基礎プログラミング+演習 ＃11– C言語入門+$f(x)=0$の求解

久野 靖 (電気通信大学)

2017.12.11

今回からC言語の内容に入ります。今回は次のことを取り上げます。

- 強い型の概念とC言語の基本、変数宣言、制御構造
- 1変数方程式の $f(x)=0$ の求解

1 演習問題解説

1.1 演習1 — 単連結リストを扱う再帰メソッド

それほど難しくないと思うのでメソッドを一通り示しましょう。まずlistsumはnilのとき0を返し、それ以外は再帰で次のセル以降の合計を求めたものに自分の値を足します。

```
def listsum(p)
  if p == nil then return 0
              else return p.data + listsum(p.next) end
end
```

listcatはnilのとき返すものが空文字列なだけで、あとは上と同じです(「+」は文字列連結にも使えるので)。なお、左右反対にするのには連結の左右を入れ換えればよいです。

```
def listcat(p)
  if p == nil then return ''
              else return p.data + listcat(p.next) end
                   # or listcat(p.next) + p.data
end
```

printmanyは再帰を2回呼ぶだけです。後で実行例を見てもらいます。

```
def printmany(p)
  if p != nil then puts(p.data); printmany(p.next); printmany(p.next) end
end
```

奇数番目のみ加算は、ヒントの通り2つのメソッドの相互再帰で奇数番目のだけ足し算します。

```
def listoddsum(p)
  if p == nil then return 0
              else return p.data + listoddsum2(p.next) end
end
def listoddsum2(p)
  if p == nil then return 0
              else return listoddsum(p.next) end
end
```

実行例は次の通り (数値のリストの課題と文字列のリストの 2 種類を用意しました)。

```
irb> Cell = Struct.new(:data, :next)
=> Cell
irb> p = Cell.new(1, Cell.new(3, Cell.new(5, nil)))
=> ...
irb> q = Cell.new('A', Cell.new('B', Cell.new('C', nil)))
=> ...
irb> listsum p
=> 9
irb> listcat q
=> "ABC"
irb> printmany q
A
B
C
C
B
C
C
=> nil
irb> listoddsum p
=> 6
```

リストの逆転は説明すると長くなるのでコードと実行の様子の図 1 だけ示します。`listrev1` は元のセルを書き換えずに新しい逆転リストを作る方法 (図の左)、`listrev2` は元のセルを書き換えて逆転リにする方法 (図の右) です。いずれも 2 つ目の引数があり、指定しないときは `nil` が初期値になります。

```
def listrev1(p, n = nil)
  if p  == nil then return n end
  q = p.next; p.next = n; return listrev1(q, p)
end

def listrev2(p, q = nil)
  if p == nil then return q end
  return listrev2(p.next, Cell.new(p.data, q))
end
```

1.2 演習 3 — エディタバッファのメソッド追加

この演習については、メソッドのみだけ掲載します。まず削除です。

```
  def delete
    if atend then return end
    @cur = @prev.next = @cur.next
  end
```

これは前回もかなり説明しましたが、要は (1) 最後の EOF は消さないようにする、(2)`@cur` は現在行の 1 つ先にする、(3)`@prev` の「次」も同じく現在行の 1 つ先にする、ということですね。どれかが足りないとおかしくなるので注意。次に交換です。

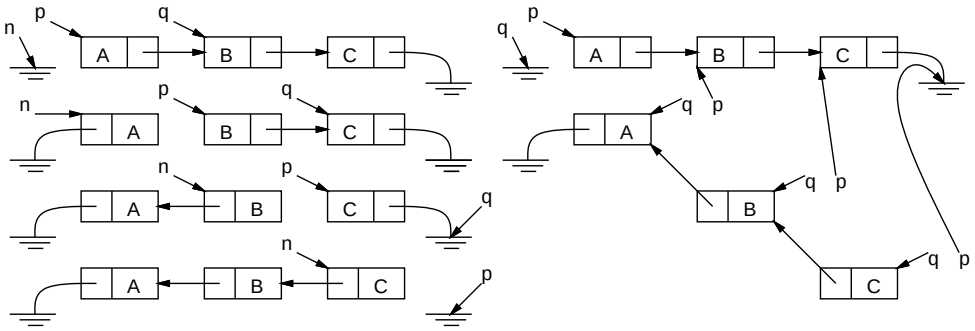

図 1: 2 通りのリスト逆転

```
def exch
  if atend || @cur.next == @tail then return end
  a = @prev; b = @cur.next; c = @cur; d = @cur.next.next
  a.next = b; b.next = c; c.next = d; @cur = b
end
```

