代数的実数を作る

荒田 実樹

2017 年 10 月 13 日

目次

第 0 章

イントロダクション

0.1　はじめに：計算機で扱える実数（よもやま話）

計算機で実数を扱うと言った時、皆さんはどのようなものを思い浮かべるだろうか。少しでもプログラミングをやったことのある方ならご存知の通り、計算機で実数を扱う際は浮動小数点数を使うことが多いと思う。しかし、浮動小数点数は単なる近似であり、誤差がつきまとう。例えば、広く使われている 2 進倍精度（精度 53 ビット）の浮動小数点数では、$0.1 + 0.1 + 0.1$ と 0.3 は同じ値にならない。

では、実数を正確に計算するにはどうしたら良いだろうか？浮動小数点数でも近似の桁数を増やせば誤差は減る。とすると、数の計算方法を覚えておいて、必要になったら必要な精度で計算するというのはどうか？

このアイディアを精密化したものが「**計算可能実数**」である。我々が扱うほとんどの実数、$\sqrt{2}, e, \pi$ 及びその四則演算、実数上の有名な関数の多くは計算可能である。（計算可能な関数はせいぜい可算個しかないので、計算可能実数は実数全体のほんの一部でしかない。具体的に「計算可能でない実数」を挙げるとすれば、「（特定のプログラミング言語で書いた）プログラムが停止する確率」であろう→「チャイティンの定数」で検索）

しかし、計算可能実数にも欠点がある。計算可能実数同士の比較演算、特に等しいかどうかは計算可能ではない。直感的だが不正確な言い方をすれば、「2 つの計算可能実数が等しいかどうか判断するには、数の全ての桁を調べる必要があるが、有限時間でそれはできない」となるだろうか。より厳密な説明は、適切な本を参照してほしい。

少し話を後退させよう。「比較演算も含めて決定可能な」実数の部分集合にはどのようなものがあるだろうか？

まず、**整数** $\mathbb{Z}$ は、多倍長計算を駆使すれば、何桁であっても計算機上で正確に取り扱える。C 言語であれば自力で実装するか GMP などのライブラリーを使って多倍長計算を行うことになるが、プログラミング言語によっては、意識せずに多倍長整数を使えるものもある。

次に、**有理数** $\mathbb{Q}$ は、整数 2 つの組で表せるので、計算機上で正確に取り扱える。これもプログラミング言語によっては標準で用意されていたりする。

有理数の有限次代数拡大も良さそうだ。 例えば、$\mathbb{Q}(\sqrt{2})$ であれば $x + y\sqrt{2}$ を有理数 2 つの組 $(x, y) \in \mathbb{Q}^2$ として表せば良い。 $\mathbb{Q}(\sqrt{2})$ の演算は、組 (x, y) に対する演算として実装できる。

さらに言うと、有理数の代数閉包 $\bar{\mathbb{Q}}$ 、すなわち**代数的数**も、計算機で正確に取り扱える。ここでは実数の部分集合についての話をしているから、それに合わせるならば、「**代数的実数**は計算機で正確に取り扱える」となるだろう。

