Lecture 02 - Solving nonlinear equations

Baojian Zhou

DATA830001, Numerical Computation
School of Data Science, Fudan University

Sep. 11th, 2024

Solving nonlinear equations

Given $f : \mathbb{R} \to \mathbb{R}$, we say that f has a **root** at $x = r$ if $f(r) = 0$.

The goal of this lecture is to find such root r

$$f(r) = 0. \tag{1}$$

To check the existence of r, we have the following theorem

Theorem 1.1 (Existence of a root)

Let f be continuous on $[a, b]$, satisfying $f(a)f(b) < 0$. Then f has a root in $[a, b]$, that is, there exists a number $r \in [a, b]$ and $f(r) = 0$.

Example: Consider $f(x) := e^x - \sin x - 2$, note

- $f(0) = -1 < 0, f(\pi) = e^\pi - 2 > 0$.
- By the above theorem, we know f has a root in $[0, \pi]$.

Useful Fact: Taylor's Theorem

Let f be an n-times differentiable function in a neighborhood of $a \in \mathbb{R}$. Recall that the Taylor polynomial of order n of f at a is the polynomial

$$P_n(x) = f(a) + f'(a)(x-a) + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n.$$

Let $f(x) = P_n(x) + R_n(x)$ where $R_n(x)$ is the remainder. We have

Theorem 1.2 (Lagrange's formula for the remainder)

If f has an $f^{(n+1)}$ in $[a, x]$ then there is some $a \leq \xi \leq x$ such that

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-a)^{n+1}.$$

- Useful in error analysis.
- For example, let $a = r$, and $x = x_t$, we have $R_2(x_t) = \mathcal{O}(e_t^3)$.

Algorithm 1 Bisect(f, a, b, ϵ)

Output: a root of f
$n = 0, a_n = a, b_n = b$
$f(a_n)f(b_n) < 0, a_n < b_n$
while $(b_n - a_n)/2 > \epsilon$ **do**
    $c_n = (a_n + b_n)/2$
    if $f(c_n) = 0$ **then**
        return c_n
    end if
    if $f(a_n)f(c_n) < 0$ **then**
        $b_n = c_n$
    else
        $a_n = c_n$
    end if
    $n = n + 1$
end while
Return c_n

- **Accuracy**
  After n iterations, $c_n = (a_n + b_n)/2$. The error $|r - c_n|$ is bounded by
  $$|r - c_n| < \frac{b - a}{2^{n+1}}.$$
- **Time complexity**
  $\mathcal{O}(n + 2)$, i.e. $n + 2$ function evaluations.
- **Iteration complexity**
  $\lceil \log_2 ((b - a)10^p) \rceil$, where p is correct number of decimal places and we want $|r - c_n| \leq .5 \times 10^p$.

Fixed-Point Iteration

The read number r is a **fixed point** of the function g if $g(r) = r$. Given an initial point x_0, we do the following iterations

$$\textbf{FPI} : x_{i+1} = g(x_i) \tag{2}$$

Theorem 1.3 (The convergences of FPI)

If g is continuous and x_i converges to r, then r is a fixed point.

$$g(r) = g(\lim_{i\to\infty} x_i) = \lim_{i\to\infty} g(x_i) = \lim_{i\to\infty} x_{i+1} = r.$$

Fact: The problem of finding root r of $f(x) = 0$ can be turned into the problem of finding fixed-point r of $g(x)$.

Examples

- $g(x) := x + f(x)$
- $g(x) := x + f(x)/f'(x)$ if $f'(x) \neq 0$
- $g(x) := x - f(x)/f'(x)$ if $f'(x) \neq 0$ (You will see this again!)

Different forms of FPI

Consider $f(x) = x^3 + x - 1 = 0$ and the following forms of g

$$g_1(x) := 1 - x^3, \quad g_2(x) := \sqrt[3]{1-x}, \quad g_3(x) = \frac{1+2x^3}{1+3x^2}$$

t	$x_t = g_1(x_{t-1})$	$x_t = g_2(x_{t-1})$	$x_t = g_3(x_{t-1})$
0	0.50000000	0.50000000	0.50000000
1	0.87500000	0.79370053	0.71428571
2	0.33007813	0.59088011	0.68317972
3	0.96403747	0.74236393	0.68232842
4	0.10405419	0.63631020	0.68232780
5	0.99887338	0.71380081	0.68232780
6	0.00337606	0.65900615	0.68232780
7	0.99999996	0.69863261	0.68232780
8	0.00000012	0.67044850	-
9	1.00000000	0.69072912	-
10	0.00000000	0.67625892	-
11	1.00000000	0.68664554	-
12	0.00000000	0.67922234	-
13	-	0.68454401	-

All three iteration procedure starts from $x_0 = 0.5$. Some observations:

- $x_{t+1} = g_1(x_t)$ cannot converge properly.
- $x_{t+1} = g_2(x_t)$ converges but relatively slow.
- $x_{t+1} = g_3(x_t)$ converges very fast.

