# Improving Floating-Point Numbers ライブラリの高速化

横山 瑞宜　川端 英之　北村 俊明

正確な数値計算を実現する枠組みとして IFN (Improving Floating-Point Numbers) が開発されている．IFN は，個々の数値を独特の浮動小数点表現に基づくコーシー列で表し，遅延評価を活用することにより，計算結果の正確さを任意に調整可能な枠組みである．IFN の性質や使用の容易さは既に言及され，IFN に基づく数値計算ライブラリのプロトタイプが Haskell で記述されて数値実験も行われているが，その実装はナイーブであり，計算速度の観点からの評価は困難であった．我々は，IFN ライブラリによる数値計算の高速化を目的とし，既存のライブラリに対し，効率的なデータ表現の導入，MPFR ライブラリの活用，および，正確な結果を得るために要する計算量の削減を試みた．大幅な桁落ちが生じる数式を用いた数値実験の結果，初期プロトタイプに対して 4 桁以上の高速化が達成され，iRRAM とも比較可能な計算速度に至ったことが確認できた．

## 1　はじめに

コンピュータを用いた数値計算において正確な計算結果を得ることは，必ずしも容易なことではない．そもそも実数は有限桁数の数値表現では正確に表せないため，一般の数値計算は浮動小数点表現などの近似表現に基づいて行われる．数値計算プログラミングにおいては，ユーザは，近似値を用いた近似的な数値計算結果が，全く意味のない値となり得ることを常に念頭に置き，計算精度について敏感であり続ける必要がある．

著者らは，正確な数値計算のための枠組み IFN (Improving Floating-Point Numbers) を提案している [1]．IFN は，データ表現の詳細を意識することなく任意の正確さの実数計算を記述できる数値計算ライブラリの実現のための数値表現である．IFN に基づく数値計算では，適応的精度改善，すなわち，必要に応じて再計算を行いつつ計算結果の精度を保つ手法がとられる．この方式に基づく数値計算ライブラリである IFN ライブラリも著者らによって開発されている．IFN ライブラリを用いれば，ユーザは，浮動小数点表現に基づく計算では配慮が必要であったような事柄，例えば丸め誤差の混入や大規模な桁落ちの発生などの問題を意識することなく，数値計算プログラムを記述できる．

文献 [1] で論じられている IFN ライブラリは，それ自体で有用であるものの，動作検証のためのプロトタイプとして開発されたものであり，その実装には，高速な数値計算の実現には不向きな点が多い．IFN ライブラリの実用化のためには，計算速度を意識した改善が必須といえる．

我々は，IFN ライブラリの高速化を目的とし，いくつかの改善を試みた．まず，従来の方式では多量の再計算が必要になる場面が多かった点に着目し，総計算量を削減することによる高速化を図った．また，すべてのデータ型およびその上での演算を Haskell で記述した IFN の初期プロトタイプに対し，可変長仮数部の浮動小数点表現などのデータ表現の再設計，FFI による C ルーチンの活用，MPFR [2] の直接的な使用等により，実装上の改善を行った．

高速化の効果を評価するために，悪条件の連立一次方程式を用いた数値実験等を行った．我々の取り組

Performance Improvment of the Improving Floating-Point Numbers Library
Mizuki Yokoyama, Hideyuki Kawabata, and Toshiaki Kitamura, 広島市立大学, Hiroshima City University.

みの一部はヒューリスティックに基づく設計だが，簡単な数値実験による評価によると，十分効果が高かった．結果として初期プロトタイプに対して 4 桁以上の高速化が達成され，その有効性が確認された．

本稿では，IFN の高速化についての取り組みについて報告する．以下，第 2 章では IFN ライブラリの概要を，第 3 章では IFN ライブラリの設計および初期プロトタイプにおける実装について述べる．続く 2 つの章では IFN ライブラリに対して我々が行った取り組みについて述べる — 第 4 章では IFN ライブラリの計算手法に対する改善，第 5 章では実装手法に対する改善について，それぞれ詳細に述べる．第 6 章では IFN ライブラリの高速化の効果を評価するために行った数値実験について述べる．第 7 章では関連研究について述べる．第 8 章はまとめである．

## 2 IFN ライブラリの概要

### 2.1 動機：浮動小数点演算の利点と欠点

科学技術計算のための数値表現としては浮動小数点表現が基本的である．多くの汎用プロセッサには IEEE754 [3] 規格に基づく高速な倍精度浮動小数点演算器が搭載されており，デスクトップ PC でも 100GFLOPS を超える計算速度を享受できる．また，MPFR ライブラリ [2] など，多倍長の浮動小数点演算のためのライブラリも容易に利用可能である．浮動小数点表現に基づく数値計算はプログラム開発環境も実行環境も整っていると言えるだろう．

しかし一方で，浮動小数点表現に基づく数値計算プログラムは，特に品質を意識した場合に，開発が容易であるとは必ずしも言えない．設計にあたって，ユーザは，計算結果の正確さに多大な注意を払うことが求められる．

浮動小数点演算の特徴を示す例として，次の Rump の例 [4] と呼ばれる式を考える．

$$\begin{aligned} y = {} & 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) \\ & + 5.5b^8 + a/(2b) \end{aligned} \tag{1}$$

式 (1) の値は，$a = 77617$，$b = 33096$ とすると（このとき $a^2 = 5.5b^2 + 1$），次のようになる．

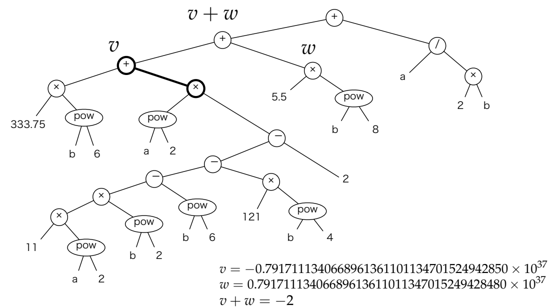



図 1 **Rump** の例の木構造表現

$$\begin{aligned} y &= -2 + a/(2b) = -54767/66192 \\ &\fallingdotseq -0.827396059946821368 14 \end{aligned}$$

ところが，この式の値を 4 倍精度浮動小数点演算で求めようとしても，正しい値とは似ても似つかない結果しか得られない．Rump の例の式の構造は図 1 に示す通りであるが，図中の $v$ および $w$ の値の絶対値が非常に近く，$v + w$ の計算においては 10 進 36 桁を超える情報を保持した計算を行わない限り大規模な桁落ちの発生によって不正確な結果しか得られないのである．

