Root Finding

Root finding refers to the general problem of searching for a solution of an equation $F(x) = 0$ for some function F. If we want to optimise a function $f(x)$ then we need to find critical points and therefore solve the equation $f'(x) = 0$.
Example quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Source

Bisection Method

The algorithm applies to any continuous function $f(x)$ on an interval a,b where the value of the function $f(x)$ changes sign from a to b . The idea is simple: divide the interval in two, a solution must exist within one subinterval, select the subinterval where the sign of $f(x)$ changes and repeat.

Algorithm

The bisection method procedure is:

1. Choose a starting interval $[a_0, b_0]$ such that $f(a_0)f(b_0) < 0$
2. Compute $f(m_0)$ where $m_0 = (a_0 + b_0)/2$ is the midpoint.
3. Determine the next subinterval $[a_1, b_1]$:
   (a) If $f(a_0)f(m_0) < 0$, then let $[a_1, b_1]$ be the next interval with $a_1 = a_0$ and $b_1 = m_0$.
   (b) If $f(b_0)f(m_0) < 0$, then let $[a_1, b_1]$ be the next interval with $a_1 = m_0$ and $b_1 = b_0$.
4. Repeat (2) and (3) until the interval $[a_N, b_N]$ reaches some predetermined length.
5. Return the midpoint value $m_N = (a_N + b_N)/2$

A solution of the equation $f(x)$ on an interval a,b is guaranteed by the Intermediate Value Theorem provided $f(x)$ is continuous on $[a,b]$ and $f(a)f(b) < 0$. In other words, the function changes sign over the interval and therefore must equal 0 at some point in the interval $[a,b]$.

Python Implementation

Write a function called `bisection` which takes 4 input parameters `f, a, b` and `N` and returns the approximation of a solution of $f(x) = 0$ given by N iterations of the bisection method. If $f(a_N)f(b_N) > 0$ at any point in the iteration (caused either by a bad initial interval or rounding error in computations), then print "`Bisection method fails.`" and return `None`.

```
def bisection(f,a,b,N):
    '''Approximate solution of f(x)=0 on interval [a,b] by bisection method.

    Parameters
    ----------
    f : function
        The function for which we are trying to approximate a solution f(x)=0.
    a,b : numbers
        The interval in which to search for a solution. The function returns
        None if f(a)*f(b) >= 0 since a solution is not guaranteed.
    N : (positive) integer
        The number of iterations to implement.

    Returns
    -------
    x_N : number
        The midpoint of the Nth interval computed by the bisection method. The
        initial interval [a_0,b_0] is given by [a,b]. If f(m_n) == 0 for some
        midpoint m_n = (a_n + b_n)/2, then the function returns this solution.
        If all signs of values f(a_n), f(b_n) and f(m_n) are the same at any
        iteration, the bisection method fails and return None.

```

```
    Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> bisection(f,1,2,25)
    1.618033990263939
    >>> f = lambda x: (2*x - 1)*(x - 3)
    >>> bisection(f,0,1,10)
    0.5
    '''
    if f(a)*f(b) >= 0:
        print("Bisection method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = (a_n + b_n)/2
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Bisection method fails.")
            return None
    return (a_n + b_n)/2
```

Source

Secant method

The secant method is very similar to the bisection method except instead of dividing each interval by choosing the midpoint the secant method divides each interval by the secant line connecting the endpoints. The secant method always converges to a root of $f(x)$ is continuous on $[a,b]$ and $f(a)f(b) < 0$.

Secant line formula

Let $f(x)$ be a continuous function on $[a, b]$ and $f(a)f(b) < 0$. A solution of the equation $f(x) = 0$ for $x \in [a, b]$ is guaranteed by the Intermediate Value Theorem. Consider the line connecting the endpoint values $(a, f(a))$ and $(b, f(b))$. The line connecting these two points is called the secant line and is given by the formula

$$y = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$

The point where the secant line crosses the x-axis is

$$0 = \frac{f(b) - f(a)}{b - a}(x - a) + f(a)$$

which we solve for x

$$x = a - f(a)\frac{b - a}{f(b) - f(a)}$$

Algorithm

The secant method procedure is:

1. Choose a starting interval $[a_0, b_0]$ such that $f(a_0)f(b_0) < 0$
2. Compute $f(x_0)$ where x_0 is given by the secant line :

$$x_0 = a_0 - f(a_0)\frac{b_0 - a_0}{f(b_0) - f(a_0)}$$

3. Determine the next subinterval $[a_1, b_1]$:
   (a) If $f(a_0)f(m_0) < 0$, then let $[a_1, b_1]$ be the next interval with $a_1 = a_0$ and $b_1 = x_0$.
   (b) If $f(b_0)f(m_0) < 0$, then let $[a_1, b_1]$ be the next interval with $a_1 = x_0$ and $b_1 = b_0$.
4. Repeat (2) and (3) until the interval $[a_N, b_N]$ reaches some predetermined length.
5. Return the value x_N, the x-intercept of the Nth subinterval.

A solution of the equation $f(x)$ on an interval a,b is guaranteed by the Intermediate Value Theorem provided $f(x)$ is continuous on $[a, b]$ and $f(a)f(b) < 0$. In other words, the function changes sign over the interval and therefore must equal 0 at some point in the interval $[a, b]$.

Python Implementation

Write a function called `secant` which takes 4 input parameters `f`, `a`, `b` and `N` and returns the approximation of a solution of $f(x) = 0$ given by N iterations of the secant method. If $f(a_N)f(b_N) > 0$ at any point in the iteration (caused either by a bad initial interval or rounding error in computations), then print "`Secant method fails.`" and return `None`.

```python
def secant(f,a,b,N):
    '''Approximate solution of f(x)=0 on interval [a,b] by the secant method.

    Parameters
    ----------
    f : function
        The function for which we are trying to approximate a solution f(x)=0.
    a,b : numbers
        The interval in which to search for a solution. The function returns
        None if f(a)*f(b) >= 0 since a solution is not guaranteed.
    N : (positive) integer
        The number of iterations to implement.

    Returns
    -------
    m_N : number
        The x intercept of the secant line on the the Nth interval
            m_n = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
        The initial interval [a_0,b_0] is given by [a,b]. If f(m_n) == 0
        for some intercept m_n then the function returns this solution.
        If all signs of values f(a_n), f(b_n) and f(m_n) are the same at any
        iterations, the secant method fails and return None.

    Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> secant(f,1,2,5)
    1.6180257510729614
```

```
    '''
    if f(a)*f(b) >= 0:
        print("Secant method fails.")
        return None
    a_n = a
    b_n = b
    for n in range(1,N+1):
        m_n = a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
        f_m_n = f(m_n)
        if f(a_n)*f_m_n < 0:
            a_n = a_n
            b_n = m_n
        elif f(b_n)*f_m_n < 0:
            a_n = m_n
            b_n = b_n
        elif f_m_n == 0:
            print("Found exact solution.")
            return m_n
        else:
            print("Secant method fails.")
            return None
    return a_n - f(a_n)*(b_n - a_n)/(f(b_n) - f(a_n))
```

Source

Newton Method

Newton's method is a root finding method that uses linear approximation. In particular, we guess a solution x_0 of the equation $f(x) = 0$, compute the linear approximation of $f(x)$ at x_0 and then find the x-intercept of the linear approximation.

Newton's formula

Let $f(x)$ be a differentiable function. If x_0 is near a solution of $f(x) = 0$ then we can approximate $f(x)$ by the tangent line at x_0 and compute the x-intercept of the tangent line. The equation of the tangent line at x_0 is

$$y = f'(x_0)(x - x_0) + f(x_0)$$

The x-intercept is the solution x_1 of the equation

$$0 = f'(x_0)(x_1 - x_0) + f(x_0)$$

and we solve for x_1

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

If we implement this procedure repeatedly, then we obtain a sequence given by the recursive formula

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

which (potentially) converges to a solution of the equation $f(x) = 0$.

