電気303/電情303 数値解析 (3)

代数方程式の解法

代数方程式

- 代数方程式は非線形方程式の一種であるが…
- 非線形方程式 $f(x) = 0$ において $f(x)$ を x の多項式に限定したものを代数方程式という.
- 以下, 変数が複素数である「雰囲気」というを出すために, 変数を z に変える.

- 複素変数 z の多項式

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

に対して, $f(z) = 0$ の複素解を求めることを, 代数方程式を解くという.

- 解のことを根とも呼ぶ (類義語).
- 最高次の係数を a_0 とする流儀と, 最高次の係数を a_n とする流儀がある.

- なぜ代数方程式を解きたいかというと …
- たとえば, 回路, 制御, 信号処理などの分野では, 伝達関数

$$\frac{B(s)}{A(s)} = \frac{b_0 s^m + b_1 s^{m-1} + \cdots + b_0}{s^n + a_1 s^{n-1} \cdots + a_n}$$

  で表現されたシステムを取り扱う.
- システムの安定性は, 分母多項式の根をすべて求めれば (代数方程式を解けば) 判定できる.

- 代数方程式の求解が一般の非線形方程式の求解と異なる点は …
  - ▷ 複素数の範囲で解を求める必要がある.
  - ▷ しばしば, 解をすべて求める必要がある.
  - ▷ 多項式の係数から解が存在する範囲を見積ることができる.

- このため, 代数方程式の実用的な数値解法は, 一般の非線形方程式の数値解法とは異なる形で発展している.
- 実用的な解法の詳細は学部生の講義で述べるには専門的過ぎるので, この講義では, まず Scilab で代数方程式を解く方法を述べ, 続いて代数方程式を解くときの注意点と, 解法の初歩を順に述べる.

Scilab における多項式

- Scilab は 1 変数多項式しか取り扱えない (多変数多項式を取り扱うときには Maple や Mathematica などが必要).
- 多項式の変数は任意.
- 今回の講義では 多項式の定義の仕方と 求解, 因数分解などの基本的な機能を説明する

多項式を定義する

- 多項式を定義するには 関数 `poly` を使う.
- `poly` には 3 個の機能がある:

  1. 根を指定して多項式を定義する
  2. 係数を指定して多項式を定義する
  3. 行列の特性多項式を定義する

`poly`の使用法1　`p=poly([1 2 3],'z')`とすると,
多項式

$$(z-1)(z-2)(z-3) = z^3 - 6z^2 + 11z - 6$$

が生成される.

Scilab の実行画面は以下の通り (-->は Scilab のプロンプト).

```
-->p=poly([1 2 3],'z')
 p  =
                2   3
  - 6 + 11z - 6z + z
```

- この使用法では, poly の第 1 引数に多項式の根 (のリスト), 第 2 引数に多項式の変数に割り振る文字を与える.
- 第 1 引数に根のリストを与えるときには, [1 2 3] のように全体を [] で囲い, 数値 (あるいは数値が代入された変数) を空白で区切ってならべる (長さは任意). コンマで区切ってもよい.

- 第 1 引数に根をひとつだけ与えるときは [] は不要. たとえば, 多項式 $z-1$ を生成するとき (根は 1 のみ) には,

  ```
  p=poly(1,'z')
  ```

  とする.

- 使用例は次ページの通り.

```
-->p=poly(1,'z')
 p  =
  - 1 + z
```

- 'z' は"z"としてもよい．文字は任意．習慣的には x, z, s, t などをよく使う．

poly の使用法 2　p=poly([4 5],'z','coeff') とすると, 多項式

$$4 + 5z$$

が生成される. Scilab の実行画面は以下の通り.

```
-->p=poly([4 5],'z','coeff')
 p  =

    4 + 5z
```

- この使用法では, poly の第 1 引数に多項式の係数のリスト (低次から順), 第 2 引数に多項式の変数に割り振る文字, 第 3 引数に'coeff'という文字列を与える.
- 第 3 引数を指定することで, 使用法 1 と挙動が変わる.
- 第 1 引数の長さが任意であること, 文字が何でもよいことは, 使用法 1 と同じ.

poly の使用法 3

- 行列 A が正方行列であるとき, p=poly(A,'z') とすると, 行列 A の特性多項式

$$\det(zI - A)$$

が求められる.

