Roots: Open Methods

Berlin Chen
Department of Computer Science & Information Engineering
National Taiwan Normal University

Reference:

1. *Applied Numerical Methods with MATLAB for Engineers*, Chapter 6 & Teaching material

Chapter Objectives (1/2)

- Recognizing the difference between bracketing and open methods for root location
- Understanding the fixed-point iteration method and how you can evaluate its convergence characteristics
- Knowing how to solve a roots problem with the Newton-Raphson method and appreciating the concept of quadratic convergence

$$ax^2 + bx + c = 0 \quad \Rightarrow \quad x = \frac{-b \mp \sqrt{b^2 - 4ac}}{2a}$$

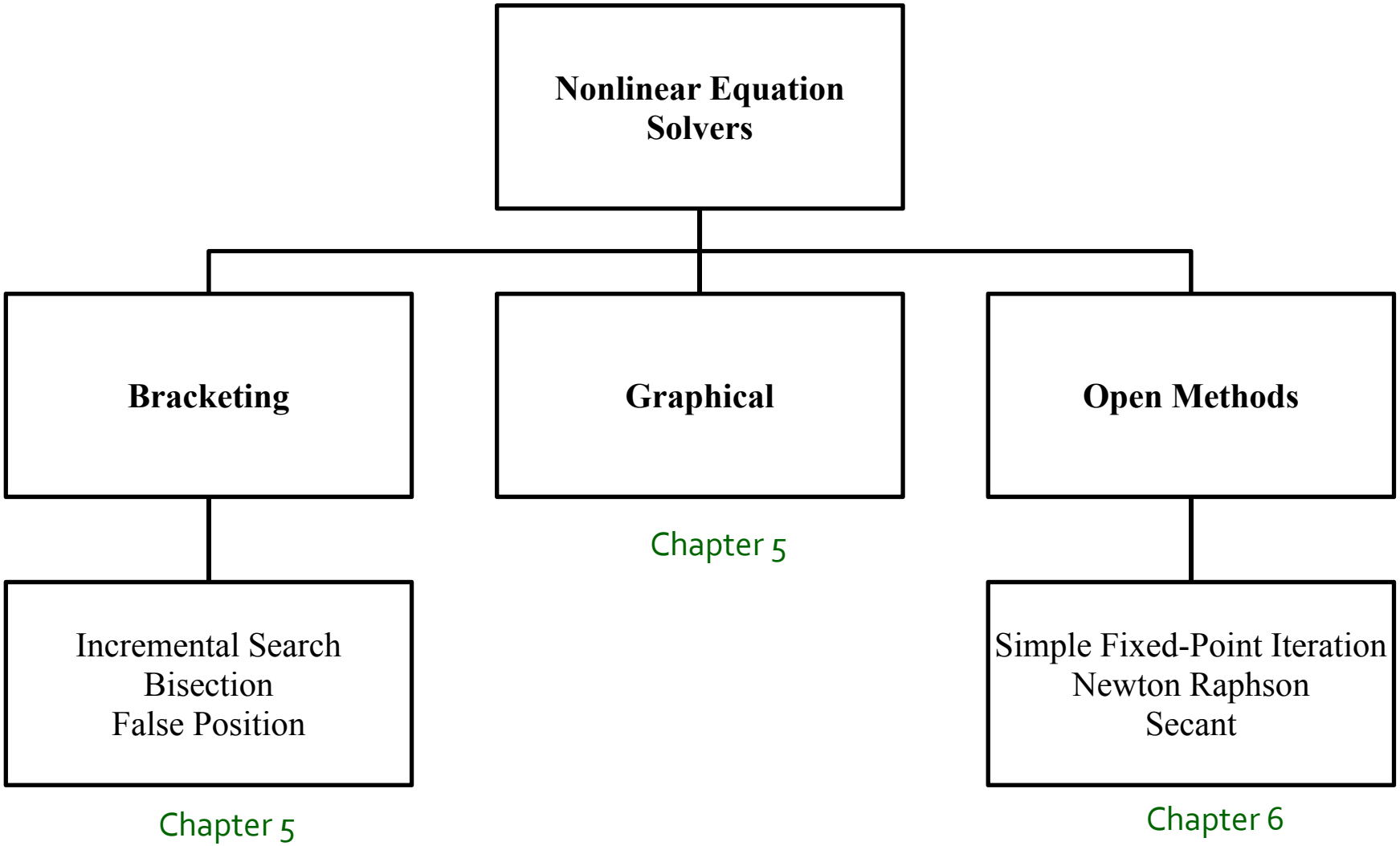
$$ax^5 + bx^4 + cx^3 + dx^2 + ex + f = 0 \quad \Rightarrow x = ?$$

$$\sin x + x = 0 \quad \Rightarrow x = ?$$

Chapter Objectives (2/2)

