情報科学 2010　久野クラス　＃6

久野　靖*

2009.11.19

はじめに

皆様、(時間) 計算量の課題はどうでしたか。計算量について一応知っているということは、まっとうなプログラムを組む人 (別にプロという意味はなく、アマチュアでも) にとって必須なので忘れないようにお願いします。さて今回は、「誤差」「数値積分」「求解」に続く数値解析の話題として、残っている次のものをやります。

- 連立一次方程式の数値解法
- 常微分方程式の数値解法

今回は駒場祭があるので、期限を 2 週間とし、少量にしました。4B と締切日が重なっていますが、4B の方は希望があれば延長します。

1　前回の演習問題解説

1.1　演習 1〜2、4〜5 — 整列アルゴリズムの時間計測

まず最初にビンソートと基数ソートのプログラムを掲載しておきます。ビンソートは非常に簡単です: [1]

```
def binsort(a)
  bin = Array.new(10000, 0)
  a.each do |i| bin[i] = bin[i] + 1 end
  k = 0
  bin.length.times do |i|
    bin[i].times do a[k] = i; k = k + 1 end
  end
end
```

基数ソートは値のビット数をパラメタで渡すようにしています。[2]各周回ごとに当該ビットの値に応じてデータを配列 b と c に振り分け、終わったらこの順で a にコピーし戻しています:

```
def radixsort(a, bits)
  b = Array.new(a.length); c = Array.new(a.length)
  bits.times do |pos|
    mask = 2**pos; bc = 0; cc = 0
    a.length.times do |i|
      if (a[i] & mask) == 0
        b[bc] = a[i]; bc = bc + 1
      else
```

*筑波大学大学院経営システム科学専攻
[1]データ (整数) の最大値をパラメタとして渡す方がいいかもしれません。
[2]10000 までの値だと 14 ビットで済むので 14 を指定すればよいです。

```
          c[cc] = a[i]; cc = cc + 1
        end
      end
      bc.times do |i| a[i] = b[i] end
      cc.times do |i| a[bc+i] = c[i] end
    end
  end
```

各種整列アルゴリズムの計算時間を N を変えて手元のマシンで計測した結果を 1 に示します。

表 1: さまざまな整列アルゴリズムの時間計測

データ数	1,000	2,000	3,000	5,000	10,000	20,000	30,000	50,000
バブルソート	1,219	4,945	11,117	-	-	-	-	-
単純選択法	305	1,242	2,766	-	-	-	-	-
単純挿入法	375	1,531	3,445	-	-	-	-	-
マージソート	23	62	94	164	351	758	1,172	2,055
クイックソート	15	31	55	109	219	508	789	1,492
ビンソート	8	8	11	14	20	34	47	74
基数ソート	27	57	86	141	280	566	838	1,402
$N+E$	11,000	12,000	13,000	15,000	20,000	30,000	40,000	60,000

$O(N^2)$ のアルゴリズムである 単純挿入法、バブルソートは N が大きくなると急激に遅くなって役に立たなくなります。一方、$O(N \log N)$ のマージソートとクイックソートは十万くらいのデータであれば十分実用になると言えます。

ビンソートは極めて速いことは分かりますね。では、このアルゴリズムの時間計算量はどうでしょう。まず、すべてのデータを順に走査するという点では $O(N)$ だと 言えますが、それだけではありません。値の数を E とすると、最後に大きさ E の配列を全部調べながら値を生成するので、このための時間も E が大きいと問題になります。なので、時間計算量は $O(N+E)$ になるわけ です。今回は E が 10,000 なので、途中まではこちらの方が主に問題になります。表 1 の一番下に $N+E$ を示しましたが、所要時間がだいたいこれに比例していることが読み取れます。

そして、ビンソートは E 個ぶんの配列を必要とすることも忘れてはいけません。アルゴリズムによっては、大量のメモリを使うことで時間を速くすることができますが、ビンソートはまさにその例です。ビンソートが要する記憶領域は元のデータ数 N と数えるための配列の数 E を併せたものだから、これを「**領域計算量**が $O(N+E)$ である」のように言い ます。つまり、値の範囲が広くなると、ビンソートは領域計算量の点でも不利になるわけでです。

なお、これまでに出て来たアルゴリズムのほとんどは領域計算量 $O(N)$ ですが、マージソートと基数ソートは「別の場所に移してから戻す」ので $O(2N)$ になって います。[3]

最後に基数ソートの時間計算量ですが、キーのビット数ぶんだけ振り分け処理をおこなうので、時間計算量は $O(N \log E)$ ということになりま す。これを $\log E$ が今回は定数 (14) と考えれば、時間計算量は**線形時間**ということになります。実際、表 1 をチェックすると所要時間が N にほぼ比例していることが分かります。

ただし、時間そのものはほとんどの場合においてクイックソートよりも劣っています。これはつまり、データが非常に多くなると、先に説明したようにオーダーの差がすべてを支配しますが、それほどでもない場合には定数項の差が無視できず、オーダーの大きいアルゴリズムでも処理時間が短くて済む場合がある、ということを意味しています。たとえば、データが数個しかないのであれば、クイックソートを使うよりも単純選択方を使うほうが適切なわけです。

