

Static Optimization with MATLAB

Day 1: Unconstrained Optimization

GSEFM Pre-Semester Course

Luca Schmitz

Institute for Monetary and Financial Stability, Goethe University Frankfurt

September 22-28, 2026 | Frankfurt

Why do we need numerical optimization?

- Analytic expressions may not exist or are too complicated to obtain
- Theorems from analytical optimization are still useful
 - existence and uniqueness of solution(s)
 - as starting point for algorithm design

Overview

Root-Finding Methods:

- Bisection
- Fixed-Point Iteration [▶ Appendix](#)
- Newton's Method
- Secant Method (Quasi-Newton) [▶ Appendix](#)
- fzero
- fsolve

Optimization Methods:

- Bracketing [▶ Appendix](#)
- Line Search Algorithms (Gradient Descent, Newton-Raphson, Quasi-Newton)
- fminbnd
- fminsearch
- fminunc

About the different algorithms

- You don't have to know all the details of these various algorithms (they are already implemented in a smart way e.g. in MATLAB)
- ⇒ Computer Science or Mathematics PhD trying to figure out to most efficiently design such an algorithm ⇒ not your research business.
- However, you should know the basic idea of the algorithms and their advantages and disadvantages.

Root-Finding Algorithms

Root-finding methods:

- Bisection
- Fixed-Point Iteration [▶ Appendix](#)
- Newton's Method
- Secant Method (Quasi-Newton) [▶ Appendix](#)
- fzero
- fsolve

Numerical Root-Finding

What does root-finding have to do with optimization?

- Most economic problems are (by assumption) well-behaved $\Rightarrow$ the root(s) of the FOCs define the optimum, e.g.

$$\nabla f(x^*) = 0$$

Bisection

Advantages:

- Simple and robust method
- Univariate
- Derivative-free

- Builds on the **Intermediate Value Theorem**: ► Definition

If $f : \mathbb{R} \rightarrow \mathbb{R}$ continuous on $[x_l, x_h]$ and $f(x_l) < 0 < f(x_h)$, then $\exists x^*$ such that $f(x^*) = 0$

Disadvantages:

- Inefficient, might be very slow
- Ignores information about the function's curvature
- not necessarily finds all roots

Bisection: Algorithm

Idea: iterative: choose bounds, take middle, update

1. Choose

- bounds $x_l < x_h$ s.th. $f(x_l)f(x_h) < 0$
- tolerance criteria $\epsilon, \delta > 0$

2. Compute the midpoint: $x_m = \frac{x_l + x_h}{2}$

3. Refine bounds:

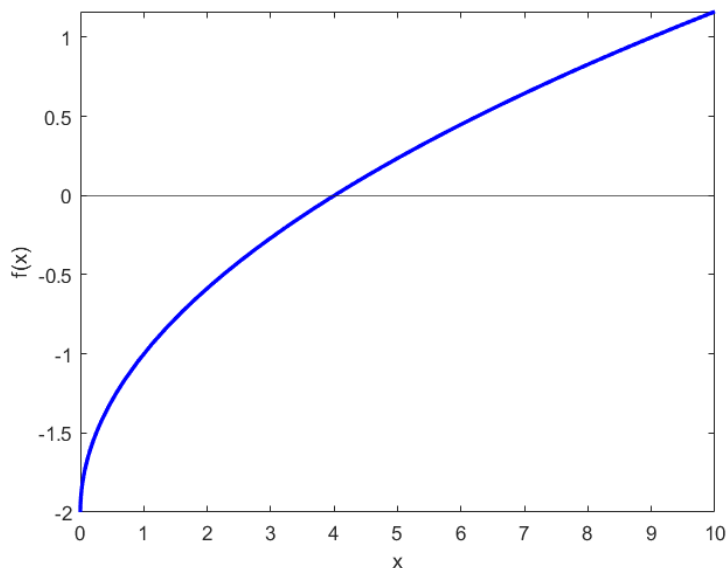
- if $f(x_l)f(x_m) < 0$, set $x_h = x_m$ and $f(x_h) = f(x_m)$
- else set $x_l = x_m$ and $f(x_l) = f(x_m)$

4. Check stopping rule:

- If $x_h - x_l \leq \epsilon(1 + |x_l| + |x_h|)$ or if $|f(x_m)| \leq \delta$ stop and report x_m
- else go back to step 2

Bisection: Example

Let's say we want to find the root of the function $f(x) = x^{\frac{1}{2}} - 2 = 0$ for $x \geq 0$:



Need initial brackets whose function values are of opposing sign: take $[x_l, x_h] = [0, 10]$.

