Lecture Notes 5

Solving non–linear systems of equations

The core of modern macroeconomics lies in the concept of equilibrium, which is usually expressed as a system of plausibly non–linear equations which can be either viewed as finding the zero of a a given function $F : \mathbb{R}^n \longrightarrow \mathbb{R}$, such that $x^\star \in \mathbb{R}^n$ satisfies

$$F(x^\star) = 0$$

or may be thought of finding a *fixed point*, such that $x^\star \in \mathbb{R}^n$ satisfies

$$F(x^\star) = x^\star$$

Note however that the latter may be easily restated as finding the zero of $G(x) \equiv F(x) - x$.

5.1 Solving one dimensional problems

5.1.1 General iteration procedures

The idea is here to express the problem as a fixed point, such that we would like to solve the one–dimensional problem of the form

$$x = f(x) \tag{5.1}$$

The idea is then particularly straightforward. If we are given a fixed point equation of the form (5.1) on an interval $\mathscr{I}$, then starting from an initial value $x_0 \in \mathscr{I}$, a sequence $\{x_k\}$ can be constructed by setting

$$x_k = f(x_{k-1}) \text{ for } k = 1, 2, \ldots$$

Note that this sequence can be constructed if for every $k = 1, 2, \ldots$

$$x_k = f(x_{k-1}) \in \mathscr{I}$$

If the sequence $\{x_k\}$ converges — *i.e.*

$$\lim_{x_k \to \infty} x_k = x^\star \in \mathscr{I}$$

then $x^\star$ is a solution of (5.1), such a procedure is called an *iterative procedure*. There are obviously restrictions on the behavior of the function in order to be sure to get a solution.

Theorem 1 (Existence theorem) *For a finite closed interval $\mathscr{I}$, the equation $x = f(x)$ has at least one solution $x^\star \in \mathscr{I}$ if*

1. *f is continuous on $\mathscr{I}$,*
2. *$f(x) \in \mathscr{I}$ for all $x \in \mathscr{I}$.*

If this theorem establishes the existence of at least one solution, we need to establish its uniqueness. This can be achieved appealing to the so–called *Lipschitz condition* for f.

Definition 1 *If there exists a number $K \in [0; 1]$ so that*

$$|f(x) - f(x')| \leqslant K|x - x'| \textit{for all} x, x' \in \mathscr{I}$$

then f is said to be Lipschitz–bounded.

A direct — and more implementable — implication of this definition is that any function f for which $|f'(x)| < K < 1$ for all $x \in \mathscr{I}$ is Lipschitz–bounded. We then have the following theorem that established the uniqueness of the solution

Theorem 2 (Uniqueness theorem) *The equation $x = f(x)$ has at most one solution $x^\star \in \mathscr{I}$ if f is lipschitz–bounded in $\mathscr{I}$.*

The implementation of the method is then straightforward

1. Assign an initial value, x_k, $k = 0$, to x and a vector of termination criteria $\varepsilon \equiv (\varepsilon_1, \varepsilon_2, \varepsilon_3) > 0$
2. Compute $f(x_k)$
3. If either
   (a) $|x_k - x_{k-1}| \leqslant \varepsilon_1 |x_k|$ (Relative iteration error)
   (b) or $|x_k - x_{k-1}| \leqslant \varepsilon_2$ (Absolute iteration error)
   (c) or $|f(x_k) - x_k| \leqslant \varepsilon_3$ (Absolute functional error)

   is satisfied then stop and set $x^\star = x_k$, else go to the next step.
4. Set $x_k = f(x_{k-1})$ and go back to 2.

Note that the stopping criterion is usually preferred to the second one. Further, the updating scheme

$$x_k = f(x_{k-1})$$

is not always a good idea, and we might prefer to use

$$x_k = \lambda_k x_{k-1} + (1 - \lambda_k) f(x_{k-1})$$

where $\lambda_k \in [0; 1]$ and $\lim_{k\to\infty} \lambda_k = 0$. This latter process smoothes convergence, which therefore takes more iterations, but enhances the behavior of the algorithm in the sense it often avoids crazy behavior of x_k.

