Numerical Solutions of nonlinear equations.

Problem: Given $f(x)$: continuous, real-valued, possibly non-linear. Find a root r of $f(x)$ such that $f(r) = 0$.

- Bisection
- Fixed point iteration
- Newton's method
- Secant method

Bisection Method

Basic idea: Given $f(x)$, a continuous function. If we find some a and b, such that $f(a)$ and $f(b)$ are of opposite sign, then, there exists a point c, between a and b, such that $f(c) = 0$.

This fact follows from the continuity of f.

Procedure:

- Initialization: Find a, b such that $f(a) \cdot f(b) < 0$. This means there is a root $r \in (a, b)$ s.t. $f(r) = 0$.
- Let $c = \frac{a+b}{2}$, mid-point.
  - If $f(c) = 0$, done (lucky!)
  - else:
    - if $f(c) \cdot f(a) < 0$, pick the interval $[a, c]$
    - if $f(c) \cdot f(b) < 0$, pick the interval $[c, b]$,
  - Iterate the procedure until stop criteria satisfied.

Stop Criteria:

1) interval small enough, i.e., $(b - a) \leq \epsilon$,

2) $|f(c)|$ very small, i.e, $|f(c)| \leq \epsilon$

3) max number of iteration reached. (to avoid dead loop, in case the method does not converge.)

4) any combination of the previous ones.

Convergence Analysis

Consider $[a_0, b_0]$, $c_0 = \frac{a_0+b_0}{2}$, let $r \in (a_0, b_0)$ be a root. The error: $\quad e_0 = |r - c_0| \leq \frac{b_0-a_0}{2}$

Denote the further intervals as $[a_n, b_n]$ for iteration number n.

$$e_n = |r - c_n| \leq \frac{b_n - a_n}{2} \leq \frac{b_0 - a_0}{2^{n+1}} = \frac{e_0}{2^n}.$$

If the error tolerance is ε, we require $e_n \leq \varepsilon$, then

$$\frac{b_0 - a_0}{2^{n+1}} \le \varepsilon \Rightarrow n \ge \frac{\ln(b-a) - \ln(2\varepsilon)}{\ln 2}, \qquad (\text{\# of steps})$$

Fixed point iteration

We rewrite the equation $f(x) = 0$ into the form $x = g(x)$. Remark: This can always be achieved, for example: $x = f(x) + x$. However, the choice of g makes a difference in convergence.

Main idea: Make a guess of the solution, say $\bar{x}$. If the function $g(x)$ is "nice", then hopefully, $g(\bar{x})$ should be closer to the answer than $\bar{x}$. If that is the case, then we can iterate.

Iteration algorithm:

- Choose a start point x_0,
- Do the iteration $x_{k+1} = g\left(x_k\right), k = 0, 1, 2, \cdots$ until meeting stop criteria.

Stop Criteria: Let ε be the tolerance

- $\left|x_k - x_{k-1}\right| \le \varepsilon$,
- max # of iteration reached,
- any combination.

Example 1

Find an approximate solution to $f(x) = x - \cos x = 0$, with 4 digits accuracy.

Choose $g(x) = \cos x$, we have $x = \cos x$. Choose $x_0 = 1$, and do the iteration $x_{k+1} = \cos(x_k)$:

In [ ]:
```
import math

# Define the function g(x) = cos(x)
def g(x):
    return math.cos(x)

# Initial guess
x0 = 1.0

# Tolerance for convergence
tolerance = 1e-4

# Perform fixed-point iteration
x_k = x0
iteration = 0

while True:
    x_k1 = g(x_k)
    print(f"x_{iteration} = {x_k:.4f}")
    if abs(x_k1 - x_k) < tolerance:
        break
    x_k = x_k1
    iteration += 1

print(f"x_{iteration + 1} = {x_k1:.4f} (converged)")
```

```
x_0 = 1.0000
x_1 = 0.5403
x_2 = 0.8576
```

```
x_3 = 0.6543
x_4 = 0.7935
x_5 = 0.7014
x_6 = 0.7640
x_7 = 0.7221
x_8 = 0.7504
x_9 = 0.7314
x_10 = 0.7442
x_11 = 0.7356
x_12 = 0.7414
x_13 = 0.7375
x_14 = 0.7401
x_15 = 0.7384
x_16 = 0.7396
x_17 = 0.7388
x_18 = 0.7393
x_19 = 0.7389
x_20 = 0.7392
x_21 = 0.7390
x_22 = 0.7391
x_23 = 0.7391 (converged)
```