1.2 演習 3 — クイックソートの弱点

先に掲げた `quicksort` のコードが整列ずみの配列に対しては遅いという弱点を実際に示すには、次のように 2 回ずつ整列してみればよいでしょう:

[3] 基数ソートでは `b` と `c` を `a` と同じサイズで取るので 3 倍と思うかもしれませんが、ケチるなら `b` と `c` を 1 つの配列にして「前から」と「後ろから」データを詰めてゆけばよいのです。

```
def test
  a = randarray(1000)
  bench(1) do quicksort(a, 0, 999) end
  bench(1) do quicksort(a, 0, 999) end
  a = randarray(2000)
  bench(1) do quicksort(a, 0, 1999) end
  bench(1) do quicksort(a, 0, 1999) end
  a = randarray(3000)
  bench(1) do quicksort(a, 0, 2999) end
  bench(1) do quicksort(a, 0, 2999) end
end
```

これを実行してみると、結果は次のようになりました:

データ数	1,000	2,000	3,000
1 回目	16	39	63
2 回目	984	3960	8859

確かに、1 回目は $O(N)$ に近い (実際には $O(N \log N)$ のはず) けれど、2 回目はずっと遅くて $O(N^2)$ に近くなっているようです。

これを改良するにはどうしたらいいでしょう? それには、ピボット値を取る時にいつも「端っこ」から取っていたからまずいので、代わりにランダムに取るようにすればよいでしょう: [4]

```
def quicksort(a, i, j)
  if j <= i then
    # do nothing
  else
    p = i + rand(j-i+1); swap(a, p, j) # ***
    pivot = a[j]; s = i
    i.step(j-1) do |k|
      if a[k] <= pivot then swap(a, s, k); s = s + 1 end
    end
    swap(a, j, s); quicksort(a, i, s-1); quicksort(a, s+1, j)
   end
end
```

これを実行してみると、上の表と異なり、1 回目でも 2 回目でもほぼ同じ時間で整列が終わるようになっています。

演習 6a — 最大公約数

2 つの数 $M, N (M < N$ とする) の最大公約数のベタなバージョン (M からカウンタを 1 ずつ減らしてゆき、それで両者が割り切れることをチェックする方法) は当然 $O(M)$ に なります:

```
def gcdenumerate(x, y)
  min = x; if min > y then min = y end
  (min-1).step(1, -1) do |i|
    if x % i == 0 && y % i == 0 then return i end
  end
end
```

では「大きい方から小さい方を引いてゆく」バージョンはどうなのでしょうか?

[4] プログラムの修正は`***`の 1 行を挿入しただけです。つまりここで、`i`〜`j` の範囲の整数 `p` を 1 つランダムに選び、`a[j]` と `a[p]` を交換してから、あとはこれまでと同様に処理するわけです。

```
def gcd(x, y)
  while x != y do
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  return x
end
```

最善の場合は $M = N$ の時で、すぐ終わります。最悪の場合は $M = 1$ のときで、$N - 1$ 回引き算をしないと終わりません。平均は…? そこで、乱数を使って実験してみました:

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.0859375
bench(10000) do gcd(rand(1000)+1, rand(1000)+1) end → 0.1640625
bench(10000) do gcd(rand(10000)+1, rand(10000)+1) end → 0.2734375
bench(10000) do gcd(rand(100000)+1, rand(100000)+1) end → 0.4453125
```

これを見ると、値が 10 倍ずつ大きくなる時に一定くらいずつ (やや多いですが) 値が増えて行きます。引き算ごとに M や N が一定比率くらいで減少して行くと考えれば $O(\log M)$ ということになるわけです。しかし M と N の比率が違うとどうでしょうか:

```
bench(10000) do gcd(rand(100)+1, rand(100)+1) end → 0.203125
bench(10000) do gcd(rand(1000)+1, rand(100)+1) end → 1.328125
bench(10000) do gcd(rand(10000)+1, rand(100)+1) end → 12.1171875
bench(10000) do gcd(rand(100000)+1, rand(100)+1) end → 130.546875
```

M の方が大きいとして、M が 10 倍になると時間も 10 倍になります (これは、M が N の 1000 倍なら 1000 回引く必要があるわけですから当り前ですね)。そうすると、計算量は全体として $O(M \log M)$ という ことになるでしょうか。

では、引き算の代わりに剰余演算を使うユークリッドの互除法ではどうでしょう?

