数値計算入門:　入門と非線形方程式の解法

Naohito Abe
一橋大学　応用マクロ経済学　講義ノート

平成 26 年 5 月 2 日

概 要

本講義では、数学的な厳密さよりも、実際にどう解くかを中心に議論する。常に Judd (1988) に戻りながらプログラムを組んでもらいたい。

1　数値計算に関して

これから数回の講義では、数値計算の考え方、および Matlab を用いたプログラミングの初歩について解説する。

数値計算そのものは、Stata や TSP 等の統計ソフトは無論、エクセル等の表計算ソフトでも背後で行われており、特に意識しなくとも、その恩恵に預かっている場合は非常に多い。例えば、統計パッケージで Probit 等の最尤法を選択すると、最適化アルゴリズムに関する Option がある。Newton 法や、Davidson-Fletcher-Powel(DFP) 法、あるいは Broyden-Fletcher-Goldfarb-Shanno(BDFGS) 等である。数値計算について学ぶ一つの理由は、こうした統計パッケージの背後に動くメカニズムを知り、正しい option の選択を行う、または、すくなくとも誤った手法を選択しないようにすることにある。

しかし、数値計算を学ぶ最大の利点は、自分の研究手法の範囲を拡大することなある。確かに、近年の Stata 等の統計パッケージは非常によくできており、かつ、常に進化し続けており、バージョンアップの度に新しい推計手法に対応するようになっている。また、多くの計量経済学者達が自分達が開発した新しい統計量を世の中に広めるため、Stata の ado ファイルで、または独自のアプリケーションを開発し無償で世界に公開するようになっている[1]。しかしながら、それでも最先端の研究で用いられる推計手法の多くは現在でも統計パッケージの組み込みコマンドでは利用できないし、たとえコマンドが存在しても、使用に耐えないくらい遅かったり、標準誤差の計算結果が不安定だったり、そもそも収束しないことも少なくない[2]。結局、自分でコマンドの

[1] Arellano and Bond による Panel GMM 推計も Stata や TSP でプログラムが提供されるようになってから、一気にに普及した。一方、複雑な尤度関数をかかねばならない Dynamic Panel の推計方法は、Hsiao の教科書で古くから解説されているが、あまり応用されていない。

[2] 例えば、Mixed LOGIT や Multinominal PROBIT など。

中身を確認し、修正せねばならないことも多い。一般に、作者により Matlab や Gauss, C,Fortran 等の言語であるプログラムが書かれた後、商用統計パッケージにそのコードが組み込まれ、広く使用されるようになるまではかなり長い時間がかかる。自分で Matlab や Gauss 等でコードを書かねば、最先端の研究手法を用いることができないことが多い。統計パッケージソフトの限界が自分の研究の限界になることは避けねばならない。

自分でコードを書かなくとも、他人の書いたコードの意味を理解するには、数値計算の基本がわかっていなければならない。世界の研究者、特に計量経済学やマクロ経済学の研究者は研究に用いたプログラムコードを web 等で公開していることが多い。多くのコードは Matlab や Gauss、または Fortran 等で書かれており、使用するにはそれらの開発環境に関する知識はもちろん、彼等のプログミングの長所と限界を知らねばならない。マクロモデルに限らず、モデル解法に普遍的なアルゴリズムは存在せず、最適なアルゴリズムは解くべき問題により大きく異なってくる。無論、同じモデルに対しても、研究者により選択するアルゴリズムは異なるし、それらに優劣をつけるのは簡単なことではない。しかしながら、自分で計算する際には、何かを選択せねばならないわけで、その際、各アルゴリズムの特徴を良く踏まえたうえで選択するのか、それともとりあえず先行研究に従って一つ適当に選ぶかでは、後の効率性が大きく変わってくるだろう。

この応用マクロ経済学では、マクロ経済学、とくに Dynamic Programmin に関わる一連の手法の修得を目標とするが、同時に一般的な数値計算の基本に関しても多少時間を割く予定である。

2　計算誤差について

コンピューターは、数式処理を行う場合を除けば無限精度の計算を行うことができない。通常、私達が使う統計ソフトや Matlab は倍精度、すなわち 15 桁から 16 桁の精度で計算することが出来る[3]。Matlab で扱うことのできる最小の数 (有効桁数という意味だが) は eps と打ち込むことで知ることが出来、Windows 版では $2.204e-16$ となる。