Example 2

Consider $f(x) = e^{-2x}(x-1) = 0$. (root: $r = 1$). Rewrite as

$$x = g(x) = e^{-2x}(x-1) + x$$

Choose an initial guess $x_0 = 0.99$, very close to the real root.

In [ ]:
```
import math

# Define the function g(x)
def g(x):
    return math.exp(-2 * x) * (x - 1) + x

# Initial guess
x0 = 0.99

# Tolerance for convergence (just for safety, though the example diverges)
tolerance = 1e-4

# Perform fixed-point iteration
x_k = x0
iteration = 0
max_iterations = 30  # Limit iterations to prevent infinite loop

while iteration < max_iterations:
    x_k1 = g(x_k)
    print(f"x_{iteration + 1} = {x_k1:.4f}")
    if abs(x_k1 - x_k) < tolerance:
        break
    x_k = x_k1
    iteration += 1

# Check if the process diverged
if iteration == max_iterations:
    print("Diverged. The iteration does not work.")
else:
    print(f"Converged to {x_k1:.4f} after {iteration + 1} iterations.")
```

```
x_1 = 0.9886
x_2 = 0.9870
x_3 = 0.9852
```

```
x_4 = 0.9832
x_5 = 0.9808
x_6 = 0.9781
x_7 = 0.9750
x_8 = 0.9715
x_9 = 0.9674
x_10 = 0.9627
x_11 = 0.9573
x_12 = 0.9510
x_13 = 0.9437
x_14 = 0.9351
x_15 = 0.9251
x_16 = 0.9133
x_17 = 0.8994
x_18 = 0.8828
x_19 = 0.8627
x_20 = 0.8382
x_21 = 0.8080
x_22 = 0.7698
x_23 = 0.7205
x_24 = 0.6543
x_25 = 0.5609
x_26 = 0.4179
x_27 = 0.1655
x_28 = -0.4338
x_29 = -3.8477
x_30 = -10659.9637
Diverged. The iteration does not work.
```

Fixed Point iteration, convergence

Our iteration is $x_{k+1}=g\left(x_k\right)$. Let r be the exact root, s.t., $r=g(r)$. Define the error: $e_k=\left|x_k-r\right|$.

$$
\begin{aligned}
e_{k+1} & =\left|x_{k+1}-r\right|=\left|g\left(x_k\right)-r\right|=\left|g\left(x_k\right)-g(r)\right| \\
& =\left|g^{\prime}(\xi)\right|\left|\left(x_k-r\right)\right| \quad\left(\xi \in\left(x_k, r\right), \text { since } g \text { is continuous }\right) \\
& =\left|g^{\prime}(\xi)\right| e_k
\end{aligned}
$$

$$
\Rightarrow \quad e_{k+1}=\left|g^{\prime}(\xi)\right| e_k .
$$

Observation:

- If $\left|g^{\prime}(\xi)\right|<1$, then $e_{k+1}<e_k$, error decreases, the iteration convergence. (linear convergence)
- If $\left|g^{\prime}(\xi)\right|>1$, then $e_{k+1}>e_k$, error increases, the iteration diverges.

Convergence condition: There exists an interval around r, say $[r-a, r+a]$ for some $a>0$, such that $\left|g^{\prime}(x)\right|<1$ for almost all $x \in[r-a, r+a]$, and the initial guess x_0 lies in this interval.

