# Traitement paresseux et optimisation des suites numériques

Jerzy Karczmarczuk

*Dept. d'Informatique, Université de Caen, France*
*(mailto:karczma@info.unicaen.fr)*

**Résumé**

Nous montrons la construction d'un petit paquetage qui opère sur des suites discrètes : $u_0, u_1, u_2, \ldots$, traitées comme des entités mathématiques à part entière, avec leur propre algèbre. Le paquetage est construit en Haskell. Si les suites sont régulières – p. ex. si elles constituent des progressions arithmétiques ou géométriques, ou si elles sont composées de fragments réguliers, on peut les représenter de manière plus compacte que les listes traditionnelles, et leur traitement peut être optimisé de façon assez significative. Nous montrons comment coder par une forme finie un polynôme quelconque d'une progression arithmétique infinie, et comment exploiter le formalisme des opérateurs de différences. Même là, où le problème est trop complexe pour l'optimisation, le codage paresseux est la source d'une compacité remarquable de code, surtout pour les algorithmes itératifs. Notre motivation première est méthodologique, nous pensons que les algorithmes numériques constituent une intéressante base applicative et pédagogique pour la programmation fonctionnelle, et d'autre part, l'enseignement des techniques numériques à l'aide des langages et algorithmes fonctionnels est beaucoup plus élégant et sûr que les méthodes traditionnelles, parfois difficiles à déboguer. Nos exemples sont simples, sans être élémentaires. Une certaine connaissance de méthodes numériques sera utile au lecteur.

## 1. Génération et traitement fonctionnel des suites

Dans le domaine des calculs numériques (et algébriques en général) tout le monde a besoin de suites. Elles constituent les sommes partielles des séries numériques, ou les coefficients des séries entières formelles $\sum_n u_n x^n$, ou les échantillons discrets des fonctions: $u_n = f(x_0 + nh)$, ou les solutions numériques des équations différentielles, ou les approximations des solutions itératives des équations algébriques, etc. Pratiquement tout livre sur les techniques numériques, par exemple [1, 2], éventuellement [3] exploite les suites indexées: $u_1, u_2, \ldots, u_n, \ldots$, et les formules récursives/itératives, également indexées: $u_n \to u_{n+1} = p(u_n)$ pour la présentation et l'analyse de presque tous les algorithmes. Les suites appartiennent au monde mathématique depuis de siècles, et après la naissance de méthodes informatiques leur implantation par des tableaux et manipulation par des codes impératifs était parfaitement naturel. La présentation des algorithmes numériques à l'aide des suites indexées est également considérée standard.

Mais les suites ou séquences – étant des ensembles ordonnés – s'expriment également très naturellement par des listes, et leur traitement incrémental par les fonctionnelles classiques connues en Haskell sous les noms de `map`, `foldr`, `zipWith`, `iterate` etc., et présentes dans d'autres langages comme OCaml sous d'autres noms: `fold_right` ou `map2`.

Pour les listes infinies ces définitions prennent les formes :

```
map f (x:xq) = f x : map f xq
zipWith op (x:xq) (y:yq) = (op x y) : zipWith op xq yq
foldr f z (x:xq)  = f x (foldr f z xq)
```

```
iterate f x = x : iterate f (f x)  -- [x,f x,f(f x),f(f(f x)),...]
```

mais nous allons définir des opérations équivalentes pour d'autres structures de données. Les fonctions d'ordre supérieur appartient à la panoplie standard d'un programmeur fonctionnel. Pour implanter en Haskell une transformation fonctionnelle par $f$ d'une suite uniforme ($u_{n+1} = u_n + h$ où $h$ est une constante), demande seulement l'écriture de

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

Les items résultant de la réalisation d'un processus itératif $u_{n+1} = p(u_n)$ sont engendrés par la fonction standard `iterate p u0`, dont l'appel crée la liste $[u_0, u_1 = p(u_0), u_2 = p(p(u_0)), \ldots]$, etc. Il n'y a pas d'administration des boucles « **for** », et la génération paresseuse peut aisément être séparée de la consommation de la suite par d'autres modules (l'intégration numérique, l'analyse de la convergence, ou l'affichage textuel ou graphique d'un segment fini).

Les formules de ce genre sont souvent utilisées pour démontrer la compacité du code fonctionnel, et pour la « publicité » de la méthodologie fonctionnelle, avec ses structures de données dynamiques et le protocole paresseux en général. Cependant, un utilisateur sérieux qui apprécie l'élégance, mais surtout l'efficacité, peut sourire, et remarquer qu'il aurait *plutôt* besoin de l'implantation facile et élégante d'une formule comme celle-ci :

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

puisque une suite trigonométrique uniforme $u_n = \sin(x_0 + nh)$ peut être calculée bien plus rapidement en exploitant l'incrémentalité. En effet, seulement trois appels des fonctions trigonométriques sont nécessaires pour une suite de longueur arbitraire. Si le profil d'une application numérique montre que les fonctions élémentaires qui satisfont des récurrences connues, comme l'exponentielle ou les fonctions trigonométriques, consomment une tranche importante du temps d'exécution, ceci est une évidence très forte que le programmeur a été négligent : ceci est le « folklore » standard, mentionné p. ex. dans [4]. Cependant, si l'algorithme se complique, l'optimisation incrémentale peut ne pas être facile, et la peur d'introduire des erreurs constitue un facteur psychologique important qui empêche une telle démarche. Quelques mécanismes automatiques seraient fort utiles.

Bien sûr, nous sommes loin de sous-estimer l'importance de la programmation impérative traditionnelle. Le coût de l'allocation dynamique de mémoire est réel. Le stockage des objets représentant les calculs différés influence aussi l'efficacité des programmes. Mais dans cet article nous sommes plus concernées par les aspects méthodologiques de la programmation.

