Lazy Processing and Optimization of Discrete Sequences

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen, France
(`mailto:karczma@info.unicaen.fr`
`http://www.info.unicaen.fr/~karczma`)

Abstract

We construct a small arithmetic package operating on discrete sequences: $u_0, u_1, u_2, \ldots$ treated as **single mathematical entities**. Applying functions to sequences by the generalized `map`, or adding them element-wise is quite trivial, but if the sequences are regular, e.g. if they are arithmetic or geometric progressions, a much more compact and in several cases much more efficient representation is possible. For example any polynomial of an arithmetic infinite sequence has an easily coded, and efficiently processed finite representation through its forward differences. Even in more general cases, the lazy manipulation of sequences results in a remarkable compactness of coding, especially where the processing algorithms are iterative. The main motivation of this paper is methodologic.

1 Indexless Sequence Processing and Generation

This is a light-weight essay on practical use of functional programming methods in the domain of *coding* of numeric computations, useful for pedagogy and for concrete applicative work.

Everybody needs sequences. We find them as coefficients of power series and their partial sums, as iterates describing the discrete solutions of differential equations, as the approximants of iterative solutions of algebraic equations, or simply as discrete samples of any function which shall be plotted. Every (or almost) textbook on numerical methods, e. g. [1, 2], see also [3], is "polluted" by indexed sequences: $u_1, u_2, \ldots, u_n, \ldots$ or by iterative formulæ: $u_n \to u_{n+1} = f(u_n)$. This representation is adapted to imperative codes and their typical, indexed data structures: arrays, usually updated in-place.

But sequences map naturally to lists, and their incremental processing using well known higher-order functionals as `map`, `fold`, `zipWith` etc. is the standard protocol of a functional programmer. A function over an equally spaced sequence implemented in Haskell needs just

```
u=[u0, u0+h ..]; y=map sin u
```

and the items resulting from an iterative process are generated by `iterate f u0`, giving $u_0, u_1 = f(u_0), u_2 = f(f(u_0)), \ldots$, etc. There is no loop administration, and the lazy generation can be decoupled from the sequence consuption and analysis (e.g. the verification of the convergence, or the ploting over a finite initial segment).

Such formulae as above are sometimes used to advertize the functional programming, but somebody who *really* needs to do numerical computations will smile, and point out that he would prefer to see a compact and easy implementation of

$$\sin(x + (i+2)h) = 2\sin(x + (i+1)h)\cos(h) - \sin(x + ih) \tag{1}$$

instead, as this form can be computed incrementally, *much* more efficiently: only three calls to trigonometric functions are needed for a sequence of any length. When the profiling of a numerical program shows that the trigonometric functions consume an important part of the processor time, the programmer most probably had been negligent; this is a standard folklore (see e. g. [4]). But the incremental algorithms may be non-trivial to derive even in simple cases, and the fear of introducing bugs is an important psychological factor which hinders the code optimization. Some automatic procedures would be very useful.

So, we shall revise, and reformulate some algorithms over sequences in an incremental fashion. We will exploit the co-recursive data generators (not just co-recursive functions), such as the computation of partial sums of $[u_0, u_1, u_2, \ldots] \to [u_0, u_0 + u_1, u_0 + u_1 + u_2, \ldots]$ through

```
psums (u0:uq)=w where w=u0:zipWith (+) uq w
```

etc. Even easier is the construction of the (very intensely used) forward differential sequence: $\Delta u_n = u_{n+1} - u_n$; it is just the term-wise difference between the tail, and the sequence itself. Some inherently recursive, and a little tortuous algorithms, like the iterated Richardson extrapolation will be reformulated also in such a way. Finally, we will use a little lazy power series package (published elsewhere, see [5]), to construct some approximation schemes based on power series of Δ, say, needing $\Delta/(1+\Delta) = \Delta - \Delta^2 + \cdots$.

Everything has been implemented in Haskell. We have used a modified "algebraic style" Prelude, with the standard numeric type classes replaced by a more abstract algebraic hierarchy: `AddGroup` which defines the additive operators, `Monoid` for the multiplication (and `sqr x = x*x`, etc.), `Group` for division, etc. We omit from the presentation the instance headers and other "administrative" details.

