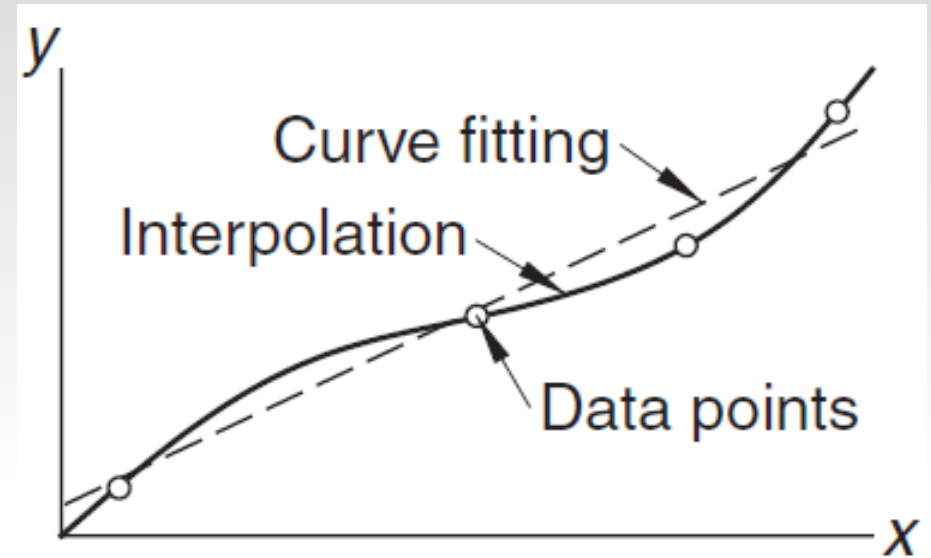


Review: Interpolation and Curve Fitting

Interpolation: Given a set of data points, construct a (simple) function going through all of them

Curve fitting: Given a set of data points with errors, find the best fitting line.

Both allow evaluation of the function at points between the data pts. and the new function is a simpler one, so easier to integrate/evaluate.



x_0	x_1	x_2	\dots	x_n
y_0	y_1	y_2	\dots	y_n

We discussed two interpolation techniques:

- polynomial interpolation
 - splines (local low-order polynomials + continuity)
- and Least Square Fitting for curve fitting.

Root finding

Most algebraic equations, $f(x)=0$, have no exact analytical solution, so must be solved numerically.

We will discuss four ways to solve such equations numerically:

- Bisection method
- Secant method
- Ridder's method
- Newton-Raphson method

All of these methods start with an initial guess and proceed via iterative refinement of that guess. All have advantages and disadvantages – no universal approach exists!

