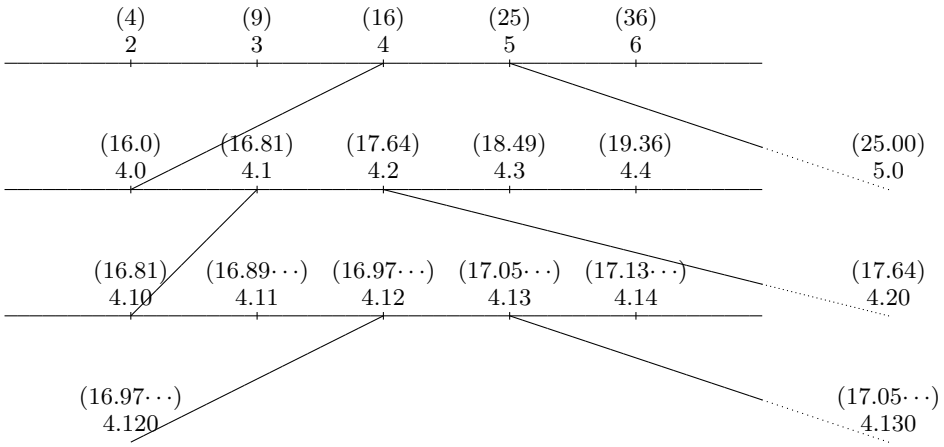
3.3 2次方程式

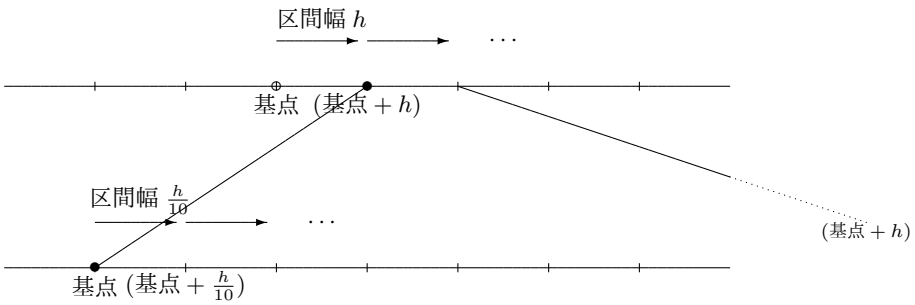
平方根を求める教科書的な方法は、区間圧搾法とでも呼べばよいだろうか。イメージとしては



のように進め、値を絞り込んでいけばよい。もちろん終わりのない作業だ。

同じことを Swift でプログラムを書いたらどうだろう。アルゴリズムは、区間の幅を 1/10 ずつ小さくしていき、小数点以下の値を 1 桁ずつ確定していく。具体的には、図において数直線の左端の値 $4 \to 4.1 \to 4.12 \to \cdots$ が $\sqrt{17}$ のより精確な値を定めることになる。

一般的なアルゴリズムは概（おおむ）ね以下の通りである。参考のため [ ] 内に、上図で用いた数値を使って第一巡目の様子を示しておく。



1. 境界値 [17] を定める
2. 基点 [4] と区間幅 [1] を決める
3. (基点 + 区間幅) [$4 + 1 = 5$] に対する平方 [$5^2 = 25$] を求める
4. (a) 境界値を超えない場合[1]：(基点 + 区間幅) を基点にし、区間幅は変えず、2. へ
   (b) 境界値を超える [$25 > 17$] 場合：基点は変えず、区間幅を 1/10 [$1 \to 0.1$] にし、2. へ

[1] 基点 [4]、区間幅 [0.1] になったとき [$4.1^2 = 16.81 < 17$] なので、[$4 + 0.1 = 4.1$] が次の基点となる。

最初は、小数点以下の値を 1 桁ずつ確定していく方法を素直にプログラムにしてみよう。1 桁確定したら区間の幅を 1/10 ずつ小さくして、ある程度の精度になったら終了だ。$\sqrt{2}$ を求めるプログラムは次のようになるだろうか。ただ、この方法は区間を圧縮するイメージではないだろう。

```
var m = 1.0 // √ 17 なら 4.0 から
var d = 0.1
while d > 0.0000001 {
    while m*m < 2 { // √ 17 なら < 17 から
        m += d
    }
    m -= d
    d /= 10
}
//
print(m)
```