2 Sequences as Algebraic Data Structures

In order to exploit different variants of the organization of sequences, we need data structures a little more heterogeneous than simple lists. We propose the following datatype:

```
data Seq a = Cs a | S a (Seq a) | A a (Seq a)
                  | G a (Seq a)
```

where the tags discriminate between the following variants:

- `(Cs h)` is a constant infinite sequence $[h, h, \ldots]$. It is equivalent to `(S h (S h (S h ...)  ...))`.
- `(S x0 xq)` is a generic sequence $[x_0, x_1, x_2, \ldots]$. But the tail `xq` may have – of course – any form.
- $s =$`(A x0 d)` is a diferential sequence: if d represents $[d_0, d_1, d_2, \ldots]$, then it is $d = \Delta s$, or $s = [x_0, x_0+d_0, x_0+d_0+d_1, \ldots]$. The arithmetic progression $[x, x+h, x+2h, \ldots]$ is represented as `(A x (Cs h))`.
- s =`(G x0 q)` represents a generalized geometric progression; if $q = [q_0, q_1, \ldots]$, then $s = [x_0, x_0q_0, x_0q_0q_1, \ldots]$. We shall use it in restricted contexts, less frequently than other forms.

Note. This paper belongs to a longer series of essays on lazy functional implementation of some mathematical objects used in scientific computing. We have mentioned already the power series, see also the implementation of an univariate differential algebra [6]. Discrete sequences were a natural step forward. While preparing these notes we found the paper [7], whose authors exploit a similar formalism – a compact representation of uniform arithmetic and geometric sequences, and the arithmetic manipulations thereof. Their objective is to optimize some indicial expressions in Computer Algebra by rewriting. They analyze thoroughly the complexity of the optimized algorithms.

2.1 Primitive Functions and Sequence Arithmetic

Our `AddGroup` class defines the addition and the negation. Hencefrom we will omit some clauses which are consequences of the symmetry of some operations. The `AddGroup` instances for sequences are:

```
neg (Cs x) = Cs (neg x)
neg (S x s) = S (neg x) (neg s)
neg (A x r) = A (neg x) (neg r)
neg (G x q) = G (neg x) q

Cs x + Cs y = Cs (x+y)
a@(Cs x) + S y s = S (x+y) (a+s)
Cs x + A y r = A (x+y) r  -- and symm.
A x xr + A y yr = A (x+y) (xr+yr)
S x xq + S y yq = S (x+y) (xq+yq)
p + q = xToS p + xToS q
```

Here `xToS` is a conversion function which expands all abbreviated variants into their generic form. This function is used also for the display – the sequences are first converted into `(S ...)` and then transformed into lists.

Before defining `xToS` we construct specific selectors which retrieve the head and the tail: `sqhd` and `sqtl` of a sequence. The head is trivial, but the tail needs some massaging:

```
sqtl p@(Cs _) = p
sqtl (S _ q) = q
sqtl (A x p) = A (x+sqhd p) (sqtl p)
sqtl (G x q) = G (x*sqhd q) (sqtl q)
```

We may now define the formard differences, partial sums, and the generic conversion operator:

```
fd (Cs _) = Cs (fromInt 0) --Constant seq.
fd (A _ p) = p  -- by definition of A.
fd p@(S _ q) = q-p
fd p@(G x (Cs q)) = (q-1)*>p
fd p = fd (xToS p)

sums p@(Cs x) = A x p
sums (S x q) = w where w = S x (w+q)
sums p = sums (xToS p)

xToS x@(S _ _) = x
xToS (Cs c) = w where w = S c w  -- cyclic
xToS (A x r) = sums (S x r)
xToS p@(G x q) = S x (xToS (sqtl p))
```

Before passing to the multiplication we introduce a useful class `Module` which is a constructor class. For every data structure `T a` the instance of `Module T` defines an infix operation `(*>) ::  a -> T a -> T a`. The expression `x*>p` multiplies by x all the elements of the compound p.

```
instance Module Seq where
  x *> Cs y  = Cs (x*y)
  x *> S y p = S (x*y) (x*>p)
  x *> A y d = A (x*y) (x*>d)
  x *> G y q = G (x*y) q
```

