

代数方程式

—数値計算(11/9/30)—

関西学院大学理工学部 西谷滋人

Copyright ©2007-11 by Shigeto R. Nishitani

概要

代数方程式の解 $f(x)=0$ を数値的に求めることを考える。標準的な二分法(bisection method)とニュートン法(Newton's method)の考え方と例を説明し、

収束性(convergency)と安定性(stability))

について議論する。さらに収束判定条件について言及する。

二分法のアイデアは単純。中間値の定理より連続な関数では、関数の符号が変わる二つの変数の間には根が必ず存在する。したがって、この方法は収束性は決して高くはないが、確実。一方、Newton法は関数の微分を用いて収束性を速めた方法である。しかし、不幸にして収束しない場合や微分に時間がかかる場合があり、初期値や使用対象には注意を要する。

Mapleでの解

Mapleでは代数方程式の解は、fsolveで求まる。

$$x^2 - 4x + 1 = 0$$

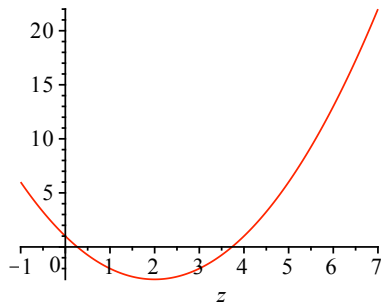
の解を考える。未知の問題では時として異常な振る舞いをする関数を相手にすることがあるので、まずは関数の概形を見ることを常に心がけるべき。

```
> restart;
```

```
> func:=x->x^2-4*x+1;
```

$$func := x \rightarrow x^2 - 4x + 1$$

```
> plot(func(z), z=-1..7);
```



もし、解析解が容易に求まるなら、その結果を使うほうがよい。Maple scriptの解析解を求めるsolveでは、

```
> solve(func(x)=0,x);
```

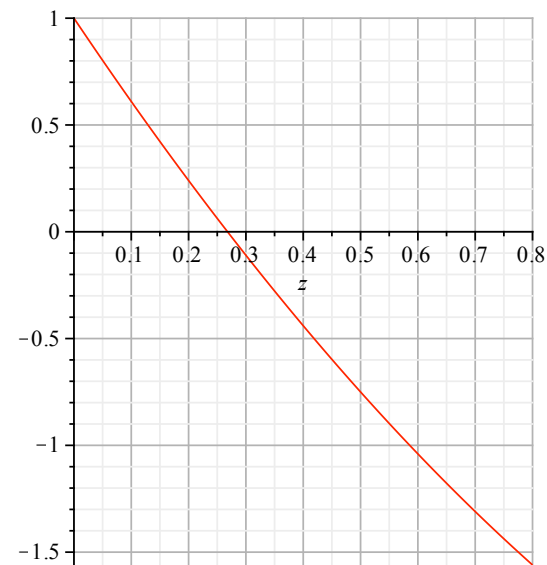
と即座に求めてくれる。数値解は以下の通り求められる。

```
> fsolve(func(x)=0,x);
```

二分法とNewton法の原理

二分法は領域の端 x_1, x_2 で関数値 $f(x_1), f(x_2)$ を求め、中間の値を次々に計算して、解を囲い込んでいく方法である。

```
> plot(func(z), z=0..0.8, gridlines=true);
```



x_1	x_2	$f(x_1)$	$f(x_2)$
0.0	0.8		

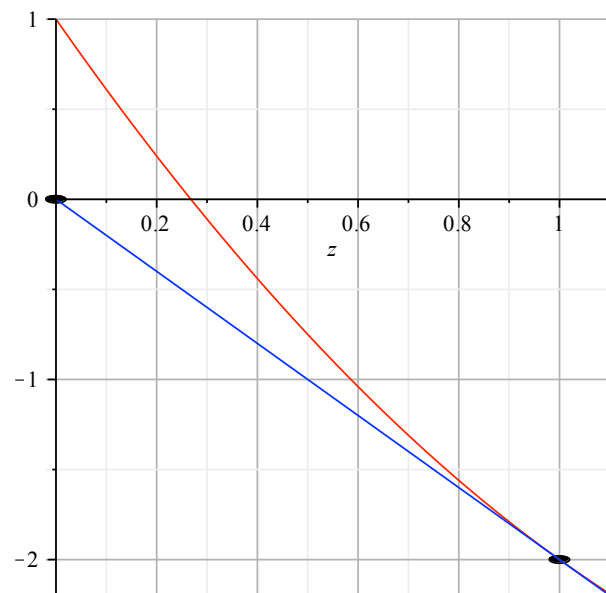
▼ Newton法(あるいはNewton-Raphson法)

Newton法は最初の点 x_1 から接線をひき、それが y 軸($=0$)と交わった点を新たな点 x_2 とする。さらにそこでの接線を求めて...
という操作を繰り返しながら解を求める方法である。関数の微分を $df(x)$ とすると、これらの間には