As an example let us take the simple function

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

such that we want to find $x^\star$ that solves

$$x^\star = \exp((x-2)^2) - 2$$

Let us start from x_0=0.95. The simple iterative scheme is found to be diverging as illustrated in figure 5.1 and shown in table 5.1. Why? simply because the function is not Lipschitz bounded in a neighborhood of the initial condition! Nevertheless, as soon as we set $\lambda_0 = 1$ and $\lambda_k = 0.99\lambda_{k-1}$ the algorithm is able to find a solution, as illustrated in table 5.2. In fact, this trick is a numerical way to circumvent the fact that the function is not Lipschitz bounded.

Figure 5.1: Non–converging iterative scheme

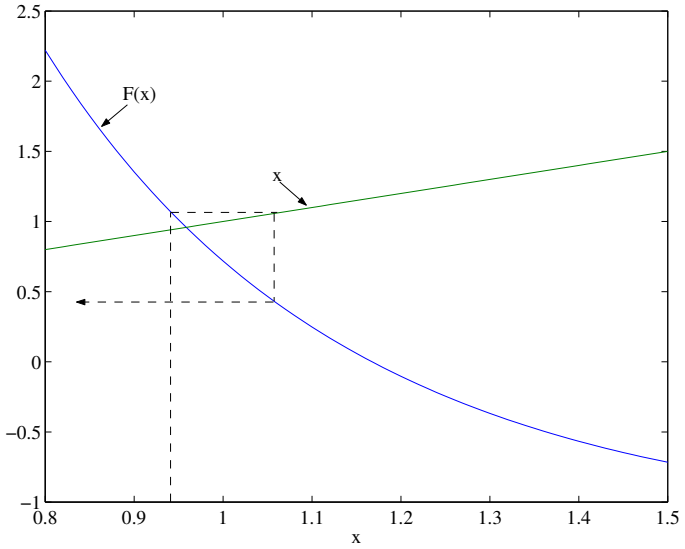


Table 5.1: Divergence in simple iterative procedure

k	x_k	$\lvert x_k - x_{k-1}\rvert/\lvert x_k\rvert$	$\lvert f(x_k) - x_k\rvert$
1	1.011686	6.168583e-002	3.558355e-001
2	0.655850	3.558355e-001	3.434699e+000
3	4.090549	3.434699e+000	7.298439e+001
4	77.074938	7.298439e+001	n.c

Table 5.2: Convergence in modified iterative procedure

k	x_k	λ_k	$\lvert x_k - x_{k-1}\rvert/\lvert x_k\rvert$	$\lvert f(x_k) - x_k\rvert$
1	1.011686	1.0000	6.168583e-002	3.558355e-001
2	1.011686	0.9900	6.168583e-002	3.558355e-001
3	1.007788	0.9801	5.717130e-002	3.313568e-001
4	7 1.000619	0.9703	4.886409e-002	2.856971e-001
5	0.991509	0.9606	3.830308e-002	2.264715e-001
6	0.982078	0.9510	2.736281e-002	1.636896e-001
7	0.973735	0.9415	1.767839e-002	1.068652e-001
8	0.967321	0.9321	1.023070e-002	6.234596e-002
9	0.963024	0.9227	5.238553e-003	3.209798e-002
10	0.960526	0.9135	2.335836e-003	1.435778e-002
11	0.959281	0.9044	8.880916e-004	5.467532e-003
12	0.958757	0.8953	2.797410e-004	1.723373e-003
13	0.958577	0.8864	7.002284e-005	4.314821e-004
14	0.958528	0.8775	1.303966e-005	8.035568e-005
15	0.958518	0.8687	1.600526e-006	9.863215e-006
16	0.958517	0.8601	9.585968e-008	5.907345e-007

5.1.2 Bisection method

We now turn to another method that relies on bracketing and bisection of the interval on which the zero lies. Suppose f is a continuous function on an interval $\mathscr{I} = [a; b]$, such that $f(a)f(b) < 0$, meaning that f crosses the zero line at least once on the interval, as stated by the intermediate value theorem. The method then works as follows.