The multiplication is quite simple

```
S x xq * S y yq=S (x*y) (xq*yq) --Generic.
p@(A x xq) * q@(A y yq) =
  A (x*y) (p*yq+xq*q+xq*yq)
Cs x * p = x*>p  -- and symm.
G x xq * G y yq = G (x*y) (xq*yq)
p * q = xToS p * xToS q
```

The only non-trivial but important clause is the multiplication of the differential sequences, in agreement with the formula

$$\Delta(u_n \cdot v_n) = u_n \Delta v_n + \Delta u_n v_n + \Delta u_n \Delta v_n. \qquad (2)$$

This equation shows clearly that for the arithmetic sequences $x =$`(A x0 (Cs h))` all polynomials of x have *closed*, finite forms. The square of `(A x (Cs h))` is equal to `(A x`^{2} `(A 2hx (Cs 2h`^{2}`)))`. The division is easy only if the divisor is constant, or if both arguments are of the `G` type, otherwise the generic conversion is performed first. More complicated functions can obviously – in the last resort – use the `Functor` instance for sequences:

```
fmap f (Cs x) = Cs (f x)
fmap f (S x s) = S (f x) (fmap f s)
-- fmap f p = fmap f (xToS p)
-- Not universal; needs context
```

but in several cases a more efficient procedure is possible, as shown on the following examples:

```
exp (A x r) = G (exp x) (exp r)
sqrt (G x q) = G (sqrt x) (sqrt q)
log (G x q) = A (log x) (log q)
```

and finally the trigonometric functions of arithmetic sequences

```
sin (A x (Cs h)) = p where
   p=S (sin x) q;  q=S (sin(x+h)) r
   r=(2.0*cos h)*>q - p
```

and almost the same formula for `cos (A x (Cs h))`. Some combinatoric sequences take particularly simple shape, for example for $n = 0, 1, 2, \ldots$, $n! =$`(G 1 (A 1 (Cs 1)))`. Its generalization, the Pochhammer symbol (or rising factorial) $x^{(n)} = \Gamma(x+n)/\Gamma(x) = x(x+1)\cdots(x+n-1)$ can be coded as `(G 1 (A x (Cs 1)))`. This is not a rocket science, and any reasonable program will compute such sequences by simple multiplications. The only, but important advantage of such representations is the possibility to code compactly some *transportable data*, and not just algorithm fragments.

3 Some compact formulæ in numerical calculus

The algebra of sequences is fairly universal, and independent of the optimization tricks which work mainly in case of regular sampling. We show here some application examples

3.1 Aitken Extrapolation

A convergent sequence $[x_0, x_1, x_2, \ldots]$ obtained by some iterative procedure, for example the solution of the equation $x = \exp(-x)$ generated by

```
x = sqiter (exp . neg) 1.0 where
  sqiter f u = S u (sqiter f (f u))
```

sometimes converges slowly, we get [1.0, 0.3679, 0.6922, 0.5005, 0.60624, 0.5454, 0.5796, 0.5601, 0.5711, 0.5649, 0.5684,...]. In the case of linear convergence the following procedure

$$x_n \to x'_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} \qquad (3)$$

can be implemented as

```
aitken u = let du = fd u
           in u - du*du/fd du
```

which stabilizes much faster: [0.58223, 0.57171, 0.56864, 0.56762, 0.56730, 0.56719, 0.56716, 0.56715, 0.56714, 0.56714,...]. This is known, of course, our only contribution is the code compactness.

3.2 Wynn process

There exists another transformation [8], relatively weakly known but fairly efficient, which independently of the convergence acceleration can be used for the rationalization (Padéization) of power series. The Wynn process for the sequence u is defined by

$$\begin{aligned} \epsilon_n^{(-1)} &= 0 \\ \epsilon_n^{(0)} &= u_n \\ \epsilon_n^{(k+1)} &= \epsilon_{n+1}^{(k-1)} + \frac{1}{\epsilon_{n+1}^{(k)} - \epsilon_n^{(k)}} \end{aligned} \qquad (4)$$