Bisection: Example (continued)

Function call

```
%% Bisection

% define a function whose root we want to find
myfun = @(x) x.^0.5 -2;

% set the inputs for the function bisec
eps = 0.00001;      % distance criterion for stopping, domain
delt = eps;         % distance criterion for stopping, range
bounds = [0, 10];   % initial bounds
                    % remember that they have to bracket  $f(x) = 0$ , i.e
                    % bound values must lead to function values of opposing
                    % sign

% Call the function bisec.m
[x,f_val,ef] = bisec(myfun, bounds, eps, delt);
```

Bisection Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% bisec.m calculates a zero of the function 'fun' between the points
% a and c by the bisection method
% Inputs:
%   fun : function whose roots are found
%   x    : initial bracket values [xl,xh]
%   tolX: tolerance level, domain
%   tolFun: tolerance level, range
%
% Output:
%   x    : root
%   fx   : function value at the root
%   ef   : exit flag: 0: initial brackets do not bracket a root
%           1: algorithm succeeded in finding a root
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [x,fx,ef] = bisec(fun,x,tolX,tolFun)
```

```
xl = x(1); fl = fun(xl); % initial low bracket value and associated function value
xh = x(2); fh = fun(xh); % initial high bracket value and associated function value
```

Bisection Function (continued)

```
% check initialization
if (fh*fl > 0)
    disp('initial [xl, xh] do not bracket a root');
    x = -Inf; fx = -Inf; ef = 0;
    return
end

% bisection
while (xh-xl)>tolX*(1+abs(xl)+abs(xh)) || abs(fm)>tolFun

    % calculate middle point and associated function value
    xm = (xl+xh)/2; fm = fun(xm);

    % Update bounds
    if fl*fm < 0
        xh = xm; fh = fm;
    else
        xl = xm; fl = fm;
    end
end

% results to be exported
x = xm; fx = fm; ef = 1;
```

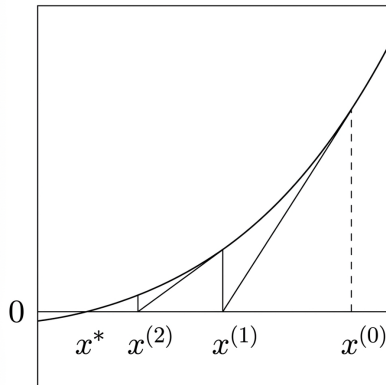
Example Results

Workspace	
Name ▲	Value
 ans	<i>1x1 ConstantLine</i>
 bounds	[0,10]
 delt	1.0000e-05
 ef	1
 eps	1.0000e-05
 f	<i>1x1 sym</i>
 f_val	3.8147e-06
 myfun	@(x)x.^0.5-2
 x	<i>1x1 sym</i>

Indeed, the algorithm found a root at $x = 4$.

Newton's Method: Idea

- Based on the principle of **successive linearization**:
⇒ replace the nonlinear problem by a sequence of simpler linear problems whose solutions converge to the solution of the nonlinear problem
- ultimate goal: find the root of the nonlinear function $f(x)$
- Calculate tangents to f by first-order Taylor series expansion and use convergence of the roots of the tangents to find the root of the original function



Newton's Method: Algorithm

Let $f : \mathbb{R} \rightarrow \mathbb{R}$ (univariate) be differentiable with $f'(x^{(i)})$ denoting the first derivative evaluated at $x^{(i)}$

- start with guess $x^{(0)}$
- $f(x)$ is approximated by its first-order Taylor series expansion about $x^{(0)}$ (i.e. the tangent to f at $x^{(0)}$) $\Rightarrow$ set this to zero instead of the original function:

$$f(x) \approx f(x^{(0)}) + f'(x^{(0)})(x - x^{(0)}) = 0$$

we can rearrange this to

$$x = x^{(0)} - [f'(x^{(0)})]^{-1}f(x^{(0)})$$

so, the left-hand side is the root of the tangent to $f(x^{(0)}) \Rightarrow$ call it $x^{(1)}$ and let it serve as our new guess for the root of f

Newton's Method: Algorithm (continued)

$\Rightarrow$ this yields the general iteration rule

$$x^{(k+1)} \leftarrow x^{(k)} - [f'(x^{(k)})]^{-1}f(x^{(k)})$$

- repeat approximation of f at next root and in turn get root of the linear approximation to f around that point through the above rule
- continue until the roots of the tangent lines converge (and then roughly coincide with the root of f)

Newton's Method: Algorithm (continued)

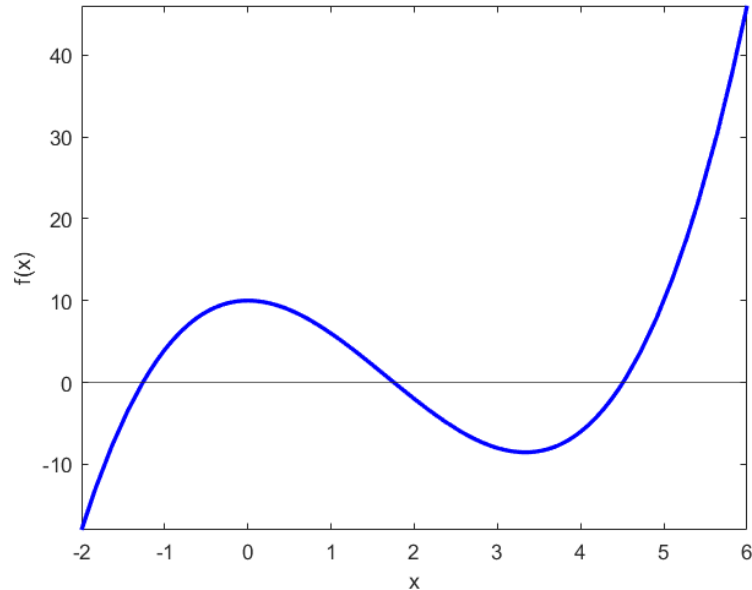
1. Set $i=0$ and choose
 - starting value $x^{(0)}$
 - stopping rule parameters $\varepsilon, \delta > 0$
 - maximum number of iterations I
2. compute next iterate according to

$$x^{(i+1)} \leftarrow x^{(i)} - [f'(x^{(i)})]^{-1}f(x^{(i)})$$

3. Check stopping rule:
 - If $|x^{(i)} - x^{(i+1)}| \leq \varepsilon(1 + |x^{(i+1)}|)$ or if $i = I \Rightarrow$ go to the next step
 - else set $i = i + 1$ and to back to step 2
4. Convergence assessment
 - if $|f(x^{(i+1)})| \leq \delta$ report success
 - otherwise report failure

Newton's Method: Example

Let's consider the root(s) of the function $f(x) = x^3 - 5x^2 + 10$



Newton's Method: Function call

