Algorithm 955: approximation of the inverse Poisson cumulative distribution function

Michael B. Giles, University of Oxford

New approximations for the inverse of the incomplete gamma function are derived, and these are used to develop efficient evaluations of the inverse Poisson cumulative distribution function. An asymptotic approximation based on the standard Normal approximation is particularly good for CPUs with MIMD cores, while for GPUs and other hardware with vector units a second asymptotic approximation based on Temme's approximation of the incomplete gamma function is more efficient due to conditional branching within each vector. The accuracy and efficiency of the software implementations is assessed on both CPUs and GPUs.

CCS Concepts: •**Mathematics of computing** → **Mathematical software; Numerical analysis;**

Additional Key Words and Phrases: Approximation, Poisson distribution, CUDA, GPU

ACM Reference Format:
Michael B. Giles, 2015. Approximation of the inverse Poisson cumulative distribution function. *ACM Trans. Math. Softw.* V, N, Article XXXX (May 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2699466

1. INTRODUCTION

Poisson random variables are used in a wide variety of applications, as diverse as the simulation of queueing systems [Gross et al. 2008] and stochastic biochemical reaction modelling [Gillespie 2007]. We are motivated by applications which do not have a fixed Poisson rate λ, so we aim to develop a method which is efficient at handling a different λ for each Poisson random variable N.

The cumulative distribution function (CDF) for Poisson rate λ is defined as

$$\overline{C}(n) \;\equiv\; \mathbb{P}(N \leq n) \;=\; e^{-\lambda} \sum_{m=0}^{n} \frac{\lambda^m}{m!}. \tag{1}$$

We are interested in evaluating the inverse CDF defined as

$$\overline{C}^{-1}(u) = n, \tag{2}$$

where n is the smallest integer such that

$$u \;\leq\; e^{-\lambda} \sum_{m=0}^{n} \frac{\lambda^m}{m!}. \tag{3}$$

Author's address: Oxford University Mathematical Institute, Radcliffe Observatory Quarter, Woodstock Road, Oxford OX2 6GG

Alternatively, it is the smallest integer n such that

$$1-u \;\geq\; e^{-\lambda} \sum_{m=n+1}^{\infty} \frac{\lambda^m}{m!}. \tag{4}$$

Given this inverse CDF, we can generate a random variable U which is uniformly distributed on the open unit interval $(0,1)$ and then compute the Poisson variate $N = \overline{C}^{-1}(U)$. There are alternative approaches to generating Poisson random variables based on rejection methods [Ahrens and Dieter 1982] or a recursion [Devroye 1991], but the advantage of this approach is that it can also be used with quasi-random uniforms, and with variance reduction approaches such as stratified and Latin Hypercube sampling [Asmussen and Glynn 2007; Glasserman 2004].

When λ is relatively small, which we take to be $\lambda \leq 4$, the most efficient approach to evaluating $\overline{C}^{-1}(u)$ is to use either (3) or (4). The former approach, usually referred to as "bottom-up" summation, has lower floating point rounding errors when $u < \frac{1}{2}$, while the latter approach ("top-down" summation) is usually more accurate when $u > \frac{1}{2}$. Practical implementation issues will be discussed later.

Most of this paper is devoted to the case of large λ. Suppose that X is a real positive random variable with a CDF corresponding to the (regularised) incomplete Gamma function

$$C(x) \;\equiv\; \mathbb{P}(X < x) \;=\; \frac{1}{\Gamma(x)} \int_{\lambda}^{\infty} e^{-t}\, t^{x-1}\, \mathrm{d}t.$$

If $N = \lfloor X \rfloor$ denotes X rounded down to the nearest integer, then integration by parts reveals that

$$\mathbb{P}(N \leq n) \;=\; \mathbb{P}(X < n{+}1) \;=\; \frac{1}{n!} \int_{\lambda}^{\infty} e^{-t}\, t^{n}\, \mathrm{d}t \;=\; e^{-\lambda} \sum_{m=0}^{n} \frac{\lambda^m}{m!}.$$

Hence, N has a Poisson distribution with rate λ, and $\overline{C}^{-1}(u) = \lfloor C^{-1}(u) \rfloor$, as illustrated in Figure 1.

Note that this inverse, $C^{-1}(u)$, is different to the one considered in the literature [Temme 1992], which considers the inverse with respect to λ rather than x.

The next two sections of the paper derive two approximations $\widetilde{Q}(u)$ to the quantile function $Q(u) \equiv C^{-1}(u)$. Before deriving the approximations and discussing their accuracy, we first address the question of how much accuracy is desired.

Suppose that $\left| \widetilde{Q}(u) - Q(u) \right| < \delta$ for some $\delta < \frac{1}{2}$. We then have

$$Q(u) \in \left(\widetilde{Q}(u){-}\delta, \widetilde{Q}(u){+}\delta \right),$$

and hence

$$n \;=\; \lfloor Q(u) \rfloor \;\in\; \left[\lfloor \widetilde{Q}(u){-}\delta \rfloor, \lfloor \widetilde{Q}(u){+}\delta \rfloor \right].$$

Since $\delta < \frac{1}{2}$, the interval contains at most two integer values. It contains only one integer with a probability which is approximately $1-2\delta$, assuming that $Q(u) - \lfloor Q(u) \rfloor$ is approximately uniformly distributed. In this case, $n = \lfloor \widetilde{Q}(u) \rfloor$. In the other case in which the interval includes two integer values then $m = \lfloor \widetilde{Q}(u){+}\delta \rfloor$ is equal to either

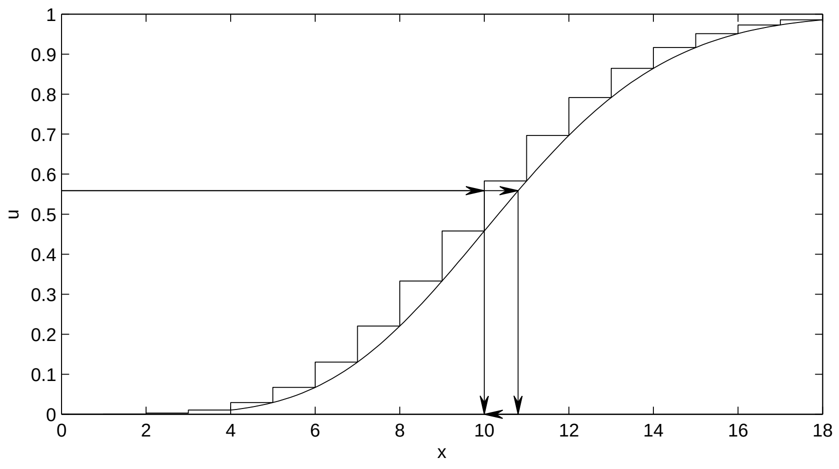

Fig. 1. Plot of $\overline{C}(x)$ and $C(x)$ for $\lambda = 10$, and an illustration of the rounding down of $C^{-1}(u)$ to give $\overline{C}^{-1}(u)$.

Table I. Average CPU and GPU costs, relative to C_C, for two different approximations $\widetilde{Q}(u)$.

	δ=0.01, $C_Q/C_C = 0.1$	δ=0.0001, $C_Q/C_C = 0.2$
CPU cost	0.1 + 0.02 = 0.12	0.2 + 0.0002 = 0.2002
GPU cost	0.1 + 0.48 = 0.58	0.2 + 0.0064 = 0.2064

n or $n+1$. We can distinguish between these two possibilities by evaluating $\overline{C}(m)$, assuming that we have an efficient way of doing this. If $\overline{C}(m) < u$, then $n = m$, otherwise $n = m-1$.

If the costs of evaluating $\widetilde{Q}(u)$ and $\overline{C}(m)$ are C_Q and C_C, respectively, then the average cost is approximately $C_Q + 2\,\delta\, C_C$. This raises an interesting tradeoff; a more accurate approximation $\widetilde{Q}(u)$ will reduce δ, but may increase C_Q, so it is not clear whether it will improve the overall average cost.

This simple analysis has assumed an execution on conventional MIMD CPU cores, with each core operating independently. Execution on functional units within an NVIDIA GPU has a SIMD style in which 32 threads in a *warp* all execute the same instruction at the same time, but using different data [NVIDIA 2014]. When there is conditional branching, effectively all of the threads still perform the same operation, but only some of the threads store the result. This is known as *warp divergence* and results in a significant loss of performance. In the algorithm described above, if any one of the threads needs to evaluate $\overline{C}(m)$ then effectively all of them execute that code. The probability that at least one thread needs to evaluate $\overline{C}(m)$ is approximately $(1 - (1-2\,\delta)^{32} \approx 64\,\delta$, if $\delta \ll 1$, and so the average cost becomes

$$C_Q + \left(1 - (1-2\,\delta)^{32}\right)\, C_C \;\approx\; C_Q + 64\,\delta\, C_C \quad \text{if}\ \delta \ll 1.$$

Table I presents the consequences of this analysis for two different approximations $\widetilde{Q}(u)$. The first gives $\delta = 0.01$ at a relative cost $C_Q/C_C = 0.1$, while the second more accurate approximation gives $\delta = 0.0001$ at double the cost, $C_Q/C_C = 0.2$. For CPU execution the first approximation gives the lowest average cost, while for GPU execution the second is more efficient. This motivates the development of the two different approximations in the next two sections.

In addition, single-precision arithmetic is 3-24 times faster than double-precision on GPUs, and therefore we develop both single-precision and double-precision approximations and implementations for the GPU. Note as well that each core in the latest Intel CPUs also has an AVX vector unit which operate on 256-bit vectors (8 single-precision or 4 double-precision variables) [Intel 2014]. Most application codes do not use these at present, but as they become used more in the future, and as the vector length increases, then the approximations developed in this paper for GPUs may become equally relevant to CPUs.

