APRI-PH-2004-12b
June 3, 2004

Coincidence, data compression, and Mach's concept of "economy of thought"

J. S. Markovitch

P.O. Box 2411, West Brattleboro, VT 05303
Email: jsmarkovitch@yahoo.com

Abstract

A case is made that Mach's principle of "economy of thought", and therefore usefulness, is related to the compressibility of data, but that a mathematical expression may compress data for reasons that are sometimes coincidental and sometimes not. An expression, therefore, may be sometimes explainable and sometimes not. A method is proposed for distinguishing coincidental data compression from non-coincidental, where this method may serve as a guide in uncovering new mathematical relationships. The method works by producing a probability that a given mathematical expression achieves its compression purely by chance.

Introduction

The following equation produces a value that differs from 20 by just 0.000900020….

$$e^{\pi} - \pi = 19.999099979... \quad , \tag{1}$$

(see Almost Integer, in Weisstein, 1999). Equation (1) is in fact very suggestive. It hints at a relationship between π and *e* different from the well-known $e^{i\pi} + 1 = 0$, but distinct in that Eq. (1) is independent of the complex plane.. But so far Eq.(1) has led to no new interesting infinite series for π and stands aloof from the remainder of mathematics.

The following two equations, which employ π, *e*, and the golden ratio $\phi = \left(1 + \sqrt{5}\right)/2$, also produce values that are close to integers:

$$e^{\pi\sqrt{43}} = 884736743.99977... \quad \text{, and,} \tag{2}$$

$$\phi^{10} = \left(\frac{1+\sqrt{5}}{2}\right)^{10} = 122.9918... \tag{3}$$

On the surface, these equations also appear to be coincidental and therefore mathematical dead ends. Only careful study reveals that, for values of p equal to 2, 3, 5, 11, 17, and 41, the expression $e^{\pi\sqrt{4p-1}}$ has a profound relationship with the complex prime numbers of Gauss (Conway and Guy, 1996; see also *j*-Function, Almost Integer, Gauss's Class Number

Problem, and Prime-Generating Polynomial, in Weisstein, 1999). The non-coincidental nature of Eq. (2) is strongly implied by noting that for p equal to 11, 17, and 41 the expression $e^{\pi\sqrt{4p-1}}$ produces

$$e^{\pi\sqrt{4\times11-1}} = 884736743.99977... \approx 960^3 + 744 \ , \tag{4a}$$

$$e^{\pi\sqrt{4\times17-1}} = 147197952743.9999986... \approx 5280^3 + 744 \ , \text{ and} , \tag{4b}$$

$$e^{\pi\sqrt{4\times41-1}} = 262537412640768743.99999999999925... \approx 640320^3 + 744 \ , \tag{4c}$$

which Conway and Guy (1996) describe as "suspiciously close to integers". (Note: Eq. (4c) is the last of this unusual sequence and is sometimes referred to as the Ramanujan Constant (Weisstein, 1999)).

The non-coincidental nature of Eq. (3), which raises the golden ratio ϕ to the power of 10, is even more decisively confirmed by observing that

$$\phi^{10} = 122.9918... \ , \tag{5a}$$

$$\phi^{20} = 15126.999933... \ , \tag{5b}$$

$$\phi^{30} = 1860497.99999946... \ , \tag{5c}$$

$$\phi^{40} = 228826126.9999999956... \text{ , and,} \tag{5d}$$

$$\phi^{50} = 28143753122.9999999999644... \text{ ,} \tag{5e}$$

which are ever nearer to integers for reasons that must be non-coincidental. In fact, given that for f(n) = ϕ^n, f(10) = 122.9918…, f(11) = 199.0050…, f(12) = 321.9968…, f(13) = 521.0019…, f(14) = 842.9988…, and f(15) = 1364.00073…, it appears that for increasing values of n, the values produced are not merely ever closer to integers, but from alternating directions. (Note: this mystery is partially resolved by noting that $f(n) = \phi^n + (-\phi)^{-n}$ produces integers for all integer values of n.)

In contrast, note that that non-integers raised to high powers typically do not produce numbers that are at all close to integers. Accordingly, $\pi^{10} = 93648.04...$, $\pi^{20} = 8769956796.08...$, $\pi^{30} = 821289330402749.58...$, $e^{10} = 22026.46...$, $e^{20} = 485165195.40...$, and $e^{30} = 10686474581524.46...$ all produce no interesting sequences of 9s or 0s to the right of the decimal place, by coincidence or otherwise.

Clearly, it would be desirable to possess a general method for determining whether Eq. (1) is coincidental, or if, like Eqs. (2) and (3), it has general significance. This issue is brought into sharper focus when considering π approximations such as

$$\pi \approx \frac{22}{7} \text{ ,} \tag{6a}$$

$$\pi \approx \frac{355}{113} \text{ , and,} \tag{6b}$$

$$\pi \approx \left(\frac{2143}{22} \right)^{\frac{1}{4}} \text{ .} \tag{6c}$$

(see Pi, in Weisstein, 1999). Are these ever more precise approximations also coincidental? And what precisely does it mean to ask this question?

Equations (6a)-(6c) appear to be useful in the sense that they serve as handy compressions of data and serve the psychological function of aiding a researcher with a limited memory. In this sense they are of at least limited "scientific" value, in the broad sense of the term used by Mach, for whom all equations, and all laws, were summaries of data (Mach, 1988).

But Eq. (1) and Eqs. (6a)-(6c) remind us that some equations or laws are more important than others. Some go beyond mere summary and are foundations for a still greater understanding. So which are they? Do they represent a blind alley? And is their initial success, though real, something that has no general significance?

One approach to answering these questions involves applying information theory to assess whether a given expression compresses data. The idea is that if an expression simplifies data significantly, then it probably does so for some important underlying reason that ultimately may be discovered .

Using Information Theory to Distinguish Coincidence from Non-Coincidence

The simplest way to assess compression is to see whether a given approximation employs fewer digits than the value it approximates. So 22 / 7 has three digits, and it produces *accurately* just three digits of π (so, 22 / 7 = 3.14̲28…, where only the underlined integers are correct). In the same way 355 / 113 contains six digits, and it produces accurately just seven digits of π (so 355 / 113 = 3.14159̲292…). Lastly, Eq. (6c) has makes use of 8 digits in order to produce 9 digits of π (or 3.141592̲6525…). This method, though crude, demonstrates the underlying difficulty of compressing data by putting it in a new form. The more digits one accounts for via the new representation, the more digits that are needed to carry it out. This suggests that if a particular approximation uses far fewer digits than the number of digits it correctly reproduces, some explanation of this feat may be possible.

As it is, the above crude measure can be improved upon by employing a more sophisticated means of assessing information content than merely counting decimal digits. Such an improvement is badly needed as 0.999999 is obviously a good approximation of 1.000000, and yet has no digits in common with it.

The most obvious improvement would be to substitute binary digits for decimal digits; this would be the equivalent of employing a ruler with a finer scale. A still more sophisticated method is to employ $\log_2($ *closeness* $)$, where *closeness* equals $\dfrac{1}{\left|1-\dfrac{R}{p/q}\right|}=\dfrac{\dfrac{p}{q}}{\left|\dfrac{p}{q}-R\right|}$, and $\dfrac{p}{q}$ is an approximation of *R*, where *p* and *q* are positive integers. So the expression $\log_2\left(\dfrac{\dfrac{22}{7}}{\left|\dfrac{22}{7}-\pi\right|}\right)$

yields the number of *bits* of π produced by the expression $\frac{22}{7}$. This solves the problem posed by 0.999999 and 1.000000.

Note that there are at least two other expressions that one might use to measure closeness: $\frac{1}{\left|1-\frac{p/q}{R}\right|}$ and $\frac{1}{\left|R-\frac{p}{q}\right|}$. The advantage that $\frac{1}{\left|1-\frac{R}{p/q}\right|}$ has over these alternatives is that later it will be tied directly to an important proof concerning what is achievable in rational approximations.

Having achieved an improvement in the measurement of the effectiveness of the approximation, the next logical step is to assess more precisely the "overhead" present in the approximating expression. This task is much more difficult because it is open-ended: there is literally no limit to the number of ways that one value may be approximated by another. It is also difficult because there is no single agreed upon set of rules for assessing the complexity of mathematical expressions.

For Eq. (6a), the information content of the expression $\frac{22}{7}$ might be estimated as $\log_2(22)+\log_2(7)$ bits. Why, you may ask, use the sum of logarithms? The answer is that these logarithms represent the cost or overhead, expressed in binary digits, incurred in reproducing π; a simple sum of these costs yields a total cost, or score, that lends itself to easy comparison with other scores that will be generated by other approximations. So the value $\log_2(7)$ equals roughly the number of bits that it takes to represent 7 as a binary number.

But why should binary digits hold the key to a consistent scoring method for the complexity of mathematical expressions? The answer may derive from communications

theory, where the simplest communications schemes involve sending messages composed of mere 1s and 0s. If one wishes to use these 1s and 0s to create an unambiguous identifier for each of 2^n messages, then n binary digits (or *bits*) are the minimum number required. So, for 4 messages, A, B, C, and D, $\log_2(4) = 2$ binary digits, or *bits*, are needed to pair each uniquely with its own identifier:

$$00 \leftrightarrow \text{A},$$

$$01 \leftrightarrow \text{B},$$

$$10 \leftrightarrow \text{C},$$

$$11 \leftrightarrow \text{D}.$$

Using the method introduced earlier, the *closeness* of $\frac{22}{7}$ to π equals

$$\log_2\left(\frac{\frac{22}{7}}{\left|\pi - \frac{22}{7}\right|}\right) \approx 11.3 \text{ bits.} \tag{7a}$$

And the cost, or overhead, incurred by $\frac{22}{7}$ in approximating π equals

$$\log_2(22) + \log_2(7) \approx 7.3 \text{ bits.} \tag{7b}$$

We see that the approximation $\frac{22}{7}$ compresses π significantly (11.3 – 7.3 = 4.0 bits). This clearly shows why $\frac{22}{7}$ might prove to be a helpful approximation: it simplifies the data it represents.

Similarly, given that

$$\log_2\left(\frac{\frac{355}{113}}{\left|\pi-\frac{355}{113}\right|}\right)\approx 23.5\text{ bits}, \tag{7c}$$

and

$$\log_2(355)+\log_2(113)\approx 15.3\text{ bits}, \tag{7d}$$

the approximation $\frac{355}{113}$ accomplishes substantial compression (23.5 – 15.3 = 8.2 bits). Because it significantly compresses the data it represents, $\frac{355}{113}$ is also clearly useful.

The above reasoning may be generalized to produce a formula for the number of binary digits of compression achieved by an approximation $\frac{p}{q}$ of R

$$Compression=\log_2\left(\frac{\frac{p}{q}}{\left|R-\frac{p}{q}\right|}\right)-\log_2(p)-\log_2(q)\ , \tag{8a}$$

$$= \log_2\left(\frac{\frac{p}{q}}{p \times q \times \left|R - \frac{p}{q}\right|}\right) = \log_2\left(\frac{1}{q^2 \times \left|R - \frac{p}{q}\right|}\right) \quad ,$$

where p and q are integers, and R, p, and q are all greater than 0.

Note that there exist infinitely many approximations $\frac{p}{q}$ to any real number R that satisfy

$$\left|R - \frac{p}{q}\right| \leq \frac{1}{\sqrt{5}q^2} \quad , \tag{8b}$$

(Conway and Guy, 1993; Hardy and Wright, 1980). This supports the conclusion that for any R, an approximation $\frac{p}{q}$ may be found such that

$$compression = \log_2\left(\frac{1}{q^2 \times \left|R - \frac{p}{q}\right|}\right) \geq \log_2\left(\sqrt{5}\right) \quad . \tag{8c}$$

Note that it is only when $\frac{p}{q}$ approximates R that Eq. (8c) can yield high values for compression, and that the smaller values of q also tend to yield higher compression scores. The

difficultly is in finding small $\left|R-\frac{p}{q}\right|$ accompanied by small q. Typically, if $\left|R-\frac{p}{q}\right|$ is to be made smaller, q must grow.