残念ながら e や π は代数的でない（**超越数**）。超越数もある程度取り扱いたい場合は... 筆者の力量を超え

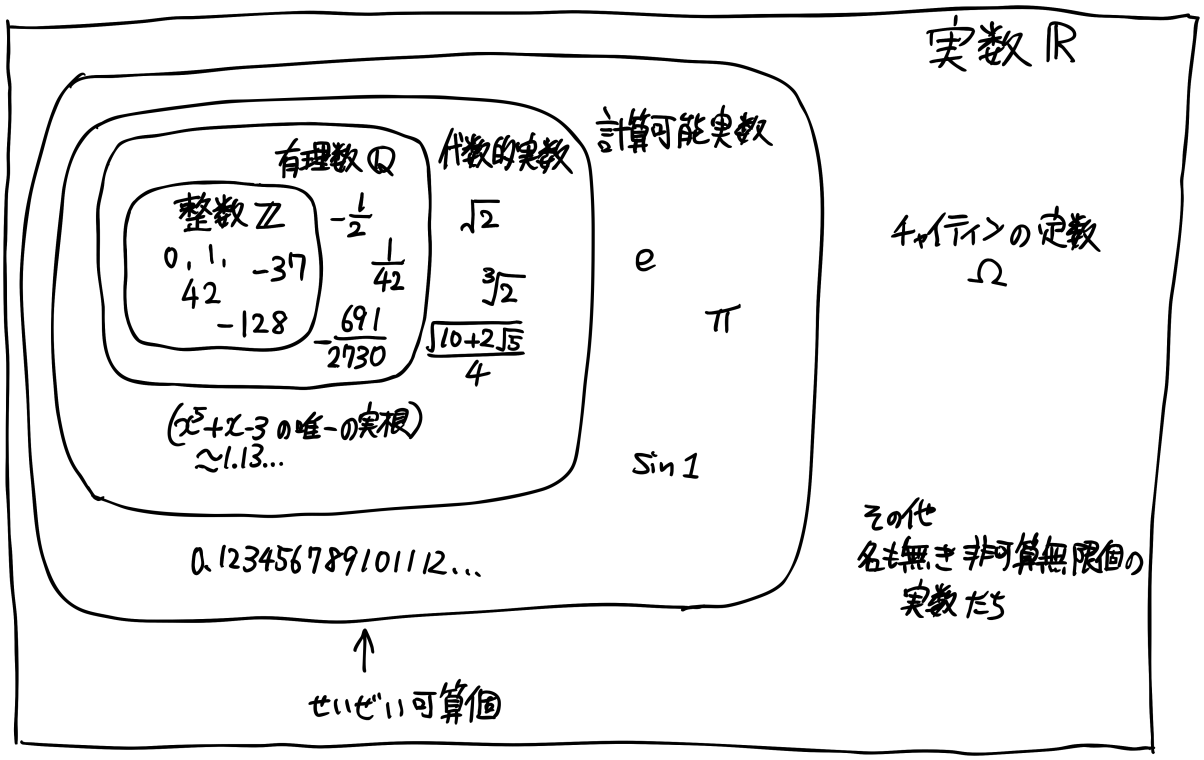

図 1

るので、本を参照して欲しい。

ともかく、代数的実数というのは、四則演算、平方根や立方根などの操作（もっと一般に、有理数係数多項式の実根を取る操作）、そして比較演算が不自由なく行える体系である。特別に扱う価値はあるだろう。そこで、これから何回かに分けて、計算機における代数的実数の実装方法を述べることにする。

（ちなみに、平方根さえ取れれば良い、つまり立方根や一般の多項式の実根を取る操作が必要ないのであれば、「作図可能な数」を実装するという手もある）

0.2　基本的な方針：代数的実数の表し方

代数的実数は、整数係数多項式の根となる実数である。具体的な代数的実数を取り扱う上では、それに伴う多項式が重要である。（この多項式のうち次数が最小のものを、**最小多項式**という。代数的数を根に持つ多項式は、最小多項式で割り切れる）

多項式は一般に複数の根を持つため、表したい代数的実数が多項式の根のうちのどれであるかを、何らかの方法で指示しなくてはならない。指示の方法としては、「小さいものから数えて k 番目」とするか、あるいは十分狭い区間を使って「1.2 より大きくて 1.5 より小さい唯一の根」とする方法がある。ここでは後者を使うことにする。（大雑把な大きさがわかっていると、他の代数的実数と大小関係を比較する際に有利である。）

つまり、ある代数的実数を表すのに

- 最小多項式 f
- 同じ多項式の根どうしを区別できるような、有理数の区間 (a, b) （この区間の中に f の根は 1 個だけ

である）

を使う。 この組 $(f,(a,b))$ について、加減乗除、比較、平方根などの演算を実装していけば良い。

アルゴリズムに関しては、まずは遅くてもいいから素朴な方法で実装し、筆者の余力があったらより洗練された方法を紹介することにする。

0.3 実装に使うプログラミング言語

その辺のプログラミング言語で扱える整数は、32 ビットとか 64 ビットとかの固定長であることが多い。しかし、整数および有理数の正確な取り扱いにはこれでは不足で、多倍長計算を行う必要がある。

多倍長計算はもちろん自力で実装することもできるが、なるべく言語の標準ライブラリーで提供されていることが望ましいことは言うまでもない。また、有理数も、多倍長整数があれば自分で実装することはできるが、標準ライブラリーで提供されていた方が楽である。

多倍長整数と有理数を組み込み型あるいは標準ライブラリーで提供している言語で、筆者の知っているものをいくつか挙げてみる：

- Python (fractions module)
- Ruby (Rational クラス)
- Scheme
- Julia (Arbitrary Precision Arithmetic, Rational Numbers)
- Haskell (Integer 型、 Rational 型)
- OCaml (Module Num)
- Go (math/big)
  - 演算子オーバーロードがない。

標準で多倍長整数はあるが有理数はない言語も、参考までにいくつか挙げておく：

- Java (java.math.BigInteger)
  - 演算子オーバーロードがない。
- Scala (scala.math.BigInt)
- C#, F# (System.Numeric.BigInteger)
- D (std.bigint)

多倍長整数があれば有理数演算の実装は比較的楽なので、これらの言語で実装してみるのも良いだろう。

それ以外に、静的型付き言語で考慮すべき重要な点として、多相（あるいはジェネリクス）の有無がある。今後、第一級の対象として多項式を扱うことになるが、その際の係数は整数、有理数、有限体、多項式などと様々である。多相があれば、多項式に関する操作を係数環ごとに書かなくても、同じコードを使い回せるので、楽である。

この他、演算子オーバーロードもあれば重宝する。

さて、ここで例示するコードであるが、筆者の好みで Haskell を使うことにする。計算のアルゴリズムはソースコードとは独立に説明するつもりなので、他の言語でも同様に実装できるだろう。リクエストがあれば、（あるいはネタ切れを起こしたら）他の言語での実装を載せるかもしれない。

