計算数学

松本　眞*

平成17 年1 月10 日

目 次

この講義ノートと、ここに書かれた多くのプログラムが
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/TEACH/teach.html
からダウンロードできる

*広島大学理学部数学科 m-mat@math.sci.hiroshima-u.ac.jp

1　π の計算

円周率 π は、円周の長さを直径で割って得られる比のことである。

$$\pi = 3.14159265358979323846264338327950288419716939\cdots$$

であることが知られている。

僕が数学へ進むきっかけになったのは、小学生のころ「π の話」（野崎明弘著、岩波科学の本書店）[2] という本に出会ったことが大きい。

そこでは、π を計算することを通して

1. 円周の長さを直径で割った比は円の大きさによらず一定なの？
2. 円周の長さってどうやって定義するの？
3. 有理数でない数って、どうやって計算するの？
4. 有理数でない数って、そもそも何？

といった根源的な疑問に触れることができる。こういった根源的な問題についてはチャンスがあったらいずれ一杯飲みながら話合うとして[1]、ここでは、計算機を用いて「いかにして計算するか」を講義する。

1.1　$22/7, 355/113$

中国では 2000 年以上前から円周率の近似として

$$22/7(\text{約率}), 355/113(\text{密率})$$

が知られてきた。これを C 言語で計算してみよう。

プログラム 1.1. yakuritsu.c

```
#include <stdio.h>
main(void)
{
  double x, y;
  x = 22.0 / 7;
```

[1] 一応簡単に答：1 は、「曲線の長さ」を定義しなければ証明できないが、定義すれば証明できる。なお、球面上などの非ユークリッド幾何の世界では比は一定でないことに注意。2 は、「線積分」を使うしかないと思う。3 は、以下である程度の説明がなされる。4 は、実は 1–3 について答えるのに前提として考えておかなくてはならない問題であり、「実数とは何か」という問いである。それは、一言でいうなら「無限小数であらわされる数の全体」のことであるのだが、実数論をきちんと扱った解析の本 (高木貞治「解析概論」[1] など) を読み、理解しない限り何のことだかわからないだろう。言い換えると、「実数とは何か」をある程度理解してはじめて 1–3 について正確に答えうる。

```
  y = 355.0 / 113;
  printf("%f \n",x);
  printf("%f \n",y);
}
```

このプログラムを、例えば yakuritsu.c というファイルに書き込んでセーブし、

```
>cc yakuritsu.c
```

と入力し Enter ボタンを押すと a.out ないし a.exe というファイルができる。このファイルが「実行可能ファイル」と呼ばれるもので、コンピュータが実行できるものである。前者のばあい

```
>./a.out
```

と入力し Enter ボタンを押すと次のような出力を得るはずである。

```
3.142857
3.141593
```

あっさりとプログラムの説明をする。

```
#include <stdio.h>
```

は、「標準 (STanDard) 入出力 (Input/Output) を利用するライブラリプログラムを使いますよ」という宣言である。

```
main(void)
```

は、ここから主ルーチンという宣言。

```
{
  double x, y;
  x = 22.0 / 7;
  y = 355.0 / 113;
  printf("%f \n",x);
  printf("%f \n",y);
}
```

がプログラムの本体で、倍精度 (double) の実数変数 x,y を用意し、x に 22/7 を、y に 355/113 を計算して代入。printf はフォーマット (Format) つき印刷 (PRINT) で、x を出力し、続いて y を出力する。

初めての人はわからないことだらけであろうが、とりあえず演習の授業などで実践してみてほしい。

printf 文を

```
printf("%.20f \n",x);
printf("%.20f \n",y);
```

に書き換えると、小数点以下 20 桁を出力するようになる。コンパイル (cc をかけること）して実行すると次のような出力を得るだろう。

```
3.14285714285714279370
3.14159292035398252096
```

実際に 22/7 を手で筆算してみると、$3.142857142857\cdots$ と周期 6 で循環することがわかる。

問題 1.2. 22/7 の小数表示が周期 6 で循環することを証明せよ。

しかし、上のように C で普通に計算すると小数点以下 16 桁目から狂ってしまう。これは、「丸め誤差」のせいである。

計算機では、倍精度実数は二進浮動点小数であらわされ、その有効桁数は 2 進で 53 桁、10 進に直すと $53 \times \log_2 10 \simeq 16$ 桁なのである。

などといわれても初心者の人は困ってしまうだろうが、とりあえずは「計算機で実数の計算をするときには有効桁数のせいで誤差がでてしまう」ということを頭の片隅においてもらえればいい。

1.2 格子点を数えて

さて、古代中国では 22/7 や 355/113 をどうやって求めたのだろうか？僕は知らない。

ここでは、どんな方法でもいいからともかく円周率を近似的に求めてみようと思う。

原点中心で半径 1 の円の第一象限の部分（つまり、$0 \leq x, y \leq 1$ の部分）の面積は $\pi/4$ である。

おおきな自然数 N を決めて、$[0, 1]$ 区間を N 等分する。すると、単位正方形

$$\{(x, y) | 0 \leq x, y \leq 1\}$$

は N^2 個の小正方形に等分される。これら小正方形のうち、円にすっぽり含まれているものの個数を $a(N)$ であらわすならば、

$$a(N)/N^2 < \pi/4$$

となる。

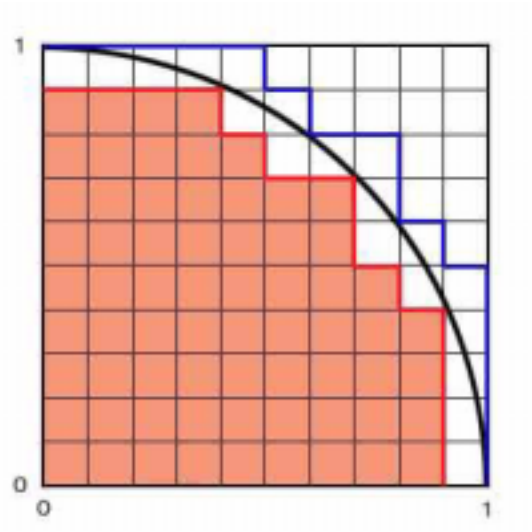

図 1: 一辺の $N = 10$ 等分による四分円の近似

では、$a(N)$ を求めてみよう。整数変数 i, j を、それぞれ 1 から $N - 1$ まで動かしてみる。点 $(i/N, j/N)$ を右上の隅とする一辺 $1/N$ の小正方形を考える。このとき、$(i/N)^2 + (j/N)^2 \leq 1$ となっていることと、この小正方形が四分円に入ることとは同値である。

これを実行するには、例えば次のようなプログラムでよい。

プログラム 1.3. koushi-1.c

```
#include <stdio.h>
#define N 100
main(void)
{
  int i, j;
  int a=0;
  for (i=1; i<N; i++)
    for (j=1; j<N; j++)
      if (i*i + j*j < N*N) a++;
  printf("pi is nearly %.10f\n", ((double) a)/(N*N)*4);
}
```

前に比べてやや難しくなっているが、再びあっさり説明する。

```
#define N 100
```

というのは、マクロ定義とよばれるものである。その意味は、「以下、N という文字が出てきたら 100 に置き換えて解釈しなさい」という宣言である。

```
  int i, j;
```

は「i, j を整数変数の名前にします」

```
  int a=0;
```

は「a を整数変数の名前とし、最初はその値を 0 にします」という宣言である。次の for 文はちょっとややこしい。

```
  for (i=1; i<N; i++)
```

は、「次の文章を繰り返し行います。まず i を 1 にして、次の文章を行います。終えたら、i に 1 を足します。(i++は、i に 1 を足す命令。)。そしてまた、次の文章を行います。…いつになったらやめるかというと、i<N という条件が満たされなくなったらやめます。」という命令である。このように、ある処理を繰り返し行うよう指示する命令文を「ループ文」という。

説明は長かったが、要は最初の for 文により i は 1 から N-1 まで順番に動く。動きながら次の文を実行するのだが、次の for 文では j が 1 から N-1 まで順番に動く。結局、変数 i,j の組は

$$(1,1),(1,2),\ldots,(1,N-1),(2,1),(2,2),\ldots,(2,N-1),\cdots,(N-1,N-1)$$

と動きながら

```
      if (i*i + j*j < N*N) a++;
```

を $(N-1)\times(N-1)$ 回繰り返し実行することになる。これだけ全部動いたら、次の printf 文に処理が移る。

繰り返し行われる文

```
      if (i*i + j*j < N*N) a++;
```

は if 文と呼ばれるもので、if の後の括弧の中身が真ならば次の文章を実行し、そうでないときは次の文章をスキップする。この場合は、$i^2+j^2<N^2$ かどうかを計算して判定し、そうである場合には a を 1 増やし、そうでないときはなにもしない。

