QF627 Programming and Computational Finance

S0608: Scientific Tools in Python and MATLAB

(part 1)

**Learning Outcomes:**

1. (Python) **scipy.optimize.fsolve(func,x0)** returns  all the roots /  one root of the (non-linear) equations defined by **func(x)=0** given a starting estimate **x0**.
2. (MATLAB) **fzero(fun,x0)** returns  all the roots /  one root of a nonlinear function. **x0** can be specified as a real scalar or a 2-element real vector.
3. (MATLAB) **fsolve(fun, x0)** returns  all the roots /  one root of a system of nonlinear equations , where is a function that returns a vector value and is a vector or a matrix. **x0** is the initial point, specified as a real vector or real array.
4. True /  False To solve ,
5. Write the equation into format.
6. Define function .

|  |  |
| --- | --- |
| **Python** | **MATLAB** |
| Define **f01** with a **lambda** expression: | Define **f01** with an anonymous function: |
| **f01=lambda x: x\*\*2-2** | **f01=@(x) x^2-2;** |
| Define **f02** with a user-define function: | Define **f02** with a user-define function: |
| **def f02(x):**  **return x\*\*2-2** | **function y=f02(x)**  **y=x^2-2;**  **end** |

1. Apply the library function.

|  |  |
| --- | --- |
| **Python** | **MATLAB** |
| **fsolve with f01** | **fsolve with f01** |
| **x=fsolve(f01, 1)** | **x=fsolve(f01, 1)** |
| **fsolve with f02** | **fsolve with f02** |
| **x=fsolve(f02, 1)** | **x=fsolve(@f02, 1)** |

1. True /  False (Python) For a one-variable equation, **fsolve** returns the solution in a list.
2. (Python) Define **myfzero** (using **fsolve**) to solve a nonlinear equation and return the solution as a scalar.

**from scipy.optimize import fsolve**

|  |
| --- |
| **def myfzero(func, x0):**  **return fsolve(func, x0)[0]** |

1. How to obtain the negative root of the equation ?

|  |  |
| --- | --- |
| Method 1: Choose a proper initial value | |
| **Python (fsolve + f02)** | **MATLAB (fzero/fsolve + f02)** |
| **x=fsolve(f02,-1)[0]** | **x=fsolve(@f02,-1) or**  **x=fzero(@f02,-1)** |
| **Python (bisect + f02)** | **MATLAB (fzero + f02)** |
| **from scipy.optimize import bisect**  **x=bisect(f02, -10, 0)** | **x=fzero(@f02,[-10, 0])** |

1. True /  False Both in Python and MATLAB, we can define functions that return a function/functions.
2. Python versus MATLAB

|  |  |
| --- | --- |
| **Python** | **MATLAB** |
| i=3  f=lambda x: i\*x  i=5  **print(f(5))**  i=7  **print(f(5))** | i=3;  f=@(x) i\*x;  i=5  **print(f(5))**  i=7  **print(f(5))** |
| **Output** | **Output** |
| **25**  **35** | **15**  **15** |

1. Example 2: Given , solve for .

|  |  |  |
| --- | --- | --- |
| **Python** | **MATLAB** | |
| **def fxab(x, a, b):**  **return a\*x\*\*3+b** | **function y=fxab(x, a, b)**  **y=a\*x^3+b;**  **end** | |
| **Use fsolve and fxab with args** | N.A. | |
| **x=fsolve(fxab, 1, args=(2, 3))[0]** |
| **Python (Define a new function)** | | **Python (Define a new function)** |
| **a=2**  **b=3**  **x=fsolve(     ,1)[0]** | | **a=2;**  **b=3;**  **x=fzero(     ,1)** |

1. How to compute ?

|  |  |
| --- | --- |
| **Python (using \*\*)** | **MATLAB (using ^)** |
|  |  |
| **Python (using mynthroot)** | **MATLAB (using nthroot)** |
| **def mynthroot(x, n):**  **if n%2 ==1 and x<0:**  **return -(-x)\*\*(1/n)**  **else:**  **return (x)\*\*(1/n)**  **mynthroot(-3/2, 3)** | **nthroot(-3/2, 3)** |

1. (Python) Implied volatility

|  |
| --- |
| **from scipy.stats import norm**  **from math import log, sqrt, exp**  **def BS\_EuroCall(S,K,r,q,sigma,T):**  **d1=(log(S/K)+(r-q+sigma\*\*2/2.)\*T)/(sigma\*sqrt(T))**  **d2=d1-sigma\*sqrt(T)**  **c=S\*exp(-q\*T)\*norm.cdf(d1)-K\*exp(-r\*T)\*norm.cdf(d2)**  **return c** |
| Example 2: Given , , , , and , use function scipy.optimize.fsolve / **myfzero** and **BS\_EuroCall** to solve the Black-Scholes equation for the implied volatility (). |
| **S=490**  **K=470**  **r=0.033**  **q=0**  **T=0.08**  **c=24.5941**  **sigma\_imp=fsolve(BS\_EuroCall, )**  **print(sigma\_imp)** |
| **sigma\_imp=myfzero(     )**  **print(sigma\_imp)** |

