An exercise on streams: convergence acceleration

Pierre Lescanne

An exercise on streams: convergence acceleration

Pierre Lescanne
University of Lyon,
École normale supérieure de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)
46 allée d'Italie, 69364 Lyon, France

December 17, 2013

Abstract

This paper presents convergence acceleration, a method for computing efficiently the limit of numerical sequences as a typical application of streams and higher-order functions.

Keywords: convergence acceleration, streams, numerical analysis, co-algebra

1 Introduction

Assume that we want to compute numerically the limit of a sequence that converges slowly. If we use the sequence itself, we will get significant figures of the limit after a very long time. Methods called *convergence acceleration* have been designed to shorten the time after which we get reasonable amount of significant digits. In other words, *convergence acceleration* is a set of methods for numerically computing the limit of a sequence of numbers. Those methods are based on sequence transformations and are a very nice domain of application of streams, with beautiful higher order functions. This allows us to present very elegantly rather complex methods and to code them in Haskell [4] replacing long, obscure and special purpose Fortran programs by short, generic and arbitrary precision Haskell programs.

In this paper we show, how given a sequence $(s_n)_{n\in\mathbb{N}}$, we can evaluate efficiently $\lim_{n\to\infty} s_n$. For that we use *Levin transforms*. There are three kinds of such transforms, which are the result of three sequence transformations labeled traditionally by t, u and v ([6], p.58).

2 Presentation of the method

In what follows we speak indistinctly of "sequences" or of "streams" We use Haskell and we work with arbitrary precision reals based on the implementation due to David Lester and called *CReal*. In Haskell the type of streams over a type A is written $[A]$.

For the numerical aspect, we follows Naoki Osada [6] and we show that the stream notation of Haskell makes the presentation much simpler. With no surprise, the size of the Haskell code is the same if not shorter than the mathematical description of the reference [6]. Moreover it provides efficient programs.

Levin transformations are somewhat generic in the sense that they are based on elementary transformations. Specialists of convergence acceleration propose three

such elementary transformations. Let s be a sequence on $CReal$, i.e., $s :: [CReal]$. We define first a basic sequence transformation on which we will found our elementary transformations:

$$
\begin{aligned}
&dELTA :: [CReal] \to [CReal] \\
&dELTA\ s = zipWith\ (-)\ (tail\ s)\ s
\end{aligned}
$$

which means that $dELTA(s)_n = s_{n+1} - s_n$. From this basic sequence transformation we define the three elementary other sequence transformations as follows. A unique function depending on a character parameter which is either $'t'$ or $'u'$ or $'v'$ is given. It corresponds to the traditional notations of numerical analysis for those sequence transformations:

$$
\begin{aligned}
&delta :: Char \to [CReal] \to [CReal] \\
&delta\ 't'\ s = dELTA \\
&delta\ 'u'\ s = zipWith\ (*)\ (dELTA\ s)\ [1..] \\
&delta\ 'v'\ s = zipWith\ (/)\ (zipWith\ (*)\ (tail\$dELTA\ s)\ (dELTA\ s)) \\
&\qquad\qquad\qquad\quad (dELTA\ (dELTA\ s))
\end{aligned}
$$

In numerical analysis, people speak about *E-algorithm*. This is a family of functions $eAlg_{n,k}$ which are also parametrized by a character either $'t'$ or $'u'$ or $'v'$. It tells which of the basic sequence transformations is chosen. $eAlg_{n,k}$ uses a family of auxiliary functions which we call $gAlg_{n,k}$ for symmetry and regularity. Here is the Haskell code for these functions:

$$
\begin{aligned}
&eAlg :: Char \to Int \to [CReal] \to [CReal] \\
&eAlg\ c\ 0\ s = s \\
&eAlg\ c\ k\ s = \mathbf{let} \\
&\qquad\qquad a = (eAlg\ c\ (k-1)\ s) \\
&\qquad\qquad b = (gAlg\ c\ (k-1)\ k\ s) \\
&\qquad\quad \mathbf{in} \\
&\qquad\quad zipWith\ (-)\ a\ (zipWith\ (*)\ b\ (zipWith\ (/)\ (dELTA\ a)\ (dELTA\ b)))
\end{aligned}
$$

$$
\begin{aligned}
&gAlg :: Char \to Int \to Int \to [CReal] \to [CReal] \\
&gAlg\ c\ 0\ j\ s = \mathbf{let} \\
&\qquad\qquad nTojMinus1\ j = zipWith\ (**)\ [1..]\ (map\ fromIntegral\ [j-1, j-1..]) \\
&\qquad\quad \mathbf{in} \\
&\qquad\quad zipWith\ (/)\ (nTojMinus1\ j)\ (delta\ c\ s) \\
&gAlg\ c\ k\ j\ s = \mathbf{let} \\
&\qquad\qquad a = gAlg\ c\ (k-1)\ j\ s \\
&\qquad\qquad b = gAlg\ c\ (k-1)\ k\ s \\
&\qquad\quad \mathbf{in} \\
&\qquad\quad zipWith\ (-)\ a\ (zipWith\ (*)\ b\ (zipWith\ (/)\ (dELTA\ a)\ (dELTA\ b)))
\end{aligned}
$$

