In-Network Fractional Calculations using P4 for Scientific Computing workloads

Shivam Patel
Illinois Institute of Technology
USA

Rigden Atsatsang
Illinois Institute of Technology
USA

Kenneth M. Tichauer
Illinois Institute of Technology
USA

Michael H. L. S. Wang
Fermilab
USA

James B. Kowalkowski
Fermilab
USA

Nik Sultana
Illinois Institute of Technology
USA

Abstract

Recent P4 research has motivated the need for in-network fractional calculations to support functions in Networking (for calculations related to active queue management and load balancing) and in Machine Learning. The P4 language and ASICs do not natively support fractional types (e.g., float). Existing P4 techniques provide incomplete emulations of the IEEE-754 standard, which was designed as a generic approach that can benefit from dedicated hardware acceleration, but whose features are difficult to fully support in P4.

This paper re-thinks the foundation of in-network fractional calculation and proposes a new approach that is more resource conscious and is straightforward to encode in P4. Instead of floating-point, it uses a fixed-point encoding of numerals; and instead of sampling functions into tables it uses Taylor Approximation to reduce dataplane calculations to simple arithmetic over pre-calculated coefficients, requiring constant space and linear time. The paper describes and evaluates a P4 code synthesis algorithm that allows users to trade-off switch resources for accuracy, grounded on an application of a well-understood mathematical theory. It describes how to encode π and various functions including cos, log and exp.

This technique is being developed to support Scientific Computing (SC) applications which typically make heavy use of fractional approximations of Real numbers. The paper applies this technique in a novel P4 program that is being open-sourced: in-network Monte Carlo simulation of photon propagation that models the analysis that is carried out in a class of cancer treatments. This technique is also being used in ongoing work on another SC application: online event detection in a large-scale neutrino detection experiment.

CCS Concepts

• **Networks → In-network processing**; • **Mathematics of computing → Numerical analysis**.

1 Introduction

Scientific Computing (SC) has an ever-increasing appetite for computing and communication capacity across a huge range of online and offline processing tasks drawn from various systems, including large-scale discrete element simulations, high-energy particle detectors and telescope arrays.

SC applications are typically distributed across many nodes in a network, and programmable switching and NICs could contribute domain-specific forwarding optimizations and simple processing tasks operating on large volumes of data flowing between nodes.

But SC computations often involve calculations over real numbers approximated using fractional types, which are not natively supported on all P4-programmable hardware platforms. The need for in-network fractional calculation is recognized in the literature, mainly clustered around two application domains: (1) approximation techniques for network functions [18, 19] including active queue management and load balancing [14], and (2) to provide in-network support to Machine Learning applications [22].

This paper describes a new technique for fractional calculations in Scientific Computing workloads. The scripts and source code are made available for others to build on and explore its application in other domains.[1] **This technique is a form of numerical analysis that can generate $\mathbb{R}$-algebras that are customized to the accuracy needs of a given application, and sensitive to the resources provided by a specific programmable network platform.** It can be applied to a large class of widely-used $\mathbb{R}$-valued functions, including cos, log and exp. The technique consists of two elements: **(1)** fixed-point representation of numerals, which provides more predictable accuracy across all numeral ranges compared to IEEE754-based floating-point encodings, and **(2)** offline computation of the Nth-degree Taylor polynomial's coefficients to approximate a given function [6]. Unlike existing techniques, this approach does not rely on table look-ups to evaluate functions; instead it reduces function approximation to simple arithmetic over coefficients calculated in step (2), and which can be carried out in the dataplane. The technique works by synthesizing a P4 program based on users' accuracy- and resource-tradeoff. This technique has been evaluated on BMv2, and is being ported to hardware targets.

Compared to existing techniques, this approach (1) provides predictable accuracy across the entire number line, (2) replaces use of

---

[1] https://github.com/ShivamPatelShivamPatel/Photon

Work	Existing/ New Target	FLoating/ FIxed Point	Variable Precision	Beyond Arithmetic
FPISA [22]	N	L	×	×
PF [17]	E	L	×	√
InREC [12]	E	L	×	√
NetFC [9]	E	L	×	×
This paper	E	I	√	√

Table 1: Features of related techniques.

Figure 1: Decomposition of SC applications' needs and mapping them to programmable switch resources.

Figure 2: In-network support for photon path modeling.

tables with ALUs, (3) provides tunable, per-application trade-off between resources and accuracy.

