FAST AND ACCURATE PARALLEL COMPUTATION OF QUANTILE FUNCTIONS FOR RANDOM NUMBER GENERATION

THOMAS LUU

Thesis submitted for the degree of Doctor of Philosophy of University College London.

2016

UCL Department of Mathematics

Declaration

I, Thomas Luu confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

The fast and accurate computation of quantile functions (the inverse of cumulative distribution functions) is very desirable for generating random variates from non-uniform probability distributions. This is because the quantile function of a distribution monotonically maps uniform variates to variates of the said distribution. This simple fact is the basis of the inversion method for generating non-uniform random numbers. The inversion method enjoys many significant advantages, which is why it is regarded as the best choice for random number generation. Quantile functions preserve the underlying properties of the uniform variates, which is beneficial for a number of applications, especially in modern computational finance. For example, copula and quasi-Monte Carlo methods are significantly easier to use with inversion. Inversion is also well suited to variance reduction techniques. However, for a number of key distributions, existing methods for the computational of their quantile functions are too slow in practice. The methods are also unsuited to execution on parallel architectures such as GPUs and FPGAs. These parallel architectures have become very popular, because they allow simulations to be sped up and enlarged.

The original contribution of this thesis is a collection of new and practical numerical algorithms for the normal, gamma, non-central χ^2 and skew-normal quantile functions. The algorithms were developed with efficient parallel computation in mind. Quantile mechanics—the differential approach to quantile functions—was used with inventive changes of variables and numerical methods to create the algorithms. The algorithms are faster or more accurate than the current state of the art on parallel architectures. The accuracy of GPU implementations of the algorithms have been benchmarked against independent CPU implementations. The results indicate that the quantile mechanics approach is a viable and powerful technique for developing quantile function approximations and algorithms.

Dedication

To my favourite aunt, who died in January 2015.

Acknowledgements

I would like to take this opportunity to acknowledge the three Fs that made this thesis possible: finance, friends and family.

I am eternally grateful to Prof. William Shaw for choosing me to work on this project. He and Dr. Robert Tong imparted invaluable knowledge and advice, which was pivotal at times. They also provided just the right amount of motivation and encouragement when I needed it. I also thank Prof. Valery Smyshlyaev for acting as my secondary supervisor.

I am thankful to the Industrial Mathematics Knowledge Transfer Network (KTN), the Engineering and Physical Sciences Research Council (EPSRC) and the Numerical Algorithms Group (NAG) for support under a KTN–EPSRC CASE award. I thank Lorcan Mac Manus of the Smith Institute for his time acting as the project liaison.

NAG were incredibly welcoming and helpful throughout this project. They also gave me access to opportunities and resources that would have otherwise been difficult to find or get. I thank all members of NAG, but especially Dr. Martyn Byng, Dr. Mike Dewar, Dr. Jacques du Toit, John Holden, Dr. Lawrence Mulholland, Mick Pont and Dr. David Sayers.

Dr. Asad Munir gave me a lot of PhD survival tips and clarified various aspects of relevant theory, for which I am very grateful.

Last, but by no means least, I would like to thank my parents and sisters for simply being there.

Contents

Chapter 1

Introduction

On 21 May 1947, John von Neumann wrote the following in a letter to Stanislaw Ulam [25]:

> I am very glad that preparations for the random numbers work are to begin soon. In this connection, I would like to mention this: Assume that you have several random number distributions, each equidistributed in $0, 1 : (x^i), (y^i), (z^i), \ldots$. Assume that you want one with the distribution function (density) $f(\xi)\,d\xi : (\xi^i)$. One way to form it is to form the cumulative distribution function: $g(\xi) = \int_\xi f(\xi)\,d\xi$ to invert it $h(x) = \xi \leftrightarrow x = g(\xi)$, and to form $\xi^i = h(x^i)$ with this $h(x)$, or some approximant polynomial. This is, as I see, the method that you have in mind.
>
> An alternative, which works if ξ and all values of $f(\xi)$ lie in $0, 1$, is this: Scan pairs x^i, y^i and use or reject x^i, y^i according to whether $y^i \leqslant f(x^i)$ or not. In the first case, put $\xi^j = x^i$ in the second case form no ξ^j at that step.
>
> The second method may occasionally be better than the first one. [*continues*]

The reader may recognise these as the now well-known inversion and rejection methods. In the previous year, Ulam described what was later called the Monte Carlo method [53] to von Neumann. The Monte Carlo method, of course, necessitates the need for random numbers, typically from one or more non-uniform probability distributions. The inversion and rejection methods are two of the most general ways to generate non-uniform random numbers. However, the former has many theoretical virtues over the latter. Von Neumann alludes to the potential impracticability, much of which remains today, of the inversion method. This thesis focuses on expanding the practical applicability of the inversion method. An extensive discussion on sampling distributions by rejection and other means is given by L. Devroye in [21], which is widely considered as the bible of non-uniform random number generation.

Von Neumann appreciated the value of Ulam's statistical sampling idea. He was also aware of Eniac[1], the first general-purpose electronic computer, which was completed in 1946. The Monte Carlo method was subsequently implemented on Eniac, to solve the neutron diffusion problem in fission devices. The Monte Carlo method has, of course, since found application in a wide range of disciplines. The random numbers for the neutron simulations on Eniac were originally read from punchcards.[2] This was relatively slow.

Von Neumann was conscious of the need for faster sourcing of uniformly distributed random numbers. Deriving randomness from physical phenomenon, e.g. radioactive decay, yields "truly" random numbers. However, the process is slow and reproducibility is an issue. Computers, being deterministic, can generate "pseudo" random numbers.

> Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin.
>
> — John von Neumann

Von Neumann proposed [82] generating a sequence of pseudo-random numbers using a method now known as middle-squaring: a seed value is squared, the middle digits of the result are taken as the next random number and the seed is reset to the generated random number. Many sequences generated using the middle-square method converge to zero, but von Neumann was aware of this. Crucially, with his method, he found that random numbers could be generated hundreds of times faster than reading them from punchcards. The middle-square method as since been supplanted by more sophisticated random number generation algorithms. Linear congruential generators were the most popular for some time, but the Mersenne

[1]Electronic Numerical Integrator And Computer

[2]The Monte Carlo method spurred the Rand Corporation, in 1947, to produce a random number table, which was significantly larger than any other publicly available table, consisting of a million random digits. The table was published in 1955, in book and punchcard form. The book was actually reissued in 2001. The "customer" reviews of the reissued *A Million Random Digits with 100,000 Normal Deviates* on Amazon.com are quite humorous.

Twister [52] algorithm is arguably the current (at the time of writing) pseudo-random number generator (RNG) of choice. The Mersenne Twister is currently the default pseudo-RNG in many industry standard software, including, but not limited to, *Matlab* and R [65]. More recent pseudo-RNGs include Marsaglia's Xorshift [49] and Random123 [66].[3] See, e.g., [64, Chapter 7] or [27, Chapter 1] for a comprehensive overview of sampling the uniform distribution. Given a source of uniform random numbers, non-uniform distributions can be simulated, as suggested by von Neumann's letter to Ulam.

The most direct way to generate a non-uniform variate is by inversion. This requires the inverse cumulative distribution function (CDF), also known as the quantile function[4], of the distribution in question to be computable. Let $F(x)$ be the CDF of the distribution. Now let $q = F^{-1}$ be the functional inverse of the CDF. If U is a standard uniform random variable then $X = q(U)$ has distribution F. If the quantile function q has a closed-form expression, the generation of variates from the distribution by inversion is clearly a straightforward matter. Table 1.1 shows a collection of distributions that have such a q. The presence of a *shape parameter*—that is, a parameter which neither shifts nor scales the distribution—does not significantly impact the computational complexity in this case. Things are quite different for distributions with no closed-form CDF. Table 1.2 shows some distributions that are non-trivial to invert, due to their non-elementary F and q functions. More often than not, a distribution with no closed-form CDF will also have one or more shape parameters. In this case, numerical inversion methods must be used to compute their quantile function. Resorting to root finding is very much commonplace. Evaluating F one or more times for this is known to be computationally expensive, rendering inversion impractical for applications such as live, real-time Monte Carlo simulation. Quantile function implementations based on root finding are typically more than an order of magnitude slower than evaluating a quantile function from Table 1.1. Being within an order of magnitude would be a more ideal situation, making inversion tractable in practice. It should be noted that inversion is very rarely the fastest method for sampling distributions.

In reality, generating variates from non-elementary distributions by inversion is usually forsaken for clever transformations or rejection sampling, which are typically more computationally feasible. However, with the inversion method,

[3]The latter is particularly interesting, because it parallelises very well.

[4]A very thorough treatment on the statistical applications of quantile functions is given by W. Gilchrist in [29].

Table 1.1: The CDF F and quantile function q of some commonly used distributions. These distributions possess elementary closed-form expressions for F and q. Each $q(u)$ is valid for $u \in (0, 1)$, which implicitly defines the domain of the adjacent F. The values of $q(0)$ and $q(1)$ are typically defined as zero or $\pm\infty$. Location parameters can be any real number. Scale and shape parameters can be any positive real number.

	Parameters	$F(x)$	$q(u) \equiv F^{-1}(u)$
Cauchy	a (location) b (scale)	$\frac{1}{\pi}\arctan\left(\frac{x-a}{b}\right)+\frac{1}{2}$	$a + b\tan\left[\pi\left(u-\frac{1}{2}\right)\right]$
Exponential	λ (shape)	$1-e^{-\lambda x}$	$-\frac{\log(1-u)}{\lambda}$
Laplace	μ (location) β (scale)	$\begin{cases}\frac{1}{2}e^{\frac{x-\mu}{\beta}} & \text{if } x<\mu\\ 1-\frac{1}{2}e^{-\frac{x-\mu}{\beta}} & \text{if } x\geqslant\mu\end{cases}$	$\begin{cases}\mu+\beta\log(2u) & \text{if } u\leqslant\frac{1}{2}\\ \mu-\beta\log\left[2(1-u)\right] & \text{if } u>\frac{1}{2}\end{cases}$
Pareto	k (scale) α (shape)	$1-\left(\frac{k}{x}\right)^{\alpha}$	$k(1-u)^{-1/\alpha}$
Uniform	$a<b$	$\frac{x-a}{b-a}$	$bu+a(1-u)$
Weibull	α (shape) β (scale)	$1-e^{-\left(\frac{x}{\beta}\right)^{\alpha}}$	$\beta\left[-\log(1-u)\right]^{1/\alpha}$

- uniform variates are monotonically mapped to variates of the non-uniform distribution; and
- only one uniform variate is required per sample generated.

These are not guaranteed with transformations or the rejection method, which are important disadvantages. The advantages of the inversion method are beneficial for a number of applications, especially in modern computational finance. Copula and quasi-Monte Carlo methods are pertinent examples, and they are significantly easier to use with inversion (see, e.g., [31, 47, 43]).

The dimension of problems solved by quasi-Monte Carlo simulation is usually equal to the number of uniform variates required to generate each sample path. Since the number of uniform variates needed to generate one non-uniform variate using the rejection method is unbounded, simulations that rely upon rejection turn into infinite-dimensional problems. Such problems are typically incompatible with quasi-Monte Carlo. Inversion is also well suited to the variance reduction techniques. Common random numbers and antithetic variates are two examples. Positively and negatively correlated variates are required for these variance reduction techniques. The inversion method allows us to do this in a very straightforward manner. Moreover, the strongest possible correlation can be induced by the inversion method. Because the rejection method does not guarantee a monotonic

Table 1.2: The CDF F and quantile function q of some commonly used distributions. These distributions *do not* possess elementary closed-form expressions for F and q. Each $q(u)$ is valid for $u \in (0,1)$, which implicitly defines the domain of the adjacent F. The values of $q(0)$ and $q(1)$ are typically defined as zero or $\pm\infty$. Location parameters can be any real number. Scale and shape parameters can be any positive real number.

	Parameters	$F(x)$	$q(u) \equiv F^{-1}(u)$
Beta	α (shape) β (shape)	$I_x(\alpha,\beta)$	$I_u^{-1}(\alpha,\beta)$
Chi-squared	ν (shape)	$P\left(\frac{\nu}{2},\frac{x}{2}\right)$	$2P^{-1}\left(\frac{\nu}{2},u\right)$
F	n (shape) m (shape)	$I_{\frac{nx}{nx+m}}\left(\frac{n}{2},\frac{m}{2}\right)$	$\frac{m}{n\left[\frac{1}{I_u^{-1}\left(\frac{n}{2},\frac{m}{2}\right)}-1\right]}$
Gamma	α (shape) β (scale)	$P\left(\alpha,\frac{x}{\beta}\right)$	$\beta P^{-1}\left(\alpha,u\right)$
Log-normal	μ (location) σ (scale)	$\frac{1}{2}\mathrm{erfc}\left[\frac{\mu-\log(x)}{\sqrt{2}\sigma}\right]$	$e^{\mu-\sqrt{2}\sigma\mathrm{erfc}^{-1}(2u)}$
Normal	μ (location) σ (scale)	$\frac{1}{2}\mathrm{erfc}\left(\frac{\mu-x}{\sqrt{2}\sigma}\right)$	$\mu-\sqrt{2}\sigma\mathrm{erfc}^{-1}(2u)$
Student t	ν (shape)	$\frac{1}{2}\left(1+\mathrm{sgn}\left(x\right)\left[1-I_{\frac{\nu}{x^2+\nu}}\left(\frac{\nu}{2},\frac{1}{2}\right)\right]\right)$	$\mathrm{sgn}\left(u-\frac{1}{2}\right)\sqrt{\nu\left[\frac{1}{I^{-1}_{2\min(u,1-u)}\left(\frac{\nu}{2},\frac{1}{2}\right)}-1\right]}$

mapping between uniform and non-uniform variates, production of correlation is non-trivial. Furthermore, with inversion, small distribution parameter perturbations cause small changes in the produced variates. This effect is useful for sensitivity analysis. Contrast this to the rejection method, where small parameter perturbations can cause large changes in the produced variates.[5] The only real disadvantage of variate generation via inversion is potentially the computational speed of existing methods. If this can be significantly improved, the benefits of inversion can be realised in practice.

Monte Carlo simulations are naturally suited to parallel computation, since samples are generated independently of each other. Indeed, performing Monte Carlo simulations is a popular use of modern many-core architectures, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs).[6] These allow substantially more samples to be simulated in a time period, compared to traditional CPUs. This is very desirable, because a sufficiently accurate solution can be found much faster. However, to take full advantage of the performance of many-core processors, the algorithms that they run have to be as branch-free as possible.

CPUs are not unduly affected by conditional statements in code, due to their sophisticated branch prediction mechanisms. This is in contrast to GPUs and other many-core architectures, which are particularly sensitive to divergent flows of execution. When branching occurs, each branch taken is effectively serialised, so the execution time of a conditional statement is roughly equal to the sum of each branch taken. *Branch divergence* is said to occur in this case. If the branches are computationally expensive, valuable parallelism is lost. (Nested conditional statements obviously compound the problem.) Peak performance is therefore achieved with straight-line code that has only one execution path. Branch divergence can otherwise significantly affect parallel performance. It is a particular issue for traditional methods of quantile function computation. The tails of distributions are usually managed separately, which causes branch divergence. Branch divergence is also an issue for the rejection method, but in a more fundamental way. This is because the number of uniform variates required to produce one non-uniform

[5]This is because, even after a slight parameter perturbation, a previously accepted variate could be rejected. When this occurs, the subsequently generated variates will be completely different from before.

[6]Interestingly, Intel have started integrating FPGAs in certain Xeon central processing units (CPUs) and in mid 2015 announced they will acquire Altera, a market leading manufacturer of FPGAs.

variate is not fixed. The wasted effort in rejecting candidate non-uniform variates is thus amplified.

Random variate generation is generally the most computationally expensive part of generating a Monte Carlo sample. It would, therefore, be desirable to have branch-free algorithms for random variate generation. This demands a fresh look at the relevant numerical algorithms. Fast, parallel random variate generation is desirable, because simulations are getting larger, and to take advantage of emerging parallel architectures. Several popular uniform pseudo- and quasi-RNGs were implemented and evaluated on the GPU in [16]. We build on this work by enabling efficient parallel non-uniform variate generation by inversion.

We can relate our work to functions from some existing mathematical software. In terms of *Mathematica* [84], the functions `RandomVariate`, `InverseCDF` and `Quantile` are very closely related to our research.

- `RandomVariate[`*`dist`*`,`*`n`*`]` gives a list of n pseudo-random variates from the distribution *dist*. For example,

  ```
  RandomVariate[NormalDistribution[],100]
  ```

  would generate a list of 100 *pseudo*-random variates that follow the standard normal distribution. At the time of writing, the *Mathematica* RNG documentation [85] says that these normal variates would be generated using the Box–Muller method [15]. No mechanism is currently available to stipulate how non-uniform random variates are generated by `RandomVariate`. *Matlab* has an analogous function to `RandomVariate`, called `random`.

- `InverseCDF[`*`dist`*`,`*`u`*`]` and `Quantile[`*`dist`*`,`*`u`*`]` give the inverse of the cumulative distribution function for the distribution *dist* as a function of the variable u. The two functions are equivalent in this context. If *`u`* is a list of uniformly distributed random numbers, the quantile function of *`dist`* would be applied to each element of the list. This would be similar to the behaviour of `RandomVariate`, except that the output is not necessarily pseudo-random and, of course, inversion is the generation method. *Matlab* has an analogous function to `InverseCDF/Quantile`, called `icdf`.

In 2006, the technology company Nvidia introduced CUDA[7], a hardware and software solution for doing general-purpose parallel computing (as opposed to computer graphics exclusively) on their GPUs. Since its introduction, CUDA has been

[7]Compute Unified Device Architecture

used extensively by scientists in industry and academia. Investment banks are heavy users of GPUs. J.P. Morgan and BNP Paribas have publicly announced that they have used Nvidia GPUs for accelerating their numerical workloads, including Monte Carlo simulations. In the light of this, other top banks are almost certainly using GPUs too.[8] In 2008, OpenCL[9] was introduced. Whereas CUDA can be used to program Nvidia GPUs only, OpenCL supports CPUs and GPUs from multiple vendors (including Nvidia). Intel released their response to Nvidia's foray into general-purpose parallel computing, called Xeon Phi, in 2012. Intel's Xeon Phi incorporates GPU-like technology and supports OpenCL.

The work in this thesis was started when CUDA was well established for many-core computing, and indeed before Intel's Xeon Phi was even announced. While all of the quantile function algorithms in this thesis were implemented and tested in CUDA, to demonstrate real-world parallel performance, they should translate very well to other vector/parallel architectures.

1.1 Contributions and structure of the thesis

This thesis looks, in detail, at the quantile function for four distributions: normal, gamma, non-central χ^2 and skew-normal. For each distribution, new ideas and formulae are introduced and they are used create algorithms that are faster or more accurate than the current state of the art on parallel architectures. Our normal and gamma algorithms fit into the former category. There is very little literature on the quantile function of the non-central χ^2 and skew-normal distributions. We believe that this thesis contains the first comprehensive analysis on these quantile functions, together with new algorithms that cover the parameter and variable space more accurately than existing approximations. The mathematical building blocks that are described in this thesis may also be useful in the future, as a basis for new quantile function algorithms. The following outlines the structure of this thesis and signposts key contributions.

[8]At the time of writing, GPUs are very much mainstream in investment banks, while FPGAs are used on a more speculative basis. If a particular bank can compute answers faster or more accurately than their competitors, the bank generally has the opportunity to make more money.

[9]Open Computing Language

Chapter 2: Approximating quantile functions in practice

- Important practicalities for computing quantile functions in general are detailed here. Subsequent chapters will refer back to this content.
- *Quantile mechanics* [76, 74]—the differential approach to quantile functions—is introduced and reviewed. This thesis makes heavy use of quantile mechanics.

Chapter 3: Normal distribution

- Existing algorithms for computing the normal quantile function are introduced and evaluated on the GPU.
- A new "hybrid" algorithm is developed, which combines several concepts. Single- and double-precision CUDA implementations of this algorithm are shown to be the quickest for computing the normal quantile function on Nvidia GPUs.
- The hybrid algorithm appears in:
  W. T. Shaw, T. Luu, and N. Brickman, *Quantile mechanics II: changes of variables in Monte Carlo methods and GPU-optimised normal quantiles*, European Journal of Applied Mathematics, 25 (2014), pp. 177–212.
  - Moreover, the implementation is integrated into the NAG GPU Library[10].

Chapter 4: Gamma distribution

- A method for parallel inversion of the gamma distribution is developed. We will concentrate on the case of generating large quantities of identically distributed gamma variates at a time. The output of our algorithm has accuracy close to a choice of single- or double-precision machine epsilon. We show that the performance of a CUDA GPU implementation of our algorithm (called Quantus) is within an order of magnitude of the time to compute the normal quantile function.

[10]http://www.nag.co.uk/numeric/GPUs/index

- The gamma quantile algorithm was published in:
  T. Luu, *Efficient and accurate parallel inversion of the gamma distribution*, SIAM Journal on Scientific Computing, 37 (2015), pp. C122–C141.
  - Also, the algorithm implementation has been added to the NAG GPU Library.

Chapter 5: Non-central χ^2 distribution

- Existing non-central χ^2 quantile approximations are known to be deficient in certain areas of the parameter and variable space. A new analytic approximation for the quantile function is derived. It is shown to complement existing approximations very well. Based on this, a hybrid approximation is formed, which has the virtue of computational simplicity, which is desirable for parallel work. The approximations typically have an accuracy of around 3–5 significant figures.
  - It should be noted that the analytic approximation is now used by Boost C++ Math Toolkit's[11] and Wolfram Mathematica's [88] non-central χ^2 quantile function implementations.
- A power series solution for the quantile function is developed.

Chapter 6: Skew-normal distribution

- A central power series solution for the skew-normal quantile function is developed.
- Analytic tail approximations for both tails of the distribution are also found.
- A new numerical algorithm (SNCDFINV) is created, by fusing the aforementioned series solution and tail approximations. While the algorithm does not completely suppress branch divergence, it works quickly on GPUs. On average, SNCDFINV yields results that have about four correct significant figures. We argue that SNCDFINV is superior to Cornish–Fisher approximations with respect to precision. Our algorithm could also be used as a basis for more accurate approximations.

[11]See the Boost 1.55 release notes at http://www.boost.org/users/history/version_1_55_0.html.

Finally, in Chapter 7, the thesis is drawn to a close. While we will focus on four distributions, many of the underlying ideas and techniques can be generalised and extended to other distributions. A number of possibilities for future research directions are also discussed in the closing chapter.

Chapter 2

Approximating quantile functions in practice

Quantile functions that are elementary can be implemented with a few keystrokes using typical built-in mathematical libraries. For example, the exponential quantile function could be implemented in C as follows.

```
double exp_inv(double u, double lambda)
{
    return -log1p(-u) / lambda;
}
```

For the sake of simplicity, it has been assumed that $0 \leqslant u \leqslant 1$ and $\lambda > 0$. Also, care has been taken to ensure accuracy for u less than machine epsilon. The C function `log1p(x)` computes $\log(1+x)$ accurately for small x values. Table 1.1 in Chapter 1 gives other examples of elementary quantile functions. While the Cauchy quantile function

$$a + b \tan\left[\pi\left(u - \frac{1}{2}\right)\right]$$

would appear at first glance to be easy to implement, some consideration is required to ensure that the resulting centre and tail variates are accurate. The following C code correctly implements the Cauchy quantile function. Again, for the sake of simplicity, it has been assumed that $0 < u < 1$, $a \in \mathbf{R}$ and $b > 0$. Infinity should be returned for $u = 0, 1$.

```
double cauchy_inv(double u, double a, double b)
{
    double x;
    x = -cospi(u) / sinpi(u);    /* x = tan(pi * (u - 0.5)) */
    return a + b * x;
}
```

Firstly, the catastrophic cancellation that would occur when a half is subtracted from $u \approx 1/2$ is dispensed with, and secondly, functions are used that accurately compute $\cos(\pi x)$ and $\sin(\pi x)$ for x that are multiples of 1/2.[1]

While elementary quantile functions are simple to implement, non-elementary quantile functions are a different story. Algorithms for computing these quantile functions generally use polynomial or rational approximations, or numerical root finding. The former are usually used for distributions with no shape parameters, while the latter is commonly used when there are. When root finding is employed, two key ingredients are required: a formula with good accuracy for the initial guesses and a sufficiently accurate forward CDF algorithm. Furthermore, both of these should be computationally efficient. This can be an issue for CDF algorithms, which are typically relatively slow. General methods for computing CDFs are surveyed in [41, Section 5.2]. An alternative to root finding is the direct approximation of quantile functions via the ordinary differential equation (ODE) that they satisfy, which is the approach that this work advocates.

In the remainder of this chapter, we will set the foundation for our work on efficiently computing non-elementary quantile functions. The content is general in nature, not overly focusing on any particular distribution—this is remit of the next four chapters. Chapters 3–6 are devoted to the normal, gamma, non-central χ^2 and skew-normal quantile functions. Existing literature for each distribution will be surveyed in the respective chapters. Since this chapter applies to *any* distribution, it should be of value to anyone embarking on the development of quantile function algorithms. Moreover, many of the ideas in Chapters 3–6 have the potential to be extended to other distributions.

[1]At the time of writing, the functions `cospi` and `sinpi` are not actually implemented by the C standard library, even though that they are recommended operators in the IEEE 754-2008 floating-point specification. However, they are both implemented by CUDA, OpenCL and C++ AMP. Furthermore, CUDA has `sincospi`, while OpenCL and C++ AMP both specify `tanpi`. These two functions further simplify the implementation of the Cauchy quantile function.

2.1 Quantile mechanics

The quantile function can be viewed as a special function in its own right. A lot of special functions are, of course, solutions to differential equations or integrals. In [76], quantile functions were characterised by differential equations, and analytic series expansions of the quantile functions for the normal, Student t, beta and gamma distributions were derived. We will review the basic ideas of quantile mechanics now.

Let $F : \mathbf{R} \to (0, 1)$ be the CDF of a continuous random variable. By the inverse function theorem, for $0 < u < 1$, the corresponding quantile function $q(u) = F^{-1}(u)$ must satisfy the ODE

$$\frac{\mathrm{d}q(u)}{\mathrm{d}u} = \frac{1}{f(q(u))}, \tag{2.1}$$

where f is the probability density function (PDF) of the random variable. In [76], (2.1) is called the *first order quantile ODE.* This ODE was used in [56] to derive series solutions of the hyperbolic, variance-gamma and generalised inverse Gaussian quantile functions. The first order quantile ODE can actually be found in [81], where it is solved numerically for the normal, exponential, Cauchy and gamma distributions.

Differentiating (2.1) with respect to u results in

$$\frac{\mathrm{d}^2 q}{\mathrm{d}u^2} = H_f(q) \left(\frac{\mathrm{d}q}{\mathrm{d}u}\right)^2, \tag{2.2}$$

where

$$H_f(x) = -\frac{\mathrm{d}}{\mathrm{d}x} \log f(x), \tag{2.3}$$

and the explicit dependence on u is suppressed for brevity. In [76], (2.2) is called the *second order quantile ODE.* $H(x)$ is a simple rational function for the Pearson system of distributions, so series solutions can be found in these cases. Table 2.1 gives H for some commonly used Pearson distributions.

2.1.1 Variate recycling

The idea of using quantile mechanics to convert samples from one distribution to another was initiated in [74]. Let G be the CDF of an *intermediary* distribution and let q be the quantile function of the *target* distribution (with PDF f). If v is

	Shape parameters	$f(x)$	$H_f(x) = -\frac{\mathrm{d}}{\mathrm{d}x} \log f(x)$
Exponential	N/A	e^{-x}	1
Normal	N/A	$\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$	x
Student t	ν	$\frac{1}{\sqrt{\nu\pi}} \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$	$\frac{(1+\nu)x}{\nu+x^2}$
Gamma	α	$\frac{\gamma(\alpha,x)}{\Gamma(\alpha)}$	$\frac{1+x-\alpha}{x}$
Beta	α, β	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$	$\frac{\alpha-x(\alpha+\beta-2)-1}{(x-1)x}$

Table 2.1: The PDF f and H-function of some commonly used Pearson distributions. Shape parameters can be any positive real number.

a sample from the intermediary distribution,

$$Q(v) = q(G(v)) \tag{2.4}$$

maps v to the target distribution. The sample v is *recycled* into one from the target distribution. Note that this mapping is monotonic, so if v is generated by inversion (using G^{-1}), the underlying properties of the uniform variates are preserved. If Q can be approximated with an approximation that covers a long enough range, branch divergence will be alleviated. In the interests of speed, G^{-1} should preferably be easy to compute.

Let us consider a change of independent variable in the second order quantile ODE, (2.2). Letting $v = G^{-1}(u)$ and writing $Q(v) = q(u)$, some differentiation and simplification gives

$$\frac{\mathrm{d}^2 Q}{\mathrm{d}v^2} + H_g(v)\frac{\mathrm{d}Q}{\mathrm{d}v} = H_f(Q)\left(\frac{\mathrm{d}Q}{\mathrm{d}v}\right)^2, \tag{2.5}$$

where

$$H_g(x) = -\frac{\mathrm{d}}{\mathrm{d}x}\log g(x) \tag{2.6}$$

with g the PDF of the intermediary distribution. In [74], (2.5) is called the *recycling ODE*.

2.2 CDF mechanics

To yield a sufficiently accurate answer, traditional quantile function algorithms based on root finding usually evaluate the CDF F several times. Suppose the successive approximations are

$$x_0, x_1, x_2, \ldots.$$

Then

$$F(x_0), F(x_1), F(x_2), \ldots$$

would be computed. The explicit computation of $F(x_1), F(x_2), \ldots$ could be avoided, by simply calculating the difference between $F(x_i)$ and $F(x_{i+1})$ implicitly. To ease approximation of this quantity, a change of variable could be used. In a sense, this is variate recycling but in "reverse". Taylor polynomials can potentially be employed to get $F(x_{i+1})$ from $F(x_i)$.

Assuming the initial guess for the quantile x_0 is sharp, the corrections δ_i to the successive approximations

$$x_0, x_1, x_2, \ldots$$

will be relatively small and progressively decrease in magnitude. In the second iteration, after $F(x_0)$ is evaluated, $F(x_1) = F(x_0 + \delta_0)$ will be required. It seems wasteful to compute $F(x_1)$ afresh, given the relationship between it and $F(x_0)$. If we expand $F(x)$ about $x = x_0$ as a Taylor series, $F(x_1)$ can be obtained by substituting $\delta_0 = x_1 - x_0$ into the expansion, assuming we are inside the radius of convergence. This process can clearly be repeated to get $F(x_2), F(x_3), \ldots$ implicitly. The use of Taylor series expansions is especially justified for this purpose since we are working locally. Alternatively, adaptive numerical integrators such Runge–Kutta–Fehlberg [26] could be used to go between $F(x_i)$ and $F(x_{i+1})$.

We will now develop an analogue to quantile mechanics. The new machinery has the potential to assist the approximation of CDFs. Let us reconsider the CDF F of a continuous random variable. We have, by definition,

$$\frac{\mathrm{d}F(x)}{\mathrm{d}x} = f(x) \tag{2.7}$$

where, as usual, f is the PDF of the random variable. This ODE can be considered as the *first order CDF ODE.*

2.2.1 Reverse variate recycling

Let q be the quantile function of a *foreground* distribution (with PDF g) and F be the CDF of the *target* distribution (with PDF f). If x is a sample from the target distribution,

$$P(x) = q(F(x)) \tag{2.8}$$

maps x to the foreground distribution. The function q basically acts as a "post filter". An application of CDF of the foreground distribution obviously recovers the CDF of the target distribution. Differentiating (2.8), we have

$$\frac{\mathrm{d}P(x)}{\mathrm{d}x} = \frac{f(x)}{g(P(x))} \tag{2.9}$$

and

$$\frac{\mathrm{d}^2P}{\mathrm{d}x^2} = \frac{\mathrm{d}P}{\mathrm{d}x}\left[H_g(P)\frac{\mathrm{d}P}{\mathrm{d}x} - H_f(x)\right], \tag{2.10}$$

the *first and second order CDF recycling ODEs.*

2.3 Floating-point considerations

Single and double-precision floating-point are the most commonly used data types for computer simulations. We consider only IEEE single- and double-precision[2] floating-point formats in this work, so there are some important simplifications that can be made when computing $q(u)$. Let us fix some notation. Let, respectively,

$$\begin{aligned} \epsilon_s &= 2^{-24} &&\approx 5.96 \times 10^{-8}, \\ \min_s &= 2^{-126} &&\approx 1.18 \times 10^{-38}, \\ \min_s' &= 2^{-149} &&\approx 1.40 \times 10^{-45} \end{aligned} \tag{2.11}$$

be machine epsilon, the smallest positive (normal) number, and the smallest positive subnormal number for IEEE single-precision. Also let

$$\begin{aligned} \epsilon_d &= 2^{-53} &&\approx 1.11 \times 10^{-16}, \\ \min_d &= 2^{-1022} &&\approx 2.23 \times 10^{-308}, \\ \min_d' &= 2^{-1074} &&\approx 4.94 \times 10^{-324} \end{aligned} \tag{2.12}$$

be the equivalent values for IEEE double-precision. Various schools of thought exist as to how accurate quantile functions—and random numbers in general—need to be. Some may take the position that the number of significant figures obtained

[2]Talking about single-precision may seem quaint. However, single-precision computations are appreciably faster than double-precision on modern parallel architectures. Single-precision is consequently coming back into fashion.

is perhaps unimportant, due to the numerous uncertainties elsewhere in problems. We will instead aim to eliminate as much uncertainty as possible from our numerical considerations and characterise the associated errors.

2.3.1 Uniform input

Pseudo- and quasi-RNGs work with unsigned integer data types—typically 32- or, to a lesser extent, 64-bit integers. Most RNGs, therefore, output random integral values between 0 and either 2^{32} or 2^{64}. For example, the Mersenne Twister MT19937 RNG [52] outputs 32-bit integers. A floating-point multiplication by 2^{-32} or 2^{-64} gives a random uniform number between 0 and 1 that can be fed into $q(u)$.

Since we are focusing on simulation applications, we will assume a 32- or 64-bit RNG is the source of randomness for non-uniform variate generation. The smallest (non-zero) uniform number we encounter is thus $2^{-32} \approx 2.33 \times 10^{-10}$ or $2^{-64} \approx 5.42 \times 10^{-20}$. It should be noted that these numbers are larger than both $\min_s$ and $\min_d$. The largest uniform number (excluding 1) produced by a 32- and 64-bit RNG is $1 - 2^{-32}$ and $1 - 2^{-64}$ respectively. However, machine precision can affect these limits. If single-precision numbers are sought, the largest number will be limited to $1 - \epsilon_s$ in either case. If double-precision numbers are requested, the largest number will be $1 - 2^{-32}$ or $1 - \epsilon_d$ depending on the RNG.

