Reliable Computing, , 1–7 ()

Interval Computation of Viswanath's Constant*

JOÃO BATISTA S. DE OLIVEIRA oliveira@inf.pucrs.br
Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga 6681, 90619-900 Porto Alegre, RS, Brazil

LUIZ HENRIQUE DE FIGUEIREDO lhf@impa.br
IMPA–Instituto de Matemática Pura e Aplicada
Estrada Dona Castorina 110, 22461-320 Rio de Janeiro, RJ, Brazil

Received ... (submitted on April 6, 2001)

Abstract. Viswanath has shown that the terms of the random Fibonacci sequences defined by $t_1 = t_2 = 1$, and $t_n = \pm t_{n-1} \pm t_{n-2}$ for $n > 2$, where each $\pm$ sign is chosen randomly, increase exponentially in the sense that $\sqrt[n]{|t_n|} \to 1.13198824\ldots$ as $n \to \infty$ with probability 1. Viswanath computed this approximation for this limit with floating-point arithmetic and provided a rounding-error analysis to validate his computer calculation. In this note, we show how to avoid this rounding-error analysis by using interval arithmetic.

1. Introduction

The Fibonacci sequence (f_n) is defined by $f_1 = f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$ for $n > 2$. It is well known that the terms of the Fibonacci sequence increase exponentially in the sense that $\sqrt[n]{f_n} \to (1+\sqrt{5})/2 = 1.61803398\ldots$ as $n \to \infty$.

A *random* Fibonacci sequence (t_n) is defined by $t_1 = t_2 = 1$ and $t_n = \pm t_{n-1} \pm t_{n-2}$ for $n > 2$, where each $\pm$ sign is independent and either $+$ or $-$ with probability $1/2$. Viswanath [4] has shown that the terms of such random Fibonacci sequences increase exponentially in the sense that $\sqrt[n]{|t_n|} \to 1.13198824\ldots$ as $n \to \infty$ with probability 1.

Viswanath computed an approximate value for this limit with floating-point arithmetic and provided a detailed rounding-error analysis to validate the computer calculation for the decimal places given above. In this note, we show that the use of interval arithmetic [2] avoids the need for this rounding-error analysis and also simplifies the computer program.

2. Preliminaries

Viswanath argued that $\sqrt[n]{|t_n|} \to e^{\gamma}$ as $n \to \infty$, where γ is the upper Lyapunov exponent of a random matrix product, and then proceeded to compute a small interval containing γ. His main result is that $\gamma \in (0.1239755980, 0.1239755995)$, which implies that $e^{\gamma} = 1.13198824\ldots$, as claimed in the title of his paper.

* The authors are partially supported by research grants from the Brazilian Council for Scientific and Technological Development (CNPq) and by the summer post-doctoral program at IMPA.

To compute an interval containing γ, Viswanath used Furstenberg's formula:

$$\gamma = 2\int_0^\infty \mathrm{amp}(m)\, d\nu(m),$$

where

$$\mathrm{amp}(m) = \frac{1}{4}\log\Big(\frac{1+4m^4}{(1+m^2)^2}\Big)$$

and ν is an invariant probability measure for a random walk associated to the random Fibonacci sequence. This integral was then computed by using the Stern–Brocot tree partition of the real line [1]. As Viswanath explains in detail, there is a simple recursive formula for computing $\nu(I)$ when I is a Stern–Brocot interval.

Let $\mathcal{S}_d$ be the set of positive Stern–Brocot intervals at depth $d+1$. Then,

$$\gamma = 2\int_0^\infty \mathrm{amp}(m)\, d\nu(m) = 2\sum_{I\in\mathcal{S}_d}\int_I \mathrm{amp}(m)\, d\nu(m),$$

and so $\gamma \in (p_d, q_d)$, where

$$p_d := 2\sum_{I\in\mathcal{S}_d}\min_{m\in I}\mathrm{amp}(m)\,\nu(I), \qquad q_d := 2\sum_{I\in\mathcal{S}_d}\max_{m\in I}\mathrm{amp}(m)\,\nu(I).$$

Viswanath carefully computed floating-point approximations for p_d and q_d with $d = 28$. He gave a computer program for this computation and a detailed rounding-error analysis that validades the results obtained by the program. However, his program needs more than a gigabyte of memory to run, and thus his results cannot be readily reproduced in most desktop machines. One of our goals in this work is to provide a program that can be used by anyone interested in reproducing Viswanath's result.