Nous allons donc réviser et reformuler quelques algorithmes de traitement de suites de manière incrémentale dans le cadre de la programmation fonctionnelle paresseuse, ce qui rendra le codage plus statique et déclaratif. Nous allons utiliser quelques générateurs co-récursifs de données (et non pas seulement des fonctions co-récursives). Par exemple, le calcul de sommes partielles de $u$ :

$$s_n = \sum_{k=0}^{n} u_n \qquad \text{pour} \quad n = 0, 1, \ldots \tag{2}$$

peut être réalisé par

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

où l'on voit que `w` est une variable définie co-récursivement.

Il est encore plus facile de construire les opérateurs qui calculent les différences finies, définies comme: $\Delta u_n = u_{n+1} - u_n$; ceci n'est rien d'autre que la différence élément par élément (construite avec `zipWith (-)` de la queue, et de la suite elle-même ; la définition complète se trouve dans la section (3)).

Quelques algorithmes connus et utiles, mais relativement intriqués, comme l'extrapolation itérée de Richardson seront reformulés de cette manière.

Nous allons également profiter d'un petit paquetage de traitement paresseux des séries formelles, publié dans [5]), pour élaborer quelques schémas d'approximation permettant de manipuler les fonctions d'opérateurs agissant sur les suites, comme $\Delta/(1+\Delta) = \Delta - \Delta^2 + \Delta^3 \cdots$ et quelques cas plus complexes.

Notre code a été écrit en Haskell, et les tests ont été effectués sous l'interprète Hugs. Ce système après le démarrage charge la librairie standard, le *Prélude* qui contient les fonctions, et les *classes* prédéfinies. Le système de classes en *Haskell* qui permet de définir les fonctions surchargées est pour nous très important, car nous voulons que le structures de données représentant les suites possèdent toutes les propriétés algébriques indispensables.

Nous avons défini et utilisé un Prélude algébrique modifié, où la classe de nombres `Num` et ses sous-classes (`Fractional`, `Floating`, etc.) ont été remplacées par une hiérarchie algébrique plus abstraite, avec des entités comme `AddGroup` qui spécifie les opérateurs additifs, `Monoid` avec la multiplication et ses dérivés comme l'élevation au carré, etc. Ceci nous a permis de déclarer que les suites sont des entités arithmétiques, sans les spécifier comme des « nombres », ce que nous paraissait méthodologiquement inapproprié.

L'opérateur de division est déclaré dans la classe `Group` qui est une sous-classe du `Monoid`.

La classe `Module` déclare un opérateur de multiplication spéciale `(*>)` d'une structure composite (suite, vecteur, polynôme) par un élément de base (un « nombre », coefficient du polynôme, etc.).

## 2. Structures de données algébriques représentant les suites

Nous voulons *optimiser* la gestion des suites dans des cas réguliers, et pour pouvoir exploiter quelques variantes spécifiques d'organisation, nous avons besoin de structures de données un peu plus hétérogènes que des listes standard. Nous proposons le type récursif linéaire comme ci-dessous:

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

où les balises (constructeurs `Cs, S, A, G`) discriminent entre les variantes suivantes:

- `(S x0 xq)` est une suite générique $[x_0, x_1, x_2, \ldots]$, dont la tête est égale à $x_0$, et la queue – $x_q = [x_1, x_2, \ldots]$. Ceci est une structure qui peut être isomorphe à une liste (infinie) d'éléments si elle est homogène: `S x0 (S x1 (S x2 ...))`, mais sa queue peut s'instancier par d'autres possibilités, par example `(S x (A y (Cs z)))` .
- `(Cs h)` est une suite *constante*: $[h, h, h \ldots]$. Ceci est une structure finie, considérée équivalente à la suite infinie `(S h (S h (S h ...) ...)`. Peut être utilisée pour terminer les suites finies si on adopte la convention que $[x_0, \ldots, x_n]$ est équivalent à $[x_0, \ldots, x_n, x_n, x_n, \ldots]$.
- $s$ `=(A x0 d)` est une représentation spéciale, une *séquence différentielle.* Si $d$ représente $[d_0, d_1, d_2, \ldots]$, alors $d = \Delta s$, ou bien $s = [x_0, x_0 + d_0, x_0 + d_0 + d_1, \ldots]$. Cette structure représente la suite des sommes partielles définies par la valeur initiale $x_0$, et les incréments formant la suite $d$. La progression arithmétique (infinie) classique $[x, x+h, x+2h, \ldots]$ sera représentée par `(A x (Cs h))`.
- $s$ `=(G x0 q)` représente une progression géométrique généralisée; si $q = [q_0, q_1, \ldots]$, alors $s = [x_0, x_0 q_0, x_0 q_0 q_1, x_0 q_0 q_1 q_2 \ldots]$. Elle sera très utile, mais dans des contextes restreints, moins fréquents que la suite différentielle, immédiatement utilisable pour l'échantillonnage uniforme d'une fonction.

**Note.** Ce texte appartient à une suite d'essais sur l'application des techniques fonctionnelles paresseuses à la description et implantation de quelques objets mathématiques utilisés dans le calcul scientifique et dans son enseignement. Nous avons mentionné déjà le traitement des séries, le lecteur peut aussi trouver quelques parallèles entre l'approche actuel, et notre codage des algorithmes de différentiation algorithmique publié dans [6]. Le traitement des suites est une continuation naturelle de cette philosophie, mais ce n'est pas une découverte fondamentale. Pendant la rédaction de ces notes nous avons découvert l'article [7], dont les auteurs ont également discuté un formalisme similaire, une représentation compacte des suites arithmétiques et géométriques, et leur manipulation algébrique. Mais ils travaillaient dans le cadre du Calcul Formel (symbolique), et leur objectif était d'utiliser les techniques de réécriture pour optimiser les expressions indexées, ainsi que l'analyse de la complexité de leurs algorithmes. Notre approche est plus directe et plus universelle, mais sans ambition dans le domaine de l'algorithmique théorique.

