A More Accurate Briggs Method for the Logarithm

Al-Mohy, Awad H.

2011

MIMS EPrint: **2011.43**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

A More Accurate Briggs Method for the Logarithm

Awad H. Al-Mohy

Abstract A new approach for computing an expression of the form $a^{1/2^k} - 1$ is presented that avoids the danger of subtractive cancellation in floating point arithmetic, where a is a complex number not belonging to the closed negative real axis and k is a nonnegative integer. We also derive a condition number for the problem. The algorithm therefore allows highly accurate numerical calculation of $\log(a)$ using Briggs' method.

Keywords logarithm function, Briggs' method, Briggs' tables, inverse scaling and squaring method

Mathematics Subject Classification (2000) 15A60, 65F30

1 Introduction

In the 17th century, Henry Briggs published his table of logarithms in *Arithmetica Logarithmica*. He used an elegant method to calculate logarithms of positive real numbers and build up tables of logarithms. To approximate $\log(a)$, where $a > 0$, Briggs exploited the relation $\log(a) = 2\log(a^{1/2})$ repeatedly and obtained $\log(a) = 2^k \log(a^{1/2^k})$, where $k \in \mathbb{N}$. Then he used the first order approximation of the logarithm function $\log(1 + x) \approx x$, which yields a good approximation for small x. Taking $x = a^{1/2^k} - 1$ for sufficiently large k (Briggs' choice was 54), he calculated logarithms to base 10 via the relation [3]

$$\log_{10}(a) \approx 2^k \log_{10}(e)(a^{1/2^k} - 1). \tag{1.1}$$

Version of May 25, 2011.

A. H. Al-Mohy
Department of Mathematics, King Khalid University, Abha, Saudi Arabia
E-mail: aalmohy@hotmail.com
http://www.maths.manchester.ac.uk/~almohy

The method of Briggs has been generalized to the matrix logarithm as proposed by Kenney and Laub [4] in a method known in the literature as the *inverse scaling and squaring* method. The weakness of the method for both the scalar and the matrix case is that the computation of the key quantity $a^{1/2^k} - 1$ in floating point arithmetic (by calculating k successive square roots then subtracting 1) is prone to loss of accuracy because of massive subtractive cancellation that can occur when $a^{1/2^k}$ approaches 1 for a large k. Cancellation happening when two nearly equal numbers are subtracted often leads to numerical instability. For more insight into the effect of cancellation and how it can sometimes be avoided by using different mathematical formulations, see [2, sec. 1.7]. To illustrate, take $a = e$ and $k = 24$ and consider IEEE double precision arithmetic, for which the unit roundoff is $u := 2^{-53} \approx 1.1 \times 10^{-16}$. As $1/2^k$ is small, we have $a^{1/2^k} - 1 \approx \log_e(a)/2^k \approx 10^{-8}$. Thus almost half of the significant digits in $a^{1/2^k}$ are lost, and hence $\log_{10}(a)$ inherits the loss of significant digits if approximated using (1.1).

Kenney and Laub [5] presented an overview of Briggs' method and gave a relative error analysis on how the number k can be selected. They pointed out the danger of cancellation in the calculation of $a^{1/2^k} - 1$. In the same context, Dieci and Papini [1] discussed this weakness and illustrated it by a similar example. They stated that "It is important to stress that the observed loss of digits is unavoidable given the finite precision representation of x, and no algorithm to approximate the logarithm can avoid it, whether or not 1 is subtracted".

In this paper we present a new algorithm for computing the quantity $a^{1/2^k} - 1$ that avoids subtractive cancellation, and illustrate its advantages in numerical experiments. For $a \in \mathbb{C} \setminus \mathbb{R}^-$, where $\mathbb{R}^-$ is the closed negative real axis, we denote by $a^{1/2}$ the *principal square root* of a, which is the solution of the equation $z^2 - a = 0$ whose real part is positive. In addition, we denote by $\log(a)$ the unique *principal logarithm* of a, which is the solution of the equation $e^z = a$ whose imaginary part lies in the strip $\{ z : -\pi < \mathrm{Im}(z) < \pi \}$ [3, Thm. 1.31].