A final preliminary comment concerns the range of finite precision floating point variables u satisfying the inequailities $0\!<\!u\!<\!1$. In single precision, the smallest value is approximately 10^{-38}, while the largest value is approximately $1-6\times 10^{-8}$. Hence, the single precision approximations to be developed will be tested over the “full single precision range” defined as $[10^{-38}, 1-6\times 10^{-8}]$.

The corresponding double precision range would be $[10^{-308}, 1\!-\!10^{-16}]$. This provides much greater resolution of the tail of the probability distribution near $u=0$ than the other tail near $u=1$. To provide equal resolution of both tails, we choose to define a second function $\overline{C}_c^{-1}(v)$ for the inverse of the complementary Possion CDF, defined by

$$\overline{C}_c(n) = 1 - \overline{C}_c(n) \quad\Longrightarrow\quad \overline{C}_c^{-1}(v) = \overline{C}^{-1}(1\!-\!v).$$

Implementing double precision approximations for both $\overline{C}^{-1}(u)$ and $\overline{C}_c^{-1}(v)$, the input v can take values as small as 10^{-308}, so we are now able to sample both tails equally. Hence, the “full double precision range” which is used for testing the approximations will be $[10^{-308}, 1\!-\!10^{-308}]$.

2. NORMAL ASYMPTOTIC APPROXIMATION

It is well known that as $\lambda\to\infty$ the Poisson CDF approaches that of a Normal distribution with mean λ and variance λ. This motivates the change of variables

$$x = \lambda + \sqrt{\lambda}\, y, \qquad t = \lambda + \sqrt{\lambda}\,(y\!-\!z),$$

which gives

$$C(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{y} I(y,z)\; \mathrm{d}z,$$

where

$$\log I = \tfrac{1}{2}\log(2\pi) - \log\Gamma(x) - t + (x\!-\!1)\log t - \tfrac{1}{2}\log\lambda.$$

Defining $\varepsilon=\lambda^{-1/2}$, an asymptotic expansion in powers of ε, followed by exponentiation and a second expansion in powers of ε, yields

$$I(y,z) = \ \exp(-\tfrac{1}{2}z^2)\ \left(1\ +\ \sum_{n=1}^{\infty} \varepsilon^n p_n(y,z)\right)$$

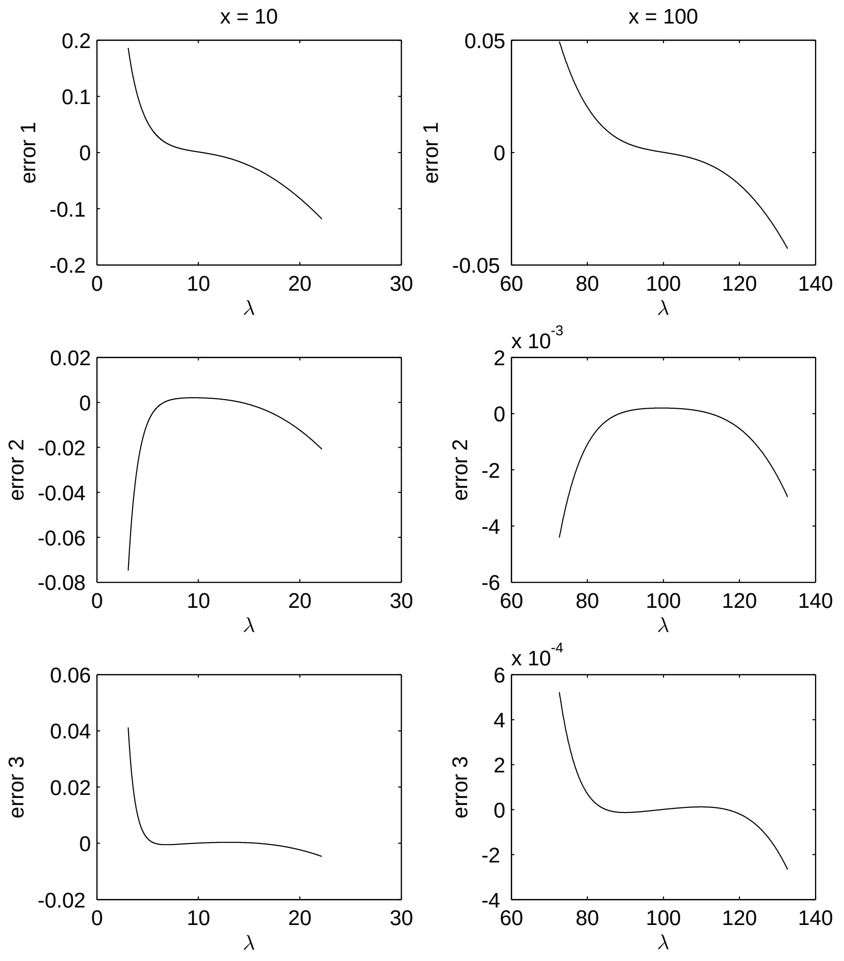

Fig. 2. Errors in Normal approximations $\widetilde{Q}_{N1}, \widetilde{Q}_{N2}, \widetilde{Q}_{N3}$, for $x\!=\!10, 100$ and $|w|\!\le\!3$.

where each of the $p_n(y,z)$ is a polynomial in both y and z. Integrating this by parts gives

$$C(x) = \Phi(y) + \phi(y)\,\Big(\;\varepsilon\,(-\tfrac{1}{3} - \tfrac{1}{6}\,y^2)\;+\;\varepsilon^2(\tfrac{1}{12}y + \tfrac{1}{72}\,y^3 - \tfrac{1}{72}\,y^5)$$
$$+\,\varepsilon^3(-\tfrac{1}{540} - \tfrac{23}{540}\,y^2 + \tfrac{7}{2160}\,y^4 + \tfrac{5}{648}\,y^6 - \tfrac{1}{1296}\,y^8) + O(\varepsilon^4)\Big)$$

where $\Phi(y)$ is the standard Normal CDF function, and $\phi(y) = \Phi'(y) = \exp(-\frac{1}{2}y^2)/\sqrt{2\pi}$ is the standard Normal probability density function. Inverting this expansion, using the methodology presented in Appendix A, gives the asymptotic expansion

$$Q(u) \;\equiv\; C^{-1}(u) = \lambda \;+\; \sqrt{\lambda}\;w \;+\; (\tfrac{1}{3} + \tfrac{1}{6}\,w^2) \;+\; \lambda^{-1/2}\,(-\tfrac{1}{36}\,w - \tfrac{1}{72}\,w^3)$$
$$+\;\lambda^{-1}(-\tfrac{8}{405} + \tfrac{7}{810}\,w^2 + \tfrac{1}{270}w^4) \;+\; O(\lambda^{-3/2}),$$

where $w = \Phi^{-1}(u)$.

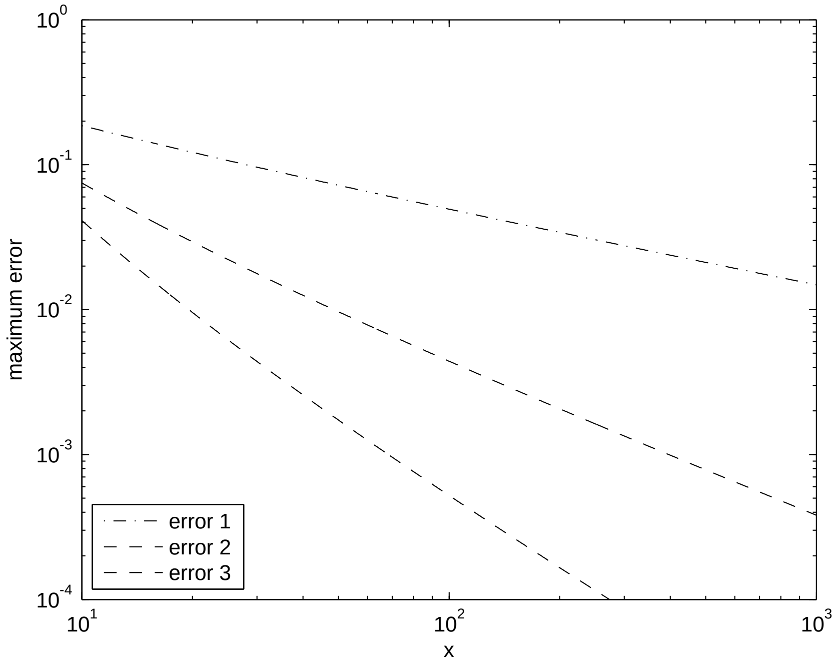

Fig. 3. Maximum errors for Normal approximations $\widetilde{Q}_{N1}$, $\widetilde{Q}_{N2}$, $\widetilde{Q}_{N3}$, for $|w|\leq 3$.

The asymptotic expansions given above were performed using MATLAB's Symbolic Toolbox. Additional validation is provided by Figures 2 and 3 which display the errors corresponding to the three approximations:

$$\begin{aligned}
\widetilde{Q}_{N1}(u) &= \lambda + \sqrt{\lambda}\, w \ + \ (\tfrac{1}{3} + \tfrac{1}{6}\, w^2) \\
\widetilde{Q}_{N2}(u) &= \widetilde{Q}_{N1}(u) \ + \ \lambda^{-1/2}\,(-\tfrac{1}{36}\, w - \tfrac{1}{72}\, w^3) \\
\widetilde{Q}_{N3}(u) &= \widetilde{Q}_{N2}(u) \ + \ \lambda^{-1}(-\tfrac{8}{405} + \tfrac{7}{810}\, w^2 + \tfrac{1}{270}\, w^4)
\end{aligned}$$

The errors are calculated for values of w in the range $[-3,3]$, corresponding to 3 standard deviations of the Normal distribution. The probability of being outside this range is less than 0.3%, assuming u is uniformly distributed over $(0,1)$. In the rare cases in which w is outside this range, the algorithm from the next section can be used to approximate $Q(u)$.