0.3.1 Haskell について

筆者のブログに書いたように、Haskell の Prelude にある Num クラスはお世辞にも洗練されているとは言えない。Num クラスを騙し騙し使う、クラス階層を自作する、Algebraic/AlgebraicPrelude などを使う、などのやり方があるだろう。 当初は Num クラスを使うことにする。

```
$ stack new real-algebraic simple-library
$ cd real-algebraic
$ tree
.
├── LICENSE
├── README.md
├── Setup.hs
├── real-algebraic.cabal
├── src
│ ^^c2^^a0^^c2^^a0 └── Lib.hs
└── stack.yaml

1 directory, 6 files
$ mkdir -p src/Numeric/RealAlgebraic
```

0.4 必要な知識

代数学

0.5 参考文献

- Joachim von zur Gathen and Jrgen Gerhard, *Modern Computer Algebra, Third Edition*, Cambridge University Press, 2013
  - 邦訳もあるようだが値段が（3 倍くらい）高めで、古い。買うなら原著だろう。
- Donald E. Knuth 著「The Art of Computer Programming Volume 2 Seminumerical Algorithms Third Edition 日本語版」アスキードワンゴ、2015 年
  - Knuth の同名の本の邦訳。
- 穴井宏和、横山和宏著「QE の計算アルゴリズムとその応用　数式処理による最適化」東京大学出版会、2011 年
  - 代数的実数について触れている。
- 吉永正彦 著「周期と実数の 0-認識問題 Kontsevich-Zagier の予想」数学書房、2016 年
  -「代数的実数よりも広くて、比較演算が決定可能な数のクラスはあるか」という問題がテーマである。計算可能実数の等号認識問題が計算不可能である件にも触れている。

第 1 章

一変数多項式環

1.1 一変数多項式環

代数的数を扱う上では、当然、多項式を操作することが必要になる。 #0 では整数係数、あるいは有理数係数の多項式に言及したが、追い追い、多項式環自身や、有限体などを係数とする多項式環を扱うことになる。そこで、多項式環の係数環は、一般の整域 R とする。

整域 R 上の一変数多項式環 (univariate polynomial) $R[x]$ は、不定元 x について $a_n x^n + \cdots + a_1 x + a_0 \quad (a_i \in R)$ の形で表される元全体である。

$R[x]$ の演算としては、まず和、差、積の環演算が挙げられる。それから、R の元による乗算（スカラー倍）や、多項式除算などもある。今後使うものをリストアップしてみよう：

- 環演算：和、差、積
- スカラー倍
- 多項式除算（余りつきの除算）
- 値の計算（x に R の元を代入する）
- 最大公約元 (GCD)
- 形式微分
- （多項式の因数分解もいずれ必要になるが、次回以降に回す）

1.1.1 多項式の表し方

多項式の表し方であるが、素朴に、係数の列として表すことにする。つまり、n 次の多項式の場合は長さ $n+1$ の列を使い、定数項を 0 番目、最高次の係数 (leading coefficient) を n 番目に格納する。 多項式がゼロの場合は長さ 0 の列を使う。

係数の列を表すのには、普通のプログラミング言語だと配列を使うところだろう。配列を直接触っても良いし、配列をラップした型を用意しても良い。

多項式に関する操作は、静的型をもつ言語で実装する場合は多相関数として実装することになるだろう。環演算を抽象化するために、型クラスのような機構があれば申し分ない。

Haskell には標準でリスト型 `[a]` と Data.Array モジュールの `Array` 型があるが、前者はランダムアクセスが苦手、後者は多次元も扱えるように一般化されすぎていて 1 次元では却って扱いづらい。そのため、ここ

では vector パッケージの Data.Vector モジュールで提供される `Vector` 型を使うことにする。`Vector` 型に対しては、リスト同様の使い勝手と、 `Array` 同様のランダムアクセス性能を期待できる。

```
module Numeric.RealAlgebraic.UniPoly where
import qualified Data.Vector as V
import Data.Vector ((!))

-- 一変数多項式 (univariate polynomial)
newtype UniPoly a = UniPoly (Vector a)
  deriving (Eq,Show)

-- 多項式としてのゼロ
zeroP :: UniPoly a
zeroP = UniPoly V.empty

-- 定数項のみの多項式
constP :: (Eq a, Num a) => a -> UniPoly a
constP 0 = zeroP
constP a = UniPoly (V.singleton a)

-- 不定元 (indeterminate)
ind :: (Num a) => UniPoly a
ind = UniPoly (V.fromList [0,1])

-- 係数のリストから多項式を作る
-- 具体的には、最高次の係数が 0 にならないようにリストの後ろの方を取り除く
fromCoeff :: (Eq a, Num a) => V.Vector a -> UniPoly a
fromCoeff xs
  | V.null xs      = zeroP
  | V.last xs == 0 = fromCoeff (V.init xs)
  | otherwise      = UniPoly xs

-- 多項式の次数
-- ゼロの場合は Nothing を返す。
degree :: UniPoly a -> Maybe Int
degree (UniPoly xs) = case V.length xs - 1 of
  -1 -> Nothing
  n -> Just n

-- 多項式の次数
```

```haskell
-- ゼロの場合はエラーとする。
degree' :: UniPoly a -> Int
degree' (UniPoly xs) = case V.length xs of
  0 -> error "degree': zero polynomial"
  n -> n - 1
```

