基礎プログラミング+演習 ＃1 – プログラム入門+様々な誤差

久野 靖 (電気通信大学)

2017.12.11

今回は初回ですが、次のことが目標となります。

- プログラムとは何であるか理解し、簡単な Ruby のプログラムを動かせるようになる。
- コンピュータによる実数計算の特性を理解し、さまざまな原因の誤差について判別できる。

ただしその前にガイダンスから始めて、その後本題に入ります。

0　ガイダンス

0.1　本科目の主題・目標・運用

本科目の主題ならびに達成目標は次のようになっています。

主題: コンピュータは、ソフトウェア (プログラム) によっていろいろな機能を実現している。将来、自分でプログラムを作ることがないとしても、コンピュータに関わることは避けられない。プログラムがどのように作られているかを知っていることは大変重要である。本授業では、新たな機能を実現するための方法論として、プログラミングの基礎を学ぶ.

達成目標: プログラミングに必要な基礎知識を Ruby 言語および C 言語を用いて習得し、簡単なプログラムの作成と読解ができるようになること、および、基礎的なアルゴリズムの理解や、ソフトウェアの開発方法を理解し、問題解決の基盤となる思考能力を身に付けることを目標とする。

本学では授業 1 単位について 45 時間の学修を必要とすることとなっています。本科目は 2 単位ですから 90 時間となります。これを 15 週で割ると週あたり 6 時間となります。授業そのものは 90 分 (1.5 時間) であるので、時間外に 4.5 時間の学修が必要です。課題等もこのことを前提に用意されていますので、留意してください。

本科目の運用ですが、各時間の内容は前もって Web で公開しますので、予習してくるようにお願いします。授業時間中は予習時に分からなかったことの質問を受けて捕捉説明を行い、あとはコンピュータ上での演習が中心となります。

LMS の「授業開始 5 分前以後の」演習室でのログインを用いて出席を把握します。授業を十分に受けていない方は個別判断の上、成績評価を行わない可能性があります。

各時の終了 10 分後までに、LMS を使用して当該時間の演習内容をレポートとして提出していただきますが (A 課題)、これは演習内容を振り返って頂くためのもので、成績には関係しません。

そして、上記とは別の課題に対する解答レポートを、次回授業の前日までに提出頂きます (B 課題)。B 課題は担当教員が次の 3 段階 (未提出は別扱い) で評価します。

C — 課された課題に対して不十分な内容である (ないし遅刻提出:担当教員の承認が必要)

B — 課された課題に対する解答として適切である

A — 課された課題に対する解答として優れている点がある

評価は「B」が通常ですが、担当教員の判断によって C、A をつけることもあります。

成績評価は課題点と試験点を 1:1 で合わせたものとしますが、課題点はすべて「B/b[1]」のとき満点とし、それより良い採点の場合は上積み (ないし C の減点を相殺) となります。

[1] b は総合課題回の「通常点」です。

0.2　プログラミングを学ぶ理由・学び方・使用言語

本科目ではプログラミングを学びますが、なぜこのことが必要なのでしょうか。それは、プログラミングを学ぶことではじめて、コンピュータとは何であり、何ができるかが分かるからです。

世の中の多くの人は、「特定のソフトウェアの機能や操作方法」を個別に学びますが、それだと別のソフトウェアに対面した時や、悪くすると同じソフトウェアの新バージョンに対面した時に、これまでの知識が通用しないことになります。これに対し、プログラミングから学ぶことによって、ソフトウェアに対するより一般的な「古くならない」理解力がつきます。

ただし、プログラミングをきちんとマスターするには、それなりに集中して学ぶ必要があります。今週の講義/演習をやって、それから 1 週間ほっておいて、翌週忘れたころに続きをやる、というのでは駄目なのです。このため本科目では、資料は予習していただき、時間中に演習したものを出席課題として提出していただき、さらに次回授業前日までに追加の演習をおこなっていただきます。

プログラミング言語としては、前半で Ruby 言語、後半で C 言語を使用します。このようにした理由は、初めてプログラミングを学ぶ人は Ruby のような「簡潔に書ける言語」が望ましく、そしてその後 C 言語を学ぶことで多様な言語に対する展望が得られるためです。

Ruby 処理系に関する情報は `http://www.ruby-lang.org/ja/`にあります。自宅などの Windows 上で動かしたい場合はここの「ダウンロード」ページから Windows 用バイナリを取って来て入れるとよいでしょう (Mac OS X では最初から入っています)。