2 Avoiding subtractive cancellation

In this section we present a formulation of the quantity $a^{1/2^k} - 1$ and show its advantage in avoiding subtractive cancellation. We begin by the following lemma.

Lemma 2.1 *For $a \in \mathbb{C} \setminus \mathbb{R}^-$, we have*

$$a^{1/2^k} - 1 = \frac{a - 1}{\prod_{i=1}^{k} (1 + a^{1/2^i})}. \tag{2.1}$$

Proof Applying the relation $a^{1/2^k} - 1 = (a^{1/2^{k+1}} - 1)(a^{1/2^{k+1}} + 1)$ repeatedly, we have

$$\begin{aligned} a - 1 &= (a^{1/2} + 1)(a^{1/2} - 1) \\ &= (a^{1/2} + 1)(a^{1/4} + 1)(a^{1/4} - 1) \\ &\vdots \\ &= (a^{1/2} + 1)(a^{1/4} + 1)(a^{1/8} + 1) \dots (a^{1/2^k} + 1)(a^{1/2^k} - 1) \\ &= (a^{1/2^k} - 1) \prod_{i=1}^{k} (1 + a^{1/2^i}). \end{aligned}$$

The formula (2.1) follows immediately. □

If a in Lemma 2.1 is a real number, then $a > 0$ and it is clear that the denominator in the right hand side of (2.1) is a product of positive real numbers, so subtractive cancellation cannot occur in floating point arithmetic. If a is nonreal, the product $\prod_{j=1}^{k}(1 + a^{1/2^j})$ can be evaluated in either polar form or conventional form. We now show that subtractive cancellation cannot occur in the product when using the polar form of a whereas it can occur when using conventional product formula for the complex numbers, and we show how that can be *completely avoided*. First, we use the polar representation $a := \rho e^{i\theta}$, where $\rho = |a| > 0$ and $\theta = \arg(a)$ with $0 < |\theta| < \pi$. Thus the principal square root of a can by written as $a^{1/2} = \rho^{1/2}\big(\cos(\theta/2) + i\sin(\theta/2)\big)$ and therefore $a^{1/2^j} = \rho^{1/2^j}\big(\cos(\theta/2^j) + i\sin(\theta/2^j)\big)$. Let $1 + a^{1/2^j} = \rho_j\, e^{i\theta_j}$, where $j = 1\colon k$. Thus we have

$$\rho_j^2 = 1 + \rho^{1/2^{j-1}} + 2\rho^{1/2^j}\cos(\theta/2^j), \quad \theta_j = \tan^{-1}\left(\frac{\rho^{1/2^j}\sin(\theta/2^j)}{1 + \rho^{1/2^j}\cos(\theta/2^j)}\right), \tag{2.2}$$

and

$$\prod_{j=1}^{k}(1 + a^{1/2^j}) = \left(\prod_{j=1}^{k}\rho_j\right) e^{i\sum_{j=1}^{k}\theta_j}. \tag{2.3}$$

We show now that the polar representation of the product in (2.3) is not prone to subtractive cancellation. There are two cases for θ. If $0 < \theta < \pi$, then the θ_j are positive for all j since $\sin(\theta/2^j)$ is positive. Similarly, if $-\pi < \theta < 0$, then the θ_j are negative. In either cases, $\sum_{j=1}^{k}\theta_j$ is a sum of one-signed real numbers, θ_j, so subtractive cancellation cannot happen.