1. (MATLAB) Implied volatility

|  |
| --- |
| **BS\_EuroCall.m** |
| **function c=BS\_EuroCall(S,K,r,q,sigma,T)**  **d1=(log(S/K)+(r-q+sigma^2/2)\*T)/(sigma\*sqrt(T));**  **d2=d1-sigma\*sqrt(T);**  **c=S\*exp(-q\*T)\*normcdf(d1)-K\*exp(-r\*T)\*normcdf(d2);**  **end** |
| Example 2: Given , , , , and , use function fzero to solve the Black-Scholes equation for the implied volatility (). |
| **S=     ;**  **K=     ;**  **r=     ;**  **q=     ;**  **T=     ;**  **c=     ;**  **sigma\_imp=fzero(     )** |

1. (Python) **mypartial**

|  |
| --- |
| **def mypartial(fun, \*args, \*\*kwargs):**  **def f(x):**  **return fun(\*args, x, \*\*kwargs)**  **return f**  **S=490**  **K=470**  **r=0.033**  **q=0**  **T=0.08**  **BS3=mypartial(BS\_EuroCall, S, K, r, q, T=T)**  **c=BS3(0.2)**  **sigma\_imp=fsolve(lambda x:BS3(x)-c, 0.5)** |

1. (MATLAB) Example 4: Compute implied volatility for every row in **data**.

|  |
| --- |
| 1. Use **readtable** to get the data from file **dataset01.csv**. Name the data imported as **data**. |
| **data=readtable('dataset01.csv');**  **disp(data)** |
| 2. Compute implied volatility for each row, and put the result in a new column with the column name **ImpVol**. |
|  |
| 3. Write **data** to a CSV file **output.csv**. |
| **writetable(data,'output.csv');** |

1. (Python) Compute implied volatility for every row in **data**.

**import pandas as pd**

**from scipy.optimize import fsolve**

|  |
| --- |
| 1. Load data from **dataset01.csv** and name the data imported as **data**. |
| **data=pd.read\_csv('dataset01.csv');** |
| 2. Compute implied volatility for each row, and put the result in a new column with the column name **ImpVol**. |
|  |
| 3. Write **data** to a CSV file **output.csv**. |
| **data.to\_csv('output.csv');** |

1. True /  False To solve the following system of nonlinear equations:
2. Write the equation into a new format.
3. Define function .

|  |  |
| --- | --- |
| **Python** | **MATLAB** |
| Define **f03** with a user-define function: | Define **f03** with a user-define function: |
| **from math import cos**  **def f03(x):**  **y = [0, 0]**  **y[0]=x[0]\*cos(x[1])-4**  **y[1]=x[0]\*x[1]-x[1]-5**  **return y** | **function y=f03(x)**        **end** |

1. Apply the library function.

|  |  |
| --- | --- |
| **Python** | **MATLAB** |
| **fsolve with f03** | **fsolve with f03** |
| **x=fsolve(     ,      )** | **x=fsolve(     ,      )** |

1. (Python) **scipy.optimize.minimize** is for the minimization of scalar function of one or more variables. In general, the optimization problems are of the form:

**f(x)** subject to

g\_i(x) **0, i = 1,…,m**

h\_j(x) **0, j = 1,…,p**

1. (MATLAB) **fmincon** finds minimum of constrained nonlinear multivariable function. The problem is specified by

**f(x)** subject to

**c(x)       0**

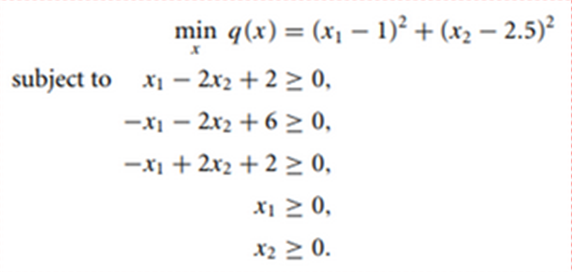
**ceq(x)       0**

**A\*x       b**

**Aeq\*x       beq**

**lb<=x<=ub**

1. Example 6: Optimization with constraints

****

|  |
| --- |
| **Python** |
| **from scipy.optimize import minimize**  **fun = lambda x:**  **cons = ({'type': '     ', 'fun': lambda x:      },**  **{'type': '     ', 'fun': lambda x:      },**  **{'type': '     ', 'fun': lambda x:      })**  **bnds = ((0, None), (0, None))**  **res = minimize(fun, (2, 0), bounds=bnds, constraints=cons)**  **print(res.x)** |
| **MATLAB** |
| **fun=@(x)      ;**  **A=     ;**  **b=     ;**  **Aeq=     ;**  **beq=     ;**  **lb=     ;**  **ub=     ;**  **x0=     ;**  **x= fmincon(     ,     ,     ,     ,     ,     ,     ,     )** |