- Knowing how to implement both the secant and the modified secant methods
- Knowing how to use MATLAB's fzero function to estimate roots
- Learning how to manipulate and determine the roots of polynomials with MATLAB

Recall: Taxonomy of Root-finding Methods



– We can also employ a hybrid approach (**Bracketing + Open Methods**)

Open Methods

- ***Open methods*** differ from bracketing methods, in that open methods require only a single starting value or two starting values that do not necessarily bracket a root
- Open methods may diverge as the computation progresses, but when they do converge, they usually do so much faster than bracketing methods

Graphical Comparison of Root-finding Methods

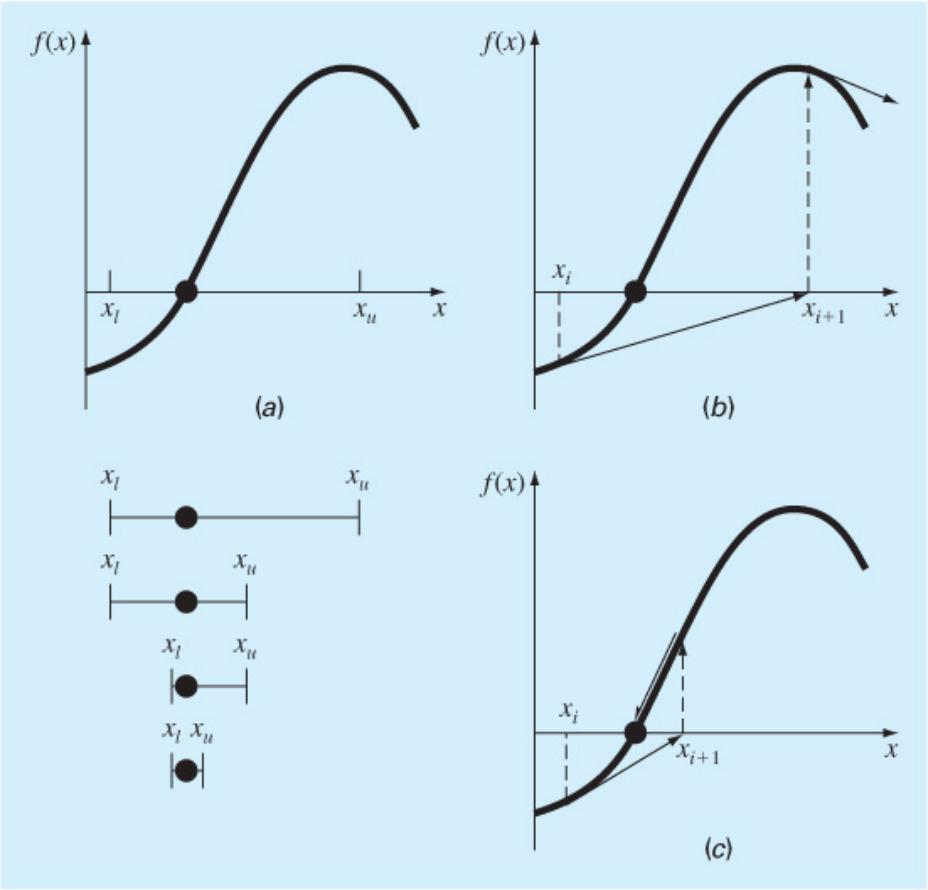


FIGURE 6.1
Graphical depiction of the fundamental difference between the (*a*) bracketing and (*b*) and (*c*) open methods for root location. In (*a*), which is bisection, the root is constrained within the interval prescribed by x_l and x_u. In contrast, for the open method depicted in (*b*) and (*c*), which is Newton-Raphson, a formula is used to project from x_i to x_{i+1} in an iterative fashion. Thus the method can either (*b*) diverge or (*c*) converge rapidly, depending on the shape of the function and the value of the initial guess.

Simple Fixed-Point Iteration

- Rearrange the function $f(x)=0$ so that x is on the left-hand side of the equation: $x=g(x)$
- Use the new function g to predict a new value of x - that is, $x_{i+1}=g(x_i)$
- The approximate error is given by:

$$\varepsilon_a = \left| \frac{x_{i+1} - x_i}{x_{i+1}} \right| \times 100\%$$

Simple Fixed-Point Iteration: An Example (1/2)

- Solve $f(x)=e^{-x}-x$
- Re-write as $x=g(x)$ by isolating x (example: $x=e^{-x}$)
- Start with an initial guess (here, 0)

i	x_i	$\|\varepsilon_a\|$ %	$\|\varepsilon_t\|$ %	$\|\varepsilon_t\|_i/\|\varepsilon_t\|_{i-1}$
0	0.0000		100.000	
1	1.0000	100.000	76.322	0.763
2	0.3679	171.828	35.135	0.460
3	0.6922	46.854	22.050	0.628
4	0.5005	38.309	11.755	0.533

The **true percent relative error** is roughly proportional (a factor of about 0.5 to 0.6) to the error from the previous iteration.