Second, consider now the conventional product formula for the complex numbers

$$(x_1 + iy_1)(x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1), \tag{2.4}$$

where $(x_1, y_1), (x_2, y_2) \in \mathbb{R} \times \mathbb{R}$. We investigate the possibility of cancellation when evaluating the denominator of (2.1) using the formula (2.4). Let $\alpha_j +$

$i\beta_j = 1 + a^{1/2^j}$ and $x_s + iy_s = \prod_{j=1}^{s}(1 + a^{1/2^j})$. We have $x_1 = \alpha_1$ and $y_1 = \beta_1$, so we can compute $x_k + iy_k$ using (2.4) via the recurrence

$$\begin{aligned} x_s &= x_{s-1}\alpha_s - y_{s-1}\beta_s \\ y_s &= y_{s-1}\alpha_s + x_{s-1}\beta_s, \quad s = 2\colon k. \end{aligned} \tag{2.5}$$

For two floating point numbers λ_1 and λ_2 resulting from several other calculations, the necessary condition for subtractive cancellation is to have $|\lambda_1 - \lambda_2|/|\lambda_1| \ll 1$. Hence if this relative difference is of order 1, it is a sign that no cancellation can occur. We use this test below to verify whether subtractive cancellation can happen at any step in real or imaginary parts of the recurrence (2.5).

As the tangent function is one-to-one in the interval $(-\pi/2, \pi/2)$ and its inverse is strictly increasing, we have from (2.2) that [1]

$$0 < |\theta_j| = \tan^{-1}\left(\frac{\rho^{1/2^j}\sin(|\theta|/2^j)}{1 + \rho^{1/2^j}\cos(|\theta|/2^j)}\right) < \tan^{-1}\big(\tan(|\theta|/2^j)\big) = |\theta|/2^j \tag{2.6}$$

for all $j \geq 1$ and therefore

$$\left|\sum_{j=1}^{k}\theta_j\right| = \sum_{j=1}^{k}|\theta_j| < \sum_{j=1}^{k}|\theta|/2^j < |\theta|\sum_{j=1}^{\infty}2^{-j} = |\theta|. \tag{2.7}$$

From (2.3) we have

$$x_s = \left(\prod_{j=1}^{s}\rho_j\right)\cos\left(\sum_{j=1}^{s}\theta_j\right), \quad y_s = \left(\prod_{j=1}^{s}\rho_j\right)\sin\left(\sum_{j=1}^{s}\theta_j\right). \tag{2.8}$$

We observe that a massive subtractive cancellation can occur in the real parts if $|\sum_{j=1}^{s}\theta_j| \approx \pi/2$. To illustrate, take $a = \rho e^{i\theta}$, where $\rho = 2$ and $\theta = 2.780027365256823$. When computing $x_k + iy_k$ using the recurrence (2.5) for $k > 10$, we obtain at $s = 9$

$$x_{s-1}\alpha_s = 1.4343726728602473, \quad y_{s-1}\beta_s = 1.4343726728603925,$$

and $x_s = -1.45\times10^{-13}$. Using (2.2), we calculate $|\pi/2 - \sum_{j=1}^{s}\theta_j| \approx 4.4\times10^{-16}$. This situation can completely be avoided if $|\theta| < \pi/2$. Then from (2.6) and (2.7), we have

$$0 < |\theta_j| < \frac{\pi}{2^{j+1}}, \quad 0 < \sum_{j=1}^{s}|\theta_j| < \frac{\pi}{2}. \tag{2.9}$$

Thus from (2.5), (2.8), (2.7), and (2.9) we obtain

$$\frac{|y_s|}{|y_{s-1}\alpha_s|} = \frac{\sin\left(\sum_{j=1}^{s}|\theta_j|\right)}{\sin\left(\sum_{j=1}^{s-1}|\theta_j|\right)\cos\left(|\theta_s|\right)} > \frac{1}{\cos\left(|\theta_s|\right)} > 1$$

[1] Note that $|f(\theta)| = f(|\theta|)$, where $f \in \{\sin, \cos, \tan\}$, for $\theta \in (-\pi/2, \pi/2)$. Also, $|\tan^{-1}(x)| = \tan^{-1}(|x|)$ for all $x \in \mathbb{R}$.