Note that Eq. (8c) assures that a compression of at least $\log_2\left(\sqrt{5}\right)=1.1609...$ bits is possible for any R. In Figure 1, an instructive puzzle is introduced that makes use of the principle behind Eq. (8b).

Tables I-III, and Figures 2-5, help demonstrate that limited compressibility is always possible to some degree, in this case for ϕ, e, and π. But Tables I-III and Figures 2-5 also illustrate the difficulty of finding really good approximations. Only $\pi \approx 22 / 7$ and $\pi \approx 355 / 113$, along with some of their multiples (e.g., 710 / 226), achieve even a single decimal digit of compression (a decimal digit equals about $\log_2(10) \approx 3.32$ bits), even though denominators were tested as high as 10,000.

The Origin of Good Rational Approximations

Good approximations may arise for reasons of luck (which must necessarily be infrequent), or because some underlying relationship has been exploited. This is made clearer by investigating *simple continued fractions*, which for x assume the form

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + ...}}}} \quad , \tag{9a}$$

which may be written more compactly as

$$x = [a_0, a_1, a_2, a_3, a_4, ...] \quad , \tag{9b}$$

where the positive integers a_0, a_1, a_2, a_3, and a_4 are the first five *partial quotients* of x.

So π, written as a simple continued fraction, equals

$$\pi = 3.1415926536... = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + ...}}}} \quad , \tag{9c}$$

or more simply

$$\pi = 3.1415926536... = [3, 7, 15, 1, 292, 1, ...] \quad . \tag{9d}$$

It is now possible to use simple continued fractions to understand the origin of the effectiveness of 355 / 113 and $\left(\frac{2143}{22}\right)^{\frac{1}{4}}$. For the first five powers of π we have the following simple continued fractions

$$\pi^1 = 3.1415926536... \quad = [3, 7, 15, 1, \mathbf{292}, 1, ...] \quad , \tag{9e}$$

$$\pi^2 = 9.8696044011... \quad = [9, 1, 6, 1, 2, \mathbf{47}, ...] \quad , \tag{9f}$$

$$\pi^3 = 31.0062766803... \quad = [31, \mathbf{159}, 3, 7, 1, 13, ...] \quad , \tag{9g}$$

$$\pi^4 = 97.4090910340... = [97, 2, 2, 3, 1, \mathbf{16{,}539}, ...] \text{ , and,} \qquad (9h)$$

$$\pi^5 = 306.0196847853... = [306, \mathbf{50}, 1, 4, \mathbf{60}, 1, ...] \text{ ,} \qquad (9i)$$

with their larger partial quotients appearing in boldface.

Now a key point is that an effective approximation for π may be achieved by truncating the continued fraction just before a large partial quotient. E.g.,

$$\pi \approx 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1}}} = \frac{355}{113} \quad . \qquad (9j)$$

So, in Figure 5, the large peak in the center of the graph occurs because π 's simple continued fraction may be truncated just before the fraction 1 / 292, thereby yielding the extremely good approximation 355 / 113.

It follows that the partial quotient 16,539 offers an especially good opportunity to create a good approximation. This expectation is justified as Eq. (6c) is successful because it chops off the continued fraction of Eq. (9h) at just the right spot

$$\pi^4 = 97.4090910340... \approx 97 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{1}}}} = \frac{2143}{22} \quad . \qquad (9k)$$

The approximation $\frac{2143}{22}$ achieves a very high compression of π^4 (29.5 – 15.5 = 14.0 bits).

Note that it is not a coincidence that $\log_2(16{,}539) \approx 14.0$ bits, just as it is not a coincidence that $\log_2(292) \approx 8.2$ bits, about the compression achieved by $\frac{355}{113}$. It follows that for a large partial quotient k one may construct an approximation achieving $\log_2(k)$ bits compression.

It is now clear that to ask whether the effectiveness of $\frac{355}{113}$ is due to chance is to ask whether the large term 292 occurs by chance in the simple continued fraction [3, 7, 15, 1, 292, 1, ...]. And to ask whether the effectiveness of $\frac{2143}{22}$ is due to chance is to ask whether 16,539 occurs by chance in the simple continued fraction [97, 2, 2, 3, 1, 16,539, ...].

As it is, the probability that an individual partial quotient of a randomly-generated number between 0 and 1 will equal or exceed k is $\log_2\left(1+\frac{1}{k}\right)$ (Khinchin, 1964). So, there is only a 1 in 11,464 chance that an individual quotient will equal or exceed 16,539, and only a 1 in 202 chance that it will equal or exceed 292 (though one should not assume that these two probabilities are independent). It follows that, for a randomly-generated number between 0 and 1, the probability $P(n)$ of achieving n bits of compression by truncating its continued fraction just before a partial quotient chosen at random is

$$P(n) \approx \log_2\left(1+\frac{1}{2^n}\right) \quad . \tag{9l}$$

Note that the presence of even a single "high compression" approximation, such as $\frac{355}{113}$, may provide a strong enough "signal" to justify an expectation that a non-coincidental relationship has been identified, that that some form of explanation of it may be possible. The use here of term "signal", borrowed from communications theory, is deliberate. Identifying the non-coincidental nature of a mathematical relationship may be seen as analogous to extracting a signal from noise. When a possible non-coincidental signal is identified, further work may, or may not, confirm it as non-coincidental.

Finding Approximations that Achieve a Predetermined Degree of Accuracy

It is important to note that it is easier to find a high compression approximation if one merely accepts the degree of accuracy produced by those approximations that happen to be effective. This is the situation with regard to 355 / 113, where π's continued fraction has been truncated at just the right point to obtain a good approximation. If one instead decides in advance that the approximation has to reproduce π to, say, 1 part in 10^9, then the likelihood of finding a good approximation by truncating π's continued fraction depends on the luck of π possessing a large partial quotient in just the right place. But such an expectation has already been shown not be to justified: large partial quotients are relatively rare.

By way of example, a brute force computer search finds that *eleven* digits are required to reproduce π to 1 part in 10^9:

$$\pi \approx \frac{103283}{32876} = \underline{3.14159265}11\ldots \quad , \tag{10a}$$

This approximation therefore cannot serve as a handy substitute for π's first nine digits, as it uses two more digits than it reproduces, and yet no value p/q with fewer digits can faithfully reproduce the first nine digits of π. This issue will become important in the next example where the data to be approximated are known as a result of physical experiment, and consequently one might want an approximation that fits the data to within its limits of experimental error.

The number of bits of data produced by the above π approximation is

$$\log_2\left(\frac{\frac{103283}{32876}}{\left|\frac{103283}{32876}-\pi\right|}\right)=\log_2\left(1300497890\right)\approx 30.3\text{ bits.} \tag{10b}$$

Now compare this against the number of bits of information used in the approximation

$$\log_2\left(103283\right)+\log_2\left(32876\right)\approx 31.7\text{ bits.} \tag{10c}$$

The approximation fails to accomplish any compression, as its score is negative (30.3 – 31.7 = –1.4 bits). If one disallows the "score inflation" that results from over-fitting π to better than 1 part in 10^9, then, with $\log_2\left(10^9\right)\approx 29.9\text{ bits}$, the score reduces to

$$\log_2\left(10^9\right)-\log_2\left(103283\right)-\log_2\left(32876\right)\approx -1.8\text{ bits}. \tag{10d}$$

In fact, without score inflation through over-fitting it is possible to state an important empirical rule, found with the aid of a computer: The probability *P(n)* that a number in the interval ($\pi - 0.1, \pi + 0.1$) around π may be approximated by p/q to an accuracy of better than 1 part per billion, while achieving *n* bits of compression, is

$$P(n) \approx \frac{1}{2^{n+0.8}} \ . \tag{11a}$$

where $n \geq 0$. Computer testing performed by the author verified this only for $n < 10$, but when the required accuracy was altered from 1 ppb to 100 ppb, it did not undermine its approximate validity; nor did substituting ϕ for π. Equation (11a) says that there is a 50 % likelihood of achieving better than 0.2 bits of compression ($n = 0.2$).

Remarkably, if instead of defining the number of bits of information overhead in an approximation as $\log_2(p) + \log_2(q)$, we instead use $\lceil \log_2(p) \rceil + \lceil \log_2(q) \rceil$, where we write $\lceil x \rceil$ for the smallest integer greater than or equal to *x*, then Eq. (11a) simplifies to

$$P(n) \approx \frac{1}{2^{n+2}} \ . \tag{11b}$$

Note that Eq. (11b) assigns a 25 % likelihood of achieving positive compression ($n = 0$).

Approximations of Values from Particle Physics

The role that information theory might play in revealing the accidental character of approximations is clarified by the following examples derived from particle physics.

According to 2002 CODATA (which is calculated by the National Institute of Standards), the experimental value of the muon-electron mass ratio is 206.7682838 with a one-standard-deviation uncertainty of ±0.0000054. The neutron-electron mass ratio equals 1838.6836598 with a one-standard-deviation uncertainty of ±0.0000013 (Mohr and Taylor, 2003). The following approximations were discovered by a brute-force computer search designed to find two approximations that reproduce these mass ratios to within four times their one-standard-deviation uncertainties, while sharing the same denominator:

$$\frac{M_{muon}}{M_{electron}} \approx \frac{596113}{2883} \tag{12a}$$

and

$$\frac{M_{neutron}}{M_{electron}} \approx \frac{5300925}{2883} \ . \tag{12b}$$

The number of bits of data they reproduce, with no allowance for over-fitting, is expressed by

$$\log_2\left(\frac{206.7682838}{4\times 0.0000054}\right) + \log_2\left(\frac{1838.6836598}{4\times 0.0000013}\right) \approx 51.6 \text{ bits}. \tag{12c}$$

Now compare this value against the number of bits of information that Eqs. (12a) and (12b) contain

$$\log_2(596113) + \log_2(5300925) + \log_2(2883) = 53.0 \text{ bits}. \tag{12d}$$

Note that in the above equation the expression $\log_2(2883)$ weights for both occurrences of 2883. Clearly, despite the economy achieved by their sharing the value 2883, and despite their having originated from a brute-force computer search, the approximations fail to accomplish any compression: the score produced is negative (51.6 – 53.0 = –1.4 bits).

Alternative numbers found via the same computer search do no better. So

$$\frac{M_{muon}}{M_{electron}} \approx \frac{608519}{2943} \tag{13a}$$

and

$$\frac{M_{neutron}}{M_{electron}} \approx \frac{5411246}{2943} \ . \tag{13b}$$

achieve a score of 51.6 – 53.1 = –1.5 bits, a value remarkably close to that achieved by Eqs. (12a) and (12b).

This illustrates what may perhaps be a general principle: If a result is random or coincidental, what it accomplishes may be nearly reproducible by other, unrelated means. That is to say the best and second-best alternatives may prove equally plausible, *with little to recommend one over the other, and nothing in common between them*. In contrast, should the best case stand apart from a cluster of alternatives, this in itself might be taken as evidence for a non-coincidental result.

Before moving on, it is interesting to compare the simple continued fractions of the particle mass ratios against those for the powers of π:

$$1838.6836598... = [1838, 1, 2, 6, 4, 1, 6, 1, ...] \ ,$$

$$206.7682838... = [206, 1, 3, 3, 5, 1, 15, 1, ...]$$

Unlike the continued fractions for the powers of π, these continued fractions offer no large partial quotients to exploit, which helps explain the failure of Eqs. (12a), (12b), (13a), and (13b) to achieve positive compression scores.

