

# Lec 07 - SciPy

**Statistical Computing and Computation**

**Sta 663 | Spring 2022**

**Dr. Colin Rundel**

# What is SciPy

Fundamental algorithms for scientific computing in Python

| Subpackage  | Description  | Subpackage | Description                             |
|-------------|--|------------|---|
| cluster     | Clustering algorithms                                  | odr        | Orthogonal distance regression          |
| constants   | Physical and mathematical constants                    | optimize   | Optimization and root-finding routines  |
| fftpack     | Fast Fourier Transform routines                        | signal     | Signal processing                       |
| integrate   | Integration and ordinary differential equation solvers | sparse     | Sparse matrices and associated routines |
| interpolate | Interpolation and smoothing splines                    | spatial    | Spatial data structures and algorithms  |
| io          | Input and Output                                       | special    | Special functions                       |
| linalg      | Linear algebra   | stats      | Statistical distributions and functions |
| ndimage     | N-dimensional image processing                         |            |   |

## **Example 1 - k-means clustering**

# Data

```
rng = np.random.default_rng(seed = 1234)

cl1 = rng.multivariate_normal([-2,-2], [[1,-0.5],[-0.5,1]], size=100)
cl2 = rng.multivariate_normal([1,0], [[1,0],[0,1]], size=150)
cl3 = rng.multivariate_normal([3,2], [[1,-0.7],[-0.7,1]], size=200)

pts = np.concatenate((cl1,cl2,cl3))
```

# k-means clustering

```
from scipy.cluster.vq import kmeans
```

```
ctr, dist = kmeans(pts, 3)  
ctr
```

```
## array([[ 2.85409537,  1.94511779],  
##          [ 0.89789235, -0.20527898],  
##          [-2.03956666, -1.85662027]])
```

```
dist
```

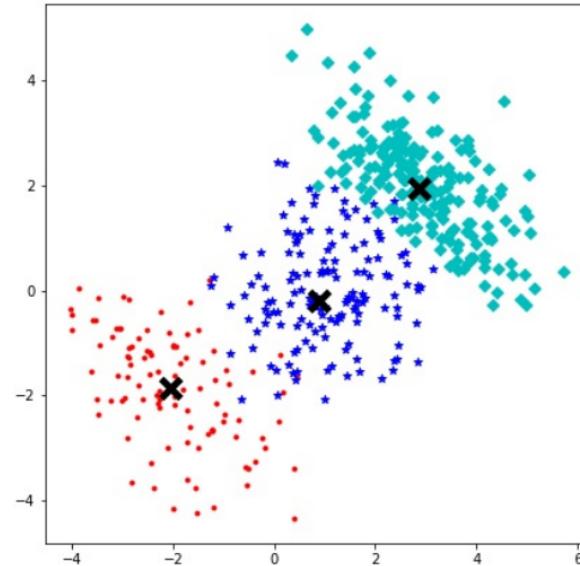
```
## 1.2206927437557962
```

```
cl1.mean(axis=0)
```

```
## array([-2.00474615, -1.87275596])
```

```
cl2.mean(axis=0)
```

```
## array([1.03849018,  0.01417119])
```



# k-means distortion plot

The mean (non-squared) Euclidean distance between the observations passed and the centroids generated.

```
ks = range(1,6)
dists = [kmeans(pts, k)[1] for k in ks]

np.array(dists).reshape((-1,1))

## array([[2.5470307 ],
##        [1.57009105],
##        [1.22069274],
##        [1.04594861],
##        [0.95269843]])
```

## Example 2 - Numerical integration

# Basic functions

For general numeric integration in 1D we use `scipy.integrate.quad()`, which takes as arguments the function to be integrated and the lower and upper bounds of integration.

```
from scipy.integrate import quad  
  
quad(lambda x: x, 0, 1)  
  
## (0.5, 5.551115123125783e-15)  
  
quad(np.sin, 0, np.pi)  
  
## (2.0, 2.220446049250313e-14)  
  
quad(np.sin, 0, 2*np.pi)  
  
## (2.0329956258200796e-16, 4.3998892617845996e-14)  
  
quad(np.exp, 0, 1)  
  
## (1.7182818284590453, 1.9076760487502457e-14)
```

