Root finding

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

Today we will discuss four ways to solve such equations numerically:

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

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

Example test problem

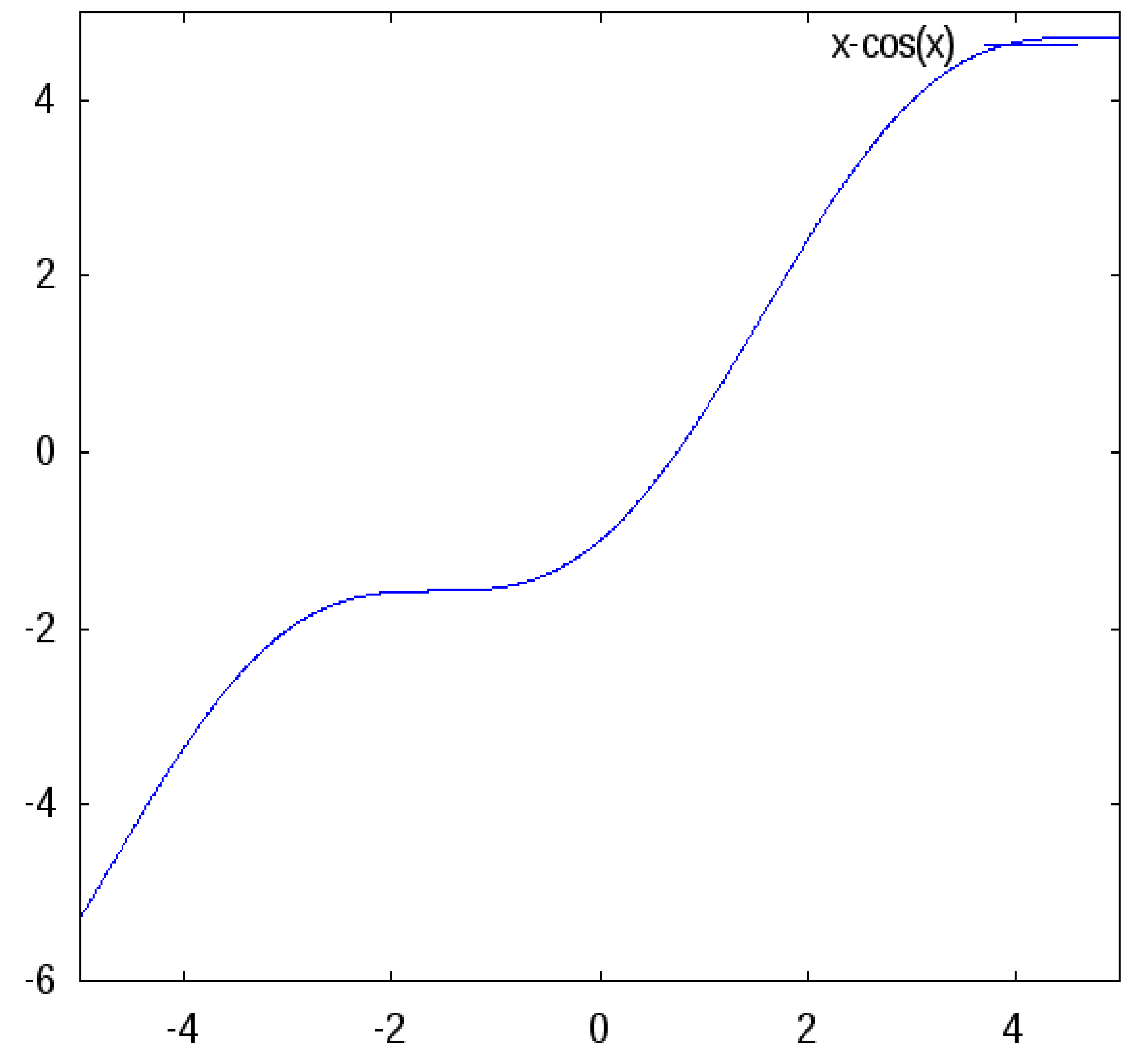
Find the solution of

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

First we need a rough guess of the solution – how do we get that?