Evaluating Complex Expressions

For equations that are not simple ratios, difficult challenges are posed in assessing weight. We met this situation before with the π approximation $\left(\frac{2143}{22}\right)^{\frac{1}{4}}$. This approximation is not a rational approximation of π, but rather is the fourth root of a rational number. In assessing the overhead of this expression, presumably some weight must be assessed for the exponent $\frac{1}{4}$. We skirted this issue earlier by treating $\frac{2143}{22}$ as a rational approximation of π^4. In this way, the issue of what weight to apply to $\frac{1}{4}$ did not arise. In what follows, this evasion will no longer be possible.

Consider the following mass ratio equations, which also produce the above mass ratios to within four times their one-standard-deviation uncertainties (Markovitch, 2003):

$$\frac{M_\mu}{M_e} \approx 3\frac{\left(\frac{41}{10}\right)^3 - 10^{-3}}{1 - 4/10^5} \qquad (14a)$$

and

$$\frac{M_n}{M_e} \approx 3\frac{\left(\frac{41}{10}\right)^3 + 6\times\left(10^3+10\right)}{10-4/10^5} \quad . \tag{14b}$$

With no allowance for over-fitting, the number of bits of data they reproduce is exactly the same as before: 51.6. But it is not clear how to assess the overhead or weight of these complex equations, which make repeated use of the same numbers. What is needed is a new, more flexible, way to assess the weight of mathematical expressions.

Ideally, any such weighting method should lead to a probability equation as convenient as the empirical Eqs. (11a) and (11b). Accordingly, an attempt has been made to create a weighting method where the most compact approximation of a random real number will achieve *n* bits of compression with a probability of less than $\log_2\left(1+\frac{1}{2^n}\right)$, which follows Eq. (9l). As before, the approximation must achieve a predetermined degree of accuracy, with no allowance of score inflation from over-fitting. The hope is that such a probability can act as a guide in determining whether an approximation under study should be examined more closely to understand why it works. It is a key goal of this article to produce such a probability.

It is important to realize that the following weighting scheme need not produce exactly correct probabilities in order to be useful. In cases where the probabilities produced are very large, say 1 in 10,000 against coincidence, it matters little if the probability is in reality 1 in 1000, or even 1 in 100. In either case, a strong case against coincidence has been established.

A Weighting Method for Complex Expressions

The weighting method to be used is as follows:

1) Well known values, such as π and *e*, as well as operations such as *ln*(*x*), will be weighted 2 bits. Values that have already appeared once in an equation will also be weighted at most 2 bits for each additional appearance. An exception will be made for any expression that is used in an identical way in two separate equations, as are the denominators of Eqs. (12a) and (12b), in which case the second use will receive no weight. This is justified as the second use would disappear entirely if the two equations were combined into a single function.
2) All remaining values, excepting exponents, will be weighted by their length in binary digits plus 1. Accordingly, as the value 10 when written in base 2 has four binary digits 1010_2, it will be weighted 5. So the weight applied to *k* will be $\lceil \log_2(k) \rceil + 1$.
3) Exponents will be weighted by their length in binary digits. Furthermore, reused exponents will be weighted 1. So an exponent of -1 will be weighted 1 bit; an exponent of 4 will be weighted 3 bits; and an exponent of $\frac{1}{4}$ will be weighted 1 + 3 = 4 bits. So the weight applied to the *exponent k* will be $\lceil \log_2(k) \rceil$.

Testing the Weighting Method

The following examples will show that this method meets its goal of maintaining negligible compression for the mass equations already examined. We begin, however, by examining a π approximation from Simon Plouffe (see Pi, Weisstein, 1999). This approximation will demonstrate that a weighting of 2 bits for repeated terms is sufficient to discourage chance compression of π. The approximation is

$$\pi \approx \frac{\frac{689}{396}}{\ln\left(\frac{689}{396}\right)} \quad . \tag{15a}$$

And the closeness it achieves equals

$$\log_2 \left(\frac{\frac{\frac{689}{396}}{\ln\left(\frac{689}{396}\right)}}{\left| \pi - \frac{\frac{689}{396}}{\ln\left(\frac{689}{396}\right)} \right|} \right) \approx 25.7 \text{ bits}, \tag{15b}$$

while an audit of its overhead reveals

11 +	/ *for the first use of 689*
10 +	/ *for the first use of 396*
2 +	/ *for reuse of 689*
2 +	/ *for reuse of 396*
2 +	/ *for ln()*
= 27 bits.	(15c)

So despite reusing two large terms, under the above weighting scheme of 2 bits for repeated terms, the above approximation only achieves a score of 25.7 – 27 = –1.3 bits. The failure of

this approximation to achieve even a single bit of compression, even though there was no requirement of a predetermined degree of accuracy, provides evidence that the weighting scheme assigns sufficient weight to repeated terms, as well as operations such as *ln*().

Additional evidence that the weighting scheme is sufficiently conservative becomes apparent from examining the consequences of rewriting 2883 (from Eqs. (12a) and (12b)) as $2 \times 10^3 + 8 \times (10^2 + 10) + 3$. In this new form it would receive a weight of

3 +	/ *for 2*
5 + 2 +	/ *for the first use of 10 to the 3rd power*
5 +	/ *for 8*
2 + 2 +	/ *for reuse of 10 raised to the 2nd power*
2 +	/ *for reuse of 10*
3 +	/ *for 3*
= 24 bits,	(16a)

which is a much larger weight than $\log_2(2883) = 11.5$ bits. Such rewriting achieves no compression.

Similarly, if one tries to reduce the weighting for 5300925 (from Eq. (12b)) by rewriting it $2305^2 - 110^2$, it would receive a weight of

13 + 2 +	/ *for* 2305^2
8 + 1	/ *for* 110^2
= 24 bits,	(16b)

which is greater than $\log_2(5300925) = 22.3$ bits.

And, similarly, if 2943 from Eqs. (13a) and (13b) is rewritten as $3^3 + (2 \times 3^3)^2$:

3 + 2 +	/ *for the first use of* 3^3
3 +	/ *for 2*
2 +	/ *for reuse of 3 to the* 3^{rd} *power*
2	/ *for the exponent 2*

= 12 bits, (17a)

which achieves no compression, as $\log_2(2943) = 11.5$ bits.

It is true, however, that if one tries to reduce the weighting for 2883 by rewriting it as 3×31^2, it would receive a weight of

3 +	/ *for 3*
6 + 2	/ *for* 31^2

= 11 bits, (17b)

slightly smaller than $\log_2(2883) = 11.5$ bits, because it takes advantage of a large square present in 2883.

As it turns out, the new scoring method when applied to the above equations only leads to equations that still produce no compression, which gives some justification for regarding the above weighting scheme as conservative means of assessing complex expressions.

So the new score for Eqs. (12a) and (12b), when taking advantage of 3×31^2, is still negative ($51.6 - (\lceil \log_2(596113) \rceil + 1 + \lceil \log_2(5300295) \rceil + 1) - 11 = 51.6 - 56 = -4.4$ bits); while the score for Eqs. (13a) and (13b)) is also negative ($51.6 - (\lceil \log_2(608519) \rceil + 1 + \lceil \log_2(5411246) \rceil) - 12 = 51.6 - 57.0 = -5.4$ bits). The new scores for Eqs. (12a), (12b), (13a), and (13b) continue to suggest that they are purely coincidental, with little reason to prefer one set of mass equations over the other. Table IV summarizes these results. (Note: see Appendix A for the application of the above weighting technique to $\frac{103283}{32876}$ and a slightly better scoring version of $\left(\frac{2143}{22}\right)^{\frac{1}{4}}$.)

Evaluating the Mass Ratio Equations that Use 41 / 10

The above method may be used to audit the number of bits of overhead in the mass ratio Eqs. (14a) and (14b), which exploit the value 41 / 10:

3 +	*/ for 3*
7 + 5 + 2 +	*/ for (41 / 10)^3*
2 + 1 +	*/ for reuse of 10 raised to the –3rd*
4 +	*/ for 6*
2 + 1 + 2 +	*/ for (10^3 + 10)*
2 +	*/ for reuse of 10*
2 +	*/ for 1*
4 +	*/ for 4*

$$2 + 3 \qquad \text{/ for reuse of 10 raised to the } -5^{th}$$

$$= 42 \text{ bits.} \qquad (18)$$

Because 42 is significantly smaller than 51.6 (51.6 – 42 = 9.6 bits), this is clear cut evidence that Eqs. (14a) and (14b) are non-coincidental. Note that earlier, under this new scoring method, Eqs. (12a), (12b), (13a), and (13b) achieved no compression. Given that Eqs. (14a) and (14b) achieve 9.6 bits compression, this yields a probability of $\log_2\left(1+\frac{1}{2^{9.6}}\right)\approx\frac{1}{500}$ that such an approximation might occur by coincidence (this calculation makes use of Eq. (9l), the equation that the weighting scheme was designed to support).

The negative compression scores of Eqs. (12a), (12b), (13a), and (13b), and the high compression scores of Eqs. (14a) and (14b), lead to the following key question: Do the values, 596113, 5300925, and 2883 from Eqs. (12a) and (12b), or 608519, 5411246, and 2943 from Eqs. (13a) and (13b), or 41 / 10 from Eqs. (14a) and (14b), appear anywhere in the physics literature?

Not surprisingly, a search on the Internet reveals no physics papers indexed by the highly distinctive integers 596113, 5300925, 2883, 608519, 5411246, or 2943. However, a study of the physics literature does reveal that the values 41 and 10 do appear in some physics equations in the form of the coefficient $b_1 = 41 / 10$. This is remarkable, as the terms 41 and 10 are fairly distinctive, and one would not necessarily expect to find such idiosyncratic terms playing an important role in any physical equation relating to particle mass.

Specifically, b_1 figures prominently in the initial version of grand-unified theory (GUT), where it serves as a coefficient that helps regulate how the strength of electromagnetic force

varies with distance (Georgi and Glashow, 1974; Georgi, Quinn, and Weinberg, 1974; Dimopoulos, Raby, and Wilczek, 1991; Zwirner, 1992), and thereby affects a particle's mass through self-interaction (Feynman, 1985). As it is easy to simplify Eqs. (14a) and (14b) by using b_1^3 in place of powers of 41 and 10, the option of using b_1 reinforces the possibility that a physical explanation lies waiting to be discovered behind the mass ratio Eqs. (14a) and (14b). (See Endnote for how the fine structure constant inverse, an important value affecting the strength of electromagnetic force and therefore particle self-interaction, also may be neatly described with the aid of powers of 10.)

Evaluating the Mass Ratio Equations That Use b_1

Simplified with the aid of b_1^3, Eqs. (14a) and (14b) become

$$\frac{M_\mu}{M_e} \approx 3\frac{b_1^3 - 10^{-3}}{1 - 4/10^5} \tag{19a}$$

and

$$\frac{M_n}{M_e} \approx 3\frac{b_1^3 + 6\times\left(10^3 + 10\right)}{10 - 4/10^5} \quad . \tag{19b}$$

As before, the two equations differ chiefly in the second term of their numerators. Because other grand-unified theories assign different values to b_1, it will be weighted 3 bits. An audit of the number of bits of information represented by Eqs. (19a) and (19b) yields:

3 + / *for 3*

$$
\begin{aligned}
&3 + 2 + && / \textit{ for } b_1^3 \\
&5 + 1 + && / \textit{ for } 10^{-3} \\
&4 + && / \textit{ for 6} \\
&2 + 1 + 2 + && / \textit{ for } (10^3 + 10) \\
&2 + && / \textit{ for reuse of 10} \\
&2 + && / \textit{ for 1} \\
&4 + && / \textit{ for 4} \\
&2 + 3 && / \textit{ for reuse of 10 raised to the } -5^{th} \\
&= 36 \text{ bits.} && (20)
\end{aligned}
$$

The value 36 is significantly smaller than 51.6 (51.6 – 36 = 15.6 bits). This compression of 15.6 bits is considerably greater than the 9.6 bits of compression achieved by Eqs. (14a) and (14b), and vastly more than the negative compression achieved by Eqs. (12a), (12b), (13a), and (13b). This underscores the possibility that Eqs. (19a) and (19b) may be more than merely handy aids to memory: they may have a physical origin. (See Table IV for a summary of these results.)