# Normal PDF

The PDF for a normal distribution is given by,

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

```
def norm_pdf(x, mu, sigma):
    return (1/(sigma * np.sqrt(2*np.pi))) * np.exp(-0.5 * ((x - mu)/sigma)**2)
```

```
norm_pdf(0, 0, 1)
```

```
## 0.3989422804014327
```

```
norm_pdf(np.Inf, 0, 1)
```

```
## 0.0
```

```
norm_pdf(-np.Inf, 0, 1)
```

```
## 0.0
```

# Checking the DPF

We can check that we've implemented a valid pdf by integrating the PDF from  $-\infty$  to  $\infty$ ,

```
quad(norm_pdf, -np.inf, np.inf)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): TypeError: norm_pdf() missing 2 required positionals: 'a' and 'b'  
##  
## Detailed traceback:  
##   File "<string>", line 1, in <module>  
##   File "/opt/homebrew/lib/python3.9/site-packages/scipy/integrate/quadpack.py", line 351, in quad  
##     retval = _quad(func, a, b, args, full_output, epsabs, epsrel, limit,  
##   File "/opt/homebrew/lib/python3.9/site-packages/scipy/integrate/quadpack.py", line 465, in _quad  
##     return _quadpack._qagie(func,bound,infbounds,args,full_output,epsabs,epsrel,limit)
```

```
quad(lambda x: norm_pdf(x, 0, 1), -np.inf, np.inf)
```

```
## (0.9999999999999997, 1.0178191380347127e-08)
```

```
quad(lambda x: norm_pdf(x, 17, 12), -np.inf, np.inf)
```

```
## (1.0000000000000002, 4.113136862574909e-09)
```

# Truncated normals

$$f(x) = \begin{cases} \frac{c}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), & \text{for } a \leq x \leq b \\ 0, & \text{otherwise.} \end{cases}$$

```
def trunc_norm_pdf(x, mu=0, sigma=1, a=-np.inf, b=np.inf):
    if (b < a):
        raise ValueError("b must be greater than a")
    x = np.asarray(x).reshape(-1)
    full_pdf = (1/(sigma * np.sqrt(2*np.pi))) * np.exp(-0.5 * ((x - mu)/sigma)**2)
    full_pdf[(x < a) | (x > b)] = 0
    return full_pdf
```

# Testing trunc\_norm\_pdf

```
trunc_norm_pdf(0, a=-1, b=1)
## array([0.39894228])

trunc_norm_pdf(2, a=-1, b=1)
## array([0.])

trunc_norm_pdf(-2, a=-1, b=1)
## array([0.])

trunc_norm_pdf([-2,1,0,1,2], a=-1, b=1)
## array([0.          , 0.24197072, 0.39894228, 0.24197072, 0.        ])

quad(lambda x: trunc_norm_pdf(x, a=-1, b=1), -np.inf, np.inf)
## (0.682689492137086, 2.0147661317082566e-11)

quad(lambda x: trunc_norm_pdf(x, a=-3, b=3), -np.inf, np.inf)
## (0.9973002039367396, 7.451935936375609e-09)
```

# Fixing trunc\_norm\_pdf

```
def trunc_norm_pdf(x, μ=0, σ=1, a=-np.inf, b=np.inf):
    if (b < a):
        raise ValueError("b must be greater than a")
    x = np.asarray(x).reshape(-1)

    nc = 1 / quad(lambda x: norm_pdf(x, μ, σ), a, b)[0]

    full_pdf = nc * (1/(σ * np.sqrt(2*np.pi))) * np.exp(-0.5 * ((x - μ)/σ)**2)
    full_pdf[(x < a) | (x > b)] = 0

    return full_pdf
```

```
trunc_norm_pdf(0, a=-1, b=1)
```

```
## array([0.58436857])
```

```
trunc_norm_pdf(2, a=-1, b=1)
```

```
## array([0.])
```

```
trunc_norm_pdf(-2, a=-1, b=1)
```

```
## array([0.])
```

```
trunc_norm_pdf([-2,1,0,1,2], a=-1, b=1)
```

```
quad(lambda x: trunc_norm_pdf(x, a=-1, b=1), -np.inf, np.inf)
```

```
## (1.0, 2.9512170485190836e-11)
```

```
quad(lambda x: trunc_norm_pdf(x, a=-3, b=3), -np.inf, np.inf)
```

```
## (0.9999999999999998, 7.472109098127788e-09)
```

