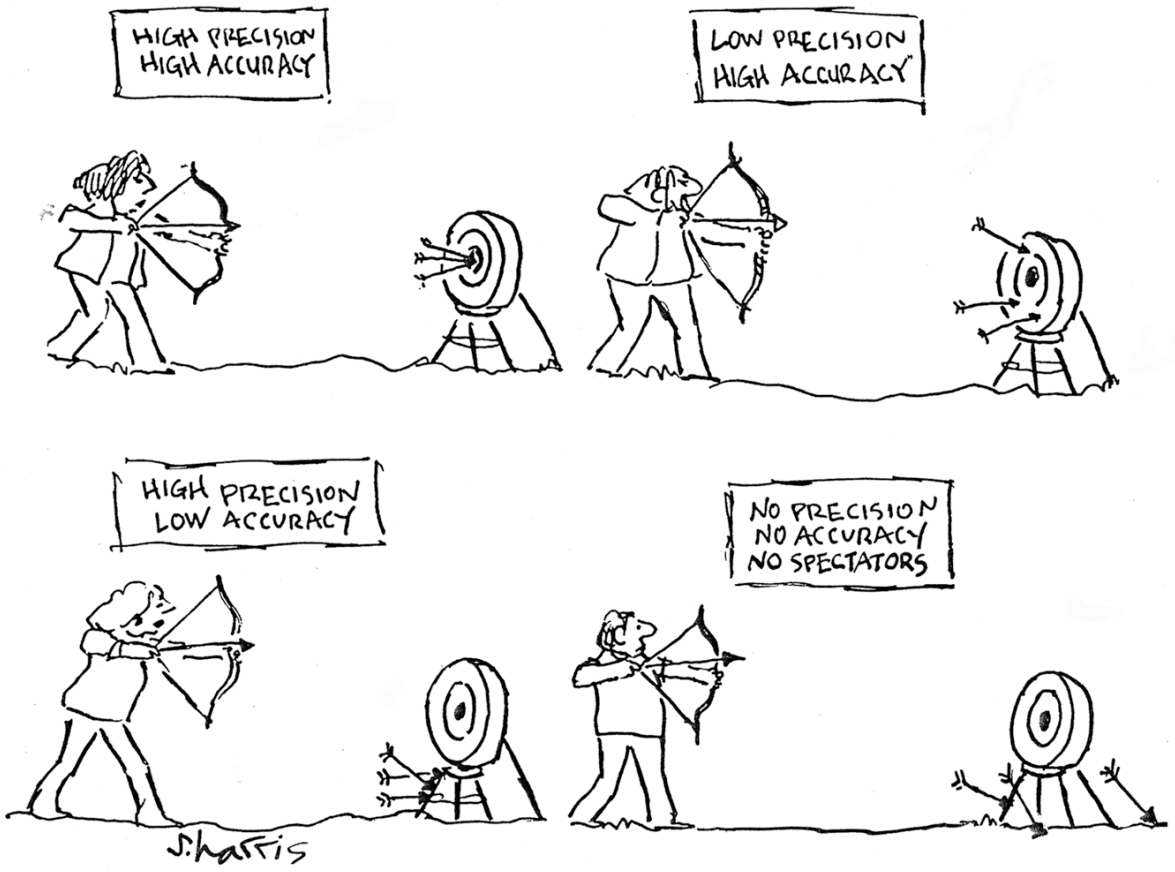
Getting started with approximate random variables: a brief guide for practitioners

Oliver Sheridan-Methven*

Saturday 17th October 2020

Abstract

We introduce the idea of *approximate random variables* and discuss when they are useful. We provide code bundles in Python and C which demonstrate various implementations and use-cases, showing key results, and how to obtain these yourself using the code provided.



*oliver.sheridan-methven@maths.ox.ac.uk

Contents

1 Pre-requisites for the demonstrations

The demonstrations we provide will mostly be written in Python, and a few higher performance versions will be in C. For the Python code, we will be using Python 2, although it should all be easily portable to Python 3 with minimal modification. The Python files can either be run from a Python IDLE, or from the command line, whichever is easier. We assume a Unix or GNU/Linux based operating system. The C files will come with their own makefiles[1], and will usually assume access to the `gcc` compiler. Some of the C programs are not hardware agnostic, and will require altering the makefiles in places. For some applications, we may be comparing performance against Intel or NAG libraries, and assessing performance on Intel, Arm, or even Nvidia hardware. As a minimum, we will require the user has

- The GNU scientific library (GLS) and the `gcc` compiler.

Further to this, some of the more advanced demonstrations may require any of the following:

- The Intel maths kernel library (MKL), the vector statistics library (VSL), and the `icc` compiler.
- The NAG library.
- The Arm `armclang` compiler.
- The Nvidia `nvcc` compiler.

Similarly, we will assume access to appropriate hardware from either Intel, Arm, or Nvidia where appropriate. We assume when compiling that the user has also setup the default paths to find the required header files and libraries by appropriately setting the environment variables `LIBRARY_PATH`, `LD_LIBRARY_PATH`, and `C_INCLUDE_PATH`.

We will try an keep the Python packages used to a minimum, although some of the notable packages which are used within our code include: `numpy`, `pandas`, `scipy`, `matplotlib`, `progressbar2`, `bisect`, `timeit`, etc.

To give an idea of the more relevant files contained, the user should expect a file structure similar to:

[1] We actually provide these as "`.sh`" files, which can be readily adapted or copied directly to the terminal.

```
approximate_random_variables/
|-- ... The files to read ...
|-- approximate_random_variables.pdf
|-- ... The Python approximations  ...
|-- approximate_gaussian_distribution.py
|-- approximate_gaussian_distribution_demonstrations.py
|-- approximate_non_central_chi_squared.py
|-- approximate_non_central_chi_squared_demonstrations.py
|-- non_central_chi_squared.m
|-- non_central_chi_squared.py
|-- ... The C approximations ...
|-- piecewise_constant_approximation.c
|-- piecewise_constant_approximation.h
|-- piecewise_constant_approximation_coefficients.h
|-- piecewise_polynomial_approximation.c
|-- piecewise_polynomial_approximation.h
|-- piecewise_polynomial_approximation_coefficients.h
|-- time_piecewise_constant_approximation.c
|-- time_piecewise_polynomial_approximation.c
|-- ... The makefiles ...
|-- make_time_piecewise_constant_approximations.sh
|-- make_time_piecewise_polynomial_approximations.sh
`-- ... etc. ...
```

2 Introduction

This report is intended to be a relatively light introduction to approximate random variables, designed primarily for an industry practitioner, but also applicable to scientists who want to achieve better code performance. In this introduction, we will give a gentle introduction to random numbers and approximate random variables, and discuss one of their major areas of application. We will give a light discussion of what's going on from a mathematical perspective, and provide code demonstrating how to implement these ideas. Most of the code is written using Python in an easy to follow manner (not particularly focused on performance), and where appropriate we will also provide some higher performance code written in C. We encourage readers to use either the code provided as it is, or adapt it to their own needs as appropriate. The code is largely demonstrative, so we make no guarantees about its performance or correctness. The code presented in the report will largely be stripped of much of its documentation and comments, and for a more descriptive version we recommend seeing the underlying source code files.

2.1 Where can I learn more?

For people who are more comfortable with maths and want to know more, there are several other sources of reference material available, of various degrees of technicality. From easiest to hardest there is:

- *High-performance low-precision vectorised arithmetic and its applications*, Oliver Sheridan-Methven, 2020, 2 pages. This is an extremely brief overview of my research with Mike Giles, giving a light overview of the core idea and some key findings, aimed at a very general reader with little technical knowledge.
- *High-performance low-precision vectorised arithmetic and its applications*, Oliver Sheridan-Methven, 2020, 8 pages. Another brief overview going into some more detail than the 2 page version. This is aimed at those with a bit more technical knowledge.
- *Analysis of nested multilevel Monte Carlo using approximate Normal random variables*, Mike Giles and Oliver Sheridan-Methven, 2020, approximately 25 pages. This is a journal article currently in preparation and due to be released very soon. This is aimed at a technical reader, and goes into depth with the main technical idea of our research and how to apply it, and is suitable for readers with a science degree.
- *Nested multilevel Monte Carlo methods and a modified Euler-Maruyama scheme utilising approximate Gaussian random variables suitable for vectorised hardware and low-precisions*, Oliver Sheridan-Methven, 2020, 250 pages. This is my PhD thesis, and goes into all the technical mathematical details surrounding these ideas. This is not a particularly light, easy, nor entertaining read, but is likely better suited as a manual for anyone worried about the technical consequences of switching to approximate random variables.

2.2 What are random variables?

Random numbers are part of everyday life, whether rolling dice, reading stock prices, or even sending encrypted messages; random numbers are a part of all of these. As random numbers occur so frequently in the world around us, they also play a central role in various branches of science and mathematics. Usually, when some "random quantity" appears as part of a scientific equation or mathematical formula, its given the more formal technical name of a being a *random variable*, which is the name we'll use going forward. Studying how random

variables behave is central to many branches of science, and has applications in: gambling, finance, computer simulations, weather prediction, security and encryption, artificial intelligence, etc., the list goes on and on.

Computers often need random numbers, either for running simulations, encrypting data, or possibly something else. However, computers don't behave randomly, and follow a very strict set of rules and instructions. So for a computer to spit out a sequence random numbers, it follows a very long and complicated set of steps, such that the numbers it spits out, while technically not random, appear random enough for most purposes. In fact, producing high quality random numbers can be a time consuming task for a computer.

The problem is made even harder when we consider that there are different varieties of random variables. In science, we say a given variety of random variables follows a *distribution*, which encompasses a particular style and flavour of random numbers. Some of these you are likely already familiar with and have names. Some common distributions include:

- The Bernoulli distribution (e.g. a coin toss).
- The discrete uniform distribution (e.g. the roll of a die).
- The continuous uniform distribution (e.g. an individual's height percentile).
- The Poisson distribution (e.g. lifetime of a lightbulb).
- The Gaussian distribution (e.g. average height of adults).

Computers by default are usually only able to produce the uniform distributions, and there are various methods to get the other more complicated distributions from these simpler ones. Amongst these methods, one which is particularly robust and versatile is known as *the inverse transform method*.

2.2.1 The inverse transform method

For some distribution of interest, let's suppose we know where the percentiles of the distribution are, for example: 50% of the values are below 0, 70% are below 1, 90% are below 2, 99% are below 3, 99.99% are below 4, etc. We can easily sample a uniform distribution, but how can we sample from this more interesting distribution? The way the inverse transform method works is surprisingly simple. Sample a uniform random variable, giving a number in 0–100, treat this as a percentile, and see what value of the distribution this percentile corresponds to. The number corresponding to this particular percentile is a random variable which exactly follows the desired distribution. There are only two things you need for this: access to uniform random numbers (which almost all computers have), and a formula for the percentiles. Repeating this for several uniformly distributed random variables produces random variables from the desired distribution, where a simple demonstration is shown in Figure 2.1.