For each pair x, w, we compute $u=\Phi(w)$, and use the MATLAB function `gammaincinv` to determine the value of λ for which $C(x) = u$. The error is then defined as $\widetilde{Q}(u)-Q(u) \equiv \widetilde{Q}(u)-x$. Figure 3 uses a lower limit of $x=10$ because if the computed approximation for x is less than 10, we will instead use bottom-up summation, based on (3), to determine the inverse Poisson CDF value.

Comparing the approximations $\widetilde{Q}_{N2}$ and $\widetilde{Q}_{N3}$, it suggests that a possible bound for the error in $\widetilde{Q}_{N2}$ is

$$\delta = \lambda^{-1}(\tfrac{1}{40} + \tfrac{1}{80}\, w^2 + \tfrac{1}{160}\, w^4).$$

Figure 4 shows the relative error $(\widetilde{Q}_{N2}-Q)/\delta$ for $x=10,100$. The maximum relative error, over the same x range as before, does not exceed 0.85 for all $x \geq 10$. If w is

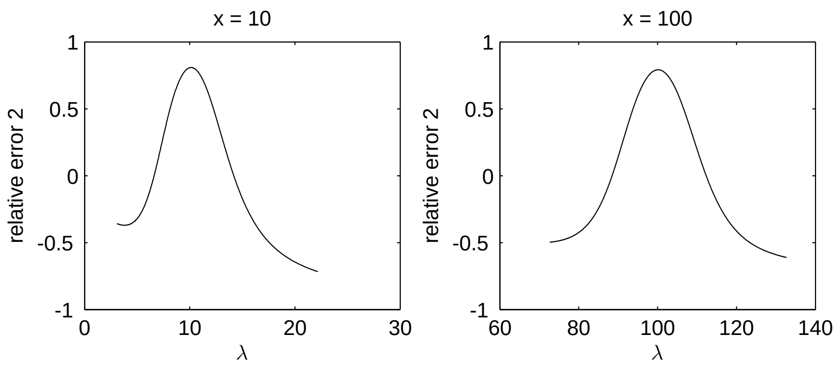

Fig. 4. Relative error of approximation $\widetilde{Q}_2$ for $x=10, 100$.

Normally distributed with zero mean and unit variance, then the expected value of δ is $\lambda^{-1}(\frac{1}{40}+\frac{1}{80}+\frac{3}{160}) = \frac{9}{160}\,\lambda^{-1}$, so that even when $\lambda=4$ there is less than a 3% chance of $\widetilde{Q}_{N2}(u)$ giving a value which is within δ of an integer value, leading to uncertainty about its correct rounding and hence requiring the additional check described in the Introduction.

3. TEMME ASYMPTOTIC APPROXIMATION

The Normal asymptotic approximations derived in the previous section have two major shortcomings. One is that they converge very slowly since the asymptotic expansion is in powers of $\lambda^{-1/2}$, and the other is that they are very inaccurate when $|w|$ is large.

Both of these shortcomings are addressed by starting from an expansion due to Temme [Temme 1979]. The leading order term in the expansion is

$$C(x) \approx \Phi(-\sqrt{x}\,\eta),$$

where

$$\eta = \sqrt{2\left(\frac{\lambda}{x} - 1 - \log\frac{\lambda}{x}\right)}, \tag{5}$$

with the square root being chosen to have the same sign as $\lambda{-}x$. i.e. $\eta > 0$ if $\lambda > x$, and $\eta < 0$ if $\lambda < x$. Hence, if $u = C(x)$ and $w = \Phi^{-1}(u)$ then $-\sqrt{x}\,\eta \approx w$, and therefore x is given (approximately) by the equation

$$\sqrt{2x\left(\frac{\lambda}{x} - 1 - \log\frac{\lambda}{x}\right)} = w$$

with the square root being chosen to match the sign of w. Equivalently,

$$\sqrt{2\left(1 - \frac{x}{\lambda} + \frac{x}{\lambda}\log\frac{x}{\lambda}\right)} = \frac{w}{\sqrt{\lambda}}$$

so if we define $r = x/\lambda$ and also define

$$f(r) \equiv \sqrt{2\,(1 - r + r\log r)},$$

with the sign of the square root matching the sign of $r{-}1$, then

$$r = f^{-1}(w/\sqrt{\lambda}).$$

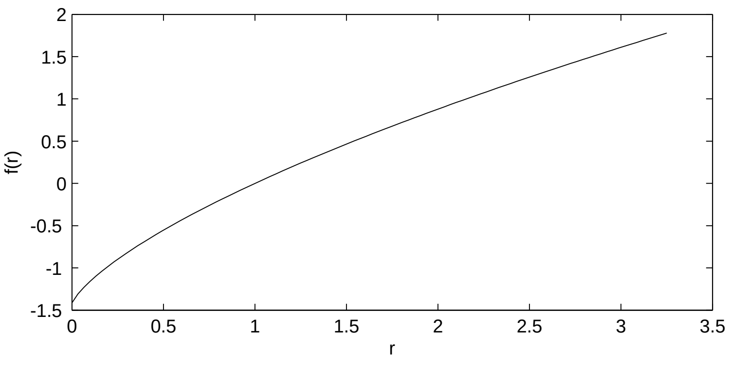

Fig. 5. The function $f(r)$.

Hence, if $u = C(x) = \Phi(w)$, then to leading order we have $x = \lambda\, r$ where $r = f^{-1}(w/\sqrt{\lambda})$.

The function $f(r)$ is plotted in Figure 5. Given $w/\sqrt{\lambda}$, the value $r = f^{-1}(w/\sqrt{\lambda})$ can be obtained by a Newton iteration, requiring 4 – 6 iterations typically to achieve full double precision accuracy. Alternatively, as we will discuss later, it can be obtained from a high order polynomial approximation to $f^{-1}(s)$. The minimum value of $f(r)$ is $-\sqrt{2}$. If $s < -\sqrt{2}$ then $f^{-1}(s)$ can be defined to equal zero. This will give $x=0$ which will lead to the use of bottom-up summation to determine the inverse Poisson CDF value.

Putting $x = \lambda\, r$, Temme derives a uniform asymptotic expansion which can be expressed in the form

$$C(\lambda\, r) = \Phi(\lambda^{\frac{1}{2}} f(r)) + \lambda^{-\frac{1}{2}}\, \phi(\lambda^{\frac{1}{2}} f(r)) \sum_{n=0}^{\infty} \lambda^{-n}\, a_n(r).$$

Differentiating this gives

$$\begin{aligned} C'(\lambda\, r) \;&=\; \lambda^{-1} \frac{\partial}{\partial r} \left\{ \Phi(\lambda^{\frac{1}{2}} f(r)) + \lambda^{-\frac{1}{2}}\, \phi(\lambda^{\frac{1}{2}} f(r)) \sum_{n=0}^{\infty} \lambda^{-n}\, a_n(r) \right\} \\ &=\; \lambda^{-\frac{1}{2}}\, \phi(\lambda^{\frac{1}{2}} f(r)) \left\{ f'(r) + \sum_{n=0}^{\infty} \lambda^{-n} \left(\lambda^{-1} a_n'(r) - f(r) f'(r)\, a_n(r) \right) \right\}, \end{aligned}$$

and we can prove inductively that the m^{th} derivative has the form

$$C^{(m)}(\lambda\, r) = \lambda^{-\frac{1}{2}}\, \phi(\lambda^{\frac{1}{2}} f(r)) \sum_{n=0}^{\infty} \lambda^{-n}\, b_{m,n}(r),$$

where, due to the fact that $\phi'(x) = -\,x\, \phi(x)$,

$$b_{m,0}(r) = \left(-\, f(r) f'(r)\right)^{m-1} \left(f'(r) - f(r) f'(r)\, a_0(r)\right).$$

We have already shown that if $u = C(x) = \Phi(w)$, and $r = f^{-1}(w/\sqrt{\lambda})$, then to leading order $x \approx \lambda\, r$. We now seek an asymptotic expansion of the form

$$x = \lambda\, r + p(\lambda, r), \qquad p(\lambda, r) \equiv \sum_{n=0}^{\infty} \lambda^{-n}\, c_n(r).$$

We have

$$
\begin{aligned}
u &= C(x) \\
&= C(\lambda\, r + p(\lambda, r)) \\
&= C(\lambda\, r) + \sum_{m=1}^{\infty} \frac{p^m(\lambda, r)}{m!}\, C^{(m)}(\lambda\, r) \\
&= \Phi(\lambda^{\frac{1}{2}} f(r)) + \lambda^{-\frac{1}{2}}\, \phi(\lambda^{\frac{1}{2}} f(r)) \sum_{n=0}^{\infty} \lambda^{-n}\, a_n(r) + \sum_{m=1}^{\infty} \frac{p^m(\lambda, r)}{m!}\, C^{(m)}(\lambda\, r).
\end{aligned}
$$

However $\Phi(\lambda^{\frac{1}{2}} f(r)) = \Phi(w) = u$, and so cancelling this term leaves

$$
\lambda^{-\frac{1}{2}}\, \phi(\lambda^{\frac{1}{2}} f(r)) \sum_{n=0}^{\infty} \lambda^{-n}\, a_n(r) + \sum_{m=1}^{\infty} \frac{p^m(\lambda, r)}{m!}\, C^{(m)}(\lambda\, r) = 0,
$$

from which the functions $c_n(r)$ may be determined by matching the coefficients for each power of λ. Considering the coefficients corresponding to the leading order power $\lambda^{-\frac{1}{2}}$ gives

$$
a_0(r) + \sum_{m=1}^{\infty} \frac{c_0^m(r)}{m!}\, \left(-f(r) f'(r)\right)^{m-1} \left(f'(r) - f(r)\, f'(r)\, a_0(r)\right) = 0,
$$

and hence

$$
a_0(r) + \Big(1 - \exp\left(-f(r)\, f'(r)\, c_0(r)\right)\Big) \left(\frac{1}{f(r)} - a_0(r)\right) = 0.
$$

Re-arranging this gives $c_0(r) = \left(f(r)\, f'(r)\right)^{-1} \log\left(1 - f(r)\, a_0(r)\right).$ Since $f(r)\, f'(r) = \frac{1}{2}\, (f(r)^2)' = \log r$, and Temme's paper [Temme 1979] gives

$$
a_0 = \frac{\sqrt{r}}{1-r} + \frac{1}{f(r)},
$$

we finally obtain

$$
c_0(r) = \frac{\log\left(\, f(r)\, \sqrt{r}/(r-1)\,\right)}{\log r}.
$$

This gives us the approximation

$$
\widetilde{Q}_{T1}(u) = \lambda\, r + c_0(r),
$$

where $r = f^{-1}(w/\sqrt{\lambda})$ and $w = \Phi^{-1}(u)$.