```
%% Newton's Method

% define a non-linear function whose root we want to find
myfun = @(x) x^3 - 5*x^2 + 10;
% and also supply its first derivative
dmyfun = @(x) 3*x^2 - 10*x;

% set the inputs for the function newton.m
x0 = -4;           % initial starting point, function has to be defined
                   % for this point
eps  = 0.00001;    % distance criterion for stopping, domain
delt = eps;        % distance criterion for stopping, range
maxit = 1000;      % maximum number of iterations

% Call the function newton.m
[xstar,f_val,ef,i] = newton(myfun,dmyfun,x0, eps, delt, maxit);
```

Newton's Method: Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% newton.m calculates a zero of the function 'fun' with the Newton Method
% Inputs:
%   fun : function whose roots are to be found
%   dfun: first derivative of that function
%   x0  : initial guess
%   tolX: tolerance level, domain
%   tolFun: tolerance level, range
%   maxit: maximum number of iterations
%
% Output:
%   xnew: root
%   fnew: function value at the root
%   ef   : exit flag: 0: no convergence achieved
%           1: algorithm succeeded in finding a root
%   i    : number of iterations needed
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xnew,fnew,ef,i] = newton(fun,dfun,x,tolX,tolFun,maxit)
```

Newton's Method: Function (continued)

```
for i=1:maxit

    % initial function values
    fx = fun(x);
    dfx = dfun(x);

    % Newton algorithm
    % get new guess according to updating rule
    xnew = x - dfx\fx;

    % function value at new point
    fnew = fun(xnew);

    % create logical convergence criterion
    convcrit = (abs(x-xnew)<=tolX*(1+abs(xnew)) && abs(fnew)<=tolFun);
```

Newton's Method: Function (continued)

```
if convcrit == 1
    ef = 1;
    disp(['The root of f found is ', num2str(xnew)])
    disp(['The function value is ', num2str(fnew)])
    disp(['The number of iterations required was ', num2str(i)])
    break;
else
    % update old guess with new value
    x = xnew;
end
end
```

Newton's Method: Pros and Cons

Advantages:

- in theory, Newton's method converges if f is continuously differentiable and if $x^{(0)}$ is sufficiently close to a root of f at which f is invertible.
- robust to starting value if f is well-behaved
- can easily extended to multivariate case

Disadvantages:

- sensitive to starting value if f behaves erratically
- must be able to compute the first derivative of the function of interest analytically
⇒ also prone to human errors

fzero

- finds root of nonlinear univariate function
- look up documentation via `doc fzero`
- look up source code via `edit fzero` or `open fzero`

fsolve

- solves system of nonlinear multivariate equations
- look up documentation via `doc fsolve`
- look up source code via `edit fsolve` or `open fsolve`

Optimization Methods

Optimization Methods:

- Bracketing [► Appendix](#)
- Line Search Algorithms (Gradient Descent, Newton-Raphson, Quasi-Newton)
- Trust Region Algorithms (not covered here)
- `fminbnd`
- `fminsearch`
- `fminunc`

Line Search Algorithms: Idea

- Determine or calculate a descent direction for f at x , that is, a $d \in \mathbb{R}^n$ with $\nabla f(x)^T d < 0$.
 - Because

$$\nabla f(x)^T d = \lim_{t \rightarrow 0^+} \frac{f(x + td) - f(x)}{t}$$

$\Rightarrow f(x + td) < f(x)$ for all sufficiently small $t > 0$.

- Determine a certain step size $t > 0$ that guarantees a certain minimum descent, and set the new approximation as

$$x^+ := x + td$$

About the different algorithms I

You perhaps already noticed that the algorithms will differ on how:

- to choose the step size t , and
- the descent direction d .

In the following we will not talk about the choice of step sizes, but only about the descent direction. These algorithms for choosing the step size are called Armijo, Wolfe-Powell rules, etc.