- Continue until some tolerance is reached

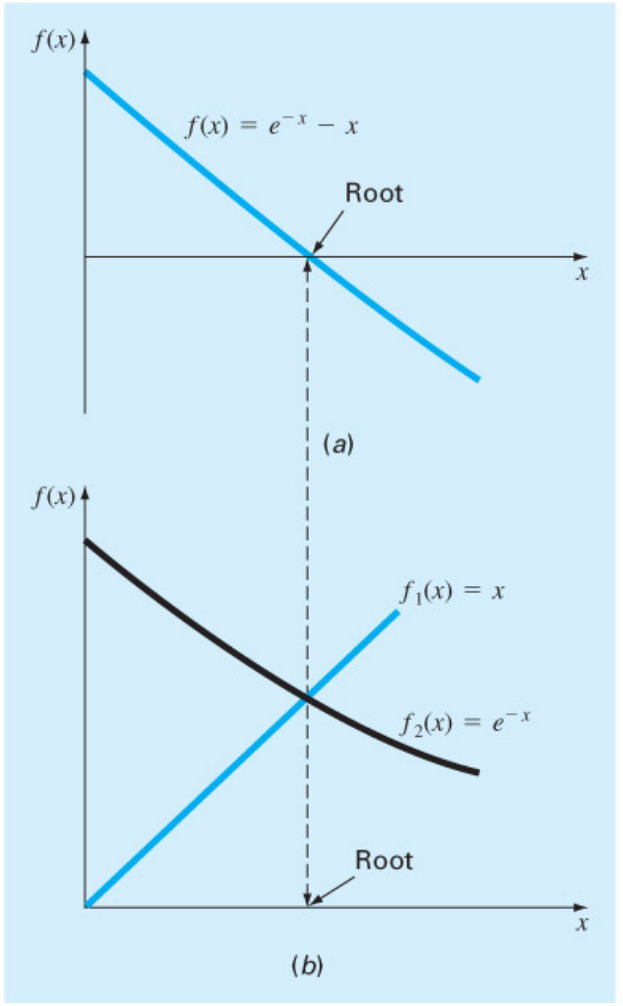
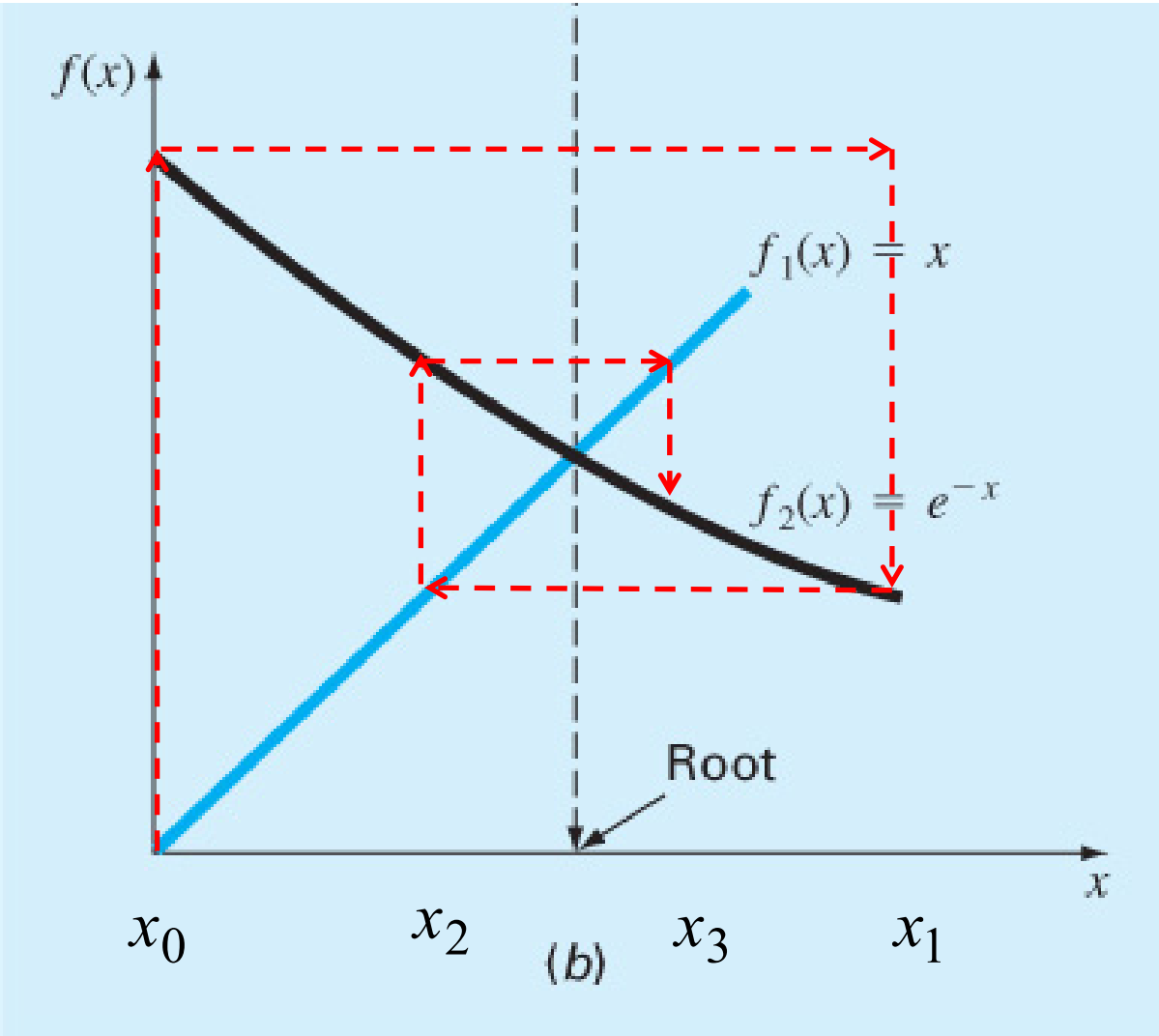