The technical name for the expression which takes a percentile and determines what value of a random variable this corresponds to is *the inverse cumulative distribution function*, sometimes called the inverse CDF or the percentage point function. Most computers and programming languages have access to these functions for some of the most popular distributions.

2.3 What are approximate random variables?

So far we have painted a trouble free picture for producing random variables from whatever distribution interests us by using the inverse transform method. But suppose we have

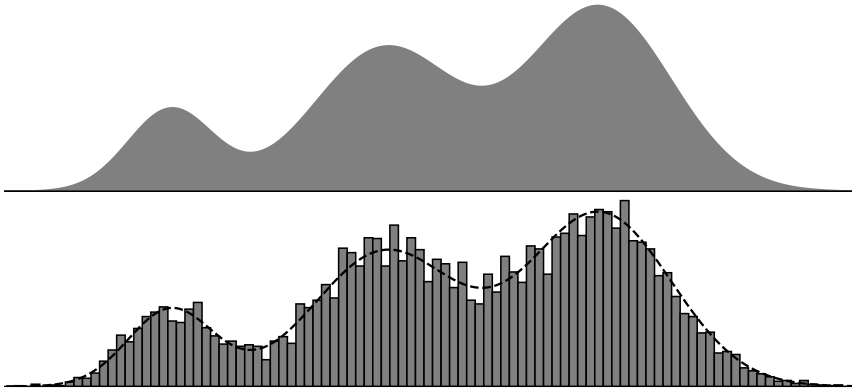
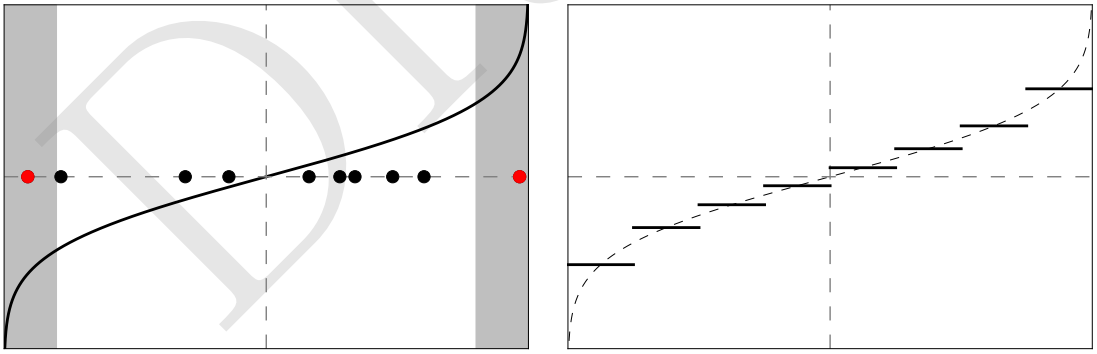


Figure 2.1 – The inverse transform method produces random variables following a desired pattern.

an application, such as a computer simulation, which requires lots of random variables from a distribution. As an example, it is not uncommon for financial simulations of stock markets to use anywhere from millions to trillions of random numbers to predict the value of various financial products. If this is the case, then we want to process these random numbers as quick as possible. Unfortunately, while the inverse transform method is versatile, if you do it exactly, it can be very slow and struggle to perform well on the latest hardware. This is where our research proposes the following idea: don't do it exactly, do it approximately.

To see where we make our approximations, let's take a very important and ubiquitous distribution known as the Gaussian distribution, which is sometimes called the Normal distribution or the bell curve. The Gaussian's inverse cumulative distribution function is plotted in Figure 2.2a, where we indicating several random positions the function might evaluate. Most of the inputs land near the centre of the function, where it's nearly a straight line and easy to evaluate. However, sometimes values land near the edges, in the more difficult shaded region where the function shoots off the edges of Figure 2.2a, and is much harder to evaluate. While these difficult scenarios are infrequent, the greater the degree of parallelisation, the more certain it is that such values are unavoidable. It is these types of edge values which contaminate an otherwise easy calculation, leading to a loss in performance.



(a) The Gaussian's inverse cumulative distribution function, and several possible random inputs (●). The more difficult regions are shaded, where some random inputs will unfortunately land (●)

(b) An example approximation of the exact function. The approximation is several flat lines of equal widths.

Figure 2.2 – The Gaussian's inverse cumulative distribution function and an approximation.

To remedy the difficult edge cases, our research proposes introducing an approximation to the exact inverse cumulative distribution function. The approximation is designed to tackle the edge cases as easily as the central ones, and being fast irrespective of

what's encountered. One example we proposed and studied was approximating the function by a series of flat lines, where an example is shown in Figure 2.2b.

In our experiments, we pitted various exact implementations (some proprietary), against our approximations. Although inexact, our approximations produced random variables in a fraction of the time it takes the exact functions. The random variables produced don't exactly follow the desired distribution, but instead something very similar. When the random variables are produced using an approximation, we call them *approximate random variables*. In our research, we study the error introduced from using these instead of the exact ones. Not only can we quantify how they affect the answer, we developed a mathematical approach to counteract their error. This means we can reap the benefits of the faster speed, without losing any accuracy, getting the best of both worlds.

2.4 When would I use approximate random variables?

In our research we investigated and analysed the consequences of using approximate random variables in the context of running computer simulations. This is extremely common in finance, weather forecasting, and other fields. As we mentioned, various other applications use random numbers and could likely benefit from using approximate random numbers, although there remains work to be done to investigate and analyse all of the remaining use-cases.

One particularly prolific scientific method is known as *Monte Carlo*. It is heavily used for financial simulations, and requires vast quantities of random numbers, usually from the Gaussian distribution. The Monte Carlo method can be described as: run lots of simulations of whatever is of interest, compute the average of what you see, and that's approximately the answer. The more simulations you run the more accurate your answer is.

For most people, computing averages is easy, but how does someone "simulate"? Running a simulation is asking a computer to do the following: given where something is now, and knowing how it behaves, guess where it will be in the future? When we say we know how the system behaves, it means we have an expression which accurately describes how it will evolve. This expression though is only accurate at predicting a tiny bit into the future. To predict far into the future, one method is to split this up into lots of smaller intermediate predictions. We then predict from one step to the next, one after another, until you're at the end.

The simplest method of simulating something step-by-step is known as the *Euler scheme*, and if the evolution is subject to random influences it's the *Euler-Maruyama scheme*. There are two factors influencing the evolution: one isn't random, and the other is (e.g. traders buying and selling stocks randomly influences their price). To simulate the random bit we need a random number, and the Euler-Maruyama scheme needs them from a Gaussian distribution. Some example Euler-Maruyama simulations are shown in Figure 2.3.

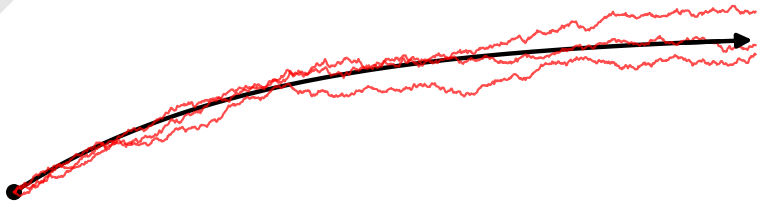


Figure 2.3 – A non-random process (—), and three Euler-Maruyama simulations (—).

The Euler-Maruyama scheme is simple, a robust problem solver, and there's not much that it struggles with, so why tamper with it? One shortcoming is that at each step it requires a random number from the Gaussian distribution. Unfortunately, this is usually the most expensive bit. To give the scheme the random numbers it needs quicker, we propose

swapping them for our approximate random numbers.

Modifying the Euler-Maruyama scheme to substitute exact random numbers with approximate ones largely alleviates the bottleneck. We can now run simulations much faster, improving the accuracy of any Monte Carlo estimate which uses the Euler-Maruyama scheme to produce the simulations. However, we cannot overlook that we compromised the quality of our simulations, and this will introducing some error into our answer. Is this significant though, or even noticeable? In our research, we've shown that the error introduced is limited in size by the error arising from approximating the Gaussian distribution. So to reduce the resulting error to an acceptable level, it's sufficient to just increase the fidelity of our approximation.

2.5 How can I get started using them?

All you need to get started using approximate random variables is an approximation and a problem to solve. In the code provided we will give some examples of approximating the Gaussian distribution, where we've taken some time and effort to ensure our C approximations are fast.

Before using approximate random variables in your own setting, we encourage you go through some of the demonstrations provided to give a better idea of how to incorporate and adapt them as necessary into your own problem. We also show how one of our approximations to the Gaussian distribution can be adapted to approximate another distribution (the non-central χ^2-distribution), and we expect our approximations can be used as templates for approximating several other distributions.

Approximate random variables are as their name suggests: approximate. Switching to approximate random variables is easy, and can gain you huge amounts of speed. Unfortunately, swapping the exact random variables for the approximate ones and doing nothing else, while easy to code up, will introduce some error. However, with a little extra effort, by utilising a technique known as *multilevel Monte Carlo*, you can counteract this error without compromising speed, and obtain the best of both worlds. This can be a little technical, and we refer a reader interested in the technicalities of the method to some of the additional material mentioned in Section 2.1.

3 The Gaussian distribution

The bulk of our approximations have been centred on approximating Gaussian random variables, as these are used in the Euler-Maruyama scheme (and several other schemes), and the Gaussian's analytic properties are sufficiently tractable that theoretical results could be proved for our approximations. As such, we frame our discussions of approximations around approximating the Gaussian distribution, for added utility and tractability.

3.1 Piecewise constant approximations

We mentioned earlier in Section 2.3 an approximation to the Gaussian's inverse cumulative distribution function which was comprised of a series of flat lines of equal width. The mathematical name for this approximation is a *piecewise constant function* which uses *equipartitioned intervals*, although we will just abbreviate this as the piecewise constant function. To construct the approximation, we first have to decide, how many intervals do we need, and what value should be use as the constant for each interval?

The number of intervals is the easy bit, so let's say we have N of them, where in most of our approximations we found $N = 1028$ to be sufficiently high to obtain a reasonable fidelity approximation. As for what value to use, a sensible choice is whatever you would expect to get from the exact function. The exact inverse cumulative distribution function for the Gaussian distribution is usually denoted Φ^{-1}. Denoting the start and end of a particular interval as a and b respectively, where $a < b$, the value used for a particular interval is

$$\frac{1}{b-a}\int_a^b \Phi^{-1}(x)\,\mathrm{d}x \equiv \frac{1}{b-a}\int_{\Phi^{-1}(a)}^{\Phi^{-1}(b)} \frac{z}{\sqrt{2\pi}}\exp\left(-\frac{z^2}{2}\right)\mathrm{d}z\,. \tag{3.1}$$

Both these integrals are equivalent to one another, where Φ is the Gaussian's cumulative distribution function (i.e. not the *inverse* CDF, but just the CDF). Both are easy for a computer to calculate, where the second is sometimes more computationally convenient for intervals touching the edge.