Error analysis of FPI

Theorem 1.4 (Linear convergence of FPI)

Assume that g is continuously differentiable, that $g(r) = r$, and that $S = |g'(r)| < 1$. Then Fixed-Point Iteration converges linearly with rate S to the fixed point r for initial guesses sufficiently close to r.

Definition 1.5 (Linear Convergence)

Let $e_t = |r - x_t|$ denote the error at step t of an iterative method. If

$$\lim_{i \to \infty} \frac{e_{t+1}}{e_t} = S < 1, \tag{3}$$

the method is said to obey *linear convergence* with rate S. An iterative method is called **locally convergent** to r if the method converges to r for initial guesses sufficiently close to r.

Iteration Complexity of FPI

Set tolerance to ϵ. We may have the following ways

- $|x_{t+1} - x_t| < \epsilon$
- $\frac{|x_{t+1}-x_t|}{|x_{t+1}|} < \epsilon$ or $\frac{|x_{t+1}-x_t|}{\max(|x_{t+1}|,\theta)} < \epsilon$ (useful when the solution is near 0)

Time complexity

$\mathcal{O}(n)$ where n is the total number of iteration needed.

Iteration complexity

Theorem 1.6 (Iteration complexity)

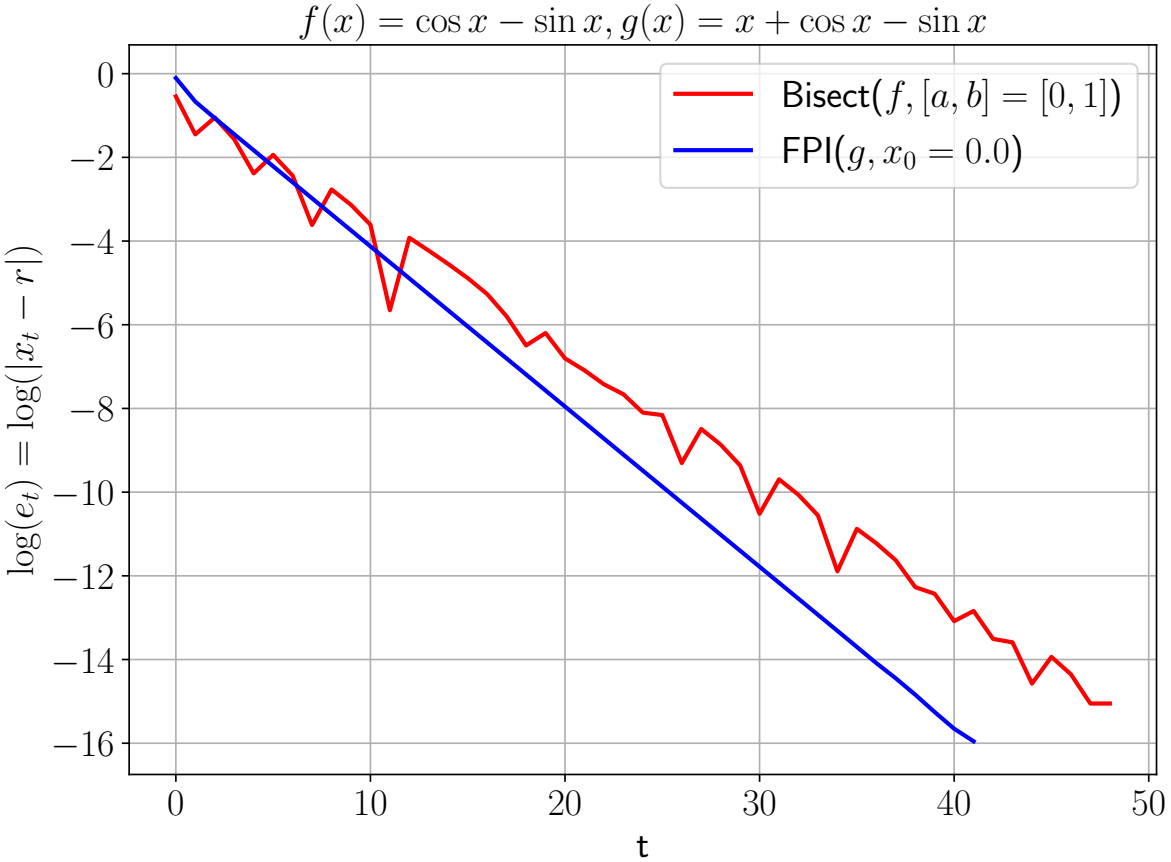
Given the tolerance ϵ and assume that $|g'(x)| \leq m$, if we want to have $e_t := |x_t - r| \leq \epsilon$, then the number of iteration needed is

$$t \geq \left\lceil \ln\left(\frac{\epsilon(1-m)}{|x_1 - x_0|}\right) \Big/ \ln m \right\rceil .$$

