Anais da Academia Brasileira de Ciências 2019 91(4): e20180268
(Annals of the Brazilian Academy of Sciences)
Printed version ISSN 0001-3765 / Online version ISSN 1678-2690
http://dx.doi.org/10.1590/0001-3765201920180268
www.scielo.br/aabc | www.fb.com/aabcjournal

Practical rules for summing the series of the Tweedie probability density function with high-precision arithmetic

NELSON L. DIAS[1] and PAULO J. RIBEIRO JR[2]

[1]Departamento de Engenharia Ambiental, Universidade Federal do Paraná, Centro Politécnico, CP 19100, 81531-990 Jardim das Américas, Curitiba, PR, Brazil
[2]Departmento de Estatística, Universidade Federal do Paraná, Centro Politécnico, 81531-990 Jardim das Américas, Curitiba, PR, Brazil

Manuscript received on March 20, 2018; accepted for publication on December 4, 2018

How to cite: DIAS NL AND RIBEIRO JR PJ. 2019. Practical rules for summing the series of the Tweedie probability density function with high-precision arithmetic. An Acad Bras Cienc 91: e20180268. DOI 10.1590/0001-3765201920180268.

Abstract: For some ranges of its parameters and arguments, the series for Tweedie probability density functions are sometimes exceedingly difficult to sum numerically. Existing numerical implementations utilizing inversion techniques and properties of stable distributions can cope with these problems, but no single one is successful in all cases. In this work we investigate heuristically the nature of the problem, and show that it is not related to the order of summation of the terms. Using a variable involved in the analytical proof of convergence of the series, the critical parameter for numerical non-convergence ("alpha") is identified, and an heuristic criterion is developed to avoid numerical non-convergence for a reasonably large sub-interval of the latter. With these practical rules, simple summation algorithms provide sufficiently robust results for the calculation of the density function and its definite integrals. These implementations need to utilize high-precision arithmetic, and are programmed in the Python programming language. A thorough comparison with existing R functions allows the identification of cases when the latter fail, and provide further guidance to their use.

Key words: Python, R Tweedie package, Tweedie probability density, Tweedie series.

INTRODUCTION

The Tweedie distribution is a member of the family of exponential dispersion models, discussed for example in Jørgensen (1987) and Jørgensen (1992). The Tweedie is a distribution for which Taylor's 1961 empirical law relating the mean to the variance, *viz.*,

$$\mathrm{Var}[X] = a\,\mathrm{E}[X]^p, \tag{1}$$

holds.

Correspondence to: Nelson Luís Dias
E-mail: nelsonluisdias@gmail.com
ORCid: https://orcid.org/0000-0002-9770-8595

In this work, we are concerned with particular ranges of parameters of the Tweedie distribution, for which serious numerical difficulties arise when computing the probability density function (pdf) with form (*c.f.* Jørgensen (1992), Section 2.7)

$$f_Z(z;\vartheta,\alpha) = N(z;\alpha) \times \exp\left[z\vartheta - \varkappa(\vartheta,\alpha)\right],\ z \geq 0, \tag{2}$$

with

$$N(z;\alpha) = \frac{1}{\pi z} \times \sum_{k=1}^{\infty} \frac{\Gamma(1+\alpha k)}{\Gamma(1+k)} \left(\varkappa(-1/z,\alpha)\right)^k \sin(-k\pi\alpha), \tag{3}$$

and

$$\varkappa(\vartheta,\alpha) = \frac{\alpha-1}{\alpha}\left(\frac{\vartheta}{\alpha-1}\right)^{\alpha}. \tag{4}$$

A more general form of the Tweedie pdf is $f_X(x) = (1/\sigma)f_Z(x/\sigma)$, where σ is a scale parameter, for which

$$E[X] = \sigma\varkappa'(\vartheta),$$
$$\mathrm{Var}[X] = \sigma^2\varkappa''(\vartheta).$$

In this work, we set $\sigma = 1$ throughout, and use Eq. (2) without loss of generality.

Clearly, Eqs. (2)–(4) define a two-parameter distribution. The ranges of the parameters that we are interested in are $0 < \alpha < 1$ and $\vartheta < 0$. The exponent p in Eq. (1) and α are related by

$$\alpha = \frac{p-2}{p-1}. \tag{5}$$

The central problem here is that for certain values of z and α the series given by Eq. (3) is exceedingly difficult, if not impossible, to sum employing "standard" double-precision (64-bit) floating-point arithmetic. A head-on approach to sum the series fails in many cases (Dunn and Smyth 2005). To the best of our knowledge, currently the most successful approaches to sum Eq. (3) are either to employ Fourier inversion (Dunn and Smyth 2008) or to exploit the link between the stable distribution and the Tweedie distribution (Dunn 2015).

In spite of the aforementioned difficulties, it turns out that, for a fairly wide range of the variables α and z, accurate calculations are possible, at the price of using much higher precision and correspondingly much slower arithmetic. Unless stated otherwise, we use Python's (Python Software Foundation 2015) **mpmath** module (http://mpmath.org/) to perform all calculations with 1 000-bit precision for the mantissa (compare with the 53-bit precision of usual double-precision arithmetic). Being interpreted, Python is intrinsically much slower than compiled languages, and the use of **mpmath** makes our calculations even slower still. Yet for many practical purposes (for example, integrating $f_Z(z)$ with 1 000 points and Gaussian quadrature), our approach is entirely feasible on a desk computer with the rules devised here. It also has the advantage that the algorithms (and the underlying mathematics) are simple and straightforward, being therefore easy to implement and use.

In this work, we compare our computations of the Tweedie densities with the ones obtained using the **tweedie** package (Dunn 2015) for R (R Core Team 2015). There are four alternatives for computations using the **tweedie** package depending on the approach desired:

Summing the series: dtweedie.series (Dunn and Smyth 2005).

Fourier series inversion: dtweedie.inversion (Dunn and Smyth 2008).

Using a stable distribution: dtweedie.stable (Nolan 1997); it uses R's library **stabledist** (Wuertz et al. 2016).

Using a saddlepoint approximation to the density: dtweedie.saddle (Dunn and Smyth 2001).

A generic call to dtweedie uses either dtweedie.series or dtweedie.inversion depending on the parameter values whereas dtweedie.stable and dtweedie.saddle must be called explicitly. Note that in the present work we never use the dtweedie.saddle function.

In spite of the mathematically more sophisticated – and usually effective, as we will see – approaches of Dunn and Smyth (2008) and Nolan (1997), the nature of the problems involved in summing Eq. (3) remains not fully explored. Considerable insight as well as useful practical guidance for the calculation of Tweedie probability density functions can be gained by looking at them in more detail. Incidentally, this will also give us the opportunity to assess the implementations in the **tweedie** R package to calculate densities in some numerically challenging situations.

We have three objectives. First, we attempt to explain (empirically, by way of example) why it is in some cases so difficult to sum Eq. (3); often, it is in practice impossible to calculate it with double-precision (53-bit mantissa) arithmetic. As we shall see, the explanation involves difficulties in calculating individual terms of the series for large k when α is very close to 0 or to 1, and/or when z is small.

Then, we also show that some simple practical rules can be devised to avoid numerical errors or the need to sum too many terms in Eq. (3). It is fortunate that in many cases the numerical problems will arise for a range of z at which $f_Z(z)$ is exceedingly small. In such cases, instead of attempting to sum the series and keeping errors under control, it is much more effective to force the calculating function to return zero. As shown in Section **Results using Python**, this does not affect the accuracy of the calculation of definite integrals of $f_Z(z)$, and therefore of the probabilities. The criterion for returning zero in these cases is explained below in Section **Practical rules and tests** after Eq. (21).

Lastly, it will be possible to reassess the capabilities of the **tweedie** package, so that the practical rules devised here can also be useful for its users. The comparison with the high-precision approach using Python adopted in this work will be mutually beneficial for assessing both the **tweedie** R package and the **nptweedie.py** module developed for the present work.

In the next section we give a few examples of the numerical difficulties that arise in the summation of the series, without going into the details of how the terms are actually summed. This is left for Sections **Absolute convergence**, where we show that the series always converges (in an analytical sense), and **Numerical implementation**, where it is shown how a standard algorithm for the summation (with 1 000-bit precision) is enough in most cases. Then, with a "good enough" algorithm at hand, a practical rule is presented in Section **Practical rules and tests** to predict cases when, even with 1 000 bits, the sum will fail (in the numerical sense) to converge in less than 10 000 terms. In this section, a first comparison with R's **tweedie** package is also made.