1.1.2 和、差、積、スカラー倍

和、差、積は素朴に筆算の通りに実装する。積に関しては必ずしもこれが最適な方法ではないが、簡単のためにこうする。

```haskell
instance (Eq a, Num a) => Num (UniPoly a) where
  negate (UniPoly xs) = UniPoly $ V.map negate xs

  UniPoly xs + UniPoly ys
    | n < m = UniPoly $ V.accumulate (+) ys (V.indexed xs)
    | m < n = UniPoly $ V.accumulate (+) xs (V.indexed ys)
    | n == m = fromCoeff $ V.zipWith (+) xs ys
    where n = V.length xs
          m = V.length ys

  -- naive method
  UniPoly xs * UniPoly ys
    | n == 0 || m == 0 = zeroP
    | otherwise = UniPoly $ V.generate (n + m - 1) (\i -> sum [(xs ! j) * (ys ! (i - j)) | j <- [0..
    where n = V.length xs
          m = V.length ys

  fromInteger n = constP $ fromInteger n

  -- the ones that should be kicked out of 'Num' class
  abs = error "abs on a polynomial is nonsense"
  signum = error "signum on a polynomial is nonsense"
```

スカラー倍も、リストのそれぞれの要素を a 倍するだけだから簡単だろう。

```haskell
-- scalar multiplication
scaleP :: (Eq a, Num a) => a -> UniPoly a -> UniPoly a
scaleP a (UniPoly xs)
  | a == 0 = zeroP
  | otherwise = UniPoly $ V.map (* a) xs
```

1.1.3 値の計算

多項式の不定元に R の元 t を代入したものを、定義式の通りに

$$f(t) = a_n \cdot t^n + a_{n-1} \cdot t^{n-1} + \cdots + a_1 \cdot t + a_0$$

と計算するのは効率的ではない。 **ホーナー法** (Horner's method) を使うと、乗算の回数を削減できる：

$$f(t) = (\cdots (a_n \cdot t + a_{n-1}) \cdot t + \cdots) \cdot t + a_0$$

Haskell では `foldr` 関数を使うと簡単にホーナー法を実装できる（多項式を、定数項が最初、最高次の係数が末尾として表す場合）。

```
valueAt :: (Num a) => a -> UniPoly a -> a
valueAt t (UniPoly xs) = V.foldr' (\a b -> a + t * b) 0 xs
```

1.1.4 合成

多項式 f に別の多項式 g を合成する演算もある。

```
-- 'f `compP` g = f(g(x))'
compP :: (Eq a, Num a) => UniPoly a -> UniPoly a -> UniPoly a
compP (UniPoly xs) g = V.foldr' (\a b -> constP a + g * b) 0 xs
```

1.1.5 除算

除算も、筆算のアルゴリズムで実装する。

```
divModP :: (Eq a, Fractional a) => UniPoly a -> UniPoly a -> (UniPoly a, UniPoly a)
divModP f g
  | g == 0    = error "divModP: divide by zero"
  | degree f < degree g = (zeroP, f)
  | otherwise = loop zeroP (scaleP (recip b) f)
  where
    g' = toMonic g
    b = leadingCoefficient g
    -- invariant: f == q * g + scaleP b p
    loop q p | degree p < degree g = (q, scaleP b p)
             | otherwise = let q' = UniPoly (V.drop (degree' g) (coeff p))
                           in loop (q + q') (p - q' * g')

divP f g = fst (divModP f g)
modP f g = snd (divModP f g)
```

整数と多項式の余りつきの除算を統一的に扱えると嬉しいが、Haskell 標準の型クラスではそのようにできない。

1.1.6 最大公約元

最大公約元の計算は、整数と同じ**ユークリッドの互除法** (Euclidean algorithm) が使える。

```
gcdP :: (Eq a, Fractional a) => UniPoly a -> UniPoly a -> UniPoly a
gcdP f g | g == 0    = f
         | otherwise = gcdP g (f `modP` g)
```

ただし、計算の途中に係数が複雑な有理数になる場合があるので、その都度モニック多項式に変換するという変種も用意しておく。

```
monicGcdP :: (Eq a, Fractional a) => UniPoly a -> UniPoly a -> UniPoly a
monicGcdP f g | g == 0    = f
              | otherwise = monicGcdP g (toMonic (f `modP` g))
```

