平成 11 年度

卒業論文

Java 言語による精度保証付き数値計算

Numerical Analysis with Guaranteed Accuracy using Java Language

平成 12 年 2 月 4 日

指導教授： 柏木 雅英 助教授

早稲田大学理工学部情報学科

学籍番号 ： G96P045-0

小泉 健

目 次

1 序論 **4**

1.1 背景 5

1.2 本論文の目的 5

1.3 本論文の構成 5

2 区間解析 **7**

2.1 はじめに 8

2.2 丸め誤差 8

2.3 区間演算の基本的概念 8

2.4 Krawczyk の方法 10

3 Java 言語で書くにあたっての利点と問題点 **14**

3.1 はじめに 15

3.2 Java 言語で書くにあたっての利点と問題点 15

3.2.1 利点 15

3.2.2 問題点 17

3.3 問題点の解決法 18

3.3.1 演算子のオーバーロードの解決法 18

3.3.2 丸めの指定の解決法 19

3.3.3 速度の点の解決法 19

4 ネイティブメソッドの区間クラス nInterval **21**

4.1 はじめに 22

4.2 ネイティブメソッド 22

4.2.1 ネイティブメソッドとは 22

4.2.2 ネイティブメソッド実装のための関数 22

4.3 ネイティブメソッドの作成 23

4.4 終わりに 29

5 クラス BigDecimal を用いた区間クラス **30**

5.1 はじめに 31

5.2 クラス BigDecimal 31

5.2.1 BigDecimal クラスの丸めモード 32

5.2.2 BigDecimal クラスの丸めの指定できるメソッド 32

5.3 BigDecimal を用いた区間を表す Interval クラス 33

5.4 おわりに 37

6 実行結果 **38**

6.1 はじめに 39

6.2 Krawczyk クラスの使用方法 39

6.3 実行結果 41

6.4 考察 43

6.5 終わりに 44

謝 辞 **45**

参考文献 **47**

表 目 次

2.1 倍精度浮動小数点数 8

3.1 C,C++から削除された機能 16

6.1 実行結果 42

第 1 章

序論

1.1 背景

数値計算において、計算を行うと同時にその結果の誤差評価をも同時に計算するような方法を総称して精度保証付き数値計算と呼び、近年急速な進歩を遂げている.

この精度保証付き数値計算のプログラムは大部分が、C、C++などのプラットフォーム依存言語で書かれている. しかしこのプラットフォーム依存言語を用いて作成したコードは、OS、CPU などのプラットフォームが変われば、プログラムに変更を施し、更に再コンパイルしなければならない. それに対し、Java という言語はソースレベル、バイナリレベルの双方においてプラットフォーム独立である. つまり Java 言語を用いて作成したコードは、どのプラットフォームに移植しても、変更を施す必要も、再コンパイルする必要もなくなるのである. 本論文ではこの Java 言語の特性を生かした、精度保証付き数値計算のプログラムを作成しようと思う.

1.2 本論文の目的

精度保証付き数値計算の実現において最も基本的かつ重要な技法に、区間演算が挙げられる. さらに区間演算を利用し、多変数の方程式の解の判定を行う一つの方法に Krawczyk 法と言う重要な方法がある.

本論文では Java 言語を用いて, 区間演算を用いた Krawczyk 法を行うプログラムを作成する.Java で区間演算を行うコードを作成するにあたって、利点はもちろんあるのだが、いくつかの問題点があるので、それについて解決していき、区間演算を用いた Krawczyk 法の Java コードの作成方法を解説していく.

1.3 本論文の構成

本論文の構成は以下の通りである.

第 2 章では, 本論文の準備として、丸め誤差、区間演算の基本的概念、Krawczyk 法につ

いて述べる.

第 3 章では,Java で精度保証のプログラムを書くにあたっての利点と問題点を考察し、問題点に関する解決法について述べる.

第 4 章では,Java で精度保証のプログラムを書く際に適用した、ネイティブメソッドについての説明、またそれを利用した区間を表すクラス nInterval の作成方法について述べる.

第 5 章では, 同様に今回のプログラムを書く際に利用した、Java クラスライブラリの BigDecimal クラスの説明、それを利用した区間を表すクラス Interval について述べる.

第 6 章では、本論文の総括として、今回作成したプログラムの実行結果、それについての考察, 今後の課題について述べる.

第 2 章

区間解析

2.1 はじめに

本章では準備として、丸め誤差、区間演算の基本的概念、Krawczyk 法について簡単に説明する.

2.2 丸め誤差

数値計算で用いられる数値 (C 言語でいうところの double、float) は、いわゆる浮動小数点形式で表される. この方式では数値を正規化することにより、符号部、指数部、仮数部の３つの部分で表す. 倍精度浮動小数点数 (double) は 64bit で表され、その内訳は以下のようになる.

表 2.1: 倍精度浮動小数点数

符号 (1bit)	指数部 (11bit)	仮数部 52bit

この方式を用いた場合、実数のうちで計算機で正確に表現できる数は高々2^{64} 個にすぎない (実際はもう少し少ない). よって例えば浮動小数点数を 2 つ加算した結果は、浮動小数点で表せるとは限らず、それに最も近い浮動小数点数で近似することになる. これによって発生する誤差を丸め誤差 という.

2.3 区間演算の基本的概念

区間演算は浮動小数点演算に置ける丸め誤差の把握を行うための基本的な手法である. 区間演算では実数値を [下限, 上限] という二つの浮動小数点数で挟まれた区間で表現する. つまりは

定義 **2.2.1** (区間)

区間演算において区間とは

$$[\underline{x}, \overline{x}] = \{x \in R | \underline{x} \leq x \leq \overline{x}\}, \underline{x} \leq \overline{x} \in R$$

と表される閉区間である.

また区間同士の加減乗除を「演算結果として有り得る集合を内包する」ように定義すると以下のようになる.