すべきことは、$\sqrt{2}$ に近い値 $m = 1$ から始めて、$m = 1.1, 1.2, 1.3, \ldots$ と値を増やすことである。この場合は 1.4 まで進んだら、次は $m = 1.41, 1.42, \ldots$ と進む。増やす量が徐々に小さくなるので、一旦 $d = 0.1$ としておいて、$d = 0.01, 0.001, 0.0001, \ldots$ とすればよいはずだ。それだけなら

```
    while m*m < 2 {
        m += d
    }
    d /= 10
```

で十分だと思われるが、間に `m -= d` が挿入されているのはなぜだろう。それは、`m*m < 2` のときは必ず `m` の値が増えるけれど、`while` 文を抜けるときの `m` の値は `m*m > 2` になってしまっているからだ。そこで増やしすぎた `m` の値を戻すのである。もし、こういう操作を嫌うなら、`while` の判定を `m*m < 2` ではなく `(m+d)*(m+d) < 2` としておけばよい。つまり、一歩先の値を 2 乗しても 2 を超えないなら `m` を増やす価値がある、ということである。もし、これを関数の形に仕立てるなら、こんな感じだろうか。

[StepBy.playground]

```
 1 func stepby(n: Double) -> Double {
 2     var m = 0.0
 3     var d = 0.1
 4     while d > 0.0000001 {
 5         while (m+d)*(m+d) < n {
 6             m += d
 7         }
 8         d /= 10
 9     }
10     return m
11 }
```

```
//
print(stepby(n: 2))
```

m = 0.0 から始めているのは、もちろん $0 < m < 1$ の平方根も計算できるようにするためである。

ex. **1** 試しに [StepBy.playground] で $\sqrt{3500000000}$ を求めてみよう。すると、明らかに実行に時間がかかるのがわかる。効率よく解を求めるプログラムに改善してみよう。

では、今度は再帰の考えを取り入れて $\sqrt{2}$ の値を求めるプログラムを書いてみることにする。

[Squeeze.playground]

```
import UIKit

var h = 1.0
func  squeeze(s: Double) {
    if h < 1/100000000 {
        print(s)
    } else if f(x: s) * f(x: s+h) > 0 {
        squeeze(s: s+h)
    } else if f(x: s) * f(x: s+h) < 0 {
        h = h/10
        squeeze(s: s)
    } else if f(x: s) == 0 {
        print(s)
    } else if f(x: s+h) == 0 {
        print(s+h)
    }
}
func f(x: Double) -> Double {
    return x*x - 2
}
//
squeeze(s: 1)
```

プログラムの本質部分は以下の数行分と func f(x: Double) である。ここはアルゴリズム 3. の処理とその後の分岐 4. に相当する。

```
if f(x: s) * f(x: s+h) > 0 { // アルゴリズム 4. の分岐 (a)
    squeeze(s: s+h)
} else if f(x: s) * f(x: s+h) < 0 { // アルゴリズム 4. の分岐 (b)
    h = h/10
    squeeze(s: s)
}

func f(x: Double) -> Double { // アルゴリズム 3. のための計算式
    return x*x - 2
}
```

アルゴリズム 1. における境界値は関数 `f` が定めている。式は x^2-2 が与えられているので、$x^2-2=0$ の解が $\sqrt{2}$ を表す。つまり、2 乗した値となる境界値は 2 だ。アルゴリズム 2. における基点はこの関数 `squeeze(s: 1)` の引数 `1` で、区間幅は `h = 1.0` で与えられている。

さて、アルゴリズム 3.–4. の判定–分岐であるが、単に `f(x: s+h) > 2` を満たすかどうかでないことに注意してもらいたい。実は、このあとの単元の先取りになるのだが、$x^2-2=0$ の解がある区間 $[s,\ s+h]$ に含まれることは、x^2-2 のグラフが x 軸上の区間 $[s,\ s+h]$ を横切ることなのである。それは言い換えると、区間の両端での関数値の符号が異なることである。両端での関数値が異符号なら、それらの積は負の値になる。この事実が単に `f(x: s+h) > 2` を判定基準に据えなかった理由だ。