2.3.2 Non-uniform output

On the other side of the coin, there are occasions where the true result of $q(u)$ is outside the range of representable floating-point numbers. The most obvious case is overflow. Here it is logical to return infinity. The other case, which is not so clear-cut, is what to do in the event of gradual underflow. Can we simply ignore this and return zero, or should we return a subnormal number that will be imprecise to some degree?

For example, gradual underflow might occur for the gamma distribution with a very small shape parameter α. Let us consider the case of $\alpha = 1/100$. Figure 2.1 shows the gamma quantile function for this particular α. The gamma quantile at $u = 37/100$ is $3.741497614 \times 10^{-44}$ to ten significant figures. This value is less than $\min_s$, but greater than $\min'_s$. Therefore, the gamma quantile at $u = 37/100$ cannot be fully represented in single-precision. The nearest subnormal number is 3.783506×10^{-44}, which has a relative error of 0.0112277. This begs the question of how to measure the accuracy of a subnormal result.

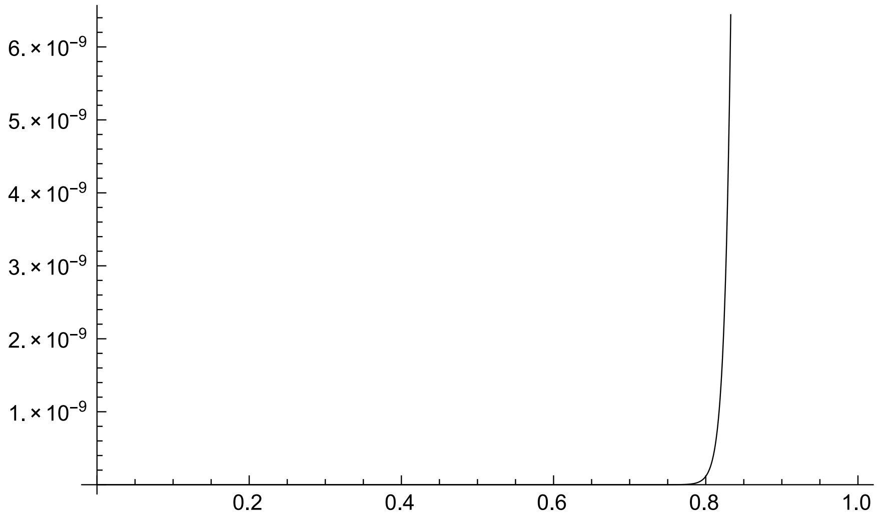


Figure 2.1: The gamma quantile function for $\alpha = 1/100$.

The finite range of floating-point numbers means that the uniform input range of concern to us can sometimes be shorter than usual. For example, if $\alpha = 1/100$, the gamma CDF

$$\frac{1}{\Gamma(\alpha)} \int_0^x t^{\alpha-1} e^{-t} \, \mathrm{d}t$$

evaluated at $x = \min'_s$ is 0.3580441447 to ten significant figures, so zero can (and should) be returned for all u less than this value when working in single-precision.

2.4 Target computer architecture

For an algorithm to be of any value it arguably has to eventually run on a computer for some greater good. While there can be advantages during algorithm design in not making any assumptions on the target computer, e.g. future applicability, a balance has to be struck. For instance, the IEEE floating-point number format has been—and almost certainly for the foreseeable future is—the prevailing data type for real numbers on computers, so the assumption of exact real numbers is impractical, to say the least. It would also be remiss not to consider trends in computer architecture.

Many-core parallel architectures are a growing trend that is especially pertinent to computer simulations. Accounting for this during the design of quantile function algorithms is, therefore, a relatively sound assumption to make. The upside of this is that the algorithm running times should be reduced on these parallel computers.

Where possible, our quantile function algorithms will address the issue of branch divergence.

2.5 Evaluation and validation

This section details how we will evaluate the speed and precision of our quantile function algorithms.

2.5.1 Speed

We will consider only the timing of code directly related to computation of quantile functions. Anything else, such as uniform random number generation, is not our concern.

The speed of our algorithms will be evaluated by generating 10^7 random inputs and measuring the time to apply the quantile function to them. This experiment will be repeated, with fresh random numbers, 100 times to get an average. Nvidia's CUDA Random Number Generation library [61] will be used to generate the uniformly distributed random numbers. The numbers will be pseudo-random and generated by MRG32k3a [46], and 19,532 blocks of 512 threads will be used, with each thread processing a single uniform input. Our algorithms will be accurately timed using CUDA events as per the CUDA C Programming Guide [59, Section 3.2.5.6.].

2.5.2 Precision

There are several error measures we could use. However, since we are working with floating-point numbers, it makes sense to restrict our attention to relative error measures. The most obvious relative error measure is probably

$$E_1 = \max_u \left| \frac{\tilde{q}(u)}{q(u)} - 1 \right|, \tag{2.13}$$

where u is a uniform input and $\tilde{q}(u)$ denotes the approximation of $q(u)$. This is a *forward* relative error measure. If $\tilde{q}(u)$ and $q(u)$ are both ∞, then this measure is taken as zero. This is the same if $\tilde{q}(u)$ and $q(u)$ are both less than $\min_s$ or $\min_d$ (effectively regarding all subnormal numbers as zero). We also consider the "roundtrip"/backward relative error measure

$$E_2 = \max_u \left| \frac{F(\tilde{q}(u))}{u} - 1 \right|, \tag{2.14}$$

which we define as zero if u is less than $F(\min_s)$ or $F(\min_d)$ and $\tilde{q}(u)$ is less than either $\min_s$ or $\min_d$.

The error measure E_2 is arguably more independent than E_1, because it does not depend on another algorithm and implementation of $q(u)$. For well-trodden quantile functions, this is less of an issue. However, for distributions that have little or no existing quantile function algorithms, E_2 should be the first port of call. As a last resort, assuming that the PDF can be evaluated, standard quadrature methods can be used to compute F. If the density of the distribution is not mathematically expressible, which is the case for the CGMY[3] [18] and stable distributions (see, e.g., [58]), another route must be found. If the characteristic function is available, an inverse Fourier transformation yields the PDF. Alternatively, a direct path from the characteristic function to the quantile function could be used [75].

The smallest possible relative error achievable is dependent on machine epsilon. It should be noted that machine epsilon accuracy is usually overkill in practical simulation applications. For example, extreme accuracy can be traded off against computational speed in many Monte Carlo applications in finance.[4]

[3]Carr–Geman–Madan–Yor

[4]Senior quantitative analyst at a Tier 1 investment bank, private communication.

Chapter 3

Normal distribution

The normal (Gaussian) PDF is

$$f_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{3.1}$$

where $\mu \in \mathbf{R}$ is the mean parameter and $\sigma \in \mathbf{R}_{>0}$ is the standard deviation parameter. If $Z \sim N(0,1)$ then $\sigma Z + \mu \sim N(\mu, \sigma)$. Without loss of generality we can set $\mu = 0$ and $\sigma = 1$, which leads to the standard normal distribution. We will thus take the normal PDF to be the familiar

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}. \tag{3.2}$$

Figure 3.1 shows the normal PDF. By definition, the normal CDF is

$$\begin{aligned} \Phi(x) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}t^2} \, \mathrm{d}t & (3.3) \\ &= \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right] & (3.4) \\ &= \frac{1}{2}\operatorname{erfc}\left(-\frac{x}{\sqrt{2}}\right), & (3.5) \end{aligned}$$

where $\operatorname{erf}(x)$ is the error function and $\operatorname{erfc}(x) = 1 - \operatorname{erf}(x)$ is the *complementary* error function. The normal quantile function is therefore

$$\begin{aligned} \Phi^{-1}(u) &= \sqrt{2}\, \operatorname{erf}^{-1}(2u - 1) & (3.6) \\ &= -\sqrt{2}\, \operatorname{erfc}^{-1}(2u). & (3.7) \end{aligned}$$

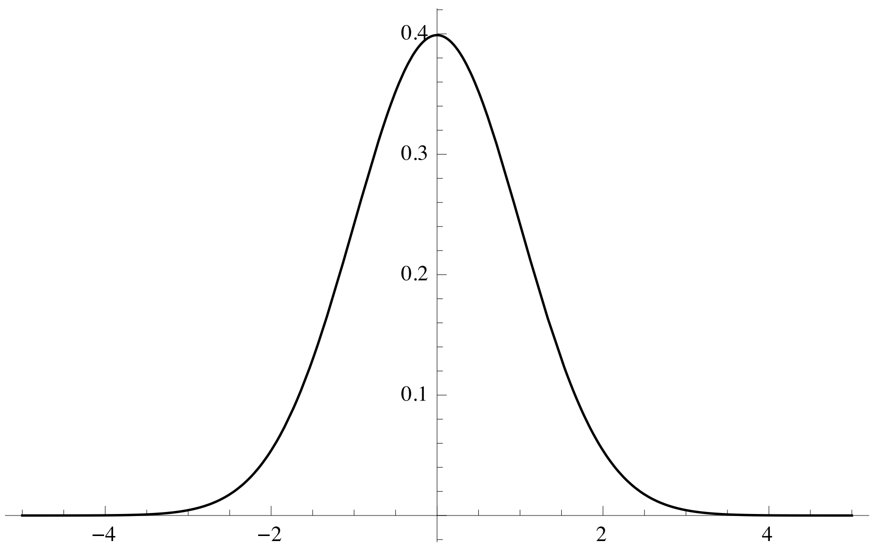


Figure 3.1: The normal PDF ϕ.

The log-normal (see, e.g., [38, Chapter 14]) and Johnson system [37] of distributions are related to the normal distribution. The quantile function of these distributions can be obtained by algebraic transformations of $\Phi^{-1}(u)$, e.g. the log-normal quantile of u is $\exp(\mu + \sigma\Phi^{-1}(u))$. Tractable skewed variants of the normal distribution can be obtained simply, by the quadratic transformations given in [72]. Azzalini's skewed normal distribution, which seems to have been adopted as the definitive skew-normal distribution, is treated in Chapter 6.

For some time, the venerable Box–Muller method [15]—or the polar variant [50] of it—was the default choice for generating normal variates, because it is very computationally efficient. However, nowadays, inversion of the normal CDF is often required when producing normal variates. Modern simulation methods such as quasi-Monte Carlo and copula methods call for direct inversion of CDFs (see Chapter 1 and, e.g., [31, 47, 43]). The normal quantile function is a standard inclusion in most numerical libraries, including Intel's Math Kernel Library (MKL), NAG's numerical libraries, Boost's C++ Math Toolkit, and Nvidia's CUDA Math Library. Standalone open-source implementations of Φ^{-1} are also freely available (e.g., [83] or [2]).[1] The normal distribution does not possess a shape parameter. This means that a one-off polynomial or rational minimax approximation can be

[1]The normal quantile function $\Phi^{-1}(u)$ is often given in the form $\sqrt{2}\,\mathrm{erf}^{-1}(2u-1)$. It should be noted that direct implementation of this, using the inverse error function, will result in loss of precision for small u. One should instead use the equivalent form $-\sqrt{2}\,\mathrm{erfc}^{-1}(2u)$ or an algorithm specially designed for computing Φ^{-1}. This matter is explained in more detail in Section 3.1.

constructed and verified offline, so most library and standalone implementations of Φ^{-1} use such approximations.

3.1 Existing methods

There are several well-known approximations for Φ^{-1}. They all use a rational approximation for $u \in [0+b, 1-b]$, where $b > 0$ is the tail *breakpoint.* The other values in $(0, 1)$ are covered by at least one additional polynomial or rational approximation, which are in terms of a computationally expensive change of variable. The reason for this two-pronged approach is because Φ^{-1} has rather a split personality. It is slowly varying in the central region where u is between about 0.1 and 0.9, and then diverges to infinity as $u \to 0^+$ or $u \to 1_-$. This is shown in Figure 3.2. The values of b for existing algorithms are as follows:

- $b = 0.08$ for Moro [55],
- $b = 0.075$ for Wichura (Algorithm AS 241) [83],
- $b = 0.02425$ for Acklam [2].

The two-pronged approach is absolutely fine for CPUs, where branch prediction means that virtually no performance penalty is incurred for branching. Things are quite different for GPUs, which are sensitive to branching. Suppose Wichura's algorithm is run on a Nvidia Fermi or Kepler GPU. Threads on such a GPU execute in groups of 32. Nvidia call these groups *warps.* Assuming uniformly distributed input, the probability that all threads in a warp use the central approximation is $0.85^{32} \approx 0.006$. The equivalent probability for the tail approximation is negligible, so there is a 99% likelihood of branch divergence occurring. When this happens each thread is forced to compute both the central and tail rational approximations, along with a logarithm and square root. A similar calculation shows that with Acklam's algorithm there is a 80% probability of branch divergence. Branch divergence will be minimised if b for a particular approximation is sufficiently small.

New approximations of special functions that deliberately address branch divergence on GPUs can be found in the literature. An earlier version of this study, [71], first gave branch-free approximations for Φ^{-1}. M. Giles has given single and double-precision approximations for erf^{-1} [30] and advocates its use for computing Φ^{-1} [30, p. 1]. The inverse error function can be found in the Intel Math Kernel Library (MKL) and the Nvidia CUDA Math Library, implemented as `erfinv`. However,

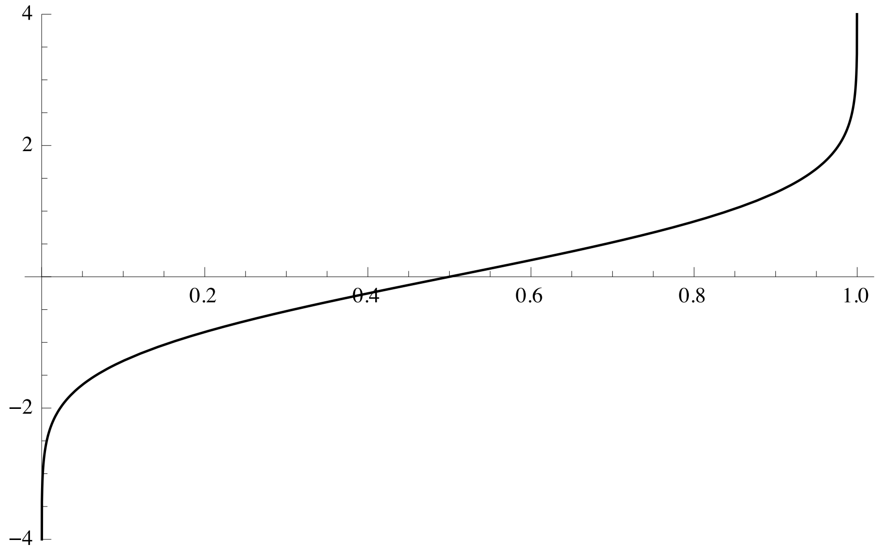


Figure 3.2: The normal quantile function Φ^{-1}.

both libraries also provide the inverse complementary error function `erfcinv`, for good reason. What is the point of `erfcinv`, since $\operatorname{erfc}^{-1}(x) = \operatorname{erf}^{-1}(1-x)$? Suppose $\Phi^{-1}(u)$ is implemented in the form $\sqrt{2}\operatorname{erf}^{-1}(2u-1)$. Working in floating-point arithmetic, if u is less than machine epsilon, then $2u-1=-1$, so $\operatorname{erf}^{-1}(2u-1)$ will be evaluated as $-\infty$ instead of a finite negative value. For u slightly larger than machine epsilon, there will be a partial loss of precision, because the density of floating-point numbers is far less around -1 than zero. One should instead implement $\Phi^{-1}(u)$ in the form $-\sqrt{2}\operatorname{erfc}^{-1}(2u)$ to maintain precision for low u, even though it is generally slower than `erfinv`. CUDA 5.0 onwards includes the normal quantile function `normcdfinv`, which is indeed based on `erfcinv`.

Wichura's and Acklam's algorithms for computing Φ^{-1} will now be analysed in detail. The Beasley–Springer–Moro algorithm will also be examined. All three algorithms actually share the same DNA, but Wichura's is slightly more evolved.

3.1.1 Wichura's algorithm AS 241

Wichura published an algorithm for Φ^{-1} in 1988 and it is currently used in R and the NAG Library. Wichura gave two versions of his algorithm in [83]: PPND7, accurate to around seven significant figures; and PPND16, accurate to around 16 significant figures. These map very nicely to single and double-precision floating-point implementations. Algorithm 1 gives the pseudocode for Wichura's algorithm.

$R_1(t)$ is a rational minimax approximation to

Algorithm 1 Wichura's algorithm (PPND)

```
Input: $u \in (0, 1)$
Output: $\Phi^{-1}(u)$
  if $|u - 0.5| \leqslant 0.425$ then
      return $(u - 0.5)R_1((u - 0.5)^2)$                ▷ $R_1(t) \approx \frac{\Phi^{-1}(0.5+\sqrt{t})}{\sqrt{t}}$
  else
      $r = \sqrt{-\log[\min(u, 1 - u)]}$
      if $r \leqslant 5$ then
          return $\mathrm{sgn}(u - 0.5)R_2(r)$           ▷ $R_2(t) \approx -\Phi^{-1}(\exp(-t^2))$
      else
          return $\mathrm{sgn}(u - 0.5)R_3(r)$           ▷ $R_3(t) \approx -\Phi^{-1}(\exp(-t^2))$
      end if
  end if
```

$$\frac{\Phi^{-1}(0.5 + \sqrt{t})}{\sqrt{t}} \tag{3.8}$$

over $0 \leqslant t \leqslant 0.425^2$. Figure 3.3 shows that this expression is relatively simple for the relevant range. Indeed PPND7 uses a rational approximation of degree 3, while one of degree 7 is used by PPND16. $R_2(t)$ and $R_3(t)$ are rational minimax approximations to

$$-\Phi^{-1}(\exp(-t^2)) \tag{3.9}$$

for $1.6 \leqslant t \leqslant 5$ and $5 \leqslant t \leqslant 27$ respectively. Figure 3.3 shows that this expression is virtually linear. The ranges correspond to $\exp(-27^2) \leqslant u \leqslant \exp(-5^2)$ and $\exp(-5^2) \leqslant u \leqslant \exp(-1.6^2)$. Note that $\exp(-27^2) \approx 2.51 \times 10^{-317} < \min_d$, the smallest *normal* positive double-precision floating-point number. PPND7 uses a rational approximation of degree $(3, 2)$ for R_2 and R_3, while a pair of degree 7 are used by PPND16. All of the polynomial coefficients that Wichura's algorithm uses are positive, which is good for numerical stability.

3.1.2 Acklam's algorithm

Acklam's algorithm [2], which was first published on the Web in 2003, is another respected approximation and is recommended by Jäckel in [36]. Acklam took a slightly different approach to Wichura. Like Wichura's algorithm, Acklam's one uses a rational minimax approximation for a central region. While Wichura uses

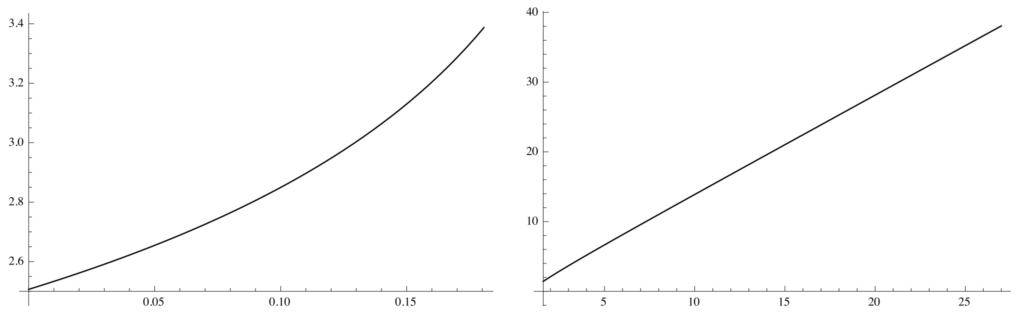


Figure 3.3: Plot of $\Phi^{-1}(0.5+\sqrt{t})/\sqrt{t}$ for $0 \leqslant t \leqslant 0.425^2$ (*left*) and $-\Phi^{-1}(\exp(-t^2))$ for $1.6 \leqslant t \leqslant 27$ (*right*).

two rational approximations for the tail, Acklam only uses one. Algorithm 2 gives the pseudocode for Acklam's algorithm. R_1 is same as Wichura's but of degree 5

Algorithm 2 Acklam's algorithm

```
Input: $u \in (0,1)$
Output: $\Phi^{-1}(u)$
  if $|u - 0.5| \leqslant 0.47575$ then
      return $(u - 0.5)R_1((u - 0.5)^2)$                  $\triangleright R_1(t) \approx \frac{\Phi^{-1}(0.5+\sqrt{t})}{\sqrt{t}}$
  else
      $q = \sqrt{-2\log[\min(u, 1 - u)]}$
      return $\operatorname{sgn}(u - 0.5)R_2'(q)$             $\triangleright R_2'(t) \approx -\Phi^{-1}(\exp(-t^2/2))$
  end if
```

and over $0 \leqslant t \leqslant 0.47575^2$. $R_2'(t)$ is a rational approximation to

$$-\Phi^{-1}(\exp(-t^2/2)) \tag{3.10}$$

of degree $(5, 4)$. This approximation covers $2^{-1073} \leqslant u \leqslant 0.02425$. Acklam's polynomial coefficients are a mixture of positive and negative ones. The degrees of the rational approximations used by Acklam suggest an accuracy between Wichura's 7 and 16 significant figures. Acklam quotes that the relative error of his approximations is at most 1.15×10^{-9} in magnitude. This has been independently verified in [70] and observed by a number of people who have also implemented Acklam's algorithm. Moreover, Acklam gives a way to refine initial approximations to full machine precision. It involves a single iteration of Halley's rational method, so assumes access to a sufficiently accurate Φ or erfc library function. In [70], Shaw observed that the use of Halley's method actually gave no improvement over a

Newton based refinement.

3.1.3 Beasley–Springer–Moro algorithm

The Beasley–Springer–Moro algorithm [55] is sometimes suggested for approximating Φ^{-1} (see, e.g., [31]). Moro's algorithm is a modification to Beasley and Springer's one from [9], improving the accuracy in the tails. Algorithm 3 gives the pseudocode for the Beasley–Springer–Moro algorithm. R_1 is the same as Wichura's

Algorithm 3 Beasley–Springer–Moro algorithm

```
Input: $u \in (0,1)$
Output: $\Phi^{-1}(u)$
  if $|u - 0.5| \leqslant 0.42$ then
      return $(u - 0.5)R_1((u - 0.5)^2)$            ▷ $R_1(t) \approx \frac{\Phi^{-1}(0.5+\sqrt{t})}{\sqrt{t}}$
  else
      $r = \log(-\log[\min(u, 1 - u)])$
      return $\operatorname{sgn}(u - 0.5)R_2''(r)$          ▷ $R_2''(t) \approx -\Phi^{-1}(\exp(-\exp(t)))$
  end if
```

but of degree $(3, 4)$ and over $0 \leqslant t \leqslant 0.42^2$. $R_2''(t)$ is a polynomial approximation to

$$-\Phi^{-1}(\exp(-\exp(t))) \tag{3.11}$$

of degree 8. In Beasley and Springer's original algorithm, if $|u - 0.5| > 0.42$, $r = \sqrt{-\log[\min(u, 1 - u)]}$ is computed and then a rational approximation to

$$-\Phi^{-1}(\exp(-r^2)) \tag{3.12}$$

of degree $(3, 2)$ is used. Unfortunately, both algorithms are optimised for *absolute* error. Moro's algorithm is not accurate enough for full double-precision calculations. It is also not quite sufficient for full single-precision. Beasley and Springer published their algorithm in 1977, nearly a decade before Wichura. (Wichura did reference the 1977 paper.)

3.1.4 Shaw's branch-free algorithm for GPUs

A branch-free algorithm for the normal quantile function was given by Shaw in [71]. A change of variable means that only one rational approximation appears in the algorithm. Let us consider

$$v = -\log[2(1 - u)] \tag{3.13}$$

for $1/2 < u < 1$. This is the quantile of the Laplace double-exponential distribution for $1/2 < u < 1$. The mapping

$$Q(v) = \Phi^{-1}\left(1 - e^{-v}/2\right) \tag{3.14}$$

transforms positive double-exponential variates to normal ones. Odd symmetry of $\Phi^{-1}(u)$ allows $0 < u \leqslant 1/2$ to be managed simply. Figure 3.4 shows a plot of $Q(v)$ for $0 \leqslant v \leqslant 44$. This range is equivalent to $0.5 \leqslant u \leqslant 1 - e^{-44}/2 \approx 1 - 3.89 \times 10^{-20}$. A single minimax rational approximation to Q suitable for single and double-precision Monte Carlo was generated in Mathematica 8 [86]. The single-precision rational approximation is of degree 6, while the double-precision one has degree 13. Both rational approximations are over $0 \leqslant v \leqslant 22$, reaching $u = 2^{-32}$. A branch-free normal quantile function approximation based on $\Phi^{-1}(u) = \operatorname{sgn}(u - 1/2) \times Q(v)$ where

$$v = \begin{cases} -\log(2u) & \text{if } 0 < u \leqslant 1/2 \\ -\log(2(1-u)) & \text{if } 1/2 < u < 1 \end{cases} \tag{3.15}$$

was implemented in CUDA. For clarity, Algorithm 4 gives the pseudocode for this. Note that only one log is used in this solution. Code implementing this is given

Algorithm 4 Shaw's branch-free algorithm

```
Input: $u \in (0,1)$
Output: $\Phi^{-1}(u)$
  if $u > \frac{1}{2}$ then
      $u \leftarrow 1 - u$
  end if
  $v \leftarrow -\log(2u)$                          ▷ double-sided exponential quantile
  return $\operatorname{sgn}\left(u - \frac{1}{2}\right) Q(v)$          ▷ $Q$ rational approximation
```

in CUDA device function form in Appendix B.1. Like Wichura's algorithm, all of the coefficients that appear in Shaw's algorithm are positive.

The CUDA implementation of Shaw's algorithm was significantly faster on the GPU than any other algorithm at its time of creation, in both single- and double-precision. This included an implementation based on Nvidia's built-in erfc^{-1} function, `erfcinv`, which was around three times slower. CUDA was at version 4.0 at the time. Nvidia's `erfcinv`(x) function in CUDA 4.0 returns: `erfinv`$(1 - x)$ for $x \geqslant 0.0625$; and the result of a rational approximation in terms of $[-\log(x)]^{-1/2}$ for other positive x (based on [12]). A pre-print [73] demonstrating the superiority

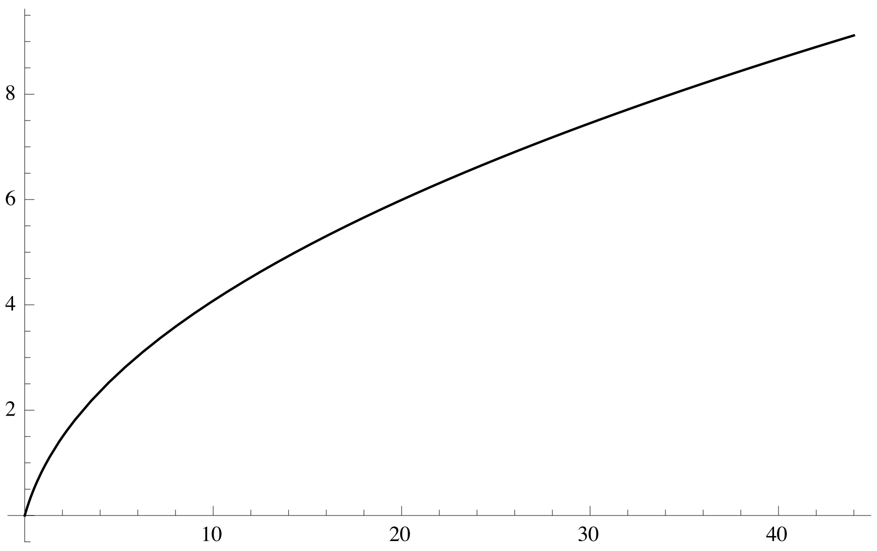


Figure 3.4: The exponential to normal transformation $Q(v) = \Phi^{-1}\left(1 - \frac{e^{-v}}{2}\right)$ for $0 \leqslant v \leqslant 44$.

of Shaw's algorithm was made available to Nvidia and by CUDA 5.5, the performance of Nvidia's erfc^{-1} function had somehow caught up with Shaw's algorithm. In response, this author blended two concepts to create an even faster algorithm.

3.2 Ideas for faster performance

We will now give details of ideas to accelerate normal quantile computation. The branch-free solution for the normal quantile function by Shaw was used as a starting point. In this section we will see the effect of applying Giles' low-probability branch divergence idea and ideas by this author. We will see novel use of `rsqrt` (reciprocal square root) and warp vote functions, combined with Giles' idea, to create the fastest known GPU normal quantile function implementation.

3.2.1 Low-probability branch divergence

Giles' [30] approach to approximating erf^{-1} is based on a branching formula with a low branch divergence probability. Nvidia's built-in CUDA inverse error function is modelled on Giles' version because of this. Despite the shortcomings of Giles' erf^{-1} algorithm for computing the normal quantile function, the idea of allowing one or more "edge case" branches, which are seldom expected to be evaluated, is an interesting one. A single, main branch can then be optimised to service the majority of cases. Suppose the probability that any group of threads executing in lockstep *all* take a common branch is, say, 0.999. Then on a CUDA GPU the probability of warp divergence not occuring is $0.999^{32} \approx 0.968$. This level branch divergence can usually be tolerated. This, of course, has to be balanced with the computational costs of each branch. A disadvantage of this low-probability branch divergence scheme is that portability (between different GPU vendors) and future proofness (Nvidia may increase the size of a warp) is somewhat sacrificed.

Shaw's branch-free solution uses a single rational approximation over the interval of interest. It covers $2^{-32} \leqslant u \leqslant 1$ by transforming two-sided exponential variates in the range $[-22, 22]$. The single-precision version uses a rational approximation of degree 6, while the double-precision one has degree 13. Most exponential variates will be in a much shorter range, most of the time; $(-8, 8)$, for example, covers $e^{-8}/2 < u < 1 - e^{-8}/2 \approx 0.999832$. Bearing in mind, because of symmetry, only $[0, 22]$ has to be considered, a shorter rational approximation can cover $[0, 8)$ and another can manage $[8, 22]$. Of course, this does introduce the possibility of branch divergence, but the probability is relatively small—most input will still be handled by the same rational approximation, most of the time. On a CUDA GPU with a warp size of 32, the probability of *not* diverging is about $0.999832^{32} \approx 0.994647$. Even branching at $v = 6$ results in an acceptable level of branch divergence: 0.038908.

A single-precision rational approximation of degree $(4, 5)$ was found for $[0, 7]$ and one of degree $(3, 4)$ to cover $[7, 22]$. Using these in a branched approximation scheme actually had a slight adverse effect on speed. It turned out that the overhead of simply evaluating an `if` statement negated the advantage gained by splitting $[0, 22]$. (Other sensible splits were tested.) However, branching allows us to cover the deep lower tail without compromising performance significantly. The smallest normal positive single-precision value is $\min_s \approx 1.18 \times 10^{-38}$, which requires an approximation over $[0, 87]$ since $e^{-87}/2 \approx 8.23 \times 10^{-39}$. A single-precision rational approximation of degree $(3, 4)$ was found for $[22, 87]$. Appendix B.2 gives the CUDA implementation of the branched algorithm for single-precision.

A double-precision rational approximation of degree 10 was found for $[0, 8]$ and one of degree $(6, 7)$ to cover $[8, 22]$. The smallest normal double-precision value is $\min_d \approx 2.23 \times 10^{-308}$, which requires an approximation over $[0, 708]$ since $e^{-708}/2 \approx 1.65 \times 10^{-308}$. A rational approximation of degree 6 was found for $[22, 44]$. Up to $v = 44$ covers 64-bit RNGs ($2^{-64} \approx 5.42 \times 10^{-20}$ and $e^{-44}/2 \approx 3.89 \times 10^{-20}$). Up to $v = 708$ is possible with several approximations, but delegating to the Nvidia algorithm turned out to be better for $44 \leqslant v \leqslant 708$. Appendix B.3 gives the CUDA implementation of the branched algorithm for double-precision.

3.2.2 Student t distribution inspired variate recycling

The branch-free solution for the normal quantile function takes a uniform variate, transforms it into a two-sided exponential variate by inversion using (3.15), and uses a rational approximation to convert it to a normal variate. An analysis of the CUDA implementation found that the call to `log` dominates the running time. This lead to alternative intermediary distributions to be considered. Given the relationship the t-distribution has with the normal distribution, it is a natural candidate. Obviously, a t-distribution with a large degree of freedom is preferable, but its quantile function has to be efficient to compute. Closed-form expressions of this for degrees of freedom $\nu = 1, 2, 4$ are known (see, e.g. [69]). The quantile function of the t-distribution with two degrees of freedom

$$\frac{2u - 1}{\sqrt{2u(1-u)}} \tag{3.16}$$

is exceedingly simple, involving no transcendental functions. An added bonus is that the quantile function involves the *reciprocal* of the square root. This is very pertinent. The primary function of GPUs is the swift manipulation of computer graphics. Real-time lighting effects are commonplace in video games, requiring the normalisation of millions of vectors per second by their Euclidean norm. GPUs are therefore very fast at computing the reciprocal (inverse) square root, i.e. $x^{-\frac{1}{2}}$. Most math libraries, including the CUDA one [60], have a `rsqrt(x)` function. In fact, on Nvidia GPUs, $\sqrt{x}$ is implemented by taking the reciprocal of `rsqrt(x)`. Since the t-distribution with $\nu = 2$ is one of the fattest tailed members of the family, we should not expect the penetration of the tails of the normal distribution to be easy (cf. using an exponential intermediary distribution).