0.3　ペアプログラミング

本科目では「ペアプログラミング」を採用します。これは次のようなものです。

- 1 つの画面の前に 2 人で座り、一人がキーボードを持ちプログラムを打つ。もう 1 人はそれを一緒に眺めて意見やコメントや考えを述べる。キーボードの担当者は適宜交替してもよい。

このような方法がよい理由としては、次のものがあげられます。

- プログラムを作って動かすのには多くの緻密で細かい注意が必要ですが、1 人でやるより 2 人でやる方がこれらの点が行き届き、無用なトラブルによる時間の空費が避けられます。
- プログラミングではたまたま「簡単な知識」が足りなくて、それを調べて使うまでにすごく時間が掛かることがありますが、2 人いればそのような知識を「どちらかは知っている」可能性が高まり、時間の無駄が省けます。
- プログラミング的な考え方を身に付けるには、さまざまな方面から思考したり、それを身体的な活動と結びつけることが有効です。2 人で互いに議論することで、思考が活発になり、多方面にわたるアイデアが出やすくなるため、上記のことがらに貢献します。

課題提出に際してはもちろん、2 人で作ったものですから、その 2 人については同一のプログラムを出して頂いて構いません。ただし次の条件があります。

- 提出するレポートにおいて、互いに「誰がペアであるか (相手の学籍番号)」を明示する。
- 「当日課題 (A 課題)」と「翌週までの課題 (B 課題)」でペアを変更したり、1 人で作業してはいけない。時間外もプログラミングについては 2 人で時間を合わせて作業すること。
- 個人的な好みや都合よりペアを組まずに作業することも認めます。ただしこの場合は A 課題から 1 人で作業すること。
- ペアの相手の変更や、ペアと個人作業の間の変更は、毎週可能です (授業開始時に決めてください)。一旦決めたらそれを翌週まで維持すること (どうしても途中で変更しなければならない事情が発生したら、教員まで相談してください)。

レポートはあくまでも個人単位で出して頂き、個人単位で採点します (試験も)。ペアで複数のプログラムを作った場合、どれを提出するかは各自で選んで構いません。では、よろしくお願いします。

1　プログラムとモデル化

1.1　モデル化とコンピュータ

モデル (model) とは、何らかの扱いたい対象があって、その対象全体をそのまま扱うのが難しい場合に、その特定の側面 (扱いたい側面) だけを取り出したものを言います。

たとえば、プラモデルであれば飛行機や自動車などの「大きさ」「重さ」「機能」などは捨てて「形」「色」だけを取り出したもの、と言えます。ファッションモデルであれば、さまざまな人が服を着る、その「様々さ」を捨てて特定の場面で服を見せる、という仕事だと言えます。

コンピュータで計算をするのに、なぜモデルの話をしているのでしょう? それは、コンピュータによる計算自体がある意味で「モデル」だからです。たとえば、「三角形の面積を求める」という計算を考えてみましょう。底辺が 10cm、高さが 8cm であれば

$$\frac{10 \times 8}{2} = 40(cm^2)$$

ですし、底辺が 6cm、高さが 5cm であれば

$$\frac{6 \times 5}{2} = 15(cm^2)$$

です。「電卓」で計算するのなら、実際にこれらを計算するようにキーを叩けばよいわけです。

1	0	×	8	÷	2	=

しかし、コンピュータでの計算はこれとは違っています。なぜなら、コンピュータは非常に高速に計算するためのものなので、いちいち人間が「計算ボタン」を押していたら人間の速度でしか計算が進まず意味がないからです。

そこで、「どういうふうに計算をするか」という**手順** (procedure) を予め用意しておき、実際に計算するときは**データ** (data) を与えてそれからその手順を実行させるとあっという間に計算ができる、というふうにします。そしてこの手順が**プログラム** (program) です。

たとえば面積の計算だったら、手順は

☆	×	◇	÷	2	=

みたいに書いてあり、あとで「☆は 10、◇は 8」というデータを与えて一気に計算します (もちろん、「☆は 6、◇は 5」とすれば別の三角形の計算ができます)。これを捉え直すと、「個々の三角形の面積の計算」から「具体的なデータ」を取り除いた「計算のモデル」が手順だ、ということになります。[2]