FIGURE 6.2
Two alternative graphical methods for determining the root of $f(x) = e^{-x} - x$. (a) Root at the point where it crosses the x axis; (b) root at the intersection of the component functions.

Simple Fixed-Point Iteration: An Example (2/2)



Convergence

- Convergence of the simple fixed-point iteration method requires that the derivative of g(x) near the root has a magnitude less than 1
  1) Convergent, $0 \le g' < 1$
  2) Convergent, $-1 < g' \le 0$
  3) Divergent, $g' > 1$
  4) Divergent, $g' < -1$

Chapra and Canale (2010) have shown that the error for any iteration is linearly proportional to the error from the previous iteration multiplied by the absolute value of the slope (derivative) of g(x):

$$E_{i+1} = g'(\xi) E_i$$

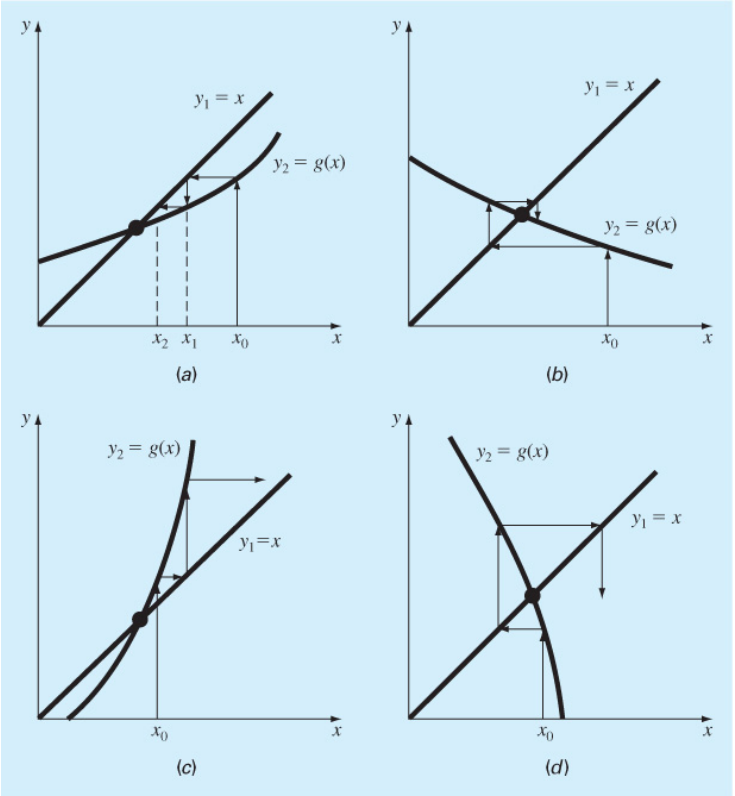


FIGURE 6.3
Graphical depiction of (*a*) and (*b*) convergence and (*c*) and (*d*) divergence of simple fixed-point iteration. Graphs (*a*) and (*c*) are called monotone patterns whereas (*b*) and (*c*) are called oscillating or spiral patterns. Note that convergence occurs when $|g'(x)| < 1$.