1.1.7 形式微分

形式微分は、定数項を取り除いて前に詰め、 n 次の項の係数だったものに n を掛ければ良い。

通常の微分は、実数上の関数に対して行うが、形式微分は一般の環に対して定義できる。なお、一般の環では、微分が消えたからといって多項式がゼロとは限らないので注意されたい。

```
diffP :: (Eq a, Num a) => UniPoly a -> UniPoly a
diffP (UniPoly xs)
  | null xs = zeroP
  | otherwise = fromCoeff $ V.tail $ V.imap (\i x -> fromIntegral i * x) xs
```

1.1.8 無平方成分

今後しばしば出てくる、多項式の**無平方成分**の計算も実装しておく。

多項式 $f(x)$ が異なる既約多項式の積として $f(x) = f_1(x)^{l_1} \dots f_m(x)^{l_m}$ と因数分解されるとしよう。この時、指数 l_i を全て 1 に変えたもの $f_1(x) \dots f_m(x)$ を f の**無平方成分** (square-free part) という。また、すでに指数 l_i が全て 1 であれば f は**無平方** (square-free) であるという。

今後、多項式が平方成分を持っているとアルゴリズムが適用できないということがあるので、そういう場合は l_i を全て 1 にした多項式、つまり無平方成分を計算しておく。その無平方成分の計算であるが、形式微分と GCD の計算を組み合わせればすぐである：

$$\mathrm{squareFree}(f) = f/\gcd(f, f')$$

原理は自分で考えて欲しい。

```
squareFree :: (Eq a, Fractional a) => UniPoly a -> UniPoly a
```

```
squareFree f = f `divP` gcdP f (diffP f)
```

第 2 章

実根の数え上げ

今回は、多項式の根を全て求める操作、および、多項式の根の存在範囲を精密化する操作を実装する。

2.1 実根の数え上げ

「多項式の根を全て求める」操作、および「多項式の根の存在範囲をより精密に求める」操作には、特定の区間にある実根の個数を数えるアルゴリズムが存在する。

多項式 $f \in \mathbb{Q}[x]$ と有理数の区間 (a, b) が与えられた時、その区間における f の実根の個数を決定する。簡単のため、端点 a, b は f の根ではないとする。 また、 f は重根を持たないとする。

2.2 スツルムの定理

多項式 $f \in \mathbb{R}[x]$ は無平方であるとする。

f の**スツルム列** (Sturm sequence, またはスツルム鎖 Sturm chain) $\{f_0, \ldots, f_l\}$ を次のように定める：

- $f_0 = f$
- $f_1 = f'$
- $1 \leq i$ について $f_{i-1} = q_i f_i - f_{i+1}$, $\deg f_{i+1} < \deg f_i$ （つまり、f_{i+1} は f_{i-1} の f_i による剰余の符号を逆転させたもの）
- $f_{l+1} = 0$

スツルム列について、次の性質が成り立つ（各自で確かめて欲しい）：

- f_i と f_{i+1} は互いに素である。特に、 f_l は定数である。
- f_i は重根を持たない。
- 実数 a について $f_i(a) = 0$ であれば、 $f_{i-1}(a) f_{i+1}(a) < 0$ である。

実数列 $\{a_0, \ldots, a_l\}$ の**符号の変化の数**とは、

i. 列から 0 を取り除いた時の、文字通りの符号の変化の数（0 を取り除いて得られた列を $\{\tilde{a}_0, \ldots, \tilde{a}_m\}$ とした時の、 $\tilde{a}_i \tilde{a}_{i+1} < 0$ となる i の個数)、もしくは

ii. $a_i > 0, a_{i+1} \leq 0$ または $a_i < 0, a_{i+1} \geq 0$ となる i の個数

のことである。 今回扱う列は、列の途中で 0 となる場合 ($a_i = 0$) は必ずその前後で符号が変化する ($a_{i-1}a_{i+1} < 0$) という性質を持つので、「0 を挟んで同符号となる」場合は考えなくて良い（つまり、 i. ii. どちらの定義を採用しても同じ)。

定理 (Sturm's theorem). 多項式 $f \in \mathbb{R}[x]$ は無平方であるとし、 f のスツルム列を $\{f_0, \ldots, f_l\}$ とする。実数 t について自然数 $w(t)$ を、実数列 $\{f_0(t), \ldots, f_l(t)\}$ の符号の変化の数と定める。 $w(t)$ は単調減少であり、実数 $b < c$ について $w(b) - w(c)$ は 半開区間 $(b, c]$ における f の実根の個数を与える。

証明． w が変化しうるのは、ある i について $f_i(a) = 0$ となるような a においてである。 そこで、$f_i(a) = 0$ となるような任意の i と $a \in \mathbb{R}$ について、

- $i = 0$ ならば a の前後で w が減少する（$\{f_0(t), f_1(t)\}$ の符号の変化の数が $t < a$ と $a \leq t$ で異なる）
- $i > 0$ ならば、 f_i に起因する w の変化は起こらない（同じ a に関して $f_0(a) = 0$ となって w が変化する可能性は否定しない）