Limitations. The technique used in this paper is only applicable to real analytic functions—this includes many commonly-used functions, as shown later. For other functions, this technique would be combined with complementary techniques.

Paper outline. The rest of the paper proceeds as follows: §2 discusses related work, §3 describes our motivating use-case, and §4-§6 describes this technique, its implementation, and its evaluation respectively. §7 discusses the findings in this paper and future work directions.

2 Related Work

This section contrasts the technique described in this paper with techniques in related work that targets P4 or PISA-like hardware [5]. Table 1 categorizes closely-related work in the following dimensions: (1) Whether they target existing P4 platforms or new ones—most work targets existing hardware or software platforms, but are restricted by the capabilities of those targets. (2) Support for floating- or fixed-point encoding of fractional numbers—virtually all existing work uses floating-point numbers, in most cases deviating from the IEEE-754 standard [1]. For example, NetFC [9] removes the NaN ("Not a Number") representation, and PF [17] (for the "Pseudo-Floating Point" representation it uses) removes the encoding of the exponent. In comparison, we use fixed-point representation which avoids unexpected behaviors [15] and provides an evenly-spaced representation of numerals to benefit the accuracy-sensitive Scientific Computing applications we target.[2] (3) Support fixed or variable precision—IEEE-754 was devised as a generic approach over fixed precision classes, but for the applications targeted in this paper we devise a technique for customizable, per-application precision. (4) Whether the approach goes beyond arithmetic (addition, subtraction, multiplication, division) to support special and trigonometric functions. Applications in Scientific Computing often require more than arithmetic.

Among existing work, Sankaran et al. [17] is the most closely related. They described a P4-based streaming analysis of image data and an approximation scheme for the natural logarithm function, and applied it to a Scientific Computing use-case. In comparison, this paper presents a mathematically-founded, general resource-aware derivation approach that works for more functions. Sankaran et al. provide an inspiring use-case that could be adapted to use the technique described in this paper, and future work could compare hardware implementations of both approaches.
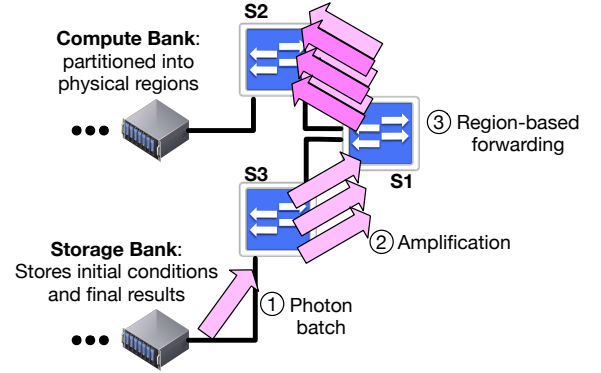
3 Use-case: Light Propagation Modeling

This section describes an application from scientific computing (SC) and the role we envisage for programmable switching. Fig. 1 summarizes our analysis of how programmable networking can contribute to SC. This paper focuses on step ❷ which presents a significant technical difficulty because of the mismatch between P4 and the needs of the SC workloads.

Use-case. Surgery is often the first line of treatment for patients diagnosed with solid cancer, where complete removal of the tumor(s) is not always achieved owing to challenges in detecting remaining cancer cells. Surgery of head and neck cancers is especially challenging because of particularly complex anatomy [11], which can lead to incomplete surgical removal. To improve the surgical removal of tumours, cancer-targeted fluorescence agents can be injected into patients prior to surgery to help better identify remaining cancer in their patients so that it can be removed. This uses specific wavelengths of light to illuminate tumours, which enables their identification and subsequent removal [4, 13, 20, 21].