For even more efficient normal quantile function computation, let us consider a rescaling. Let t_2 be the t-distribution with location $\mu = 0$, *scale* $\sigma = \frac{1}{\sqrt{2}}$ and degrees of freedom $\nu = 2$. The quantile function of this distribution is simply

$$\frac{u - \frac{1}{2}}{\sqrt{u(1-u)}}. \tag{3.17}$$

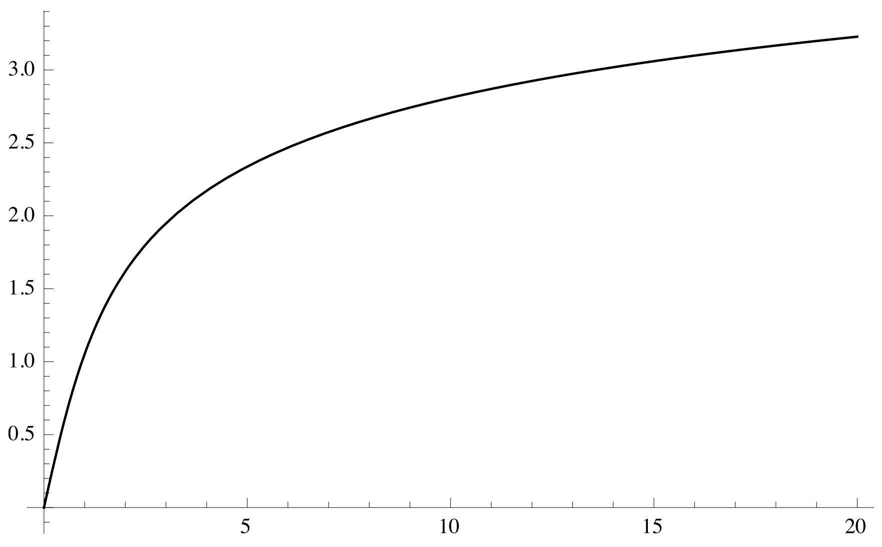


Figure 3.5: The t_2 to normal transformation $Q_t(v) = \Phi^{-1}\left[\frac{1}{2}\left(1+\frac{v}{\sqrt{1+v^2}}\right)\right]$ for $0 \leqslant v \leqslant 20$.

If v is a sample from the distribution,

$$Q_t(v) = \Phi^{-1}\left[\frac{1}{2}\left(1+\frac{v}{\sqrt{1+v^2}}\right)\right] \tag{3.18}$$

maps v to the normal distribution. Figure 3.5 shows a plot of $Q_t(v)$ for $0 \leqslant v \leqslant 20$. As expected, the reach of the t_2 distribution into the normal tail is nothing to write home about. However, $Q_t(15.5) \approx 3.07933$, for which $u \approx 0.998963$, giving a low branch divergence probability. Branch divergence would still be relatively low. The remaining can be managed by the improved exponential transform machinery. The result is a hybrid algorithm. For double-precision, a rational approximation of degree $(14, 15)$ for $Q_t(v)$ was found for $0 \leqslant v \leqslant 15.5$.

We can actually do better and make more economical approximations. Let us consider

$$\Psi(u) = \frac{\Phi^{-1}(u)}{u - \frac{1}{2}}. \tag{3.19}$$

Let $v = \tilde{q}(u)$, where $\tilde{q}$ is any invertible function and write

$$\tilde{Q}(v) = \Psi(\tilde{q}^{-1}(v)) = \Psi(u). \tag{3.20}$$

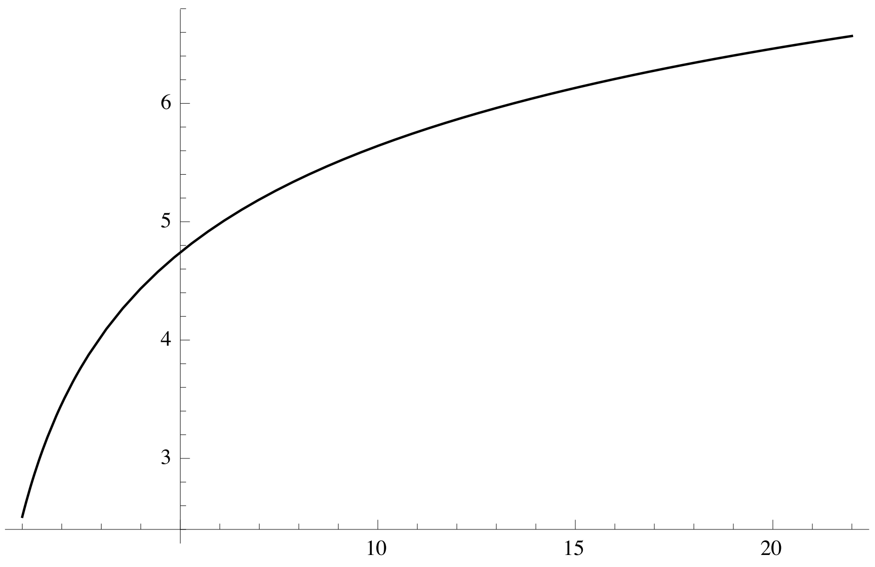


Figure 3.6: The $\tilde{Q}$ transformation on $[1, 22]$.

If

$$\tilde{q}(u) = \frac{1}{2\sqrt{u(1-u)}} \tag{3.21}$$

then

$$\tilde{q}^{-1}(v) = \frac{1}{2}\left(1 + \sqrt{1 - \frac{1}{v^2}}\right) \tag{3.22}$$

for $u \geqslant 1/2$. The function $\tilde{Q}(v)$ is slightly easier to approximate than $Q_t(v)$, as shown in Figure 3.6. For single-precision, a rational approximation of degree 5 for $\tilde{Q}(v)$ was found for $1 \leqslant v \leqslant 21$. For double-precision, a rational approximation of degree 11 was found for $1 \leqslant v \leqslant 22$. These ranges are very good, providing lower probabilities of branch divergence, since $\tilde{q}^{-1}(21) \approx 0.999433$ and $\tilde{q}^{-1}(22) \approx 0.999483$. The aforementioned change of variable will be referred to as $\tilde{t}_2$ henceforth.

Hybrid approximation for single-precision

Algorithm 5 gives the pseudocode for our new normal quantile algorithm based on an hybrid approximation, which uses the $\tilde{t}_2$ and exponential changes of variable. Appendix B.4 gives the single-precision CUDA implementation of this.

The single-precision branch-free and branched implementations could not use CUDA's fast, intrinsic `__logf` function, because it results in poor relative error near $u = 0.5$. There were no such problems elsewhere, around zero and one in particular.

Algorithm 5 Hybrid approximation of the normal quantile for single-precision

```
Input: $u \in (0, 1)$
Output: $\Phi^{-1}(u)$
  $v \leftarrow \frac{1}{2}\sqrt{u(1-u)}^{-1}$                    ▷ $\tilde{q}(u)$
  if $v \leqslant 21$ then                                        ▷ $v > 21$ cannot be handled
      return $\left(u - \frac{1}{2}\right)\tilde{Q}(v)$            ▷ $\tilde{Q}$ rational approximation
  else
      if $u > \frac{1}{2}$ then
          $u \leftarrow 1 - u$
      end if
      $v \leftarrow -\log(2u)$                                     ▷ double-sided exponential quantile
      return $\operatorname{sgn}\left(u - \frac{1}{2}\right) Q(v)$ ▷ $Q$ rational approximation
  end if
```

A side benefit of recycling $\tilde{t}_2$ variates is that the intrinsic function can be used in the hybrid implementation.

Hybrid approximation for double-precision

All CUDA-enabled GPUs that support double-precision floating-point numbers also allow the threads within a warp to cooperate. Specifically, the 32 threads in a warp can take a binary vote (see [59, Section B. 13.]). In this scheme, each thread simultaneously casts their yes/no (`true`/`false`) vote and then is notified of everyone else's vote. The threads can alternatively receive a summary of the votes in the form of logical AND or inclusive OR of them.

This warp voting functionality can be exploited in the hybrid solution to further alleviate branch divergence. We have the fast $\tilde{t}_2$ transformation that is suitable for most warps for most of the time. We also have the comparatively slow exponential transformation that is suitable for all warps. In the (rare) event of a thread in a warp needing to use this transformation because the $\tilde{t}_2$ transformation is unsuitable, branch divergence occurs, which means that two sets of quantile and rational functions are computed. Only one set per warp is actually required: if *any* of the threads in the warp strictly needs to use the exponential transformation, force every other thread to also use the transformation (irrespective of their $\tilde{t}_2$ eligibility); otherwise, *all* of the threads can just use the $\tilde{t}_2$ transformation. This all-or-nothing strategy can be implemented using warp voting. The relevant CUDA code is `__all(v < X)`, where `X` is upper limit of the $\tilde{t}_2$ range, i.e. `21` or `22` depending on single or double-precision. The `__all` function here returns `true` if and only if all `v` in the warp are less than X. The threads in a warp are executing in lockstep anyway, so there are no synchronisation issues when taking a warp vote. The equivalent of CUDA warp voting is described in the OpenCL 2.0 specification.

The pseudocode for our double-precision hybrid normal quantile algorithm is essentially the same as Algorithm 5, but '$v \leqslant 21$' is substituted for 'all v in the warp are $\leqslant 22$'. Appendix B.5 gives the CUDA implementation of this.

3.3 Performance and accuracy on the GPU

We benchmarked the GPU performance of the new algorithms against the well-established candidates. This was done in accordance with Section 2.5. We also considered the optimised normal quantile function in Nvidia's CUDA Math Library, `normcdfinv`. Both single and double-precision variants of each algorithm were evaluated where possible. Unlike Wichura, neither Moro nor Acklam give single-precision coefficients for their approximations, so they only feature in our double-precision tests.

Faithful CUDA GPU implementations of the algorithms were written by this author. Wichura's original Fortran implementation of his algorithm was ported to CUDA. The implementation of Acklam's algorithm was based on the pseudocode on his website. CUDA's built-in erfc function was used for the Halley refinement suggested by Acklam. Moro's algorithm was implemented using the pseudocode in [31]. All of the code written was optimised as far as possible for the GPU architecture, e.g. divisions were factored out of conditional statements.

The performance of the algorithms was evaluated on three high-end Nvidia GPUs:

- GeForce GTX Titan;
- Tesla K80;
- Tesla C2050.

The C2050 is based on Nvidia's previous generation Fermi architecture, while the other two are Kepler GPUs. We used CUDA 7.0 software, the most recent production release at the time of writing.

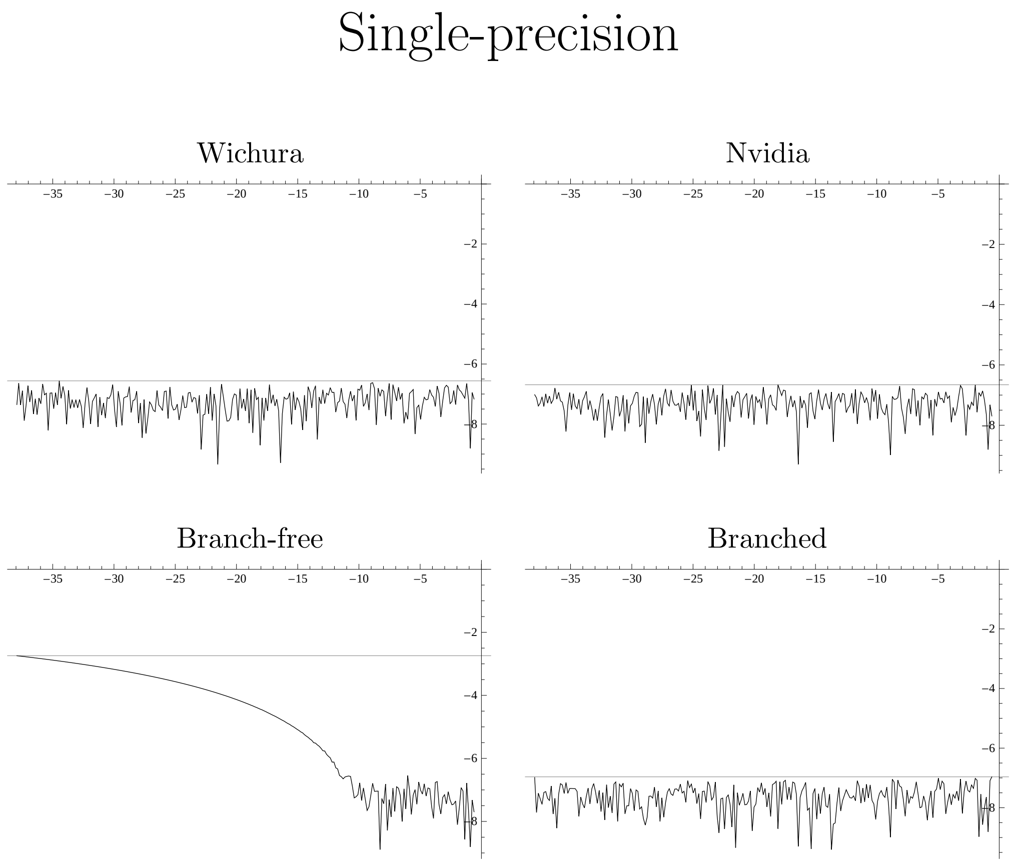


Figure 3.7: Lower tail $|\Phi_{approx}(u)/\Phi(u)-1|$ log-log plots for single-precision CUDA versions of Wichura, Nvidia and the branch-free and branched solutions.

3.3.1 Precision

Mathematica 8 [86] was used by this author to benchmark the accuracy of the algorithms, because it has the built-in capability to run user-defined CUDA functions on the GPU. A reference based on Mathematica's arbitrary-precision `InverseErfc` function was used. Random numbers were again generated on the GPU's memory by Nvidia's CUDA Random Number Generation library [61]. These were then copied to Mathematica for the computation of reference output. Each algorithm was then ran and their output was compared with the reference. Figures 3.7 and 3.8 show the accuracy of the algorithms for the lower tail. The plots in this section show $\log_{10}|\text{approx}/\text{exact}-1|$ as a function of $\log_{10}(u)$. Shaw's branch-free formula preserves a precision better than 4.1×10^{-7} down to about $u = 10^{-10}$.

Figure 3.9 shows our single-precision hybrid approximation maintains accuracy

Double-precision

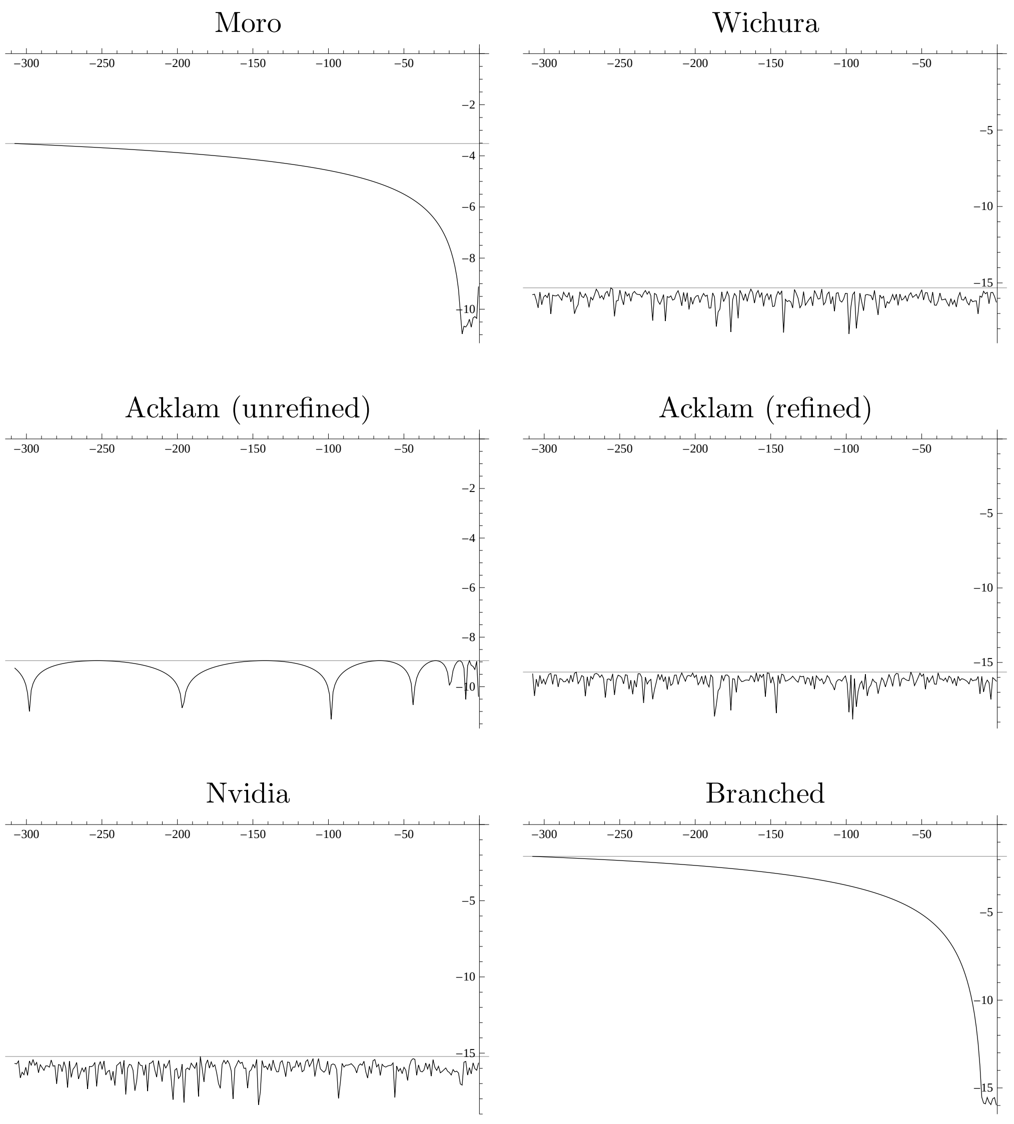


Figure 3.8: Lower tail $|\Phi_{approx}(u)/\Phi(u)-1|$ log-log plots for double-precision CUDA versions of Moro, Wichura, Acklam (unrefined), Acklam (refined), Nvidia, and the branched solution.

Table 3.1: Percentage difference of the time taken for each single-precision algorithm, relative to Nvidia's built-in `normcdfinv` function, to compute the normal quantile for 1 billion pseudo-random uniform variates on the test Nvidia GPUs. (Lower is better.)

Single-precision algorithm	GTX Titan	K80	C2050
Wichura	+19.36	+32.29	+33.53
Branch-free	+11.10	+17.49	+18.06
Branched	+16.09	+24.75	+23.54
Hybrid	−4.90	−5.17	−2.51

Table 3.2: Percentage difference of the time taken for each double-precision algorithm, relative to Nvidia's built-in `normcdfinv` function, to compute the normal quantile for 1 billion pseudo-random uniform variates on the test Nvidia GPUs. (Lower is better.)

Double-precision algorithm	GTX Titan	K80	C2050
Moro	+52.23	+67.58	+70.82
Wichura	+43.71	+54.12	+56.83
Acklam (unrefined)	+11.46	+25.91	+27.87
Acklam (refined)	+244.12	+241.27	+248.22
Branched	+13.20	+27.74	+29.75
Hybrid	−27.24	−4.27	−5.01

right down to the smallest non-zero input. Extensive tests suggest its relative error peaks at about 3.91×10^{-7}. Figure 3.10 shows the left-tail outcome for the double-precision hybrid approximation. This confirms that double-precision is realised consistently throughout the region of interest. Between 2^{-32} and 1, the hybrid approximation's peak relative error is about 8.58×10^{-16}.

3.3.2 Speed

The timings for single-precision are given in Table 3.1. The hybrid algorithm is the fastest, having a slight edge over Nvidia. The branch-free method comes third behind the hybrid and Nvidia algorithms, but has the merit of robustness with respect to any future increase in the number of threads per warp. It may also be of value for other vector/parallel architectures such as Intel's Xeon Phi.

The timings for double-precision, reported in Table 3.2, are very interesting.

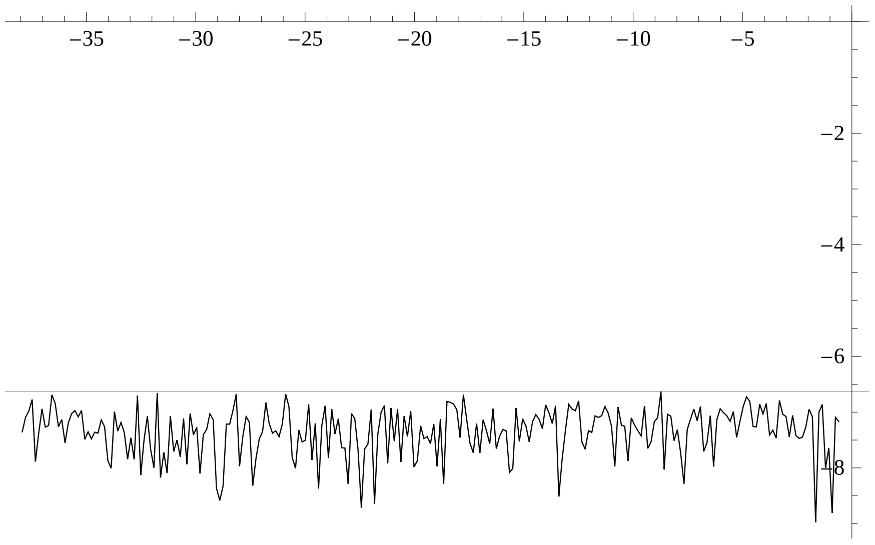


Figure 3.9: Precision of the *single-precision* hybrid approximation in the left tail.

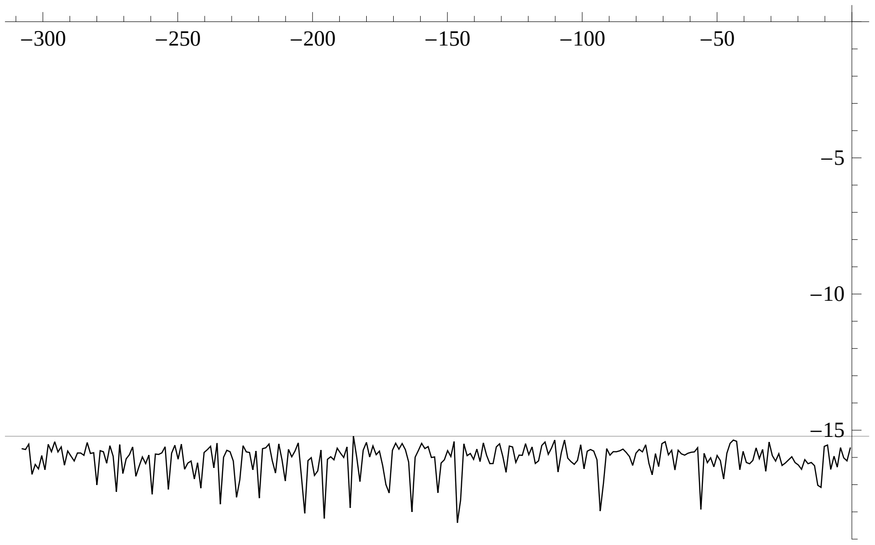


Figure 3.10: Precision of the *double-precision* hybrid approximation in the left tail.

The use of the $\tilde{t}_2$ distribution as an intermediate distribution results in a clear win for this method on performance grounds, and given that it also provides clear double-precision accuracy over the range of interest for Monte Carlo work, we argue that it is a good candidate for the preferred method on GPUs. The advantage arises from the relatively fast speed of the inverse square root operation compared to the logarithm in double-precision on Nvidia GPUs. We do not quite know why the difference for the GTX Titan time is markedly exaggerated. We also noticed the effect on a GeForce GTX 480 card. The GTX GPUs are consumer-level products (aimed at gamers), whereas the Tesla K80 and C2050 cards are both primarily designed for high-performance computing. Tesla GPUs have first-class support for double-precision computations, so this may be the reason.

It should be noted that Nvidia's built-in `normcdfinv` function makes use of tailored implementations of mathematical functions. These implementations only consider the input range of relevance. For example, where a generic `log` could be used, instead there is a stripped down version of `log` that does not support special cases such as infinity and NaN[2] values, which would cause overheads and are unneeded in the context of `normcdfinv`. Our implementations of Φ^{-1} only use standard functions from [60]. The timings in Tables 3.1 and 3.2 could, therefore, be improved.

3.4 Commentary

We have described a new algorithm for computing the normal quantile function on GPUs. The normal distribution is almost unique from our perspective in that its quantile function is not elementary, yet there are no shape parameters to be concerned of. This means that rational minimax approximations are a viable solution with very little downside.

As we will see in the next three chapters, the presence of a shape parameter in a distribution complicates quantile function approximation considerably.

[2]Not a Number

Chapter 4

Gamma distribution

The gamma distribution is used in physical and financial modelling. Rainfall [35] and insurance claims [13, p. 43] are two examples. Let

$$F_\alpha(x) = \frac{1}{\Gamma(\alpha)} \int_0^x t^{\alpha-1} e^{-t} \, \mathrm{d}t = \frac{\gamma(\alpha, x)}{\Gamma(\alpha)} \tag{4.1}$$

be the CDF of the gamma distribution with shape parameter $\alpha > 0$ and unit scale parameter.[1] Now let $q_\alpha = F_\alpha^{-1}$ be the functional inverse of the gamma CDF. The gamma quantile function q_α clearly does not have a closed-form expression, so numerical methods must be used to compute it. The derivative of F_α, the gamma PDF

$$f_\alpha(x) = \frac{\mathrm{d}}{\mathrm{d}x} F_\alpha(x) = \frac{x^{\alpha-1} e^{-x}}{\Gamma(\alpha)}, \tag{4.2}$$

becomes very small in the right tail. This makes root finding in this region problematic. Figure 4.1 shows the gamma PDF f_α for various α.

The normal distribution can be sampled via the Box–Muller transformation, but there is no known equivalent for sampling the gamma distribution. However, a simple and efficient rejection-based algorithm for generating gamma variates is given by Marsaglia and Tsang in [51]. Changing the value of α does not have a significant effect on the computational complexity of this algorithm, which is a desirable feature. Moreover, only one normal and one uniform variate are required per gamma variate. If the generation of gamma variates via inversion is not required, then Marsaglia and Tsang's algorithm should be considered.

[1]The gamma distribution has a scale parameter $\beta > 0$, such that if $X \sim \Gamma(\alpha, 1)$, then $\beta X \sim \Gamma(\alpha, \beta)$. The gamma CDF with general scale parameter is as per (4.1) but x is replaced with x/β. We can thus always assume $\beta = 1$.

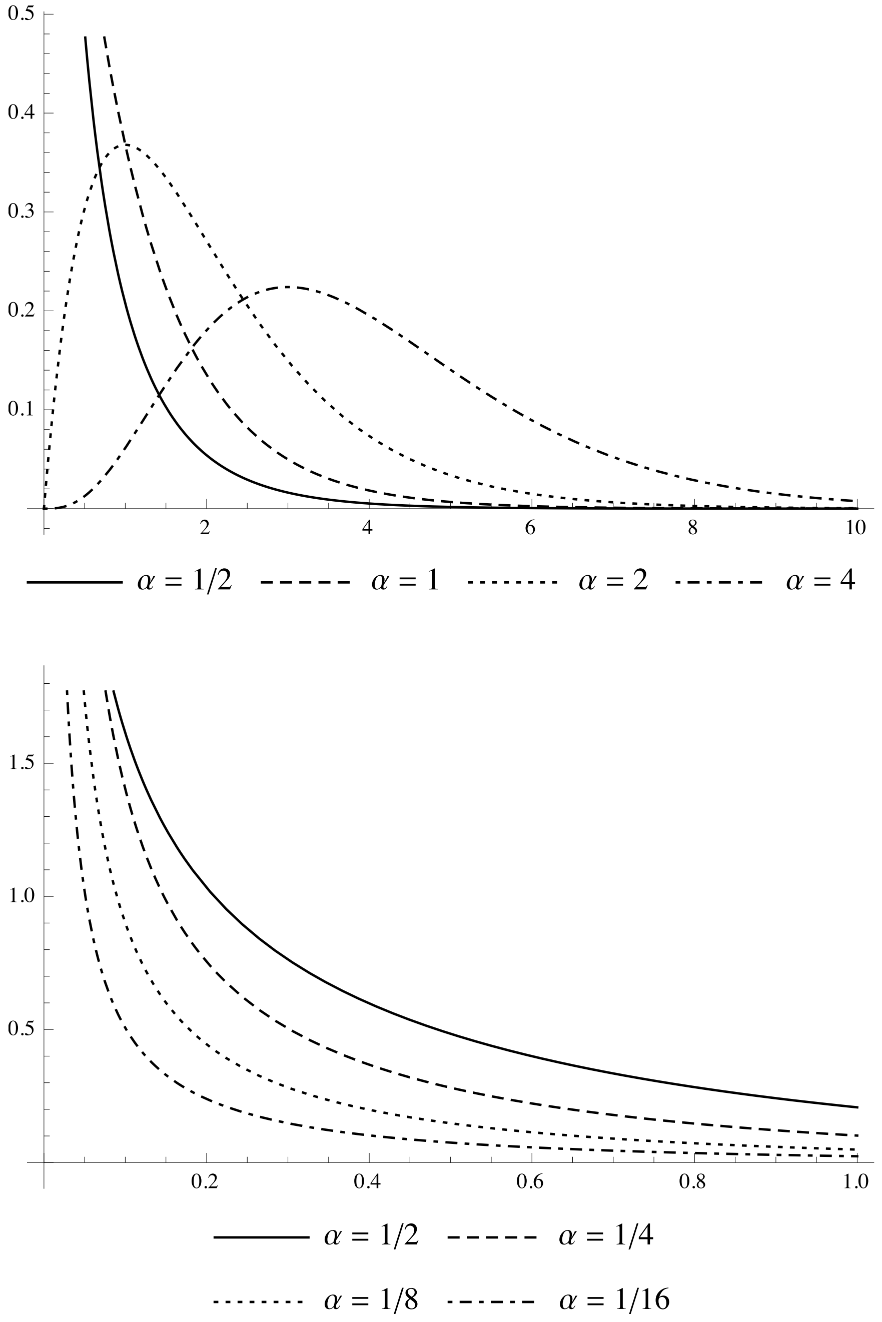


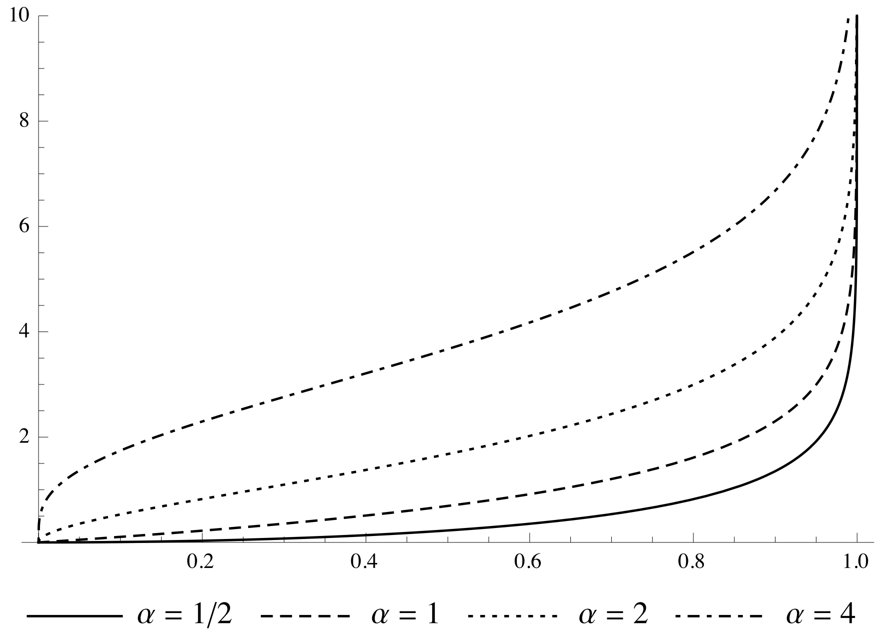
Figure 4.1: The gamma PDF $f_\alpha(x)$ for various α.

A method for parallel inversion of the gamma distribution is described in this chapter. This is very desirable for random number generation in Monte Carlo simulations where gamma variates are required. In this chapter, we will concentrate on the case of generating large quantities of identically distributed gamma variates at a time. When the shape parameter α is fixed, it opens the possibility of pre-computing a fast approximation to q_α. Let α be a fixed but arbitrary shape parameter. Explicitly, given a list of uniformly distributed random numbers our algorithm applies q_α to each element. The result is, therefore, a list of random numbers distributed according to the gamma distribution with shape α. The output of our algorithm has accuracy close to a choice of single- or double-precision machine epsilon.

Inversion of the gamma distribution is traditionally accomplished using some form of root finding. This is known to be computationally expensive. Our algorithm departs from this paradigm by using an initialisation phase to construct, on the fly, a piecewise Chebyshev polynomial approximation to a transformation function, which can be evaluated very quickly during variate generation. The Chebyshev polynomials are high order, for good accuracy, and generated via recurrence relations derived from recycling ODEs (2.5).

We are not aware of any research or commercial solutions for GPU-optimised inversion of the gamma distribution. We remodel the gamma quantile function for efficient evaluation on GPUs and other parallel architectures by using various ideas. More specifically, we leverage quantile mechanics with tailored changes of variable to facilitate fast and accurate computation of the gamma quantile function. A novelty of our approach is that the same change of variable is applied to each uniform random number before evaluating the transformation function. This is particularly amenable to implementation on single instruction, multiple data (SIMD) architectures, whose performance is sensitive to frequently diverging execution flows due to conditional statements. We show that the performance of a CUDA GPU implementation of our algorithm (called Quantus) is within an order of magnitude of the time to compute the normal quantile function. An open-source implementation of our new GPU-optimised algorithm is publicly available.[2]

[2]https://github.com/thomasluu/quantus



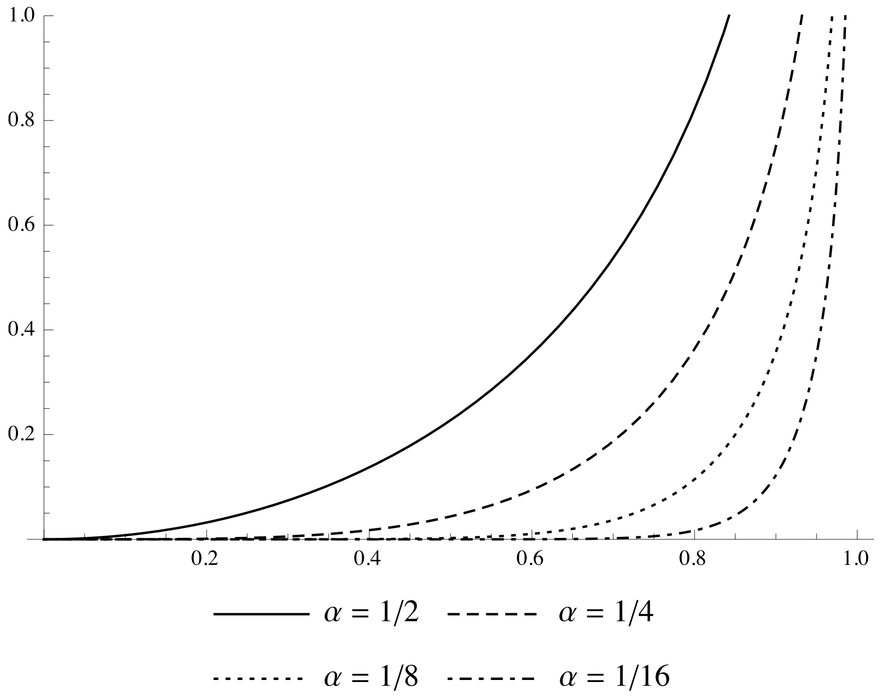


Figure 4.2: The gamma quantile function $q_\alpha(u)$ for $0 \leqslant u < 1$ and various α.