As already mentioned, because failure of numerical convergence is associated with very small values of $f_Z(z)$, it is enough to return zero in such cases. This rule is then incorporated into a simple summation algorithm that is always successful (in this narrower sense) for $0.01 < \alpha < 0.99$. It is important to note that reliable computations of the pdf for the range of α considered here are important not only for problems

characterized by parameter values in that range, but also when running algorithms for statistical inference either based on optimization or sampling methods such as Monte Carlo Markov Chain and similar ones which may have excursions off of those regions of the parameter space.

In Section **Results using Python**, several tests on the proposed algorithm are performed: the results are tested against an analytical closed form for $f_Z(z)$ when $\vartheta = -1/2$, $\alpha = 1/2$ which corresponds to the inverse Gaussian distribution; and the accuracy of the integrals of $f_Z(z)$ is checked by numerical integration. We will show that the proposed algorithm always integrates to 1 (given enough integration points) within an error of 10^{-6} (at most). In this section we also go back to calculating Eq. (3) with double-precision, obtaining additional insight at where and why standard double-precision implementations fail. Many of the tests performed in Section **Results using Python** are then run again in R, using the **tweedie** package, in Section **Performance in comparison with R's tweedie package**: this provides an objective assessment of current R implementations for situations where straightforward summation is very difficult. Our conclusions are given in the final section.

TERMWISE BEHAVIOR AND THE DIFFICULTY OF COMPUTING N(Z;α)

Let

$$B = \left|\varkappa(-1/z, \alpha)\right|; \tag{6}$$

it is worth writing

$$b_k = \frac{\Gamma(1+\alpha k)}{\Gamma(1+k)} B^k, \tag{7}$$

$$c_k = (-1)^k b_k, \tag{8}$$

$$d_k = c_k \sin(-k\pi\alpha), \tag{9}$$

and the corresponding series

$$S_b(k) = \sum_{j=1}^{k} b_j, \tag{10}$$

$$S_c(k) = \sum_{j=1}^{k} c_j, \tag{11}$$

$$S_d(k) = \sum_{j=1}^{k} d_j. \tag{12}$$

While the series that we are ultimately interested in is $N(z, \alpha) = S_d(\infty)/(\pi z)$, considerable insight about the problems that plague the summation will be obtained by looking at the b_k terms. Using these auxiliary series will also help to establish analytically the convergence of $S_d(k)$ in the next section.

An upper bound for $\max_k b_k$ can be found as follows (Dunn and Smyth 2005):

$$k_{max} = \frac{z^{2-p}}{p-2}, \tag{13}$$

$$b_{k_{max}} = \exp\left[(1-\alpha)k_{max} + \frac{1}{2}\ln(\alpha)\right]. \tag{14}$$

In the following, we show all plots normalized by the value of $b_{k_{max}}$.

We now discuss the problems involved with the summation of Eq. (3) by means of 3 examples. In this section, we do not correct those problems yet, and the discussion that follows has the sole objective of shedding light on the nature of the numerical problems discovered by Dunn and Smyth (2005). We proceed to sum the series almost straightforwardly, except that we use 1 000-bit precision arithmetic, and avoid loss of precision due to large variations in the order of magnitude of the terms in the sum. The numerical convergence criterion used is

$$\varepsilon = \frac{b_k}{|S_d(k)|} < 10^{-10}. \tag{15}$$

Figures 1(a) - 1(c) show the behavior of b_k, $S_d(k)$, and ε. The parameters chosen for this example are $\alpha = +1/2$, $\vartheta = -1/2$, because then the Tweedie coincides with an inverse Gaussian distribution, the only case in the range $0 < \alpha < 1$ for which an alternative analytical expression for $f_Z(z)$ is available. Figures 1(a) - 1(c) correspond to the cases (a) $z = 0.0006$, (b) $z = 0.0008$ and (c) $z = 0.0020$.

The first striking feature in the figures is the range covered by the b_k's (shown as a solid line): in excess of 10^{300} in the first two cases, and 2.82×10^{216} in the last one. Indeed, the range is so wide that the plotting program used to produce the figures, which itself uses double-precision arithmetic, is unable to plot all the values calculated, thereby forcing the graphs to be clipped at the lower 10^{-300} end.

To prevent that the summation of terms varying across this very wide range of orders of magnitude suffer from the "catastrophic loss of precision" mentioned by Malcolm (1971), we implemented an adaptation of the **msum** function found in http://code.activestate.com/recipes/393090/ (Shewchuck 1996), which is designed to deal with this kind of problem. The adaptation, in the form of 3 functions in the module **nsum** (written by us), is presented in the Appendix. Therefore, in principle, the sum in Eq. (12) is being carried out without loss of precision. Yet, in cases (a) and (b), correct numerical convergence was not achieved for $k < 10\,000$ (although this is not readily apparent in the figures themselves): later on, we will find more evidence that loss of precision is not the critical issue with summing Eq. (3). Rather, the lack of convergence to the right value can happen due to the following reasons:

1. Although the sum itself is carried out without loss of precision, apparently inaccurate evaluation of the terms d_k for large values of k leads the algorithm to "converge" to a wrong value.

2. The algorithm reaches the stopping criterion $k = 10\,000$ without achieving the convergence criterion specified in Eq. (15).

Case (a) is perhaps the worst of all: the algorithm's convergence criterion given by Eq. (15) is satisfied, but the value obtained for the density is wrong and positive (reason 1 above). In this particular case, because we have the alternative to calculate the density with the inverse Gaussian formula, this is easily verifiable; for $\alpha \neq 1/2$, however, there seems to be no way to check within the algorithm itself if reason 1 is happening. In the case of Figure 1(a), the algorithm (employing 1 000 bits of precision!) calculated a density of $2.735443272 \times 10^{+68}$, whereas the correct value given by the inverse Gaussian formula is $9.03190022 \times 10^{-358}$.

Case (b)'s failure is another instance of reason 1: here, the algorithm found a negative value of S_d. This case is somewhat easier to handle, because the user can verify the implausibility of a negative density with a simple **if** clause. Note that the negative values of $S_d(k)/b_{k_{max}}$ to which the algorithm converges do not appear in the log-log plot.