コンピュータでの計算はモデル、と言うのには別の意味もあります。三角形は 3 つの直線 (正確に言えば線分) から成りますが、世の中には完璧な直線など存在しませんし、まして鉛筆で紙の上に引いた線は明らかに「幅」を持っていて縁はギザギザ曲がっています。また、10cm とか 8cm とか「きっかり」の長さも世の中には存在しません。でも、そういう細かいことは捨てて「理想的な三角形」に抽象化してその面積を考えて計算しているわけです。

逆に言えば、コンピュータでの計算は常に、現実世界をそのまま扱うのではなく、必要な部分をモデルとして取り出し、それを計算している、ということです。この意味での抽象化やモデル化には、皆様はこれまで数学を通して多く接してきたと思いますが、これからはコンピュータでプログラムを扱う時にもこのようなモデル化を多く扱っていきます。

[2]モデルを作る時の「不要な側面を捨てる」という作業を**抽象化** (abstraction) と言います。つまり、具体的な計算を抽象化したものが手順、という言い方をしてもよいわけです。

1.2 アルゴリズムとその記述方法

「三角形の面積の計算方法」のような、計算 (や情報の加工) の手順のことを**アルゴリズム** (algorithm) と言います。ある手順がアルゴリズムであるためには、次の条件を満たす必要があります。

- 有限の記述でできている。
- 手順の各段階に曖昧さがない。
- 手順を実行すると常に停止して求める答えを出す。[3]

1 番目は、「無限に長い」記述は書くこともコンピュータに読み込ませることも不可能だからです。2 番目は、曖昧さがあるとそれをコンピュータで実行させられないからです。3 番目はどうでしょうか。実際にコンピュータのプログラムを書いてみると、手順に問題があって実行が止まらなくなることも頻繁に経験しますが、そのようなものはアルゴリズムとは言えないのです。[4]

アルゴリズムを考えたり検討するためには、それを何らかの方法で記述する必要があります。その記述方法には色々なものがありますが、ここでは手順や枝分かれ等をステップに分けて日本語で記述する、**擬似コード** (pseudocode) と呼ばれる方法を使います。**コード** (code) とは「プログラムの断片」という意味で、「擬似」というのはプログラミング言語ではなく日本語を使うから、ということです。

三角形の面積計算のアルゴリズムを擬似コードで書いてみます。[5]

- triarea: 底辺 w、高さ h の三角形の面積を返す
- 　s ← $\frac{w \times h}{2}$。
- 　面積 s を返す

1.3 変数と代入/手続き型計算モデル

上のアルゴリズム中で次のところをもう少しよく考えてみましょう。

- s ← $\frac{w \times h}{2}$。

この「←」は**代入** (assignment) を表します。代入とは、右辺の**式** (expression) [6] で表された値を計算し、その結果を左辺に書かれている**変数** (variable — コンピュータ内部の記憶場所を表すもの) に「格納する」「しまう」ことを言います。つまり、「w と h を掛けて、2 で割って、結果を s のところに書き込む」という「動作」を表していて、数式のような定性的な記述とは別物なのです。

数式であれば $s = \frac{w \times h}{2}$ ならば $h = \frac{2s}{w}$ のように変形できるわけですが、アルゴリズムの場合は式は「この順番で計算する」というだけの意味、代入は「結果をここに書き込む」というだけの意味ですから、そのような変形はできないので注意してください (困ったことに、多くのプログラミング言語では代入を表すのに文字「=」を使うので、普通の数式であるかのような混乱を招きやすいのです)。

モデルという立場からとらえると、式は「コンピュータ内の演算回路による演算」を抽象化したもの、変数は「コンピュータ内部の**主記憶**ないし**メモリ** (memory) 上のデータ格納場所」を抽象化したもの、そして代入は「格納場所へのデータの格納動作」を抽象化したもの、と考えることができます。

このような、式による演算とその結果の変数への代入によって計算が進んでいく計算のモデルを**手続き型計算モデル**と呼び、そのようなモデルに基づくプログラミング言語を**命令型言語** (imperative

[3] 実は、計算の理論の中に「答えを出すかどうか分からないが、出したときはその答えが正しい」という手順を扱う部分もありますが、ここでは扱いません。

[4] 停止することを条件にしておかないと、アルゴリズムの正しさについて論じることが難しくなります。たとえば、「このプログラムは永遠に計算を続けるかもしれませんが、停止したときは億万長者になる方法を出力してくれます」と言われて、それを実行していつまでも止まらない (ように思える) とき、上の記述が正しいかどうか確かめようがありません。