4.1 Approximating the gamma quantile function

Figure 4.2 shows the gamma quantile function q_α for the same α values in Figure 4.1. By definition, we have $q_\alpha(0) = 0$ and $q_\alpha(1) = \infty$ for all α.

The exponential and χ^2 distributions are special cases of the gamma distribution.[3] The quantile function of the exponential distribution with rate parameter $\lambda > 0$ is

$$\frac{1}{\lambda} q_1(u) = -\frac{1}{\lambda} \log(1-u), \tag{4.3}$$

so $q_1(u)$ is, therefore, trivial to implement. The quantile function of the χ^2 distribution with $\nu > 0$ degrees of freedom is

$$2\, q_{\nu/2}(u). \tag{4.4}$$

If $\nu = 1$,

$$q_{1/2}(u) = [\mathrm{erf}^{-1}(u)]^2, \tag{4.5}$$

so $q_{1/2}(u)$ is another case that is straightforward to implement.

In general, q_α is typically computed by numerical root finding. This is the approach taken by the Boost C++ Math Toolkit [14], which uses [22]. NAG [57] and R [65] instead use [11] to first compute the χ^2 quantile function and then transform the result into the equivalent gamma variate. The most recent (at the time of writing) algorithm for q_α in the literature is in [28]. The authors claim their inversion algorithm, which uses Newton root finding, is more accurate than the one in [22].

The first order gamma quantile ODE,

$$\frac{\mathrm{d}q_\alpha}{\mathrm{d}u} = e^{q_\alpha} \Gamma(\alpha)\, q_\alpha^{1-\alpha}, \tag{4.6}$$

is actually used in [81], where it is solved numerically. However, they encountered accuracy problems when α was near zero. We will treat this with a change of variable.

[3]While not a special case, the beta distribution is related to the gamma distribution. If $X_1 \sim \Gamma(\alpha, 1)$ and $X_2 \sim \Gamma(\beta, 1)$ are independent, $X_1/(X_1 + X_2) \sim \mathrm{B}(\alpha, \beta)$. So if one has a fast gamma variate generator, a fast beta variate generator is always available.

4.1.1 Analytical estimates

As $u \to 0$,

$$\frac{\mathrm{d}q_\alpha}{\mathrm{d}u} \sim \Gamma(\alpha)\, q_\alpha^{1-\alpha}. \tag{4.7}$$

Treating (4.7) as an exact ODE, it has the solution $q_\alpha(u) = [u\Gamma(\alpha+1)]^{1/\alpha}$. This motivates the approximation

$$x_\alpha(u) = [u\Gamma(\alpha+1)]^{1/\alpha} \tag{4.8}$$

for $q_\alpha(u)$. The derivative of $F_\alpha(x_\alpha(u))$ is just $\exp(-x_\alpha(u))$. Consequently, the difference of this quantity from one is a measure of how accurate $x_\alpha(u)$ is to $q_\alpha(u)$. This would be useful for q_α algorithms based on root finding, because the expensive iterative process can potentially be short-circuited. Solving $\epsilon = 1 - \exp(-x_\alpha(u))$ for u allows one to find, for a particular α and tolerance ϵ, the range of u that can be managed by $x_\alpha(u)$. Let

$$u_\alpha(\epsilon) = \frac{[-\log(1-\epsilon)]^\alpha}{\Gamma(1+\alpha)} \tag{4.9}$$

be the upper limit of the approximation x_α for ϵ. Table 4.1 gives u_α for various α in single- and double-precision. An analogous procedure can be used to derive approximations for certain distributions, such as the non-central χ^2 distribution. See Chapter 5 for an in-depth analysis of the non-central χ^2 quantile function.

The form of $x_\alpha(u)$ in (4.8) is unsuitable for practical implementation for very small α, because of the resultant large power, which is problematic for u close to unity. Fortunately, this can be managed:

$$\begin{aligned} x_\alpha(u) &= [u\Gamma(\alpha+1)]^{1/\alpha} \\ &= \exp\left\{\frac{1}{\alpha}[\log u + \log\Gamma(1+\alpha)]\right\}. \end{aligned} \tag{4.10}$$

The power series of $\log\Gamma(1+z)$ for $|z| < 2$ (given by equation 6.1.33 in [1, p. 256]) is especially useful here, for α near zero. The series converges rapidly for $|z| < 1/2$ and the obvious implementation of $\log\Gamma(1+z)$ is absolutely fine for z outside this range.

We will rely on (4.10) when it will yield a sufficiently accurate result. Section 4.1.2 will give details of approximations for other inputs.

Table 4.1: Upper u-limits of the approximation $[u\Gamma(\alpha{+}1)]^{1/\alpha}$ to the gamma quantile function $q_\alpha(u)$ for various α in single- and double-precision. Asterisked values are not representable in the respective floating-point number formats and would be flushed to zero. This is the same for the double-asterisked value, but it would be rounded to one.

	$u_\alpha(\epsilon)$	
α	Single-precision ($\epsilon = \epsilon_s$)	Double-precision ($\epsilon = \epsilon_d$)
10^{-9}	9.9999998×10^{-1}**	$9.999999638404157 \times 10^{-1}$
10^{-8}	9.9999984×10^{-1}	$9.999996384042162 \times 10^{-1}$
10^{-7}	9.9999839×10^{-1}	$9.999963840480389 \times 10^{-1}$
10^{-6}	9.9998394×10^{-1}	$9.999638410680227 \times 10^{-1}$
10^{-5}	9.9983943×10^{-1}	$9.996384694366355 \times 10^{-1}$
10^{-4}	9.9839545×10^{-1}	$9.963905630200882 \times 10^{-1}$
10^{-3}	9.8406912×10^{-1}	$9.644855708647861 \times 10^{-1}$
10^{-2}	8.5157729×10^{-1}	$6.965068173834265 \times 10^{-1}$
10^{-1}	1.9915322×10^{-1}	$2.668089225353158 \times 10^{-2}$
10^{0}	5.9604647×10^{-8}	$1.110223024625157 \times 10^{-16}$
10^{1}	$1.5596895 \times 10^{-79}$*	$7.840418869435904 \times 10^{-167}$
10^{2}	$3.6141656 \times 10^{-881}$*	$3.724082781223321 \times 10^{-1754}$*

4.1.2 Numerical approximations

The gamma quantile function is neither analytic at zero nor easy to approximate without treating the tail separately. We can actually kill two birds with one stone by considering a transformation of the gamma distribution. This indirect route, which turns out to be fruitful, does not appear to have been explored in the literature. Let $X \sim \Gamma(\alpha, 1)$. We will consider $Y \sim \log X$. Since the range of X is $[0, \infty)$, the range of Y is the entire real line. Y follows the exp-gamma distribution with shape α, unit scale, and location zero. The CDF of Y is

$$\hat{F}_\alpha(x) = F_\alpha(e^x), \tag{4.11}$$

which implies its inverse is

$$\hat{q}_\alpha(u) = \log q_\alpha(u), \tag{4.12}$$

so if we can approximate $\hat{q}_\alpha(u)$, then $q_\alpha(u)$ follows immediately. The PDF of Y is

$$\hat{f}_\alpha(x) = \frac{e^{x\alpha - e^x}}{\Gamma(\alpha)}, \tag{4.13}$$

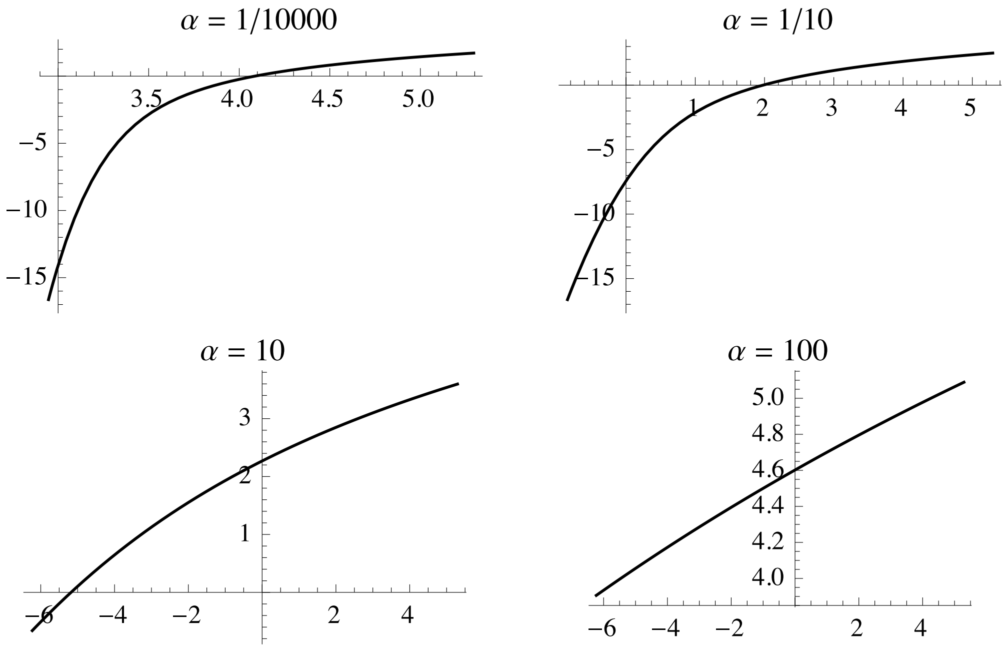


Figure 4.3: The normal to exp-gamma transformation $\hat{q}_\alpha \circ \Phi$ for various α. The ranges in the top two plots are truncated due to the analytic approximation from Section 4.1.1.

so

$$H_{\hat{f}_\alpha}(x) = e^x - \alpha, \tag{4.14}$$

as per (2.3). In contrast to

$$H_{f_\alpha}(x) = \frac{1 + x - \alpha}{x}, \tag{4.15}$$

which diverges as $x \to 0$, $H_{\hat{f}_\alpha}(x)$ clearly converges as $x \to -\infty$.

We observed that the density of Y is remarkably similar to the skew-normal distribution[4] [5] with shape $-\alpha^{-1}$. This suggests that the skew-normal distribution would be useful for variate recycling. Unfortunately, this distribution's quantile function is only straightforward to evaluate for a very limited number of cases—none of which that are of use to us, but all is not lost. The left tail of the skew-normal distribution with negative shape (skew) "decreases at the same rate as the normal distribution tail" [6, pp. 52–53]. This points to the normal distribution as

[4]The PDF of the skew-normal distribution with shape parameter $\alpha \in \mathbf{R}$ is $2\phi(x)\Phi(\alpha x)$. The quantile function of this distribution is, therefore, easy to compute for $\alpha \in \{-1, 0, 1\}$. See Chapter 6 for an in-depth analysis of the skew-normal quantile function.

being a good alternative for variate recycling. We have

$$\begin{aligned} v_0 &= \Phi^{-1}(2^{-32}) \approx -6.23 \text{ (for 32-bit RNGs)}, \\ v_0 &= \Phi^{-1}(2^{-64}) \approx -9.08 \text{ (64-bit RNGs)} \end{aligned} \tag{4.16}$$

and

$$\begin{aligned} \Phi^{-1}(1-\epsilon_s) &\approx 5.29 \text{ (for single-precision)}, \\ \Phi^{-1}(1-2^{-32}) &\approx 6.23 \text{ (double-precision 32-bit RNGs)}, \\ \Phi^{-1}(1-\epsilon_d) &\approx 8.21 \text{ (double-precision 64-bit RNGs)}, \end{aligned} \tag{4.17}$$

which gives an idea of the ranges we typically have to approximate over. Nevertheless, for completeness, we will also show that our method—very naturally—accommodates for input down to $\min'_d$. However, $q_\alpha(u)$ with a very small α parameter skirts the u-axis and suddenly goes to infinity. As shown in Section 4.1.1, the start of the range to approximate over can sometimes be brought in. Figure 4.3 shows the appropriateness of the normal to exp-gamma transformation

$$Q(v) = \hat{q}_\alpha(\Phi(v)) \tag{4.18}$$

for various α. Note that the transformation becomes more linear as α increases. This is because the exp-gamma distribution converges to the normal distribution as $\alpha \to \infty$.

Recalling the normal PDF (3.2), H for the normal distribution is easily found to be

$$H_\phi(x) = x. \tag{4.19}$$

Combining (2.5) with (4.14) and (4.19), the normal to exp-gamma quantile recycling ODE and initial conditions are

$$\begin{aligned} \frac{\mathrm{d}^2 Q}{\mathrm{d}v^2} &= \frac{\mathrm{d}Q}{\mathrm{d}v}\left[\left(e^Q - \alpha\right)\frac{\mathrm{d}Q}{\mathrm{d}v} - v\right], \\ Q(v_i) &= \hat{q}_\alpha(\Phi(v_i)), \\ Q'(v_i) &= \frac{\phi(v_i)}{\hat{f}_\alpha(Q(v_i))}, \end{aligned} \tag{4.20}$$

or equivalently

$$\begin{aligned} Q'_0 &= Q_1, \\ Q'_1 &= Q_1\left[\left(e^{Q_0} - \alpha\right)Q_1 - v\right], \end{aligned} \tag{4.21}$$

with $Q_0(v_i) = Q(v_i)$ and $Q_1(v_i) = Q'(v_i)$. The first order system can be represented as

$$\begin{aligned} V_1 &= e^{Q_0}, \\ V_2 &= V_1 - \alpha, \\ V_3 &= V_2 \cdot Q_1, \\ V_4 &= V_3 - v, \\ V_5 &= Q_1 \cdot V_4, \\ (Q_0)_1 &= Q_1, \\ (Q_1)_1 &= V_5, \end{aligned} \tag{4.22}$$

where $(X)_k$ denotes the kth coefficient in the Taylor series of $X(v)$. So

$$\begin{aligned} (V_1)_k &= \begin{cases} e^{(Q_0)_0} & \text{if } k = 0, \\ \sum_{j=0}^{k-1}(1 - j/k)(V_1)_j(Q_0)_{k-j} & \text{otherwise,} \end{cases} \\ (V_2)_k &= (V_1)_k - \delta_k \alpha, \\ (V_3)_k &= \sum_{j=0}^{k}(V_2)_j(Q_1)_{k-j}, \\ (V_4)_k &= (V_3)_k - (v)_k, \\ (V_5)_k &= \sum_{j=0}^{k}(Q_1)_j(V_4)_{k-j}, \\ (Q_0)_{k>0} &= \frac{1}{k}(Q_1)_{k-1}, \\ (Q_1)_{k>0} &= \frac{1}{k}(V_5)_{k-1}, \end{aligned} \tag{4.23}$$

with

$$\begin{aligned} (v)_k &= \begin{cases} v & \text{if } k = 0, \\ 1 & \text{if } k = 1, \\ 0 & \text{otherwise,} \end{cases} \\ (Q_0)_0 &= Q(v), \\ (Q_1)_0 &= Q'(v), \end{aligned} \tag{4.24}$$

where we have employed the technique from [54, Section 3.4] to derive the recurrence relations for the Taylor coefficients of Q about any point. (See also [20] and [56]. The latter novelly extends the method to several non-Pearson distributions, which have relatively complicated H-functions.) These recurrence relations are straightforward to implement on a computer. Moreover, for $1, 2, \dots, n$ the total number of elementary operations required to compute $(Q_0)_n$ is of order n^2 (assuming $(Q_0)_1, (Q_0)_2, \dots, (Q_0)_{n-1}$, and the previously computed auxiliary variables are cached so they can be reused). The complexity of evaluating $(Q_0)_k$ is thus relatively low. The accuracy of the summations corresponding to V_1 and the product auxiliary variables (V_3 and V_5) can be improved by Kahan/compensated summation (see, e.g., [33]).

Given a suitably accurate numerical approximation of $Q(v_i)$ and setting $v = v_i$ in the recurrence relations above, we can compute the coefficients of the Taylor expansion of $Q(v)$ about $v = v_i$, $(Q_0)_k$, up to an *arbitrary order*. The truncated Taylor series of $Q(v)$ about $v = v_i$ is thus

$$\sum_{k=0}^{n} (Q_0)_k (v - v_i)^k. \tag{4.25}$$

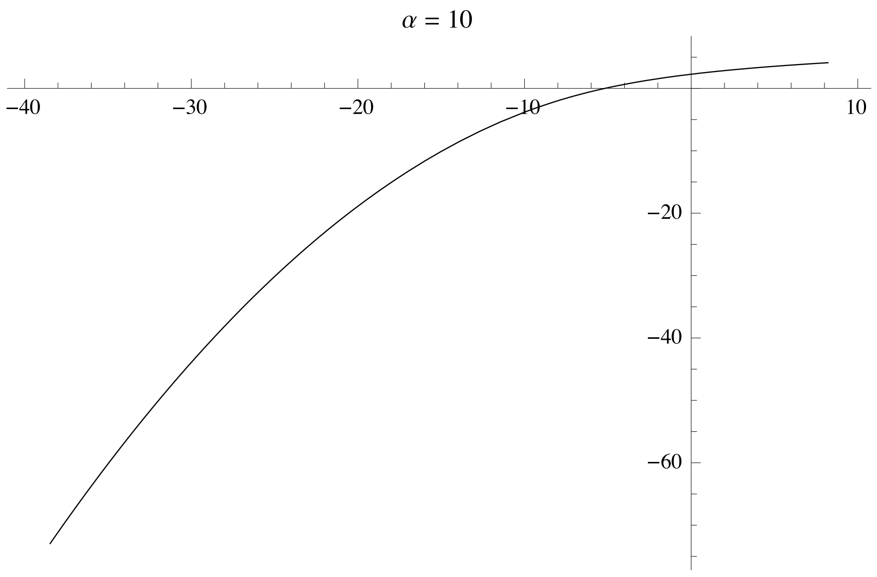
We have shown that the normal to exp-gamma transformation regularises q_α well for uniform output from 32- and 64-bit RNGs. We have also shown that the analytic approximation in Section 4.1.1 can handle input close to zero for small to moderate α. For other α, the normal to exp-gamma transformation can be extended right down to $\min'_d$. Figure 4.4 shows that this transformation is perfectly serviceable for this case.

Very large shape parameter values

It is well known that the gamma distribution tends to the normal distribution as $\alpha \to \infty$. Since the normal distribution approximates the gamma distribution well for large α, it is a natural candidate for variate recycling in this case. While the normal to exp-gamma transformation is theoretically suitable for very big α values, directly transforming normal variates to gamma ones is more efficient. Details for this are given here for completeness. The ranges we have to approximate over are as per (4.16) and (4.17). Figure 4.5 shows the appropriateness of the normal to gamma transformation

$$Q(v) = q_\alpha(\Phi(v)) \tag{4.26}$$

for various large α.



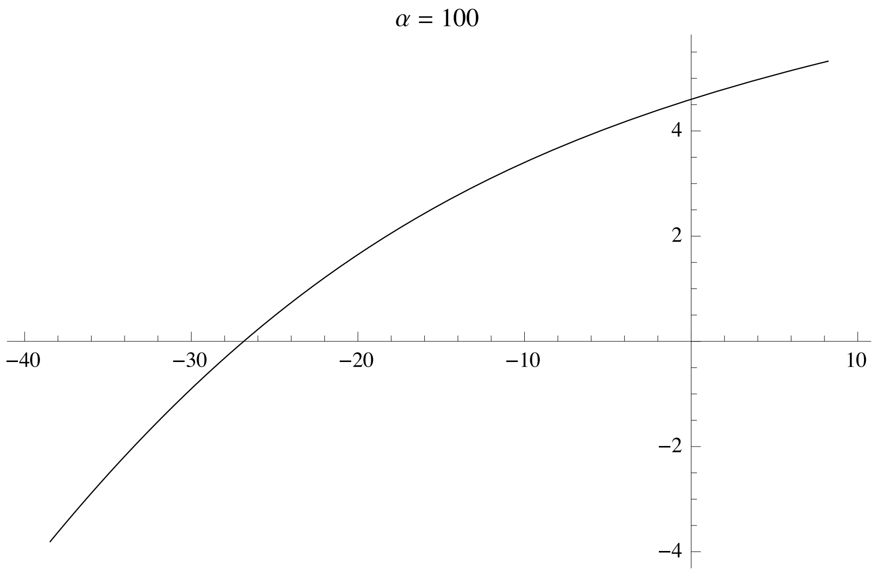


Figure 4.4: The normal to exp-gamma transformation $\hat{q}_\alpha \circ \Phi$ for $\alpha = 10$ and $\alpha = 100$, applicable to the full complement of double-precision numbers.

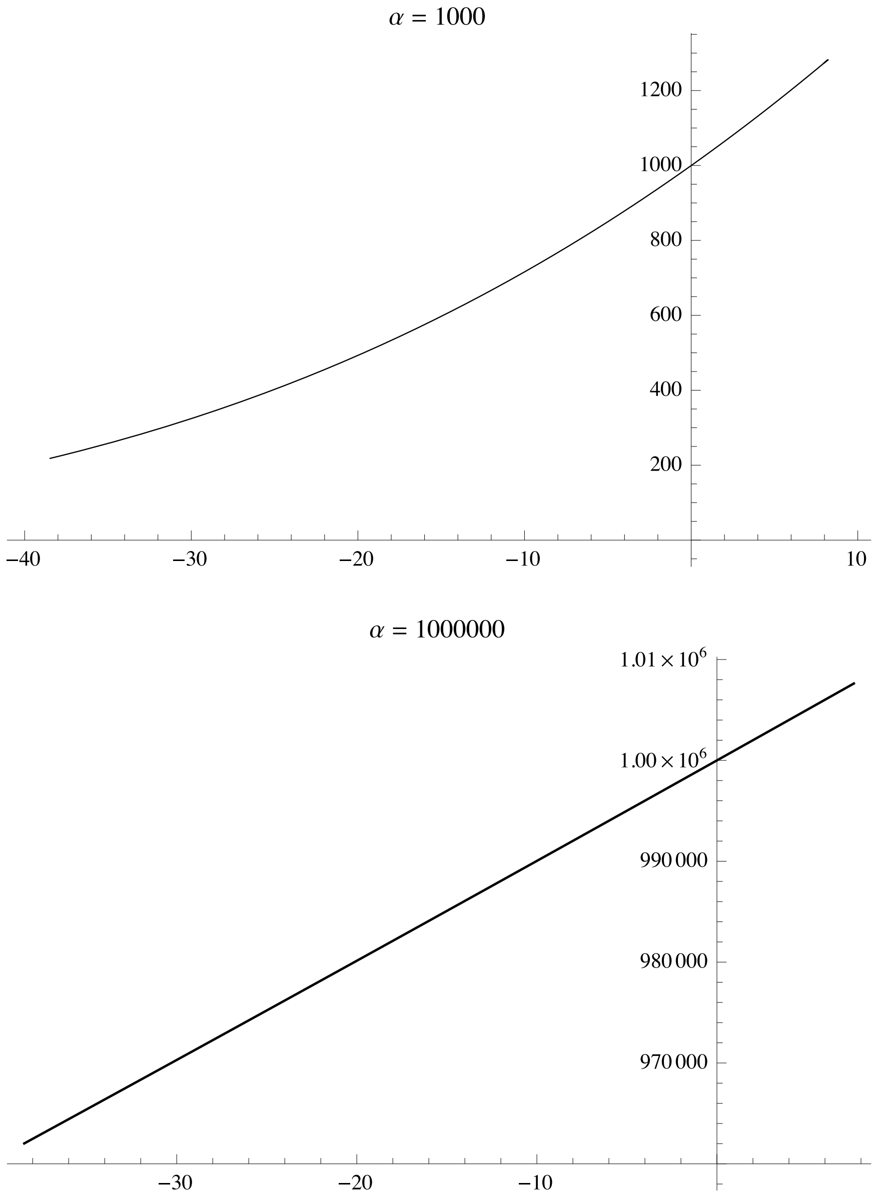


Figure 4.5: The normal to gamma transformation $q_\alpha \circ \Phi$ for $\alpha = 10^3$ and $\alpha = 10^6$, applicable to the full complement of double-precision numbers.

The normal to gamma quantile recycling ODE and initial conditions are

$$\begin{aligned}
\frac{\mathrm{d}^2Q}{\mathrm{d}v^2} &= \frac{\mathrm{d}Q}{\mathrm{d}v}\left(\frac{1-\alpha+Q}{Q}\frac{\mathrm{d}Q}{\mathrm{d}v} - v\right), \\
Q(v_i) &= q_\alpha(\Phi(v_i)), \\
Q'(v_i) &= \frac{\phi(v_i)}{f_\alpha(Q(v_i))},
\end{aligned} \tag{4.27}$$

or equivalently

$$\begin{aligned}
Q_0' &= Q_1, \\
Q_1' &= Q_1\left(\frac{1-\alpha+Q_0}{Q_0}Q_1 - v\right),
\end{aligned} \tag{4.28}$$

with $Q_0(v_i)$ and $Q_1(v_i)$ as appropriate. This leads to

$$\begin{aligned}
V_1 &= 1-\alpha+Q_0, & (V_1)_k &= \delta_k(1-\alpha) + (Q_0)_k, \\
V_2 &= V_1 \cdot Q_1, & (V_2)_k &= \sum_{j=0}^{k}(V_1)_j(Q_1)_{k-j}, \\
V_3 &= \frac{V_2}{Q_0}, & (V_3)_k &= \frac{1}{(Q_0)_0}\left[(V_2)_k - \sum_{j=1}^{k}(Q_0)_j(V_3)_{k-j}\right], \\
V_4 &= V_3 - v, & (V_4)_k &= (V_3)_k - (v)_k, \\
V_5 &= Q_1 \cdot V_4, & (V_5)_k &= \sum_{j=0}^{k}(Q_1)_j(V_4)_{k-j}, \\
(Q_0)_1 &= Q_1, & (Q_0)_{k>0} &= \frac{1}{k}(Q_1)_{k-1}, \\
(Q_1)_1 &= V_5, & (Q_1)_{k>0} &= \frac{1}{k}(V_5)_{k-1},
\end{aligned} \tag{4.29}$$

where $(v)_k$, $(Q_0)_0$, and $(Q_1)_0$ are as per (4.24).

4.2 Parallel inversion algorithm

We will now give the design of our new gamma quantile function algorithm, and evaluate a CUDA GPU implementation of it.

4.2.1 Algorithm design

Our algorithm for q_α is split into two phases:

- *initialisation* for a given shape parameter α; and
- *generation* of gamma variates with shape α.

For the sake of simplicity, this section will ignore the normal to gamma transformation (4.26). If the transformation is used for large α values, the algorithm steps are broadly identical.

Initialisation

Routine 6 Initialisation.

```
Input: The value of $\alpha$, requested precision $\epsilon$ (either $\epsilon_s$ or $\epsilon_d$), target RNG (either
    32- or 64-bit).
                                        ▷ Determine relevant $u$-range (Section 2.3.1)
 1: $u_{\min} \leftarrow$ the smallest (non-zero) uniform number produced by the target RNG
 2: $u_{\max} \leftarrow$ the largest (excluding 1) uniform number produced by the target RNG
 3: if $u_{\min} < u_\alpha(\epsilon)$ then
 4:     $u_{\min} \leftarrow u_\alpha(\epsilon)$                              ▷ See Section 4.1.1
 5: end if
 6: if $u_{\min} \geqslant u_{\max}$ then
 7:     $x_\alpha(u)$ can be used for all $0 \leqslant u \leqslant u_{\max}$ so no further initialisation is neces-
    sary.
 8: end if
                    ▷ Compute the smallest and largest expected inputs into the recycling
    function $Q$ (4.18)
 9: $v_0 \leftarrow \Phi^{-1}(u_{\min})$
10: $v_m \leftarrow \Phi^{-1}(u_{\max})$
11: An instance of the recycling ODE (4.20) is solved over $[v_0, v_m]$.
```

Routine 6 gives the pseudocode for this phase. Our ODE solution is an approximation, which is in the form of a piecewise Taylor polynomial approximation. Supposing $[v_0, v_m]$ is partitioned as $v_0 < v_1 < \cdots < v_m$, Q is thus approximated by polynomials of the form

$$\sum_{k=0}^{n} \frac{Q^{(k)}(v_i)}{k!}(v - v_i)^k, \tag{4.30}$$

where we compute the Taylor coefficients using the recurrence relations developed in Section 4.1.2. The task of computing the piecewise polynomial approximations can be done sequentially or in parallel, by concurrently expanding about different points. The ODE solution is basically an $(m+1)\times(n+1)$ matrix with the coefficients of the polynomial expansion about v_i in the ith row. For an arbitrary $v \in [v_0, v_m]$

Table 4.2: The recommended parameters for the initialisation phase of our algorithm.

	Single-precision	Double-precision
Maximum order	10th	20th
Initial step size	1/4	1/8
Tolerance target	$50\epsilon_s$	$50\epsilon_d$

the relevant row for evaluating $Q(v)$ is easily found. If $v_0 < v_1 < \cdots < v_m$ is an equidistant partition of $[v_0, v_m]$, the calculation is even simpler: the correct row is the integer part of $(v - v_0)/h$, assuming h is the distance between any two partition points. We will actually use equidistant partitions, because they are feasible due to our changes of variable. We can estimate the accuracy of an approximant about v_i by evaluating it at v_{i+1} and comparing the result with $Q(v_{i+1})$, since this is where the error will peak. The partition of $[v_0, v_m]$ and order n can be refined and adjusted so as to meet a desired tolerance.

A simple yet reliable procedure therefore exists for solving the recycling ODEs. It works by successively refining a mesh. Given an initial step size, a piecewise Taylor polynomial approximation to Q is generated. Derivatives of up to a specified maximum order can be used to iteratively form piecewise polynomials of increasing accuracy. If an approximation is found that is accurate to some tolerance, initialisation is complete. If such an approximation is not found, the search process is repeated with the step size halved. Table 4.2 gives the initialisation parameters that we recommend. They were found to give a good balance of performance and accuracy. Starting with step sizes with an integral power of two and expanding about points that are a whole multiple of the step size allows for more efficient polynomial selection and evaluation.

In the interests of numerical stability we would prefer to evaluate Chebyshev polynomials instead of Taylor ones. Armed with our Taylor coefficients, an efficient method to accomplish this is given in [78]. (See [56, section 8.5.2] for an application to the hyperbolic distribution.) Our Taylor polynomials can be recast into the Chebyshev form

$$\frac{c_0}{2} + \sum_{k=1}^{n} c_k T_k\left(\frac{v - v_i}{h}\right), \tag{4.31}$$

where $T_k(x)$ are the Chebyshev polynomials of the first kind, defined as

$$T_k(x) = \cos(k \arccos x), \tag{4.32}$$

and c_k are Chebyshev coefficients. These coefficients are computed as

$$c_k = \sum_{r=k}^{n} a_r \theta_{r,k}, \tag{4.33}$$

where

$$a_r = \frac{Q^{(r)}(v_i)}{r!} h^r \tag{4.34}$$

and

$$\theta_{r,k} = \begin{cases} 2^{1-r}\binom{r}{(r-k)/2} & \text{if } r-k \text{ is even,} \\ 0 & \text{otherwise,} \end{cases} \tag{4.35}$$

which can be evaluated recursively.

While our approximants are not guaranteed to preserve complete monotonicity, it is not unreasonable to expect them by and large to be monotonic. We found this to be true in practice. Also, whenever monotonicity between two consecutive variates was violated, the deviation from monotonicity was very close to machine epsilon. Such deviations are irrelevant in practice for most simulations.

Generation

Routine 7 Generation.

```
Input: A uniform variate $u \in (0,1)$
Output: $q_\alpha(u)$
1: if $u \leqslant u_\alpha(\epsilon)$ then
2:     return $x_\alpha(u)$
3: end if
4: $v \leftarrow \Phi^{-1}(u)$          ▷ Evaluate change of variable
5: $y \leftarrow Q(v)$                 ▷ The recycling function is computed
6: return exp($y$)                     ▷ Since $Q(v) = \hat{q}_\alpha(u)$
```

Routine 7 gives the pseudocode for the variate generation phase. The recycling function $Q(v)$ is computed by looking up the appropriate Chebyshev polynomial and evaluating it using Clenshaw's formula [19]. It should be noted that this *does not* introduce additional branch divergence. Once the correct polynomial index is determined by each thread, the polynomials are evaluated synchronously—the instructions to do this are identical, but the coefficients may be different. The variate generation portion of the algorithm is incredibly simple, so the barrier to execution on present and future computer architectures is low.

4.2.2 Computational experiments

We will now demonstrate the parallel performance of our gamma quantile function algorithm. This was done in accordance with Section 2.5. The Oxford English Dictionary says *quantile* originates from the Latin word *quantus*, which means *how great, how much.* Quantus, therefore, seems an apt name for our algorithm implementation. The performance of Quantus was evaluated on two high-end Nvidia GPUs:

- a Kepler-class GeForce GTX Titan; and
- a Fermi-class Tesla C2050.

The test GPUs were hosted in a system with

- an Intel Core i5-4670K (overclocked to 4.2 GHz); and
- 8 GB of RAM.

The system was running

- Ubuntu Server 12.04.2 LTS with GCC 4.6.3;
- Nvidia CUDA 6.5.14; and
- Boost 1.56.0.

The freely available Boost C++ Math Toolkit provides a high-quality quantile function implementation for the gamma distribution (along with several other distributions). Suitably precise initial conditions can hence be computed via Boost's quantile function implementation.

The Quantus initialisation code was parallelised with OpenMP and compiled using GCC with the –O2 optimisation flag. For both single- and double-precision initialisation, double working precision was used. This yields recycling function approximations with accuracy close to machine epsilon. A caveat is that the initial conditions have to be accurate to target precision. This is easily achieved for single target precision. However, for double target precision this is problematic for values in the right tail. We took a brute force approach, simply computing these initial conditions with software-simulated arbitrary precision for $u > 9/10$. We used the normal recycling scheme from Section 4.1.2 for all $\alpha \geqslant 1000$. The normal quantile function was computed using the hybrid GPU-optimised algorithm from Chapter 3.

Speed

Using the parameters in Table 4.2, the coefficients table for the recycling function approximation always comfortably fits[5] in the level 1 (L1) cache on Nvidia GPUs, so evaluation of the function is very fast.

Table 4.3 shows the performance of Quantus for a wide range of shape parameters. The performance is always within an order of magnitude of the time to compute the normal quantile function, which is what we were aiming for. Table 4.3 also shows the initialisation times of Quantus. They are relatively high compared to generation, but bear in mind initialisation is a fixed cost.

Precision

The precision of Quantus was assessed by inspecting the output from ten out of the 100 runs of each speed test. The gamma output and corresponding uniform input were copied from the GPU to the CPU. 80-bit extended precision references were computed using Boost and compared to the copied values.