for all $s > 1$ since the sine function is increasing over the interval $(0, \pi/2)$, so subtractive cancellation cannot occur in the imaginary parts. For the real parts, it seems not straightforward to have such a lower bound by inspection since the cosine function is decreasing over the interval $(0, \pi/2)$, but we find that

$$\frac{|x_s|}{|x_{s-1}\alpha_s|} = \frac{\cos\left(\sum_{j=1}^{s} |\theta_j|\right)}{\cos\left(\sum_{j=1}^{s-1} |\theta_j|\right)\cos\left(|\theta_s|\right)} > \frac{1}{2},$$

using the optimization function `fmincon` of MATLAB, where we minimized $x_s/(x_{s-1}\alpha_s)$ subject to the constraints in (2.9) for $s = 2 : 100$. Accordingly, the real parts cannot suffer from subtractive cancellation.

In summary, if a belongs strictly to the right half of the complex plane, the evaluation of the denominator of (2.1) by the conventional product formula in the recurrence (2.5) is immune to subtractive cancellation. The trick to extend this conclusion to an arbitrary complex number a with $|\arg(a)| < \pi$ is to first perform one square root of a, which lies strictly in the right half of the complex plane. Then we use (2.1) for $a^{1/2}$, that is, replace a and k by $a^{1/2}$ and $k - 1$ in (2.1), respectively. We are now in a position to write our algorithm.

3 Algorithms and conditioning

We present in this section two algorithms for computing $g_k(a) := a^{1/2^k} - 1$ and derive a condition number for this function. The next algorithm is the standard approach to compute the expression $a^{1/2^k} - 1$.

Algorithm 3.1 (standard version) *This algorithm computes the quantity* $r := a^{1/2^k} - 1$ *directly from this formula.*

1 for $i = 1{:}\, k$
2 $\quad a \leftarrow a^{1/2}$
3 end
4 $r = a - 1$

Based on the analysis in the previous section we have the following algorithm.

Algorithm 3.2 (improved version) *This algorithm computes the quantity* $r := a^{1/2^k} - 1$ *from the formula* (2.1).

1 $\widehat{k} = k$
2 if $\arg(a) \geq \pi/2$
3 $\quad a \leftarrow a^{1/2}$, $\widehat{k} = k - 1$
4 end
5 $z_0 = a - 1$
6 $a \leftarrow a^{1/2}$
7 $r = 1 + a$

```
 8  for j = 1:k̂ − 1
 9      a ← a^{1/2}
10      r ← r(1 + a)
11  end
12  r = z_0/r
```

Algorithm 3.2 involves more arithmetic operations than Algorithm 3.1 if they are executed using the same precision and the same algorithm for computing $a^{1/2}$. Thus, Algorithm 3.2 has $2k$ and $7k+3$ extra flops for a real and complex a, respectively. However, these extra arithmetic operations are worth paying provided that we gain better accuracy. The operation counts for the complex case are based on the fact that one invocation in line 10 costs 7 flops when using the recurrence (2.5) after adding 1 to the real part. The division in the last step costs 9 flops assuming we use the division formula [6, sec. 5.4]

$$\frac{x+iy}{c+id} = \begin{cases} \dfrac{[x+y(d/c)]+i[y-x(d/c)]}{c+d(d/c)}, & |c| \geq |d|, \\ \dfrac{[x(c/d)+y]+i[y(c/d)-x]}{c(c/d)+d}, & |c| < |d|. \end{cases}$$

To have insight into the conditioning of the quantity $\prod_{j=1}^{k}(1+a^{1/2^j})$, consider the principal branch of the complex function $f_k(z) = \prod_{j=1}^{k}(1+z^{1/2^j})$, with $-\pi < \arg(z) < \pi$, so f is single-valued. The condition number of f_k at z is given by [3, sec. 3.1]

$$\operatorname{cond}(f_k, z) = \frac{|zf_k'(z)|}{|f_k(z)|} = \sum_{j=1}^{k} \frac{1}{2^j} \frac{|z^{1/2^j}|}{|1+z^{1/2^j}|}. \tag{3.1}$$