[5] 以下ではこのように、何を受け取って何を行う手順 (アルゴリズム) かを明示するようにします。上の例で「返す」というのは、底辺と高さを渡されて計算を開始し、求まった結果 (面積) を渡されたところに答えとして引き渡す、というふうに考えてください。

[6] プログラミングで言う式とは、計算のしかたを数式に似た形で記述したものを言います。先に説明した、電卓で計算する手順を記したようなものと思ってください。

language) ないし**手続き型言語** (procedural language) と呼びます。手続き型計算モデルは、今日のコンピュータとその動作をそのまま素直に抽象化したものになっています。このため手続き型計算モデルは、最も古くからある計算モデルでもあるのです。

コンピュータによる計算を表すモデルとしては他に、関数とその評価に土台を置く関数型モデルや、論理に土台を置く論理型モデルなどもあるのですが、上記のような理由から、手続き型モデルが今のところもっとも広く使われています。

2 アルゴリズムとプログラミング言語

2.1 プログラミング言語

プログラムとは、アルゴリズムを実際にコンピュータ与えられる形で表現したものであり、その具体的な「書き表し方」ないし「規則」のことを**プログラミング言語** (programming language) と呼びます。これはちょうど、人間が会話をする時の「話し方」として日本語、英語など多くの言語があるのと同様です。ただし、**自然言語** (natural language — 日本語や英語など、人間どうしが会話したり文章を書くのに使う言語) とは違い、プログラミング言語はあくまでもコンピュータに読み込ませて処理するための人工言語であり、書き方も杓子定規です。

ひとくちにプログラミング言語といっても、実際にはさまざまな特徴を持つ多くのものが使われています。ここでは、プログラムが簡潔に書けて簡単に試して見られるという特徴を持つ、**Ruby 言語** (Ruby language) を用います。

2.2 Ruby 言語による記述 exam

では、三角形の面積計算アルゴリズムを Ruby プログラムに直してみましょう。本クラスでは入力と出力は基本的に **irb** コマンド[7] の機能を使わせてもらって楽をするので、計算部分だけを Ruby の**メソッド** (method)[8] として書くことにします。先にアルゴリズムを示した、三角形の面積計算を行うメソッドは次のようになります。

```
def triarea(w, h)
  s = (w * h) / 2.0
  return s
end
```

詳細を説明しましょう。

1. 「`def` メソッド名」〜「`end`」の範囲が 1 つのメソッド定義になる。
2. メソッド名の後に丸かっこで囲んだ名前の並びがあれば、それらは**パラメタ** (parameter) ないし**引数**[9] の名前となる。メソッドを呼び出す時、各パラメタに対応する値を指定する。
3. メソッド内には**文** (statement — プログラムの中の個々の命令のこと) を任意個並べられる。各々の文は行を分けて書くが、1 行に書く場合は「`;`」で区切る。たとえば上の例のメソッド本体を 1 行に書きたければ「`s = (w * h) / 2.0; return s`」とする。
4. 文は原則として先頭から順に 1 つずつ実行される。
5. **return 文**「`return` 式」を実行するとメソッド実行は終了 (式の値がメソッドの結果となる)。

上の例は擬似コードに会わせるように、面積の計算結果を変数 `s` に入れてからそれを `return` していましたが、`return` の後ろに計算式を直接書くこともできるので、次のようにしても同じです。

[7] Ruby の実行系に備わっているコマンドの 1 つで、さまざまな値をキーボードから入力し、それを用いてプログラムを動かす機能を提供してくれます。

[8] メソッドは他の言語での**手続き・サブルーチン** (subroutine) に相当し、一連の処理に名前をつけたもののことです。

[9] メソッドを使用するごとに、毎回異なる値を引き渡して、それに基づいて処理を行わせるための仕組みです。

```
def triarea(w, h)
  return (w * h) / 2.0
end
```

このように、たったこれだけのコードでも、大変細かい規則に従って書き方が決まっていることが分かります。要は、プログラミング言語というのはコンピュータに対して実際にアルゴリズムを実行する際のありとあらゆる細かい所まで指示できるように決めた形式なのです。

そのため、プログラムのどこか少しでも変更すると、コンピュータの動作もそれに相応して変わるか、(もっとよくある場合として) そういうふうには変えられないよ、と怒られることになります。いくら怒られても偉いのは人間であってコンピュータではないので、そういうものだと思って許してやってください。

2.3 プログラムを動かす exam

では、このコードを動かしてみましょう。本科目ではシステム sol.edu.cc.uec.ac.jp での実習を前提とします。sol で Ruby を動かすためには「module load ruby」というコマンドの実行が必要です (ログインのつど 1 回必要。.tcshrc に書いておけば自動で実行させられます)。

