Python 8: SciPy

금융공학 프로그래밍 I

Intro to SciPy

- What is SciPy?
  - Common tools for scientific programming
    - Linear algebra
    - Numeric integration
    - Interpolation
    - Optimization
    - Distributions and random number generation
    - Signal processing, etc.
  - Industry-standard Fortran libraries such as LAPACK, BLAS, etc.

SciPy Package

The majority of SciPy's functionality resides in its subpackages

- `scipy.optimize`, `scipy.integrate`, `scipy.stats`, etc.

```
import scipy.optimize
from scipy.integrate import quad
```

- Although SciPy imports NumPy, the standard approach is to start scientific programs with

```
import numpy as np
```

Linear Algebra

- scipy.linalg
  - Matrix inverse & multiplication

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
array([[1, 2],
       [3, 4]])
>>> linalg.inv(A)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> A.dot(linalg.inv(A)) #double check
array([[  1.00000000e+00,   0.00000000e+00],
       [  4.44089210e-16,   1.00000000e+00]])
```

Linear Algebra

- Solving linear system
  - For example,

$$x + 3y + 5z = 10$$
$$2x + 5y + z = 8$$
$$2x + 3y + 8z = 3$$

```
>>> import numpy as np
>>> from scipy import linalg
>>> A = np.array([[1,2],[3,4]])
>>> A
array([[1, 2],
       [3, 4]])
>>> b = np.array([[5],[6]])
>>> b
array([[5],
       [6]])
>>> linalg.inv(A).dot(b) #slow
array([[-4. ],
       [ 4.5]])
>>> A.dot(linalg.inv(A).dot(b))-b #check
array([[  8.88178420e-16],
       [  2.66453526e-15]])
>>> np.linalg.solve(A,b) #fast
array([[-4. ],
       [ 4.5]])
>>> A.dot(np.linalg.solve(A,b))-b #check
array([[ 0.],
       [ 0.]])
```

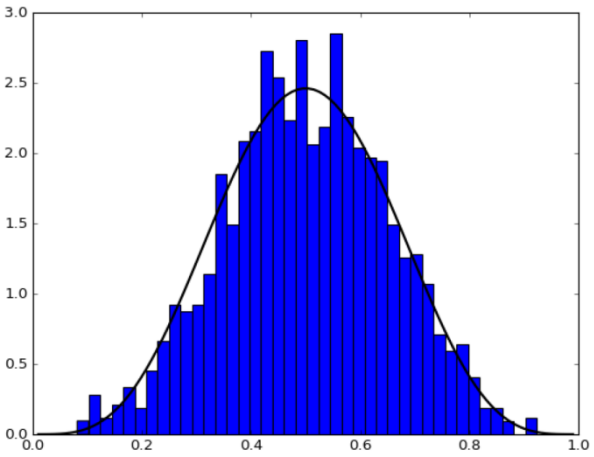
Linear Algebra

- Determinant
  - linalg.det(A)
- Eigenvalue & eigenvector
  - lambda, v = linalg.eig(A)
- Singular value decomposition
  - U, s, Vh = linalg.svd(A)
- LU decomposition
- Cholesky decomposion
- QR decomposition
- Schur decomposition

Statistics

- scipy.stats
  - Random variable objects (distributions, random sampling …)
  - Estimation procedures
  - Statistical tests

```
import numpy as np
from scipy.stats import beta
from matplotlib.pyplot import hist, plot, show
q = beta(5, 5)       # Beta(a, b), with a = b = 5
obs = q.rvs(2000)    # 2000 observations
hist(obs, bins=40, normed=True)
grid = np.linspace(0.01, 0.99, 100)
plot(grid, q.pdf(grid), 'k-', linewidth=2)
show()
```



Statistics

- methods

```
In [14]: q.cdf(0.4)      # Cumulative distribution function
Out[14]: 0.26656768000000002

In [15]: q.pdf(0.4)      # Density function
Out[15]: 2.0901888000000004

In [16]: q.ppf(0.8)      # Quantile (inverse cdf) function
Out[16]: 0.63391348346427079