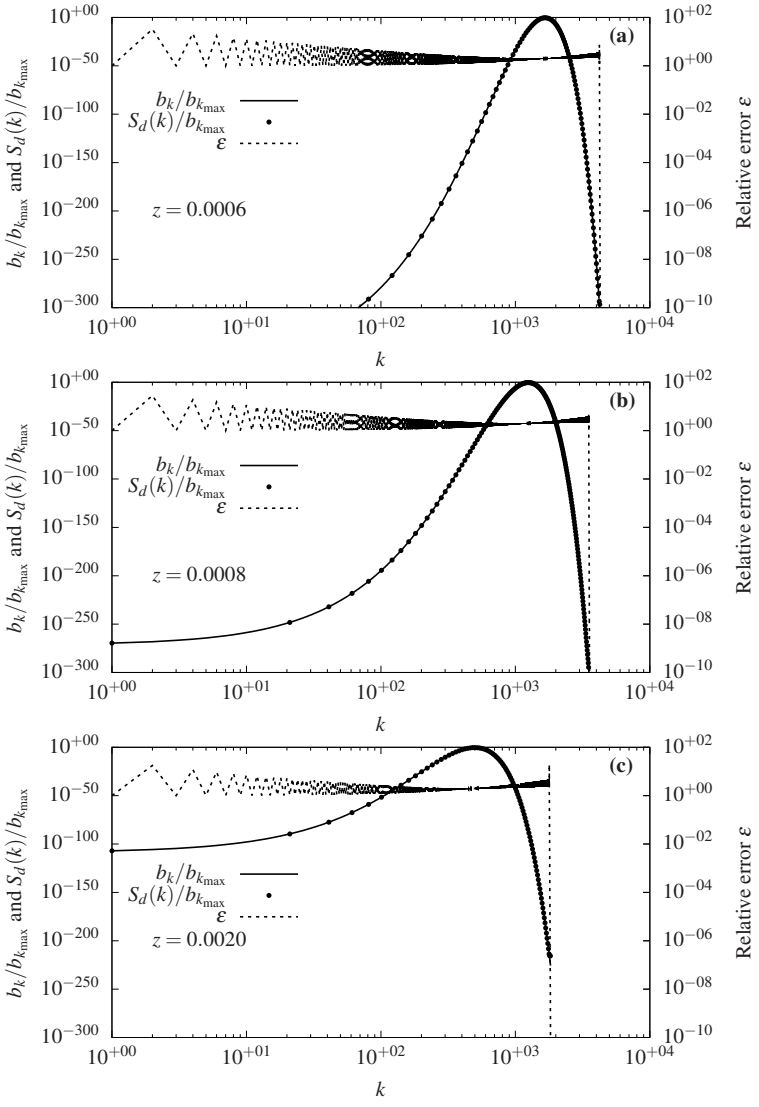

Figure 1 - Individual terms $b_k/b_{k_{max}}$ (solid line), sum $S_d(k)/b_{k_{max}}$ of the $N(z;\alpha)$ series (filled circles), and relative error ε (dashed line) with $\alpha = -1/2$, $\vartheta = -1/2$. **(a)** z = 0.0006, **(b)** z = 0.0008 and **(c)** z = 0.0020. Only **(c)** converged to the correct value with less than 10 000 sums.

In short, neither case (a) nor case (b) failed to converge because the stopping criterion k = 10 000 was reached: they either converged to a wrong positive value (a) or to a wrong negative value (b), in spite of being calculated with 1 000 bits of precision and the use of a summation algorithm specifically designed to avoid loss of precision.

Finally, case (c) is successful. The range of the b_k terms is somewhat smaller (albeit still daunting). Convergence to the right value of $3.2329931463 \times 10^{-105}$, confirmed by the inverse Gaussian formula, is only achieved at k = 1826.

In all cases, the relative error (shown as a dashed line, with the corresponding axis scale on the right) takes a long time to reach the convergence criterion $\varepsilon < 10^{-10}$. Close to spurious (cases (a) and (b)) or

correct (case (c)) convergence, ε plunges down abruptly, there being no qualitative difference between the three cases that could help to devise an identification pattern within the algorithm for spurious convergence.

Finally, Figure 1 shows a last disconcerting feature of Eq. (3): because of the alternating character of Eq. (11), compounded by the $\sin(-k\pi\alpha)$ term in Eq. (12), successive values of d_k tend to almost cancel out each other (in order of magnitude), the residue being carried over to the next term. For this reason, the order of magnitude of $S_d(k)$ (given by the points in the figure) and of b_k (and therefore of c_k and d_k as well) remains the same for a very long time, while the magnitude of b_k varies non-monotonically with k over a very wide range of values (the reader should be aware that the points and the solid line in Figure 1 are **not** exactly equal, the visual impression being an artifact of the very wide range in the plot).

All that adds to the difficulty of summing the series, casting serious doubts on the feasibility of a direct attack, as found out by Dunn and Smyth (2005) and Dunn and Smyth (2008).

If one accepts the use of very large floating-point precision as a useful, and perhaps unavoidable, tool, however, there is hope in practical situations, although definitely not for all possible combinations of z and α. In the next section, we discuss the analytical convergence of the series. In Section **Numerical implementation**, we give the details of how the series summations, in this section, and elsewhere in the manuscript, were implemented. In Section **Practical rules and tests**, we give a practical procedure to sum the series using 1 000-bit precision that avoids the errors observed, for example, in cases (a) and (b) of this section. The procedure is able to predict when the density $f_Z(z)$ will be very close to zero, and in this case will return zero instead of attempting the sum. Unavoidably, this results in a relative error of 100% for the null estimate of $f_Z(z)$, but has no practical consequences insofar as integration of the density and calculation of probabilities is concerned.

ABSOLUTE CONVERGENCE

The series $S_d(k)$ is absolutely convergent. The proof is based on two theorems:

1. If a series is absolutely convergent, then it is convergent (Dettman (1984), Theorem 4.3.4). We shall show that $S_b(k)$ converges. Therefore, $S_c(k)$ is absolutely convergent ($b_k = |c_k|$), and $S_c(k)$ converges.

2. If a series is bounded term-by-term by a convergent series of positive terms, the series is absolutely convergent (Dettman (1984), Theorem 4.3.12). Since $|d_k| \leq b_k$ and the latter is convergent, $S_d(k)$ is absolutely convergent, and therefore it is convergent.

To prove that $S_b(k)$ is convergent, we use the ratio test:

$$b_k = B^k \frac{\Gamma(1+\alpha k)}{k!} \Rightarrow$$
$$\frac{b_{k+1}}{b_k} = \frac{B^{k+1}}{(k+1)!}\frac{k!}{B^k}\frac{\Gamma(1+\alpha(k+1))}{\Gamma(1+\alpha k)}$$
$$= \frac{B}{k+1}\frac{\Gamma(1+\alpha(k+1))}{\Gamma(1+\alpha k)}.$$

Using Stirling's approximation for large x,

$$\Gamma(x+1) \sim \sqrt{2\pi x}\left(\frac{x}{e}\right)^x,$$

we get

$$\lim_{k\to\infty}\frac{b_{k+1}}{b_k} = \lim_{k\to\infty}\frac{B}{k+1}\frac{\sqrt{2\pi(\alpha(k+1))}}{\sqrt{2\pi\alpha k}}\frac{\left(\frac{\alpha(k+1)}{e}\right)^{\alpha(k+1)}}{\left(\frac{\alpha k}{e}\right)^{\alpha k}}$$
$$= \lim_{k\to\infty}\frac{B}{k+1}e^{-\alpha}\sqrt{\frac{\alpha k+\alpha}{\alpha k}}\times\frac{[\alpha(k+1)]^{\alpha(k+1)}}{(\alpha k)^{\alpha k}}.$$

Note that the square-root term has limit 1 as $k\to\infty$. Also,

$$\frac{[\alpha k+\alpha]^{\alpha k+\alpha}}{[\alpha k]^{\alpha k}} = \left[\frac{\alpha k+\alpha}{\alpha k}\right]^{\alpha k}[\alpha k+\alpha]^{\alpha}$$
$$= \underbrace{\left[1+\frac{1}{k}\right]^{\alpha k}}_{\to e^{\alpha}}[\alpha k+\alpha]^{\alpha}$$

We obtain, finally:

$$\lim_{k\to\infty}\frac{b_{k+1}}{b_k} = \frac{B}{k+1}e^{-\alpha}e^{\alpha}\alpha^{\alpha}(k+1)^{\alpha} = B\alpha^{\alpha}(k+1)^{\alpha-1}\to 0, \tag{16}$$

for $0<\alpha<1$. This means that for any α in this range, and for any finite B, the series $S_b(k)$ is convergent. From the two theorems above, therefore, both $S_c(k)$ and $S_d(k)$ are absolutely convergent.

NUMERICAL IMPLEMENTATION

We use a very simple scheme to calculate $S_d(k)$. The b_k terms are calculated with

$$b_k = \exp\left[\ln\left[\frac{\Gamma(1+\alpha k)}{\Gamma(1+k)}B^k\right]\right]$$
$$= \exp\left[\ln\Gamma(1+\alpha k)+k\ln(B)-\ln\Gamma(1+k)\right] \tag{17}$$

to take advantage of the function loggamma in **mpmath**. Then c_k and d_k are calculated straightforwardly with Eqs. (8)–(9). As mentioned above, the sum of the d_k terms is performed with the help of a modification of the publicly available function **msum**: see module **nsum** in the Appendix.

The summation stops when the criterion given in Eq. (15) is reached. If $S_d(k)$ is found to be negative, k is incremented and the algorithm continues. Usually, this forces the algorithm to reach k = 10 000, at which point it flags non-convergence.

PRACTICAL RULES AND TESTS

As mentioned in the Introduction, the algorithm as is will often fail to converge to the right numerical value with less than 10 000 terms in the sum. In order to make it more robust, and to curtail unnecessary calculations when $f_Z(z)$ is actually very small, we observe that B defined in Eq. (6) is a good predictor of the "difficulty" of the summation. This observation is found in the caption of Table 1 of Dunn and Smyth (2008) about a variable they call ξ, which is given by (in our notation, and setting $\sigma = 1$)

$$\xi = \frac{1}{z^{2-p}}. \tag{18}$$

For comparison, our

$$B = \left| \frac{1}{2-p} \left(\frac{p-1}{z} \right)^{\frac{2-p}{1-p}} \right| \tag{19}$$

is not equal to ξ, but it is related to it.

