IEEE754と数値計算 : 浮動小数点演算の特徴とは?

皆本, 晃弥
佐賀大学理工学部知能情報システム学科

https://doi.org/10.15017/1470409

出版情報：九州大学情報基盤センター広報 : 全国共同利用版. 2 (1), pp.63-84, 2002-03. 九州大学情報基盤センター
バージョン：
権利関係：

IEEE754と数値計算

－浮動小数点演算の特徴とは？－

皆本 晃弥*
佐賀大学理工学部知能情報システム学科

1 はじめに

この解説記事の表題にあるIEEE754とは，多くのメーカが採用している2進浮動小数点演算規格で，正式名称はANSI/IEEE std 754-1985, IEEE Standard for Binary Floating-Point Arithmeticです．通常，IEEE754とかIEEE標準と呼ばれていますが，IEEE標準といった場合，IEEE854[1]を含む場合もありますので，本稿ではIEEE754という用語を使うことにします．

「浮動小数点規格は知っているよ，例えば倍精度浮動小数点数は，浮動小数点フォーマット

$$\pm(\frac{d_1}{\beta}+\frac{d_2}{\beta^2}+\cdots+\frac{d_p}{\beta^p})\times\beta^e \qquad (0\leq d_i<\beta) \tag{1}$$

において，$p=53, \beta=2, -1022\leq e\leq 1023$を満たすものでしょ」という方は，まず，次の文章を読んでください([2])．

> When discussing the floating-point capabilities of a new machine, we always ask the manufacturer two questions:
>
> Does the machine use IEEE arithmetic?
>
> Does it support graceful underflow and provide user control of rounding mode and exception flags?
>
> Frequently the designer believes his machine is using IEEE arithmetic when it is using only the IEEE formats without the other required features.
>
> - W. J. Cody -

いかがでしょうか? IEEE754は浮動小数点フォーマットだけを定めている訳ではないのです．しかし，読者が冒頭のように答えたからといっても何ら珍しいことではありません．というのも，数値計算や数値解析の本を見ると，意外に浮動小数点に関する記述は少なく，IEEE754をサポートしているコンピュータを使った場合，どのような計算結果が得られるかを規格に沿って述べているものが少ないからです．

*E-mail : minamoto@is.saga-u.ac.jp
[1]IEEE754をベースに作られた10進浮動小数点規格です．

そこで，今回はIEEE754を前面に出し，例を通じてそれをサポートしているコンピュータ上で実行された数値計算結果を吟味していくことにします．とはいえ，細かく規格([6])を見ていくと，実に実装しなければならない項目が約90，推奨している項目が約20ある[2]ので，これらをすべて解説することはできません．そこで本稿では，最初に概要を述べた後，いくつかの項目にしぼって話を進めることとします．

2　IEEE754の概要

IEEE754とは，IEEE[3]の浮動小数点ワーキンググループ(IEEE Task 754)が1985年に定めた2進浮動小数点演算規格です([6])．ワーキンググループでは，いくつかのモデル[4]が検討されましたが，IEEE754で採用されている規格は，1978年にW.Kahan, J.Coonen, H.S.Stoneによって開発されたKCSという規格です．

2.1　IEEE754の目的

IEEE754の目的は，すべてのソフトウェア，すべてのハードウェア，またはすべてのハードウェアとソフトウェアの組み合せにおいて実装できるような規格になるということです．これが実現すると，あるコンピュータから別のコンピュータにプログラムを移植した場合でも，両方のコンピュータがIEEE754をサポートしていれば，共に全く同じ結果を得ることができるようになります．

IEEE754が検討された背景として，1970年代までに小型計算機のための浮動小数点方式が数多く登場したため，「規格を定めることに意義がある」と考える研究者が増えたということが挙げられます．

2.2　IEEE754の規定事項と規定外事項

主な規定事項は，

1. 基本および拡張浮動小数点フォーマット
2. 加算，減算，乗算，除算，平方根，剰余，比較操作
3. 整数と浮動小数点形式の変換
4. 異なった浮動小数点フォーマット間の変換
5. 浮動小数点の基本フォーマットと10進数との変換
6. 浮動小数点例外と非数(NaN)の扱い

[2]文献[6]には，"shall"は義務，"should"が推奨を表すと書いてあり，単純にこれらを数えるか，文脈を考えて，例えば，2つの"shall"を1つの義務ととらえるかで数が変わってきます．

[3]Institute of Electrical and Electronics Engineers(アメリカ電気電子技術者協会)．

[4]主に3つの規格が検討されました([1])．

で，逆に規定されていない事項は

1. 10進数と整数のフォーマット
2. NaN の符号と仮数 (significand) の解釈
3. 拡張フォーマットにおける 2 進数と 10 進数間の変換

です.

IEEE754 が検討されていたころや定められた後にも，それに対抗するかのごとく浮動小数点規格が提案されました ([4],[7]). IEEE754 に対する批判もありますが，ここでは，IEEE754 が最もすぐれた浮動小数点演算規格であるかどうかは議論せず，あくまで規格として受け入れて IEEE754 とそれによる数値計算結果について述べたいと思います.

2.3 IEEE754 を採用している CPU

IEEE754 を採用している CPU には次のようなものがあります.

Intel x86 系 (Pentium シリーズ)，Compaq/DEC Alpha 21264,
SUN SPARC，Motorola 68xxx 88xxx，HP PA-RISC，IBM PowerPC [5]

これらの CPU を搭載しているコンピュータであれば，IEEE754 に基づいた浮動小数点演算が実行されることになります．ほとんどの研究室や演習室には，いわゆる Windows マシンや Solaris マシンといったものがあり，それらは Pentium シリーズや UltraSPARC シリーズといった CPU を搭載しているので，これらのコンピュータで計算している人は知らず知らずのうちに IEEE754 に基づく演算を行っていることになります．なお，自分の使用している計算機が IEEE754 に準拠しているかどうか知りたい場合は，J.Hauser 氏が作成した TestFloat というソフトを使ってみるという手があります[6]．ただし，TestFloat が出力する結果を確実に理解するには，TestFloat のドキュメントにも書かれているように IEEE754 の知識が必要となります．また，Intel と SPARC 以外の環境では，TestFloat のコンパイルのために各種設定ファイルを自分で用意しなければなりません.