1. Define an interval $[a; b]$ $(a < b)$ on which we want to find a zero of f, such that $f(a)f(b) < 0$ and a stopping criterion $\varepsilon > 0$.

2. Set $x_0 = a$, $x_1 = b$, $y_0 = f(x_0)$ and $y_1 = f(x_1)$;

3. Compute the bisection of the inclusion interval

$$x_2 = \frac{x_0 + x_1}{2}$$

   and compute $y_2 = f(x_2)$.

4. Determine the new interval

   - If $y_0 y_2 < 0$, then $x^\star$ lies between x_0 and x_2 thus set

$$\begin{array}{lll} x_0 = x_0 & , & x_1 = x_2 \\ y_0 = y_0 & , & y_1 = y_2 \end{array}$$

   - else $x^\star$ lies between x_1 and x_2 thus set

$$\begin{array}{lll} x_0 = x_1 & , & x_1 = x_2 \\ y_0 = y_1 & , & y_1 = y_2 \end{array}$$

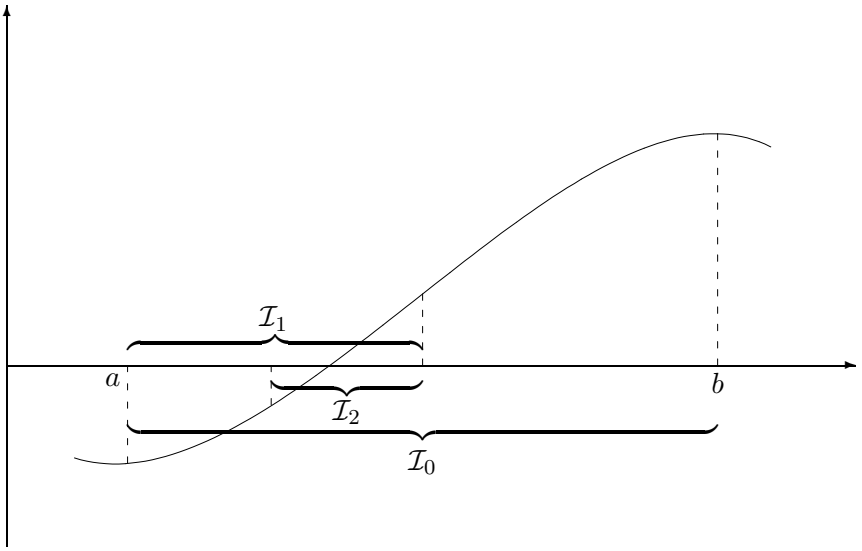
5. if $|x_1 - x_0| \leqslant \varepsilon(1 + |x_0| + |x_1|)$ then stop and set

$$x^\star = x_3$$

This algorithm is illustrated in figure 5.2. Table 5.3 reports the convergence scheme for the bisection algorithm when we solve the problem for finding the fixed point of

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

Figure 5.2: The Bisection algorithm



As can be seen, it takes more iterations than in the previous iterative scheme (27 iteration for a=0.5 and b=1.5, but still 19 with a=0.95 and b=0.96!), but the bisection method is actually implementable in a much greater number of cases as it only requires continuity of the function while not imposing the Lipschitz condition on the function.

MATLAB CODE: BISECTION ALGORITHM

```
function x=bisection(f,a,b,varargin);
%
% function x=bisection(f,a,b,P1,P2,...);
%
% f          : function for which we want to find a zero
% a,b        : lower and upper bounds of the interval (a<b)
% P1,...     : parameters of the function
%
% x solution
%
epsi    = 1e-8;
x0      = a;
x1      = b;
y0      = feval(f,x0,varargin{:});
```