定義 **2.2.2** (二項演算)

区間 $X = [p, q], Y = [v, w]$ に対して, 二項演算 $* \in \{+, -, \cdot, /\}$ を,

$$X * Y = \{x * y | x \in X, y \in Y\}$$

で定義する. ただし, 除算 $* = /$ の場合には $0 \notin Y$ とする. 実際の区間演算は,

$$X + Y = [p + v, q + w]$$

$$X - Y = [p - w, q - v]$$

$$X \cdot Y = [a, b](a = min(pv, pw, qv, qw), b = max(pv, pw, qv, qw)$$

$$X/Y = [p, q] \cdot [1/w, 1/v]$$

のように表現できる.

区間演算では区間の両端を計算する際に、丸めの向きを「外向き」にしておくことにより, 丸め誤差の影響分を区間内に収めるという方法で計算を行う. すなわち, 区間の下限を計算するときには「切り捨て」、区間の上限を計算するときには「切り上げ」で計算を行う. こうすれば, 厳密計算に比べて多少区間幅が大きくなるもの、区間内に真の値を包含しながら計算するという区間演算の性質は損なわれない. つまりは浮動小数点演算を用いた区間演算を実現するためには確実に切り捨て (または切り上げ) で計算を行う必要がある.IEEE 754 Std. に従って設計された CPU では、加減乗除における丸めの発生の仕方を制御することができる. 普段は真の値に最も近い浮動小数点数に丸めるモードになっているが、これを

「常に切り上げ」や「常に切り捨て」を行うように変更することができる. これを用いて区間同士の加減乗除は、

1. 丸めモードを切り捨てに変更
2. 区間の下限を計算
3. 丸めモードを切り上げに変更
4. 区間の上限を計算
5. (必要ならば丸めモードを通常のモードに戻す.)

という手順で計算を行えば良い.

2.4 Krawczykの方法

まず準備として、不動点定理のうちで最も基本的な縮小写像原理 を示す.

定理 2.3.1 (縮小写像の原理)

X を $d(x,y)$ を距離とする完備距離空間とし,$g : X \to X$ を全ての $x, y \in X$ に対して

$$d(g(x), g(y)) \leq \alpha d(x,y) \tag{2.1}$$

を満たすような写像とする. ただし、$\alpha < 1$ とする.(このような g を縮小写像と言い、実数 α を縮小定数と言う.) このとき、以下が成立する：

1. $g(x) = x$ を満たす点 x^*(不動点と言う) が X に唯一存在する.
2. x_0 を X 内の任意の点とし、点列 $\{x_k\}$ を

$$x_{k+1} = g(x_k) \tag{2.2}$$

で定めると、x_k は x^* に収束し, その収束速度は

$$d(x_k, x^*) \leq \frac{\alpha^k}{1-\alpha} d(x_1, x_0) \tag{2.3}$$

で表される.

3. X 内の任意の点 x_0 に対して、

$$d(x_0, x^*) \leq \frac{d(g(x_0), x_0)}{1-\alpha} \tag{2.4}$$

□

上の定理は一般的な形で書いてあるが、本稿では n 次元ユークリッド空間 R^n を扱っているので、

X : R^n内の閉集合

$d(x,y)$: $\|x-y\|$.ただし ノルムは $\|x\| = max|x_i|$(最大値 ノルム) で定める.

のように読み替えれば良い.

非線形方程式

$$f(x) = 0, f : R^n \to R^n \tag{2.5}$$

の解の存在を保証するために、これを同値な不動点形式に変換し、不動点定理を適用することを考える. 単純に

$$x = x + f(x) \tag{2.6}$$

のようにしても良いが、これでは右辺の縮小性が成立するかどうかは f によることになってしまう. 近似解の近辺での f の傾きを近似する行列とし、これを用いて f の傾きを補正して、

$$x = x - L^{-1} f(x) \tag{2.7}$$

のようにする ($f(x) = 0$ を $-L^{-1}f(x) = 0$ だったと考える). 以下この右辺を $g(x)$ とおく：

$$g(x) = x - L^{-1} f(x) \tag{2.8}$$

L が逆行列をもつならば x が $f(x) = 0$ を満たすことと x が $x = g(x)$ を満たすことは同値なので, 以下 $x = g(x)$ の不動点の存在を示すことを考えれば良い. 以下, 区間 $I \subset R^n$ とし,I に g の不動点が存在することを示すことを考える. 示すべきことは、

1. $\{g(x)|x \in I\} \subset I$($g$ が I から I への写像であること)

2. g が I 上で縮小写像であること.

の 2 つである.

前者は, 区間演算を用いて

$$G(I) = I - L^{-1}F(I) \tag{2.9}$$

のように g に対する区間演算を用い、$G(I) \subset I$ を確認すれば容易に確認できるように見える.しかし, 区間演算の減算の定義の仕方では、$c = a - b$ ならば c の区間幅は a の区間幅と b の区間幅の和となることを考えれば、決して $G(I) \subset I$ は成立しない! 写像 $g(x) = x - L^{-1}f(x)$ は, 傾きを 0 に近づけるためによく似た傾きの関数の減算を行なっている訳で、これは区間演算の苦手な計算の形そのものである. よって、区間演算の過大評価を抑えるための何らかの手法を併用する必要がある.Krawczyk の方法では、平均値形式を用いて、c を I の中心として

$$\begin{aligned} K(I) &= g(c) + G'(I)(I - c) \\ &= c - L^{-1}f(c) + (E - L^{-1}F'(I))(I - c) \end{aligned} \tag{2.10}$$

(E は単位行列) のような **Krawczyk** 写像 K について、

$$K(I) \subset I \tag{2.11}$$

を確認する.

後者の条件については, 次のように確認する. 平均値の定理により,$x, y \in I$ ならば

$$\|g(x) - g(y)\| \leq \max_{x \in I} \|g'(x)\| \|x - y\| \tag{2.12}$$

が成立する. ここで行列 $g'(x)$ のノルムは、

$$\|A\| = \sup_{\|x\|=1} \|Ax\| \tag{2.13}$$

となるように定義されたもので、R^n のノルムが $\max|x_i|$(最大値ノルム) ならば

$$\|A\| = \max_i \sum_j |A_{ij}| \tag{2.14}$$

で計算できる. すなわち、区間行列のノルムを

$$\|A\| = \max_i \sum_j \max_{x \in A_{ij}} |x| \tag{2.15}$$

で定義し、

$$\|G'(I)\| = \|E - L^{-1}F'(I)\| < 1 \tag{2.16}$$

が成立すれば、g は I 上で縮小写像であることが言える.