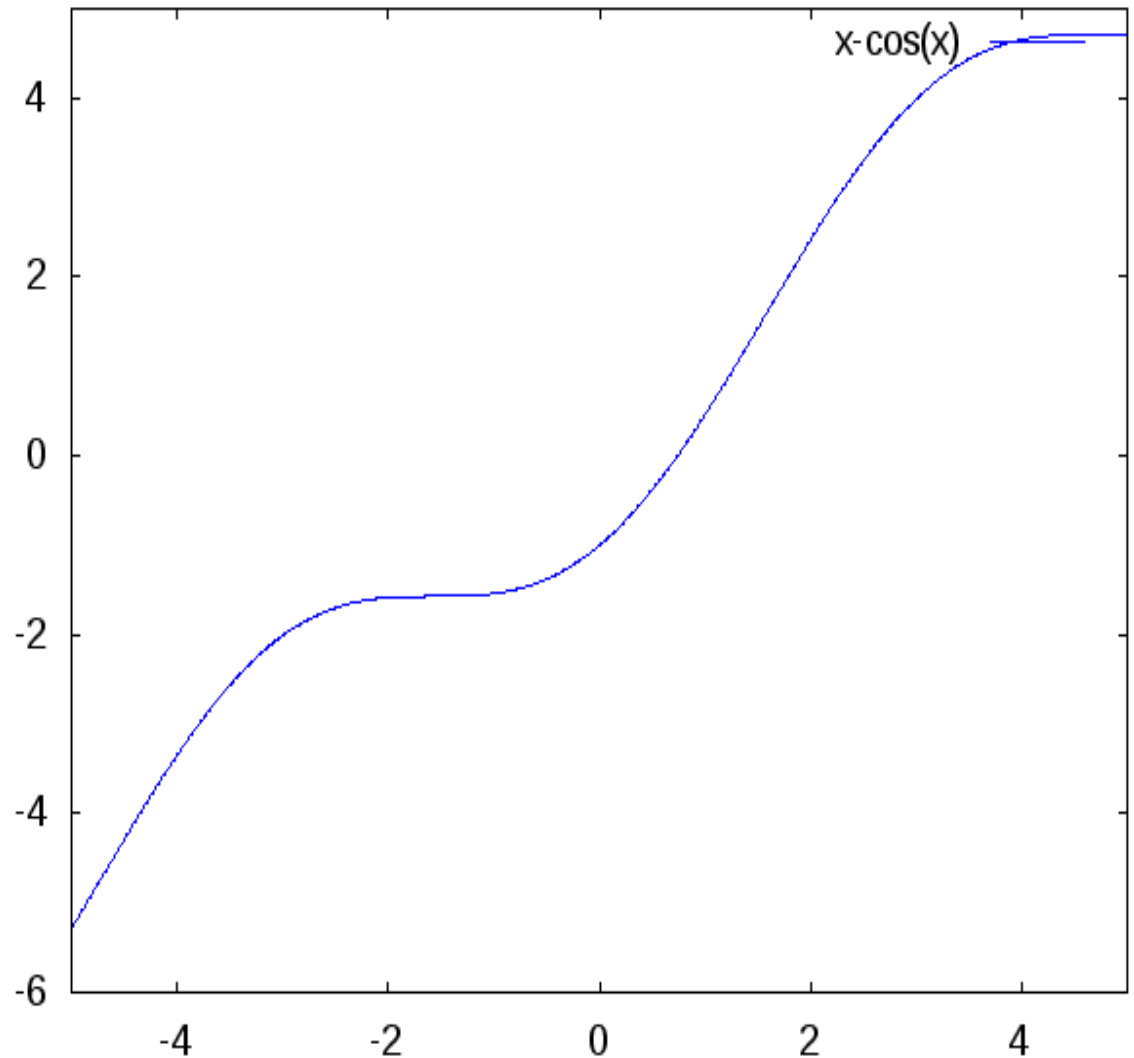
Picking the initial guess

Problem: Find a good guess for the the solution of $f(x)=0$, e.g.

$$x - \cos(x) = 0.$$

Method I: Use equation's **properties** – e.g. it is easy to see that the above equation has a root between 0 and 1 (Why?)

Method II: **plot** the function over some interval of interest – gives you a rough idea about the root(s): how many and where?



Solution method 1: bisection

One of the simplest methods to implement.

Works for **continuous** functions $f(x)$ with a single root in a given interval $[a,b]$. In that case it is easy to see that $f(a)*f(b)<0$ (why is that?).

Let's **halve the interval**, making two: $[a,(a+b)/2]$ and $[(a+b)/2,b]$ – for one of those it remains true that $f(x)$ has opposite signs at ends.

Repeat procedure **iteratively** with the new interval.

Drawback: uses no knowledge of the function properties, so convergence is relatively **slow**.

Bisection method: properties

- Method needs 39 iterations to solve example equation for $\text{tol}=\epsilon=1\text{e-}12$ (root is 0.739085133216).
- Works for $f(a)*f(b)<0$ and f continuous on interval $[a,b]$
- Relatively slow convergence: interval always halved \rightarrow need $\log_2((b-a)/\epsilon)-1$ iterations for initial interval of length $(b-a)$ and tolerance ϵ , since the solution error is
$$|x^{(k)}-a|<(1/2)^{k+1}(b-a)$$