「という関係が成り立つ。

```
> df:=unapply(diff(func(x),x),x);
```

```
> with(plots):with(plottools):
x1:=1.0:x0:=0.0:
p:=plot(func(z),z=0..1.1):
p1:=plot(df(x1)*(z-x1)+func(x1),z=0..1.1,color=blue):
p2:=[disk([x1,func(x1)],0.02),
disk([x0,0],0.02)]:
display(p,p1,p2,gridlines=true);
```



x	$f(x)$	$df(x)$
1.0		

▼二分法とNewton法のコード

▼ 二分法(bisection)

```
> x1:=0:
x2:=0.8:
f1:=func(x1):
f2:=func(x2):
for i from 1 to 5 do
  x:=(x1+x2)/2;
  f:=func(x);
  if f*f1>=0.0 then
    x1:=x;
    f1:=f;
  else
    x2:=x;
    f2:=f;
  end if;
  printf("%20.15f, %20.15f\n",x,f);
end do:
0.400000000000000, -0.440000000000000
0.200000000000000, 0.240000000000000
0.300000000000000, -0.110000000000000
0.250000000000000, 0.062500000000000
0.275000000000000, -0.024375000000000
```

▼ Newton法(あるいはNewton-Raphson法)

```
> dfunc:=unapply(diff(func(z),z),z);  
                                     dfunc := z → 2 z - 4  
  
> x:=1:  
f:=func(x):  
printf("%15.10f, %24.25f\n",x,f);  
for i from 1 to 5 do  
    x:=x-f/dfunc(x);  
    f:=func(x);  
    printf("%15.10f, %24.25f\n",x,f);  
end do:
```

1.0000000000	-2.000000000000000000000000000000
0.0000000000	+1.000000000000000000000000000000
0.2500000000	+0.062500000000000000000000000000
0.2678571429	+0.000318877551000000000000000000
0.2679491900	+0.000000000084726737970000000000
0.2679491924	+0.000000000000000000000000000000

以下のようにDigitsを変更すれば、Mapleでは浮動小数点演算の有効数字を変えることができる。

```
> Digits:=40;
```

Digits := 40

収束性と安定性

実際のコードの出力からも分かる通り、解の収束の速さは2つの手法で極端に違う。2分法では一回の操作で解の区間が半分になる。このように繰り返しごとに誤差幅が前回の誤差幅の定数(<1)倍になる方法は1次収束(linear convergence)するという。Newton法では関数・初期値が素直な場合($f'(x) \neq 0$)に、収束が誤差の2乗に比例する2次収束を示す。以下はその導出を示した。

```
> restart;
ff:=subs(xi-x[f]=ei,series(f(xi),xi=x[f],4));
ff:=f(x_f) + D(f)(x_f) ei + 1/2 D^(2)(f)(x_f) ei^2 + 1/6 D^(3)(f)(x_f) ei^3 + O(ei^4)

> dff:=subs({0=x[f],x=ei},series(diff(f(x),x),x,3));
dff:=D(f)(x_f) + D^(2)(f)(x_f) ei + 1/2 D^(3)(f)(x_f) ei^2 + O(ei^3)

> ei1:=ei-ff/dff;
ei1:=ei
- (f(x_f) + D(f)(x_f) ei + 1/2 D^(2)(f)(x_f) ei^2 + 1/6 D^(3)(f)(x_f) ei^3
+ O(ei^4)) / (D(f)(x_f) + D^(2)(f)(x_f) ei + 1/2 D^(3)(f)(x_f) ei^2 + O(ei^3))

> ei2:=simplify(convert(ei1,polynomial));
ei2:= 1/3 (3 D^(2)(f)(x_f) ei^2 + 2 D^(3)(f)(x_f) ei^3 - 6 f(x_f)
2 D(f)(x_f) + 2 D^(2)(f)(x_f) ei + D^(3)(f)(x_f) ei^2)

> ei3:=series(ei2,ei,3);
ei3:= -f(x_f)/D(f)(x_f) + f(x_f) D^(2)(f)(x_f)/(D(f)(x_f)^2) ei
+ 1/6 (3 D^(2)(f)(x_f) + 3 f(x_f) D^(3)(f)(x_f)/D(f)(x_f) - 6 f(x_f) D^(2)(f)(x_f)^2/D(f)(x_f)^2) ei^2
+ O(ei^3)

> subs(f(x[f])=0,ei3);
1/2 D^(2)(f)(x_f)/(D(f)(x_f)) ei^2 + O(ei^3)
```

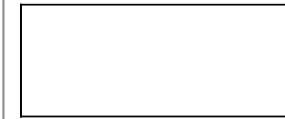
注意すべきは、この収束性には一回の計算時間の差は入っていないことである。Newton法で解析的に微分が求まらない場合、数値的に求めるという手法がとられるが、これにかかる計算時間はばかにできない。二分法を改良した割線法(secant method)がより速い場合がある(NumRecipe9章参照)。二分法では、収束は遅いが、正負の関数値の間に連続関数では必ず解が存在するという意味で解が保証されている。しかし、Newton法では、収束は速いが、必ずしも