Since $\operatorname{Re}(a^{1/2^j}) > 0$ for all $j \geq 1$, we have $|a^{1/2^j}|/(|1+a^{1/2^j}|) \leq 1$ and conclude that

$$\operatorname{cond}(f_k, a) \leq \sum_{j=1}^{k} \frac{1}{2^j} < \sum_{j=1}^{\infty} \frac{1}{2^j} = 1. \tag{3.2}$$

Thus $f_k(z)$ is a very well-conditioned function of z. Now we analyze the conditioning of the original problem, $a^{1/2^k} - 1$. The principal branch of the complex function $g_k(z) = z^{1/2^k} - 1$, where $-\pi < \arg(z) < \pi$, is analytic. Thus with a little manipulation the condition number of g_k at $a := \rho e^{i\theta}$ is

$$\operatorname{cond}(g_k, a) = \frac{|a^{1/2^k}|}{2^k |g_k(a)|} \leq \frac{\rho^{1/2^k}}{2^k |g_k(\rho)|}. \tag{3.3}$$

To obtain a lower bound for the condition number, suppose first that $\rho < 1$. Then from (2.2) we have $\rho_{k+1} < 2$. Using (2.1) we obtain

$$\frac{2^{k+1}|g_{k+1}(a)|}{2^k |g_k(a)|} = \frac{2}{|1+a^{1/2^{k+1}}|} = \frac{2}{\rho_{k+1}}.$$

Thus the sequence $2^k|g_k(a)|$ is increasing since $2/\rho_{k+1} > 1$. In addition, this sequence converges to $|\log(a)|$. Therefore, $|\log(a)|$ is the least upper bound of the sequence, that is, for all k we have

$$2^k|g_k(a)| \le |\log(a)| = (\log(\rho)^2 + \theta^2)^{1/2}.$$

Combining this bound with (3.3) yields

$$\frac{\rho^{1/2^k}}{(\log(\rho)^2 + \theta^2)^{1/2}} \le \operatorname{cond}(g_k, a) \le \frac{\rho^{1/2^k}}{2^k|g_k(\rho)|}. \tag{3.4}$$

There is a nice relation between $\operatorname{cond}(g_k, a)$ and $\operatorname{cond}(g_k, 1/a)$. We have

$$\operatorname{cond}(g_k, 1/a) = \frac{|a^{-1/2^k}|}{2^k|g_k(1/a)|} = \frac{1}{2^k|g_k(a)|} = \frac{\operatorname{cond}(g_k, a)}{|a^{1/2^k}|}.$$

Now, if $\rho > 1$ then $|1/a| = 1/\rho < 1$, so we have using (3.4) the inequality

$$\frac{1}{(\log(\rho)^2 + \theta^2)^{1/2}} \le \rho^{1/2^k}\operatorname{cond}(g_k, 1/a) = \operatorname{cond}(g_k, a). \tag{3.5}$$

The bounds (3.4) and (3.5) tell us that the condition number can be large if ρ and θ are sufficiently close to 1 and 0, respectively, at the same time, and it is not k that makes the condition number large. An example is instructive. Take $a = 1 - 2^{-50} + 2^{-24}i$. For $k = 1\colon 50$, we have $\operatorname{cond}(g_k, a) \approx 1.7 \times 10^7$, using (3.3), and the relative error $|g_k(a) - \widehat{r}|/|g_k(a)| \approx 1.5 \times 10^{-8}$, where $\widehat{r}$ is computed by Algorithm 3.2.

The numerical experiment in the next section gives more insight into stability of the algorithms.

4 Numerical experiment

In this section we conduct two numerical experiments. The first will show the improved accuracy of Algorithm 3.2 over Algorithm 3.1, and the second will show the accuracy of Briggs' method for computing the logarithm when equipped with Algorithm 3.2. Both were carried out in MATLAB R2010a on a machine with Core i7 processor. For ease of notation, let the functions g_k and $\operatorname{cond}(g_k, \cdot)$ above operate in an elementwise fashion when given vector arguments.

