ARV & PyARV: fast approximate random numbers in C & Python

Dr Oliver Sheridan-Methven*
Sunday 27th April 2025

Contents

There are some things I want to mention in this article:

- The distributions included and omitted:
  - Gaussian.
  - Non-central χ^2.
  - χ^2.
  - Poisson.
  - Student-t.
  - β.
  - Cauchy.

  And measure how these compare to what's available in Python already, (skip and what's in C or C++ as this was done in a previous paper and for now the front end is Python.).
- What are the current difficulties:
  - Compilers with scatter and gather.
  - Writing agnostic implementations that are well suited to AVX512.
  - User defined specialisations and parametrisations.

AMS MSC classifications: 11K45, 62-04, 62E17, 65-04, 65C05, 65C10, 65D15, 65Y05, 65Y20, 68W10, and 68W25.

Keywords: Approximate random variables, random number generation, numerical analysis, Monte Carlo, C, Python, and high performance computing.

Abstract

abstract

1 Introduction

In this section we introduce the reader to the topic of efficient random number generation, and introduce the ARV and PyARV packages in this context, outlining the our contributions, and detailing the structure of this report.

1.1 Random number generation

Random numbers are a staple component of the mathematical, physical, and computation sciences. Their significance stems both from their theoretical

*oliver.sheridan-methven@hotmail.co.uk.
The code used to generate the results and figures herein is hosted at github.com/oliversheridanmethven/pyarv_article.

DRAFT

underpinning of probabilistic and stochastic systems, to their applications in scientific computing, with notable examples *par excellence* including Monte Carlo simulations and cryptography. The utility of random numbers cannot be understated, as surmised by Knuth [10, p. 189]: "*Almost all good computer programs contain at least one random number generator*". Random numbers, while simple enough to comprehend intuitively, are very tricky to tackle analytically and computationally. This article details the ARV and PyARV software packages, which make available ultra fast random numbers for both low and high level applications.

Not all random numbers are created equal, and there are low quality random numbers, and high quality ones, and differing metrics by which we can judge and compare random number and their generators. Random numbers come in different flavours, such a pseudo or quasi random numbers. Furthermore, they can have differing strengths, such as whether they are cryptographically secure or not. Random numbers can come from an endless selection of possible distributions (e.g. uniform, Gaussian, Poisson, etc.). Lastly, random number generators can be computationally cheap or extremely expensive. While there are numerous random numbers which should almost always be avoided (e.g. the infamous `RANDU` [10, p. 107, 188]), no single random number generator is optimal under the lens of all these metrics. Consequently, it is left to the discretion and expertise of an application's architect to choose the random number generator(s) most suitable for their specific application's needs.

The setting we will focus our attention on concerns the speed with which random numbers can be generated. Our aim is to generate vast quantities of random numbers *en masse*, so the quicker a random number can be generated, the better. This setting is self evidently highly applicable to Monte Carlo simulation. There are two mechanism by which we can achieve our goal. The first is to ensure that the chosen random number generator's algorithm is implemented such that it achieves maximal performance on the available computational hardware, efficiently leveraging parallel calculation units (e.g. SIMD units, GPUs, FPGAs, etc.), the cache hierarchy, reduced precision data-types, etc. The second is to choose [or construct] the quickest random number generator algorithm appropriate for the desired task alongside the distribution and hardware, utilising clever approximations, dynamically adjusting and reducing precisions where permissible, etc. The ARV package exploits both these mechanisms, and the underlying implementation showed speed improvements by a factor of 7 for the simple Gaussian distribution [6, § 3.4], up to factors of 6'000 for the more complicated non-central χ^2 distribution [6, p. 26:22, tab. 5(b)].

1.2 The ARV and PyARV packages

In this paper we introduce the ARV and PyARV packages. The acronym ARV stands for *approximate random variables* [5, §,2], which are random variables sampled using the inverse transform method [7, § 2.2.1], but with the inverse cumulative distribution function replaced by an approximation. (The different nomenclature of "random numbers" and "random variables" is a minor mathematical detail we will overlook, and henceforth we will freely interchange between the two). Numerous approximations (and analyses) have been produced, including the approximation by constants, polynomials, piecewise polynomials, dyadic intervals, Padé approximations, Chebyshev polynomials, and several other methods [1, 3, 5, 9, 16]. The ARV package utilises piecewise polynomial approximations on dyadic subintervals, the analysis and implementation of which is detailed by Giles and Sheridan-Methven [5], and highly optimised specialist implementations utilising inlined assembly targetting Intel AVX512 and Arm ThunderX2 SVE VLA architectures are detailed by Sheridan-Methven [16, § 4.3.3]. The ARV package is a C library implementing and extending upon the algorithms outlined in [5], and PyARV is a library which wraps ARV, exposing a Python interface.

The goal of the ARV and PyARV packages is to take the algorithms detailed in [5] and their prototype implementations from [15], and package these as a polished, coherent, structured, professional grade software library, with thorough documentation, tests, and error handling, and to make these available to the Python and C-style languages. The motivation to do so is to ease and encourage further work utilising approximate random variables (both from academia and industry), to consolidate and refine the current implementations, and to improve the accessibility for both high and low level programming languages.