次に、Emacs 等のエディタで上と同じ内容を sample1.rb というファイルに打ち込んで保存してください。この、人間が打ち込んだプログラムを (プログラムを動かす「源」という意味で) ソースないし**ソースコード** (source code) と呼びます。Ruby のソースファイルは最後を「.rb」にするのが通例です。そして、先に進む前に ls を使って、作成したファイルがあることを確認してください。ディレクトリが違っている場合はソースファイルのあるディレクトリに移動しておくこと。

上記が済んだらいよいよ **irb** コマンドを実行して Ruby 実行系を起動してください (「%」はプロンプト文字列のつもりなので打ち込まないでください)。

```
% irb
irb(main):001:0>
```

この「irb なんとか>」というのは irb の**プロンプト** (prompt — 入力をどうぞ、という意味の表示) で、ここの状態で Ruby のコードを打ち込めます。

プロンプトの読み方を説明すると、main というのは現在打ち込んでいる状態がメインプログラム (最初に実行される部分) に相当することを意味しています。次の数字は何行目の入力かを表しています。最後の数字はプログラムの**入れ子** (nesting — 「はじめ」と「おわり」で囲む構造の部分) の中に入るごとに 1 ずつ増え、出ると 1 ずつ減ります。とりあえずあまり気にしなくてよいでしょう。以後の実行例では見た目がごちゃごちゃしないように「irb>」だけを示すことにします。

次に load(ファイルからプログラムを読み込んでくる、という意味です) で sample1.rb を読み込ませます。ファイル名は**文字列** (string) として渡すので'' または""で囲んでください。[10]

```
irb> load 'sample1.rb'
=> true
irb>
```

true が表示されたら読み込みは成功で、ファイルに書かれているメソッド triarea が使える状態になります。成功しなかった場合は、ファイルの置き場所やファイル名の間違い、ファイル内容の打ち間違いが原因と思われるので、よく調べて再度 load をやり直してください。

なぜわざわざ 3〜4 行程度の内容を別のファイルに入れて面倒なことをしているのでしょうか? それは、メソッド定義の中に間違いがあった時、定義を毎回 irb に向かって打ち直すのでは大変すぎる

[10] 本来ならメソッドに渡すパラメタは丸かっこで囲むのですが、Ruby では曖昧さが生じない範囲でパラメタを囲む丸かっこを省略できます。本資料ではプログラム例の丸かっこは省略しませんが、irb コマンドに打ち込む時は見た目がすっきりするので丸かっこを適宜省略します。

からです。このため、以下でもメソッド定義はファイルに入れて必要に応じて直し、irb では load とメソッドを呼び出して実行させるところだけを行う、という分担にします。

load が成功したら triarea が使えるはずなので、それを実行します。

```
irb> triarea 8, 5
=> 20.0
irb> triarea 7, 3
=> 10.5
irb>
```

確かに実行できているようです。irb は quit で終わらせられます。

```
irb> quit
%
```

苦労の割に大した結果ではない感じですが、初心者の第 1 歩として、着実に進んでいきましょう。

演習 1 例題の三角形の面積計算メソッドをそのまま打ち込み、irb で実行させてみよ。数字でないものを与えたりするとどうなるかも試せ。

演習 2 三角形の面積計算で、割る数の指定を「2.0」でなくただの「2」にした場合に何か違いがあるか試せ。

演習 3 次のような計算をするメソッドを作って動かせ。[11]

a. 2 つの実数を与え、その和を返す (ついでに、差、商、積も)。何か気づいたことがあれば述べよ。
b. 「%」という演算子は**剰余** (remainder) を求める演算である。上と同様に剰余もやってみよ。何か気づいたことがあれば述べよ。
c. 与えられた数値の 8 乗を返す。ついでに 6 乗、7 乗もやるとなおよい。なお、Ruby のべき乗演算「**」は使わず、なおかつ乗除算が少ないことが望ましい。
d. 円錐の底面の半径と高さを与え、体積を返す。
e. 実数 x を与え、x の平方根を出力する。さまざまな値について計算し、精度がどれくらいあるか検討せよ。[12]
f. その他、自分が面白いと思う計算を行うメソッドを作って動かせ。