Secant Method

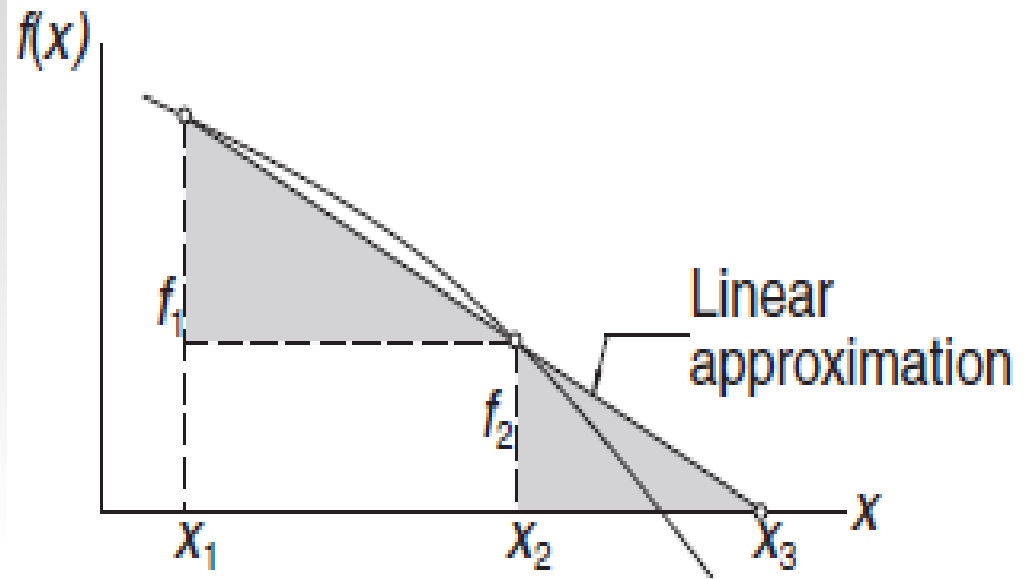
Start with **two estimates** for the root, x_1 and x_2 (not necessarily bracketing it), then, from the similar triangles we have:

$$\frac{f_2}{x_3 - x_2} = \frac{f_1 - f_2}{x_2 - x_1}$$

Or, solving for x_3 : $x_3 = x_2 - f_2 \frac{x_2 - x_1}{f_2 - f_1}$

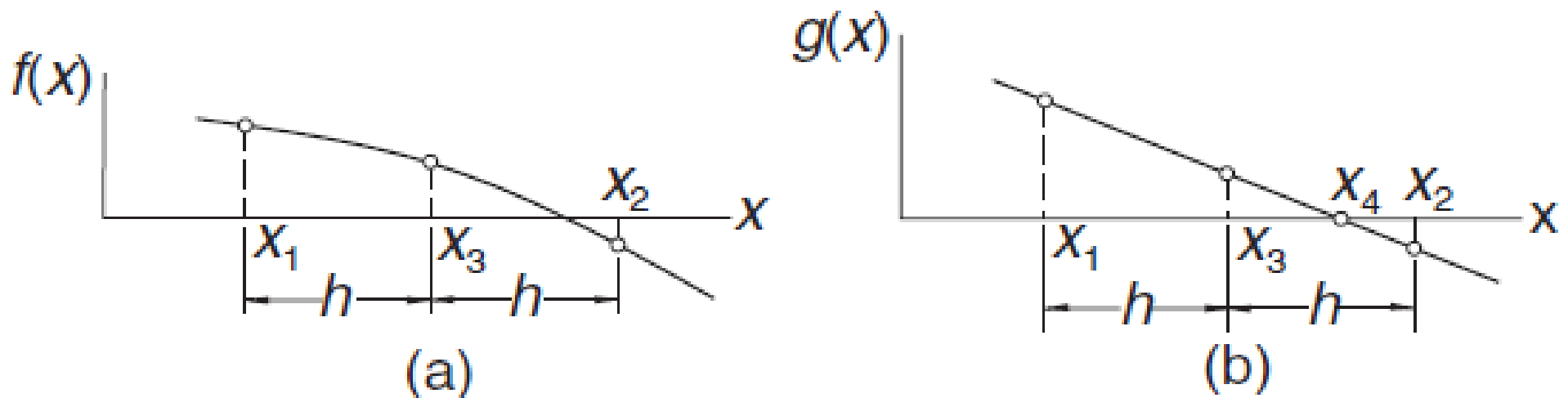
We then replace oldest value $x_1 \leftarrow x_2, x_2 \leftarrow x_3$ and **repeat** the process until convergence:

$$x_{n+1} = x_n - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$$



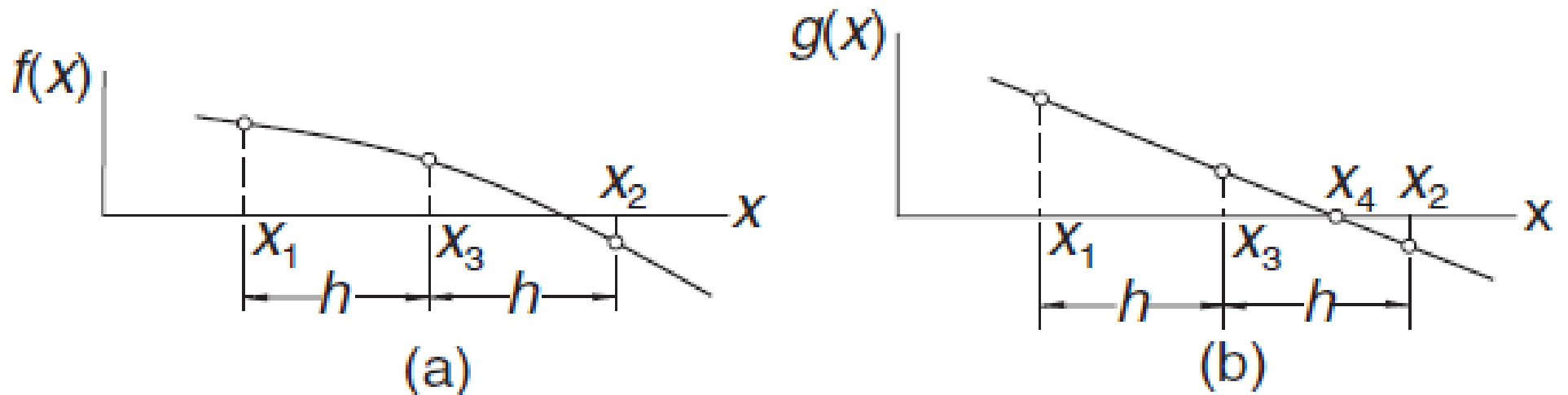
This can be shown to exhibit **super-linear convergence**, ~60% more correct digits per iteration than bisection.

Ridder's method



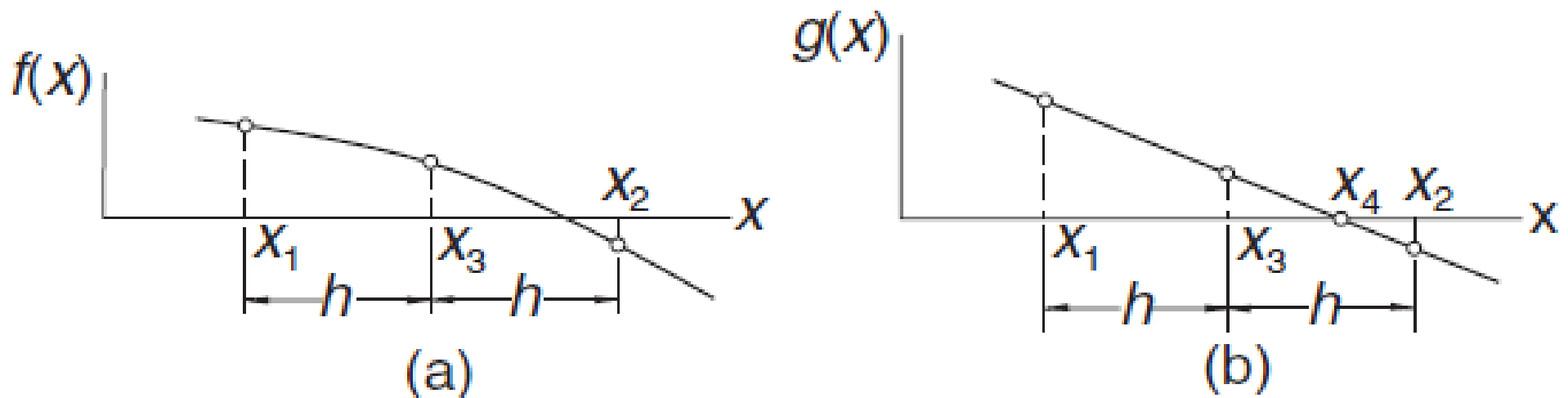
- Another method based on linear interpolation.
- Faster than the secant method (**quadratic** convergence).
- Has nice property that a **bracketed root remains bracketed** by subsequent approximations, making this method very reliable.