The role of B in the rate of convergence is seen in Eq. (16). For a specified (desired) rate of convergence η after n terms, one has

$$\begin{aligned} B\alpha^{\alpha}(n+1)^{\alpha-1} &< \eta, \\ \frac{1}{B} > \Phi(\alpha) &\equiv \frac{\alpha^{\alpha}(n+1)^{\alpha-1}}{\eta}. \end{aligned} \tag{20}$$

If n and η are fixed, Eq. (20) provides a practical criterion to predict that the algorithm will be successful: this will happen whenever 1/B is larger than $\Phi(\alpha)$. This of course may be over-optimistic: B is also a function of z, and there is no *a priori* guarantee that n and, most importantly, η, are independent of z and of α.

On the bright side, Eq. (20) is independent of ϑ: only α appears in it. The idea therefore is to vary α over its allowed range of $0 < \alpha < 1$, and for each α to vary z from below until the algorithm starts to converge to the "correct" value of $f_Z(z)$. For a single z it may be difficult to identify if the convergence is to the true value, but a synoptic view of several values of z in a graph is quite enough to identify the correct values.

For a fixed α, therefore, it is possible to determine, by trial and error, the values of z and B at which the algorithm converges numerically with less than 10 000 terms.

To pursue this idea, we (manually) searched for the z-range where the algorithm converges using the values $\alpha = 0.01$, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 0.99. For $\alpha < 0.01$ or $\alpha > 0.99$, manually trying to identify a range of z's where the algorithm converges proved unsuccessful. We consider that the interval $\alpha \in [0.01, 0.99]$ is wide enough in practice; therefore, if α falls outside of it, we simply do not attempt to sum the series. The value of $\vartheta = -1/2$ (the same as for the inverse Gaussian) was used.

The graphs obtained with 6 of the 11 values of α above are shown in Figure 2. Each sub-figure shows the calculated densities $f_Z(z)$ (with 1 000-bit precision) as well as 1/B for a range of z values around which the algorithm starts to be successful. Non-convergence in 10 000 iterations (or convergence to a negative value) is flagged as 10^{+150}. Also shown is the value of the density calculated by the **tweedie** package with the "generic" call to dtweedie (see the comment in the introduction about how it chooses which method to employ). In contrast to the 1 000-bit calculation of $f_Z(z)$ by series summation, usually dtweedie does not suffer from spurious values at the left tail of the distribution, but for $\alpha = 0.9$ it clearly does not return correct values there, and in fact high-precision series summation fares a little better (Figure 2, lower right panel). We observed the same problem for our maximum "acceptable" α, 0.99 (not shown). We did not try to determine a more precise value of α, above 0.8, where dtweedie starts to give incorrect values at the left tail.

Wrong convergence to positive values simply shows as discernible spurious values. Similar behavior is observed for the other values of α mentioned above. Note the monotonically increasing 1/B in all cases.

In Figure 2, the very small values of $f_Z(z)$ for the first successful value of z are noteworthy. Table I lists α, z, $f_Z(z)$ and 1/B for the first successful instance of the algorithm (subjectively decided) in each case.

With the values in Table I, it is possible to plot 1/B *versus* α, in Figure 3. We can now draw a smooth curve through the data points. This can be done by adjusting a suitable equation using (in our case) the available non-linear Levenberg-Marquardt least squares procedure built-in in the plotting program (Gnuplot:

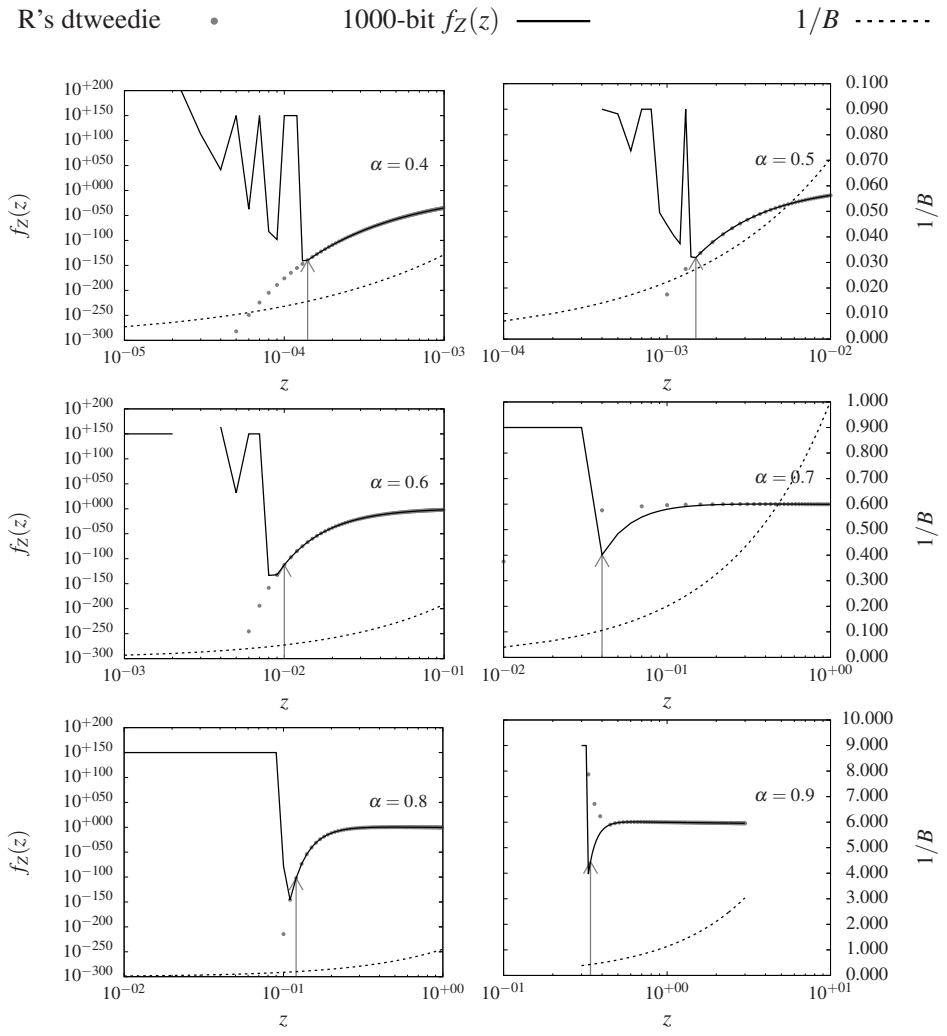

Figure 2 - Transition from unsuccessful to successful calculation of Tweedie densities, for $\alpha = 0.4$, 0.5, 0.6, 0.7, 0.8 and 0.9. On the left axis scale, the solid line shows the density calculated from summing the series with 1 000-bit precision, and the grey circles are the densities calculated by dtweedie in R. On the right axis scale, the dashed line is 1/B. The arrows indicate the first point at which the series summation is considered "successful". In all cases, $\vartheta = -1/2$.

www.gnuplot.info). We used a weighted version, giving weight 1 to all the ten lowest values of α, and weight 100 to the largest (0.99). This is necessary because, as α gets closer to 1, the pdf becomes more and more pointed close to zero, and small changes in z and in 1/B make all the difference between numerical non-convergence and convergence. Thus, an accurate value of the minimal 1/B is much more critical for $\alpha = 0.99$.

Our first try was to use Eq. (20) and to adjust n and η as parameters of the least-squares fit. Although Eq. (20) is (encouragingly) able to capture the overall α-dependency, the result, shown in Figure 3 as a dashed line, is relatively poor. Therefore, we changed to a purely empirical equation with 3 instead of 2 adjustable paramters, *viz.*

$$\Psi(\alpha) = \exp\left[a + b\alpha^c\right]. \tag{21}$$

TABLE I
Early values, in the direction of increasing z, of α, z, f_Z (z) and 1/B for which the Tweedie densities are calculated correctly.