ことを示す。

$i = 0$ の場合。 f は重根を持たないので $f'(a) > 0$ または $f'(a) < 0$ のいずれかである。 それぞれの場合について、 f と f' の a の近傍における符号を表にしてみよう。 ただし、 ε は十分小さい正の数とする。

$f'(a) > 0$ の場合：

t	$a-\varepsilon$	a	$a+\varepsilon$
$f(t)$	$-$	0	$+$
$f'(t)$	$+$	$+$	$+$

$f'(a) < 0$ の場合：

t	$a-\varepsilon$	a	$a+\varepsilon$
$f(t)$	$+$	0	$-$
$f'(t)$	$-$	$-$	$-$

いずれの場合も、部分列 $\{f_0(t), f_1(t)\}$ の符号の変化の数は、$t < a$ と $t \geq a$ で 1 から 0 と減少することが表からわかる。

$i > 0$ の場合。 f_i と f_{i-1} は互いに素なので、$f_{i-1}(a) \neq 0$ である。 a の前後で f_{i-1} と f_{i+1} の符号は変化しないこと、そして $f_{i-1}(a)$ と $f_{i+1}(a)$ の符号は逆であることに注意する。

$f_{i-1}(a) > 0$ と $f_{i-1}(a) < 0$ のそれぞれの場合について、f_{i-1}, f_i 及び f_{i+1} の a の近傍における符号を表にしてみよう。 ただし、 ? には、それぞれ任意の符号が入る。

$f_{i-1}(a) > 0$ の場合：

t	$a-\varepsilon$	a	$a+\varepsilon$
$f_{i-1}(t)$	$+$	$+$	$+$
$f_i(t)$	?	0	?
$f_{i+1}(t)$	$-$	$-$	$-$

$f_{i-1}(a) < 0$ の場合：

t	$a-\varepsilon$	a	$a+\varepsilon$
$f_{i-1}(t)$	$-$	$-$	$-$
$f_i(t)$	?	0	?
$f_{i+1}(t)$	$+$	$+$	$+$

いずれの場合も、部分列 $\{f_{i-1}(t), f_i(t), f_{i+1}(t)\}$ の符号の変化の数は $t = a$ の近傍で 1 のまま変化しないことが表からわかる。

以上が証明の要点である。端点 b, c の扱いなど、細かい部分は自分で詰めていただきたい。 証明終わり。

なお、スツルム列の要素 f_i は正の実数倍しても定理の適用に支障はない。特に、係数の膨張を抑えるために、剰余の計算の際に（± 1 倍を除いた）モニック多項式を利用しても構わない。

TODO: 実際の多項式においてスツルム列と符号の変化を図示する。

2.3 根の限界

有限の区間における根の個数を求めるアルゴリズムがあるのはわかった。多項式の全ての根を決定するには、実根の上界と下界を求めれば良い。ここでは、簡単な方法を紹介する。

定理. 多項式 $f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \in \mathbb{R}[x]$ に対し、

$$M = \max\left\{\frac{|a_{n-1}|}{|a_n|}, \ldots, \frac{|a_1|}{|a_n|}, \frac{|a_0|}{|a_n|}\right\}$$

とおく。 この時、 f の実根の絶対値は $M+1$ 未満である。

証明. $|t| \geq M+1$ の時に $f(t)$ が 0 でないことを示す。

$$\begin{aligned}\left|\frac{a_{n-1}}{a_n}t^{n-1} + \cdots + \frac{a_1}{a_n}t + \frac{a_0}{a_n}\right| &\leq \left|\frac{a_{n-1}}{a_n}\right| \cdot |t|^{n-1} + \cdots + \left|\frac{a_1}{a_n}\right| \cdot |t| + \left|\frac{a_0}{a_n}\right| \\ &\leq M(|t|^{n-1} + \cdots + |t| + 1) \\ &\leq M\frac{|t|^n - 1}{|t| - 1} \\ &\leq |t|^n - 1\end{aligned}$$

より、

$$f(t) = a_n\left(t^n + \frac{a_{n-1}}{a_n}t^{n-1} + \cdots + \frac{a_1}{a_n}t + \frac{a_0}{a_n}\right) \neq 0$$

である。

2.3.1 正負の無限大における符号の変化

$t \to \pm\infty$ における多項式の符号は、次数 n と最高次の係数 a_n を使って

$$\mathrm{sign}(f(t)) = \begin{cases}\mathrm{sign}(a_n) & (t \to +\infty) \\ (-1)^n\mathrm{sign}(a_n) & (t \to -\infty)\end{cases}$$

と表現できる。

特に、根の限界における符号は、多項式に値を代入することなく求められる。

2.4 実根の数え上げの実装

第 3 章

代数的数の演算

3.1　代数的数の加減乗除

代数的数どうしの加減乗除を考える。

代数的数 α と β の定義多項式を

$$f(X) = a(X-\alpha_1)\cdots(X-\alpha_n),$$
$$g(X) = b(X-\beta_1)\cdots(X-\beta_m)$$