- 行列 A が非正方行列である場合には, $\det(zI - A)$ は計算できないのであるが, Scilab 6.1.0 では根のリストが行列の形で与えられていると解釈され, 使用法 1 と同じ結果になる.

- たとえば, $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} (= I)$ (単位行列) に対し, `poly(A,'z')` とすると, $\det zI - A = \det (z-1)I = (z-1)^2$ となる.

```
-->A=[1 0;0 1];p=poly(A,'z')
 p  =
                2
    1 - 2z + z
```

注意事項

- Scilab では多項式の加減乗除ができる (通常の除算をすると有理式になり, 多項式の剰余を求めるには別の方法を使う) が …
- `p=1+2 * z + z^2` などのように, 通常の多項式の記法に近い形で多項式を定義するには準備が必要 (ただし乗算記号*は省略できない).

- たとえば, 多項式の変数の記号が z であるときには …
- 最初に z=poly(0,'z') と宣言しておく. これにより, 変数 z に, 根が 0 の z の多項式, すなわち多項式 z が割り振られる.
- この準備をしておけば, あとは p=1+2*z+z^2 などのようにして多項式を定義できる.

使用例は以下の通り.

```
-->z=poly(0,'z')
 z  =
    z

-->p=1+2*z+z^2
 p  =
                2
    1 + 2z + z
```

代数方程式を解く

- 関数 `roots()` を使うと, 複素数の範囲で, 代数方程式の解をすべて求めることができる. 使い方は 2 通りで …

  1. 事前に多項式を定義しておき, その多項式を引数として与える.
  2. 多項式の係数リストを高次から順に与える.

- 例として, Scilab で

$$z^2 + 2z + 5$$

  を解く.

- 厳密解は $z = -1 \pm 2i$.

- まず, 虚数単位について注意しておく
- Scilab では, 画面の表示では, 虚数単位は i と表示されるが, 入力するときには
  %i
  と打ち込む.

方法 1

```
-->p=poly([5 2 1],'z','coeff')
 p  =
                2
    5 + 2z + z
-->roots(p)
 ans  =
  - 1. + 2.i
  - 1. - 2.i
```

方法 2

```
-->roots([1 2 5])
 ans  =
  - 1. + 2.i
  - 1. - 2.i
```

- 当然, 結果は同じであるが …
- poly(,,'coeff') による多項式の定義では多項式の係数を低次から順に与える.
- roots([多項式の係数リスト]) では, 多項式の係数を高次から順に与える.
- poly と roots で, 多項式の係数を与える順番が逆

多項式を低次の項の積に分解する

- `roots` を使うと多項式の根がすべて求められるので, 複素数の範囲で多項式を 1 次多項式の積で表現することができるが …
- 応用上は, 実係数多項式を実係数の 1 次あるいは 2 次の多項式の積で表現することが必要になることもある.
- 後者の場合は, 関数 `factors` を用いる.

- p を多項式としたとき,
  `[lnum,g]=factors(p)`
  とすると, g に実数が, lnum に 1 次あるいは 2 次の多項式のリストが返される. lnum で与えられた多項式の積に g を乗じたものが, 数値計算の誤差を除き, もとの多項式に一致する.

- 例として,

$$p(z) = 2 + 2z + 2z^2 + 2z^3$$
$$= 2(1 + z)(1 + z^2)$$

を分解表現する.

```
-->p=poly([2 2 2 2],'z','coeff');

-->[lnum g]=factors(p)
```

分量の都合で, 結果を次ページにまわす.

```
g   =
    2.
lnum   =
       lnum(1)
                          2
    1 + 8.882D-16z + z

       lnum(2)
    1 + z
```

- 数値計算の誤差を除き, $2(1+z^2)(1+z)$ という分解表現が得られている
- `1+z^2` が `1 + 8.882D-16*z + z^2` に変わってしまっているが, このようになるのは, 数値計算の誤差のため避けられない. `8.882D-16` は 8.882×10^{-16} が, 数値計算の結果発生した誤差である.