In Example 1, $g(x)=\cos x, g^{\prime}(x)=\sin x, r=0.7391$, $\left|g^{\prime}(r)\right|=|\sin (0.7391)|<1$. OK, convergence.

In Example 2, we have

$$
\begin{aligned}
g(x) & =e^{-2 x}(x-1)+x, \\
g^{\prime}(x) & =-2 e^{-2 x}(x-1)+x^{-2 x}+1
\end{aligned}
$$

With $r=1$, we have

$$|g'(r)| = e^{-2} + 1 > 1, \qquad \text{Divergence.}$$

A practical error estimate:

Assume $|g'(x)| \leq m < 1$ in $[r-a, r+a]$. We have $e_{k+1} \leq m e_k$. This gives

$$e_1 \leq m e_0, \quad e_2 \leq m e_1 \leq m^2 e_0, \quad \cdots \quad e_k \leq m^k e_0.$$

This result is useless unless we find a way to estimating e_0.

$$e_0 = |r - x_0| = |r - x_1 + x_1 - x_0| \leq e_1 + |x_1 - x_0| \leq m e_0 + |x_1 - x_0|$$

then

$$e_0 \leq \frac{1}{1-m} |x_1 - x_0|, \qquad \text{(can be computed)}$$

Put together

$$e_k \leq \frac{m^k}{1-m} |x_1 - x_0|$$

$$e_k \leq \frac{m^k}{1-m} |x_1 - x_0|$$

f the error tolerance is ε, then

$$\frac{m^k}{1-m} |x_1 - x_0| \leq \varepsilon,$$

If the error tolerance is ε, then

$$\frac{m^k}{1-m} |x_1 - x_0| \leq \varepsilon,$$

$$m^k \leq \frac{\varepsilon(1-m)}{|x_1 - x_0|}$$

$$k \geq \frac{\ln(\varepsilon(1-m)) - \ln|x_1 - x_0|}{\ln m}$$

which give the minimum number of iterations needed to achieve an error $\leq \varepsilon$.

Example 3. We want to solve $\cos x - x = 0$ with the fixed point iteration

$$x = g(x) = \cos x, \quad x_0 = 1,$$

with error tolerance $\varepsilon = 10^{-5}$. Find the $\min \#$ iterations. We know $r \in [0, 1]$. We see that the iteration happens between $x = 0$ and $x = 1$. For $x \in [0, 1]$, we have

$$|g'(x)| = |\sin x| \leq \sin 1 = 0.8415, \quad m \doteq 0.8415$$

And $x_1 = \cos x_0 = 0.5403$. Using the formula

$$k \geq \frac{\ln(\varepsilon(1-m)) - \ln|x_1 - x_0|}{\ln m} \approx 73 \quad \# \text{ iterations needed}$$

In actual simulation $k = 25$ is enough.

Newton's Method

Goal: Given $f(x)$, find a root r s.t. $f(r)=0$.

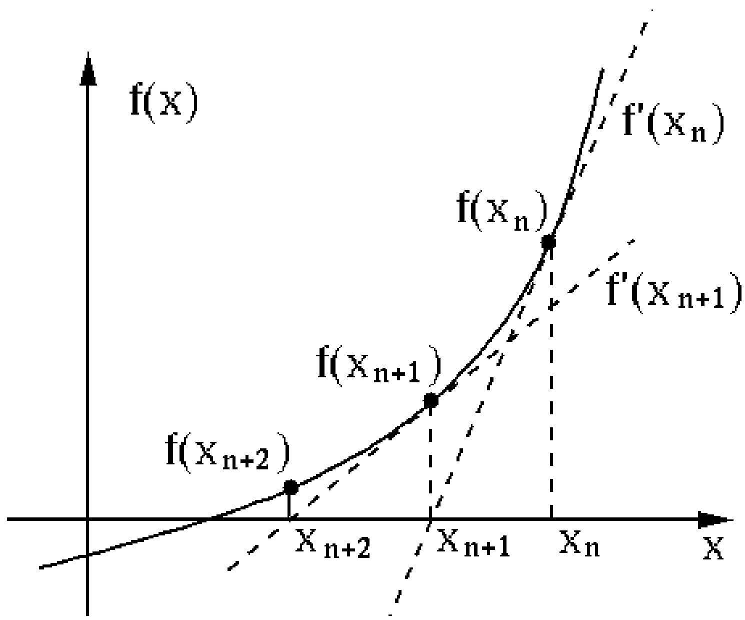
Main idea: Given x_k, the next approximation x_{k+1} is determined by approximating $f(x)$ as a linear function at x_k.