Newton-Raphson Method

- Based on forming the tangent line to the $f(x)$ curve at some guess x, then following the tangent line to a point where it crosses the x-axis
  - Such a point usually represents an improved estimate of the root

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

$$\Rightarrow x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

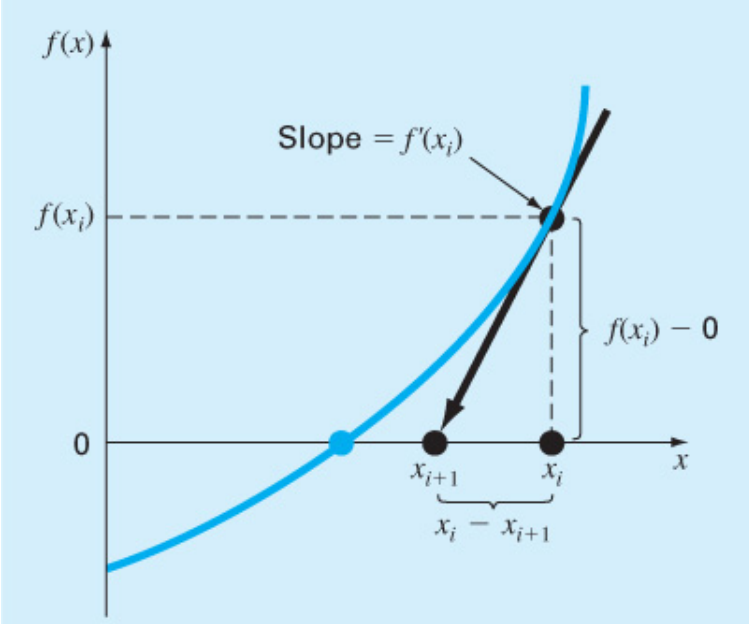


FIGURE 6.4
Graphical depiction of the Newton-Raphson method. A tangent to the function of x_i [that is, $f'(x)$] is extrapolated down to the x axis to provide an estimate of the root at x_{i+1}.

Newton-Raphson Method: Pros and Cons

- **Pro:** The error of the i+1th iteration is roughly proportional to the square of the error of the ith iteration - this is called ***quadratic convergence***
- **Con:** Some functions show slow or poor convergence

Chapra and Canale (2010) have shown that the error is roughly proportional to the square of the previous error:

$$E_{t,i+1} = \frac{-f''(x_i)}{2f'(x_i)} E_{t,i}^2$$

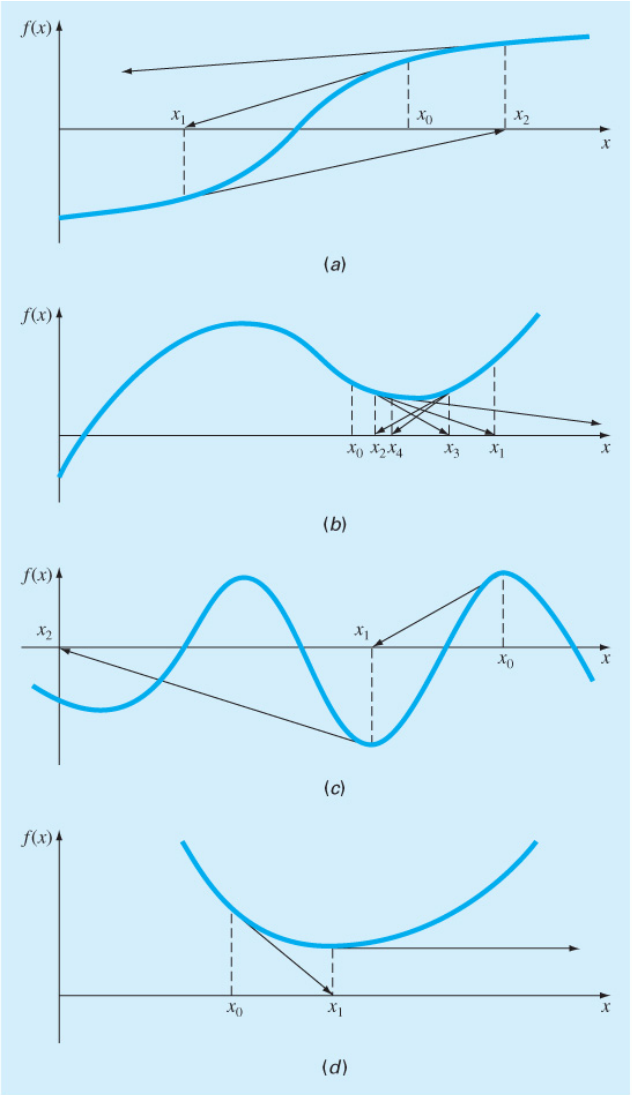


FIGURE 6.6
Four cases where the Newton-Raphson method exhibits poor convergence.