多項式の四則演算

- 加算: +
- 減算: −
- 乗算: ∗
- 除算: / (結果は有理式になる)

多項式の剰余

- ユークリッドの互除法を用いて, 多項式 $A(z)$, $B(z)$ に対し,

$$A(z) = P(z)B(z) + Q(z)$$

となる多項式を求める (ただし, $Q(z)$ は多項式としての $A(z)$ の $B(z)$ による剰余) には…

- 関数 pdiv を用いる.
- 次ページに

$$1 + 2z + 3z^2 + 2z^3 + z^4$$

を

$$1 + z + z^2$$

で除算した例を示す.

```
-->A=1+2*z+3*z^2+2*z^3+z^4;
-->B=1+z+z^2;
-->[R,Q]=pdiv(A,B)
 Q  =
                2
    1 + z + z
 R  =
    0.
```

- 返却値の第 1 要素が剰余, 第 2 要素が商.
- この例では, 実は A=B^2 なので, 剰余 R は零となっている.

その他の関数

- 関数 `degree` は多項式の次数を返す.
- 関数 `coeff` は多項式の係数を返す.
- これらの使用例を次ページに示す.
- Scilab には他にも様々な多項式・有理式 (行列) を操作するための関数があるが, この講義ではこれ以上深入りしない.

degree と coeff の使用例

```
-->z=poly(0,'z');
-->p=-1+z+2*z^2+3*z^3;
-->degree(p)
 ans  =
    3.
-->coeff(p)
 ans  =
  - 1.    1.    2.    3.
```

アルゴリズムやソフトウェアの比較

- 教科書には代数方程式を解くための2種類のアルゴリズムが記載されている:
- 第1のアルゴリズムは$\boldsymbol{x}$が2次のベクトルの場合のNewton法を修正なしで適用したもの.
- 第2のアルゴリズムはBairstowによるもの(1920年).

- 素朴な Newton 法には初期値の設定に関する問題があるが, 考え方を理解するためには重要.
- Bairstow 法はすべての根を求められることが特徴だが, 数値的な性質は良くない.
- Scilab では Jenkins & Traub のアルゴリズム (1970 年) などが用いられる (この講義ではこのアルゴリズムには立ち入らない)

- Bairstow 法と, Scilab の `roots()`, MATLAB の `roots()`, Mathematica の `NSolve[]` の挙動を比較してみる.
- 最高次の係数が 1 で, 残りの係数が $[-1, 1]$ に値を取る一様分布の擬似乱数から生成された多項式 1000 個の根を求め, 比較した.
- 動作環境等は次の通り.

- Scilab 5.5.2 for Windows 64bit および MATLAB R2015b for Windows 64bit : Intel Core i5-4690 3.50GHz, 32GB of Memory, Windows7 Professional 64bit Service Pack 1
- Mathematica: Mathematica 10.1.0 for Microsoft Windows (64-bit), Intel(R) Core(TM) i5-2500K CPU 3.30GHz, 16.0GB of Memory, Windows7 Professional 64Bit Service Pack 1

Scilab で Bairstow 法と roots() を比較

- Bairstow 法は多項式の次数が 10 のときは動作したが, 20 のときは動作しなった (零による除算が発生した可能性がある).
- Scilab の `roots()` は次数 100 まで一応動作したが, 高次では大きな誤差が出ることがあった.

次数 10 の多項式では …

	Bairstow 法	roots()
平均求解時間	5.68×10^{-3} 秒	1.25×10^{-4} 秒
$\lvert f(x) \rvert$ の平均	2.16×10^{-10}	9.51×10^{-15}

$f(x) = 0$ を解いているので, 理想的には $|f(x)|$ は零になることに注意.

次数が 20, 50, 100 のときには … $|f(x)| \geq 10^{-3}$ となる x を誤答, それ以外を正解と見做し, 次数を変えたときの, 平均求解時間, 正解に関する $|f(x)|$ の平均, 誤答率, $|f(x)|$ の最悪値を次に示す.

roots()	次数 20	次数 50	次数 100
平均求解時間	4.68×10^{-5}	5.93×10^{-4}	5.97×10^{-3}
正解の $\|f(x)\|$ の平均	2.24×10^{-13}	1.28×10^{-7}	5.41×10^{-7}
誤答率	0%	0.004%	0.316%
$\|f(x)\|$ の最悪値	9.14×10^{-10}	2.05×10^{-2}	8.09×10^{10}