Optimization of these strategies requires accurate understanding of how fluorescent light propagates through biological tissue at the margins of cancer resections. However, accurate simulations are restrictively time-consuming using current methods. To assist with these efforts, we have been developing advanced imaging strategies to help detect even the smallest amounts of cancer. This understanding is obtained through detailed simulations of photon propagation

---

[2]Recall that the encoding standarized in IEEE-754 separates numbers into disjoint classes that are afforded different (sometimes variable) accuracy, and involves a normalization stage. This is difficult to emulate in P4.
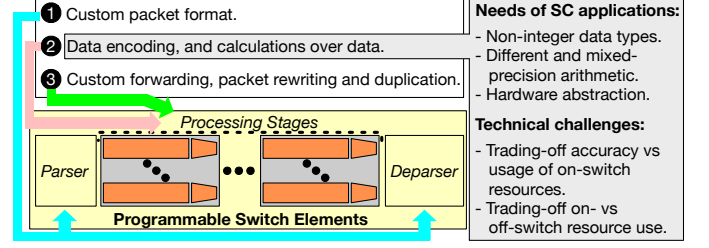
across different tissues. However, accurate simulations are resource intensive and time-consuming to produce. To assist with these efforts, we explore the use of programmable switching to distribute simulation load consistent with photon trajectories in a volume, and also absorb some of that load through in-switch calculation.

Algorithm and primitive constituents. Through collaborators who are active researchers in biomedical science specializing in this application area, we obtained a description of the basic algorithm. This was first prototyped in Python then ported to P4. Photons are modeled as decaying energy. Once a sufficient amount of energy has been lost, the photon's location can be determined. We model this as a pseudo random distance d that is the natural logarithm of a physical constant, where the randomness corresponds to the percentage which the photon is obeying the constant. We exponentially decay the energy, and use the most recently computed d as the distance traveled. After this calculation, we obtain the photon's location in $\mathbb{R}^3$ in polar coordinates, and we then use standard trigonometric formulas to convert to Cartesian coordinates.

Contribution from Programmable Switching. Fig. 2 shows how switch programmability supports this experiment. ① Initial photon coordinates are loaded from the Storage Bank and sent to a host on the Compute Bank. ② Switch **S3** amplifies the number of photons, volumetrically placing them according to a given distribution. Our prototype amplifies by ×100. Amplification involves the switch: (i) parsing the packet contents and interpreting photon coordinates; (ii) storing an approximation of the distribution on the dataplane, in tables and registers; (iii) cloning and recirculating packets to amplify the original batch of photons. ③ Switch **S1** rewrites the destination address and forwards packets according to the photons' coordinates they carry, so that photons in the same sub-volume will be sent to the same host. This partitioning of the Compute Bank to handle different sub-volumes of a space is intended to model different materials that photons encounter—such as different skin layers, blood, and blood vessels. Switches **S1** and **S2** can amplify, process, and forward photons further.

4 $\mathbb{R}$-algebra Approximation in P4

Taylor Approximation [6] is a well-established technique to approximate non-linear $\mathbb{R}$-valued functions such as cos, log and exp. We adapt this technique to develop a P4 code synthesizer (§4.2) for tunable accuracy and resource usage.

This approach provides tunability by varying the degree of the Taylor polynomial that is generated. This degree correlates with the accuracy of the function's approximation and with the resources that will be needed to evaluate the approximation at runtime. The coefficients of this polynomial are pre-calculated and stored on the P4 target. The target uses the coefficients to evaluate the function approximation at runtime in the dataplane.

This tunability is essential to balance generality with practicality. A person working in hard science may require precision levels of 10^{-6} or more, but someone needing to compute probabilities in another domain may only need precision levels of 10^{-2}. The precision needed by the function and that needed to encode numerals are scaled in tandem; we give worked examples of this below.

4.1 Approximating Real numbers

Recall the definition of 2's complement integer representations. For $n \in \mathbb{Z}$, n can be represented as:

$$n = (-1)x_{p-1}2^{p-1} + \sum_{j=0}^{p-2} x_j 2^j \quad \text{where} \quad \begin{aligned} p &\geq \lceil log_2(|x|+1) \rceil + 1 \\ \text{and} \quad x_k &\in \{0,1\} \end{aligned}$$

(1)

For example, to encode $n = 42$ then we would require $p = \lceil log_2(|42|+1) \rceil + 1 = 6 + 1 = 7$ bits. Therefore $p = 7$ is the smallest possible choice in Eqn 1. When written as a sum of powers of 2, $42 = (-1) \cdot 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, corresponding to the 2's complement bit string 0101010. This paper relies on fixed-point encoding, which extends Eqn 1 to the real line by adding a component that represents the fractional part. Suppose now that $x \in \mathbb{R}$, then using a construction based on Dedekind cuts we can show that it has a representation as:[3]

$$x = (-1)x_{p-1}2^{p-1} + \sum_{j=0}^{p-2} x_j 2^j + \lim_{q \to \infty} \sum_{i=1}^{q} x_{-i} 2^{-i}$$

(2)

We tune the representation of a numeral by taking a finite truncation of the sum on the right (finite $q \in \mathbb{N}$) leading to what is referred to as "$p.q$ fixed-point encoding".