Example 1: Comparison with Bisect

$$f(x) = \cos x - \sin x$$

$$g(x) := x + \cos x - \sin x$$

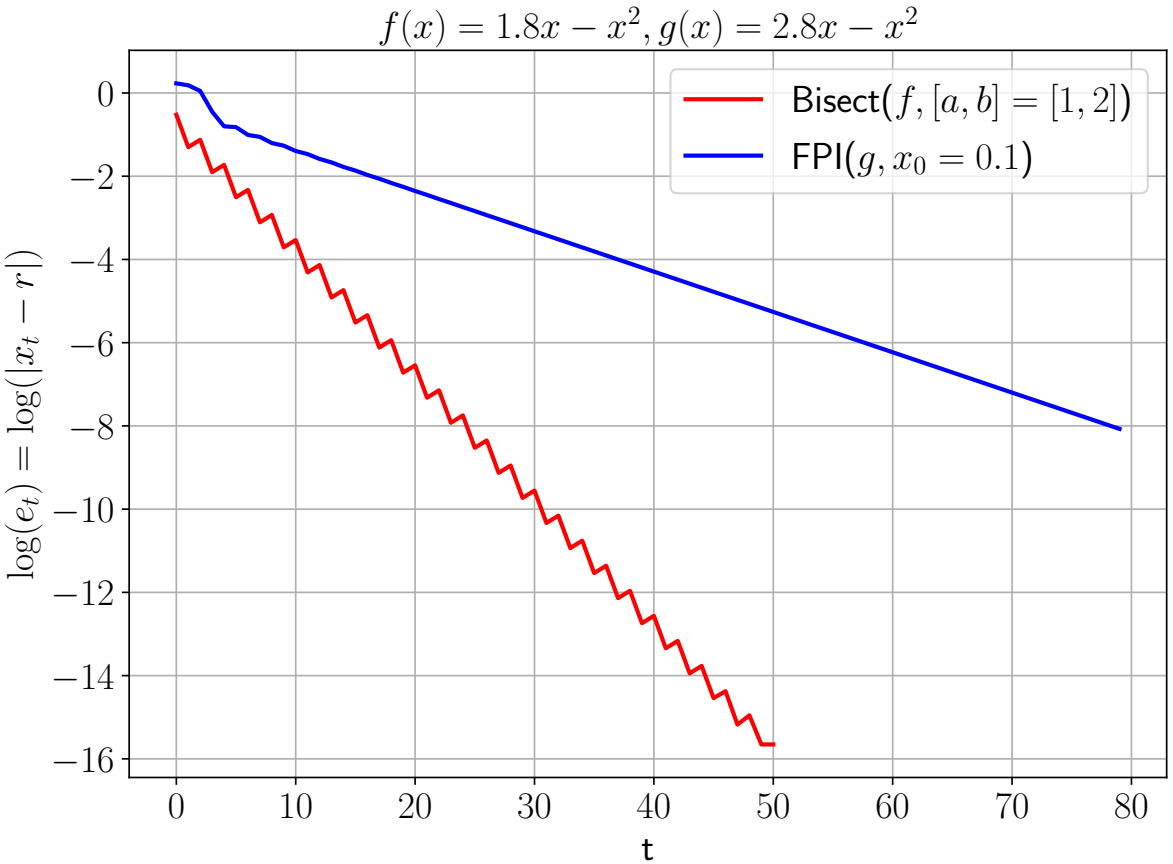


- FPI is faster than Bisect (expected as $|g'(r)| \approx 0.414$)
- Bisect is "unsteady"
- FPI is "steady": error is monotonically decreasing!

Example 2: Comparison with Bisect

$$f(x) = 1.8x - x^2$$

$$g(x) := 2.8x - x^2$$



- FPI is slower than Bisect (expected as $|g'(r)| = 0.8$)
- FPI is "unsteady" at the beginning
- FPI is "steady": error is monotonically decreasing!

FPI for $x = \cos x$

Recall that we introduced an interesting case when we use the calculator: push the cosine button when some initial value will always converge to a fixed number. The iteration procedure is FPI!

$$x_{t+1} = \cos x_t$$

t	x_t	$e_t = \lvert x_t - r \rvert$	e_t / e_{t-1}
0	1.0000000000000	2.609148667848393e-01	-
5	7.0136877362e-01	3.7716359592e-02	0.693376
10	7.4423735490e-01	5.1522216854e-03	0.670767
15	7.3836920412e-01	7.1592909284e-04	0.674004
20	7.3918439977e-01	9.9266556333e-05	0.673558
25	7.3907136530e-01	1.3767916216e-05	0.673620
30	7.3908704270e-01	1.9094801715e-06	0.673611
35	7.3908486839e-01	2.6482844651e-07	0.673612
40	7.3908516994e-01	3.6729393860e-08	0.673612
45	7.3908512812e-01	5.0940468510e-09	0.673612
50	7.3908513392e-01	7.0649985862e-10	0.673612