素直に解に収束するとは限らない。解を確実に囲い込む、あるいは解に近い値を初期値に選ぶ手法が種々考案されている。解が安定であるかどうかは、問題、解法、初期値に大きく依存する。収束性と安定性のコントロールが数値計算のツボとなる。

収束判定条件

どこまで値が解に近づけば計算を打ち切るかを定める条件を収束判定条件と呼ぶ。以下のような条件がある。

ε(イプシロン, epsilon)法



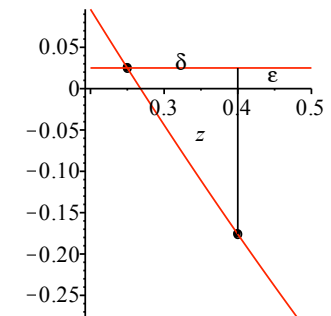
δ(デルタ, delta)法



占部法

数値計算の際の丸め誤差までも含めて判定する条件で、 $|f(x_{i+1})| > |f(x_i)|$ とする。

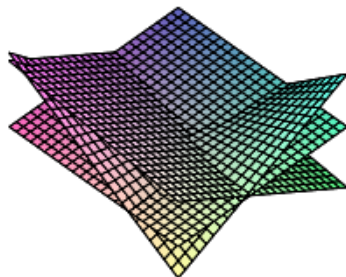
```
> with(plots):with(plottools):
f2:=x->0.4*(x^2-4*x+1):x1:=0.25:x0:=0.4:
p:=plot(f2(z),z=x1-x1/5..x0+x0/5):
p1:=plot([f2(x1)],z=0.2..0.5):
p2:=disk([x1,f2(x1)],0.005),
disk([x0,f2(x0)],0.005):
l1:=line([x0,f2(x0)], [x0,f2(x1)]):
t1:=textplot([0.45,0.0,`epsilon`],align=above):
t2:=textplot([0.325,0.05,`delta`],align=BELOW):
> display(p,p1,p2,l1,t1,t2);
```



2変数の関数の場合

2変数の関数では、解を求める一般的な手法は無い。この様子は実際に2変数の関数で構成される面の様子をみれば納得されよう。解のある程度近くからは、Newton法で効率良く求められることが知られている。

```
> restart;
> f:=(x,y)->4*x+2*y-6*x*y;
g:=(x,y)->10*x-2*y+1;
f:=(x,y)->4*x+2*y-6*x*y
g:=(x,y)->10*x-2*y+1
> plot3d({f(x,y),g(x,y),0},x=-2..2,y=-2..2);
```



```
> fsolve({f(x,y)=0,g(x,y)=0},{x,y});
{x = -0.07540291160, y = 0.1229854420}
```

課題

1 Newton法の $f(x)$, $df(x)$ の関係を示す式を導け.

2 次の関数

$$f(x) = \exp(-x) - 2 \exp(-2x)$$

の解を二分法, Newton法で求めよ.

3 代数方程式に関する次の課題に答えよ. (2004年度期末試験)

i $\exp(-x) = x^2$ を二分法およびニュートン法で解け.

ii n 回目の値 x_n と小数点以下10桁まで求めた値 $x_f=0.7034674225$ との差 Δx_n の絶対値(abs)のlogを n の関数としてプロットし, その収束性を比較せよ. また, その傾きの違いを両解法の原理から説明せよ.

4 次の方程式

$$f(x) = x^4 - x - 0.12$$

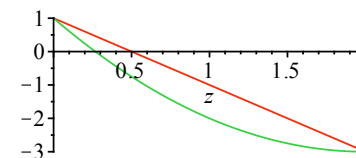
の正数解を二分法で求めよ. (2008年度期末試験)

5 収束条件がうまく機能しない例を示せ.

6 割線法は, 微分がうまく求まらないような場合に効率がよい, 二分法を改良した方法である. 二分法では新たな点を元の2点の midpoint に取っていた. そのかわりに下図に示すごとく, 新たな点を元の2点を直線で内挿した点に取る. 二分法のコードを少し換えて, 割線法のコードを書け. また, 収束の様子を二分法, Newton法と比べよ.

```
> func:=x->x^2-4*x+1;
x1:=0;
```

```
x2:=2;
f1:=func(x1);
f2:=func(x2);
plot({(z-x1)*(f1-f2)/(x1-x2)+f1, func(z)}, z=0..2);
```



7 次の方程式

$$f(x) = \cos(x) - x^2$$

の正数解を二分法で求めよ. 割線法でも求め, 収束性を比べよ. (2009年度期末試験)

8 次の方程式

$$f(x) = x^3 - 3x + 3$$

の解をニュートン法で求めよ. 初期値をそれぞれ $x=-3$, $x=2$ とした時を比べ, その差について論ぜよ. (2010年度期末試験)