TestFloat による出力例

```
No errors found in float32_to_floatx80.
No errors found in float64_to_floatx80.
No errors found in floatx80_to_int32, rounding nearest_even.
No errors found in floatx80_to_int32, rounding to_zero.
No errors found in floatx80_to_int32, rounding down.
No errors found in floatx80_to_int32, rounding up.
```

[5] IBM としましたが，Motorola，IBM，Apple の共同開発です.
[6] http://www.cs.berkeley.edu/~jhauser/arithmetic/index.html

出力結果の意味をすべて理解するにはIEEE754の詳細な知識が必要ですが，とりあえず計算機がIEEE754に準拠しているかどうか知りたい場合には，上記の例のように，すべての項目において"No errors"と出力されていることを確認すればいいでしょう．

ちなみに，本稿で使用した計算機環境をは次の通りです．すべてIEEE754に準拠しています．

CPU	OS	コンパイラ
UltraSPARC-II	Solaris 2.6	gcc 2.8.1
Alpha 21264	Kondara MNU/Linux 1.0	gcc 2.91.66
Pentium4	Laser5 Linux 7.1	gcc 2.96

本稿では，これらのCPUで計算した結果をそれぞれSPARC，Alpha，Pentium4による結果と呼ぶことにします．また，gccによるコンパイルでは原則としてコンパイラオプションは使用していません．使用したところだけ，そのことを明記するようにします．

3　浮動小数点演算の基礎知識

この章では，第4章以降に登場する基本的な用語を簡単に解説します．

3.1　丸めと丸め誤差

実数を一定の有限桁で近似することを丸めといい，そのときに発生する誤差のことを丸め誤差といいます．丸め誤差は有限桁で実数を近似するために生じる誤差なので，丸め誤差だけはすべてのアルゴリズムに存在します．

浮動小数点を使う限り，丸め誤差は避けては通ることはできません．相対誤差が小さいという意味で数学の各種公式を使ってコンピュータ上で計算してよいという妥当性を得ることができるのです．

3.2　前進誤差解析と後退誤差解析

前進誤差解析 (forward error analysis) は計算結果 $\hat{y}$ と真の解 y との差 $|y-\hat{y}|$ を直接評価しようとするものです．なお，$\hat{y}$ の絶対値誤差 $|y-\hat{y}|$ や相対誤差 $\frac{|y-\hat{y}|}{|y|}$ を前進誤差と呼びます．

それに対し，後退誤差解析 (backward error analysis) では，データ誤差があるだけで計算誤差は発生しないものと考えます．このように，計算誤差をすべてデータ誤差に押しつけて，実際に得られた計算値はデータ誤差を含んだ入力データによる正しい計算と見なすのです．

後退誤差解析の良さは，計算された値を必要としないということです．また，どのように計算した値を表現するかは本質的ではありません．要は (真の入力+誤差) という形をしていればいいのです．例えば，入力 x と出力 y の関係が $y=f(x)$ と表されているとき，出力の近似 $\hat{y}$ を入力誤差 Δx を使って $\hat{y}=f(x+\Delta x)$ と表します．このとき，$|\Delta x|$ や $\frac{|\Delta x|}{x}$ を後退誤差と呼びます．

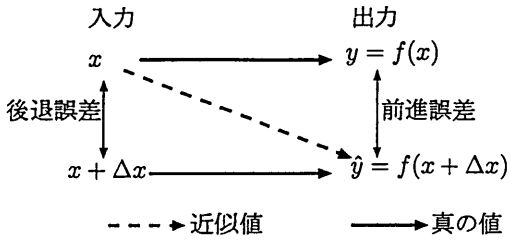


図 1: 前進誤差と後退誤差の関係

3.3 ulp

ulp(unit in the last place) とは，浮動小数点数の最終桁のずれを見るための単位で，浮動小数点結果の正確さを表すのに使われます．

正規化された浮動小数点数[7]

$$y = \pm(\frac{d_1}{\beta} + \frac{d_2}{\beta^2} + \cdots + \frac{d_p}{\beta^p}) \times \beta^e \quad (0 \leq d_i < \beta,\ d_1 \neq 0,\ e_{min} \leq e \leq e_{max})$$

の 1ulp は，

$$ulp(y) = 0.00\cdots01 \times \beta^e = \beta^{e-p}$$

であり，任意の実数 $x \in \mathbf{R}$ に対して x と y との誤差は $\frac{|y-x|}{ulp(y)}$ulp となります．

例えば，$\beta = 2$, $p = 3$, $e_{min} = -1$, $e_{max} = 3$ とすると，$ulp(y) = 2^{e-3}$ となります．このとき，正規化された浮動小数点数の分布は図 2 のようになります．

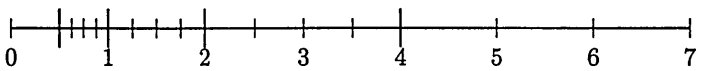
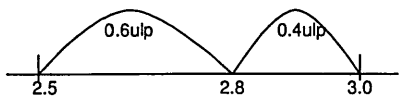


図 2: $\beta = 2, p = 3, e_{\min} = -1, e_{\max} = 2$ の時の正規化数

ここで，真の値を $x = 2.8$ とし，その近似値を $y = fl(x)$ とすると，$fl(x) = (\frac{1}{2} + \frac{1}{2^2}) \times 2^2 = 3$ なので，$ulp(y) = 2^{2-3} = \frac{1}{2}$ となります．よって，x と y との誤差は $\frac{|y-x|}{ulp(y)} = \frac{0.2}{0.5} = 0.4$ulp となります．



この例から分かるように，計算結果が正しい答えに最も近くても，0.5ulp の誤差がある可能性があります．この 0.5ulp に対応する絶対値誤差は，

$$|0.d_1d_2\ldots d_p - 0.d_1d_2\ldots d_p\frac{\beta}{2}| \times \beta^e = \frac{\beta}{2} \times \beta^{-p-1} \times \beta^e$$