To check that this is consistent with the Normal asymptotic expansion derived in the previous section, note that a Taylor series expansion of $f(r)$ around $r=1$ gives

$$
f(r) = (r-1) - \tfrac{1}{6}(r-1)^2 + \tfrac{5}{72}(r-1)^3 + O((r-1)^4),
$$

from which it follows that

$$
f^{-1}(w/\sqrt{\lambda}) = 1 + w/\sqrt{\lambda} + \tfrac{1}{6}(w/\sqrt{\lambda})^2 - \tfrac{1}{72}(w/\sqrt{\lambda})^3 + O(\lambda^{-2}).
$$

Furthermore, a Taylor series expansion of $c_0(r)$ around $r=1$ gives

$$
c_0(r) = \tfrac{1}{3} - \tfrac{1}{36}(r-1) + O((r-1)^2),
$$

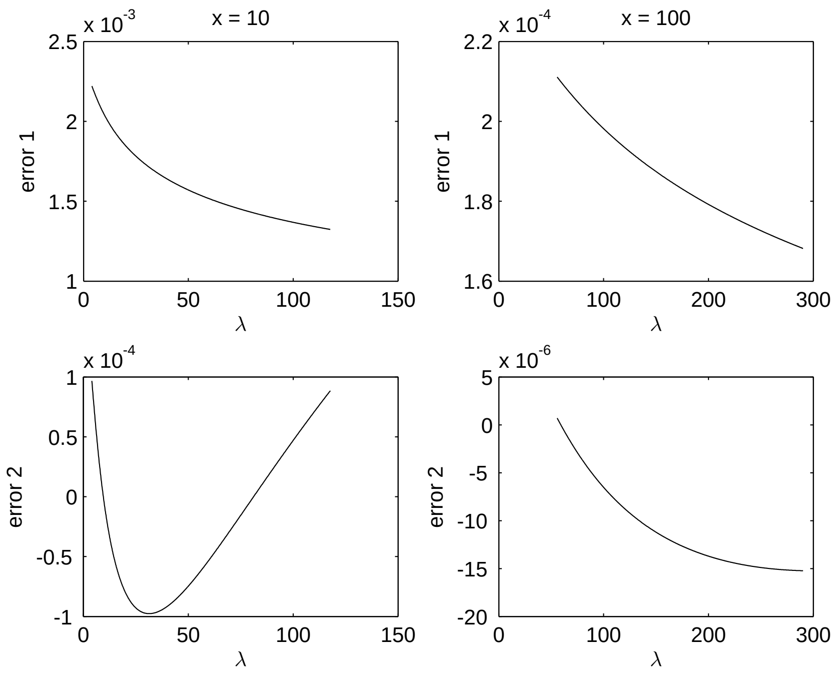

Fig. 6. Error in $\widetilde{Q}_{T1}$, $\widetilde{Q}_{T2}$ approximations based on Temme expansion.

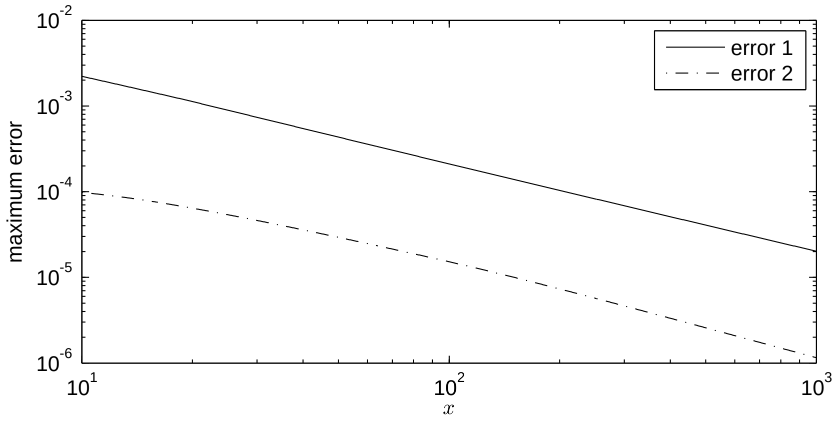

Fig. 7. Maximum error in $\widetilde{Q}_{T1}$, $\widetilde{Q}_{T2}$ approximations over the single precision range.

and hence we obtain

$$\widetilde{Q}_{T1}(u) \ = \ \lambda + \lambda^{1/2}\, w + (\tfrac{1}{3} + \tfrac{1}{6}\, w^2) + \lambda^{-1/2}\, (-\tfrac{1}{36}\, w - \tfrac{1}{72}\, w^3) \ + \ O(\lambda^{-1}),$$

which matches the leading order terms in the Normal asymptotic expansion.

The advantage of this new approximation, compared to the Normal asymptotic approximation $\widetilde{Q}_{N2}$ is its accuracy over the full range of possible values for w, not

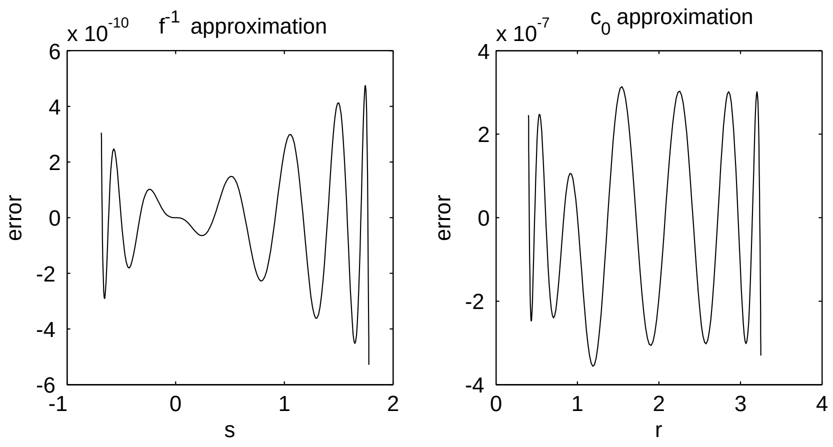

Fig. 8. Errors in $f^{-1}(s)$ and $c_0(r)$ approximations.

just $|w| < 3$. The top plots in Figure 6 show the error in this $\widetilde{Q}_{T1}$ approximation for $x = 10, 100$ and u taking values in the single precision range $[10^{-38}, 1-6\times10^{-8}]$.

Based on the form of the error, and bearing in mind that the Normal asymptotic analysis gave

$$\widetilde{Q}_{N3}(\tfrac{1}{2}) = \lambda + \tfrac{1}{3} - \tfrac{8}{405}\,\lambda^{-1}$$

some numerical experimentation led to an ad-hoc correction giving the improved approximation

$$\widetilde{Q}_{T2}(u) = \widetilde{Q}_{T1}(u) - \frac{0.0218}{\widetilde{Q}_{T1}(u) + 0.065\lambda}.$$

The bottom plots in Figure 6 show that this is significantly more accurate. Figure 7 plots the maximum errors for the two approximations as a function of x over the single precision range of values for u.. A bound on the maximum error in $\widetilde{Q}_{T2}$ over the full double precision range for u is given by $\delta = 0.01/x$, and further testing shows that this same bound applies also to the double precision range $[10^{-308}, 1-10^{-308}]$.

The cost of evaluating the functions $c_0(r)$ and $f^{-1}(s)$ can be reduced greatly by constructing polynomial approximations for $0.4 < r < 3.25$ (most of the range plotted in Figure 5) and $s_{min} < s < s_{max}$ with $s_{min} = f(0.4) \approx -0.6834$, $s_{max} = f(3.25) \approx 1.778$.

Since $f^{-1}(s) = 1 + s + \frac{1}{6}s^2 + O(s^3)$, the polynomial approximation to $f^{-1}(s)$ is defined to be a degree 14 polynomial

$$p_1(s) = 1 + s + \tfrac{1}{6}s^2 + \sum_{n=3}^{14} a_n s^n,$$

with the coefficients a_n chosen through a weighted least-squares minimisation. The left-hand plot in Figure 8 shows the resulting error over the corresponding s interval.