1. (MATLAB) **quadprog** solves quadratic objective functions with linear constraints. It finds a minimum for a problem specified by

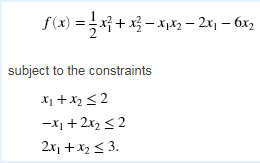
**(1/2)\*(x')\*H\*x + (f')\*x** subject to

**A\*x       b**

**Aeq\*x       beq**

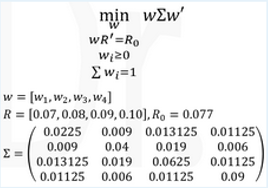
**lb<=x<=ub**

1. Example 7: Quadratic Programming with linear constraints

****

|  |
| --- |
| **Python** |
| **from scipy.optimize import minimize**  **fun = lambda x:**  **cons = ({'type': '     ', 'fun': lambda x:      },**  **{'type': '     ', 'fun': lambda x:      },**  **{'type': '     ', 'fun': lambda x:      })**  **bnds = ((None, None), (None, None))**  **res = minimize(fun, (0, 0), bounds=bnds, constraints=cons)**  **print(res.x)** |
| **MATLAB (using quadprog)** |
| **H=     ;**  **f=     ;**  **A=     ;**  **b=     ;**  **x=quadprog(     ,     ,     ,     )** |

1. Example 8: Minimum-Variance Portfolio

****

|  |
| --- |
| **Python (using np.matrix)** |
| **from scipy.optimize import minimize**  **import numpy as np**  **R=np.matrix([0.07,0.08,0.09,0.1])**  **x0=[0.1,0.1,0.1,0.1]**  **#x0=np.array([0.1,0.1,0.1,0.1])**  **#x0=np.array([[0.1,0.1,0.1,0.1]])**  **R0=0.077**  **sigma=np.matrix([[0.0225, 0.009, 0.013125, 0.01125],**  **[0.009, 0.04, 0.019, 0.006],**  **[0.013125, 0.019, 0.0625, 0.01125],**  **[0.01125, 0.006, 0.01125, 0.09]])**  **def fun(w, sigma):**  **Mw=np.matrix(w)**  **return (Mw     sigma     (Mw.T))**  **cons = ({'type': 'eq', 'fun': lambda x: (np.matrix(x)     (R.T))     -R0},**  **{'type': 'eq', 'fun': lambda x: np.sum(x)-1.0})**  **bnds = ((0, 1), )\*4**  **res = minimize(fun, x0, args=sigma, bounds=bnds, constraints=cons)**  **res.x** |
| **Python (using np.array and a 2D array for R)** |
| **from scipy.optimize import minimize**  **import numpy as np**  **R=np.array([[0.07,0.08,0.09,0.1]])**  **x0=[0.1,0.1,0.1,0.1]**  **#x0=np.array([0.1,0.1,0.1,0.1])**  **#x0=np.array([[0.1,0.1,0.1,0.1]])**  **R0=0.077**  **sigma=np.array([[0.0225, 0.009, 0.013125, 0.01125],**  **[0.009, 0.04, 0.019, 0.006],**  **[0.013125, 0.019, 0.0625, 0.01125],**  **[0.01125, 0.006, 0.01125, 0.09]])**  **def fun(w, sigma):**  **return (w     sigma     (w.T))**  **cons = ({'type': 'eq', 'fun': lambda x: (x     (R.T))     -R0},**  **{'type': 'eq', 'fun': lambda x: np.sum(x)-1.0})**  **bnds = ((0, 1), )\*4**  **res = minimize(fun, x0, args=sigma, bounds=bnds, constraints=cons)**  **res.x** |
| **Python (using np.array and a 1D array for R)** |
| **from scipy.optimize import minimize**  **import numpy as np**  **R=np.array([0.07,0.08,0.09,0.1])**  **x0=[0.1,0.1,0.1,0.1]**  **#x0=np.array([0.1,0.1,0.1,0.1])**  **#x0=np.array([[0.1,0.1,0.1,0.1]])**  **R0=0.077**  **sigma=np.array([[0.0225, 0.009, 0.013125, 0.01125],**  **[0.009, 0.04, 0.019, 0.006],**  **[0.013125, 0.019, 0.0625, 0.01125],**  **[0.01125, 0.006, 0.01125, 0.09]])**  **def fun(w, sigma):**  **return (w     sigma     (w.T))**  **cons = ({'type': 'eq', 'fun': lambda x: (x     (R.T))     -R0},**  **{'type': 'eq', 'fun': lambda x: np.sum(x)-1.0})**  **bnds = ((0, 1), )\*4**  **res = minimize(fun, x0, args=sigma, bounds=bnds, constraints=cons)**  **res.x** |

1. (to be continued)