(Here the sequences $\epsilon^{(2k-1)}$ are auxiliary only.) A strict, indicial algorithm requires a few moments of reflection in order to avoid useless computations. Our lazy algorithm consists in creating a sequence of sequences, and mapping the sequence head selector through it. The solution is $w_n = \epsilon_0^{(n)}$. Here is the whole code (with the omission of some cosmetic `fromInt` conversions), together with the test of the convergence acceleration of the notorious, weakly convergent series $1 - 1/2 + 1/3 - 1/4 + \ldots = \log 2$.

```
xfd p = fmap fd p    -- Auxiliary
wynn u = fmap sqhd e where
  o = A (Cs 0) (xfd o + (recip (xfd e)))
  e = A u (xfd e + (recip (xfd (sqtl o))))

res = wynn (sums ((G 1.0 (-1))*recip(A 1.0 1)))
```

which gives [1.0, 0.7, 0.69333, 0.69315, 0.693147, 0.693147185, 0.693147181, 0.693147181...]. The code is easier to recognize when we separate the even and the odd subsequences, and if the iterative clause of the definition (4) is rewritten as

$$\epsilon_n^{(k+1)} - \epsilon_n^{(k-1)} = \epsilon_{n+1}^{(k-1)} - \epsilon_n^{(k-1)} + \frac{1}{\epsilon_{n+1}^{(k)} - \epsilon_n^{(k)}} \qquad (5)$$

3.3 Differential Equations

The choice among the enormous number of numerical equation solvers is rarely dictated by its elegance; the precision, stability and complexity are more important criteria, but a handy and compact formula which yields the *whole* numeric, iterative solution of an equation might be very useful for testing.

We show how to code the order 2 Runge-Kutta algorithm for $y(t)$ fulfilling the equation $y' = f(y,t)$, with $y(t_0) = y_0$. Its usual presentation goes as follows:

$$\begin{aligned} k_1 &= h \cdot f(y_n, t_n) \\ k_2 &= h \cdot f(y_n + k_1/2, t_n + h/2) \\ y_{n+1} &= y_n + k_2 \end{aligned} \qquad (6)$$

If the function f can be lifted from numbers to sequences, which can be *always* done with the appropriate declaration of arithmetic instances, the entire solution of this equation is given by y defined through

```
t  = A t0 (Cs h)
k1 = 0.5*h *> f t y
y  = A y0 (h *> f (y+k1) (t+0.5*h))
```

For example, when $f(t,y) = -ty$, we get the approximation of the full trajectory $y = \exp(-t^2/2)$ without further work.

3.4 Evaluation of Continued Fractions

According to [4] the best general method for evaluation of continued fractions

$$f = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots}}}, \qquad (7)$$

is the Lentz algorithm [9], much more stable than the Wallis iterative formula invented 300 years ago. It goes as follows:

```
Set $f_0 = b_0; C_0 = f_0; D_0 = 0$;
for $j = 1, 2, \ldots$
    Set $D_j = 1/(b_j + a_j D_{j-1})$
    Set $C_j = b_j + a_j/C_{j-1}$
    Set $f_j = f_{j-1}(C_j D_j)$
    Stop when $C_j D_j$ becomes almost equal to 1.
```

with some security valves preventing the division by zero. Our code, together with this guard is shorter than the algorithm presentation:

```
cfeval b0 b a =
  let enh x = if x==0.0 then 1.0e-35 else x
      tiny s = fmap enh s
      d = S 0.0 (recip (tiny (b+a*d)))
      c = tiny (S b0 (b+a/c))
  in G (enh b0) (sqtl (c*d))
```

and for the exponential funtion e^z for $z = 1$

$$e^z = 1 + \cfrac{z}{1 - \cfrac{z}{2 + \cfrac{z}{3 - \cdots}}}, \tag{8}$$

we get the code

```
aa = S 1.0 (neg aa)
bb = S 1.0 (S 2.0 (tw + bb))
 where  tw = S 2.0 (S 0.0 tw)

res = cfeval 1.0 bb aa
```

yielding [1.0, 2.0, 3.0, 2.75, 2.714, 2.7179, 2.71831, 2.718283, 2.71828172, 2.71828182, 2.71828183,…]. The algorithm works for classical continued fraction expansions which begin with $b_0 = 0$, for example for $\tan(x)$.