Secant Methods (1/2)

- A potential problem in implementing the Newton-Raphson method is the evaluation of the derivative - there are certain functions whose derivatives may be difficult or inconvenient to evaluate
- For these cases, the derivative can be approximated by a backward finite divided difference:

$$f'(x_i) \cong \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}$$

Secant Methods (2/3)

- Substitution of this approximation for the derivative to the Newton-Raphson method equation gives:

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1} - x_i)}{f(x_{i-1}) - f(x_i)}$$

- Note - this method requires ***two*** initial estimates of *x* but does ***not*** require an analytical expression of the derivative

Secant Methods (3/3)

- Modified Secant Method
  - Rather than using two arbitrary values to estimate the derivate, an alternative approach involves **a fractional perturbation** of the independent variable to estimate *f'*(*x*)

$$f'(x_i) \cong \frac{f(x_i + \delta x_i) - f(x_i + \delta x_i)}{\delta x_i}$$

$$\Rightarrow x_{i+1} = x_i - \frac{\delta x_i f(x_i)}{f(x_i + \delta x_i) - f(x_i + \delta x_i)}$$

Brent's Root-location Method

- A hybrid approach that combines the reliability of bracketing with the speed of open methods
  - Try to apply a speedy open method whenever possible, but revert to a reliable bracketing method if necessary
    - That is, in the event that the open method generate an unacceptable result (i.e., an estimate falling outside the bracket), the algorithm reverts to the more conservative bisection method
  - Developed by Richard Brent (1973)
- Here the bracketing technique being used is ***the bisection method***, whereas two open methods, namely, ***the secant method*** and ***inverse quadratic interpolation***, are employed
  - Bisection typically dominates at first but as root is approached, the technique shifts to the fast open methods

Inverse Quadratic Interpolation (1/4)

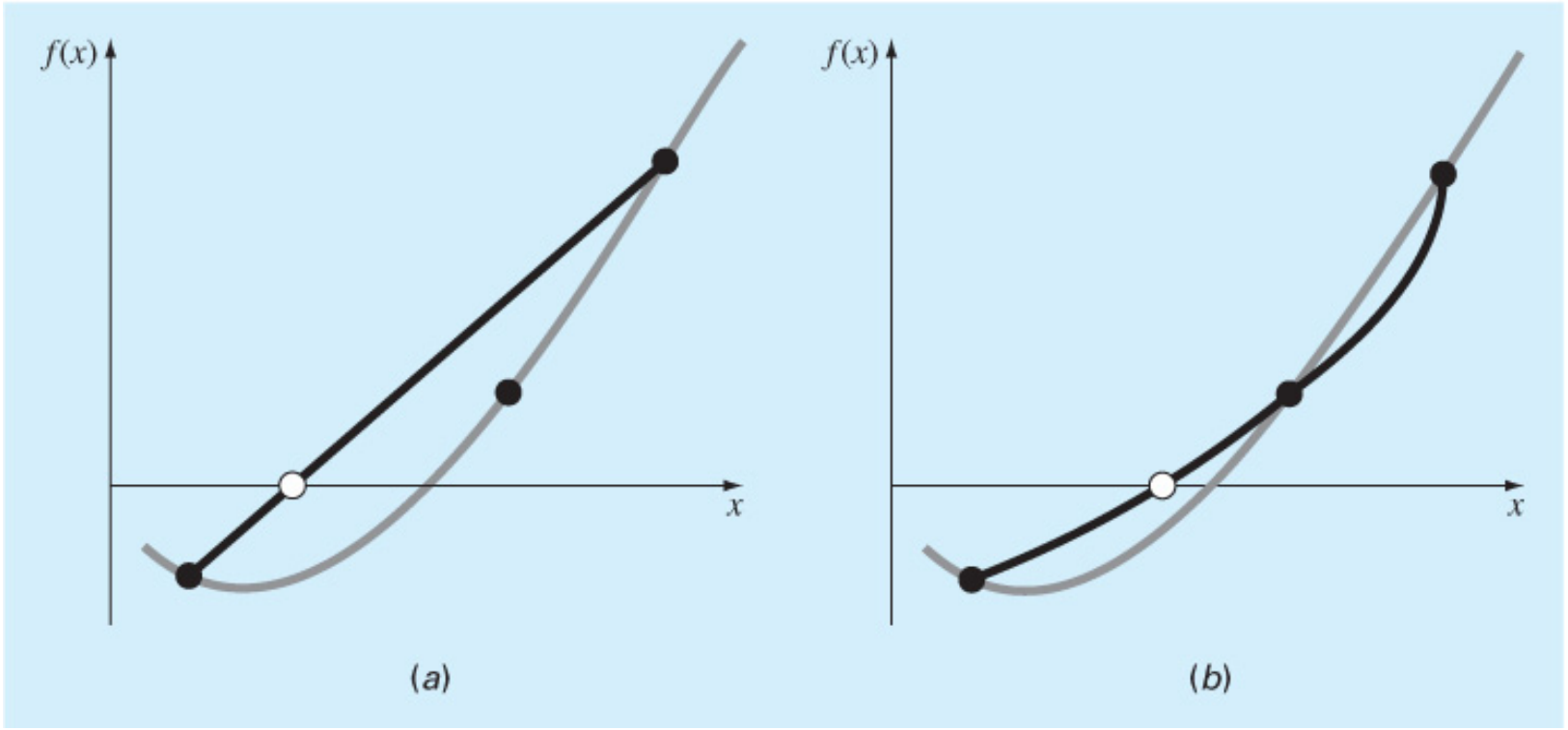
- Inverse quadratic interpolation is similar in spirit to the secant method
  - **The secant method:** compute a straight line that goes through two guesses and take the intersection of the straight line with the x axis as the new root estimate
  - **Inverse quadratic interpolation**: compute parabola (quadratic curve), a function of x, that goes through three points and take the intersection of the parabola with the x axis as the new root estimate
    - However, it is possible that the parabola might not intersect the x axis
    - Inverse quadratic interpolation rectifies the difficulty by fitting the points with a parabola in y (a function of y)