α	*z*	$f_Z(z)$	*1/B*
0.01	1.00000000e-50	2.32936769e-44	0.00319390
0.10	2.40000000e-15	3.64498662e-130	0.00379492
0.20	1.10000000e-08	2.71264953e-126	0.00612120
0.30	4.00000000e-06	1.26481645e-139	0.00925062
0.40	1.40000000e-04	2.37723364e-140	0.01561782
0.50	1.50000000e-03	3.20563918e-141	0.02738613
0.60	1.00000000e-02	4.14874551e-113	0.05461693
0.70	4.00000000e-02	6.96585298e-100	0.10553674
0.80	1.20000000e-01	6.97320117e-103	0.20240861
0.90	3.40000000e-01	6.82753609e-78	0.42911484
0.99	8.82330000e-01	6.05303014e-09	0.91581959

This is shown as a solid line in Figure 3. Not surprisingly, the fit is better. The adjusted parameters are directly coded in module **Psialpha.py**, given in the Appendix. It is very simple, and just defines the values of a, b and c in Eq. (21) above.

With Ψ(α) thus obtained, there are only two changes that need to be made to the algorithm of calculating Tweedie densities:

1. Do not allow $\alpha < 0.01$ or $\alpha > 0.99$; in these cases we do not attempt to sum the series, and the calculating function aborts. As discussed above, the range [0.01, 0.99] is probably wide enough for most applications.

2. If $1/B < \Psi(\alpha)$, return 0; otherwise, sum using 1 000-bit precision.

The full implementation is given in the Appendix. The probability density function is called pdfz_tweedie. It is designed to receive and to return standard floating-point variables: all the multiple-precision work is done internally and kept hidden from the user. In this way, he or she can use the ordinary Python **float** type transparently while using the module **nptweedie**.

RESULTS USING PYTHON

A first comparison is to check the proposed algorithm against the densities calculated with the inverse Gaussian distribution. This is only possible for $\alpha = 1/2$, $\vartheta = -1/2$, and is done visually in Figure 4. The agreement is actually very good: for 1 000 points evenly distributed between 0 and 20 the two methods agree exactly up to the eighth decimal place.

For cases when an analytical benchmark is not available, a good alternative is to check (numerically) how close the integral $I = \int_0^\infty f_Z(z)\,dz$ is to 1.

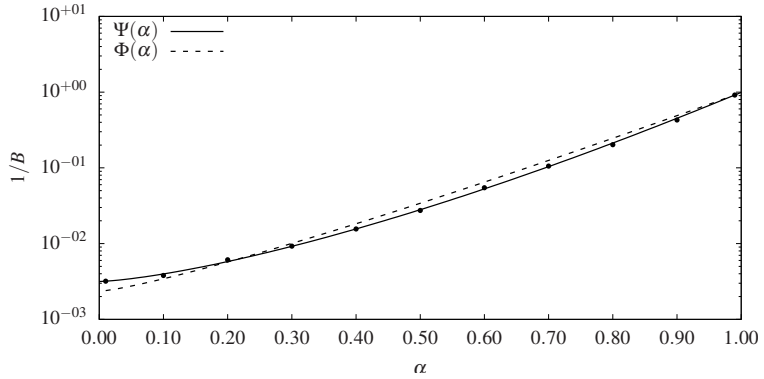

Figure 3 - Early values of 1/B for which the Tweedie densities are calculated correctly, in the direction of increasing z, for each α, and adjusted curves Φ(α) (dashed line) and Ψ(α) (solid line).

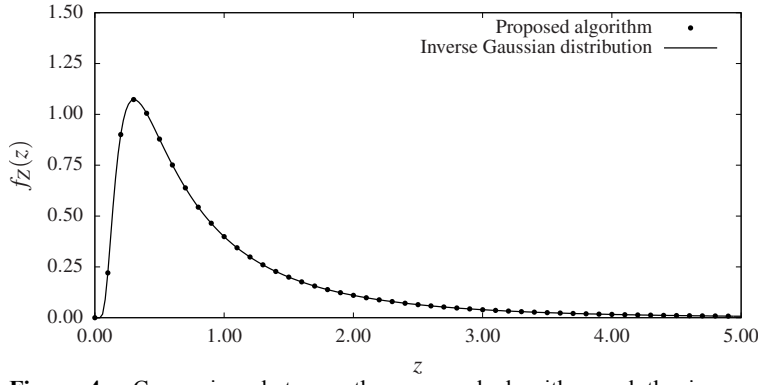

Figure 4 - Comparison between the proposed algorithm and the inverse Gaussian distribution, using $\alpha = 1/2$, $\vartheta = -1/2$.

At this point, also, one raises the question of how much the 1 000-bit precision in use is overkill: to what extent would double-precision suffice? Moreover, how important is it to have a sophisticated handling of possible loss of precision in the summation?

In order to answer these questions, we re-implemented the pdfz_tweedie function using double-precision only, while keeping most of the remainder of the algorithm intact. Two versions of the pure double-precision version were implemented: the first version uses the **nsum** module and does the sum without loss of precision ("no loss"); it is called here "dp-nl"; the other version does not use the **nsum** package, and performs all the sums with standard arithmetic only ("with loss") ; it is called here "dp-wl". For easy reference, we call the 1 000-bit precision version "np-nl". Note that "dp-nl", "dp-wl" and "np-nl" are the mnemonic names of the versions for summing the series, and not actual function names.

We now show the results of integrating the density numerically for these three versions, for all values of α considered above, keeping $\vartheta = -1/2$ fixed. Except for $\alpha = 0.99$, all numerical integrals were calculated for a range $z \in [1.0 \times 10^{-6}, 50]$ with 1 000 points using Gauss-Legendre quadrature (we used a translation to Python of the routine gauleg in Numerical Recipes (Press et al. 1992). For $\alpha = 0.99$, the integration was for $z \in [1.0 \times 10^{-6}, 20]$ with 10 000 points. The adjustment of range and number of points used is relatively easy to do, and somewhat unavoidable.

Results are shown in Table II: for each α, the rows show the number of points used in the quadrature, and the value of the numerical integral I and its absolute difference from 1, δI, for each of "np-nl", "dp-nl" and "dp-wl".

Clearly, with 1 000-bit precision, we are able to calculate very accurate integrals and, consequently, probabilities, for all values of α in the range.

The double precision versions, on the other hand, are at first sight disappointing. Except for $\alpha = 0.5$ and $\alpha = 0.99$, these versions are unable to produce accurate values of I. On the other hand, they fail or suceed simultaneously and produce essentially the same results. This is an indication that "catastrophic loss of precision" is not an issue, because dp-nl avoids loss of precision, while dp-wl does not. In hindsight, this is probably because the order of magnitude of the b_k terms varies slowly with k, so that each new term added is close, in order of magnitude, to the previous one and to the partial sum as well (*c.f.* Figure 1).

TABLE II

Numerical integration of Tweedie densities using Gaussian quadrature for versions np-nl, dp-nl and dp-wl of bit precision / summation method: $I = \int f_Z(z)\,dz$ (the integral calculated with each version) and $\delta I = I - 1$, with $\vartheta = -1/2$.

		np-nl		dp-nl		dp-wl	
α	points	*I*	δ	*I*	δ	*I*	δ
0.01	1000	1.0000e+00	7.1056e-07	2.4029e+69	2.4029e+69	2.4043e+69	2.4043e+69
0.10	1000	1.0000e+00	2.0786e-08	1.0141e+00	1.4119e-02	1.0140e+00	1.4009e-02
0.20	1000	1.0000e+00	2.9919e-10	2.1073e+02	2.0973e+02	2.1099e+02	2.0999e+02
0.30	1000	1.0000e+00	1.5745e-10	4.5974e+29	4.5974e+29	4.5929e+29	4.5929e+29
0.40	1000	1.0000e+00	9.3596e-11	1.1959e+62	1.1959e+62	1.1958e+62	1.1958e+62
0.50	1000	1.0000e+00	6.3911e-11	1.0000e+00	1.6025e-08	1.0000e+00	1.6028e-08
0.60	1000	1.0000e+00	4.5433e-11	9.8963e+26	9.8963e+26	9.8884e+26	9.8884e+26
0.70	1000	1.0000e+00	3.2674e-11	3.2814e+28	3.2814e+28	3.2770e+28	3.2770e+28
0.80	1000	1.0000e+00	2.2272e-11	1.3940e+43	1.3940e+43	1.3941e+43	1.3941e+43
0.90	1000	1.0000e+00	3.1027e-11	7.9491e+24	7.9491e+24	7.9629e+24	7.9629e+24
0.99	10000	1.0000e+00	3.7514e-09	1.0000e+00	1.8141e-08	1.0000e+00	1.7466e-08

Moreover, in practice the double precision versions fare better than suggested by Table II. This can be verified in Figure 5, where we show the pointwise numerical convergence (or not) of the double precision version "dp-nl" (gray dots) against the 1 000-bit precision version "np-nl" (solid black lines). The behavior of "dp-wl" is essentially the same as that of "dp-nl", and is not shown. In the figure, non-convergence of the sum for 10 000 terms (or spurious convergence for values greater than 10) is flagged as $f_Z(z) = 10$ for easy identification. Negative spurious values are simply not shown on the log-log plot.