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:



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: implementation

```
## module bisection
''' root = bisection(f,x1,x2,switch=0,tol=1.0e-9).
    Finds a root of f(x) = 0 by bisection.
    The root must be bracketed in (x1,x2).
    Setting switch = 1 returns root = None if
    f(x) increases upon bisection.
'''
from math import log,ceil
import error

def bisection(f,x1,x2,switch=1,tol=1.0e-9):
    f1 = f(x1)
    if f1 == 0.0: return x1
    f2 = f(x2)
    if f2 == 0.0: return x2
    if f1*f2 > 0.0: error.err('Root is not bracketed')
    n = ceil(log(abs(x2 - x1)/tol)/log(2.0))
    for i in range(n):
        x3 = 0.5*(x1 + x2); f3 = f(x3)
        if (switch == 1) and (abs(f3) > abs(f1)) \
                         and (abs(f3) > abs(f2)):
            return None
        if f3 == 0.0: return x3
        if f2*f3 < 0.0: x1 = x3; f1 = f3
        else:            x2 = x3; f2 = f3
    return (x1 + x2)/2.0
```

int

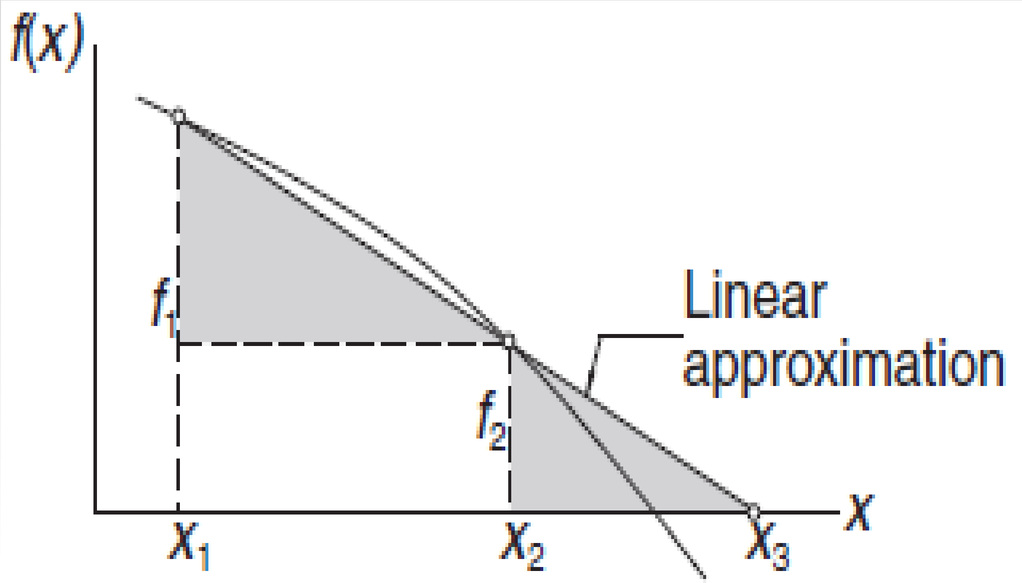
Bisection method: properties

- Method needs 39 iterations to solve example equation for tol=e=1e-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 → need $\log_2((b-a)/e)-1$ iterations for initial interval of length (b-a) and tolerance e, 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:



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

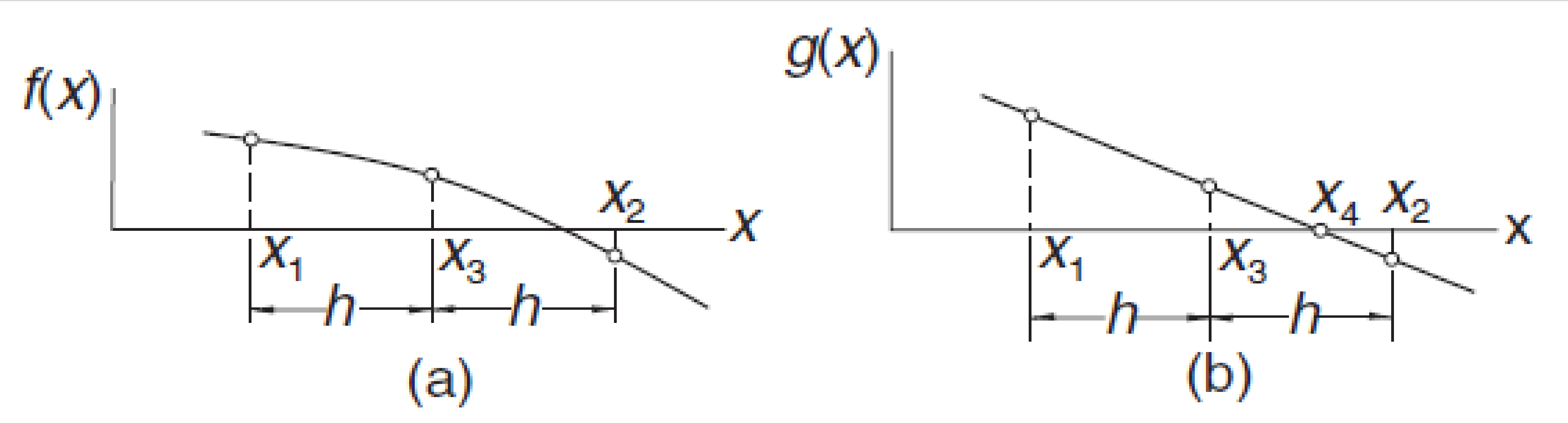
Or, solving for x_3: $$x_3 = x_2 - f_2 \frac{x_2 - x_1}{f_2 - f_1}$$