Experiment 1. We generate a row vector v of 20 points linearly spaced between 2 and 10 using the MATLAB function `linspace` as $v =$`linspace(2,10,20)`. Then we build up the vector $a = [10^{-8}v, v, 10^8 v]$ of length 60. The reason for writing the vector a in this way is to test the behavior of the algorithms for small, medium, and large elements. Denote by $\widehat{g_k}(a)$ the computed values of $g_k(a)$ in floating point arithmetic using Algorithm 3.1 and Algorithm 3.2, where $k = 1\colon 60$. The "exact" $g_k(a)$ were evaluated in 100 decimal digit arithmetic using the Symbolic Math Toolbox. Figure 4.1 (top) displays the relative

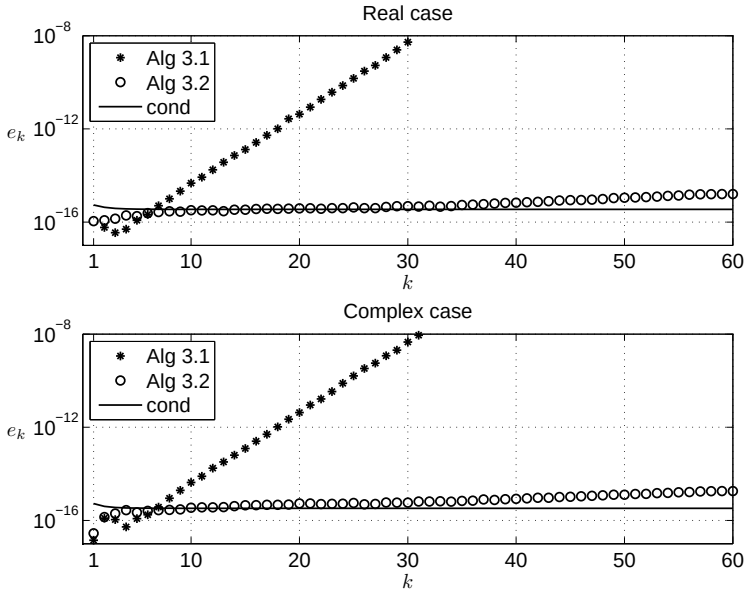

Fig. 4.1 Experiment 1: relative errors, e_k, from Algorithm 3.1 and Algorithm 3.2 plotted versus the number of square roots, k, applied to the entries of the vectors a (top) and b (bottom).

errors $e_k := \|g_k(a) - \widehat{g_k}(a)\|_2 / \|g_k(a)\|_2$ and $\|\mathrm{cond}(g_k, a)\|_2 u$ (solid line), where $k = 1\colon 60$. Notice that for each k in the top part two points are plotted representing the relative errors e_k associated with each algorithm.

We repeat the test above for the vector a replaced by a vector b with $b_j = a_j + 2^{j(-1)^j} e^{\pi i j/60}$, where $j = 1\colon 60$. Figure 4.1 (bottom) shows the result.

This experiment reveals the superiority of Algorithm 3.2 over Algorithm 3.1. When the number of square roots k increases, the relative error for Algorithm 3.1 deteriorates rapidly until it reaches 1 at $k = 58$. However, the relative error remains of order u for each k when using Algorithm 3.2.

Experiment 2. In this experiment we test the computation of $\log_{10}(a)$ from (1.1) using Algorithm 3.2 to evaluate $a^{1/2^k} - 1$ versus the MATLAB function `log10`. We set $k = 54$ (Briggs' choice) [2] and use the vector a described at the beginning of Experiment 1. We apply the MATLAB function `log10` and (1.1) to the entries of a, so denote the computed value by each of them by $\widehat{x}$. We calculate the relative errors $e_j = |\log_{10}(a_j) - \widehat{x}|/|\log_{10}(a_j)|$, where $j = 1\colon 60$. For the "exact" $\log_{10}(a_j)$, we used the Symbolic Math Toolbox in 100 decimal digits precision. We notice that `log10` produces many errors of zero, but to facilitate the plots we replace a zero error by 10^{-18}. Figure 4.2 shows the

[2] Of course, the choice of k should depend on a, but we want to see what would have happened if Briggs had used the improved algorithm.