このように込み入ったつなぎ換えは、作業変数を使った方が間違えないで済みます。まず現在行か次の行が「おしまい」だったら交換できないのでそれを除外し、あとは最終的に並ぶ 4 つのセル (中央の 2 つが交換) を変数 a、b、c、d に入れて、つなぎ直し、@cur を変更します。

次は 1 つ戻るですが、せっかくある程度メソッドが作ってあるわけですから、「@prev を覚えておき、先頭に行ってから現在行が覚えておいた行になるまで 1 行ずつ進む」方法で作ってみました。

```
def backward
  if @prev == @head then return end
  a = @prev; top; while @cur != a do forward end
end
```

全部反転はやや大変ですが、先頭から順にたどりながら、今まで「前→後」の対だったものを「後←前」の順になるように参照をつなぎ換える (ただし先頭と末尾はそれなりに対処)、という方針です。

```
def invert
  top; if atend then return end
  a = @cur; b = @cur.next; a.next = @tail
  while b != @tail do c = b.next; b.next = a; a = b; b = c end
  @head.next = a; top
end
```

先頭と末尾の対処はまず最初に先頭の次を@tail にし、最後にループを抜けてきた時のセルを先頭 (@head の次) にする、ということです。なお、バッファ内に 1 行しかない時はループ周回数が 0 で、その時もちゃんと動作することに注意。

1.3 演習 5 — エディタの機能強化

「指定した行へ行く」機能はバッファ側で「何行目」を管理するのがよいので、エディタバッファ全体を示します。インスタンス変数@lineno を追加し、現在行が変化するメソッドでこれを更新します。invert のように大幅に直す場合は最後に top を呼び、ここで 1 にリセットされるので問題ありません。そして gogo があれば backward はずっと簡単です。行番号を間違いなく維持するのはきわどそうに見えますが、@lineno をアクセスするのもバッファ内容や現在位置を変更するのも Buffer 内だけなので、この中できちんと処理すれば大丈夫です。つまり、オブジェクト指向の持つ**カプセル化**の機能によって、プログラムが正しく構成し易くなるのです。

```
  class Buffer
    Cell = Struct.new(:data, :next)
    def initialize
      @tail = @cur = Cell.new("EOF", nil)
      @head = @prev = Cell.new("", @cur)
      @lineno = 1
    end
    def getlineno
      return @lineno
    end
    def goto(n)
      top; (n-1).times do forward end
    end
    def atend
      return @cur == @tail
    end

    def top
      @prev = @head; @cur = @head.next; @lineno = 1
    end
    def forward
      if atend then return end
      @prev = @cur; @cur = @cur.next; @lineno = @lineno + 1
    end
    def insert(s)
      @prev.next = Cell.new(s, @cur)
      @prev = @prev.next; @lineno = @lineno + 1
    end
    def print
      puts("  " + @cur.data)
    end
# delete、exch は上掲のとおり。backward は以下のように変更
    def backward
      goto(@lineno - 1)
    end
    def invert
      top; if atend then return end
      a = @cur; b = @cur.next; a.next = @tail
      while b != @tail do
        c = b.next; b.next = a; a = b; b = c
      end
      @head.next = a; top
    end
# subst, read, write は前回資料掲載
  end
```