- 最悪で $|f(x)| = 8.09 \times 10^{10}$ であり (本来は零になる筈), うまく解けていないことがわかる.
- 100 次の多項式の場合, 根は 100 個あるので, 誤答率 0.3%といっても, 無視できない. 100 個の根を求めたとき, すべての根が誤答でない確率は, 確率的な独立性を仮定すれば, $(0.997)^{100} = 0.74$ で, 74%程度

- これはScilabが無償だから駄目というわけではなく, 浮動小数点数を使った計算の誤差が原因なので, 有償ソフトでも工夫せずに使えば同様の問題が出る.
- 有償ソフトのMATLABの関数`roots()`, Mathematicaの`NSolve[]`と比較する.

	Scilab	MATLAB	Mathematica
平均求解時間	5.97×10^{-3} 秒	4.4×10^{-3} 秒	6.96×10^{-3} 秒
正解の $\|f(x)\|$ の平均	5.41×10^{-7}	5.67×10^{-7}	3.82×10^{-7}
誤答率	0.316%	0.288%	0.197%
$\|f(x)\|$ の最悪値	8.09×10^{10}	1.20×10^{12}	1.20×10^{9}

- どのソフトでも解が異常な可能性あり
- 数値計算結果の妥当性をチェックする習慣をつけることが必要. 盲信は危険.
- 教科書等のアルゴリズムを盲信するのも危険 (Bairstow 法の結果を思い出すこと)

なぜこんなことが起こるのか?

- この問題では, 倍精度浮動小数点数では桁数が足りない.
- 素性の良いアルゴリズムは, 数値計算の誤差がないなどの理想的な条件の下では, 正解への収束性を保証するが, 実用上は, 理想的な条件が満たされていないことも多い.

- アルゴリズムによっては, 計算手順を定めているのみで, 正解への収束性が保証していないものもある.
- 数値計算の誤差を見込んだ上で解の精度を保証する考え方 (精度保証付き数値計算) もあるが, どんな問題にでも使えるというわけではない.

- 実はMathematicaには, 精度保証とは違うが, 対策がある. 計算時間が増えるが, 演算に使う桁数を明示的に指定することで, 精度を高めることができる.
- Mathematicaの擬似乱数生成器および`NSolve[]`において`WorkingPrecision->100`として内部計算における桁を 100 桁確保した場合を, 倍精度の場合と比較する.

	Mathematica 倍精度	Mathematica 内部計算 100 桁
平均求解時間	6.96×10^{-3} 秒	1.32×10^{-1} 秒
正解の $\lvert f(x)\rvert$ の平均	3.82×10^{-7}	0.0×10^{-75}
誤答率	0.197%	0%
$\lvert f(x)\rvert$ の最悪値	1.20×10^{9}	0

まとめ

- コンピュータによる数値計算の結果は必ずしも信用できないので結果の検証が必要.
- ソフトウェアによっては, 計算精度を指定できる. その場合は比較的安心.

代数方程式の数値解法の初歩

- 代数方程式を解くときにも, もっとも基本的なのは Newton 法であるが…
- Newton 法には, $f'(z) = 0$ となる場合には使えないという制限がある. Newton 法の系列のアルゴリズムの大半はこの制限を受ける (例外あり (後述)).

- 代数方程式が重根を含むときには, 重根となる点において $f'(z) = 0$ となる. したがって, 一般的には, Newton 法を工夫なしに使った場合は, 代数方程式の解すべてを求めることはできない.

- 代数方程式が重根を持つことは「確率的には稀」と考える受講者もいるかもしれないが, 応用上, 重根を持つ代数方程式はしばしば現れるため, 重根の求解を断念するわけにはいかない.

- 重根への対応法は後回しにして，当面，重根はないものと仮定し，素朴な Newton 法で代数方程式を解くことを考える

- Newton 法の主要部 (反復解法) は,

$$x(k+1) = x(k) - \frac{f(x(k)}{f'(x(k))}$$

であった.

- 前ページの式では, $x(k)$, $f(x(k))$, $f'(x(k))$ がスカラーであることは想定されているが, それが実数であることは要請されておらず, スカラーを実数から複素数に変更しても, 処理系が複素数に対応しているのであれば, アルゴリズムを変更する必要はない.