16 桁の精度があれば地球から太陽までの距離をミリの単位で測ることが可能であり、通常の四則演算をするかぎり、計算精度の問題を意識する必要はない。しかしながら、数値計算で複雑な作業を始めると、16 桁でも精度が不足することが多々ある。特に引き算を繰り返して行うときに、発生する可能性が高い。

例えば、

$a = 123456789123456789$

[3]なお、stata では default は float であり、もっと精度が低い。例えば、商品コードなど、13 桁以上の数値を用いる場合は、double に宣言しないと、勝手に stata が四捨五入してしまい、商品コードとして使えないので注意すること。

とすると、これは計算精度を超えた数値となる。したがって
$b = a - 1$ と定義しても
$b - a$ は 1 にならず、0 が帰ってくる。
実際に直面しそうな問題の一例は、二次方程式の解である。

$$ax^2 + bx + c = 0 \tag{1}$$

の解は

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2}$$

で与えられる。もしも ac が b に比して小さければ、$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$ はゼロに近い数字となる。注意すべきは、Matlab は、b や a を 16 桁数字で扱うのみならず、この解もまた、あたかも 16 桁精度として表現してしまうことである。しかしながら、もしも $b = 100$ で $a = 1$,$c = 0.01$ だとしたら、この解の精度は 10 桁程度しかない。もしも $b = 10000$ ならば、精度は 6 桁程度しかない。極端な例あるが、

$$\left(x - 10^8\right)\left(x - 10^{-8}\right) = 0$$

の解を計算すると、Matlab は
ans =
100000000
ans =
1.4901e-008
を返す。10^8 は正しく計算されているが、10^{-8} となるべき解には非常に大きな計算誤差が含まれていることがわかる。これが非常に近い数値の引き算をとることによる誤差 (桁落ち) である。このような場合は、引き算により問題が生じているわけであるから、極力引き算をとらないで計算することで回避可能なときがある。例えば、まず解の公式を用い、足し算の解を x_1、

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

として計算し、次に、解と係数の関係

$$x_1 \times x_2 = \frac{c}{a} \tag{3}$$

$$x_2 = \frac{c}{ax_1}$$

を用いてもう一つの解を計算すると、Matlab は正しく $1.0000e - 008$ を解として返してくる。

上記の問題は、二階微分を扱うときにより深刻になる。微分を、公式を使わずに微小な差分で定義すると、極めて似た数値の引き算を行うことになる。

一回の微分で精度が 6 桁減少すると、二回目の微分ではさらに 6 桁減少し、16 桁精度でも 4 桁程度の精度しかなくなってしまう。多くの統計パッケージでヤコビアンやヘシアンを外生的に与えるオプションがあるのは、この数値計算による桁オチの問題を回避するためでもある。数値計算による誤差を極力減らすためには、解析的に微分係数を求めることができるときには、極力、数値計算を用いない微分係数を使用することが望ましい。二次方程式の解の公式や解の関係、あるいは悪ラス報告や双対性などの経済理論を駆使し、コンピューターにかかる負荷が少なくなるように工夫することが、時に非常に重要になってくる。

アルゴリズムを何度みても、間違いがみつからず、また数学的には正しい解を予測できているにも関わらず、出力結果がおかしいときには、どこかで、計算誤差が拡大するような処理を行っていないかどうか確認する必要がある。とくに、経済学では、繰り返し代入や反復法を多用するため、数値計算誤差が蓄積されていく可能性がある。

数値計算に潜む誤差は大きな問題であり、Judd [1998] の P.39 以降を読むことをお勧めする。

3　非線形解法の基本問題

$$f : R^n \Longrightarrow R^n, x \in R^n$$

この関数 f の解 (root) の一つを求める。

$$f(x) = 0$$

または、経済学ではよく不動点として解を定義する。その場合は、

$$g(x) = f(x) - x$$

と定義すれば、

$$g(x) = x$$

で定義される g の不動点が f の解となる。

なお、解を求めるアルゴリズムは、次回で解説する予定が最適化アルゴリズムと密接な関係がある。解を求めるアルゴリズムを用いて最適化コードを書くことが可能であるし、逆に最適化コードから非線型方程式の解を求めることも可能である。