## 3. Fonctions primitives et arithmétique des suites

Il n'est pas nécessaire de présenter ici la description complète de nos classes algébriques, mais une courte introduction peut être utile pour comprendre comment nous avons surchargé les opérateurs arithmétiques. Voici quelques classes minimalistes, où le lecteur reconnaîtra quelques méthodes récupérées de la classe `Num` du Prélude Haskell orthodoxe et de ses dérivés, mais aussi quelques nouveautés, comme la reconnaissance du fait qu'un Groupe Additif est automatiquement un Module sur les entiers. Ceci *ne remplace vraiment pas* la classe `Num`; les convertisseurs standard comme `fromInt`, `fromDouble` etc. qui sont déclarées dans `Num` et qui permettent l'injection des nombres entiers ou flottants dans n'importe quel domaine algébrique défini par l'utilisateur, sont partiellement intégrés au noyau du compilateur (ce que nous considérons comme une erreur de conception). Mais Haskell n'est pas un langage de Calcul Formel. Plusieurs lois algébriques comme le fait que *tout* Anneau, possédant l'unité et l'addition, étend automatiquement le domaine des entiers, ne sont pas reconnues par le système de types, tout doit être spécifié explicitement par le programmeur.

Tout type de données `a` qui représente des objets mathématiques suffisamment riches en propriétés algébriques, sera l'instance des classes suivantes :

```
class AddGroup a where
 mZero        :: a         -- un 'Zéro' générique
 (+),(-),subt :: a->a->a
 neg          :: a->a      -- négation
 x-y = x + neg y           -- déf. cycliques : (-) et neg
 neg x = mZero-x           -- l'instance DOIT rédéfinir une d'eux
 abs          :: a->a
 signum       :: a->a
 subt = flip (-)           -- déf.: flip f x y = f y x
 (#)          :: Int->a->a -- multiplication par les entiers
 n # x = iterOp (+) n x mZero

-- "Puissance" d'un opérateur par itération linéaire
-- iterOp :: (a -> a -> a) -> Int -> a -> a -> a
iterOp _  0 _ buf = buf
iterOp op n x buf = iterOp op (n-1) x (x `op` buf)

class Monoid a where
 mOne :: a                -- un 'Un' générique
 (*)  :: a->a->a          -- multiplication interne
 sqr  :: a->a
 sqr x = x*x              -- carré générique

class (Monoid a)=>Group a where
```

```
 (/)    :: a->a->a
 recip  :: a->a
 recip x = mOne/x         -- définitions par défaut cycliques...
 x/y     = x*recip y      -- l'instance DOIT rédéfinir une !
```

Les structures `Seq` instancient ces classes et quelques autres. Désormais nous allons omettre des définitions citées les clauses triviales, résultant par exemple de la commutativité de l'addition, etc. Pour le Groupe Additif nous aurons:

```
instance (AddGroup a, Monoid a)=>AddGroup (Seq a) where
 mZero = Cs mZero
 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    -- et symétriquement.
 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
```

La dernière clause pour l'addition exprime le fait que pour ajouter deux progressions géométriques, il faut le faire explicitement, même si quelques identités remarquables sont parfois exploitables. La fonction `xToS` est un convertisseur qui développe les formes abrégées et les transforme en leur variantes génériques. Elle sera utilisée pour l'impression « standard » – les suites sont converties d'abord en `(S ...)`, et ensuite transformées en listes qui peuvent être affichées par des procédures standard.

Avant de définir `xToS`, spécifions les sélecteurs qui récupèrent la tête (`sqhd`) et la queue (`sqtl`) d'une suite. La tête est triviale pour toutes les variantes d'une structure `Seq` , mais la construction de la suite restante demande un procédé plus laborieux:

```
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)
```

Avec ces utilitaires nous pouvons définir les différences discrètes (avancées, $\texttt{fd} = \Delta$, on verra plus tard les différences symétriques), les sommes partielles, etc.:

```
fd (Cs _) = Cs (fromInt 0)   -- Suite constante.
fd (A _ p) = p               -- par définition de A.   Cool, non?
fd p@(S _ q) = q-p           -- Déf. générique, déjà mentionnée
fd p@(G x (Cs q)) = (q-1)*>p -- Multiplication dans Module
fd p = fd (xToS p)           -- Tous les cas restants

-- Sommes partielles
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  -- structure "infinie" cyclique
xToS (A x r) = sums (S x r)
xToS p@(G x q) = S x (xToS (sqtl p))
```

Nous avons introduit une classe représentant la structure mathématique `Module` qui est une *classe de constructeurs*. Rappelons qu'en Haskell une classe est une collection (abstraite) de *types* qui possèdent quelques propriétés communes. Ainsi le type `(Seq a)` où `a` est le type des éléments de la suite, peut appartenir à la classe `Group`, etc. Mais en toute indépendance de ce type `a`, le constructeur `Seq` lui-même peut aussi appartenir à une classe. Par exemple, la fonctionnelle `map` qui applique une fonction à tous les éléments d'une liste, est une instance d'une méthode plus abstraite `fmap` qui appartient à la classe de constructeurs `Functor`. Ici :

```
instance Functor Seq where
 fmap f (Cs x) = Cs (f x)
 fmap f (S x s) = S (f x) (fmap f s)
```

Pour d'autres variantes l'applicateur universel ne peut être défini. Un autre type composite, par exemple `(Arbre a)` admet la fonction `fmap` qui applique récursivement une fonction à tous les noeuds d'un arbre, si le constructeur `Arbre` est déclaré comme une instance de la classe `Functor`.

