Two Approximate-Programmability Birds, One Statistical-Inference Stone

Adrian Sampson
University of Washington
asampson@cs.washington.edu

Abstract

We describe two central problems in the programmability of approximate systems—*assisted approximate programming* and *dynamic check generation*—and argue that they are, in fact, different views on the same underlying challenge. The common underpinning is the problem of statistical inference over probabilistic programs. We suggest research avenues borrowing from probabilistic programming that can help address the general problem.

1. Background

Approximate computing promises to advance computing efficiency by eliminating wasted resources spent on excessive accuracy. But a significant hurdle to approximation's adoption is *programmability:* developers need to compose unreliable computational components into reliable software. Languages, tools, and methodologies will be critical to equipping programmers to exploit the potential of approximate computing.

Development tools for approximate computing rely on domain-specific descriptions of program output accuracy [10–12]. For our purposes, we assume that programmers write constraints following the general form:

$$\Pr\big[d(f(x), f'(x)) \leq b\big] \geq p$$

where f and f' are the original and relaxed programs, x ranges over inputs, d is some distance metric in the output space, b is a distance bound, and p is a desired probability. This formulation is flexible: programmers can express hard constraints on legal output ranges by setting $p = 1$, the chance of achieving a specific correct output by setting $b = 0$, or anything in between [1].

The rest of this paper describes two important categories of hypothetical development tools founded on this statistical definition of correctness. We argue that they share a foundational problem statement and that solving either will address both. We suggest directions in statistical reasoning than may help the community address the underlying problem.

2. Assisted Approximate Programming

When writing software to run on approximate hardware, a programmer needs to make an intractably large number of local decisions [4]. Every operation and every storage location may be approximate or precise—or worse still, in some machine models, each operation can have many possible precision levels [15]. Each fine-grained approximate operation can have far-reaching, nondeterministic effects on overall program behavior. Most crucially, individual approximations can compose in unintuitive ways to create constructive or destructive interference. If it is done manually, approximate programming is an exercise in frustration.

The community has begun to approach the problem of automatically assisting the programmer in making approximation decisions [3, 8–10]. In an ideal setting, programmers would provide only an original, precise program and a probabilistic correctness constraint in the form described above. An assistant would then produce the appropriate precision level for each operation in the program necessary to meet the specification. Clearly, this problem is hard: the space of possible approximation decisions is large and their impact on the output is not obviously efficient to measure.

3. Generating Cheap, Dynamic Quality Checks

Even in well-tuned software, an approximation strategy can work better for some inputs and worse for others. *Introspection* of approximation's effects is crucial for this reason [5, 11]. If applications can detect situations where approximation is particularly deleterious, they can fall back to precise execution or adjust precision parameters. This kind of dynamic self-healing behavior can enable more aggressive approximation: if we know we can catch outliers, we could optimize better for the common case.

To enable quality introspection, we need ways to forecast the distance bound $d(f(x), f'(x)) \leq b$ at run time to judge a particular x. Crucially, this check must be *cheap*—the naive approach of obtaining $f(x)$ via precise re-execution and then invoking the distance metric d would obviate any benefits offered by approximation. Some applications have straightforward, domain-specific cheap quality checks: for example, Grigorian et al. [5] observe that inverse kinematics solutions can be efficiently checked using the corresponding forward kinematics algorithm, which is far cheaper as a rule. But cheap checks are not so readily available for every algorithm. An important challenge in approximate programming is to automatically generate cheap dynamic checks based on arbitrary code and distance metrics.

Specifically, given a program f and its relaxed counterpart f', we want to generate a checking function c that decides—with some confidence—whether $f'(x)$ will be a good approximation of $f(x)$. The "goodness" of an approximate output is defined using a constraint of the above form. The program can query $c(x)$ to decide whether to run $f'(x)$ or fall back to the precise $f(x)$.

4. The Common Thread: Statistical Inference

These two problems differ sharply in their use cases: one proposes a design-time programmer assistant while the other constructs run-time introspection routines. But their underlying congruence is clear: both problems need to *find probability distributions* at one point in a program that satisfy a later probabilistic constraint.

As an example, consider a simple function for computing the Euclidean distance in two-dimensional space:

```
def dist(x1, y1, x2, y2):
  return sqrt((x1 − x2) ** 2 + (y1 − y2) ** 2)
```

where the $+$ and $-$ operators are executed with approximate hardware, inducing normally distributed error on their outputs. We will

also imagine a quality constraint following the generic form above:

$$\Pr[|\mathsf{dist}_p(x_1, y_1, x_2, y_2) - \mathsf{dist}_a(x_1, y_1, x_2, y_2)| \leq 0.01] \geq 0.9$$

where dist_p and dist_a are the precise and approximate versions of the function, respectively, and the distance function is the absolute value of the difference between the outputs.