FPI for $x^2 = a$, i.e., $\sqrt{a}$

Recall that we introduced a modern way to calculate $\sqrt{a}$ with

$$x_{t+1} = \frac{1}{2}\left(x_t + \frac{a}{x_t}\right).$$

This can be derived via FPI. We showcase $a = 2$.

t	x_t	$e_t = \lvert x_t - r \rvert$	e_t / e_{t-1}^2
0	5.000000000000000000000000	3.585786437626905e+0	-
1	2.700000000000000000000000	1.285786437626905e+0	0.10000000
2	1.720370370370370370370370	3.061568079972753e-1	0.18518519
3	1.441455368177650201331579	2.724180580455515e-2	0.29063509
4	1.414470981367771002489897	2.574189946759537e-4	0.34687165
5	1.414213585796883763046612	2.342378871424492e-8	0.35348905
6	1.414213562373095242787195	1.939855065882499e-16	0.35355338
7	1.414213562373095048801688	1.330434729502770e-32	0.35355339
8	1.414213562373095048801688	6.258095016769949e-65	0.35355339
9	1.414213562373095048801688	1.384647774597878e-129	0.35355339

Newton's method

Key Idea: Draw the tangent line at point x_t and use the intersection point of this line with the x-axis as an approximation root. One point on the tangent line is $(x_t, f(x_t))$. The point-slope formula for the equation of a line is $y - f(x_t) = f'(x_t)(x - x_t)$. The intersection point can be found by letting $y = 0$. That is,

$$y - f(x_t) = f'(x_t)(x - x_t)$$
$$x - x_t = -\frac{f(x_t)}{f'(x_t)}$$
$$x = x_t - \frac{f(x_t)}{f'(x_t)}.$$

Quadratically Convergent of Newton's

Definition 2.1 (Quadratically convergent)

Let $e_t = |x_t - r|$ denote the error after step t of an iterative method. The iteration is quadratically convergent if

$$M = \lim_{t \to \infty} \frac{e_{t+1}}{e_t^2} < \infty. \tag{4}$$

Theorem 2.2 (Quadratically convergent of the Newton's)

Let f be twice continuously differentiable and $f(r) = 0$. If $f'(r) \neq 0$, then Newton's Method is locally and quadratically convergent to r. The error e_t at step t satisfies

$$\lim_{t \to \infty} \frac{e_{t+1}}{e_t^2} = M, \text{ where } M = \frac{f''(r)}{2f'(r)}. \tag{5}$$

Newton's method for $x^3 + x - 1 = 0$

Algorithm 2 Newton(f, x_0)

1: $x_0 =$ initial guesses
2: **for** $t = 0, 1, 2, \ldots,$ **do**
3: $\quad x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$
4: **end for**
5: **Return** x_{t+1}

Consider $f(x) = x^3 + x - 1$ and use the Newton's method to find a root of $f(x) = 0$. The iteration table of the Newton's method shows as the following: (use Python decimal with getcontext().prec = 200)

t	x_t	$e_t = \lvert x_t - x^* \rvert$	e_t / e_{t-1}^2
0	-0.70000000000000000000000000	1.382327803828019e+0	-
1	0.12712550607287449392712550	5.552022977551448e-1	0.29055555
2	0.95767811917566125895252010	2.753503153476419e-1	0.89327066
3	0.73482779499450153790970260	5.249999116648221e-2	0.69244945
4	0.68459177068492666790987680	2.263966856907341e-3	0.82139415
5	0.68233217420448415354840460	4.370376464826179e-6	0.85266556
6	0.68232780384433235138256250	1.631302401307875e-11	0.85407651
7	0.68232780382801932736971100	2.272830855039658e-22	0.85407924
8	0.68232780382801932736948370	4.411968456107219e-44	0.85407924
9	0.68232780382801932736948370	1.662505011372050e-87	0.85407924
10	0.68232780382801932736948370	2.360609180609911e-174	0.85407924

Initial point x_0 is important

Consider the following equation

$$f(x) = 4x^4 - 6x^2 - 11/4 = 0$$

The iteration procedure of Newton's is

$$x_{t+1} = x_t - \frac{4x_t^4 - 6x_t^2 - 11/4}{16x_t^3 - 12x_t}.$$

Given $x_0 \in [-10, 10]$

- $x_0 \in [-.62, -.35] \cup [.35, .62]$, it alternatively gives $x_t = -\frac{1}{2}, x_{t+1} = \frac{1}{2}$.
- $x_0 \in \{0\}$, Newton's cannot apply.
- It converges in other cases.

Newton's for multiplicative roots

Newton's method does not always converge quadratically.
Consider $f(x) = x^2$, when it applies Newton's method, we have

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)} = x_t - \frac{x_t^2}{2x_t} = \frac{x_t}{2}.$$

The above convergence is linear. It is similar for $f(x) = x^m$, we have $x_{t+1} = x_t - \frac{x_t^m}{m x_t^{m-1}} = \frac{m-1}{m} x_t$.

Example: $f(x) = \sin x + x^2 \cos x - x^2 - x$, and estimate the number of steps of Newton's method required to converge within six correct places.