Worked example: approximating π. To observe how the $p.q$ encoding works, take $x = \pi$ and observe that $\pi = \lfloor \pi \rfloor + (\pi - \lfloor \pi \rfloor) = 3 + (\pi - 3)$. Using the rule for p in (1), since $\lceil log_2(|3|+1) \rceil + 1 = 2 + 1 = 3$ we require at minimum $p = 3$ for a 2's complement representation. We see that $3 = (-1) \cdot 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$. Now to approximate $\pi - \lfloor \pi \rfloor$, suppose we want to estimate it to be within $\epsilon = 10^{-10}$ of its real value. Since π is irrational, it does not have a representation as a finite sum of powers of 2. If we wish to approximate it up to $\epsilon = 10^{-10}$, then we need $q \geq \lceil |log_2(\frac{1}{10^{-10}})| \rceil = 34$. We will take $q = 38$ since position 2^{-34} has a coefficient of 0, with the next "1" appearing alongside 2^{-38}. The techniques used to compute the significant terms are expressed as for $0 < y < 1, s_1 = \lfloor \log_2(y) \rfloor, s_n = \lfloor \log_2(y - \sum_{i=0}^{n-1} 2^{s_i}) \rfloor$. Given our requirement of $q \geq 34$ and our choice of $q = 38$ we can terminate the expansion whenever $|s_n| \geq 38$ to have our estimate be within $\epsilon = 10^{-10}$. For the sake of completeness we will enumerate the sum in an inequality $|(\pi-3)-(2^{-3}+2^{-6}+2^{-11}+2^{-12}+2^{-13}+2^{-14}+2^{-15}+2^{-16}+2^{-18}+2^{-19}+2^{-21}+2^{-23}+2^{-25}+2^{-29}+2^{-33}+2^{-38})| < 10^{-10}$. Observe the -3 in parenthesis with π. If we acknowledge that $3 = (-1) \cdot 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$, factor out a negative, and merge it with the estimate of $(\pi - 3)$ we can observe that $|(\pi - ((-1) \cdot 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 2^{-3} + 2^{-6} + 2^{-11} + 2^{-12} + 2^{-13} + 2^{-14} + 2^{-15} + 2^{-16} + 2^{-18} + 2^{-19} + 2^{-21} + 2^{-23} + 2^{-25} + 2^{-29} + 2^{-33} + 2^{-38})| < 10^{-10}$. This can be made more explicit by padding-in the 2^{-i} that have 0 as a coefficient, and recognize this as the 41-bit 2's complement fixed point number 011.00100100001111110110101010001000100001 with scaling factor 38.

4.2 Approximating functions

To approximate function f we start with the Taylor Polynomial of degree N: $\sum_{k=0}^{N} \frac{f^{(k)}(a)}{k!}(x-a)^k$, where $f^{(k)}$ is the kth derivative of f, and a is an *expansion point* where the domain of the approximation is centered. We gather the coefficients of this polynomial *offline*

---

[3]The accompanying technical report provides more background [16].

and generate P4 code that evaluates the polynomial at runtime. N represents accuracy and is related to the p and q choices. If we are seeking specific $p.q$ (to operate on π from the previous example) then we iteratively search for an N that yields the closest match for $p.q$.

Once a polynomial that approximates a function has been generated, performing the calculations using that polynomial uses constant space and linear time. The linearity is in terms of N—the degree of the polynomial.

To evaluate the polynomials, we synthesize P4 functions that implement a Multiply-Add-Accumulate(MAC) operation that corresponds to polynomial evaluation. The technique is a variant of Horner's method [8] and is computed by updating an index variable that selects the k-th coefficient and then performing fixed-point multiplies and adds.

Algorithm 1 generates a P4 function called operation_f_p_q(x) and supporting definitions to approximate a given function f. Note that the algorithm: (1) also takes parameter a, which is a point on the line for which we expect highest accuracy; (2) generates a P4 program P by accumulating definitions and statements; (3) is executed offline before the in-network computation begins; (4) generates P4 that only involves arithmetic on unsigned words that we use to store $p.q$ fixed-point encodings of $\mathbb{R}$-values. The computation of derivatives and factorials is done offline. Once the P4 program is synthesized, it is compiled and installed on the switch.