Rump の例からは他にもいくつかの事実が確認できる．図中，太字の丸で示した演算を除いては 4 倍精度程度の浮動小数点演算でも問題は生じない．また，$a$ や $b$ を僅かに変更しただけで大規模な桁落ちは抑えられ，計算全体を 4 倍精度程度の浮動小数点演算で行うことが可能になる．これらの観察からわかるように，浮動小数点表現を用いる数値計算プログラムの特徴は次のようにまとめられる．

- 個々の演算をどの程度の精度で行えば十分かを予め知ることは難しい．
- 浮動小数点表現での計算結果からはその数値の正確さの程度を判別することはできない．
- 入力データの変動が計算結果に与える影響を予め把握することは難しい．

### 2.2 IFN を用いたプログラミング

著者らは，正確な数値計算のための枠組みとして，データ型 IFN とそれに基づく計算手法を提案してい

る[1]．また，IFN を用いた数値計算のための IFN ライブラリも設計しており，Haskell で記述されたプロトタイプが実現されている（以下，IFN ライブラリの初期プロトタイプと呼ぶ）．IFN を用いたプログラミングの特徴は，上で列挙した浮動小数点表現によるプログラミングの特徴（問題点）と対比して次のようにまとめることができる．

- ユーザは，プログラムを構成する個々の変数に割り当てる語長を意識する必要がない — プログラム中の変数は，IFN 型にすればよい．
- 数値計算の結果として得られる値は正確である — ユーザは，必要なだけの精度の（多倍長浮動小数点表現の）値を，計算結果の IFN 型変数から取り出すことができる．
- 入力データの変動が計算結果に与える影響を考慮することは依然として難しい — ただし，入力データが正確であるという前提の下で任意の精度の計算結果を得ることができる．

以上のことを可能にするデータ型 IFN は，特殊な多倍長浮動小数点表現（データ型 Q）の無限列（遅延ストリーム）として定義される．IFN は実数値を表すが，その実体は，当該実数値を近似する多倍長浮動小数点表現 Q が正確さの昇順に並ぶ一種のコーシー列である．

IFN を用いた数値計算では，入力データの IFN 化，IFN 演算子の合成による IFN 同士の演算，および，IFN からの多倍長精度浮動小数点数の取り出しの組み合わせによってプログラムが構成される．IFN に関する操作はすべて lazy に行われる．IFN および Q の持つべき性質とそれを保った演算の実現手法が，IFN ライブラリの設計における要といえる．

## 3 IFN ライブラリの設計と実装

本章では，IFN ライブラリの設計および初期プロトタイプの実装[1] について述べる．なお，データ型やアルゴリズムの説明にあたっては Haskell の記法に従った記述を用いる．

### 3.1 Q: 実数値を近似する多倍長浮動小数点表現

多倍長浮動小数点表現 Q は，初期プロトタイプにおいては下のように定義されている．

```
data Q = Q { sign     ::  Int
           , mantissa ::  [Int]
           , expo     ::  Int
           , zeroFlag ::  Bool }
```

すなわち，一般の多倍長浮動小数点表現と同様，符号，可変長仮数部，および指数部を持っている．IFN では，（真のゼロを意味するゼロフラグを有すること以外に）Q のインスタンス $q$ が近似する実数 $v$ の存在範囲を次のように定義する点が特徴的である．

$$\langle q\rangle - \frac{2^e}{2^{n+1}} \le v \le \langle q\rangle + \frac{2^e}{2^{n+1}} \qquad (2)$$

ここで，$q$ の仮数部が $[b_1,\ldots,b_n]$ （$b_i$ は 0 か 1），指数部と符号がそれぞれ $e$，$s$ であるとき（$s$ は 1 か $-1$），$\langle q\rangle$ は次のように定義される．

$$\langle q\rangle = s \times \left(\frac{b_1}{2^1} + \frac{b_2}{2^2} + \cdots + \frac{b_n}{2^n}\right) \times 2^e$$

式 (2) は，当該範囲外の実数値の近似値として $q$ が用いられることはないということを明確に述べている．浮動小数点表現 $q$ の可変長仮数部に「余計なビット」は存在しない．

また，$q$ に対して $n-e$ を $ac\ q$ と表記し「（$q$ の）正確さ」と定義する．式 (2) は $ac\ q$ を用いて下のように書ける．$ac\ q$ は，$v$ を $q$ によって近似したときの絶対誤差の指標に他ならない．

$$\langle q\rangle - \frac{1}{2^{(ac\ q)+1}} \le v \le \langle q\rangle + \frac{1}{2^{(ac\ q)+1}}$$

なお，$ac\ q$ は初期プロトタイプでは次のように定義されている．

```
ac ::  Q -> Int
ac q = lenM q - expo q
lenM ::  Q -> Int
lenM q = length (mantissa q)
```

### 3.2 IFN: 実数値を表現する無限リスト

実数値を表すデータ型 IFN は，Q が正確さの昇順に並ぶ無限リストであり，初期プロトタイプでは次のように定義される．

```
type IFN = [Q]
```

IFN $qs = [q_0, q_1, \ldots]$ とすると，IFN は次の性質を満たすものと定義される．

- すべての $i \ge 0$ に対して $ac\ q_i < ac\ q_{i+1}$ である．

```
addIFN ::  Int -> IFN -> IFN
addIFN = addIFN' a0 where a0 = minBound::Int
addIFN' ::  Int -> IFN -> IFN -> IFN
addIFN' a (p:ps) (q:qs)
  | zeroFlag p = (q:qs)
  | zeroFlag q = (q:qs)
  | ac p < ac q =
  |  if a' <= a then addIFN a ps (q:qs)
  |             else r :  addIFN a' ps (q:qs)
  | ac p > ac q =
  |  if a' <= a then addIFN a (p:ps) qs
  |             else r :  addIFN a' (p:ps) qs
  | otherwise =
  |  if a' <= a then addIFN a ps qs
  |             else r :  addIFN a' ps qs
  where r = addQ p q
        a' = ac r
```

図 **2　IFN** 同士の加算の実装例

- ある実数 $v$ が存在して，すべての $i \geq 0$ について，$q_i$ が $v$ を近似する．

$q_i$ の正確さ $ac\ q_i$ は整数値であり $i$ を大きくするといくらでも大きくできるので，ある IFN $qs$ のすべての要素が近似する $v$ は唯一であり，それが $qs$ の表す実数値である．