Notes:

DRAFT

1.3 Applications suitable for approximate random variables

Before proceeding with the remainder of this report, we should make clear our intended audience by detailing which applications would benefit from using approximate random variables, and which would not.

Our target audience who are most likely to benefit from approximate variables, and find the ARV and PyARV packages beneficial and interesting, include practitioners and researchers investigating:

- High performance computing applications that have random number generation as their primary bottle neck. Obvious examples typically include Monte Carlo applications in finance, weather forecasting, etc.
- Programs computing random numbers on vectorised hardware, such as modern CPUs, GPUs, FPGAs, etc.
- Applications using very low precisions data types.
- Mathematical and statistical software libraries.
- Numerical analysis of quasi-random numbers, low discrepancy sequences, Monte Carlo methods, function approximation, rounding error models, stochastic rounding, alternative number representations (e.g. posits), etc.

Importantly though, we should detail **inappropriate applications** for approximate random variables, which would include the scenarios where:

- Applications have not undergone considerable optimisation and performance profiling.
- Programs run on antiquated or low-end hardware, or embedded devices without floating point support.
- Random number generation is not the dominant computational cost.
- Random variables are drawn only from the uniform distribution.
- Cryptographically secure random numbers are required or applications are security critical running in malicious environments.[1]
- Applications require highly accurate evaluations of certain statistical properties, such as double precision accurate estimation of percentile values in some statistical tests.

[1] These settings do not necessarily preclude the use of approximate random variables, but such settings were not prioritised nor analysed in any way during the development of the underlying methods used in our construction of approximate random variables. Consequently, we make no guarantees about their utility or suitability for such situations.

1.4 Contributions of this report

Our research has generated the following contributions, which we list in the order of their significance:

ARV: A coherent, polished, and hardware agnostic C library implementing approximate random variables for a range of distributions. The implementation is highly optimised to fully exploit SIMD capable vectorised CPUs. The library is easily portable to other C-style languages such as e.g. C++.

PyARV: A Python library exposing the ARV functionality.

More distributions implemented: Alongside the Gaussian and non-central χ^2 distributions, we now support implementations for

the Poisson distribution?

Programming know-how and compiler shortcomings: Several limitations of compilers and their ability to effectively vectorise code patterns have been identified. In doing so we have learnt, and can herein share, much of the know-how about how to write code amenable to vectorising compilers.

1.5 Structure of this report

The remains of this report are structured as follows:

Section 2: Introduces the mathematical preliminaries necessary to understand what ARV is doing in its implementation and to explain the PyARV interface.

Section 3: Showcases how to get started with the PyARV package, covering installation, performance checks, and a simple usage example.

Section 4: Discusses several of the design choices made, what the package's goals are and are not.

Section 5: Explains the more advanced features and capabilities, such as dynamically generated approximations and parametrised distributions.

Section 6: Demonstrates numerous examples benefiting from our packages.

Section 7: Explores the future plans for the project

Notes:

DRAFT

and how people can meaningfully contribute to the effort.

Section 8: Presents the conclusions from this report.

2 Mathematical preliminaries

To get started with the ARV and PyARV packages, we will first need to know at a sufficiently high level what ARV it is doing. In this section we very briefly present the basic mathematical preliminaries.

2.1 A brief mathematical overview

Beginning with a very brief mathematical introduction, we say a random variable X is described by a probability distribution $\mathcal{D}$, which details how likely a given outcome is (e.g. flipping a coin, rolling a dice, etc.), and denote X as *following* a distribution $\mathcal{D}$ by $X \sim \mathcal{D}$. The probability of a random variable X taking a value less than or equal to some parameter x is described by the *cumulative distribution function* $C_{\mathcal{D}}(x)$, where $C(x) \coloneqq \mathbb{P}(X \leq x)$. Supposing we wished to simulate n random variables $X_i \sim \mathcal{D}$ for $i \in \{1, 2, \ldots, n\}$, then we could sample the random variable X_i by computing

$$X_i \coloneqq C_{\mathcal{D}}^{-1}(U_i), \tag{2.1}$$

where $U_i \sim \mathcal{U}$ is a uniform random variable where $C_{\mathcal{U}}(u) \coloneqq u$ for $u \in (0, 1)$ [10, § 3.4.1.B]. The function C^{-1} is the *inverse cumulative distribution function*, and (2.1) is known as the *inverse transform method* [7, § 2.2.1]. Approximate random variables are produced by replacing $C^{-1} \to \widetilde{C}^{-1}$ in (2.1), where $\widetilde{C}^{-1}$ is some approximation to C^{-1}, giving

$$\widetilde{X}_i \coloneqq \widetilde{C}_{\mathcal{D}}^{-1}(U_i). \tag{2.2}$$

We call an X produced by (2.1) an *exact random variable* following the *exact distribution* $\mathcal{D}$, and an $\widetilde{X}$ produced by (2.2) an *approximate random variable* following the *approximate distribution* $\widetilde{\mathcal{D}}$. The approximations ARV uses are piecewise polynomial approximations on dyadic subintervals, where the polynomials are usually linear or cubic, only a handful of dyadic subintervals are used. A precise definition, construction, analysis, and computational exposition is given in [5], and a detailed error analysis of their use in multilevel Monte Carlo applications is given in [4].