Ridder's method



- Let's assume the root is **bracketed** by (x_1, x_2)
- Introduce the function $g(x) = f(x)e^{(x-x_1)Q}$ where Q is such that $(x_1, g_1), (x_2, g_2)$ and (x_3, g_3) lie on a straight line.
- Improved value of the root is obtained by **linear interpolation of $g(x)$** , instead of $f(x)$ as before, as follows:
- We have: $g_1 = f_1$ $g_2 = f_2 e^{2hQ}$ $g_3 = f_3 e^{hQ}$ $h = (x_2 - x_1)/2$
- Straight line means $\rightarrow g_3 = (g_1 + g_2)/2$ or $f_3 e^{hQ} = \frac{1}{2}(f_1 + f_2 e^{2hQ})$

Ridder's method



- The last equation is quadratic one for the $\exp(hQ)$:

$$f_3 e^{hQ} = \frac{1}{2}(f_1 + f_2 e^{2hQ}) \longrightarrow e^{hQ} = \frac{f_3 \pm \sqrt{f_3^2 - f_1 f_2}}{f_2}$$

- Then, linear interpolation on (x_1, g_1) and (x_3, g_3) yields the improved root:

$$x_4 = x_3 - g_3 \frac{x_3 - x_1}{g_3 - g_1} = x_3 - f_3 e^{hQ} \frac{x_3 - x_1}{f_3 e^{hQ} - f_1}$$

- Finally, we substitute the exponent from above:

+: for $f_1 - f_2 > 0$

- : for $f_1 - f_2 < 0$

$$x_4 = x_3 \pm (x_3 - x_1) \frac{f_3}{\sqrt{f_3^2 - f_1 f_2}}$$

$$\text{VOLUME}(R) = (4 / \text{INT}(\text{PI})) * \text{PI} * R^{\text{INT}(\text{PI})}$$

PROGRAMMING TIP: THE NUMBER "3" IS CURSED. AVOID IT.

The real programming tip: keep your code as simple and straightforward as possible!

Over-complicating things is error-prone and typically means trouble.

Newton-Raphson method

Based on Taylor expansion:

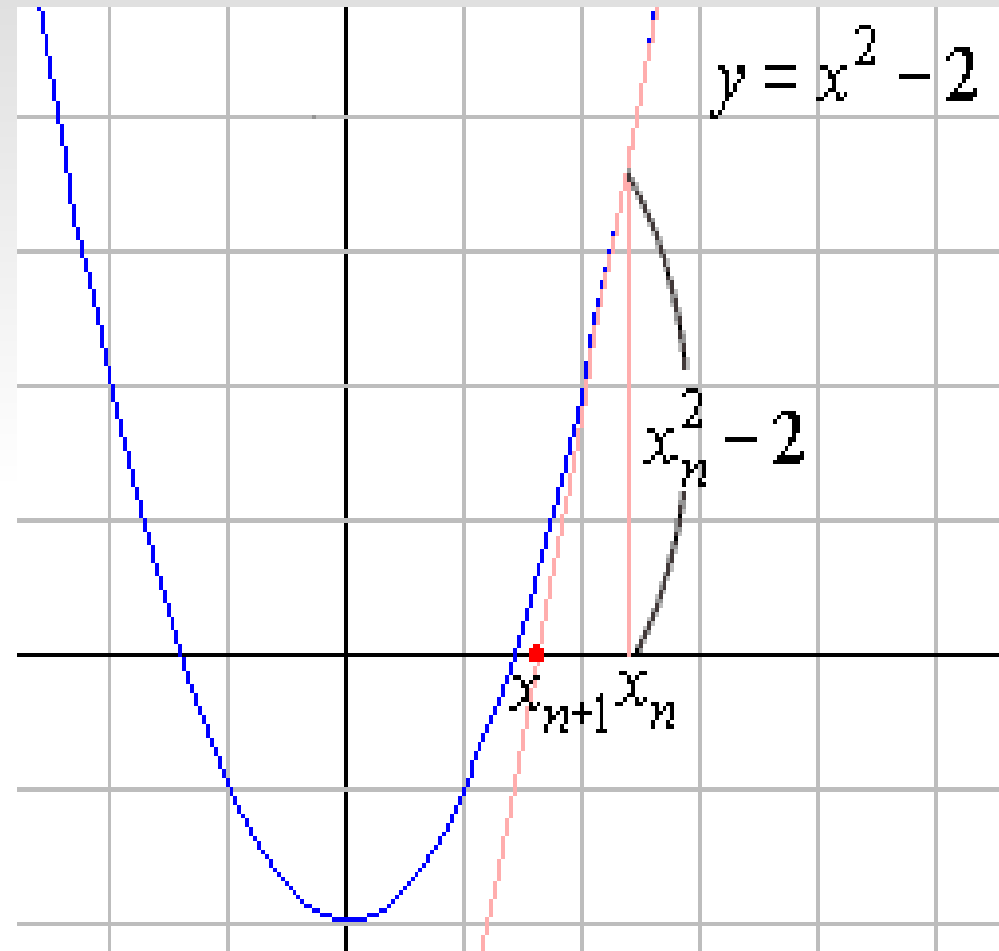
$$f(x) = f(x_0) + (x-x_0) f'(x_0) + \dots$$