Table 4.4 gives peak relative error statistics for Quantus with the same shape parameters from the speed test. See Section 2.5.2 for the definitions of E_1 and E_2. We found Quantus achieves better or comparable accuracy with respect to Boost's gamma quantile function in both single- and double-precision, over all distribution parameters and uniform inputs for both relative error measures. The results suggest our algorithm is stable for $\alpha \leqslant 10^3$. The peak backward errors for $\alpha \geqslant 10^4$ deteriorate, because of the large magnitude of the variates, but the forward errors are excellent.

4.3 Commentary

We have described a method for the efficient and accurate parallel inversion of the gamma distribution. Quantile mechanics was used with select changes of variable to accomplish this. We showed that the performance of a CUDA GPU implementation of our algorithm is similar to the time to compute the normal quantile function. The underlying algorithmic ideas should translate well to other parallel architectures, e.g., Intel Xeon Phi.

Devroye in [21, p. 404] said a good gamma variate generator should

- have uniform generation speed over all shape parameters α;

[5]The average table size for single- and double-precision was about 0.4 and 4 KB, respectively.

Table 4.3: Timings in ms to compute the gamma quantile function q_α for 10^7 pseudo-random uniform variates using an implementation of the algorithm described in Section 4.2.1 averaged over 100 runs on the two test Nvidia GPUs. All standard deviations were negligible. Initialisation times for the implementation on an Intel Core i5-4670K system are also shown. Timings for the normal quantile function Φ^{-1} using the hybrid GPU-optimised algorithm from Chapter 3 are given on the bottom row.

	Single-precision			Double-precision		
		Generation			Generation	
α	Init.	Titan	C2050	Init.	Titan	C2050
10^{-9}	0.06	0.70	2.44	10.39	4.32	3.73
10^{-8}	0.37	0.71	2.32	19.84	4.31	3.73
10^{-7}	0.68	0.69	2.32	16.27	4.32	3.74
10^{-6}	0.70	0.72	2.32	17.25	4.38	3.81
10^{-5}	0.83	0.75	2.35	16.88	4.81	4.57
10^{-4}	0.85	1.01	2.65	10.75	7.42	6.63
10^{-3}	0.83	1.50	4.04	9.83	13.72	11.94
10^{-2}	0.79	1.75	5.70	9.47	15.77	13.03
10^{-1}	0.80	1.72	5.47	10.98	13.20	11.36
10^{1}	0.71	1.25	3.69	6.11	9.09	8.21
10^{2}	0.60	1.17	3.44	5.30	8.51	7.65
10^{3}	0.60	1.10	3.08	5.54	6.93	6.51
10^{4}	0.60	0.96	2.59	5.12	6.64	6.23
10^{5}	0.58	0.97	2.59	5.06	6.63	6.23
10^{6}	0.58	0.96	2.59	4.96	6.63	6.23
10^{7}	0.77	0.97	2.59	4.66	6.34	5.96
10^{8}	1.52	0.96	2.59	4.61	6.34	5.96
10^{9}	3.67	0.96	2.59	7.12	6.33	5.96
Φ^{-1}	n/a	0.44	0.97	n/a	4.00	3.61

Table 4.4: Peak relative error statistics for an implementation of the algorithm described in Section 4.2.1, over 10^8 pseudo-random uniform variates for each α. See Section 2.5.2 for the definitions of E_1 and E_2.

	Single-precision		Double-precision	
α	E_1	E_2	E_1	E_2
10^{-9}	nil	nil	2.42×10^{-13}	5.42×10^{-20}
10^{-8}	4.13×10^{-5}	4.13×10^{-13}	2.43×10^{-13}	1.08×10^{-19}
10^{-7}	7.44×10^{-5}	7.44×10^{-12}	2.58×10^{-13}	1.63×10^{-19}
10^{-6}	5.03×10^{-5}	5.03×10^{-11}	2.73×10^{-13}	2.71×10^{-19}
10^{-5}	6.29×10^{-5}	6.29×10^{-10}	3.26×10^{-13}	3.25×10^{-18}
10^{-4}	4.14×10^{-5}	4.14×10^{-09}	2.15×10^{-13}	2.15×10^{-17}
10^{-3}	2.77×10^{-5}	2.77×10^{-08}	1.62×10^{-13}	1.62×10^{-16}
10^{-2}	1.28×10^{-5}	1.28×10^{-07}	1.32×10^{-13}	1.32×10^{-15}
10^{-1}	8.76×10^{-6}	8.76×10^{-07}	4.88×10^{-14}	4.88×10^{-15}
10^{1}	8.15×10^{-7}	7.20×10^{-06}	1.92×10^{-15}	1.45×10^{-14}
10^{2}	1.23×10^{-6}	3.87×10^{-05}	3.01×10^{-15}	6.96×10^{-14}
10^{3}	1.81×10^{-7}	1.49×10^{-05}	6.34×10^{-16}	5.07×10^{-14}
10^{4}	2.23×10^{-6}	1.10×10^{-03}	9.70×10^{-15}	4.94×10^{-12}
10^{5}	2.84×10^{-7}	3.99×10^{-04}	3.27×10^{-16}	4.50×10^{-13}
10^{6}	5.44×10^{-8}	2.66×10^{-04}	2.19×10^{-16}	8.35×10^{-13}
10^{7}	1.02×10^{-7}	1.43×10^{-03}	1.90×10^{-15}	2.90×10^{-11}
10^{8}	7.88×10^{-8}	3.67×10^{-03}	1.99×10^{-16}	7.25×10^{-12}
10^{9}	6.34×10^{-8}	9.71×10^{-03}	1.19×10^{-16}	1.63×10^{-11}

- be simple to implement; and
- have small or non-existent initialisation times.

We believe our algorithm meets the first and last of these goals. While our algorithm is simple in principle, it is certainly not one of the easiest algorithms to implement. Much of the effort is in efficiently implementing the recurrence relations and automating the generation of the gamma quantile function approximation. However, our generation times are more or less uniformly bounded, and our initialisation times are relatively small and get amortised for practical simulation sizes.

Chapter 5

Non-central χ^2 distribution

The non-central χ^2 distribution features in the Cox–Ingersoll–Ross and Heston models in mathematical finance. The PDF of the distribution is

$$f_{\nu,\lambda}(x) = \frac{1}{2} e^{-(x+\lambda)/2} \left(\frac{x}{\lambda}\right)^{\nu/4-1/2} I_{\nu/2-1}\left(\sqrt{\lambda x}\right) \tag{5.1}$$

on $x \in [0, \infty)$, where $\nu > 0$ is the degrees of freedom parameter, $\lambda \geqslant 0$ is the non-centrality parameter and I is the modified Bessel function of the first kind. The PDF can also be written in terms of the normalised confluent hypergeometric limit function ${}_0\tilde{F}_1(a; z) = {}_0F_1(a; z)/\Gamma(a)$:

$$f_{\nu,\lambda}(x) = 2^{-\nu/2} e^{-\frac{1}{2}(\lambda+x)} x^{\nu/2-1} {}_0\tilde{F}_1\left(; \frac{\nu}{2}; \frac{x\lambda}{4}\right). \tag{5.2}$$

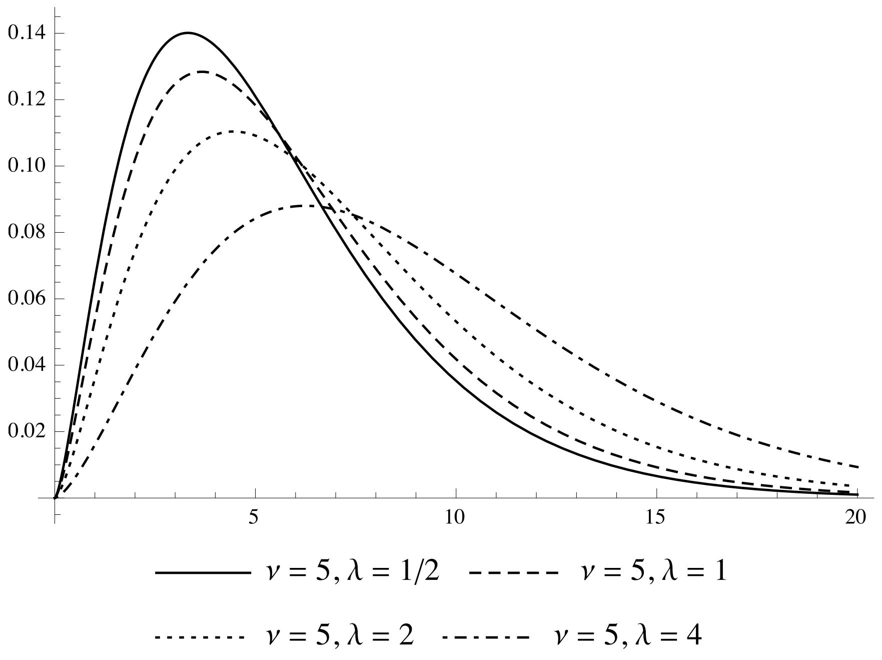
This is the form that we will primarily work with. Figure 5.1 shows the non-central χ^2 PDF for various ν and λ. When $\nu < 2$, the distribution has an infinite peak at the origin. This suggests that the non-central χ^2 quantile function for $\nu < 2$ is harder to compute. When $\lambda = 0$, the non-central χ^2 distribution reduces to the (central) χ^2 distribution. The χ^2 quantile function can be handled with the methods developed in Chapter 4, since it is a special case of the gamma distribution.

The non-central χ^2 CDF is

$$F_{\nu,\lambda}(x) = 1 - Q_{k/2}\left(\sqrt{\lambda}, \sqrt{x}\right) \tag{5.3}$$

where Q_m is Marcum's Q function, which is defined as

$$Q_M(\alpha, \beta) = \frac{1}{\alpha^{M-1}} \int_\beta^\infty x^M e^{-(x^2+\alpha^2)/2} I_{M-1}(\alpha x) \, \mathrm{d}x, \tag{5.4}$$



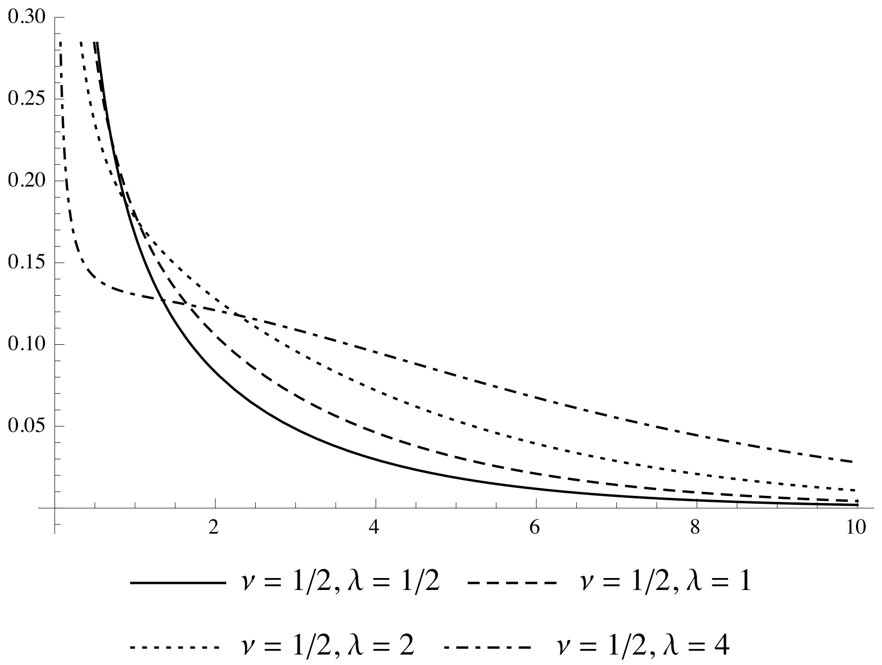


Figure 5.1: The non-central χ^2 PDF $f_{\nu,\lambda}(x)$ for various ν, λ.

with $M, \alpha, \beta \geqslant 0$.

If the generation of non-central χ^2 random variates via inversion is not required, then this is relatively straightforward to accomplish, by exploiting the additive properties of the χ^2 distribution: $X + Y \sim \chi^2_\nu(\lambda)$ if $X \sim \chi^2_\nu$ and $Y \sim \chi^2_{2V}$ where $V \sim Poi(\lambda/2)$.

5.1 A new analytic approximation

Let $q_{\nu,\lambda} = F^{-1}_{\nu,\lambda}$ be the non-central χ^2 quantile function. In the interests of brevity, q will be used instead of $q_{\nu,\lambda}$ when there is no ambiguity. The first order non-central χ^2 quantile ODE is

$$\begin{aligned} \frac{dq}{du} &= \frac{1}{f_{\nu,\lambda}(q)} && (5.5)\\ &= \frac{1}{2^{-\nu/2} e^{-\frac{1}{2}(\lambda+q)} q^{\nu/2-1} {}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda q}{4}\right)} && (5.6)\\ &= \frac{2^{\nu/2} e^{\frac{1}{2}(\lambda+q)} q^{1-\nu/2}}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda q}{4}\right)}. && (5.7) \end{aligned}$$

So, as $u \to 0$,

$$\frac{dq}{du} \sim 2^{\nu/2} e^{\frac{1}{2}\lambda} q^{1-\nu/2} \Gamma(\nu/2). \tag{5.8}$$

This suggests that the asymptotic formula

$$x_{\nu,\lambda}(u) = [2^{\nu/2-1} e^{\lambda/2} u\nu\Gamma(\nu/2)]^{2/\nu} = [2^{\nu/2} e^{\lambda/2} u\Gamma(1+\nu/2)]^{2/\nu} \tag{5.9}$$

would be a useful approximation for when $q_{\nu,\lambda}(u)$ is very small. Indeed, the formula is now used by the Boost C++ Math Toolkit[1] [14] and Wolfram Mathematica [88]. Note that when $\lambda = 0$ (the central χ^2 special case), the formula matches (4.8) scaled by two with $\alpha = \nu/2$. The precision of the approximation is given by

$$1 - f_{\nu,\lambda}(x_{\nu,\lambda}(u)) \frac{2x_{\nu,\lambda}(u)}{u\nu}. \tag{5.10}$$

[1]See https://svn.boost.org/trac/boost/changeset/85074 and https://svn.boost.org/trac/boost/changeset/85103 for the accepted source code changes.

5.2 Existing approximations

There are a handful of established approximations for the non-central χ^2 quantile function, including one due to Pearson [63]. However, the approximations typically fail for small enough inputs—Pearson's approximation can yield a negative result and the approximation from [67] can give a complex-valued result. These are known issues. They are especially problematic for refinement via root finding, since these invalid values obviously cannot be fed into $F_{\nu,\lambda}(x)$. Several popular existing approximations will now be detailed, with emphasis on when they can break down.

5.2.1 Sankaran's approximation

Sankaran gives an analytic approximation for the non-central χ^2 CDF [67, 68]:

$$F_{\nu,\lambda}(x) \approx \Phi\left[\frac{\frac{x}{\nu+\lambda} - \mu}{\sigma}\right] \tag{5.11}$$

where Φ is the standard normal CDF and

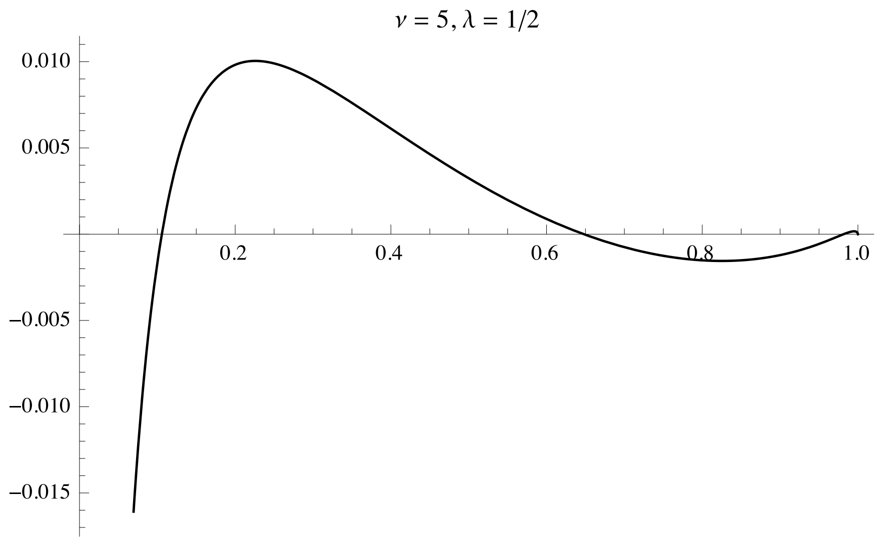
$$\begin{aligned} h &= 1 - \frac{2}{3}\frac{(\nu+\lambda)(\nu+3\lambda)}{(\nu+2\lambda)^2} \\ p &= \frac{\nu+2\lambda}{(\nu+\lambda)^2} \\ m &= (h-1)(1-3h) \\ \mu &= 1 + hp(h-1-(1-h/2)mp) \\ \sigma &= h\sqrt{2p}(1+mp/2). \end{aligned} \tag{5.12}$$

Sankaran's CDF approximation is readily invertible:

$$q_{\nu,\lambda}(u) \approx (\nu+\lambda)\left[\Phi^{-1}(u)\,\sigma + \mu\right]^{1/h}. \tag{5.13}$$

Figure 5.2 shows the backward relative error (2.14) of Sankaran's non-central χ^2 quantile function approximation for $\nu = 5, \lambda = 1/2$ and $\nu = 1/2, \lambda = 1/2$. The approximation is generally good in the central region, but deteriorates badly in the left tail.

For small enough u, when $\Phi^{-1}(u)\,\sigma + \mu < 0$, (5.13) will give a complex-valued result. Judging from error messages about invalid comparisons with complex values, Mathematica 9.0.1's [87] non-central χ^2 quantile function implementation appears to be based on root finding using a slight modification of Sankaran's approximation.



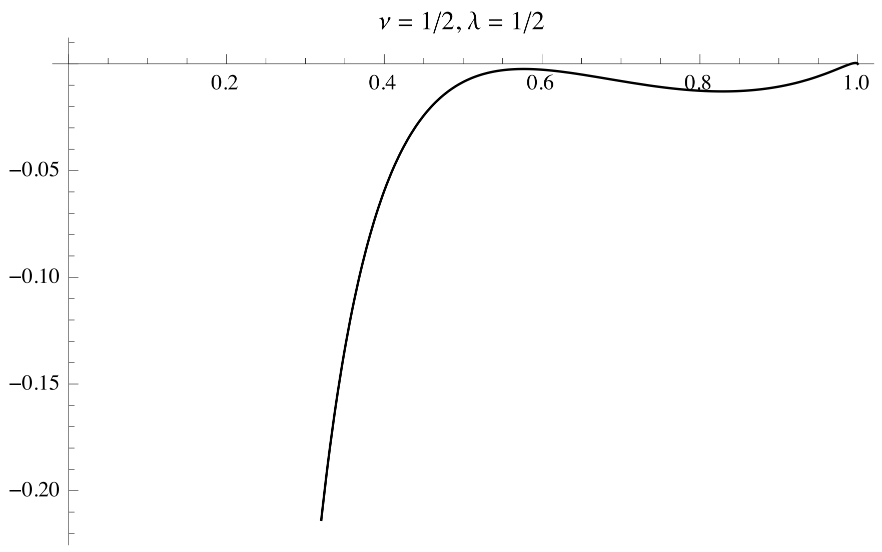


Figure 5.2: The backward relative errors of Sankaran's non-central χ^2 quantile approximation for $\nu = 5, \lambda = 1/2$ and $\nu = 1/2, \lambda = 1/2$.

For instance, substituting $(u, \nu, \lambda) = \left(\frac{1}{100}, \frac{1}{2}, \frac{1}{10}\right)$ into (5.13) produces $-0.740868 + 0.287014i$. Calling Mathematica 9.0.1's non-central χ^2 quantile function with the same parameters results in the following output.

```
Quantile[NoncentralChiSquareDistribution[1/2,1/10],1/100]//N
Greater::nord: Invalid comparison with -0.739859+0.286623 I attempted. >>
LessEqual::nord: Invalid comparison with -0.739859+0.286623 I attempted. >>
LessEqual::nord: Invalid comparison with -0.739859+0.286623 I attempted. >>
Greater::nord: Invalid comparison with -0.739859+0.286623 I attempted. >>
LessEqual::nord: Invalid comparison with -0.739859+0.286623 I attempted. >>
General::stop: Further output of LessEqual::nord will be suppressed during this calculation. >>
FindRoot::srect: Value If[-0.739859+0.286623 I<=0.,0.5 +0.1,-0.739859+0.286623 I] in search specification
    {Statistics‘NoncentralDistributionsDump‘x$308,Statistics‘NoncentralDistributionsDump‘startval[0.5,0.1,0.01]}
    is not a number or array of numbers. >>
Greater::nord: Invalid comparison with -0.739859+0.286623 I attempted. >>
General::stop: Further output of Greater::nord will be suppressed during this calculation. >>
FindRoot::srect: Value If[-0.739859+0.286623 I<=0.,0.5 +0.1,-0.739859+0.286623 I] in search specification
    {Statistics‘NoncentralDistributionsDump‘x$308,Statistics‘NoncentralDistributionsDump‘startval[0.5,0.1,0.01]}
    is not a number or array of numbers. >>
Quantile[NoncentralChiSquareDistribution[0.5,0.1],0.01]
```

This author forwarded the output to Wolfram, and $x_{\nu,\lambda}$ (5.9) was quickly incorporated into Mathematica 10.0.0 [88].

5.2.2 Pearson's approximation

E. S. Pearson gives an approximation to $q_{\nu,\lambda}$ in [63] (see also [39]) of the form $b + c\chi_f^2$, where

$$\begin{aligned} b &= -\frac{\lambda^2}{\nu + 3\lambda} \\ c &= \frac{\nu + 3\lambda}{\nu + 2\lambda} \\ f &= \frac{(\nu + 2\lambda)^3}{(\nu + 3\lambda)^2} = \frac{\nu + 2\lambda}{c^2} \end{aligned} \tag{5.14}$$

and χ_f^2 is the quantile of the central χ^2 with f degrees of freedom. The first three moments of $(b + c\chi_f^2)$ agree with those of the non-central χ^2 distribution. Figure 5.3 shows the backward relative error of Pearson's non-central χ^2 quantile approximation for $\nu = 5, \lambda = 1/2$ and $\nu = 1/2, \lambda = 1/2$. The approximation, like Sankaran's, is relatively poor for u near zero, but is otherwise relatively good.

For small enough u, Pearson's approximation will give a result less than zero. This effect is demonstrated in the second plot of Figure 5.3. To compute the non-central χ^2 quantile function, the Math Toolkit in Boost [14] up to and including version 1.54.0 uses Pearson's aforementioned approximation as an initial guess and refines it by root finding using the CDF (whose implementation is based on [23] and [10]). In the event of a negative initial guess, Boost would use a simple heuristic, setting the guess to the smallest positive number available, e.g. $\min_d$ for double-precision. Finding the root of $F_{\nu,\lambda}(x) - u = 0$ with such a crude guess is very inefficient. A better approximation for this case is given in Section 5.1. Boost 1.55.0 uses $x_{\nu,\lambda}$ due to private communication with this author.

5.2.3 Temme's median approximation

It is worth mentioning that Temme in [77] gives the relation

$$F_{\nu,\lambda}(\nu + \lambda) \approx 1/2, \tag{5.15}$$

which immediately yields an asymptotically good approximation for the non-central χ^2 median. When ν and λ are both close to zero, the true median is much smaller than $\nu + \lambda$. For example, if we fix $\nu = 1/10$ and allow λ to vary and vice versa, Figure 5.4 graphically compares the forward accuracy (2.13) of Temme's median approximation to that of (5.9) with $u = 1/2$. Our approximation from Section 5.1 clearly complements Temme's median approximation well.

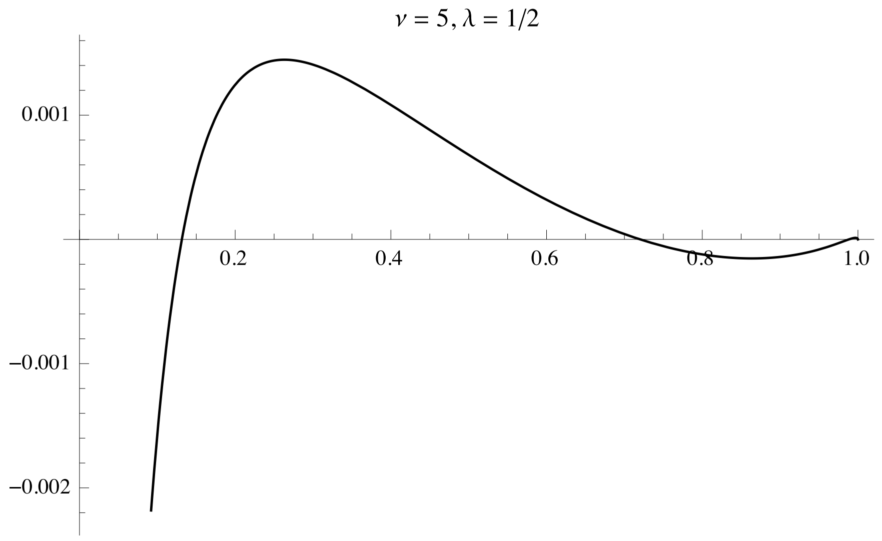
5.2.4 Ding's algorithm

There is very little on computing high-precision non-central χ^2 quantiles in the literature. In [24], an algorithm for this is given. The algorithm uses Newton's method, but the novelty is that the non-central χ^2 PDF and CDF are computed concurrently and only using the *central* χ^2 PDF and CDF. However, the algorithm employs a crude Cornish–Fisher initial guess to start its iterations. This author found similar issues with this approach to Sankaran's and Pearson's approximations. Substituting the Cornish–Fisher guess with the approximation we will develop in Section 5.4 cures this, simultaneously improving the speed and precision of Ding's algorithm.

5.3 Power series solution

We will now develop a series solution for the non-central χ^2 quantile function. The quantile function q of the non-central χ^2 distribution satisfies

$$\frac{\mathrm{d}^2 q}{\mathrm{d}u^2} = \left[\frac{2 + q - \nu}{2q} - \frac{\lambda}{4}\frac{{}_0\tilde{F}_1\left(;\frac{\nu}{2}+1;\frac{\lambda}{4}q\right)}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda}{4}q\right)}\right]\left(\frac{\mathrm{d}q}{\mathrm{d}u}\right)^2 \tag{5.16}$$



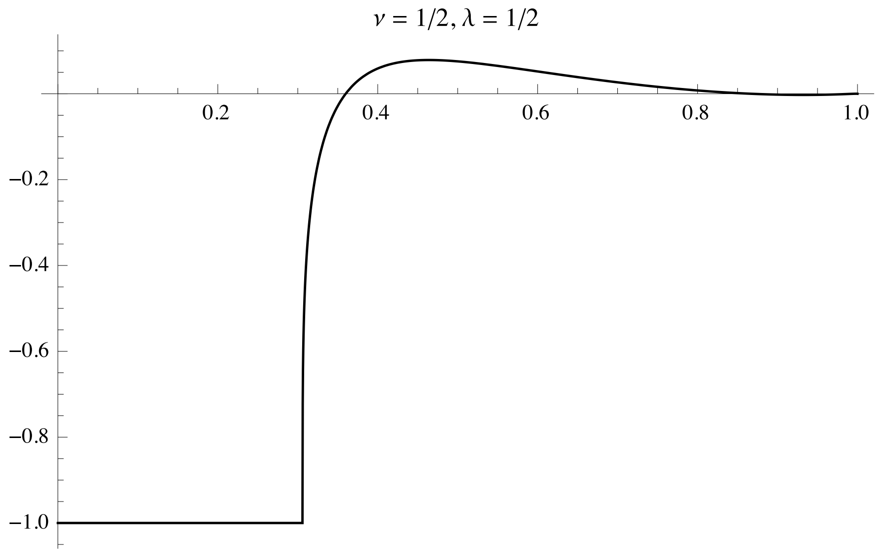
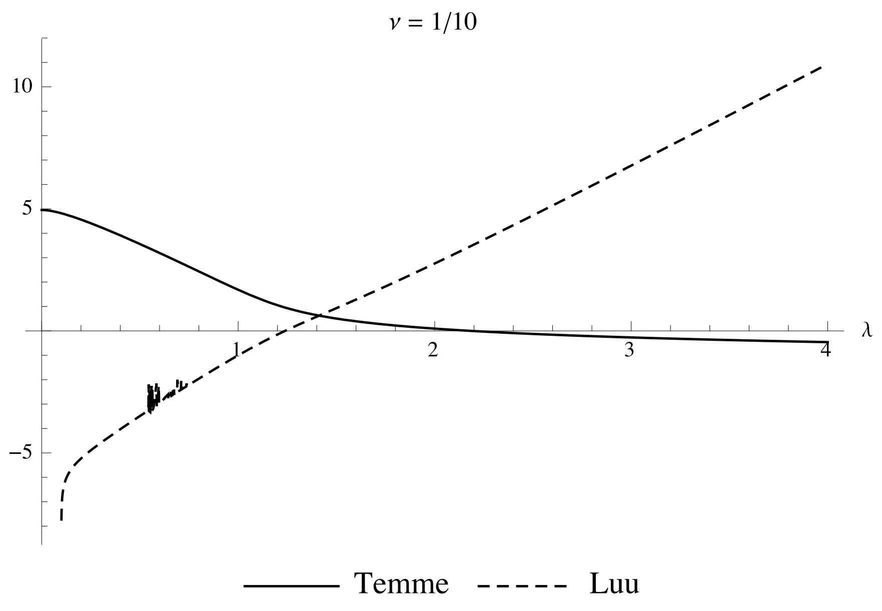


Figure 5.3: The backward relative errors of Pearson's non-central χ^2 quantile approximation for $\nu = 5, \lambda = 1/2$ and $\nu = 1/2, \lambda = 1/2$.



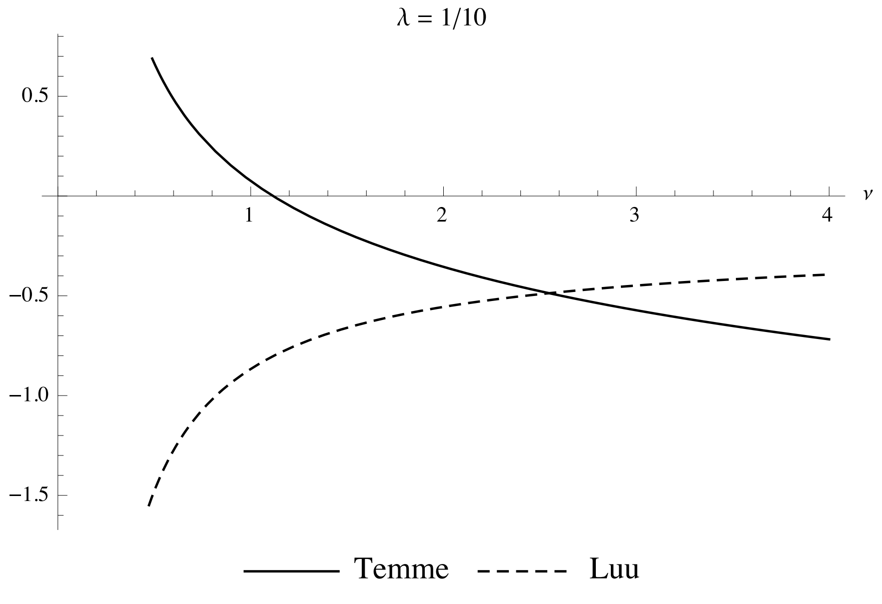


Figure 5.4: The $\log_{10}$ forward relative errors of Temme's median approximation $\nu + \lambda$ and $x_{\nu,\lambda}(1/2)$ for $\nu = 1/10$ and $\lambda = 1/10$.

conditional on $q(0) = 0$ and

$$q(u) \sim \left[2^{\nu/2-1} e^{\lambda/2} u \nu \Gamma(\nu/2)\right]^{2/\nu} \quad \text{as } u \to 0. \tag{5.17}$$

Let

$$v = \left[2^{\nu/2-1} e^{\lambda/2} u \nu \Gamma(\nu/2)\right]^{2/\nu}. \tag{5.18}$$

If

$$q(v) = \sum_{n=0}^{\infty} g_n v^n \tag{5.19}$$

with $g_0 = 0$ and $g_1 = 1$, then

$$\begin{aligned} n(2n+\nu)g_{n+1} = &\sum_{k=1}^{n-1} g_{k+1} g_{n-k+1} (n-k+1) \left[(2-\nu)(k+1) - 2(n-k) - (2-\nu)\right] \\ &+ a(n)(1-\lambda/\nu) - \frac{\lambda}{2} \sum_{i=1}^{n-1} a(i) f_{n-i} \end{aligned}, \tag{5.20}$$

where

$$a(i) = \sum_{j=1}^{i} \sum_{k=0}^{i-j} g_j g_{k+1} g_{i-j-k+1} (k+1)(i-j-k+1) \tag{5.21}$$

$$c_n(b) = \sum_{k=1}^{n} d_k(b) e_n(k) \tag{5.22}$$

$$d_k(b) = \begin{cases} \frac{1}{\Gamma(b)} & \text{if } k = 0 \\ \frac{d_{k-1}(b)}{(b+k-1)k} & \text{if } k > 0 \end{cases} \tag{5.23}$$

$$e_n(\alpha) = \begin{cases} \left(\frac{\lambda}{4}\right)^{\alpha} & \text{if } n = \alpha \\ \frac{1}{n-\alpha} \sum_{k=1}^{n-\alpha} ((\alpha+1)k - (n-\alpha)) g_{k+1} e_{n-k}(\alpha) & \text{if } n > \alpha \end{cases} \tag{5.24}$$

$$f_n = \Gamma(\nu/2) \left[c_n(\nu/2+1) - \frac{2}{\nu} c_n(\nu/2) - \sum_{k=1}^{n-1} f_k c_{n-k}(\nu/2)\right]. \tag{5.25}$$

The (lengthy but mechanical) derivation of this is in Appendix C. The first few terms are

$$\begin{aligned} g_1 &= 1 \\ g_2 &= \frac{\nu-\lambda}{\nu(\nu+2)} \\ g_3 &= \frac{4\lambda^2(\nu+3)-2\lambda\nu(3\nu+10)+\nu^2(3\nu+10)}{2\nu^2(\nu+2)^2(\nu+4)} \end{aligned} . \tag{5.26}$$

When $\nu = 2, \lambda = 0$, experimentation with many terms confirms that the coefficients are consistent with the series of the exact solution, $q(u) = -2\log(1-u)$ with $u = \frac{v}{2}$. Also, if $\nu = 1, \lambda = 0$ then the coefficients are in accordance with those of $2\left[\text{erf}^{-1}(u)\right]^2$, where $u = \sqrt{\frac{2v}{\pi}}$.