IFN のインスタンスは，プログラム中の定数や入力データから次の関数により生成できる．なお `fromString` は指定した仮数部長の Q のインスタンスを作る関数，`initN` および `diffN` はそれぞれ IFN の先頭要素の仮数部長の初期値および増分を表す定数である（仮数部長がゼロでない場合，仮数部長の昇順は正確さの昇順を意味する）．

```
genIFNfromString ::  String -> IFN
genIFNfromString s
  = map (\n -> fromString n s)
        [initN, initN + diffN..]
```

変数 $qs$ から，所望の正確さの多倍長浮動小数点数を得るには，次の関数を使えばよい．

```
accurateValue ::  Int -> IFN -> Q
accurateValue a qs
   = head $ dropWhile (\q -> ac q < a) qs
```

### 3.3　IFN 演算子と Q 演算子

IFN を用いた算術演算は，次のように実現できる．例えば，IFN $ps = [p_0, p_1, \ldots]$ および $qs = [q_0, q_1, \ldots]$ の加算結果 $rs$ を，IFN の加算演算子 addIFN を用いて次のように得るとする．

$$rs = \mathrm{addIFN}\ ps\ qs$$

$rs = [r_0, r_1, \ldots]$ と置くとき，$r_0$ は Q の加算演算子 addQ を用いて $r_0 = \mathrm{addQ}\ p_0\ q_0$ とする．$r_1$ については，

- $r_1 = \mathrm{addQ}\ p_1\ q_0$
- $r_1 = \mathrm{addQ}\ p_0\ q_1$
- $r_1 = \mathrm{addQ}\ p_1\ q_1$
- その他（$r_1 = \mathrm{addQ}\ p_2\ q_3$ など）

といった選択肢が考えられるが，IFN ライブラリでは，図 2 に示す通り，$ac\ r_0 < ac\ r_1$ となるように適宜オペランドを選択するように addIFN が定義される．

IFN 演算子は Q 演算子（上の例では addQ）を用いて構成される．Q 演算子では基本的には多倍長浮動小数点演算が行われるが，単純な浮動小数点演算を用いただけでは不十分で，計算結果の正確さの調整処理が必要となる．正確に言えば，Q 型の値 $p$，$q$ がそれぞれ実数値 $v$，$w$ を近似している場合に，加算結果 $r = \mathrm{addQ}\ p\ q$ が実数値 $v + w$ を近似するようにしなければならない．そうでなければ，$ps$ および $qs$ が表す実数値 $v_p$ および $v_q$ に対して，$rs = \mathrm{addIFN}\ ps\ qs$ が実数値 $v_p + v_q$ を表さなくなってしまう．

Q 演算子における計算結果の浮動小数点表現の正確さの調整とは，適切な位置における仮数部の丸めである（一般の浮動小数点演算での丸め誤差混入による「あてにならないビットの発生」を防ぐ措置）．適切な丸め位置はデータによって変わる（演算結果の仮数部長も変わる）．

### 3.4　IFN における論理演算

IFN ライブラリには，IFN 同士の大小比較など，論理値を返す IFN 演算子も備えられている．ただし，比較演算は IFN が表す実数の同一性判定やゼロ判定に帰着され，それらを厳密に取り扱うことは困難である．IFN では，ユーザが与える基準値に従い，ある程度以上近い実数値の差はゼロと同一視する．なお，プログラム中の定数由来のゼロである「真のゼロ」は，実数計算の結果として生じる見かけ上のゼロとは異なり，特別扱いする．

### 3.5 IFN ライブラリとプログラミング

前述の通り，IFN ライブラリを用いたプログラムは，IFN 型を用いて構成する．IFN 演算子の実装は遅延評価を前提にしており，IFN 型からの浮動小数点数の取り出し操作が，データフローを遡った部分式の適応的な再計算要求の伝搬に対応付けられる．データフローの末端にあるのは定数や外部入力値であり，これらを表す IFN から，正確さが大きくなる順に，すなわち仮数部長の昇順に，多倍長浮動小数点数が取り出され，計算に用いられることになる．

なお，IFN 演算子の使用は，多くのプログラミング言語の持つ演算子オーバーローディング機能等によりコードの表面上から隠せるため，数値計算プログラムの記述時の困難は特に生じないと考えられる．例えば，Haskell では，IFN 型を型クラス Num や Fractional のインスタンスと定義すれば，`+` や `/` などの算術演算子を直接使用したプログラミングが可能である．

### 3.6 精度と正確さ

一般の浮動小数点演算においては,「精度」は仮数部の（有効）桁数を表す．つまり，精度は相対誤差の指標である．一方，IFN の設計において定義されている「正確さ」（3.1 節参照）は，浮動小数点表現 Q の絶対誤差を表す指標である．両者の定義は異なるが，具体的な値に言及しない限りにおいては互いに置き換え可能な表現である．例えば，数値計算における「計算精度の改善」と「正確さの改善」は同義と言える．

IFN ライブラリの設計にあたっては正確さの絶対値が言及されているものの[1]，IFN ライブラリの実体は,「正確さ」という尺度による単調増加列同士の演算の合成により，任意の尺度のデータを得ることができる適応的な改善機構を実現したものである．従って，単調増加列どうしの演算（IFN 演算）が定義できるならば，単調増加列を構成するための尺度は他のものに置き換えることができる．実際，IFN は，Q のインスタンス $q$ の正確さ $ac\ q$ ではなく仮数部長 $lenM\ q$ に基づく単調増加列とした構成も可能で，インターフェースをほとんど変えることなく IFN ライブラリを実装することができる．

本稿では，誤解の恐れのない範囲で,「精度」という言葉を「正確さ」と互いに同義として用いる．

### 3.7 IFN の課題：高速化の余地

IFN ライブラリは，ユーザに対し，計算精度を意識しない数値計算手段を提供する．しかしながら，IFN ライブラリの初期プロトタイプは，計算手法や実装に，高速処理には適さない部分があり，IFN ライブラリによる計算速度等の定量的な評価が難しい．主要な改善の余地としては次が挙げられる．