$$g(y)=\frac{(y-y_{i-1})(y-y_i)}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2}+\frac{(y-y_{i-2})(y-y_i)}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1}+\frac{(y-y_{i-2})(y-y_{i-1})}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i$$

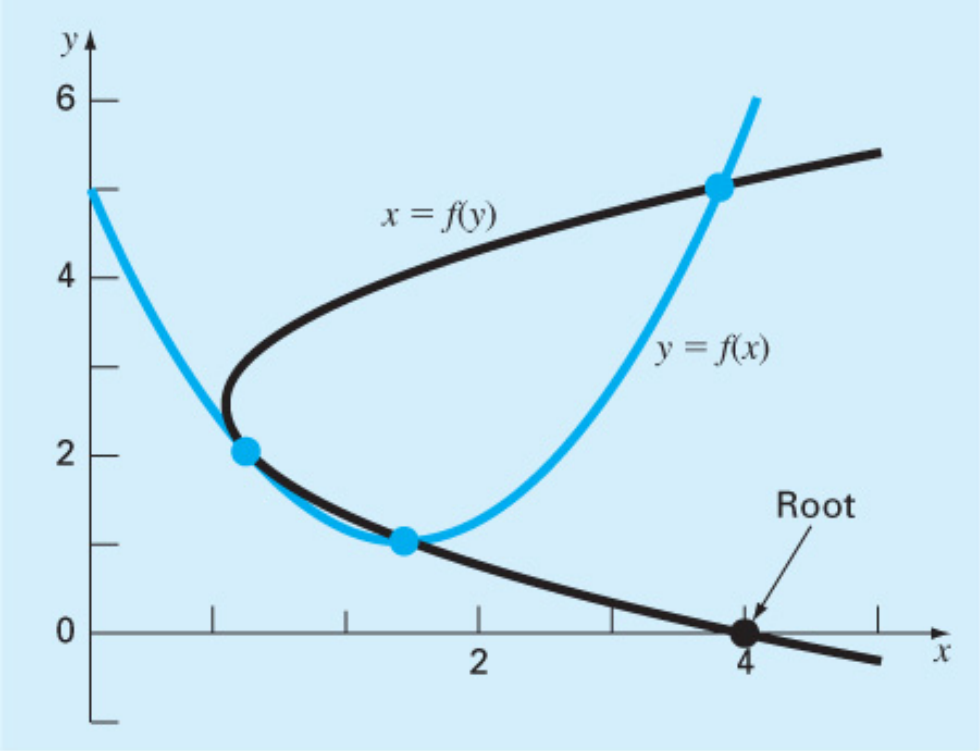
This form is also called a **Lagrange polynomial**.

Inverse Quadratic Interpolation (2/4)

FIGURE 6.8
Comparison of (*a*) the secant method and (*b*) inverse quadratic interpolation. Note that the approach in (*b*) is called "inverse" because the quadratic function is written in y rather than in x.



Inverse Quadratic Interpolation (3/4)



The inverse quadratic interpolation x=f(y) always intersect the x axis.

FIGURE 6.9
Two parabolas fit to three points. The parabola written as a function of x, $y = f(x)$, has complex roots and hence does not intersect the x axis. In contrast, if the variables are reversed, and the parabola developed as $x = f(y)$, the function does intersect the x axis.