3.1 Bisection

一般的なケースは Judd[1988] を参照することにし、ここでは、$f : R \Longrightarrow R$ のケースのみを考える。

f が連続、かつ f(a) <0, f(b) >0 であれば、中間値の定理より (a,b) の中に f(x)=0 を満たす x が存在する。この中間値の定理を用いて f(x)=0 を満たす x を求める手法を Bisection 法という。

アルゴリズム 1 *Bisection*

目的*:* 関数 $f(x)$, $f : R \Longrightarrow R$ のゼロ点を見つけること

Step 0: $x^L < x^R, f\left(x^L\right) f\left(x^R\right) < 0$ となるような x^L, x^R を見つける。また、解の誤差許容度 δ, ε を設定する。

Step 1: 中間値 $x^M = \frac{x^L + x^R}{2}$ を計算する。

Step 2: もしも $f\left(x^L\right) f\left(x^M\right) < 0$ であれば、$x^R = x^M$ とする。$f\left(x^M\right) f\left(x^R\right) < 0$ であれば、$x^L = x^M$ とする。

Step 3: $x^L - x^R < \varepsilon \left|1 + \left|x^L\right| + \left|x^R\right|\right|, or \quad \left|f\left(x^M\right)\right| < \delta$ かどうかを確認。いずれも満たされてない場合は Step1 に戻る。

誤差許容度 δ, ε を大きくとると早く収束するが、不正確となる。一方、誤差許容度 δ, ε を極めて小さくすると収束に時間がかかり、ゼロとするといつまでも収束しない。「適切」な誤差許容度を設定することは以外に難しい。まず覚えておくべきはコンピューターの計算精度である。Matlab の default は double の 16 桁である。したがって $\varepsilon = 10^{-21}$ などとしても意味はない。また、x^L, x^R が非常に大きな値, 12345543321123 などの場合は、$\varepsilon = 0.001$ としてもコンピューターが識別できる精度を超えてしまうことに注意が必要である。

上記のアルゴリズムでは x^L, x^R との「相対的な」大きさとして許容度を設定しているのは、x^L, x^R の絶対的な大きさにより、誤差許容度 δ, ε が数値計算で意味のない値になることを防ぐためである。

Matlab でのコード例

まず、ゼロ点を求めたい関数を m ファイルとして作成する。例えば

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```
function y=bisecfun1(x)
y= x^5 + 2*x -2
end
```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

とだけ書いたファイルを bisecfun1.m として保存する。

次に、bisection.m として

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x=bisection(f,a,b)
%
%  bisection 法による Root finding
%  a,b:初期区間
%  2014 年応用マクロ経済学講義用
%  naohito abe
%
% 初期設定
%
tol = 0.001; % 相対的誤差許容量
%
sa = sign(feval(f,a));
sb = sign(feval(f,b));
%
if sa ==sb  % 中間値の定理を利用可能か否かのチェック
    error('In Bisection: Root shoule be in the initial range')
end
%
dx = 0.5*(b-a);
%
tol = dx*tol;
%
x=a+dx;% 中間値で評価
%
% メインループ
%
while abs(dx)>tol
%
    dx=0.5*dx;   % 縮小
%
    if sa ==sign(feval(f,x))
        x=x+dx;  % 中間値が左にあるときには x を増やす

    else
        x=x-dx;  % 中間値が右にあるときには x を減らす
    end
end
```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

x=bisection('bisecfun1',0,1) とタイプすると、0,1 区間内で bisection 手法で Root を求めることができ、その解の値を変数 x に与えることになる。

なお、Matlab では sign は値が正であれば 1, 負であれば-1, ゼロであれば 0 を返す関数である。

また、% はコメント文であり、% から改行までのコードは無視される。このコメント文はデバッグ時に頻繁に使うことになる。

最初のうちは面倒でも大量にコメントを書いておくことをお勧めする。書いている途中では当たり前のことに見えることも、一週間もたつと自分が何を考えいたかわからなくなってしまうからである。また、あるプロジェクトのために比較的大規模のプログラミングを行う際には、できるだけ作業日記をつけると良い。どこに悩み、どうやって解決したかを記録しておくと、例えば、レフェリーから計算やり直しを命じられた時等、長期間経過した後に読み返さねばならない時に非常に参考になる。また、ファイル名も作業途中のものは日時等をつけて、整理しやすくすることをお勧めする。

