数値計算入門:　導入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 18 年 10 月 26 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　数値計算に関するメモ

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、計量分析や数値解析を用いない研究でも、その恩恵に預かっている場合は非常に多い。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 が常に存在する。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ある。確かに統計パッケージは常に進化し、計量経済学者達は自分の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応用計量経済学者達に提供するようになっている。しかしながら、それでも非線形の推定や新しい手法に基づく最尤法プログラム等多くのものが現在でも統計パッケージで利用できない。また、コマンドは存在しても、使用に耐えないくらい遅かったり、標準偏差の計算結果に疑問が生じることが多い。一般に、Matlab や Gauss, C,Fortran 等の言語で書かれた後、商用統計パッケージにコードが組み込まれるまではかなり長い時間がかかることを覚悟せねばならない。自分で Matlab や Gauss 等でコードを書くことで、最先端の研究手法を用いることができるのできるのである。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利点と限界を知らねばならない。数値計算の基本に関する知識は、定量的分析を現在行う際にはどうしても必要になってくるのである。

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。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、通常の演算では精度の問題を意識することは少ない。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=1234567891234567 89

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、二時方程式の解である。

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

の解は

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

で与えられる。もしも ac が b に比して小さければ、$-b + \sqrt{b^2 - 4ac}$ 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。

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。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: $x^L < x^R, f(x^L) f(x^R) < 0$ 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 $f(x^L) f(x^M) < 0$ であれば、$x^R = x^M$ とする。$f(x^M) f(x^R) < 0$ であれば、$x^L = x^M$ とする。

Step 3: $x^L - x^R < \varepsilon \left|1 + \left|x^L\right| + \left|x^R\right|\right|, or \quad \left|f(x^M)\right| < \delta$ 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。しかしながら、誤差許容度 δ, ε を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 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

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

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
%   2006 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;   %x を増やす
  else
    x=x-dx;
  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、プログラミングの最中は、できるだけ作業日記をつけると良い。そして、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、凹性等の仮定を課していない、強力な手法である。また、必ずゼロ点に収束する。しかしながら、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 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)
%
%    2006 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。

3.3 Secant Method

Newton 法の問題点の一つは数値微分を用いることにある。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、より荒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。

3.4 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

$$f^1\left(x_1, x_2, ..., x_n\right) = 0$$

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

.
.
.

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

をみたす $x \in R^n$ を見つける問題を考える。

3.5 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-Jacboi*
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。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ことが望ましいが、うまく行くかどうかは問題の性質に依存してくる。

3.6 Newton's Method for Multivariate case

各方程式をばらばら扱うのではなく、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が可能である。

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。