[7] $d_1 \neq 0$ とすることを正規化するといいます．

となります．上の例では $\beta = 2$, $p = 3$, $e = 2$ なので $\frac{2}{2} \times 2^{-4} \times 2^2 = 0.25$ が 0.5ulp に対応する絶対値誤差になっています．また，0.5ulp に対応する相対誤差 E_r は，真の値は必ず β^{e-1} と β^e の間にあるので

$$\frac{\frac{\beta}{2} \times \beta^{-p-1} \times \beta^e}{\beta^e} \leq E_r \leq \frac{\frac{\beta}{2} \times \beta^{-p-1} \times \beta^e}{\beta^{e-1}},$$

つまり，

$$\frac{1}{2}\beta^{-p} \leq E_r \leq \frac{\beta}{2}\beta^{-p}$$

を満たします．なお，上式を

$$\frac{1}{2}\beta^{-p} \leq \frac{1}{2}\mathrm{ulp} \leq \frac{\beta}{2}\beta^{-p}. \tag{2}$$

と書くことがあります ([3])．

不等式 (2) の上限を $u = \frac{\beta^{1-p}}{2}$ とすると，ある実数が最も近い浮動小数点数に丸められたとき，その相対誤差は u 以下になります．この u のことを丸めの単位 (unit roundoff) といい，これは誤差解析においてよく使われる単位です．この丸めの単位を使うと，実数と浮動小数点数の集合 $\mathbf{F}$ との関係を次のように表すことができます ([5])．

> 定理 1 $\mathbf{F}$ の最小値および最大値をそれぞれ $x_{\min}$, $x_{\max}$ とする．このとき，$\forall x \in \mathbf{R}$ が $x_{\min} \leq x \leq x_{\max}$ を満たすならば，
>
> $$fl(x) = x(1+\delta), \qquad |\delta| < u \tag{3}$$
>
> が成り立つ．

4　IEEE754 フォーマットと演算

この章では IEEE754 の各種フォーマットと演算について解説します．また，数学関数についてもここで触れることにします．

4.1　フォーマット

IEEE754 では，浮動小数点フォーマット (1) におけるパラメータは表 1 のように定められています．図 3 において，msb は最上位ビット (most significant bit) を表し，lsb は最下位ビット (least significant bit) を表します．また，s は符号ビット，e は指数部，f は仮数部です．

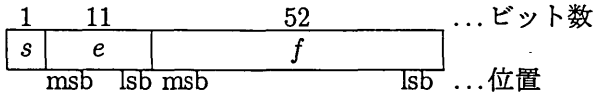


図 3: 倍精度形式

表 1 のように，IEEE754 では単精度，倍精度，拡張単精度，および拡張倍精度という 4 つのフォーマットを定義しています．このようにパラメータを定義していますが，必ずサポー

表 1: IEEE 754形式のパラメータ

	形式			
パラメータ	単精度	拡張単精度	倍精度	拡張倍精度
p	24	≥ 32	53	≥ 64
$e_{\max}$	$+127$	$\geq +1023$	$+1023$	$> +16383$
$e_{\min}$	-126	≤ -1022	-1022	≤ -16382
指数長 [bit]	8	≥ 11	11	≥ 15
形式長 [bit]	32	≥ 43	64	≥ 79

トしなければならないのは単精度だけで，倍精度に関しては推奨事項もありません．これは，IEEE754 では拡張精度にもっとも重点がおかれているためで，代わりに，基本精度 (単精度・倍精度) に対応して，拡張フォーマット (拡張単精度・拡張倍精度) を実装することを推奨しています．

また，IEEE754 では，浮動小数点表現を一意にするために，仮数部の先頭ビットを 1 とします．このことを浮動小数点を正規化するといいます．これは，浮動小数点フォーマット (1) において，$d_1 = 1$ とすることに対応します．正規化された浮動小数点を表現する場合は，仮数部の先頭ビットはつねに 1 なので，これに対応するビットを用意しておく必要がありません[8]．したがって，図 3 において仮数部が 52 ビットとなっていますが，実際は 53 ビット分の情報を持っていることになります．

4.2 基本演算

IEEE754 のすべての演算は正確に計算したものを最後に浮動小数点に丸めたものに一致しなくてはならないので，浮動小数点演算モデルは，$\mathbf{F}$ を IEEE754 形式で表現可能な浮動小数点数とする[9]と，次のようになります．

浮動小数点モデル

任意の $x, y \in \mathbf{F}$ に対して，

$$fl(x \odot y) = (x \odot y)(1+\delta), \qquad |\delta| \leq u, \qquad \odot = +, -, /, *. \tag{4}$$

IEEE754 では平方根も基本演算として定められているので，このモデルは平方根に対しても成立します．

また，このモデルを使うと基本演算の後退誤差解析が可能になります．例えば，任意の浮動小数点数 $x_i \in \mathbf{F} (i = 1, 2, \cdots, n)$ に対して，$s_n = \sum_{i=1}^{n} x_i$ の近似値 $\hat{s}_n$ を考えます．ここで，誤差解析に必要な補題を挙げておきます．この補題は数学的帰納法により比較的容易に証明することができます ([5])．

[8]このビットのことを隠れビット (hidden bit) といいます．
[9]ただし，NaN, 無限大，符号付きゼロは除きます．

補題 1 $|\delta_i| \leq u$, $\rho_i = \pm 1 (i = 1, 2, \cdots, n)$, $nu < 1$ とする．このとき，

$$\prod_{i=1}^{n} (1+\delta_i)^{\rho_i} = 1 + \theta_n \tag{5}$$

が成り立つ．ただし，$|\theta_n| \leq \dfrac{nu}{1-nu} =: \gamma_n$ である．

最初に，モデル (4) を用いて $n = 2$ まで計算すると，

$$\begin{aligned} \hat{s}_1 &= x_1(1+\delta_1) \qquad |\delta_1| \leq u \\ \hat{s}_2 &= (\hat{s}_1 + x_2)(1+\delta_2) = (x_1(1+\delta_1) + x_2)(1+\delta_2) \qquad |\delta_2| \leq u \end{aligned}$$