Similarly, since $c_0(r) = \frac{1}{3} + O(r)$, the approximation to $c_0(r)$ is defined to be the degree 12 polynomial

$$p_2(r) = \tfrac{1}{3} + \sum_{n=1}^{12} b_n r^n,$$

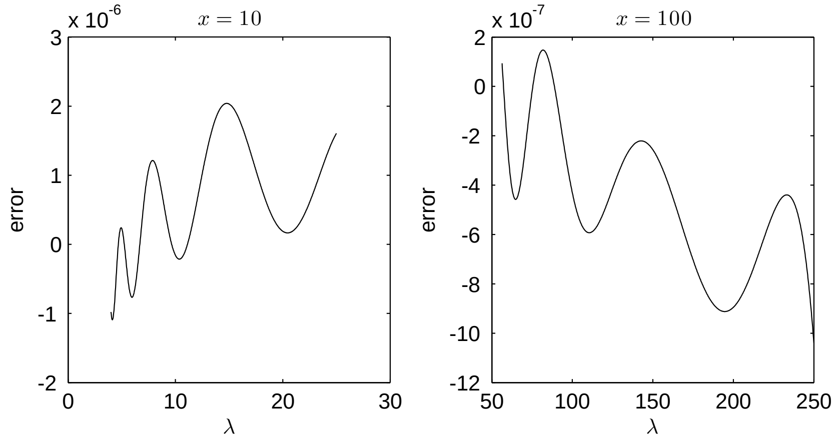

Fig. 9. Error in $\widetilde{Q}_{T3}$ approximation based on Temme expansion.

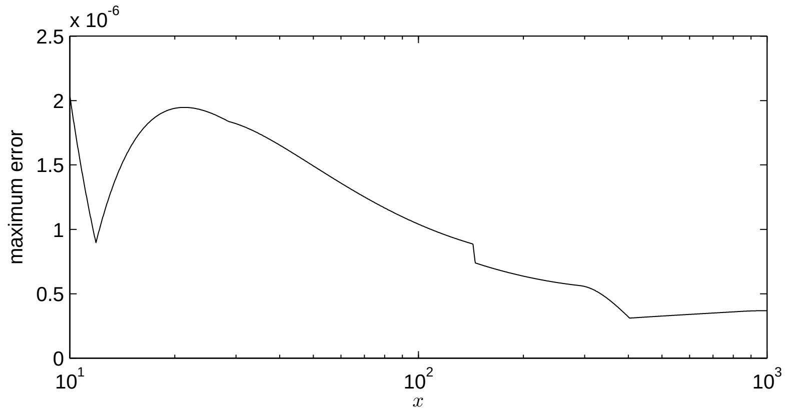

Fig. 10. Maximum error in $\widetilde{Q}_{T3}$ approximation based on Temme expansion.

with the coefficients b_n again chosen through a weighted least-squares minimisation. The right-hand plot in Figure 8 shows the resulting error.

In addition to these polynomial approximations to $f^{-1}(s)$ and $c_0(r)$, we improve the approximation of $Q(u)$ by defining

$$\widetilde{Q}_{T3}(u) = \lambda\, r + p_2(r) + p_3(r)/\lambda,$$

where $r = p_1(w/\sqrt{\lambda}) = p_1(\Phi^{-1}(u)/\sqrt{\lambda})$. Here $p_3(r)$ is a degree 10 polynomial with coefficients chosen through a weighted least-squares minimisation, minimising the difference from $Q(u)$ over the range $0.4 < r < 3.25$, $10 \leq \lambda \leq 100$.

Figures 9 and 10 show the errors in this new approximation. The $\widetilde{Q}_{T3}$ approximation can use the error bound $\delta = 2 \times 10^{-6}$.

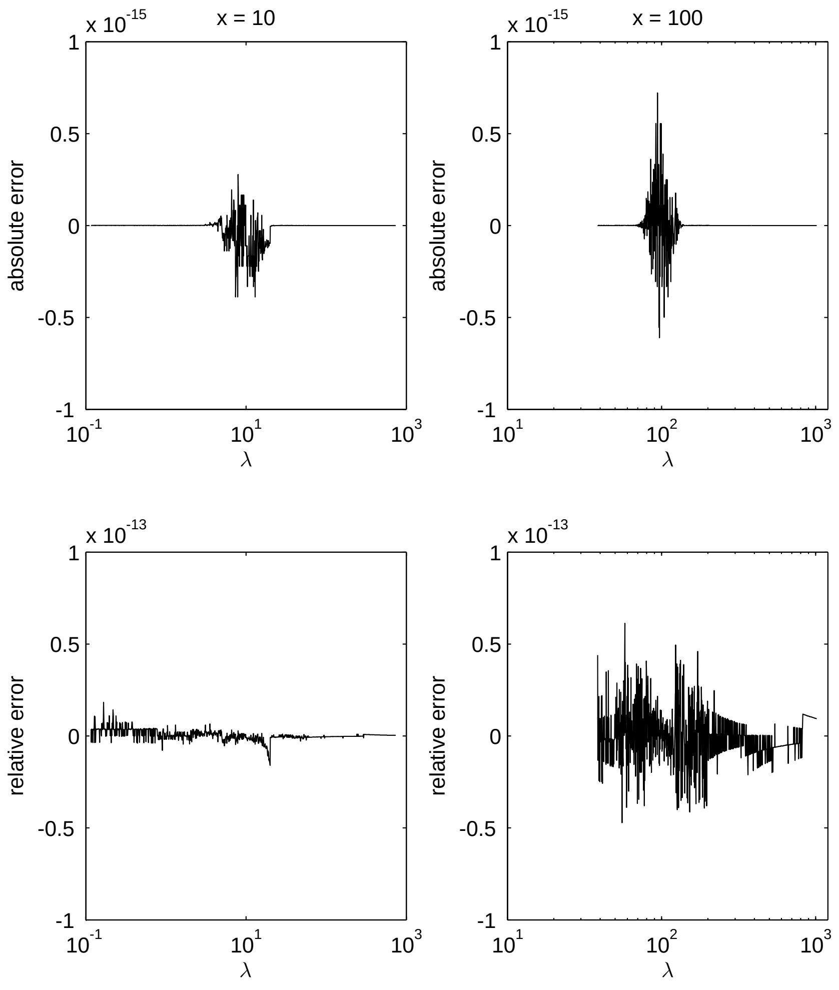

Fig. 11. Error and relative error in Temme approximation for $C(x)$ for $x = 10, 100$.

4. EVALUATION OF $C(X)$ FOR $X \geq 10$

In the rare cases when the approximations are not sufficiently precise to know which integer to round to, it becomes necessary to evaluate $C(x)$ for integer values of $x \geq 10$.

If $\lambda/2 \leq x \leq 2\lambda$, then Temme's 1987 algorithm [Temme 1987] can be used with $N = 12$ terms (see [Temme 1987] for full details). If $x \leq \lambda/2$, then since x is an integer we can use $C(x) = \overline{C}(x-1)$ and use a simple summation, starting at $x-1$ and then working downwards. The ratio of successive terms in the summation is less than $1/2$, because $x \leq \lambda/2$, so no more than 50 terms will be required to achieve full double precision accuracy. Similarly, if $x \geq 2\,\lambda$, we can compute $1 - C(x)$ by summing upwards, starting at x and finishing when the increments become negligible. These summations require the computation of $\exp(-\lambda)\, \lambda^{x-1} \,/\, \Gamma(x)$, which is discussed in Appendix B.

This is similar to the implementation of the incomplete gamma function in the NAG mathematical library [NAG 2014b], which uses Temme's method when $x \geq 20$ and

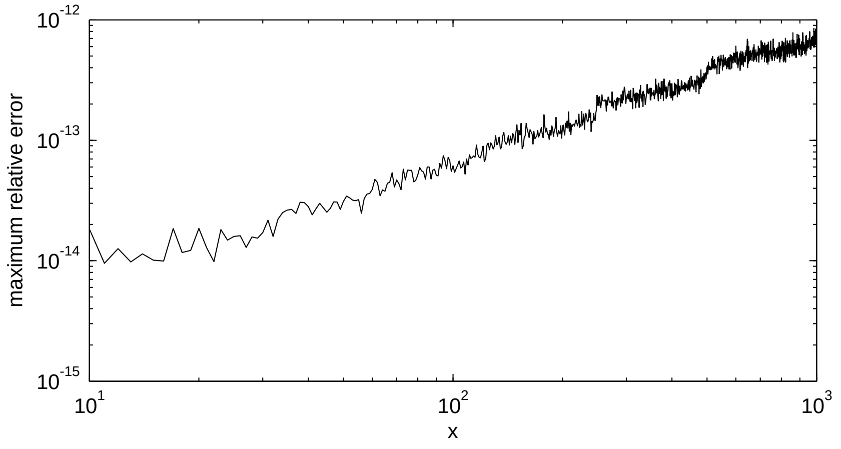

Fig. 12. Maximum relative error in Temme approximation for $C(x)$.

$0.7x \le \lambda \le 1.4x$, and otherwise follows the approach due to Gautschi [Gautschi 1979]. Other references on computing the incomplete gamma function include [Temme 1994; Winitzki 2003].

Figure 11 plots the error compared to MATLAB's `gammainc` function for $x = 10, 100$, and λ in the range corresponding to the full range of double precision values u in the open interval $(0, 1)$.

Figure 11 also shows the error relative to $C'(x)$, which gives an estimate for the equivalent error in x since $\Delta x \approx \Delta C/C'(x)$. Figure 12 plots the maximum relative error over a range of values for x. The irregular nature of the error suggests it is due to floating point rounding error, not the error in Temme's asymptotic approximation, and further investigation confirms it is due to the rounding error in computing η, as defined in (5).

5. BOTTOM-UP AND TOP-DOWN SUMMATION

As explained in the Introduction, it is standard to use bottom-up summation when $u \le \frac{1}{2}$, and top-down summation for $u > \frac{1}{2}$, because this gives the greatest accuracy. However, when executing on a GPU, this approach almost doubles the computational cost because some of the 32 threads in a warp will perform the bottom-up summation, while others work top-down, and so the computational cost is the sum of the two.