とおく。 f, g が既約であれば、 α_i は α の共役、$beta_j$ は β の共役だる。

このとき、 $\alpha+\beta, \alpha-\beta, \alpha\beta, \alpha/\beta$ はそれぞれ

$$h_{\alpha+\beta}(X) = \prod_{i=1}^{n}\prod_{j=1}^{m}(X-(\alpha_i+\beta_j)) = (X-(\alpha_1+\beta_1))\cdots(X-(\alpha_n+\beta_m))$$
$$h_{\alpha-\beta}(X) = \prod_{i=1}^{n}\prod_{j=1}^{m}(X-(\alpha_i-\beta_j)) = (X-(\alpha_1-\beta_1))\cdots(X-(\alpha_n-\beta_m))$$
$$h_{\alpha\beta}(X) = \prod_{i=1}^{n}\prod_{j=1}^{m}(X-\alpha_i\beta_j) = (X-\alpha_1\beta_1)\cdots(X-\alpha_n\beta_m)$$
$$h_{\alpha/\beta}(X) = \prod_{i=1}^{n}\prod_{j=1}^{m}(X-\alpha_i/\beta_j) = (X-\alpha_1/\beta_1)\cdots(X-\alpha_n/\beta_m) \quad (\beta\neq 0)$$

の根となる。 一般に α_i や β_j は R の元とは限らないが、それらの対称式は R の元となる。 上に挙げた多項式 $h_{\alpha\star\beta}$ の係数は α_i および β_j の対称式で書けるので、 R 上の多項式である。

3.1.1　例題

$\alpha=\sqrt{2}, \beta=\sqrt{3}$ のとき、それぞれの最小多項式は $f(X)=X^2-2, g(X)=X^2-3$ である。まずは感覚を掴むため、$\alpha+\beta$ の最小多項式を、自分の手で計算してみよう。

それができたら、 $\alpha=\sqrt{2}, \beta=\sqrt[3]{2}$ でやってみよう。

...

手計算では手に負えないと感じて頂けただろうか？

$h_{\alpha\star\beta}$ の係数を α_i および β_j の対称式で書いて根と係数の関係を使って... と計算してやるのは、アルゴリズムを書き下すのすら大変なので、もっと機械的に計算しやすい方法があると良い。

3.2 終結式

多項式 $f(X) = a_nX^n + \cdots + a_0 \in R[X]$, $g(X) = b_mX^m + \cdots + b_0 \in R[X]$ に対し、 **Sylvester 行列** $\mathrm{Syl}(f,g)$ と**終結式** (resultant) $\mathrm{res}(f,g)$ を次のように定義する：

$$\mathrm{Syl}(f,g) = \begin{pmatrix} a_n & & 0 & b_m & & & 0 \\ \vdots & \ddots & & \vdots & \ddots & & \\ \vdots & & a_n & b_0 & & \ddots & \\ a_0 & & \vdots & & \ddots & & b_m \\ & \ddots & \vdots & & & \ddots & \vdots \\ 0 & & a_0 & 0 & & & b_0 \end{pmatrix},$$

$$\mathrm{res}(f,g) = \det \mathrm{Syl}(f,g)$$

$\mathrm{Syl}(f,g)$ は $n+m$ 次の正方行列である。

Sylvester 行列も終結式も一変数多項式について定義されるものであるから、f,g が多変数の多項式は、どの変数についての（多項式係数）多項式と見るかを明示する必要がある。ここでは単に、消去される変数を右下に書くことにする。 つまり、$f,g \in R[X,Y]$ を $f,g \in R[X][Y]$ と見る場合の終結式は $\mathrm{res}_Y(f,g)$ と書き、これは $R[X]$ の元である。

次数が n 以下の多項式の全体（$n+1$ 次元線形空間）を $R[X]^{(n)}$ と書けば、 $\mathrm{Syl}(f,g)$ は線形写像

$$\begin{array}{ccc} R[X]^{(m-1)} \oplus R[X]^{(n-1)} & \longrightarrow & R[X]^{(m+n-1)} \\ (s,t) & \longmapsto & sf+tg \end{array}$$

の表現行列であり、 $\mathrm{res}(f,g)$ が非 0 であることはこの線形写像が同型であることと同値である。

多項式 $a_nX^n + \cdots + a_0$ を列ベクトル $(a_n,\ldots,a_0)^T$ と同一視すれば、

$$\mathrm{Syl}(f,g) = (X^{m-1}f, X^{m-2}f, \ldots, Xf, f, X^{n-1}g, X^{n-2}g, \ldots, Xg, g),$$
$$\mathrm{res}(f,g) = \det(X^{m-1}f, X^{m-2}f, \ldots, Xf, f, X^{n-1}g, X^{n-2}g, \ldots, Xg, g),$$

と書ける。

便宜上、 f または g が 0 の場合は $\mathrm{res}(f,g) = 0$ と定める。

定理. f, g が非自明な共通因数を持つ（$\gcd(f,g)$ が定数でない）とき、かつその時に限り $\mathrm{res}(f,g) = 0$.