One can see that the same problems that afflicted the 1 000-bit version are present, except that with more gravity, in the double-precision version. Thus, for small values of z, the simple algorithm used here fails, except that these z values are now greater than they were with the 1 000-bit version. Not surprisingly,

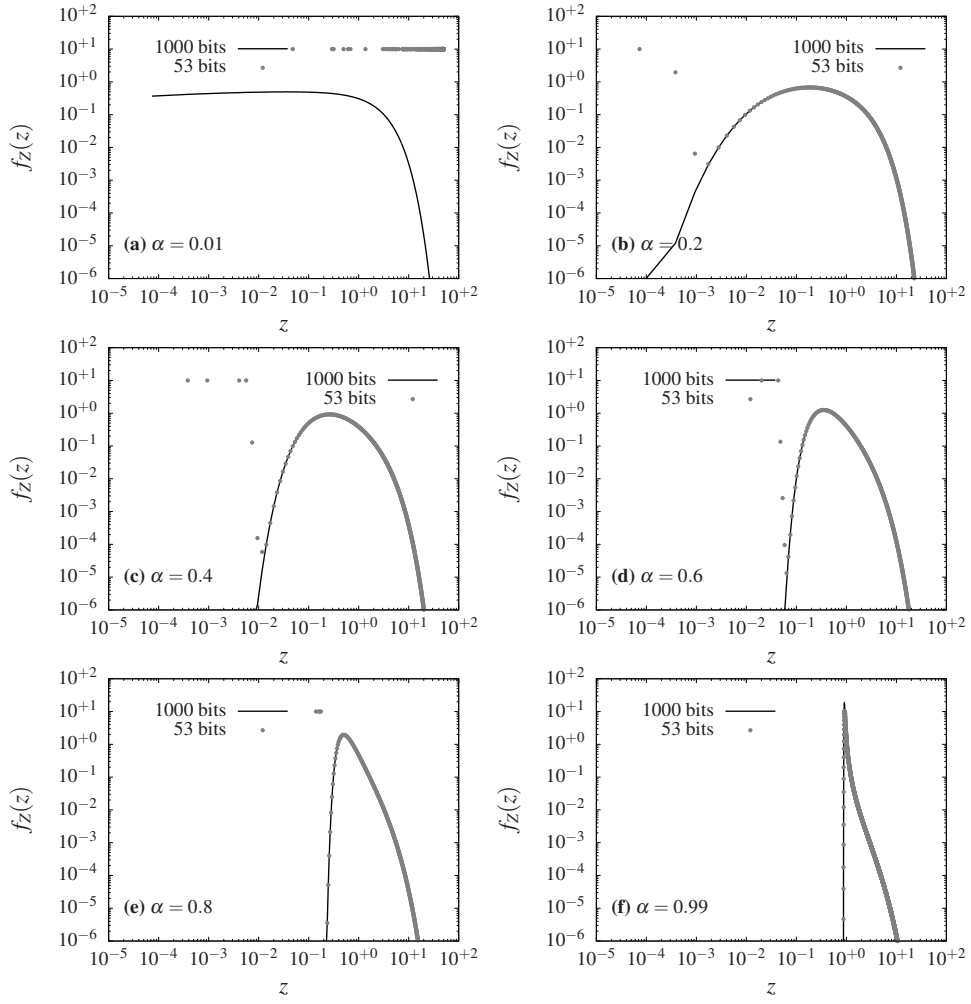

Figure 5 - Detailed verification of the convergence behavior of versions "np-nl" (continuous black line), and "dp-nl" (gray dots, with "dp-wl" being the same as "dp-nl"): **(a)**, $\alpha = 0.01$; **(b)**, $\alpha = 0.2$; **(c)**, $\alpha = 0.4$; **(d)**, $\alpha = 0.64$; **(e)**, $\alpha = 0.8$ and **(f)**, $\alpha = 0.99$.

the filter of Eq. (21), which was designed and calibrated with the 1 000-bit version, "lets through" a few points on the left tail of the distribution that lead to non-convergence.

Of course, one can re-calibrate the parameters in Eq. (21) to censor those points, but especially for the smaller α's, this may result in "chopping off" two much of the left tail (in the case $\alpha = 0.01$, the whole pdf is lost with double precision).

PERFORMANCE IN COMPARISON WITH R'S TWEEDIE PACKAGE

We now repeat the analysis in the previous section except that, instead of running np-nl against its double-precision versions, we run it against two options available in R: dtweedie, and dtweedie.stable. The results are shown in Table III. In this table, columns 3 and 4 are repeated from Table II, for ease of comparison. dtweedie.stable now fails for $\alpha = 0.01$ (it was not tried to check at exactly what value of $\alpha < 0.1$

it fails for the first time), whereas, at the other extreme, dtweedie fails for $\alpha = 0.9$ and $\alpha = 0.99$ (it was not tried to check at exactly what value of $\alpha > 0.8$ it fails for the first time).

TABLE III
Numerical integration of Tweedie densities using Gaussian quadrature for np-nl (1000- bit precision), dtweedie and dtweedie.stable: $I = \int f_Z(z)\,dz$ **the integral calculated with each version) and** $\delta I = I - 1$**, with** $\vartheta = -1/2$**.**

		np-nl		dtweedie		dtweedie.stable	
α	points	*I*	δ	*I*	δ	*I*	δ
0.01	1000	1.0000e+00	7.1061e-07	1.0000e+00	7.1058e-07	4.9356e-16	1.0000e+00
0.10	1000	1.0000e+00	2.0667e-08	1.0000e+00	2.0938e-08	1.0000e+00	3.7887e-08
0.20	1000	1.0000e+00	4.5261e-10	1.0000e+00	5.3930e-10	1.0000e+00	5.0887e-10
0.30	1000	1.0000e+00	1.3885e-10	1.0000e+00	4.2949e-10	1.0000e+00	4.0474e-10
0.40	1000	1.0000e+00	3.7200e-11	1.0000e+00	4.0054e-10	1.0000e+00	3.7758e-10
0.50	1000	1.0000e+00	5.2246e-11	1.0000e+00	3.8441e-10	1.0000e+00	3.8441e-10
0.60	1000	1.0000e+00	9.8298e-11	1.0000e+00	4.3890e-10	1.0000e+00	4.0424e-10
0.70	1000	1.0000e+00	1.4200e-10	1.0000e+00	4.2062e-10	1.0000e+00	3.8182e-10
0.80	1000	1.0000e+00	3.6224e-11	1.0000e+00	5.1426e-10	1.0000e+00	3.1015e-10
0.90	1000	1.0000e+00	1.5194e-10	0.0000e+00	1.0000e+00	1.0000e+00	3.4215e-11
0.99	10000	1.0000e+00	3.7498e-09	0.0000e+00	1.0000e+00	1.0000e+00	2.0088e-08

The same parameters of Gaussian quadrature previously employed to generate Table II were used. Figure 6 shows the corresponding densities for six cases: in the figure, failures are flagged at constant values equal to 1.0×10^{-6}.

There is always at least one function in package **tweedie** that performs as well, in practice, as our high-precision implementation in Python, but the choice must be made by the user. In its interval of validity, $0.01 < \alpha < 0.99$, **nptweedie** always succeeds and calculates the integral of the density with an accuracy better than 1×10^{-6}.