Instead, we use bottom-up summation for all values of u, but monitor the accuracy when $u > \frac{1}{2}$ to check whether it is necessary to re-do the calculation with a top-down summation. Given a bound δ on the maximum summation error, if we define

$$S_n = \exp(-\lambda) \sum_{m=0}^{n} \frac{\lambda^m}{m!},$$

then when $S_{n-1}+\delta < u < S_n-\delta$, we know that n is the correct value despite the floating point errors in the computation of S_{n-1} and S_n, whereas if $S_n-\delta < u < S_n+\delta$, we switch to top-down summation to determine the correct value. The pseudo-code to do this, starting from $n = 0$ is :

```
$a := \exp(-\lambda)$;
$S := a$;
while $S + \delta < u$ do
    $n := n + 1$;
    $a := a\,\lambda/n$;
    $S := S + a$;
end
if $S - \delta > u$ then
    return $n$;
else
    perform top-down summation instead
end
```

When $u \le \frac{1}{2}$, δ is set to zero since the bottom-up summation gives the best accuracy. For $u > \frac{1}{2}$, we use $\delta = 10^{-6}$ for single precision arithmetic. This is because we use bottom-up summation only for n up to approximately 10, giving a maximum of 20 floating point operations, each with a maximum relative error of 5×10^{-8}. This bound assumes the worst-case scenario in which each of the rounding errors has the same sign, and has been verified by exhaustive investigation of the accumulated errors in computing $\exp(-\lambda) \sum_{m=1}^{50} \frac{\lambda^m}{m!}$ for $1 < \lambda < 10$. Given 10 possible return values, each with an exclusion zone of width 2×10^{-6}, then if u is uniformly distributed on $(0, 1)$ this gives a probability of approximately 2×10^{-5} that the single precision bottom-up summation will fail to return a value, and we will instead have to proceed to the top-down summation to determine the correct value for n.

For double precision, we use $\delta = 10^{-13}$ when $u > \frac{1}{2}$.

One drawback of the bottom-up algorithm is that it requires a relatively expensive division by n in each iteration. This can be avoided by subtracting $u - \delta$ from S, and then dividing both S and δ by the factor $\exp(-\lambda)\ \lambda^n/n!$ (which changes during the calculation as n increases) resulting in:

```
$S := 1 - \exp(\lambda)\,(u - \delta)$;
$\delta := \exp(\lambda)\,\delta$;
while $S < 0$ do
    $n := n + 1$;
    $S := S\,n/\lambda + 1$;
    $\delta := \delta\,n/\lambda$;
end
if $S > 2\delta$ then
    return $n$;
else
    perform top-down summation instead
end
```

The reciprocal λ^{-1} can be pre-computed so that now the computational cost is equal to the cost of the exponential $\exp(\lambda)$ plus approximately $5n$ floating point operations (each iteration requires 1 addition, 2 multiplications, 1 fused multiply/add, and 1 conditional test).

One last technical detail is that when λ is large, but u is exceptionally small so that $n < 10$ and bottom-up summation is still required, it is possible for $\exp(\lambda)$ to exceed the

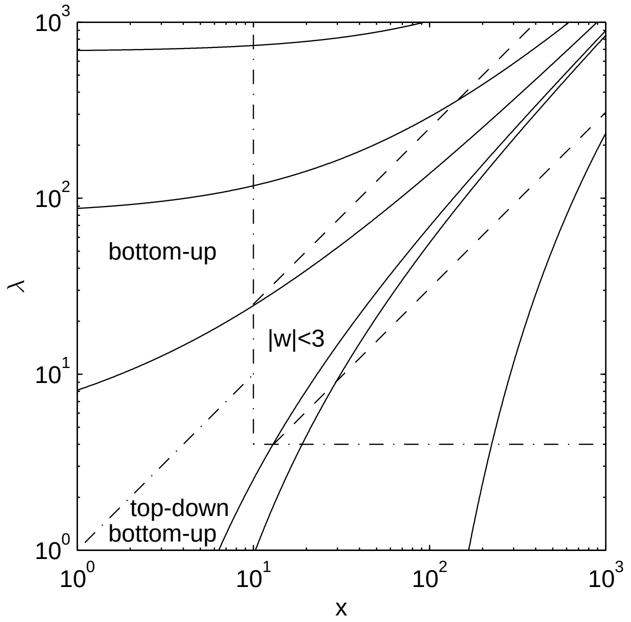

Fig. 13. Different regions important to the CPU and GPU algorithms. The innermost curved lines bound the region in which the CPU algorithm uses the Normal approximation. The outer pairs of curved lines correspond to the ranges of single-precision and double-precision floating point variables. The region on the left uses bottom-up summation; the region at the bottom starts with bottom-up summation then switches to top-down if additional accuracy is needed.

maximum floating point value. This problem can be avoided by replacing $\exp(\lambda)(u-\delta)$ by $e((u-\delta)e)$, where $e = \exp(\lambda/2)$.

Multiplying the terms a and S by $\exp(\lambda)$, the pseudo-code for top-down summation, starting from some suitable value n is:

```
$a := \lambda^n/n!$;
$S := a - \exp(\lambda)\,(1-u)$;
**while** $S < 0$ **do**
    $a := a\,n/\lambda$;
    $S := S + a$;
    $n := n - 1$;
**end**
**return** $n$;
```

The only remaining issue is the initial value n which is chosen so that the sum of the terms which are ignored through truncating the series representation is negligible compared to $1-u$.

6. INVERSE POISSON CDF ALGORITHMS

Combining all of the elements of the previous sections, the pseudo-codes for the CPU and GPU algorithms to compute the inverse Poisson CDF function $\overline{C}^{-1}(u)$ are given in Algorithms 1 – 3. For the reasons explained in the Introduction, and illustrated in Table I, the CPU algorithm uses the Normal approximation derived in Section 2, while the GPU algorithms use the approximations based on Temme's asymptotic expansion, as derived in Section 3.

In the rare cases when the approximations are not sufficiently accurate to be certain about the rounding to the appropriate integer, the CPU algorithm and the double precision GPU algorithm both use the methods of computing $C(x)$ outlined in Section 4. The approximations in the single precision GPU algorithm are sufficiently accurate that there is no need to improve upon them.

Figure 13 illustrates the different regions which are important to the three algorithms. The dot-dash lines show the boundaries for bottom-up/top-down summation, used by all three algorithms. The dashed lines specify the central region $0.4 < r < 3.25$, which corresponds to $s_{min} < s < s_{max}$, as used by the GPU algorithms. The innermost pair of curved lines correspond to $|w| = 3$, which is the boundary for the Normal approximation in the CPU algorithm. The next pair of curved lines correspond to $u = 10^{-38}$ and $u = 1 - 6 \times 10^{-8}$, which are the extreme single precision floating point values in the open interval $(0, 1)$. Finally, the outermost two curved lines correspond to $u = 10^{-308}$ and $u = 1 - 10^{-308}$, which are the extreme double precision floating point values when implementing both the inverse CDF function and its complementary counterpart, as discussed in the Introduction.

ALGORITHM 1: CPU algorithm to compute the inverse Poisson CDF function $n = \overline{C}^{-1}(u)$.

```
Data: $\lambda$, $u$
Result: $n = \overline{C}^{-1}(u)$
if $\lambda > 4$ then
    $w := \Phi^{-1}(u)$;
    if $|w| < 3$ then
        $x := \widetilde{Q}_{N2}(w)$;
        $\delta := (\frac{1}{40} + \frac{1}{80}\, w^2 + \frac{1}{160}\, w^4) \,/\, \lambda$;
    else
        $r := f^{-1}(w/\sqrt{\lambda})$ ;                /* computed by Newton iteration */
        $x := \lambda\, r + c_0(r)$;
        $x := x - 0.0218/(x{+}0.065\lambda)$;
        $\delta := 0.01/\lambda$;
    end
end
$n := \lfloor x + \delta \rfloor$;
if $x > 10$ then
    if $x{-}n > \delta$ then
        return $n$;
    else if $C(n) < u$ then
        return $n$;
    else
        return $n{-}1$;
    end
end
use bottom-up summation to determine $n$
if $u{>}0.5$ and not accurate enough then
    use top-down summation to determine $n$
end
```

ALGORITHM 2: Single precision GPU algorithm to compute the inverse Poisson CDF function.

```
Data: $\lambda$, $u$
Result: $n = \overline{C}^{-1}(u)$
if $\lambda > 4$ then
    $w := \Phi^{-1}(u)$;
    $s := w/\sqrt{\lambda}$;
    if $s_{min} < s < s_{max}$ then
        $r := p_1(s)$;
        $x := \lambda\, r + p_2(r) + p_3(r)/\lambda$;
    else
        $r := f^{-1}(s)$ ;                                /* computed by Newton iteration */
        $x := \lambda\, r + c_0(r)$;
        $x := x - 0.0218/(x + 0.065\lambda)$;
    end
    $n := \lfloor x \rfloor$;
    if $x > 10$ then
        return $n$;
    end
end
use bottom-up summation to determine $n$;
if $u > 0.5$ and not accurate enough then
    use top-down summation to determine $n$;
end
return $n$
```

7. IMPLEMENTATION, VALIDATION AND PERFORMANCE

The double precision CPU function `poissinv(u,lam)` and its complementary inverse `poisscinv(u,lam)` are defined in a header file `poissinv.h`. Their implementations use a common core function `poissinv_core(u,v,lam)` with $u+v=1$, so that

$$\texttt{poissinv(u,lam)} = \texttt{poissinv\_core(u,1-u,lam)}$$
$$\texttt{poisscinv(v,lam)} = \texttt{poissinv\_core(1-v,v,lam)}$$

For the purposes of performance comparison, the header file also includes a CPU version of the vector algorithm `poissinv_v(u,lam)`.

The single-precision and double-precision GPU algorithms have been implemented in CUDA C [NVIDIA 2014] in the header file `poissinv_cuda.h`, defining `poissinvf(u,lam)` and `poissinv(u,lam)` as inline *device* functions which are called from a user's *kernel* function executing on the GPU.

The implementations all require two special functions. The complementary error function `erfc`, which is used in the computation of $C(x)$ from Temme's 1987 algorithm, as detailed in Section 4, is part of the C++11 standard, and so is available as part of the standard math library on almost all systems.