- ユーザは最終結果に要求する精度（正確さ）を指定することはできても，部分式については指定できないため，適応的精度改善を部分式単位で行うことができない．結果として，長大な数式の値を正確に求めるために要する各部分式の再計算回数が多大となりがちである．
- 計算速度や所要記憶容量を考慮したデータ構造が用いられていない．また，可変長仮数部の特殊な浮動小数点演算（Q 演算）がすべて Haskell で記述されている．つまり，実用的な計算速度が得られるとは言い難い設計になっている．

我々は，これらの項目について，特に高速化を目的として改善を試みた．以降の章では，我々の取り組みの詳細について述べる．

## 4 高速化 1: 部分式単位での精度改善

### 4.1 概要

IFN ライブラリを用いた数値計算では，一連の計算の結果に対してユーザが要求する正確さを満足するために，必要に応じた適応的精度改善が行われる．「精度改善」は計算式を構成する個々の演算に対応する IFN 演算子で制御される．図 2 に例示されているように，IFN 演算子は，生成する IFN が正確さの昇順に整列されるという制約を満たす関数として設計される．この制約を満たしている限り，当該関数は，IFN 演算子として再計算要求を部分式に伝える機能を実現する．

この方式の問題は，ユーザが与える要求精度が，一連の計算のデータフローにおける末端部分に直接届かないということである．すなわち，数式の末端の定数参照について，どの程度の精度を初期値として計

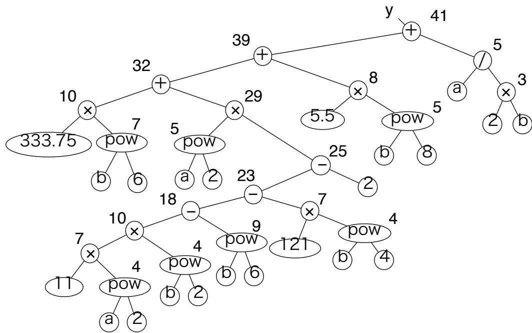



図 3　**Rump** の例における各演算の実行回数

算を開始するべきかがわからないのである．また，適当な初期精度に基づいて一連の計算を終えて初めて，ユーザの要求が満たされるかどうかが判明し，不十分な場合の再計算の連鎖は，データフローの（一つ以上の）末端にまで届く．結果として，再計算回数は増大しがちである．

IFN を用いた数値計算における再計算の様子を示すために，式 (1) を $a = 77617$，$b = 33096$ としたときの各 IFN 演算子における計算回数（すなわち IFN の何番目の要素まで実際に計算したか）を図 3 に示す．計算にあたっては，定数の IFN の先頭要素の仮数部長および増分（要素間の仮数部長の差）はいずれも 64 とし，計算結果には正確さが 128 を超えることを要求した．図 3 中，演算子の右肩に添えられている整数値は，その演算によって実際に計算された近似値の個数である．図より，再計算が頻繁に行われていることがわかる．

### 4.2　要求精度を伝搬できるデータ表現

再計算の頻発を防ぐには，ユーザの要求する精度を踏まえて，末端の定数参照における初期精度が決定できるようにする方法が考えられる．これを実現するために，我々は，計算に用いるデータ型 IFN に対し，与えられた要求精度に従って IFN の値（無限リスト）を生成する関数型としてデータ型 IFNgen を定義し，IFN による計算を，IFN 演算子ではなく IFNgen 演算子，すなわち IFNgen 同士の演算を行う関数で包んで行うようにした．

データ型 IFNgen の定義は次の通りである．

```
addIFN ::  IFNgen -> IFNgen -> IFNgen
addIFN v w a = addIFN' a (v a) (w a)

genIFNfromString ::  String -> IFNgen
genIFNfromString s a = map f [a,a+diffN..]
 where f n = fromString (fromIntegral n) s

accurateValue ::  Int -> IFNgen -> Q
accurateValue a v = head (v a)
```

図 4　**IFNgen** による計算の構成（主要な関数のみ記載）

```
addIFN ::  IFNgen -> IFNgen -> IFNgen
addIFN v w a =
 let a' = f a in addIFN' a (v a') (w a')
```

図 5　**IFNgen** での加算における精度の修正と伝搬

```
type IFNgen = Int -> IFN
```

IFNgen の導入は，図 4 に示すようにいくつかの関数の置き換えだけで実現できる．ライブラリのインターフェースを構成する関数の型が変わるが，IFN の生成，演算子，IFN からの Q の取り出しが一貫して変わるので，数値計算プログラムのソースコードの変更は必要ない．

IFNgen を用いると，ユーザが計算結果に求める正確さが，データーフローの末端の定数の IFN 生成に用いられるようになる．また，個々の IFN 演算子における正確さの下限指定もなされるため，部分式レベルで再計算の必要性の判定がなされ易くなる．結果として，個々の IFN 演算子での再計算回数は低く抑えられる．

IFNgen を用いる場合，図 5 に示すように，オペランドに要求する正確さを個々の演算子において修正するようにすれば（図 5 における関数 `f` の導入），再計算の頻度を更に抑えることも可能になる [†1]．

---

†1　Q 演算子のうち，加算に関しては，オペランドの正確さよりも和の正確さは必ず減少するため，例えば `f a = a + 3` とすることは合理的である．乗除算に関しては，オペランドの正確さと積や商の正確さの関係は一概に言えないので，図 4 や図 5 の `addIFN` の形式による正確さの伝搬・修正は必ずしも妥当ではない．一方，3.6 節で述べた方法で IFN を仮数部長に基づいて再構成すれば，これらの関係は逆になる．

$$
\begin{array}{l}
\text{for } k = 1, \ldots, n \\
\quad \text{for } i = k+1, \ldots, n \\
\quad\quad a_{ik} \leftarrow a_{ik}/a_{kk} \\
\quad\quad \text{for } j = k+1, \ldots, n \\
\quad\quad\quad a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}
\end{array}
$$

図 **6** **LU** 分解（ピボッティング無し）

### 4.3 部分式の計算結果の共有

4.2 節で述べた IFNgen の導入は，単純な数式に対する数値計算に対しては顕著な効果があり，大幅な実行時間の短縮に繋がる．しかしながら，部分式を複数箇所で共有しているような複雑な数式に対しては，計算時間の大幅な増大を招き得る．なぜなら，IFNgen 演算子により数値計算プログラムを構成すると，数値計算プログラムの字面上で共有される部分式の実体が IFN から IFNgen に，つまり無限リストから「無限リストを返す関数」に置き換わるからである．