Table 5.3: Bisection progression

iteration	x_2	error
1	1.000000	5.000000e-001
2	0.750000	2.500000e-001
3	0.875000	1.250000e-001
4	0.937500	6.250000e-002
5	0.968750	3.125000e-002
10	0.958008	9.765625e-004
15	0.958527	3.051758e-005
20	0.958516	9.536743e-007
25	0.958516	2.980232e-008
26	0.958516	1.490116e-008

```
y1      = feval(f,x1,varargin{:});

if a>=b
   error('a should be greater than b')
end

if y0*y1>=0
   error('a and b should be such that f(a)f(b)<0!')
end

err = 1;
while err>0;
   x2   = (x0+x1)/2;
   y2   = feval(f,x2,varargin{:});
   if y2*y0<0;
      x1    = x2;
      y1    = y2;
   else
      x0    = x1;
      x1    = x2;
      y0    = y1;
      y1    = y2;
   end
   err      = abs(x1-x0)-epsi*(1+abs(x0)+abs(x1));
end
x   = x2;
```

5.1.3 Newton's method

While bisection has proven to be more stable than the previous algorithm it displays slow convergence. Newton's method will be more efficient as it will take advantage of information available on the derivatives of the function. A simple way to understand the underlying idea of Newton's method is to go back to the Taylor expansion of the function f

$$f(x^\star) \simeq f(x_k) + (x^\star - x_k) f'(x_k)$$

but since $x^\star$ is a zero of the function, we have

$$f(x_k) + (x^\star - x_k) f'(x_k) = 0$$

which, replacing x_{k+1} by the new guess one may formulate for a candidate solution, we have the recursive scheme

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k + \delta_k$$

with $\delta_k = -f(x_k)/f'(x_k0$ Then, the idea of the algorithm is straightforward. For a given guess x_k, we compute the tangent line of the function at x_k and find the zero of this linear approximation to formulate a new guess. This process is illustrated in figure 5.3. The algorithm then works as follows.

1. Assign an initial value, x_k, $k = 0$, to x and a vector of termination criteria $\varepsilon \equiv (\varepsilon_1, \varepsilon_2) > 0$

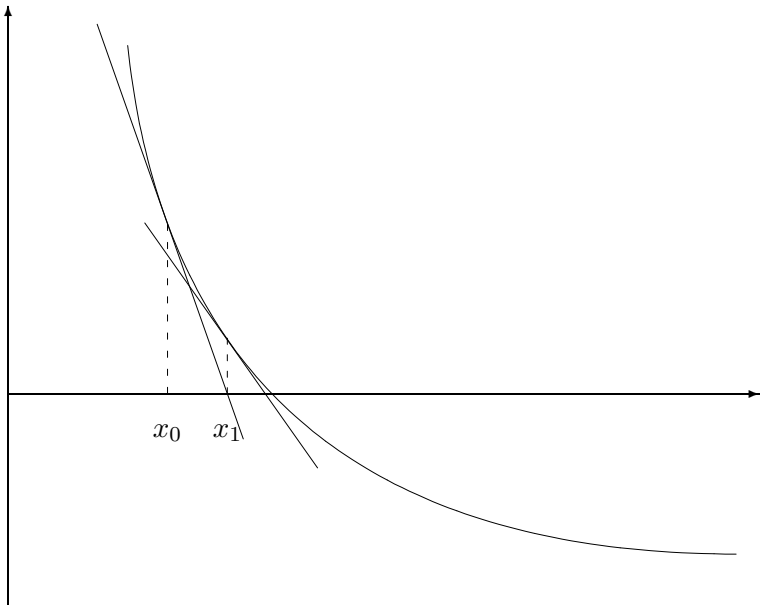
2. Compute $f(x_k)$ and the associated derivative $f'(x_k)$ and therefore the step $\delta_k = -f(x_k)/f'(x_k)$

3. Compute $x_{k+1} = x_k + \delta_k$

4. if $|x_{k+1} - x_k| < \varepsilon_1(1 + |x_{k+1}|)$ then goto 5, else go back to 2

5. if $|f(x_{k+1})| < \varepsilon_2$ then stop and set $x^\star = x_{k+1}$; else report failure.

Figure 5.3: The Newton's algorithm



A delicate point in this algorithm is that we have to compute the derivative of the function, which has to be done numerically if the derivative is not known. Table 5.4 reports the progression of the Newton's algorithm in the case of our test function.