エディタドライバ側も一応示します。「位置変更しない指定行数プリント」も、最初に行番号を覚え、印刷し終わったらそこに戻ればよいので簡単です。

```
def edit
  e = Buffer.new
  while true do
    printf(">")
    line = gets; c = line[0..0]; s = line[1..-2]
    if c == "q"    then return
    elsif c == "t" then e.top; e.print
    elsif c == "p" then
      e.print; l = e.getlineno;
      s.to_i.times do e.forward; e.print end; e.goto(l)
    elsif c == "i" then e.insert(s)
    elsif c == "r" then e.read(s)
    elsif c == "w" then e.save(s)
    elsif c == "s" then e.subst(s); e.print
    elsif c == "d" then e.delete
    elsif c == "x" then e.exch
    elsif c == "b" then e.backward
    elsif c == "v" then e.invert
    elsif c == "a" then e.forward; e.insert(s); e.backward
    elsif c == "c" then e.delete; e.insert(s); e.backward
    elsif c == "g" then e.goto(s.to_i)
    else                e.forward; e.print
    end
  end
end
```

2 C言語入門

2.1 弱い型と強い型

これまで使ってきたRubyでは、変数は使ったときに自動的に用意され、どのような種類の値でも格納できました。それは確かに便利なのですが、変数名を間違ったり入れる値の種類を間違えても処理系は教えてくれないという弱点がありました。どのみちプログラムは実際に動かして間違いを調べる必要があるから、と思うかも知れませんが、苦労して動かしながら調べるよりも処理系が「これは違います」と言ってくれる方がずっと簡単に直せるのも確かです。

そのため世の中には、次のような設計のプログラミング言語も多くあります。

- 変数は使う前に**宣言** (declaration — これからこの変数を使うという指定) を書く必要がある。
- 宣言時に変数に**データ型** (値の種別) を明示し、それと異なる値を入れることを許さない。

なお、ここでは「変数」とだけ書きましたが、関数のパラメタや関数や返す値についても同様です (そうしないと検査がきちんとできない)。このような言語を、型を厳密に検査することから**強い型の言語** (strongly-typed language) と呼びます。[1] 繁雑で不自由なようですが、その方が結局プログラムの誤りを速やかに修正できる、というのが、この方式を支持する人の主張です。

また、強い型の言語のほうがCPU命令への変換がやりやすく、結果としてプログラムが高速に実行できる場合が多い、という点もあります。さらに、CPUの動作との対応がはっきりしていて、**組み込みシステム** (embedded system — 様々な機器にCPUが搭載されていてそこでプログラムが動く

[1] Rubyのようにどの変数にどの種類の値を入れてもよい言語は**弱い型の言語** (weakly-typed language) です。

もの) で使いやすい、という性質もあります。この 2 つの利点はこれから取り上げる C 言語にとくにあてはまります。つまり、皆様がこれから研究や仕事でハードウェアに近いプログラミングをやる場合、C 系列の言語を使う可能性が高いと言えます。

そして本科目の立場としては、Ruby で弱い型をやったので、それと異なる強い型の言語も知って欲しいということと、2 つ異なる言語を体験することで「さまざまな言語といっても似たところも多い」ことを学んで頂きたいということから、**C 言語** (C language) を取り上げています。

2.2 C 言語のバージョンについて

本題に入る前に、C 言語のバージョンについて説明しておきます。C には現在おおよそ「K&R」「C89」「C99」「C11」の 4 つの版があります。K&R は最初に C 言語を解説した本「The C Programming Langauge」の著者 Kernighan、Ritche の名前からそう呼ぶもので、古い版です。そのあと標準化活動により C89、C99、C11 ができました (数字はそれぞれの規格ができた年の西暦の下 2 桁です)。

C99 が C89 の使いにくいところを改良していて処理系も普及しているので、本科目では C99 にしたがって説明しています。ただし今日でも C89 の処理系を使う場面があるかも知れないので、C89 と互換性のないところはその旨を説明し、C89 に書き換える方法も注記するようにします。面倒な話ですみませんでしたが、ではいよいよ本題に進みます。

2.3 最初の C プログラム exam

本科目で最初にやった Ruby プログラムが三角形の面積だったので、C 言語でもそうします。ずっと長いですが、それは主に irb に相当するものが無くて入力も自分で扱う必要があるためです。

```
// triarea --- area of triangle
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  if(argc != 3) {fprintf(stderr,"needs 2 args.\n");return 1;}
  double w = atof(argv[1]), h = atof(argv[2]);
  printf("%.8g\n", w * h / 2.0);
  return 0;
}
```