Pour toute structure de données `T a`, l'instance de `Module T` définit une opération binaire infixe `(*>)`. L'expression `x*>p` multiplie par $x$ tous les composants de la structure composite $p$.

```
class Module t where
 (*>) :: (Monoid a, AddGroup a) => a->t a->t a

instance Module Seq where
  x *> Cs y  = Cs (x*y)          --  fmap (x*) arg_2
  x *> S y p = S (x*y) (x*>p)    --  idem
  x *> A y d = A (x*y) (x*>d)
  x *> G y q = G (x*y) q
```

La multiplication des suites (élément par élément) est définie dans la classe `Monoid` (nous n'avons pas besoin des Semi-groupes sans unité) :

```
instance (AddGroup a, Monoid a) => Monoid (Seq a) where
 mOne = Cs mOne
 S x xq * S y yq=S (x*y) (xq*yq)   -- Générique.
 p@(A x xq) * q@(A y yq) = A (x*y) (p*yq + xq*q + xq*yq)
 Cs x * p = x*>p                   -- = p * Cs x.
 G x xq * G y yq = G (x*y) (xq*yq)
 p * q = xToS p * xToS q           -- Sinon, conversion générique
```

La multiplication des suites différentielles `A`, est l'implantation de la formule

$$\Delta(u_n \cdot v_n) = u_n \Delta v_n + \Delta u_n v_n + \Delta u_n \Delta v_n. \tag{3}$$

Le carré de `(A x (Cs h))` est égal à (`A` $x^2$ (`A 2hx` + $h^2$ (`Cs` $2h^2$))). En effet, si $\Delta u = h$, alors $\Delta u^2 = 2hu + h^2$. L'algèbre polynomiale générée par les suites arithmétiques representées par $u =$`(A x (Cs h))` est close, c'est-à-dire, tous les polynômes de $u$ possèdent des formes finies.

La division est simple seulement si le diviseur est une constante, où si les deux arguments appartiennent à la variante géométrique `G`, sinon la conversion générique est effectuée d'abord.

L'application d'une fonction quelconque à une suite générique utilise l'applicateur `fmap`: `exp u@(S _ _) = fmap exp u`, etc., mais ceci est justement l'approche qui doit être évitée tant que possible. Dans de nombreux cas, les fonctions élémentaires peuvent être spécifiées de façon plus efficace, par exemple:

```
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)
```

où la sémantique des variantes s'adapte bien aux récurrences satisfaites par telle ou telle fonction. Les fonctions trigonométriques sont légèrement plus longues, et utilisent les identités :

$$\sin(x+2h) = 2\cos(h)\sin(x+h) - \sin(x)\,, \quad (4)$$
$$\cos(x+2h) = 2\cos(h)\cos(x+h) - \cos(x)\,. \quad (5)$$

```
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

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

Quelques suites combinatoires prennent aussi des formes simplifiées, parfois closes, par exemple pour $n = 0, 1, 2, \ldots$, $n! =$ `(G 1 (A 1 (Cs 1)))`.

La généralisation de la suite factorielle, le symbole de Pochhammer: $x^{(n)} = \Gamma(x+n)/\Gamma(x) = x(x+1)\cdots(x+n-1)$ peut être codée comme `(G 1 (A x (Cs 1)))`. Tout programme impératif raisonnable utilisera aussi des récurrences et la multiplication itérée pour calculer ces suites. L'avantage de la formulation fonctionnelle est la possibilité de coder de manière compacte des séquences numériques *complètes*, et non pas seulement des fragments d'algorithmes. Le code devient plus orienté vers le style de programmation par flots de données. Nous pensons que ceci augmente la modularité des programmes.

# 4. Formulation compacte de quelques formules numériques

Les optimisations présentées ci-dessus sont des « trucs » qui fonctionnent surtout dans le cas de l'échantillonnage régulier (progressions arithmétiques). Toutefois, l'algèbre de suites est assez universelle, ce qui facilite le codage des algorithmes en général. Passons donc aux quelques exemples d'applications concrètes.

## 4.1. Extrapolation d'Aitken

Les suites représentent souvent les résultats partiels d'un processus itératif: $x_{n+1} = f(x_n)$, par exemple les approximants de la solution itérative d'une équation algébrique $x = f(x)$. Nous pouvons générer une telle suite par la fonctionnelle `sqiter` :

```
sqiter f x0 = S x0 (sqiter f (f x0))
```

qui est l'analogue de la fonctionnelle `iterate`, définie dans le domaine des listes, et génère `(S x0 (S (f x0) (S (f(f x0)) ...)))`. Par exemple, la suite qui est la solution de l'équation algébrique $x = \exp(-x)$ avec $x_0 = 1$ peut être codée tout simplement comme

```
x = sqiter (exp . neg) 1.0
```

Le résultat est $x$ = [1.0, 0.3679, 0.6922, 0.5005, 0.60624, 0.5454, 0.5796, 0.5601, 0.5711, 0.5649, 0.5684,. . . ].

La convergence d'un tel processus peut être très lente, et les techniques d'accéleration appartiennent à l'arsenal standard de tous les numériciens. Si la convergence est linéaire : $\Delta u_{n+1} \approx \alpha \Delta u_n$, une des techniques connues et populaires, celle de Aitken a prouvé son efficacité dans des cas typiques.

Sa définition formelle

$$x_n \to x'_n = x_n - \frac{(x_{n+1} - x_n)^2}{x_{n+2} - 2x_{n+1} + x_n} , \tag{6}$$

peut être implantée par la procédure