Is it possible to fix this issue?

Modified Newton's for multiplicative roots

Key Idea: Try to shift more on $f(x_t)/f'(x_t)$, that is

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)} - \frac{(m-1)\cdot f(x_t)}{f'(x_t)}$$

Theorem 2.3

If f is $(m+1)$-times continuously differentiable on $[a,b]$, which contains a root r of multiplicity $m > 1$, then Modified Newton*'s Method*

$$x_{t+1} = x_t - \frac{m \cdot f(x_t)}{f'(x_t)} \tag{6}$$

converges locally and quadratically to r.

- Bad side: We need to know m explicitly.

Secant method

The Newton's method converges very fast. But, it needs to have derivative information, which may not be available. Can we make any approximation based on Newton's method?

Key Idea: Approximate the derivative by constructing a secant line!

$$\text{An approximation of } f'(x_t) \text{ at} x_t : f'(x_t) \approx \frac{f(x_t) - f(x_{t-1})}{x_t - x_{t-1}}. \tag{7}$$

Algorithm 3 Secant(f, x_0, x_1)

1: x_0, x_1 be initial guesses
2: **for** $t = 1, 2, \ldots,$ **do**
3: $\quad x_{t+1} = x_t - \frac{f(x_t)(x_t - x_{t-1})}{f(x_t) - f(x_{t-1})}$
4: **end for**
5: **Return** x_{t+1}

Convergence analysis of Secant

Theorem 2.4 (Convergence of Secant)

under the assumption that the Secant Method converges to r and $f(r) = 0$, the approximate error relationship

$$e_{t+1} \approx \left| \frac{f''(r)}{2f'} \right| e_t e_{t-1}, \tag{8}$$

this implies that

$$e_{t+1} \approx \left| \frac{f''(r)}{2f'} \right| e_t^{\alpha}, \tag{9}$$

where $\alpha = (1 + \sqrt{5})/2 \approx 1.62$.

- The convergence factor $1 < \alpha < 2$.
- It is superlinear!

Secant for $x^3 + x - 1 = 0$

Consider $f(x) = x^3 + x - 1$ and use the Secant method to find a root of $f(x) = 0$. The iteration table of the Secant method shows as the following: ($e_{t+1} = |x_{t+1} - x^*|$, use Python decimal with getcontext().prec = 200)

t	x_t	e_t	$e_t / e_{t-1}^{\alpha}, \alpha = \frac{1+\sqrt{5}}{2}$
0	0.00000000000000000000000000	6.823278038280193e-1	-
1	1.00000000000000000000000000	3.176721961719807e-1	0.58963610
2	0.50000000000000000000000000	1.823278038280193e-1	0.58963610
3	0.63636363636363636363636363	4.596416746438296e-2	1.16591657
4	0.69005235602094240836769634	7.724552192923081e-3	0.72174624
5	0.68202041964818558443655010	3.073841798337429e-4	1.12751983
6	0.68232578140989279837544690	2.022418126528994e-6	0.80384864
7	0.68232780435902570912687990	5.310063817573961e-10	0.97481374
8	0.68232780382801841015864900	9.172108347492460e-16	0.86774892
9	0.68232780382801932736948330	4.159748582088937e-25	0.93233725
10	0.68232780382801932736948370	3.258625294159891e-40	0.89187309
11	0.68232780382801932736948370	1.157709700479240e-64	0.91666904
12	0.68232780382801932736948370	3.222049900695358e-104	0.90126413
13	0.68232780382801932736948370	3.185885035992595e-168	0.91075405
14	0.68232780382801932736948370	0.000000000000000e-185	0.90487723

Secant method with False Position

- Hybird idea of Bisect and Secant
- Different from Bisect, the midpoint is replaced by a Secant method-like approximation. Given x_0, x_1 such that $f(x_0)f(x_1) < 0$, we can make the next point be

$$x_2 = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)},$$

  for next iteration, we $x_0 = x_1$ and $x_1 = x_2$.
- Unlike Secant's method, False Position can make sure the point is within the range $[x_0, x_1]$ (i.e., assume that $f(x_0) < f(x_1)$).
- Bisection Method guarantees to cut the uncertainty by $1/2$ on each step, but False Position makes no such promise.

Dekker's Method

Dekker's Idea

Try Secant first and switch to Bisection when needed.