それにこうしておけば、x^2-2 のような式だけでなく、$x^2+2x-5=0$ のような式にも応用できるので、2 次方程式の解を求める役に立つ。そればかりか、x^3+(うんぬん) といった 3 次方程式や 4 次方程式の解まで求められるプログラムになるのだ。すると、`if`〜`else` 文に記述された式が、アルゴリズム 3.–4. を忠実に表していることがわかるだろう。

しかし、3 次方程式の解も求められるとはいえ、近似値になってしまうのはどうなんだろう。たとえば 2 次方程式なら解の公式が使えるわけだから、根号を用いてきっちり正確な解を示したいと考えることは自然だ。そうなると 2 次方程式も、1 次方程式の解を `print` 文で表示させたようにすればよい。それ自体は以前のプログラムを書き換えれば済む。ちょっと `if` 文による分岐を増やす必要はあるだろうが、...というほど実は簡単ではない。

たとえば $x^2+2x-5=0$ の解は $a=1$、$b=2$、$c=-5$ を $x=\dfrac{-b\pm\sqrt{b^2-4ac}}{2a}$ に埋め込めばお終い、とはならない。なぜなら単に埋め込んだけでは $x=\dfrac{-2\pm\sqrt{24}}{2}$ が表示される。$\sqrt{24}=2\sqrt{6}$ にしたいだろうし、その上約分までできる。これは単純に場合分けだけで済む話ではない。

実は、x の係数である b が偶数のときは必ず根号内が簡単になって約分ができる。b が偶数であることを $2b'$ で表すと解の公式は

$$x=\frac{-(2b')\pm\sqrt{(2b')^2-4ac}}{2a}=\frac{-b'\pm\sqrt{b'^2-ac}}{a}\quad ※$$

まで省略できるのだ。こういうとき文字式というのは便利だね。

では、b の偶奇による場合分けを増やせば解決かというとそうでもない。$x^2+4x-4=0$ は※を用いてもなお根号内が簡単になる。そう、根号内の数を簡単にするプログラムを用意しないと解決されないのだ。そうなると、砂遊び程度の活動では収まらないので、詳しくはサイト内にあるファイル “solvswft.code” を見てもらうことにして、ここでは簡単に必要部分を抜き出しておくにとど

めたい。

まずは根号内の数を簡単にするコードだ。

```
func outFromSqrt(d: Int) -> Int {
    var i = d
    while i > 1 {
        if d%(i*i) == 0 { return i }
        i = i-1
    }
    if i == 0 { return 0 } else { return 1 }
}
func leaveInSqrt(d: Int) -> Int {
    var i = d
    while i > 1 {
        if d%(i*i) == 0 { return d/(i*i) }
        i = i-1
    }
    return d
}
//
let n = 24
print("√ \(n) = \(outFromSqrt(d: n))√ \(leaveInSqrt(d: n))")
```

もっとマシなやり方はあるが、単純な方法で解決している。マシなやり方はタプルを使うことだ。そうすれば関数はひとつで済む。この例はふたつの関数を合わせて使う。`n = 24` として実行すると`√ 24 = 2√ 6` が出力される。

ex. 2 タプルを使って根号内の数を簡単にする関数 `extractSqrt(d: Int) -> (Int, Int)` を書いてみよう。また、このままでは`√ 4 = 2√ 1` などと出力されるので、改善してみよう。

また、根号内を簡単にできても約分ができないと意味がない。2 次方程式の解の公式では a、b、$m\sqrt{n}$ の 3 種の値が同時に約分できなくてはならないので、3 数の最大公約数を求めるものが必要だ。次は 3 数の最大公約数を求めるコードだ。

```
func findGCD(A: Int, B: Int, C: Int) -> Int {
    var i = max(abs(A), abs(B), abs(C))
    while i > 1 {
        if A%i == 0 && B%i == 0 && C%i == 0 { return i }
        i = i-1
    }
    return 1
}
//
let a = 6
let b = 18
let c = 12
print("GCD(\(a), \(b), \(c)) = \(findGCD(A: a, B: b, C: c))")
```

最大公約数は再帰を用いて**ユークリッドの互除法**を使うのがよいが、これも単純に済ませている。3 数 6, 18, 12 で実行すると CGD(6, 18, 12) = 6 が出力される。ユークリッドの互除法は 「3.6 **三平方の定理**」で説明しているので参考にしてもらいたい。