```
def gcd3(x, y)
  while true do
    if x > y
      x = x % y; if x == 0 then return y end
    else
      y = y % x; if y == 0 then return x end
    end
  end
end
```

これでまずアンバランスな方から測ってみましょう:

```
bench(10000) do gcd3(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcd3(rand(1000)+1, rand(100)+1) end → 0.046875
bench(10000) do gcd3(rand(10000)+1, rand(100)+1) end → 0.046875
bench(10000) do gcd3(rand(100000)+1, rand(100)+1) end → 0.0390625
```

まったく変わりませんね。それは、CPU の割算命令の時間は値が変わってもほとんど一定時間で実行されるからです (ただし Ruby で多倍長演算が必要なくらい大きい値になるとそれは 1 命令ではできなくなるのでこのようには行きません)。では値の大きさの影響はどうでしょうか?

```
bench(10000) do gcd3(rand(100)+1, rand(100)+1) end → 0.0390625
bench(10000) do gcd3(rand(1000)+1, rand(1000)+1) end → 0.0546875
bench(10000) do gcd3(rand(10000)+1, rand(10000)+1) end → 0.0625
bench(10000) do gcd3(rand(100000)+1, rand(100000)+1) end → 0.078125
bench(10000) do gcd3(rand(1000000)+1, rand(1000000)+1) end → 0.0859375
```

こちらは 10 倍になるごとにおよそ一定ずつ増えてゆくようです。それは先と同じで、ループを 1 回まわるごとにだいたい一定比率で 2 つの値が小さくなるからでしょう。これを総合すると、このアルゴリズムの時間計算量は $O(\log M)$ ということに なります。[5]

1.3 演習 6b — フィボナッチ数

再帰的定義そのままの**フィボナッチ数**の計算は、`fib(`N`)` の計算に `fib(`$N-1$`)` と `fib(`$N-2$`)` を 実行し、それがさらに `fib(`$N-2$`)` と `fib(`$N-3$`)`、`fib(`$N-3$`)` と `fib(`$N-4$`)` をそれぞれ呼び、というふうに「倍々」になるので、時間計算量は $O(2^N)$ と なります。[6]これに対し、ループで計算する場合は当然 $O(N)$ に なります:

```
def fibloop(n)
  x0 = 1; x1 = 1
  n.times do
    t = x0 + x1; x0 = x1; x1 = t
  end
  return x1
end
```

一応計測してみましょう:

```
bench(10000) do fibloop(10) end → 0.0625
bench(10000) do fibloop(100) end → 0.8359375
bench(10000) do fibloop(1000) end → 11.9140625
```

「10 倍になるごとに時間も 10 倍」から外れていますが、これは `fib(50)` くらいからもう結果が多倍長計算が必要な大きさ になってしまうからでしょう。

では、行列計算を使ってなおかつ行列の N 乗の計算を工夫する方法だとどうでしょうか。その「工夫」の漸化式を再掲します:

$$Q^n = \begin{cases} E & (n=0) \\ QQ^{n-1} & (n \text{ が正の奇数}) \\ (Q^{\frac{n}{2}})^2 & (n \text{ が正の偶数}) \end{cases}$$

これを用いるプログラムは次のとおり: [7]

```
def mat22multvec(a, v)
  c = [0, 0]   # 結果用の 1 次元配列
  c[0] = a[0][0]*v[0] + a[0][1]*v[1]
  c[1] = a[1][0]*v[0] + a[1][1]*v[1]
  return c
end
def mat22mult(a, b)
  c = [[0,0],[0,0]]   # 結果用の 2 次元配列
  c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0]
```

[5]数が大きくなって 1 語に入らなくなった場合は、除算の実行が一定時間とは見なせなくなります。その場合には、除算の計算にその数の桁数に比例する時間、すなわち $O(\log M)$ を要するので、全体としての時間計算量は $O(\log^2 M)$ となります。

[6]これは**指数時間**で、極めて遅いということです。

[7]2×2 行列の積、行列とベクトルとの積を下請けに用意して、それを用いて上の漸化式を使った N 乗を定義し、最後にそれを用いてフィボナッチ数を計算しています。

```
  c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1]
  c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0]
  c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1]
  return c
end
def mat22power(a, n)
  if n == 0
    return [[1,0],[0,1]]  # 2x2 単位行列
  elsif n % 2 == 1
    return mat22mult(mat22power(a, n-1), a)
  else
    b = mat22power(a, n/2); return mat22mult(b, b)
  end
end
def fibmat(n)
  return mat22multvec(mat22power([[1,1],[1,0]], n), [1,1])[0]
end
```

実験してみましょう:

```
irb(main):101:0> bench(10000) do fibmat(10) end → 0.703125
irb(main):102:0> bench(10000) do fibmat(100) end → 1.46875
irb(main):103:0> bench(10000) do fibmat(1000) end → 2.921875
```

10 倍になるごとにだいたい一定ずつ時間が増えています (最後のほうは多倍長になるので外れていますが…)

この場合、計算量はどうでしょうか。「正の奇数」が選ばれるのは N を 2 進表現した時に「1」が現れる回数と等しくなり、「正の偶数」が選ばれるのは N を (奇数なら 1 を引きながら) 半分ずつにしていき 0 になるまでやるので、2 進表現の桁数つまり $\log N$ が回数となります。上記「1」の数は平均すると桁数の半分くらいだから $\frac{\log N}{2}$ となります。これらを合わせると、全体として $O(\log N)$ と なります。

1.4 演習 6c — 組み合わせの数

再帰的定義はフィボナッチと同様、倍々の呼び出しになるので、$O(2^N)$ と なります。では「パスカルの三角形」の場合はどうでしょうか。図 1 を N 段目まで作るとなると、要素数が $\frac{N(N+1)}{2}$ ありますから、計算量としては $O(N^2)$ ということになる はずです。コードを書いてみると次のようになります:

```
def combarray(n, r)
  a = Array.new(n+1, 1)
  1.step(n) do |i|
    (i-1).step(1, -1) do |k| a[k] = a[k-1] + a[k] end
    # p(a)
  end
  return a[r]
end
```

これは、N 段までのパスカルの三角形を作るために、要素数 $N+1$ の「1」ばかりが詰まった配列を用意し、それを図 1 のように隣の要素どうし足すことを繰り返していくものです。内側ループを添字が大きい方から順に処理しているのは、そうしないと「1 つ前の値」を使うことができないからです。

では時間を計測してみます:

```
bench(10000) do combarray(10,5) end → 0.609375
bench(10000) do combarray(20,10) end → 2.2578125
bench(10000) do combarray(30,15) end → 4.9765625
```

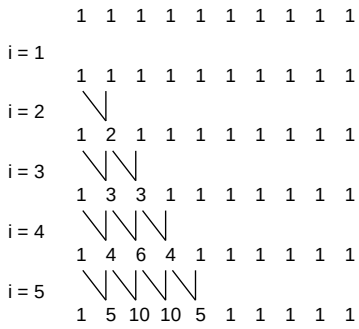

図 1: パスカルの三角形の計算

確かに $O(N^2)$ のよう です。ところで、前にやった「普通に掛け算する」バージョンはどうでしょうか?

```
def combloop(n, r)
  result = 1
  1.step(r) do |i|
    result = result * (n - r + i) / i
  end
  return result
end
```

これならループが r 回だから ($r \sim N$ として) $O(N)$ と なります:

```
bench(10000) do combloop(10,5) end → 0.0703125
bench(10000) do combloop(20,10) end → 0.1171875
bench(10000) do combloop(30,15) end → 0.2265625
```

たしかにずっと速いので、結局、ループで普通に計算するのがよいというオチでした。ただし、「何回もさまざな値を」使うのであれば、パスカルの三角形を「2 次元配列」の上で作成しておいて、そこから値を取り出すようにするのが良さそうです。

2 連立 1 次方程式の数値解法

2.1 消去法

連立 1 次方程式 (simaltaneous linear equations) とは、複数の変数を含んだ **1 次式** (linear equation) を並べたもので、変数の数と 1 次式の数が等しければ、[8]各変数の値を一意に決めることができます。たとえば次に 3 元連立 1 次方程式の例を示します:

$$\begin{aligned} 2x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ 2x_0 &+ 5x_1 &+ 4x_2 &= 4 \\ 4x_0 &+ 8x_1 &+ 8x_2 &= 7 \end{aligned}$$

これを手で解くとしたら、どうするでしょう? まず、一番上の式を 2 番目、3 番目から引く[9]ことで x_0 の**係数** (coefficient) を消去できます:

$$\begin{aligned} 2x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ &\quad 2x_1 &+ 2x_2 &= 3 \\ &\quad 2x_1 &+ 4x_2 &= 5 \end{aligned}$$

[8]正確にはこれに加えて、**不定** (indeterminate — 解が無数にあること) や**不能** (impossible — 解が存在しないこと) となるような特別な場合を除けば、という条件も必要です。

[9]もちろん、3 番目については 2 倍してから引く必要があります。

同様にして 2 番目の式を 3 番目から引くことで x_1 の係数を消去できます:

$$\begin{aligned} 2x_0 &+ 3x_1 &+ 2x_2 &= 1 \\ 2x_1 &+ 2x_2 &= 3 \\ 2x_2 &= 2 \end{aligned}$$

これで $x_2 = 1$ が求まるので、それを 2 番目、1 番目に代入でき、それにより引続き $x_1 = 0.5$ が求まるので、それを 1 番目に代入できます:

$$\begin{aligned} 2x_0 &= -2.5 \\ x_1 &= 0.5 \\ x_2 &= 1 \end{aligned}$$

これで $x_0 = -1.25$ となり、全ての解が求まりました。

このように、まず**前進消去** (forward elimination) により順に係数を消去して残った係数が**三角形** (triangular form) になるようにして、その後**後退代入** (backward substitution) により順に値を代入していくことで、連立 1 次方程式を解くことができます。このアルゴリズムを**ガウスの消去法** (Gaussian elimination) と呼びます。実際にこれをプログラムで扱う場合は、変数の名前はどうでもよいわけなので、係数だけを配列のデータとして与えます。[10]Ruby プログラムは、単に配列の上で前進消去、後退代入の操作を行うものとします。そうすると、実行が終わった時に各行の右端の値が方程式の解になるわけです。ではプログラムを次に示しましょう: [11]