- Interval values a and b containing root
  - $\text{sign}(f(a)) \neq \text{sign}(f(b))$
  - If $|f(a)| < |f(b)|$ swap a and b
- c previous value of b (initially a)
- Midpoint $m = \frac{a+b}{2}$
- Try to compute secant intercept s using b and c
  - If dividing by zero $s = m$
- Update a, b, and c
- Repeat until $|b - a| < \epsilon$

Dekker's Method

Algorithm 4 Dekker(f, a, b, ϵ)

```
1:  Assert f(a)f(b) < 0;
2:  c = a;
3:  while True do
4:      if f(b)f(c) ≥ 0 then
5:          c = a;
6:      end if
7:      if |f(c)| < |f(b)| then
8:          a = b, b = c, c = a;
9:      end if
10:     m = (b + c)/2;
11:     if |m − b| ≤ ϵ(|b|) then
12:         Return b
13:     end if
14:     p = (b − a) ∗ f(b);
15:     if p ≥ 0 then
16:         q = f(a) − f(b);
17:     else
18:         q = f(b) − f(a);
19:         p = −p;
20:     end if
21:     a = b;
22:     if p ≤ ϵ(q) then
23:         b = b + sign(c − b) ∗ ϵ(b); // Minimal step
24:     else if p ≤ (m − b) ∗ q then
25:         b = b + p/q; // Secant
26:     else
27:         b = m; // Bisection
28:     end if
29: end while
30: Return x_{t+1}
```

$\epsilon(x)$: x has data type single or double, returns the positive distance from abs(x) to the next larger floating-point number of the same precision as x. $\epsilon(1.0) = 2^{-52} \approx 2.2204 \times 10^{-16}$.

Brent's method

An algorithm with Guaranteed Convergence for Finding a Zero of a Function, The Computer Journal 14.4, 1971, pp. 422-425. By R. P. Brent. (He also has a great book, "Algorithms for Minimization Without Derivatives, Prentice Hall, 1973.")

The most practical method! Matlab: fzero, scipy brenth,brentq. Idea:

- Try interpolating with a parabola (inverse quadratic interpolation)
- If that does not work, then use Secant Method
- Otherwise, use Bisect

It is more complicated than previous method we introduced.

Code: `https://github.com/scipy/scipy/blob/main/scipy/optimize/Zeros/brentq.c`

Ridder's

Ridders, C. "A new algorithm for computing a single root of a real continuous function." IEEE Transactions on circuits and systems 26.11 (1979): 979-980.

Idea: Given two values of the independent variable, x_0 and x_2, which are on two different sides of the root being sought, i.e., $f(x_0)f(x_2) < 0$, the method begins by evaluating the function at the midpoint $x_1 = (x_0 + x_2)/2$. One then finds the unique exponential function e^{ax} such that function $h(x) = f(x)e^{ax}$ satisfies $h(x_1) = (h(x_0) + h(x_2))/2$. Specifically, parameter a is determined by

$$e^{a(x_1 - x_0)} = \frac{f(x_1) - \text{sign}[f(x_0)]\sqrt{f(x_1)^2 - f(x_0)f(x_2)}}{f(x_2)}.$$

$$x_3 = x_1 + (x_1 - x_0)\frac{\text{sign}\left[f\left(x_0\right)\right] f\left(x_1\right)}{\sqrt{f\left(x_1\right)^2 - f\left(x_0\right) f\left(x_2\right)}}$$

Ridder's

The false position method is then applied to the points $(x_0, h(x_0))$ and $(x_2, h(x_2))$, leading to a new value x_3 between x_0 and x_2,

$$x_3 = x_1 + (x_1 - x_0)\frac{\text{sign}[f(x_0)]f(x_1)}{\sqrt{f(x_1)^2 - f(x_0)f(x_2)}}. \tag{10}$$

Comments from scipy.optimize: Ridders' method is faster than Bisect, but not generally as fast as Brent's method.

Application of root finding method

Let $f(x) : \mathbb{R}^d \to \mathbb{R}$ be differentiable convex. We want to minimize f over $\mathbb{R}^d$

$$\min_{\boldsymbol{x} \in \mathbb{R}^d} f(\boldsymbol{x})$$

Recall Steepest Descent

- **Step 0**: Given $\boldsymbol{x}_0$, set $t := 0$
- **Step 1**: $\boldsymbol{d}_t := -\nabla f(\boldsymbol{x}_t)$. If $\|\boldsymbol{d}_t\|_2 \leq \epsilon$, then stop.
- **Step 2**: Solve $\min_\lambda h(\lambda) := f(\boldsymbol{x}_t + \lambda \boldsymbol{d}_t)$ for the step-length λ_t, perhaps chosen by an exact or inexact line-search.
- **Step 3**: Set $\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \lambda_t \boldsymbol{d}_t$, $t = t + 1$; Go to **Step 1**

Application of root finding method

Consider

$$h(\lambda) := f(\boldsymbol{x}_t + \lambda \boldsymbol{d}_t),$$

where f is a convex function (i.e., $h(\lambda)$ is convex). Since $h(\lambda)$ is convex differentiable, the optimal λ^* can be found when $h'(\lambda) = 0$, that is

$$h'(\lambda) = \nabla f(\boldsymbol{x}_t + \lambda \boldsymbol{d}_t)^\top \boldsymbol{d}_t = 0.$$

That is, we need to find a root. We can apply any root-finding algorithm to resolve the step size problem of steepest descent method.

How to find effective range $[a, b]$?

Matlab fzero:
`https://www.mathworks.com/help/matlab/ref/fzero.html`

How do you find a suitable range of $[a, b]$?