But how should one regard the apparent probability of $\log_2\left(1+\frac{1}{2^{15.6}}\right)$, roughly 1 in 34,000, that Eqs. (19a) and (19b) are coincidental? Although the complex weighting scheme just applied was devised to meet the goal of producing less than a $\log_2\left(1+\frac{1}{2^n}\right)$ probability of achieving *n* bits of compression by chance, there can be no guarantee that it succeeds. This issue of how seriously one should take this probability is unclear, partly because the weighting

for the above equations is more complex than that applied to the earlier 1 ppb π approximation, and partly because there are literally an infinite number of ways of constructing an approximation. Despite these uncertainties, the probability of 1 in 34,000 is so large that, even allowing for significant error in its calculation, it must strongly suggest a physical, rather than coincidental, origin for effectiveness of b_1. (See Appendix B for the results of a computer-driven search for approximations of randomly-generated numbers, where the program attempts to find approximations that do better than the above weighting scheme is designed to allow. And see Appendix C for a versions of the mass equations that treat mass as a membrane stretched over the surface. These equations, because of their symmetry, are arguably simpler than Eqs. (19a) and (19b). Appendix D contains the source code for the above program.)

Lastly, how should we regard the decision to assign the expression b_1 a weight of just $\log_2(8) = 3$? Is this done on the assumption that there are a total of 8 plausible values that might be used in its place? Or is it to be justified by saying that, for a physicist familiar with grand-unified theory, the "psychological overhead" of remembering b_1 is just 3 bits, 1 bit more than was assigned to π?

This latter type of reasoning appears to follow Ernst Mach's concept of "economy of thought". For Mach, theories partly serve the *psychological* function of summarizing and condensing data. Whether or not an equation meets Mach's goal therefore depends on the mind of the person involved; for a GUT physicist for whom b_1 is already a "given", presumably the psychological overhead of memorizing and recalling 41 / 10 is significantly less than the burden imposed by 41 / 10 for the layman, who necessarily lacks this prior knowledge.

Conclusion

Finally, what are we to make of the original question regarding Eq. (1)? Is it coincidental? We can calculate the information content of Eq. (1) by using the techniques outlined above. First, note that

$$\pi \approx \ln\left(\pi + 2 \times 10 - \left(3/10^2\right)^2 \right) \tag{21a}$$

and

$$\log_2 \left(\frac{\ln\left(\pi + 2 \times 10 - \left(3/10^2\right)^2 \right)}{\left| \ln\left(\pi + 2 \times 10 - \left(3/10^2\right)^2 \right) - \pi \right|} \right) \approx 31.7 \text{ bits.} \tag{21b}$$

An audit of the information content of Eq. (21a) reveals that

2 +	*/ for ln()*
2 +	*/ for π*
3 +	*/ for 2*
5 +	*/ for the first use of 10*
3 + 2 + 2 + 1	*/ for (3 / 10^2)^2*
= 20 bits.	(23c)

The value 20 is far less than 31.7 (31.7 – 20 = 11.7 bits). The above compression score, while clearly suggesting non-coincidence, does little to shed light on the "why" of Eq. (21a). But perhaps one day this mystery will be resolved.

Of course, estimation is involved in saying that the informational content of π and *ln()* warrants a value of $\log_2(4) = 2$ bits, which is the same weight given to 3. But how else is one to proceed? One obvious alternative is to turn the problem over to psychologists who could experimentally decide which is easier to memorize: π or 3? a particular approximation, or the digits that approximation faithfully reproduces? Ernst Mach, with his principle of "economy of thought", might approve, as simplicity, like beauty, must ultimately lie in the eye of the beholder.

Endnote

Powers of 10 may also be used to reproduce neatly at least one other well-known physical constant, the fine structure constant α, which within the limits of error equals 1/137.036 (Mohr and Taylor, 2003; Feynman, 1985). This value may be elegantly reproduced as follows

$$\frac{1}{\alpha} \approx 137.036 = \frac{10^3 - 10^{-3}}{3^3} + 10^2 - 10^{-3} = \frac{999.999}{3^3} + 99.999 \quad . \tag{24}$$

Acknowledgements

The author wishes to thank Joe Mazur, C. Y. Lo, and Erik Ramburg for their useful communications and discussions and J. Parrillo for his diligent checking of the manuscript.

References

Conway, J. H. and Guy, R. K., *The Book of Numbers* (Springer-Verlag, New York, 1996), pp. 186-187, 217-226.

Dienes, K. R., Dudas, E., and T. Gherghetta, Nucl. Phys. B, **537**, 47 (1999).

Dimopoulos, S., Raby S., and Wilczek, F., Physics Today, **44**, 25 (1991).

Feynman, R. P., *QED* (Princeton University Press, Princeton,1985), pp. 126-130.

Georgi, H. and Glashow, S. L., Phys. Rev. Lett., **32**, 438 (1974).

Georgi, H., Quinn, H. R., and Weinberg, S., Phys. Rev. Lett., **33**, 451 (1974).

Hardy, G. H. and Wright, E. M., *An Introduction to the Theory of Numbers*, 5th ed. (Oxford, England: Clarendon Press, 1979), pp. 154-170.

Khinchin, A., *Continued Fractions*, (New York, Dover, 1997), pp. 92-93.

Mach, E., *Science of Mechanics* (Open Court, La Salle, Illinois,1988), pp. 6-8, 577-595.

Markovitch, J. S., A Precise, Particle Mass Formula Using Beta-Coefficients From a Higher-Dimensional, Nonsupersymmetric GUT, available at www.slac.stanford.edu/spires/find/hep/www?r=apri-ph-2003-11 (Applied and Pure Research Institute, Nashua, NH, APRI-PH-2003-11, 2003).

Mohr, P. J. and Taylor, B. N., "The 2002 CODATA Recommended Values of the Fundamental Physical Constants, Web Version 4.0," available at physics.nist.gov/constants (National Institute of Standards and Technology, Gaithersburg, MD 20899, 9 December 2003).

Ramanujan, S., “Modular Equations and Approximations to π”, *Quart. J. Pure and Appl. Math*., 45, 350-372, 1914.

Weisstein, E. W., *CRC Concise Encyclopedia of Mathematics* (CRC Press, New York, 1999).

Zwirner, F. Properties of SUSY particles, Erice 1992, hep-ph/9307293.

Table I. All integer ratio approximations of $\phi = \left(1+\sqrt{5}\right)/2$, where the compression as defined by Eq. (8a) is positive, and the denominator is less than 10,000. The numerators and denominators in columns 1 and 2 turn out to be the Fibonacci numbers. For the larger Fibonacci numbers the compression achieved in column 4 is close to $\log_2\left(\sqrt{5}\right) = 1.16096404\ldots$.

Numerator	*Denominator*	*Approximation of Golden Ratio*	*Compression in Binary Digits*
2	1	2	1.388483882
3	2	1.5	1.082725763
5	3	1.6666667	1.192005157
8	5	1.6000000	1.149281502
13	8	1.6250000	1.165451527
21	13	1.6153846	1.159253716
34	21	1.6190476	1.161617875
55	34	1.6176471	1.160714388
89	55	1.6181818	1.161059380
144	89	1.6179775	1.160927653
233	144	1.6180556	1.160977960
377	233	1.6180258	1.160958767
610	377	1.6180371	1.160966039
987	610	1.6180328	1.160963297
1597	987	1.6180344	1.160964370
2584	1597	1.6180338	1.160964012
4181	2584	1.6180341	1.160964012
6765	4181	1.6180340	1.160964251
10946	6765	1.6180340	1.160963297

Table II. All integer ratio approximations of *e*, where the compression as defined by Eq. (8a) is positive, and the denominator is less than 10,000.

Numerator	*Denominator*	*Approximation of e*	*Compression in Binary Digits*
3	1	3	1.827675462
5	2	2.5	0.195736066
8	3	2.6666667	1.106136322
11	4	2.7500000	0.978546560
19	7	2.7142857	2.352476597
38	14	*same as above*	0.352476627
87	32	2.7187500	1.060675144
106	39	2.7179487	0.980907083
193	71	2.7183099	2.823138714
386	142	*same as above*	0.823138654
1264	465	2.7182796	1.033987284
1457	536	2.7182836	0.989045382
2721	1001	2.7182817	3.179216623
5442	2002	*same as above*	1.179216743
8163	3003	*same as above*	0.009291716
23225	8544	2.7182818	1.021727085
25946	9545	2.7182818	0.992906749

Table III. All integer ratio approximations of π, where the compression as defined by Eq. (8a) is positive, and the denominator is less than 10,000. In column 4, all values that achieve at least a single decimal digit of compression (about 3.32 bits) are shaded.

Numerator	*Denominator*	*Approximation of π*	*Compression in Binary Digits*
3	1	3	2.8867153
6	2	*same as above*	0.8867153
19	6	3.1666667	0.1362694
22	7	3.1428571	4.0119391
44	14	*same as above*	2.0119391
66	21	*same as above*	0.8420141
88	28	*same as above*	0.0119391
333	106	3.1415094	0.0969140
355	113	3.1415929	8.1975736
710	226	*same as above*	6.1975736
1065	339	*same as above*	5.0276486
1420	452	*same as above*	4.1975736
1775	565	*same as above*	3.5537174
2130	678	*same as above*	3.0276486
2485	791	*same as above*	2.5828638
2840	904	*same as above*	2.1975736
3195	1017	*same as above*	1.8577236
3550	1130	*same as above*	1.5537174
3905	1243	*same as above*	1.2787104
4260	1356	*same as above*	1.0276486
4615	1469	*same as above*	0.7966942
4970	1582	*same as above*	0.5828638
5325	1695	*same as above*	0.3837924
5680	1808	*same as above*	0.1975736
6035	1921	*same as above*	0.0226480

Table IV. Compression scores achieved for various approximations in descending order of bits compressed. Rows representing apparently chance compression appear shaded. All scores are calculated using the weighting method designed to handle complex expressions.

Approximation	*Compression (without over-fitting)*	*Compression (allowing over-fitting)*
Mass Equations (19a) and (19b) using b_1	15.6 bits	—
$\pi \approx \ln\left(\pi + 20 - \left(3/10^2\right)^2\right)$	—	11.7 bits
$\pi \approx \left(101 + 1 - \frac{101}{22}\right)^{\frac{1}{4}}$	—	10.5 bits
Mass Equations (14a) and (14b) without b_1	9.6 bits	—
$\pi \approx \frac{\frac{689}{396}}{\ln\left(\frac{689}{396}\right)}$	—	–1.3 bits
Mass Equations (12a) and (12b) using 3×31^2	–4.4 bits	—
$\pi \approx \frac{103283}{32876}$	–5.1 bits *(to 1 ppb)*	–4.7 bits
Mass Equations (13a) and (13b) using $3^3 + (2 \times 3^3)^2$	–5.4 bits	—

Figure 1. The role of continued fractions in finding effective rational approximations (Conway and Guy, 1993).

A formula called *payout* is supplied by me

$$payout(R, p, q) = \frac{p \times q \times \left| R - \frac{p}{q} \right|}{\frac{p}{q}} = q^2 \times \left| R - \frac{p}{q} \right| \quad .$$

I agree to surrender to you payout(R ,p ,q) × 1000 dollars under the following terms:

You must first supply the value R, a real number greater than 0.

I then get to choose p and q, which must be positive integers. Only then will all three values be plugged into the formula to determine the payout to you. I am free to choose whatever values minimize my payout to you, and consequently minimize your profit.

Question: What value should you choose for R, given that you know in advance that I will choose p and q so as to approximately equal R, and thereby reduce my payout to the lowest possible amount?