The other function which is needed is `normcdfinv` which computes $\Phi^{-1}(u)$. Alternatively, one could use `erfcinv` which is the inverse of `erfc` and equivalent to $\Phi^{-1}(u)$ with a simple scaling. Unfortunately, neither of these is part of the C++11 standard. NVIDIA provides `normcdfinv` as part of its math library. For the CPU implementations, we used a C implementation of an approximation developed by Wichura [Wichura 1988], having independently verified that its relative error is less than 10^{-16} compared to the value computed by MATLAB.

ALGORITHM 3: Double precision GPU algorithm to compute the inverse Poisson CDF function.

```
Data: $\lambda$, $u$
Result: $n = \overline{C}^{-1}(u)$
if $\lambda > 4$ then
    $w := \Phi^{-1}(u)$;
    $s := w/\sqrt{\lambda}$;
    if $s_{min} < s < s_{max}$ then
        $r := p_1(s)$;
        $x := \lambda r + p_2(r) + p_3(r)/\lambda$;
        $\delta := 2\times 10^{-5}$;
    else
        $r := f^{-1}(s)$ ;                                /* computed by Newton iteration */
        $x := \lambda r + c_0(r)$;
        $x := x - 0.0218/(x+0.065\lambda)$;
        $\delta := 0.01/\lambda$;
    end
    $n := \lfloor x+\delta \rfloor$;
    if $x > 10$ then
        if $x-n > \delta$ then
            return $n$;
        else if $C(n) < u$ then
            return $n$;
        else
            return $n-1$;
        end
    end
end
use bottom-up summation to determine $n$;
if $u > 0.5$ and not accurate enough then
    use top-down summation to determine $n$
end
return $n$;
```

To check the accuracy for a given value of λ, GCC's `quadlib` quadruple-precision mathematical library [Foundation] (which requires GCC version 4.6 or later) is used to compute $u_n = \overline{C}(n)$ for all integers n for which $u_n < 1 - 10^{-300}$. For each n, u_n is rounded both down and up to the nearest single or double-precision floating point values u_n^-, u_n^+.

These are compared to the corresponding values $\widetilde{u}_n^-, \widetilde{u}_n^+$ obtained from `poissinvf` or `poissinv` by using repeated interval bisection to determine two consecutive floating point values $\widetilde{u}_n^-, \widetilde{u}_n^+$ with $Q(\widetilde{u}_n^-) < n$ and $Q(\widetilde{u}_n^+) \geq n$, where $Q(u)$ represents the value returned by `poissinvf` or `poissinv`. [1]

Since

$$\begin{aligned} \overline{C}^{-1}(u) = n &\implies u_n^+ \leq u \leq u_{n+1}^- \\ Q(u) = m &\implies \widetilde{u}_m^+ \leq u \leq \widetilde{u}_{m+1}^- \end{aligned}$$

[1]In most cases $Q(\widetilde{u}_n^-) = n-1$ and $Q(\widetilde{u}_n^+) = n$, but the difference between the two can be greater than 1 when either u or $1-u$ is extremely small.

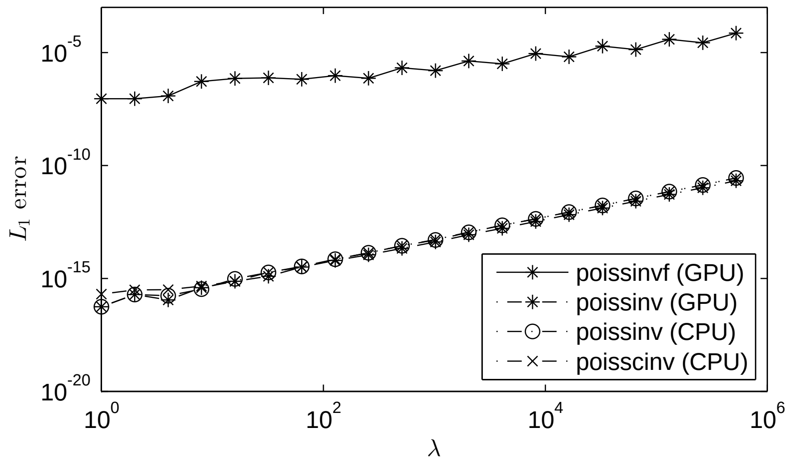

Fig. 14. L_1 errors of `poissinvf`, `poissinv` and `poisscinv` functions.

it follows that

$$m \geq n+2 \implies \widetilde{u}^+_{n+2} \leq u \leq u^-_{n+1}$$
$$m \leq n-2 \implies u^+_n \leq u \leq \widetilde{u}^-_{n-1}$$

However, for all values of λ tested, it is found that $\widetilde{u}^+_{n+1} > u^-_n$ and $u^+_{n+1} > \widetilde{u}^-_n$ for all n, and hence $|\overline{C}^{\,-1}(u) - Q(u)| \leq 1$.

The L_1 error

$$\int_0^1 |\overline{C}^{\,-1}(u) - Q(u)| \, \mathrm{d}u,$$

can be computed as

$$\sum_n \tfrac{1}{2} \left| (u^- - \widetilde{u}^-_n) + (u^+ - \widetilde{u}^+_n) \right|.$$

Figure 14 plots this error for both the single precision `poissinvf` and the double precision `poissinv` and `poisscinv`. The accuracy is excellent, and in line with what one would expect from single and double precision computations. In the case of `poissinvf`, the main error at larger values of λ is due to the error in computing $\Phi^{-1}(u)$; since this is multiplied by $\sqrt{\lambda}$, the resulting L_1 error is approximately proportional to $\sqrt{\lambda}$. In the case of `poissinv`, the error is due to the error in computing $C(n)$, and this is approximately proportional to λ as shown in Section 4.

Finally, Tables II and III document the performance of the algorithms, measured in samples per second. The first table gives performance on a 4-core 130W 3.6GHz Intel Xeon E5-1620 based on the Sandy Bridge architecture. The results are obtained using two compilers, GCC 4.6.3 and Intel ICC 13.0, both with full optimisation enabled (`-O3`). The computations use only one core; performance would scale linearly if additional cores were used.

The most important observation from the results is that for $\lambda \geq 4$ the algorithm designed for the CPU is approximately twice as fast as the algorithm which was designed for vector execution on GPUs. Further testing reveals that about the cost of the

Table II. Samples/sec for `poissinv` and `poissinv_v` on Intel Xeon E5-1620 using two compilers: GCC and Intel's ICC

	gcc 4.6.3		icc 13.0	
λ	poissinv	poissinv_v	poissinv	poissinv_v
2	3.30e07	3.30e07	4.58e07	4.64e07
8	1.34e07	7.51e06	1.47e07	8.37e06
32	1.60e07	8.91e06	1.76e07	9.73e06
128	1.61e07	8.91e06	1.76e07	9.73e06
normcdfinv	2.75e07		5.87e07	

Table III. Samples/sec for `poissinvf` and `poissinv` on two NVIDIA GPUs: GTX750 Ti (Maxwell) and K40 (Kepler)

	GTX750 Ti		K40	
λ	poissinvf	poissinv	poissinvf	poissinv
2	6.04e09	4.76e08	2.53e10	7.54e09
8	2.97e09	1.67e08	1.31e10	2.91e09
32	4.35e09	3.19e08	1.79e10	6.13e09
128	4.35e09	3.19e08	1.79e10	6.13e09
mixed	2.53e09	1.37e08	1.07e10	2.33e09
normcdfinvf	1.25e10		4.54e10	
normcdfinv		4.36e08		1.16e10

correction procedure in the algorithm contributes 2-4% of the total cost of the CPU algorithm, compared to less than 0.5% of the vector algorithm. Referring back to the discussion in the Introduction, this corresponds to $2\delta\, C_C/C_Q \approx 0.03$ for the CPU algorithm and 0.003 for the vector algorithm. Hence, it is to be expected that the average cost is least for the CPU algorithm which has the simpler primary approximation.

Note also that the performance of the CPU algorithm using GCC is very similar to the performance of the `normcdfinv` function given in the final line; for an unknown reason, ICC gives double the performance for `normcdfinv`.

Table III gives the performance of `poissinvf` and `poissinv` using CUDA 6.0 on two NVIDIA GPUs: a 60W GTX 750 Ti which is an entry-level consumer graphics card based on the latest Maxwell GPU architecture, and a 235W Tesla K40 which is the current high-end HPC card based on the previous generation Kepler architecture. Full optimisation was enabled (`-O3 --use_fast_math`), with the second flag enabling use of the SFU (Special Function Unit) for single precision intrinsics such as `log`. In the case of the GTX 750 Ti the double precision performance is 24$\times$ worse than single precision, whereas with the K40 the difference is only 3$\times$. The rows for specified values of λ compute the inverse CDF for a uniform sequence of 2^{24} values u_n covering the whole $(0,1)$ interval. The row marked "mixed" uses mixed pairs of values (u,λ) so that each warp is likely to have some threads performing bottom-up summations, while others in the same warp have to evaluate the asymptotic approximation. Because of this warp divergence, the average cost of each sample in the mixed case is roughly equal to the sum of the costs of the samples for $\lambda\!=\!2$ and $\lambda\!=\!32$. It is also similar to the cost for $\lambda\!=\!8$ since most threads in that region must first evaluate the asymptotic approximation, and then perform the bottom-up summation.

8. CONCLUSIONS

In this paper we have derived a number of approximations of the inverse incomplete gamma function which can be used to implement the inverse Poisson cumulative distribution function.

The Normal asymptotic approximation labelled $\widetilde{Q}_{N2}$ is well suited to CPU implementations since it directly gives the correct integer output more than 99% of the time. In the other 1% there is a more expensive secondary step which is used to determine the correct value, but this contributes little to the overall average cost.