となります．ここで，最悪の誤差を見積もるためには，補題 1 より $\delta = \delta_1 = \delta_2$ とし，γ_2 を使って過大評価すればよいことが分かるので，$\delta = \delta_1 = \cdots = \delta_n$ として，上記の計算を繰りかえすと，

$$\begin{aligned} \hat{s}_n &= x_1(1+\delta)^n + x_2(1+\delta)^{n-1} + \cdots + x_n(1+\delta) \\ &= x_1(1+\theta_n) + x_2(1+\theta_{n-1}) + \cdots + x_n(1+\theta_1) \end{aligned} \tag{6}$$

となります．従って，後退誤差解析による結果 (6) より，前進誤差評価

$$|s_n - \hat{s}_n| \leq \gamma_n \sum_{i=1}^{n} |x_i| \tag{7}$$

を得ることができます．このように後退誤差 $\theta_i (1 \leq i \leq n)$ の上限 γ_n を使うと，近似値 $\hat{s}_n$ を計算することなく前進誤差 $|s_n - \hat{s}_n|$ を見積もることができます．

4.3　三角関数，指数関数

IEEE754 では，三角関数や指数関数といった超越関数について何も規定されていません．これは，超越関数については，無限大の精度で計算した後，その結果を丸めた場合と同じ精度を短時間で得るようなアルゴリズムが存在しないためです．そのため，超越関数については各 CPU メーカの言葉を信じるしかありません．

例えば，インテルのマニュアルには次のように書かれています ([12])．

> Pentium プロセッサや IA-32 以降のプロセッサでは，超越関数におけるワースト・ケースの誤差は，偶数の最近値へ丸める場合で 1ulp 未満，その他のモードで丸める場合で 1.5ulp 未満になる．超越関数は，入力オペランドに関しては，命令がサポートする領域全体を通じて単調関数であることが保証されている．

このマニュアルを素直に読むと，入力オペランドに関して単調性があるとはいえ，最大で 1.5ulp の誤差があるので，超越関数を含んだ値を厳密に評価しようとした場合，Pentium シリーズの超越関数を利用するのは危険だと言えるでしょう．従って，確実に超越関数の真値を包み込みたい場合，スピードは犠牲になりますが，別に必要な関数を用意する必要があります[10]．

[10] ここでいう単調性が，丸めモードを切り替えた場合に必ず真の値を包み込んでくれるという意味ならば，そ

5 IEEE754 の特別な数と例外

IEEE754 の例外とデフォルトの結果は表 2 のようになります．また，IEEE754 の特別な値は表 3 のようになります．これらの特別な数は丸めモード指定と並んで IEEE754 の大きな特徴となっています．この節では，これらの数について解説します[11]．

表 2: IEEE 演算の例外とデフォルト結果

例外	例	デフォルト値
無効演算 (Invalid operation)	$0/0$, $\sqrt{-1}$	NaN
オーバーフロー (Overflow)		$\pm\infty$
アンダーフロー (Underflow)		非正規化数
ゼロ除算 (Divided by zero)	$2/0$	$\pm\infty$
不正確 (Inexact)	$fl(x \odot y) \neq x \odot y$ のとき	正確な丸め

表 3: IEEE 754 の特別な値

指数	仮数	表現値
$e = e_{\min} - 1$	$f = 0$	± 0
$e = e_{\min} - 1$	$f \neq 0$	$0.f \times 2^{e_{\min}}$
$e_{\min} \leq e \leq e_{\max}$	—	$1.f \times 2^{e}$
$e = e_{\max} + 1$	$f = 0$	$\pm\infty$
$e = e_{\max} + 1$	$f \neq 0$	NaN

5.1 非数 (NaN)

NaN(Not a Number) は $0/0$ や $\sqrt{-1}$ など，数でなくなる状況のときに生成される特殊な数であり，それはどこかで Invalid operation が起こったことを示しています (表 2)．いったんある部分が NaN として評価されると，それ以降の評価はすべて NaN として評価されます．また，NaN には順序や同値関係はありません．表 4 に NaN を発生させる演算例を挙げておきますのでプログラムと共に参考にしてください．

の存在範囲を見積もるためにメーカが用意した関数を利用してもいいのですが，このマニュアルからだけでは判断がつかないので，メーカに問い合わす必要があるでしょう．

[11] これらの一部については拙著 [11] にも書いてあります．

表 4: NaN を発生する演算

演算	NaN を発生する原因
+	$\infty + (-\infty)$
×	$0 \times \infty$
/	$0/0, \infty/\infty$
$\sqrt{}$	$\sqrt{x}$ $(x < 0$ の時$)$

NaN を含んだプログラム (Linux 用) [12]

```
/*
  NaN のテスト
 */
#include <stdio.h>

int main(void)
{
  double a, b;
  a=0.0/0.0;        /* NaN の生成 */
  printf("a=%lf isnan=%d \n ", a, isnan(a));
  b=1.0+a;         /* NaN による演算 */
  printf("b=%lf \n", b);
  return 0;
}
```

計算結果

```
a=nan isnan=1
b=nan
```

なお，このプログラム中の isnan 関数は，引数が NaN のとき 1 を返す関数です.

5.2 無限大 (Infinity)

無限大 (Infinity) は，オーバーフローが起きたときでも処理を続けることができるようにするために導入された特殊な数で，コンピュータ上では Inf と表示されます．Inf の生成は数学の無限大記号 ∞ と同じような扱いができるように定められています．そのため，いったん Inf と評価されたとしても，$\lim_{x\to\infty} \frac{1}{x} = 0$ のようなルールがあるので，最終的な演算結果は浮動小数点数になる場合があります.

[12]Solaris には isnan 関数がないようなので，SPARC 上でプログラムが動かない場合は isnan 関数の部分を削除してください.