Answer: It appears that you should choose the golden ratio $\phi = \frac{1+\sqrt{5}}{2}$. Then, in order to minimize my payout to you, I should choose 2 and 1 for p and q. You will then get paid exactly $1^2 \times \left| \frac{1+\sqrt{5}}{2} - \frac{2}{1} \right| \times 1000$ dollars, or about $381.96.

Why the golden ratio? The answer lies in the golden ratio expressed as a continued fraction

$$\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}} \quad .$$

Effective rational approximations arise whenever large terms appear in the continued fraction for a value. For the golden ratio, these terms are the lowest possible–they always equal 1. For this reason, when you select the golden ratio, I find it difficult to reduce my payout. Had you chosen π, however, then by my letting p and q equal 355 and 113, you would collect just $3.40.

Figure 2. Data from Column 4 of Tables I-III, representing all approximations of *e*, π, and ϕ that achieve positive compression as defined by Eq. (8a), and which have denominators less than 10,000.

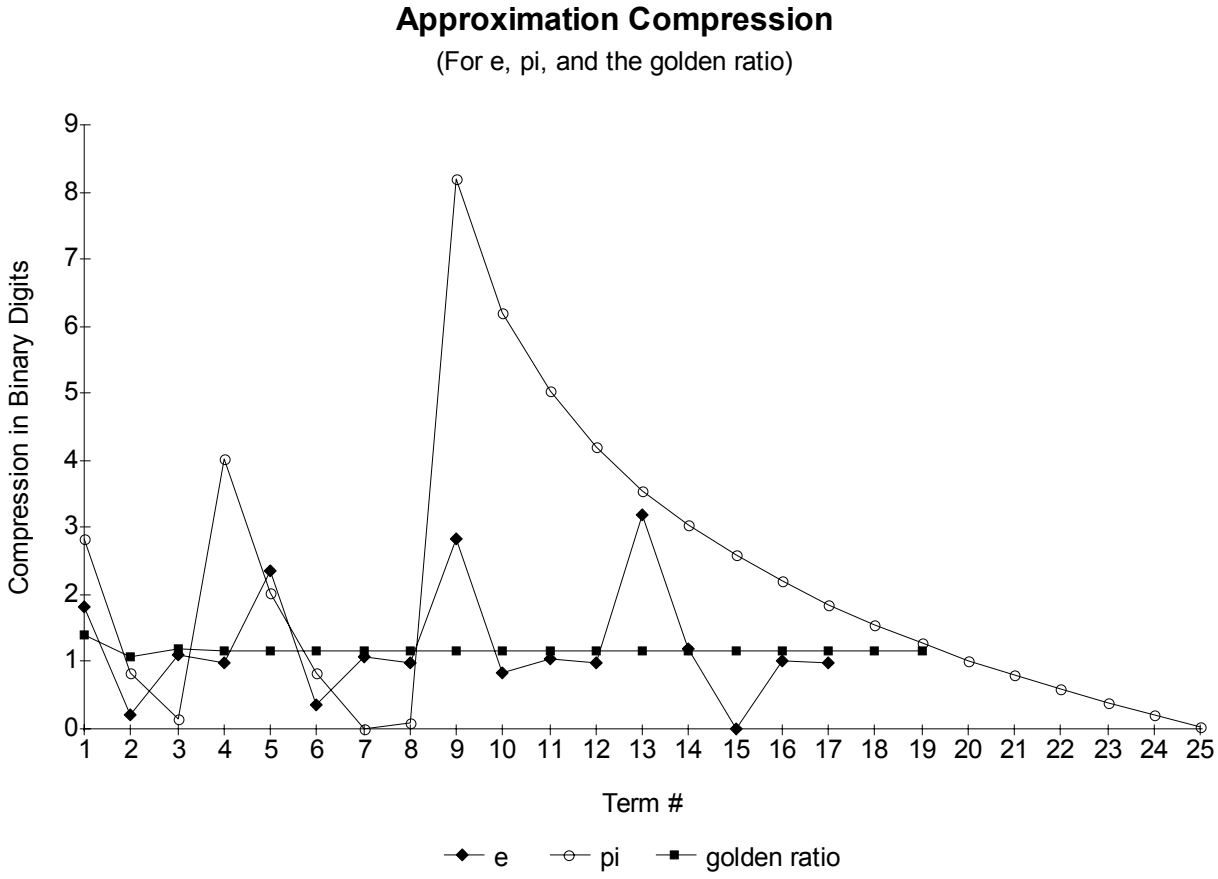


Figure 3. Data from Column 4 of Table I, representing all approximations of ϕ that achieve positive compression as defined by Eq. (8a), and which have denominators less than 10,000. The highest compression (about 1.3884… bits) is achieved by approximating the golden ratio with 2 / 1 (see below). Other compression values are achieved by the use of ratios of the Fibonacci numbers.

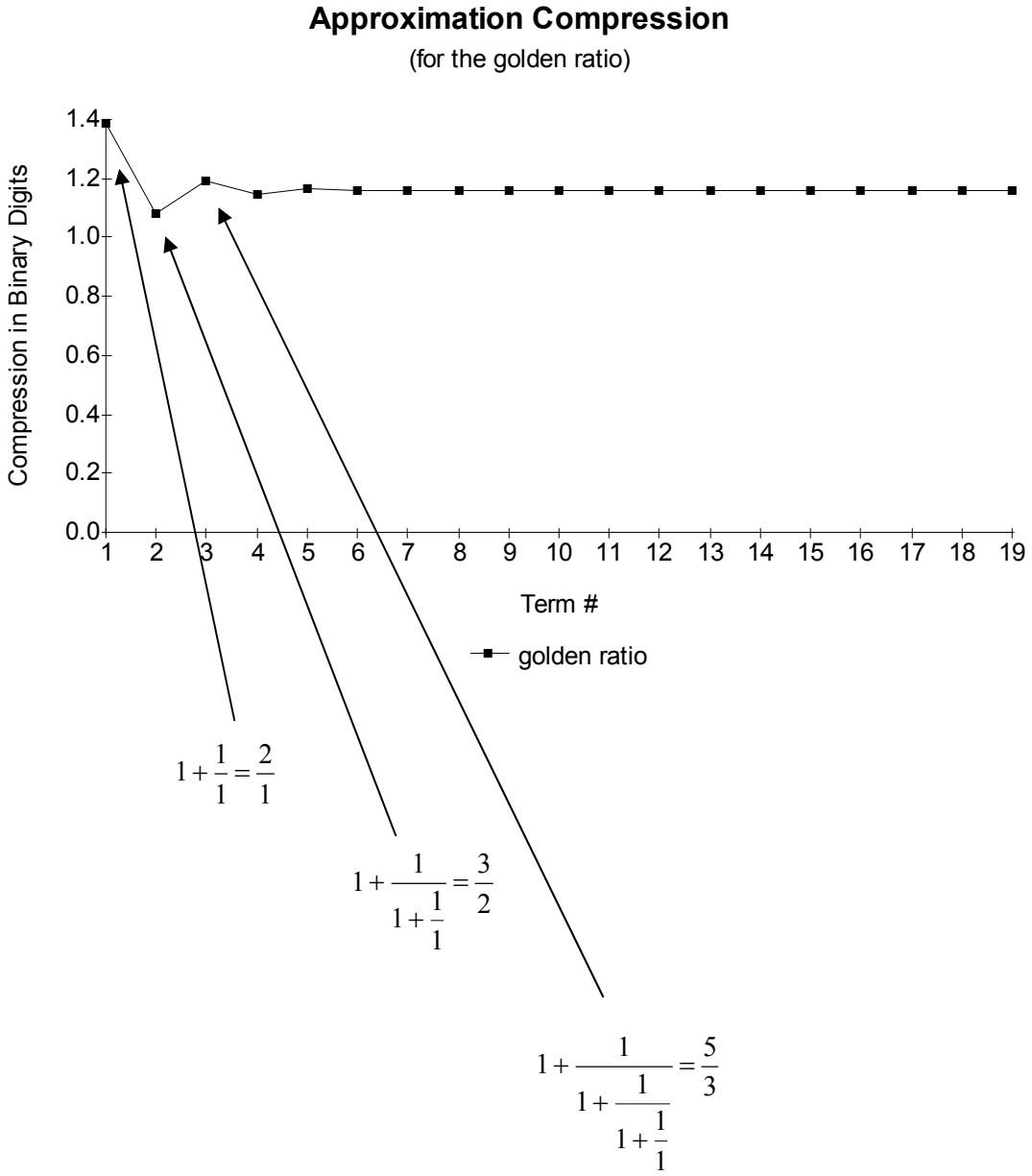


Figure 4. Data from Column 4 of Table II, representing all approximations of *e* that achieve positive compression as defined by Eq. (8a), and which have denominators less than 10,000. The pattern formed by successive peaks in the graph is not coincidental. The terms of the simple continued fraction for *e* form the sequence 2, 1, 2, 1, 1, 4, 1, 1, 6, …, and an effective approximation for *e* is achieved by truncating the fraction just before a large term.