では最初なので丁寧に説明しましょう。

1. 「`// ...`」はそこから行末までがコメントになります。C では最初に実行する関数は `main` という名前に決められているので、何をするプログラムかのコメントは書いた方がいいでしょう。なお、このほかに「`/* ... */`」というコメントの書き方もできます。[2]
2. 「`#include <ファイル名>`」はシステムの決まった場所にあるファイルを取り込む指示です。[3] 最後が「`.h`」で終わるファイルは**ヘッダファイル**と呼ばれ、関数等の宣言が書かれています。前述のように、C では型検査のために、使用する各関数についてパラメタや結果の型を記述する必要があり、入出力など誰もが使う関数も同様です。しかしそれを毎回マニュアルを見て手で書き込むのは大変なので、宣言を集めたファイルが用意されています。`stdlib.h` は変換関数 `atoi` や `atof`、`stdio.h` は出力関数 `printf` の宣言を含んでいます。当面はこの 2 つを取り込めばよいです (追加はそのつど説明します)。

[2] C89 まででは「`/* ... */`」型のコメントしか使えないので注意。ここでは見やすいので「`// ...`」を主に使います。
[3] 「`#`」で始まる命令は行の先頭文字が「`#`」である必要があるので注意。

3. 上述の通り、C では実行を開始する関数は `main` であり、プログラムに必ず `main` を含めます。その宣言ですが、`int` は整数型で、`main` は必ず整数を返します (C ではこのように「型名関数や変数の名前」の形で型を宣言します)。そして `main` はプログラム起動時にシステムから文字列の配列としてパラメタを受け取るので、パラメタも宣言します。後述しますが、起動するコマンドが「`./a.out 7 5`」だと、`"./a.out"`、`"7"`、`"5"`の 3 つの文字列が渡されます。

   そこでパラメタですが 1 番目の `int argc` はパラメタ文字列の数で、上記の個数です (個数なので `int` つまり整数です)。2 番目の `char *argv[]` は、文字列 (`char *`) の配列 (`[]`) です。こちらの書き方は少し後で説明しますので、当面こう書くと思ってください。なお、`argc` と `argv` という名前は別の名前でも動きますが、この 2 つの名前にしておく習慣になっています。

4. 次は if 文ですが、Ruby と書き方を対比して見て頂くのが簡単なのでそうします (図 2)。

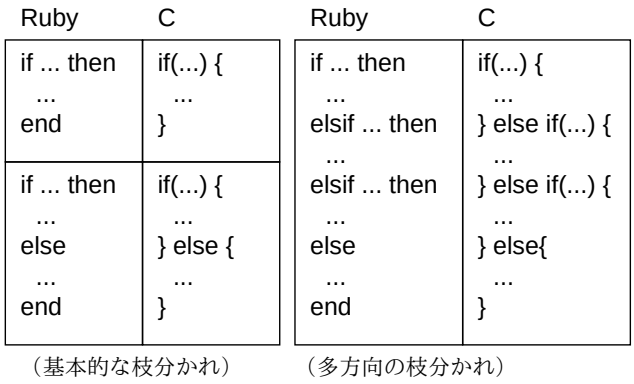

図 2: Ruby と C の if 文の対比

   C は「`{ ... }`」だらけですが、`{ ... }`の中の文が 1 つだけなら中かっこを書かなくてもすみます。しかし間違いが起きやすくなるので、本資料では常に書きます。

5. if の中身ですが、比較演算子、かつ/または/〜でない等は Ruby と同じです。ここでは底辺・高さの 2 つを指定するので、0 番目のコマンド名と合わせ `argc` は 3 のはずで、そうでない場合は指定が違うというエラーを出して終わります。標準エラー出力への書き出しは「`fprintf(stderr,` 文字列`);`」です。stderr という変数は `stdio.h` で宣言され使えるようになっています。C では文字列はダブルクォート`"..."`のみ使えます。`main` は正常終了なら 0、そうでない場合は 0 以外を返す約束なので、ここでは 1 を返します。