Gradient Descent

- The simplest line search algorithm is the **gradient descent** method.
- The descent direction is given by the negative gradient, i.e., $d = -\nabla f(x)$.
- The step size is determined by a **line search** algorithm, which is a one-dimensional optimization problem.
- The new approximation is given by

$$x^+ = x - t\nabla f(x)$$

Newton-Raphson method: Idea

- The **Newton-Raphson method** is a second-order method, i.e., it uses the Hessian matrix.
- The descent direction is given by the solution of the linear system

$$\nabla^2 f(x)d = -\nabla f(x)$$

- The step size is determined by a line search algorithm.
- The new approximation is given by

$$x^+ = x + td$$

Newton-Raphson Method: Procedure

- Guess a scalar $x^{(0)}$
- Given $x^{(k)}$, the subsequent iterate $x^{(k+1)}$ is computed by maximizing the second-order Taylor Approximation to f about $x^{(k)}$:

$$f(x) \approx f(x^{(k)}) + f'(x^{(k)})(x - x^{(k)}) + \frac{1}{2}f''(x^{(k)})(x - x^{(k)})^2$$

- Solving the first-order condition

$$f'(x^{(k)}) + f''(x^{(k)})(x - x^{(k)}) = 0$$

yields the iteration rule

$$x^{(k+1)} \leftarrow x^{(k)} - [f''(x^{(k)})]^{-1}f'(x^{(k)})$$

Newton-Raphson Method: Algorithm

1. Set $i = 0$ and choose
 - starting point $x^{(0)}$
 - stopping rule parameters $\epsilon, \delta > 0$
 - maximum number of iterations l
2. Compute next iterate according to

$$x^{(k+1)} \leftarrow x^{(k)} - [f''(x^{(k)})]^{-1} f'(x^{(k)})$$

3. Check stopping rule
 - if $|x^{(k)} - x^{(k+1)}| \leq \epsilon(1 + |x^{(k+1)}|)$ or if $i = l$ go to the next step
 - else $i = i + 1$ and go back to step 2
4. Check outcome:
 - if $|f'(x^{(k+1)})| \leq \delta(1 + |f(x^{(k+1)})|)$ report success
 - otherwise report failure

Newton-Raphson Method: Function Call

```
%% Newton-Raphson
```

```
% set inputs to the function newtonraph.m
```

```
eps = 0.00001;      % distance criterion for stopping, domain
```

```
delt = eps;          % distance criterion for stopping, range
```

```
maxit = 1000;        % maximum number of iterations
```

```
myfun = @(x) x^3 - 5*x^2 + 10; % function whose optimum we want to find
```

```
dmyfun = @(x) 3*x^2 - 10*x;    % first derivative
```

```
ddmyfun = @(x) 6*x - 10;      % second derivative
```

```
x0 = 4;                  % starting value
```

```
[xstar,dfxstar,ef,it] = newtonraph(myfun,dmyfun,ddmyfun, x0, eps,delt, maxit);
```

Newton-Raphson Method: Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% newtonraph.m calculates a local optimum of the function 'fun' with the
%           Newton-Raphson Method
% Inputs:
%   fun : function whose roots are to be found
%   dfun: first derivative of that function
%   x0  : initial guess
%   tolX: tolerance level, domain
%   tolFun: tolerance level, range
%   maxit: maximum number of iterations
%
% Output:
%   xnew: root
%   dfxnew : derivative value at optimum point
%   ef   : exit flag: 0: no convergence achieved
%           1: algorithm succeeded in finding a local optimum
%   i    : number of iterations needed
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [xnew,dfxnew,ef,i] = newtonraph(fun,dfun,ddf,fun,x,tolX,tolFun,maxit)
```

Newton-Raphson Method: Function (continued)

```
ef = 0;
for i=1:maxit
    % calculate function and derivatives at current guess
    fx = fun(x);
    dfx = dfun(x);
    ddfx = ddfun(x);

    % update the guess according to iteration rule
    xnew = x - ddfx\dfx;
    dfxnew = dfun(xnew);

