, then $2 = 2^1$, then $1 = 2^0$, effectively simulating binary digit addition. It is clear that we can extend this to handle k-bit numbers with k instructions.

However, consider an array of 5 4-bit integers, and suppose that we need to pop the 1st element. First, passing the array into our program requires an input of at least $2^{2^{5 \cdot 4}} = 2^{2^{20}}$. Next, we need to perform division of $2^{2^{20}}$ by 2^{2^4} for the pop operation. If we use our naive division algorithm, we will require $2^{20}/2^4 = 2^{16}$ operations. For a 10 element array, we need $2^{40-4} = 2^{36} \approx 6 \cdot 10^{10}$ operations. Even without considering the exponential growth of the input values, running such a program is not feasible. Assuming we use our optimization, we would need 36 instructions to process our 10 element array. What if instead we wanted to run programs with larger inputs (say 10000 elements)? Writing such programs is extremely impractical.

Another solution would be to improve our interpreter so as to run multiple instructions at a time. Making an efficient interpreter could likely be a project of its own; however, some simple optimizations can be very effective for the specific programs that we seek to write (e.g. arrays, stacks, ALU operations).

Most lines of our programs have the following structure:

```
a/b [n], 1/1 [n+1]
```

We can optimize this line by noticing that if a and b are coprime, then we can compute the number of iterations of the first instruction by counting multiplicities of primes in the factorization of boht the instruction and the input. If a and b are not coprime, this only affects whether or not we can run the instruction in the first place, so we can check against the denominator and then exclude it when computing the number of iterations of the instruction that we do.

This dramatically speeds up computations; for instance, we can add in $O(1)$ time (technically $O(k)$ since the python interpreter still needs to do basic arithmetic operations). With our array programs, we still have issues with input size (an array of size n with k-bit integers still requires an input of 2^{nk} as opposed to $n \cdot 2^k$). While we could modify the way that input is taken, this would be difficult without fundamentally changing the structure of the language. Unfortunately, this means that within the constraints of the current project, one can only represent arrays of up to ≈ 25 bits, i.e. 4 bit arrays of size 6.

7. Conclusion

In this project we wrote a variety of Fractran programs in order to explore the limits of this programming language. We started by writing basic programs to compute recurrences (such as powers and Fibonacci numbers), then moved on to compute more complicated arithmetic operations, including modulo and bitwise operations. We implemented a complete set of such instructions, which are given in the ALU folder of the repository. In the process, we developed a method to simulate a stack using the registers given in the format of this language. This allowed us to extend our programs to store arrays, which led to an exploration of the limitations of the Fractran language, in particular exponential growth of the number of operations and amount of memory required for programs.

Throughout this project we considered also the optimization of Fractran programs, first in program size (through FRACTRAN-1), then through time optimizations, and finally through optimization of our Fractran interpreter. With additional time one could try to optimize the interpreter further to compress loops of larger size than 1, or to construct a representation of arrays that would allow input to be taken directly. This would allow more interesting programs to be written; for instance, if this framework were to be developed, almost any OISC could be implemented fairly simply.

It appears Fractrans largest strength is its ability to compute recurrences (and more generally accumulator-based programs) with simple programs, although these may take exponential time; Conway famously wrote programs to compute π using the Wallis product, and to iterate through the prime numbers. More recent work on this topic has included the use of recurrence relations to compute digits of $\sqrt{2}$ and other radicals [KMW24] using Fractran.

References

[Con87] John Conway. Fractran: A simple universal programming language for arithmetic. `https://www.cs.cmu.edu/~cdm/resources/Conway87.pdf`, 1987. [Accessed 05-05-2025].

[KMW24] Khushi Kaushik, Tommy Murphy, and David Weed. Computing $\sqrt{2}$ with fractran, 2024.