current point: x_0 , we want x such that $f(x) = 0$. Truncating expansion to 2 terms and setting $f(x)=0$, we get

$$0 = f(x_0) + (x-x_0) f'(x_0),$$

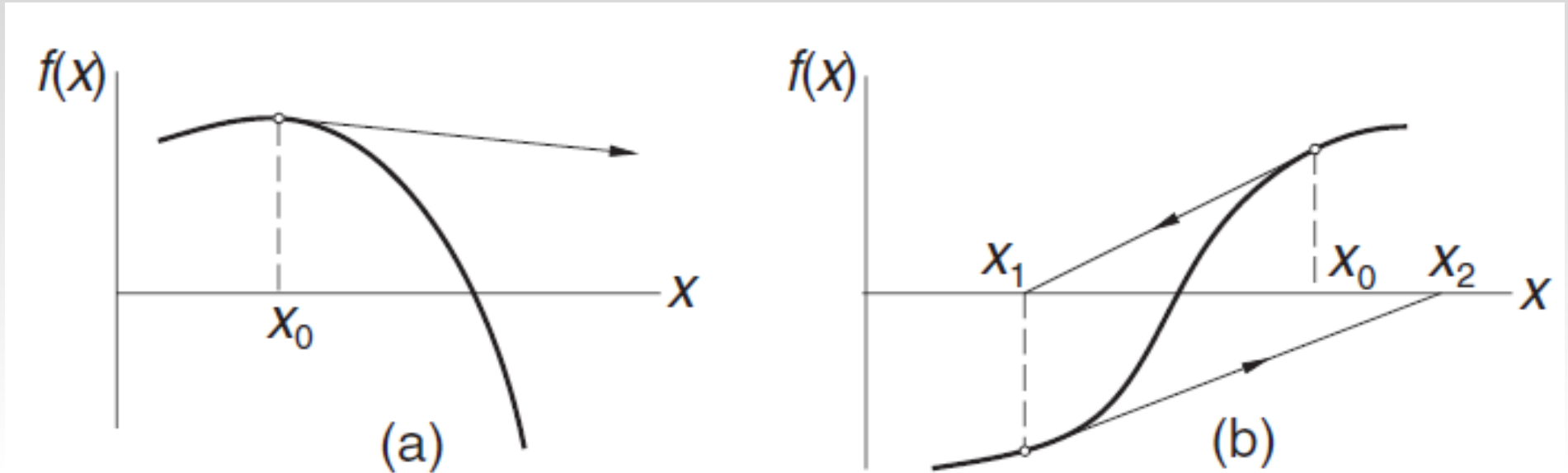
solving for x :

$x_1 = x_0 - f(x_0)/f'(x_0) \rightarrow$ next approximation to x , etc.



$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton-Raphson convergence



- The Newton-Raphson method converges to a root very **fast**, but only if the initial guess is **reasonably good**.
- Global (i.e. away from the root) convergence is **poor**.
- Reason for that is illustrated above: while locally the tangent is a good approximation to how the function behaves, further away that might not be the case.
- Essentially, the Taylor expansion is good approximation only for $(x-x_0) \ll 1$.