The accuracy and resource usage of this algorithm is evaluated in the next section. We now analyze the algorithm a little further. The offline part of the algorithm involves computing the coefficients to a Taylor polynomial of degree N. It also builds a program—consisting of $p.q$-sized register definitions and initializations, and an unrolled loop that carries out the evaluation of the polynomial in the dataplane. The unrolled loop consists of repetitions of the final block of code, which updates the res value that is returned by the action.

Worked example: log. Take $\log(x)$ on the domain $(\frac{1}{1000}, 2^{-5})$ with expansion point $a = 2^{-6}$. Following Algorithm 1, initialize res = 0 and m = 0. The mth derivative of $\log(x)$ on our domain evaluated at our expansion point a is $\log(2^{-6}) = -6\log(2)$ if m is 0, otherwise it is $(-1)^{m-1}(m-1)!(2^{-6})^{-m} = (-1)^{m-1}(m-1)!(2^6)^m$. From the definition of d_i in Algorithm 1, for all the c_m where $m > 0$, we have $c_m = (-1)^{m-1}\frac{(2^6)^m}{m}$. In this case x_sub_a $= (x - 2^{-6})$, and at the end of the mth repetition (before incrementing the register reader variable m) we have x_sub_a_m $= (x - 2^{-6})^m$. At the end of the code repetition we will see res contain the value of $-6\log(2) + \sum_{m=1}^{N}(-1)^{m-1}\frac{(2^6)^m}{m}(x - 2^{-6})^m$.

4.3 Optimized approximation

There are two optimizations we found useful. First, when the derivative terms are complicated, it is beneficial to simplify the polynomial, since it will use fewer resources without compromising accuracy. A typical example would be to simply precompute fixed-point numerals for each of the needed reciprocal factorial terms and then incrementally compute the numerator of the kth term from within the P4 code. For example, the final expression in the previous example can be simplified to $\sum_{m=1}^{N}\frac{(-1)^{m-1}}{m}(2^6x - 1)^m$. Another example

---

Algorithm 1 Generate approximator for f.

Require: Parameters f, N, p, q, a where a is center point and f is real analytic.
Ensure: Program P has length $\leq f(N, p, q)$

$\quad P \leftarrow \emptyset$ $\qquad\qquad\qquad\qquad$ ▷ Initialize P to empty P4 program.
$\quad$ **for** $1 \leq k < N$ **do**
$\qquad d_i \leftarrow |\frac{(f^{(k)}(a))}{k!}|$ $\qquad$ ▷ Compute k^{th} derivative scaled by $k!$.
$\qquad p_i, q_i \leftarrow \text{conv}_2(p, q, d_i)$ $\qquad$ ▷ Encode d_i as $p.q$.
$\qquad P \leftarrow \text{defReg}(\text{Reg}_i, p + q)$ $\quad$ ▷ Define register, width $p + q$.
$\qquad P \leftarrow \text{writeReg}(\text{Reg}_i, \text{concat}(p_i, q_i))$ $\qquad$ ▷ Initialize.
$\quad$ **end for**
$\quad d \leftarrow \frac{(f^{(k)}(a))}{k!}$
$\quad$ ▷ Define approximation that's contributed to the function's value.
Typedef 'R' is a bit vector of width $p + q$.
$P \leftarrow \text{defineAction}(\text{op}\_f\_p\_q(\text{x})$ with P4 template:
$\quad$ ▷ Handle case $m = 0$, and x^0

```
R m = 0; R prod = 0; R c_m = read_reg(m);
R res = c_m; R xsuba = x-a; R xsuba_m = x-a;
m = m + 1;
```

$\quad$ ▷ Make $N - 1$ copies of the following block:

```
c_m = read_reg(m;)
prod = (int<p+q>)((((int<2p+2q>)(xsuba_m)
   * (int<2p+2q>)(c_m)) >> q);
if (d > 0) res = res + prod;
else res = res - prod;
xsuba_m = (int<p+q>)((((int<2p+2q>)(xsuba_m)
   * (int<2p+2q>)(xsuba)) >> q);
m = m + 1;
```

---

involves the use of the half-angle formula[4] to obtain good precision with a smaller $p.q$-type by mapping the value of x to a subset of the domain.