In [17]: q.mean()
Out[17]: 0.5
```

rvs: Random Variates			
pdf: Probability Density Function			
cdf: Cumulative Distribution Function			
sf: Survival Function (1-CDF)			
ppf: Percent Point Function (Inverse of CDF)			
isf: Inverse Survival Function (Inverse of SF)			
moment: non-central moments of the distribution			

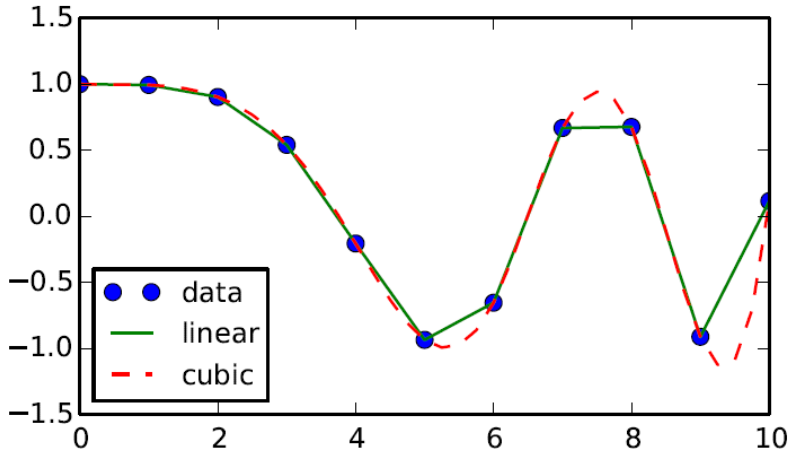
Linear Regression

❖ linregress

```
In [19]: from scipy.stats import linregress

In [20]: x = np.random.randn(200)

In [21]: y = 2 * x + 0.1 * np.random.randn(200)

In [22]: gradient, intercept, r_value, p_value, std_err = linregress(x, y)

In [23]: gradient, intercept
Out[23]: (1.9962554379482236, 0.008172822032671799)
```

1-D Interpolation

```
from scipy.interpolate import interp1d

x = np.linspace(0, 10, num=11, endpoint=True)
y = np.cos(-x**2/9.0)
f = interp1d(x, y)
f2 = interp1d(x, y, kind='cubic')

xnew = np.linspace(0, 10, num=41, endpoint=True)
import matplotlib.pyplot as plt
plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')
plt.legend(['data', 'linear', 'cubic'], loc='best')
plt.show()
```

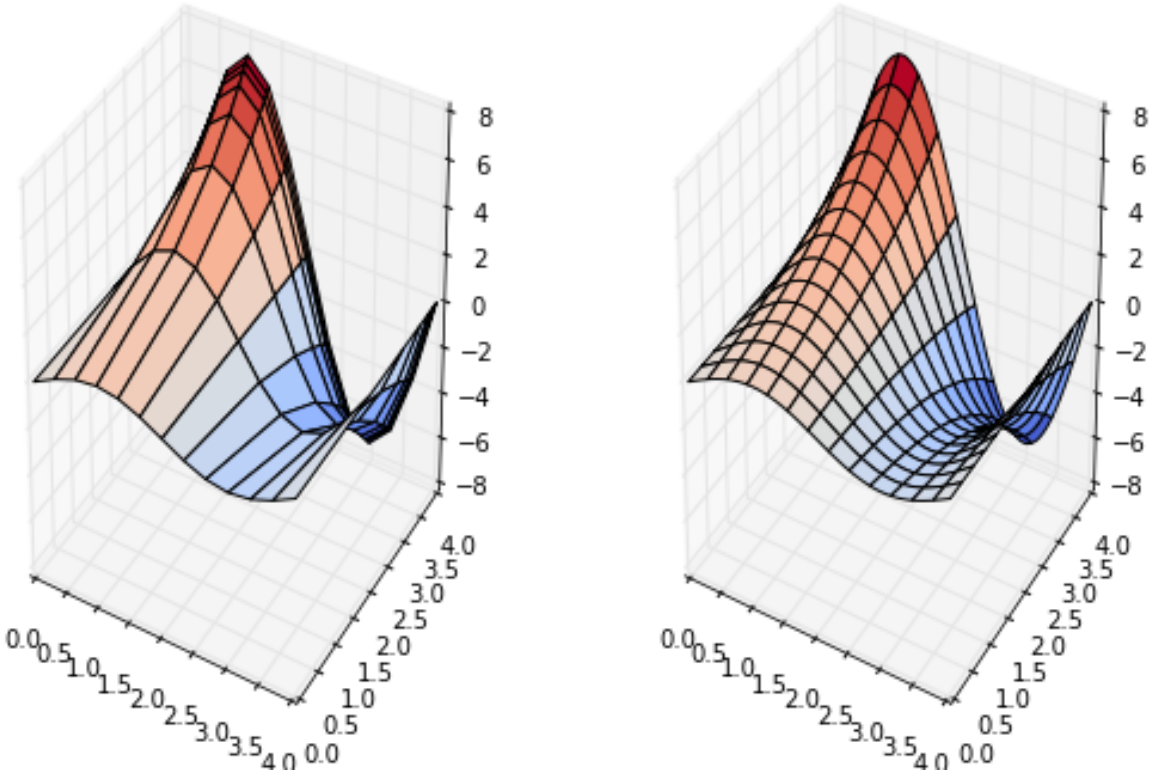


2-D Interpolation