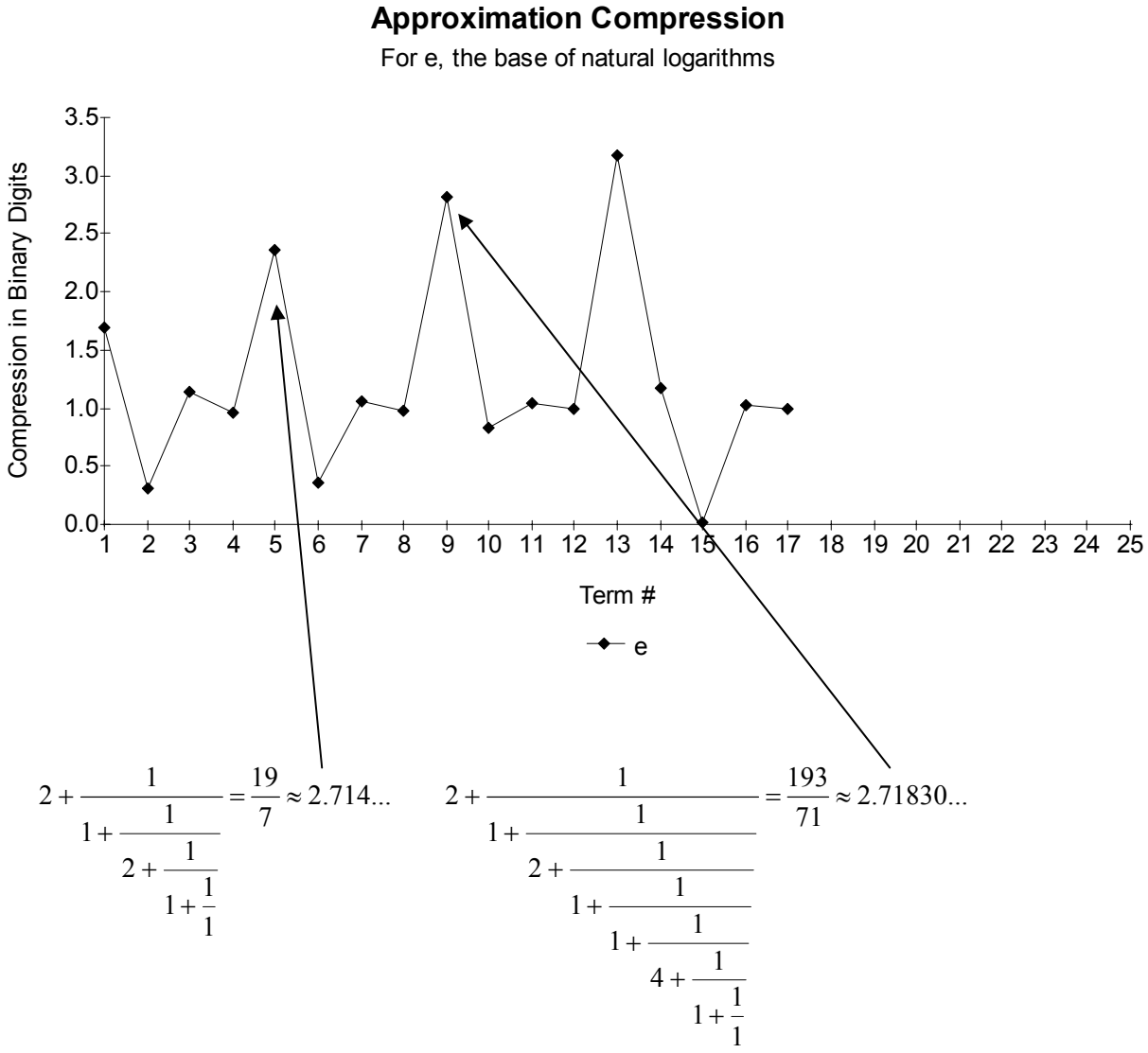
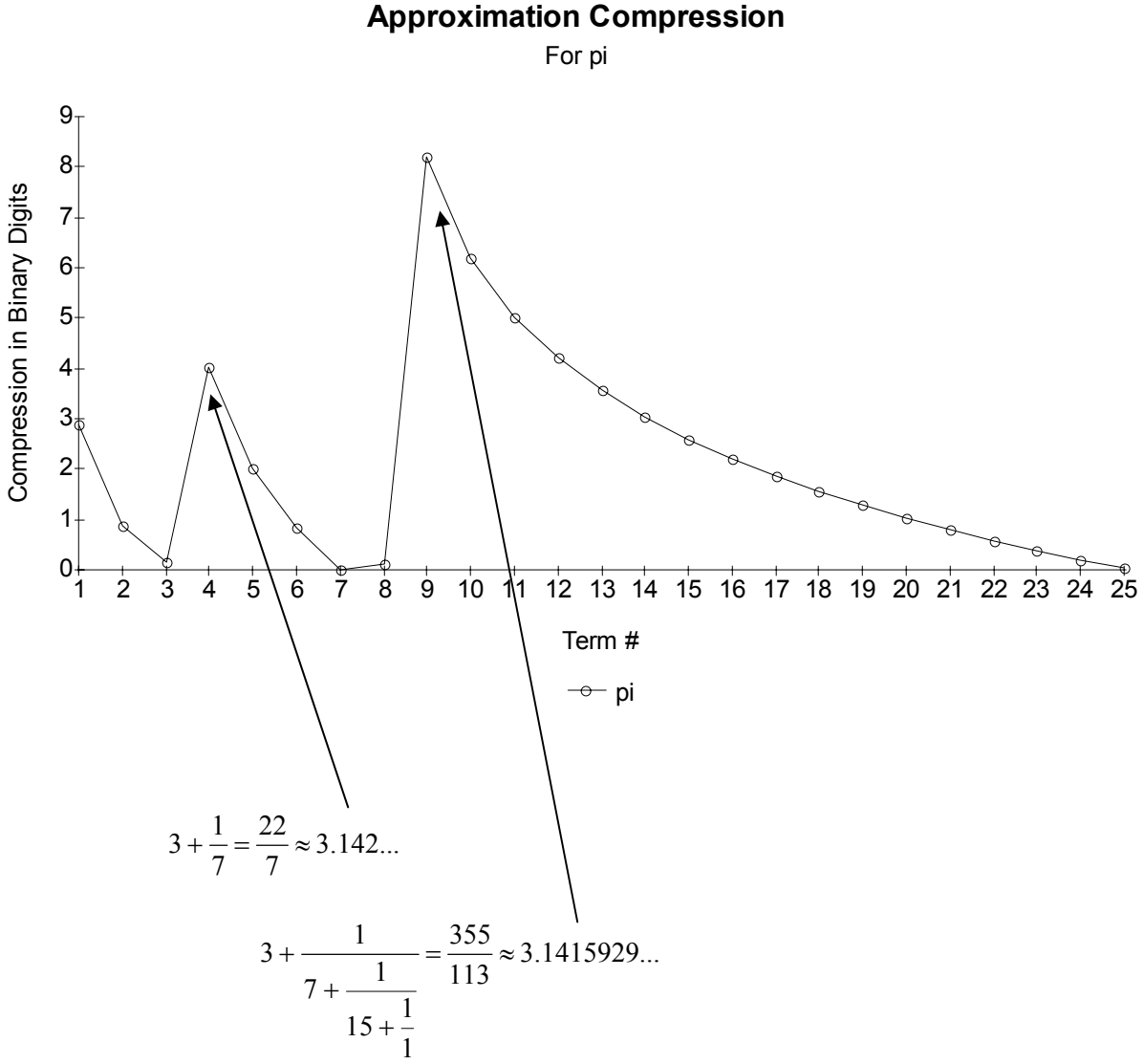


Figure 5. Data from Column 4 of Table III, representing all approximations of π that achieve positive compression as defined by Eq. (8a), and which have denominators less than 10,000. The pattern formed by successive peaks in the graph appears to be coincidental, as the terms of the simple continued fraction for π form the chaotic sequence 3, 7, 15, 1, 292, 1, 1, 2, 1, 3 …. An effective approximation for π is achieved by truncating the fraction just before a large term. The large peak in the center of the graph occurs because π 's continued fraction may be truncated just before the term 1 / 292, thereby yielding the extremely good approximation 355 / 113.



Appendix A: Two Examples of the Weighting Scheme

It is possible to gain some perspective on the weighting method for complex equations by revisiting the earlier attempt to find a 1 ppb π approximation that achieves positive compression.

Note that

$$\left(\frac{2143}{22}\right)^{\frac{1}{4}} = \left(101+1-\frac{101}{22}\right)^{\frac{1}{4}} = \underline{3.14159265}258\ldots \qquad \text{(A1)}$$

is a variant of Eq. (9h). This rewriting will allow a slightly higher compression. Like Eq. (10a), Eq. (A1) produces π within 1 ppb.

An audit of its overhead yields

8 +	*/ for the first use of 101*
2 +	*/ for 1*
2 +	*/ for reuse of 101*
5 +	*/ for 22*
1 + 3	*/ for the exponent 1 / 4*
= 21 bits.	(A2)

With over-fitting allowed this produces a score that is overwhelmingly positive (31.5 – 21 = 10.5 bits).

Lastly, with the above weighting method the nine-digit π approximation's compression score produced by Eq. (10a) is substantially worse (29.9 – 35 = –5.1 bits. With over-fitting allowed its new score is 30.3 – 35 = –4.7 bits).

Appendix B: Computer-Driven Test of Weighting Scheme

A computer program employed the weighting/probability scheme that was used to create Table IV to search for effective approximations of real numbers randomly-generated in the interval (0,1). When reduced to their simplest form (a reduction that the program does not carry out), the following expressions achieve the compression scores indicated. The scores are within the margin allowed for by the weighting/probability scheme, but there remains the possibility that a computer program that carried out a more comprehensive search–which is certainly possible–might be able to achieve scores higher than the scheme is supposed to allow.

The approximations produced were required to achieve 16 bit accuracy. Proper allowance was made in the scoring for repeated terms and opportunities for reduction. The following is the raw output from the program:

```
SEED=[104] COMP=[16] VAL MAX=[25] EXPS MAX=[6,3] VAL WT=[1,2,2] EXP WT=[0,1,1]
0.97061144 = {[ 1^1 - ( 5/9 )^6] /  1^1}^1: Compression = 16 - 14 = +2 bits
0.90962267 = {[12^1 + (12/14)^2] / 14^1}^1: Compression = 16 - 16 = +0 bits
0.76040668 = {[ 3^1 + ( 1/24)^1] /  4^1}^1: Compression = 16 - 14 = +2 bits
0.22610559 = { 5^5 / [24^3 - ( 3/1 )^1]}^1: Compression = 16 - 18 = -2 bits
0.44202356 = { 2^2 / [ 9^1 + ( 2/9 )^2]}^1: Compression = 16 - 15 = +1 bits
0.06997305 = { 7^1 / [10^2 + ( 1/3 )^3]}^1: Compression = 16 - 16 = +0 bits
0.53631785 = {[ 7^1 - ( 1/6 )^2] / 13^1}^1: Compression = 16 - 15 = +1 bits
0.87684198 = {11^1 / [11^1 + (12/11)^5]}^1: Compression = 16 - 17 = -1 bits
0.99360318 = {[ 1^1 - ( 2/25)^2] /  1^1}^1: Compression = 16 - 13 = +3 bits
0.74868211 = {[ 3^1 - ( 4/23)^3] /  4^1}^1: Compression = 16 - 17 = -1 bits
0.10174560 = { 5^1 / [ 7^2 + ( 1/7 )^1]}^1: Compression = 16 - 13 = +3 bits
0.10696970 = {[ 7^1 - ( 2/13)^1] /  2^6}^1: Compression = 16 - 17 = -1 bits
0.15502914 = { 1^1 / [ 2^1 + ( 6/7 )^4]}^2: Compression = 16 - 16 = +0 bits
0.87812215 = {[ 2^2 + ( 5/8 )^2] /  5^1}^1: Compression = 16 - 17 = -1 bits
0.30410115 = {[ 4^1 + ( 4/7 )^6] /  6^1}^3: Compression = 16 - 19 = -3 bits
0.47901246 = {[13^1 - ( 1/15)^1] /  3^3}^1: Compression = 16 - 16 = +0 bits
0.82098128 = {[ 1^1 - ( 5/11)^3] /  1^1}^2: Compression = 16 - 15 = +1 bits
0.00341615 = { 6^1 / [ 5^1 + ( 5/2 )^5]}^2: Compression = 16 - 18 = -2 bits
0.49146059 = {12^1 / [ 5^2 - ( 7/12)^1]}^1: Compression = 16 - 17 = -1 bits
0.99462779 = { 6^4 / [ 6^4 + ( 7/1 )^1]}^1: Compression = 16 - 14 = +2 bits
Percent Above One = 25.00 Highest Compression = +3.00...Done...
```

Appendix C: Toroidal Versions of the Mass Equations

One way to interpret particle mass is as a membrane stretched over the surface of one or more toruses, with mass proportional to the surface area of these toruses. Toroidal versions of Eqs. (19a) and (19b) are particularly interesting as the torus is a candidate for the shape of the extra dimensions of space in string theory, and, in some versions of string theory, a membrane stretched over a torus is a candidate for mass.