6. Ruby では「`;`」は文を続けて書く際の区切りですが、C では文は常に終わりに「`;`」をつけます。

7. 高さと幅は実数をですが、C では実数として通常使うのが「倍精度実数型」(`double`) なので、ここでもそう宣言します。初期値としてそれぞれ、コマンド引数の 1 番目と 2 番目を **atof**(文字列を実数に変換する関数、Ruby でいう `.to_f`) で実数に変換して代入します。[4]

8. `printf` の機能は (フォーマット文字列も含めて)Ruby と同じです。ここでは 2 番目のパラメタのところに直接計算式を書いていますが、もちろん「`double z = w * h / 2.0;`」のように別の変数に入れてからここで `z` を出力しても構いません。

9. ここで終わりですが、`main` の正常終了なので 0 を返します。

10. 関数の本体も「`{ ... }`」で囲みます。

続いて動かし方を説明します。プログラムのファイルは C では「`.c`」で終わる慣例なので、Emacs で `sample1.c` に打ち込んだとします (`ls` でファイルがあることを確認)。次に C では**コンパイラ** (compiler) と呼ばれるプログラムでソースプログラムを翻訳し、CPU 命令の列に変換します。sol では GCC(GNU

[4] C89 まででは、宣言はブロック (`{ ... }` のことです) の先頭にしか書けないという制約があります。その場合は、`main` の直後に「`double w, h;`」という宣言だけ入れて、この箇所は宣言なしの代入にしてください。

C Compiler) というコンパイラを使用し、gcc というコマンドで翻訳します。GCC はエラーがなければ a.out というファイルに実行可能形式を出力するので、次のようにコマンド引数を与えて動かします (%はプロンプトのつもりなので打ち込まないこと)。

```
% gcc sample1.c  ←コンパイル。何も出力が無いなら OK
% ./a.out 7 5    ←引数を指定して実行
17.5             ←出力
```

演習 1 例題をそのまま打ち込み実行しなさい。実行できたら、プログラムの一部をわざと色々に壊してコンパイルし、エラーの出かたを体験しなさい。OK なら次の課題をやりなさい。

a. 2 つの実数を与え、その和を返す (差、商、積も)。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。(ちなみに文字列を整数に変換する関数 atoi(s) を使う必要があると思われます。)
c. 円錐の底面の半径と高さを与え、体積を返す。
d. 実数 x を与え、その平方根を出力する。さまざまな値で試し、精度を検討せよ。
e. その他、自分が面白いと思う計算を行う関数を作って動かせ。

わざと Ruby の初期の演習と同じにしましたが、どうでしょうか。あと、**平方根** (square root) は **sqrt**(x) で計算できますが、これを使うには冒頭に「#include <math.h>」を追加する必要があり、また次の実行例のように gcc コマンドの末尾にオプション-lm を追加してください (ちなみに桁数を増やすため printf の書式文字列は「"%.20g\n"」にしています)。

```
% gcc sqrt1.c -lm
% ./a.out 2
1.4142135623730951455
```

なお、math.h 取り込みと-lm 指定を行うことで、次の**数学関数**が使えるようになります。

- sqrt(x) — 平方根, cbrt(x) — 立方根、pow(x,y) — べき乗 x^y (実数)
- abs(x) — 絶対値 (整数)、fabs(x) — 絶対値 (実数)
- exp(x) — e^x、log(x) — $\ln x$、log10(x) — $\log_{10} x$
- sin(x)、 cos(x)、tan(x) — $\sin, \cos, \tan$
- asin(x), acos(x), atan(x), atan2(y,x) — 逆三角関数で、最後のは $\arctan \frac{y}{x}$ を計算し、x が 0 のとき y の正負に応じ $\pm\frac{\pi}{4}$(± 90 度) を返すので便利。

2.4 C 言語の演算子 exam

Ruby については演算子をまとめて説明しませんでしたが、C についてはここで主なものをまとめて説明してしまいます (説明がややこしいものは後で必要なところで追加します)。演算子には結び付きの強さがあります (たとえば a * b + c は (a * b) + c ですから、* は-より**優先順位**が高い、と言います。表 1 に、主要な演算子を優先順位順に記します。