MATLAB CODE: SIMPLE NEWTON'S METHOD

```
function [x,term]=newton_1d(f,x0,varargin);
%
% function x=newton_1d(f,x0,P1,P2,...);
%
% f          : function for which we want to find a zero
% x0         : initial condition for x
% P1,...     : parameters of the function
%
% x          : solution
% Term       : Termination status (1->OK, 0-> Failure)
%
eps1    = 1e-8;
eps2    = 1e-8;
dev     = diag(.00001*max(abs(x0),1e-8*ones(size(x0))));
```

Table 5.4: Newton progression

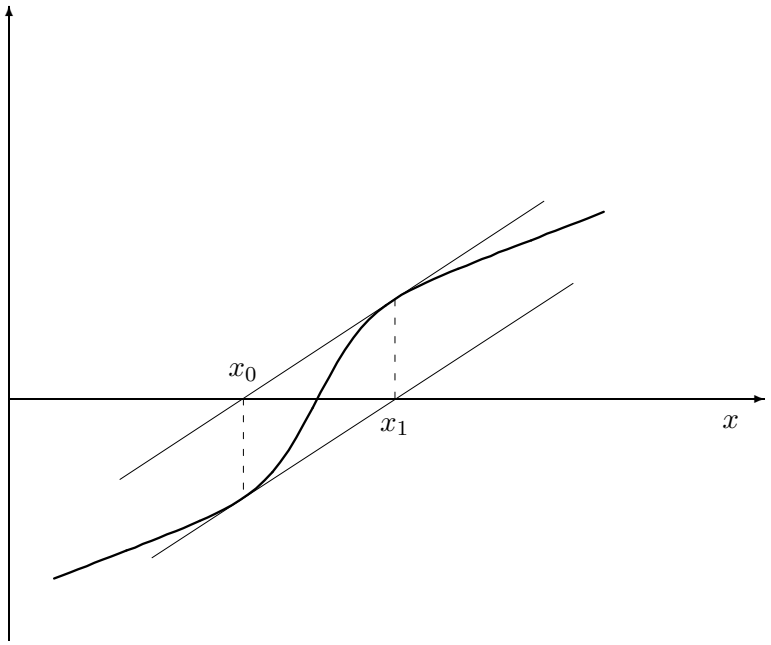
iteration	x_k	error
1	0.737168	2.371682e-001
2	0.900057	1.628891e-001
3	0.954139	5.408138e-002
4	0.958491	4.352740e-003
5	0.958516	2.504984e-005
6	0.958516	8.215213e-010

```
y0      = feval(f,x0,varargin{:});

err     = 1;
while err>0;
   dy0  = (feval(f,x0+dev,varargin{:})-feval(f,x0-dev,varargin{:}))/(2*dev);
   if dy0==0;
      error('Algorithm stuck at a local optimum')
   end
   d0   = -y0/dy0;
   x    = x0+d0
   err  = abs(x-x0)-eps1*(1+abs(x));
   y0   = yeval(f,x,varargin{:})
   ferr = abs(y0);
   x0   = x;
end
if ferr<eps2;
   term = 1;
else
   term = 0;
end
```

Note that in order to apply this method, we need the first order derivative of f to be non–zero at each evaluation point, otherwise the algorithm degenerates as it is stuck a a local optimum. Further the algorithm may get stuck into a cycle, as illustrated in figure 5.4. As we can see from the graph, because the function has the same derivative in x_0 and x_1, the Newton iterative scheme cycles between the two values x_0 and x_1.

Figure 5.4: Pathological Cycling behavior



A way to escape from these pathological behaviors is to alter the recursive scheme. A first way would be to set

$$x_{k+1} = x_k + \lambda\delta_k$$

with $\lambda \in [0, 1]$. A better method is the so–called damped Newton's method which replaces the standard iteration by

$$x_{k+1} = x_k + \frac{\delta_k}{2^j}$$

where

$$j = \min\left\{i : 0 \leqslant i \leqslant i_{max}, \left| f\left(x_k - \frac{1}{2^j}\frac{f(x_k)}{f'(x_k)}\right)\right| < |f(x_k)|\right\}$$

Should this condition impossible to fulfill, one continues the process setting $j = 0$ as usual. In practice, one sets $i_{max} = 4$. However, in some cases, i_{max}, should be adjusted. Increasing i_{max} helps at the cost of larger computational time.