If we label the intervals $0, 1, 2, \ldots, N-1$, we can store these values in an list. For a uniform random input U in the range $0 \leq U < 1$, where 1 is not included, we can obtain the desired value by simply looking up the index corresponding to the relative size of U compared to $N-1$. Mapping U to the index is done by multiplying by N and only keeping the integer part, and hence computes $\lfloor UN \rfloor$, which in code can be done by something equivalent to `int(U * N)`.

In the file

`approximate_gaussian_distribution.py`

there is a function

`construct_piecewise_constant_approximation`

which produces a piecewise constant approximation to the Gaussian's inverse cumulative distribution function. A condensed version is shown in Code 3.1.

In the file

`approximate_gaussian_distribution_demonstrations.py`

there is the function

`plot_piecewise_constant_approximation_of_gaussian`

```
def expected_value_in_interval(func, a, b):
    """ Calculates the expected value of a function inside an interval. """
    return integrate(func, a, b) / (b - a)


def build_lookup_table(func, n_table_entries):
    """ Builds a lookup table. """
    interval_width = 1.0 / n_table_entries
    lookup_table = zeros(n_table_entries)
    for n in progressbar(range(n_table_entries)):
        a = n * interval_width
        b = a + interval_width
        lookup_table[n] = expected_value_in_interval(func, a, b)
    return lookup_table


def construct_piecewise_constant_approximation(func, n_intervals):
    """ Constructs a piecewise constant approximation. """
    lookup_table = build_lookup_table(func, n_intervals)

    def piecewise_constant_approximation(u):
        """ A piecewise constant approximation. """
        return lookup_table[array(n_intervals * u).astype(int)]

    return piecewise_constant_approximation
```

Code 3.1 – Constructing a piecewise constant approximation.

which demonstrates how to use this approximation, and how you can reproduce Figure 2.2b for yourself. A condensed version of the function is shown in Code 3.2, where we exclude the values 0 and 1 as the exact function struggles with these values (returning infinite values). The output from Code 3.2 is shown in Figure 3.1.

```
uniform_input = linspace(0, 1, 1000)[1:-1]  # We exclude the end points.
approximate_inverse_gaussian_cdf = \
    construct_piecewise_constant_approximation(
        inverse_gaussian_cdf,
        n_intervals=8)
plot(uniform_input, inverse_gaussian_cdf(uniform_input))
plot(uniform_input, approximate_inverse_gaussian_cdf(uniform_input))
```

Code 3.2 – Comparing the exact and approximate functions.

Although the Python implementation is not optimised to be very high speed (compared to our C version), it is still much faster than the exact routine (`norm.ppf` from `scipy.stats`). On my machine, the function

`piecewise_constant_approximation_of_gaussian_timing`

from the file

`approximate_gaussian_distribution_demonstrations.py`

gives the following estimates for the average time:

```
Average time for the approximate function: 3.23281e-08 s.
Average time for the exact function: 8.69456e-07 s.
```

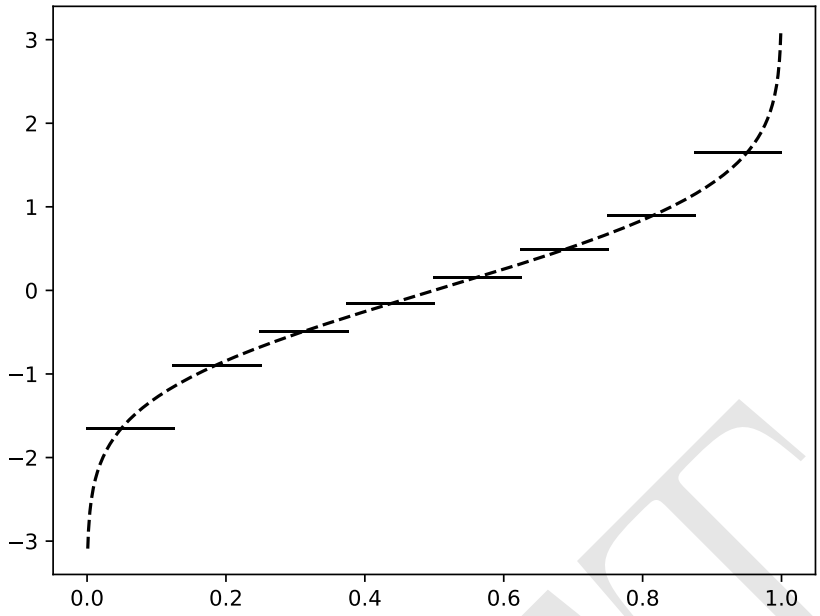


Figure 3.1 – The piecewise constant approximation produced by Code 3.2.

from which we can see the approximation is approximately 30 times faster than the exact function. Of course, both of these times are much slower than equivalent C code, but it is at least indicative that there is a lot of speed to be gained.

It is worth noting of course that the lookup table only needs to be computed once, and this can be done offline, so is not performance critical.

So how can a higher performance version be implemented in C? Easily. Assuming we've precomputed the values for the lookup table, then these can be stored in an array, and the piecewise constant approximation is effectively a single line of C code. A slightly expanded version of the function

`piecewise_constant_approximation`

from the file

`piecewise_constant_approximation.c`

is shown in Code 3.3. We can see the implementation condenses down to a single simple line of code. The reason this is so fast is because the lookup table is small enough that it will easily fit within the L2-cache. Furthermore the lookup is unconditional, and so well suited to vectorisation, and hence we explicitly mark the process as suitable for vectorisation using `#pragma omp simd`.

To compare this against the equivalent GSL function (`gsl_cdf_ugaussian_Pinv`), there is a simple series of compiler instructions in the file

`make_time_piecewise_constant_approximations.sh`

which resembles

```
gcc -I. -c piecewise_constant_approximation.c
gcc -I. -O0 -c time_piecewise_constant_approximations.c
gcc -I. -o time_piecewise_constant_approximations [object_files] -lgsl
```

and produces the executable

```c
#include <omp.h>

#define LOOKUP_TABLE_SIZE 1024
const double lookup_table[LOOKUP_TABLE_SIZE] = {-3.37,-2.98,...,2.98,3.37};

void piecewise_constant_approximation(
    unsigned int n_samples,
    const double *restrict input,
    double *restrict output)
{
    #pragma omp simd
    for (unsigned int n = 0; n < n_samples; n++)
    {
        output[n] = lookup_table[(unsigned int) (LOOKUP_TABLE_SIZE * input[n])];
    }
}
```

Code 3.3 – Constructing a piecewise constant approximation.

`time_piecewise_constant_approximations`

On my machine, which is not vector capable, and running with minimal optimisations, we obtain

```
Average time for the approximate function: 2.28712e-09 s.
Average time for the exact (GSL) function: 2.18994e-08 s.
```

Similar to before the approximation is 10 times faster than the exact function from GSL, without any optimisation. Furthermore, both functions are running 10 times faster in C than in Python, which is not too surprising. If we compile

`piecewise_constant_approximation.c`

under `-O3` then this changes to

```
Average time for the approximate function: 9.3289e-10 s.
Average time for the exact (GSL) function: 2.18471e-08 s.
```

widening the speed increase to nearly 25 times. With vector capable hardware and some fine tuning with OpenMP, the method can be very competitive.

Moving this onto some Intel Skylake AVX512 hardware and compiling using

```
icc -I. -std=c11 -O3 -simd -p -mkl -qopenmp -xCORE-AVX512 \
    -march=skylake-avx512 -qopt-zmm-usage=high -c \
    piecewise_constant_approximation.c
icc -I. -mkl -O0 -c time_piecewise_constant_approximation.c \
    -D__PURE_INTEL_C99_HEADERS__
icc -I. -o time_piecewise_constant_approximation [object_files] \
    -lgsl -lgslcblas
```

our simple implementation does okay, achieving

```
Average time for the approximate function: 8.9733e-10 s.
Average time for the exact (GSL) function: 2.2618e-08 s.
```

and when this is adapted to use `-DUSE_VECTOR_BATCHES` (see the makefile and source code for an explanation) this improves to

```
Average time for the approximate function: 7.5127e-10 s.
Average time for the exact (GSL) function: 2.26445e-08 s.
```

which corresponds to about 1.7 clock-cycles on average (the machine runs at 2.3 GHz), in keeping with the findings from previous work.

3.2 Piecewise polynomial approximations

There are a few natural ways to extend the piecewise constant approximation, and the two most obvious are: increase the order of the polynomial, and change the spacing of the intervals so they're denser around the edges.

Starting with increasing the order of the polynomial, the next improvement from a piecewise constant is a piecewise linear function. So the next question is which linear function best approximates the function in a given interval. There are various measures of the error, including the worse case error, or the average case error. For Monte Carlo applications, we used a variant called *the mean squared error*, known mathematically as *the L^2-error* (some definitions use the *root mean squared error*). One of the advantages of the L^2-error, is that it is relatively easy to determine the optimal parameters. For a function f, let's approximate this using a function $\tilde{f}$, where we write it as an m-th order polynomial

$$f(u) \approx \tilde{f}(u) = c_0 + c_1 u + c_2 u^2 + \cdots + c_m u^m. \tag{3.2}$$

So how can we determine the optimal coefficients? The error can be expressed as

$$\int_a^b \left(f(u) - \sum_{i=0}^{m} c_i u^i \right)^2 \mathrm{d}u\,. \tag{3.3}$$

We can minimise this by differentiating with respect to each coefficient c_i and finding the values which make the derivative zero. Doing this for each of the coefficients, we obtain the set of $m+1$ simultaneous equations

$$\sum_{i=0}^{m} c_i \left(\frac{b^{i+j+1} - a^{i+j+1}}{i+j+1} \right) = \int_a^b u^j f(u)\,\mathrm{d}u \tag{3.4}$$

for $j = 0, 1, 2, \ldots, m$, which is equivalent to an equation of the form

$$A\boldsymbol{x} = \boldsymbol{b}. \tag{3.5}$$

Luckily, solving simultaneous equations of this form is easy for computers, and so the coefficients are easily determined, where

$$\boldsymbol{b}_i = \int_a^b u^i f(u)\,\mathrm{d}u \tag{3.6}$$

and

$$A_{i,j} = \frac{b^{i+j+1} - a^{i+j+1}}{i+j+1}, \tag{3.7}$$

and we are solving for values of $\boldsymbol{x}$, where

$$\boldsymbol{x}_i = c_i. \tag{3.8}$$

The improvement we also want to make is to make the intervals denser near the edges of the function, as these are the trickiest bits which require the most effort from an

approximation. One way to achieve this is to have the intervals geometrically decaying in size near the edge, and one decay rate which is particularly well suited for computer calculations is in powers of two (as computers work in binary). When the intervals are decaying powers of two, we give these the special name of *dyadic intervals*.