For general ν, λ, sanity checks are not as straightforward. However, for odd degrees of freedom, progress can be made. If $\nu = 2k+1$ with $k \in \mathbf{N}$, [42] proves that the non-central χ^2 PDF is

$$f_{\nu,\lambda}(x) = \left(\frac{2x}{\lambda}\right)^k \frac{\exp\left[-(x+\lambda)/2\right]}{\sqrt{2\pi x}} \frac{\mathrm{d}^k \cosh\sqrt{\lambda x}}{\mathrm{d}x^k}. \tag{5.27}$$

So

$$\begin{aligned} f_{1,\lambda}(x) &= \frac{e^{-\frac{\lambda}{2}-\frac{x}{2}}\cosh\left(\sqrt{\lambda x}\right)}{\sqrt{2\pi}\sqrt{x}} \\ f_{3,\lambda}(x) &= \frac{e^{-\frac{\lambda}{2}-\frac{x}{2}}\sinh\left(\sqrt{\lambda}\sqrt{x}\right)}{\sqrt{2\pi}\sqrt{\lambda}} \\ f_{5,\lambda}(x) &= \frac{e^{-\frac{\lambda}{2}-\frac{x}{2}}\left(\lambda x\cosh\left(\sqrt{\lambda x}\right)-\sqrt{\lambda x}\sinh\left(\sqrt{\lambda x}\right)\right)}{\sqrt{2\pi}\lambda^2\sqrt{x}} \\ f_{7,\lambda}(x) &= \frac{\sqrt{x}e^{-\frac{\lambda}{2}-\frac{x}{2}}\left((\lambda x+3)\sinh\left(\sqrt{\lambda x}\right)-3\sqrt{\lambda x}\cosh\left(\sqrt{\lambda x}\right)\right)}{\sqrt{2\pi}\lambda^2\sqrt{\lambda x}} \\ f_{9,\lambda}(x) &= \frac{\sqrt{x}e^{-\frac{\lambda}{2}-\frac{x}{2}}\left(\sqrt{\lambda x}(\lambda x+15)\cosh\left(\sqrt{\lambda x}\right)-3(2\lambda x+5)\sinh\left(\sqrt{\lambda x}\right)\right)}{\sqrt{2\pi}\lambda^3\sqrt{\lambda x}} \\ \ldots &= \ldots . \end{aligned} \tag{5.28}$$

The resultant PDFs can be integrated exactly. Finite series of the corresponding CDFs can hence be found and manually reversed. Numerous experiments have been conducted in *Mathematica* by this author, and perfect agreement of symbolic coefficients from the manually inverted series and the our recurrence relation always occurred.

5.4 A simple and general approximation

We will now supplement the existing approximations with our new power series solution, to form a more general and complete approximation. We will also be aiming for computational simplicity. Our algorithm would, therefore, be especially ideal for computing the initial guess in non-central χ^2 quantile function implementations.

5.4.1 Formulation

We favour Sankaran's approximation, given in Routine 8, over Pearson's, because the normal quantile function is easier to compute than the central χ^2 quantile function. The remaining question is when to use the power series solution and also how much of it to use.

Well, in the interests of simplicity we will actually only use the leading order term of the series. With regard to which approximant to use, if a Sankaran approximation is *not* real, we will unconditionally use the series solution. If the Sankaran approximation is real, the second non-zero term of the series will be used to decide which answer to use. More specifically, a cut-off point $u^* \in (0, 1)$ will be computed using the term. Now, if the input u is greater than u^*, the series solution will be rejected in favour of Sankaran's approximation. Routine 9 gives the pseudocode for computing the series solution and cut-off point. Finally, Algorithm 10, designated NCX2CDFINV, gives the complete procedure in pseudocode form.

Routine 8 `sankaran`

```
Input: $u \in (0,1), \nu > 0, \lambda \geqslant 0$.
Output: Sankaran's approximation of $q_{\nu,\lambda}(u)$.
1: $h \leftarrow 1 - \frac{2}{3}\frac{(\nu+\lambda)(\nu+3\lambda)}{(\nu+2\lambda)^2}$
2: $p \leftarrow \frac{\nu+2\lambda}{(\nu+\lambda)^2}$
3: $m \leftarrow (h-1)(1-3h)$
4: $\mu \leftarrow 1 + hp(h - 1 - (1 - h/2)mp)$
5: $\sigma \leftarrow h\sqrt{2p}(1 + mp/2)$
6: $z \leftarrow \Phi^{-1}(u)$
7: $x \leftarrow z\sigma + \mu$
8: return $(\nu + \lambda)x^{1/h}$
```

Routine 9 `luu`

Input: $u \in (0,1), \nu > 0, \lambda \geqslant 0$.
Output: Luu's approximation of $q_{\nu,\lambda}(u)$.

```
1: $\epsilon \leftarrow 1/10$
2: $c \leftarrow 2^{\nu/2-1} e^{\lambda/2} \nu \Gamma(\nu/2)$
3: $v \leftarrow [cu]^{2/\nu}$                                    ▷ $v(u)$
4: $g_2 \leftarrow \frac{\nu-\lambda}{\nu(\nu+2)}$
5: $g_3 \leftarrow \frac{4\lambda^2(\nu+3)-2\lambda\nu(3\nu+10)+\nu^2(3\nu+10)}{2\nu^2(\nu+2)^2(\nu+4)}$
6: if $\nu = \lambda$ then                                        ▷ $g_2 = 0$
7:     $h \leftarrow \sqrt[3]{\frac{\epsilon}{|g_3|}}$
8: else
9:     $h \leftarrow \sqrt{\frac{\epsilon}{|g_2|}}$
10: end if
11: $u^* \leftarrow \frac{1}{c} h^{\nu/2}$                         ▷ Using the inverse of $v(u)$.
12: return $(v, u^*)$
```

Algorithm 10 Non-central χ^2 quantile algorithm (`ncx2cdfinv`)

Input: $u \in (0,1), \nu > 0, \lambda \geqslant 0$.
Output: Hybrid approximation of $q_{\nu,\lambda}(u)$.

```
1: if $u = 0$ then
2:     return 0
3: end if
4: if $u = 1$ then
5:     return $\infty$
6: end if
7: $x_{\text{sankaran}} \leftarrow$ sankaran$(u, \nu, \lambda)$
8: $(x_{\text{luu}}, u^*) \leftarrow$ luu$(u, \nu, \lambda)$
9: if $x_{\text{sankaran}} \notin \mathbf{R}$ then     ▷ In practice, isnan would be used to test this.
10:     return $x_{\text{luu}}$
11: end if
12: if $u < u^*$ then
13:     return $x_{\text{luu}}$
14: else
15:     return $x_{\text{sankaran}}$
16: end if
```

5.4.2 Precision

We will now give an indication of the accuracy of Algorithm 10. The algorithm was implemented in double-precision for this purpose. 80-bit extended precision references were computed using Boost 1.57.0 and compared to the results from NCX2CDFINV. Table 5.1 shows average and peak backward relative errors (2.14) for a wide range of ν and λ pairs.

5.5 Commentary

In this chapter, we have described new advances in the computation of the non-central χ^2 quantile function. A new analytic approximation, which addresses deficiencies of existing approximations, was derived first. This led to a power series solution for the quantile function, which is interesting from a theoretical and practical point of view. Finally, a new algorithm, called NCX2CDFINV, was created by combining two approximations. We believe that NCX2CDFINV is the first non-central χ^2 quantile function algorithm to reliably cover the whole parameter and variable space of the distribution. No effort has been made to make the algorithm accurate to machine precision, but it is computationally efficient. We believe our algorithm would, therefore, be an excellent candidate for initial guesses in high-precision non-central χ^2 quantile algorithms and low-precision simulations.

Table 5.1: Average and peak backward relative error (2.14) statistics for an implementation of NCX2CDFINV, over 10^8 pseudo-random uniform variates for each ν and λ pair. Standard deviations are in brackets. Peak statistics are in **bold**.

	λ							
ν	10^{-4}	10^{-3}	10^{-2}	10^{-1}	10^{0}	10^{1}	10^{2}	10^{3}
10^{-4}	1.68×10^{-9} (1.51×10^{-7}) $\mathbf{3.46 \times 10^{-5}}$	1.24×10^{-7} (5.75×10^{-6}) $\mathbf{3.93 \times 10^{-4}}$	1.24×10^{-5} (2.01×10^{-4}) $\mathbf{4.86 \times 10^{-3}}$	1.19×10^{-3} (6.11×10^{-3}) $\mathbf{5.22 \times 10^{-2}}$	9.40×10^{-3} (1.42×10^{-2}) $\mathbf{4.91 \times 10^{-2}}$	1.22×10^{-2} (3.40×10^{-2}) $\mathbf{3.63 \times 10^{-1}}$	2.08×10^{-4} (5.80×10^{-4}) $\mathbf{5.79 \times 10^{-2}}$	5.57×10^{-6} (1.40×10^{-5}) $\mathbf{1.15 \times 10^{-3}}$
10^{-3}	2.15×10^{-7} (5.41×10^{-6}) $\mathbf{3.47 \times 10^{-4}}$	1.32×10^{-7} (4.12×10^{-6}) $\mathbf{3.46 \times 10^{-4}}$	1.23×10^{-5} (1.81×10^{-4}) $\mathbf{3.92 \times 10^{-3}}$	1.19×10^{-3} (6.05×10^{-3}) $\mathbf{6.15 \times 10^{-2}}$	9.43×10^{-3} (1.42×10^{-2}) $\mathbf{4.97 \times 10^{-2}}$	1.22×10^{-2} (3.41×10^{-2}) $\mathbf{3.63 \times 10^{-1}}$	2.08×10^{-4} (5.80×10^{-4}) $\mathbf{5.79 \times 10^{-2}}$	5.57×10^{-6} (1.40×10^{-5}) $\mathbf{1.15 \times 10^{-3}}$
10^{-2}	1.34×10^{-5} (1.44×10^{-4}) $\mathbf{1.09 \times 10^{-1}}$	1.33×10^{-5} (1.29×10^{-4}) $\mathbf{3.46 \times 10^{-3}}$	5.10×10^{-6} (6.62×10^{-5}) $\mathbf{3.46 \times 10^{-3}}$	1.19×10^{-3} (5.53×10^{-3}) $\mathbf{1.60 \times 10^{-1}}$	9.57×10^{-3} (1.42×10^{-2}) $\mathbf{5.29 \times 10^{-2}}$	1.22×10^{-5} (3.40×10^{-2}) $\mathbf{3.65 \times 10^{-1}}$	2.08×10^{-4} (5.79×10^{-4}) $\mathbf{5.78 \times 10^{-2}}$	5.57×10^{-6} (1.40×10^{-5}) $\mathbf{1.15 \times 10^{-3}}$
10^{-1}	1.68×10^{-3} (6.11×10^{-3}) $\mathbf{4.35 \times 10^{-2}}$	1.69×10^{-3} (6.16×10^{-3}) $\mathbf{4.38 \times 10^{-2}}$	2.10×10^{-3} (8.15×10^{-3}) $\mathbf{5.89 \times 10^{-2}}$	3.43×10^{-3} (1.72×10^{-2}) $\mathbf{1.39 \times 10^{-1}}$	9.86×10^{-3} (1.25×10^{-2}) $\mathbf{5.27 \times 10^{-2}}$	1.19×10^{-2} (3.33×10^{-2}) $\mathbf{3.65 \times 10^{-1}}$	2.07×10^{-4} (5.73×10^{-4}) $\mathbf{5.61 \times 10^{-2}}$	5.57×10^{-6} (1.40×10^{-5}) $\mathbf{1.14 \times 10^{-3}}$
10^{0}	1.95×10^{-2} (2.50×10^{-2}) $\mathbf{9.20 \times 10^{-2}}$	1.95×10^{-2} (2.49×10^{-2}) $\mathbf{9.20 \times 10^{-2}}$	1.94×10^{-2} (2.48×10^{-2}) $\mathbf{9.15 \times 10^{-2}}$	1.82×10^{-2} (2.34×10^{-2}) $\mathbf{8.73 \times 10^{-2}}$	4.90×10^{-3} (5.93×10^{-3}) $\mathbf{2.73 \times 10^{-2}}$	7.91×10^{-3} (1.86×10^{-2}) $\mathbf{2.22 \times 10^{-1}}$	1.96×10^{-4} (5.10×10^{-4}) $\mathbf{4.03 \times 10^{-2}}$	5.51×10^{-6} (1.36×10^{-5}) $\mathbf{1.06 \times 10^{-3}}$
10^{1}	4.70×10^{-3} (2.10×10^{-2}) $\mathbf{5.74 \times 10^{-1}}$	4.70×10^{-3} (2.10×10^{-2}) $\mathbf{5.74 \times 10^{-1}}$	4.70×10^{-3} (2.10×10^{-2}) $\mathbf{5.73 \times 10^{-1}}$	4.70×10^{-3} (2.10×10^{-2}) $\mathbf{5.70 \times 10^{-1}}$	4.97×10^{-3} (2.23×10^{-2}) $\mathbf{6.59 \times 10^{-1}}$	3.10×10^{-3} (1.22×10^{-2}) $\mathbf{3.29 \times 10^{-1}}$	1.33×10^{-4} (2.12×10^{-4}) $\mathbf{8.44 \times 10^{-2}}$	5.06×10^{-6} (1.03×10^{-5}) $\mathbf{2.77 \times 10^{-4}}$
10^{2}	3.22×10^{-4} (1.17×10^{-3}) $\mathbf{2.80 \times 10^{-1}}$	3.22×10^{-4} (1.17×10^{-3}) $\mathbf{2.80 \times 10^{-1}}$	3.22×10^{-4} (1.17×10^{-3}) $\mathbf{2.80 \times 10^{-1}}$	3.22×10^{-4} (1.17×10^{-3}) $\mathbf{2.80 \times 10^{-1}}$	3.23×10^{-4} (1.17×10^{-3}) $\mathbf{2.80 \times 10^{-1}}$	3.40×10^{-4} (1.23×10^{-3}) $\mathbf{2.96 \times 10^{-1}}$	2.23×10^{-4} (8.07×10^{-4}) $\mathbf{2.03 \times 10^{-1}}$	8.37×10^{-6} (2.28×10^{-5}) $\mathbf{5.95 \times 10^{-3}}$
10^{3}	2.96×10^{-5} (9.71×10^{-5}) $\mathbf{2.02 \times 10^{-2}}$	2.96×10^{-5} (9.71×10^{-5}) $\mathbf{2.02 \times 10^{-2}}$	2.96×10^{-5} (9.71×10^{-5}) $\mathbf{2.02 \times 10^{-2}}$	2.96×10^{-5} (9.71×10^{-5}) $\mathbf{2.02 \times 10^{-2}}$	2.96×10^{-5} (9.71×10^{-5}) $\mathbf{2.02 \times 10^{-2}}$	2.96×10^{-5} (9.72×10^{-5}) $\mathbf{2.02 \times 10^{-2}}$	3.11×10^{-5} (1.02×10^{-4}) $\mathbf{2.12 \times 10^{-2}}$	2.07×10^{-5} (6.78×10^{-5}) $\mathbf{1.44 \times 10^{-2}}$

Chapter 6

Skew-normal distribution

The normal distribution is symmetric about its mean. Azzalini's skew-normal distribution [5] is a generalisation of the normal distribution that allows for non-zero skewness. The PDF of this skew-normal distribution is

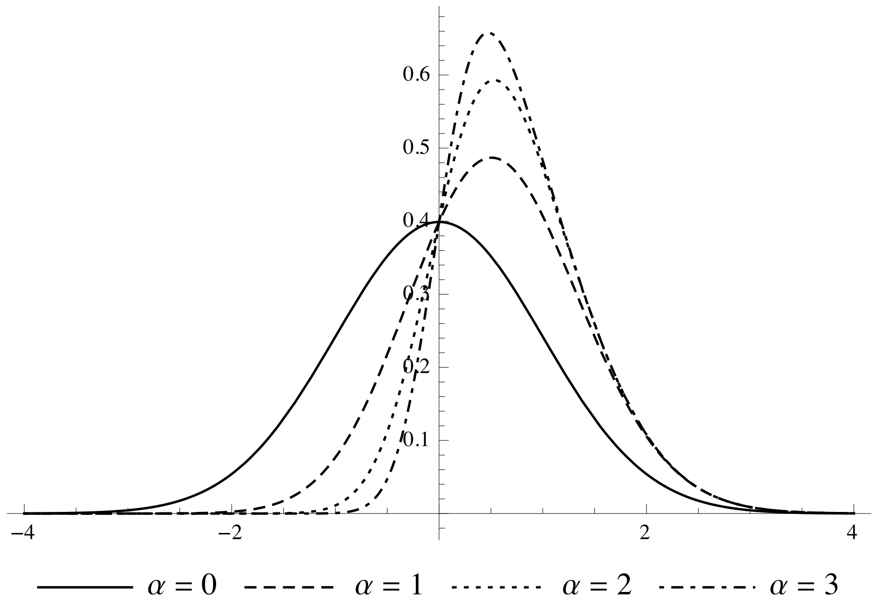
$$f_{\mu,\sigma,\alpha}(x) = \frac{2}{\sigma}\phi\left(\frac{x-\mu}{\sigma}\right)\Phi\left(\alpha\frac{x-\mu}{\sigma}\right), \tag{6.1}$$

where μ and σ are location and scale parameters as per the normal distribution and $\alpha \in \mathbf{R}$ is the shape parameter that controls skewness. The functions ϕ and Φ are the PDF and CDF of the standard normal distribution, as defined in (3.2) and (3.5). As in the normal distribution case, the location and scale parameters are not of material concern to us because we can assume $\mu = 0$ and $\sigma = 1$ while approximating the skew-normal quantile function. We will thus take the skew-normal PDF to be

$$f_{\alpha}(x) = 2\,\phi\,(x)\,\Phi\,(\alpha x)\,. \tag{6.2}$$

When $\alpha = 0$, the skew-normal distribution reduces to the standard normal distribution (since $\Phi(0) = 1/2$). The skew-normal distribution for $-\alpha$ is the mirror of that for α, so we will restrict our attention to $\alpha > 0$. Figure 6.1 shows the skew-normal PDF for various $\alpha \geqslant 0$. It is worth mentioning that as $\alpha \to \infty$, the skew-normal distribution tends to the half-normal distribution with density

$$\sqrt{\frac{2}{\pi}}e^{-\frac{x^2}{2}}, \tag{6.3}$$



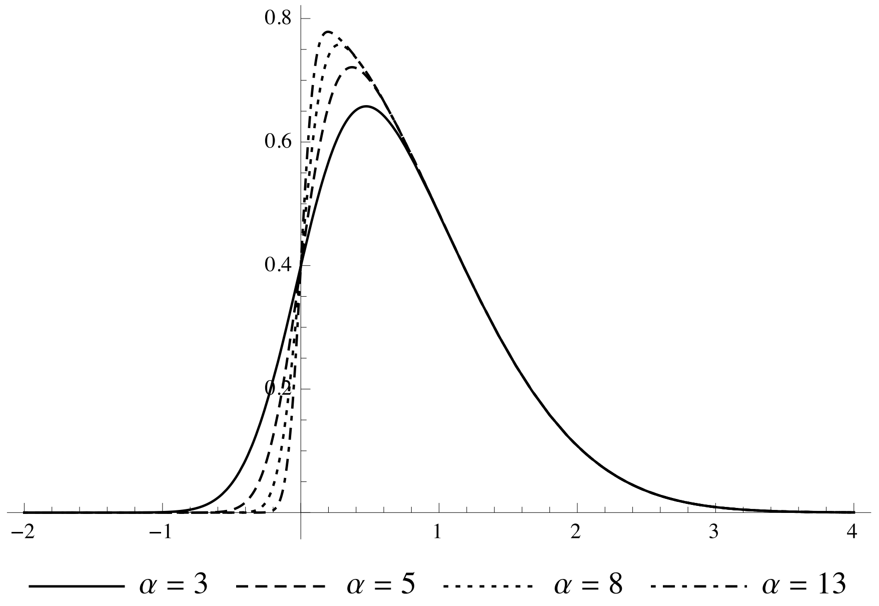


Figure 6.1: The skew-normal PDF for various $\alpha \geqslant 0$.

whose CDF is easily invertible. The quantile function of the skew-normal distribution with $\alpha = 1$ can be handled with ease since

$$f_1(x) = \frac{\mathrm{d}}{\mathrm{d}x}\Phi^2(x). \tag{6.4}$$

For $\alpha \notin \{-1, 0, 1\}$, the skew-normal quantile function is not known to have a closed-form expression. However, the skew-normal CDF can be expressed in terms of Owen's T function, which is defined as

$$T(x, a) = \frac{1}{2\pi}\int_0^a \frac{e^{-x^2(1+t^2)/2}}{1+t^2}\,\mathrm{d}t, \tag{6.5}$$

with $x, a \in \mathbf{R}$. Let

$$F_\alpha(x) = \Phi(x) - 2\,T(x, \alpha) \tag{6.6}$$

be the skew-normal CDF. From (6.5), we have

$$T(0, a) = \frac{1}{2\pi}\arctan(a), \tag{6.7}$$

which implies that

$$F_\alpha(0) = \frac{1}{2} - \frac{\arctan(\alpha)}{\pi}. \tag{6.8}$$

This leads to a general quantile result, namely that the skew-normal quantile function is precisely zero at

$$u_0 = \frac{1}{2} - \frac{\arctan(\alpha)}{\pi} \tag{6.9}$$

for all shape parameters α. We will use this to develop a series solution for the skew-normal quantile function momentarily. Before we look at this, it is worth saying that skew-normal random variates can be easily generated without inverting $F_\alpha(x)$. If z_0, z_1 are standard normal variates with correlation δ, then

$$x = \begin{cases} z_1 & \text{if } z_0 \geqslant 0 \\ -z_1 & \text{otherwise} \end{cases} \tag{6.10}$$

is a random variate of the skew-normal distribution with $\alpha = \delta/\sqrt{1-\delta^2}$ (see, e.g., [3]).

6.1 Central power series solution

Let $q_\alpha(u) = F_\alpha^{-1}$ denote the skew-normal quantile function. In the interests of brevity, q will be used instead of q_α when there is no ambiguity. For $0 < u < 1$ we have

$$\frac{\mathrm{d}q}{\mathrm{d}u} = \frac{\sqrt{2\pi}\exp\left(\frac{q^2}{2}\right)}{\operatorname{erfc}\left(-\frac{\alpha q}{\sqrt{2}}\right)}, \tag{6.11}$$

conditional on $q(u_0) = 0$. Before proceeding, it actually turns out to be advantageous to make the change of variable

$$v = \Phi^{-1}(u), \tag{6.12}$$

which maps the ODE to

$$\frac{\mathrm{d}q}{\mathrm{d}v} = \frac{\exp\left(\frac{1}{2}\left(q^2 - v^2\right)\right)}{\operatorname{erfc}\left(-\frac{\alpha q}{\sqrt{2}}\right)}. \tag{6.13}$$

This is conditional on $q(\Phi^{-1}(u_0)) = 0$. The presence of the dependent variable as a function of erfc in the ODE complicates matters somewhat. To deal with this, we will work out the series of $\operatorname{erfc}(x(t))$ where x is an analytic function in t.

The complementary error function $\operatorname{erfc}(z)$ satisfies (see, e.g., [79]) the linear ODE

$$w''(z) + 2zw'(z) = 0 \tag{6.14}$$

where

$$\begin{aligned} w(z) &= \operatorname{erfc}(z) \\ w(0) &= 1 \\ w'(0) &= -\frac{2}{\sqrt{\pi}}. \end{aligned} \tag{6.15}$$

Setting $z = x(t)$, where x is an analytic function in t, the chain and quotient rules give

$$w'(z) = w'(t)/u'(t) \tag{6.16}$$

and

$$w''(z) = \frac{u'(t)w''(t) - w'(t)u''(t)}{u'(t)^3}. \tag{6.17}$$

Therefore $\operatorname{erfc}(x(t))$ satisfies

$$w''(t) = w'(t)\frac{x''(t)}{x'(t)} - 2x(t)w'(t)x'(t) \tag{6.18}$$

where

$$\begin{aligned} w(t) &= \operatorname{erfc}(x(t)) \\ w(t_0) &= \operatorname{erfc}(x(t_0)) \\ w'(t_0) &= -\frac{2}{\sqrt{\pi}}e^{-x(t_0)^2}x'(t_0). \end{aligned} \tag{6.19}$$

Let

$$x(t) = \sum_{i=0}^{\infty}(x)_i(t-t_0)^i. \tag{6.20}$$

Applying appropriate recurrence relations to $w''(t)$ and adjusting for x' and x'' where necessary, we obtain

$$\begin{aligned} (T_1)_k &= -2(x)_k, \\ (T_2)_k &= \sum_{j=0}^{k}(w')_j(T_1)_{k-j}, \\ (T_3)_k &= \sum_{j=0}^{k}(j+1)(x)_{j+1}(T_2)_{k-j}, \\ (T_4)_k &= \sum_{j=0}^{k}(j+1)(j+2)(x)_{j+2}(w')_{k-j}, \\ (T_5)_k &= \frac{1}{(x)_1}\left[(T_4)_k - \sum_{j=1}^{k}(j+1)(x)_{j+1}(T_5)_{k-j}\right], \\ (T_6)_k &= (T_5)_k + (T_3)_k, \\ (w)_k &= \begin{cases} \operatorname{erfc}((x)_0) & \text{if } k=0 \\ \frac{1}{k}(w')_{k-1} & \text{otherwise} \end{cases}, \\ (w')_k &= \begin{cases} -\frac{2}{\sqrt{\pi}}e^{-(x)_0^2}(x)_1 & \text{if } k=0 \\ \frac{1}{k}(T_6)_{k-1} & \text{otherwise} \end{cases}. \end{aligned} \tag{6.21}$$

After some simplification, we have

$$w(t) = \operatorname{erfc}(x(t)) = \sum_{i=0}^{\infty} (w)_i (t - t_0)^i \tag{6.22}$$

where

$$\begin{aligned}
(w)_n &= \frac{1}{n(n-1)(x)_1} \left[\sum_{i=0}^{n-2} \delta_i (i+1)(w)_{i+1} \left[(n-i-1)(n-i)(x)_{n-i} - 2D_n(i) \right] - E_n \right] \\
D_n(i) &= \sum_{k=0}^{n-i-2} (x)_{n-2-i-k} \sum_{j=0}^{k} (k-j+1)(x)_{k-j+1}(j+1)(x)_{j+1} \\
E_n &= \sum_{j=1}^{n-2} (j+1)(x)_{j+1}(n-j-1)(n-j)(w)_{n-j}
\end{aligned} \tag{6.23}$$

We can now develop the series solution for the skew-normal quantile function. Let

$$q(v) = \sum_{n=0}^{\infty} c_n (v - v_0)^n. \tag{6.24}$$

The coefficients of the series can be computed with the following recurrence relations.

$$
\begin{aligned}
B_n &= \begin{cases} v^2 & \text{if } n = 0 \\ 2v & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 0 & \text{otherwise} \end{cases} \\
A_n &= \frac{1}{2} \sum_{j=0}^{n} c_{n-j} c_j - B_j \\
a_n &= \begin{cases} \exp(A_0) & \text{if } n = 0 \\ \frac{1}{n} \sum_{i=0}^{n-1} (n-i) a_i A_{n-i} & \text{otherwise} \end{cases} \\
b_n &= -\frac{\alpha c_n}{\sqrt{2}} \\
d_n(i) &= \sum_{k=0}^{n-i-2} b_{n-2-i-k} \sum_{j=0}^{k} (k-j+1) b_{k-j+1} (j+1) b_{j+1} \\
e_n &= \sum_{j=1}^{n-2} (j+1) b_{j+1} (n-j-1)(n-j) f_{n-j} \\
f_n &= \begin{cases} \operatorname{erfc}(b_0) & \text{if } n = 0 \\ -\frac{2}{\sqrt{\pi}} e^{-b_0^2} b_1 & \text{if } n = 1 \\ \frac{1}{n(n-1)b_1} \left[\sum_{i=0}^{n-2} \delta_i (i+1) f_{i+1} \left[(n-i-1)(n-i) b_{n-i} - 2 d_n(i) \right] - e_n \right] & \text{otherwise} \end{cases} \\
g_n &= \frac{1}{f_0} \left[a_n - \sum_{i=1}^{n} f_i g_{n-i} \right] \\
c_n &= \begin{cases} q(v_0) & \text{if } n = 0 \\ \frac{1}{n} g_{n-1} & \text{otherwise} \end{cases}
\end{aligned}
\tag{6.25}
$$

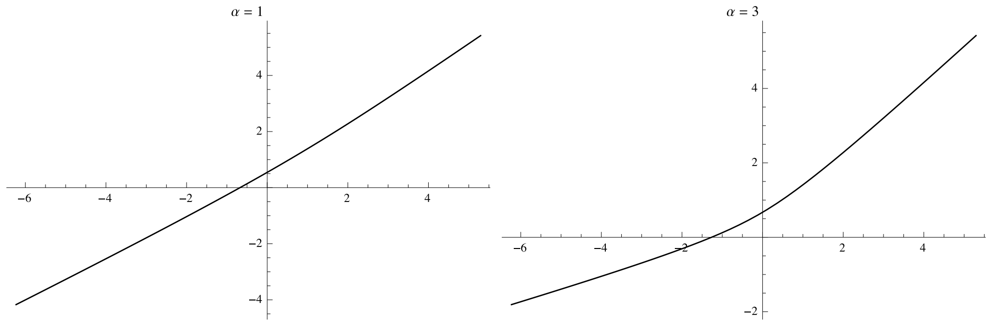


Figure 6.2: The normal to skew-normal transformation for $\alpha = 1$ and $\alpha = 3$.

For $v_0 = \Phi^{-1}(u_0)$, when $q(v_0) = 0$, we have

$$
\begin{aligned}
c_0 &= 0\\
c_1 &= e^{-\frac{v_0^2}{2}}\\
c_2 &= -\frac{1}{2}e^{-v_0^2}\left(\sqrt{\frac{2}{\pi}}\alpha + e^{\frac{v_0^2}{2}}v_0\right)\\
c_3 &= \frac{e^{-\frac{1}{2}\left(3v_0^2\right)}\left(3\alpha\left(2\alpha+\sqrt{2\pi}e^{\frac{v_0^2}{2}}v_0\right)+\pi e^{v_0^2}\left(v_0^2-1\right)+\pi\right)}{6\pi}.
\end{aligned}
\tag{6.26}
$$

Moreover, when $\alpha = 1$, experimentation with many terms confirms that the coefficients are consistent with the series of the exact solution, $q(v) = \Phi^{-1}(\sqrt{\Phi(v)})$, about $v = \Phi^{-1}(1/4)$.

Due to the Gaussian change of variable, $q(v)$ becomes more linear as $\alpha \to 0$. The applicability of the series solution that we have just developed is, therefore, widest when $\alpha \approx 0$. As we increase the magnitude of α, $q(v)$ appears more and more boomerang-shaped. This effect is shown in Figure 6.2. For $|\alpha| \lessapprox 1$, it is evident that $q(v)$ is a near linear function. In this case, our power series really comes into its own. For example, for $\alpha = 1/2$, the series coefficients, to twenty significant figures, are

$$\begin{aligned}
c_0 &= 0 \\
c_1 &= 0.93076675216545461682 \\
c_2 &= 0.0034823090353740146853 \\
c_3 &= 0.00023027801455712131008 \\
c_4 &= 1.9142647842329812281 \times 10^{-6} \\
c_5 &= -1.9389408012460763489 \times 10^{-6} \\
c_6 &= -2.6830046692737864968 \times 10^{-7} \\
c_7 &= -1.0834703556008779814 \times 10^{-8} \\
c_8 &= 2.2647847528405086592 \times 10^{-9} \\
c_9 &= 4.8950228045400391666 \times 10^{-10} \\
c_{10} &= 3.7564194584734528925 \times 10^{-11}.
\end{aligned} \tag{6.27}$$

Using these eleven terms yields approximations with a relative accuracy of less than 2.1×10^{-5} on $v \in [v_0, 4]$, which is equivalent to

$$0.352416 \approx \frac{1}{2} - \frac{\arctan(1/2)}{\pi} \leqslant u \leqslant \Phi(4) \approx 0.999968,$$

and less than 2.0×10^{-8} on $v \in [-4, v_0]$. In contrast, if $\alpha = 3$ then the relative accuracy is less than 9.3×10^{-4} over $v \in [-3, 0]$.

6.2 Tail approximations

Figure 6.1 suggests that the behaviour of the left and right tails of the skew-normal distribution are quite distinct. Let us consider the first order skew-normal quantile ODE (6.11) in an asymptotic setting. As $u \to 1$,

$$\frac{\mathrm{d}q}{\mathrm{d}u} \sim \sqrt{\frac{\pi}{2}} \exp\left(\frac{q^2}{2}\right), \tag{6.28}$$

with the condition that $q(1) = \infty$. The solution to this is just

$$\sqrt{2}\, \mathrm{erf}^{-1}(u). \tag{6.29}$$

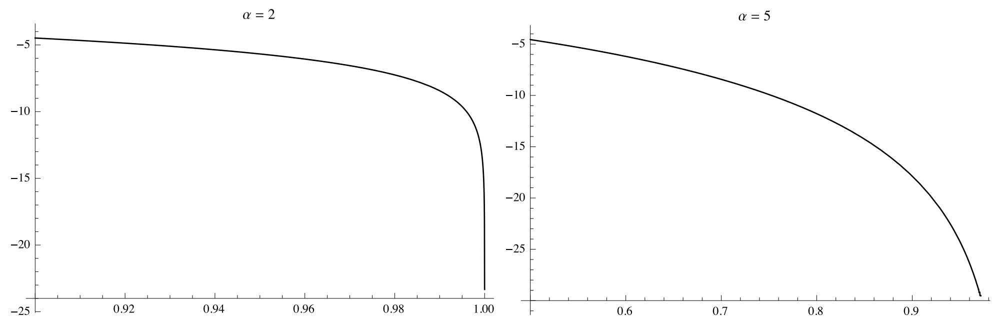


Figure 6.3: The $\log_{10}$ relative errors of the skew-normal quantile approximation $\sqrt{2}\,\mathrm{erf}^{-1}(u)$ for $\alpha = 2$ (on $9/10 \leqslant u < 1$) and $\alpha = 5$ (on $5/10 \leqslant u < 1$).