Our other goal is to show that the rounding-error analysis can be replaced by bounds computed a posteriori with interval arithmetic. Here is a formulation for computing γ that is suitable for interval arithmetic:

$$\gamma = 2\sum_{I\in\mathcal{S}_d}\int_I \mathrm{amp}(m)\, d\nu(m) \in 2\sum_{I\in\mathcal{S}_d}\mathrm{AMP}(I)\nu(I),$$

where $\mathrm{AMP}(I)$ is an interval enclosure for $\mathrm{amp}(I)$ computed with interval arithmetic. (By *interval enclosure* we mean an interval that contains $\mathrm{amp}(I)$.)

Viswanath exploited the simple behavior of $\mathrm{amp}(m)$ to divide the computation into parts where $\mathrm{amp}(m)$ is monotone and thus was able to compute $\mathrm{amp}(I)$ exactly (except for rounding errors). We decided to use as "AMP" the natural interval extension of "amp", obtained by replacing arithmetic operators and logarithms with their interval extensions, because this leads to a simpler program. The main point here is that interval arithmetic keeps track of rounding errors automatically.

3. Computing enclosures for γ with interval arithmetic

The main changes we made to Viswanath's original program were the following:

1. To reduce memory requirements the original program so that anyone could reproduce Viswanath's result, it was necessary to replace the huge array `num`, containing the Stern–Brocot coefficients, by a recursive computation, thus trading memory for execution time. So, instead of precomputing the array `num` and using `num[k]` in the computation, our program uses `num(k)`, a call to a subroutine providing such coefficients. In the original code from Viswanath, the array `num` is filled as follows:

   ```
   num[0] = 1; num[1] = 1;
   for (i=2; i<N; i=i+2) {
     num[i]   = num[i/2];
     num[i+1] = num[i/2]+num[i/2+1];
   }
   num[N] = 1;
   ```

   and so the necessary recurrence is

   $$\mathtt{num}(k) = \begin{cases} 1, & \text{for } k = 1; \\ \mathtt{num}(k/2), & \text{for } k \text{ even}; \\ \mathtt{num}((k-1)/2) + \mathtt{num}((k-1)/2+1), & \text{for } k \text{ odd}. \end{cases}$$

   Of course, much of the running time will be now spent on computing coefficients, even though this recurrence can be computed with effort $O(\log_2 k)$ for each k. To alleviate this computation, a cache buffer `buff` of pre-computed coefficients was created, storing (in our case) the first 2^{23} coefficients, which are used when needed. Thus, we have a new, faster recurrence:

   $$\mathtt{num}(k) = \begin{cases} \mathtt{buf[k]}, & \text{for } k \le 2^{23}; \\ \mathtt{num}(k/2), & \text{for } k \text{ even}; \\ \mathtt{num}((k-1)/2) + \mathtt{num}((k-1)/2+1), & \text{for } k \text{ odd}. \end{cases}$$

   This new recurrence is faster because it stops as soon as k gets below 2^{23}, which only takes about 9 steps for $k \le 2^{32}$. The cache `buff` is filled at the beginning of our program with a loop exactly like the one in the original program.

   This change allows the program to run in machines with modest memory without being too slow. The size of our cache was chosen as the largest that can still hold all needed values in two bytes per entry (i.e., `buf[k]` ≤ 65535).

2. Generate the `table` of the values of $\nu(I)/2$ automatically, for I in $\mathcal{S}_d$, given the desired depth $d+1$ of the Stern–Brocot tree. These values were computed to high accuracy using the arbitrary-precision calculator `bc` with the code below:

   ```
   scale = 50; g = (1 + sqrt(5))/2;
   for (i=0; i <= d; i++) g^(d-i)*(1+g)^(-d)/4;
   ```

We also generated macros for `N`, whose value is 2^d, for `NRT`, whose value is $2^{d/2}$, and for the `filter` used to index the values of $\nu(I)$ in `table`. This parametrization allowed the modified program to run (still in plain floating-point arithmetic) for d up to 32, even in machines with modest memory. The only price to be paid is running time.