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

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

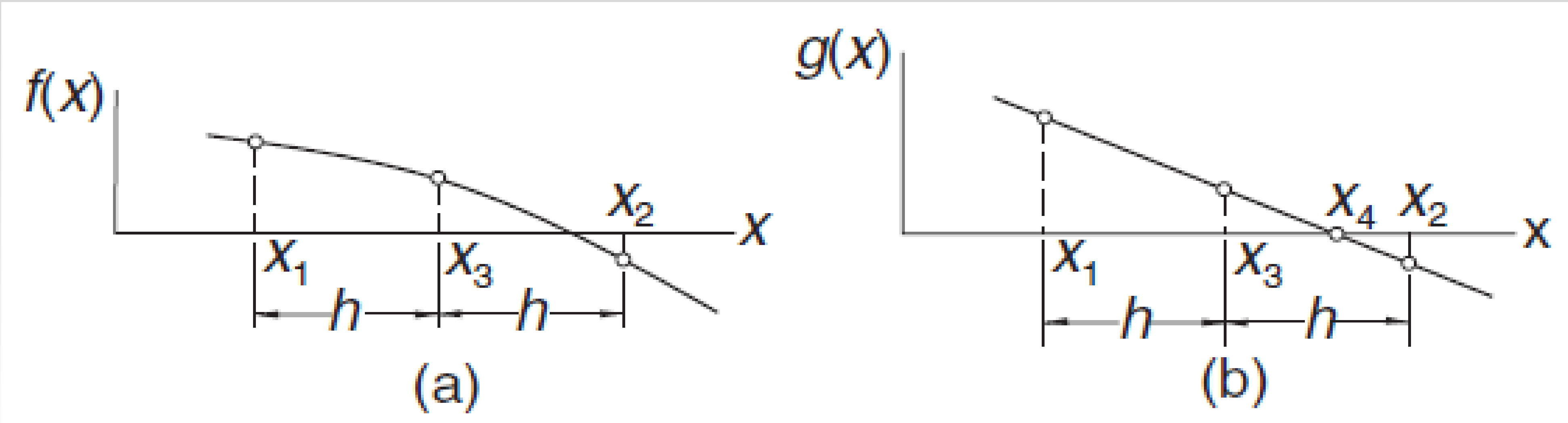
This can be shown to exhibit super-linear convergence, ~60% more correct digits per iteration than bisection.