Bisection 法は、中間値の定理のみに依存する方法であり、ゼロ点を求めたい関数が連続であることだけが必要となっている。したがって、スムースであることや二回微分の存在、凹性等の仮定を課す必要のない、広い応用可能性をもつ手法である。また、存在する限り、また計算精度が確保されている限りは、いつかは必ずゼロ点に収束する。もしも一変数しかない場合は、この手法は強力であり、微分情報を用いる手法よりも信頼できるものである。したがって、bisection はかなり一般的に用いられている。しかしながら、この手法は時間がかかるという欠点もある。もしも関数が微分可能であれば、関数に付随する情報を利用し、より計算量の少ない手法でゼロ点を求めることができる。

3.2 Matlab での応用

Bisection は、マクロ経済学で実際によく使用される。ここでは、非常に単純な、人口成長や技術革新が存在しない、労働供給が外生であるような、標準的な最適成長モデルの定常状態を Bisection で求めてみよう。対数効用の場合のオイラー方程式は

$$-1 + \beta \left[f'(k) + (1 - \delta) \right] = 0.$$

資源制約は

$$f(k) - \delta k - c = 0.$$

と書くことができる。生産関数が CES で

$$Y = F(K, L) = \left(aK^{\psi} + (1-\alpha) L^{\psi}\right)^{1/\psi}$$

のとき、一人当たり資本ストックで書きなおすと

$$f(k) = \left(ak^{\psi} + (1-\alpha)\right)^{1/\psi}.$$

したがって、

$$f'(k) = a\psi k^{\psi-1} \left(ak^{\psi} + (1-\alpha)\right)^{(1-\psi)/\psi}.$$

解くべき非線形連立方程式は消費 c と資本ストック k の関数であり、下記のようになる。

$$\left(ak^{\psi} + (1-\alpha)\right)^{1/\psi} - \delta k - c = 0, \tag{4}$$

$$-1 + \beta \left[a\psi k^{\psi-1} \left(ak^{\psi} + (1-\alpha)\right)^{(1-\psi)/\psi} + (1-\delta)\right] = 0 \tag{5}$$

となる。非線形連立方程式の解法は簡単ではないが、この連立方程式体系では (5) に消費が含まれていないので、事実上一変数の最適化問題となっている。まず (5) を k に関して解き、そこで得られた k の値を (4) に代入して C を求めるのが一番効率的である。そこで、(5) を Bisection で解いてみよう。

まず Euler 方程式の関数を作る。

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function y=euler1(x)
a=0.3;
b=0.9;
phai=0.5;
d=0.1;
y= -1+b*[a*phai*xˆ(1-phai)*(a*xˆ(phai)+(1-a))ˆ((1-phai)/phai)+(1-d)];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

上記を euler1.m として保存し、main program を、例えば下記のようにして作る。

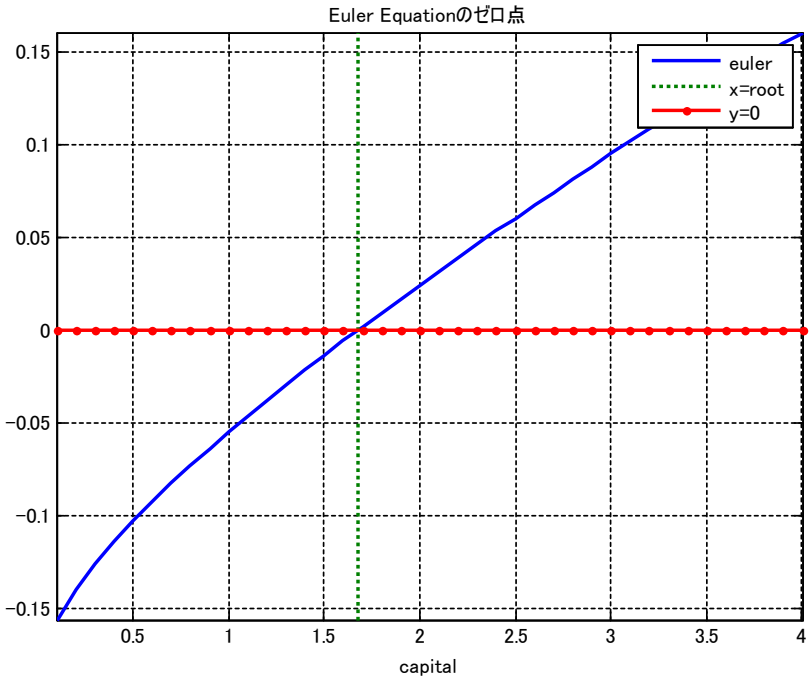
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   2014 年応用マクロ経済学講義用、Root Finding の例
%   naohito abe
clear all
a=0.3;
b=0.9;
```