Matlab's Idea
Start at a single given x. Take an interval centered at x with half-width $x/50$. Evaluate $f(x)$ at the interval endpoints. If the signs match, increase the interval width by a factor $\sqrt{2}$ and repeat until a sign change is detected.

Find $[a, b]$ use Change Detection

Algorithm 5 Signchange(f, x)

1: $a = x, b = x$
2: **if** $x! = 0$ **then**
3: $\quad dx = x/50$
4: **else**
5: $\quad dx = 1/50$
6: **end if**
7: **while** sign($f(a)$) == sign($f(b)$) **do**
8: $\quad dx = \sqrt{2}dx$
9: $\quad a = x - dx$
10: $\quad b = x + dx$
11: **end while**
12: Return a, b

Try this algorithm multiple times until you find one range.

How to check signs effectively?

To check $f(x_0)f(x_1) < 0$, there are two ways:

- if $f(x_0)f(x_1) < 0$
- if $\text{sign}(f(x_0)) != \text{sign}(f(x_1))$

Consider: $f(x) = np.exp(x) - 1.9151695967140057e^{-174}$ where we know that $f(-400) = 0$. Given $a = -450.0, b = -350.0$, if we use $f(x_0)f(x_1) < 0$, the algorithm will return 0.

scipy.optimize.brentq has this issue before v1.7.2(11/06/2021). Check more details at: `https://github.com/scipy/scipy/pull/14007` and `https://github.com/scipy/scipy/issues/13737`.

Open Source Codes

Python Implementations: C code for scipy.optimize

- scipy.optimize.brenth: Find a root of a function in a bracketing interval using Brent's method with hyperbolic extrapolation. This algorithm is proposed by Dekker in 1975.
- scipy.optimize.bisect: Find root of a function within an interval using bisection.
- scipy.optimize.ridder: Find a root of a function in an interval using Ridder's method.
- scipy.optimize.brentq: Find a root of a function in a bracketing interval using Brent's method with inverse quadratic interpolation.

To read the code, go to `https://github.com/scipy/scipy/tree/main/scipy/optimize/Zeros`.
There are four methods, including bisect.c, brenth.c, brentq.c, ridder.c.

Limits of Accuracy

Consider $f(x) = x^3 - 2x^2 + 4/3x - 8/27$ with $f(r = 2/3) = 0$. Consider using Bisect to solve $f(x) = 0$.

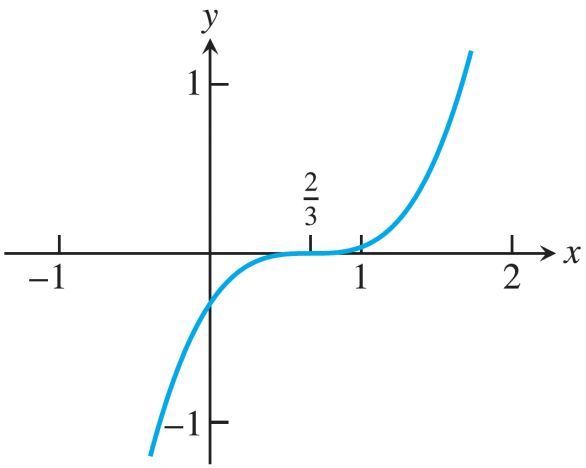
t	x_t	e_t	e_t/e_{t-1}
0	0.50000000000000000000	1.666666666666667e-1	-
2	0.50000000000000000000	1.666666666666667e-1	2.00000000
3	0.75000000000000000000	8.333333333333330e-2	0.25000000
7	0.67187500000000000000	5.208333333333300e-3	0.25000000
8	0.66406250000000000000	2.604166666666700e-3	1.00000000
9	0.66796875000000000000	1.302083333333300e-3	0.25000000
25	0.66667300462722800000	6.337960561300000e-6	1.00000000
53	0.66667306043228800000	6.393765621300000e-6	1.00000000
54	0.66667306043228800000	6.393765621300000e-6	1.00000000
55	0.66667306043228800000	6.393765621300000e-6	1.00000000

We cannot have six or more digits of precision! It indicates that Bisect loses some or all significant digits.