5.1.4 Secant methods (or Regula falsi)

Secant methods just start by noticing that in the Newton's method, we need to evaluate the first order derivative of the function, which may be quite costly. Regula falsi methods therefore propose to replace the evaluation of the derivative by the secant, such that the step is replaced by

$$\delta_k = -f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \text{ for } k = 1, \ldots$$

therefore one has to feed the algorithm with two initial conditions x_0 and x_1, satisfying $f(x_0)f(x_1) < 0$. The algorithm then writes as follows

1. Assign 2 initial value, x_k, $k = 0, 1$, to x and a vector of termination criteria $\varepsilon \equiv (\varepsilon_1, \varepsilon_2) > 0$

2. Compute $f(x_k)$ and the step

$$\delta_k = -f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

3. Compute $x_{k+1} = x_k + \delta_k$

4. if $|x_{k+1} - x_k| < \varepsilon_1(1 + |x_{k+1}|)$ then goto 5, else go back to 2

5. if $|f(x_{k+1})| < \varepsilon_2$ then stop and set $x^\star = x_{k+1}$; else report failure.

MATLAB CODE: SECANT METHOD

```
function [x,term]=secant_1d(f,x0,x1,varargin);
eps1    = 1e-8;
eps2    = 1e-8;
y0      = feval(f,x0,varargin{:});
y1      = feval(f,x1,varargin{:});

if y0*y1>0;
   error('x0 and x1 must be such that f(x0)f(x1)<0');
end

err = 1;
while err>0;
   d    = -y1*(x1-x0)/(y1-y0);
   x    = x1+d;
   y    = feval(f,x,varargin{:});
   err  = abs(x-x1)-eps1*(1+abs(x));
   ferr = abs(y);
   x0       = x1;
   x1       = x;
   y0       = y1;
   y1       = y;
end
if ferr<eps2;
   term=1;
else
   term=0
end
```

Table 5.5 reports the progression of the algorithm starting from $x_0 = 0.5$ and $x_1 = 1.5$ for our showcase function. This method suffers the same convergence problems as the Newton's method, but may be faster as it does not involve the computation of the first order derivatives.

Table 5.5: Secant progression

iteration	x_k	error
1	1.259230	2.407697e-001
2	0.724297	5.349331e-001
3	1.049332	3.250351e-001
4	0.985310	6.402206e-002
5	0.955269	3.004141e-002
6	0.958631	3.362012e-003
7	0.958517	1.138899e-004
8	0.958516	4.860818e-007
9	0.958516	7.277312e-011

5.2 Multidimensional systems

We now consider a system of n equations

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, \ldots, x_n) &= 0 \end{aligned}$$

that we would like to solve for the vector n. This is a standard problem in economics as soon as we want to find a general equilibrium, a Nash equilibrium, the steady state of a dynamic system... We now present two methods to achieve this task, which just turn out to be the extension of the Newton and the secant methods to higher dimensions.

5.2.1 The Newton's method

As in the one–dimensional case, a simple way to understand the underlying idea of Newton's method is to go back to the Taylor expansion of the multi-dimensional function F

$$F(x^\star) \simeq F(x_k) + \nabla F(x_k)(x^\star - x_k)$$

but since $x^\star$ is a zero of the function, we have

$$F(x_k) + \nabla F(x_k)(x^\star - x_k) = 0$$

which, replacing x_{k+1} by the new guess one may formulate for a candidate solution, we have the recursive scheme

$$x_{k+1} = x_k + \delta_k \text{ where } \delta_k = (\nabla F(x_k))^{-1} F(x_k)$$

such that the algorithm then works as follows.