```
import numpy as np
from scipy.interpolate import interp2d
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d.axes3d import Axes3D

x = np.linspace(0, 4, 13)
y = np.array([0, 2, 3, 3.5, 3.75, 3.875, 3.9375, 4])
X, Y = np.meshgrid(x, y)
Z = np.sin(np.pi*X/2) * np.exp(Y/2)

x2 = np.linspace(0, 4, 65)
y2 = np.linspace(0, 4, 65)
f = interp2d(x, y, Z, kind='cubic')
Z2 = f(x2, y2)

fig = plt.figure(figsize=(9,6))
ax0 = fig.add_subplot(121, projection='3d')
ax1 = fig.add_subplot(122, projection='3d')

ax0.plot_surface(X, Y, Z, cstride=1, rstride=1, cmap=cm.coolwarm)

X2, Y2 = np.meshgrid(x2, y2)
ax1.plot_surface(X2, Y2, Z2, cstride=5, rstride=5, cmap=cm.coolwarm)

fig.suptitle(r"$z(x,y) = \sin\left(\frac{\pi x}{2}\right)e^{y/2}$", fontsize=20)
fig.show()
```

2-D Interpolation

$$z(x, y) = \sin\left(\frac{\pi x}{2}\right) e^{y/2}$$



Optimization

- The module contains:
  - Unconstrained and constrained minimization of multivariate scalar functions (minimize) using a variety of algorithms (e.g. BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP)
  - Global (brute-force) optimization routines (e.g. basinhopping, differential_evolution)
  - Least-squares minimization (leastsq) and curve fitting (curve_fit) algorithms
  - Scalar univariate functions minimizers (minimize_scalar) and root finders (newton)
  - Multivariate equation system solvers (root) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov).

scipy.optimize.minimize

❖ Minimizing Rosenbrock function:

$$f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$$

```
>>> import numpy as np
>>> from scipy.optimize import minimize

>>> def rosen(x):
...     """The Rosenbrock function"""
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)

>>> x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
>>> res = minimize(rosen, x0, method='nelder-mead',
...                options={'xtol': 1e-8, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571

>>> print(res.x)
[ 1.  1.  1.  1.  1.]
```

❖ OptimizaResult

x	(ndarray) The solution of the optimization.
success	(bool) Whether or not the optimizer exited successfully.
status	(int) Termination status of the optimizer. Its value depends on the underlying solver. Refer to *message* for details.
message	(str) Description of the cause of the termination.
fun, jac, hess, hess_inv	(ndarray) Values of objective function, Jacobian, Hessian or its inverse (if available). The Hessians may be approximations, see the documentation of the function in question.
nfev, njev, nhev	(int) Number of evaluations of the objective functions and of its Jacobian and Hessian.
nit	(int) Number of iterations performed by the optimizer.
maxcv	(float) The maximum constraint violation.

Constrained minimization

$$\begin{aligned} &\min F(x) \\ &\text{subject to} \quad C_j(X) = 0, \quad j = 1, ..., \text{MEQ} \\ &\qquad\qquad C_j(x) \geq 0, \quad j = \text{MEQ} + 1, ..., M \\ &\qquad\qquad XL \leq x \leq XU, \quad I = 1, ..., N. \end{aligned}$$

Max. $f(x, y) = 2xy + 2x - x^2 - 2y^2$

s.t. $x^3 - y = 0$

$y - 1 \geq 0$

```
>>> def func(x, sign=1.0):
...     """ Objective function """
...     return sign*(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)