定理（終結式と根の関係）. $f(X) = a(X-\alpha_1)\cdots(X-\alpha_n)$, $g(X) = b(X-\beta_1)\cdots(X-\beta_m)$ の時、

$$\mathrm{res}(f,g) = a^m \prod_{i=1}^{n} g(\alpha_i) = (-1)^{nm} b^n \prod_{j=1}^{m} f(\beta_j) = a^m b^n \prod_{i=1}^{n}\prod_{j=1}^{m} (\alpha_i - \beta_j).$$

証明. 根 α_i, β_j が不定元、つまり $\tilde{f}, \tilde{g} \in R[A_1,\ldots,A_n,B_1,\ldots,B_m][X]$, $\tilde{f}(X) = a(X-A_1)\cdots(X-A_n)$, $\tilde{g}(X) = b(X-B_1)\cdots(X-B_m)$ として考える。

各 i,j について、仮に $A_i = B_j$ とすれば、終結式の基本性質より $\mathrm{res}_X(f,g) = 0$ となる。 すなわち、$\mathrm{res}_X(\tilde{f},\tilde{g})$ は $A_i - B_j$ で割り切れる。

これが全ての i,j の組み合わせについて成り立つので、$\mathrm{res}_X(\tilde{f},\tilde{g})$ は $\prod_{i=1}^{n}\prod_{j=1}^{m}(A_i - B_j)$ で割り切れる。つまり、ある $c \in R[A_1,\ldots,A_n,B_1,\ldots,B_m]$ について

$$\mathrm{res}_X(\tilde{f},\tilde{g}) = c\prod_{i=1}^{n}\prod_{j=1}^{m}(A_i - B_j)$$

である。

全ての B_j に 0 を代入した時、右辺は $c(B_j=0)\prod_{i=1}^{n}A_i^m$ となる。 左辺は、行列式による定義より直接計算できて、

$$\det\begin{pmatrix} a_n & & 0 & b_m & & 0 \\ \vdots & \ddots & & 0 & \ddots & \\ \vdots & & a_n & \vdots & \ddots & \ddots \\ \vdots & & \vdots & 0 & & b_m \\ a_0 & & \vdots & & \ddots & 0 \\ & \ddots & \vdots & & & \vdots \\ 0 & & a_0 & 0 & & 0 \end{pmatrix} = (-1)^{nm}a_0^m b_m^n$$

となる。 $a_0=(-1)^n a_n\prod_{i=1}^{n}A_i$ なので、

$$a_n^m b_m^n\prod_{i=1}^{n}A_i^m = c(B_j=0)\prod_{i=1}^{n}A_i^m$$

であり、

$$c(B_j=0)=a_n^m b_m^n$$

がわかる。

同様に、 $c(A_i=0)=a_n^m b_m^n$ もわかるので、 $c=a_n^m b_m^n\in R$ である。

よって、

$$\mathrm{res}_X(\tilde{f},\tilde{g})=a_n^m b_m^n\prod_{i=1}^{n}\prod_{j=1}^{m}(A_i-B_j)$$

である。

具体的な f と g については、環準同型 $A_i\mapsto\alpha_i$, $B_j\mapsto\beta_j$ を適用すれば

$$\mathrm{res}(f,g)=a^m b^n\prod_{i=1}^{n}\prod_{j=1}^{m}(\alpha_i-\beta_j).$$

を得る。 残りの 2 つの等式については省略する。

3.2.1 終結式の計算

命題. $n=\deg f$, $m=\deg g$ のとき、

$$\mathrm{res}(f,g)=(-1)^{nm}\,\mathrm{res}(g,f).$$

命題. $f,g,s,t\in R[X]$ に対し、$\deg(tg)\le\deg f$ のとき $r=f-tg$ と置いて

$$\mathrm{res}(f,g)=(-1)^{(\deg f-\deg r)\deg g}\,\mathrm{lc}(g)^{\deg f-\deg r}\,\mathrm{res}(r,g).$$

また、$\deg(sf)\le\deg g$ のとき $r=g-sf$ と置いて

$$\mathrm{res}(f,g)=\mathrm{lc}(f)^{\deg g-\deg r}\,\mathrm{res}(f,r).$$

3.3　終結式による代数的数の加減乗除

定理.

$$
\begin{aligned}
h_{\alpha+\beta}(X) &= (-1)^{nm}\operatorname{res}_Y(f(X-Y),g(Y)) = (-1)^{nm}\operatorname{res}_Y(g(X-Y),f(Y)),\\
h_{\alpha-\beta}(X) &= (-1)^{nm}\operatorname{res}_Y(f(X+Y),g(Y)) = \operatorname{res}_Y(g(Y-X),f(Y)),\\
h_{\alpha\beta}(X) &= (-1)^{nm}\operatorname{res}_Y(Y^m g(X/Y),f(Y)),\\
h_{\alpha/\beta}(X) &= b_0^{-n}\operatorname{res}_Y(X^m g(Y/X),f(Y)).
\end{aligned}
$$