3 コンピュータ上での実数の扱い

3.1 実数の表現と浮動小数点 exam

コンピュータ上での正負の整数が 2 進法を用いて表現されていることはすでに学んできましたが、それでは小数点付きの数値はどうでしょうか。数学の世界では整数は実数の特別な場合ですが、コンピュータ上の数の表現の場合は**整数型** (integral type) と**実数型** (real type) はまったく違った性質を持っていて、プログラムの上でもきっぱり区別されます。なお、型ないし**データ型** (data type) とはデータの種類を意味する用語です。

たとえば、先の三角形の面積のプログラムで割る数を「2.0」としたのと「2」としたのでは挙動が違うのに気づいた人もいるかと思います。「10 を 3 で割る」例で確認してみましょう。実は irb では、いちいちプログラムを書かなくても式を直接計算できます。

[11] 1 つのファイルにメソッド定義 (def ... end) はいくつ入れても構わないので、ファイルが長くなりすぎない範囲でまとめて入れておいた方が扱いやすいと思います。
[12] x の**平方根** (square root) は Math.sqrt(x) で計算できます。

```
irb> 1 / 3                          ←両方とも整数だと
=> 0                                ←整数の (切捨ての) 割り算
irb> 1.0 / 3                        ←片方が実数なら
=> 0.3333333333333333               ←小数点つきの結果
```

つまり「1」や「3」は整数を表し、「1.0」のように小数点がついていると実数、という区別があり、さらに整数の割り算と実数の割り算は現然と違っているのです。

ところで、小数点つきの方は本当は無限に「3」が続くはずですが、途中で打ち切られていますね。この「表示の桁数」を自分で指定するには次のようにします。

```
irb> printf("%.20g\n", 1.0 / 3) ←小数点以下 20 桁指定
0.33333333333333331483          ← printf の出力
=> nil                          ←結果は「なし」
```

printf というのは出力の命令で、ただしそのときに「桁数や出力の幅などを指定」できるので、それを使って 20 桁表示させています。こうしてみると最後が「1483」となっていますが、つまりコンピュータでは有限の桁数で計算するため、精度に限界があります。そして「おまかせ」で表示させているときはその限界より長くは (どうせ誤差なので) 表示しないわけです。

では具体的には、有限のビット数で実数を表すのにはどうしたらよいでしょうか? たとえば、十進表現で 8 桁ぶんの整数を表す方法があるのなら、そのうちの下から 4 桁が小数点以下、その上が小数点以上、のように考えればそれで小数点付きの数が表せる、という考えもあります。

□□□□. □□□□

このような考え方を、小数点が決まった位置に固定されていることから**固定小数点** (fixed point) による実数表現と呼びます。しかし実際には、この方法はあまりうまくいきません。というのは、科学技術計算ではすぐに「30,000,000」だとか「0.0000001」のような数値が出てくるので、この方法ではすぐに限界になってしまうからです。

ではどうしましょう? たとえば理科では、上のような数値の表現ではなく、「3×10^8」とか「1×10^{-6}」のような記法が使われますね。つまり、1 つの数値を**指数** (exponent — 桁取り) と**仮数** (mantissa — 有効数字) に分けて扱うことで、広い範囲の数値を柔軟に扱うことができます。この方法は、指数によって小数点の位置を動かすものと考えて**浮動小数点** (floating point) と呼ばれます。

たとえば、同じ十進表現で 8 桁ぶんでも、6 桁の有効数字と 2 桁の指数に分けた浮動小数点表現を扱うとすれば、表せる絶対値のもっとも大きい数は「$\pm\ 9.99999\times10^{99}$」、0 でない絶対値のもっとも小さい数は「$0.00001\times10^{-99}$」ということになり、ずっと広い範囲の数が扱えることになるわけです。

注意! コンピュータでは「小さい字」が使えないので、伝統的に指数部分を「e ±指数」で表します (e は exponent の e)。たとえば「3.0×10^{22}」であれば「3.0e+22」です。このような表示は「エラー」とかではないのでそのつもりで。

コンピュータでは 2 進法を使うため、上と同様のことを 2 進表現で行います。多くの環境では、符号 1 ビット、仮数部 52 ビット、指数部 (符号含む)11 ビット、合計 64 ビットの浮動小数点表現が使われています。(このビットの割り当ては、**IEEE754** と呼ばれる標準に従ったものです。)