Second, formulas can be simplified if the problem does not require the general case. For an example from our use-case, we take the natural logarithm of a value used to calculate d—the distance traveled by the photon in our simulation. To determine how much energy it lost—which is used to model its passage through different tissues—we evaluate $e^{-0.02d}$ and use this to exponentially decay the energy remaining for that photon. Since $e^{-0.02(\cdot)}$ is only used for this purpose in our procedure, we restricted its domain to be the image of log prior to constructing an approximator for it. The reason for restricting the domains of functions that need to be approximated is that typically a smaller domain corresponds to less computational effort and higher precision. This is similar to how for the case of log, since we knew that it was only going to be used on a subinterval of $(0, 1)$ corresponding to random percentages, it was advantageous to approximate log on that subinterval rather than all of $(0, 1)$.

5 Prototype Implementation

The implementation was 847 lines of P4$_{16}$ code targeting BMv2's simple_switch. Most of this code consisted of the implementations of the functions such as sin, cos, arccos, log, and exp. It uses 8.56 fixed-point encoding which amounted to using 64-bit words. For each function that we sought to make a Taylor Polynomial in P4,

---

[4]$\sin(x) = 2\sin(\frac{x}{2})\cos(\frac{x}{2})$

Function f	Degree N	LOC	#Registers
$\log(x)$	20	133	20
$e^{-0.02x}$	18	120	18
$\cos(x)$	9	77	9
$\sin(x)$	$9+8$	$77+65$	$9+8$
$\arccos(x)$	20	137	20

Table 2: Resource use for the functions in "photon.p4" (§3). Since sin **is expressed in terms of** cos**, it adds to the resources needed by** cos**. In §7 we discuss eliminating the need for registers by inlining constants. #Registers is the total registers that are needed, across all stages.**

Function f	Domain	#Coefficients	#Constants
$\log(x)$	$\left(\frac{1}{1000}, 2^{-5}\right)$	N	6
$e^{-0.02x}$	$\text{Im}[\log]\left(\frac{1}{1000}, 2^{-5}\right)$	N	11
$\cos(x)$	$(-\pi, \pi)$	N	8
$\sin(x)$	$(-\pi, \pi)$	$2N$	16
$\arccos(x)$	$(-1, 1)$	N	8

Table 3: Resources used for functions used in the prototype of our use-case (§3). Notation $\text{Im}[f](S)$ **denotes the image of** f **under** S**.** sin **requires** $2N$ **because it includes the definition of** cos**.**

we estimated how many terms were needed for the polynomial, and computed the coefficients using the `spfpm` module in Python.

We ran the prototype in a small network model for in-network simulation of photon propagation and captured the computed values. The captured values were analyzed for accuracy by comparing them to the outputs from a reference Python implementation that uses floating-point numbers. The next section shows graphs that compare the accuracy of P4-computed values with that of reference values computed using Python's approximation of real numbers.

We designed a packet format to encode each photon's coordinates. The format includes a a field to encode the photon's energy. Each packet contains information about a single photon that was emitted by the light source. Packets were then streamed through the network and processed by the P4 program.

The P4 program receives a photon packet, computes its new coordinates and recirculates it if needed to exponentially decay its energy. The calculation done in the switch involves updating polar coordinates, which involves trigonometric functions that were approximated using approach described in §4. Once the energy is sufficiently low, coordinates were changed from Polar to Cartesian and the packet is sent to a destination host for capture.

After capture, we analyzed the photon values against a reference implementation that was written in Python, and which used floating point types. We also visualized the distribution of photons in the 3D volume representing the tissue that was traversed by photons.

6 Evaluation

We evaluate **(1)** the accuracy of the resulting functions when compared to that of Python implementations that scientists will be familiar with; **(2)** the resource complexity of the resulting function approximations: how much space (registers) and steps they require to execute, giving us a back-of-the-envelope approximation of this technique's use of ALUs when it will be ported on real hardware.

Accuracy. Fig 3 shows the accuracy of function approximations when using different p and q. For each function in our use-case (§3) we generated 1000 points at random in the domain we were targeting. These points were converted into 8.56 fixed-point encodings. We then ran the workload through the P4 program and captured packets containing the resulting calculations. We then took the absolute difference with the result of Python's Math library. The precision of the P4 implementations were on the order of 10^{-4} to 10^{-7}.