    % check convergence criterion
    if (abs(x-xnew)<= tolX*(1+abs(xnew))) ==1

        % check whether derivative at point is actually zero
        if (abs(dfx) <= tolFun*(1+abs(fx))) == 1
            ef = 1;
            break;
        else
            break;
        end
    else
        x = xnew;
    end
end
```

Newton-Raphson Method: Pros and Cons

Advantages:

- Robust to starting value if f is well-behaved, for example globally concave

Disadvantages:

- if f is not well-behaved, the method can be sensitive to the starting value
- requires the computation of both the first and second derivatives of the objective function
- only guarantee that the objective function value is increased in the direction of the Newton step if $f''(x^{(k)})$ is negative definite, otherwise might find saddle point or minimum

⇒ mostly only used when objective function is globally concave

Quasi-Newton Methods

- The **quasi-Newton methods** are also second-order methods.
- The descent direction is given by the solution of the linear system

$$B_k d = -\nabla f(x)$$

where B_k is an approximation of the Hessian matrix.

- The step size is determined by a line search algorithm.
- The new approximation is given by

$$x^+ = x + td$$

fminsearch

- finds minimum of unconstrained multivariable function using derivative-free method
- look up documentation via `doc fminsearch`
- look up source code via `edit fminsearch` or `open fminsearch`

fminunc

- finds minimum of unconstrained multivariable function using (optional) derivative-based methods
- look up documentation via `doc fminunc`
- look up source code via `edit fminunc` or `open fminunc`

fminbnd

- finds minimum of univariate function on fixed interval
- look up documentation via `doc fminbnd`
- look up source code via `edit fminbnd` or `open fminbnd`

References I

- Alt, W. (2011). *Nichtlineare optimierung eine einführung in theorie, verfahren und anwendungen: Eine einführung in theorie, verfahren und anwendungen*. Vieweg+Teubner Verlag.
- Chiang, A. C., & Wainwright, K. (2005). *Fundamental methods of mathematical economics*. McGraw-Hill Education.
- de la Fuente, A. (2000). *Mathematical methods and models for economists*. Cambridge University Press.
- Geiger, C., & Kanzow, C. (1999). *Numerische verfahren zur lösung unrestringierter optimierungsaufgaben (springer-lehrbuch)*. Springer.
- Geiger, C., & Kanzow, C. (2002). *Theorie und numerik restringierter optimierungsaufgaben*. Springer.
- Ulbrich, M. (2011). *Nichtlineare optimierung*. Birkhäuser Verlag.
- Werner, J. (2002). Unrestringierte optimierungsaufgaben [Last visited on 2023/09/19]. <http://num.math.uni-goettingen.de/werner/uncopt.pdf>
- Werner, J. (2012). Vorlesung über optimierung [Last visited on 2023/09/19]. <https://num.math.uni-goettingen.de/werner/optim.pdf>

Intermediate Value Theorem

Theorem (Intermediate Value Theorem)

Let f be a continuous real-valued function on the closed and bounded interval $[a, b]$. Then for each number γ (strictly) between $f(a)$ and $f(b)$, there exists a point $c \in (a, b)$ such that $f(c) = \gamma$.

$\Rightarrow$ It implies that if f is continuous and $f(a)$ and $f(b)$ have different signs, then f must have **at least one root in $[a, b]$** .

[► Go Back](#)

Fixed Point Iteration I

Consider the univariate case:

- Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be continuous.
- Consider the problem of finding the fixed-point $x = f(x)$, i.e. the intersection of the function with the 45-degree line
- can also be applied to root-finding problems when we redefine the original problem $d(x) = 0$ into $x = f(x) = x - d(x)$

Convergence

- since f is continuous, if the iterates converge, they converge to a fixed point of f
- in theory, it works if f is differentiable and the initial value is closed to a fixed point x^* of f at which $|f'(x^*)| < 1$ (Banach fixed-point theorem)

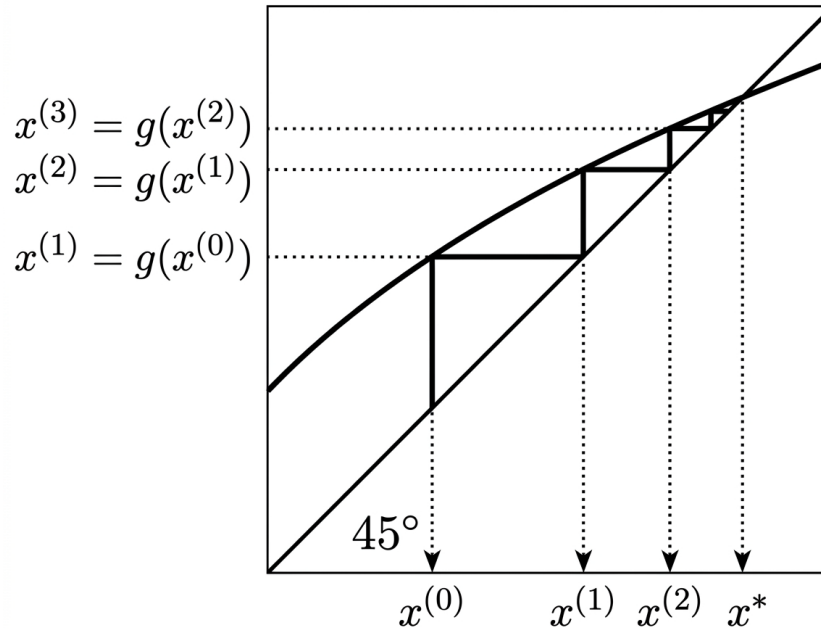
Fixed Point Iteration II

- Idea: Start with a guess $x^{(0)}$

⇒ subsequent iterates are generated using the iteration rule

$$x^{(k+1)} \leftarrow f(x^{(k)})$$

Fixed Point Iteration III

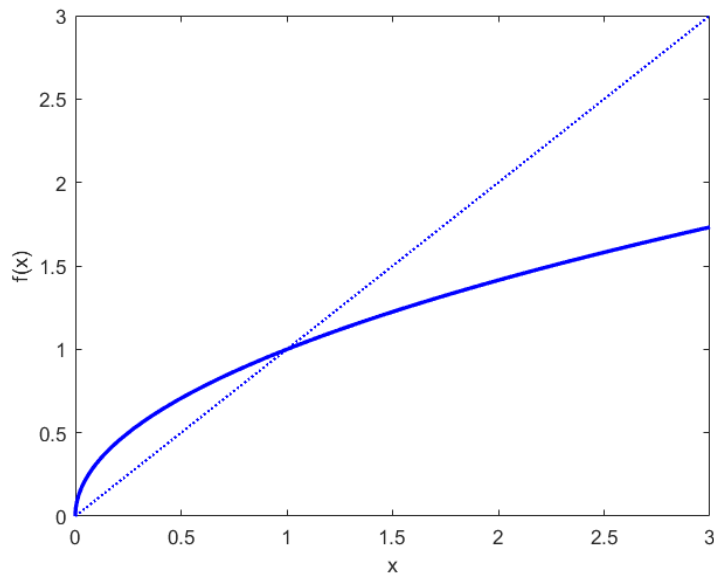


Fixed Point Iteration IV

1. Set
 - the starting point $x^{(0)}$
 - Stopping rule parameters ϵ , δ , maximum number of iterations l
2. Compute next iterate: $x^{(i+1)} = f(x^{(i)})$
3. Check stopping rule:
 - if $|x^{(i)} - x^{(i+1)}| \leq \epsilon(1 + |x^{(i+1)}|)$ or if $i = l$ continue to next step
 - else set $i = i + 1$ and go back to step 2
4. Solution assessment
 - if $|f(x^{(i+1)}) - x^{(i+1)}| \leq \delta$ report success
 - otherwise report failure

Fixed Point Iteration V

Let's say we want to find the fixed point of the function $f(x) = x^{\frac{1}{2}}$



Fixed Point Iteration VI

%% Fixed Point Iteration

% define a function whose fixed point we want to find