3.2 浮動小数点計算の誤差 exam

浮動小数点を用いた実数表現には、整数の表現とは異なる注意点があります。まず、有効数字は当然ながら有限なので、その範囲で表せない結果の細かい部分は**丸め** (rouding — 十進表現で言えば四捨五入) が行われて、**丸め誤差** (roundoff error) となります。言い替えれば、コンピュータによる実数計算は基本的に近似値による計算を行っているものと考えるべきなのです。

また、絶対値が大きく異る 2 つの数を足したり引いたりすると、絶対値が小さいほうの数値の下の桁は (演算のための桁揃えの結果) 捨てられてしまうので、これも誤差の原因となります。これを**情**

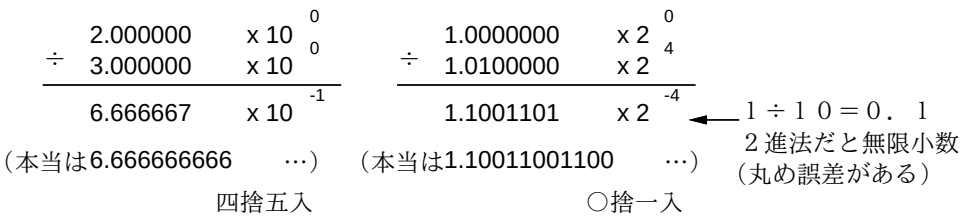

図 1: 丸め誤差

報落ち (loss of information) と言います。極端な例として、演算した結果が元の (絶対値が大きいほうの) 数のまま、ということも起こります。これは、たとえば図 2 左のような例を思い浮かべてみれば分かると思います。

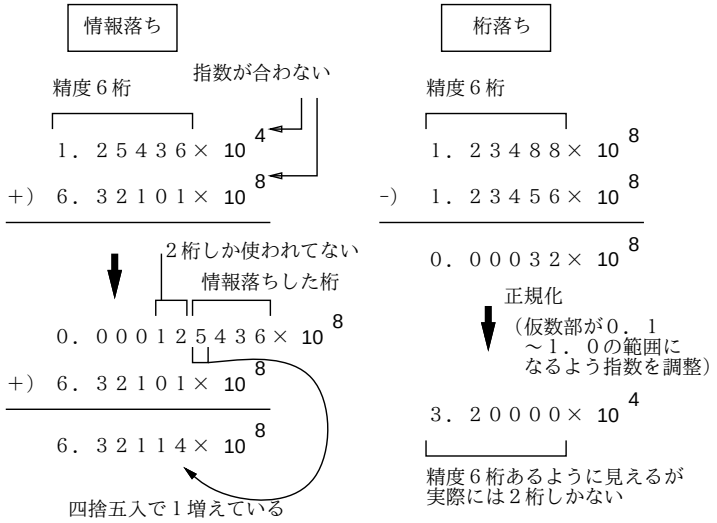

図 2: 情報落ちと桁落ち

逆に、非常に値が近い数値どうしを引き算する場合も、上のほうの桁がすべて 0 になるため、結果は元の数の下の部分だけから得られたものとなり、やはり誤差が大きくなります。これを**桁落ち** (cancellation) と言います。素朴に計算すると桁落ちが問題になる例として、次のものを考えてみます。

$$\sqrt{x+1}-1$$

x が 0 に近いとき、$\sqrt{x+1}$ も 1 に近いので桁落ちが起きます。これを避けるためには、次のように変形します。

$$\sqrt{x+1}-1=\frac{\sqrt{x+1}-1}{1}=\frac{(\sqrt{x+1}-1)(\sqrt{x+1}+1)}{\sqrt{x+1}+1}=\frac{x}{\sqrt{x+1}+1}$$

難しくしただけみたいに見えるかも知れませんが、最後の式であれば引き算をしないので桁落ちから逃れられるわけです。ところで、こう変形してみると、$x\sim 0$ のときにはこの式はおよそ $\frac{x}{2}$ だと分かりますね。実際に両方のやり方で計算してみて確認しましょう。

```
def calc1(x)
  return Math.sqrt(x + 1.0) - 1.0
end
def calc2(x)
  return x / (Math.sqrt(x + 1.0) + 1.0)
end
```