We have

$$\frac{f\left(x_k\right)}{x_k-x_{k+1}}=f^{\prime}\left(x_k\right)$$

which gives

$$x_{k+1}=x_k-\frac{f\left(x_k\right)}{f^{\prime}\left(x_k\right)}$$



Newton's method is a fixed point iteration

$$x_{k+1}=g\left(x_k\right), \quad g(x)=x-\frac{f(x)}{f^{\prime}(x)}.$$

If r is a fixed point, assume $f^{\prime}(r) \neq 0$, then

$$r=g(r), \quad r=r-\frac{f(r)}{f^{\prime}(r)}, \quad \frac{f(r)}{f^{\prime}(r)}=0, \quad f(r)=0.$$

then r is a root.

Observation: A fixed point iteration $x=g(x)$ is optimal if $g^{\prime}(r)=0$ where $r=g(r)$. For Newton, we have

$$g^{\prime}(x)=1-\frac{f^{\prime}(x) f^{\prime}(x)-f^{\prime \prime}(x) f(x)}{\left(f^{\prime}(x)\right)^2}=\frac{f^{\prime \prime}(x) f(x)}{\left(f^{\prime}(x)\right)^2}$$

Then

$$g'(r)=\frac{f''(r)f(r)}{(f'(r))^2}=0$$

which is the "best" possible scenario!

Newton's iteration, convergence

Let r be the root so $f(r)=0$ and $r=g(r)$. Recall $g'(r)=0$. Define Error:
$e_{k+1}=|x_{k+1}-r|=|g(x_k)-g(r)|$

Taylor expansion for $g(x_k)$ at r :

$$\begin{aligned} g(x_k) =& g(r)+(x_k-r)\,g'(r)+\frac{1}{2}(x_k-r)^2 g''(\xi), \quad \xi\in(x_k,r) \\ =& g(r)+\frac{1}{2}(x_k-r)^2 g''(\xi) \\ e_{k+1} =& \frac{1}{2}(x_k-r)^2\left|g''(\xi)\right|=\frac{1}{2}e_k^2\left|g''(\xi)\right| \end{aligned}$$

Write now $M=\frac{1}{2}\max_x\left|g''(\xi)\right|$, we have

$$e_{k+1}\le M(e_k)^2$$

This is called quadratic convergence.

Theorem: If $e_{k+1}\le Me_k^2$, then $\lim_{k\to+\infty}e_k=0$ if e_0 is sufficiently small. (This means, M can be big, but it would not effect the convergence!)

Proof for the convergence: We have

$$e_1\le(Me_0)\,e_0$$

If e_0 is small enough, such that $(Me_0)<1$, then $e_1<e_0$. This means $(Me_1)<Me_0<1$, and so

$$e_2\le(Me_1)\,e_1<e_1, \quad \Rightarrow \quad Me_2<Me_1<1$$

By an induction argument, we conclude that $e_{k+1}<e_k$ for all k, i.e., error is strictly decreasing after each iteration. $\Rightarrow$ convergence.