増加/減少演算子は C 言語における発明の 1 つで、「値を 1 増やす/減らす」専用の演算です。たとえば「++i」とすると、変数 i の値が 1 増えます。プログラムで 1 増やす/減らすことは多いので結構役に立ちます。さらに特異な点として、これらの演算子は後置と前置の両方で使えます。その違いは「増減する前の値/増減した後の値」が式の値となることです。

```
int i = 10; // i は 10
int j = i++; // i は 11 になるが、j は増やす前の 10 が入る
int k = ++i; // i は 12 になり、k は増やした後の 12 が入る
```

表 1: C 言語の主要な演算子 (優先順位順)

種別	演算子
単項演算子	++ (増加)、-- (減少)、+ (プラス符号)、- (符号反転)、~ (ビット反転)、! (論理否定)
2 項演算子	*、/、% (乗算、除算、剰余)
2 項演算子	+、- (加算、減算)
2 項演算子	<<、>> (左シフト、右シフト)
2 項演算子	>、>=、<、<= (比較演算子)
2 項演算子	==、!= (等しい/等しくない)
2 項演算子	& (ビット毎 and)
2 項演算子	| (ビット毎 or)
2 項演算子	&& (論理 and)
2 項演算子	|| (論理 or)
3 項演算子	x?y:z (if x then y else z end)
2 項演算子	= (代入)、+=、-=、*=、/=、%=、&=、|= (複合代入)
2 項演算子	, (順次評価)

次に、~、&、|はビット毎演算で、たとえば sol の環境では int は 32 ビットの 2 の補数表現で整数を表しますが、それをビットの列とみなしてビット毎に NOT、AND、OR をとります。さらに<<、>>はシフト演算で、左側の項をビットの列とみなして右側で指定した値だけずらします (空いた場所には 0 のビットが入ってきます)。これらは整数の値専用です。

そして論理演算や比較演算は Ruby と同様です。なお、C では「はい」は 1、「いいえ」は 0 で表すので、これらの演算は 0 または 1 を返します。さらに、&&は左の項が 0 なら右の項は計算せずに 0 を返しますし、||も左の項が 1 なら右の項は計算せずに 1 を返します。

次に代入=は、「右辺の値を左辺の変数に入れる」演算であり、入れた値が結果となります。代入には+=(x += 1 は x = x + 1 と同等) など演算と代入がくっついたものが一通りあります。

最後に「,」は文が書けないところで順番に実行したいときに使われます。たとえば「x = 1, y = 2」は変数 x に 1、y に 2 を入れ、右の項の値 2 全体の値となります。

演習 2 表 1 から興味ある 2 項演算子を 1 つ以上選び、整数を 2 つ与えてその演算の結果を表示するプログラムを作り、結果を検討しなさい。文字列を整数に変換するのには atof の代わりに **atoi** を呼べばできます。整数を出力するのには「printf("%d\n", 式);」でできます。

2.5 繰り返しの構文 exam

if 文については最初の例題のついでに説明してしまいましたが、繰り返しの構文についてここで説明しておきます。まず **while ループ**については Ruby と同等で、書き方が少し違うだけです。

```
while(条件) {
  ...
}
```

問題は **for ループ**で、C 言語はこれがとっつきにくいので定評 (?) があります。そして、Ruby の times や step のようなメソッドも無いので、**計数ループ**にはこれを使うしかありません。具体的には、C の for ループは図 3 のように while ループを書き換えたものになっています。たとえば、変数 i を 0 から 9 の手前まで 1 ずつ増やす計数ループであれば次のように書くことになります。

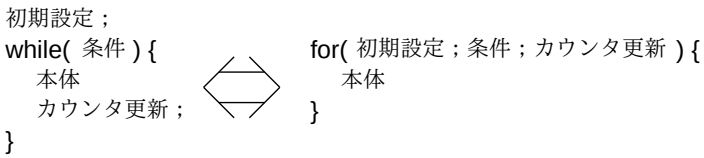

図 3: C 言語における for 文の意味づけ

```
for(int i = 0; i < 10; ++i) {
  ...
}
```

なお、この例のように for の初期化部分に変数宣言を書くのは C99 で許されていますが、なぜか `gcc` では「`-std=c99`」という指定を追加してコンパイルする必要があります。[5]

while、for とも途中で繰り返しを抜けるのに「`break;`」という文が使えます (Ruby と同じ)。途中で繰り返しの残りを飛ばして次の周回に進むには「`continue;`」です (Ruby の `next`)。[6]