```
phai=0.5;
d=0.1;
x=bisection('euler1',0,3)
m=40;
res=zeros(m,5);
z=0;
for i=1:m
    z=z+0.1;
    y=euler1(z);
    res(i,1)=z;
    res(i,2)=y;
    res(i,3)=x;
    res(i,4)=y;
    res(i,5)=0;
end
figure(1)
plot(res(:,1),res(:,2), res(:,3),res(:,4),':',res(:,1),res(:,5),'.-','LineWidth',2)
title('Euler Equation のゼロ点'); xlabel('capital'); grid on; legend('euler','x=root',
'x=0');
ylim([res(1,2),res(m,2)]); xlim([res(1,1),res(m,1)]);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

とすると、

```
x =
1.6743
```

という解をえる。なお、出力される図は下記の通り。



3.3 Newton 法

Newton 法とは、非線形関数 f をいくつもの線形関数で近似する手法である。

x^k が我々の最初のゼロ点の Guess だとしよう。その点で関数 f を線形近似すると

$$g(x) \equiv f'\left(x^k\right)\left(x - x^k\right) + f\left(x^k\right)$$

g は x に関して線形の関数であり、x^k において関数 $f(x)$ と接している。この線形関数 g のゼロ点 x^{k+1} を求めると

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

この x^{k+1} を初期値としてまた関数 f を線形近似することで、新たなゼロ点を求めることができる。これを繰り返していくことで、f のゼロ点を求めることができる。