Example

Find a numerical method to compute $\sqrt{a}$ using only $+,-,*,/$ arithmetic operations. Test it for $a=3$.
Answer. It's easy to see that $\sqrt{a}$ is a root for $f(x)=x^2-a$. Newton's method gives

$$x_{k+1}=x_k-\frac{f(x_k)}{f'(x_k)}=x_k-\frac{x_k^2-a}{2x_k}=\frac{x_k}{2}+\frac{a}{2x_k}$$

Test it on $a=3$: Choose $x_0=1.7$.

$$
\begin{array}{ll}
x_0 = 1.7 & \text{error} \\
x_1 = 1.7324 & 7.2 \times 10^{-2} \\
x_2 = 1.7321 & 3.0 \times 10^{-4} \\
x_3 = 1.7321 & 2.6 \times 10^{-8} \\
 & 4.4 \times 10^{-16}
\end{array}
$$

Note the extremely fast convergence

Example

To find the root of $f(x) = e^{-2x}(x-1)$ using Newton's method, we need to follow these steps:

1. Define the function $f(x)$ and its derivative $f'(x)$.
2. Choose an initial guess x_0.
3. Perform the iteration $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ until the result converges.

First, let's compute the derivative $f'(x)$:

$$f(x) = e^{-2x}(x-1)$$

Using the product rule:

$$f'(x) = \frac{d}{dx}\left[e^{-2x}(x-1)\right] = e^{-2x}(-2(x-1)) + e^{-2x} = e^{-2x}(-2x+2+1) = e^{-2x}(-2x+3)$$

Now we have:

$$
\begin{gathered}
f(x) = e^{-2x}(x-1) \\
f'(x) = e^{-2x}(-2x+3)
\end{gathered}
$$

In [ ]:

```
import math

# Define the function f(x)
def f(x):
    return math.exp(-2 * x) * (x - 1)

# Define the derivative f'(x)
def f_prime(x):
    return math.exp(-2 * x) * (-2 * x + 3)

# Initial guess
x0 = 0.99  # Close to the root

# Tolerance for convergence
tolerance = 1e-6

# Perform Newton's method iteration
x_k = x0
iteration = 0
max_iterations = 100  # Limit iterations to prevent infinite loop

while iteration < max_iterations:
    f_xk = f(x_k)
    f_prime_xk = f_prime(x_k)
```

```
    # Ensure we don't divide by zero
    if f_prime_xk == 0:
        print("Derivative is zero. No solution found.")
        break

    x_k1 = x_k - f_xk / f_prime_xk
    print(f"Iteration {iteration + 1}: x = {x_k1:.6f}")

    if abs(x_k1 - x_k) < tolerance:
        print(f"Converged to {x_k1:.6f} after {iteration + 1} iterations.")
        break

    x_k = x_k1
    iteration += 1

if iteration == max_iterations:
    print("Did not converge within the maximum number of iterations.")
```

```
Iteration 1: x = 0.999804
Iteration 2: x = 1.000000
Iteration 3: x = 1.000000
Converged to 1.000000 after 3 iterations.
```

Secant Method

If $f(x)$ is complicated, $f'(x)$ might not be available. Solution for this situation: using approximation for f', i.e.,

$$f'\left(x_k\right) \approx \frac{f\left(x_k\right)-f\left(x_{k-1}\right)}{x_k-x_{k-1}}$$

This is secant method:

$$x_{k+1}=x_k-\frac{x_k-x_{k-1}}{f\left(x_k\right)-f\left(x_{k-1}\right)} f\left(x_k\right)$$

Initial guess needed: x_0, x_1.

Advantages include

- No computation of f';
- One $f(x)$ computation each step;
- Also rapid convergence.

A bit on convergence: One can show that

$$e_{k+1} \leq C e_k^\alpha, \quad \alpha=\frac{1}{2}(1+\sqrt{5}) \approx 1.62$$

This is called super linear convergence. $(1<\alpha<2)$

It converges for all function f if x_0 and x_1 are close to the root r.