All inverse cumulative distribution functions are defined for valued between 0 and 1, and so we only need dyadic intervals in this range. If we again consider N intervals in this range, each corresponding to an index, they are as shown in Table 3.1.

Dyadic interval	Index
$[\frac{1}{2}, 1)$	0
$[\frac{1}{4}, \frac{1}{2})$	1
$[\frac{1}{8}, \frac{1}{4})$	2
$\vdots$	$\vdots$
$[\frac{1}{2^{n+1}}, \frac{1}{2^n})$	n
$\vdots$	$\vdots$
$[0, \frac{1}{2^{N-1}})$	$N-1$

Table 3.1 – The indices for various dyadic intervals.

These intervals are dense near zero, but they are not dense near 1. Is this a problem? Not really. For the Gaussian's inverse cumulative distribution function, this is rotationally symmetric about $\frac{1}{2}$ such that $\Phi^{-1}(U) \equiv -\Phi^{-1}(1-U)$, and so values near one are easily computed by the equivalent value reflected about $\frac{1}{2}$. Notice that if we reflect about $\frac{1}{2}$, then there will be one interval, the 0 index interval, which will contain just the one value $\frac{1}{2}$. While this is a slight computational redundancy, ensuring this value has an interval in which it belongs ensures any approximation won't break if it is given the perfectly valid value of $\frac{1}{2}$.

Taking an order 1 polynomial, corresponding to a piecewise linear function, in the file

`approximate_gaussian_distribution.py`

the function

`construct_symmetric_piecewise_polynomial_approximation`

provides a relatively simple demonstration of how to build and implement such a function. A small demonstration of how to use this can be found in function

`plot_piecewise_polynomial_approximation_of_gaussian`

from

`approximate_gaussian_distribution_demonstrations.py`

where the output is shown in Figure 3.2. If the number of intervals in increased to a modest amount (e.g. 8 or 16) then the approximation achieves a very high fidelity very quickly. In Figure 3.2 we only use a very small number of intervals, else the approximation and the exact function would be indistinguishable.

It is worth noting that the Python implementation provided in

`approximate_gaussian_distribution_demonstrations.py`

is only demonstrative, but is not particularly high performance, where the function

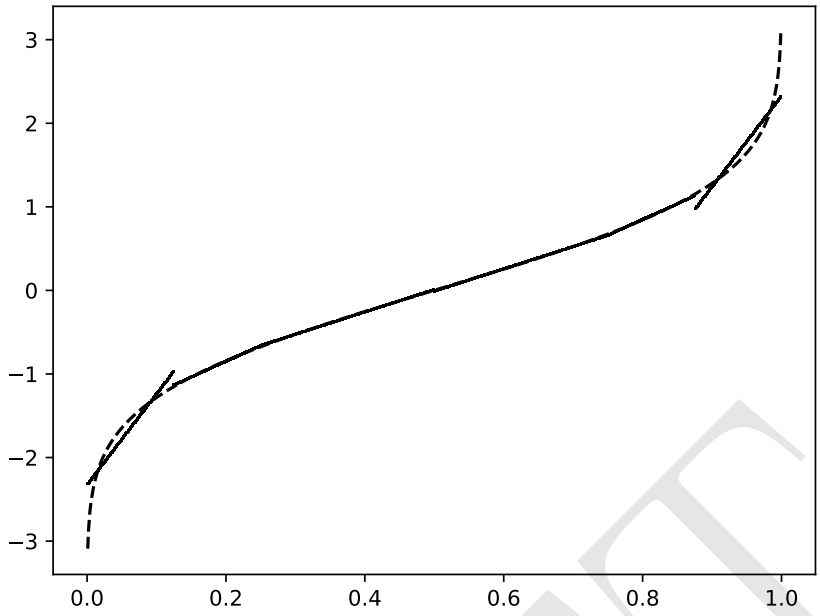


Figure 3.2 – A piecewise linear approximation using dyadic intervals.

`piecewise_constant_polynomial_of_gaussian_timing`

produces

```
Average time for the exact function: 8.93741e-08 s.
Average time for the approximate function: 1.22783e-06 s.
```

showing that our Python implementation is approximately 15 times slower than the exact function! (It will be our C implementations which will be fast).

To get an idea of how accurate these approximations are for different numbers of intervals and varying polynomial orders, the function

`plot_error_of_piecewise_polynomial_approximation_of_gaussian`

computes the RMSE for various configurations, with the results shown in Figure 3.3.

We are not particularly discouraged by this Python result, as this approximation can be extremely performant on vector hardware, assuming a sufficiently wide vector length. The reason for this is simple, let us suppose we are working in 32-bit single-precision, and we are using the reasonably high fidelity 16-interval piecewise cubic approximation. If we store the possible coefficients for a given term in a list, then this will only require 512-bits ($32 \times 16 = 512$), which is the vector width on a lot of modern hardware, such as Intel's AVX512 Skylake hardware or newer. This means that when we lookup the coefficients for a given term, instead of querying the cache, we can query a vector register, where we will only need 4 vector registers to hold all the coefficients we need. Additionally, if we are using dyadic intervals, then we can easily determine the index by reading the exponent bits of the floating-point number using bit-wise operations (much quicker than how we did this in Python). This all means we never even need to look in the cache, everything we require is held in vector registers, and we only need a few simple arithmetic operations (which can even use FMAs) and bit manipulations. This means that a C implementation can achieve very high speeds.

In the file

`piecewise_polynomial_approximation.c`

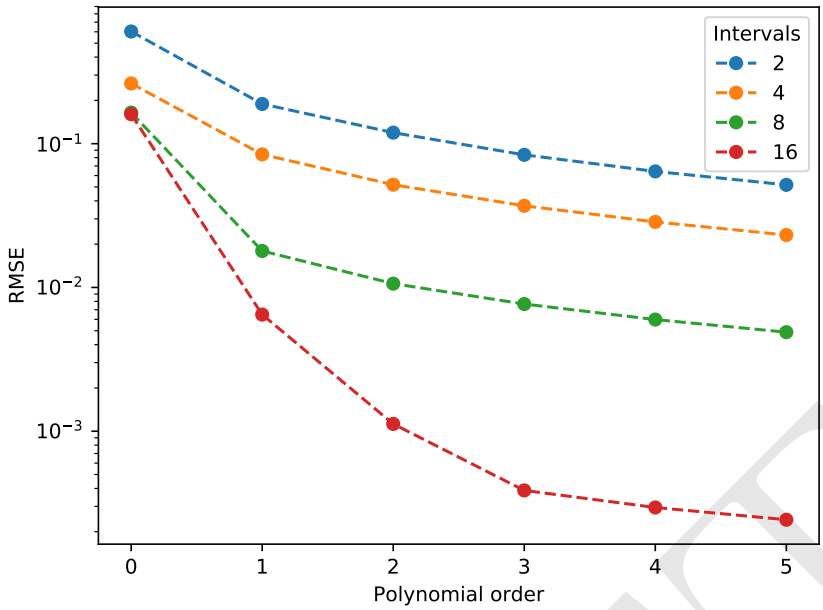


Figure 3.3 – Error of piecewise polynomial approximations.

we provide several implementations of a piecewise polynomial approximation using the function

`piecewise_polynomial_approximation`

where there several implementations targeting different architectures. Different implementations are compiled depending on the value of various macro definitions passed to the compiler (using "`-D<macro-name>`"), as described in Table 3.2. To use a cubic approximation implemented using OpenMP you would compile with the flags

```
-DUSE_OPENMP_SIMD_APPROX -DPOLYNOMIAL_ORDER=3
```

Macro	Target	Vectors	`POLYNOMIAL_ORDER`
`USE_OPENMP_SIMD_APPROX`	Any	Any	1, 3
`USE_INTEL_INTRINSICS_APPROX_CUBIC`	Intel	AVX512	3
`USE_INTEL_INTRINSICS_APPROX_LINEAR`	Intel	AVX512	1
`USE_ARM_SVE_INLINE_APPROX_CUBIC`	Arm	SVE 512-bits	3

Table 3.2 – Piecewise polynomial approximation implementations.

All of the implementations are written assuming IEEE 32-bit single-precision floats, require 16 dyadic intervals, and assume vector widths of 512-bits (even for SVE). Our set of implementations is not complete nor exhaustive, and there is likely some room for more fine tuning and optimisation. Nonetheless, some can be readily copied, while we hope that others serve as useful templates for constructing other approximations.

Some condensed versions of these implementations (with most of the comments removed for brevity) are shown in Codes 3.4 to 3.6, demonstrating an OpenMP SIMD implementation, an Arm SVE implementation, and an Intel intrinsics implementation. An

437 explanation of what is going on in each of these implementations is provided in Algorithm 3.1.

Input: Floating-point uniform random number $U \in [0, 1)$.
Output: Floating-point piecewise polynomial approximation.
1 Form predicate based on whether $U > \frac{1}{2}$.
2 Reflect about $\frac{1}{2}$ to obtain $U \in [0, \frac{1}{2}]$.
3 Alias U as an unsigned integer.
4 Read the exponent bits using bitwise-AND.
5 Right shift away the mantissa's bits.
438 6 Obtain an array index by correcting for exponent bias.
7 Cap the array index to avoid overflow.
8 Read the polynomial coefficients using the array index.
9 Re-interpret U as a float.
10 Form the polynomial approximation.
11 Correct sign of approximation based on predicate.