The asymptotic approximations $\widetilde{Q}_{T2}$ and $\widetilde{Q}_{T3}$ based on the Temme expansion are a better choice for GPU implementation because the primary approximation $\widetilde{Q}_{T3}$ gives the correct output more of the time, reducing the costs due to different threads within the same GPU warp branching differently. Indeed, in the case of single precision arithmetic the approximations are so accurate that no correction step is required.

The approximations in this paper could be used to initialise a Newton iteration to determine the inverse of the incomplete gamma function [DiDonato and Morris 1986; Temme 1992; Gil et al. 2012]. Another direction for future work is the generation of similar asymptotic approximations for the inverse of the incomplete beta function to facilitate the fast inversion of the CDF for the Binomial distribution.

A. INVERSE FUNCTION ASYMPTOTIC EXPANSION

Suppose that a smooth function $f(x)$ has the expansion

$$f(x) = f_0(x) + \varepsilon\, f_1(x) + \varepsilon^2 f_2(x) + \varepsilon^3 f_3(x) + \ldots$$

where $f_0(x)$ is monotonic, and we seek a similar expansion

$$g(x) = g_0(x) + \varepsilon\, g_1(x) + \varepsilon^2 g_2(x) + \varepsilon^3 g_3(x) + \ldots$$

for the inverse function defined by $g(f(x)) = x$. Performing a Taylor series expansion in ε of $g(f(x))$, and equating the coefficients for each power of ε, we obtain:

$$\begin{aligned}
\varepsilon^0 :&\quad g_0(y) = x \\
\varepsilon^1 :&\quad g_1(y) + g_0'(y)\, f_1(x) = 0 \\
\varepsilon^2 :&\quad g_2(y) + g_1'(y)\, f_1(x) + g_0'(y)\, f_2(x) + \tfrac{1}{2} g_0''(y)\, (f_1(x))^2 = 0 \\
\varepsilon^3 :&\quad g_3(y) + g_2'(y)\, f_1(x) + g_1'(y)\, f_2(x) + \tfrac{1}{2} g_1''(y)\, (f_1(x))^2 \\
&\quad + g_0'(y)\, f_3(x) + g_0''(y)\, f_1(x)\, f_2(x) + \tfrac{1}{6} g_0'''(y)\, (f_1(x))^3 = 0
\end{aligned}$$

where $y = f_0(x)$. Hence we have

$$\begin{aligned}
g_0(y) &= f_0^{-1}(y) \\
g_1(y) &= -g_0'(y)\, f_1(x) \\
g_2(y) &= -g_1'(y)\, f_1(x) - g_0'(y)\, f_2(x) - \tfrac{1}{2} g_0''(y)\, (f_1(x))^2 \\
g_3(y) &= -g_2'(y)\, f_1(x) - g_1'(y)\, f_2(x) - \tfrac{1}{2} g_1''(y)\, (f_1(x))^2 \\
&\quad -g_0'(y)\, f_3(x) - g_0''(y)\, f_1(x)\, f_2(x) - \tfrac{1}{6} g_0'''(y)\, (f_1(x))^3
\end{aligned}$$

with $x = f_0^{-1}(y) = g_0(y)$.

This allows us to determine $g_0(y)$, $g_1(y)$, $g_2(y)$, $g_3(y)$ sequentially, and the construction is easily extended to obtain additional terms in the asymptotic expansion.

B. COMPUTING $\mathrm{EXP}(-\lambda)\,\lambda^{X-1}\,/\,\Gamma(X)$

The summations in Section 4 require the computation of

$$t = \exp(-\lambda)\,\lambda^{x-1}\,/\,\Gamma(x),$$

but it is surprisingly tricky to do this accurately. When x and λ are large, the first term is very small, and the last two terms are extremely large, and even the range of double precision floating point variables is insufficient.

This leads to the idea of first computing

$$\log t = -\lambda + (x-1)\log\lambda - \log\Gamma(x).$$

The problem here is that $(x-1)\log\lambda$ and $\log\Gamma(x)$ are large and similar in magnitude. For example, when $\lambda = 200, x = 400$, then $(x-1)\log\lambda \approx 2114.0$ and $\log\Gamma(x) \approx 1994.5$, while the final result is relatively small, with $\log t \approx -80.5$. Hence, there is a large cancellation which greatly increases the rounding error due to floating point precision.

The solution to this is to not use the $\log\Gamma(x)$ function which is made available in mathematical libraries, but instead use the asymptotic approximation which is typically used internally. One standard approximation is

$$\log\Gamma(x) \approx (x-\tfrac{1}{2})\,\log x - x + \tfrac{1}{2}\log(2\,\pi) + S(x),$$

where

$$S(x) = \tfrac{1}{12}\,x^{-1} - \tfrac{1}{360}\,x^{-3} + \tfrac{1}{1260}\,x^{-5} - \tfrac{1}{1680}\,x^{-7} + \tfrac{1}{1188}\,x^{-9}.$$

Using this gives

$$\log t \approx -x\log(x/\lambda) + (x-\lambda) - \log\lambda + \tfrac{1}{2}\log x - \tfrac{1}{2}\log(2\,\pi) - S(x),$$

and therefore

$$t \;\approx\; \sqrt{\frac{x}{2\pi\lambda^2}}\;\exp\Big(-x\log(x/\lambda) + (x-\lambda) - S(x)\Big).$$

In the above derivation, the key step which improves the accuracy which can be achieved is $x\log\lambda - x\log x = -x\log(x/\lambda)$. Since $\log\lambda$ and $\log x$ are both typically much larger than $\log(x/\lambda)$, the expression on the right can be evaluated much more accurately than the one on the left.

The final expression for t can be evaluated relatively cheaply. When $\lambda, x > 10$, it gives a relative error of 10^{-13} in double precision.

This is similar to the way in which the gamma distribution probability density function is computed in the NAG mathematical library [NAG 2014a].

REFERENCES

J.H. Ahrens and U. Dieter. 1982. Computer generation of Poisson deviates from modified Normal distributions. *ACM Trans. Math. Software* 8, 2 (1982), 163–179. DOI:http://dx.doi.org/10.1145/355993.355997

A. Asmussen and P. Glynn. 2007. *Stochastic Simulation*. Springer, New York.

L. Devroye. 1991. Expected time analysis of a simple recursive Poisson random variate generator. *Computing* 46 (1991), 165–173. DOI:http://dx.doi.org/10.1007/BF02239170

A.R. DiDonato and A.H. Morris. 1986. Computation of the incomplete gamma function ratios and their inverse. *ACM Trans. Math. Software* 12, 4 (1986), 377–393. DOI:http://dx.doi.org/10.1145/22721.23109

Free Software Foundation. The GCC Quad-Precision Math Library. Retrieved Dec 1, 2014 from http://gcc.gnu.org/onlinedocs/libquadmath/.

W. Gautschi. 1979. A computational procedure for incomplete gamma functions. *ACM Trans. Math. Software* 5, 4 (1979), 466–481. DOI:http://dx.doi.org/10.1145/355853.355863

A. Gil, J. Segura, and N.M. Temme. 2012. Efficient and accurate algorithms for the computation and inversion of the incomplete Gamma function ratios. *SIAM Journal on Scientific Computing* 34, 6 (2012), A2965–A2981.

D.T. Gillespie. 2007. Stochastic simulation of chemical kinetics. *Annual Reviews of Physical Chemistry* 58 (2007), 35–55. DOI:http://dx.doi.org/10.1146/annurev.physchem.58.032806.104637

P. Glasserman. 2004. *Monte Carlo Methods in Financial Engineering*. Springer, New York.

D. Gross, J.F. Shortle, J.R. Thompson, and C.M. Harris. 2008. *Fundamentals of queueing theory (fourth edition)*. John Wiley & Sons, New Jersey.

Intel. 2014. Intel Advanced Vector Extensions Intrinsics Guide. Retrieved Dec 1, 2014 from http://software.intel.com/sites/landingpage/IntrinsicsGuide.

NAG. 2014a. `nag_gamma_pdf_vector`, Numerical Algorithms Group, C Library, Mark 23. Retrieved Dec 1, 2014 from http://www.nag.co.uk/numeric/cl/nagdoc_cl23/html/G01/g01kkc.html.

NAG. 2014b. `nag_incomplete_gamma`, Numerical Algorithms Group, C Library, Mark 23. Retrieved Dec 1, 2014 from http://www.nag.co.uk/numeric/cl/nagdoc_cl23/html/S/s14bac.html.

NVIDIA. 2014. CUDA C Programming Guide, version 6.0. Retrieved Dec 1, 2014 from http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

N.M. Temme. 1979. The asymptotic expansion of the incomplete gamma functions. *SIAM Journal of Mathematical Analysis* 10, 4 (1979), 757–766. DOI:http://dx.doi.org/10.1137/0510071

N.M. Temme. 1987. On the computation of the incomplete gamma functions for large values of the parameters. In *Algorithms for Approximation*. Clarendon Press, New York, 479–489.

N.M. Temme. 1992. Asymptotic inversion of incomplete gamma functions. *Math. Comp.* 58, 198 (1992), 755–764. DOI:http://dx.doi.org/10.1090/S0025-5718-1992-1122079-8

N.M. Temme. 1994. A set of algorithms for the incomplete gamma functions. *Probability in the Engineering and Informational Sciences* 8, 2 (1994), 291–307. DOI:http://dx.doi.org/10.1017/S0269964800003417

M.J. Wichura. 1988. Algorithm AS 241: The percentage points of the normal distribution. *Applied Statistics* 37, 8 (1988), 477–484. DOI:http://dx.doi.org/10.2307/2347330

S. Winitzki. 2003. *Computing the incomplete gamma function to arbitrary precision*. Lecture Notes in Computer Science, Vol. 2667. Springer, 790–798. DOI:http://dx.doi.org/10.1007/3-540-44839-X_83