```
aitken x = let dx = fd x in x - dx*dx/fd dx
```

ce qui stabilise la solution beaucoup plus rapidement, ici: [0.58223, 0.57171, 0.56864, 0.56762, 0.56730, 0.56719, 0.56716, 0.56715, 0.56714, 0.56714,...]. Il serait probablement difficile de trouver une implantation plus courte de cet algorithme.

## 4.2. Processus de Wynn

Il existe une autre transformation plus récente, [8], qui reste relativement peu connue, mais qui est assez efficace. Elle peut – indépendamment de l'accéleration de convergence – être utilisé pour la rationalisation (Padéisation) des séries. Le procédé de Wynn pour la suite $u$ est défini par

$$\begin{aligned} \epsilon_n^{(-1)} &= 0 \\ \epsilon_n^{(0)} &= u_n \\ \epsilon_n^{(k+1)} &= \epsilon_{n+1}^{(k-1)} + \frac{1}{\epsilon_{n+1}^{(k)} - \epsilon_n^{(k)}} , \quad k = 1, 2, \ldots \end{aligned} \tag{7}$$

où les suites $\epsilon^{(2k-1)}$ sont accessoires; seuls les termes d'indice supérieur pair constituent les convergents. La dernière clause peut être réformulée de manière suivante :

$$\epsilon_n^{(k+1)} - \epsilon_n^{(k-1)} = \epsilon_{n+1}^{(k-1)} - \epsilon_n^{(k-1)} + \frac{1}{\epsilon_{n+1}^{(k)} - \epsilon_n^{(k)}} , \tag{8}$$

ce qui permet d'utiliser l'opérateur des différences discrètes `fd`. Les objets $\epsilon$ possèdent deux indices, alors notre procédure consiste à créer une suite de suites. En fait, nous en construirons deux, séparant les convergents $c^{(k)} \equiv \epsilon^{(2k)}$, et les suites accessoires $a^{(k)} \equiv \epsilon^{(2k-1)}$. Il suffit ensuite de récupérer les têtes (termes avec l'indice inférieur égal à zéro) de la suite des convergents, et la solution est $w_k = c_0^{(k)}$. La formule itérative (7) exprimée par $c$ et $a$ prendra la forme

$$\begin{aligned} c_n^{(k+1)} - c_n^{(k)} &= c_{n+1}^{(k)} - c_n^{(k)} + \frac{1}{a_{n+1}^{(k+1)} - a_n^{(k+1)}} , & (9) \\ a_n^{(k+1)} - a_n^{(k)} &= a_{n+1}^{(k)} - a_n^{(k)} + \frac{1}{c_{n+1}^{(k)} - c_n^{(k)}} . & (10) \end{aligned}$$

Voici cette solution, avec l'ommission de quelques conversions `fromInt` ou `fromFloat` cosmétiques. Pour tester la convergence nous avons pris la suite harmonique (alternante) dont le taux de convergence est très faible : $1 - 1/2 + 1/3 - 1/4 + \ldots = \log 2$.

```
xfd p = fmap fd p     -- Auxiliaire
wynn u = fmap sqhd e where
  c = A u (xfd c + (recip (xfd (sqtl a))))    -- k pair
  a = A (Cs 0) (xfd a + (recip (xfd c)))      -- k impair

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

ce qui donne [1.0, 0.7, 0.69333, 0.69315, 0.693147, 0.693147185, 0.693147181, 0.693147181...].

## 4.3. Équations différentielles

Le choix parmi les innombrables quadratures numériques des équations différentielles est rarement dicté par leur élégance. Ce qui compte vraiment ce sont la précision, la stabilité, et la complexité générale. Ces attributs sont parfois difficiles à évaluer. Par conséquent, pour les tests, et surtout pour le prototypage, la possibilité d'écrire un code simple et portable, facilement « injectable » dans une procédure de simulation etc., ne doit pas être sous-estimée.

Nous montrons comment coder l'algorithme de Runge-Kutta d'ordre 2 (mais facile à étendre) pour $y(t)$ satisfaisant l'équation

$$y' = f(y,t)\,, \qquad \text{avec} \quad y(t_0) = y_0. \tag{11}$$

La solution de l'équation est la suite $y_n$ exprimée en fonction de $t_n = t_0 + nh$.

La présentation traditionnelle est la suivante :

$$\begin{aligned} k_1 &= h \cdot f(y_n, t_n) \\ k_2 &= h \cdot f(y_n + k_1/2, t_n + h/2) \\ y_{n+1} &= y_n + k_2\ . \end{aligned} \tag{12}$$

Si la fonction numérique $f$ est écrite de manière suffisamment polymorphe, en utilisant les opérations arithmétiques standard, et si nous pouvions effectuer son « lifting », au domaine des suites, à l'aide des déclarations des instances arithmétiques appropriées, la solution numérique *complète* de cette équation serait donnée par

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

## 4.4. Calcul des réduites des fractions continues

Selon [4] la meilleur méthode générale de calculer les sommes partielles (les réduites) des fractions continues

$$f = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots}}}, \tag{13}$$

c'est l'algorithme de Lentz [9], beaucoup plus stable que la solution itérative de Wallis, inventée il y a 300 ans. Sa description dans un pseudo-code impératif est la suivante :

Assigner $f_0 = b_0; C_0 = f_0; D_0 = 0;$
**Pour** $j = 1, 2, \ldots$ **itérer**
  Affecter $D_j = 1/(b_j + a_j D_{j-1})$
  Affecter $C_j = b_j + a_j/C_{j-1}$
  Affecter $f_j = f_{j-1}(C_j D_j)$
  Arrêter quand $C_j D_j$ devient très proche de 1.

La valeur $f_n$ est la $n$-ième réduite. Dans le code la clause d'arrêt est absente, nous généreons une suite infinie de réduites.

La spécification doit être complétée par un dispositif de sécurité, pour prévenir la division par zéro. Notre code est plus court que sa description informelle :

```
cfeval b0 b a =                                -- b commence par b1
  let enh x = if x==0.0 then 1.0e-35 else x    -- Protection
      tiny s = fmap enh s                      -- Remplacement des zéros
      d = S 0.0 (recip (tiny (b+a*d)))
```

```
    c = tiny (S b0 (b+a/c))
  in G (enh b0) (sqtl (c*d))
```

même si sa lecture demande un peu d'effort, ou d'expérience.