Newton-Raphson method: simple usage example

```
def f(x): return x**4 - 6.4*x**3 + 6.45*x**2 + 20.538*x - 31.752
def df(x): return 4.0*x**3 - 19.2*x**2 + 12.9*x + 20.538
```

← Define function
and its derivative

```
def newtonRaphson(x,tol=1.0e-9):
    for i in range(30):
        dx = -f(x)/df(x)
        x = x + dx
        if abs(dx) < tol:
            return x,i
    print('Too many iterations\n')
```

← Simple Newton-Raphson
implementation

```
root,numIter = newtonRaphson(2.0)
print 'Root =',root
print 'Number of iterations =',numIter
raw_input("Press return to exit")
```

← Usage

```
Root = 2.09999997862
Number of iterations = 22
Press return to exit
```

← Results

Newton-Raphson: systems of equations

- We want to solve the (nonlinear) system:

$$f_1(x_1, x_2, \dots, x_n) = 0$$

$$f_2(x_1, x_2, \dots, x_n) = 0$$

$$\vdots$$

$$f_n(x_1, x_2, \dots, x_n) = 0$$

- Similarly to the single equation case we use Taylor expansion and drop high-order terms:

$$f_i(\mathbf{x} + \Delta\mathbf{x}) = f_i(\mathbf{x}) + \sum_{j=1}^n \frac{\partial f_i}{\partial x_j} \Delta x_j + \cancel{O(\Delta x^2)}$$

- This can be written as:

$$\mathbf{f}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \Delta\mathbf{x} \quad J_{ij} = \frac{\partial f_i}{\partial x_j} \leftarrow \begin{array}{l} \text{Jacobian} \\ \text{matrix} \end{array}$$

Newton-Raphson: systems of equations

- Let \mathbf{x} is the current approximation to the solution of $\mathbf{f}(\mathbf{x})=\mathbf{0}$ and $\mathbf{x}+\Delta\mathbf{x}$ is the improved one. To find $\Delta\mathbf{x}$ we set $\mathbf{f}(\mathbf{x}+\Delta\mathbf{x})=\mathbf{0}$.
- The result is a system of linear equations:
$$\mathbf{J}(\mathbf{x}) \Delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$$
- The steps for solving system of equations are then:
 - Estimate solution \mathbf{x} and evaluate $\mathbf{f}(\mathbf{x})$.
 - Compute the Jacobian matrix $\mathbf{J}(\mathbf{x})$.
 - Set up the linear system above and solve for $\Delta\mathbf{x}$.
 - Let $\mathbf{x}+\Delta\mathbf{x} \rightarrow \mathbf{x}$ and repeat steps until convergence.

Newton-Raphson method: features

- Function $F(x)$ has to be **differentiable**.
- Converges fast: Newtons method exhibits **quadratic** or **second order** convergence, roughly doubling the number of correct digits in every iteration. But **it does not always converge!**
- A good initial guess is quite crucial, if it is not a good one the method **might not converge** at all!
- Of the four methods discussed here, this is the only one which can be generalized for **systems of equations** (as opposed to a single equation) – we needed to write the method in vector/matrix form for that.

Newton-Raphson method: useful applications

Common, if somewhat surprising use for Newtons method - fast computing of e.g. $1/c$ and $\sqrt[n]{c}$ (why would we want to do that?), as follows: solve $f(x)=x^2-c=0$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = \frac{1}{2} \left(x_n + \frac{c}{x_n} \right)$$

convergent for all $x_0 > 0$. Example:

```
def f(x): return x**2 - 5  
def df(x): return 2.*x
```

Results:

Root = 2.2360679775

Number of iterations = 3

Press return to exit

← Square root of 5

Exercise: do the same for $1/c$ and n^{th} root of c .

Some final remarks

Newtons method can be used for systems of equations (requires matrix inversion, which replaces the division).

We should always try to make software robust (do tests, check for correct input/results, guaranteed to stop, ...)

We could (and generally should) combine several methods to be more robust (e.g. Newton-Raphson [fast] & bisection [certain]).

Some useful Python internal functions for root finding:

- `scipy.optimize.fsolve` : general routine for solving nonlinear equation (or system of equations)
- `numpy.roots` : zeros of polynomials

(try them out on your own). Usage examples provided in Jupyter notebook.