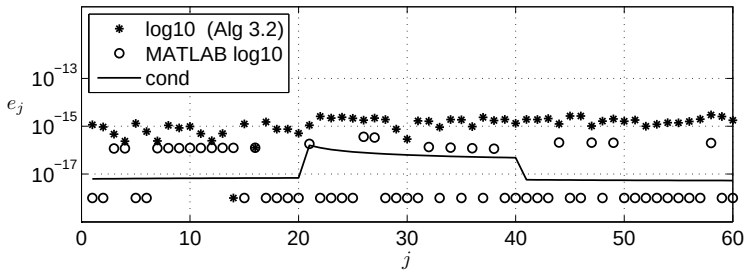

Fig. 4.2 Experiment 2: relative errors, e_j, from computing $\log_{10}(a)$ via (1.1) by using Algorithm 3.2 and the MATLAB function `log10`.

result. The solid line represents the values $\mathrm{cond}(\log_{10}, a_j)u$ for $j = 1\colon 60$, where $\mathrm{cond}(\log_{10}, \cdot)$ is the condition number for the logarithm function to base 10 given by $\mathrm{cond}(\log_{10}, z) = 1/|\log_e(10)\log(z)|$.

Briggs' method equipped with Algorithm 3.2 is highly accurate and returns results almost as good as those obtained by using the MATLAB function `log10`. It is natural to ask how Briggs succeeded in obtaining 14 significant digits in his tables while taking up to 54 successive square roots using Algorithm 3.1! The answer [3, sec. 11.5], [5] is that Briggs calculated to about 30 decimal digits in order to obtain the 14-digit logarithms. If he had used Algorithm 3.2, the calculation to 16 decimal digits would have been enough for him to generate his tables.

5 Conclusion

Some of the key properties of the field of the real numbers are not valid in the floating point arithmetic. For instance, associativity and the distribution law are not properties of floating point arithmetic. Thus different mathematical formulations of problems can lead to more or less accurate and stable numerical algorithms. This is what makes numerical computing so interesting. Poor accuracy in the computation of the expression $a^{1/2^k} - 1$ has been the weakness of the inverse scaling and squaring method. We solved this problem by using a variant of this expression. Algorithm 3.2 behaves in a stable manner reflecting the conditioning of the problem. The importance of Algorithm 3.2 stems from the fact that it is readily extended to matrix case. We are currently applying this idea to derive a new inverse scaling and squaring algorithm for the matrix logarithm.

Acknowledgments

I am very grateful to Professor Nick Higham for his valuable comments and suggestions on several drafts of this manuscript. I also thank the University of Manchester for allowing me to use their electronic resources.

References

1. L. Dieci and A. Papini. Conditioning and Padé approximation of the logarithm of a matrix. *SIAM J. Matrix Anal. Appl.*, 21(3):913–930, 2000.
2. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002. ISBN 0-89871-521-0. xxx+680 pp.
3. N. J. Higham. *Functions of Matrices: Theory and Computation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008. ISBN 978-0-898716-46-7. xx+425 pp.
4. C. S. Kenney and A. J. Laub. Condition estimates for matrix functions. *SIAM J. Matrix Anal. Appl.*, 10(2):191–209, 1989.
5. C. S. Kenney and A. J. Laub. A Schur–Fréchet algorithm for computing the logarithm and exponential of a matrix. *SIAM J. Matrix Anal. Appl.*, 19(3):640–663, 1998.
6. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992. ISBN 0-521-43064-X. xxvi+963 pp.