CONCLUSIONS

In this work we have investigated empirically a few of the problems that afflict the calculation of Tweedie probability density functions for the range $0 < \alpha < 1$ of one of its parameters. The α parameter is identified as the one whose values affect the summing of the corresponding series, the ϑ parameter playing no role in the problem. We have also shown that the series converges in the analytical sense, in spite of the occurrence of severe numerical problems for its summation for some values of z and α.

The terms in the series span a very large range of orders of magnitude, close to that allowed by double precision. This wide range of values, however, turns out not to lead to catastrophic loss of precision, likely because the order of magnitude varies slowly in the summation (nearby terms have close to the same orders of magnitude).

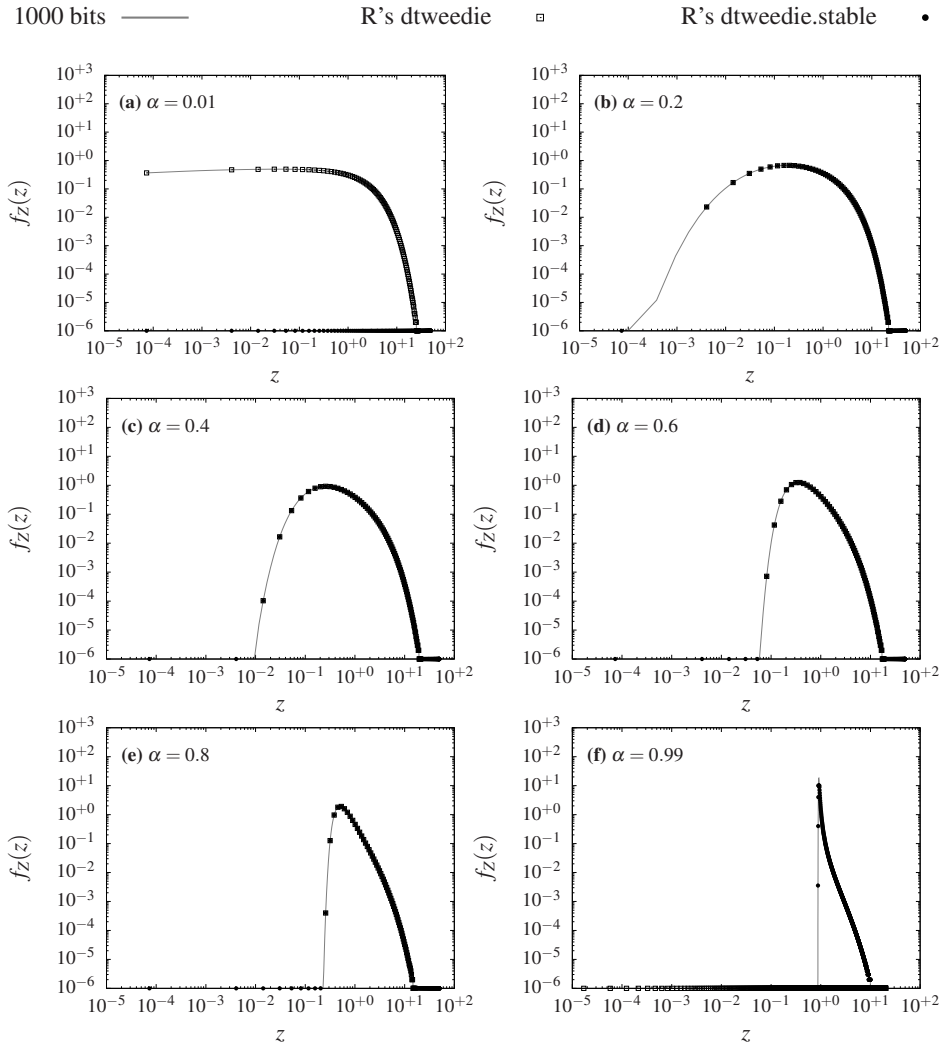

Figure 6 - Detailed verification of the convergence behavior of 1 000-bit densities "np-nl" (continuous gray line), in comparison with dtweedie (open squares) and dtweedie.stable (filled circles): (**a**), $\alpha = 0.01$; (**b**), $\alpha = 0.2$; (**c**), $\alpha = 0.4$; (**d**), $\alpha = 0.64$; (**e**), $\alpha = 0.8$ and (**f**), $\alpha = 0.99$. In **(a)**, dtweedie.stable values lower than 1.0×10^{-6} are shown at this value. In **(e)**, dtweedie values lower than 1.0×10^{-6} are shown at this value.

Still, for very large indices k in the summation, the calculation of the individual term remains inaccurate, leading in some cases to either the inability of reaching a stable value even after 10 000 terms, or to convergence to spurious values. A key parameter that allows to predict this event, here called 1/B, was identified and used to construct an *ad-hoc* procedure to avoid it.

We have visually identified the thresholds in z and the corresponding 1/B values as a function of α. For a practical range of $\alpha \in [0.01, 0.99]$, this allows the Tweedie pdfs to be calculated successfully using 1 000-bit precision.

Attempts to use only double precision fail to various degrees, with the worst cases being α close to the lower limit above (and vice-versa). In specific cases, the procedures adopted here may be adapted for double precision, if one is willing to accept a shorter range for α than $[0.01, 0.99]$. One of the three functions available

in R's **tweedie** package that we tested is always able to deal with the numerical problems satisfactorily, but the specific function varies, and its name may need to be enforced by the user: the results presented here may be useful in this regard.

ACKNOWLEDGMENTS

The authors wish to thank the comments and suggestions by two anonymous reviewers, which contributed substantially to improve the manuscript.

AUTHOR CONTRIBUTIONS

N.L. Dias suggested the use of high-precision arithmetic to sum the Tweedie series, derived the analytical results and wrote the Python code. P.J. Ribeiro Jr. oversaw the writing of the R codes needed to use the **tweedie** R package.

REFERENCES

DETTMAN JW. 1984. Applied Complex Variables. New York: Dover Publications.

DUNN PK. 2015. Package 'tweedie' (Tweedie exponential family models), version 2.2.1. URL http://www.r-project.org/package=tweedie.

DUNN PK AND SMYTH GK. 2001. Tweedie family densities: methods of evaluation. Odense, Denmark: In Proceedings of the 16th International Workshop on Statistical Modelling.

DUNN PK AND SMYTH GK. 2005. Series evaluation of tweedie exponential dispersion model densities. Stat Comp 15(4): 267-280.

DUNN PK AND SMYTH GK. 2008. Evaluation of tweedie exponential dispersion model densities by Fourier inversion. Stat Comp 18(1): 73-86.

JØRGENSEN B. 1987. Exponential dispersion models. J Roy Stat Soc B Met 49(2): 127-162.

JØRGENSEN B. 1992. The theory of exponential dispersion models and analysis of deviance. 2nd edition. Rio de Janeiro: Instituto de Matemática Pura e Aplicada.

MALCOLM MA. 1971. On accurate floating-point summation. Communape ACMape 14(11): 731-736.

NOLAN JP. 1997. Numerical calculation of stable densities and distribution functions. Comm Statist Stochastic Models 13(4): 759–774.

PRESS WH, TEUKOLSKY SA, VETTERLING WT AND FLANNERY BP. 1992. Numerical Recipes in C; The Art of Scientific Computing. 2nd edition. New York, NY, USA: Cambridge University Press.

PYTHON SOFTWARE FOUNDATION. 2015. Python Language Reference, version 3.5. URL https://docs.python.org/3/reference/index.html.

R CORE TEAM. 2015. R: A Language and Environment for Statistical Computing .R Foundation for Statistical Computing. URL https://www.R-project.org/.

SHEWCHUCK JR. 1996. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Pittsburgh PA 15213: Technical Report CMU-CS-96-140, School of Computer Science, Carnegie Mellon University.

TAYLOR LR. 1961. Aggregation, variance and the mean. Nature 189: 732-735.

WUERTZ D, MAECHLER M AND RMETRICS CORE TEAM MEMBERS. 2016. Stabledist: Stable distribution functions. R package version 0.7-1. URL https://CRAN.R-project.org/package=stabledist.

