QF627 Programming and Computational Finance

S0608: Scientific Tools in Python and MATLAB

(part 2)

1. True /  False (Python) **scipy.misc.derivative(func,x0,dx,n)** computes the **n**th derivative of **func** at **x0** with spacing **dx**. When **n=1**, it computes

**(func(x0+dx)-func(x0-dx))/(2\*dx)**

1. True /  False (MATLAB) diff can be used to approximate derivatives with the syntax **diff(f)/h**.
2. (Python) Use **scipy.misc.derivative** and **BS\_EuroCallV** to approximate Delta and Vega for every row in **data**.

|  |  |
| --- | --- |
| **import pandas as pd**  **from scipy.stats import norm**  **from scipy.misc import derivative**  **from math import log, sqrt, exp**  **def BS\_EuroCallV(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**  **data=pd.read\_csv('dataset01.csv',header=0)** |  |
| **Use one command** to apply **scipy.misc.derivative** and **BS\_EuroCallV** to each row of **data** to compute Delta and save the result to a new column in **data** and name this column **Delta**. | |
| **data['Delta']=data[['S','K','r','q','sigma','T']].apply(**  **lambda x: derivative(lambda s: BS\_EuroCallV(     ),     ,dx=0.01), axis=1)** | |
| **Use one command** to apply **scipy.misc.derivative** and **BS\_EuroCallV** to each row of **data** to compute Vega and save the result to a new column in **data** and name this column **Vega**. | |
| **data['Vega']=data[['S','K','r','q','sigma','T']].apply(**  **lambda x: derivative(lambda s: BS\_EuroCallV(     ),     ,dx=0.01), axis=1)** | |

1. (Python) **scipy.integrate.quad(fun, a, b)** computes the definite integral . For example, to compute , we can use

|  |
| --- |
| **from scipy import integrate**  **integrate.quad(lambda x:      ,      ,      )** |

1. (MATLAB) **integral(fun, xmin, xmax)** numerically integrates function **fun** from **xmin** to **xmax** using global adaptive quadrature and default error tolerances. For example, to compute , we can use

|  |
| --- |
| **integral(@(x):      ,      ,      )** |

1. Compute .

|  |
| --- |
| **Python** |
| **from scipy import integrate**  **import numpy as np**  **func=lambda x:**  **integrate.quad(func,      ,      )** |
| **MATLAB** |
| **fun=@(x):      ;**  **integral(fun,      ,      )** |

1. In Python, **scipy.integrate.dblquad(fun, a, b, gfun, hfun)** computes the definite integral . In MATLAB, **integral2(fun, xmin, xmax, ymin, ymax)** approximates the integral of the function **z=fun(x,y)** over the planar region **xminxxmax** and **ymin(x)yymax(x)**. For example, to compute , we can use

|  |
| --- |
| **Python** |
| **from scipy import integrate**  **func=lambda x, y:**  **integrate.dblquad(func,      ,      ,      ,      )** |
| **MATLAB** |
| **fun=@(x,y):      ;**  **integral(fun,      ,      ,      ,      )** |

1. True /  False (Python) **scipy.interpolate.interp1d** is a class which produces callable instances. The **\_\_init\_\_** method has compulsory arguments **x** and **y** and the default interpolation method is the **linear** interpolation. To use cubic spline interpolation, use **kind=**. **x** and **y** are two of the instance attributes.
2. (Python) Define **class myinterp1d**.

|  |
| --- |
| **import matplotlib.pyplot as plt**  **import numpy as np**  **class myinterp1d(object):**  **def \_\_init\_\_(self, x, y):**  **self.x=x**  **self.y=y**  **def \_\_call\_\_(self, xnew):**  **ynew=**  **for x0 in xnew:**  **if x0<=self.x[0]:**  **ynew.append(     )**  **elif x0>=self.x[-1]:**  **ynew.append(     )**  **else:**  **hi=next(filter(lambda x:      , enumerate(self.x)))[0]**  **lo=hi-1**  **m=**  **ynew.append(     )**  **ynew=np.array(ynew)**  **return ynew**  **x = np.arange(0, 10)**  **y = np.exp(-x/3.0)**  **f = myinterp1d(x, y)**  **xnew = np.arange(0, 9, 0.1)**  **ynew = f(xnew)**  **plt.plot(x, y, 'o', xnew, ynew, '-')**  **plt.show()** |

1. Fix the bug.

|  |
| --- |
| **import numpy as np**  **import matplotlib.pyplot as plt**  **class myinterp1d(object):**  **def \_\_init\_\_(self, x, y):**  **self.x=x**  **self.y=y**  **self.nInt=len(x)-1 #number of intervals**  **self.f=self.\_linear()**    **def \_linear(self):**  **f=[]**  **for i in range(self.nInt):**  **m=(y[i+1]-y[i])/(x[i+1]-x[i])**  **b=y[i]-m\*x[i]**  **#f.append(lambda x: m\*x+b)**    **return f**  **def \_linear\_interp(self, x0):**  **if x0<=self.x[0]:**  **return y[0]**  **elif x0>=self.x[-1]:**  **return y[-1]**  **else:**  **i=next(filter(lambda s: x0<s[1], enumerate(self.x)))[0]**  **return self.f[i-1](x0)**    **def \_\_call\_\_(self, xnew):**  **return np.array([self.\_linear\_interp(x0) for x0 in xnew])**  **x = np.arange(0, 10)**  **y = np.exp(-x/3.0)**  **f = myinterp1d(x, y)**  **xnew = np.arange(0, 9, 0.1)**  **ynew = f(xnew) # use interpolation function returned by `interp1d`**  **plt.plot(x, y, 'o', xnew, ynew, '-')**  **plt.show()** |

1. True /  False (Python) **numpy.polyfit(x,y,deg)** fits a polynomial of degree **deg** to points (**x**, **y**). It returns a vector of coefficients **p**. **numpy.polyval(p,x)** evaluates a polynomial at **x**. When **deg** is greater than **len(x)-1**, the polynomial can be used as an interpolation function.
2. True /  False (MATLAB) **interp1(x,v,xq,method)** returns interpolated values of a 1-D function at specific query points **xq** using **method**. Vector **x** contains the sample points, and **v** contains the corresponding values. When **method** is **spline**, cubic spline interpolation with not-a-knot end conditions is used.
3. True /  False (MATLAB) **polyfit(x,y,n)** and **polyval(p,x)** are similar to the Python functions **numpy.polyfit** and **numpy.polyval**.