以上をまとめると、次のようになる：

定理 2.3.2 (Krawczyk の方法)

$f : R^n \to R^n, I \in IR^n, c$ を I の中心、$L \simeq f'(c)$ を正則な行列,F' を f の区間包囲とする.

$$K(I) = c - L^{-1}f(c) + (E - L^{-1}F'(I))(I - c) \tag{2.17}$$

としたとき、

$$K(I) \subset I \tag{2.18}$$

$$\|E - L^{-1}F'(I)\| < 1 \tag{2.19}$$

が成立するならば、I に解が唯一存在することが保証される.

第 3 章

Java言語で書くにあたっての利点と問題点

3.1 はじめに

本論文では Java 言語により上述の区間演算を用いた Krawczyk 法のプログラム作成方法を説明する. そのために区間を以下のような Interval クラスを用いて表す.

```
 public class Interval
    type min,max;
//*******コンストラクタ 1 ***********
    Interval(type a,type b){
           this.min=a;
           this.max=b;
    }
//*******コンストラクタ 2 ***********
    Interval(type a){
           this.min=a;
           this.max=a;
    }
```

この区間を表す Interval クラスを用いて Krawczyk 法を行う.

3.2 Java 言語で書くにあたっての利点と問題点

3.2.1 利点

Java で精度保証プログラムを書くときの利点は

1. **Java** 言語特有の移植性
2. シンプルなプログラムが書ける事
3. **Java** がオブジェクト 指向言語である事

の 3 点が挙げられる.

まず 1 について考えると、前述のように Java 言語はソースレベル、バイナリレベルの双方においてプラットフォーム独立である. ソースレベルで見ると,Java の基本データ型は, 全ての開発プラットフォームにおいて一貫した大きさを持っている. また, 全ての Java 環境に共通した Java の基本クラスライブラリを使う事で, あるプラットフォームで作成したコードを移植先のプラットフォームで動くよう書き直す必要も無くなる. さらに Java はバイナリレベルでもプラットフォーム独立であり, ソースコードを再コンパイルする必要も無く, 複数のプラットフォームで実行できる.

2 についてもこれも Java の利点であるが、Java はとてもシンプルにできていて、C++を手本にして書かれたのだが、たちの悪いバグの元凶となるポインタや, 明示的なメモリの管理(malloc、free)などを仕様からはずしている. ここで表 1 に C,C++から削除された機能をあげておく.

表 3.1: C,C++から削除された機能

削除された機能	代用方法
#define *	定数仕様 (final 指定)
typedef *	クラス宣言
構造体と共用体	クラスのインスタンス変数
多重継承機能	インターフェイス
演算子のオーバーロード	
自動強制型変換	キャストを明示的に仕様
ポインタ構造	配列と文字列をオブジェクトとして用意
プログラマのメモリ管理	Java のガーベッジコクレクタが行う

(*) プリプロセッサを必要としない. したがってヘッダファイルも必要無い.Java 言語コンパイラでは, クラス定義がタイプ情報を全てリンク時に保持するバイナリ形式でコンパイル

される.

Java では上記のように様々な機能は削除されたが、豊富なクラスライブラリにより、C などほかの言語でかける作業や問題解決は全て行う事ができるのである.

3 については,Java はオブジェクト指向の言語として設計されている.Java 言語を, オブジェクト指向言語と見たとき,smalltalk の持つオブジェクト指向言語としての純粋さと,C++言語の持つ実行効率を重視した設計のほぼ中間に位置する. 基本的なデータタイプを除き,Java 言語のあらゆる要素がオブジェクトとして設計されている.

つまりは Java はオブジェクト指向言語であるために必要な 3 つの特性

- カプセル化 (Encappsulation)
- 継承 (Inheritance)
- ポリモーフィズム (polymorphism)

をサポートしている. これによりオブジェクト指向言語の特性を活かしたプログラムを作成することができる.

3.2.2 問題点

Java 言語で精度保証のプログラムに書くにあたって少し問題になる点がある. それは以下の 3 点である.

1. **Java** には「 演算子のオーバーロード 」がない事
2. **Java** では丸めの指定をすることができない事
3. インタプリタで実行するため **C** に比べて速度が遅い事

1 での問題は、区間をクラスや構造体を用いて表し、区間と区間、区間と数値を演算したいときに問題が生じてくる. 例をあげると, 区間を Interval クラスで表し、演算子のオーバーロードを用いて作った次の C のソースを見てもらいたい.

```
Interval a(1,2),b(2,2),c;
  c=a+b; c=2*a;
```

このように演算子のオーバーロードを用いるとあたかも区間と区間を演算しているようなソースを無理なく書くことができる. しかし Java にはこの機能がない. (演算子のオーバーロードにより演算子の本来の意味とは違う意味を持たす事が出来てしまうからしい.) 私はこの問題に対してはとりあえずは関数のオーバーロードを使うことで補おうと思う.

2 について考えると、丸めの制御が出来ない事は区間演算のプログラムを書くにあたっては致命的である. この問題に関しては「ネイティブメソッド」と java.math パッケージの「BigDecimal クラス」を用いてこの問題を解決し、Java でのコードを作りたいと思う.

3 については、どのプログラムについてもいえる事だが, たとえどんなに正確な結果を導くプログラムであろうが, 遅ければ何にもならない. これについても解決法を考察する.

3.3 問題点の解決法

3.3.1 演算子のオーバーロードの解決法

演算子のオーバーロードに関しては上述のように、関数のオーバーロードを用いることで補う.Java は関数のオーバーロード機能があるので、区間 + 区間、区間 + 数値、などもひとつの関数 (例えば Add) で表すことができる. 上のクラス Interval を用いて先ほどの演算は

```
Interval a(1,2),b(2,2),c;
  c=c.Add(a,b); c=c.Mul(2,a);
```

のように書けるようにする.

3.3.2 丸めの指定の解決法

区間演算を行うときに丸めの指定ができないことは, 区間の正当性を示すことができないので致命的である. 今回は区間を Interval というクラスで表す. このクラスの演算で丸めを指定できるようにする.