Both the approximation-assistance problem and the check-generation problem amount to inferring probability distribution parameters to satisfy the above constraint. The only difference is where the inferred distributions are introduced. We now show specifically how to formulate each as an inference problem.

Assisted approximation as inference. Say that the variance parameter σ^2 of the approximate operations' error is configurable: we can spend less energy to get higher-variance errors on each $+$ and $-$. If N represents a normal distribution, then we can write the approximate version of the function as:

```
def dist_a(x1, y1, x2, y2):
  return sqrt((x1 − x2 + N(σ1^2)) ** 2
      + (y1 − y2 + N(σ2^2)) ** 2
      + N(σ3^2))
```

The assisted approximate programming problem asks what the parameters σ_2 in each distribution call must be so that the probabilistic constraint on dist's output is met.

Check generation as inference. To formulate cheap-check generation, we introduce probability distributions at the *inputs* to the computation. (In this setting, the distribution parameters for the approximate operations are fixed—either via hand tuning or by an automatic assistant.) For example, we can assign uniform distributions $U(a, b)$ to the inputs, were a and b are minimum and maximum of the uniform's interval:

```
def dist_checked():
  x1 = U(a_x1, b_x1)
  y1 = U(a_y1, b_y1)
  x2 = U(a_x2, b_x2)
  y2 = U(a_y2, b_y2)
  return dist_a(x1, y1, x2, y2)
```

If we pair this amended program with a modified probabilistic constraint stating that outputs are likely to be *bad* rather than good:

$$\Pr[|\mathsf{dist}_p(\dots) - \mathsf{dist}_a(\dots)| \geq 0.1] \geq 0.3$$

then inferring the distribution parameters a_{x1}, b_{x1}, etc. identifies those input ranges that are likely to lead to poor approximations. A cheap run-time check can compare inputs using a simple conditional **if** $a_{\mathsf{x1}} <= \mathsf{x1} <= b_{\mathsf{x1}}$ **and** ... to identify executions that are dangerous to run approximately.

5. First Steps

Both of the challenges in this paper boil down to *statistical inference* problems where:

- The input is a *probabilistic program* in Kozen's sense [7]: an ordinary, imperative program with calls to sampling functions.
- The inference targets are the parameters of the distributions invoked by the program.
- Inference is subject to a probabilistic constraint on the output with the general form $\Pr[d(f(x), f'(x)) \leq b] \geq p$.

Completely solving this general inference problem over arbitrary probabilistic programs is clearly challenging and possibly intractable. But advances toward even partial solutions for restricted approximate kernels will pay off doubly since they help with both programming and dynamic checking with only minor adaptation.

Fortunately, the statistical inference problem is well-studied in the machine learning and statistics communities. More recently, work in the area of *probabilistic programming languages* has extended inference to statistical models expressed as programs [2, 6, 14, etc.]. But this work traditionally focuses on machine learning and models of real-world phenomena: programming languages are typically restricted and available inference algorithms tend to focus on learning from examples. Instead, the approximate programmability context requires a reframing to focus on (a) general-purpose programming languages in all their real-world complexity, and (b) inference based on probabilistic output constraints rather than concrete observations.

We have recently taken early steps toward this goal with a technique for translating general-purpose programs into graphical models [13]. This approach enables statistical reasoning over a broad class of realistic approximate programs. Future work in the approximation community should focus on bringing constraint-directed statistical inference to this model of probabilistic programs.

References

[1] Discussion at ISAT/DARPA *Workshop on Accuracy Trade-Offs Across the System Stack for Performance and Energy*, February 2014.

[2] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *AISTATS*, 2013.

[3] H. Esmaeilzadeh, K. Ni, and M. Naik. Expectation-oriented framework for automating approximate programming. Technical Report GT-CS-13-07, Georgia Institute of Technology. URL `https://smartech.gatech.edu/handle/1853/49755`.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[5] B. Grigorian and G. Reinman. Improving coverage and reliability in approximate computing using application-specific, light-weight checks. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[6] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI*, 1997.

[7] D. Kozen. Semantics of probabilistic programs. In *Symposium on Foundations of Computer Science*, pages 101–114, Oct 1979.

[8] S. Misailovic and M. Rinard. Synthesis of randomized accuracy-aware map-fold programs. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[9] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. Rinard. Reliability-aware optimization of approximate computational kernels with Rely. Technical Report MIT-CSAIL-TR-2014-001, MIT. URL `http://dspace.mit.edu/handle/1721.1/83843`.

[10] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard. Quality of service profiling. In *ICSE*, 2010.

[11] M. F. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Dynamic analysis of aproximate program quality. Technical Report UW-CSE-14-03-01, University of Washington. URL `ftp://ftp.cs.washington.edu/tr/2014/03/UW-CSE-14-03-01.PDF`.

[12] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[13] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *PLDI*, 2014.

[14] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI*, 2013.

[15] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Quality programmable vector processors for approximate computing. In *MICRO*, 2013.