Here is the formula as it is given in [6]. R_n is the generic value of $(delta\ c\ s)_n$. $gAlg$ is written g. $E_k^{(n)}$ is the n^{th} element of the sequence $eAlg\ c\ k\ s$, the same for $g_{k,j}^{(n)}$. Δ is the notation for what we write $dELTA$.

$$
\begin{aligned}
E_0^{(n)} &= s_n,\ \ g_{0,j}^{(n)} = n^{1-j} R_n \quad , \qquad n = 1, 2, ...;\ j = 1, 2, ..., \\
E_k^{(n)} &= E_{k-1}^{(n)} - g_{k-1,k}^{(n)} \frac{\Delta E_{k-1}^{(n)}}{\Delta g_{k-1,k}^{(n)}} \ , \qquad n = 1, 2, ...;\ k = 1, 2, ..., \\
g_{k,j}^{(n)} &= g_{k-1,j}^{(n)} - g_{k-1,k}^{(n)} \frac{\Delta g_{k-1,j}^{(n)}}{\Delta g_{k-1,k}^{(n)}}, \qquad n = 1, 2, ...;\ k = 1, 2, ..., j > k
\end{aligned}
$$

3 Levin's formulas

There is another formula for the $E-algorithm$:

$$T_k^{(n)} = \frac{\displaystyle\sum_{j=0}^{k}(-1)^j\binom{k}{j}\binom{n+j}{n+k}^{k-1}\frac{s_{n+j}}{R_{n+j}}}{\displaystyle\sum_{j=0}^{k}(-1)^j\binom{k}{j}\binom{n+j}{n+k}^{k-1}\frac{1}{R_{n+j}}}$$

$T_k^{(n)}$ is not easily amenable to a Haskell program.[1] We give only the functions for $k = 0, 1, 2$, which we call *levin* 0, *levin* 1 and *levin* 2. For $k = 0$, $T_0^{(n)} = s_n$ and *levin* 1 and *levin* 2 are the result of small calculations.

```
levin :: Char → Int → [CReal] → [CReal]
levin c 0 s = s
levin c 1 s = zipWith (−) s (zipWith (/) (zipWith (∗) (dELTA s) (delta c s))
                                         (dELTA (delta c s)))
```

levin $'u'$ 1 s is called *Aitken's delta-squared process*. One notices that the numerator and the denominator of the above formula differ slightly. Indeed s_{n+j} in the numerator is just replaced by 1 in the denominator.

```
formulaForLevinTwo :: Char → [CReal] → [CReal] → [CReal]
formulaForLevinTwo c s' s =
 zipWith (+) (zipWith (−) (foldl (zipWith (∗)) [2..] [tail$tail s', tail$delta c s, delta c s])
                          (foldl (zipWith (∗)) [2, 4..] [tail s', tail$tail$delta c s, delta c s]))
             (foldl (zipWith (∗)) [0..] [s', tail$tail$delta c s, tail$delta c s])
```

```
levin c 2 s = zipWith (/) (forLevinTwo c s s) (forLevinTwo c [1, 1..] s)
```

Brezinski [2] proves that the sequences $E_k^{(n)}$ and $T_k^{(n)}$ are the same, in other words:

```
eAlg == levin
```

4 Examples

> *More than any other branch of numerical analysis, convergence acceleration is an experimental science. The researcher applies the algorithm and looks at the results to assess their worth.*
>
> Dirk Laurie [5]

Given a sequence we use the convergence acceleration to find the main coefficient of the asymptotic equivalent. More precisely given an integer sequence s_n, we want to find a number $a > 1$ such that $s \sim a^n f(n)$ where $f(n)$ is subexponential. In other words, for all $b \in \mathbb{R}$, $b > 1$, $\lim_{n\to\infty} \dfrac{f(n)}{b^n} = 0$. Actually we compute the limit of the sequence s_{n+1}/s_n. For that we create the function:

[1] In particular due to many divisions by 0 and to the complexity of the formula.

```
expCoeffAC :: ([CReal] → [CReal]) → [Integer] → Int → CReal
expCoeffAC tranform sequence n = last tranform $ zipWith (/) (tail u) u
  where
    u = map fromIntegral (take n sequence)
```

Thus *expCoeffAC* (*levin′u′* 2) *s* 300 gives the approximation of the coefficient one can get after 300 iterations using the sequence transformation *levin′u′* 2.