ネイティブメソッド

Java ライブラリでは提供されていないハードウェアや OS の機能を利用したいときには、ネイティブメソッドを用いて解決する事ができる. 今回はこの機能を用いて丸めの指定のできる C のプログラムを Java に組み込んで, この問題を解決して行く.

クラス BigDecimal

ここでは詳しい説明は省くが,Java クラスパッケージの BigDecimal クラスは数値を

$$unscaled\ Val \div 10^{scale}$$

で表す事のできるクラスである. 本来丸め誤差とは, 小数部を削除する事によって発生する問題なので, ここでは BigDecimal の小数部の桁数 (Scale) を増やしたまま計算する事により, 丸め誤差を減らし, 区間の正当性を計る.

3.3.3 速度の点の解決法

現在のコンピュータの性能を考慮に入れるとそこまで、気にするほどではないが、もし速度が気になるなら解決策として

1. ネイティブコードを Java プログラムにリンクする
2. Java バイトコードをネイティブコードに変換する

の方法があるが、Java の最も優れている移植性が失われることになるので、なるべく用いないほうが好ましい. 今回のように Java の機能だけでは実現できないハードや OS の機能

を利用したいときなどやむをえない場合を除いて、ネイティブメソッドはなるべく使わないほうがよい.
もっと良い解決法として自分のプログラムのアルゴリズムをできるだけシンプルにしたり計算量を減らしたりすることで、速度の向上を計ることにする.

第 4 章

ネイティブメソッドの区間クラス

nInterval

4.1 はじめに

本章では Java の機能の一つであるネイティブメソッドについての紹介と具体的な作成方法について説明する. それに合わせて, 区間を表すクラス nInterval でのネイティブメソッドを用いた丸めの制御を行うプログラムの説明をしていく.

4.2 ネイティブメソッド

4.2.1 ネイティブメソッドとは

ネイティブメソッドとはネイティブコード (各 CPU に固有の機械語コード) で作られるメソッドである. 具体的には、C などの言語でメソッド本体を書いて、コンパイラで翻訳してネイティブコードにするわけである. このネイティブメソッドが必要な理由とは,

1. **Java** の機能だけでは実現できないシステム関数を使いたい
2. 速度を上げたい

の 2 点が挙げられる. しかし注意しなくてはならないのが, ネイティブメソッドを使う事により Java の特徴の一つであるコードの移植性が失われてしまうという事である.1 の場合のように, ネイティブメソッドを用いるのがやむを得ない場合を除いて,2 のためだけにコードの移植性が失われるのは問題である. 今回は,CPU の機能である丸めの指定を利用するためにこれを用いる.

4.2.2 ネイティブメソッド実装のための関数

ここでネイティブメソッドを C で実装する際に役立つ,Java の実行時データ構造にアクセスするためのマクロや関数が幾つかあるので, 簡単ではあるがその紹介をしておく.

```
Object *unhand(Handle *)
```

この関数は, オブジェクトのハンドルからそのデータ部のポインタを得る関数である. 返されるポインタの型は常に Object*ではなく, 渡したオブジェクトの型に応じて変わる. 今回もこの関数を利用して,C 言語から nInterval クラスのインスタンスへのポインタを得た.

さらに,C から Java クラスのコンストラクタと,static メソッド ,dynamic メソッドを呼びたい時に使用する関数を以下に示す.

```
HObject *execute_java_constructor(ExecEnv *e,char *classname,
                                  ClassClass *c,char *signature, ...)
long execute_java_static_method(ExecEnv *e,ClassClass *c,
                                  char *method_name,char *signature, ...)
long execute_java_dynamic_method(ExecEnv *e,HObject *obj,
                                  char *method_name,char *signature, ...)
```

ここでの最初の引数 e は実行時コンテキストである. ほとんどの場合, この引数の値には"0"を使用し,Java 実行時にシステムに現在の実行コンテキストを使用するよう指示する. コンストラクタと static メソッドを呼ぶ関数の引数 c はインスタンス化したいクラスの構造である. これは

```
ClassClass *FindClass(struct execenv *e,char *name,bool_t resolve)
```

で探したりもできるが,"0"で代用することもできる.
今回は上述の関数の中では, コンストラクタを呼ぶ関数を用いてプログラムを作成した.

4.3 ネイティブメソッドの作成

ここでネイティブメソッドの作成方法を,nInterval クラスと, それを利用して Krawczyk 法を行う Krawczyk クラスを例にとって紹介する.

Step1 Java コードを書いてコンパイル

- ネイティブメソッドを定義する

ネイティブメソッドにしたいメソッドには native 指定をして, 本体には何も書かないようにする.(後から C 言語で実装する.) 他のメソッドと同様, ネイティブメソッドは Java クラス内で定義する.nInterval クラスを例に取ると以下のようになる.

```
public class nInterval{
  double min,max;

    ..........

    public native nInterval add1(nInterval a,nInterval b);
    public native nInterval add2(double x,nInterval a);
    public native nInterval add3(nInterval a,double x);

    .....

}
```

今回は C 言語で実装するので関数のオーバーロードはできない. よって関数名は上記のように 3 つ必要となる.

- ライブラリをロードする

ネイティブメソッドを実装する C 言語コードは動的にロード可能なライブラリにコンパイルし, それを必要とする Java クラスにロードされなければならない.(この方法については後で説明する.)Java クラスにライブラリをロードすると, ネイティブメソッドの実装がクラスの定義にマップされる.

System クラスの loadLibrary() を利用して, 動的にライブラリをロードできるようにする. 今回は"nInterval"というライブラリをロードするので以下のようになる.

```
static {
   System.loadLibrary("nInterval");
}
```

この static initializer がクラスが実行系にロードされる時に一度だけ実行される事になる. このコードによってクラスとそれが必要とするネイティブコードが結合されるのである.

このクラスのコードを作成したら javac で通常と同様にコンパイルする.

Step2 .h ファイルを作成

前に作成した nInterval クラスに対しヘッダファイルを作成する.Unix では

```
% javah Java クラス名
```

と打つことにより,"Java クラス名.h"という名前をもつヘッダファイルができる. この C 言語ヘッダーファイルには,nInterval クラスを表す構造体, そのクラスに定義されているネイティブメソッドのシグニチャなどが入っている. これを利用して C 言語の実装を行う.