簡単な Matlab コードを書いてみると

%%%%%%%%%%%%%%%%%%%%%%

function [x,fval] = newton1(f,x)

```
%
%   2014 年応用マクロ経済学講義用
%   Naohito Abe
% Newton's Method
%
%
maxsteps =1000;
maxit =1000;
tol = 0.001;
%
for it=1:maxit
      fval= feval(f,x);
      fnorm = norm(fval);
      %
      if fnorm<tol
            return
      end
%
% 数値微分 (だいぶいい加減なやり方)
if x ==0
      dx=0.01;
end
if x~=0
      dx=0.01*x;
end
%
x2=x+dx;
df=(feval(f,x2)-feval(f,x))/dx;
%
%   Newton's Update Formula
%
dx = -(fval/df);
x = x+dx;
%
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

上記のコードを newton1.m と名づけて保存し、前に作った関数のゼロ点を求めてみると、当然のことながら同じ答えを得ることができる。このコードやアルゴリズムをみてわかるように、Newton 法では一階の微分の情報が必

要になっており、かつ crucial な役割を果たしている。そして、割り算を数回行っている。

コンピューターにとり、割り算というのは苦手なものである。特に、数値微分のように、分母がゼロに近くなればなるほど誤差が極めて大きくなる。数値計算では積分よりも微分で苦労することが多いのである。上記の数値微分はずいぶんいい加減なやり方をしている野で、実際に自分で数値微分を行うときには、Judd[1998] の 7 章を参考にし、様々な近似法を利用することをお勧めする。

もしもゼロ点をもとめたい関数の導関数を手で求めることができるのであれば、上記の数値微分は使わずに、自分で導関数を情報としてアルゴリズムに与えるほうがはるかに精度の高い計算を行うことができる。実際、様々なパッケージソフトの最適化アルゴリズムや Equation Solver では自分で導関数を設定する Option がついていることが多い。

3.4 Secant Method

Newton 法の問題点の一つは数値微分を用いるため、誤差が大きくなること、および時間がかかることにある。secant は数値微分を使わずに update する手法である。具体的には、二回の差分方程式として

$$x^{k+1} = x^k - \frac{f\left(x^k\right)\left(x^k - x^{k-1}\right)}{f\left(x^k\right) - f\left(x^{k-1}\right)}$$

と x^{k+1} を定義する。これは、よく見れば明らかであるが、Newton 法の数値微分の箇所を、より粗い近似で置き換えたものである。

Newton 法も Secant 法も、関数形が複雑な場合は収束しないことが多い。また、正しい初期値を与えないとすぐに無限大、無限小に発散していく。Bisection 法が常に収束するのに比して、収束が保障されない Newton、Secant 法の欠点はかなり深刻である。しかしながら、Newton 法は、非線型方程式の解法では標準的なツールとなっている。それは、多変数の場合のスピードゲインが極めて大きいためである。

4 多変数ケース

これまでの一次元ケースを発展し、R^n から R^n への関数を考える。

$$f : R^n \Longrightarrow R^n, x \in R^n$$

つまり、f は n 次元ベクトルで定義され、n 個の値を返す関数である。すなわち

$$f^1(x_1, x_2, ..., x_n) = 0$$
$$f^2(x_1, x_2, ..., x_n) = 0$$

.
.
.

$$f^n(x_1, x_2, ..., x_n) = 0$$

をみたす $x \in R^n$ を見つける問題を考える。

4.1 Gauss-Jacobi, Gauss-Seidel

n 個の非線形連立方程式を同時に解くのは困難であるが、一変数の非線型方程式の場合はこれまで議論してきたように、Bisection や Newton 法で容易に求めることができる。したがって、iteration の反復の際に、一変数の問題に置き換え、ループを一つ増やすことで n 変数の問題を扱うことは自然な発想である。

アルゴリズム 2 *Gauss-Jacobi*
x^k を所与とし、x^{k+1} を下記の n 個の非線形方程式の解で与える

$$f^1\left(x_1^{k+1}, x_2^k, ..., x_n^k\right) = 0$$
$$f^2\left(x_1^k, x_2^{k+1}, ..., x_n^k\right) = 0$$

.
.
.

$$f^n\left(x_1^k, x_2^k, ..., x_n^{k+1}\right) = 0$$

アルゴリズム 3 *Gauss-Seidel*
x^k を所与とし、x^{k+1} を下記の n 個の非線形方程式の解で与える

$$f^1\left(x_1^{k+1}, x_2^k, ..., x_n^k\right) = 0$$
$$f^2\left(x_1^{k+1}, x_2^{k+1}, ..., x_n^k\right) = 0$$

.
.
.

$$f^n\left(x_1^{k+1}, x_2^{k+1}, ..., x_n^{k+1}\right) = 0$$

これらは、単に一次元非線形方程式の解法を組み合わせただけなので、プログラミングは容易であるが、非常に長い時間がかかることが多い。また、関数の並べ方にスピードや収束は依存してくるし、常に収束するとは限らない。そのため、できるだけ、対角行列に近いように、すなわち、argumentsの少ない方程式から順に並べていく、あるいは事前に、極力単純な式に変換してから作業を行うことことが望ましい。

なお、Gauss-Seidel のほうが、update される変数が多い分、Gauss-Jacobi よりも高速に収束することが多いようである。マクロ経済学では、世代重複モデルを数値的に解くときに Gauss-Seidel が使用されることがある。Dynamic Programmin 等の形式に変換せずに、最適化のための一階条件等を多数並べて、それらを満たす解を求めたいときに活用されている。

4.2 Newton's Method for Multivariate Case

Gauss-Seidel や Gauss-Jacobi では各方程式をばらばらに扱っていたが、多変数、多値関数の微分情報、すなわち正方行列になるヤコブ行列を用い、Newton 法を多変数ケースに拡張することも可能である。

アルゴリズム 4 *Newton*

Step1 x^k でヤコブ行列を計算　$A^k = J\left(x^k\right)$。このヤコブ行列が正則であることを確認。

Step2 $x^{k+1} = x^k - \left(A^k\right)^{-1} f\left(x^k\right)$

Step3　$\left\|x^{k+1} - x^k\right\| \leq \varepsilon\left(1 + \left\|x^{k+1}\right\|\right)$ でなければ *Step1* に戻る。

Step4 $\left\|f\left(x^{k+1}\right)\right\| \leq \delta$ であれば *stop.* でなければ収束失敗。異なる初期値を試す。

無論、一変数時と同様に、ヤコブ行列を求めるという作業が必要であり、このため誤差が大きくなる恐れがある。Secant Method の多変数 version も存在する。詳しくは Jadd[1988] を参照すること。

Matlab では、組み込み関数では多変数の方程式を解くコマンドが存在しない。Optimization Toolbox には、fsolve という関数が用意されており、Newton 法により計算することができる。ヤコブ行列に関しては、自分で指定しないかぎり、数値微分で求められたヤコブ行列が使用される。