A torus is uniquely specified by its outer radius R, and its inner radius r measured from the torus's center point. Its surface area is then equal to

$$S(R,r) = \pi^2 (R + r)(R - r) = \pi^2 \left(R^2 - r^2\right) . \tag{C1}$$

Now the Eqs. (19a) and (19b) may be rewritten to employ the function $S(R,r)$. In these equations the values $b_1 = \frac{41}{10}$ and $\tilde{b}_1 = \frac{1}{10}$ are the beta coefficients of the extra-dimensional GUT described by Dienes, Dudas, and Gherghetta, 1998. The added use of $\tilde{b}_1 = \frac{1}{10}$ allows the muon-electron mass ratio equation in particular to achieve an interesting symmetry.

$$\frac{M_{\mu}}{M_e} = \frac{S\left(\frac{b_1^{\frac{3}{2}}}{\frac{1}{\sqrt{2}}}, \frac{b_1^{\frac{3}{2}}}{\sqrt{2}}\right) + S\left(\frac{\tilde{b}_1^{\frac{3}{2}}}{\sqrt{2}}, \frac{\tilde{b}_1^{\frac{3}{2}}}{\frac{1}{\sqrt{2}}}\right)}{S\left(\frac{1}{\sqrt{2}}, \sqrt{2}\tilde{b}_1^{\frac{5}{2}}\right)} \tag{C2}$$

$$\frac{M_n}{M_e} = \frac{S\left(\frac{b_1^{\frac{3}{2}}}{\frac{1}{\sqrt{2}}}, \frac{b_1^{\frac{3}{2}}}{\sqrt{2}}\right) + S\left(\frac{1}{\tilde{b}_1^{\frac{4}{2}}}, \frac{1}{\tilde{b}_1^{\frac{3}{2}}}\right) + S\left(\frac{1}{\tilde{b}_1^{\frac{2}{2}}}, \frac{1}{\tilde{b}_1^{\frac{1}{2}}}\right)}{S\left(\frac{1}{\sqrt{2}\tilde{b}_1^{\frac{1}{2}}}, \sqrt{2}\tilde{b}_1^{\frac{5}{2}}\right)} \tag{C3}$$