無限大を含んだプログラム (Linux用) [13]

```
/*
  inf のテスト
 */
#include <stdio.h>

int main(void)
{
  double a, b;
  a=1.0/0.0;        /* 無限大の生成 */
  printf("a=%lf  isinf=%d \n ", a, isinf(a));
  b=1.0/a;         /* 無限大による除算 */
  printf("b=%lf \n", b);
  return 0;
}
```

計算結果

```
a=inf  isinf=1
b=0.000000
```

なお，このプログラム中のisinf関数は，引数が無限大ならば1を返す関数です．

5.3 符号付きゼロ (signed zero)

IEEE754では，2種類のゼロ，つまり+0と-0を規定しています．ただし，$+0 = -0$と定められています．もし符号付きゼロがなかったら，右極限，左極限ということは考えられません．つまり，

$$\lim_{x \to +\infty} \frac{1}{1/x} = \lim_{x \to +\infty} x = +\infty, \qquad \lim_{x \to -\infty} \frac{1}{1/x} = -\infty$$

ということが成り立たなくなります．というのも，符号付きゼロがなければ，

$$\frac{1}{\frac{1}{+\infty}} = \frac{1}{0}, \qquad \frac{1}{\frac{1}{-\infty}} = \frac{1}{0}$$

となってしまうからです．符号付きゼロにはデメリットもありますが，IEEE754ではメリットの方が大きいと判断したようです．

[13] Solarisにはisinf関数がないようなので，SPARC上でプログラムが動かない場合はisinf関数の部分を削除してください．

符号付きゼロを含むプログラム

```
/*
  符号付きゼロのテスト
 */
#include <stdio.h>

int main(void)
{
  double a, b, c, d;
  a=1.0/0.0; b=-1.0/0.0;        /* inf の生成 */
  printf("a=%lf b=%lf \n", a, b);
  c=1.0/a; d=1.0/b;           /* 符号付きゼロの生成 */
  printf("c=%lf d=%lf \n", c, d);
  printf("ans1=%lf, ans2=%lf\n", 1.0/c, 1.0/d);

  return 0;
}
```

計算結果

```
a=inf b=-inf
c=0.000000 d=-0.000000
ans1=inf, ans2=-inf
```

5.4　非正規化数

IEEE754 では正規化された最小数より小さい数を表現するために，仮数部の最初の桁を 1 とするのをやめ，そこを 0 としてもいいことになっています．この数のことを非正規化数 (denormalized number) といいます．また，非正規化数の範囲に数が入ることを段階的アンダーフロー (gradual underflow，graceful underflow) といいます．図 4 は，非正規化数の様子を表しています．

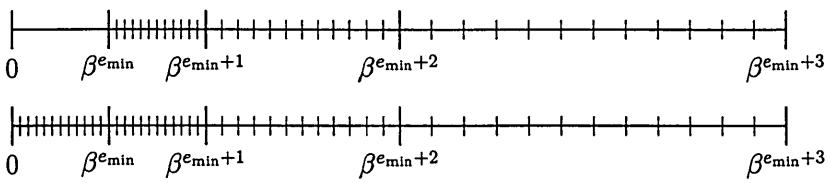


図 4: ゼロへの切り捨てと段階的アンダーフローの比較

正規化数のままでは，$\beta^{e_{min}}$ より小さい正の数は，0 へ切り捨てられてしまいますが，非正規化数を導入すると有効桁数は減りますが，更に小さい数を表示することができます．

6　丸めモード

IEEE754 では，4 つの丸めモードを選択できるようになっています．この点は，IEEE754 がそれ以前の規格と大きく違うところです．

6.1 4つの丸めモード

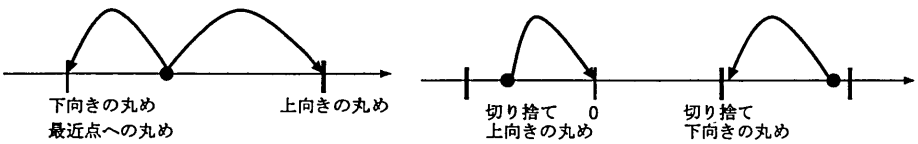
IEEE754 で定められている丸めモードは次の通りです．以下では，$x \in \mathbf{R}$ とします．

(1) 上向きの丸め (round upward)
x 以上の浮動小数点数の中で最も小さい数に丸めます．

(2) 下向きの丸め (round downward)
x 以下の浮動小数点数の中で最も大きい数に丸めます．

(3) 最近点への丸め (round to nearest)
x に最も近い浮動小数点数に丸めます．もしも，このような点が2点ある場合は，仮数部の最終ビットが偶数である浮動小数点数に丸めます．これを偶数への丸め (round to even) といいます．

(4) 切り捨て (round toward 0)
絶対値が x 以下の浮動小数点の中で最も x に近いものに丸めます．



丸めモード指定の応用例としては区間演算や連立一次方程式の高速精度保証などが挙げられます ([10]).

6.2 丸めの操作方法

ここでは，以下のようなプログラムを使って，実際に丸めモードの指定によりどのように結果が異なるかを見てみます．x86 および SPARC 用は文献 [10] のものを使い，Alpha 用はそれらと区間演算ライブラリ PROFIL([8]) を参考に作成しました[14].

[14]アーキテクチャの変更があった場合は，ここに挙げているレジスタの値を変える必要があります．その場合は，各 CPU のマニュアルを見るか，UNIX 系 OS の場合は，floatingpoint.h や fpu_control.h といったヘッダファイルを追跡してください．

x86用丸めモード変更プログラム

```
/* 丸めモード変更による計算 (x86 + Linux + gcc用) */
#include <stdio.h>

#define Near asm volatile("fldcw _RoundNear")
#define Up   asm volatile("fldcw _RoundUp")
#define Down asm volatile("fldcw _RoundDown")

int _RoundNear = 0x133a;
int _RoundUp   = 0x1b3a;
int _RoundDown = 0x173a;

int main(void)
{
  int i;
  double a;

  Down; /* 下向きの丸め */
  a=0.0;
  for(i=1;i<=1000000;i++) a+=0.1;
  printf("%30.20lf\n", a);

  Up; /* 上向きの丸め */
  a=0.0;
  for(i=1;i<=1000000;i++) a+=0.1;
  printf("%30.20lf\n", a);

  return 0;
}
```