Python Implementation

Write a function called newton which takes 5 input parameters `f, Df, x0, epsilon` and `maxinter` and returns the approximation of a solution of $f(x) = 0$ given by Newton's method.
The function may terminate in 3 ways:

1. If `abs(f(xn)) < epsilon`, the algorithm has found an approximate solution and returns `xn`.
2. If `f'(xn) == 0`, the algorithm stops and returns `None`.
3. If the number of iterations exceeds `maxinter`, the algorithm stops and returns `None`.

```
def newton(f,Df,x0,epsilon,max_iter):
'''Approximate solution of f(x)=0 by Newton's method.

Parameters
----------
f : function
    Function for which we are searching for a solution f(x)=0.
Df : function
    Derivative of f(x).
x0 : number
    Initial guess for a solution f(x)=0.
epsilon : number
    Stopping criteria is abs(f(x)) < epsilon.
max_iter : integer
    Maximum number of iterations of Newton's method.
```

```

    Returns
    -------
    xn : number
        Implement Newton's method: compute the linear approximation
        of f(x) at xn and find x intercept by the formula
            x = xn - f(xn)/Df(xn)
        Continue until abs(f(xn)) < epsilon and return xn.
        If Df(xn) == 0, return None. If the number of iterations
        exceeds max_iter, then return None.

    Examples
    --------
    >>> f = lambda x: x**2 - x - 1
    >>> Df = lambda x: 2*x - 1
    >>> newton(f,Df,1,1e-8,10)
    Found solution after 5 iterations.
    1.618033988749989
    '''
    xn = x0
    for n in range(0,max_iter):
        fxn = f(xn)
        if abs(fxn) < epsilon:
            print('Found solution after',n,'iterations.')
            return xn
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('Exceeded maximum iterations. No solution found.')
    return None


```

Source

Lecture Code: Newton Solver, we try to find x such that $f(x) = 0$.

```
# Newton Solver
def our_newton_solver(funcname,startvalue,arglist):
    ''' Parameters:
            funcname = Function to optmize
            startvalue = Value to start the resesrch of optimal value
```

```
            arglist = optimal values for the function
        Returns:
            Optimal value for which the function is solved'''
    current=startvalue
    fval = funcname(current,arglist)
    grad = (funcname(current+0.5*1e-5,arglist)-funcname(current
-0.5*1e-5,arglist))*1e+5
    while (abs(fval)>1e-8):
        current = current - fval/grad
        fval = funcname(current,arglist)
        grad = (funcname(current+0.5*1e-5,arglist)-funcname(
current-0.5*1e-5,arglist))*1e+5
    return current
```

Lecture Code: Newton Maximizer, we try to find x such that $f'(x) = 0$.

```
    # Newton Maximizer
def our_newton_maximizer(funcname,startvalue,arglist):
    ''' Parameters:
            funcname = Function to optmize
            startvalue = Value to start the resesrch of optimal
value
            arglist = optimal values for the function
        Returns:
            Optimal value for which the function is maximized'''
    current=startvalue
    fval = funcname(current,arglist)
    grad = (funcname(current+0.5*1e-5,arglist)-funcname(current
-0.5*1e-5,arglist))*1e+5
    secgrad1 = (funcname(current+0.5*1e-5+0.5*1e-5,arglist)-
funcname(current-0.5*1e-5+0.5*1e-5,arglist))*1e+5
    secgrad2 = (funcname(current+0.5*1e-5-0.5*1e-5,arglist)-
funcname(current-0.5*1e-5-0.5*1e-5,arglist))*1e+5
    secderiv = (secgrad1-secgrad2)*1e+5
    while (abs(grad)>1e-8):
        current = current - grad/secderiv
        fval = funcname(current,arglist)
        grad = (funcname(current+0.5*1e-5,arglist)-funcname(
current-0.5*1e-5,arglist))*1e+5
        secgrad1 = (funcname(current+0.5*1e-5+0.5*1e-5,arglist)-
funcname(current-0.5*1e-5+0.5*1e-5,arglist))*1e+5
        secgrad2 = (funcname(current+0.5*1e-5-0.5*1e-5,arglist)-
funcname(current-0.5*1e-5-0.5*1e-5,arglist))*1e+5
        secderiv = (secgrad1-secgrad2)*1e+5

```

```
    return current

```

Utility Functions

There are several classes of utility functions that are frequently used to generate demand functions.