Algorithm 3.1: Piecewise polynomial approximation using dyadic intervals.

```c
#include "piecewise_polynomial_approximation.h"
#include "piecewise_polynomial_approximation_coefficients.h"

// For IEEE 754
#define N_MANTISSA_32 23
#define FLOAT32_EXPONENT_BIAS 127
#define FLOAT32_EXPONENT_BIAS_TABLE_OFFSET (FLOAT32_EXPONENT_BIAS - 1)
#define TABLE_MAX_INDEX (TABLE_SIZE - 1) // Zero indexing...
#define FLOAT32_AS_UINT32(x) (*((uint32 *) &x))  // Type-punning is performed.

#pragma omp declare simd
static inline float32 polynomial_approximation(float32 u, uint32 b);

#pragma omp declare simd
static inline uint32 get_table_index_from_float_format(float32 u);

static inline float32 polynomial_approximation(float32 u, uint32 b)
{
    #if (POLYNOMIAL_ORDER == 3)
    float32 z, z_even, z_odd;
    float32 x = u * u;
    z_even = poly_coef_0[b] + poly_coef_2[b] * x;
    z_odd = poly_coef_1[b] + poly_coef_3[b] * x;
    z = z_even + z_odd * u;
    return z;
    #elif (POLYNOMIAL_ORDER == 1)
    float32 z =  poly_coef_0[b] + poly_coef_1[b] * u;
    return z;
    #else
    #endif
}

static inline uint32 get_table_index_from_float_format(float32 u)
{
    uint32 b;
    b = FLOAT32_AS_UINT32(u) >> N_MANTISSA_32; // Keeping only the exponent.
    b = FLOAT32_EXPONENT_BIAS_TABLE_OFFSET - b; // Getting the table index.
    b = b > TABLE_MAX_INDEX ? TABLE_MAX_INDEX : b;  // Avoiding overflow.
    return b;
}

#ifdef USE_OPENMP_SIMD_APPROX
void piecewise_polynomial_approximation(
    unsigned int n_samples,
    const float32 *restrict input,
    float32 *restrict output)
{
    #pragma omp simd
    for (unsigned int i = 0; i < n_samples; i++)
    {
        float32 u, z;
        u = input[i];
        bool predicate = u < 0.5f;
        u = predicate ? u : 1.0f - u;
        uint32 b = get_table_index_from_float_format(u);
        z = polynomial_approximation(u, b);
        z = predicate ? z : -z;
        output[i] = z;
    }
}
#endif
```

Code 3.4 – Piecewise polynomial approximation implementations using OpenMP.

```
// This is not VLA, but assumes vector lengths sufficient to hold the arrays.
#ifdef USE_ARM_SVE_INLINE_APPROX_CUBIC
void piecewise_polynomial_approximation(unsigned int n_samples,
    const float32 *restrict input, float32 *restrict output)
{
    unsigned int i;  // A dummy variable.
    asm ("" 
        "\tcbz %[n_samples], 2f                                    "
        "\n\tmov %[i], xzr                                         "
        "\n\tfmov    z0.s, #5.00000000000000000e-01                "
        "\n\tptrue   p0.s                                          "
        "\n\tfmov    z1.s, #1.00000000000000000e+00                "
        "\n\tld1w    {z2.s}, p0/z, [%[poly_coef_0], %[i], lsl #2]  "
        "\n\tld1w    {z3.s}, p0/z, [%[poly_coef_1], %[i], lsl #2]  "
        "\n\tld1w    {z4.s}, p0/z, [%[poly_coef_2], %[i], lsl #2]  "
        "\n\tld1w    {z5.s}, p0/z, [%[poly_coef_3], %[i], lsl #2]  "
        "\n\tmov z6.s, #126                                        "
        "\n\tmov z7.s, %[table_max_index]                          "
        "\n\twhilelo p1.s, xzr, %[n_samples]                      "
        "\n1:                                                      "
        "\n\tld1w    {z8.s}, p1/z, [%[input], %[i], lsl #2]        "
        "\n\tfsub    z9.s, z1.s, z8.s                              "
        "\n\tfcmgt   p2.s, p0/z, z0.s, z8.s                        "
        "\n\tsel z8.s, p2, z8.s, z9.s                              "
        "\n\tlsr z9.s, z8.s, #23                                   "
        "\n\tsub z9.s, z6.s, z9.s                                 "
        "\n\tcmplo   p3.s, p0/z, z9.s, %[table_max_index]         "
        "\n\tsel z9.s, p3, z9.s, z7.s                              "
        "\n\tfmul    z10.s, z8.s, z8.s                             "
        "\n\ttbl z11.s, {z2.s}, z9.s                               "
        "\n\ttbl z12.s, {z4.s}, z9.s                               "
        "\n\tfmad    z12.s, p0/m, z10.s, z11.s                     "
        "\n\ttbl z11.s, {z3.s}, z9.s                               "
        "\n\ttbl z13.s, {z5.s}, z9.s                               "
        "\n\tfmad    z13.s, p0/m, z10.s, z11.s                     "
        "\n\tfmad    z13.s, p0/m, z8.s, z12.s                      "
        "\n\tfneg    z10.s, p0/m, z13.s                            "
        "\n\tsel z10.s, p2, z13.s, z10.s                           "
        "\n\tst1w    {z10.s}, p1, [%[output], %[i], lsl #2]        "
        "\n\tincw    %[i]                                          "
        "\n\twhilelo p1.s, %[i], %[n_samples]                      "
        "\n\tb.any 1b                                              "
        "\n2:                                                      "
        "\n\t                                                      "
        : [i] "=&r" (i)
        : "[i]" (0), [input] "r" (input), [output] "r" (output),
        [n_samples] "r" (n_samples), [poly_coef_0] "r" (poly_coef_0),
        [poly_coef_1] "r" (poly_coef_1), [poly_coef_2] "r" (poly_coef_2),
        [poly_coef_3] "r" (poly_coef_3), [table_max_index] "i" (TABLE_MAX_INDEX)
        : "memory", "cc", "p0", "p1", "p2", "p3", "z0", "z1", "z2", "z3", "z4",
        "z5", "z6", "z7", "z8", "z9", "z10", "z11", "z12", "z13"
        );
}
#endif
```

Code 3.5 – Piecewise polynomial approximation implementations using Arm SVE inline assembly.

```
// Using AVX512 intrinsics.
#ifdef USE_INTEL_INTRINSICS_APPROX_CUBIC
void piecewise_polynomial_approximation(unsigned int n_samples,
    const float32 *restrict input, float32 *restrict output)
{
    unsigned int n_iterations = n_samples / VECTOR_LENGTH;
    /* Loading in the constants. */
    __m512 half_float = _mm512_set1_ps(0.5f);
    __m512 one_float = _mm512_set1_ps(1.0f);
    __m512 minus_zero_float = _mm512_set1_ps(-0.0f);  // For negation.
    __m512i exponent_bias_table_offset =
        _mm512_set1_epi32(FLOAT32_EXPONENT_BIAS_TABLE_OFFSET);
    __m512i table_max_index = _mm512_set1_epi32(TABLE_MAX_INDEX);
    __m512 coef_0 = _mm512_load_ps(poly_coef_0);
    __m512 coef_1 = _mm512_load_ps(poly_coef_1);
    __m512 coef_2 = _mm512_load_ps(poly_coef_2);
    __m512 coef_3 = _mm512_load_ps(poly_coef_3);
    __m512 c_0, c_1, c_2, c_3;
    __m512i c;  // A useful temporary variable.
    __m512 u, z, z_even, z_odd, u_squared;
    __m512i b;
    for (int i = 0, offset = 0; i < n_iterations; i++, offset += VECTOR_LENGTH)
    {
        // Loading in the data.
        u = _mm512_loadu_ps(input + offset);
        // Forming the predicate.
        __mmask16 predicate = _mm512_cmplt_ps_mask(half_float, u);
        // Transforming the input to [0, 0.5].
        u = _mm512_mask_sub_ps(u, predicate, one_float, u);
        b = _mm512_castps_si512(u);
        // Getting the indices.
        b = _mm512_srli_epi32(b, N_MANTISSA_32);
        b = _mm512_sub_epi32(exponent_bias_table_offset, b);
        b = _mm512_min_epi32(b, table_max_index);
        // Obtaining the relevant coefficients.
        c = _mm512_castps_si512(coef_0);
        c = _mm512_permutexvar_epi32(b, c);
        c_0 = _mm512_castsi512_ps(c);
        c = _mm512_castps_si512(coef_1);
        c = _mm512_permutexvar_epi32(b, c);
        c_1 = _mm512_castsi512_ps(c);
        c = _mm512_castps_si512(coef_2);
        c = _mm512_permutexvar_epi32(b, c);
        c_2 = _mm512_castsi512_ps(c);
        c = _mm512_castps_si512(coef_3);
        c = _mm512_permutexvar_epi32(b, c);
        c_3 = _mm512_castsi512_ps(c);
        // Building the polynomial.
        u_squared = _mm512_mul_ps(u, u);
        z_even = _mm512_fmadd_ps(c_2, u_squared, c_0);
        z_odd = _mm512_fmadd_ps(c_3, u_squared, c_1);
        z = _mm512_fmadd_ps(z_odd, u, z_even);
        z = _mm512_mask_xor_ps(z, predicate, minus_zero_float, z); // Negation.
        // Writing the result.
        _mm512_storeu_ps(output + offset, z);
    }
}
#endif
```

Code 3.6 – Piecewise polynomial approximation implementations using Intel intrinsics.

On my personal scalar machine, compiling the OpenMP SIMD version using

```
make_time_piecewise_polynomial_approximations.sh
```

which for a piecewise cubic approximation looks like

```
gcc -I. -fopenmp -Ofast -DUSE_OPENMP_SIMD_APPROX -DPOLYNOMIAL_ORDER=3 \
    -c piecewise_polynomial_approximation.c
gcc -I. -O0 -c time_piecewise_polynomial_approximation.c
gcc -I. -o time_piecewise_polynomial_approximation [object_files] -lgsl
```

I obtain

```
Average time for the approximate function: 5.5168e-09 s.
Average time for the exact (GSL) function: 2.2597e-08 s.
```

GSL is as expected, and the SIMD cubic is slightly slower than the piecewise constant. Given the scalar hardware, this is not too surprising.

Moving onto an Intel machine[1] we can inspect the performance of the C implementations using AVX512 intrinsics. Building a cubic approximation using

```
icc -I.  -DUSE_INTEL_INTRINSICS_APPROX_CUBIC -DPOLYNOMIAL_ORDER=3 \
    -std=c11 -O3 -simd -p -mkl -qopenmp -xCORE-AVX512 \
    -march=skylake-avx512 -qopt-zmm-usage=high -c \
    piecewise_polynomial_approximation.c
icc -I. -mkl -O0 -c time_piecewise_polynomial_approximation.c \
    -D__PURE_INTEL_C99_HEADERS__
icc -I. -o time_piecewise_polynomial_approximation \
    [object_files] -lgsl -lgslcblas
```

we obtain the executable

```
time_piecewise_polynomial_approximation
```

where for the piecewise linear function we instead use the flags

```
-DUSE_INTEL_INTRINSICS_APPROX_LINEAR -DPOLYNOMIAL_ORDER=1
```

the cubic obtains

```
Average time for the approximate function: 3.93652e-10 s.
Average time for the exact (GSL) function: 2.70001e-08 s.
```

and the linear obtains

```
Average time for the approximate function: 2.49551e-10 s.
Average time for the exact (GSL) function: 2.6856e-08 s.
```

We can see that at a frequency of 2.3 GHz these correspond to average times of 0.9 and 0.6 clock-cycles per input, which is in keeping with our previous research, and is very fast.

These C implementations are designed to be fast and high performance. Nonetheless, we are sure with some more profiling, trying some other compiler flags, and perhaps a bit more

[1] `lscpu` gives `Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz`.

effort they may yet yield a bit more performance. It is worth noting that our approximations are much faster than the exact functions from Intel's MKL, which can be included in our comparisons using the flag

```
-DCOMPARE_AGAINST_MKL
```

when compiling

```
time_piecewise_polynomial_approximation.c
```

and gives the results

```
Average time for the approximate function: 2.13887e-10 s.
Average time for the exact (GSL) function: 2.40896e-08 s.
Average time for the exact (Intel HA) function: 2.93076e-09 s.
Average time for the exact (Intel LA) function: 2.16174e-09 s.
```

where not only is our approximation averaging around 0.5 clock-cycles, it is around 10 times faster than Intel.