SPARC用丸めモード変更プログラム

```
/* 丸めモード変更による計算 (SPARC + Solaris + gcc用) */
#include <stdio.h>

static int _RoundNear = 0x00000000L;
static int _RoundUp = 0x80000000L;
static int _RoundDown = 0xC0000000L;

#define Near()  asm volatile("ld %0,%%fsr" : : "g" (_RoundNear))
#define Up()    asm volatile("ld %0,%%fsr" : : "g" (_RoundUp))
#define Down()  asm volatile("ld %0,%%fsr" : : "g" (_RoundDown))

int main(void)
{
  int i;
  double a;

  Down(); /* 下向きの丸め */
  a=0.0;
  for(i=1;i<=1000000;i++) a+=0.1;
  printf("%30.20lf\n", a);

  Up(); /* 上向きの丸め */
  a=0.0;
  for(i=1;i<=1000000;i++) a+=0.1;
  printf("%30.20lf\n", a);

  return 0;
}
```

Alpha用丸めモード変更プログラム

```
/*
   丸めモード変更による計算(Alpha + Linux + gcc用)
*/
#include <stdio.h>

#define RND_MASK 0x0C00000000000000
#define RND_DOWN 0x0400000000000000
#define RND_NEAR 0x0800000000000000
#define RND_UP   0x0C00000000000000

void Near(void){
  register long int fpcontrol;
  asm volatile ("mf_fpcr %0" : "=f" (fpcontrol));
  fpcontrol &= ~RND_MASK;
  fpcontrol |= RND_NEAR;
  asm volatile ("mt_fpcr %0" : : "f" (fpcontrol));
}

void Up(void){
  register long int fpcontrol;
  asm volatile ("mf_fpcr %0" : "=f" (fpcontrol));
  fpcontrol &= ~RND_MASK;
  fpcontrol |= RND_UP;
  asm volatile ("mt_fpcr %0" : : "f" (fpcontrol));
}

void Down(void){
  register long int fpcontrol;
  asm volatile ("mf_fpcr %0" : "=f" (fpcontrol));
  fpcontrol &= ~RND_MASK;
  fpcontrol |= RND_DOWN;
  asm volatile ("mt_fpcr %0" : : "f" (fpcontrol));
}

int main(void)
{
  int i;
  double a;

  Down();
  a=0.0;
  for(i=1;i<=1000000;i++) a+=0.1;
  printf("%30.20lf\n", a);

  Up();
  a=0.0;
  for(i=1;i<=1000000;i++) a+=0.1;
  printf("%30.20lf\n", a);

  return 0;
}
```

これらのプログラムの実行結果はすべて以下のようになります.

丸めモード変更プログラムの実行結果

```
    99999.99999613071850035340
   100000.00000432481465395540
```

この結果より，丸め方向を上向きと下向きに指定して計算を行うことにより，正しい答え100000を包み込むことができることが分かります.

なお，x86およびSPARC上ではgccを使って上記のプログラムをコンパイルする場合，オプションは何も必要ありませんが，Alphaマシンで丸めモードを切替える場合は，-mfp-rounding-mode オプションを指定する必要があります．例えば，プログラム名が rnd_alpha.c だとすると，

```
gcc -mfp-rounding-mode=d rnd_alpha.c
```

とする必要があります[15]．

6.3 Double rounding(2重丸め)

拡張フォーマットを使うと丸め操作が2回入る可能性があります[16]．このように2回丸め操作が行われることをdouble rounding(2重丸め)と呼びます．例えば，同じ倍精度計算でもシステムが内部で拡張フォーマットを使用している場合，直接，倍精度に結果を丸めた場合とは計算結果が異なる場合があります．その例を以下で見てみましょう．

Double roundingの影響を見るプログラム

```
#include <stdio.h>
int main(void)
{
  double a, b, c, d;
  a=4600000000000006.0;
  b=1.5;
  c = a + b; c+= b;      /* 2つに分けて足すと切り上げは2回 */
  d = a + b + b;         /* 切り上げは最後に1度だけ */
  printf("c=%lf\n", c);
  printf("d=%lf\n", d);
  printf("d-c=%lf\n", d - c);
  return 0;
}
```

このプログラムの実行結果は次のようになります，なお，ここで，使用したaの値は，2^{52}から2^{53}の間であれば何でも構いません[17]．

Pentium4の結果

```
c=4600000000000010.000000
d=4600000000000009.000000
d-c=-1.000000
```

Pentium4では，なぜ，このような結果になるのでしょうか? それは，Pentium4では，まず拡張倍精度で計算をして，その後，倍精度に丸めているからです．c=a+b+bでは，a+b+bが拡張倍精度で計算され，その結果を倍精度に丸めます．これにより，正しい答えであるd=4600000000000009.000000

[15] Alphaマシンでは，コンパイル毎にオプションを指定すれば丸めモードを切り替えることができます．mは下への丸め，cがゼロへの丸めです．dは動的丸めモードで，浮動小数点制御レジスタを変更しなければ，上への丸めとなり，この例のように変更すれば，レジスタの値に応じて丸めモードが変更されます．

[16] 実数から拡張フォーマットで1回，拡張フォーマットから基本フォーマットで1回なので，合計2回です．

[17] モデル(4)および倍精度では$u = 2^{-53}$であることを考えれば，実行結果がより理解しやすくなると思います．

を得ることができます．一方，c=a+b;c+=bでは，まず，a+bが拡張倍精度で計算され，その値が倍精度に丸められます．このときに，切り上げが起こり，c=4600000000000008.000000となります．次に，c+bが拡張倍精度で計算され，その値が倍精度に丸められます．このときも，先程と同様に切り上げが起こり，c=4600000000000010.000000となります．

では，同じ計算をSPARCでするとどうなるでしょうか?