4 Romberg Integration

If a function is uniformly sampled within an interval, the natural integration algorithms are based on various Newton-Cotes schemes, for example the trapezoidal or Simpson rule. But we can do much better without augmenting the number of sampled points. If a quantity I depends on the size of the discretization sub-interval h: $I(h) = I_0 + \zeta h^p + \ldots$, a better approximation to $\lim_{h\to 0} I(h)$ is obtained by the popular Richardson iterated extrapolation scheme, whose first step here would be the subtraction

$$I \to \left(2^p I(h/2) - I(h)\right) / \left(2^p - 1\right) \tag{9}$$

which eliminates the ζ term. The knowledge of the leading power of h in the remaining (error) terms permits the iteration of this procedure. The Simpson rule is equivalent to the trapezoidal algorithm iterated once.

4.1 Adaptive trapezoidal rule

We begin thus with the generation of an infinite sequence of approximants for

$$\int_a^b f(x)dx = h(f(a)/2 + f_1 + \ldots + f_{n-1} + f(b)/2) \tag{10}$$

where $h = (b - a)/n$, with varying $n \to 2n$. Of course, when augmenting the density of points we reuse all the former computations. The code goes as follows, beginning with a simple function which yields the sum of n terms of a sequence. (It is redundant, but we don't want to propose code which uses `sums`, and needs an explicit deforestation.)

```
sumn n s = sn n (sqhd s) (sqtl s) where
  sn 1 x _ = x
  sn m x p = sn (m-1) (x+sqhd p) (sqtl p)
```

```
trapez a b f =
  let intr h n =
        let hp=0.5*h
        in S (sumn n (f (A (a+hp) (Cs h))))
            (intr hp (2*n))
      h0=b-a
  in (sums (S (0.5*sumn 2 (f (S a (Cs b)))) --ugh!
             (intr h0 1)))*(G h0 (Cs 0.5))
```

It is a bit ugly, but reasonable. $\int_0^3 \exp(-2x)dx$ produces [1.5, 0.83, 0.589, 0.522, 0.5046, 0.5002, 0.4991, 0.49885, 0.498783, 0.498766, 0.498762, 0.498761,…]. (Of course, the function f should be written so as to enable its lifting to the sequence domain.)

4.2 Implementation of the iterated Richardson Scheme

It can be proven that the convergence of the trapezoidal rule is: $I(h) = I_0 + \zeta_1/n^2 + \zeta_2/n^4 + \zeta_3/n^6 + \ldots$. In (9) $p = 2$. Normally there is no point in starting with $n = 1$, the scheme is valid when h is small, and the expansion of $I(h)$ has some sense.

The final coding of the Romberg iterative scheme is very simple and compact. The sequence of approximants u passes through the following generator:

```
rombg c u = S (sqhd w) (rombg (4.0*c) w)
  where w = (1.0/(c-1.0))*>(c*>sqtl u - u)
```

with the initial value of c equal to 4. (Of course, if we begin with a quadrature which converges faster, we choose the appropriate $c = 2^p$.) For the example above, we get [0.6, 0.50, 0.49882, 0.498760862, 0.498760624, 0.498760624,…]. 32 subdivisions suffice to get the precision equivalent to more than 4000 in the trapezoidal case, and the algorithm is robust.

5 Symbolic Calculus of Finite Differences

Many useful formulae in processing of power series, or finite difference schemes in differential equation solving are obtained through a formal, symbolic treatment of the forward difference operator and some other related entities. We might define: $Eu : Eu_n = u_{n+1}$, and formally we may write $E = 1 + \Delta$. The Euler acceleration formula for the alternating power series may be derived as follows

$$\begin{aligned}\sum_{n=0} (-z)^n u_n &= \frac{1}{1+zE} u_0 = \frac{1}{1+z} \frac{1}{1 + z\Delta/(1+z)} u_0 \\ &= \frac{1}{1+z} \sum_{k=0} \left(\frac{-z}{1+z}\right)^k \Delta^k u_0, \end{aligned} \tag{11}$$

which can be coded in three lines. We may imagine a little more involved example, where we have to sum not just numbers: $\Delta^k u_0$, but whole sequences: $s = \sum_{k=0} a_k \Delta^k u$. If the coefficients a_k form a list, then we have to choose the upper limit m of the summation, and apply `sumn m` to

```
sseq (a0:aq) u = S (a0*>u) (sseq aq (fd u))
```

This can be useful for the interpolation of uniformly spaced tabulated functions by people who hate searching textbooks for interpolation formulae. If $u_n = u(x_0 + nh)$, then $u(x_0 + (n + \kappa)h) = E^\kappa u_n = (1 + \Delta)^\kappa u_n$, and we need to apply the binomial series of Δ to u, truncated at the desired degree of the interpolating polynomial. The same formula applies for the extrapolation, for example we can compute the element u_{-1} as $(1+\Delta)^{-1}u_0$, which, when truncated at $m=2$ gives the quadratic extrapolation $u_{-1} = 3u_0 - 3u_1 + u_2$, etc.