Prenons à titre d'exemple la fonction exponentielle $e^z$ qui possède une représentation régulière :

$$e^z = 1 + \cfrac{z}{1 + \cfrac{z}{-2 + \cfrac{z}{-3 + \cdots}}}, \tag{14}$$

et dont les termes $b_1, \ldots$ forment la suite [1, -2, -3, 2, 5, -2, -7, 2, 9, ...]. Cette suite est générée par le programme suivant

```
bb = bx + S 0.0 tw where
  tw=S (-2.0) (S 0 (neg tw))       -- = -2,0,-2,0,-2,0,...
  bx=S 1.0 (S 0 (neg bx + tw))     -- = 1,0,-3,0,5,0,-7,0,9,...
```

L'exécution de `res = cfeval 1.0 bb (Cs 1.0)` génère [1.0, 2.0, 3.0, 2.75, 2.714, 2.7179, 2.71831, 2.718283, 2.71828172, 2.71828182, 2.71828183,...]. L'algorithme marche également pour les développements qui commencent par $b_1$ ($b_0 = 0$), par exemple pour $\tan(x)$.

# 5. Intégration de Romberg

Si une fonction est échantillonnée uniformément dans un intervalle $(a, b)$, les techniques classiques d'intégration numériques sont usuellement basées sur des schémas de Newton-Cotes, instanciées par exemple par la méthode des trapèzes ou de Simpson, où l'évaluation de $I = \int_a^b f(x)\,dx$ prend la forme $I(h) = \sum_{k=0}^{n} c_k f(a + kh)$, avec $h = (b - a)/n$, où $n$ est le nombre de points d'échantillonnage.

On sait depuis longtemps qu'il est possible d'obtenir une meilleure précision sans incrémenter le nombre $n$. Si la valeur numérique $I(h)$ dépend de la longueur du sous-intervalle élémentaire de sorte que $I(h) = I_0 + \zeta h^p + o(h^p)\ldots$, on peut essayer d'estimer mieux la « vraie » valeur $I_0$ en comparant deux résultats avec deux valeurs différentes de $h$ (et de $n$).

Une meilleure approximation à $\lim_{h\to 0} I(h)$ peut être obtenue par le schéma d'éxtrapolation itérée de Richardson, dont la première étape serait ici la soustraction

$$I \to \left(2^p I(h/2) - I(h)\right) / \left(2^p - 1\right) \tag{15}$$

ce qui élimine la contribution du terme $\zeta$ dans le développement asymptotique de $I(h)$. Le résultat aura le comportement $I_0 + \zeta' h^q + \ldots$, où $q > p$. Notre connaissance de l'exposant principal $p$ des termes restants (erreur de discrétisation) nous permet d'itérer cette procédure. Ainsi on peut construire la règle de Simpson, qui est équivalente à la méthode des trapèzes itérée une fois.

## 5.1. Méthode des trapèzes adaptative

Commençons donc par la génération d'une suite infinie d'approximants pour

$$\int_a^b f(x)\,dx = h\left(f(a)/2 + f_1 + \ldots + f_{n-1} + f(b)/2\right), \qquad n = n_0, 2n_0, 4n_0, \ldots \tag{16}$$

où $n$ change avec chaque itération par $n \to 2n$. Naturellement, en doublant le nombre de points nous pouvons et nous *devons* réutiliser toutes les valeurs précédentes. Dans le code ci-dessous la première fonction `sumn` calcule tout simplement la somme des premiers $n$ termes. (Ceci est redondant, mais nous avons voulu éviter d'utiliser `sums`, ce qui serait formellement correct, mais demanderait une déforestation (élimination des structures de données intermédiaires) si nous pensions à l'utilité pratique de notre algorithme).

On sépare la contribution de $f(a)$ et $f(b)$, et on augmente la densité des points intermédiaires à chaque itération effectuée par la fonction `intr`.

```
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)

trapeze 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)))) (intr h0 1)))*(G h0 (Cs 0.5))
```

Ceci n'est pas très simple, mais la présentation impérative, avec les boucles, *du même algorithme* n'est pas simple non plus. Soulignons qu'il ne s'agit pas de la méthode des trapèzes simple, qui peut être codée en une ligne, mais de l'algorithme adaptatif, plus concrètement : de l'étape génératrice de l'algorithme adaptatif, et qu'une bonne partie de la complexité vient de l'optimisation, nous n'évaluons jamais deux fois $f(x)$ pour le même $x$. Le choix de la valeur maximale de $n$ est laissé à la discrétion du module qui consomme le résultat.

Notez l'expression `sumn 2 (f (S a (Cs b)))` qui représente $f(a) + f(b)$ ; La fonction $f$ est surchargée – elle doit agir sur les suites, et non pas sur les nombres, ce qui rend la présentation de l'algorithme moins lisible.

Le test effectué sur l'intégrale $\int_0^3 \exp(-2x)\,dx$ produit le résultat final [1.5, 0.83, 0.589, 0.522, 0.5046, 0.5002, 0.4991, 0.49885, 0.498783, 0.498766, 0.498762, 0.498761,...], où chaque nouvel item double le nombre de points d'échantillonnage. La convergence est loin d'être excellente.

## 5.2. Implantation du schéma d'extrapolation itérée de Richardson

On peut prouver que le taux de convergence de la méthode des trapèzes est: $I(h) = I_0 + \zeta_1/n^2 + \zeta_2/n^4 + \zeta_3/n^6 + \ldots$. Dans la formule (15) $p = 2$. Le schéma est valide si $h$ est petit et le développement de $I(h)$ est convergent, donc il faut commencer par $n = n_0$ suffisamment grand. Il suffit de regarder la convergence de la suite produite ci-dessus : pour notre fonction de test, la valeur $n_0 = 8$ est un point de départ raisonnable.