Increasing the number of fractional bits improves precision. The difference in the error distributions vary according to the function

and the polynomial used. In particular, for the computation of cos we used the half-angle formula to define $\cos(x)$ in terms of $\cos(x/4)$ to obtain greater precision.

Resource complexity. The general method we use statically bounds the resource complexity as described in §4.2: constant space and linear time. Our evaluation provides a sampling of resource usage for functions that were approximated for our use-case (§3). This sampling is reported in two tables. Table 2 shows the lines-of-code of synthesized P4 and the number of registers that it contains. Note that the definition of sin includes that of cos because of the half-angle formula (described in §4.3). Table 3 shows the linear complexity of **#Coefficients**—note that sin's cost is higher because of the definition in terms of cos. The **#Constants** column shows actual values for the number of coefficients.
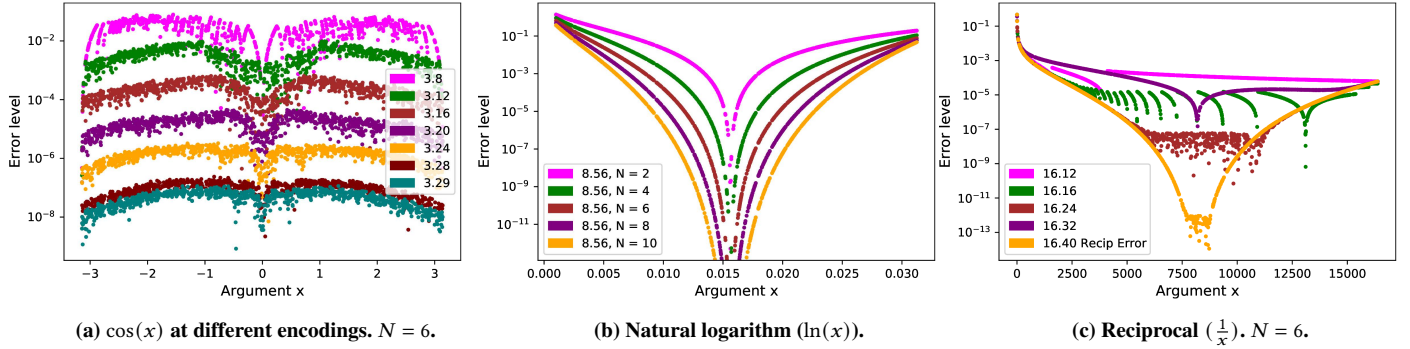
7 Discussion

This paper described and evaluated a proof-of-concept of a new approach for fractional calculations in P4. This approach is based on Taylor approximation, and the resulting P4 approximation uses linear resources and does not use tables.

The focus of this paper involved adapting this technique to work within the constraints of the P4 language. The next phase of this work involves porting this approach to run on programmable network hardware, and addressing the constraints of specific hardware targets as distinguished from constraints of the P4 language. These targets are typically conservatively designed around networking use-cases [10] and present target-specific constraints. For example, public documentation of a Tofino-like target [2] describes support for registers of sizes 8 bits, 16 bits, and 32 bits, or pairs of values of those sizes. According to that documentation, ALUs can add on sizes 8, 16, and 32 bits, but not multiply. Thus the P4 code synthesis will be adapted to only invoke specific operations over specifically-sized registers. We also plan to explore the P4-to-FPGA generation path, to evaluate this paper's technique on non-ASIC targets.

Another direction of future work involves optimizing this technique further. An easy first step involves eliminating the need for coefficient-storing-registers by in-lining constants into the code. Note that the coefficients are not updated at runtime, and therefore they can be hardcoded in the synthesized P4. This will simplify Algorithm 1, shorten the generated P4 code, and lessen the values in the "#Registers" column. Another optimization technique involves using a technique similar to half-angle reduction for non-trigonometric function. We are developing an approach based on a conjecture that provides this for the log function, which we will then evaluate against the approximation provided by the Taylor polynomial.

(a) $\cos(x)$ **at different encodings.** $N = 6$. **(b) Natural logarithm** $(\ln(x))$. **(c) Reciprocal** $(\frac{1}{x})$. $N = 6$.

Figure 3: Absolute Error between functions encoded in P4 compared to their Python counterparts.