UltarSPARC-IIの結果

```
c=4600000000000010.000000
d=4600000000000010.000000
d-c=0.000000
```

また，Alphaで計算してもSPARCと同じ結果になります．この結果より，SPARCとAlphaは今のところ倍精度で宣言した演算は内部でも倍精度で行われていると考えられます．

IEEE754を満たしているコンピュータ上で計算した結果は，どのコンピュータ上でも一致するようにするのが，IEEE754の目的でしたが，この例ではそれが崩れています．このようなことが起こらないようにするためにも，IEEE754では拡張フォーマットを推奨していると思われます．

ちなみに，上記の計算を拡張倍精度で計算すると同じ結果になります．以下がそのプログラムです．ただ単に，倍精度のところを拡張倍精度にしただけです[18]．

倍精度拡張による計算

```
#include <stdio.h>
int main(void)
{
  long double a, b, c, d;  /* 拡張倍精度を使う */
  a=4600000000000006.0;
  b=1.5;
  c = a + b; c+= b;      /* 2つに分けて足すと切り上げは2回 */
  d = a + b + b;         /* 切り上げは最後に1度だけ */
  printf("c=%Lf\n", c);
  printf("d=%Lf\n", d);
  printf("d-c=%Lf\n", d - c);
  return 0;
}
```

拡張倍精度の結果 (Pentium4, SPARC)

```
c=4600000000000009.000000
d=4600000000000009.000000
d-c=0.000000
```

なお，Alphaでは拡張フォーマットには対応していないらしく倍精度と同じ結果となります．ちなみに，Compaq C Compiler V6.2を利用して上記のプログラムをコンパイルすると，

```
type long double has the same representation as type double on this platform
```

[18]マニュアルによるとPentiumの拡張倍精度フォーマットは80ビット，SPARCの拡張倍精度フォーマットは128ビットなので，拡張倍精度計算の正確さという点では，SPARCの方が有利だと言えます．

と表示されます.

7　コンパイラと最適化

IEEE754では，プログラム言語からその機能を利用できるかどうかについては何も保証されていません．したがって，IEEE754をサポートしているコンピュータ上でも開発元が異なるFortran, C言語などを使った場合，計算した結果が異なるということは珍しくありません．例えば，IEEE754では拡張フォーマットが推奨されていますが，使用する言語が拡張フォーマットをサポートしていなければ利用することはできません．

また，多くのコンパイラは最適化を行うと演算順序を変更してしまいます．例えば，$(a+b)+c=a+(b+c)=a+b+c$としてしまいます．これは，近似解を高速に求める場合には有効ですが，浮動小数点独特の考えを利用したアルゴリズムには有害となります．

ここでは，最適化による演算結果の違いをみるために，単純な和を求める方法 (recursive summation) と高精度な和を求める埋め合わせ法 (compensated summation[19]) を考えます．ここで，埋め合わせ法[20]の基本的なアイディアは演算により失われた情報を後で加えて補正するというものです．

コンパイラの影響を見るプログラム例

```
/*   0.1をN回掛けるプログラム */
#include <stdio.h>

#define N 1000000

int main(void)
{
  double a, b, c, d, tmp;
  int i;

  /* Recursive summation */
  a=0.0;
  for (i=1;i<=N;i++)     a+=0.1;
  printf("%30.20lf \n", a);

  /* Compensated summation*/
  b=0.0; c=0.0;
  for (i=1;i<=N;i++){
    tmp = b;
    d = 0.1 + c;                  /* コンパイラの影響を見るため， */
    b = tmp + d;                  /* c=(tmp-b)+dとせず，このように*/
    c = (tmp - (tmp + d) ) + d; /* c=(tmp-(tmp+d))+dとしている*/
  }
  printf("%30.20lf \n", b);

  return 0;
}
```

[19]Kahan summation formulaと呼ぶ場合もあります ([3]).

[20]Compensated summationという一般的な日本語訳が調べても分からなかったので，ここでは埋め合わせ法と呼ぶことにします.

Pentium4+gcc(オプションなし) の結果

```
100000.00000133288267534226
100000.00000133288267534226
```

SPARC+gcc(オプションなし) の結果

```
100000.00000133288267534226
100000.00000000000000000000
```

Pentium4+gcc(-O2 オプション) の結果

```
100000.00000000087311491370
100000.00000000000000000000
```

SPARC+gcc(-O2 オプション) の結果

```
100000.00000133288267534226
100000.00000000000000000000
```

なお，Alpha と SPARC は同じ結果になりました．これは，Alpha と SPARC は double rounding の影響を受けていないためだと思われます．

この例で分かるように最適化オプションをつけると演算結果が異なります．恐らく演算順序を変更しているのでしょう．これは実数で成り立つ数学の規則を浮動小数点数にも適用するというかなり無謀ともいえることをしている結果です．また，double rounding の影響により同じオプションをつけた場合でも計算結果は異なります．従って，数値結果を他人に見せる場合には，自分がどのようなコンピュータ上で，どのようなコンパイラを，どのようなオプションと共に，利用したのかを明記するべきでしょう．

なお，この例における前進誤差は，式 (7) において $\gamma_n = 1.1102 \times 10^{-10}, \sum_{i=1}^{n} |x_i| = 100000$ なので，$|s_n - \hat{s}_n| \leq 0.0000111022$ と評価するとができます[21]．上記の数値結果からは $|s_n - \hat{s}_n| \approx 0.000001333$ といえるので，評価 (7) は最悪の誤差限界を表していることが分かります．

8 IEEE754 で未定義な値の計算

ここではIEEE754で定義されていない数をいくつか計算してみることにしましょう．IEEE754では，基本的な数学関数やベキ乗の計算については定義されていないので，以下のような数はどのような結果になるのか，IEEE754 からだけでは分かりません．

IEEE754 で定められていない数

(1)0^0	(2)1^0	(3)1^∞	(4)1^{NaN}	(5)2^∞
(6)NaN^0	(7)log(0)	(8)log$(-\infty)$	(9)exp(∞)	(10)exp$(-\infty)$

これらを計算するために次のようなプログラムを用意しました．

[21] より正確には定理 1 より x_i を $x_i(1+\delta)$ として計算する必要があります．

IEEE754で未定義な数の計算

```
/*
  IEEE754で定義されていない数を計算
*/
#include<stdio.h>
#include<math.h>

int main(void)
{
  double a[10], tmp1, tmp2;
  int i;

  tmp1=pow(2.0,3000.0); tmp2=sqrt(-1.0);  /* tmp1=Inf, tmp2=NaN */

  a[0]=pow(0.0,0.0) ; a[1]=pow(1.0,0.0) ;
  a[2]=pow(1.0,tmp1); a[3]=pow(1.0,tmp2);
  a[4]=pow(2.0,tmp1); a[5]=pow(tmp2,0.0);
  a[6]=log(0.0)     ; a[7]=log(-tmp1);
  a[8]=exp(tmp1)    ; a[9]=exp(-tmp1);

  for(i=0;i<10;i++) printf("(%d)=%lf \n",i+1,a[i]);

  return 0;
}
```

このプログラムの実行結果を各CPU別にまとめると次のようになります[22].