Le codage final du schéma de Romberg est très simple et compact. La suite initiale des approximants $u$ est « digérée » par le générateur/itérateur suivant:

```
rombg c u = S (sqhd w) (rombg (4*c) w)
  where w = (1.0/(c-1.0))*>(c*>sqtl u - u)
```

où la valeur initiale de $c = 2^p$ doit être égale à 4, en accord avec le taux de convergence de la méthode des trapèzes. Naturellement, si nous avions commencé par une quadrature primitive qui converge plus rapidement, par exemple Simpson ou Gauss, il faudrait choisir la valeur $c$ appropriée, et aussi changer la constante 4 dans `(rombg (4*c) w)`. Pour notre exemple nous obtenons [0.6, 0.50, 0.49882, 0.498760862, 0.498760624, 0.498760624,...]. $2^5 = 32$ subdivisions suffisent pour obtenir la précision équivalente à plus de 4000 si la méthode des trapèzes simple avait été appliquée. L'algorithme de Romberg est très robuste et populaire, mais parfois considéré difficile à coder, car il est intrinsèquement récursif, et son implantation avec des boucles imbriquées est un mécanisme de torture pédagogique très efficace.

# 6. Calcul symbolique des différences

Dans cette section nous continuons à manipuler les suites numériques en utilisant nos structures de données et la sémantique paresseuse, mais l'arithmétique sur les suites sera enrichie par l'usage des opérateurs de différences finies, l'opérateur $\Delta$, et ses variantes. Beaucoup de formules très intéressantes dans le domaine des séries entières, et de très nombreux schémas de différences finies pour résoudre les équations différentielles peuvent être dérivées par des manipulations formelles de l'opérateur $\Delta$.

L'opérateur qui permet de passer au successeur d'un élément peut être spécifie comme $Eu : Eu_n = u_{n+1}$, et ceci implique la relation formelle $E = 1 + \Delta$. La réalisation de $E$ dans notre formalisme est l'opérateur `sqtl`.

Nous pouvons exploiter cet opérateur pour *extrapoler* la suite $u$ et obtenir l'approximation polynomiale de $u_{-1}$. Formellement $u_{-1} = E^{-1}u_0$. Ceci peut être reformulé de manière suivante :

$$u_{-1} = \frac{1}{1+\Delta}u_0 = (1 - \Delta + \Delta^2 - \Delta^3 + \cdots)u_0\,. \tag{17}$$

L'approximation linéaire est $u_{-1} = 2u_0 - u_1$, le terme suivant génère l'approximation parabolique : $u_{-1} = u_2 - 3u_1 + 3u_0$, etc. Ceci peut être utile pour calculer automatiquement les fragments terminaux des splines ou autres courbes polynomiales tracées par les logiciels graphiques. Mais *a priori* il est difficile de tronquer la série en $\Delta$ de manière universelle. Nous allons donc générer une suite paresseuse d'approximations de $u_{-1}$, et le module consommateur pourra tester la convergence de cette suite et choisir le degré du polynôme extrapolateur selon la précision souhaitée. Voici le code, qui génère d'abord une suite de suites, et ensuite récupère les têtes :

```
extrap u = fmap sqhd (sums (gen u)) where
  gen u = S u (gen (fd (neg u)))
```

Pour la suite $u_n = \cos(0.2n)$, ou `u = cos (A 0.0 (Cs 0.2))`: $u$ = [1.0, 0.980066578, 0.921060994,...], la valeur de `extrap u` est [1.0, 1.01993342, 0.98086126, 0.978508894, 0.979972796, 0.980124939, 0.980072643, 0.980064493, 0.980066253, 0.980066648, 0.980066594,...].

On peut évidemment utiliser le même principe pour effectuer l'interpolation polynomiale des suites (en particulier les suites résultant d'un échantillonnage uniforme) sans chercher des formules de Lagrange, Newton, etc. Si $u_n = u(x_0 + nh)$, alors $u(x_0 + (n+\kappa)h) = E^\kappa u_n = (1+\Delta)^\kappa u_n$, et il suffit d'appliquer le développement de la série binomiale en $\Delta$ à $u$, tronquée au degré désiré du polynôme interpolateur.

$$u(x_0 + (n+\kappa)h) = \sum_k \binom{\kappa}{k} \Delta^k u_n\,. \tag{18}$$

En général, si nous voulons calculer la somme des séquences $s = \sum_{k=0} c_k \Delta^k u$, où les coefficients $c_k$ forment une liste, il nous faut choisir la borne supérieure de la sommation $m$, et appliquer `(sumn m)` à

```
sseq (c0:cq) u = S (c0*>u) (sseq cq (fd u))
```

Si la somme est infinie, on génère la suite de sommes partielles pour $m = 1, 2, \ldots$, comme dans l'exemple d'extrapolation. Si le point de départ est une formule de genre $F(\Delta)\,u$, la liste (ou suite) $c_k$ peut être obtenue par le développement de Taylor de la fonction $F$. Nous avons utilisé le paquetage décrit dans ([5]) qui est maintenant integré dans notre Prélude algébrique.

Une autre application possible d'une telle représentation des suites par les successeurs exprimés par les différences est la technique d'Euler d'accéleration de convergence de sommes. Formellement nous pouvons écrire $u_n = E^n u_0$, et pour les séries entières alternées, pour lesquelles la technique donne des meilleurs résultats et peut même permettre la sommation (continuation analytique) d'une suite divergente, nous aurons la dérivation suivante:

$$\begin{aligned} \sum_{n=0} (-z)^n u_n &= \sum_{n=0} (-zE)^n u_0 \\ &= \frac{1}{1+zE} u_0 \quad = \quad \frac{1}{1+z} \cdot \frac{1}{1+z\Delta/(1+z)} u_0 \\ &= \frac{1}{1+z} \sum_{k=0} \left( \frac{-z}{1+z} \right)^k \Delta^k u_0 \,, \end{aligned} \tag{19}$$

ce qui peut être codé en trois lignes. Voici le code d'« Eulerisation » d'une série alternante $\sum (-z)^n u_n$, où les signes $\pm 1$ sont attachés à la variable formelle $z$ plutôt qu'à la suite de coefficients.

```
euler z u = let z1=recip (1.0+z); fc=neg z * z1
                eusum u = S (sqhd u) (fc*>eusum (fd u))
            in z1 *> eusum u
```

La somme $1-1/3+1/5-1/7+\cdots = \pi/4$ demande plus de 500 termes pour fixer 3 chiffres significatifs (ici $z=1$). 15 termes de la suite transformée donnent déjà 5.