3. Replace all occurrences of `double` with a C++ class `interval`, which implements interval data types and operations. We used `RVInterval`, a library freely available in the Internet [3]. This library takes care of rounding control in arithmetic operations and runs in both Linux and Solaris (among others).

4. Compute a table `itable` of intervals containing the values of $\nu(I)/2$ that are stored in floating-point in `table`. These intervals have diameter at most $4u$, where u is the "machine epsilon" ($= 2^{-52}$). Intervals of diameter u would be ideal, but cannot be obtained easily because we cannot control how the compiler rounds floating-point constants read from the program text. So, we were forced to explicitly multiply the values in `table` by $1 - u$ and $1 + u$, rounding down in the first case and up in the second case. This gave intervals with diameter at most $4u$, which were wider than strictly necessary but good enough for our purposes.

5. Compute the sums for p_d and q_d with interval arithmetic. The core of this computation is the single line below:

```
result += AMP(interval(left.min(),right.max())) * itable[b];
```

This code implements the summation

$$2 \sum_{I \in \mathcal{S}_d} \mathrm{AMP}(I)\nu(I)$$

in interval arithmetic, according to Viswanath's rule for the value of $\nu(I)$ from the index of I in $\mathcal{S}_d$. (Note how the code is easy to read!) We explicitly replaced the pairwise summation used in the original program by a left-to-right summation to make the program clearer, because interval arithmetic controls rounding errors automatically. Actually, we factored out the (interval) multiplication by `itable[b]`, and used an array of accumulators `R`, one for each `b`:

```
R[b] += AMP(interval(left.min(),right.max()));
```

The final interval result was computed by summing `R[b]*itable[b]`. This avoids 2^d interval multiplications and the corresponding loss of significant bits due to rounding.

The changes described above are not extensive, and the resulting program has essentially the same size as the original program, but is arguably easier to read. The complete program is shown in the Appendix.

Table 1. Enclosures for γ.

d	Left bound	Right bound	Diameter	CPU time
6	0.11283281476075978	0.13258611571571169	2.0×10^{-2}	0.38
8	0.12095052025785517	0.12639934806962877	5.4×10^{-3}	0.33
10	0.12317103249016015	0.12463761240153247	1.5×10^{-3}	0.30
12	0.12376461949077926	0.12415279732484592	3.9×10^{-4}	0.35
14	0.12392085135129263	0.12402234102445128	1.0×10^{-4}	0.40
16	0.12396150759669380	0.12398779267063693	2.6×10^{-5}	0.60
18	0.12397199563321587	0.12397875219502701	6.8×10^{-6}	1.55
20	0.12397468243410505	0.12397640852895681	1.7×10^{-6}	5.23
22	0.12397536680911644	0.12397580550758186	4.4×10^{-7}	20.22
24	0.12397554029636378	0.12397565131195952	1.1×10^{-7}	82.74
26	0.12397558408863271	0.12397561209514275	2.8×10^{-8}	351.79
28	0.12397559507726642	0.12397560219184581	7.1×10^{-9}	1667.34
30	0.12397559772623701	0.12397559980033829	2.1×10^{-9}	8988.48
32	0.12397559798522664	0.12397559960223987	1.6×10^{-9}	55545.80

Table 2. Enclosures for e^{γ}.

d	Left bound	Right bound	Diameter
6	1.11944476266250548	1.14177733682818805	2.2×10^{-2}
8	1.12856906967925408	1.13473523213640970	6.2×10^{-3}
10	1.13107785546740947	1.13273788851645607	1.7×10^{-3}
12	1.13174944788364762	1.13218885321122009	4.4×10^{-4}
14	1.13192627701833826	1.13204116167593716	1.1×10^{-4}
16	1.13197229782634290	1.13200205219293926	3.0×10^{-5}
18	1.13198417005540253	1.13199181840225527	7.6×10^{-6}
20	1.13198721147556314	1.13198916539454775	2.0×10^{-6}
22	1.13198798617958896	1.13198848278109043	5.0×10^{-7}
24	1.13198818256508571	1.13198830823343544	1.3×10^{-7}
26	1.13198823213741773	1.13198826384045814	3.2×10^{-8}
28	1.13198824457642178	1.13198825263004244	8.1×10^{-9}
30	1.13198824757502536	1.13198824992288394	2.3×10^{-9}
32	1.13198824786819863	1.13198824969863887	1.8×10^{-9}