- 教科書では 2 変数版の Newton 法が利用されているが, Scilab は複素数を取り扱えるので, 複素変数を使えば, 1 変数の Newton 法で十分. プログラムは実変数と同じ.
- $x^2 + 1 = 0$ を解くプログラムのサンプルを次のシートに示す.

```
deff('[y]=f(x)','y=x^2+1');
deff('[y]=df(x)','y=2*x');
x0=0.9*%i; //初期値, %i は虚数単位
x=x0;maxItr=100;smallTh=1e-8;
for itr=1:maxItr
  x=x-f(x)/df(x);
  if(abs(f(x))<smallTh)
   break();
  end
end
```

- アルゴリズムの主要部

  `x=x-f(x)/df(x);`

  は数式

$$x(k+1) = x(k) - \frac{f(x(k)}{f'(x(k))}$$

  をプログラムの形で書き下しただけで, 実変数の場合と同じ.

- 素朴な Newton 法は初期値次第で発散することがある.
- 直線探索あるいは信頼領域法を用いた大域的に収束する Newton 法は, 重根などの場合に, この「一定の条件」が満たされないこともある.
- この問題への対処は後述.

- Scilabで採用されている Jenkins & Traub のアルゴリズムはポピュラー.
- 連立法 (Durand – Kerner 法および Ehrlich – Aberth 法) と呼ばれる手法もある.
- いずれも複雑なのでこの講義では内容には立ち入らない.

- Newton 法の変形で, 必ずいずれかの解に収束することが保証されたアルゴリズムに, 平野法と呼ばれる方法がある (平野菅保が 1967 年に提案). 以下でこれについて解説する.

平野法

平野法は, 次のような特徴を持つ方法.

- Taylor 展開の高次項をすべて利用
- 重根にも適用できる
- 初期値をどのように取ってもいずれかの解に大域的に収束する

平野法のアルゴリズムを次ページ以降で述べる.

- $p(z) = a_0 z^n + a_1 z^{n-1} + \cdots + z_n$ とし, $p(z) = 0$ を解きたいものとする.
- 仮の解を z_0 とし, $p(z_0 + \zeta)$ を z_0 のまわりで Taylor 展開すると

$$p(z_0 + \zeta) = p(z_0) + \sum_{k=1}^{n} \frac{p^{(k)}(z_0)}{k!} \zeta^k$$

- $p(z_0+\zeta) = p(z_0) + \sum_{k=1}^{n} \frac{p^{(k)}(z_0)}{k!}\zeta^k$ という展開を利用して $\cdots$
- $k = 1, 2, \ldots, n$ に対し, Taylor 展開の第 k 次の項 $\frac{p^{(k)}(z_0)}{k!}\zeta^k$ の $p(z_0+\zeta)$ への影響のみを考え, 他の項を無視する. すなわち, 次のように近似する.

$$p(z_0+\zeta) \simeq p(z_0) + \frac{p^{(k)}(z_0)}{k!}\zeta^k$$

- $p(z_0+\zeta)=0$ を解きたいので, $0=p(z_0)+\frac{p^{(k)}(z_0)}{k!}\zeta^k$ の解, すなわち

$$\zeta(k)=\left(-\frac{k!p(z_0)}{p^{(k)}(z_0)}\right)^{1/k}$$

を k 番目の解の候補とする (k 乗根は複素数の範囲).

- $\zeta(1), \zeta(2), \ldots, \zeta(n)$ を上記の手順で求める.
- $\zeta(1), \zeta(2), \ldots, \zeta(n)$ の中で絶対値がもっとも小さいものが「解の改善の効率が高い」と考え, これを使って z_0 を更新する: $z_0 = z_0 + \zeta(k)$ (ただし $\zeta(k)$ は $\zeta(1), \zeta(2), \ldots, \zeta(n)$ の中で絶対値が最小のもの.

- より詳しく言うと, backtracking と呼ばれる手法 (減速とも呼ばれる) を使って, $|p(z_0 + \mu\zeta(k))|$ が $|p(z_0)|$ より小さくなるよう, $\zeta(k)$ に乗ずる適切な「倍率」μ を定める必要がある.