3 $f(x) = 0$ の求解

3.1 数え上げによる求解

C の書き方の話ばかりではつまらないので、こんどは関数 $f(x)$ について、$f(x) = 0$ を満たす x を求めるという問題、つまり **1 変数方程式の求解**を取り上げます。これも解析的に解けなくても、次の条件が満たされていればプログラムで解を求めることができます。

> ある区間 $[a, b]$ において、$f(x)$ が単調増大、連続、かつ $f(a) < 0$、$f(b) > 0$ となるような a、b が分かっている

a でマイナス、そこからなめらかに増えていって、b でプラスになっているのなら、その間のどこかに解があるわけですから、それを求めればよいわけです。

たとえば「$N(> 1)$ の平方根を求める」ことを考えます。$f(x) = x^2 - N$ とすれば、$f(0) < 0$、$f(N) > 0$ なのでここで説明する方法で解を求められます (そしてそれが N の平方根なわけです)。

では具体的にどうやったら $f(x) = 0$ の解が求まるでしょうか? たとえば、次の方針はどうでしょう?

> 小さい値 d を決めて、a から初めて $f(a + d)$、$f(a + 2d)$、$f(a + 3d)$、$\cdots$ を求めて行く。
> はじめてその値が 0 以上になったところが解である。

これで確かに「誤差 d で」解が求まります。これを**数え上げ** (enumeration) 法と呼びます。

演習 3 数え上げ法によって平方根を求める C プログラムを作成しなさい。精度をあげた時にどれくらいまで実用になるか検討しなさい。

3.2 区間 2 分法

数え上げ法はコンピュータらしいとは言えますが、いかにも効率が悪そうです。そこで端からちょっとずつ計算するかわりに、a と b の中間の値を計算するように、次の方針を考えます。

> $c = \frac{a+b}{2}$ を求め、$f(c)$ を計算する。もしも $f(c) < 0$ であれば、解がある範囲は区間 $(c, b]$。
> そうでなければ、解がある範囲は区間 $[a, c]$ とわかる。そこでこのどちらかに応じ、a、b いずれかを c で置き換え、同様に繰り返すことを、$|b - a|$ が十分小さくなるまで行なう。

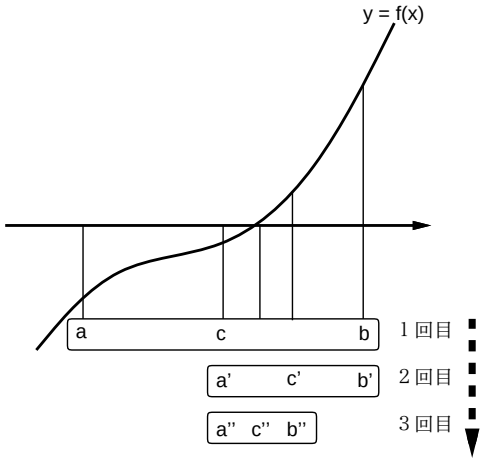

図 4: 区間 2 分法による求根

これは 1 ステップごとに区間を 2 つに分けるため、**区間 2 分法** (binary search method) と呼びます。

$2^{10} = 1024$ ですから、区間を半分にすることを 10 回繰り返すと区間の幅はおよそ 1000 分の 1、40 回繰り返すとおよそ 1 兆分の 1 になります。言い換えれば、その精度で解が求まるわけです。

演習 4 区間 2 分法によって平方根を求める C プログラムを作成しなさい。必要と思われる精度にしたとき、繰り返し回数がいくつになるか検討しなさい。

3.3 ニュートン法

ニュートン法 (Newton's method) は、万有引力の発見者ニュートンに由来する方法で、[7] 適当な近似値 r から始め、その近似値を改良していくことで解に到達します。具体的には、$f(x)$ の $x = r$ における接線を求め、接線と X 軸が交わる点の X 座標を新たな r とし、これを反復していきます。その i 回目の値を r_i と書き、各回の計算内容を**漸化式** (recurrence formula) として表しましょう。