Step3 スタブファイルを作成

スタブファイルを作成するために javah の-stubs オプションを使用する. この場合も javah は Java クラスに対して実行する.

```
% javah -stubs Java クラス名
```

これにより"Java クラス名.c"というスタブファイルができる. このスタブファイルは Java クラスとそれに対応する C 言語構造体をつなぎ合せる接着剤のような役割を果たす.

Step4 C 言語でネイティブメソッドの実装を記述

javah で作成した nInterval.h の関数のシグニチャを利用して,C 言語でネイティブメソッドを実装するコードを作成する.nInterval.h に格納された nInterval 同士の加算 add1 のシグネチャは以下のようになっている.

```
extern struct HnInterval
  *nInterval_add1(struct HnInterval *,
                      struct HnInterval *,struct HnInterval *);
```

これを利用して関数 add1 の実装をしていく.
ここで add1 関数の引数が一つ増えていることに注意する. この増えた引数は C++ の"this"変数とみなすことができる. このメソッドのシグニチャを用いて add1 実装を行う. 今回は nIntervalNative.c ファイルでこのメソッドの実装を行うことにする. そのコードは以下のようになる.

```
#include <StubPreamble.h>
#include "nInterval.h"      // 作成したヘッダファイル

#define near() fpsetround(FP_RN)  //丸めを四捨五入モードへ
#define up()   fpsetround(FP_RP)  //丸めを上へのモードへ
#define down() fpsetround(FP_RM)  //丸めを下へのモードへ

    ..........

struct HnInterval
    *nInterval_add1(struct HnInterval *this,
                        struct HnInterval *a,
                              struct HnInterval *b){
   double h,k;
```

```
    HnInterval *temp;
     temp=(HnInterval *)execute_java_constructor(0,"nInterval",0,"()");
     down();  //下への丸めを指定
     h=unhand(a)->min+unhand(b)->min;
     up();    //上への丸めを指定
     k=unhand(a)->max+unhand(b)->max;
     near();  //丸めを四捨五入モードに戻す
     unhand(temp)->min=h; unhand(temp)->max=k;
     return temp;
}
     ..........
```

他の演算もこの様にヘッダファイルのメソッドのシグニチャを利用して実装する.
ここでは 2 つのファイルをインクルードしている. StubPeamble.h は Java 実行時, システムと情報を交換するための十分な情報を C 言語コードに提供するヘッダである. ネイティブメソッドを書く時は, このファイルを常に自分の C 言語ソースファイルにインクルードする必要がある. nInterval.h は **Step2** で作成した.h ファイルである. このファイルもインクルードする必要がある.

Step5 作成した **C** のコードとスタブファイルをコンパイル

gcc などを使って作成した C のコードとスタブファイルをコンパイルする. この時には **Step4** で,StubPreamble.h をインクルードできるようにするために,JDK(Java Development Kit) の includeディレクトリをコンパイラのオプションで指定するとともに,-fpic オプションで後で結合するために穴を開けておく. よって C のコードのコンパイルは

```
% gcc -fpic -c -I/usr/local/jdk1.1.7/include
```

```
        -I/usr/local/jdk1.1.7/include/freebsd nIntervalNative.c
```

スタブファイルのコンパイルは

```
% gcc -fpic -c -I/usr/local/jdk1.1.7/include
        -I/usr/local/jdk1.1.7/include/freebsd nInterval.c
```

となる.

Step6 用意したオブジェクトファイルを結合してダイナミックリンクライブラリへ

Step5 で作成した 2 つのオブジェクトファイルを結合して動的にロード可能なライブラリを作成する. このときに-shared オプションで 2 つのオブジェクトファイルを結合する.

```
% gcc -shared -o libnInterval.so nInterval.o nIntervalNative.o
```

これにより,"libnInterval.so"という動的にロードできるライブラリができる.

Step7 Java インタプリタで実行

最後に Java インタプリタを使用し Java アプリケーションを実行する. 今回は nInterval クラスを利用し Krawczyk 法を行う, Krawczyk クラスをコンパイルして実行する.
ところでシステムは"libnInterval.so"という名前のファイルをどこから探してくるのであろうか? Unix では System.loadLibrary() はシステムのダイナミックリンカを呼び出すが, ダイナミックリンカは環境変数 LD_LIBARARY_PATH に指定されてるディレクトリのリストをもとにライブラリファイルを探索する. ライブラリパスが設定されていない場合は以下のような例外割込みが表示される.

```
java.lang.NullPointerException
        at java.lang.Runtime.loadLibrary(Runtime.java)
        at java.lang.System.loadLibrary(System.java)
```

```
        at
        at Krawczyk.<init>(Krawczyk.java:23)
        at Krawczyk.main(Krawczyk.java:231)
java.lang.UnsatisfiedLinkError .....
```

この場合はライブラリパスを設定しなければならない.

```
% setenv LD_LIBRARY_PATH .:dir1:dir2: ...
```

のようにするとLD_LIBRARY_PATHが設定され,ダイナミックリンカがlibnInterval.soファイルを探し出す事ができるようになる.(「.」がカレントディレクトリ,dir1以下はそれ以外の探索すべきディレクトリを表す.)
これによりJava実行系が最初にクラスnIntervalをロードする時,static initializerの中でSystem.loadLibrary("nInterval")が実行され,ネイティブメソッドのロードが可能となり,ネイティブメソッドの実装が行われ,nIntervalクラスの実行ができるようになる.

4.4 終わりに

このように,ネイティブメソッドを使用することにより、Javaの機能だけでは実現できないことも、他のプラットフォーム依存言語をJavaにロードさせることで可能となる.しかしこのネイティブコードを使うと上述のように,特定のプラットフォームでしか動かせなくなる問題が発生する.そうなると今回の"どのプラットフォームでも使用できる"ということに反することになるが、残念ながら現段階では、このネイティブメソッドを用いて環境に合わせたネイティブコードをその環境の分だけ用意するしかないのである.