Appendix D: Source Code for Program to Find Approximations

```
#include "stdafx.h"

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <limits.h>
#include <ctype.h>
#include <float.h>
#include <errno.h>
#include <signal.h>
#include <locale.h>
#include <setjmp.h>
#include <stdarg.h>

#ifdef DOCUMENTATION

/*---------------------------------------------------------------------------*/
//
//  Written by J. S. Markovitch
//
//  Copyright (c) 2004  J. S. Markovitch  All rights Reserved.
//
// find best (most compact) approximation of a number in the form:
//
// {[ dwFraction1Numer^a + dwSign * ( dwFraction2Numer / dwFraction2Denom )^c ] / dwFraction1Denom^b }^d
//
// or
//
// { dwFraction1Numer^a / [ dwFraction1Denom^b + dwSign * ( dwFraction2Numer / dwFraction2Denom )^c ]}^d
//
/*---------------------------------------------------------------------------*/

#endif

/*---------------------------------------------------------------------------*/

#define TOTAL_TRIALS                                    20

/*---------------------------------------------------------------------------*/
```

```
#define MAX_BITS_COMPRESSION_START		16.0
#define MAX_BITS_COMPRESSION_END		16.0

#define MAX_VALUE				25

#define MAX_POWER				6
#define MAX_OVERALL_POWER			3

//------------------------- WEIGHTING METHOD ---------------------------

// NON-EXPONENTS

#define WEIGHT_ADDED_FOR_FIRST_USE		1
#define WEIGHT_FOR_REUSE			2
#define WEIGHT_FOR_ONE				2

// EXPONENTS

#define WEIGHT_ADDED_FOR_EXPONENT		0
#define WEIGHT_FOR_EXPONENT_REUSE		1
#define WEIGHT_FOR_NEG_ONE_EXPONENT		1

/*-------------------------------------------------------------------------*/
// speed things up by skipping cases where scores must be negative?

#define SPEED_SEARCH				0
#define SPEED_SEARCH_THRESHOLD		(-0.0)

/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

void SetRandom( unsigned long  ulValue );
double GetRandom( void );
double GetRandomTestValue();

/*-------------------------------------------------------------------------*/

double SearchForApproximation( double dwMaxBitsCompression, double dwTestValue, double dwNumer1Limit, double dwDenom1Limit, double dwNumer2Limit, double dwDenom2Limit );

double NumberOfBits( double dwVal );
double NumberWeight( double dwVal );

/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

int main(int argc,char *argv[])
{
```

```
	double dwCompression = 0.0, dwMaxBitsCompression;
	double nTotalTrials = TOTAL_TRIALS;
	double dwAboveOne = 0.0, dwHighestCompression = 0.0;
	short nSeed;
	time_t ltime;

/*---------------------------------------------------------------------------*/

	time( &ltime );
	//nSeed = (int) (ltime & 0xff);
	nSeed = 104;
	SetRandom( (unsigned long) nSeed );

	if( SPEED_SEARCH )
		printf("SPEED_SEARCH on (so the best score may not be found if it is less than %1.0f.)\n",
			(float) SPEED_SEARCH_THRESHOLD );

//---------------------------------------------------------------------------


	dwAboveOne = 0.0;
	dwHighestCompression = -100000.0;

	for ( dwMaxBitsCompression = MAX_BITS_COMPRESSION_START; dwMaxBitsCompression <= MAX_BITS_COMPRESSION_END; dwMaxBitsCompression += 2 )
	{
		printf("SEED=[%d] COMP=[%1.0f] VAL MAX=[%1.0f] EXPS MAX=[%1.0f,%1.0f] VAL WT=[%1.0f,%1.0f,%1.0f] EXP WT=[%1.0f,%1.0f,%1.0f]\n",

			(int)nSeed,

			(float) dwMaxBitsCompression,

			(float) MAX_VALUE,

			(float) MAX_POWER,
			(float) MAX_OVERALL_POWER,

			(float) WEIGHT_ADDED_FOR_FIRST_USE,
			(float) WEIGHT_FOR_REUSE,
			(float) WEIGHT_FOR_ONE,

			(float) WEIGHT_ADDED_FOR_EXPONENT,
			(float) WEIGHT_FOR_EXPONENT_REUSE,
			(float) WEIGHT_FOR_NEG_ONE_EXPONENT
			);

		for( nTotalTrials = 0; nTotalTrials < TOTAL_TRIALS; nTotalTrials++ )
		{
			dwCompression = SearchForApproximation( dwMaxBitsCompression, GetRandomTestValue(), MAX_VALUE, MAX_VALUE, MAX_VALUE, MAX_VALUE );
```

```
                        if( dwCompression > dwHighestCompression )
                                dwHighestCompression = dwCompression;
                        if( dwCompression > 1.000001 )
                                dwAboveOne++;
                }
        }

        printf( "Percent Above One = %5.2f Highest Compression = %+5.2f...Done...\n" ,
                (float) (100.0 * ( dwAboveOne / TOTAL_TRIALS )),
                (float) dwHighestCompression
                );

//-------------------------------------------------------------------------
//-------------------------------------------------------------------------
//-------------------------------------------------------------------------

        return 0;
}

/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/
/*-------------------------------------------------------------------------*/

double SearchForApproximation( double dwMaxBitsCompression, double dwTestValue, double dwNumer1Limit, double dwDenom1Limit , double dwNumer2Limit, double dwDenom2Limit )
{
        double dwNumer1, dwNumer2, dwDenom2, dwDenom1, dwNumer1Power, dwNumer2Power, dwDenom1Power, dwDenom2Power, dwSign;
        double dwBestNumer1 = 0.0, dwBestNumer2 = 0.0, dwBestDenom2 = 0.0, dwBestDenom1 = 0.0, dwBestSign = 0.0;
        double dwFraction1Numer, dwFraction1Denom, dwFraction2Numer, dwFraction2Denom;
        double dwTmp, dwApproxN, dwApproxD, dwError, dwErrorN, dwErrorD, dwCompression;
        double dwBestCompression = -100000.0;

        int    nTmp, nNumer1Power, nDenom1Power, nFraction2Power, nOverallPower;
        int    nBestNumer1Power = 0, nBestDenom1Power = 0, nBestFraction2Power = 0, nBestOverallPower = 0;

        bool   bRecipricalInNumer, bBestRecipricalInNumer, bFraction2IsFraction;
        bool   bDenom1PowerIsDuplicate, bFraction2PowerIsDuplicate, bOverallPowerIsDuplicate;


        // ------------------------------------------------------------------------------------
        // --------Outer part of nested for loop-----------------------------------------------
        // ------------------------------------------------------------------------------------

        // --------Loop thru all possibilities
        // --------Must do powers last, as they use variables from earlier loops

        for( dwNumer1 = 1.0;  dwNumer1 <= dwNumer1Limit; dwNumer1++ )
        for( dwDenom1 = 1.0;  dwDenom1 <= dwDenom1Limit; dwDenom1++ )
        for( dwNumer2 = 0.0;  dwNumer2 <= dwNumer2Limit; dwNumer2++ )  // only Numer2 starts at 0
        for( dwDenom2 = 1.0;  dwDenom2 <= dwDenom2Limit; dwDenom2++ )
```

```
{
	// ---------------------------------------------------------------------------------------
	// --------Speed things up by skipping cases where scores must be negative?--------------------
	// ---------------------------------------------------------------------------------------

	if( SPEED_SEARCH &&

		// check for repeats

		dwNumer1 != dwDenom1 &&  // not a fraction equal to 1
		dwNumer2 != dwDenom2 &&  // not a fraction equal to 1

		dwNumer1 != dwNumer2 &&  // no other repeats
		dwNumer1 != dwDenom2 &&  // no other repeats

		dwDenom1 != dwNumer2 &&  // no other repeats
		dwDenom1 != dwDenom2 &&  // no other repeats

		dwNumer2 != 0                       // fraction2 will not be zero-ed out
		)
	{
		// no repeats, so we may easily anticipate whether it's score is going to be too large
		if( NumberWeight( dwNumer1 ) +
			NumberWeight( dwDenom1 ) +
			NumberWeight( dwNumer2 ) +
			NumberWeight( dwDenom2 ) > dwMaxBitsCompression + SPEED_SEARCH_THRESHOLD )
			continue;  // score would have to be below SPEED_SEARCH_THRESHOLD
	}

	// ---------------------------------------------------------------------------------------
	// --------Inner part of nested for loop-----------------------------------------------------
	// ---------------------------------------------------------------------------------------

	for( nOverallPower = 1; nOverallPower <= MAX_OVERALL_POWER; nOverallPower++)
	for( dwSign = -1; dwSign <= 1; dwSign += 2 )
	for( nNumer1Power = 1, dwNumer1Power = 1.0; nNumer1Power <= MAX_POWER; nNumer1Power++, dwNumer1Power *= dwNumer1 )
	for( nDenom1Power = 1, dwDenom1Power = 1.0; nDenom1Power <= MAX_POWER; nDenom1Power++, dwDenom1Power *= dwDenom1 )
	for( nFraction2Power = 1, dwNumer2Power = dwDenom2Power = 1.0;   nFraction2Power <= MAX_POWER;
			nFraction2Power++, dwNumer2Power *= dwNumer2, dwDenom2Power *= dwDenom2)
	{
		// ---------------------------------------------------------------------------------------
		// ---------Compute powers-------------------------------------------------------------------
		// ---------------------------------------------------------------------------------------

		// compute powers: make 1st fraction out of these:

		dwFraction1Numer = dwNumer1 * dwNumer1Power;
		dwFraction1Denom = dwDenom1 * dwDenom1Power;
```

```
// compute powers: make 2nd fraction out of these:

dwFraction2Numer = dwNumer2 * dwNumer2Power;
dwFraction2Denom = dwDenom2 * dwDenom2Power;

// ---------------------------------------------------------------------------------------
// ---------form of formula searched for-----------------------------------------------------
// ---------------------------------------------------------------------------------------
//
// {[ dwFraction1Numer^a + dwSign * ( dwFraction2Numer / dwFraction2Denom )^c ] / dwFraction1Denom^b }^d
//
// or
//
// { dwFraction1Numer^a / [ dwFraction1Denom^b + dwSign * ( dwFraction2Numer / dwFraction2Denom )^c ]}^d
//
// ---------------------------------------------------------------------------------------
// ---------------------------------------------------------------------------------------

// ---------------------------------------------------------------------------------------
// -----Decide whether putting 2nd fraction in numerator or denominator yields best score------
// ---------------------------------------------------------------------------------------

dwTmp = dwApproxN = (dwFraction1Numer + (dwSign * dwFraction2Numer) / dwFraction2Denom) / dwFraction1Denom;
for( nTmp = nOverallPower; nTmp > 1; nTmp-- )
        dwApproxN = dwTmp * dwApproxN;

dwTmp = dwApproxD =  dwFraction1Numer / (dwFraction1Denom + (dwSign * dwFraction2Numer) / dwFraction2Denom);
for( nTmp = nOverallPower; nTmp > 1; nTmp-- )
        dwApproxD = dwTmp * dwApproxD;

dwErrorN = fabs( dwApproxN - dwTestValue );

dwErrorD = fabs( dwApproxD - dwTestValue );

bRecipricalInNumer = (dwErrorN < dwErrorD);
dwError = (bRecipricalInNumer) ? dwErrorN : dwErrorD;

// ---------------------------------------------------------------------------------------
// --------Does it surpass threshold?-------------------------------------------------------
// ---------------------------------------------------------------------------------------

if( (log( dwTestValue / dwError ) / log(2) - dwMaxBitsCompression) > 0 )
{
        // -----Compute score-----------------------------------------------------------------------

        dwCompression = dwMaxBitsCompression;  // do not allow score to be inflated by over-fitting

        // ---------------------------------------------------------------------------------------
        // -----Powers-----------------------------------------------------------------------------
```

```
// ---------------------------------------------------------------------------------------------

// adjust for powers
// powers of 1 receive no weight, but -1 may be weighted
// duplicate values and exponennts are weighted according to rules

// identify duplicated powers
// start with nNumer1Power first, so it cannot be duplicate

bDenom1PowerIsDuplicate = bFraction2PowerIsDuplicate = bOverallPowerIsDuplicate = 0;

if( nDenom1Power == nNumer1Power )
        bDenom1PowerIsDuplicate = 1;                        // duplicates earlier power

if( nOverallPower   == nNumer1Power ||
        nOverallPower   == nDenom1Power
        )
        bOverallPowerIsDuplicate = 1;                   // duplicates earlier power

if( nFraction2Power == nNumer1Power ||
        nFraction2Power == nDenom1Power ||
        nFraction2Power == nOverallPower
        )
        bFraction2PowerIsDuplicate = 1;                     // duplicates earlier power

if( nNumer1Power > 1)
        dwCompression = dwCompression
                - NumberOfBits( (double) nNumer1Power )
                - WEIGHT_ADDED_FOR_EXPONENT;

if( nDenom1Power > 1 &&
        !(dwNumer2 == 0 && nNumer1Power == nDenom1Power) )  // do not weight duplicated power if Fraction2 is missing
        dwCompression = dwCompression
                - ((bDenom1PowerIsDuplicate)    ?
                        WEIGHT_FOR_EXPONENT_REUSE   :    // duplicates earlier power
                        NumberOfBits( (double) nDenom1Power )    + WEIGHT_ADDED_FOR_EXPONENT );

if( nFraction2Power > 1 && dwNumer2 > 0 && dwNumer2 != dwDenom2  )
        dwCompression = dwCompression
                - ((bFraction2PowerIsDuplicate) ?
                        WEIGHT_FOR_EXPONENT_REUSE   :    // duplicates earlier power
                        NumberOfBits( (double) nFraction2Power ) + WEIGHT_ADDED_FOR_EXPONENT );

if( nOverallPower > 1)
        dwCompression = dwCompression
                - ((bOverallPowerIsDuplicate)   ?
                        WEIGHT_FOR_EXPONENT_REUSE   :    // duplicates earlier power
                        NumberOfBits( (double) nOverallPower )   + WEIGHT_ADDED_FOR_EXPONENT );
```

```
// ---------------------------------------------------------------------------------------
// ----------Fraction2-------------------------------------------------------------------
// ---------------------------------------------------------------------------------------

// -------------- is Fraction2 really a fraction? --------------

// or is it really just 0 or 1?
// this must be answered to know whether to discount for repeats

if( dwNumer2 == 0 || dwNumer2 == dwDenom2 )
        bFraction2IsFraction = 0;  // Fraction2 is really 0 or 1; do NOT allow it to be used for repeats
else
        bFraction2IsFraction = 1;  // Fraction2 not 0 or 1; allow it to be used for repeats

// -------------- calculate weight for Numer2 --------------

if( dwNumer2 == 0 )
        dwCompression -= 0;  // Fraction2 is unused
else if( dwNumer2 == dwDenom2 )
        dwCompression -= WEIGHT_FOR_ONE;  // Fraction2 is just 1
else if( dwNumer2 == 1.0 )
        // in effect, a negative exponent
        // if it equals -1, we adjust weight here
        dwCompression -= ( 1 == nFraction2Power ) ? WEIGHT_FOR_NEG_ONE_EXPONENT : 0 ;
else
        dwCompression = dwCompression - (NumberWeight( dwNumer2 ));

// -------------- calculate weight for Denom2 --------------

if( dwNumer2 == 0 )
        dwCompression -= 0;  // Fraction2 is unused
else if( dwDenom2 == 1.0 || dwNumer2 == dwDenom2 )
        dwCompression -= 0;  // Denom2 is uneeded
else
        dwCompression = dwCompression - (NumberWeight( dwDenom2 ));

// ---------------------------------------------------------------------------------------
// ----------Fraction1-------------------------------------------------------------------
// ---------------------------------------------------------------------------------------

// -------------- calculate weight for Numer1 --------------
// allow that Numer1 may already have been weighted for in Fraction2
// or that a numerator of 1 may be replaced by -1 exponent

if( dwNumer1 == 1.0 && !bReciprocalInNumer && nOverallPower == 1)
        dwCompression -= WEIGHT_FOR_NEG_ONE_EXPONENT;  // in effect an exponent of -1
else if( dwNumer1 == 1.0 && !bReciprocalInNumer && nOverallPower > 1)
        dwCompression -= 0;  // in effect an exponent of -nOverallPower, weighted for elsewhere
else if ( bFraction2IsFraction && (dwNumer1 == dwDenom1 || dwNumer1 == dwNumer2 || dwNumer1 == dwDenom2) )
```

```
                        dwCompression -= WEIGHT_FOR_REUSE;  // weight for repeat
                else
                        dwCompression = dwCompression - (NumberWeight( dwNumer1 ));

                // ---------- calculate weight for Denom1 ----------
                // allow that Denom1 may already have weighted for in Fraction2

                if( dwDenom1 == 1.0 && bRecipricalInNumer )
                        dwCompression -= 0;  // Denom1 is uneeded
                else if ( bFraction2IsFraction && (dwDenom1 == dwNumer2 || dwDenom1 == dwDenom2) )
                        dwCompression -= WEIGHT_FOR_REUSE;  // weight for repeat
                else
                        dwCompression = dwCompression - (NumberWeight( dwDenom1 ));
        }
        else
        {
                dwCompression = -100.0;
        }

        // is it best score so far?

        if( dwBestCompression < dwCompression )
        {
                // save score

                bBestRecipricalInNumer = bRecipricalInNumer;

                dwBestNumer1 = dwNumer1;
                dwBestNumer2 = dwNumer2;
                dwBestDenom2 = dwDenom2;
                dwBestDenom1 = dwDenom1;

                dwBestSign   = dwSign;

                nBestNumer1Power       = nNumer1Power;
                nBestDenom1Power       = nDenom1Power;
                nBestFraction2Power = nFraction2Power;
                nBestOverallPower        = nOverallPower;
                dwBestCompression      = dwCompression;
        }
    }
}

// ------------------------------------------------------------------------------------
// -------print score of best approximation found----------------------------------------
// ------------------------------------------------------------------------------------

if( bBestRecipricalInNumer )
        printf( "%10.8f = {[%2d^%1d %s (%2d/%-2d)^%1d] / %2d^%1d}^%d: Compression = %d - %02.0f = %+02.0f bits\n",
```

```
                        (float) dwTestValue,

                        (int) dwBestNumer1,
                        (int) nBestNumer1Power,

                        (dwBestSign > 0) ? "+" : "-",

                        (int) dwBestNumer2,
                        (int) dwBestDenom2,
                        (int) nBestFraction2Power,

                        (int) dwBestDenom1,
                        (int) nBestDenom1Power,
                        (int) nBestOverallPower,

                        (int) dwMaxBitsCompression,
                        (float) dwMaxBitsCompression - (float) dwBestCompression,
                        (float) dwBestCompression
                );
        else
                printf( "%10.8f = {%2d^%1d / [%2d^%1d %s (%2d/%-2d)^%1d]}^%d: Compression = %d - %02.0f = %+02.0f bits\n",

                        (float) dwTestValue,

                        (int) dwBestNumer1,
                        (int) nBestNumer1Power,

                        (int) dwBestDenom1,
                        (int) nBestDenom1Power,

                        (dwBestSign > 0) ? "+" : "-",

                        (int) dwBestNumer2,
                        (int) dwBestDenom2,
                        (int) nBestFraction2Power,
                        (int) nBestOverallPower,

                        (int) dwMaxBitsCompression,
                        (float) dwMaxBitsCompression - (float) dwBestCompression,
                        (float) dwBestCompression
                );

        return dwBestCompression;
}

/*---------------------------------------------------------------------------*/

// number of bits in integer
```

```
// guard against failure to round up correctly

double NumberOfBits( double dwVal )
{
	// ceil( x + 0.00000001 ) rounds up all x INCLUDING integers

	return ceil( log( dwVal ) / log(2) + 0.00000001 );
}

double NumberWeight( double dwVal )
{
	return NumberOfBits( dwVal ) + WEIGHT_ADDED_FOR_FIRST_USE;
}

double NumberWeight2( double dwVal )
{
	double dwExtraWeight;

	if( dwVal <= 2 )
		dwExtraWeight = 0;
	else if( dwVal <= 15 )
		dwExtraWeight = 1;
	else if( dwVal <= 63 )
		dwExtraWeight = 2;
	else
		dwExtraWeight = 3;
	return NumberOfBits( dwVal) + dwExtraWeight;
}

/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/

double GetRandomTestValue()
{
	return GetRandom();
}

/* starting values for random generator */

/* See Computer Language, Dec. 1989 for info. on random numbers */

#define RANDOM_MOD_DIV1 32771L
#define RANDOM_MOD_DIV2 32779L
#define RANDOM_MOD_DIV3 32783L

#define RANDOM_MULT1 179L
#define RANDOM_MULT2 183L
#define RANDOM_MULT3 182L
```

```
#define RANDOM_SEED1 7397L
#define RANDOM_SEED2 29447L
#define RANDOM_SEED3 802L

#define RANDOM_TOP ( 32000L  )

static unsigned long ulSeed1;
static unsigned long ulSeed2;
static unsigned long ulSeed3;

// Note: Generally use 0 seed

void SetRandom( unsigned long  ulValue )
{
   ulSeed1 = ulValue + RANDOM_SEED1;
   ulSeed2 = ulValue + RANDOM_SEED2;
   ulSeed3 = ulValue + RANDOM_SEED3;

   ulSeed1 = ulSeed1 % RANDOM_TOP;
   ulSeed2 = ulSeed2 % RANDOM_TOP;
   ulSeed3 = ulSeed3 % RANDOM_TOP;

   if( 0L == ulSeed1 )
      ulSeed1 = 1L;
   if( 0L == ulSeed2 )
      ulSeed2 = 1L;
   if( 0L == ulSeed2 )
      ulSeed2 = 1L;
}

double GetRandom( void )
{
       double dwTemp;

   ulSeed1 = RANDOM_MULT1 * ulSeed1;
   ulSeed2 = RANDOM_MULT2 * ulSeed2;
   ulSeed3 = RANDOM_MULT3 * ulSeed3;

   ulSeed1 = ulSeed1 % RANDOM_MOD_DIV1;
   ulSeed2 = ulSeed2 % RANDOM_MOD_DIV2;
   ulSeed3 = ulSeed3 % RANDOM_MOD_DIV3;

       do {
            dwTemp =
                 (((double) ulSeed1) / ((double) RANDOM_MOD_DIV1         )) - // assure number is not simple ratio by
                 (((double) ulSeed2) / ((double) RANDOM_MOD_DIV2 * 1000.0))      + // making small downward adj. and
                 (((double) ulSeed3) / ((double) RANDOM_MOD_DIV3 * 1000.0));   // making small upward adj.
       } while ( dwTemp <= 0.0 || dwTemp >= 1.0 );
```

```
    return dwTemp;
}

/*---------------------------------------------------------------------------*/
```