This, alone, turns out to be a very good approximation to the right tail of the skew-normal distribution. Note the similarity to the normal quantile function. While, naturally, getting better as $u \to 1$, (6.29) also becomes more and more viable as $\alpha \to \infty$. Figure 6.3 illustrates this point. The derivative of $F_\alpha(\sqrt{2}\,\mathrm{erf}^{-1}(u))$ is

$$\frac{1 + \mathrm{erf}(\alpha\,\mathrm{erf}^{-1}(u))}{2}. \tag{6.30}$$

Consequently, the difference of this quantity from one is a measure of how accurate our tail approximation is to $q_\alpha(u)$. Solving

$$\epsilon = \frac{\mathrm{erfc}(\alpha\,\mathrm{erf}^{-1}(u))}{2} \tag{6.31}$$

for u allows one to find, for a particular α and tolerance ϵ, the range of u that can be managed by (6.29). The upper limit of the approximation (6.29) for ϵ is

$$\mathrm{erf}\left[\frac{\mathrm{erfc}^{-1}(2\epsilon)}{\alpha}\right]. \tag{6.32}$$

A formula for the case $u \to 0$ cannot be derived using the same method as for $u \to 1$, so another route must be used.

For the left tail of the skew-normal distribution with $\alpha > 0$, [17] proves the inequality

$$\sqrt{\frac{2}{\pi}}\frac{\phi(x\sqrt{1+\alpha^2})}{|\alpha|\,(1+\alpha^2)x^2} - \sqrt{\frac{2}{\pi}}\left(2 + \frac{1+\alpha^2}{\alpha^2}\right)\frac{\phi(x\sqrt{1+\alpha^2})}{|\alpha|\,(1+\alpha^2)^2x^4} < F_\alpha(x) < \sqrt{\frac{2}{\pi}}\frac{\phi(x\sqrt{1+\alpha^2})}{|\alpha|\,(1+\alpha^2)x^2}. \tag{6.33}$$

The upper bound is interesting to us, because it is invertible. To see this, recall the

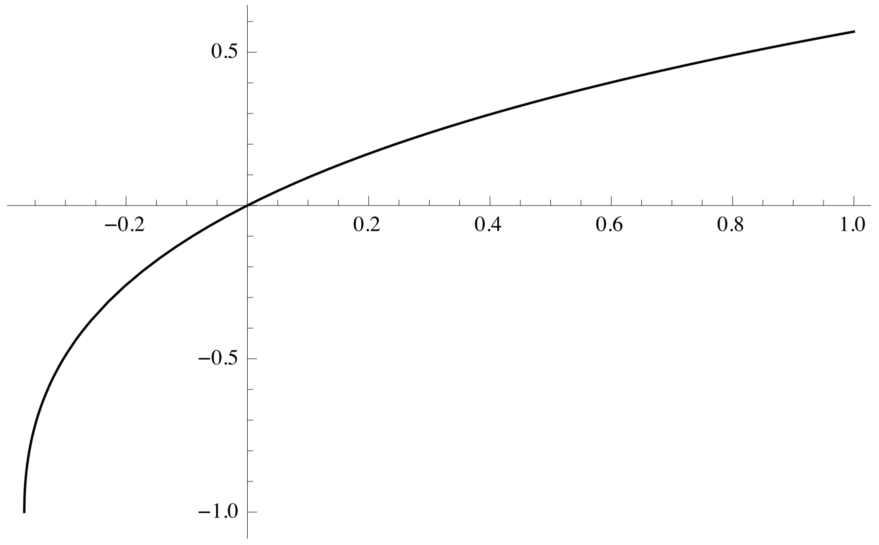


Figure 6.4: The Lambert W-function $W(z)$ for $-1/e \leqslant z \leqslant 1$.

definition of the normal PDF ϕ. So

$$F_\alpha(x) < \sqrt{\frac{2}{\pi}} \frac{\phi(x\sqrt{1+\alpha^2})}{|\alpha|\,(1+\alpha^2)x^2} = \frac{e^{-\frac{1}{2}(1+\alpha^2)x^2}}{\pi\,|\alpha|\,(1+\alpha^2)x^2}. \tag{6.34}$$

Let $u = F_\alpha(x)$, then, assuming the upper bound is reasonably sharp,

$$\frac{1}{2}(1+\alpha^2)x^2 e^{\frac{1}{2}(1+\alpha^2)x^2} \approx \frac{1}{2\pi u\,|\alpha|} \tag{6.35}$$

$$\Longrightarrow x \approx \pm\sqrt{\frac{2\,W\!\left(\frac{1}{2\pi u|\alpha|}\right)}{1+\alpha^2}} \tag{6.36}$$

where $W(z)$ is the Lambert W-function[1]. The negative solution happens to be the approximation we want. We found that it gives up to around three significant figures of accuracy.

The Lambert W-function, also known as the product log function, is a relatively simple quantity to compute. Figure 6.4 gives a plot of $W(z)$ for $-1/e \leqslant z \leqslant 1$. Since $W(z)$ is the solution to

$$f(w) = we^w - z = 0, \tag{6.37}$$

[1]$W(z)$ gives the principal solution for w in $z = w\exp(w)$.

and

$$\begin{aligned} f'(w) &= e^w(1+w) \\ \frac{f''(w)}{f'(w)} &= \frac{2+w}{1+w}, \end{aligned} \tag{6.38}$$

the use of Halley's method in the form

$$w_{n+1} = w_n - \frac{f(w_n)}{f'(w_n)} \left[1 - \frac{f(w_n)f''(w_n)}{2f'(w_n)^2}\right]^{-1} \tag{6.39}$$

is very natural. The question now is how to choose the initial guess w_0. Power series expansions for $W(z)$ at $z = 0, -1/e, \infty$ are certainly suitable, but better formulae are available.

We adopt the analytic approximations in [7]. For $-1/e \leqslant z \leqslant 0$,

$$W(z) = -1 + \frac{\sqrt{\eta}}{1 + ((N_1\sqrt{\eta})/(N_2 + \sqrt{\eta}))} \tag{6.40}$$

is given, where

$$\eta = 2 + 2ez \tag{6.41}$$

and

$$N_1 = \left(1 - \frac{1}{\sqrt{2}}\right)(N_2 + \sqrt{2}). \tag{6.42}$$

If, as per [8],

$$N_2 = 3\sqrt{2} + 6 - \frac{[(2237 + 1457\sqrt{2})e - 4108\sqrt{2} - 5764]\sqrt{\eta}}{(215 + 199\sqrt{2})e - 430\sqrt{2} - 796}, \tag{6.43}$$

then approximations with a relative error of at least 0.013% will be yielded, which is excellent. For $z \geqslant 0$, [7] gives

$$W(z) \approx \log\left\{\frac{6}{5}\frac{z}{\log[(12/5)(z/\log(1 + (12z/5)))]}\right\}, \tag{6.44}$$

which is quoted to have a maximum relative error of 2.39%. There are more complicated approximations in [7] that offer more accuracy, but a balance between the computational effort for the initial guess and subsequent refinements has to be struck. We found the aforementioned pair of approximants to be good enough. Two or three Halley iterations are typically sufficient for double-precision results.

For completeness, Routine 11 gives the pseudocode for computing $W(z)$.[2]

Routine 11 Lambert W-function

```
Input: $z \in [-1/e, \infty)$
Output: $W(z)$
 1: if $z = -1/e$ then
 2:     return $-1$
 3: end if
 4: if $z = 0$ then
 5:     return 0
 6: end if
 7: if $z > 0$ then
 8:     $w_0 \leftarrow \log\left\{\frac{6}{5}\frac{z}{\log[(12/5)(z/\log(1+(12z/5)))]}\right\}$
 9: else
10:     $\eta \leftarrow 2 + 2ez$
11:     $N_2 \leftarrow 3\sqrt{2} + 6 - \frac{[(2237+1457\sqrt{2})e-4108\sqrt{2}-5764]\sqrt{\eta}}{(215+199\sqrt{2})e-430\sqrt{2}-796}$
12:     $N_1 \leftarrow \left(1 - \frac{1}{\sqrt{2}}\right)(N_2 + \sqrt{2})$
13:     $w_0 \leftarrow -1 + \frac{\sqrt{\eta}}{1+((N_1\sqrt{\eta})/(N_2+\sqrt{\eta}))}$
14: end if
15: while true do
16:     $f \leftarrow w_0 \exp(w_0) - z$
17:     $w_1 \leftarrow w_0 - f/((\exp(w_0)(w_0+1) - (w_0+2)f/(2w_0+2)))$
18:     if $|w_0/w_1 - 1| < \epsilon$ then          ▷ $\epsilon$ represents machine epsilon here
19:         break
20:     end if
21:     $w_0 \leftarrow w_1$
22: end while
23: return $w_1$
```

In summary, we now have tail approximations for both ends of the skew-normal distribution. The four cases are outlined below.

$(\alpha > 0, u \to 1)$

$$\sqrt{2}\,\mathrm{erf}^{-1}(u) \tag{6.45}$$

$(\alpha > 0, u \to 0)$

$$-\sqrt{\frac{2\,W\left(\frac{1}{2\pi u|\alpha|}\right)}{1+\alpha^2}} \tag{6.46}$$

[2]An open-source implementation of the Lambert W-function algorithm is available from the public GitHub repository https://github.com/thomasluu/plog.

$(\alpha < 0, u \to 0)$

$$-\sqrt{2}\,\operatorname{erfc}^{-1}(u) \tag{6.47}$$

$(\alpha < 0, u \to 1)$

$$\sqrt{\frac{2\,W\left(\frac{1}{2\pi(1-u)|\alpha|}\right)}{1+\alpha^2}} \tag{6.48}$$

In the last case there is no loss of precision, because u is not near zero. The nature of these tail approximations means that they are also general upper and lower bounds for skew-normal quantiles. There is overlap between these tail approximations and our central power series solution. We will use this observation in Section 6.4 to create a complete algorithm for the skew-normal quantile function. Let us first review the prevailing method to approximate skew-normal quantiles.

6.3 Cornish–Fisher approximations

The use of Cornish–Fisher expansions [34], which are in terms of cumulants, are a natural method to crudely approximate the skew-normal quantile function. Indeed, Azzalini's skew-normal software package [4] (written in R) and the Boost C++ Math Toolkit [14] both use a Cornish–Fisher expansion with the first four cumulants of the (normalised) skew-normal distribution. The approximation that uses this particular expansion is

$$q_\alpha(z) \approx \sigma\left(z + \frac{z^2-1}{6}\kappa_3 + \frac{z^3-3z}{24}\kappa_4 - \frac{2z^3-5z}{36}\kappa_3^2\right) + \mu, \tag{6.49}$$

where $z = \Phi^{-1}(u)$ and

$$\begin{aligned}
\mu &= \frac{\sqrt{\frac{2}{\pi}}\alpha}{\sqrt{\alpha^2+1}} \\
\sigma &= \sqrt{1 - \frac{2\alpha^2}{\pi\alpha^2+\pi}} \\
\kappa_3 &= -\frac{\sqrt{2}(\pi-4)\alpha^3}{(\alpha^2+1)^{3/2}\left(\frac{2}{\alpha^2+1}+\pi-2\right)^{3/2}} \\
\kappa_4 &= \frac{8(\pi-3)\alpha^4}{((\pi-2)\alpha^2+\pi)^2}.
\end{aligned} \tag{6.50}$$

The Cornish–Fisher approximations are then refined with Newton's method. Boost uses the algorithm from [62] to implement the the skew-normal CDF.

The Cornish–Fisher approximation that uses the first five cumulants is

$$q_\alpha(z) \approx \sigma\left(z + \frac{z^2-1}{6}\kappa_3 + \frac{z^3-3z}{24}\kappa_4 - \frac{2z^3-5z}{36}\kappa_3^2 \right.$$
$$\left. + \frac{z^4-6z^2+3}{120}\kappa_5 - \frac{z^4-5z^2+2}{24}\kappa_3\kappa_4 + \frac{12z^4-53z^2+17}{324}\kappa_3^3\right) + \mu, \tag{6.51}$$

where

$$\kappa_5 = \frac{\sqrt{2}(\pi(3\pi-40)+96)\alpha^5}{(\alpha^2+1)^{5/2}\left(\frac{2}{\alpha^2+1}+\pi-2\right)^{5/2}}. \tag{6.52}$$

Cornish–Fisher expansions are asymptotic in nature, so increasing the number of cumulants used will, of course, not necessarily yield more accurate results. Moreover, the quality of the (tail) approximations deteriorate as $z \to \pm\infty$. This is a well known issue, which will be addressed in the next section.

6.4 A new numerical algorithm

We will now combine our power series solution and tail approximations to develop an improved algorithm for the skew-normal quantile function. The combination of the polynomial and analytic approximants means that inversion is very computationally efficient, comparable to that of Cornish–Fisher expansions.

6.4.1 Algorithm design

For a given u, α pair, we are able to compute, using (6.32), the point from which the right tail approximation (6.45) can be used. However, we cannot do this for the left tail approximation (6.46). In the light of this, the power series solution of $Q(v) = q(\Phi(v))$ is formed. The recurrence relation from Section 6.1 is used to compute the required coefficients. The symbolic form of the coefficients can be pre-computed or computed on-the-fly. We chose to do the former. In any case, the last coefficient computed is used to estimate the radius of convergence. If $v = \Phi^{-1}(u)$ is outside this range, we switch to the left tail approximation. Using the last coefficient is justified, because it is precisely what adaptive ODE solvers such as the Runge–Kutta–Fehlberg method [26] are based on. We calculate the difference between a Taylor series approximation of order n from one of order $n-1$

to gauge the error estimate. This quantity is

$$\epsilon \approx |c_n|\, h^n, \tag{6.53}$$

where h is the step-size, so, allowing for a generous safety factor, we take the radius of convergence as

$$h \approx \frac{3}{4} \left| \frac{\epsilon}{c_n} \right|^{1/n}. \tag{6.54}$$

Algorithm 12, designated SNCDFINV, gives the pseudocode for all of this.

6.4.2 Computational experiments

We will now demonstrate the parallel performance of our skew-normal quantile function algorithm SNCDFINV. This was done in accordance with Section 2.5. Our implementation uses the first five non-zero coefficients of the central series solution. The symbolic form of $c_1, c_2, \ldots, c_5$ were generated for this purpose using (6.25). All computations were implemented in double-precision. The performance of SNCDFINV was evaluated on three high-end Nvidia GPUs:

- a GeForce GTX Titan;
- a Tesla K80; and
- a Tesla C2050.

The C2050 is based on Nvidia's previous generation Fermi architecture, while the other two are Kepler GPUs. We used CUDA 6.5.14 software, the most recent production release at the time of writing.

We benchmarked the performance of SNCDFINV against the Cornish–Fisher expansions (6.49) and (6.51). We will denote these by CF4 and CF5 respectively.

Speed

Table 6.1 shows the performance of SNCDFINV, CF5 and CF4 for a wide range of shape parameters. The performance of SNCDFINV is always within an order of magnitude of the time to compute CF5 and CF4.

Precision

The precision of SNCDFINV, CF5 and CF4 were assessed by inspecting the output from ten out of the 100 runs of each speed test. The skew-normal output and cor-

Algorithm 12 Skew-normal quantile algorithm (`sncdfinv`)

```
Input: $u \in (0,1), \alpha \in \mathbf{R}$.
Output: $q_\alpha(u)$
 1: $\epsilon \leftarrow 1/100$
                                        ▷ Check right tail approximation applicability
 2: $u_r \leftarrow \operatorname{erf}\left[\frac{\operatorname{erfc}^{-1}(2\epsilon)}{|\alpha|}\right]$
 3: if $\alpha > 0 \wedge u > u_r$ then
 4:     return $\sqrt{2}\operatorname{erf}^{-1}(u)$
 5: end if
 6: if $\alpha < 0 \wedge (1-u) > u_r$ then
 7:     return $-\sqrt{2}\operatorname{erfc}^{-1}(u)$
 8: end if
 9: $v \leftarrow \Phi^{-1}(u)$                 ▷ Evaluate the Gaussian change of variable
10: if $\alpha < 0$ then
11:     $v = -v$
12: end if
                                        ▷ Form series of $Q(v)$ about $v_0$
13: $v_0 \leftarrow \Phi^{-1}(u_0)$              ▷ See (6.9) for the definition of $u_0$
14: $c_0 \leftarrow 0$
15: $c_1 \leftarrow e^{-\frac{v_0^2}{2}}$
16: $c_2 \leftarrow -\frac{1}{2}e^{-v_0^2}\left(\sqrt{\frac{2}{\pi}}|\alpha| + e^{\frac{v_0^2}{2}}v_0\right)$
17: $\ldots$
18: $c_n \leftarrow \ldots$
19: $h \leftarrow \frac{3}{4}\left|\frac{\epsilon}{c_n}\right|^{1/n}$          ▷ Calculate a safe step-size
                                        ▷ Check left tail approximation applicability
20: $v_l \leftarrow v_0 - h$
21: if $v < v_l$ then
22:     if $\alpha > 0$ then
23:         return $-\sqrt{\frac{2W\left(\frac{1}{2\pi u|\alpha|}\right)}{1+\alpha^2}}$
24:     else
25:         return $\sqrt{\frac{2W\left(\frac{1}{2\pi(1-u)|\alpha|}\right)}{1+\alpha^2}}$
26:     end if
27: end if
                                        ▷ The normal to skew-normal recycling function is computed.
28: $Q \leftarrow \sum_{k=0}^{n} c_k(v-v_0)^k$    ▷ In practice, this would be evaluated using Horner's scheme.
29: if $a < 0$ then
30:     return $-Q$
31: else
32:     return $Q$
33: end if
```

Table 6.1: Timings in ms to compute the skew-normal quantile function q_α for 10^7 pseudo-random uniform variates using implementations of SNCDFINV, CF5 and CF4 averaged over 100 runs on the three test Nvidia GPUs. All standard deviations were negligible.

	SNCDFINV			CF5			CF4		
α	Titan	K80	C2050	Titan	K80	C2050	Titan	K80	C2050
2^{-5}	42.95	9.57	31.57	39.54	10.88	30.96	24.14	6.55	16.37
2^{-4}	42.99	8.17	31.57	39.53	9.46	30.96	23.81	5.31	16.37
2^{-3}	43.69	8.21	31.58	40.04	9.45	30.96	23.85	5.41	16.37
2^{-2}	42.44	8.22	31.57	39.95	9.44	30.95	24.53	5.37	16.37
2^{-1}	43.80	8.27	31.55	40.61	9.46	30.95	24.91	5.43	16.37
2^{1}	49.15	9.01	36.35	42.61	9.44	30.96	25.89	5.38	16.37
2^{2}	54.61	9.80	44.11	41.17	9.47	30.93	25.15	5.42	16.33
2^{3}	53.78	9.59	43.31	41.39	9.51	30.95	25.36	5.42	16.35
2^{4}	50.67	9.08	40.31	41.06	9.46	30.95	25.21	5.43	16.35
2^{5}	46.73	8.24	35.78	41.05	9.46	30.95	25.35	5.35	16.35
2^{6}	39.66	7.04	30.38	41.01	9.46	30.95	24.87	5.36	16.35
2^{7}	34.09	5.91	26.32	41.30	9.56	30.95	25.16	5.48	16.35

responding uniform input were copied from the GPU to the CPU. 80-bit extended precision references were computed using Boost 1.57.0 and compared to the copied values. Table 6.2 gives average and peak *forward* relative error (2.13) statistics for the implementations with the same shape parameters from the speed test. Table 6.3 shows the same for the *backward* relative error (2.13).

The error statistics are very interesting, especially in the light of the the speed results. An immediate observation is that SNCDFINV's average and maximum errors are consistently lower than those of the Cornish–Fisher approximants. Moreover, the peak relative errors of SNCDFINV are always at a sensible level. As one would expect, all approximations are very good for $\alpha \approx 0$. Also, as α increases away from zero, the error statistics of both CF5 and CF4 deteriorate. It should be noted that their errors for even small to moderate α are quite poor.

6.5 Commentary

We have described the development of an algorithm, called SNCDFINV, for computing the skew-normal quantile function. The algorithm compares favourably with commonly used Cornish–Fisher expansions. We believe our algorithm would, therefore, be a strong candidate for initial guesses in high-precision skew-normal

Table 6.2: Average and peak *forward* relative error (2.13) statistics for implementations of SNCDFINV, CF5 and CF4, over 10^8 pseudo-random uniform variates for each α. Standard deviations are in brackets. Peak statistics are in **bold**.

α	SNCDFINV	CF5	CF4
	1.02×10^{-14}	3.10×10^{-12}	5.03×10^{-11}
2^{-5}	(3.24×10^{-13})	(9.17×10^{-10})	(3.10×10^{-08})
	$\mathbf{1.71 \times 10^{-09}}$	$\mathbf{4.91 \times 10^{-06}}$	$\mathbf{1.66 \times 10^{-04}}$
	1.24×10^{-12}	3.07×10^{-10}	1.81×10^{-09}
2^{-4}	(5.92×10^{-12})	(1.93×10^{-07})	(1.58×10^{-06})
	$\mathbf{5.56 \times 10^{-09}}$	$\mathbf{1.10 \times 10^{-03}}$	$\mathbf{9.02 \times 10^{-03}}$
	1.59×10^{-10}	2.65×10^{-08}	4.32×10^{-08}
2^{-3}	(7.46×10^{-10})	(1.25×10^{-05})	(2.18×10^{-05})
	$\mathbf{2.43 \times 10^{-07}}$	$\mathbf{5.94 \times 10^{-02}}$	$\mathbf{1.04 \times 10^{-01}}$
	8.09×10^{-09}	3.79×10^{-06}	8.93×10^{-07}
2^{-2}	(6.01×10^{-08})	(1.20×10^{-02})	(1.37×10^{-03})
	$\mathbf{1.54 \times 10^{-05}}$	$\mathbf{1.19 \times 10^{+02}}$	$\mathbf{1.37 \times 10^{+01}}$
	2.50×10^{-06}	3.19×10^{-04}	1.47×10^{-04}
2^{-1}	(2.32×10^{-05})	$(1.03 \times 10^{+00})$	(4.23×10^{-01})
	$\mathbf{1.32 \times 10^{-02}}$	$\mathbf{1.01 \times 10^{+04}}$	$\mathbf{4.17 \times 10^{+03}}$
	6.28×10^{-04}	6.54×10^{-02}	1.21×10^{-01}
2^{1}	(1.25×10^{-03})	$(5.34 \times 10^{+01})$	$(1.00 \times 10^{+02})$
	$\mathbf{3.50 \times 10^{-02}}$	$\mathbf{4.25 \times 10^{+05}}$	$\mathbf{7.97 \times 10^{+05}}$
	7.95×10^{-04}	2.26×10^{-01}	4.15×10^{-01}
2^{2}	(5.75×10^{-03})	$(8.16 \times 10^{+01})$	$(1.52 \times 10^{+02})$
	$\mathbf{1.29 \times 10^{-01}}$	$\mathbf{2.78 \times 10^{+05}}$	$\mathbf{5.18 \times 10^{+05}}$
	4.54×10^{-04}	4.99×10^{-01}	9.25×10^{-01}
2^{3}	(6.55×10^{-03})	$(1.04 \times 10^{+03})$	$(1.96 \times 10^{+03})$
	$\mathbf{1.72 \times 10^{-01}}$	$\mathbf{1.00 \times 10^{+07}}$	$\mathbf{1.90 \times 10^{+07}}$
	2.88×10^{-04}	4.15×10^{-01}	9.57×10^{-01}
2^{4}	(4.72×10^{-03})	$(2.95 \times 10^{+02})$	$(7.32 \times 10^{+02})$
	$\mathbf{1.73 \times 10^{-01}}$	$\mathbf{1.43 \times 10^{+06}}$	$\mathbf{3.54 \times 10^{+06}}$
	1.77×10^{-04}	2.25×10^{-01}	$1.09 \times 10^{+00}$
2^{5}	(2.89×10^{-03})	$(2.55 \times 10^{+02})$	$(2.09 \times 10^{+03})$
	$\mathbf{1.56 \times 10^{-01}}$	$\mathbf{2.42 \times 10^{+06}}$	$\mathbf{1.98 \times 10^{+07}}$
	1.00×10^{-04}	4.56×10^{-01}	6.73×10^{-01}
2^{6}	(1.67×10^{-03})	$(3.03 \times 10^{+02})$	$(4.37 \times 10^{+02})$
	$\mathbf{1.33 \times 10^{-01}}$	$\mathbf{2.32 \times 10^{+06}}$	$\mathbf{3.34 \times 10^{+06}}$
	5.57×10^{-05}	$1.04 \times 10^{+00}$	2.94×10^{-01}
2^{7}	(1.04×10^{-03})	$(8.55 \times 10^{+02})$	$(4.43 \times 10^{+01})$
	$\mathbf{1.09 \times 10^{-01}}$	$\mathbf{5.97 \times 10^{+06}}$	$\mathbf{3.10 \times 10^{+05}}$

Table 6.3: Average and peak *backward* relative error (2.14) statistics for implementations of SNCDFINV, CF5 and CF4, over 10^8 pseudo-random uniform variates for each α. Standard deviations are in brackets. Peak statistics are in **bold**.

α	SNCDFINV	CF5	CF4
	3.12×10^{-14}	1.17×10^{-12}	3.07×10^{-12}
2^{-5}	(4.00×10^{-13})	(5.23×10^{-12})	(8.38×10^{-12})
	$\mathbf{2.96 \times 10^{-10}}$	$\mathbf{4.77 \times 10^{-09}}$	$\mathbf{9.98 \times 10^{-10}}$
	3.74×10^{-12}	7.46×10^{-11}	1.11×10^{-10}
2^{-4}	(4.87×10^{-11})	(3.28×10^{-10})	(2.26×10^{-10})
	$\mathbf{3.65 \times 10^{-08}}$	$\mathbf{2.94 \times 10^{-07}}$	$\mathbf{9.46 \times 10^{-08}}$
	4.16×10^{-10}	4.70×10^{-09}	5.26×10^{-09}
2^{-3}	(5.56×10^{-09})	(2.02×10^{-08})	(1.05×10^{-08})
	$\mathbf{4.23 \times 10^{-06}}$	$\mathbf{1.74 \times 10^{-05}}$	$\mathbf{9.14 \times 10^{-06}}$
	3.88×10^{-08}	2.87×10^{-07}	3.05×10^{-07}
2^{-2}	(5.40×10^{-07})	(1.19×10^{-06})	(6.22×10^{-07})
	$\mathbf{4.23 \times 10^{-04}}$	$\mathbf{9.49 \times 10^{-04}}$	$\mathbf{4.69 \times 10^{-04}}$
	3.31×10^{-06}	1.57×10^{-05}	1.79×10^{-05}
2^{-1}	(5.02×10^{-04})	(6.27×10^{-05})	(3.98×10^{-05})
	$\mathbf{3.00 \times 10^{-01}}$	$\mathbf{4.61 \times 10^{-02}}$	$\mathbf{1.05 \times 10^{-02}}$
	6.36×10^{-04}	1.05×10^{-02}	1.00×10^{-03}
2^{1}	(6.77×10^{-03})	$(3.73 \times 10^{+01})$	(2.66×10^{-02})
	$\mathbf{3.58 \times 10^{-01}}$	$\mathbf{3.72 \times 10^{+05}}$	$\mathbf{9.99 \times 10^{-01}}$
	2.64×10^{-03}	2.48×10^{-02}	3.79×10^{-02}
2^{2}	(3.53×10^{-02})	(6.38×10^{-02})	(7.27×10^{-02})
	$\mathbf{6.84 \times 10^{-01}}$	$\mathbf{9.99 \times 10^{-01}}$	$\mathbf{1.00 \times 10^{+00}}$
	2.10×10^{-03}	5.09×10^{-02}	8.81×10^{-02}
2^{3}	(3.55×10^{-02})	(1.15×10^{-01})	(1.94×10^{-01})
	$\mathbf{7.86 \times 10^{-01}}$	$\mathbf{1.00 \times 10^{+01}}$	$\mathbf{1.00 \times 10^{+00}}$
	1.13×10^{-03}	8.89×10^{-02}	2.32×10^{-01}
2^{4}	(2.53×10^{-02})	(2.35×10^{-01})	(7.61×10^{-01})
	$\mathbf{7.89 \times 10^{-01}}$	$\mathbf{1.18 \times 10^{+00}}$	$\mathbf{4.75 \times 10^{+00}}$
	5.37×10^{-04}	1.63×10^{-01}	$1.76 \times 10^{+00}$
2^{5}	(1.60×10^{-02})	(5.64×10^{-01})	$(9.57 \times 10^{+00})$
	$\mathbf{7.49 \times 10^{-01}}$	$\mathbf{3.69 \times 10^{+00}}$	$\mathbf{8.54 \times 10^{+01}}$
	2.43×10^{-04}	3.94×10^{-01}	$1.55 \times 10^{+03}$
2^{6}	(9.37×10^{-03})	$(1.90 \times 10^{+00})$	$(1.36 \times 10^{+04})$
	$\mathbf{6.91 \times 10^{-01}}$	$\mathbf{1.61 \times 10^{+01}}$	$\mathbf{1.75 \times 10^{+05}}$
	1.16×10^{-04}	$2.64 \times 10^{+00}$	$1.14 \times 10^{+14}$
2^{7}	(5.49×10^{-03})	$(1.92 \times 10^{+01})$	$(1.46 \times 10^{+15})$
	$\mathbf{6.25 \times 10^{-01}}$	$\mathbf{2.17 \times 10^{+02}}$	$\mathbf{2.92 \times 10^{+16}}$

quantile algorithms and low-precision simulations. In contrast to Cornish–Fisher approximations, our power series solution will give more accuracy by dialling up the number of terms used.

The skew-normal quantile function algorithm we have described is not a truly branch-free solution, due to the separate tail formulae. Further work would be necessary to remove this dependency. We simply take the hit of branch divergence on GPUs in this first-cut solution. Nevertheless, our skew-normal algorithm does not need an initialisation stage, which means that it can be used when the shape parameter α varies between calls. For the fixed parameter case, the algorithm from Section 4.2.1 could be adapted for the skew-normal distribution.

Chapter 7

Conclusions

In this thesis, we have shown that the quantile mechanics approach is a viable and powerful technique for developing quantile function approximants. Along the way, we have applied the technique to the normal, gamma, non-central χ^2 and skew-normal distributions. The main research output for each distribution was a quantile function algorithm that is faster or more accurate than the current state of the art on GPUs and other many-core architectures. Our normal and gamma algorithms fit into the "faster" category. We believe that this thesis contains the first comprehensive analysis on the non-central χ^2 and skew-normal quantile functions, together with new algorithms that cover the parameter and variable space more accurately than existing approximations. In particular, we derived new analytic approximations and power series solutions for the non-central χ^2 and skew-normal quantile functions. These are interesting from a theoretical and practical perspective.

We believe that the practical applicability of the inversion method has been expanded in a meaningful way. This was the goal of this thesis. A main idea was to find efficient methods for constructing the polynomial approximations that John von Neumann talked about in his letter (partially reproduced on page 8) to Stanislaw Ulam [25]. We have concurrently looked at algorithms that are particularly suitable for parallel implementation. The algorithms are, therefore, especially useful for random number generation. This work is, of course, not the last word on the matter.

There are a number of possibilities for future research directions. Concrete ideas include improving the non-central χ^2 and skew-normal quantile function algorithms, especially their respective accuracies. The foothold gained in this work is likely to be valuable. Other distributions are also certainly ripe for research. This is especially true for Pearson distributions, which have pleasant quantile ODEs that are relatively easy to work with. Non-Pearson distributions are slightly more involved, in that their density function typically includes a special function. However, as we have shown with the non-central χ^2 and skew-normal distributions, if the special function itself satisfies an ODE then significant progress can be made. For the development of fast GPU quantile function algorithms, there is the issue of branch divergence. Handling this in an efficient manner for the whole parameter and variable space of a distribution is somewhat more of an art than a science. We saw this with the normal and gamma distributions, where multiple and even compositions of inventive changes of variable were called upon. Nevertheless, worthy candidates are distributions that—intentionally or otherwise—mimic the target distribution. For example, the Kumaraswamy distribution [45] (see also [40]) is remarkably beta-like, yet it has a closed-form CDF[1]. Some other interesting tractable beta-like distributions are reviewed in [44].

An open problem in fast and accurate quantile function computation is the case of when the shape parameter(s) of a distribution is not fixed, but varies. This more or less precludes the use of pre-computing approximants via quantile mechanics. However, this author has experimented with an idea of working with the ODE of the *forward* CDF (as opposed to the inverse function), with a view to accelerating the repeated CDF computations in root-finding algorithms. A sizeable speed-up (without compromising accuracy) was observed for the gamma distribution, but the performance is still not quite appropriate for real-time simulation purposes. Nevertheless, details of the underlying mathematics—coined 'CDF mechanics'—were given in Section 2.2.

[1]The Kumaraswamy CDF on $x \in [0, 1]$ is simply $\left[1 - (1 - x^{\alpha})^{\beta}\right]$, where $\alpha, \beta > 0$ are shape parameters. The corresponding PDF for this distribution is $\alpha\beta x^{\alpha-1}(1 - x^{\alpha})^{\beta-1}$. Compare this with the PDF of the beta distribution: $\frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1 - x)^{\beta-1}$.

Appendix A

Availability of software

Normal distribution

Open-source implementations of the GPU-optimised normal quantile approximations in Chapter 3 are given in Appendix B. Similar implementations are also available in the NAG Numerical Routines for GPUs (see http://www.nag.co.uk/numeric/GPUs/index). Anyone who wishes to obtain this should contact NAG either through the website www.nag.co.uk, or via email at infodesk@nag.co.uk.

Gamma distribution

An open-source implementation of the gamma quantile function algorithm described in Chapter 4 is available from the public GitHub repository https://github.com/thomasluu/quantus. A production grade GPU and multithreaded CPU implementation of the algorithm is available from NAG. Please see http://www.nag.co.uk or contact the author for more details.

Non-central χ^2 and skew-normal distributions

An open-source implementation of the quantile function algorithms described in Chapters 5 and 6 is available from the public GitHub repositories https://github.com/thomasluu/ncx2cdfinv and https://github.com/thomasluu/sncdfinv.