しかし、Java環境というのはまだ発展途上であり、現在でもJava環境に欠けているさまざまな機能を追加しようという動きもある.だから多くのユーザが必要に感じたら、将来Javaパッケージに"丸めの制御"を追加してもらえるかもしれない.

第 5 章

クラスBigDecimalを用いた区間クラス

5.1 はじめに

この章では Java.math.BigDecimal の紹介と、これを用いて作成した丸め誤差の影響を少なくしたクラス Interval の区間演算のプログラムの説明をする.

5.2 クラス BigDecimal

Public class BigDecimal

extends Number

impliments Comparable

このクラス"BigDecimal"は以下のような階層構造のもとにある.

java.lang.Object

|

+ − −java.lang.Number

|

+ − −**java.math.BigDecimal**

BigDecimal は、任意精度の「スケールなし整数値」と小数点以下の桁数を表す負でない 32 ビット整数「スケール」で構成される.BigDecimal で表される数値は

$$unscaled\ Value \div 10^{scale}$$

である.このクラスは、基本算術、スケール操作、比較、ハッシング、および書式変換の演算を提供するクラスである.

またこのクラスは、小数部を破棄できる演算 (divide と setScale) に対してユーザに明示的に丸め動作を指定させて、ユーザが丸め動作を完全に制御できるようにすることも提供している.このために 8 つの丸めモードが用意されている.

5.2.1 BigDecimalクラスの丸めモード

このクラスにはクラス変数として 8 つの丸めモードが用意されている. これを以下に示す.

- **public static final int ROUND_UP**

  0 から離れるように丸めるモード.

- **public static final int ROUND_DOWN**

  0 に近づくように丸めるモード.

- **public static final int ROUND_CEILING**

  正の無限大に近づくように丸めるモード. 正の場合はROUND_UP、負の場合はROUND_DOWN のように動作する.

- **public static final int ROUND_FLOOR**

  負の無限大に近づくように丸めるモード.ROUND_CEILING の逆の動作をする.

- **public static final int ROUND_HALF_UP**

  最も近い数字に丸めるモード. ただし両隣の数字が等距離の場合は切り上げる.

- **public static final int ROUND_HALF_DOWN**

  最も近い数字に丸めるモード. ただし両隣の数字が等距離の場合は切り捨てる.

- **public static final int ROUND_HALF_EVEN**

  最も近い数字に丸めるモード. ただし両隣の数字が等距離の場合は偶数に丸める.

- **public static final int ROUND_UNNECESSARY**

  要求される演算の結果が正確であり、丸めが必要でないことを表す丸めのモード.

5.2.2 BigDecimalクラスの丸めの指定できるメソッド

このクラスに用意されている丸めの指定のできる関数を以下に示す.

1. **public BigDecimal divide(BigDecimal val, int scale, int roundingMode)**
   値が (this/val) で、スケールが指定されたものである BigDecimal を返す. 丸めを行い指定したスケールで結果を生成する必要がある場合は、指定した丸めモードが適用される.

2. **public BigDecimal divide(BigDecimal val ,int roundingMode)**
   値が (this/val) で、スケールが this.scale() である BigDecimal を返す. 丸めを行い、特定のスケールで結果を生成する必要がある場合は、指定された丸めモードが適用される.

3. **public BigDecimal setScale(int scale, int roundingMode)**
   スケールが指定された値であり、かつスケールなしの値が、この BigDecimal のスケールなしの値と、総体値を維持できる適当な 10 の累乗の積または商により決定される BigDecimal を返す. スケールが演算で減らされる場合、スケールなしの値は割る必要があり、値が変わる可能性がある. この場合指定した丸めモードが除算に適用される.

5.3 BigDecimal を用いた区間を表す Interval クラス

BigDecimal を用いた区間を作成することにより, 丸め誤差を少なくする事ができる. 丸め誤差は小数部を破棄する事により発生するので,BigDecimal の Scale を多く取り, 小数部分を多くする事で, 区間の広がりを抑える. 以下に BigDecimal を用いた区間 Interval のコンストラクタのコードを示す.

```
import java.math.BigDecimal; //BigDecimal クラスを import


public class Interval{
  BigDecimal min,max;


   Interval(BigDecimal a,BigDecimal b){
```

```
        this.min=a;
        this.max=b;
    }
    Interval(BigDecimal a){
        this.min=a;
        this.max=a;
    }
    Interval(double a,double b){
        BigDecimal ba = new BigDecimal(a);
        BigDecimal bb = new BigDecimal(b);
        this.min=ba;
        this.max=bb;
    }
    Interval(double a){
        BigDecimal ba = new BigDecimal(a);
        this.min=ba;
        this.max=ba;
    }
    .......
}
```

この様に区間を 2 つの BigDecimal で表すクラス Interval を作成する.またこの区間の Scale を設定できるような関数を,5.2.2 の 3 の関数を用いて以下のように作成する.

```
    Interval setScale(int x){
        BigDecimal a = new BigDecimal(0.0);
        BigDecimal b = new BigDecimal(0.0);
        a=min.setScale(x,BigDecimal.ROUND_FLOOR);   //下へ丸める
```

```
        b=max.setScale(x,BigDecimal.ROUND_CEILING); //上へ丸める
        return(new Interval(a,b));
    }
```

当然だが、区間の小さい方の数値は下へ, 区間の大きい方の数値は上へ丸める事で解の正当性を保証する事ができる.
また、計算していくうちにBigDecimalのScaleが自動的に増え, 計算量が増加をもたらすことで, 結果を出すまでに時間が掛かってしまう事がある. これを防ぐために, 先程の区間のScaleを設定する関数を使ってそれぞれの演算でもScaleを設定できるようにしておく.Interval同士の加算を例に取りこれを示す.
ここでBigDecimalクラスのインスタンスメソッドであるaddを用いる. その仕様は

public BigDecimal add(BigDecimal val)