一般の数値計算プログラムでは，部分式の結果が多くの場面で参照される．典型的な例としては反復計算や行列計算が挙げられる．例えば，ピボッティング無しの LU 分解は図 6 のように表現され，図 7 のように実装できる．このとき，例えば 3 元連立一次方程式の LU 分解に基づく求解において構成されるデータフローは図 8 の通りとなる．図中，丸で囲まれた値が入力データ（行列 $(a_{ij})$ および右辺ベクトル $b_i$），四角で囲まれた値が計算結果（解ベクトル $x_i$）である（LU 分解後の係数行列 $(a'_{ij})$ および前進代入結果のベクトル $y_i$ も示している）．図 8 から明らかなように，一連の計算の中で，多数の部分式の計算結果が多くの箇所で参照されている．部分式の計算結果の共有を行わない場合，計算量は膨大になり得る．

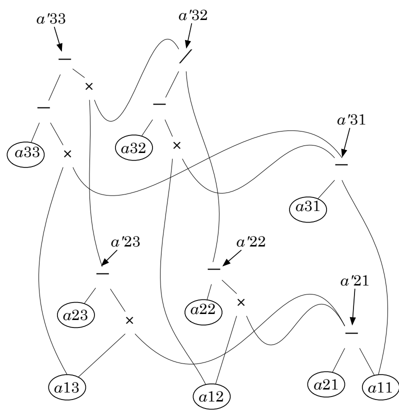
初期プロトタイプにおいては，部分式の計算結果は，IFN すなわち無限リストとして共有され，複数箇所において同一の IFN が参照される．無限リストは複数の要素を含んでおり，個々の参照箇所において必要とされる要素は異なり得るが，同じ部分式に対応する無限リストが多重に生成されることはない．しかしながら，部分式の計算結果を IFNgen すなわち関数にすると，無限リストではなく関数が共有されることになり，共有された関数が生成する IFN は共有さ

```haskell
import Data.Array.IO

luLoop3 ::  IOArray (Int, Int) IFNgen
      -> Int -> Int -> Int -> Int
      -> IO (IOArray (Int, Int) IFNgen)
luLoop3 a k i j n =
  if j <= n then do
    t1 <- readArray a (i,j)
    t2 <- readArray a (i,k)
    t3 <- readArray a (k,j)
    writeArray a (i, j) (t1 - t2 * t3)
    luLoop3 a k i (j+1) n
  else
    return a

luLoop2 ::  IOArray (Int, Int) IFNgen
       -> Int -> Int -> Int
       -> IO (IOArray (Int, Int) IFNgen)
luLoop2 a k i n =
  if i <= n then do
    t <- readArray a (i,k)
    t' <- readArray a (k,k)
    writeArray a (i, k) (t / t')
    luLoop3 a k i (k+1) n
    luLoop2 a k (i+1) n
  else
    return a

luLoop1 ::  IOArray (Int, Int) IFNgen
        -> Int -> Int
        -> IO (IOArray (Int, Int) IFNgen)
luLoop1 a k n =
  if k <= n-1 then do
    luLoop2 a k (k+1) n
    luLoop1 a (k+1) n
  else
    return a

lu ::  IOArray (Int, Int) IFNgen -> Int
     -> IO (IOArray (Int, Int) IFNgen)
lu a n = do
  luLoop1 a 1 n
```

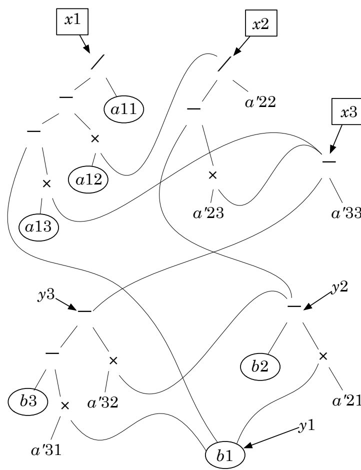
図 **7** **LU** 分解の **Haskell** での実装例

れない．従って，同一の実数を表す多数の IFN が部分式ごとに独立に生成されることになる．このままでは，計算時間の観点から，行列計算などに IFNgen 演算子を単純に用いることは困難である．

部分式共有時の IFN の共有に関する問題に対処するために，個々の IFNgen 演算子には，いわゆるスタティック変数を持たせる方法をとった．そして，IFN




(a) LU 分解




(b) 前進代入と後退代入

**図 8　3 元連立一次方程式の求解におけるデータフロー**

を生成しようとする時点で，過去に生成された IFN があればそれを参照するようにした．この方法は，図 9 に示すコーディングにより可能である．図 9 中，下線部により，数値計算プログラム中における個々の IFNgen 演算子の初回の評価（及び末端の定数参照）の際に，`IORef Maybe IFN` 型の値が一つずつ生成されて対応付けられ，各箇所で一度生成された IFN は二度目以降の同一演算子の評価時には再利用される．

なお，本章で述べた一連の取り組みでは，IFN ライブラリのインターフェースは不変であるため，数値計算アプリケーションの修正は必要ない．

```
import Data.IORef

saveIFN ::  IFNgen -> IFNgen
saveIFN f = unsafePerformIO $ do
  ref <- newIORef Nothing
  let f' a = case t of
    Nothing -> unsafePerformIO $ do
      let v = f a
      writeIORef ref (Just v)
      return v
    Just v' -> unsafePerformIO $ do
      let newv
          = dropWhile (\n -> ac n < a) v'
      return newv
  where t = unsafePerformIO $ readIORef ref
  return f'

addIFN ::  IFNgen -> IFNgen -> IFNgen
addIFN v w = saveIFN $ \a ->
  let a' = f a in addIFN' a (v a') (w a')

genIFNfromString ::  String -> IFNgen
genIFNfromString s
   = saveIFN $ \a -> map f [a,a+diffN..]
 where f n = fromString (fromIntegral n) s
```

**図 9　IFNgen での IFN 共有のための仕組み**

## 5　高速化 2: MPFR ライブラリの導入

### 5.1　概要

初期プロトタイプにおける IFN ライブラリは，動作の実証を目的として開発されたものであり，処理速度を意識した観点からは，実装方式に関連するいくつかの明らかな改善の余地があった．本章では，次の項目について述べる．