```
def gauss(a)
  n = a.length
  n.times do |i|                    # i を 0〜n-1 まで (a[i][i] を消去)
    (i+1).step(n-1) do |j|          # j を i+1〜n-1 まで (i 行より下を掃き出す)
      r =  a[j][i] / a[i][i].to_f   # r は j 行目を掃き出す時の係数
      i.step(n) do |k| a[j][k] = a[j][k] - a[i][k]*r end   # 掃き出し
    end
    # p(a)
  end
  (n-1).step(0, -1) do |i|          # i を n-1〜0 まで逆順に変化
    a[i][n] = a[i][n] / a[i][i]; a[i][i] = 1.0          # a[i][i] で割る→解が求まる
    i.times do |j| a[j][n] = a[j][n] - a[j][i]*a[i][n] end    # 求まった解を差し引く
    # p(a)
  end
  return a
end
```

ではこれを動かしたところを見てみましょう:

```
irb> gauss([[2,3,2,1],[2,5,4,4],[4,8,8,7]])
=> [[1.0, 3, 2, -1.25], [0.0, 1.0, 2.0, 0.5],
    [0.0, 0.0, 1.0, 1.0]]
```

確かに解が求まっています。ところで、つぎのデータだとどうでしょうか:

```
irb> gauss([[2,3,2,1],[2,3,4,4],[4,8,8,7]])
=> [[1.0, 3, 2, NaN], [0.0, 1.0, 2.0, NaN],
    [0.0, NaN, 1.0, NaN]]
```

何がいけないのでしょう? この場合、2 番目の式から 1 番目の式を引くと最初の 2 つの係数がともに 0 になります。そのため、係数 0 を消去しようとして 0 で割り算をしてしまい、NaN になってしまうわけです。

しかし、この連立方程式が解けないということは全くなくて、次のように順番を入れ換えれば問題なく解くことができます:

[10] ここでは 2 次元配列を直接 `irb` に手で打ち込むことにします。連立 1 次方程式なので当然、幅が高さより 1 大きくなければいけません。

[11] `p` というのは配列などを `irb` の出力と同じように表示してくれるメソッドで、途中経過を見たい時はコメントを外して表示させてみるとよいでしょう。

```
irb> gauss([[2,3,2,1],[4,8,8,7],[2,3,4,4]])
=> [[1.0, 3, 2, -0.25], [0.0, 1.0, 4.0, -0.5],
    [0.0, 0.0, 1.0, 1.5]]
```

もちろん、人間が順番を考えるようでは困るので、実際にはプログラムの方で次に消去しようとする係数が 0 だったら、別の行と入れ換えてから消去を行う必要があります。[12]これを**ピボット選択** (pivoting) と呼びます。[13]ただし、ピボット選択を行おうとしても、すべての係数が 0 で選べないこともあります。これは、方程式がもともと不定/不能の場合に起きます。

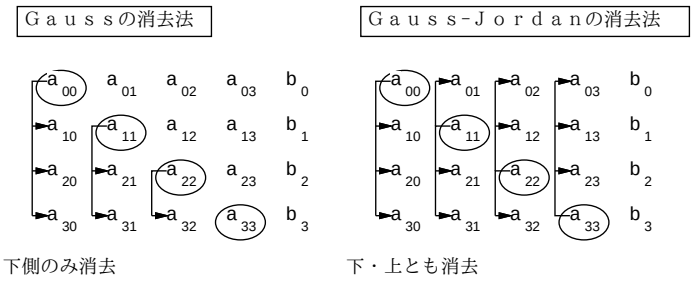

図 2: ガウス-ジョルダンの消去法

ガウスの消去法では処理が前進消去と後退代入の 2 つの段階に分かれていましたが、消去の時にこれまでのように「自分より下の行について消去する」代わりに「自分以外のすべての行について消去」することで、1 段で解を求めることができ、プログラムがやや簡単になります (図 2)。これを**ガウス–ジョルダンの消去法** (Gauss–Jordan elimination) と呼びます。[14]

演習 1 上の例題をそのまま打ち込んで動かせ。動いたら次のように改造せよ。

a. ガウス–ジョルダンの消去法に改造しコードが短くなることを確認。
b. ピボット選択を入れ、例題で駄目だったデータが扱えることを確認。
c. 方程式が不能/不定の時にその旨表示する機能を追加。

2.2 反復法

連立 1 次方程式の消去法とは別の原理による数値解法として、**反復法**と呼ばれるカテゴリのものがあります。ここではそのうちから、**ヤコビ法** (Jacobi method) と呼ばれる方法を説明します。ここまでに見てきたような連立 1 次方程式を行列・ベクトルを使って書き直すと、次のように書くことができます:

$$A\,\vec{x} \;=\; \vec{b}$$

ところで、行列 A を下三角行列 L、対角行列 D、上三角行列 U の 3 つに分解して考えると、次のように変形できます: [15]

$$(L+D+U)\,\vec{x} \;=\; \vec{b}$$

$$D\,\vec{x} \;=\; \vec{b} - (L+U)\,\vec{x}$$

$$\vec{x} \;=\; D^{-1}\{\vec{b} - (L+U)\,\vec{x}\}$$

連立 1 次方程式を解くということは、この式を満たす $\vec{x}$ を求めることに等しいわけです。ところで、この式を**漸化式**と考え、適当な $\vec{x_0}$ から始めて次の式により値を計算していくものとします:

$$\vec{x_{i+1}} \;=\; D^{-1}\{\vec{b} - (L+U)\,\vec{x_i}\}$$

[12]さらに、完全に 0 でなかったとしても、非常に 0 に近い (絶対値の小さい) 値だと誤差が出やすくなるので、常に絶対値の大きい行を選んで来てそれで消去を行うのがよいのです。

[13]たとえば $2x+3y=1$ と $200x+300y=100$ とは同じ方程式なので、実際にはまず各行の係数を絶対値の最大が 1 になるように定数倍してそろえておき、それから最大を選択するほうがよいのです。この操作を**スケーリング** (scaling) と呼びます。

[14]ただし、計算量はやや不利になります。

[15]D は対角行列ですから**逆行列** (inverse matrix) は各要素を逆数にするだけで簡単に作れます。

ここで $\vec{x_i}$ が**収束**すれば、つまり値が変化しなくなれば、その値 $\vec{x_*}$ は連立 1 次方程式の解となっています。[16]

これを Ruby プログラムにしてみましょう。上の漸化式の計算は面倒そうですが、よく考えれば次の手順で行えると分かります:

- b_i は $a_{i,n}$ に入っている。
- そこから、$\sum_j a_{j,i}x_j$ を引く (ただし対角要素は飛ばす)。
- 最後に $a_{i,i}$ で割る。

収束は $\vec{x_i}$ の各成分の前回との差の絶対値の和が 10^{-8} 以下になったことで判定し、100 回反復しても収束しない場合はあきらめるようにしました:

```
def jacobi(a)
  n = a.length; x = Array.new(n, 0); count = 0
  while true do
    x1 = Array.new(n, 0); d = 0.0
    n.times do |i|
      v = a[i][n].to_f
      n.times do |j| if j != i then v = v - a[i][j]*x[j] end end
      x1[i] = v / a[i][i]; d = d + (x1[i]-x[i]).abs
    end
    x = x1; count = count + 1
    # p(x)
    if d < 0.000000001 then return x end
    if count > 100 then return nil end
  end
  return x
end
```

では実際に動かしてみましょう:

```
irb> jacobi([[2,3,2,1],[2,5,4,4],[4,8,8,7]])
=> nil
```

あれ、収束しないようです。別のもので試すと…

```
irb> jacobi([[5,2,-1,19],[2,-3,2,-1],[1,1,-2,8]])
=> [2.99999999991689, 0.999999999949859, -1.99999999987636]
```

これは確かに大丈夫のようです。なお、ヤコビ法では「次の」ベクトルを完全に計算し終わってから現在のものと入れ換えますが、上の例で言うと `x1` を使わないで「`x1[i] = …`」を「`x[i] = …`」に直してしまい、計算し終わった成分は以後その新しい値を使う方法を**ガウス–ザイデル法** (Gauss–Seidel method) と呼び、その方が収束が速いことが知られています。

演習 2 上の例題をさまざまなデータで動かし、さらに次の検討を行え。

a. どのような場合に解が収束する/しないかを検討。
b. 途中経過を打ち出させ、収束しない場合の振舞いや、収束する場合どのように収束するかを検討。
c. ガウス–ザイデル法に直した場合、収束までの計算回数や振舞いがどのように違うかを検討。

[16] この漸化式は常に収束するとは限りませんが、行列が**対角優位** (diagonal dominant — 対角成分が他の成分に比べて相対的に大きい) の場合には収束することが知られています。

3　常微分方程式の数値解法

3.1　常微分方程式

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、とくに**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、式が次の形のものを取り上げます:

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

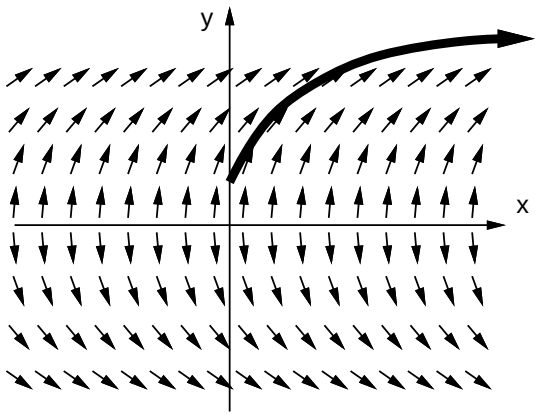

図 3: 常微分方程式と初期値問題

直観的に説明するなら、XY 平面の至るところにおいて傾きを表す矢印が定義されていて、その方向に沿った曲線を表すのが常微分方程式である、という風に考えることができます (図 3)。この曲線を**解曲線** (solution curve) と呼びます。この場合、解曲線は無限にありますが、その一般式を求めたものを**一般解** (general solution) と呼びます。これに対して、**初期値** (initial value) として任意の点 (x_0, y_0) を与えて、そこを通る曲線を求めるとすれば、その曲線は 1 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。

簡単な例として、次の常微分方程式を考えます:

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

これを次のように変形します:

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

次に両辺を不定積分します:

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

両辺とも積分の結果、次のようになります:

$$y^2 \;=\; x + C$$

よって次の式が求まり、これが一般解となります: [17]

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

そして、たとえば点 $(1,0)$ を通る特殊解を求めるなら、上式の y に 1、x に 0 を代入して $C = 1$ が求まるので、$y = \sqrt{x+1}$ が特殊解となるわけです。

[17] 当然 $y = 0$ は除きます。

3.2 オイラー法

前節のような簡単な常微分方程式の初期値問題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多いです。そのような場合には、数値的に解を求める方法が使われます。

一番素朴な方法として、初期値 (x_0, y_0) における解曲線の接線の方向は $\frac{dy}{dx} = f(x, y)$ として分かっているわけ なので、この方向に微小な値 h ぶんだけ動いた点を (x_1, y_1) と し、以下同様にして次々に曲線上の値を (近似的に) 求めていくという方法が考えられます。これを**オイラー法** (Euler method) と呼びます。整理すると、次の式で x_i, y_i を計算していくのがオイラー法です:

$$x_{i+1} = x_i + h$$
$$y_{i+1} = y_i + h\ f(x_i, y_i)$$

これを使って x が指定値になるまで指定した分割数で計算し、最後の y の値を返す Ruby プログラムを示します: [18][19]