The most important point to take away from this is that to sample approximate random variables from a given distribution, we must first generate samples following a uniform distribution, and then we *transform* these uniform samples into the desired distribution. Notice that in (2.1) and (2.2), both transform exact uniform random variables.[2] We do not sample from the desired distribution directly. Consequently, there is an extremely subtle and important point of clarification that must be emphasised: **the ARV and PyARV packages do not provide random number generators**. What they do provide are libraries which *transform* uniform random variables into approximate random variables approximately following the desired distribution. The random number generators used to generate the uniform random variables must be externally provided *a priori*, and can include pseudo or quasi-random number generators.

Do not be fooled though by the words "approximation" and "linear polynomials" into thinking that the approximate random variables produced are of a poor or unusable quality, when in fact quite the opposite is true. The quality of the approximations, as measured using a mathematical quantity called the L^2-norm, are surprisingly accurate. This measure of quality is quite different to how most libraries measure the quality of their approximations. A "normal" library would measure its quality by quoting its error measured using the L^∞-norm, corresponding to the "worst case error". Instead, by measuring ARV's error using the L^2-norm, this corresponds to the "average case error". This means that while our library is usually quite accurate, for certain inputs it can have very large errors, but on average these happen so incredibly infrequently for uniformly random inputs that the overall accuracy is still very good. Overall, the linear approximations using only a small numbers of subintervals are very accurate with incredible speed for appropriate applications (e.g. Monte Carlo simulations). If the user wishes to reduce the frequency of the high error cases, they can easily do so by slightly increasing the number of subintervals, and to reduce the typical approximation error, they just increase the polynomial order a small amount. To give a sense of speed, these approximations are typically an order of magnitude (or several) faster than readily available solutions (both proprietary and open source), out competing C++'s Boost, Matlab, Intel's MKL, GNU's GSL, Arm, NAG, C and NumPy's Cephes library, etc. [5, § 3.4]. We should

[2] That (2.1) and (2.2) transform the same uniform random variables is the key for the coupling mechanism in multilevel Monte Carlo applications.

Notes:

DRAFT

emphasise though that convergence is only in the L^2-norm. Users who are expecting convergence in any other sense may be sorely mistaken. For example, attempts to plot the convergence in the probability mass functions produce very wild looking results [6, fig. 2.1b]!

Readers familiar with methods for efficiently generating Gaussian random variables may ask about other methods besides the inverse transform method, such as the Ziggurat [12], Box-Muller [2], and Marsaglia [11] methods, which are all heavily used in practice. While reasonable for scalar computational modes, they are ill position for high performance on vectorised SIMD hardware, and a detailed explanation of this is found in [5, § 3.1.1]. In addition to these pitfalls, such methods are restricted to one specific distribution, whereas the inverse transform method is applicable to all probability distributions, and by extension so too are approximate random variables.

Lastly the fact we provide a transformation library, rather than a random number generation library, is an advantage, rather than a short coming. This means the user can input either pseudo random or quasi random variables as inputs. Additionally, this provided a coupling mechanism central for multilevel Monte Carlo simulations. Giles and Sheridan-Methven [4] show how this coupling mechanism can be utilised to ensure accuracy is maintained alongside gaining the speed improvements.

2.2 Examples of approximate random variables

The reader may already encountered approximate random variables without realising it, and arguably the simplest possible non-trivial example would be the use of Rademacher random variables [**?**]. Rademacher random variables are an approximation of Gaussian random variables and are sampled from $\{-1, +1\}$ with equal probability. This is an exceptionally crude approximation (where only the mean is correct), but is computationally exceedingly fast. These Rademacher random variables can be seen as a special case of the general piecewise polynomial approximation using dyadic intervals presented in [5], where there is a single dyadic interval in use, the polynomial is the zeroth order constant approximation, and the constant is chosen to be ± 1 for computational convenience rather some other value determined by an L^p minimisation. Approximating Gaussian random variables with Rademacher random variables for simulating stochastic differential equations is a long standing practice and is known as *the weak Euler-Maruyama scheme* [7, p. XXXII].

2.3 Using low precision approximate random variables with confidence

For a reader first coming across approximate random variables, especially low precision ones (which might use e.g. half precision), they may have several concerns and reservations. The most typical hesitation is whether they can still be used in applications which don't wish to compromise on accuracy, but wish to benefit from the vast potential for speed improvements. When presented with examples such as the weak Euler-Maruyama scheme and approximations as drastically crude as Rademacher random variables, it's not obvious what level of error may be introduced by switching to approximate random variables, and if this can be mitigated. Thankfully, several of these theoretical and operational issues have undergone a rigorous mathematical analyses, and there are positive and encouraging results to each of these concerns. We very briefly surmise three papers by Giles and Sheridan-Methven which directly address these various questions, and refer the reader to them if they want the full mathematical proofs and precise definitions and problem formulations.

Simulation accuracy: [4] demonstrates using approximate random variables in a nested multilevel Monte Carlo framework so their speed benefits are realised whilst full accuracy is maintained. Error bounds are constructed, and limitations surrounding non differentiability analysed.