- データ表現の再設計と FFI の活用
- MPFR ライブラリの導入

### 5.2　データ表現の再設計と FFI の活用

IFN の初期プロトタイプは，ほぼすべての記述が Haskell でなされていた．IFN ライブラリは，遅延評価による無限リストの活用の仕方がその特徴として挙げられるものの，要素演算である可変長浮動小数点演算等の低レベル処理では遅延評価やリストによる仮数部保持等が合理的と考えられる場面はほとんどなく，処理速度の観点からは最適な実装方式とはいえ

```
import Foreign hiding (newForeignPtr)
import Foreign.Concurrent

data CQ -- Q allocated in C layer
data IFNElem -- wraps Q
   = IFNElem (ForeignPtr CQ) (Ptr CQ)
type IFN = [IFNElem]

-- attach finalizer to CQ
addFinalizer ::  Ptr CQ -> IO IFNElem
addFinalizer q = do
  p <- newForeignPtr q $ c_gc_free q
  return (IFNElem p q)
```

図 10　FFI の活用

```
typedef struct {
  FP *_mp;
  bool _zeroFlag;
} Q;
typedef __mpfr_struct FP;
```

図 11　MPFR を直接利用した Q の定義

なかった．

実装方式の改善策として，IFN 演算の構成要素である Q 演算およびそれより下の階層を，実装言語として C 言語を用いて再設計し，Haskell の FFI (Foreign Function Interface) により IFN ライブラリに統合した．図 10 に，FFI 版におけるデータ型の定義を示す．

C 言語層でのメモリ管理には Boehm GC ライブラリを使用した．C 言語層でのポインタを Haskell 上で扱うにあたっては，Foreign.Concurrent モジュールの newForeignPtr により，Haskell の GC と連携させている（図 10 の c_gc_free は Boehm GC の GC_FREE() を呼び出す）．

FFI 導入のための再設計においては，IFN 演算子の修正はほぼ必要なく，IFN ライブラリとしてのインターフェースにも変更はない †2．

### 5.3　MPFR ライブラリの導入

可変長浮動小数点表現 Q に対する演算以下の階層を C 言語化するのに合わせ，データ型 Q にも変更を加えた（図 11）．具体的には，仮数部の各桁を，符号無し整数の配列の各ビットに対応させた．更に，ビット配置を MPFR ライブラリでの数値表現に一致させることにより，Q 演算を構成する基本演算を MPFR ライブラリの関数に置き換えた．

```
typedef struct {
  mpfr_prec_t _mpfr_prec;
  mpfr_sign_t _mpfr_sign;
  mpfr_exp_t _mpfr_exp;
  mp_limb_t *_mpfr_d;
} __mpfr_struct;
typedef __mpfr_struct mpfr_t[1];
```

図 12　MPFR における浮動小数点表現

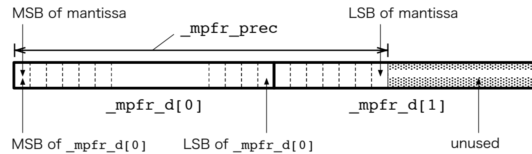



図 13　MPFR における浮動小数点表現の仮数部

MPFR ライブラリにおける数値は図 12 に示す構造体__mpfr_struct で保持される．MPFR では任意長の仮数部の数値を扱うことができる．ユーザが指定する仮数部長は整数値（メンバ_mpfr_prec）で保持されて，その値を下回らないだけの長さのビット長の mp_limb_t 型（典型的には unsigned long int）の配列が仮数部（メンバ_mpfr_d）として確保される．メンバ_mpfr_d の指す領域のビット配置は図 13 に例示する通りである．

MPFR での浮動小数点演算（関数 mpfr_add() など）においては，結果を格納する構造体 __mpfr_struct のインスタンスのメンバ_mpfr_prec で指定された桁数の仮数部が，適切な丸めの元で正確に計算されることが保証されている．従って，仮数部長を適切に管理すれば，Q 演算は MPFR 関数を用いて構成できる．

MPFR ライブラリを使用するにあたっては，関数 mpfr_add() 等の四則演算のみを利用し，データ領域の管理や数値データの整形は自前で行っている．また，Q 演算に特有の状況への対応も MPFR の外で行っている．例えば，MPFR では仮数部長の下限が

†2　一部の環境（Mac OS X など）での実行時には Boehm GC のための初期化ルーチンの呼び出しが必要となる．

定数 MPFR_PREC_MIN で規定されているが IFN における データ型 Q では長さゼロの仮数部を許す．また，データ型 Q はゼロフラグを持っていて，真のゼロの扱いや，(浮動小数点表現における見かけ上の) 結果がゼロとなる場合の対応が MPFR とは異なる．これらへの対応は明示的に記述し，Q 演算子を再設計した．

## 6 数値実験および評価

本稿で述べた IFN の高速化の効果の評価のため，いくつかの数値計算アプリケーションを題材に用いて実測を行った．

### 6.1 実測環境と比較項目

実測に用いた環境は次の通りである：

- ハードウェア: MacBook Pro, Intel Core i7 3GHz, メモリ 16GB
- ソフトウェア: Mac OS X 10.10.3, GHC 7.8.4, Boehm GC 7.4.2, GCC 4.8.4, MPFR 3.1.1, GMP 5.1.1

実測結果は主として次の 4 種類の実装を比較する．

**ORG** 初期プロトタイプ

**IFNgen** IFNgen の導入に加えて部分式ごとの IFN を共有化したもの（第 4 章参照）

**MPFR** FFI により MPFR を使用したもの（5.3 節参照）

**MPFR+IFNgen** IFNgen 導入の上で MPFR 化したもの

また，IFN 以外のライブラリとの比較のために，iRRAM [5] を単独で使用して同様の計算を行った結果も添える．

### 6.2 実験 1: 単純な数式の計算

Rump の例を題材として，数値実験を行った．用いたソース記述は図 14 の通りで，式 (1) に対して部分式を明示的に共有させる記述とした．`rump 77617 33096` を様々な精度で求めた．結果を図 15 に示す．図 15 の横軸は計算結果に対する要求精度（正確さ），縦軸は実行に要した時間（単位は秒）である．図 15 中，ORG，IFNgen，MPFR，MPFR+IFNgen とラベルづけされた曲線は，それぞれ，4 種類の比較項目

```
rump a b =
  let b2 = b*b in
  let b4 = b2*b2 in
  let b6 = b4*b2 in
  let b8 = b4*b4 in
  let a2 = a*a in
  let t1 = 333.75 * b6 + a2 *
    (11 * a2 * b2 - b6 - 121 * b4 - 2)
      + 5.5 * b8 in
  let t2 = a / (2 * b) in
    t1 + t2
```