値が(this+val)でスケールがMAX(this.scale(),val.scale())のBigDecimalを返す

である. このメソッドを利用し、区間Interval同士の加算は以下のようになる.

```
//***********Scaleを設定しない関数************************
    Interval add(Interval a,Interval b){
         BigDecimal h = new BigDecimal(0.0);
         BigDecimal k = new BigDecimal(0.0);
         h=a.min.add(b.min);
         k=a.max.add(b.max);
         return(new Interval(h,k));
    }
```

```
//***********Scaleを val で設定する関数**********************
    Interval add(Interval a,Interval b,int val){
         BigDecimal h = new BigDecimal(0.0);
         BigDecimal k = new BigDecimal(0.0);
   // Scaleを設定
         a=a.setScale(val);
         b=b.setScale(val);
         h=a.min.add(b.min);
         k=a.max.add(b.max);
         return(new Interval(h,k));
    }
```

同様に減算, 乗算でも Scale を設定できるようにしておく. 除算では 5.2.2 の 2 に示す関数を用いる事で丸めモードを指定する. 更に同様に,5.2.2 の 1 の関数を用い,Scale の増加を防ぐために,Scale を自分で設定するような関数も作っておく. ここでの Min,Max の関数はそれぞれ引数の最小値, 最大値を返す関数である.

```
//*********Scaleを設定しない関数************************
 Interval div(Interval a,Interval b){
     BigDecimal h = new BigDecimal(0.0);
     BigDecimal k = new BigDecimal(0.0);
     h=Min(a.min.divide(b.min,BigDecimal.ROUND_FLOOR),
            a.min.divide(b.max,BigDecimal.ROUND_FLOOR),
             a.max.divide(b.min,BigDecimal.ROUND_FLOOR),
              a.max.divide(b.max,BigDecimal.ROUND_FLOOR));
     k=Max(a.min.divide(b.min,BigDecimal.ROUND_CEILING),
            a.min.divide(b.max,BigDecimal.ROUND_CEILING),
```

```
            a.max.divide(b.min,BigDecimal.ROUND_CEILING),
             a.max.divide(b.max,BigDecimal.ROUND_CEILING));
    return(new Interval(h,k));
}

//*********Scaleを引数Valによって設定する関数************
    Interval div(Interval a,Interval b,int val){
      BigDecimal h = new BigDecimal(0.0);
      BigDecimal k = new BigDecimal(0.0);
      h=Min(a.min.divide(b.min,val,BigDecimal.ROUND_FLOOR),
            a.min.divide(b.max,val,BigDecimal.ROUND_FLOOR),
             a.max.divide(b.min,val,BigDecimal.ROUND_FLOOR),
              a.max.divide(b.max,val,BigDecimal.ROUND_FLOOR));
      k=Max(a.min.divide(b.min,val,BigDecimal.ROUND_CEILING),
            a.min.divide(b.max,val,BigDecimal.ROUND_CEILING),
             a.max.divide(b.min,val,BigDecimal.ROUND_CEILING),
              a.max.divide(b.max,val,BigDecimal.ROUND_CEILING));                    ret
    }
```

このように,除算では丸めの指定をする事で区間の正当性を保証することができる.

5.4 おわりに

この様にBigDecimalクラスを用いると,小数点部を多く取ったまま計算するために,計算量が増えるといった欠点もでてくるが、Javaの仕様だけで区間演算を行う事ができ,Javaの利点も損なわれずにすむ.しかしこの方法は丸め誤差の影響を少なくするというだけで,丸めの制御を行っているわけではないので、区間の正当性という点では多少問題が残る.

第 6 章

実行結果

6.1 はじめに

この章では, ネイティブメソッドを用いたクラス nInterval と、BigDecimal を用いたクラス Interval を利用して作成した Java コードを用いて,Krawczyk 法を実行した結果を示す. 例として 4 変数の方程式,

$$f(x) = f\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 + x_3 - x_4^2 + 2 \\ x_1^2 - 2x_2^2 + 2x_4 - 1 \\ 3x_1 + 2x_2 - x_3^2 + x_4 \\ x_1 - x_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{6.1}$$

の解が, どの区間に存在するのかを示し、Krawczyk 法によりその縮小性と唯一性を確認する. まずは近似解として

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -0.8 \\ 2.0 \\ 1.5 \\ 3.0 \end{pmatrix} \tag{6.2}$$

を与えることにする.

6.2 Krawczyk クラスの使用方法

実際, 方程式に今回の Krawczyk 法のプログラムを使用する際に, 少し加えなければならない点がある. それを以下に示す.

1. Krawczyk と自動微分を行うクラス AutoBibun2,doubleAutoBibun2 に方程式の変数の数を代入する.

2. Krawczyk クラス中の方程式を代入するメソッド Func() と自動微分を行うメソッド funcBibun() に, 今回使用する方程式を代入する.

3. Krawczyk クラスに変数の初期値を代入する.

上述の式を例に取ると,1 については,Krawczyk クラスのクラス変数 SIZE に

```
 static final int SIZE=4;
```

のように代入する.

2 については以下のようになる.

まずはメソッド Func() に,double 用として方程式を

```
void Func(double x[],double f[]){
   f[0]=x[0]+x[1]+x[2]-x[3]*x[3]+2.0;
   f[1]=x[0]*x[0]-2.0*x[1]*x[1]+2*x[3]-1.0;
   f[2]=3.0*x[0]+2.0*x[1]-x[2]*x[2]+x[3]-1.0;
   f[3]=x[0]-x[3];
}
```

のように代入する.

さらに Interval クラス用に方程式を代入しなければならない. 演算子のオーバーロードは使用できないので, 関数のオーバーロードで代用する. それは以下のようになる.

```
void Func(Interval x[],Interval Fc[]){
  for(int i=0;i<SIZE;i++)
    Fc[i] = new Interval(0.0);
  Interval temp = new Interval(0.0);
  Fc[0]=temp.add(temp.add(temp.add(x[0],x[1]),
                 temp.sub(x[2],temp.mul(x[3],x[3]))),new Interval(2.0));
  Fc[1]=temp.add(temp.sub(temp.mul(x[0],x[0]),
                 temp.mul(new Interval(2.0),temp.mul(x[1],x[1]))),
                  temp.sub(temp.mul(new Interval(2.0),x[3]),new Interval(1.0)));
  Fc[2]=temp.add(temp.sub(temp.add(temp.mul(new Interval(3.0),x[0]),
                 temp.mul(new Interval(2.0),x[1])),temp.mul(x[2],x[2])),
```

```
                    temp.sub(x[3],new Interval(1.0)));
    Fc[3]=temp.sub(x[0],x[3]);
  }
```

同様に自動微分を行うために, メソッド FuncBibun() の中にも代入する.