Finally, we are developing another use-case that will be used to evaluate this approach further. It involves online processing of signal data from a model of the upcoming Deep Underground Neutrino Experiment [7] (DUNE), whose detectors will stream data continuously at several TB/s. We are exploring the use of programmable switching to move the simple data acquisition (DAQ) and reduction algorithms involving fractional calculations into the network to support this use-case. We envisage that data streaming out of the detector will be processed in-flight before they reach the DAQ compute cluster, freeing up resources in the DAQ for more sophisticated algorithms normally performed offline and offsite. We are developing a P4 prototype and validating it using waveform data generated by a DUNE simulator featuring high-fidelity particle models [3]. Collaborators provided us the DUNE filtering algorithms and these are being prototyped in P4. We envisage that other SC use-cases can benefit from the fractional calculation primitives that this paper described.

Acknowledgements

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] *p4_16 Tofino Native Architecture Application Note - Public Version 2021*. Intel, 2021.

[3] John Allison, Katsuya Amako, JEA Apostolakis, HAAH Araujo, P Arce Dubois, MAAM Asai, GABG Barrand, RACR Capra, SACS Chauvie, RACR Chytracek, et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*, 53(1):270–278, 2006.

[4] Stefan Andersson-Engels, Claes af Klinteberg, K Svanberg, and S Svanberg. In vivo fluorescence imaging for tissue diagnostics. 42(5):815–824, May 1997.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, aug 2013.

[6] Ole Christensen and Khadija Laghrida Christensen. *Approximation theory: from Taylor polynomials to wavelets*. Springer Science & Business Media, 2004.

[7] DUNE collaboration et al. Deep Underground Neutrino Experiment (DUNE): Far detector technical design report. Volume I. Introduction to DUNE. *Journal of Instrumentation*, 15(8), 2020.

[8] Thomas H. et al Cormen. *Introduction to algorithms*. MIT Press, 3 edition, 2009.

[9] Penglai Cui, Heng Pan, Zhenyu Li, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Hongtao Guan, and Gaogang Xie. NetFC: Enabling Accurate Floating-point Arithmetic on Programmable Switches. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11, 2021.

[10] Andy Fingerhut. Why doesn't P4 have floating point types? https://github.com/jafingerhut/p4-guide/blob/master/docs/floating-point-operations.md, February 2020. Accessed: 2022-08-06.

[11] Ulrich Harréus. Surgical errors and risks - the head and neck cancer patient. *GMS current topics in otorhinolaryngology, head and neck surgery*, 12, 12 2013.

[12] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. InREC: In-network REal Number Computation. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 358–366, 2021.

[13] Scott K Lyons. Advances in imaging mouse tumour models in vivo. *The Journal of Pathology*, 205(2):194–205, 2005.

[14] Mojtaba Malekpourshahraki, Brent E Stephens, and Balajee Vamanan. ADA: Arithmetic Operations with Adaptive TCAM Population in Programmable Switches.

[15] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):1–41, 2008.

[16] Shivam Patel. Continuous Generalization of 2's Complement Arithmetic. Technical Report: repository.iit.edu, 2022.

[17] Ganesh C. Sankaran, Joaquin Chung, and Raj Kettimuthu. Leveraging In-Network Computing and Programmable Switches for Streaming Analysis of Scientific Data. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 293–297, 2021.

[18] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Changhoon Kim, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 67–82, USA, 2017. USENIX Association.

[19] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, page 1–16, USA, 2018. USENIX Association.

[20] H. J. C. M. Sterenborg, M. Motamedi, R. F. Wagner, M. Duvic, S. Thomsen, and S. L. Jacques. In vivo fluorescence spectroscopy and imaging of human skin tumours. *Lasers in Medical Science*, 9(3):191–201, 1994.

[21] Gooitzen M van Dam, George Themelis, Lucia M A Crane, Niels J Harlaar, Rick G Pleijhuis, Wendy Kelder, Athanasios Sarantopoulos, Johannes S de Jong, Henriette J G Arts, Ate G J van der Zee, Joost Bart, Philip S Low, and Vasilis Ntziachristos. Intraoperative tumor-specific fluorescence imaging in ovarian cancer by folate receptor-α targeting: first in-human results. *Nature Medicine*, 17(10):1315–1319, 2011.

[22] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. Unlocking the Power of Inline Floating-Point Operations on Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 683–700, Renton, WA, April 2022. USENIX Association.