1. Assign an initial value, x_k, $k = 0$, to the vector x and a vector of termination criteria $\varepsilon \equiv (\varepsilon_1, \varepsilon_2) > 0$
2. Compute $F(x_k)$ and the associated jacobian matrix $\nabla F(x_k)$
3. Solve the linear system $\nabla F(x_k)\delta_k = -F(x_k)$
4. Compute $x_{k+1} = x_k + \delta_k$
5. if $\|x_{k+1} - x_k\| < \varepsilon_1(1 + \|x_{k+1}\|)$ then goto 5, else go back to 2
6. if $\|f(x_{k+1})\| < \varepsilon_2$ then stop and set $x^\star = x_{k+1}$; else report failure.

All comments previously stated in the one–dimensional case apply to this higher dimensional method.

MATLAB CODE: NEWTON'S METHOD

```
function [x,term]=newton(f,x0,varargin);
%
% function x=newton(f,x0,P1,P2,...);
%
% f         : function for which we want to find a zero
% x0        : initial condition for x
% P1,...    : parameters of the function
%
% x         : solution
% Term      : Termination status (1->OK, 0-> Failure)
%
eps1    = 1e-8;
eps2    = 1e-8;
x0      = x0(:);
y0      = feval(f,x0,varargin{:});
n       = size(x0,1);
dev     = diag(.00001*max(abs(x0),1e-8*ones(n,1)));

err     = 1;
```

```
while err>0;

   dy0  = zeros(n,n);
   for i= 1:n;
     f0        = feval(f,x0+dev(:,i),varargin{:});
     f1        = feval(f,x0-dev(:,i),varargin{:});
     dy0(:,i)  = (f0-f1)/(2*dev(i,i));
   end

   if det(dy0)==0;
      error('Algorithm stuck at a local optimum')
   end
   d0   = -y0/dy0;
   x    = x0+d0
   y    = feval(f,x,varargin{:});
   tmp  = sqrt((x-x0)'*(x-x0));
   err  = tmp-eps1*(1+abs(x));
   ferr = sqrt(y'*y);
   x0   = x;
   y0   = y;

end

if ferr<eps2;
   term = 1;
else
   term = 0;
end
```

As in the one–dimensional case, it may be useful to consider a damped method, for which the j is computed as

$$j = \min \left\{ i : 0 \leqslant i \leqslant i_{max}, \left\| f\left(x_k + \frac{\delta_k}{2^j} \right) \right\|_2 < \|f(x_k)\|_2 \right\}$$

5.2.2 The secant method

As in the one–dimensional case, the main gain from using the secant method is to avoid the computation of the jacobian matrix. Here, we will see a method developed by Broyden which essentially amounts to define a $\mathbb{R}^n$ version of the secant method. The ideas is to replace the jacobian matrix $\nabla F(x_k)$ by a matrix S_k at iteration k which serves as a guess for the jacobian. Therefore

the step, δ_k, now solves

$$S_k \delta_k = -F(x_k)$$

to get

$$x_{k+1} = x_k + \delta_k$$

The remaining point to elucidate is how to compute S_{k+1}. The idea is actually simple as soon as we remember that S_k should be an approximate jacobian matrix, in other words this should not be far away from the secant and it should therefore solve

$$S_{k+1} \delta_k = F(x_{k+1}) - F(x_k)$$

This amounts to state that we are able to compute the predicted change in F(.) for the specific direction δ_k, but we have no information for any other direction. Broyden's idea is to impose that the predicted change in F(.) in directions orthogonal to δ_k under the new guess for the jacobian, S_{k+1}, are the same than under the old one:

$$S_{k+1} z = S_k z \text{ for } z' \delta_k = 0$$

This yields the following updating scheme:

$$S_{k+1} = S_k + \frac{(\Delta_k^F - S_k \delta_k) \delta_k'}{\delta_k' \delta_k}$$

where $\Delta_k^F = F(x_{k+1}) - F(x_k)$. Then the algorithm writes as

1. Assign an initial value, x_k, $k = 0$, to the vector x, set $S_k = I$, $k = 0$, and a vector of termination criteria $\varepsilon \equiv (\varepsilon_1, \varepsilon_2) > 0$

2. Compute $F(x_k)$

3. Solve the linear system $S_k \delta_k = -F(x_k)$

4. Compute $x_{k+1} = x_k + \delta_k$, and $\Delta_k^F = F(x_{k+1}) - F(x_k)$

5. Update the jacobian guess by

$$S_{k+1} = S_k + \frac{(\Delta_k^F - S_k\delta_k)\delta_k'}{\delta_k'\delta_k}$$

6. if $\|x_{k+1} - x_k\| < \varepsilon_1(1 + \|x_{k+1}\|)$ then goto 7, else go back to 2

7. if $\|f(x_{k+1})\| < \varepsilon_2$ then stop and set $x^\star = x_{k+1}$; else report failure.