結果、$(N-1)\times(N-1)$ 通りの (i,j) のうち、四分円に入る正方形の右上隅になるようなものの個数が a に格納されることになる。

$\pi/4$ の下からの（つまり小さいほうからの）近似は $a/N^2 < \pi/4$ で与えられるのだから

$$a/N^2 \times 4 < \pi.$$

プログラムでは

```
printf("pi is nearly %.10f\n", ((double) a)/(N*N)*4);
```

により a を倍精度実数とみなして N の自乗で割って 4 倍した値を、「% .10f」すなわち小数点以下 10 桁出力してみている。

実際に動かしてみよう。コンパイルして実行すると

```
pi is nearly 3.1000000000
```

なる出力を得る。

冒頭の# define 文を

```
#define N 1000
```

に変えてみよう。コンパイルして実行すると

```
pi is nearly 3.1375240000
```

を得る。

なかなか 3.14 に到達しないのを、意外に思うか当然に思うか？

N の値を 1000 から 10000 に増やして実行してみよう。すると少し待ち時間がかかった上で（数秒かかるだろう）、

```
pi is nearly 3.1411902000
```

を得るだろう。

この計算には $(N-1)^2$ 回の繰り返しが必要となる。大体 N^2 だと思えば、N を 10 倍にすると全体では 100 倍の計算時間がかかることになる。

ここで、いくつかの疑問が少なくとも僕には生じる。

- 古代中国人はどうやって 355/113 に到達したのか？正方形を 10000 × 10000=1 億個に細分しても 3.14119 にしかならない。
- 下からの評価

  $$3.14119 < \pi$$

  は証明できたが、π の上からの評価、すなわち「大きくてもせいぜいこのくらい」という評価はどうしたら求められるだろうか。

前者はともかく、後者については「四分円を覆い隠すだけの小正方形の数」を計算すればよい。それには、今までのように「右上端が四分円に入る」という条件でなく「左下隅が四分円に入る」というよりゆるい条件を満たす小正方形の数を数えればいい。それをやるには

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    if (i*i + j*j < N*N) a++;
```

と i, j の動く範囲を 0 からに広げるだけで十分である。

問題 1.4. π を上から評価するプログラムを書き、実行せよ。なぜ i, j の動く範囲を 0 からに広げるだけでいいのか？

注意 1.5. `N` を余りに大きくすると (例えば 40000)、これらのプログラムは正しく動かない（ことが多い。C 言語の処理系に依存する）。それは、$N \times N$ の計算において桁あふれが起きたり符号の問題がおきるからである。

このようなプログラムの不具合（プログラムに紛れ込んだ「虫 (bug)」というニュアンスで、普通バグという）については、この講義の範疇をやや超えている。興味のあるひとは演習の先生か私に質問に来て欲しい。

注意 1.6. 実は、このような計算方法が「面積の定義＝積分」の定義の本質を与えている。ある図形の面積とは、それに含まれる小正方形たちの面積の和で下から押さえられ、それを含む小正方形たちの面積の和で上から押さえられる。小正方形を細かくしていったとき、これらの極限が一致するときその値をその図形の面積という。これは微積分学で習うリーマン積分の概念である。

このプログラムのように、ある値（図形の面積など）を計算機を用いて近似的に計算することを（近似）数値計算という。この場合、

1. 真の値と、求めた近似値との誤差の評価
2. どのくらいの時間で、上の誤差評価をどれだけ小さくできるか

が問題となる。

プログラム 1.3 では、図 1 を見てわかるとおり誤差は「円周にひっかかる小正方形の面積の和」で抑えられる。すなわち誤差 $\pi/4 - a(N)/N^2$ の評価は

$$0 < \pi/4 - a(N)/N^2 < \text{円周に交わる小正方形の数}/N^2$$

となる。

問題 1.7. 1. 円周に交わる小正方形の数は $2N-1$ 以下であることを証明せよ。

2. プログラム 1.3 で求められる値の、真の円周率からの誤差が $4(2N-1)/N^2 \simeq 8/N$ 以下であることを示せ。

さて、上の問題によれば円周率を小数点以下６桁まで計算しようとすると $N = 800000$ くらいにしなくてはならない。実は桁あふれのためプログラムも書き直さなくては動かないが、なにより for 文による繰り返しが N^2 回必要であり、うんと時間がかかる。($N = 10000$ の場合の、80×80 倍の時間がかかる。仮に $N = 10000$ が 1 秒でできたとすると、6400 秒かかることになる。)

もう少し、賢い方法はないだろうか。問題になるのは N^2 個の点全てではなく、円周の近くにある点だけであることに注意しよう。少しプログラミングに慣れれば、次のような方法を C で書けるようになるであろう。

問題 1.8. 1. 図 1 の二つの折れ線を求める方法を考えよ。左上の点 $(0, 1)$ から始めて、きざみはば $1/N$ で右に進むか下に進む。円周のぎりぎり下（ぎりぎり上）を通るようにして、$(1, 0)$ に到達するまで進む。
2. この折れ線を用いて、円周率を上と下から評価する方法を考えよ。
3. その方法を C 言語でプログラムし、実行せよ。

この方法だと、折れ線を通るのに $2N - 1$ 回の計算をすればよいことになりさきの N^2 に比べると格段に高速になっている。

本当は自分でプログラムするのがいいのだが、初心者へのヒントとして下からの評価を与える C プログラムを以下に挙げる。

プログラム 1.9. oresen.c

```
#include <stdio.h>
#define N 10000
main(void)
{
  long i;
  long a=0;
  long x=0, y=N;
  for (i=0; i<2*N-1; i++) {
    if ((x+1)*(x+1) + y*y < N*N) {
      x++;
      a = a + y;
    } else {
      y--;
    }
  }
```

```
  printf("pi is nearly %.10f\n", ((double) a)/(N*N)*4);
}
```

これだと一瞬にして

```
pi is nearly 3.1411902000
```

が得られる。

一般に、ある問題を解くための計算方法のことをその問題を解くアルゴリズムという。プログラム 1.3(koushi-1.c) プログラム 1.9(oresen.c) は「一辺 $1/N$ の小正方形が四分円にいくつ入るか？」という問題を解くためのアルゴリズムを与えている。そして、後者の方が高速であるという点で優れたアルゴリズムであるといえる。

1.3 計算時間とランダウの O

unix 系の計算機では、time というコマンドで実行時間を計測することができる。N の値を 10000 として

```
> cc koushi-1.c
```

を実行すると a.exe または a.out なる実行可能ファイルができる。

```
> ./a.out
```

と入力することにより $N^2 = 10^8$ 回の計算が行われて答えがでる。

この際、

```
> time ./a.out
```

と入力すると a.out を実行するのにかかった時間が表示される。

```
pi is nearly 3.1411902000


real 0m1.525s
user 0m1.452s
sys  0m0.050s
```

といった出力を得る。ここで、real のところにこのプログラムの実行にかかった時間 (上の例だと 0 分 1.525 秒) が出力され、そのうちユーザー側でプログラムの実行に消費した時間が user のところに、システムが消費した時間が sys のところに表示される。

といっても意味がわからないと思うが、私も良く分かっていない。ともかく、user のところの時間がプログラム実行にかかった正味の時間である。

折れ線によるアルゴリズム oresen.c の方の時間を測ってみよう。

```
> cc oresen.c
> time ./a.out
```

を実行すると、

```
pi is nearly 3.1411902000


real 0m0.153s
user 0m0.030s
sys  0m0.060s
```

を得る。user の部分の時間をくらべて、後者の方が約 50 倍速く解答を得ていることがわかる。ためしに $N = 30000$ として時間を比較してみると、`koushi-1.c` の方が 12.758 秒、`oresen.c` の方は 0.020 秒であり、後者が 600 倍速い。

注意 1.10. 前者の繰り返し回数は N^2, 後者は $2N - 1$ である。ので、後者が約 $N/2$ 倍早そうなものだが、そうはならない。出力文 (`printf`) などに結構時間がかかるからであろうと思うが、確かなところは私は知らない。

注意 1.11. コンパイルするたびに `a.out` は上書きされてしまう。これだと、二つのアルゴリズムの速度比較をするときなど、二つ以上の実行可能ファイルをとっておきたいときに不便である。

ひとつの方法は、コンパイルするたびに

```
> cc koushi-1.c
> mv a.out koushi
```

として、`a.out` を別のファイル (上の例では koushi) にとっておくことである。

もう一つの方法としては、

```
> cc -o koushi koushi-1.c
```

としてやると `koushi` (Windows 系では `koushi.exe`) というファイルを `cc` が作って、そこにコンパイル結果を書き出してくれる。`-o` は output(出力) を指定するオプションである。

問題 1.12.

```
> cc -o koushi koushi-1.c
> cc -o oresen oresen.c
> time koushi
> time oresen
```

の４つをこの順に実行してみよ。

さて、プログラム 1.3 は N^2 回の繰り返しを行い、プログラム 1.9 は $2N-1$ 回の繰り返しを行う。しかしながら、繰り返されている部分の計算は違うので、かかる時間の比が単純に N^2 対 $2N-1$ になるわけではない。一回当たりでは、後者の方が複雑な計算を行うために余計に時間が必要になる。

こうしてみると、時間の比較を行うことに限れば、N^2 とか $2N-1$ とかいう精密な回数はあまり意味をもたない。$2N$ と $2N-1$ の違いはさして重要ではない。

そこで、ランダウという物理学者が導入したとされる $O(N)$, $O(N^2)$ などの記号を使うと便利である。これは、関数の大きさをおおまかに評価するときに使われるもので、O はオーダー（地震でいうマグニチュードのようなニュアンス）の頭文字である。

定義 1.13. N を自然数を走る変数とする。$f(N)$ を自然数から正の実数への関数とする。このとき、$O(f(N))$ で「N が増えていくとき、$f(N)$ よりそう大きくはならない関数」の集合を表す。より正確には、

$$g(N) \in O(f(N)) \Leftrightarrow \exists N_0 \in \mathbb{N}, \exists C > 0, \forall N > N_0, |g(N)| \leq Cf(N)$$

により定義される集合である。

上の定義の右辺はどんなことを言っているか。

「ある N_0 が存在して、N_0 よりも大きな N に対しては、$|g(N)| < f(N)$ が成立、つまり絶対値をとっても $g(N)$ の方が $f(N)$ より小さくなっている。」といいたいところだが、もっとおおらかな気分で「ある正定数 C が存在して、N_0 より先では $|g(N)| < Cf(N)$ となっている」ということを主張している。

例 1.14.

1. $2N-1 \in O(N)$。なぜならば、$C=2$, $N_0=1$ ととればよい。
2. $N^2 \notin O(N)$。どんなに $C>0$ を大きくとっても、どんなに N_0 を大きくとっても、
$$N^2 < CN$$
はいつか崩れてしまう。（いつか、とは $N > C$ となるときから。）
3. $f(N)$ を正の実数に値をとる関数とする。
$$\lim_{N\to\infty} \frac{g(N)}{f(N)}$$
が有限の値になるとき (0 でも良い)、
$$g(N) \in O(f(N))$$
である。（逆は必ずしも正しくない。）

この記法を用いて、次のようにいう。「プログラム 1.3 の (時間) 計算量は $O(N^2)$ に属するが、$O(N)$ には属さない。プログラム 1.9 の (時間) 計算量は $O(N)$ に属する。」

解析学（微積分学）では、この大文字の O によく似てちょっと違う、小文字の o が用いられる。

定義 1.15. N を自然数を走る変数とする。$f(N)$ を自然数から正の実数への関数とする。このとき、$o(f(N))$ で「N が増えていくとき、$f(N)$ との比をとればいくらでも小さくなる関数」の集合を表す。より正確には、

$$g(N) \in o(f(N)) \Leftrightarrow \forall \epsilon > 0, \exists N_0 \in \mathbb{N}, \forall N > N_0, |g(N)| \leq \epsilon f(N)$$

により定義される集合である。

この意味は、「神様がどんなに小さな ϵ をおっしゃっても、それが正である限り、我々はある $N_0 \in \mathbb{N}$ を見つけることができる。N_0 より先の $N > N_0$ では、

$$|g(N)|/f(N) < \epsilon$$

となっているように。」

これは、解析学で習う収束の定義の一種である。実際、

$$g(N) \in o(f(N)) \Leftrightarrow \lim_{N \to \infty} g(N)/f(N) = 0$$

である。

直感的に言えば、$g(N) \in O(f(N))$ とは $g(N)$ の大きさが $f(N)$ の大きさとせいぜい互角 (正の定数倍は互角と思う) であることを意味し、$g(N) \in o(f(N))$ とは $g(N)$ の大きさは $f(N)$ の大きさにいくらでも負けていくことを意味している。

問題 1.16.

1 で、N によらず常に値 1 を取る定数関数を表す。

$$g(N) \in O(1)$$

ということと、$|g(N)|$ が有界であることとは同値であることを示せ。

$$g(N) \in o(1)$$

ということと、

$$\lim_{N \to \infty} g(N) = 0$$

ということとは同値であることを示せ。

定義 1.17. 実数 x と実数の部分集合 S に対し、

$$x + S := \{x + s | s \in S\}$$

という記法が広く用いられている。集合と元を足して、結果として集合が得られることになる。同様に、

$$x \times S := \{x \times s | s \in S\}$$

が定義される。

$g(N)$ と $h(N)$ が「比較的近い関数」であることを表すのに、

$$g(N) \in h(N) + O(f(N))$$

といった記述をすることがある。これは、上の定義から

$$g(N) - h(N) \in O(f(N))$$

と同値であるから、「$g(N)$ と $h(N)$ の差の絶対値は、せいぜい $f(N)$ と互角の大きさである」ということを意味している。

同様に、

$$g(N) \in h(N) + o(f(N))$$

と書いたら、「$g(N)$ と $h(N)$ の差の絶対値は、$f(N)$ との比で言えばいくらでも小さくなっていく」ことを意味している。

例 1.18.

- $2N - 1 \in 2N + O(1)$.
- プログラム 1.3 や 1.9 の計算結果 $\in \pi + O(\frac{1}{N})$.

  実際、π と計算結果の誤差は $4(2N-1)/N^2$ 以下であることを問題 1.7 で示した。

  $$\lim_{N\to\infty} |4(2N-1)/N^2|/(1/N) = \frac{1}{8}$$

  であるから、例 1.14 の最後によって

  $$|4(2N-1)/N^2| \in O(\frac{1}{N})$$

  である。

  この例のように、$4(2N-1)/N^2$ という複雑な情報を扱う代わりに、$O(1/N)$ という荒っぽいが扱いやすい情報を使って誤差や速度を比較するのがランダウの記法の利便さである。

問題 1.19. 次を示せ。

1. $g(N), h(N) \in O(f(N)) \Rightarrow g(N) + h(N) \in O(f(N))$
2. $g(N) \in O(f(N)) \Rightarrow -g(N) \in O(f(N))$
3. $0 \in O(f(N))$

代数学で群論を習った人は、この問題が次の事実を示していることに気づくとよい：「$O(f(N))$ は、自然数から実数への関数の集合がなす加法群の部分群である。」

問題 1.20.

$$g(N) \in h(N) + O(f(N))$$

であることを、

$$g(N) \equiv h(N) \mod O(f(N))$$

と記す。次の三つを示せ。

1. $g(N) \equiv h(N) \mod O(f(N)), h(N) \equiv k(N) \mod O(f(N))$
   $\Rightarrow g(N) \equiv k(N) \mod O(f(N))$
2. $g(N) \equiv h(N) \mod O(f(N))$
   $\Rightarrow h(N) \equiv g(N) \mod O(f(N))$
3. $g(N) \equiv g(N) \mod O(f(N))$

どこかで同値関係という概念を習った人は、この問題が「$\equiv \mod O(f(N))$ が同値関係である」ことを示していると納得できると良い。

代数学で「部分群による剰余群」「合同関係」を習った人は、上の二つの問題により「N を変数とする実数値関数の加法群 $/O(f(N))$」なる剰余群が定義されることに気づくと良い。

と、やや脱線したが別に以上のことに気づかなくてもこの講義では良い。次の文章の意味が分かれば十分である。

- プログラム 1.3 は、誤差を $O(1/N)$ にするのに計算時間が $O(N^2)$ かかるアルゴリズムである。
- プログラム 1.9 は、誤差を $O(1/N)$ にするのに計算時間が $O(N)$ かかるアルゴリズムである。

注意 1.21. $g(N) \in h(N) + O(f(N))$ であることを、通常の書物では

$$g(N) = h(N) + O(f(N))$$

と記すのであるが、これは僕には気持ち悪い。

$$g(N) + O(f(N)) = h(N) + O(f(N))$$

なら正確である。

プログラム 1.9 を解説しておく。long は int より「大きな」整数変数を宣言している。(多くの処理系で 32 ビット (すなわち 2 進 32 桁) 整数変数をあらわす。 -2^{31} から $2^{31}-1$ までの整数を表すことができる。)

最初に宣言されている long 整数変数 i は単に $2N-1$ 回ループを回す際のカウンタである。a は $a(N)$ を蓄えるためのもので、初期値は 0 である。(x,y) は一辺 N の正方形の左上隅 $(0,N)$ を出発し、四分円の円周の内側を通って $(N,0)$ へと向かう。for 文では、次の { から } までを $2N-1$ 回繰り返す。繰り返されるのは

```
if ((x+1)*(x+1) + y*y < N*N) {
  x++;
  a = a + y;
} else {
  y--;
}
```

である。これは if-else 文と呼ばれるもので、

if (式) 文 1 else 文 2

という書式を持っている。() の中の式を計算し、それが真ならば文 1 を実行して次に移る。偽ならば文 2 を実行して次に移る。

上の例では、

- 式が (x+1)*(x+1) + y*y < N*N
- 文 1 が {x++; a = a + y;}
- 文 2 が {y--;}

である。したがって、点 (x,y) において x を 1 増やしたときにもし円の外にでなければ、x を 1 増やして a には y を足してやる。もし円の外に出てしまうならば、y を 1 減らしてやる。

これを $2N-1$ 回繰り返す。点 (x,y) は右または下に 1 ずつ動いていく。もし右に動いても円の外にでないならば右に動き、そのときはその下にある正方形の個数である y を a に足してやる。右に動くと円の外に出てしまうときは、下に動く。

1.4 モンテカルロ法で

単位正方形を細かく分断するかわりに、でたらめな点を単位正方形にたくさん打って、そのうちの何個が四分円に入るかを数えることでも円周率は近似できる。

プログラム 1.22. pi-monte1.c

```
#include <stdio.h>
#define  M 10
main(void)
{
  long i, j=0;
  double x, y;
  for (i=0; i<M; i++) {
    x = ((double) rand())/0x80000000;
    y = ((double) rand())/0x80000000;
    if (x*x + y*y < 1) j++;
    printf("(%f,%f)\n",x,y);
  }
  printf("pi is nearly %.10f\n", ((double) 4*j)/M);
}
```

なるプログラムをコンパイルして実行すると、処理系に大いに依存するが例えば

```
(0.000000,0.690001)
(0.505418,0.591491)
(0.554785,0.378429)
(0.257732,0.207382)
(0.626262,0.340127)
(0.843852,0.068778)
(0.409907,0.879994)
(0.319480,0.980568)
(0.085005,0.907629)
(0.102509,0.921978)
pi is nearly 3.6000000000
```

なる出力結果を得る。プログラム 1.22 は、単位正方形に M 回でたらめに点を打ち、そのうちで四分円に入ったものの個数 j を求める。j/M が大体四分円

の面積$\pi/4$になるはずであるから、$4*j/M$を計算することでπの近似値が求まる。

`rand()`は、31ビット符号なし整数に値をとる一様擬似乱数を生成する関数である。これは、呼ばれるたびに$0 \sim 2^{31}-1$の間の整数を、それぞれほぼ等確率$1/2^{31}$ででたらめに生成する。そこで、それを(double)で倍精度実数だと思い、2^{31}で割ってやれば[0,1)半開区間に一様に分布する実数擬似乱数とほぼみなすことができる。

2^{31}を記すのに、計算して2147483648と書く方法もあるがここでは16進数表記を用いて0x80000000と書いた。C言語では0xで始まる数字は16進数で書かれているとみなす。$0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f$で0から15の数を表す。例えば0x1aは$16\times 1+10=26$をあらわす。0x8は$8=2^3$をあらわし、0x80は$8\times 2^4=2^7$をあらわす。0x80000000は$8\times(16)^7=2^{31}$をあらわすことになる。

途中の動作を示すために

```
printf("(%f,%f)\n",x,y);
```

を入れて、打った点の座標を表示するようにしてある。M=10000などにすると、10000回座標を表示するのは邪魔なのでこのprintf文を取り除く必要がある。実際にM=10000で実験すると$\pi\sim 3.1728$を得た。

このように、乱数を用いて確率的な方法で面積や体積などを近似的に求める方法の総称を**モンテカルロ法**という。上の場合、M回点を打って得られる近似値の誤差は、大体$O(1/\sqrt{M})$となると期待される。

より一般に、面積1の図形Xの中に面積pの図形Cがあるとする（体積でも良い）。Xにランダムに一様（面積あたり落ちる確率が一定）に点を打ったとき、それがCに入る確率はpである。M個の点を独立に打ったとき、そのうちのちょうどa個がCに入る確率は

$$\binom{M}{a}p^a(1-p)^{M-a}$$

である。モンテカルロ法では、このときCの面積pを

$$p\simeq a/M$$

で近似する。

問題 1.23. 上の式で確率が求まることを示せ。上のような確率分布を二項分布（binomial distribution）という。

問題 1.24. 1. a/Mとpの差（つまり、誤差）の期待値は

$$E(a/M-p)=0$$

であることを示せ。（これは、誤差がプラスになることとマイナスになることとが打ち消しあっている、ということを言っているだけである。）

2. a/M と p の差（つまり、誤差）の自乗の期待値は

$$E((a/M-p)^2) = p(1-p)/M$$

でることを示せ。

上の結果により、誤差の自乗の期待値は (p を定数とするならば) $O(1/M)$ である。そのため、誤差の絶対値はおよそ $O(1/\sqrt{M})$ になると考えられる。

注意 1.25. 上でみたようにモンテカルロ法は「誤差を $O(1/M)$ にするのに、計算回数は $O(M^2)$ かかる」方法である（M を M^2 にとりなおした）。したがって、円の面積を近似するという目的においてはプログラム 1.9 の方法の方がすぐれている。しかし、モンテカルロ法は

- 求めたい図形 C の形がよくわからないとき
- 3 次元図形の体積、あるいはより大きな次元（例えば 4 次元以上）の図形の体積を求めたいとき

には威力を発揮する。というより、この方法でしか体積が求まらないことが多い。

例えば、100 次元超立方体内の図形の体積を求めようと思ったとき、一辺を N 等分するとこの超立方体は N^{100} 個に分割されてしまう。このような状況では、図形に入る小超立方体を数えたり、境界にある小超立方体を数えるのは困難である。

1.5 $\tan^{-1}(x)$ で

三角関数と微積分学を用いると、はるかに効率よく π を求めることができる。

命題 1.26.

$$\tan^{-1}(y) = \int_0^y \frac{1}{x^2+1}dx$$

証明は、右辺の積分を $x = \tan\theta$ により変数変換すると良い。

命題 1.27. $0 \le y \le 1$ のとき

$$\begin{gathered} y - \tfrac{1}{3}y^3 + \tfrac{1}{5}y^5 \cdots - \tfrac{1}{4n-1}y^{4n-1} \\ \le \textstyle\int_0^y \frac{1}{x^2+1}dx \le \\ y - \tfrac{1}{3}y^3 + \tfrac{1}{5}y^5 \cdots - \tfrac{1}{4n-1}y^{4n-1} + \tfrac{1}{4n+1}y^{4n+1} \end{gathered}$$

証明.

$$1 - x^2 + x^4 \cdots - x^{4n-2} \leq \frac{1}{x^2+1} \leq 1 - x^2 + x^4 \cdots - x^{4n-2} + x^{4n}$$

は高校の範囲で証明できる。あとは両辺を積分すればいい。 □

これにより、$y = 1$ を代入すれば

$$1 - 1/3 + 1/5 - 1/7 \cdots$$

なる交代級数（符号が毎回逆転する数列の無限和）は、$\pi/4$ を上下しながら収束していく。（この方法を発見したのはニュートンであるといわれている。）

これを C プログラムにしよう。

プログラム 1.28. newton.c

```
#include <stdio.h>
#define N 10000
main(void)
{
  long i;
  double a=0;
  double sign=1;
  for (i=1; i<N; i++) {
    a += sign / (2*i-1);
    sign = -sign;
  }
  printf("pi is nearly %.10f\n", 4*a);
}
```

for 文は、i が 1 から N-1 まで、1 ずつ増やしながら次の二行を繰り返すことをしめしている。交代級数であるので符号 (sign) は+,-を交互にとる。繰り返しのたびに sign / (2*i-1) を加えていく。

実行すると、

```
pi is nearly 3.1416926636
```

を得る。

さて、N を変えるのにいちいちコンパイルして実行するのは面倒くさい。そこで、N を入力できるようにしよう。

C 言語で入力を処理する関数の一つに、scanf というものがある。

```
scanf("%d",&n)
```

を実行すると、ここで入力待ちが行われる。%d は「整数 decimal の形での入力を行う」という意味であり、& n は「変数 n に入力を行う」という意味である。この& 記号は「アドレスを取り出す」という操作をあらわしているのだが、これについては後述する。

プログラム 1.29. newton-input.c

```
#include <stdio.h>
main(void)
{
  long i,n;
  double a=0;
  double sign=1;
  printf("何回繰り返す？");
  scanf("%d",&n);
  for (i=1; i<n; i++) {
    a += sign / (2*i-1);
    sign = -sign;
  }
  printf("pi is nearly %.10f\n", 4*a);
}
```

コンパイルして実行してみよう。「何回繰り返す？」と出力されたら、好きな整数を入力し、Enter キーを押してみよう。例えば 1 千万を入力すると、次のような出力を得るだろう。

```
何回繰り返す？10000000
pi is nearly 3.1415927536
```

問題 1.30. 上のプログラム newton.c において、N 回の繰り返しを行ったときの誤差が $O(1/N)$ であることを示せ。

これにより、小数点以下 10 桁を正確に出すには大体 10^{10} 回、すなわち 100 億回繰り返せば良い。しかし、time を使って測ってみると 1 億回の繰り返しで 3 秒ほどかかっており、100 億回だと 300 秒=5 分ほどかかる計算になる。

次の式を使うと、あっというまに正確な値がでる。

命題 1.31.

$$\tan^{-1}(1) = 4\tan^{-1}(1/5) - \tan^{-1}(1/239)$$

問題 1.32. この等式を証明せよ。両辺の tan をとり、加法定理を繰り返し使えばできる。

命題 1.26, 1.27 を用いれば

$$tan^{-1}(1/5), tan^{-1}(1/239)$$

を計算できるが、誤差のオーダーは前者が N 回の繰り返しで $O(5^{-2N}/2N)$, $O(239^{-2N}/2N)$ でありすごい速さで小さくなる。

実際にプログラムを組む際には、前者の誤差と後者の誤差のオーダーが一致する程度に繰り返しの回数を調整してやると良い。

$$5^{3.4} \sim 239$$

であるから、前者の繰り返し回数と後者のそれの比を 3.4:1 くらいにするとよい。次のコードがそれを実現している。

プログラム 1.33.
```
#include <stdio.h>
main(void)
{
  long i,n;
  double a=0, b;
  double sign=1;
  printf("何回繰り返す？");
  scanf("%d",&n);
  b=1.0/5;
  for (i=1; i<n; i++) {
    a += b / (2*i-1);
    b = -b /(5*5);
  }
  a *= 4;
  b=-1.0/239;
  for (i=1; i<n/3.4; i++) {
    a += b / (2*i-1);
    b = -b /(239*239);
  }
  printf("pi is nearly %.20f\n", 4*a);
}
```

10 回の繰り返しで 3.14159265359086203873 を得るだろう。（厳密には 13 回の繰り返し。）

この時の誤差のオーダーは $5^{-20}/20$ くらいであり、$\log_{10} 5 \sim 0.7$ を使えば $5^{-20} \sim 10^{-14}$ であるから 14 桁目ぐらいまでは正確にもとまっているはずである。実際にはそこまで正確ではないが。

さて、先にプログラム 1.1 を実行したとき、C の倍精度計算 (double を使った計算) での有効桁数は 16 桁までであることをみた。このため、16 桁以上を正確に計算することはこのままではできない。この講義ではこれ以上深追いしないが、double 程度ではなくもっと桁数の多い計算を行うライブラリが必要となる。

1.6 e はどうよ

ここでは、単に問題を述べて終わっておく。

問題 1.34. 自然対数の底 e をなるべく正確に計算せよ。

2　循環小数

2.1　a/n の計算

§1.1 では 22/7, 355/113 の値を計算したが、小数点以下１６桁目から狂ってしまっていた。これは、C 言語では倍精度計算をしても 有効桁数が１０進で１６桁程度しかないためである。

普通に割り算の筆算をやって周期を求めれば、循環小数として正確に値を求めることができる。このプログラムを書いてみよう。

プログラム 2.1. shousuu-1.c

```
#include <stdio.h>
main(void)
{
  unsigned long state, bunbo, keta;
  int i;
  printf("a ÷ n の小数展開を計算します。a,n を入力ください。\n");
  printf(" a=");
  scanf("%d",&state);
  printf(" n=");
  scanf("%d",&bunbo);
  printf("何桁もとめますか？桁数を入力ください。\n 桁数=");
  scanf("%d",&keta);
  printf("%d.",state / bunbo);
  state = state % bunbo;
  for (i=0; i<keta; i++) {
    printf("%1d", (state * 10)/bunbo);
    state = (10 * state) % bunbo;
  }
  printf("\n");
}
```

コンパイルして実行し、a, n の値を入力すると次のような計算結果を得る。

```
a ÷ n の小数展開を計算します。a,n を入力ください。
 a= 355
 n= 113
何桁もとめますか？桁数を入力ください。
 桁数= 50
3.14159292035398230088495575221238938053097345132743
```

はじめてみる演算子%は、「割った余り」を求めるものである。

```
state = state % bunbo;
```

では、state を bunbo で割った余りが state に格納される。例えば、22 ÷ 7 を計算しよう。state には 22 が、bunbo には 7 が入力される。そののち、

```
printf("%d.",state / bunbo);
state = state % bunbo;
```

を実行すると、state ÷ bunbo の商を出力し、小数点を打つ。state には bunbo を割った余りが入る。よって、まず 22 ÷ 7 の商である 3 を出力し、小数点を打つ。そして、state には余りである 1 が入る。その後、keta 回にわたり、

```
printf("%1d", (state * 10)/bunbo);
state = (10 * state) % bunbo;
```

が実行される。これは、直前の余りが入った state の中身に対し、下一桁に 0 をつけ、bunbo で割った商（一桁）を出力する。そして、そのときの余りを state に格納する。

これは、

$$
\begin{aligned}
22 \div 7 &= 3 \text{ 余り } 1 \\
10 \div 7 &= 1 \text{ 余り } 3 \\
30 \div 7 &= 4 \text{ 余り } 2 \\
20 \div 7 &= 2 \text{ 余り } 6 \\
&\cdots
\end{aligned}
$$

という、筆算による小数割り算に他ならない。

2.2 周期の計算

さて、上のプログラム 2.1 を使うと例えば

$$3/52 = 0.05769230769230\cdots$$

である。この場合、0.05 の部分は循環せず、その後の 769230 は周期 6 で循環する。

このような循環小数に対しては、非循環部分の長さが 2 桁、循環部分の長さが 6 桁であるという。

一般に a/n の小数展開をすると、いくつか（0 個かも）の非循環部分のあとで、循環部分が現れることが知られている。非循環部分の長さと循環部分の長さを求めるプログラムを書いてみる。

プログラム 2.2. shuuki-1.c

```
#include <stdio.h>
#define N 2000 /*展開の長さの上限*/
main(void)
{
  unsigned long state, bunbo;
  unsigned long state_seq[N];
  int i=0;
  int found = 0; /*見つかってないとき 0，見つかったら 1*/
  int exhaust = 0; /*配列を使い果たしたら 1*/
  printf("a ÷ n の周期を計算します。a,n を入力ください。\n");
  printf(" a=");
  scanf("%d",&state);
  printf(" n=");
  scanf("%d",&bunbo);
  printf("%d.",state / bunbo);
  state = state % bunbo;
  while (found==0 && exhaust==0) { /*見つからず、使い果たしてない*/
    int j;
    printf("%1d", (state * 10)/bunbo);
    /* 今までに、state が現れたことがあるか探す */
    for (j=0; (j<i) && (state_seq[j] != state) ; j++);
    if (j < i) { /* 見つかった */
      printf("非循環部%d 桁周期%d 桁\n", j, i-j);
      found = 1; /*見つかったというフラグを立てる*/
    }
    if (i>=N-1) { /* 用意した桁数が足らない */
      printf("\n%d 桁見ても循環しません\n", i);
      exhaust = 1; /*配列を使い果たしたというフラグを立てる*/
    }
    state_seq[i] = state;
    state = (10 * state) % bunbo;
    i++;
  }
}
```

コンパイルして実行してみる。

a ÷ n の周期を計算します。a,n を入力ください。
 a=355
 n=113
3.14159292035398230088495575221238938053097345132743362831858407079646017699115044247787610619469026548672566371681 非循環部 0 桁周期 112 桁

を得る。これは、小数点以下打ち出されたの出力のうち、非循環部が 0 桁あり、残りの（数えると１１３個ある）１１２桁が循環しているということを意味する。

先の 3 ÷ 52 の場合は

 0.057692307 非循環部 2 桁周期 6 桁

を得る。(7 から 7 までが循環している。)

プログラムの解説をする。まず、先のプログラム 2.1 を考えよう。state が、「今までに計算して現れた余りに一致した瞬間」から小数展開は循環する。今までに現れたかどうかをチェックするためには、過去の state の履歴を全部保存しておかなくてはならない。

そういったことをするのに適したデータ構造が、「配列 (アレイ、array)」である。main 関数の本体の 2 行目の

```
unsigned long state_seq[N];
```

が、配列を宣言する部分である。この宣言により、

$$\text{state\_seq[0]}, \text{state\_seq[1]}, \ldots, \text{state\_seq[N-1]}$$

の N 個の変数がメモリ上に確保される。そして、整数変数 i に対し、state_seq[i] は普通の変数のように代入したり、値を読み出したりすることができる。

もう一つ、はじめてみるのは while 文であろう。この文は、

```
while (<論理式>) <文>
```

という構造を持っている。まず < 論理式 > を計算し、それが偽であれば下に実行を移す。真であれば、右の < 文 > を実行し、その後で < 論理式 > を計算する。偽であれば下に実行を移し、真であれば < 文 > を実行する。

つまり、< 論理式 > が真である間は右の文を繰り返し実行し続ける。ループを回るように実行を繰り返すので、しばしば (for ループのように）while ループといわれる。

プログラム 2.2 では、

```
while (found == 0 && exhaust==0) {...}
```

という while 文がある。==は等しいかどうかを調べる演算子である。&&は、「かつ（AND)」を表す論理演算子である。結局、括弧の中身は「found が 0 でかつ exhaust が 0」ということになる。

found と exaust という二つの変数は最初 0 に設定されている。後で述べるように、もし「今まで記録しておいた state の中に、今計算した state が見つかった (found)」ら、found は 1 にセットされる。

もし、「記録してきた state の数が配列のサイズ N を超えてしまった（使い果たし、exhaust)」ら、exaust は 1 にセットされる。

このどちらかが起きるまで、while の次の {} で囲まれた部分が繰り返される。この部分をパート 1 と呼ぼう。

パート 1 では、state には直前に計算された余りが入っている。整数変数 i は 0 にセットされている。printf で、小数展開の次の桁を計算して出力する。その後で、「その前までの計算の中で、state が現れたことがあるかどうか」を計算する。

この while 文の中では、常に次の性質が成り立つようにプログラムは書かれている。

- `state_seq[0]` から `state_seq[i-1]` の中に、小数点以下 1 桁目から `i` 桁目を計算するときの直前の余りが格納されている。
- `state` には、`i+1` 桁目を計算する直前の余りが格納されている。

上の条件が満たされているかどうかは、

1. while 文に入る直前でこの性質が成り立つ
2. while 文の中で `i`, `state`, `state_seq[]` が書き換えられているたびに、この性質が成り立つ

の二点を確かめれば良い。

まず、while 文に入る直前では、$i = 0$ である。前半の条件は空な条件である。後半の条件は、それを成り立たせるために `state = state % bunbo` を実行している。)

その後、`state` や `state_seq`、`i` が書き換えられるのは最後の 3 つの文

```
state_seq[i] = state;
state = (10 * state) % bunbo;
i++;
```

のみであり、これを実行後も先の二つの条件が満たされていることは考えればわかる。(このように、ループ文の中で常に成り立つ命題のことをループ不変量 loop invariant という。この程度複雑なプログラムを書く場合、ループ不変量を何にするかを決めて書いておくことが不可欠であると、筆者は思う。)

これがわかれば、あとはそう難しくない。for 文

```
for (j=0; (j<i) && (state_seq[j] != state) ; j++);
```

では、j を 0 から 1 ずつ増やしていき、次のいずれかの条件がくずれたときにループを抜ける。

j<i または state_seq[j] != state

ここで、!=は「左右が違うときに真、等しいときに偽」という比較演算子である。

条件 j<i が崩れたということは、今まで計算した i 個の余りである state_seq[j], $j = 0, \ldots, i-1$ の中に、i+1 桁目を計算する直前の余りである state が現れなかったことになる。

条件 state_seq[j] != state が崩れたということは、今まで計算した中で、j+1 桁目計算の直前の余りに i+1 桁目計算の直前の余りが一致したことになる。このとき、j+1 桁目からと i+1 桁目からの出力は一致する。すなわち、非循環部分の長さは j で循環部の長さは i-j となる。

そこで、for 文を抜けた後、「一致する余りが見つかった」場合には非循環部、循環部の長さを出力して found を 1 にする。（このように、何かが成功したか否かを記録している変数をフラグ（旗）という。成功したら旗を立てる、ということである。）

もし、用意した配列を使い果たしても余りの重複がなければ、つまり i>=N-1 (すなわち i が N-1 以上) となってしまったら、そう出力して exhaust フラグを立てる。(正確には、次の行で使い果たす。)

どちらでもなければ、もう一桁計算し（先に見た 3 行)、ループを繰り返す。

以上で、プログラム 2.2 の解説を終わる。人の書いたプログラムを見るより、自分で書いてみることを進める。（この程度のプログラムでも、筆者は動くまでに数時間を要した。恥ずかしながら。）

と思ったが、プログラムの読解が難しいようなのでもう少し解説を試みる。

```
  state = state % bunbo;
  while (found==0 && exhaust==0) {
    /* 見つからず、使い果たしてない間 */
    int j;
/3  printf("%1d", (state * 10)/bunbo);
/1  /* 今までに、state が現れたことがあるか探す */
/1  for (j=0; (j<i) && (state_seq[j] != state) ; j++);
/1  if (j < i) { /* 見つかった */
/1    printf("非循環部%d 桁周期%d 桁\n", j, i-j);
```

```
/1    found = 1; /*見つかったというフラグを立てる*/
/1  }
/2  if (i>=N-1) { /* 用意した桁数が足らない */
/2    printf("\n%d 桁見ても循環しません\n", i);
/2    exhaust = 1; /*配列を使い果たしたというフラグを立てる*/
/2  }
/3  state_seq[i] = state;
/3  state = (10 * state) % bunbo;
/3  i++;
  }
```

であるが、最初の/1 の部分で書き換えられえる変数は found のみである。次の/2 の部分で書き換えられる変数は exhaust のみである。よって、これらの変数が書き換えられない間は while ループを回り続ける、すなわち/3 の部分を実行し続けるのである。するとこれは

```
/3  printf("%1d", (state * 10)/bunbo);
/4
/3  state_seq[i] = state;
/3  state = (10 * state) % bunbo;
/3  i++;
```

の 4 行を実行し続ける。このとき、このループにおける不変量として、/4 のところでは常に次がなりたつ。「state_seq[0] から state_seq[i-1] までに、今までの余りの列が入っている。正確には、state_seq[j] には小数点以下 j 桁目を計算する直前の余りが入っている。state には、i 桁目を計算する直前の余りが入っている。」

/4 のところに実際に入っている/1,/2 の部分では、これらの変数を変更しない。したがって、この性質は/1, /2 では常に保たれている。したがって、この性質を信じてプログラムを組んで良い。

/1 では、i 桁目計算直前の余りである state と、いままで計算した 1 から $i-1$ 桁目を求める直前の余りの中に一致したものがあるかないかを調べる。一致したものがあったら、非循環部の長さと周期を出力して found フラグを立てて終わり。/3 部分は実行されるが、while 文を抜けて終了する。

/2 では、配列を使い果たしていたら exhaust フラグを立てて終わり。/3 部分が実行されたとき、カウンタ i は N 未満でないとならない。そのため、配列使い果たし条件は i>=N でなく i>=N-1 となる。

2.3 関数呼び出し

上の「循環小数」の計算において、state がとる値は次の漸化式によって定まっている。

$$x_0 := a \mod n, \quad x_{j+1} = 10x_j \mod n.$$

ここに、 mod は割った余りをとるという二項演算子である。(割られる数が負の時には、余りを負で返す。という点がちょっと使いにくいことがある。)

より一般に、f を unsigned long 整数から unsigned long 整数への関数として、

$$x_0 := a, x_{j+1} := f(x_j)$$

により数列を生成することを考えよう。

C 言語で書けば、例えば次のようなものである。

プログラム 2.3. zenka-1.c

```
#include <stdio.h>
#define  BUNBO 7

unsigned long
next_state(unsigned long x)
{
 return((10 * x) % BUNBO);
}

main(void)
{
  unsigned long state, kosu;
  int i;
  printf("漸化式による数列を計算します。初期値 a を入力ください。\n");
  printf(" a=");
  scanf("%d",&state);
  printf("何個もとめますか？個数を入力ください。\n 個数=");
  scanf("%d",&kosu);
  for (i=0; i<kosu; i++) {
    printf("%d ", state);
    state = next_state(state);
  }
  printf("\n");
}
```

今までと違い、main() の上にもプログラムがある。結論から言うと、このプログラムは次のプログラムと等しい動作をする。

```
#include <stdio.h>
#define  BUNBO 7
main(void)
{
  unsigned long state, kosu;
  int i;
  printf("漸化式による数列を計算します。初期値 a を入力ください。\n");
  printf(" a=");
  scanf("%d",&state);
  printf("何個もとめますか？個数を入力ください。\n 個数=");
  scanf("%d",&kosu);
  for (i=0; i<kosu; i++) {
    printf("%d ", state);
    state = (10 * state) % BUNBO
  }
  printf("\n");
}
```

こちらは、すでに見た知識で読めるはずである。入力された数 a を 10 倍して BUNBO で割り、その余りを出力する。余りを 10 倍して BUNBO で割り、その余りを出力する。これを、入力した kosu 回繰り返す。

コンパイルして実行すると次のような出力を得るだろう。

```
漸化式による数列を計算します。初期値 a を入力ください。
 a=1
何個もとめますか？個数を入力ください。
 個数=20
1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3
```

このプログラムとプログラム 2.3 との違いは、上の方に next_state という「関数」が定義されていること、state への代入文が

```
    state = next_state(state);
```

に変わっていることである。

さて、

```
unsigned long
```

```
next_state(unsigned long x)
{
 return((10 * x) % BUNBO);
}
```

は「関数定義」というものである。

```
<型宣言 1>  <関数名>(<型宣言子 2> <変数名>)
<複文>
```

という形をしているのだが、平たく言うと「< 関数名 >(next_state に当たる) というのは、返す値を < 型宣言 1>(先頭の unsigned long に当たる) 型とする関数である。もらう値は、< 型宣言 2> (括弧のなかの unsigned long に当たる) 型である。< 変数名 >（x にあたる）なる変数を作ってそこにもらった値をコピーする。それから < 複文 >（{return((10 * x) % BUNBO);} にあたる）を実行する。」という動作をする。全然平たく言ってないが。

next_state の場合、例えば

```
state = next\_state(3);
```

なる文を実行するとしよう。state に代入する値を計算するため、右辺の next_state(3) を計算しようとする。すると、上で定義された next_state なる関数が呼び出される。よびだされると、まず x という unsigned long 変数を用意し、そこに 3 という値をコピーする。それから本体を実行する：

```
 return((10 * x) % BUNBO);
```

であるが、この return という命令はその後の括弧の中身を計算し、その値を next_state の返す値として返し、この関数から出て行くという命令である。

この場合、next_state(3) が呼ばれると 10*3 % BUNBO が実行されて 30 を 7 で割った余りである 2 が計算される。そして、その値が返される。

```
state = next\_state(3);
```

ならば、state には 2 が代入されることになる。

2.4 角谷の問題

角谷の問題というのは、いまだに未解決である次の予想のことである。

問題 2.4. x_0 を正整数として、漸化式

$$x_{j+1} = f(x_j)$$

を考える。ここに f は

$$f(x) = \begin{cases} x/2 & x \text{ が偶数} \\ 3x+1 & x \text{ が奇数} \end{cases}$$

で定義される。このとき、「x_0 がどんな正整数であっても、数列 x_j はいずれ 1 になる」というのが角谷予想（角谷コラッツ予想）である。

これを C 言語で実験する。まずはさきほどのコードのうち、next_state の関数定義を変えるだけでよい。

プログラム 2.5. kakutani-1.c

```
#include <stdio.h>

unsigned long
next_state(unsigned long x)
{
  if (x % 2 == 0) {return(x/2);}
  else {return(3*x+1);}
}

main(void)
{
  unsigned long state, kosu;
  int i;
  printf("漸化式による数列を計算します。初期値 a を入力ください。\n");
  printf(" a=");
  scanf("%d",&state);
  printf("何個もとめますか？個数を入力ください。\n 個数=");
  scanf("%d",&kosu);
  for (i=0; i<kosu; i++) {
    printf("%d ", state);
    state = next_state(state);
  }
  printf("\n");
}
```

next_state の定義しか変わっていないことに注意しておく。

```
if (x%2 == 0) {return(x/2);} else {return(3*x+1)}
```

では、x を 2 で割った余りが 0 なら x/2 を返し、さもなくば 3x+1 を返す、という関数を定義している。

コンパイルして実行してみる。次のような出力を得るだろう。

```
漸化式による数列を計算します。初期値 a を入力ください。
 a=7
何個もとめますか？個数を入力ください。
 個数=30
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4
```

問題 2.6. 角谷予想で、入力した数が 1 になるまで漸化式による計算をつづけて出力するプログラムを書け。

このようにプログラムで関数を用いるメリットは、多くある。最初に有用なのが、「あるまとまった操作」に一つ「名前」をつけることによるプログラムの読みやすさの向上であろう。そのプログラムが何を受け取って何を返すのかが良くわかる。（専門的な言葉でいうとプログラムのモジュール化という。）

3 再帰的呼び出し

3.1 階乗

N を入力して、その階乗 $N!$ を計算するプログラムを書こう。

プログラム 3.1. kaijo-1.c

```
#include <stdio.h>
main(void)
{
  long n, x;
  int i;
  printf("N の階乗を計算します。N を入力ください。\n");
  printf(" N=");
  scanf("%d",&n);
  x = 1;
  for (i=1; i<=n; i++) {
```

```
    x *= i;
  }
  printf(" %d!=%d\n",n,x);
}
```

for 文の中の、x *= i が実行されたあとにおけるループ不変性質は、

$$x = i!$$

である。そして、$i = n$ まではループの繰り返しを実行し、$i = n + 1$ では実行せず for 文を抜けるのであるから、x には $n!$ が格納される。コンパイルして実行すると次のような結果を得る。

```
N の階乗を計算します。N を入力ください。
 N=5
 5!=120
```

ちょっと工夫すれば、もう少し短いプログラムも書ける。

プログラム 3.2. kaijo-2.c

```
#include <stdio.h>
main(void)
{
  long n, x;
  printf("N の階乗を計算します。N を入力ください。\n");
  printf(" N=");
  scanf("%d",&n);
  printf(" %d!=",n);
  x = 1;
  for (; n>0; n--) {
    x *= n;
  }
  printf("%d\n",x);
}
```

このプログラムだと、x にまず 1 を入れ、次に n を掛け、(n-1) を掛け、と n を一つ減らしながら掛けて行き、n=1 まで掛けたらとまるようになっている。(しかし、このコードにするとループ不変性質が書きにくい。$x = N(N-1)\cdots(n+1)$ であるが。)

さて、階乗を計算するのに、「帰納法的」あるいは「再帰的」な方法を用いることもできる。再帰的呼び出しを使っても、上の簡単なプログラムに比べて何一つついいことがない。しかし、再帰的呼び出しを説明する一番簡単な例として、階乗が採用されることが多い。

再帰的呼び出し (recursive call) とは、$f(n)$ を計算するのに $f(n-1)$（または、そのほかのより小さい同種の計算）を用いて処理をするということである。

まずプログラムを見てみよう。

プログラム 3.3. kaijo-rec.c

```
#include <stdio.h>
long factorial(long n)
{
  if (n==0) return (1);
  else return(n*factorial(n-1));
}

main(void)
{
  long n, x;
  printf("N の階乗を計算します。N を入力ください。\n");
  printf(" N=");
  scanf("%d",&n);
  printf(" %d!=%d\n",n,factorial(n));
}
```

動作させたときの入出力はまったく同じなので省略する。このプログラムでは、factorial(n) という関数が定義されている。その内容は、n=0 であれば 1 を返し（数学では 0!=1 と決めてある）、さもなければ n∗factorial(n) を返す。

$$f(n) := \begin{cases} 1 & (n = 0) \\ n \times f(n-1) & (n \neq 0) \end{cases}$$

このプログラムは次のように働く。3 を入力したとしよう。main ルーチンから、関数 factorial(3) が呼ばれる。呼び出された factorial では n=3 は n==0 を満たさないので、3∗factorial(2) を計算しようとする。そのとき、factorial(2) を呼び出す。

結果、プログラムは

1. main() で、factorial(3) を計算しようと factorial を呼び出す。この呼び出された factorial（を計算する環境）を f-1 としよう。
2. f-1 の内部では、$3 \neq 0$ なので、3∗factorial(2) を計算しようと factorial を呼び出す。このとき呼び出される factorial では、変数 n は新たに作り出されて、f-1 において使われている n とは違う部分に格納場所を持つ（物知りの人は、スタック領域に格納場所を作ることを知っているかも）。
   ここで呼び出された factorial(2) は、f-1 とは計算方法は同じだが、変数などは名前は同じだが違う格納場所を持っている。この呼び出された関数を計算する環境を f-2 と書く。
3. f-2 の内部では、$2 \neq 0$ なので、2∗factorial(1) を計算しようとする。このとき、factorial(1) を計算するためにあらたに変数 n が作り出される。この計算を行う環境を f-3 と書く。
4. f-3 では 1∗factorial(0) を計算しようとする。factorial(0) を計算するためにまたあらたな変数 n が作り出される。この計算を行う環境を f-4 と書こう。
5. f-4 においては、factorial(0) が呼ばれるのであるから n==0 が成立している。(ここで、左辺の n は環境 f-4 における n であることに注意しておく。)
   ここではじめて、return(1) が実行される。すなわち、f-4 において計算された factorial(0) の値は 1 となり、値が返される。
6. 値が f-3 に戻り、f-3 の環境での計算があとを引き継ぐ。f-3 の環境では、n=1 である。n∗factorial(0) を計算しようとしていたのだが、f-4 の計算により 1∗1 を計算することになったのでその値は 1。return(1) となり、1 が factorial(1) の計算結果として f-2 に戻される。
7. 値が f-2 に戻る。f-2 の環境では、n=2 である。n∗factorial(1) を計算しようとしていたので、2∗1 を計算してその結果である 2 を factorial(2) の値として f-1 の環境に返す。
8. 値が f-1 に戻る。f-1 の環境では、n=3 である。n∗factorial(2) を計算しようとしていたので、3∗2 である 6 が求まる。それを factorial(3) の値として main に返す。
9. main では、factorial(3) の値として 6 が戻ってくるので、その値を出力して終わる。

「環境」と呼んでいるものは何か。それは、「それぞれの変数名がさしているメモリ空間のアドレス」に他ならない。... と説明してもわかりませんよね。

例え話で話すならば次のような感じなのである。N=2 が入力されたとしよう。

- main という名前の作業机で作業をしている人 A が、「おおい、誰か factorial(2) を計算してくれ」と叫ぶ。
- すると別の人 B と、factorial という名の作業机が魔法のように出現し、そこに n という名のノートが出現する。そのノートには 2 と書かれている。

- facgtorial という作業机には、「ノート n に書かれている数 □ に対して、□ が 0 なら 1 を返し、そうでなければ □×factorial(□) を計算して、その値を返せ」と書いてある。
- そこでその人 B は、ノート n の中身を見て □ = 2 だから、2 × factorial(1) を計算しようとして、「おおい、誰か factorial(1) を計算してくれ」と叫ぶ。
- するとまた別の人 C と、factorial という名の作業机が魔法のように出現し、そこに n という名のノートが出現する。そのノートには 1 と書かれている。
- factorial という作業机には、「ノート n に書かれている数 □ に対して、□ が 0 なら 1 を返し、そうでなければ □×factorial(□) を計算して、その値を返せ」と書いてある。
- そこでその人 C は、ノート n の中身を見て □ = 1 だから、1 × factorial(0) を計算しようとして、「おおい、誰か factorial(2) を計算してくれ」と叫ぶ。
- するとまた別の人 D と、factorial という名の作業机が魔法のように出現し、そこに n という名のノートが出現する。そのノートには 0 と書かれている。
- factorial という作業机には、「ノート n に書かれている数 □ に対して、□ が 0 なら 1 を返し、そうでなければ □×factorial(□) を計算して、その値を返せ」と書いてある。
- そこでその人 D は、ノート n の中身を見て □ = 0 だから、「factorial(0) が計算できました、1 ですよ」と、自分を呼んだ C さんに言って、消滅する。ノートも机も消滅する。
- C さんは、「ははあ、factorial(0) は 1 ですか。では、私の返す値は □×factorial(0)= 1 × 1 = 1 ですよ。」と、自分を呼んだ B さんに言って、消滅する。ノートも机も消滅する。
- B さんは、「ははあ、factorial(1) は 1 ですか。では、私の返す値は □×factorial(1)= 2 × 1 = 2 ですよ。」と、自分を呼んだ A さんに言って、消滅する。ノートも机も消滅する。
- A さんは、「ほう、factorial(2) は 2 ですか。では、作業机 main に書いてあるとおり、その 2 を出力しましょう。」と言って出力して、消滅する。机も消滅する。

3.2 置換とバックトラック

次のプログラムは、n を入力したとき、$0, 1, \ldots, n-1$ の n 個の文字の置換（permutation, 高校では順列といった）を全て列挙し出力するものである。ただし、n は１０までの正整数。

プログラム 3.4. perm-1.c

```
#include <stdio.h>
#define N 10

void perm(int start, int n, int x[])
{
  int i;
  int y[N];
  for (i=0; i<N; i++) y[i]=x[i];
  if (start == n) {
    for (i=0; i<n; i++)
      printf("%d",y[i]);
    printf("\n");
    return;
  }
  for (i=0; i<n-start; i++) {
    int temporary;
    /* y[start] と y[start + i] を入れ替える*/
    temporary = y[start];
    y[start] = y[start + i];
    y[start + i] = temporary;
    perm(start+1, n, y);
  }
  return;
}

main(void)
{
  int i, n, x[N];
  printf("n 個の元の置換を全て求めます。n を入力ください。\n");
  printf(" n=");
  scanf("%d",&n);
```

```
  for (i=0;i<n;i++) x[i]=i;
  perm(0,n,x);
}
```

関数 perm(int start, int n, int x[]) がやることは、「整数 start, n, 配列 x[] を受け取って、x[start] から x[n-1] の部分の配列の中身の置換を全て生成すること」である。length が 0 の時は、配列の中身を出力する。そうでないときは、x[start] の中身を x[start] から x[n-1] のそれぞれと入れ替えつつ、残りの x[start+1] から x[n-1] の置換を perm(start+1,n,x[]) を用いて全て生成する。

ちょっとプログラムを変更すると、「隣り合う二数の差が ±1 にならないような置換を全部生成する」次のコードが得られる。

プログラム 3.5. perm-non-pm.c

```
#include <stdio.h>
#define N 10

void perm_non_pm(int start, int n, int x[])
{
  int i;
  int y[N];
  for (i=0; i<N; i++) y[i]=x[i];
  if (start == n) {
    for (i=0; i<n; i++)
      printf("%d",y[i]);
    printf("\n");
    return;
  }
  for (i=0; i<n-start; i++) {
    int temporary;
    /* y[start] と y[start + i] を入れ替える*/
    temporary = y[start];
    y[start] = y[start + i];
    y[start + i] = temporary;
    /* もし その結果、隣合う数の差が 1 ならその先探さない*/
    if ((start == 0)
        || ((y[start-1]-y[start]!=1)
            && (y[start-1]-y[start]!=-1)))
```

```
      perm_non_pm(start+1, n, y);
  }
  return;
}

main(void)
{
  int i, n, x[N];
  printf("隣り合う数が2以上離れているようなn個の元の置換を全て求めます。\n");
  printf("nを入力ください。\n");
  printf(" n=");
  scanf("%d",&n);
  for (i=0;i<n;i++) x[i]=i;
  perm_non_pm(0,n,x);
}
```

コンパイルして実行し、5を入力すると次のような出力を得る。

```
隣り合う数が2以上離れているようなn個の元の置換を全て求めます。
nを入力ください。
 n=5
02413
03142
13024
13042
14203
20314
20413
24031
24130
30241
31402
31420
41302
42031
```

このように、「ある条件を満たすような解を全て探索する」という問題をコンピュータで解くことは多い。そして、そのようなときには、「一手進めてみて、

残りの全ての可能性を再帰呼び出して探す」というプログラムを書かなくてはならないことが多い。このようなタイプの探索アルゴリズムをバックトラックという。

参考文献

[1] 高木貞治著 解析概論 岩波書店

[2] 野崎明弘著 π の話（岩波科学の本）岩波書店、1998 年再版

[3] Matsumoto, M. and Nishimura, T. “Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator” ACM Trans. on Modeling and Computer Simulation **8** (1998), 3–30.