4.1 Catalan numbers

Among astonishing examples are Catalan numbers:

```
catalan = 1 : [ let
                  cati = take i catalan
                in
                sum (zipWith (∗) cati (reverse cati)) | i ← [1..]]
```

We know that

$$catalan!!n \sim \frac{4^n}{\sqrt{\pi n^3}}$$

Actually we get *expCoeffAC* (*levin′u′* 2) *catalan* 5 == 4.0 showing a very fast acceleration. Indeed we get the exponential coefficient after 5 iterations [2] whereas the ratio *catalan*!!5 over *catalan*!!4 is 42/14 that is 3. After 300 iterations we get only 3.98 and after 600 iterations we get 3.99.

4.2 Counting plain lambda terms

Unfortunately, not all the sequences are alike. Now we want to use this technique to address a conjecture, on the asymptotic evaluation of the exponential coefficient of the numbers of typable terms of size n when n goes to ∞. First let us give the recursive definition of the numbers S_∞ of plain lambda terms of size n. This sequence appears on the *On-line Encyclopedia of Integer Sequences* with the entry number **A114851**. We assume that abstractions and applications have size 2 and variables have size $2 + k$ where k is the depth of the variable w.r.t. its binder.

$$\begin{aligned} S_{\infty,0} &= S_{\infty,1} = 0, \\ S_{\infty,n+2} &= 1 + S_{\infty,n} + \sum_{k=0}^{n} S_{\infty,k} S_{\infty,n-k}. \end{aligned}$$

It has been proved in [3] that

$$S_{\infty,n} \sim A^n \cdot \frac{C}{n^{3/2}},$$

where $A \doteq 1.963447954$ and $C \doteq 1.021874073$. After 300 iterations and using *levin′u′* 2 we found

$$1.96344895227352832916191477135699933556 16.$$

giving six exact digits.

[2] 4 iterations give no result.

4.3 Counting typable lambda terms

The question is now to find the exponential coefficients for the numbers of typable terms. We have no formula for computing those numbers. The only fact we know is the following table of the numbers $T_{\infty,n}$ till 42 which has been obtained after heavy computations (more than 5 days for the 42^{nd}). The method consists in generating all the lambda terms of a given size and sieving those that are typable to count them.

n	$T_{\infty,n}$
0	0
1	0
2	1
3	1
4	2
5	2
6	3
7	5
8	8
9	13
10	22
11	36
12	58
13	103
14	177
15	307
16	535
17	949
18	1645
19	2936
20	5207

n	$T_{\infty,n}$
21	9330
22	16613
23	29921
24	53588
25	96808
26	174443
27	316267
28	572092
29	1040596
30	1888505
31	3441755
32	6268500
33	11449522
34	20902152
35	38256759
36	70004696
37	128336318
38	235302612
39	432050796
40	793513690
41	1459062947
42	2683714350

Therefore the best method to guess the exponential coefficient is by acceleration of convergence. After 43 iterations we found 1.8375065809.... Knowing that with the same number 43 we get 1.8925174623... for S_∞, this is not enough to conclude. But this allows us to speculate that the exponential coefficient for the $T_{\infty,n}$ could be 1.963447954 like for the $S_{\infty,n}$'s.

5 Conclusion

We have shown how streams can be applied to a field of numerical analysis. It makes no doubt that they can also be applied to other fields. For instance, one may imagine applications of acceleration convergence to the computation of limits of non numerical sequences.

References

[1] Folkmar Bornemann, D. P. Dirk Pieter Laurie, S. Wagon, and Jörg Waldvogel. *The SIAM 100-digit challenge : a study in high-accuracy numerical computing.* Society for Industrial and Applied Mathematics, Philadelphia, 2004.

[2] C. Brezinski. A general extrapolation algorithm. *Numer. Math.*, 35, 1980.

[3] Katarzyna Grygiel and Pierre Lescanne. Counting terms in the binary lambda calculus. unpublished note.

[4] Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[5] D. P. Dirk Pieter Laurie. *Appendix A: Convergence acceleration*, pages 227–261. In [1], 2004.

[6] Naoki Osada. *Acceleration Methods for Slowly Convergent Sequences and their Applications*. PhD thesis, Nagoya University, 1993.