At the time of writing, SVE capable hardware is in the preliminary stages of release, and we don't currently have access to a machine. However, to enable SVE compilation we recommend exploring the follow flags for `armclang`:

```
-armpl -fsimdmath -fvectorize -march=armv8.4-a+sve -mcpu=thunderx2t99
```

amongst several others.

4 The non-central χ^2-distribution

The non-central χ^2-distribution $\chi^2_\nu(\lambda)$ with ν degrees of freedom and non-centrality parameter λ is another common distribution, where $\nu > 0$ and $\lambda \geq 0$. One application of note where this occurs is the Cox-Ingersoll-Ross (CIR) model for interest rates. The inverse cumulative distribution function for this distribution, which we call $C_\nu^{-1}(\cdot;\lambda)$ is very expensive to calculate, where the results from a relative comparison to the Gaussian are shown in Table 4.1. Although there are two parameters, ν and λ, for a simulation with a constant time increment ν is a constant, and so $C_\nu^{-1}(\cdot;\lambda)$ can be regarded as only being parametrised by λ. Thus we need to build a parametrised approximation.

λ \ ν	1	5	10	50	100
1	37	36	40	54	73
5	40	46	48	62	85
10	54	56	63	69	97
50	101	103	103	144	143
100	191	190	192	189	185
200	243	246	240	233	221
500	465	474	465	446	416
1000	459	458	455	471	474

(a) Using Python and `ncx2.ppf` from Scipy's `scipy.stats`.

λ \ ν	1	5	10	50	100
1	168	214	259	456	294
5	651	782	840	1510	2046
10	935	1086	1050	1838	2496
50	3000	2969	2562	4118	5333
100	4929	3461	5039	6046	6299
200	9456	9603	10129	11524	12766
500	22691	22713	22702	23328	26273
1000	45872	43968	43807	44563	46780

(b) Using MATLAB and `ncx2inv` from MATLAB's statistics and machine learning toolbox.

Table 4.1 – The ratio of average computational times for evaluating the inverse non-central χ^2 cumulative distribution function against the inverse Gaussian cumulative distribution function for several values of the non-centrality parameter λ and degrees of freedom ν.

You can generate the results from Table 4.1 yourself in MATLAB and Python. For MATLAB run the file

`non_central_chi_squared.m`

to obtain

```
            nu
         ------------------------------
lambda   |1      5       10       50
-------------------------------------
1        | 182   213     267      431
5        | 136   675     801      1382
10       | 927   879     951      1756
50       | 2815  2009    2690     3946
100      | 4951  5179    4781     6700
200      | 9639  8702    9539     11054
```

and for Python run the file

`non_central_chi_squared.py`

to obtain

nu	1	5	10	50
lambda				
1	33	40	42	57
5	43	50	50	63
10	55	58	66	74
50	105	108	108	151
100	196	172	196	203
200	248	252	250	245

It is not particularly wise to try and directly approximate the inverse cumulative distribution function $C_\nu^{-1}(\cdot;\lambda)$, as it is not particularly well scaled for various values of its parameters. Instead, if we define the function $P_x(\cdot;y)$ as

$$P_x(U;y) \coloneqq \sqrt{\frac{x}{4y}}\left(\frac{y}{x}C_x^{-1}\left(U;\frac{(1-y)x}{y}\right)-1\right) \tag{4.1}$$

for $x>0$ and $0<y<1$, then we have

$$C_\nu^{-1}(U;\lambda) = \lambda+\nu+2\sqrt{\lambda+\nu}P_\nu\left(U;\frac{\nu}{\lambda+\nu}\right). \tag{4.2}$$

The reason this is preferable is because $P_x(\cdot;y)$ is much better scaled for the range of possible parameters, where it has the limits

$$P_x(U;y) \xrightarrow{y\to 0} \Phi^{-1}(U) \tag{4.3}$$

and

$$P_x(U;y) \xrightarrow{y\to 1} \frac{C_x^{-1}(U)}{2\sqrt{x}}-\frac{\sqrt{x}}{2}, \tag{4.4}$$

where C_ν^{-1}, without the parameter λ, is the inverse cumulative distribution function of the central χ^2-distribution. We can see its convergence to these two limits in Figure 4.1.

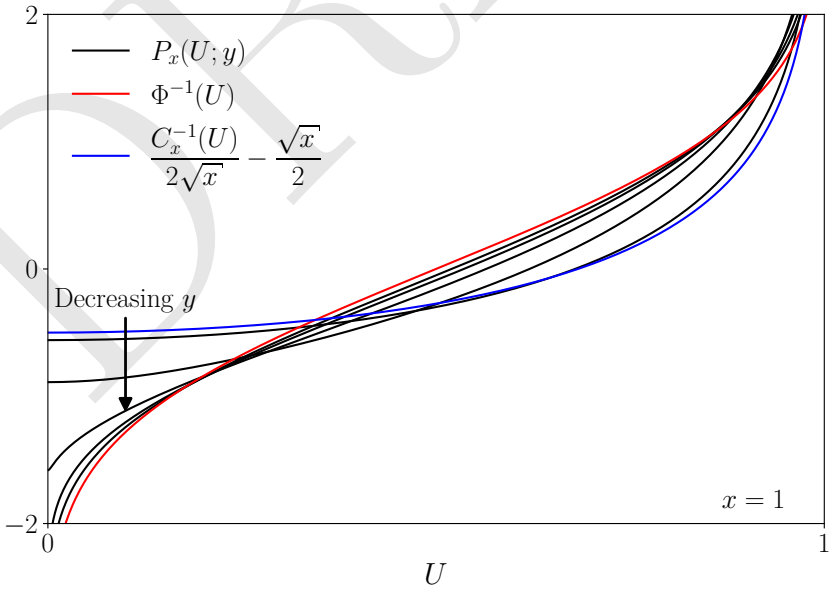


Figure 4.1 – The limiting convergence of the non-central χ^2-distribution.

We notice that $P_x(\cdot;y)$ is not symmetric, but given it looks so similar to the Gaussian, we approximate it similarly. We again use a piecewise polynomial approximation using dyadic intervals. However, as it is not rotationally symmetric, we use one approximation between 0 and $\frac{1}{2}$, and a second between $\frac{1}{2}$ and 1.

As we have a parametrised distribution for $0 < y < 1$, we construct several approximations for a discrete set of different and equally spaced values of $\sqrt{y}$, where the value for some specific y is obtained from linear interpolation in the value of $\sqrt{y}$.

A polynomial approximation to the non-central χ^2-distribution's inverse cumulative distribution function, produced in the way we've highlighted, is implemented in

`approximate_non_central_chi_squared.py`

and its use is demonstrated in the file

`approximate_non_central_chi_squared_demonstrations.py`

with the function

`plot_non_central_chi_squared_polynomial_approximation`

which produces the approximations shown in Figure 4.2.

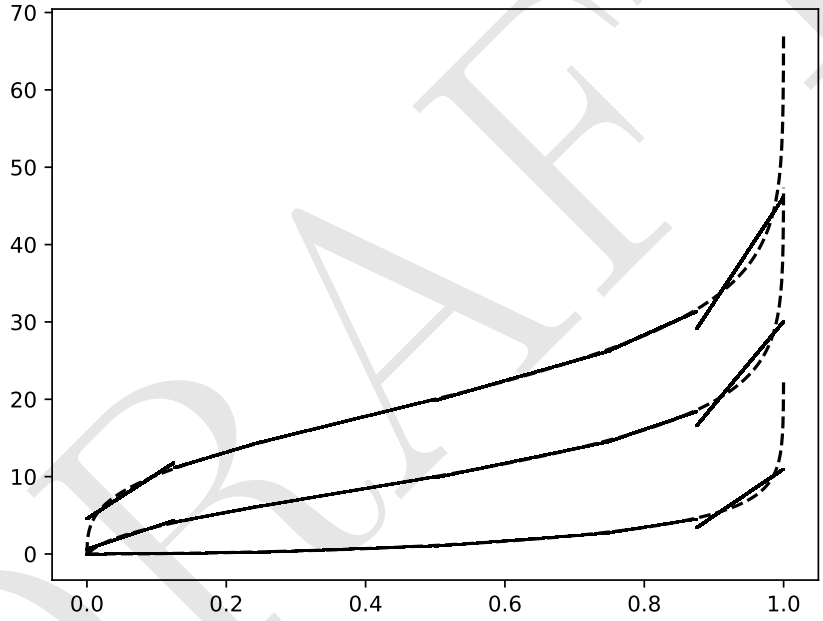


Figure 4.2 – Several piecewise polynomial approximations using dyadic intervals.

Similar to before, we can compare the relative time taken between the exact function and our polynomial approximation using the function

`non_central_chi_squared_polynomial_approximation_timing`

which outputs

```
nu        1    5    10   50
lambda
1        1.3  1.5  1.5  2.0
5        1.5  1.7  1.7  2.2
10       2.0  2.1  2.4  2.7
50       3.7  3.9  3.9  5.2
```

and a more comprehensive collection of results are presented in Table 4.2. We can see from this that our Python implementation is usually slightly faster than the exact function when ν and

λ are both $\mathcal{O}(1)$. However, as λ increases such that $\lambda \gg 1$ we can see that even our modest Python implementation is much faster.

λ	ν				
	1	5	10	50	100
1	1.2	1.5	1.5	2.1	2.7
5	1.2	1.7	1.8	2.3	3.2
10	1.2	2.2	2.3	2.6	3.7
50	4.0	3.9	3.8	5.4	5.4
100	6.7	7.3	7.4	7.2	6.8
200	8.9	9.3	8.8	8.7	8.0
500	16.8	16.8	16.3	16.5	16.0
1000	17.4	17.4	15.9	16.6	17.8

Table 4.2 – The relative time required in Python using `ncx2.ppf` from Scipy's `scipy.stats` against our polynomial approximation.

5 Monte Carlo simulations using the Euler-Maruyama scheme

We mentioned in Section 2.4 that random variables (and hence also approximate random variables) play a central role in Monte Carlo simulations, and we will elaborate a bit more on this here. There are many classical problems which can be described by what's called a *first order ordinary differential equation*, which takes the form

$$\frac{\mathrm{d}x}{\mathrm{d}t} = a(t, x), \tag{5.1}$$

and is often written as

$$\mathrm{d}x = a(t, x)\,\mathrm{d}t\,. \tag{5.2}$$

Examples of problems taking the form of (5.2) can be found in models for: population growth, thermal cooling, circuit theory, and projectile motion. How these systems evolve is not at all random, but completely deterministic.