```
myfun = @(x) x.^0.5;
```

% set the inputs for the function fixpoint.m

```
x0 = 3; % initial starting point, function has to be defined  
      % for this point
```

```
eps = 0.00001; % distance criterion for stopping, domain
```

```
delt = eps; % distance criterion for stopping, range
```

```
maxit = 1000; % maximum number of iterations
```

% Call the function fixpoint.m

```
[xstar,f_val,ef] = fixpoint(myfun, x0, eps, delt, maxit);
```

Fixed Point Iteration VII

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% fixpoint.m calculates the fixed point of the function 'fun' by fixed
% point iteration.
% Inputs:
%   fun : function whose fixed point is found
%   x0  : initial starting point
%   tolX: tolerance level, domain
%   tolFun: tolerance level, range
%
% Output:
%   x    : fixed point
%   fx   : function value at the fixed point
%   efout: exit flag: 0: no convergence
%           1: algorithm succeeded in finding a fixed point
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [x,fx,efout] = fixpoint(fun,x0,tolX,tolFun,maxit)

% set exit flag to no convergence:
ef = 0;
```

Fixed Point Iteration IX

```
% fixed-point iteration
for i=1:maxit

    % calculate the function value
    fx0 = fun(x0);

    % set new x0 to freshly obtained function value
    xnew = fx0;
    % calculate distance between new guess and its function value
    df = fun(xnew)-xnew;

    % Check the stopping rule: logical=1 if fulfilled
    conveys = abs(x0-xnew) <= tolX*(1+abs(xnew)) && (abs(df) <= tolFun);
    if conveys == 1
        % set exit flag to positive value and exit the for-loop
        ef = 1;break;
    else % convergence not yet achieved
        % update new guess
        x0 = xnew;
    end
end

% results to be exported
x = xnew; fx = fx0; efout = ef;
```

Fixed Point Iteration X

Workspace	
Name ▲	Value
 delt	1.0000e-05
 ef	1
 eps	1.0000e-05
 f_val	1.0000
 maxit	1000
 myfun	@(x)x.^0.5
 x0	3
 xstar	1.0000

Indeed, fixed point iteration converged and found fixed-point of 1.
But, try e.g. $f(x) = x^2$ for $x^{(0)} = 0.3, x^{(0)} = 0.9$ and $x^{(0)} = 3$.

Fixed Point Iteration XI

Advantages:

- Easy to implement
- Often converges when the sufficiency conditions are not met

Disadvantages:

- Difficult to adjust to multidimensional functions
- Often also non-convergence

► [Go Back](#)

Quasi-Newton Methods: Idea

- same successive linearization approach
- but: replace Jacobian (f') by an approximation ($\tilde{J}$) which is easier to compute $\Rightarrow$ analyst does not need to provide an explicit derivative expression
- Univariate case: Secant Method
- Multivariate case: i) Broyden, ii) Inverse Broyden, iii) Broyden update of QR-Decomposition of $\tilde{J}$

Disadvantages:

- Converges more slowly than Newton's method
- requires an initial approximation of the function's Jacobian

The Secant Method: Idea

- the most widely used univariate Quasi-Newton method
- in fact identical to the univariate Newton method except that it replaces the derivative of f with an approximation constructed from the function values at the two previous iterates:

$$f'(x^{(k)}) \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$$

- This yields the iteration rule

$$x^{(k+1)} \leftarrow x^{(k)} - \frac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})} f(x^{(k)})$$