>>> def func_deriv(x, sign=1.0):
...     """ Derivative of objective function """
...     dfdx0 = sign*(-2*x[0] + 2*x[1] + 2)
...     dfdx1 = sign*(2*x[0] - 4*x[1])
...     return np.array([ dfdx0, dfdx1 ])
```

Constrained minimization

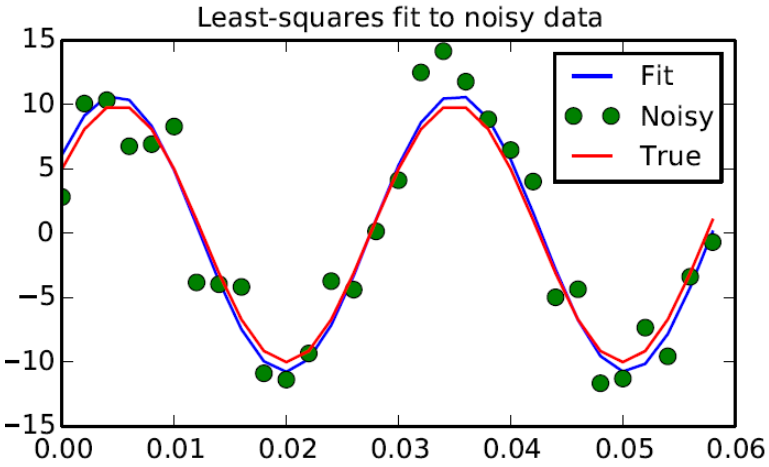
```
>>> cons = ({'type': 'eq',
...           'fun' : lambda x: np.array([x[0]**3 - x[1]]),
...           'jac' : lambda x: np.array([3.0*(x[0]**2.0), -1.0])},
...          {'type': 'ineq',
...           'fun' : lambda x: np.array([x[1] - 1]),
...           'jac' : lambda x: np.array([0.0, 1.0])})
```

```
>>> res = minimize(func, [-1.0,1.0], args=(-1.0,), jac=func_deriv,
...                constraints=cons, method='SLSQP', options={'disp': True})
Optimization terminated successfully.    (Exit mode 0)
            Current function value: -1.00000018311
            Iterations: 9
            Function evaluations: 14
            Gradient evaluations: 9
>>> print(res.x)
[ 1.00000009  1.        ]
```

Least-square fitting

$$e_i(\mathbf{p}, \mathbf{y}_i, \mathbf{x}_i) = \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i, \mathbf{p})\|$$

Min. $J(\mathbf{p}) = \sum_{i=0}^{N-1} e_i^2(\mathbf{p})$.



$$y_i = A\sin(2\pi k x_i + \theta)$$

where the parameters A, k, and θ are unknown. The residual vector is

$$e_i = |y_i - A\sin(2\pi k x_i + \theta)|.$$

```
>>> from numpy import arange, sin, pi, random, array
>>> x = arange(0, 6e-2, 6e-2 / 30)
>>> A, k, theta = 10, 1.0 / 3e-2, pi / 6
>>> y_true = A * sin(2 * pi * k * x + theta)
>>> y_meas = y_true + 2*random.randn(len(x))

>>> def residuals(p, y, x):
...     A, k, theta = p
...     err = y - A * sin(2 * pi * k * x + theta)
...     return err
```

Root Finding

```
>>> import numpy as np
>>> from scipy.optimize import root
>>> def func(x):
...     return x + 2 * np.cos(x)
>>> sol = root(func, 0.3)
>>> sol.x
array([-1.02986653])
>>> sol.fun
array([ -6.66133815e-16])
```

$$\begin{aligned} x_0 \cos(x_1) &= 4, \\ x_0 x_1 - x_1 &= 5. \end{aligned}$$

- Method:
  - hybr (default): hybrid method of Powell
  - lm: Levenberg-Marquanrdt

```
>>> def func2(x):
...     f = [x[0] * np.cos(x[1]) - 4,
...          x[1]*x[0] - x[1] - 5]
...     df = np.array([[np.cos(x[1]), -x[0] * np.sin(x[1])],
...                    [x[1], x[0] - 1]])
...     return f, df
>>> sol = root(func2, [1, 1], jac=True, method='lm')
>>> sol.x
array([ 6.50409711,  0.90841421])
```

Q & A

khhwang78@gmail.com