- $\zeta(k)$ は複数の分枝を持つが, その中でもっとも $p(z_0+\zeta)$ が零に近付くものを採用する.
- 以上を, $p(z_0)$ の値が十分小さくなるまで繰り返す解法が平野法.
- 収束性については杉原, 室田, 数値計算法の数理, 岩波書店, 1994 などを参照.
- 平野法は, 解をひとつ求めたいときには便利.

- 高次代数方程式の根を多項式の剰余算によって逐次的すべて求めようとすると誤差の集積の問題が発生しやすいので, そのような場合には Jenkins & Traub のアルゴリズムや連立法を使うべき.
- 代数方程式 $p(z) = 0$ を解くためのアルゴリズムを詳しく書き下すと, 次のようになる.

初期化: z の初期値とパラメータ β, λ を定める (ただし $0 < \beta < 1, \lambda > 1$).

ループ: $p(z)$ の絶対値が指定した値以下になるまで以下を繰り返す.

1. μ=1 とする.
2. $k-1, \ldots, n$ に対し, $\zeta(k) = \left(-\mu \frac{k! p(z_0)}{p^{(k)}(z_0)}\right)^{1/k}$ とする.
3. $\{\zeta(1), \ldots, \zeta(n)\}$ の中で絶対値が最小のものを $\zeta(s)$ とする (複数の分枝を含むことがある).
4. $\zeta(s)$ の分枝を $\{\xi_1, \ldots, \xi_s\}$ とする. $\{|p(z+\xi_1)|, \ldots, |p(z+\xi_s)|\}$ の中で最小のものを $|p(z+\xi_t)|$ とする.
5. $|p(z+\xi_t)| < (1-(1-\beta)\mu)p(z)$ なら $z = z+\xi_t$ としてループ冒頭に戻る. そうでなければ $\mu = \mu/\lambda$ としてステップ 2 に戻る.

Horner の方法と組立除法

- 次数が高い多項式やその導関数のある点における値を求めるには, 多数回の乗算と加算が必要になる. 高速に多項式の値を評価するためには, 乗算回数を減らすことが望ましい.

- このための方法がHornerの方法 (および組立除法). 杉原, 室田, 数値計算法の数理, 岩波書店, 1994 にしたがって, これを説明する.

- 例として, $p(z) = a_0z^3 + a_1z^2 + a_2z + a_3$ において, z に定数 c を代入した値を計算することを考える.
- 次のような計算をしてみる.
- 左の欄の数字に c を掛けたものと上の欄に 1 を掛けたものを足す
  (右上がりの対角線まで)

	$a_1^0 = a_1$	$a_2^0 = a_2$	$a_3^0 = a_3$
$a_0^1 = a_0$	$a_1^1 = a_0^1 c + a_1^0$	$a_2^1 = a_1^1 c + a_2^0$	$a_3^1 = a_2^1 c + a_3^0$
$a_0^2 = a_0$	$a_1^2 = a_0^2 c + a_1^1$	$a_2^2 = a_1^2 c + a_2^1$	
$a_0^3 = a_0$	$a_1^3 = a_0^3 c + a_1^2$		
$a_0^4 = a_0$			

上記の計算をおこなうと, 次のような表が得られる.

	a_1	a_2	a_3
a_0	$a_0c + a_1$	$a_0c^2 + a_1c + a_2$	$a_0c^3 + a_1c^2 + a_2c + a_3$
a_0	$2a_0c + a_1$	$3a_0c^2 + 2a_1c + a_2$	
a_0	$3a_0c + a_1$		
a_0			

$p(z) = a_0 z^3 + a_1 z^2 + a_2 z + a_3$ と見比べると …

	a_1	a_2	a_3
a_0	$a_0 c + a_1$	$a_0 c^2 + a_1 c + a_2$	$\boxed{p(c)}$
a_0	$2a_0 c + a_1$	$\boxed{p'(c)}$	
a_0	$\boxed{\frac{1}{2!} p''(c)}$		
$\boxed{\frac{1}{3!} p'''(c)}$			

- 以上のように, 単純に計算する場合より少ない乗算と可算の回数で $p(c), p'(c), p''(c), p'''(c)$ が求められている.
- 一般の次数の場合も計算法は同じ.
- このようにして, 多項式とその導関数に定数 c を代入した値を効率良く求めることができる.