$\Rightarrow$ it thus requires 2 starting values instead of 1

Secant Method: Example

Let's take the same function as for Newton's Method and see if we get the same roots.

$$f(x) = x^3 - 5x^2 + 10$$

Secant Method: Function Call

```
%% Secant Method
```

```
% define a non-linear function whose root we want to find
```

```
myfun = @(x) x^3 - 5*x^2 + 10;
```

```
% set the inputs for the function secant.m
```

```
x0 = [3.5,4];           % initial starting vector, function has to be defined
```

```
% for both points
```

```
eps = 0.00001;          % distance criterion for stopping, domain
```

```
delt = eps;              % distance criterion for stopping, range
```

```
maxit = 1000;            % maximum number of iterations
```

```
% Call the function secant.m
```

```
[xstar,f_val,ef,i] = secant(myfun,x0, eps, delt, maxit);
```

Secant Method: Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% secant.m calculates a zero of the function 'fun' with the Secant Method
% Inputs:
%   fun : function whose roots are to be found
%   x0  : initial guess, vector with two scalars
%   tolX: tolerance level, domain
%   tolFun: tolerance level, range
%   maxit: maximum number of iterations
%
% Output:
%   xnew: root
%   fnew: function value at the root
%   ef   : exit flag: 0: no convergence achieved
%           1: algorithm succeeded in finding a root
%   i    : number of iterations needed
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [xnew,fnew,ef,i] = secant(fun,x,tolX,tolFun,maxit)
```

Secant Method: Function (continued)

```
ef = 0;      % set exit flag to 0
x1 = x(1);
x2 = x(2);

□ for i=1:maxit

    % initial function values
    fx1 = fun(x1);
    fx2 = fun(x2);

    % inverse slope between the two points
    m = (x2-x1)/(fx2-fx1);

    % Secant algorithm
    % get new guess according to updating rule
    xnew = x2 - m*fx2;

    % function value at new point
    fnew = fun(xnew);
```

Secant Method: Function (continued)

```
% create logical convergence criterion
convcrit = (abs(x2-xnew)<=tolX*(1+abs(xnew)) && abs(fnew)<=tolFun);

if convcrit == 1
    ef = 1;
    disp(['The root of f found is ', num2str(xnew)])
    disp(['The function value is ', num2str(fnew)])
    disp(['The number of iterations required was ', num2str(i)])
    break;
else
    % update old guess with new value
    x1 = x2;
    x2 = xnew;
end
end
```

Quasi-Newton Methods: Broyden Algorithm

0. Set $k = 0$. Choose:
 - a) x^k , and initial guess for approximate Jacobian, $\tilde{J}^k$
 - b) Stopping rule parameters $\epsilon, \delta > 0$
 - c) Maximum number of iterations K
1. Compute next iterate: $x^{k+1} = x^k - \tilde{J}^{k-1} \cdot f(x^k)$.
2. Update approximate Jacobian: $\tilde{J}^{k+1} = \tilde{J}^k + \frac{(\Delta f^k - \tilde{J}^k \cdot \Delta x^k) \cdot (\Delta x^k)^T}{(\Delta x^k)^T \cdot \Delta x^k}$
with $\Delta f^k = f(x^{k+1}) - f(x^k)$, $\Delta x^k = x^{k+1} - x^k$
3. Check stopping rule: if $\|x^k - x^{k+1}\| \leq \epsilon \cdot (1 + \|x_{k+1}\|)$ or if $k = K$ go to step 4; else $k = k + 1$ and go to step 1.
4. If $\|f(x^{k+1})\| \leq \delta$ report success; else report failure.

Quasi-Newton Methods: Inverse Broyden Algorithm

0. Set $k = 0$. Choose:
 - a) x^k , and initial guess for inverse Jacobian, $\tilde{B}^k = \tilde{J}^{k-1}$
 - b) Stopping rule parameters $\epsilon, \delta > 0$
 - c) Maximum number of iterations K
1. Compute next iterate: $x^{k+1} = x^k - \tilde{B}^k \cdot f(x^k)$.
2. Update approximate inverse Jacobian:
$$\tilde{B}^{k+1} = \tilde{B}^k + \frac{(\Delta x^k - \tilde{B}^k \cdot \Delta f^k) \cdot (\Delta x^k)^T \cdot \tilde{B}^k}{(\Delta x^k)^T \cdot \tilde{B}^k \cdot \Delta f^k} \text{ with}$$
$$\Delta f^k = f(x^{k+1}) - f(x^k), \Delta x^k = x^{k+1} - x^k$$
3. Check stopping rule: if $\|x^k - x^{k+1}\| \leq \epsilon \cdot (1 + \|x^{k+1}\|)$ or if $k = K$ go to step 4; else $k = k + 1$ and go to step 1.
4. If $\|f(x_{k+1})\| \leq \delta$ report success; else report failure.

Bracketing: Idea

- Derivative-free method
- place successively smaller brackets around a local maximum of a univariate function
- Goal: try to find the local maximum of a continuous univariate function $f(x)$ on the interval $[a, c]$
- if $\exists b \in [a, c]$ such that $f(a), f(c) < f(b)$, then there exists a local maximum in (a, c)

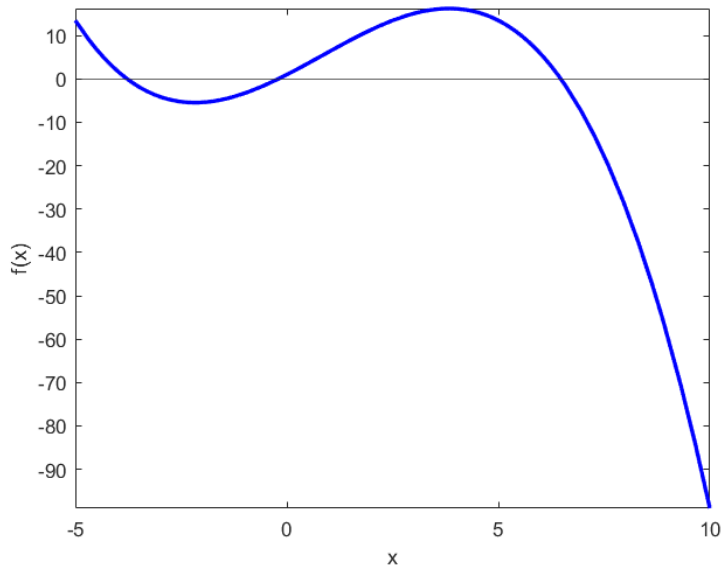
Bracketing: Algorithm

1. Choose
 - $a < b < c$ such that $f(a), f(c) < f(b)$
 - stopping rule parameter $\epsilon > 0$
2. Choose new point by taking middle of larger bracket
 - if $b - a < c - b$, set $d = \frac{b+c}{2}$
 - else set $d = \frac{a+b}{2}$
3. compute $f(d)$
4. Refine the triplet (a, b, c)
 - if $d < b$ and $f(d) > f(b)$, replace (a, b, c) with (a, d, b)
 - else if $d < b$ and $f(d) < f(b)$, replace (a, b, c) with (d, b, c)
 - else if $d > b$ and $f(d) > f(b)$, replace (a, b, c) with (b, d, c)
 - else, replace (a, b, c) with (a, b, d)
5. Check stopping rule
 - if $c - a < \epsilon(1 + |a| + |c|)$ stop and report solution d
 - else go back to step 2

Bracketing: Example

Let's say we want to find a local maximum of the following univariate function

$$f(x) = -0.2x^3 + 0.5x^2 + 5x + 1$$



Bracketing: Function call