Appendix B

CUDA device functions for Φ^{-1}

B.1 Branchless approximation for single-precision

```
__device__ float branchless(float u)
{
    float ushift = u - 0.5f;
    if (ushift > 0.0f) u = 1.0f - u;
    float v = -logf(u + u);

    float p =   1.68267776058639e-6f;
    p = p * v + 0.0007404314351202936f;
    p = p * v + 0.03602364419560667f;
    p = p * v + 0.4500443083534446f;
    p = p * v + 1.861100468283588f;
    p = p * v + 2.748475794390544f;
    p = p * v + 1.253314132218524f;

    float q =   0.00003709787159774307f;
    q = q * v + 0.004513659269519104f;
    q = q * v + 0.1101701640048184f;
    q = q * v + 0.8410203004476538f;
    q = q * v + 2.402969434512837f;
    q = q * v + 2.692965915568952f;
    q = q * v + 1.0f;
    return __fdividef(p, q) * copysignf(v, ushift);
}
```

B.2 Branched approximation for single-precision

```
__device__ float branched(float u)
{
    float ushift = u - 0.5f;
    if (ushift > 0.0f) u = 1.0f - u;
    float v = -logf(u + u);

    float p, q;
    if (v < 22.0f) {
        p =         1.68267776058639e-6f;
        p = p * v + 0.0007404314351202936f;
        p = p * v + 0.03602364419560667f;
        p = p * v + 0.4500443083534446f;
        p = p * v + 1.861100468283588f;
        p = p * v + 2.748475794390544f;
        p = p * v + 1.253314132218524f;

        q =         0.00003709787159774307f;
        q = q * v + 0.004513659269519104f;
        q = q * v + 0.1101701640048184f;
        q = q * v + 0.8410203004476538f;
        q = q * v + 2.402969434512837f;
        q = q * v + 2.692965915568952f;
    } else {
        p =         0.00001016962895771568f;
        p = p * v + 0.003330096951634844f;
        p = p * v + 0.1540146885433827f;
        p = p * v + 1.045480394868638f;
        q =            1.303450553973082e-7f;
        q = q * v + 0.0001728926914526662f;
        q = q * v + 0.02031866871146244f;
        q = q * v + 0.3977137974626933f;
    }
    p *= copysignf(v, ushift);
    q = q * v + 1.0f;

    return __fdividef(p, q);
}
```

B.3 Branched approximation for double-precision

```
__device__ double branched(double u)
{
    double ushift = u - 0.5;
    if (ushift > 0.0) u = 1.0 - u;
    double v = -log(u + u);

    double p, q;
    if (v < 8.0) {
        p =            1.349518868381678058753249e-8;
        p = p * v + 8.52746651977185243009898e-6;
        p = p * v + 0.0007406377273502195713300782;
        p = p * v + 0.02110410727013085360335842;
        p = p * v + 0.2623738494513625927357995;
        p = p * v + 1.636409525694839308796912;
        p = p * v + 5.518808617987600457113932;
        p = p * v + 10.40118297266547564032922;
        p = p * v + 10.8611239302525037936894;
        p = p * v + 5.835965523943366494409442;
        p = p * v + 1.253314137315500218846638;

        q =            3.475390584395848523528879e-7;
        q = q * v + 0.00006713843966407750619673244;
        q = q * v + 0.003234732137701730881039638;
        q = q * v + 0.06142247251750825316341193;
        q = q * v + 0.5598983561302253505510779;
        q = q * v + 2.723933211326168795847825;
        q = q * v + 7.495380651029058089810514;
        q = q * v + 11.93179043337747424406973;
        q = q * v + 10.81567043691618587425845;
        q = q * v + 5.156426788932205027416249;
    } else if (v < 22.0) {
        p =         5.905547081121762506516589e-8;
        p = p * v + 0.00002242239919190552389396371;
        p = p * v + 0.001823372963609771796770297;
        p = p * v + 0.04741412290058875930061007;
        p = p * v + 0.4414912980032717990549048;
        p = p * v + 1.424983759201438344148404;
        p = p * v + 1.250429746707532155567877;

        q =         7.820715032526777904083752e-10;
        q = q * v + 1.038245803426121099241835e-6;
        q = q * v + 0.0001691152956136853577153963;
        q = q * v + 0.007734585142330988979414947;
        q = q * v + 0.1248230877587629006979956;
        q = q * v + 0.7546908613706284650243254;
        q = q * v + 1.628953198232099316377859;
    } else if (v < 44.0) {
        p =         5.610390426863731506852026e-11;
        p = p * v + 1.421171074845048244184302e-7;
        p = p * v + 0.00004247705115079861566951468;
        p = p * v + 0.003356723402904164805382096;
        p = p * v + 0.0846819487619891724948853;
        p = p * v + 0.6645701735755075991918887;
        p = p * v + 1.218054142898073209279639;

        q =         2.938536605592884511709757e-9;
        q = q * v + 2.108867766800134266521303e-6;
        q = q * v + 0.0003132939517941178397687649;
        q = q * v + 0.01414790841185618108831795;
        q = q * v + 0.2116313408806531549023063;
        q = q * v + 0.9697478641141374383359639;
    } else {
        return -CUDART_SQRT_TWO * erfcinv(u + u);
    }
    p *= copysign(v, ushift);
    q = q * v + 1.0;
    return p / q;
}
```

B.4 Hybrid approximation for single-precision

```
__device__ float hybrid(float u)
{
    float v, p, q, ushift;

    ushift = u - 0.5f;

    v = copysignf(ushift, 0.0f);

    if (v < 0.499433f) {
        asm("rsqrt.approx.ftz.f32 %0,%1;" : "=f"(v) : "f"(u - u * u));
        v *= 0.5f;

        p =         0.001732781974270904f;
        p = p * v + 0.1788417306083325f;
        p = p * v + 2.804338363421083f;
        p = p * v + 9.35716893191325f;
        p = p * v + 5.283080058166861f;
        p = p * v + 0.07885390444279965f;
        p *= ushift;
        q =         0.0001796248328874524f;
        q = q * v + 0.02398533988976253f;
        q = q * v + 0.4893072798067982f;
        q = q * v + 2.406460595830034f;
        q = q * v + 3.142947488363618f;
    } else {
        if (ushift > 0.0f) u = 1.0f - u;
        asm("lg2.approx.ftz.f32 %0,%1;" : "=f"(v) : "f"(u + u));
        v *= -0.6931471805599453f;

        if (v < 22.0f) {
            p =         0.000382438382914666f;
            p = p * v + 0.03679041341785685f;
            p = p * v + 0.5242351532484291f;
            p = p * v + 1.21642047402659f;
            q =         9.14019972725528e-6f;
            q = q * v + 0.003523083799369908f;
            q = q * v + 0.126802543865968f;
            q = q * v + 0.8502031783957995f;
        } else {
            p =         0.00001016962895771568f;
            p = p * v + 0.003330096951634844f;
            p = p * v + 0.1540146885433827f;
            p = p * v + 1.045480394868638f;
            q =         1.303450553973082e-7f;
            q = q * v + 0.0001728926914526662f;
            q = q * v + 0.02031866871146244f;
            q = q * v + 0.3977137974626933f;
        }
        p *= copysignf(v, ushift);
    }
    q = q * v + 1.0f;
    asm("rcp.approx.ftz.f32 %0,%1;" : "=f"(v) : "f"(q));
    return p * v;
}
```

B.5 Hybrid approximation for double-precision

```
__device__ double hybrid(double u)
{
    double v, p, q, ushift;

    ushift = u - 0.5;

    v = copysign(ushift, 0.0);

    if (all(v < 0.499483203996)) {
        v = rsqrt(u - u * u);
        v *= 0.5;

        p =         7.2744228279773304710401873082e-8;
        p = p * v + 0.0000421743871813882581153268755947;
        p = p * v + 0.00520379665440516934032831487884;
        p = p * v + 0.225264560844470327491137486146;
        p = p * v + 4.14932318033989880708071080023;
        p = p * v + 35.4767630504161379997123610715;
        p = p * v + 145.073376038130631263518147487;
        p = p * v + 279.375312117440143683363022333;
        p = p * v + 236.365498586700680221243225944;
        p = p * v + 75.0305448544398694183806332634;
        p = p * v + 6.39207201315300014457052458023;
        p = p * v + 0.0167925887515079331460230479124;
        p *= ushift;
        q =         6.75247954960365765875437842188e-9;
        q = q * v + 4.65945444291242666572785117414e-6;
        q = q * v + 0.000655200752555879764745895049551;
        q = q * v + 0.0321185176417740220294374574723;
        q = q * v + 0.675906789081877876894258344513;
        q = q * v + 6.75231637016836002442853621662;
        q = q * v + 33.6182510700950789325981739877;
        q = q * v + 84.7300851352254981643307375148;
        q = q * v + 106.642594343396090577609079223;
        q = q * v + 63.3625936203786444600988073568;
        q = q * v + 15.2022982212427166686341278383;
    } else {
        if (ushift > 0.0) u = 1.0 - u;
        v = -log(u + u);

        if (all(v < 44.0)) {
            if (v < 8.0) {
                p =         1.349518868381678058753249e-8;
                p = p * v + 8.52746651977185243009898e-6;
                p = p * v + 0.0007406377273502195713300782;
                p = p * v + 0.02110410727013085360335842;
                p = p * v + 0.2623738494513625927357995;
                p = p * v + 1.636409525694839308796912;
                p = p * v + 5.518808617987600457113932;
                p = p * v + 10.40118297266547564032922;
                p = p * v + 10.8611239302525037936894;
                p = p * v + 5.835965523943366494409442;
                p = p * v + 1.253314137315500218846638;
                q =         3.475390584395848523528879e-7;
                q = q * v + 0.00006713843966407750619673244;
                q = q * v + 0.003234732137701730881039638;
                q = q * v + 0.06142247251750825316341193;
                q = q * v + 0.5598983561302253505510779;
                q = q * v + 2.723933211326168795847825;
                q = q * v + 7.495380651029058089810514;
                q = q * v + 11.93179043337747424406973;
                q = q * v + 10.81567043691618587425845;
                q = q * v + 5.156426788932205027416249;
            } else {
                p =         3.520313516116902104718364e-14;
                p = p * v + 1.290150467609741469439886e-10;
                p = p * v + 6.205583506207842717279345e-8;
                p = p * v + 9.158932787882144364471905e-6;
                p = p * v + 0.0005374401369638208453883002;
                p = p * v + 0.01394912091056901348391011;
                p = p * v + 0.16549242452273177206985 01;
                p = p * v + 0.87667479370384158573122 23;
                p = p * v + 1.88468136100870139998674 8;
                p = p * v + 1.25236124398308722407455 5;
                q =         2.19276488942086051209795 2e-12;
                q = q * v + 2.38894462713794020396471 4e-9;
                q = q * v + 6.09142515484072031801978 2e-7;
                q = q * v + 0.00005611957534780981729681127;
                q = q * v + 0.00221028996511963578495 3798;
                q = q * v + 0.03993946264733258917755 805;
                q = q * v + 0.33620298562326883578626 47;
                q = q * v + 1.27819116349780849070965 2;
                q = q * v + 2.00077171403615841498062 8;
            }
            p *= copysign(v, ushift);
        } else {
            return -CUDART_SQRT_TWO * erfcinv(u + u);
        }
    }
    q = q * v + 1.0;
    return p / q;
}
```

Appendix C

Non-central χ^2 quantile function power series derivation

This appendix details the derivation of the power series solution in Section 5.3. The quantile function q of the non-central χ^2 distribution satisfies

$$\frac{\mathrm{d}^2 q}{\mathrm{d}u^2} = \left[\frac{2+q-\nu}{2q} - \frac{\lambda}{4} \frac{{}_0\tilde{F}_1\left(;\frac{\nu}{2}+1;\frac{\lambda}{4}q\right)}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda}{4}q\right)} \right] \left(\frac{\mathrm{d}q}{\mathrm{d}u} \right)^2, \tag{C.1}$$

conditional on $q(0) = 0$ and

$$q(u) \sim \left[2^{\nu/2-1} e^{\lambda/2} u \nu \Gamma(\nu/2) \right]^{2/\nu} \quad \text{as } u \to 0. \tag{C.2}$$

Let

$$v = \left[2^{\nu/2-1} e^{\lambda/2} u \nu \Gamma(\nu/2) \right]^{2/\nu}. \tag{C.3}$$

This maps the ODE to

$$\frac{\mathrm{d}^2 q}{\mathrm{d}v^2} + \frac{1-\nu/2}{v} \frac{\mathrm{d}q}{\mathrm{d}v} = \left[\frac{2+q-\nu}{2q} - \frac{\lambda}{4} \frac{{}_0\tilde{F}_1\left(;\frac{\nu}{2}+1;\frac{\lambda}{4}q\right)}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda}{4}q\right)} \right] \left(\frac{\mathrm{d}q}{\mathrm{d}v} \right)^2. \tag{C.4}$$

Let us rearrange this into

$$2\, vq \frac{\mathrm{d}^2 q}{\mathrm{d}v^2} + (2-\nu)\, q \frac{\mathrm{d}q}{\mathrm{d}v} = v\,(2-\nu+q) \left(\frac{\mathrm{d}q}{\mathrm{d}v} \right)^2 - \frac{\lambda}{2} vq \left(\frac{\mathrm{d}q}{\mathrm{d}v} \right)^2 \frac{{}_0\tilde{F}_1\left(;\frac{\nu}{2}+1;\frac{\lambda}{4}q\right)}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda}{4}q\right)}. \tag{C.5}$$

Try

$$q(v) = \sum_{n=0}^{\infty} g_n v^n \tag{C.6}$$

with $g_0 = 0$ and $g_1 = 1$. So

$$\begin{aligned}
2\,v &\left(\sum_{n=0}^{\infty} g_n v^n\right)\left(\sum_{n=0}^{\infty}(n+2)(n+1)g_{n+2}v^n\right) \\
&= (2-\nu)\,v\left(\sum_{n=0}^{\infty}(n+1)g_{n+1}v^n\right)^2 \\
&- (2-\nu)\left(\sum_{n=0}^{\infty} g_n v^n\right)\left(\sum_{n=0}^{\infty}(n+1)g_{n+1}v^n\right) \\
&+ v\left(\sum_{n=0}^{\infty} g_n v^n\right)\left(\sum_{n=0}^{\infty}(n+1)g_{n+1}v^n\right)^2 \\
&- \frac{\lambda}{2}v\left(\sum_{n=0}^{\infty} g_n v^n\right)\left(\sum_{n=0}^{\infty}(n+1)g_{n+1}v^n\right)^2\left(\sum_{n=0}^{\infty} f_n v^n\right)
\end{aligned}, \tag{C.7}$$

where

$$\sum_{n=0}^{\infty} f_n v^n = \frac{{}_0\tilde{F}_1\left(;\frac{\nu}{2}+1;\frac{\lambda}{4}q(v)\right)}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda}{4}q(v)\right)}, \tag{C.8}$$

which we will defer the handling of. We have

$$\begin{aligned}
2\,v &\sum_{n=0}^{\infty} v^n \sum_{i=0}^{n} g_i g_{n-i+2}\left[(n-i+2)\,(n-i+1)\right] \\
&= (2-\nu)\,v\sum_{n=0}^{\infty} v^n \sum_{i=0}^{n} g_{i+1}g_{n-i+1}\left[(i+1)\,(n-i+1)\right] \\
&- (2-\nu)\sum_{n=0}^{\infty} v^n \sum_{i=0}^{n} g_i g_{n-i+1}\,(n-i+1) \\
&+ v\sum_{n=0}^{\infty} v^n \sum_{j=0}^{n} g_j\left(\sum_{i=0}^{n-j} g_{i+1}g_{n-j-i+1}\left[(i+1)\,(n-j-i+1)\right]\right) \\
&- \frac{\lambda}{2}v\sum_{n=0}^{\infty} v^n \sum_{k=0}^{n}\sum_{j=0}^{k} g_j\left(\sum_{i=0}^{k-j} g_{i+1}g_{k-j-i+1}\left[(i+1)\,(k-j-i+1)\right]\right) f_{n-k}
\end{aligned} \tag{C.9}$$

$$\begin{aligned}\Longrightarrow \sum_{n=0}^{\infty} v^{n+1} 2 \sum_{i=0}^{n} g_i g_{n-i+2} \left[(n-i+2)(n-i+1)\right] \\ = \sum_{n=0}^{\infty} v^{n+1} (2-\nu) \sum_{i=0}^{n} g_{i+1} g_{n-i+1} \left[(i+1)(n-i+1)\right] \\ - \sum_{n=0}^{\infty} v^{n+1} (2-\nu) \sum_{i=0}^{n+1} g_i g_{n-i+2} (n-i+2) \\ + \sum_{n=0}^{\infty} v^{n+1} \sum_{j=0}^{n} g_j \left(\sum_{i=0}^{n-j} g_{i+1} g_{n-j-i+1} \left[(i+1)(n-j-i+1)\right] \right) \\ - \sum_{n=0}^{\infty} v^{n+1} \frac{\lambda}{2} \sum_{k=0}^{n} \sum_{j=0}^{k} g_j \left(\sum_{i=0}^{k-j} g_{i+1} g_{k-j-i+1} \left[(i+1)(k-j-i+1)\right] \right) f_{n-k}\end{aligned} \tag{C.10}$$

$$\begin{aligned}\Longrightarrow 2 \sum_{i=1}^{n} g_i g_{n-i+2} \left[(n-i+2)(n-i+1)\right] \\ = (2-\nu) \sum_{i=0}^{n} g_{i+1} g_{n-i+1} \left[(i+1)(n-i+1)\right] \\ - (2-\nu) \sum_{i=1}^{n+1} g_i g_{n-i+2} (n-i+2) \\ + \sum_{j=1}^{n} g_j \left(\sum_{i=0}^{n-j} g_{i+1} g_{n-j-i+1} \left[(i+1)(n-j-i+1)\right] \right) \\ - \frac{\lambda}{2} \sum_{k=1}^{n} \sum_{j=1}^{k} g_j \left(\sum_{i=0}^{k-j} g_{i+1} g_{k-j-i+1} \left[(i+1)(k-j-i+1)\right] \right) f_{n-k}\end{aligned} \tag{C.11}$$

$$\Longrightarrow n\left(2n+\nu\right)g_{n+1}$$
$$= \left(2-\nu\right)\sum_{i=1}^{n-1} g_{i+1}g_{n-i+1}\left[\left(i+1\right)\left(n-i+1\right)\right]$$
$$- 2\sum_{i=2}^{n} g_i g_{n-i+2}\left[\left(n-i+2\right)\left(n-i+1\right)\right]$$
$$- \left(2-\nu\right)\sum_{i=2}^{n} g_i g_{n-i+2}\left(n-i+2\right)$$
$$+ \sum_{j=1}^{n} g_j\left(\sum_{i=0}^{n-j} g_{i+1}g_{n-j-i+1}\left[\left(i+1\right)\left(n-j-i+1\right)\right]\right)$$
$$- \frac{\lambda}{2}\sum_{k=1}^{n}\sum_{j=1}^{k} g_j\left(\sum_{i=0}^{k-j} g_{i+1}g_{k-j-i+1}\left[\left(i+1\right)\left(k-j-i+1\right)\right]\right)f_{n-k} \tag{C.12}$$

$$\Longrightarrow n\left(2n+\nu\right)g_{n+1}$$
$$= \sum_{i=1}^{n-1} g_{i+1}g_{n-i+1}\left(n-i+1\right)\left[\left(2-\nu\right)\left(i+1\right)-2\ \left(n-i\right)-\left(2-\nu\right)\right]$$
$$+ a(n) - \frac{\lambda}{2}\sum_{k=1}^{n} a(k)f_{n-k} \tag{C.13}$$

where

$$a(n) = \sum_{j=1}^{n} g_j\left(\sum_{i=0}^{n-j} g_{i+1}g_{n-j-i+1}\left[\left(i+1\right)\left(n-j-i+1\right)\right]\right). \tag{C.14}$$

We will now give a recursive formula for the coefficients in

$$\sum_{n=0}^{\infty} f_n v^n = \frac{{}_0\tilde{F}_1\left(;\frac{\nu}{2}+1;\frac{\lambda}{4}q(v)\right)}{{}_0\tilde{F}_1\left(;\frac{\nu}{2};\frac{\lambda}{4}q(v)\right)}. \tag{C.15}$$

Let us focus on ${}_0\tilde{F}_1\left(;b;\frac{\lambda}{4}q(v)\right)$ for some b. The power series expansion of ${}_0\tilde{F}_1\left(;b;z\right)$ at $z=0$ is

$${}_0\tilde{F}_1\left(;b;z\right) = \sum_{k=0}^{\infty}\frac{1}{\Gamma(b+k)k!}z^k \tag{C.16}$$

(see, e.g., [80]). Let us define the coefficients of this series recursively.

$$d_k(b) = \begin{cases} \frac{1}{\Gamma(b)} & \text{if } k = 0 \\ \frac{d_{k-1}(b)}{(b+k-1)k} & \text{if } k > 0 \end{cases}. \tag{C.17}$$

Now let

$$\sum_{n=0}^{\infty} c_n(b) v^n = {}_0\tilde{F}_1\left(;b;\frac{\lambda}{4}q(v)\right). \tag{C.18}$$

We have (see, e.g., [32, 56])

$$c_n(b) = \begin{cases} d_0(b) & \text{if } n = 0 \\ \sum_{k=1}^{n} d_k(b) e_n(k) & \text{if } n \geqslant 1 \end{cases}, \tag{C.19}$$

where

$$e_n(\alpha) = \begin{cases} \left(\frac{\lambda}{4}\right)^{\alpha} & \text{if } n = \alpha \\ \frac{1}{n-\alpha} \sum_{k=1}^{n-\alpha} ((\alpha+1)k - (n-\alpha)) g_{k+1} e_{n-k}(\alpha) & \text{if } n > \alpha \end{cases}. \tag{C.20}$$

Returning to our original problem,

$$\begin{aligned} f_n &= \frac{1}{c_0(\nu/2)} \left(c_n(\nu/2+1) - \sum_{k=0}^{n-1} f_k c_{n-k}(\nu/2) \right) \\ &= \Gamma(\nu/2) \left(c_n(\nu/2+1) - \sum_{k=0}^{n-1} f_k c_{n-k}(\nu/2) \right) \end{aligned}. \tag{C.21}$$

The formulae in Section 5.3 follow immediately.

List of Abbreviations and Symbols

$I_x(a,b)$ normalised incomplete beta function of a, b and x : $I_x(a,b) = \frac{B_x(a,b)}{B(a,b)}$

$I_x^{-1}(a,b)$ inverse of $I_x(a,b)$ with respect to x

CDF cumulative distribution function

CPU central processing unit

$\mathrm{erf}(x)$ error function

$\mathrm{erfc}(x)$ complementary error function

FPGA field-programmable gate array

$P(a,x)$ normalised lower incomplete gamma function of a and x : $P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)}$

$P^{-1}(a,x)$ inverse of $P(a,x)$ with respect to x

GPU graphics processing unit

ODE ordinary differential equation

PDF probability density function

RNG random number generator

SIMD single instruction, multiple data

Bibliography

[1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*, vol. 55 of Applied Mathematics Series, National Bureau of Standards, 1964.

[2] P. J. Acklam, *An algorithm for computing the inverse normal cumulative distribution function.* http://home.online.no/~pjacklam/notes/invnorm/.

[3] A. Azzalini, *Random numbers with skew-normal or skew-t distribution.* http://azzalini.stat.unipd.it/SN/faq-r.html.

[4] ——, *The Skew-Normal Probability Distribution.* http://azzalini.stat.unipd.it/SN/index.html.

[5] A. Azzalini, *A class of distributions which includes the normal ones*, Scandinavian Journal of Statistics, 12 (1985), pp. 171–178.

[6] ——, *The Skew-Normal and Related Families*, vol. 3, Cambridge University Press, 2013.

[7] D. A. Barry, J.-Y. Parlange, L. Li, H. Prommer, C. J. Cunningham, and F. Stagnitti, *Analytical approximations for real values of the Lambert W-function*, Mathematics and Computers in Simulation, 53 (2000), pp. 95–103.

[8] ——, *Erratum to analytical approximations for real values of the Lambert W-function*, Mathematics and Computers in Simulation, 59 (2002), pp. 543–543.

[9] J. D. Beasley and S. G. Springer, *Algorithm AS 111: The percentage points of the normal distribution*, Applied Statistics, 26 (1977), pp. 118–121.

[10] D. Benton and K. Krishnamoorthy, *Computing discrete mixtures of continuous distributions: noncentral chisquare, noncentral t and the distribution of*

the square of the sample multiple correlation coefficient, Computational Statistics & Data Analysis, 43 (2003), pp. 249–267.

[11] D. J. Best and D. E. Roberts, *Algorithm AS 91: The percentage points of the χ^2 distribution*, Applied Statistics, 24 (1975), pp. 385–388.

[12] J. M. Blair, C. A. Edwards, and J. H. Johnson, *Rational Chebyshev approximations for the inverse of the error function*, Mathematics of Computation, 30 (1976), pp. 827–830.

[13] P. J. Boland, *Statistical and Probabilistic Methods in Actuarial Science*, Chapman & Hall/CRC, 2007.

[14] Boost, *Boost C++ Math Toolkit*, 2014.

[15] G. E. P. Box and M. E. Muller, *A note on the generation of random normal deviates*, The Annals of Mathematical Statistics, 29 (1958), pp. 610–611.

[16] T. Bradley, J. du Toit, M. Giles, R. Tong, and P. Woodhams, *Parallelization techniques for random number generators*, in GPU Computing Gems Emerald Edition, W.-m. W. Hwu, ed., Morgan Kaufmann, 2011, pp. 231–246.

[17] A. Capitanio, *On the approximation of the tail probability of the scalar skew-normal distribution*, METRON, 68 (2010), pp. 299–308.

[18] P. Carr, H. Geman, D. B. Madan, and M. Yor, *The fine structure of asset returns: An empirical investigation*, The Journal of Business, 75 (2002), pp. 305–333.

[19] C. W. Clenshaw, *A note on the summation of Chebyshev series*, Mathematics of Computation, 9 (1955), pp. 118–120.

[20] G. Corliss and Y. F. Chang, *Solving ordinary differential equations using Taylor series*, ACM Transactions on Mathematical Software, 8 (1982), pp. 114–144.

[21] L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

[22] A. R. DiDonato and A. H. Morris Jr, *Computation of the incomplete gamma function ratios and their inverse*, ACM Transactions on Mathematical Software, 12 (1986), pp. 377–393.

[23] C. G. Ding, *Algorithm AS 275: Computing the non-central χ^2 distribution function*, Applied Statistics, 41 (1992), pp. 478–482.

[24] ——, *An efficient algorithm for computing quantiles of the noncentral chi-squared distribution*, Computational Statistics & Data Analysis, 29 (1999), pp. 253–259.

[25] R. Eckhardt, *Stan Ulam, John von Neumann, and the Monte Carlo method*, Los Alamos Science, 15 (1987), pp. 131–143.

[26] E. Fehlberg, *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems*, NASA Technical Report 315.

[27] J. E. Gentle, *Random Number Generation and Monte Carlo Methods*, Springer-Verlag, New York, 2003.

[28] A. Gil, J. Segura, and N. M. Temme, *Efficient and accurate algorithms for the computation and inversion of the incomplete gamma function ratios*, SIAM Journal on Scientific Computing, 34 (2012), pp. A2965–A2981.

[29] W. Gilchrist, *Statistical Modelling with Quantile Functions*, CRC Press, 2000.

[30] M. Giles, *Approximating the erfinv function*, in GPU Computing Gems Jade Edition, W.-m. W. Hwu, ed., Morgan Kaufmann, 2011, pp. 109–116.

[31] P. Glasserman, *Monte Carlo methods in financial engineering*, vol. 53, Springer, 2004.

[32] P. Henrici, *Applied and Computational Complex Analysis*, vol. 1, Wiley, 1988.

[33] N. J. Higham, *The accuracy of floating point summation*, SIAM Journal on Scientific Computing, 14 (1993), pp. 783–799.

[34] G. W. Hill and A. W. Davis, *Generalized asymptotic expansions of Cornish-Fisher type*, The Annals of Mathematical Statistics, 39 (1968), pp. 1264–1273.

[35] G. J. Husak, J. Michaelsen, and C. Funk, *Use of the gamma distribution to represent monthly rainfall in Africa for drought monitoring applications*, International Journal of Climatology, 27 (2007), pp. 935–944.

[36] P. Jäckel, *Monte Carlo methods in finance*, vol. 9, Wiley, 2002.

[37] N. L. Johnson, *Systems of frequency curves generated by methods of translation*, Biometrika, 36 (1949), pp. 149–176.

[38] N. L. Johnson, S. Kotz, and N. Balakrishnan, *Continuous Univariate Distributions*, vol. 1 of Probability and Mathematical Statistics, Wiley, 2 ed., 1994.

[39] ——, *Continuous Univariate Distributions*, vol. 2 of Probability and Mathematical Statistics, Wiley, 2 ed., 1995.

[40] M. Jones, *Kumaraswamy's distribution: A beta-type distribution with some tractability advantages*, Statistical Methodology, 6 (2009), pp. 70–81.

[41] W. J. Kennedy Jr and J. E. Gentle, *Statistical computing*, Dekker, 1980.

[42] H. Kettani, *Contributions to the theory of the non-central chi-square distribution*, in Proceedings of the 2006 International Conference on Scientific Computing, H. R. Arabnia, ed., Las Vegas, Nevada, USA, June 2006, CSREA Press, pp. 48–54.

[43] R. Korn, E. Korn, and G. Kroisandt, *Monte Carlo Methods and Models in Finance and Insurance*, CRC Press, 2010.

[44] S. Kotz and J. R. van Dorp, *Beyond Beta: Other Continuous Families of Distributions with Bounded Support and Applications*, World Scientific, 2004.

[45] P. Kumaraswamy, *A generalized probability density function for double-bounded random processes*, Journal of Hydrology, 46 (1980), pp. 79–88.

[46] P. L'Ecuyer, *Good parameters and implementations for combined multiple recursive random number generators*, Operations Research, (1999), pp. 159–164.

[47] C. Lemieux, *Monte Carlo and Quasi-Monte Carlo Sampling*, vol. 20, Springer, 2009.

[48] T. Luu, *Efficient and accurate parallel inversion of the gamma distribution*, SIAM Journal on Scientific Computing, 37 (2015), pp. C122–C141.

[49] G. Marsaglia, *Xorshift RNGs*, Journal of Statistical Software, 8 (2003), pp. 1–6.

[50] G. Marsaglia and T. A. Bray, *A convenient method for generating normal variables*, SIAM Review, 6 (1964), pp. 260–264.

[51] G. Marsaglia and W. W. Tsang, *A simple method for generating gamma variables*, ACM Transactions on Mathematical Software, 26 (2000), pp. 363–372.

[52] M. Matsumoto and T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modeling and Computer Simulation, 8 (1998), pp. 3–30.

[53] N. Metropolis and S. Ulam, *The Monte Carlo method*, Journal of the American Statistical Association, 44 (1949), pp. 335–341.

[54] R. E. Moore, *Methods and Applications of Interval Analysis*, vol. 2, SIAM, 1979.

[55] B. Moro, *The full monte*, RISK, 8 (1995), pp. 57–58.

[56] A. U. K. Munir, *Series Representations and Approximation of some Quantile Functions appearing in Finance*, PhD thesis, University College London, 2012.

[57] NAG, *The NAG Library*, The Numerical Algorithms Group (NAG), Oxford, United Kingdom, 2014.

[58] J. P. Nolan, *Stable Distributions - Models for Heavy Tailed Data*, Birkhauser, Boston, 2015. In progress, Chapter 1 online at academic2.american.edu/∼jpnolan.

[59] NVIDIA Corporation, *CUDA C Programming Guide*. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[60] ——, *CUDA Math API*. http://docs.nvidia.com/cuda/cuda-math-api/.

[61] ——, *cuRAND Library Programming Guide*. http://docs.nvidia.com/cuda/curand/.

[62] M. Patefield and D. Tandy, *Fast and accurate calculation of Owen's T function*, Statistical Software, 5 (2000), pp. 1–25.

[63] E. S. Pearson, *Note on an approximation to the distribution of non-central* χ^2, Biometrika, 46 (1959), p. 364.

[64] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 3 ed., 2007.

[65] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2014.

[66] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, *Parallel random numbers: As easy as 1, 2, 3*, in High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for, IEEE, 2011, pp. 1–12.

[67] M. Sankaran, *On the non-central chi-square distribution*, Biometrika, 46 (1959), pp. 235–237.

[68] ——, *Approximations to the non-central chi-square distribution*, Biometrika, 50 (1963), pp. 199–204.

[69] W. T. Shaw, *Sampling Student's T distribution – use of the inverse cumulative distribution function*, Journal of Computational Finance, 9 (2006), pp. 37–73.

[70] ——, *Refinement of the normal quantile: A benchmark normal quantile based on recursion, and an appraisal of the Beasley-Springer-Moro, Acklam, and Wichura (AS241) methods.* http://www.mth.kcl.ac.uk/~shaww/web_page/papers/NormalQuantile1.pdf, 2007.

[71] W. T. Shaw and N. Brickman, *Differential equations for Monte Carlo recycling and a GPU-optimized normal quantile.* http://arxiv.org/abs/0901.0638v3, 2009.

[72] W. T. Shaw and I. R. C. Buckley, *The alchemy of probability distributions: beyond Gram-Charlier expansions, and a skew-kurtotic-normal distribution from a rank transmutation map.* http://arxiv.org/abs/0901.0434, 2009.

[73] W. T. Shaw, T. Luu, and N. Brickman, *Quantile mechanics II: changes of variables in Monte Carlo methods and GPU-optimized normal quantiles.* http://arxiv.org/abs/0901.0638v5, 2011.

[74] W. T. Shaw, T. Luu, and N. Brickman, *Quantile mechanics II: changes of variables in Monte Carlo methods and GPU-optimised normal quantiles*, European Journal of Applied Mathematics, 25 (2014), pp. 177–212.

[75] W. T. Shaw and J. McCabe, *Monte Carlo sampling given a characteristic function: Quantile mechanics in momentum space.* http://arxiv.org/abs/0903.1592, 2009.

[76] G. Steinbrecher and W. T. Shaw, *Quantile mechanics*, European Journal of Applied Mathematics, 19 (2008), pp. 87–112.

[77] N. M. Temme, *Asymptotic and numerical aspects of the noncentral chi-square distribution*, Computers & Mathematics with Applications, 25 (1993), pp. 55–63.

[78] H. C. Thacher Jr, *Conversion of a power to a series of Chebyshev polynomials*, Communications of the ACM, 7 (1964), pp. 181–182.

[79] The Wolfram Functions Site, *Complementary error function.* http://functions.wolfram.com/GammaBetaErf/Erfc/13/01/01/0001/.

[80] ——, *Regularized confluent hypergeometric function.* http://functions.wolfram.com/HypergeometricFunctions/Hypergeometric0F1Regularized/06/01/02/01/01/0003/.

[81] G. Ulrich and L. T. Watson, *A method for computer generation of variates from arbitrary continuous distributions*, SIAM Journal on Scientific Computing, 8 (1987), pp. 185–197.

[82] J. von Neumann, *Various techniques used in connection with random digits*, National Bureau of Standards Applied Mathematics Series, 12 (1951), pp. 36–38.

[83] M. J. Wichura, *Algorithm AS 241: The percentage points of the normal distribution*, Applied Statistics, 37 (1988), pp. 477–484.

[84] S. Wolfram, *The Mathematica Book*, Wolfram Media/Cambridge University Press, 5 ed., 2003.

[85] Wolfram Research, Inc., *Random number generation.* http://reference.wolfram.com/language/tutorial/RandomNumberGeneration.html.

[86] ——, *Mathematica, Version 8.0.1*, Champaign, Illinois, 2011.

[87] ——, *Mathematica, Version 9.0.1*, Champaign, Illinois, 2013.

[88] ——, *Mathematica, Version 10.0*, Champaign, Illinois, 2014.