Naturally though, there are systems which are subject to random effects which influence how they evolve. Examples of systems experiencing random influences can be found in models from finance, dilute chemistry, and various branches of physics. If a system behaves randomly we say it is *stochastic*, and these can regularly be described by using what's called a *stochastic differential equation* of the form

$$\mathrm{d}X_t = a(t, X_t)\,\mathrm{d}t + b(t, X_t)\,\mathrm{d}W_t\,. \tag{5.3}$$

The new term $\mathrm{d}W_t$ is the underlying stochastic process, where W_t is called a *Brownian motion*. The most important thing about a Brownian motion is that between two times s and t, with $s < t$, the change $W_t - W_s$ is a zero mean Gaussian random variable with variance $t - s$. This means that over a time interval Δt, the change in the Brownian motion ΔW_t can be expressed using a standard Gaussian random variable Z by $\Delta W_t = \sqrt{\Delta t}\,Z$.

When simulating a path described by (5.3), it is important not to forgetting that no two simulations of (5.3) will be the same, as each path is subjected to unique random influences, as was demonstrated in Figure 2.3.

A popular and robust way to simulate processes described by (5.3) up to time T is to use *the Euler-Maruyama scheme*, which makes the approximation

$$\Delta X_t \approx a(t, X_t)\Delta t + b(t, X_t)\Delta W_t, \tag{5.4}$$

which gives rise to the Euler-Maruyama scheme

$$\hat{X}_{n+1} = \hat{X}_n + a(t_n, \hat{X}_n)\delta + b(t_n, \hat{X}_n)\sqrt{\delta}\,Z_n, \tag{5.5}$$

where we simulate from $\hat{X}_0 \to \hat{X}_1 \to \hat{X}_2 \to \cdots \to \hat{X}_n \to \cdots \to \hat{X}_N$, using N equally spaced time increments $\delta = \frac{T}{N}$ where $t_n = n\delta$.

To estimate exactly where the final position X_T is expected to be, the Monte Carlo scheme uses M simulations to approximate

$$\mathbb{E}(X_T) \approx \mathbb{E}\Big(\hat{X}_N\Big) \approx \frac{1}{M}\sum^{M} \hat{X}_N. \tag{5.6}$$

Performing all these M simulations, each requiring N random numbers, the amount of random numbers used can quickly become enormous, and generating them becomes the computational bottleneck. This is where our approximate random variables come into play. Instead of the random variables Z_n, which exactly follow the Gaussian distribution, we propose using approximate random variables $\tilde{Z}_n$ and the modified version of the Euler-Maruyama scheme

$$\tilde{X}_{n+1} = \tilde{X}_n + a(t_n, \tilde{X}_n)\delta + b(t_n, \tilde{X}_n)\sqrt{\delta}\,\tilde{Z}_n. \tag{5.7}$$

Switching to the modified Euler-Maruyama scheme will give a considerable speed improvement, but it will introduce some error. Is there a way to benefit from the increased speed but without losing accuracy? Fortunately, there is an extension of regular Monte Carlo called *multilevel Monte Carlo* which does just this, and the idea behind it is surprisingly simple. We suppose that in some setting we have the means to run simulations in some very high fidelity with a high computational cost, or a low fidelity with a low computational cost, where these are approximations to some exact quantity. By convention we denote these as *fine* and *coarse* simulations respectively. The multilevel estimate uses the combination

$$\overbrace{\mathbb{E}(X_{\text{exact}}) \approx \underbrace{\mathbb{E}(X_{\text{fine}})}_{\text{Multilevel Monte Carlo}}}^{\text{Regular Monte Carlo}} = \mathbb{E}(X_{\text{coarse}}) + \mathbb{E}(X_{\text{fine}} - X_{\text{coarse}}) \,. \tag{5.8}$$

If we inspect the portion marked regular Monte Carlo, everything is as one might expect, and we would estimate the exact quantity using the finest resolution possible. The interesting bit is the equality we've marked as the multilevel Monte Carlo. The fact that this is an equality means that the same accuracy as the regular Monte Carlo is achieved, and no accuracy is lost. So why change to the multilevel scheme? The multilevel scheme's first expectation $\mathbb{E}(X_{\text{coarse}})$ is our new primary estimation tool, which we can see is made from coarse simulations, and so is computationally cheaper, giving a speed improvement. The second expectation $\mathbb{E}(X_{\text{fine}} - X_{\text{coarse}})$ is the correction needed to adjust from a coarse simulation to a fine simulation, and is how the fine simulation's accuracy is maintained. This second expectation only requires a negligible number of simulations, and hence doesn't affect the overall cost. This is why the multilevel scheme has all the speed of a coarse simulation and the accuracy of a fine simulation. Using this multilevel Monte Carlo, the higher fidelity simulations are produced from the Euler-Maruyama scheme using the exact random numbers, and faster but lower fidelity simulations are produced using the approximate random numbers. We gain speed without losing accuracy.

Usually the fine and coarse simulations differ by using smaller and larger time increments. Our approximate random variables present a separate way to boost speed without losing performance. Unfortunately, incorporating approximate random variables into a pre-existing regular Monte Carlo or multilevel Monte Carlo is not without some technical difficulties. Approximate random variables cannot be used for all simulations, there are limitations, and these are too mathematically technical to repeat here. For anyone looking to implement a Monte Carlo scheme using them, we recommend reading some of the more advanced material mentioned in Section 2.1.

The next few sections need to be checked and largely stripped of their references, as this is a simple getting started guide.

5.1 Compiler optimisations

When discussing the Kahan compensated summation, we mentioned that an optimising compiler might interfere with the desired computation which was specified or described by the programmer's code. It is likely the case that if a practitioner is willing to explore the use of approximate random variables, low-precision floating-point numbers, programming in a low-level compiled language (e.g. C, C++, Fortran, etc.), and using SIMD and vector instructions, that they are trying to extract the maximum computational performance achievable, and so can be assumed to also be exploring various degrees of compiler optimisation. We would like to take this opportunity to warn against one possible and rather subtle optimisation the compiler might perform, which will ultimately violate the telescopic summation central to the multilevel Monte Carlo framework.

```
1  inline double difference_quantised(const double * brownian_path)
2  {
3      /* Difference between levels using the approximate Euler-Maruyama scheme. */
4      double Payoff_fine_quantised, Payoff_coarse_quantised;
5      /* Computing the payoffs from the same underlying Brownian path. */
6      return Payoff_fine_quantised - Payoff_coarse_quantised;
7  }
8
9  inline double difference(const double * brownian_path)
10 {
11     /* Difference between levels using the exact Euler-Maruyama scheme. */
12     double Payoff_fine, Payoff_coarse;
13     /* Computing the payoffs from the same underlying Brownian path. */
14     return Payoff_fine - Payoff_coarse;
15 }
16
17 double quantised_estimator(const double * brownian_path)
18 {
19     return difference_quantised(brownian_path);
20 }
21
22 double quantised_estimator_correction(const double * brownian_path)
23 {
24     return difference(brownian_path) - difference_quantised(brownian_path);
25 }
```

Code 5.1 – Compiler and hardware optimisation of inlined quantised multilevel Monte Carlo.

Suppose an implementation wishes to follow the DRY[1] design pattern and has a modular structure using inlined functions as shown in Code 5.1. (Such function inlining may be explicit, as in Code 5.1, or may be implicitly performed by the compiler under a sufficiently high optimisation flag[2]). If in Code 5.1 the functions `difference` and `difference_quantised` are inlined by the compiler, then the calculations may be optimised differently within `quantised_estimator` and `quantised_estimator_correction`. The compiler could re-order the arithmetic, utilise FMA operations (recalling FMA operations work in infinite intermediate precision[3]), reorder the typecasting and arithmetic, or perform any number of optimisations. There is no guarantee that the optimisations performed on one inlined function be the same as another, thus these optimisations can results in the telescopic sums in **????** being violated, nullifying the multilevel Monte Carlo analysis.

As an example of how subtle this behaviour can be, let us consider the effect of implicit typecasting. For brevity we denote $\hat{P}^{\mathrm{f}}$, $\hat{P}^{\mathrm{c}}$, $\bar{P}^{\mathrm{f}}$, and $\bar{P}^{\mathrm{c}}$ by the floating point-variables `P_f_32`, `P_c_32`, `P_f_16`, and `P_c_16` respectively, where `P_f_32` and `P_c_32` are single-precision floats, and `P_f_16` and `P_c_16` are half-precision floats. Suppose we implement the multilevel estimator and correction as in Code 5.2. On the surface, everything in Code 5.2 seems mathematically correct, but when calculating the multilevel estimator `estimator`, the fine and coarse path are differenced in low-precision, equivalent to using $\ominus$. However, addition and subtraction in C are left-to-right associative, and hence the compiler can evaluate the multilevel correction `correction` in a manner equivalent to `((P_f_32 - P_c_32) - P_f_16) + P_c_16`. This would mean all the variables are typecast to single-precision, and no half-precision difference is ever computed. We can see then, that

[1] *Don't repeat yourself* (DRY).

[2] GCC for example will inline functions used only once under `-O1`, and will optimise unmarked small functions and perform partial inlining under `-O2`.

[3] "The operation `fusedMultiplyAdd(x, y, z)` computes $(\mathtt{x}\times\mathtt{y})+\mathtt{z}$ as if with un-bounded range and precision, rounding only once to the destination format", **?** , page 4, definition 2.1.28].

```
float P_f_32, P_c_32;
half P_f_16, P_c_16;
/* Performing the necessary path simulations. */
half estimator = P_f_16 - P_c_16;
float correction = P_f_32 -  P_c_32 - P_f_16 + P_c_16;
```

Code 5.2 – Possible implicit typecasting within the finite-precision multilevel Monte Carlo framework.

this is equivalent to the Monte Carlo estimator

$$\mathbb{E}\Big(\bar{P}^{\mathrm{f}} \ominus \bar{P}^{\mathrm{c}}\Big) + \mathbb{E}\Big(\Big(\hat{P}^{\mathrm{f}} - \hat{P}^{\mathrm{c}}\Big) - \Big(\bar{P}^{\mathrm{f}} - \bar{P}^{\mathrm{c}}\Big)\Big) \neq \mathbb{E}\Big(\hat{P}^{\mathrm{f}} - \hat{P}^{\mathrm{c}}\Big), \tag{5.9}$$

which is not the intended multilevel Monte Carlo telescopic summation. The compiler can readily jumble the order of arithmetic and associativity under heavy enough optimisation settings, such as `-Ofast`. Ensuring the telescoping summation is respected remains the responsibility of the programmer, and we acknowledge that spotting if or when it is violated can be a very tricky challenge.

Instinctively, an inexperienced programmer might speculate that such divergences of the actual computation from the desired computation will only be minor and likely inconsequential. However, an experienced programmer needn't look far to realise that code which is only slightly different from what was intended is typically the most dangerous, causing erroneous results which are difficult to identify and devilish to debug [**?** , Chapter 27]. Furthermore, the wider the adoption of an approach, the sooner and more readily corner cases will arise, either from real world applications or carefully construed counter examples. Ultimately, we cannot stress enough the importance of being vigilant in working to exactly preserve the multilevel Monte Carlo telescoping summation.