最初の素朴版から見てみます。

```
irb> calc1 0.00000000001
=> 5.000000413701855e-12
irb> calc1 0.000000000001
=> 5.000444502911705e-13
irb> calc1 0.0000000000001
=> 4.9960036108132044e-14
irb> calc1 0.00000000000001
=> 4.884981308350689e-15
irb> calc1 0.000000000000001
=> 4.440892098500626e-16
```

x が小さくなると、どんどん $\frac{x}{2}$ から外れて行きます。では修正版ではどうでしょうか。

```
irb> calc2 0.00000000001
=> 4.9999999999875e-12
irb> calc2 0.000000000001
=> 4.99999999999875e-13
irb> calc2 0.0000000000001
=> 4.999999999999876e-14
irb> calc2 0.00000000000001
=> 4.999999999999988e-15
irb> calc2 0.000000000000001
=> 4.999999999999999e-16
```

確かにこちらは大丈夫です。

最後にあと 1 つだけ、浮動小数点表現に関する注意があります。整数では全てのビットのパターンを数値の表現として使っていましたが、浮動小数点では指数部と仮数部の組み合わせ方に制約があるので (たとえば仮数部が 0 であれば値が 0 なので指数部には意味がなく、この時は指数部も 0 にしておくのが普通)、これを利用して正負の**無限大** (infinity — $\pm\infty$) や**非数** (NaN — Not a Number) などの特別な値を用意しています。また、0 にも「+0」と「−0」があったりします。だから、演算の結果としてこれらのヘンな値が表示されても驚かないようにしてください。

演習 4 2 次方程式の解の公式「$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$」について、次のことをやりなさい。

a. 係数 a, b, c の値を引数として渡すと、2 つの解を打ち出す (または「[値, 値]」の形で返す) メソッドを作成しなさい。いくつかの値で実行例を示すこと。

b. 上記に加えて、$|b|$ と $|\sqrt{b^2-4ac}|$ が非常に近い場合を解かせてみて、桁落ちによる誤差が現れることを観察しなさい。いくつかの実行例を示すこと。

ヒント: $(x+d)(x+1)$ で d が非常に 0 に近い値 (たとえば 0.000000000012345 とか) はそうなるでしょう。この式を展開して a, b, c を決めればいいわけです。

c. 仮に $b \geq 0$ とする (負なら a, b, c すべてに -1 を掛ければ解は同じで $b \geq 0$ とできる)。すると、$\pm$ のうち $-$ については両方の符号が同じなので桁落ちなしに解が求まる。これを α とおき、解と係数の関係 $\alpha\beta = \frac{c}{a}$ を利用して他方の解 β を求めることができる。この方法で 2 つの解を求めて打ち出す (または「[値, 値]」の形で返す) メソッドを作成しなさい。いくつかの値で実行例を示すこと。

d. 上記の設問 b で桁落ち誤差のあった実行例が c のプログラムでは問題なく計算できることを観察しなさい。いくつかの実行例を示すこと。

演習 5 実数の演算で誤差が現れるような、次のような計算をプログラムにおこなわせて、確かに誤差が現れることを確認しなさい。いずれも複数の実行例を示すこと。必要なら printf で表示桁数を増やしてみること。次のようなプログラムの形が想定されます。

```
def kadai5a
  printf("%.20g\n", 1.12345 - 1.0);
  (同様の行がいくつも…)
end
```

a. (桁落ち): 「1.12345 - 1.0」は「0.12345」になるでしょうか。「1.1234512345 - 1.12345」はどうでしょうか。「12345」をさらに増やすとどうなるでしょうか。

b. (情報落ち);「$1.0 + 0.0012345$」は「1.0012345」になるでしょうか。「$1.0 + 0.00000000012345$」はどうでしょうか。「0」をさらに増やすとどうなるでしょうか。

c. (丸め誤差): 「ある数に 0.1 を掛ける」場合と「ある数を 10.0 で割る (10 ではなく 10.0 にすること!)」場合では結果が違うような数があります。ところが、「0.125 を掛ける」のと「8.0 で割る (8 ではなく 8.0 にすること!)」の場合は違いはないかも知れません。どんな数がそうなるかとかその理由とかを探究してみてください。

d. その他、コンピュータの実数計算が数学と違っている具体例を示すようなプログラムを好きなように探究してみてください。

本日の課題 1A

「演習 3」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. プログラム、って恐そうですか? 第 2 外国語と比べてどう?

Q2. Ruby 言語のプログラムを打ち込んで実行してみて、どのような感想を持ちましたか?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 1B

「演習 3」(ただし 1A で提出したものは除外、以後も同様)「演習 4」「演習 5」の小課題全体から選択して 1 つ以上プログラムを作り、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. プログラムを作るという課題はどれくらい大変でしたか?

Q2. コンピュータでの数値の計算に対する数学とは違う挙動についてどう思いましたか?

Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。