```
def euler(xmin, y, xmax, count)
  h = (xmax-xmin).to_f / count
  count.times do |i|
    # x = xmin + h * (i+1)
    y = y + h * 0.5/y  # f(x,y) = 1/2y
  end
  return y
end
```

実際に動かしてみましょう。x として 1 を指定することで、2 の平方根を計算するようにし、刻み数を変えることで精度の変化を見ています:

```
irb> euler(0,1,1,100)
=> 1.41482796736591
irb> euler(0,1,1,1000)
=> 1.41427484589246
irb> euler(0,1,1,10000)
=> 1.41421968916029
```

参照のために、1 桁の平方数でない数の平方根を掲げておきましょう:

$$\sqrt{2} \approx 1.41421356237309504880168872420969807856 96$$
$$\sqrt{3} \approx 1.73205080756887729352744634150587236694 28$$
$$\sqrt{5} \approx 2.23606797749978969640917366873127623544 06$$
$$\sqrt{6} \approx 2.44948974278317809819728407470589139196 59$$
$$\sqrt{7} \approx 2.64575131106459059050161575363926042571 02$$
$$\sqrt{8} \approx 2.82842712474619009760337744841939615713 93$$

double 型の精度は 16 桁くらいなので、もうちょっと精度良く求まってほしいわけですが、オイラー法の場合、「個々の点における」接線を延長して次の点を求めているため、解曲線がまっすぐな所ではよいけれど、カーブしているところではどんどん「外側」にずれていってしまいます。これを補う方法は以下で説明します。

3.3 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、それでは解曲線がカーブしている時に誤差が大きいのでした。そこで、始点と終点の両方で接線の向きを求め、それを平均すれば、より正確になると考えられます。しかし実際には、各ステップの終点は「これから求める」ので分かりません。そこで、オイラー法で終点の近似値を求め、そこの傾きを求め、これと始点での傾きとの平均を取ることを考えます (図 4)。つまり次のようにするわけです:

[18] $(0, 1)$ から始めた 場合、答えは $\sqrt{x+1}$ になるはずでしたね。
[19] この方程式の場合は x を使わない ($f(x, y)$ が x に依存しない) ので、x の計算はコメントアウトしてあります。

$$x_{i+1} = x_i + h$$
$$k1 = h\ f(x_i, y_i)$$
$$k2 = h\ f(x_i + h, y_i + k1)$$
$$y_{i+1} = y_i + \frac{1}{2}(k1 + k2)$$

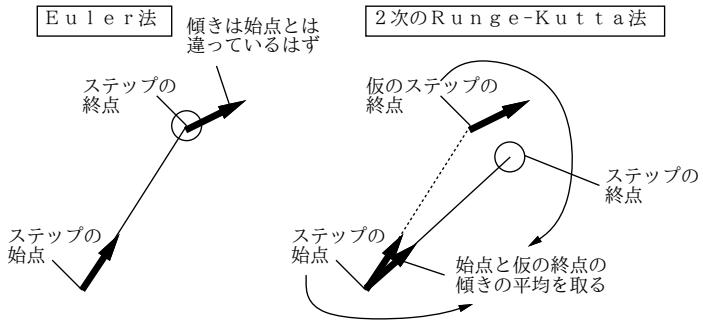

図 4: オイラー法と 2 次のルンゲ–クッタ法

これを **2 次のルンゲ–クッタ法** (2nd order Runge–Kutta method) と呼びます。これを Ruby プログラムにしたものを示しておきます:

```
def rungekutta2(xmin, y, xmax, count)
  h = (xmax-xmin).to_f / count
  count.times do |i|
    # x = xmin + h * (i+1)
    k1 = h * 0.5/y
    k2 = h * 0.5/(y+k1)
    y = y + 0.5*(k1+k2)
  end
  return y
end
```

これによる計算結果を見ると、オイラー法よりずっと精度がよくなっていることが分かります:

```
irb> rungekutta2(0,1,1,100)
=> 1.41421356445272
irb> rungekutta2(0,1,1,1000)
=> 1.41421356237517
irb> rungekutta2(0,1,1,10000)
=> 1.41421356237309
```

さらに正確さを増すため、h の半分だけ上記の方法で進み、そこから終点までを 2 番目の近似値で進み、始点、最初の近似値、2 番目の近似値、終点の近似値を 1 : 2 : 2 : 1 の比率で混ぜる方法 があり、**4 次のルンゲ–クッタ法** (4th-order Runge–Kutta method) として知られています: [20]

$$x_{i+1} = x_i + h$$
$$k1 = h\ f(x_i,\ y_i)$$
$$k2 = h\ f(x_i + \frac{h}{2},\ y_i + \frac{k1}{2})$$
$$k3 = h\ f(x_i + \frac{h}{2},\ y_i + \frac{k2}{2})$$
$$k4 = h\ f(x_i + h,\ y_i + k3)$$
$$y_{i+1} = y_i + \frac{1}{6}(k1 + 2k2 + 2k3 + k4)$$

[20] 単にルンゲ-クッタ法と言った場合はこちらを指すことが多いです。

なぜ 1 : 2 : 2 : 1 の比率がよいかは ここでは説明しませんが、**テイラー展開** (Taylor expansion) による 1 次の項のあてはめがオイラー法、2 次の項まであてはめるのが 2 次のルンゲ–クッタ法、4 次の項まであてはめるのが 4 次のルンゲ–クッタ法となります。

誤差についてもおおまかに説明すると、オイラー法では各ステップの誤差が (1 次まであてはめた残りなので) $O(h^2)$、それを $\frac{1}{h}$ ステップ累計するため全体の誤差は $O(h)$ となります。同様に 2 次のルンゲ–クッタ法では各ステップ $O(h^3)$、全体で $O(h^2)$、4 次のルンゲ–クッタ法では各ステップ $O(h^5)$、全体で $O(h^4)$ となります。これを言い換えると、h を $\frac{1}{2}$ にした時、オイラー法では全体の誤差がほぼ $\frac{1}{2}$ になるのに対し、2 次のルンゲ–クッタ法では $\frac{1}{2^2}$、4 次のルンゲ–クッタ法では $\frac{1}{2^4}$ になります。誤差を減らすのには刻みをどんどん細かくするほどよいかというと、細かくするほど計算時間が掛かるだけでなく、丸め誤差や情報落ち誤差が累積するため、結局精度もある程度以上は上がらなくなります。このため、2 次や 4 次のルンゲ–クッタ法のように最初から性能のよい公式を使うことが重要になるわけです。

演習 3 オイラー法と 2 次のルンゲ–クッタ法のプログラムを打ち込み、これらでさまざまな値の平方根を刻み幅を変えて計算し、上で説明した誤差に関する性質が成り立っているかどうか確認してみよ。

演習 4 さらに 4 次のルンゲ–クッタ法のプログラムも作成し、同様に検討せよ。

A　本日の課題 6A

「演習 1」または「演習 2」の小課題から 1 つ選び、今日中に久野までメールで送ってください。

1. Subject: は「Report 6A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 1」または「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 検討結果のまとめ (とできれば簡単な考察)
5. 以下のアンケートの回答。

   Q1. 連立一次方程式がプログラムで解けるという点を理解しましたか。

   Q2. 常微分方程式についてはどうですか。

   Q3. 本日の全体的な感想と今後の要望をお書きください。

B　次回までの課題 6B

次回までの課題は「演習 1」〜「演習 4」(の小課題) から 2 つ以上選んで報告することです。

各課題のために作成したプログラムは複数ある場合もすべてレポートに掲載してください。期限は「12/3 の授業開始 10 分前」とし、いつも通り久野までメールで送付してください。

1. Subject: は「Report 6B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 1 つ目のプログラムのソース。
4. その説明と分析/考察。
5. 2 つ目のプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 2 次元配列を操作するプログラムが作れるようになりましたか。

   Q2. ここまでに学んだ各種の解法 (数値積分、求解、連立一次方程式、常微分方程式) を、今後 (別の科目等で) 出会う問題に対して使えそうですか。

   Q3. 課題に対する感想と今後の要望をお書きください。