- One of the most common is the Cobb-Douglas utility function, which has the form
$$u(x,y) = x^a y^{1-a} \text{ with } a \in [0,1]$$
- Another common form for utility is the Constant Elasticity of Substitution (CES) utility function. This function has the form
$$u(x,y) = (ax^r + by^r)^{1/r}$$
- A third common utility function is quadratic, which has the form
$$u(x,y) = 2ax - (b-y)^2$$

Cobb-Douglas Utility Function

Constant Elasticity of Substitution (CES)

The constant elasticity of substitution applied to utility can use the formula

$$u(x,y) = (ax^r + by^r)^{1/r} \text{ where } -\infty < r < 1 \text{ and } r \neq 0$$

Marginal rate of substitution (MRS) is computed by

$$MRS = -\frac{a}{b}\left(\frac{x}{y}\right)^{r-1}$$

The demand functions are computed

$$x(p_x, p_y, I) = \frac{p_x^{1/(r-1)}}{p_x^{r(r-1)} + p_y^{r(r-1)}} \cdot I$$
$$y(p_x, p_y, I) = \frac{p_y^{1/(r-1)}}{p_x^{r(r-1)} + p_y^{r(r-1)}} \cdot I$$

where (p_x, p_y, I) are price of good x, price of good y and income.
Conquences of variations of r:

- If $r \to 0$ then $u(x,y) \to$ Cobb Douglas Utility Function
$$u(x,y) = x^a y^{1-a}$$
- If $r \to -\infty$ then $u(x,y) \to$ Leontief utility Function (inputs are perfect complements)
$$u(x,y) = Min(ay, bx)$$
- If $r \to 1$ then $u(x,y) \to$ Linear Production (inputs are perfect substitutes)
$$u(x,y) = ay + bx$$

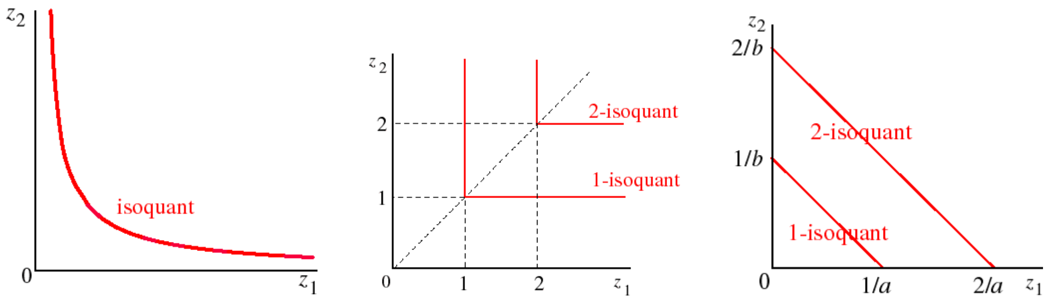

Figure 1: Utility Curves

Quasilinear Utility Functions

Utility function that is independent of the income effect.

$$u(x,y) = v(x) + y$$

where v is an arbitrary function that is strictly increasing if good x is desired. Indifference curve for α utility level:

$$v(x) + y = \alpha$$
$$y = \alpha - v(x)$$

Marginal rate of substitution is computed by

$$MRS = \frac{\partial u}{\partial x} / \frac{\partial u}{\partial y}$$
$$\text{where } \frac{\partial u}{\partial x} = v'(x) \text{ and } \frac{\partial u}{\partial y} = 1$$
$$\text{Therefore } MRS = v'(x)$$

Slutsky decomposition: Income and substitution effects

Slutsky decomposition is the total effect of substitution and income.

Normal Goods

Normal goods are goods for which demand increases when income increases. In the slutsky decomposition, the income and substitution effect reinforce each other when the the good's price change.
Income and substitution effects cause an increase in demand when prices decrease.

Income Inferior Goods

Demand reduced with higher income.
Substitution and Income effects oppose each other. When income increases, demand decreases, the substitution effect is the same as normal goods.

Griffon Goods

Extreme income-inferiority, the income effect may be larger in size than the substitution effect, causing quantity demanded demanded to fall as own-prices rises.
Slutsky's decomposition of the effect of a price change into a pure substitution effect and an income effect thus explains why the law of downwards-slopping demand is violated for extremely income-inferior goods.