Constructing approximations: [5] details the precise construction of approximate random variables, the quality of their approximation, and the implementation details necessary to achieve ultra fast computational performance.

Low precisions: [6] inspects the consequences of moving to low precisions, such as e.g. 16 bit half precision, and the accumulation of rounding error. A novel rounding error model is produced, showing that such low precisions can (and should) be utilised in the majority of coarse grained simulations, and that accuracy needn't be compromised.

Like any technology or mathematical technique, approximate random variables can be misused, but

Notes:

DRAFT

the aforementioned papers can give the reader confidence that they can leverage the ARV and PyARV packages effectively and safely. Furthermore, these provide precise technical references which they can consult to assure themselves whether they have a setting appropriate for approximate random variables or otherwise.

While [4–6] directly address our specific class of approximate random variables, there is similar (but less specific) literature on the same broader topic. For example, approximations using moment matching schemes are discussed by Müller et al. [14], approximate random variables and dynamic precisions for use on FPGAs is discussed by Haas and Giles [8], etc.

Present a set of reference material

3 Getting started with PyARV

In this section we show how to use the PyARV package, covering the required dependencies, installation, tests, and writing your first program using the PyARV package. We will primarily showcase the PyARV frontend for simplicity and because it is currently the primary means of installing the functionality, but nonetheless the ARV library be similarly interfaced. PyARV is currently only tested on Unix-style operating systems (e.g. MacOS and GNU/Linux), and support for Windows may come at a later date, (any operability on Windows is currently entirely good fortune and coincidental).

3.1 Dependencies

This package is targetting users who care considerably about high performance computing. Consequently, we have given ourselves license to operate at the cutting edge of what modern software and hardware can support. As such, our libraries have been written and developed with the latest toolchains in mind. At the time of writing, this has meant we have used Python 3.13, C 23, SIMD capable hardware, GCC 14 and Clang 19, recent versions of CMake, etc.[3] However, the author tries to avoid making assumptions about the hardware and user's environment as much as possible. For this reason, while our implementations are very high performance, if the reader wants the *ultimate* in performance, really they should be using a compiled language, using the ARV library, and they have to take responsibility for multi-threading, thread pinning, compilations targetting specific hardware, memory alignment, data precisions, and related high performance computing matters.

[3] Please consult the repository for the exact requirements, as these are continually incrementing.

3.2 Installation

Installing the package is as simple as running `pip install pyarv`. To check the package is installed correctly, you can run the package's tests with `python -m unittest discover pyarv`.

3.3 A simple example

To use the PyARV library in Python applications, you pick the desired distribution and transform NumPy arrays of uniform random variables. An example is given in code 1.

```
1  import numpy as np
2  from numpy.random import uniform
3  from scipy.stats import norm
4  from pyarv.gaussian import Gaussian
5
6  size = 10_000_000
7  u = uniform(size=size).astype(np.float32)
8  # Approximate random variables
9  approx = Gaussian(order=1).transform(u)
10 # Exact random variables
11 exact = norm.ppf(u)
```

Code 1 – Example usage of the PyARV API.

In code 1 we import `numpy` and `scipy` as one usually would, but additionally on line 4 we import the piecewise linear approximation to the Gaussian distribution from the `pyarv` package. We generate a large number of uniform random variables, and for compatibility with [and computationally efficiency within] PyARV these uniforms random variables are cast on line 7 to 32 bit single precision floating point data types. On line 9 we transform these into their corresponding approximate random variables `approx` using an order 1 (a.k.a. linear) piecewise polynomial. For comparison, the transformation to give exact random variables `exact` using (2.1) is shown on line 11.

3.4 Checking the performance

The package's installation involves extensive and heavily optimised compilation of C modules, so it is important to double check that the package has not only compiled, but been well optimised too. To assess

Notes:

DRAFT

this, the package proved some example scripts to demonstrate its high performance. To see these demonstrations run `python -m <module>`, substituting `<module>` with either of the modules `pyarv.non_central_chi_squared.demos.speed` or `pyarv.gaussian.demos.speed`. Consult the output generated to verify that indeed PyARV is much fast than the equivalent routines coming from either NumPy or SciPy. The results from the Gaussian example are surmised in table 3.1.

Package	Function	Time
SciPy	`ppf`	220(20)
PyARV	`transform`	5(1)

(a) The inverse transform method.

Package	Function	Time
NumPy	`uniform`	45(5)
PyTorch	`rand`	25(5)

(b) Uniform random numbers.

Package	Function	Time
NumPy	`normal`	110(5)
PyTorch	`randn`	75(5)

(c) Gaussian random numbers.

Table 3.1 – Computational costs (ms) for generating 10^7 random numbers using 32 bit precision with PyARV and the other major Python packages.