```
%% Bracketing
```

```
% set the inputs for the function bracket.m
```

```
myfun = @(x) x^3 - 5*x^2 + 10;
```

```
x0 = [-2;-1;3];           % initial starting vector [a,b,c]',  
                                % such that a < b < c and f(b) > f(a),f(c)
```

```
eps = 0.00001;           % distance criterion for stopping, domain
```

```
% 2nd example
```

```
myfun2 = @(x) - 0.2*x^3+ 0.5*x^2 + 5*x + 1;
```

```
x02 = [0;2;7];
```

```
% call the function bracket.m
```

```
[xstar,f_val,ef] = bracket(myfun2, x02, eps);
```

Bracketing: Function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% bracket.m calculates a local maximum of the function 'fun' on the interval
%           [a,c] by bracketing
% Inputs:
%   fun : function whose roots are to be found
%   x0  : initial brackets, column vector with three scalars, ordered
%   tolX: tolerance level, domain
%
% Output:
%   x    : local maximizer
%   fx   | : function value at maximizer
%   ef   : exit flag: 0 if initialization failed,
%           1 if everything worked well
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [x,fx,ef] = bracket(fun,x0,tolX)
```

Bracketing: Function (continued)

```
% extract brackets from vector and evaluate them
xa = x0(1,1); fa = fun(xa);
xb = x0(2,1); fb = fun(xb);
xc = x0(3,1); fc = fun(xc);

% check the initialization: fb shall have the highest value
if (fa > fb) || (fc > fb)
    fprintf('Initialization failed;')
    x = -Inf; fx = -Inf; ef=0;
    return;
end

% bracketing algorithm
while (xc-xa) > tolX*(1+abs(xa)+abs(xc))
    % set new middle point to evaluate
    if (xb-xa) > (xc-xb) % lower interval is bigger
        xd = (xb+xa)/2;
    else
        xd = (xc+xb)/2;
    end
end
```

Bracketing: Function (continued)

```
% evaluate mid point:
fd = fun(xd);

% set new brackets
if (xd < xb) && (fd > fb)
    xc = xb; xb = xd;
elseif (xd < xb) && (fd < fb)
    xa = xd;
elseif (xd > xb) && (fd > fb)
    xa = xb; xb = xd;
else
    xc = xd;
end

end

% values to export
x = xd; fx = fd; ef = 1;
```

Bracketing: Pros and Cons

Advantages:

- does not require the evaluation of function derivatives
- guaranteed to find a local optimum in a known number of steps

Disadvantages:

- slow
- only local minimum in $[a, c]$

► Go Back