問題	Pentium4	SPARC	Alpha
(1)	1.000000	1.000000	1.000000
(2)	1.000000	1.000000	1.000000
(3)	nan	NaN	nan
(4)	nan	NaN	nan
(5)	inf	Inf	inf
(6)	1.000000	NaN	1.000000
(7)	-inf	-Inf	-inf
(8)	nan	NaN	nan
(9)	inf	Inf	inf
(10)	0.000000	0.000000	0.000000

IEEE754で定義されていない数を計算しましたが，(6)を除いてすべて一致しました．どうしてこのような結果になるのかIEEE754に基づいて考察してみましょう．

まず，IEEE754では1度NaNと評価されたら，それ以降はすべてNaNとして評価されるので，式にNaNを含んでいる(4)はNaNとなるのが自然です．また，$\log(-\infty)$という実数は存在しないので，(8)もNaNとなるのが自然です．

IEEE754の基本的な精神は，数学における極限操作が成り立つようにするということなので，(5)(7)(9)(10)がそれぞれ，inf, -inf, inf, 0をとるというのも自然です．また，(2)が1になるのも問題はないでしょう．

次に，(3)を検討してみます．まず，

$$1^{\infty} = \lim_{x \to 0} 1^{\frac{1}{x}} = \lim_{x \to 0} (1-x)^{\frac{1}{x}}$$

[22]画面に表示された通りに書いていますので，大文字と小文字が混在しています．

が，極限の性質より成り立ちます．$\lim_{x\to 0} 1^{\frac{1}{x}} = 1$が成り立つのはいいでしょう．一方，$\lim_{x\to 0}(1-x)^{\frac{1}{x}} = \frac{1}{e}$となります．よって，極限は存在しないことになるので，(3) がNaNと表示されるのは，数学的には正しいことになります．

後は，(1) と (6) ですが，(1) は1にしておくと都合がいいことが多いので各CPUでそのように値を設定していると考えられます．例えば，$p(x) = \sum_{i=1}^{n} a_i x^i$とするとき，$p(0) = a_0 = a_0 \times 0^0$であり，$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$において，$x = 0, y = 1$とすると，$1 = 0^0$となります．

最後に (6) についてですが，SPARCはNaNが1度出たのだからNaNと評価するべきであると考えているのでしょう．私もこれは自然だと思います．それに対して，PentiumとAlphaは，すべての数に対する0乗の値は1で返した方がいいと考えているようです[23]．

9 最後に

本稿では，IEEE754とそれによる基本的な計算結果について解説しました．

IEEE754では，あいまいにしているところやそのハードウェアやソフトウェアへの実装の点で不十分なところがあり，今のところ目標は達成されていないといえるでしょう．そのため，IEEE754に対する批判もあるのは事実です ([3],[4],[7])．例えば，次のようなことが挙げられます．

1. 1970年代の末からIEEE754は検討されてきたが，それ以前の浮動小数点システムと比べて進歩しているとは言いがたい．規格を決めることのみに力を注ぎすぎ，浮動小数点表現にとって望ましい仕様についての研究が十分ではない．

2. 非数 (NaN) の具体的なビットパターンが規定されていない．そのため，互換性を損なう恐れがある．

3. 4種類の丸めモードを導入したのは進歩だが，ハードウェアの負担が大きい．最近点への丸めだけで十分である．

4. 本当に符号付きゼロが必要なのか?

5. 正規化数に徹するべきである．非正規化数の導入により正規化数か非正規化数かの判定が必要となる．

結局，1970年代までにいろいろな小型計算機用の浮動小数点方式が乱立したため，とにかく規格を決めてしまおうと考えたことに問題があるようです．また，メーカにとっては，その規格が優れたものであろうがなかろうが，共通の規格を採用すれば開発コストの軽減につながるのでIEEE754は魅力的だったのでしょう．

とはいえ，現在のほとんどのコンピュータがIEEE754を採用していることを考えるとIEEE754を使用することを前提とした計算法を考えるのも必要なことなのかもしれません．

割と数値計算を行っているときには，浮動小数点規格のことを見落としがちです．どうして，floating exceptionやNaNが表示されるのかが分かるだけでもプログラムのデバッグが楽にな

[23] ちなみに∞^0も1と表示されます．

ります．規格にも目を通さず，プログラミングに没頭するのも大事ですが，1度は自分が使用している計算機環境や浮動小数点規格を調べることをお勧めします．

参考文献

[1] W. J. Cody: Analysis of Proposals for the Floating-Point Standard, COMPUTER, 14(3), 63-69, 1981.

[2] W. J. Cody: Floating-point standards - Theory and Practice -, In Reliability in Computing, The Role of Interval Methods in Scientific Computing, R. E. Moore(Eds.), Academic Press, 99-107, 1988.

[3] D. Goldberg: What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys, 23(1), 5-48, 1991.

[4] H. Hamada: A new real number representation and its operation, In Proceedings of the Eighth Symposium on Computer Arithmetic, Como, Italy, M.J. Irwin and R. Stefanelli(eds.), IEEE Computer Society, Washington, DC, 153-157, 1987.

[5] N.J. Higham: Accuracy and Stability of Numerical Algorithms, SIAM, 1996.

[6] IEEE Standard for binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Institute of Electrical and Electronics Engineers, New York, 1985. Reprinted in SIGPLAN Notices 22(2), 9-25, 1987.

[7] S. Matsui and M. Iri: An overflow/underflow-free floating-point representation of numbers, J. Inform. Process., 4(3), 123-133, 1981.

[8] O. Knüppel: PROFIL/BIAS - A fast interval library, Computing 53, 277-288, 1994.

[9] J.H. Wilkinson: Rounding Errors in Algebraic Processes, Prentice-Hall, 1963.

[10] 大石 進一: 精度保証付き数値計算，コロナ社，2000年.

[11] 皆本 晃弥: UNIXユーザのためのトラブル解決Q&A，サイエンス社，2000年.

[12] IA-32 インテル(R) アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル，上巻：基本アーキテクチャ

http://www.intel.co.jp/jp/developer/design/pentium4/manuals/