At the time of writing, on the author's laptop[4] the inverse transform method is over 30 and 300 to 4'500 times faster than SciPy for the Gaussian and non-central χ^2 distributions respectively, and 20 times faster than NumPy for the Gaussian distribution. The approximation is so fast that in fact the uniform random number generation in NumPy has become the computational bottleneck. Switching the uniform random generation from NumPy to PyTorch considerably improved the speed to generate the uniform random numbers, but does not change the conclusions. Accounting for the costs of generating the uniform random numbers in the first place, then PyARV is still several thousand times faster for the non-central χ^2 distribution, and for the Gaussian distribution is 3 times faster than SciPy and 2 times faster than NumPy. Such speed improvements are not to be scoffed at, and even though we are so fast that we now violate an *a priori* presumptions of approximate random variables—that generating the uniforms is negligible in comparison to the transform in (2.1)—the approach still offers considerable time savings. As can be seen in this example, the savings only improve as we transition from simpler to more challenging distributions.

[4] A 48 GB 2023 Apple MacBook Pro M3 Max 16 core running MacOS Sequoia 15.3.2 Darwin 24.3.0 AArch64 ArmV8.3 which is NEON SIMD capable.

Hardware support for 16 bit half precision floating point is not universal, and so ARV currently implements our approximations for 32 bit single precision floating point data types. For users wondering if such low precisions can be meaningfully utilised in Monte Carlo simulations, Giles and Sheridan-Methven [6] develop an appropriate round error model and derive round error bounds and convergence rates for multilevel Monte Carlo simulations using low precision approximate random variables. There results indicate potential speed ups by a factor of 14 from switching to 16 bit half precision floating point data types. Recent work by Haas and Giles [8] into varying precision multilevel Monte Carlo applications using approximate random variables on FPGAs indicate further speed improvements by a factor of 5 to 7 are possible.

4 Design choices

In this section we discuss several of choices made when designing and implementing ARV and PyARV.

4.1 Implementation decisions and frequently asked questions

Several decisions were made in the implementation of ARV and PyARV, and based on preliminary discussions, we collect together and answer some of the more pivotal and frequently asked questions.

4.1.1 Why make a Python frontend?

Many developers may consider the phrase "high performance Python" an oxymoron, and certainly there is a degree of truth in this. For the ultimate performance it is necessary for applications to concern themselves with multithreading, memory management, alignment, cache usage, data sizes, data bandwidths, vector hardware and accelerator utilisation, etc., and most of these are inaccessible in Python. Consequently, high performance computing is usually done in low level compiled languages such as C, C++ and Fortran. However, Python regularly acts as a front end providing a user friendly interface on top of lower level and higher performance back

Notes:

DRAFT

end implementations (examples include `numpy`, `polars`, etc.). As Python is increasingly popular for both research and production applications, rather than forsake the Python community, we wrote PyARV to enable them to benefit in large part from the performance achievable from the ARV package.

4.1.2 Why implement ARV in C?

A high performance implementation of approximate random variables requires the use of a low level compiled language. The most important capabilities required is being able to specify the sizes of the data types, and have fine grained control over performing vectorised floating point operations. This leaves C, C++, Fortran, and Rust as notable candidates. The author chose C for the following reasons:

Simplicity: C is simple and minimalist.

Portability: C is the *lingua franca* for high performance computing and portability. Most languages support C extensions and bindings. Importantly, NumPy only offers a C based API, and C++ can use C functionality through C++'s `extern` keyword.

Memory aliasing: C supports the `restrict` keyword, which is core to our implementation.

Compiler support: The language is extremely well supported by compilers.

Type punning: This is defined behaviour in C using `union` types, and is utilised for very fast bit manipulation and indexing operations on floating point data types.

4.1.3 Why two separate packages?

We separated out the high performance ARV implementation from the Python binding. This is because we want the core library to be accessible to non Python high performance users. Although ARV and PyARV are very closely coupled, in the future we hope ARV will have several other binding interfaces beyond PyARV, so it was important not to keep the ARV implementation stand alone. However, packaging and distributing the two separately is difficult, so in lieu of further demand and developer support, the two are bundled together in one repository and build process for now.

4.1.4 Why use CMake and `setuptools`?

The use of C bindings and custom compilations and builds in PyARV necessitates `setuptools` and `setup.py` to manage building the Python package, and precludes use of other tools and methods such as uv, poetry, hatchling, `pyproject.toml`, etc. To manage the build system for the ARV C library and the NumPy and Python C APIs we chose CMake as this is well maintained, documented, and supports build process intermixing Python and C via `scikit-build`.

4.1.5 Why single precision?

The package is intended to work with floating point data types, and the C standard defines `Float16`, `float`, `double`, and `long double`, which *usually* correspond to 16, 32, 64, and 80 bit data types respectively.

double check the float16 data type name and long double size

Of these, the ARV package implements approximations using the single precision `float` type assuming a 32 bit precision IEEE floating point representation.

clarify and cite that this is IEEE 754, and possibly also the IEEE-656(?) spec as well.

In the NumPy C API, the NumPy `float32` data type maps to the C `float` type, and hence in code 1 we perform the cast operation on line 7. The question remains though as to why other precisions are not yet supported? As was demonstrated in [6], for stochastic simulation settings, which are the motivating example *par excellence* for using approximate random variables, double precision was superfluously precise,[5] and half precision quickly required compensation schemes (such as Kahan compensated summation [**?**]) to fend off rounding error [6, fig. 3.1]. Consequently, single precision presents a suitable primary precision for a first release of our packages. Users who haven't yet investigated dropping the precisions of their floating point calculations should almost certainly do so *before* using our packages.