図 14 Rump の例（部分式を明示的に共有）

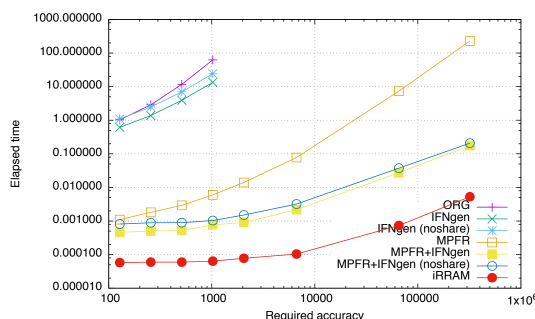



図 15 Rump の例の実測結果

に対応している．また，“noshare” と記された曲線は，部分式ごとの IFN を共有しない場合（4.3 節参照）の実行結果である．

図 15 中，曲線 ORG と MPFR，IFNgen と MPFR +IFNgen をそれぞれ比較すると，要求精度によらず，実行速度が約 3 桁異なることがわかる．すなわち，FFI により MPFR を導入した効果は非常に大きい．なお，MPFR 導入による IFN ライブラリの挙動の変化は，(処理速度の違いを除けば) 大きくはないと考えられる．

図 15 中，ORG と IFNgen，MPR と MPFR +IFNgen をそれぞれ比較すると，要求精度を大きくするに従い速度差が広がっている．無限リストである IFN の初期精度（先頭要素の精度）の変更と部分式の IFN を共有する仕組みの導入は，要求される精度が大きくなるほど，大きな効果となって現れるといえる．

曲線 ORG と MPFR+IFNgen を比較すると，要求精度（正確さ）が 512 の段階で 4 桁以上の高速化が達成されており，要求精度を大きくするほど大きな効果が得られることが予想される．

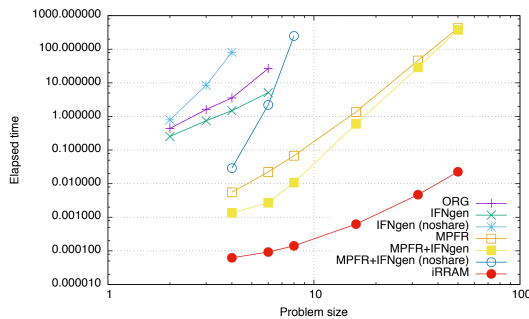



図 16 **Hilbert** 行列を係数行列に持つ連立一次方程式の求解の実測結果（横軸は問題サイズ）

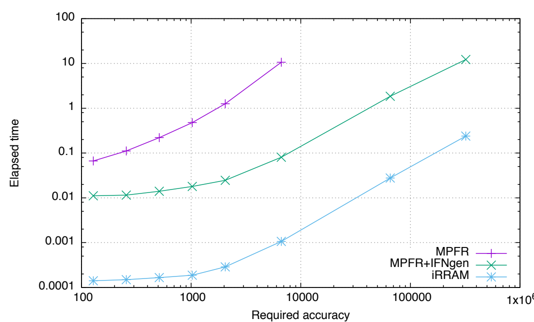



図 17 **Hilbert** 行列を係数行列に持つ **8** 元連立一次方程式の求解の実測結果（横軸は結果に対する要求精度（正確さ））

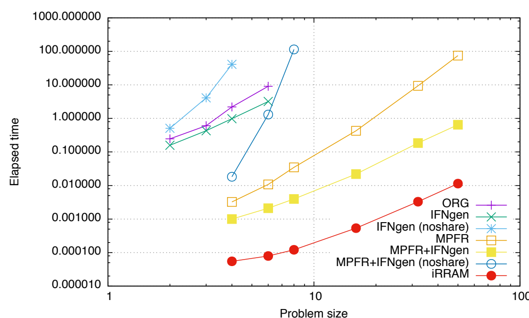



図 18 単純な行列を係数行列に持つ連立一次方程式の求解の実測結果（横軸は問題サイズ）

図 15 には，比較のため，iRRAM ライブラリによる実測結果も示している．IFNgen+MPFR は iRRAM よりも 1 桁以上低速であった．

### 6.3 実験 2: 行列計算

Hilbert 行列を係数行列に持つ連立一次方程式 $Hx = b$ の求解を題材に実行速度を測定した．Hilbert 行列は各要素が $h_{i,j} = 1/(i + j - 1)$ の正方行列であり，条件数が大きい．Hilbert 行列を係数行列にもつ連立一次方程式は精度よく求めることが困難であることが知られているが，IFN ライブラリを用いれば，任意桁数の正確さで求解結果を得ることができる．実測にあたっては，求解プログラムの LU 分解部分は図 7 の記述をそのまま用いた．右辺ベクトル $b$ は $(1\ 0\ \cdots\ 0)^T$ とした．実測結果を図 16 および図 17 に示す．

図 16 は，計算結果に要求する正確さを 128 とし，問題サイズ（次元数）を変化させた実測結果である．縦軸は実行に要した時間（単位は秒）である．図 16 中，ORG，IFNgen，MPFR，MPFR+IFNgen とラベルづけされた曲線は，それぞれ，4 種類の比較項目に対応している．

図 16 中，曲線 ORG と MPFR，IFNgen と MPFR +IFNgen をそれぞれ比較すると，問題サイズによらず，4 桁を超える高速化が達成されていることがわかる．曲線 ORG と MPFR+IFNgen の比較でも，問題サイズによらず 4 桁以上の高速化が確認できる．iRRAM による実行結果は，IFN を用いたいずれの場合よりも 1 桁以上高速である．

図 17 は，問題サイズを固定し（行列サイズ $8 \times 8$），結果に要求する精度を変化させた結果を示している．MPFR，MPFR + IFNgen，および比較対象の iRRAM での結果を載せている．結果に対する要求精度を高くすると計算時間は伸びるが，その増加の傾向は，iRRAM と同様であることがわかる．

図 18 は，図 16 と同様の測定に際して係数行列を Hilbert 行列ではなくより単純で解き易い行列に置き換えた場合 [†3] の実測結果である．行列サイズを変化させたときの傾向は図 16 の場合とほぼ同様であるが，曲線 MPFR + IFNgen は形状が大きく異なっている．iRRAM の実行の様子と比較すると，IFN（とくに MPFR+IFNgen）は，数値的な条件が悪い場合