Inverse Quadratic Interpolation (4/4)

- The new root estimate, x_{i+1}, therefore corresponds to y=0
  -Substituted into the equation shown above, we can have

$$x_{i+1} = \frac{y_{i-1}y_i}{(y_{i-2}-y_{i-1})(y_{i-2}-y_i)}x_{i-2} + \frac{y_{i-2}y_i}{(y_{i-1}-y_{i-2})(y_{i-1}-y_i)}x_{i-1} + \frac{y_{i-2}y_{i-1}}{(y_i-y_{i-2})(y_i-y_{i-1})}x_i$$

An Example Function for the Brent's Method

```
function b = fzerosimp(x1,xu)
a = xl; b = xu; fa = f(a); fb = f(b);
c = a; fc = fa; d = b - c; e = d;
while (1)
  if fb == 0, break, end
  if sign(fa) == sign(fb) %If needed, rearrange points
    a = c; fa = fc; d = b - c; e = d;
  end
  if abs(fa) < abs(fb)
    c = b; b = a; a = c;
    fc = fb; fb = fa; fa = fc;
  end
  m = 0.5*(a - b); %Termination test and possible exit
  tol = 2 * eps * max(abs(b), 1);
  if abs(m) <= tol | fb == 0.
    break
  end
  %Choose open methods or bisection
  if abs(e) >= tol & abs(fc) > abs(fb)
    s = fb/fc;
    if a == c                    %Secant method
      p = 2*m*s;
      q = 1 - s;
    else              %Inverse quadratic interpolation
      q = fc/fa; r = fb/fa;
      p = s * (2*m*q * (q - r) - (b - c)*(r - 1));
      q = (q - 1)*(r - 1)*(s - 1);
    end
    if p > 0, q = -q; else p = -p; end;
    if 2*p < 3*m*q - abs(tol*q) & p < abs(0.5*e*q)
      e = d; d = p/q;
    else
      d = m; e = m;
    end
  else                                 %Bisection
    d = m; e = m;
  end
  c = b; fc = fb;
  if abs(d) > tol, b=b+d; else b=b-sign(b-a)*tol; end
  fb = f(b);
end
```

FIGURE 6.10
Function for Brent's root-finding algorithm based on a MATLAB M-file developed by Cleve Moler (2005).

MATLAB's `fzero` Function

- MATLAB's `fzero` provides the best qualities of both bracketing methods and open methods.
  - Using an initial guess:

    `x = fzero(function, x0)`

    `[x, fx] = fzero(function, x0)`
    - *`function`* is a function handle to the function being evaluated
    - *`x0`* is the initial guess
    - *`x`* is the location of the root
    - *`fx`* is the function evaluated at that root
  - Using an initial bracket:

    `x = fzero(function, [x0 x1])`

    `[x, fx] = fzero(function, [x0 x1])`
    - As above, except *`x0`* and *`x1`* are guesses that *must* bracket a sign change

`fzero` Options

- Options may be passed to fzero as a third input argument - the options are a data structure created by the `optimset` command
- *options* = optimset('par_1', val_1, 'par_2', val_2,…)
  - par_n is the name of the parameter to be set
  - val_n is the value to which to set that parameter
  - The parameters commonly used with `fzero` are:
    - `display`: when set to 'iter' displays a detailed record of all the iterations
    - `tolx`: A positive scalar that sets a termination tolerance on x

`fzero` Example

- `options = optimset('display', 'iter');`
  - Sets options to display each iteration of root finding process
- `[x, fx] = fzero(@(x) x^10-1, 0.5, options)`
  - Uses fzero to find roots of $f(x)=x^{10}-1$ starting with an initial guess of $x=0.5$
- MATLAB reports `x=1, fx=0` after 35 function counts

Polynomials (1/2)

- MATLAB has a built in program called `roots` to determine all the roots of a polynomial - including imaginary and complex ones.
- `x = roots(c)`
  - `x` is a column vector containing the roots
  - `c` is a row vector containing the polynomial coefficients
- Example:
  - Find the roots of

$$f(x)=x^5-3.5x^4+2.75x^3+2.125x^2-3.875x+1.25$$

  - `x = roots([1 -3.5 2.75 2.125 -3.875 1.25])`

Polynomials (2/2)

- MATLAB's `poly` function can be used to determine polynomial coefficients if roots are given:
  - `b = poly([0.5 -1])`
    - Finds $f(x)$ where $f(x)$ =0 for x=0.5 and x=-1
    - MATLAB reports `b = [1.000 0.5000 -0.5000]`
    - This corresponds to $f(x)=x^2+0.5x-0.5$
- MATLAB's `polyval` function can evaluate a polynomial at one or more points:
  - `a = [1 -3.5 2.75 2.125 -3.875 1.25];`
    - If used as coefficients of a polynomial, this corresponds to $f(x)=x^5-3.5x^4+2.75x^3+2.125x^2-3.875x+1.25$
  - `polyval(a, 1)`
    - This calculates $f(1)$, which MATLAB reports as -0.2500