[5] The calculation of derivatives and sensitivities by finite difference based methods (a.k.a. Greeks) is an obvious case where higher precision data types are usually still desirable.

Notes:

DRAFT

4.1.6 Why are there precomputed coefficient tables?

Users perusing the C implementation of ARV will notice that coefficient tables for the linear and cubic approximations may be pre-computed. This is atypical, as most of the approximations will have the coefficient tables passed in from the Python interface in PyARV. The explanation for this is improved performance. The default tables contain 16 coefficients, which for 32 bit floating point data types, totals 512 bit. Being able to pack the polynomial coefficients into this many bits means that they can all be read in a single cache line, and are small enough to fit inside a single vector register. Furthermore, if there are only one a small few of these tables (such as for low order linear or cubic approximations), this means all the coefficients can reside exclusively in vector registers, and so is even quicker to access than the L1-cache. As these arrays are of a known fixed size and are known at compile time, the compiler is free to exploit this for sufficiently high optimisation settings. Of course, users compiling the package on machines with smaller vector widths (typical for laptops and desktops) will not be able to enjoy these benefits to the same degree. Similarly, users who wish to use more arbitrary polynomial orders and table sizes will have to pass in arrays of coefficients into ARV which are not known at compile time, so won't be as amenable to compiler optimisations.

4.1.7 Logging, tests, documentation, etc.

The ARV package is relatively light weight, and as it is intended to be used in an intensive computational pipeline, it does not perform any logging. There are such a small number of supporting routines in the ARV package that we have not required written a testing library installation, but instead enforce...

Decide what I want to do here for testing. I can either require some tests written using Criterion, write my own tests, (e.g. using static assert), or just have all my tests written in the Python layer. All that really needs testing from ARV is the approximation.h functions which do some type punning, getting indices from floats. I am hesitant to say that it is really so trivial it doesn't need testing... (certainly doesn't warrant an entire testing library).

The PyARV front end has unit tests, but not regression tests (as these are too tightly coupled to the underlying hardware and compilers). Again there is no logging in PyARV as it is not needed. The API for both packages is documented and hosted on github.

Try the C-API for mkdocstring.

There is not much that can be tested in general in the PyArv package, other than handling bad input, but otherwise we do some tests for zero values for symmetric distributions and certain things are monotonically increasing, but much of this doesn't hold, so there are very few things that in face make sense to test.

Mention that it is the users responsibility to ensure input is sufficiently sanitised, and we do not make any guarantees about subnormal numbers, the exact values of zero and 1, minus zero, nan, signalling nan, etc. Similarly, we require the user to preallocate correct arrays with suitable alignment and correct lengths, and all the rest. We trust the user to read the documentation and write correct code.

Mention that as we do some checking on the arrays (contiguous, right length, etc., and then in the C backend we rely on the compiler to check the types, we don't do *much* python type checking with tools such as mypy.

4.2 The package's goals and omissions

The package's goals are quite simple:

ARV: To provide and showcase a reusable, lean, and highly optimised implementation of approximate random numbers which out competes all other major random number generators and commercial libraries.

PyARV: To make many of the benefits of ARV accessible to the Python community.

These are clearly very lofty ambitions, and to achieve these our packages have had to undertake certain responsibilities and make several assumptions. Even more importantly perhaps are the range of omissions and what has been designated as being outside of the scope for the packages.

In order to achieve high performance, we assume that the user cares sufficiently enough about high performance that they are using the most up-to-date technologies and hardwares available. Our assumptions and requirements include:

- Extremely recent versions of Python 3.13 and C 23, and similarly Clang 19 or GCC 14
- A high end CPU supporting vectorised SIMD floating point operations (e.g. AVX512, SVE,

Notes:

etc.).

- The user will manage populating and pinning processes to each available thread.
- A large, aligned, and contiguous supply of uniform random variables in the correct data type in the appropriate container with the correct access specifiers (e.g. non `volatile` arrays).
- A UNIX-style operating system.
- A POSIX file system.
- No competition against others programs (or possibly a hypervisor) for system resources.

Just as important are what we consider outside of the purview of our packages, which include:

- Specialisations targetting specific hardwares, such as Intel AVX512 or Arm SVE. Specific optimisations for these written using inline assembly can be found in [**?**], and are discussed in depth in [16, § 4.3.3].
- Specialisations targetting specific compilers, such as Intel's `icc` or Nvidia's `nvcc`.
- Distributed systems running across multiple threads and machines.
- Non standard systems and implementations. We assume the C compiler is fully standards compliant and implements all the optional parts of the C standard pertaining to floating point arithmetic. We do not support extensions of the C language offered by compilers. Analogous assumptions hold for Python and NumPy.
- Non IEEE floating point implementations.
- Floating point rounding modes.
- Big or little endian storage.
- The memory model and whether this is unified, NUMA, etc., and also the cache hierarchy on the specific machine.
- The atomicity of operations.
- Commercial or non-mainstream systems.
- Languages other than C and Python.
- Legacy systems and non-recent language versions.