We ran our program on a Linux machine with 128 megabytes of memory for several values of d. Table 1 shows the enclosures computed for γ and also the CPU time in seconds required in each case. Table 2 shows the corresponding enclosures for e^{γ}. For comparison, Viswanath's enclosure was $(0.1239755980, 0.1239755995)$, computed with $d = 28$. The diameter of this interval is 1.5×10^{-9}.

The results computed with interval arithmetic are wider than Viswanath's, but the diameters for $d \geq 28$ are of the same order of magnitude. Considering that the interval program is easier to read, and that special care was taken neither in the summation of the terms nor in the computation of the interval enclosure AMP, we conclude that the computation with interval arithmetic can replace the detailed rounding-error analysis performed by Viswanath. Slightly better results could possibly be obtained by using a tighter interval enclosure AMP, as Viswanath did (approximately) in pure floating-point, and by using pairwise summation instead of the simpler left-to-right summation.

The complete set of programs and results is available from the authors.

Acknowledgments

We thank S. Rump for reading and commenting on a previous version of this paper.

Appendix – program listing

```
#include <stdio.h>
#include "interval.h"           /* RVInterval */
#include "npu.h"
#include "data.h"               /* D, N, table */

#define bitcount(x,b) {b = 0; for( ; x!=0; x&=(x-1)) b++; }

#define MAXBUF ( 1 << 23 )
unsigned short buf[MAXBUF];

unsigned int num(unsigned int i) {
    if (i < MAXBUF) return buf[i];
    if ((i % 2) == 0) return num(i / 2);
    return num(i / 2) + num(i / 2 + 1);
}

inline interval sternbrocot(unsigned int k) {
    interval numerador = (int) num(k);
    interval denom = (int) num(N - k);
    return numerador / denom;
}

inline interval AMP(interval m) {
    interval m2 = sqr(m);
    interval m2p1 = m2 + 1;
    return log((sqr(m2) * 4 + 1) / sqr(m2p1));
}

int main() {
    unsigned int i, b;

    initnpu();

    buf[0] = buf[1] = 1;
    for (i = 2; i < MAXBUF; i = i + 2) {
        buf[i] = buf[i / 2];
        buf[i + 1] = buf[i / 2] + buf[i / 2 + 1];
    }
```

```
    interval itable[D+1];
    for (i = 0; i <= D; i++)
        itable[i] = interval(rounddown(table[i]),roundup(table[i]));

    interval R[D+1];
    for (i = 0; i <= D; i++) R[i] = 0;

    interval right, left = sternbrocot(0);
    for (i = 0; i < N-1; i++) {
        right = sternbrocot(i + 1);
        int x = i ^ filter;
        bitcount(x, b);
        R[b] += AMP(interval(left.min(), right.max()));
        left = right;
    }

    right = log((interval) 4.0);
    int x = i ^ filter;
    bitcount(x, b);
    R[b] += AMP(interval(left.min(), right.max()));

    interval result = 0;
    for (i = 0; i <= D; i++) result += R[i] * itable[i];

#define show(I) \
    printf("%d   %.17f %.17f  %.1e\n",D,I.min(),I.max(),I.width())

    show(result);
    result = exp(result);
    show(result);

    return 0;
}
```

References

1. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics – A foundation for computer science.* Addison-Wesley Publishing Company, Reading, MA, second edition, 1994.
2. R. E. Moore. *Interval analysis.* Prentice-Hall Inc., Englewood Cliffs, N.J., 1966.
3. R. Van Iwaarden. RVInterval, a C++ interval arithmetic package. Available at `http://www-math.cudenver.edu/~rvan/Software.html`.
4. D. Viswanath. Random Fibonacci sequences and the number 1.13198824 . . . . *Math. Comp.*, 69(231):1131–1155, 2000.