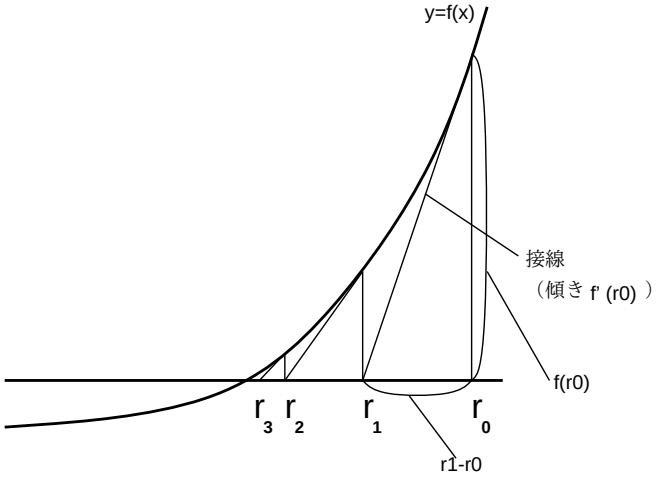

図 5: ニュートン法による求解

[5] C89 まではどのみち、変数宣言はブロックの先頭に置く必要があり、初期化部分に書けるのは代入だけです。

[6] for 文ではこの場合、カウンタ更新に進みます (次の周回でカウンタが増えないと不便)。この点で、図 3 に示した「while と for の書き換え」は完全に同等ではありません (while で残りをスキップした場合すぐ次の条件テストに進む)。

[7] 彼は微積分学の発明者の一人でもあります

具体的にやってみましょう。$x = r_i$ の時の接点の座標は $(r_i, f(r_i))$、そこでの接線の傾きは $f'(r_i)$ (もちろん関数は微分可能でないといけません)。

$$\frac{f(r_i)}{r_i - r_{i+1}} = f'(r_i)$$

より、

$$r_{i+1} = r_i - \frac{f(r_i)}{f'(r_i)}$$

となります。$f(x) = x^2 - n$ の場合、$f'(x) = 2x$ より、

$$r_{i+1} = r_i - \frac{{r_i}^2 - n}{2r_i} = \frac{r_i}{2} + \frac{n}{2r_i}$$

となります。そこで $r_0 = N$ とおき、$r_1, r_2, \cdots$ を計算していくと、その値は $\sqrt{N}$ に**収束** (converge) していくわけです。一般にこのような、近似値を反復によって改良していく方法を**反復解法** (iterative method) と呼びます。反復の結果、値がほとんど変化しなくなったら、つまり $|r_{i+1} - r_i| < \epsilon$ となったら収束したこととし、そこでの近似値を解とするわけです。

なお、収束する値が大きい値であるような計算をする場合は、**絶対誤差** (absolute error) ではなく**相対誤差** (relative error) に基づいて収束を判定するのがよいかもしれません。その場合は反復をやめる条件は次のようになります。

$$\left|\frac{r_{i+1} - r_i}{r_i}\right| < \epsilon$$

ニュートン法は収束すれば高速なことで知られていますが、収束しない場合もあります。平方根の計算の場合は、最初の近似値として N から始めれば問題ありません。

演習 5 ニュートン法によって平方根を求める C プログラムを作成しなさい。必要と思われる精度にしたとき、繰り返し回数がいくつになるか検討しなさい。(ヒント: 繰り返しごとに現在の近似値を書き出すのでもよいですね。)

演習 6 C 言語で書いてみたいと思う自分にとって興味深い題材をプログラムとして作成しなさい。

本日の課題 11A

「演習 1」または「演習 2」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. 強い型の言語、とくに C 言語についてどう思いましたか。

Q2. 言語が異なるとプログラミングの方法も異なると思いますか。

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 11B

「演習 1」〜「演習 6」の (小) 課題から 1 つ以上を選択してプログラムを作り、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. C 言語でプログラムが書けるようになりましたか。

Q2. C と Ruby はどのように違うと感じていますか。

Q3. 課題に対する感想と今後の要望をお書きください。