In an effort to keep the implementation lean and generic, we have purposefully excluded settings specific to any particular hardware or compiler. To keep the code base manageable and succinct, we assume standards compliance, IEEE representations. Additionally, as a matter of pragmatism, scalability, and expense, we are unable to test the fully range of commercial compilers, software libraries, and hardwares that exist, whose multiplicity is far too expansive, and so limit our report to what is readily available to the author.

5 Advanced usage

5.1 Using the API

- Higher order polynomials
- More dyadic intervals.
- high accuracy approximations.
- Parametrised distributions.
- Asymmetric distributions.
- Custom distributions.
- For the custom distributions, showcase for a variety of these how fast they are.
- Discuss distributions which do not have an L2 norm, such as the Cauchy distribution, and how this related to the proofs in the ACM toms paper.

Mention that if the polynomial order is known at compile time, then rather than a generic for loop, the polynomial evaluation can be unrolled, hence why we provide linear and cubic routines out of the box. Similarly for a fixed number of dyadic intervals we know the storage requirements for these and they can live on the stack rather than the heap, and sufficiently small intervals and low polynomial orders for non-parametric distributions can fit in the L1 cache or even in the registers, and can be read in a single cache line!

Notes:

DRAFT

6 Examples

Give an example in the following settings:

- using a naive Rademacher Random variables and using a zeroth order single interval dyadic.
- Give a Gaussian example with simple Monte Carlo estimation (e.g. estiamte pi or something similar as an example of numerical integration unrelated to SDEs, or some high dimensional problem).
- Give an SDE example pricing a call option, so we can compare to the analytic formula.
- Give a more involved CIR example with the non central chi-squared distribution.
- Give a multi-level Monte Carlo application showing the impact of the MLMC correction step, showing accuracy is maintained.
- Find some example using a Possion distribution.
- Perhaps I can give an example for hypothesis testing parameters from some empirical distributions, such as parameter estimation for a Dirchlet distribution from a Bayesian framework.
- Perform a Bayesian Monte Carlo or Markov Chain Monte Carlo using some empirical distribution or some custom distribution which might be some polynomial (maybe a polynomial is a poor example, so perhaps a sinusoidal distribution with a discontinuity)
- Can I give an example using a Cauchy distribution?
- Find examples from outside Monte Carlo. (Chaos theory, weather simulation, random matrices, etc.)
- Give an example using quasi-random numbers.

For several of these examples, they could easily be measured by the metric of "time to achieve a given accuracy", or conversely "accuracy achieved in a given time" (e.g. 5-10 mins). I think the "accuracy achieved in a given time" for the Monte Carlo applications could be readily measured by using a timeout decorator (like the one in orsm) and then the error in the estimate (given by the empirical variance) can be compared. We can do a sanity check that the exact answer is within the expected bounds and see the exact level of error (although this is a random variable), so this might require several repetitions and some bootstrapping.

7 Future plans

In this section we outline our plans for the future of ARV and PyARV. In addition, we invite contributions to the project, highlighting the avenues of work which would most benefit and advance the project.

7.1 Major changes and features

In this report, we are announcing only the first release of ARV and PyARV. Whilst we hope the packages will both pique the readers' interest and benefit numerous computer applications, should our packages gain traction, we anticipate there are several new features and major changes that would follow in future releases. These would include (but are certainly not limited to):

More data types: Currently only 32 bit single precision is supported. We do anticipate utility in supporting 16 bit half precision, and perhaps even 64 bit double precision.

Shipping the packages separately: PyARV is so tightly coupled to ARV that in this first release is makes sense to ship the two packages together. In the future we would prefer to split these into two separately managed projects.

More distributions of interest: While we support dynamically constructed user approximations, having more distributions with native support would increase the utility of ARV and likely offer superior performance resulting from compiler optimisations.

More language bindings: Numerous other languages besides Python would benefit from being able to interface to ARV.

Equally important perhaps is to mention what we anticipate as very unlikely to change in any future releases:

Exclusively percentiles: Most statistical libraries offer functions to evaluate numerous properties of a probability distribution, such as its cumulative distribution function, the probability mass function, the moment generating function, etc. The *only* function we offer is the inverse cumulative distribution function.

Multithreading: The user is solely responsible for ensuring threads are utilised and handling distributed workloads.

Accelerators: Support for accelerators, such as GPUs, while possible, will not be supported by ARV.

Specalisations: Hardware specific specialisations will not be supported by ARV.

Non IEEE floats: The only floating point standard

Notes:

DRAFT

supported by ARV will be IEEE floating point for the foreseeable future.

7.2 Contributing to the project

At the time of writing, birthing this project into a publishable state has been the individual effort of the author. However, the author's time and skills are finite, and the project would benefit hugely from a wider pool of contributors. A non exhaustive list of tasks which would considerably benefit the project include:

CICD: The project is currently built and tested manually, and would benefit immensely from a proper CICD framework, whereby the tests, builds, documentation generation, package publishing, formatters, sanitizers, type-checkers, etc., are run in an automated and continuous fashion of GitHub for a range of platforms and environments.

Packaging and distribution: The ARV package is currently a subcomponent of the PyARV package, and is not stand-alone. Splitting this off into a separate stand-alone library and setting up the build and packaging system to manage dependency of PyARV on ARV would greatly improve the project's organisation and dependencies.