The convergence properties of the Broyden's method are a bit inferior to those of Newton's. Nevertheless, this method may be worth trying in large systems as it can be less costly since it does not involve the computation of the Jacobian matrix. Further, when dealing with highly non–linear problem, the jacobian can change drastically, such that the secant approximation may be particularly poor.

MATLAB CODE: BROYDEN'S METHOD

```
function [x,term]=Broyden(f,x0,varargin);
%
% function x=Broyden(f,x0,P1,P2,...);
%
% f         : function for which we want to find a zero
% x0        : initial condition for x
% P1,...    : parameters of the function
%
% x         : solution
% Term      : Termination status (1->OK, 0-> Failure)
%
eps1    = 1e-8;
eps2    = 1e-8;
x0      = x0(:);
y0      = feval(f,x0,varargin{:});
S       = eye(size(x0,1));

err = 1;
while err>0;
   d    = -S\y0;
   x    = x0+d;
   y    = feval(f,x,varargin{:});
   S    = S+((y-y0)-S*d)*d'/(d'*d);
   tmp  = sqrt((x-x0)'*(x-x0));
   err  = tmp-eps1*(1+abs(x));
   ferr = sqrt(y'*y);
   x0   = x;
   y0   = y;
```

```
end
if ferr<eps2;
   term = 1;
else
   term = 0;
end
```

5.2.3 Final considerations

All these methods are numerical, such that we need a computer to find a solution. Never forget that a computer can only deal with numbers with a given accuracy, hence these methods will be more or less efficient depending on the scale of the system. For instance, imagine we deal with a system for which the numbers are close to zero (let's think of a model that is close to the non–trade theorem), then the computer will have a hard time trying to deal with numbers close to machine precision. Then it may be a good idea to rescale the system.

Another important feature of all these methods is that they implicitly rely on a linear approximation of the function we want to solve. Therefore, the system should not be too non–linear. For instance, assume you want to find the solution to the equation

$$\frac{c^{\alpha}}{(c-\overline{c})^{\sigma}} = p$$

where p is given. Then there is a great advantage to first rewrite the equation as

$$c^{\alpha} = p(c-\overline{c})^{\sigma}$$

and then as

$$c^{\frac{\alpha}{\sigma}} = p(c-\overline{c})$$

such that the system is "more" linear. Such transformation often turn out to be extremely useful.

Finally, in a number of cases, we know a solution of a simpler system, and we may use a continuation approach to the problem. For instance, assume we want to solve a system $F(x) = 0$ which is particularly complicated, and we know the solution to the system $G(x) = 0$ to be particularly simple. We may then restate the problem of solving $F(x) = 0$ as solving

$$\lambda G(x) + (1 - \lambda)F(x) = 0$$

with $\lambda \in [0; 1]$. We first start with $\lambda = 1$ get a first solution x_0, and then take it as an initial condition to solve the system for $\lambda_1 = 1 - \varepsilon$, $\varepsilon > 0$ and small, to get x_1. This new solution is the used as an initial guess for the problem with $\lambda_2 < \lambda_1$. This process is repeated until we get the solution for $\lambda = 0$. This may seem quite a long process, but in complicated method, this may actually save a lot of time instead of spending hours to finding a good initial value for the algorithm. Judd [1998] (chapter 5) reports more sophisticated continuation methods — known as *homotopy* methods — that have proven to be particularly powerful.

Bibliography

Judd, K.L., *Numerical methods in economics*, Cambridge, Massachussets: MIT Press, 1998.

Index

Contents

List of Figures

List of Tables