Example

Use secant method for computing $\sqrt{a}$. Answer. The iteration now becomes

$$x_{k+1} = x_k - \frac{\left(x_k^2 - a\right)\left(x_k - x_{k-1}\right)}{\left(x_k^2 - a\right) - \left(x_{k-1}^2 - a\right)} = x_k - \frac{x_k^2 - a}{x_k + x_{k-1}}$$

Test with $a = 3$, with initial data $x_0 = 1.65$, $x_1 = 1.7$.

$$\begin{array}{ll} x_1 = 1.7 & \text{error} \\ x_2 = 1.7328 & 7.2 \times 10^{-2} \\ x_3 = 1.7320 & 7.9 \times 10^{-4} \\ x_4 = 1.7321 & 7.3 \times 10^{-6} \\ x_5 = 1.7321 & 3.7 \times 10^{-9} \end{array}$$

It is a little but slower than Newton's method, but not much.

Example

1. Define the function $f(x)$.
2. Choose two initial guesses x_0 and x_1.
3. Perform the iteration $x_{k+1} = x_k - f\left(x_k\right) \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$ until the result converges.

In [ ]:

```
import math

# Define the function f(x)
def f(x):
    return math.exp(-2 * x) * (x - 1)

# Initial guesses
x0 = 0.5  # First initial guess
x1 = 0.6  # Second initial guess

# Tolerance for convergence
tolerance = 1e-6

# Perform secant method iteration
iteration = 0
max_iterations = 100  # Limit iterations to prevent infinite loop

while iteration < max_iterations:
    f_x0 = f(x0)
    f_x1 = f(x1)

    # Ensure we don't divide by zero
    if f_x1 == f_x0:
        print("f(x1) and f(x0) are equal. Division by zero encountered.")
        break

    x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0)
    print(f"Iteration {iteration + 1}: x = {x2:.6f}")

    if abs(x2 - x1) < tolerance:
        print(f"Converged to {x2:.6f} after {iteration + 1} iterations.")
        break

    x0, x1 = x1, x2
    iteration += 1

if iteration == max_iterations:
    print("Did not converge within the maximum number of iterations.")
```

```
Iteration 1: x = 0.789842
Iteration 2: x = 0.896355
Iteration 3: x = 0.966937
Iteration 4: x = 0.993979
Iteration 5: x = 0.999617
Iteration 6: x = 0.999995
Iteration 7: x = 1.000000
Iteration 8: x = 1.000000
Converged to 1.000000 after 8 iterations.
```

System of Linear Equations

Direct methods for systems of linear equations

The problem: n equations, n unknowns,

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = & b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = & b_2 \\ & \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = & b_n \end{cases}$$

In matrix-vector form:

$$A\vec{x} = \vec{b},$$

where $A \in \boldsymbol{R}^{n\times n}, \quad \vec{x} \in \boldsymbol{R}^n, \quad \vec{b} \in \boldsymbol{R}^n$

$$A = \{a_{ij}\} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}.$$

```
plaintext
Algorithm GaussianElimination(A, b)
    Input: Matrix A (n x n) and vector b (n)
    Output: Solution vector x (n) such that Ax = b

    Step 1: Forward Elimination
    for k from 1 to n-1 do
        for i from k+1 to n do
            if A[k][k] == 0 then
                swap rows k and a row below it with a non-zero element
in the k-th column
            end if
            factor = A[i][k] / A[k][k]
            for j from k to n do
                A[i][j] = A[i][j] - factor * A[k][j]
            end for
            b[i] = b[i] - factor * b[k]
        end for
    end for

    Step 2: Back Substitution
```

```
x[n] = b[n] / A[n][n]
for i from n-1 to 1 do
    sum = 0
    for j from i+1 to n do
        sum = sum + A[i][j] * x[j]
    end for
    x[i] = (b[i] - sum) / A[i][i]
end for

return x
```