Build systems: As PyARV is built on bindings over C modules, we are required (to the best of our knowledge) to rely on `setuptools` and `setup.py` to manage python packaging. This is not easy. Similarly, to manage the ARV component, we rely on `cmake`, and managing the reliance on compilers and C libraries is trickier still. If we can port these setups to easier to use and more portable solutions, this would improve the robustness and purview of the library's packaging and shipping. Ideally then we could install the package easily on a range of operating systems, and install the necessary dependencies where required.

Type safety: The ARV suite is written in C, so leverages the type system there, and the PyARV suite uses Python type annotations, but marrying the two should be done both for documentation tools and type checking tools (e.g. `mypy`).

Lower precisions: The implementation utilises 32 bit single precision floating point data types for its calculations. Support for 16 bit half precision floating point types would be interesting.

Windows: The code is written on a MacOS platform, but targets all Unix-style operating systems (i.e. GNU/Linux). Windows support would be beneficial.

Documentation: The documentation can always be improved, both being more thorough, having wider coverage, and having automated components, such as code outputs being dynamically generated rather than manually copied as text. The C codebase would also benefit from adopting the very new `mkdocstrings` C-handler [13].

Tests: There can always be more tests, targetting different Python versions, hardwares, etc., and these should be automated, include coverage reports, and possibly also use sanitisers and touch upon fuzzing.

Compilation: Having several compilers optimise the code fully has proved a tricky and fragile dark art. In several places the compiler struggles or fails, and expertise on improving, the code, the compiler support, and the compiler diagnostics would be invaluable.

Other languages: The core ARV package has only be wrapped for Python. It would be a small effort to port or wrap the implementations to other languages. An obvious and immediate candidate would be C++, and then further candidates include Rust, Julia, etc. Expertise on how to handle the type-punning, memory restrictions, and compiler directives would be required, and possibly also the packaging.

If the reader is able to provide *any* assistance or guidance in these areas (or knows someone who could), **please contact the author**, or submit pull requests, issues, code reviews, feature requests, etc., at the project's repository:

`github.com/oliversheridanmethven/pyarv`

8 Conclusions

write the conclusions, and perhaps thank anyone who wants to proof read or volunteer code

References

[1] J. M. Blair, C. A. Edwards, and J. H. Johnson. Rational Chebyshev approximations for the inverse of the error

Notes:

DRAFT

function. *Mathematics of computation*, 30(136):827–830, 1976.

[2] George E.P. Box and Mervin E. Muller. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29:610–611, 1958.

[3] Ray C. C. Cheung, Dong-U Lee, Wayne Luk, and John D. Villasenor. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(8):952–962, 2007.

[4] Michael B. Giles and Oliver Sheridan-Methven. Analysis of nested multilevel Monte Carlo using approximate normal random variables. *SIAM/ASA Journal on Uncertainty Quantification*, 10(1):200–226, 2022.

[5] Michael B. Giles and Oliver Sheridan-Methven. Approximating inverse cumulative distribution functions to produce approximate random variables. *ACM Transactions on Mathematical Software*, 49(3), September 2023.

[6] Michael B. Giles and Oliver Sheridan-Methven. Rounding error using low precision approximate random variables. *SIAM Journal on Scientific Computing*, 46(4):B502–B526, 2024.

[7] Paul Glasserman. *Monte Carlo methods in financial engineering*, volume 53 of *Stochastic modelling and applied probability*. Springer, 2003.

[8] Irina-Beatrice Haas and Michael B. Giles. A nested MLMC framework for efficient simulations on FPGAs. *Monte Carlo Methods and Applications*, 2025. Accepted for publication, pending journal issue.

[9] Cecil Hastings Jr, Jeanne T. Wayward, and James P. Wong Jr. *Approximations for digital computers*. Princeton University press, 1955.

[10] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms, Volume 2*. Addison-Wesley, 3rd edition, June 2019. 37th printing.

[11] George Marsaglia and T.A. Bray. A convenient method for generating normal variables. *SIAM Review*, 6(3):260–264, 1964.

[12] George Marsaglia and Wai Wan Tsang. The Ziggurat method for generating random variables. *Journal of statistical software*, 5(1):1–7, 2000.

[13] Timothée Mazzucotelli. mkdocstrings-c: a C handler for mkdocstrings, 2025. URL `github.com/mkdocstrings/c`. GitHub repository.

[14] Eike H. Müller, Rob Scheichl, and Tony Shardlow. Improving multilevel Monte Carlo for stochastic differential equations with application to the Langevin equation. *Proceedings of the royal society of London A: mathematical, physical and engineering sciences*, 471 (2176), 2015.

[15] Oliver Sheridan-Methven. Getting started with approximate random variables: a brief guide for practitioners, 2020. URL `github.com/oliversheridanmethven/approximate_random_variables`. GitHub repository.

[16] Oliver Sheridan-Methven. Nested multilevel Monte Carlo methods and a modified Euler-Maruyama scheme utilising approximate Gaussian random variables suitable for vectorised hardware and low-precisions, 2021. DPhil. thesis, Mathematical Institute, University of Oxford.

DRAFT

Notes: