アルゴリズム入門　＃4

地引　昌弘

2024.10.24

はじめに

一般に、アルゴリズムをコンピュータ上で実行する場合、コンピュータで処理し易い手順に修正したプログラムを作成し、これを実行します。「コンピュータで処理し易い」手順の一つとして、「“少しだけ or 一つだけ実行する” を何度も繰り返し、その結果を集計する」手順があります[1]。今回は、この考えに沿って、前回学んだ繰り返しの制御構造を利用した数値計算の手法について説明します。

1　前回の演習問題の解説

1.1　演習 3-1a — 枝分かれの復習

演習 3-1a は、前回の資料にある例題とほとんど同じです (0 と比較するか、b と比較するかの違い)。まずは擬似コードを見てみましょう:

- max: 数 a、b の大きい方を返す
- 　もし a > b であれば、
- 　　$result \leftarrow a$
- 　そうでなければ、
- 　　$result \leftarrow b$
- 　(枝分かれ終わり)
- 　$result$ を返す。

Python では次の通り:

```
def max(a, b):
  if a > b:
    result = a
  else:
    result = b
  return(result)
```

これも、次のような「別解」があり得ます:

- max2: 数 a、b の大きい方を返す
- 　$result \leftarrow a$
- 　もし $b > result$ であれば、
- 　　$result \leftarrow b$
- 　(枝分かれ終わり)
- 　$result$ を返す。

[1] この手法を取り上げる度に、「少しだけ or 一つだけ…」と言うのは面倒なので、取り敢えず本講義では、これを “逐次細分最適化法” と呼ぶことにします (これは一般的な名称ではありません/本講義内だけの名称です)。

これの Python 版は次の通り:

```
def max2(a, b):
  result = a
  if b > result:
    result = b
  return(result)
```

これらはどちらが正解ということはありません。皆さんは、どちらが好みですか?

ところで、「2 数が等しい場合はどうするのか」について、皆さんの中には迷った人がいると思います。問題には「異なる数」と書いてあるので、今回は考えなくてもよいのですが、仮にそれが書いていなかったとします。そうなると、等しい場合について何らかの指示が本来あるべきですよね。例えば、次のような方針が考えられます。

- 「等しい場合はその等しい数を返す」
- 「等しい場合は何が返るかは分からない」
- 「等しい数を渡してはならない」

上から二つ目までの場合は、それほど問題にはならないでしょう (2 番目では何が返ってもよいので、等しい数でもよい)。最後の場合はどうでしょう。これも次の考え方があり得ます。

(a) 「渡してはならない」以上、渡されることはないのだから、特別な処理は必要ない

(b) 「渡してはならない」値が渡されたのだから、エラーを表示するなどして警告するべき

どちらにも (互いに裏返しの) 利点と弱点があります。(a) の方は、簡潔で短いプログラムを作ることができます。(b) の方は、起きるべきでないことが起きていることが分かるので、対処が必要な場合には有用です。現実世界の多くでは、発注者 (例えば地引) の注文を受けてプログラムを作成することになります。よって、解釈が曖昧な場合は、発注者に仕様 (プログラムの振舞い) を確認することになります。特に大きなプログラムでは、複数人が開発したプログラムを結合して全体を構成するため、互いに仕様の解釈が異なった場合は正しい動作をしないだけでなく、不良個所の特定が大変難しくなります (各自、自分の解釈が正しいと考えているため)。

1.2 演習 3-1b — 枝分かれの入れ子

演習 3-1b はもう少し複雑です。まず考えつくのは、a と b の大きい方はどちらかを判断し、それぞれの場合について、それを c と比べるというものでしょう:

- max3: 数 a、b、c で最大のものを返す
- 　もし $a > b$ であれば、
- 　　もし $a > c$ であれば、
- 　　$result \leftarrow a$
- 　　そうでなければ、
- 　　$result \leftarrow c$
- 　　(枝分かれ終わり)
- 　そうでなければ、
- 　　もし $b > c$ であれば、
- 　　$result \leftarrow b$
- 　　そうでなければ、
- 　　$result \leftarrow c$
- 　　(枝分かれ終わり)
- 　(枝分かれ終わり)
- 　$result$ を返す。

かなり大変ですね。これを Python にしたものは次の通り:

```
def max3(a, b, c):
  if a > b:
    if a > c:
      result = a
    else:
      result = c
  else:
    if b > c:
      result = b
    else:
      result = c
  return(result)
```

このプログラムを見ると、インデント (字下げ) の効果は一目瞭然ですね。これがないと、どの `if` とどの `elif` や `else` が対応しているかを見極めるのは、少々面倒です。これまで何度も述べて来たように、多くの言語では “{” と “}” や “begin” と “end” によりブロックの境界を示しますが、これだけでは、どのブロックがどのヘッダに属するかを理解するのは無理なので、インデントも併用します。Python のデザイン哲学は、文法を極力単純化することで、プログラムの可読性 · 作業性 · 信頼性を高めることを目指しており、begin · end とインデントを両方使うなら、どちらか一つ必須な方で十分という考えにより、このような仕様になっています。ところで、先の別解から発展させるとどうなるでしょう?

- max3a: 数 a、b、c で最大のものを返す
- result ← a
- もし $b > result$ であれば、
- $result \leftarrow b$
- (枝分かれ終わり)
- もし $c > result$ であれば、
- $result \leftarrow c$
- (枝分かれ終わり)
- $result$ を返す。

Python では次の通り (今度はどちらが好みですか?):

```
def max3a(a, b, c):
  result = a
  if b > result:
      result = b
  if c > result:
      result = c
  return(result)
```

一般に、枝分かれ (if) の中に枝分かれを入れるより、枝分かれを並べるだけで済ませられるならば、その方が制御構造を理解し易い場合が多いです。因みに、上の方法では、入力の数 N が何個あってもプログラム自体は簡単に拡張できますね。

実は、さらなる別解もあります。それは、前に作成した `max2` を利用するというものです。

```
def max3b(a, b, c):
  return(max2(a, max2(b, c))
```

このように、一度作って完成したプログラムを、後から別のプログラムを作る時の「部品」として使う、という考え方は大変重要です。是非、覚えておいて下さい。

1.3 演習 3-1c — 多方向の枝分かれ

演習 3-1c では、3 種類の結果を表示しなければならないため、必ず三つに枝分かれします。よって、if の中にまた if が入るのはやむを得ないでしょう。Python コードを見てみましょう:

```
def sign1(x):
  if x > 0:
    return("positive.")
  else:
    if x < 0:
      return("negative.")
    else:
      return("zero.")
```

このような「複数の条件判断」はよく使うので、`if` の入れ子にしなくても書けるような構文が用意されています。具体的には、`if` 文の後に、「`elif` 条件: 動作」という分岐を何回でも入れられるようになっています[2]。それを使うと次のようになります:

```
def sign2(x):
  if x > 0:
    return("positive.")
  elif x < 0:
    return("negative.")
  else:
    return("zero.")
```

擬似コードだと次のようになります:

- 実数 x を入力する。
- もし $x > 0$ ならば、
- 　　「positive.」を返す。
- そうでなくて $x < 0$ ならば、
- 　　「negative.」を返す。
- そうでなければ、
- 　　「zero.」を返す。
- (枝分かれ終わり)

「そうでなくて〜ならば、」は何回現われても構いません。また、そのどれもが成り立たない場合は、最終的に「そうでなければ」に至りますが、この部分は不要なら無くても構いません[3]。これを、先ほどの最大値の問題に適用してみましょう。複合条件を使えば「`a > b and a > c`」なら `a` が最大だと分かりますから、`elif` を用いることにより、次のような 3 方向枝分かれで書くこともできます (変数を使わず、その場で値を返すスタイルにしてみました):

```
def max3c(a, b, c):
  if (a > b) and (a > c):
    return(a)
  elif b > c:
    return(b)
  else:
    return(c)
```

[2] “elseif” ではなく “elif” なので、注意して下さい。

[3] とは言え、あらゆる条件を事前に明記することは、現実的には難しい (誤りの温床になりやすい) ので、多くの場合は「そうでなければ」· “else” を追記します

このプログラムでは、最初の枝分かれにある条件が二つの条件式より構成されています。複数の条件式がある場合、どんな条件式が書かれているか一見するだけでは分かりにくい、条件式間に優先度を付けたい[4]といった理由より、条件式を括弧で括る書き方がよく用いられます。上のプログラムも、そのようか書き方に沿って書いてあります。

最後に、例えば今回取り上げたような、与えられた N 個の数値から最大値を求める問題において、単純に枝分かれを使うだけでは、N が 4、5 と増えてくると条件内の比較演算が大幅に増えてしまいます (一般に最大値の決定では N^2 に比例して増えます)。N の個数が多くなれば、別の比較方法 (アルゴリズム) を検討する必要がありそうです (このテーマは、後で取り上げます)。

1.4 演習 3-2 — 誤差の補正

この演習では、これまで学んだ誤差の知識を全て活用すると共に、枝分かれの制御構造を用いて数値計算における誤差の影響を可能な限り排除し、計算精度を向上させる具体的な事例を取り上げました。数学的には、2 次方程式 $ax^2+bx+c=0$ の解は、$x=(-b\pm\sqrt{D})/2a,\ D=b^2-4ac$ より求められますが、これをそのままプログラムにしただけでは、2 次方程式の種類によっては、得られた解に誤差の影響が見られます (参考までに、2 次方程式の解を求める素直なプログラムを下記に再掲しておきます)。

```
def fa(a, b, c):
  D = b**2 - 4.0*a*c
  x1 = (-b + math.sqrt(D)) / (2.0*a)
  x2 = (-b - math.sqrt(D)) / (2.0*a)
  return(x1, x2)
```

これは、“数学による計算 $\rightarrow$ プログラムによる計算” のあらゆる場合に現れます。そこで、プログラムによる計算では、誤差の影響を可能な限り排除して、計算精度を向上させる手段がないかどうかを**常に考える**必要があります。一般に、丸め誤差はある意味、実数そのものが持ってしまう誤差とも考えられるので、これを減らすことは難しいです。これに対し、桁落ち誤差や情報落ち誤差は計算に伴い発生する誤差なので、計算の順番を変えるなどの工夫により、減らせる可能性があります。その考え方は前回のヒントに述べてあるので、再度確認して身に付けて下さい (これから先も使う基本的な考え方です)。

さて、2 次方程式の解を求めるプログラムにおいて、どこに着眼し、どのように修正するかの方針については、これも前回のヒントで述べました。具体的には、解の公式にある $-b$ と $\sqrt{D}$ の足し算 · 引き算のうち、「足し算」により得られる解を利用し、もう一方は解と係数の関係を用いて求めようというものでした。2 解を α,β とすると、解と係数の関係のうち引き算の可能性がない方は $\alpha\beta=\frac{c}{a}$ です。もう一方は、a,b,β の値により、引き算となる可能性があります (おっと、これは大ヒントですね)。

最後は、fa 関数内にある x1, x2 のうち、どちらが「引き算」であるかを考えることになります。一見すると、x1 の方が「引き算」になっているように見えます。x2 については、`-(b + math.sqrt(D))` より、「足し算」+「符号反転」となっているように見えます。しかし、これは `b` と `math.sqrt(D)` が正数であることが前提です。`math.sqrt(D)` が正数であることは明らかですが、(上でも述べましたが) `b` の正負は 2 次方程式の種類によって変わります。つまり、x1, x2 のどちらも、`b` の正負に応じて「引き算」を使うわけです。以上の考察をもとに、fa 関数を拡張して誤差を可能な限り取り除いた fb 関数のプログラムを示します:

```
import math

def fb(a, b, c):
  D = b**2 - 4.0*a*c
  x1 = (-b + math.sqrt(D)) / (2.0*a)
  x2 = (-b - math.sqrt(D)) / (2.0*a)
  if b > 0: x1 = c / (a*x2)
  else:     x2 = c / (a*x1)
  return(x1, x2)
```

[4]例えば、条件式 1 が成り立ちかつ、条件式 2 あるいは条件式 3 が成り立つ場合は、“条件式 1 and 条件式 2 or 条件式 3” ではなく、“条件式 1 and (条件式 2 or 条件式 3)” と書く必要があります。

この fb 関数より 2 次方程式 $x^2 - 100x + 1 = 0$ の解を求め、check 関数により検査した結果は、下記の通りです。

```
>>> x1, x2 = fb(1, -100, 1)
>>> x1
99.98999899979995
>>> x2
0.010001000200050014
>>> check(1, -100, 1, x1, x2)
(0.0, 0.0)
```

x1 については、fa 関数と fb 関数で値に変化はありませんが、x2 については変化があります (前回の資料にある値と比べてみて下さい)。この結果を見る限り、fb 関数で得られた解については、元の式に入れて丸めるときっちり 0 になるため (つまり、check 関数では誤差を検出できなかったため)、システムの限界まで精度を向上できていると言えます。

1.5 演習 3-3 — 計算の繰り返し

演習 3-3 は、testdiv2 関数を参考に計算の繰り返しを使って、このシステムで扱える最大数を調べるプログラムを作ってみようというものでした。testdiv2 関数は、最小数を調べるため、与えられた値を繰り返し 2 で割って行きましたが、今回は最大値を求めるため、与えられた値を繰り返し 2 倍して行きましょう。具体的なプログラムは、下記の通りです:

```
import math

def testdouble(x):
  while x != math.inf:
    print(x)
    x = x * 2.0
```

testdouble 関数に 1.0 を渡して実行した結果は、こんな感じになります:

```
>>> testdouble(1.0)
...
1.1235582092889474e+307
2.247116418577895e+307
4.49423283715579e+307
8.98846567431158e+307
>>>
```

この結果より、ここで使われている浮動小数点表現では、最も大きい数というのはおよそ「10^{307}」くらいであることが分かります。

1.6 演習 3-4a — 条件の組み合わせ

演習 3-4a は、繰り返しと枝分かれの基本的な組み合わせです。まずは、2 の倍数と 3 の倍数以外を打ち出す Python コードを示します:

```
def fizz2a(n):
  i = 0
  while i <= n - 1:
    if (i % 2 != 0) and (i % 3 != 0):
      print(i)
    i = i + 1
```

この例は、「2 の倍数でなく、3 の倍数でもないもの」を選ぶ条件を一つにまとめてあります。別案として、条件を簡素化して「素直に」枝分かれさせ、打ち出さない場合は (つまり、2 の倍数あるいは 3 の倍数の場合は) 「何もしない」という案もあります:

```
def fizz2b(n):
  i = 0
  while i <= n - 1:
    if i % 2 == 0:   pass
    elif i % 3 == 0: pass
    else:            print(i)
    i = i + 1
```

但し、Python の文法では、複合文のブロック部分にはコメント以外の文を必ず書かないといけないので、「何もしない」ことを示す “pass” 命令を入れてあります。参考までに、Ruby では、「何も書かない」ことにより「何もしない」ことを示すので、ブロック部分は空欄にします (しかしながら、「何も書いてない」のは不安なので、通常は、「何もしない」というコメントを書きます)。

1.7 演習 3-4b — 条件の排他性

次に演習 3-4b ですが、これは elif の連鎖を用いて、「3 の倍数」「5 の倍数」「3 と 5 の公倍数 (15 の倍数)」「それ以外」の四つに場合分けするのが一番素直です:

```
def fizzbuzz1(n):
  i = 0
  while i <= n - 1:
    if i % 15 == 0:  print("fizzbuzz")
    elif i % 3 == 0: print("fizz")
    elif i % 5 == 0: print("buzz")
    else:            print(i)
    i = i + 1
```

なぜ、15 の倍数を最初に調べているのか分かりますか。elif の連鎖では上から順に条件を調べて行くので、先に「3 の倍数」や「5 の倍数」を調べてしまうと、「15 の倍数」は 3 や 5 の倍数でもあることから、これらの条件が先に成立してしまいます。そのため、「15 の倍数」の場合は、必ず「3 の倍数」あるいは「5 の倍数」の枝が選ばれてしまい、「15 の倍数」だけの枝に決して来なくなってしまうからです。

条件が複数ある場合は、これらの特徴を考慮した場合分けも考えられます。今回の条件は、「3 の倍数なら fizz」「5 の倍数なら buzz」「両者の公倍数なら fizzbuzz」となっているので、この関係をうまく利用したプログラムも作れそうです。Python の `print` 関数は、文字列や数値を表示した後に必ず改行が入ります。今回は、fizz と buzz をくっ付けて 1 行に出力したいので、改行なしを示す「`end=""`」オプションを付けています:

```
def fizzbuzz2(n):
  i = 0
  while i <= n - 1:
    num = i
    if i % 3 == 0:
      print("fizz", end="")
      num = ""
    if i % 5 == 0:
      print("buzz", end="")
      num = ""
    print(num)
    i = i + 1
```

このプログラムは、まずは変数 num に表示する数を入れておき、3 の倍数なら fizz、5 の倍数なら buzz を表示すると共に、num には空白の文字列を入れ直します。15 の倍数だった場合は、3 の倍数として fizz、5 の倍数として buzz を表示しているので、結果として fizzbuzz が表示されます。3 や 5 の倍数でない場合は、num にカウンタ i が入っているので、print(num) によりその数が出力されます (3 や 5 の倍数である場合は、num が空文字になっているはずなので、何も出力されません)。ところで、このコードは二つの if が排他的な関係になっておらず、条件により、どちらか片方が処理される/両方とも処理される/ 両方とも処理されない、という関係にあります。また、数を出力するための変数 num の利用も、数を入れたり、空文字で消したりと変則的です。そのため、先に示した 4 方向に枝分かれするコードと比べて、流れを追うのが難しいように見えます。どちらが良いかは、条件の種類により一概に言えませんが、このようなプログラムを作る場合は、やはりコメントを追記して、制御の流れや変数の使い方を明示 · 確認しておく方が良いですね。

1.8 演習 3-4c — フラグを用いた条件

最後は演習 3-4c ですが、ヒントにもあるように、要は各桁の数字が 3 かどうかを調べるわけです。但し、33 が与えられても hoge の出力は 1 回だけ、3 が入っていない場合は数値を出力、といった対応が必要で、これを比較演算子による論理演算の組み合わせだけで行なうことは、少々面倒です。そこで、フラグ (詳細は第 3 回資料の 8, 10 ページを参照) を導入し、例えば、各桁の数字に一つでも 3 があればフラグを True にして以後の制御に利用する、という処理を考えることにします。

まずは、「3 が付く数」とはどう扱えばよいかを考えましょう。1 桁目が 3 の場合とは、10 で割った “余り” が 3 となる場合です。そして、2 桁目が 3 の場合とは、10 で割った “結果” が 3 となる場合ですが、1 桁目と 2 桁目で扱いが異なるのは嫌ですね。そこで、もう少し考えてみましょう。実は、数を 10 で割るということは、その数を 1 桁小さくすることに該当します。つまり、2 桁目が 3 の場合に、これを 10 で割るということは、1 桁目が 3 になるということです。これより、2 桁目が 3 かどうかを調べることは、まず 10 で割り (1 桁小さくし)、もう一度 10 で割って余りを調べることになります。後は、これを全桁数分行なえばよいのですが、与えられた数が何桁あるかは分かりません。よって、10 で割り続けて行くことになりますが、「3 を発見したら “hoge” を出力して途中で抜ける」、「(最後まで) 発見できなかったら元の数字を出力する」必要があります。この制御を、比較演算子による論理演算の組み合わせだけで行なうことは少々面倒なので、識別用の変数 found を導入します。found には、まずは「3 が付いていない」ことを表わす False を入れておきます。そして 3 を発見したら、「3 が付いている」ことを表わす True に入れ替えて、最後に結果を見ます。このような使い方をする変数を**旗** (Flag) と呼びます。最初は「旗」が降りていて、後で見ると「旗」が立っていたとすれば、誰が立てたかは分からないまでも、少なくとも誰かが旗を立てたことは確実なので、調べたい事象が発生したことが分かります (これとは逆に、“旗を降ろす” という使い方もあります)。では、Python のプログラムを見てみましょう:

```
def fizz3b(n):
  i = 0
  while i <= n - 1:
    found = False
    j = i
    while 1 <= j:                 # 未調査の桁がある間は調べ続ける。
      if j % 10 == 3:             # 1 桁目は 3 か?
        print('hoge')
        found = True              # 3 を発見したことを記録し、break でループを抜ける。
        break
      j = int(j / 10)             # 1 桁小さくするために 10 で割る。
    if found == False:            # while ループを抜けてきたが、最終的に 3 はあったか?
      print(i)
    i = i + 1
```

調査対象の数字 (つまり、i) に 3 が入っているかどうかを調べるだけならば、これを 10 で割って行けばよいのですが、3 が入っていない場合は、元の数字を表示する必要があります。そこで、i のコピー j を用意し、j を調べて行きます (フラグの利用に劣らず、このような配慮もお題の一つでした)。内側の while ループでは、「j の 1 桁目を調べる」+「j を

1 桁小さくする」処理を繰り返しています。繰り返しの条件にある “1 <= j” に注意して下さい。10 で割り続けた結果、1 より小さくなった場合は全桁数を調べたことになるので、次の数値へ移る必要があるのです。

2 数値積分

これまでの知識をもとに、いよいよアルゴリズムをコンピュータ上で実行する代表的な手法の一つを取り上げてみます。具体的には本資料の冒頭でも述べましたが、「“少しだけ or 一つだけ実行する” を何度も繰り返し、その結果を集計する」という手法です[5]。以下では、**数値積分** (Numerical Integration — 定積分の値を数値計算により求めること) を題材に、この手法について少し検討してみましょう。

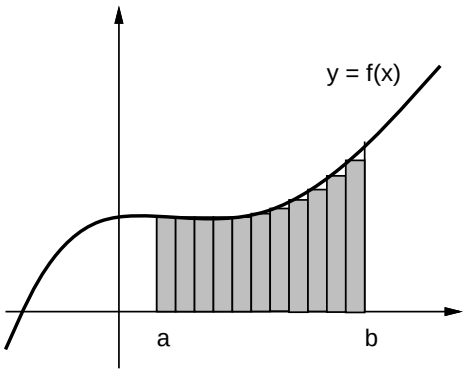

図 1: 数値積分の原理

参考: 関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分というのは、図 1 のように関数のグラフを描いたとして、区間 $[a, b]$ の範囲における関数の下側の面積です。そこで、図 1 にあるように、その部分を多数の細長い長方形に区分けしてその面積を合計すれば、知りたい面積の値、つまり定積分の値が求まることになります。各長方形の幅は区間 $[a, b]$ を n 等分した値であり (これを dx とする)、高さは $f(x)$ の値なので、その面積を計算するのは簡単です。これが皆さんよく御存じの高校数学で習うリーマン和ですが、なぜこのような話をしたかと言うと、コンピュータ上にリーマン和を求めるプログラムを作成することで、$f(x)$ の不定積分が簡単に求められないような場合であっても、定積分を近似計算できるようになるからです (数式の形で一般的に問題の解を求めることを**解析的** (Analytical) に解くと言い、数値計算により特定の問題の近似解を求めることを**数値的** (Numerical) に解くと言います)。不定積分のみならず微分方程式などでも、簡単に解を求められないが、その現象については具体的に捉えたい事例が数多く存在します。コンピュータによる数値的な解法の進化は、工学を始めとした様々な分野の革新的な発展を促しました。

今回は「正しい」値が求まるかどうかをチェックしたいので、簡単な関数 $y = x^2$ で試してみます。不定積分は $\frac{1}{3}x^3$ なので、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ となります。例えば、$[1, 10]$ ならば $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ です。では、アルゴリズムを作ってみましょう:

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- $dx \leftarrow \frac{b-a}{n}$
- $s \leftarrow 0$
- $x \leftarrow a$
- $x < b$ が成り立つ間、繰り返し:
- $y \leftarrow x^2$ # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$
- $x \leftarrow x + dx$
- (繰り返し終わり)
- s を返す。

[5] これも本資料の冒頭で述べましたが、この手法を “逐次細分最適化法” と呼ぶことにします (これは一般的な名称ではありません/本講義内だけの名称です)。

まずは、x に a を格納しておき、繰り返しの中で $x \leftarrow x + dx$、つまり x に dx を足した値を作ります。それを x に入れ直すことで、x を徐々に (dx 刻みで) 動かしていき、b まで来たら繰り返しを終わります。このように、繰り返しでは「こういう条件で変数を動かしていき、こうなったら終わる」という考え方が必要なのです。面積の方は、s を最初 0 にしておき、繰り返しの中で細長い長方形の面積を繰り返し加えていくことで、合計を求めます。

では、Python のプログラムを示しましょう。「#」の右側に書かれている部分は**注記**ないし**コメント** (Comment) と呼ばれ、Python ではこの書き方でプログラム中に覚え書きを入れておくことができます。コードの意味が分かりづらい (何のためにこのような計算をしているのか読み取りにくい) 箇所には、必ずその意図を注記しておくようにして下さい (**重要:** “コメント” のないプログラムは素人が書いたプログラムです)。また、一時的に命令を実行しないようにするために (言い換えれば、その命令が存在しないようにするために)、コメントを使う場合もあります。これを**コメント アウト** (Comment Out) と呼びます。この例でも後で使うコードをコメント アウトしてあります:

```
def integ1(a, b, n):
  dx = (b - a) / n
  s = 0.0
  x = a
# count = 0
  while x < b:
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
    x = x + dx
#   count = count + 1
#   print("count=%d x=%.20f" % ((count), (x)))
  return(s)
```

やっていることは、先の擬似コードそのままだと分かるはずです。さて、333 が求まるでしょうか? 実行させてみます:

```
>>> integ1(1.0, 10.0, 100)
337.5571499999994                ← ふーん?
>>> integ1(1.0, 10.0, 1000)
332.55462150000733               ← 小さい
>>> integ1(1.0, 10.0, 10000)
333.04545121491196               ← 大きい…
```

何だか変ですね。そこで、繰り返しの回数が幾つになっているかをチェックすることにして、上の行頭の「#」を削って動かし直してみました[6]:

```
>>> integ1(1.0, 10.0, 100)
...
count=98 x=9.819999999999990     ← x の値に誤差がある
count=99 x=9.909999999999989
count=100 x=9.999999999999989
count=101 x=10.089999999999989   ← えっ、101 回目…
337.5571499999994                ← 101 回目の影響で大きくなっている
```

理由が分かりました。区間数が 100 個なのに、長方形を 1 個余計に加えてしまい、値が大き過ぎたわけです。何故こんなことが起きるのでしょうか? それは「$x \leftarrow x + dx$ で x を増やして行き、b になったら止める」というアルゴリズムに問題があるのです。そもそもコンピュータでの浮動小数点計算は近似値の計算なので、dx を区間長の $\frac{1}{100}$ にしたとしても、そこに (丸め) 誤差があります。このため 100 回足しても僅かに b より小さい場合があり、その時は余分に繰り返しを実行してしまいます (2 次方程式の解を求める演習でも見た通り、このような部分に、アルゴリズムに素直に従って “実際に計算するプログラム作成の難しさ” が潜んでいたりします)。

[6] つまり、変数 `count` で回数を数えつつ、`x` を表示するようにするわけです。このように、コメント アウトしてあったコードを活かして (コメントの記号を削って) 動かすことを、「コメント アウトを外す」と言います。また、ここでは、`x` の値を細かく見たいので、print 関数の書式に “%.20f” を指定し、十進法浮動小数点数として 20 桁を強制的に表示させている点にも注意して下さい。

3 計数ループ

では、どうすれば良いでしょうか。繰り返し回数を 100 回と決めているので、回数を数える際は整数型で行ない[7]、それをもとに各回の x を計算するのが良さそうです。つまり、次のようなループを書くことになります (**カウンタ** (Counter) とは「数を数える」ために使う変数のことを言います):

```
i = 0              # i はカウンタ
while i < n:       # 「n 未満の間」繰り返し
  ...              # ここはループ内側（ブロック）の動作
  i = i + 1        # カウンタを 1 増やす
```

このように、指定した上限まで数を数えながら繰り返して行くような繰り返しを、**計数ループ** (Counting Loop) と呼びます。計数ループはプログラムで頻繁に使われるため、ほとんどのプログラミング言語は計数ループのための専用機能や構文を持っています (while 文でも計数ループは書けますが、専用の構文の方が書き易くて読み易いからです)。

Python では、計数ループ用の構文として **for 文** (For Statement) を用意しています[8]。これを使って上の while 文による計数ループと同等のものを書くと、次のようになります:

```
for i in range(n):
  ...
```

これは、カウンタ変数 `i` を 0 から始めて一つずつ増やしながら `n-1` まで繰り返していくループとなります。ここで、カウンタ変数 `i` の範囲は、$1 \sim n$ ではなく、$0 \sim n-1$ である点に注意して下さい。十進法における各桁の数字は、1 から 10 ではなく 0 から 9 なので、大半のプログラミング言語では、カウンタの始まりを 1 ではなく 0 にしています。インデントと同様、これも是非慣れて下さい。

以後、擬似コードでは計数ループを次のように記します (“繰り返し終わり。” については、これまでの慣例通りです):

- 変数 i を 0 から n の手前まで変えながら繰り返し、
- ... # ループ内の動作
- (繰り返し終わり)

では、先の積分プログラムを、計数ループにより書き直してみましょう:

- integ1: 関数 x^2 の区間 $[a,b]$ の定積分を区間数 n で計算
- $dx \leftarrow \frac{b-a}{n}$
- $s \leftarrow 0$
- 変数 i を 0 から n の手前まで変えながら繰り返し、
- $x \leftarrow a + i \times dx$
- $y \leftarrow x^2$ # 関数 $f(x)$ の計算
- $s \leftarrow s + y \times dx$
- (繰り返し終わり)
- s を返す。

先のプログラムと違うのは、毎回 x を i から計算している部分です。では、これの Python プログラムを示します:

```
def integ2(a, b, n):
  dx = (b - a) / n
  s = 0.0
  for i in range(n):
    x = a + i * dx
    y = x**2          # 関数 f(x) の計算
    s = s + y * dx
  return(s)
```

[7] 整数ならば、溢れない限り誤差はありません。

[8] 多くのプログラミング言語では、計数ループを表わすのに for というキーワードを使うので、計数ループのことを **for ループ** (For Loop) と呼ぶこともあります。

これを動かしてみましょう:

```
>>> integ2(1.0, 10.0, 100)
328.5571499999999
>>> integ2(1.0, 10.0, 1000)
332.5546214999998
>>> integ2(1.0, 10.0, 10000)
332.9554512149995
>>>
```

今度は、刻みを小さくすると順当に誤差が減少して行きます。しかし、常に正しい面積である 333 より小さいのは、何故でしょうか? それは、長方形の面積を計算する際に、微小区間の左端にある x を使って高さを求めているため、増加関数では、図 2 のように微小な三角形の分だけ面積が小さめに計算されてしまうからです (逆に、減少関数だと大きめに計算されます)。

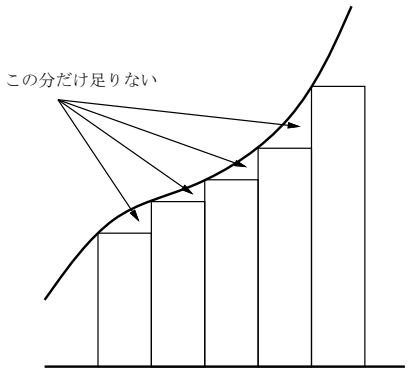

図 2: 区間の左端を使う場合の誤差

演習 4-1 上の演習問題にあるプログラムを打ち込んで動かせ。動いたら「減少する関数だと値が大きめに出る」ことを確認せよ。また、左端ではなく右端を用いた計算もしてみよ。最後に、次のような考え方で誤差が減少できるかどうか、実際にプログラムを書いて試してみよ。

a. 左端の $f(x)$ だけでも右端の $f(x)$ だけでも欠点があるので、両方で計算して平均を取る。
b. 左端や右端だから良くないので、区間の中央の $f(x)$ を使う。
c. 上記 a と b をうまく組み合わせてみる (どうすれば “うまい” のか、各自で考えよ)。

演習 4-2 次のような、繰り返しを使ったプログラムを作成せよ。

a. 整数 n を受け取り、2^n を計算する。
b. 整数 n を受け取り、$n! = n \times (n-1) \times \cdots \times 2 \times 1$ を計算する。
c. 整数 n と整数 $r\,(\leq n)$ を受け取り、${}_nC_r = \dfrac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1}$ を計算する。

4 微分方程式の数値解法

ここまでは、逐次細分最適化法の例として、数値積分を取り上げました。次に別の事例として、微分方程式の数値的解法を見てみます。

4.1 微分方程式

一般に、未知の関数とその**導関数** (Derived Function) から成る等式で定義される方程式を**微分方程式** (Differential Equation) と呼びます。このうち、未知の関数が (本質的に)1 つの変数を持つようなものを、特に**常微分方程式** (Ordinary Differential Equation) と呼びます。ここでは一番簡単な場合として、次の形の方程式を取り上げます:

$$\frac{dy}{dx} = f(x, y)$$

微分方程式を直観的に説明すると、XY 平面のあらゆる場所に傾きを示す矢印が定義されていて、その方向に沿った曲線を表わすものと見なすことができます (図 3)。このような曲線を**解曲線** (Solution Curve) と呼びます。傾きだけを見た場合、解曲線は無限に存在するので、その一般式を**一般解** (General Solution) と呼びます。これに対して、ある点 (x_0, y_0) を**初期値** (Initial Value) として与え、そこを通る曲線を求めるとすれば、その曲線は一つだけになります。これを**特殊解** (Particular Solution) と呼び、初期値から特殊解を求める問題を**初期値問題** (Initial Value Problem) と呼びます。

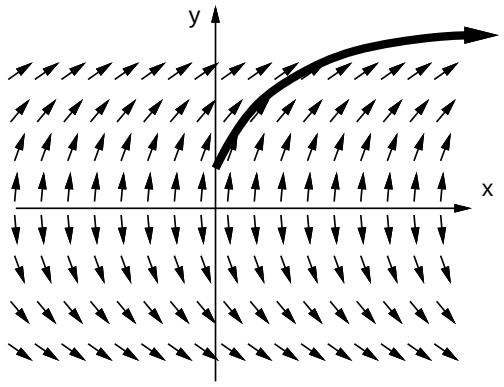

図 3: 微分方程式と初期値問題

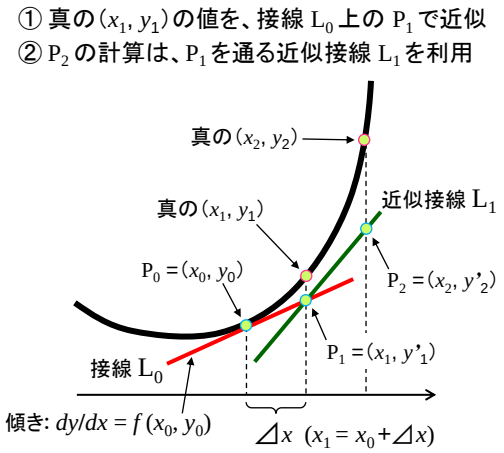

図 4: オイラー法のイメージ

簡単な例として、次の微分方程式を考えてみましょう:

$$\frac{dy}{dx} = \frac{1}{2y}$$

これを次のように変形します (本来、dy/dx は分数ではないのに、このような変形ができる理由を、付録に補足しておきました/興味のある方は是非読んでみて下さい):

$$2y\ dy = 1\ dx$$

次に両辺を不定積分します:

$$\int 2y\ dy = \int 1\ dx$$

両辺を積分した結果、次のようになります:

$$y^2 = x + C$$

これより次の式が求まり、これが一般解となります (当然 $y = 0$ は除きます):

$$y = \pm\sqrt{x + C}$$

この一般解から、例えば点 (0, 1) を通る特殊解を得るには、上式の x に 0、y に 1 を代入して $C = 1$ を算出します。これより、$y = \sqrt{x + 1}$ が特殊解となります。

4.2 オイラー法

前節のような簡単な微分方程式は簡単に解が求まりましたが、現実の問題では解析的には解が求められないことがあります。しかし、正確な解は得られずとも (解析的には解を求められずとも)、例えばグラフの形が分かるだけでも現象の傾向を掴むことは可能であり、新たな事実の解明に繋がります。よって以下では、微分方程式の解を数値的に求める方法について、考えてみましょう。

一番素朴な考え方として、初期値 (x_0, y_0) における解曲線の接線方向は、$\frac{dy}{dx} = f(x, y)$ として分かっているので、この方向に微小な値 h だけ動いた解曲線上の点 (x_1, y_1) は、(近似的に) この接線上にあると見なすというものがあります (図 4)。これより、以下同様にして次々と曲線上の点を求めて行くという方法が考えられます (以後、1 回分の計算を**ステップ**と呼ぶことにします)。これを**オイラー法** (Euler Method) と呼びます。形式的に定義すれば、次の漸化式で x_i, y_i を計算して行くのがオイラー法です:

$x_{i+1} = x_i + h$
$y_{i+1} = y_i + h\frac{dy}{dx} = y_i + hf(x_i, y_i)$　　y の増分 $=$ x の増分 $\times$ 傾き

この定義に従い、先ほどの微分方程式を数値的に解く Python のプログラムを示します。このプログラムは、x0 ～ xmax 間を count 個に分割し、その 1 区間の長さを h として (x0, y0) から count 回の近似計算をしています。そして、近似グラフを表示するために、各近似計算毎の結果 (y) を配列 s にしまっています[9]。本来ならば、各 (x_i, y_i) を記録するべきですが、x0 ～ xmax 間は等分に分割されるので、グラフの傾向を見るぐらいであれば、各 y_i を記録するだけでもできます。また、この微分方程式では、$f(x, y)$ が x に依存しないので、x の計算は省くこともできます (この例では、分かり易さを重視するために、入れていますが):

```
!pip install ita                        # Google Colaboratory へのログイン毎に 1 度実行して下さい。
import ita

def euler1(x0, y0, xmax, count):
  h = (xmax - x0)/count
  x = x0; y = y0
  s = ita.array.make1d(count)           # 結果を収納する配列を用意
  for i in range(count):
    x = x + h
    y = y + h*(0.5/y)                   # f(x, y) = 1/(2y)
    s[i] = y                            # 配列内の要素 (変数) に結果を収納
  ita.plot.plotdata(s, line=True)
  return(y)                             # x 座標が xmax の時の y 座標を表示
```

このプログラムを動かした結果、得られるグラフを図 5 に示します (このグラフは、あくまで傾向を見るだけなので、グラフにある x 座標の値は実際の x 座標の値と合っていません)。

```
>>> euler1(0, 1, 100, 1000)
10.055647112943161
```

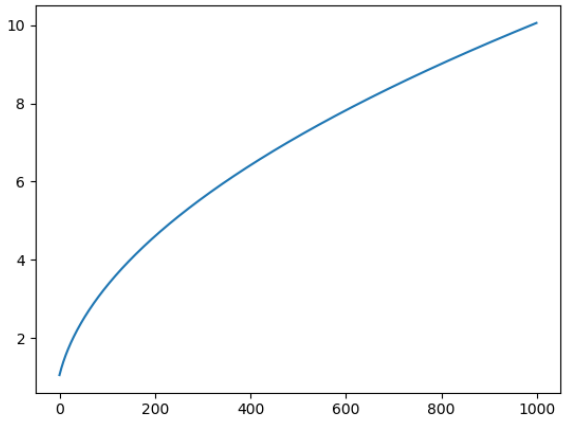

図 5: オイラー法により得られた微分方程式のグラフ

ところで、オイラー法は、個々の点における「近似」接線を延長して次の点を求めています。図 4 で言えば、真の (x_1, y_1) における傾きを $f(x_1, y'_1)$ で近似し ($f(x, y)$ に (x_1, y'_1) を代入した値を計算できるので、これを利用し)、その延長として P_2 を求めているというわけです。これはシンプルで分かり易いのですが、解曲線のカーブがきつい場合は、各近似点毎のずれの蓄積も大きくなり ($\because$ 前の近似点より次の近似点を求めているため)、真の解曲線と近似解曲線とのずれも大きくなってしまいます。これを補う方法として、次に説明するルンゲ–クッタ法があります。

[9]配列については、次のテーマで取り上げます。ここでは、配列は変数の並びであり、各変数とは番号を指定して (s[i])、値をやり取りする、ぐらいの理解で十分です。

4.3 ルンゲ–クッタ法

オイラー法では、各ステップの始点における接線を延長して次の点を求めていましたが、それでは解曲線のカーブがきつい場合に誤差が大きくなってしまいます。数値積分では、各微小矩形の左端/中点/右端をほど良く混ぜることで、精度を上げることができました。この考えを応用して、始点と終点の両方で接線の向きを求め、両者をうまく混ぜれば、より正確になることが予想されます。しかし実際には、各ステップの終点は「これから求める」値なので分かりません。そこで、まずはオイラー法で終点の近似値を求めてこれを仮の終点 1 とし、続いて仮の終点における傾きを改めて始点に用いて得られた終点を仮の終点 2 とします。そして、仮の終点 1 における y の増分 k_1 と、仮の終点 2 における y の増分 k_2 との平均を取ることを考えます (図 6)。

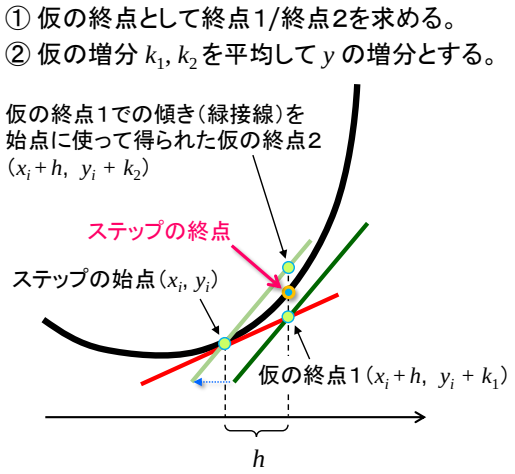

図 6: 2 次のルンゲ–クッタ法

つまり、次のようにするわけです:

$x_{i+1} = x_i + h$
$k1 = h\frac{dy}{dx} = hf(x_i, y_i)$ $k1$: 始点における傾きを用いて x が h 増えた時の y の増分
$k2 = hf(x_i + h, y_i + k1)$ $(x_i + h, y_i + k1)$: 仮の終点
$k2$: 仮の終点における傾きを用いて x が h 増えた時の y の増分
$y_{i+1} = y_i + \frac{1}{2}(k1 + k2)$

これを **2 次のルンゲ–クッタ法** (2nd-order Runge–Kutta Method) と呼びます (2 次と呼ばれる理由は、次回に説明します)。以下に Python のプログラムを示します:

```
# !pip install ita                    # Google Colaboratory へのログイン毎に 1 度実行して下さい。
import ita

def rungekutta2(x0, y0, xmax, count):
  h = (xmax - x0)/count
  x = x0; y = y0
  s = ita.array.make1d(count)         # 結果を収納する配列を用意
  for i in range(count):
    x = x + h
    k1 = h*(0.5/y)                    # k1, k2 の意味はルンゲ-クッタ法の漸化式を参照のこと
    k2 = h*(0.5/(y + k1))
    y = y + 0.5*(k1 + k2)
    s[i] = y                          # 配列内の要素 (変数) に結果を収納
  ita.plot.plotdata(s, line=True)
  return(y)                           # x 座標が xmax の時の y 座標を表示
```

表示されるグラフは、あくまで傾向を見るためのものなので、これらを比較しても両者の違いはあまり見えません。そこで、$(0, 1)$ を始点として計算を始め、x 座標が 1 となった時点での y 座標の値を比べてみましょう。この微分方程式の解は $y = \sqrt{x+1}$ なので、本来ならば 1.4142135623730950... となるはずです:

```
>>> euler1(0, 1, 1, 1000)
1.4142748458924588
>>> rungekutta2(0, 1, 1, 1000)
1.4142135623751702
>>>
```

これを見ると、確かにオイラー法より精度が良くなっていることが分かりますね。ところで、実はルンゲ–クッタ法には、終点の混ぜ方などに応じて様々なバリエーションがあります。ここでは、次回も含め議論の見通しが良くなるように、ステップ幅 h をそのまま利用した終点 $1/2$ の平均値を用いました。

演習 4-3 以下の微分方程式について、オイラー法とルンゲ–クッタ法により点 $(0, 1)$ を通る近似解曲線を求めるプログラムを作成し、両者の値を比較しなさい。

$$\frac{dy}{dx} = \frac{1}{4}\, y^{-1}$$

付録: dx と dy について

厳密な議論をしたい方に向け、dy/dx と dx, dy との関係について、以下に補足しておきます。これまで、微分 $dy/dx = f(x,y)$ における左辺は分数ではない (つまり、dx, dy は単独では意味を持たない) という説明を受けた方が多いかと予想します。しかし本来、微分は $\Delta y/\Delta x$ における $\Delta x \to 0$ の極限として扱われ、元々は分数であったものが極限を取った瞬間に分数ではなくなる、と言うのも不思議な話です。dy/dx は分数ではないという考え方は、主に以下の理由によるものです。

例えば、$y = F(x) = x^2$ について考えます。この場合、Δx, Δy の関係は、次のようになります。

$$y + \Delta y = F(x + \Delta x) = (x + \Delta x)^2 = x^2 + 2x \cdot \Delta x + (\Delta x)^2 \tag{1}$$

ここで、$\Delta x \to 0$ とすると、$(\Delta x)^2$ は Δx より小さいので、$\lim_{\Delta x \to 0}(\Delta x)^2 = 0$ となります。よって、式 (1) は下記の式になります。

$$y + \Delta y = x^2 + 2x \cdot \Delta x \tag{2}$$

式 (2) に $y = x^2$ を代入し、両辺を Δx で割ると、最終的に下記の式が導かれます。

$$\frac{\Delta y}{\Delta x} = 2x$$

この式変形は、一見良さそうに見えますが、実は $\lim_{\Delta x \to 0}(\Delta x)^2 = 0$ とした時点で、$(\Delta x)^2 = 0$ より $\Delta x = 0$ と考えているにもかかわらず、その後で Δx による割り算をしています。つまり、0 で割っているのではないか、という疑問が生じ、この部分に論理の曖昧性があると考えられました。そこで、$\lim_{h \to 0}\{F(a+h) - F(a)\}/h$ を考え、まずは $\{F(a+h) - F(a)\}/h$ を計算します。その後、$h \to 0$ とした場合の極限値を、関数 $F(x)$ の $x = a$ における微分係数と呼び、$F'(a)$ と表わすことにしました。$y = F(x) = x^2$ については、下記のようになります。

$$\lim_{h \to 0} \frac{F(a+h) - F(a)}{h} = \lim_{h \to 0} \frac{(a+h)^2 - a^2}{h} = \lim_{h \to 0} \frac{2ah + h^2}{h} = \lim_{h \to 0} 2a + h = 2a$$

ここでは、$h \to 0$ とした後の割り算が入りません。また、微分係数については、x の各値 (要は様々な a) 毎にその微分係数を対応させることが可能であり (つまり、新しい関数を作ることが可能であり)、これを関数 $F(x)$ の導関数 dy/dx $(= F'(x))$ と定義しました。この場合の dy/dx は、それ自体が数学記号です (つまり分数ではない)。

ところで、皆さんは微分係数や導関数の定義を見ることはあっても、“微分” の定義と言われるものはあまり見たことがないと予想します。本来ならば、dF や dx などの無限小変分を微分と定義したいところですが、上ではこれらを避けた議論をしています。このため、上の定義による微分係数や導関数に基づく議論を行なう場合は、導関数を求めることを微分と呼ぶ事例が多いです。但し、これでは、厳密な定義を求めた結果、逆に微分そのものの定義が曖昧となってしまったため、$F(x_1, x_2, \ldots, x_n)$ の全微分 $dF = (\partial F/\partial x_1) \cdot dx_1 + (\partial F/\partial x_2) \cdot dx_2 + \cdots + (\partial F/\partial x_n) dx_n$ を微分と呼ぶ再定義化が行なわれています (これが、本付録の後半です)。

以下では、$\Delta y/\Delta x$ の意味について、改めて考えてみましょう。関数 $y = F(x)$ のグラフを考えると、$\Delta y/\Delta x$ は、$F(x)$ 上の 2 点 (x, y), $(x+\Delta x, y+\Delta y)$ からなる弦の傾きであり、$\Delta y/\Delta x$ は x の関数として表わせます $(\Delta y = F(x+\Delta x) - F(x))$。この時、$\Delta x \to 0$ とすると、この 2 点は 1 点に収束し、また弦は $y = F(x)$ と点 (x, y) だけを共有する線に収束します。この線を、傾き $\lim_{\Delta x \to 0} \Delta y/\Delta x = \lim_{\Delta x \to 0}\{F(x + \Delta x) - F(x)\}/\Delta x = F'(x)$ を持ち、$y = F(x)$ と点 (x, y) で接する接線と呼ぶことにします。さて、ここで問題となるのは、接線は一意に決定できるか、ということです。もちろん、図形的 or 経験的に接線が一本なのは当たり前と言いたいところですが、無限小での Δx と Δy の計算における論理の曖昧性から微分係数 · 導関数の議論を展開したにもかかわらず、接線については経験則を根拠にするというのでは、やはり疑問が残ります。

接線については、以下のように考えることで、その一意性が保証されます。$y = F(x)$ と点 $(a, F(a))$ で接する、微分係数を用いて定義された接線を $y = F'(a)(x-a) + F(a)$ とします。また、これ以外に点 $(a, F(a))$ で接する接線を、次のように表わします (どんな形の式か分からないので、以後の見通しが良くなるように、取り敢えず $y = F'(a)(x-a) + F(a)$ が見える形を作ってみた)。

$$y = F'(a)(x-a) + F(a) + R(x) \tag{3}$$

接線なので、どのような接線であれ全て 1 次式になります。よって、$R(x)$ は、x の高々1 次式です。次に、$y = F(x)$ を用いて、式 (3) を次のように変形します。

$$\frac{F(x) - F(a)}{x - a} - F'(a) = \frac{R(x)}{x - a}$$

ここで、$x \to a$ とすると、$\{F(x) - F(a)\}/(x - a) \to F'(a)$ より、$R(x)/(x - a) \to 0$ となります。これは、$x \to a$ とした時、$R(x)$ が $x - a$ より速く 0 に収束することを意味します。$R(x)$ は x の高々1 次式なので、$R(x) = px + q$ とすると、$R(x)/(x - a) = p + (pa + q)/(x - a)$ より、$p = q = 0$ でない場合は、$\lim_{x \to a} R(x)/(x - a) \to \infty$ と発散してしまいます。つまり、$R(x) \equiv 0$ です。以上より、式 (3) は、微分係数を用いて定義された接線と同じものになります。

さて、上の議論により接線の一意性が保証されたとして、以下では、$\Delta x \approx 0$ の近傍について考えてみます。$\Delta x \approx 0$ の近傍では、$\Delta y/\Delta x = F'(x) + \epsilon$ より、$\Delta y = F'(x) \cdot \Delta x + \epsilon \cdot \Delta x$ となります。$\lim_{\Delta x \to 0} \Delta y/\Delta x$ が $F'(x)$ に収束することから、$\Delta x \to 0$ では $\epsilon \to 0$ となり、$\epsilon \cdot \Delta x$ は Δx より速く 0 に収束します。これより、Δy の主要部 (Δy に一番影響を与えている項) は $F'(x) \cdot \Delta x$ となります。よって、$F'(x) \cdot \Delta x$ を改めて関数 $y = F(x)$ の微分と定義し、dy と表わすことにします。つまり、$dy = F'(x) \cdot \Delta x$ です。ここで、関数 $y = F(x) = x$ を考えると、$F'(x) = 1$ であり、これを先ほどの微分の定義に当てはめると、$dy = dF(x) = dx = F'(x) \cdot \Delta x = 1 \cdot \Delta x = \Delta x$ より、$\Delta x = dx$ となります。$\Delta x \approx 0$ の近傍に関する議論から微分の定義に至る議論において、無限小による割り算が行なわれていない点に注意して下さい。以上より、この定義に従えば、$dy = F'(x) \cdot dx$ と扱うことは合理的です。

ところで、上では 2 次元の場合を説明しましたが、3 次元以上ではどう考えるのか、簡単に触れておきます。2 次元では微分係数をもとに接線やその傾きについて議論しましたが、3 次元以上では、各次元毎の微分係数 (偏微分係数) をもとに超接平面やその傾きについて議論します。

超接平面の一意性については、2 次元の時と同じ考え方を適用できます。$y = F(x_1, x_2, \ldots, x_n)$ に点 $A = (a_1, a_2, \ldots, a_n)$ で接する超接平面は、次の式で表わされます。

$$y = F(a_1, a_2, \ldots, a_n) + \left.\frac{\partial F}{\partial x_1}\right|_A (x_1 - a_1) + \left.\frac{\partial F}{\partial x_2}\right|_A (x_2 - a_2) + \cdots + \left.\frac{\partial F}{\partial x_n}\right|_A (x_n - a_n) \tag{4}$$

ここで、式 (4) 以外に点 A で接する超接平面を、次のように表わします。

$$y = F(A) + \left.\frac{\partial F}{\partial x_1}\right|_A (x_1 - a_1) + \left.\frac{\partial F}{\partial x_2}\right|_A (x_2 - a_2) + \cdots + \left.\frac{\partial F}{\partial x_n}\right|_A (x_n - a_n) + R_1(x_1) + R_2(x_2) + \cdots + R_n(x_n)$$

超接平面は線形式なので、$R_i(x_i)$ は x_i の高々1 次式です。以下、x_i 以外の x に $(a_1, a_2, \ldots, a_n)$ を代入して値を固定した後、2 次元の時と同様、$x_i \to a_i$ における "$\{F(x) - F(A)\}/(x_i - a_i) - \partial F/\partial x_i|_A = \{R_i(x_i) + \alpha\}/(x_i - a_i)$" の収束性を考えます ($\alpha$ は x_i 以外の x に $(a_1, a_2, \ldots, a_n)$ を代入した定数項, $R_i(x_i) + \alpha$ は x_i の高々1 次式)。

以後も 2 次元の時と同様な議論を行なうことで、最終的に $y = F(x_1, x_2, \ldots, x_n)$ の微分 dF が定義されますが、これは F の全微分と呼ばれるものです。

最後に、別の定義を紹介します。まずは、$i = ah$ という関係にある各変数を考えます。h は有限値 (つまり、無限小や無限大ではない何らかの値) とします。また、a, i については、$a \to 0$ とした時、a および h の値に応じて $i \to 0$ となる i が存在するとします。$y = F(x)$ にこれらの変数を適用すると、以下の関係が成り立ちます。

$$\frac{F(x + i) - F(x)}{i} = \frac{F(x + ah) - F(x)}{ah}$$

この式を次のように変形します。

$$\frac{F(x + ah) - F(x)}{a} = \frac{F(x + i) - F(x)}{i} \cdot h$$

ここで、$a \to 0$ とした時の左辺の極限値を関数 $y = F(x)$ の微分と定義し、dy と表わします。さらに、$a \to 0$ となる a に応じて $i \to 0$ となる i を取れるので、$\{F(x + i) - F(x)\}/i$ は $F(x)$ の導関数 $F'(x)$ であり、$dy = F'(x) \cdot h$ となります。最後に、関数 $y = F(x) = x$ に上と同じ議論を適用することで $h = dx$ が導かれ、$dy = F'(x) \cdot dx$ となります (この別定義は、微分係数や導関数を定義した**オーギュスタン・ルイ・コーシー** (Augustin Louis Cauchy) 自身によるものです)。