†3 具体的には，行列サイズ $N \times N$ に対して $a_{ii} = N + i$ とし，その他の要素は 1 とした．

の挙動に改善の余地があると言える．

図 16 および図 18 において，“noshare” と記された曲線は，図 15 の Rump の例の場合と大きく異なっている．4.3 節で述べた通り，反復計算等によって部分式が多数箇所で参照される場合には，個々の部分式に対応する IFN を共有させる必要があることがわかる．

## 7　関連研究

IFN の基本アイデアは，IS (Improving Sequences) [6] から得られた．IS はある真の値に対する近似数値の遅延リストで，近似の度合いの昇順に並ぶ有限列である．IS 同士の演算の合成により，不要な計算を省いた効率の良いプログラムを構成することができ，組み合わせ最適化問題等を題材に有用性が示されている．IFN は遅延ストリームが実数を表すことに特化している点や無限リストである点などが IS とは異なっている．

数値計算プログラムの実行時に桁落ちを検出し，結果の不正確さを判定するツールはいくつか提案されている [7] [8]．桁落ちの要因まで解析する機能等も開発されている．しかしながらこれらのツールは計算結果の精度を保証する仕組みは持っていない．

浮動小数点表現ではなく，有理数表現（整数の分母と分子の組による表現）を用いる方法もある．有理数表現でも Rump の例のような単純な数式であれば正確な計算ができるが，計算に要するコストが大きいため複雑なプログラムでの使用には向いていない．場合によっては，一種の丸めによる近似が必要になる．

計算結果の精度を把握する必要のある数値計算では，区間演算も有用である [9]．IFN における浮動小数点表現 Q は，それで表現される実数の存在区間が厳密に定義されるという意味で，区間演算の一種とみなすこともできる．

正確な実数値の計算を計算機上で実現する方法は数多く研究されている [10]．例えば，連分数表現 [11] [12]，linear fractional transformations と呼ばれる手法によるもの [13]，各桁を符号付き数列で表すもの，両端を有理数で表現する入れ子区間列表現 [14]，等が挙げられる．scaled-integer representation と呼ばれる手法 [15] では，実数は関数で表現され，個々の算術演算は関数呼び出しや関数の構築（関数適用や関数抽象）を有理数係数を用いて組み合わせることにより実現される．これらの手法には実数値表現に遅延ストリームを用いるものもある．ライブラリとしてまとめられて使用可能なものもある [16] [17] [5] [18] [19]．IFN は入れ子区間列を用いたコーシー列での数値表現に相当する．単純な優劣の比較は難しいが，IFN は近似値列の表現方法が浮動小数点表現に基づいておりシンプルで，効率改善の余地が多いと考えられる．

## 8　おわりに

本稿では，著者らによって開発された IFN ライブラリの初期プロトタイプの高速化の試みについて述べた．IFN レベルでの算術演算の改善，および，Q 演算レベルでの実装方式の改善により，4 桁以上の高速化の達成が確認された．IFN ライブラリは依然としてプロトタイプの域を出ないが，卓上計算機的な利用には十分使用可能な性能であるといえる．

IFN ライブラリの高速化により，規模の大きい数値計算を用いた評価が可能になった．今後の課題としては，実用を視野に入れた効率改善や更なる性能分析や，アプリケーション記述を支援するサポートツールの開発が挙げられる．

謝辞 本研究の一部は広島市立大学特定研究費（課題番号 1030301）の助成を受けている．

## 参 考 文 献

[1] Kawabata, H., and Iwasaki, H.: Improving Floating-Point Numbers: a Lazy Approach to Adaptive Accuracy Refinement for Numerical Computations, 第 16 回プログラミングおよびプログラミング言語ワークショップ PPL2014, Mar. 2014.
[2] The GNU MPFR Library, http://www.mpfr.org
[3] Microprocessor Standards Committee of the IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic, IEEE Standard 754, 2008.
[4] Rump, S.M.: Algorithms for Verified Inclusion, in R. Moore, editor, Reliability in Computing, Perspectives in Computing, pp.109–126. Academic Press, New York, 1988.
[5] Müller, N.T.: The iRRAM: Exact Arithmetic in C++, http://irram.uni-trier.de
[6] Iwasaki, H., Morimoto, T., Takano, Y.: Prun-

ing with Improving Sequences in Lazy Functional Programs, Higher-Order Symb. Comput., Vol.24, pp.281–309, 2011.

[ 7 ] Jeffrey, K.H., Lam, M.O., Stweart, G.W.: Dynamic floating-point cancellation detection, WHIST 2011, 2011.

[ 8 ] Benz, F., Hildebrandt, A., Hack, S.: A Dynamic Program Analysis to Find Floating-Point Accuracy Problems, SIGPLAN Not., PLDI 2012, Vol.47, No.6, pp.453-462, 2012.

[ 9 ] Knuth, D.E: The Art of Computer Programming, 3rd ed., Sec.4.2.2, Addison-Wesley, 1997.

[10] Gowland, P., and Lester, D.: A Survey of Exact Arithmetic Implementations, CCA2000, LNCS2064, pp.30–47, 2001.

[11] Gosper, W.: Continued fractions, `http://www.inwap.com/pdp10/hbaker/hakmem/cf.html`

[12] Vuillemin, J.: Exact real computer arithmetic with continued fractions, IEEE Transactions on Computers Vol.39, pp.1087–1105, 1990.

[13] Potts, P.: Exact Real Arithmetic using Möbius Transformations, PhD thesis, Depertment of Computing, Imperial College of Science, Technology and Medicine, University of London, 1990.

[14] Escardó, M.: Introduction to exact numerical computation, Notes for a tutorial at ISSAC2000, 2000.

[15] Boehm, H.-J., Cartwright, R., Riggle, M., and O'Donnell, M.J.: Exact real arithmetic: A case study in higher order programming, ACM Symosium on Lisp and Functional Programming, pp.162–173, 1986.

[16] Lester, David.R.: The world's shortest correct exact real arithmetic program?, Information and Computation, Vol.216, pp.39–46, 2012.

[17] Guy, Martin: bignum / BigFloat, `http://medialab.freaknet.org/bignum/`

[18] Edalat, A.: Exact Real Number Computation Using Linear Fractional Transformations, Final Report on EPSRC grant GR/L43077/01, 2001.

[19] Lambov, B.: RealLib: An efficient implementation of exact real arithmetic, Mathematical Structures in Computer Science, Vol.17, pp.81–98, 2007.