最後に 3 については,Krawczyk クラスの Main() で

```
  double xn[] ={-0.8,2.0,1.5,3.0};
```

のように初期値を与える.

また Java を実行する時には

```
  % java Krawczyk Newton法を回す回数
```

のように引数を 1 つ指定しなければならない.

6.3 実行結果

参考のために例に挙げた方程式を実行した結果を以下に示す. ただし Newton 法は 6 回回すとする. また BigDecimal の Scale は, 数値をそろえる為に 15 に設定しておく.

計算機環境としては

OS:FreeBSD2.2.8

CPU:PentiumII 300MHz

Memory:128Mbyte

Compiler:JDK1.1.7

を用いた.

表 6.1: 実行結果

	丸め誤差無し
実行時間	0.345s
x_1	[3.6330870326470417 3.633129548901388]
x_2	[3.119708096675766 3.1198480860251916]
x_3	[4.446541156562136 4.446607985075619]
x_4	[3.6330870326470417 3.633129548901388]

	ネイティブメソッドで丸めの指定
実行時間	0.353s
x_1	[3.633087032647041 3.633129548901389]
x_2	[3.119708096675765 3.119848086025193]
x_3	[4.446541156562135 4.44660798507562]
x_4	[3.633087032647041 3.633129548901389]

	BigDecimal を使用 (Scale は 15)
実行時間	1.159s
x_1	[3.633087032647041 3.633129548901389]
x_2	[3.119708096675766 3.119848086025192]
x_3	[4.446541156562136 4.446607985075619]
x_4	[3.633087032647041 3.633129548901389]

この表 6.1 を見ると, 丸めの制御を入れると多少であるが, 区間の幅が広くなる事がわかる. この丸め誤差を考慮する事ではじめて,「 区間内に真の値を包含しながら計算する 」という区間演算の性質を保つ事ができる. また実行時間を見ると,BigDecimal を使用したときが, 格段と時間が掛かっている事もわかる. やはりネイティブ メソッドを使用し、C 言語によって区間演算をする方法と Java インタプリタで実行した違いがでたのであろう.
また Krawczyk 法により縮小性と,$f(x) = 0$ の解が上記の実行結果の区間に唯一存在する事も確認できた.

6.4 考察

以上のように,Java 言語を用いて精度保証付き数値計算のプログラムを作成した.Java 言語で作成するに当たって,

1. **Java** には「 演算子のオーバーロード 」がない事
2. **Java** では丸めの指定をすることができない事
3. インタプリタで実行するため **C** に比べて速度が遅い事

のような問題点があったが, 関数のオーバーロード , ネイティブ メソッド ,Java クラスライブラリの BigDecimal などを用いて解決する事ができた.3 についてはなるべくプログラムの計算量を減らしたり, プログラムをシンプルにする事で解決した. 実際, 今回の場合は速度は全く気にならないほどであった.
今後の課題としては,

1. プログラムを使用する際に加えなければならない事を減らすこと
2. 速度を向上するためにネイティブ メソッドの利用を考える

が挙げられる. これらの点については, まだまだ考察しなければならない事が多いので, じっくり考えていきたいと思う.

6.5 終わりに

実際に Java で精度保証付き数値計算のプログラムを作成して,C++に比べて様々な点で問題点が発生したが,Java のコードの移植性,Java がまだまだ発展途上である事を考えると,この問題点を克服して作成する意義は十分にあったと考えられる. また, 多くの Java プログラマが必要に感じたら今回の問題点であった演算子のオーバーロード, 丸めの制御を Java の仕様に加えてもらえる日も必ず来るであろう.

謝 辞

本論文の制作, 中間報告などにおいて非線形数値計算の分野,Java 言語の仕様にとどまらず, 度重なる御指導, 御鞭撻を賜わりました, 大石進一教授, 柏木雅英 助教授に心より感謝致します.

柏木研において柏木研助手 相馬隆郎 氏 , 博士課程 2 年 宮田孝富 氏には, 非線形の分野について熱心に御指導してくださり, 又本論文においてプログラムの Gauss の消去法, 自動微分に対する非常に貴重な御意見を頂きました. また, 日頃の柏木研究室内でのあらゆる面において御教示頂いたことに, 心より感謝申し上げます.

柏木研修士課程 1 年高崎大輔 氏には, 中間発表の準備の際に適切なアドバイスを頂き, 心より感謝申し上げます. 柏木研修士課程 1 年岩折朱希嗣 氏には, 私のコンピュータに関する稚拙な疑問にも親身に御指導を賜り, 心より感謝申し上げます.

最後に, 非線形班と CG 班の隔てなく, 楽しい時間をともに過ごした柏木研究室学部 4 年の皆様, 金谷卓充氏, 川上修氏, 櫻井幹夫氏, 白井健一氏, 洲濱陽一氏, 長友泰崇氏, 波多野伸哉氏, 深谷光統氏, 村竹範彦氏, 山田浩之氏, 吉田直史氏, 渡部啓氏 に心より感謝致します.

参考文献

[1] 大石進一著,“応用解析セミナー数値計算”, 裳華房,1999

[2] 柏木雅英著,“精度保証つきシミュレーション [1] -区間解析-”, 日本シミュレーション学会 “シミュレーション第 18 巻第 4 号 (平成 11 年 12 月)”

[3] ローラ · リメイ + チャールズ ·L· パーキンズ著, 武舎広幸 + 久野禎子 + 久野靖訳、“Java 言語入門”, プレンスティスホール,1996

[4] “ネイティブメソッドを Java プログラムに統合する”
http://www.n-sun.com/java/Tutorial/native/TOC.html

[5] “ネイティブメソッドについて”
http://www.db.is.kyushu-u.ac.jp/tsubo/rinkou/native/native.html

[6] “Java Platform 1.1.1 Core API”
http://www.catnet.ne.jp/terada/ja/api/packages.html

[7] “Java House Mailing List HomePage”
http://java-house.etl.go.jp/ml/