APPENDIX
PYTHON IMPLEMENTATION

For completeness, we give here the full Python implementation of 3 modules (**nsum**, **Psialpha** and **nptweedie**). The following observations may be helpful for non-Python programmers:

- In Python, there is no **end**, **endif**, **end do**, etc., statement. The scope of **if** statements or loops is defined by indentation. To close long stretches of **if** statements or loops, we use the Python keyword **pass**, which does nothing.
- Python modules are libraries, and functions in a module can be imported into other programs, in a fashion similar to MODULA-2, ADA and R.
- Although Python functions are much more flexibile, the functions in the present modules are similar to C functions (returning a single value, or none) and they can be used on the right side of an assignment in exactly the same way.
- Calling programs do not need to manage the inner workings of the modules. In particular, there is no need to understand what a dictionary is in **nsum** (just use the recipe) or multiple precision in **nptweedie** (just use ordinary floating point; all conversions necessary are performed on the input and output values).

nsum.py

The adaptation of the function **msum** found in http://code.activestate.com/recipes/393090/, which is what is actually used to perform the sums, consists of a small Python module **nsum** with three functions: startsum, sumthis, and sumall. startsum creates a new entry in a internal dictionary called partials. A Python dictionary is like an associative array: instead of an integer, it is indexed by an entry; in our case, the name of the sum in a string. This allows the flexibility of keeping tab of several partial sums in parallel. Although this feature was not actually used in our implementation for summing the series (see module **nptweedie** in this appendix), it is useful, for example, for debugging.

Although not necessary, we adopt the convention of using strings as dictionary keys with the same name as the floating-point variable that ultimately stores the sum.

It is easy to understand how to use the functions in **nsum.py**, and this is all that is needed to understand how sums are calculated in the implementation **nptweedie.py** given below: startsum('sumd') starts a sum that will ultimately be stored in variable sumd; sumthis(dk,'sumd') adds a new term dk, without actually calculating the sum, and sumd = sumall('sumd') calculates the partial sum. Note that, because the Python dictionary is internal to **nsum**, knowledge of how Python dictionaries work, although useful, is not strictly necessary.

Here is the full implementation of **nsum.py**:

```
#!/usr/bin/python3
partials = {}
def startsum(mysum):
   partials[mysum] = []
def sumthis(x,mysum):
   i = 0
   for y in partials[mysum]:
      if abs(x) < abs(y):
         x, y = y, x
      pass
```

```
      hi = x + y
      lo = y - (hi - x)
      if lo:
         partials[mysum][i] = lo
         i += 1
      pass
      x = hi
   partials[mysum][i:] = [x]
   return
def sumall(mysum):
   return( sum(partials[mysum]))
```

Psialpha.py

```
#!/usr/bin/python3
Psi_a = -5.7573749548413
Psi_b = 5.74962006661501
Psi_c = 1.39742867043842
```

nptweedie.py

This module implements the summation of the Tweedie series using 1 000-bit precision. The most useful function in the module is pdfz_tweedie(z,theta,alpha); its use is self-explanatory.

```
#!/usr/bin/python3
#-----------------------------------------------------------------------
# nptweedie.py
#
# N. L. Dias (nldias@ufpr.br)
#-----------------------------------------------------------------------
from  mpmath import loggamma, sqrt, sin, pi, fabs,\
                    exp, log, mp, mpf
from math import exp as mexp
from sys import exit
from nsum import startsum, sumthis, sumall
from Psialpha import Psi_a, Psi_b, Psi_c

bitprecision = 1000

control = True          # by default returns zero if z is too small

def setcontrol(bool):   # sets/unsets control
   global control
   control = bool
```

```
dbgoutput = False       # in the calling program, change this var to
                        # True with setdbg() if you want debugging
                        # output (Figure 1)

def setdbg(bool):       # sets/ unsets debugging info in auxiliary
                        # files
   global dbgoutput
   dbgoutput = bool


def testdbg():          # tests the variable dbgoutput
   if dbgoutput:
      print('True')
   else:
      print('False')
   pass

kconv = 0
ra = mpf(0)

def kcon():
   return kconv

def rate():
   return ra


def kappa(theta,alpha):
   res = ((alpha-1)/alpha )*((theta/(alpha-1))**alpha)
   return res

def bkshort(alpha,BB,k):
   xx = loggamma(1 + alpha*k)
   yy = k*log(BB)
   zz = loggamma(1 + k)
   bk = exp(xx + yy - zz)
   return bk

def pdfx_tweedie(x,theta,alpha,lambdaa):
   return (1.0/lambdaa)*pdfz_tweedie(x/lambdaa,theta,alpha)
```

```
def pdfz_tweedie(z,theta,alpha):
   Nz = Nz_tweedie(z,theta,alpha)
   return Nz*mexp(theta*z - kappa(theta,alpha))

def Nz_tweedie(z,theta,alpha):
# ----------------------------------------------------------------------
# sum the series
# ----------------------------------------------------------------------
# set the precision for mpf
# ----------------------------------------------------------------------
   mp.prec = bitprecision
# ----------------------------------------------------------------------
# convert from floats to mpfs with exactly the same digits: hence the
# need for str()
# ----------------------------------------------------------------------
   z = mpf(str(z))
   theta = mpf(str(theta))
   alpha = mpf(str(alpha))
# ----------------------------------------------------------------------
# I can't do miracles
# ----------------------------------------------------------------------
   assert mpf('0.01') < alpha < mpf('0.99')
# ----------------------------------------------------------------------
# calculate p initially as a float
# ----------------------------------------------------------------------
   p = (alpha-2)/(alpha-1)
   if dbgoutput:
      bckname = "mp-s-p%6.4fz%6.4f.dat" % (p,z)
      fbck = open(bckname,'wt')
      fbck.write("#234567890"+"1234567890"*11)
      fbck.write("\n")
      header = '#  k                          '+\
               'bk                       '+\
               'sumd                     '+\
               'relerr'
   pass
   eps = mpf('1.0e-10')           # a relative error
   nmax = 10000                   # maximum number of terms to sum
   larg = -1/z                    # auxiliary
   BB = fabs(kappa(larg,alpha))   # it is possible that BB < 0
   oB = 1/BB                      # the reciprocal
```

```
   def abc(alpha):                 # is it possible?
      return exp(Psi_a + Psi_b*alpha**Psi_c)
   if (control) and (oB < abc(alpha)):     # returns zero
      return(float(0.0))
   pass
   startsum('sumd')                # the sum of dk
   sumd = mpf(0)                   # initialize
   relerr = mpf(1)                 # a relative error
   k = 1                           # k
   sign = -1                       # the sign of (-1)**k
   maxter = mpf(0)                 # the maximum term so far
   fkmax = z**(mpf(2)-p)/(p-mpf(2))# the estimated locus of the max
   kmax = int(fkmax)
# ---------------------------------------------------------------------
# for double precision, check if kmax is less than 10000
# ---------------------------------------------------------------------
   if (kmax > nmax) :
      print('too many expected iterations (> %d)' % nmax)
      return(float('nan'))
   pass
   lbmax = (1-alpha)*fkmax + mpf('0.5')*log(alpha)# the log of the max
   bmax = exp(lbmax)
# ---------------------------------------------------------------------
# check for debugging output
# ---------------------------------------------------------------------
   if dbgoutput :
      fbck.write(
      ("# theta=%10.6f alpha=%10.6f BB = %15.5e period = %d"+
       " kmax = %d, bmax = %15.8e eps=%15.8e\n") %
      (theta,alpha,BB,9999,kmax,bmax,eps))
   pass
# ---------------------------------------------------------------------
# loop to sum the series
# ---------------------------------------------------------------------
   while ((sumd < 0) or (fabs(relerr) > eps)):
      bk = bkshort(alpha,BB,k)/bmax
      ck = bk*sign
      dk = ck*sin(-k*pi*alpha)
      sign = -sign
# ---------------------------------------------------------------------
# sum without loss of precision
# ---------------------------------------------------------------------
```

```
      sumthis(dk,'sumd')
      sumd = sumall('sumd')
      relerr = fabs(bk/sumd)
      if dbgoutput :
         fbck.write("%4d %25.15e %25.15e %25.15e\n" %
            (k,bk,sumd,relerr))
      pass
      k += 1
      if (k > nmax):
         kconv = k - 1
         print('Nz_tweedie reached %d terms without convergence' % nmax)
         return(float('nan'))
      pass
   pass # end of while
   sumd *= mpf(1)/(pi*z)
   sumd *= bmax
   if dbgoutput:
      fbck.close()
   pass
   kconv = k - 1
   return float(sumd)
```