Les exemples plus complexes demandent parfois un peu d'attention. Par exemple, si nous voulons calculer la dérivée d'une fonction tabularisée, les différences standard $\Delta$ ajoutent une déviation, et il serait mieux de pouvoir appliquer l'opérateur de différences finies centrales $\delta u(x) = u(x+h/2) - u(x-h/2)$.

Ceci correspond dans le domaine de suites discrètes à l'opérateur des différences symétriques $\delta u_n = u_{n+1/2} - u_{n-1/2}$. Mais les éléments de genre $u_{n\pm 1/2}$ ne sont pas directement disponibles, et il faut utiliser quelques techniques d'interpolation/extrapolation, par exemple développer $\delta = E^{1/2} - E^{-1/2} = \sqrt{1+\Delta} - 1/\sqrt{1+\Delta}$ en série de $\Delta$. Ceci n'est pas idéal.

Mais nous savons comment appliquer *effectivement* l'opérateur $\delta^2$: $\delta^2 u_n = u_{n+1} - 2u_n + u_{n-1}$, donc $\delta^2 = E - 2 + E^{-1}$. L'opérateur $E^{-1}$ passe tout simplement de $u_n$ à $u_{n-1}$, il faut seulement adopter un schéma d'extrapolation d'indices en dessous de zéro, ce qui a déjà été discuté. La stratégie permettant de calculer la dérivée numérique d'une fonction échantillonnée : $u_n = f(x_n) = f(x_0 + nh)$ par les techniques de différences finies, peut être codée de manière élégante et compacte. Si $D$ dénote l'opérateur de différentiation, nous pouvons écrire la série de Taylor formelle pour le successeur:

$$f(x+h) = \sum_{n=0} h^n \frac{D^n f(x)}{n!} = \exp(hD) f(x) \,, \tag{20}$$

ou :

$$E = e^{hD} \,, \quad \text{d'où} \quad \delta = e^{hD/2} - e^{-hD/2} \,. \tag{21}$$

La solution formelle de cette équation est

$$\frac{h}{2} D = \log\left( \frac{\delta}{2} + \sqrt{1+\delta^2/4} \right) \,. \tag{22}$$

Le développement de la formule (22) est un exercice standard, cependant $D$ est une fonction impaire (arsinh) de $\delta$ : $D = \delta F(\delta^2)$. Ceci n'est pas commode pour des raisons évoquées ci-dessus, il serait mieux de pouvoir exprimer $D$ par une fonction rationnelle de $E$ et $E^{-1}$. Nous pouvons utiliser le « truc » suivant. L'opérateur qui calcule la moyenne symétrique : $\mu\, u(x) = 1/2\,(u(x+h/2) + u(x-h/2))$ possède la représentation symbolique $\mu = 1/2(E^{1/2} + E^{-1/2})$. Or, en regardant la définition de $\delta$ il est facile de prouver que $W = \mu/\sqrt{1+\delta^2/4} = 1$. Si on multiplie l'expression (22) par $W$, le développement en série s'exprime par $\delta^2$ et par $\mu\delta = 1/2(E - E^{-1})$.

Tout ceci peut être directement appliqué à la génération de plusieurs formules d'interpolation ou extrapolation, mais ce sujet est trop détaillé.

# 7. Conclusions

Les calculs numériques sérieux demandent d'habitude l'usage des plus rapides parmi les algorithmes disponibles, et l'élégance de la programmation est assez souvent rapportée au second plan. Les techniques impératives typiques, les boucles qui parcourent les tableaux, et qui modifient sur place les éléments indexés sont normalement considérés plus efficaces que la manipulation des listes. L'allocation de mémoire est un facteur important pour l'évaluation de la complexité du code.

Cependant, nous devons parfois *enseigner et tester* quelques algorithmes conceptuellement difficiles, par exemple: adaptatifs, ou possédant une structure récursive non-triviale, comme la technique de Romberg, et dans ce contexte les ressources *humaines* peuvent être beaucoup plus importantes que la vitesse d'exécution. Dans la présentation mathématique des algorithmes, les objets définis par l'induction abondent, et les techniques fonctionnelles en offrent des implantations plus naturelles.

Les exemples montrés ici sont relativement simples, mais ils sont utilisables dans la pratique courante, ils ne sont pas des jouets théoriques ou des maquettes qui montrent uniquement l'élégance de la forme. Nous avons voulu montrer comment économiser le temps du codage (et du déboguage...), et comment trouver un plaisir intellectuel dans le domaine des calculs numériques, normalement considérés comme un exercice assez ennuyeux. Mais nous ne voulions pas sacrifier l'efficacité de l'exécution, au contraire, nous avons exploité jusqu'au bout le fait que très souvent le parcours par les suites est régulier, séquentiel, et l'arithmétique des indices gérée par l'administration des boucles est inutile.

Il faut remarquer que la sémantique non-stricte *est* un obstacle psychologique important pour un utilisateur typique du logiciel numérique : physicien ou ingénieur qui utilise Fortran ou C++. Une boucle infinie est considérée « naturelle », une liste infinie est une abomination. Même les algorithmes récursifs sont peu connus, sans parler de données co-récursives. Pour la vulgarisation des techniques fonctionnelles dans ce milieu il faudra montrer la solution de plusieurs problèmes réels, beaucoup plus complexes que nos exercices.

# Références


[1] Germund Dahlquist, Åke Björck, *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, 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, *Generating Bessel functions in Mie scattering calculations using continued fractions*, Applied Optics, **15**, (1976), pp. 668 – 671. Voir aussi *Numerical Recipes in "C"*.