Ridder's method



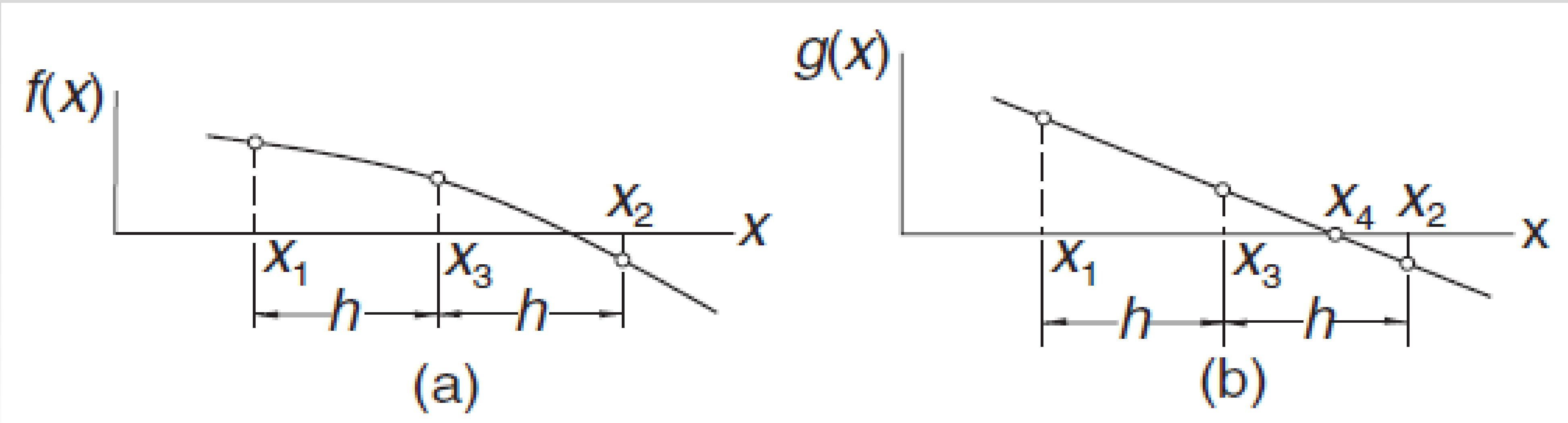
- Another method based on linear interpolation.
- Faster than the secant method (quadratic convergence).
- Has nice property that a bracketed root remains bracketed by subsequent approximations, making this method very reliable.

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: $g_1 = f_1 \qquad g_2 = f_2e^{2hQ} \qquad g_3 = f_3e^{hQ} \qquad h = (x_2 - x_1)/2$
- Straight line means $\rightarrow$ $g_3 = (g_1 + g_2)/2$ or $f_3e^{hQ} = \frac{1}{2}(f_1 + f_2e^{2hQ})$

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 (x1,g1) and (x3,g3) yields the improved root: $x_4 = x_3 - g_3 \dfrac{x_3 - x_1}{g_3 - g_1} = x_3 - f_3 e^{hQ} \dfrac{x_3 - x_1}{f_3 e^{hQ} - f_1}$

- Finally, we substitute the exponent from above:

  +: for $f_1 - f_2 > 0$

  - : for $f_1 - f_2 < 0$

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

Ridder's method: implementation

```
## module ridder
''' root = ridder(f,a,b,tol=1.0e-9).
    Finds a root of f(x) = 0 with Ridder's method.
    The root must be bracketed in (a,b).
'''
import error
from math import sqrt

def ridder(f,a,b,tol=1.0e-9):
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if fa*fb > 0.0: error.err('Root is not bracketed')
    for i in range(30):
      # Compute the improved root x from Ridder's formula
        c = 0.5*(a + b); fc = f(c)
        s = sqrt(fc**2 - fa*fb)
        if s == 0.0: return None
        dx = (c - a)*fc/s
        if (fa - fb) < 0.0: dx = -dx
        x = c + dx; fx = f(x)
      # Test for convergence
        if i > 0:
            if abs(x - xOld) < tol*max(abs(x),1.0): return x
        xOld = x
      # Re-bracket the root as tightly as possible
        if fc*fx > 0.0:
            if fa*fx < 0.0: b = x; fb = fx
            else:           a = x; fa = fx
        else:
            a = c; b = x; fa = fc; fb = fx
    return None
    print 'Too many iterations'
```

Newton-Raphson method

Based on Taylor expansion:

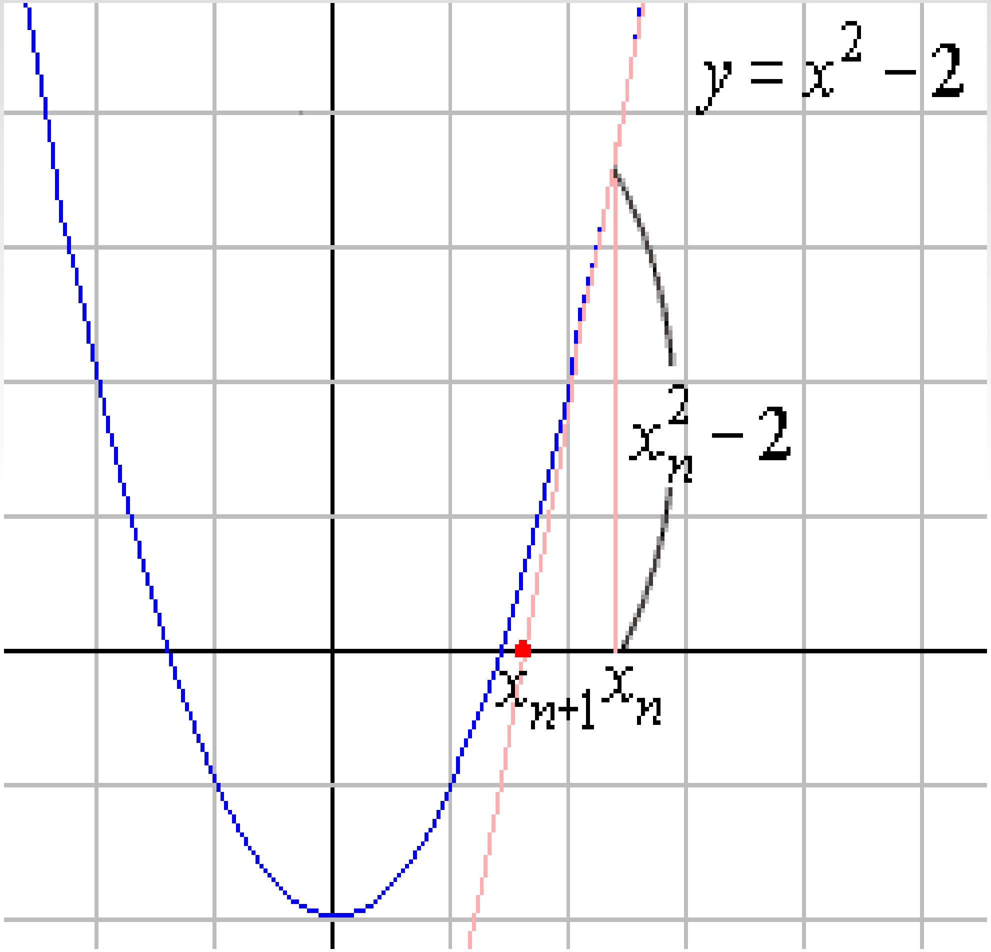
$f(x) = f(x_0) + (x-x_0)\, f'(x_0) + \ldots$

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

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

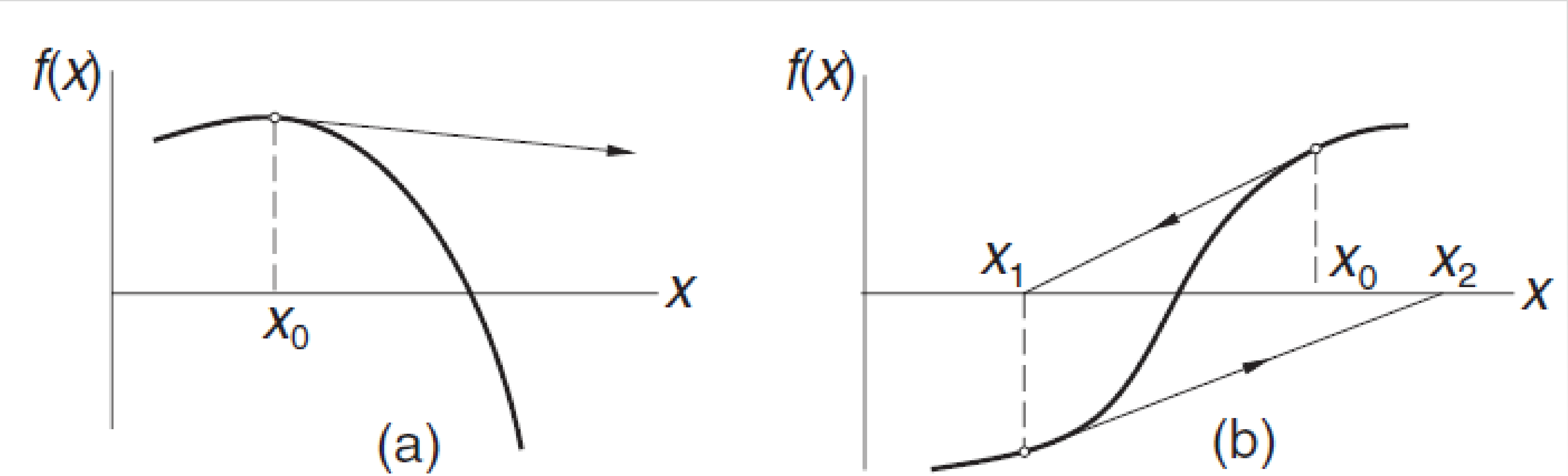
solving for x:

$x_1 = x_0 - f(x_0)/f'(x_0)$ → 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

```
## module newtonRaphson
''' root = newtonRaphson(f,df,a,b,tol=1.0e-9).
    Finds a root of f(x) = 0 by combining the Newton--Raphson
    method with bisection. The root must be bracketed in (a,b).
    Calls user-supplied functions f(x) and its derivative df(x).
'''
def newtonRaphson(f,df,a,b,tol=1.0e-9):
    import error
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if fa*fb > 0.0: error.err('Root is not bracketed')
    x = 0.5*(a + b)
    for i in range(30):
        fx = f(x)
        if abs(fx) < tol: return x
      # Tighten the brackets on the root
        if fa*fx < 0.0:
            b = x
        else:
            a = x
      # Try a Newton-Raphson step
        dfx = df(x)
      # If division by zero, push x out of bounds
        try: dx = -fx/dfx
        except ZeroDivisionError: dx = b - a
        x = x + dx
      # If the result is outside the brackets, use bisection
        if (b - x)*(x - a) < 0.0:
            dx = 0.5*(b-a)
            x = a + dx
      # Check for convergence
        if abs(dx) < tol*max(abs(b),1.0): return x
    print 'Too many iterations in Newton-Raphson'
```

Newton-Raphson method: more sophisticated implementation

Here bisection method is used to find better approximation if the first guess is not sufficiently good for the Newton-Raphson method to converge quickly.

Note that both the function and its derivative are required, along with a bracketing interval.

Newton-Raphson: systems of equations

- We want to solve the (nonlinear) system:

$$
\begin{aligned}
f_1(x_1, x_2, \ldots, x_n) &= 0 \\
f_2(x_1, x_2, \ldots, x_n) &= 0 \\
&\vdots \\
f_n(x_1, x_2, \ldots, x_n) &= 0
\end{aligned}
$$

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

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

- This can be written as:

$$\mathbf{f}(\mathbf{x} + \mathbf{\Delta x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\, \mathbf{\Delta x} \qquad J_{ij} = \frac{\partial f_i}{\partial x_j} \longleftarrow \text{Jacobian matrix}$$

Newton-Raphson: systems of equations

- Let x is the current approximation to the solution of **f(x)=0** and **x+Δx** is the improved one. To find **Δx** we set **f(x+Δx)=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 **x** and evaluate **f(x)**.
  - Compute the Jacobian matrix **J(x)**.
  - Set up the linear system above and solve for **Δx**.
  - Let **x+Δx** → **x** and repeat steps until convergence.

Newton-Raphson method for systems: implementation

```
## module newtonRaphson2
''' soln = newtonRaphson2(f,x,tol=1.0e-9).
    Solves the simultaneous equations f(x) = 0 by
    the Newton-Raphson method using {x} as the initial
    guess. Note that {f} and {x} are vectors.
'''
from numpy import zeros,dot
from gaussPivot import *
from math import sqrt

def newtonRaphson2(f,x,tol=1.0e-9):

    def jacobian(f,x):
        h = 1.0e-4
        n = len(x)
        jac = zeros((n,n))
        f0 = f(x)
        for i in range(n):
            temp = x[i]
            x[i] = temp + h
            f1 = f(x)
            x[i] = temp
            jac[:,i] = (f1 - f0)/h
        return jac,f0

    for i in range(30):
        jac,f0 = jacobian(f,x)
        if sqrt(dot(f0,f0)/len(x)) < tol: return x
        dx = gaussPivot(jac,-f0)
        x = x + dx
        if sqrt(dot(dx,dx)) < tol*max(max(abs(x)),1.0): return x
    print 'Too many iterations'
```

Jacobian matrix is calculated internally by finite-differencing (see next topic in course).

Note that at every iteration the Jacobian is calculated and a linear system is solved, which could be fairly expensive.

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 √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 ← Square root of 5
Number of iterations = 3
Press return to exit

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, 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).