More intricate examples need sometimes a few trade tricks. In order to compute the derivative of a tabulated function, it is *much* better to use the symmetric difference operator $\delta u(x) = u(x+h/2) - u(x-h/2)$, than the forward differences. We can use the symbolic form of the Taylor expansion: $E = \exp(hD)$, where D is the differentiation operator, to conclude that

$$\frac{h}{2}D = \log\left(\frac{\delta}{2} + \sqrt{1+\delta^2/4}\right) \tag{12}$$

which is, as expected, an odd function of δ. This is awkward. We know how to apply effectively $\delta^2 = E - 2 + E^{-1}$ to a sequence (adopting some extrapolation scheme below u_0). However, after having developed the series (12) – e. g. using the lazy power series package presented in [5], we multiply it by

$$\frac{\mu}{\sqrt{1+\delta^2/4}} \quad \text{where} \quad \mu u(x) = \frac{1}{2}\left(u\left(x+\frac{h}{2}\right) + u\left(x-\frac{h}{2}\right)\right) \tag{13}$$

which is equal to 1. But then the series is in δ^2, multiplied by $\mu\delta = 1/2(E - E^{-1})$, and the technique becomes effective, and fairly efficient.

6 Conclusions

Serious numerical computations demand usually the fastest possible algorithms, and the elegance of programming is often relegated to the second plan. Functional techniques will lose with brutal, but more efficient, lower level imperative coding, implemented using arrays. However, when we have to *teach and test* some tortuous recursive and/or iterative algorithms dealing with differential equations, evaluation of power series, etc., human resources *are* important even if we know how to administrate the memory allocation for our arrays, which is not always the case. We have shown how to economize the human time, and how to take a pleasure at writing numerical codes, which is usually considered to be a very boring exercice. Our examples are simple, but they are *not* toys. All algorithms presented are practically useful, and they are coded without sacrificing the engineering efficiency.

Our code is sometimes slightly frenzy, but this is obviously not the main issue, apart from the psychodramatic goal of this talk... Lazy functional programming shortens the paradoxal gap between the standard mathematical textbooks full of inductively defined data objects, and the practice of programming for engineers or physicists, where even recursive *functions* are not always well accepted. A good deal of these recursive functions, and all the classical loop administration, together with the iteration/recursion stop criteria are eliminated through the usage of co-recursive data.

We noticed that the laziness *is* a serious psychological obstacle for a traditional user of the numerical sofware. The separation between the generating process, and the iteration stopping criteria, which are entirely left for the consumer module, needs not only good compilers of lazy languages, but some mental adjustment of the programmer. This seems to be more delicate than the "standard" disinclination to use the APL-style generalized folds by people formed on "C" and similar languages.

References

[1] Germund Dahlquist, Åke Bjrck, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, (1974).

[2] R. Bulirsch, J. Stoer, *Introduction to Numerical Analysis*, Springer, N.Y., (1980).

[3] Donald E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, (1981).

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical recipes in "C". The Art of Scientific Computing*, 2nd ed., Cambridge Univ. Press, (1993).

[5] Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*, Theoretical Computer Science **187**, (1997), pp. 203–219.

[6] Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Proceedings of the III ACM SIGPLAN International Conference on Functional Programming, Baltimore, (1998), pp. 195–203.

[7] Olaf Bachmann, Paul S. Wang, Eugene V. Zima, *Chains of Recurrences – a Method to Expedite the Evaluation of Closed-Form Functions*, ACM Proc. of the International Conference on Symbolic and Algebraic Computation (ISSAC), ACM Press, Oxford, July 1994, pp. 242 – 249.

[8] P. Wynn, *On a Device for Computing the $e_m(S_n)$ transformation*, MTAC **10**, (1956), pp. 91 – 96.

[9] W.J. Lentz, *Applied Optics*, **15**, (1976), pp. 668 – 671.