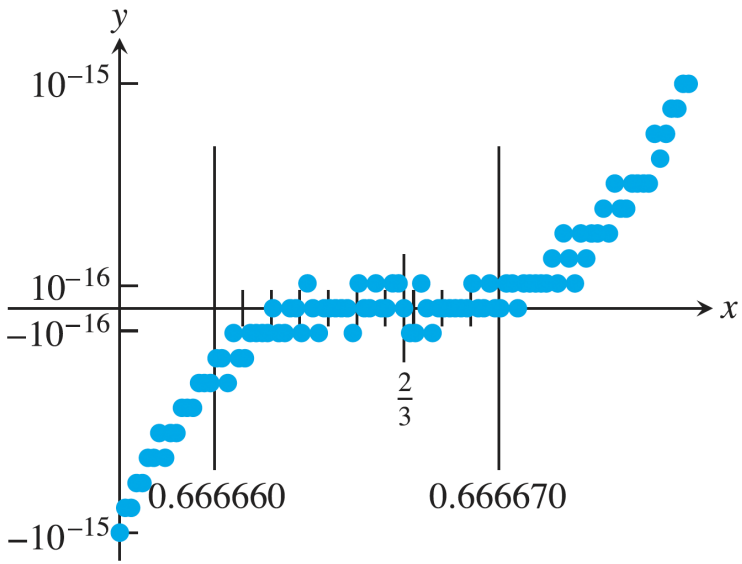
Limits of Accuracy

Two quantities we need to consider for checking stop condition

- $|f(x_t) - 0| = |f(x_t)|$: How fast does f value go to zero?
- $|x_t - r|$: How fast we can get solution r?



(a) $f(x)$;



(b) Zoom in of $f(x)$

- All other methods have the same issue!

Limits of Accuracy

Definition 3.1 (Backward error, Forward error)

Assume r is a root of f, i.e., $f(r) = 0$. Suppose x_t is an approximation to r. For the root-finding problem, we define two types of errors

- **Forward error**: $|r - x_t|$
- **Backward error**: $|f(x_t) - f(r)| = |f(x_t)|$

Problem: Equation $f(x) \rightarrow$ Equation solver $\rightarrow$ Solution

- Forward error comes from algorithm
- Backward error comes from f itself
- Example in previous slide: $|f(x_t)| \approx 2.2 \times 10^{-16}$ while the forward error is $\approx 10^{-5}$. In this case, backward error $\ll$ forward error!

Limits of Accuracy

Consider the multiple roots finding

$$f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27}$$
$$= \left(x - \frac{2}{3}\right)^3 = 0.$$

- $r = \frac{2}{3}$ is a multiple root with multiplicity $m = 3$
- Use Taylor's Theorem at point r

$$f(b) = f(a) + (b-a)f'(a) + \frac{(b-a)^2}{2!}f''(a) + \cdots$$
$$+ \frac{(b-a)^n}{n!}f^{(n)}(a) + \frac{(b-a)^{n+1}}{(n+1)!}f^{(n+1)}(\xi)$$

- Backward Error decreases quickly when forward error decreases!

Sensitivity Formula for Roots

Assume that the problem is to find a root r of $(x) = 0$, but that a small change $g(x)$ is made to the input, where it is small. Let Δr be the corresponding change in the root, so that

$$f(r + \Delta r) + \epsilon g(r + \Delta r) = 0. \tag{11}$$

Expanding f and g in degree-one Taylor polynomial implies that

$$f(r) + (\Delta r)f'(r) + \epsilon g(r) + \epsilon(\Delta r)g'(r) + \mathcal{O}((\Delta r)^2) = 0, \tag{12}$$

where "big $\mathcal{O}$" notation $\mathcal{O}((\Delta r)^2)$ to stand for terms involving $(\Delta r)^2$ and higher powers of Δr. It can be neglected if Δr is small.

$$(\Delta r)(f'(r) + \epsilon g'(r)) \approx -f(r) - \epsilon g(r) = -\epsilon g(r) \tag{13}$$

or

$$(\Delta r) \approx \frac{-\epsilon g(r)}{f'(r) + \epsilon g'(r)} \approx -\epsilon \frac{g(r)}{f'(r)}, \tag{14}$$

assuming that ϵ is small compared with $f'(r)$.

Sensitivity Formula for Roots

Assume that r is a root of $f(x)$ and $r + \Delta r$ is a root of $f(x) + \epsilon g(x)$. Then

$$\Delta r \approx -\frac{\epsilon g(r)}{f'(r)} \text{ if } \epsilon \ll f'(r). \tag{15}$$

Example: estimate the largest root of $P(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6) - 10^{-6}x^7$. Set $f(x) = (x-1)(x-2)(x-3)(x-4)(x-5)(x-6), \epsilon = -10^{-6}$ and $g(x) = x^7$. Without the $\epsilon g(x)$ term, the largest root is $r = 6$.

$$\Delta r \approx -\frac{\epsilon 6^7}{5!} = -2332.8\epsilon, \tag{16}$$

the error magnification factor is the relative forward error divided by the relative backward error.

$$\left|\frac{\Delta r/r}{\epsilon g(r)/g(r)}\right| = \frac{-\epsilon g(r)/(rf'(r))}{\epsilon} = \frac{|g(r)|}{|rf'(r)|} \tag{17}$$

Summary

Bisect method:

- Guarantee to converge
- Relative slow even when candidate solution is near the root

Fixed-Point-Iteration (FPI):

- Very general and easy to implement
- It may converge and diverge
- Sometime converge faster than Bisect

Newton's and its variants:

- Much faster (quadratically convergent)
- Need to have derivative information of f (fixed by Secant method)
- Slowed down when r is a multiple roots

Sensitivity analysis:

- Limits of Accuracy
- EMF