5.2 Coupling uniform random variables in different precisions

We have seen that the compiler can interfere with the intended functionality of the code. In addition to this, we would like to further highlight two other instances where an implementation may seemingly appear correct and sensible, but where extra care and vigilance needs to be taken to ensure correctness. Both of these considerations surround the low-precision uniform random numbers.

5.2.1 Low-precision uniform distributions when typecasting through intermediate precisions

For our multilevel Monte Carlo simulations we would like to couple our simulations. This can be done by the simulations sharing the same underlying Weiner process. If the Euler-Maruyama scheme is being implemented with the inverse transform method used to generate the Gaussian random variables, this is achieved by the simulations using the same sequence of underlying uniform random variables.

For simplicity, let us assume we are only trying to couple two path simulations. If these are both in the same precision, such as double-precision Euler-Maruyama schemes using exact and approximate random variables, then both will use the same underlying uniforms, and all is well. However, let us now suppose that we are trying to drive paths at different precisions, such as a high and low-precision Euler-Maruyama scheme, both using exact Gaussian random variables. Let the higher precision have r bits for its mantissa, and let the lower precision have r', where $r > r' > 0$. For the multilevel correction, we would like our underlying uniform

712 random number generator to produce a uniform random number, represented twice: once in
713 high-precision, and the same number again in low-precision. Let us consider how ideally this
714 would proceed in Algorithm 5.1, and how a naïve implementation of this may look in Code 5.3.

Input: A random sample ω (equivalent to a seed for a random number generator).
Output: High and low-precision uniform random numbers.
1 Sample a uniform random number $U \equiv U(\omega)$.
2 Obtain a high-precision representation $\bar{U}^{(r)}$ of U.
3 Obtain a low-precision representation $\bar{U}^{(r')}$ of U.

Algorithm 5.1: Obtaining two varying precision representations of an underlying uniform random variable.

```
1 float u_32 = uniform_random_number_generator();
2 half u_16 = u_32;
```

Code 5.3 – A naïve implementation of Algorithm 5.1 for producing coupled and varying precision uniform random variables for the multilevel Monte Carlo correction term.

715 Algorithm 5.1 is evidently the desired procedure for producing our multilevel
716 correction, and at first sight Code 5.3 appears a valid implementation. To see where the pitfall
717 lies in this implementation, we will need to dig slightly deeper into the uniform random
718 number generator. For more information on uniform random number generation we
719 recommend **?**], **?**], **?**], and **?**].

720 Many uniform random number generators[4] proceed by constructing a sequence of
721 integers $\{m_1, m_2, \ldots\}$ modulo a very large maximum number M (typically one less than a
722 power of 2), where the integer then is mapped to the range $[0, 1)$ by dividing by M. As the
723 random number generators are constructed to have extremely long periods, they typically have
724 some large state (or state vector), and support values of M which are frequently greater than
725 2^{32}. When scaling a sequence value m to $[0, 1)$ by dividing by M, to avoid losing precision, the
726 two values of m and M are typically converted to floating-point numbers first, and then the
727 division performed on the two floats. As the integer sizes are typically quite large, these are
728 first typecast to double-precision floating-point data-types, and then a floating-point division
729 operation is performed. This means that a uniform random number generators might typically
730 have a function signature resembling that depicted in Code 5.4. As an example, the standard
731 uniform random number generator from GSL `gsl_rng_uniform` matches the function signature
732 from Code 5.4, as does the equivalent NAG function `nag_random_continuous_uniform`[5].

```
1 double uniform_random_number_generator(void)
2 {
3     static unsigned long int m, M;
4     // Compute the sequence value m and scale to [0, 1) by dividing by M.
5     return (double) m / (double) M;
6 }
```

Code 5.4 – A typical uniform random number generator function signature.

[4] A similar procedure also occurs for low-discrepancy sequences producing quasi-random numbers. Examples of low-discrepancy sequences include; Halton, Hammersley, Kronecker [**?** , 1.1.53], Richtmyer [**?**], Ramshaw [**?**], Lapeyre-Pagès [**?** , 2.C.4.3], Niederrieter, Sobol, Faure, and van der Corput [**? ? ? ?**].

[5] The function `nag_random_continuous_uniform` has been replaced by `nag_rand_basic` and `nag_rand_uniform` in recent versions of the NAG library.

We can see from this consideration that the usual data-type output by uniform random number generators can be expected to be a double-precision floating-point value. If we then incorporate the function signature from Code 5.4 into the implementation in Code 5.3, we can see that the uniform random number generator `uniform_random_number_generator` will output a `double`. This is then implicitly typecast to a single-precision `float` when assigning a value to `u_32`. Then, when assigning a value to `u_16`, the single-precision `float` is then typecast a second time from a `float` down to a half-precision `half`. To show why this is problematic, we now need to consider how the uniforms for the new multilevel Monte Carlo estimator will be produced, which will likely be equivalent to Code 5.5.

```
1 half u_16 = uniform_random_number_generator();
```

Code 5.5 – Producing low-precision uniform random variables for the new multilevel Monte Carlo estimator.

We can see from Code 5.5 that the half-precision uniforms are now obtained directly by typecasting from `double` to `half`, without passing through the intermediate `float` format. Thus we can see that in the multilevel Monte Carlo estimator, the low-precision cheap simulations are produced by $\mathscr{R}(U;r')$, whereas if we use Code 5.3 then in the correction term they are produced by $\mathscr{R}(\mathscr{R}(U;r);r')$. The subtle point here is that these are not the same distribution, and thus are violating the telescoping summation central to the multilevel Monte Carlo framework.

Two distributions can differ drastically if there is a small decrement in the precision in going from r to r'. As an example, consider the set of representable values in the range $(\frac{1}{2},1)$ for the precision levels r and r' when $r=r'+1$, as depicted in Figure 5.1. We can see the impact of the round to nearest even rounding mode causes the two distributions to differ considerably. Based on these considerations, the correct way to implement Algorithm 5.1 is by Code 5.6 rather than Code 5.3.

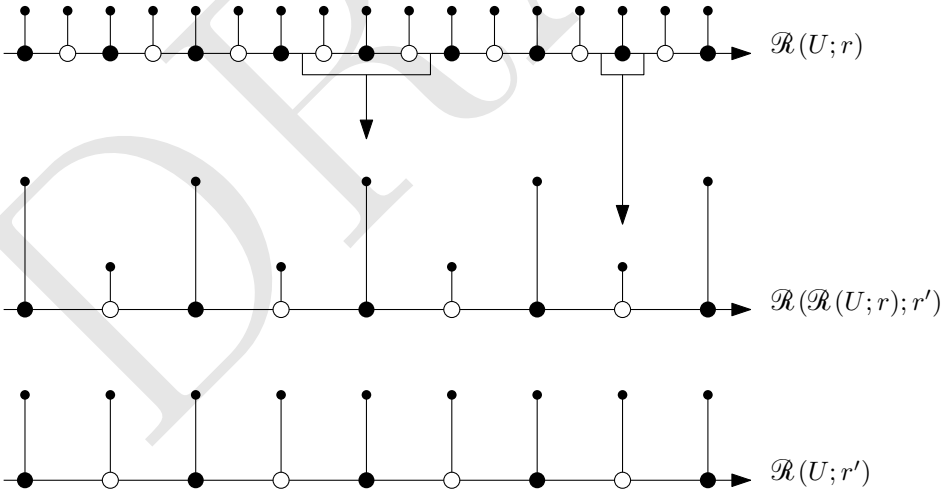


Figure 5.1 – The representable floating-point values in the range $(\frac{1}{2},1)$ for precisions r and r' with $r=r'+1$. The even numbers are denoted by solid markers (●), and odd numbers by hollow markers (○). We indicate the probability density functions for each value, indicating the contributions for $\mathscr{R}(\mathscr{R}(U;r);r')$ for an even and odd value.

The reason this is particularly important concerns low-precision SIMD implementations. It is crucial for a performant multilevel simulation using approximate random variables that input vectors to $\tilde{\Phi}^{-1}(\cdot)$ contain the same data-type as the output.

This is a very subtle implementation pitfall, and the risk of this is compounded by using an optimising compiler. If a compiler is optimising heavily enough, and desires to keep the

```
double u_64 = uniform_random_number_generator();
float u_32 = u_64;
half u_16 = u_64;
```

Code 5.6 – An improved implementation of Algorithm 5.1 for producing coupled and varying precision uniform random variables for the multilevel Monte Carlo correction term.

number of registers used to a minimum, then a compiler may optimise Code 5.6 into Code 5.3. Ensuring this does not occur, (or at least is not unknowingly performed), is the responsibility of the programmer.

5.2.2 Overflow resulting from low-precision typecasting

There is a second risk present when typecasting high-precision uniform random variables down to a low-precision, which is manifestly present if a practitioner reduces the precision without also switching to approximate random variables. The issue is that the singularities of $\Phi^{-1}(\cdot)$ become attainable with an appreciable probability when the uniform random numbers are typecast to a low-precision.

To make this point clearer, we recall that for the Gaussian distribution, $\Phi^{-1}(\cdot)$ has singularities at $\{\pm 0, 1\}$, for which it is typical for library implementations of $\Phi^{-1}(\cdot)$ to return either of the floating-point values NaN or $\pm\infty$. Encountering either of these is likely to contaminate any Monte Carlo calculation. When the floating-point value is typecast down to `half` from either `double` or `float`, then for IEEE 16-bit half-precision floating-point variables, there is a 0.024% chance[6] a uniform random variable $U \sim \mathcal{U}([0, 1))$ will be round in the typecasting to the value 1. Thus typecasting to `half` changes the range of U from $[0, 1) \to [0, 1]$.

If our low-precision simulations are using approximate random number distributions which are non-singular on the domain $[0, 1]$, then this result from typecasting presents no problem. However, if a practitioner only reduces the precision without introducing approximate random variables, instead choosing to maintain an exact and singular implementation of $\Phi^{-1}(\cdot)$, then very quickly they will encounter singular Gaussian random variables. Our framework utilising approximate random variables circumvents this problem. However, if a practitioner or application should remain averse to using approximate random numbers, then they will need to decide how to try and circumvent this issue. One method we propose to mediating this is that if the typecast number produces 1, instead compute the associated Gaussian random variable using the inverse of the complementary cumulative distribution function, as demonstrated in Code 5.7.

```
bool non_singular = u_16 != 1.0f16;
u_16 = non_singular ? u_16 : 1.0 - u_64;
z_16 = uniform_to_gaussian_16(u_16);
z_16 = non_singular ? z_16 : -z_16;
```

Code 5.7 – Computing a low-precision Gaussian random variable using the inverse complementary CDF.

[6]For an IEEE 16-bit half-precision floating-point number, 10 bits are reserved for the mantissa, meaning the half-precision value just below 1 is $1.1111111111_2 \times 2^{-1} \approx 0.999511_{10}\ldots$, and hence the probability of rounding up to 1 is $\frac{1-0.999511}{2} = 0.000244\ldots$