# Multivariate normal

$$f(\mathbf{x}) = \det(2\pi\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

```
def mv_norm(x, mu, Sigma):
    x = np.asarray(x)
    mu = np.asarray(mu)
    Sigma = np.asarray(Sigma)

    return np.linalg.det(2*np.pi*Sigma)**(-0.5) * np.exp(-0.5 * (x - mu).T @ np.linalg.solve(Sigma, (x-mu)) )
```

```
norm_pdf(0,0,1)
## 0.3989422804014327
mv_norm([0], [0], [[1]])
## 0.3989422804014327
mv_norm([0,0], [0,0], [[1,0],[0,1]])
## 0.15915494309189535
mv_norm([0,0,0], [0,0,0], [[1,0,0],[0,1,0],[0,0,1]])
```

```
from scipy.integrate import dblquad, tplquad
dblquad(lambda y, x: mv_norm([x,y], [0,0], np.identity(2)),
        a=-np.inf, b=np.inf,
        gfun=lambda x: -np.inf, hfun=lambda x: np.inf)
## (1.000000000000322, 1.3150127836618008e-08)

tplquad(lambda z, y, x: mv_norm([x,y,z], [0,0,0], np.identity(3),
        a=0, b=np.inf,
        gfun=lambda x: 0, hfun=lambda x: np.inf,
        qfun=lambda x,y: 0, rfun=lambda x,y: np.inf)
```

## **Example 3 - (Very) Basic optimization**

# Scalar function minimization

```
def f(x):
    return x**4 + 3*(x-2)**3 - 15*(x)**2 + 1
```

```
from scipy.optimize import minimize_scalar
minimize_scalar(f, method="Brent")
```

```
##      fun: -803.3955308825884
##      nfev: 17
##      nit: 11
##  success: True
##          x: -5.528801125219663
```

```
minimize_scalar(f, method="bounded", bounds=[0, 6])
```

```
##      fun: -54.21003937712762
##  message: 'Solution found.'
##      nfev: 12
##      status: 0
##  success: True
##          x: 2.668865104039653
```

```
minimize_scalar(f, method="bounded", bounds=[-8, 6])
```

```
##      fun: -803.3955308825871
```

# Results

```
res = minimize_scalar(f)

type(res)

## <class 'scipy.optimize.optimize.OptimizeResult'>

dir(res)

## ['fun', 'nfev', 'nit', 'success', 'x']

res.success

## True

res.x

## -5.528801125219663
```

# More details

```
from scipy.optimize import show_options
show_options(solver="minimize_scalar")

##
## brent
## -----
##
## Options
## -----
## maxiter : int
##     Maximum number of iterations to perform.
## xtol : float
##     Relative error in solution `xopt` acceptable for convergence.
##
## Notes
## -----
## Uses inverse parabolic interpolation when possible to speed up
## convergence of golden section method.
##
## bounded
## -----
##
## Options
## -----
## maxiter : int
##     Maximum number of iterations to perform.
## disp: int, optional
##     If non-zero, print messages.
##     0 : no message printing.
```

# Local minima

```
def f(x):  
    return -np.sinc(x-5)
```

```
res = minimize_scalar(f)  
res  
  
##      fun: -0.049029624014074166  
##      nfev: 15  
##      nit: 10  
##  success: True  
##            x: -1.4843871263953001
```

# Random starts

```
rng = np.random.default_rng(seed=1234)

lower = rng.uniform(-20, 20, 100)
upper = lower + 1

sols = [minimize_scalar(f, bracket=(l,u)) for l,u
        in zip(lower, upper)]
functs = [sol.fun for sol in sols]

best = sols[np.argmin(functs)]
best

##      fun: -1.0
##      nfev: 12
##      nit: 8
##  success: True
##      x: 5.000000000618556
```



# Back to Rosenbrock's function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

```
def f(x):
    return (1-x[0])**2 + 100*(x[1]-x[0]**2)**2

from scipy.optimize import minimize

minimize(f, [0,0])

##      fun: 2.844030241790906e-11
##  hess_inv: array([[0.49482454,  0.98957635],
##                  [0.98957635,  1.98394216]])
##      jac: array([-3.98673382e-06, -2.84416264e-06])
##  message: 'Optimization terminated successfully.'
##      nfev: 72
##       nit: 19
##       njev: 24
##      status: 0
##     success: True
##         x: array([0.99999467,  0.99998932])

minimize(f, [-1,-1]).x
```

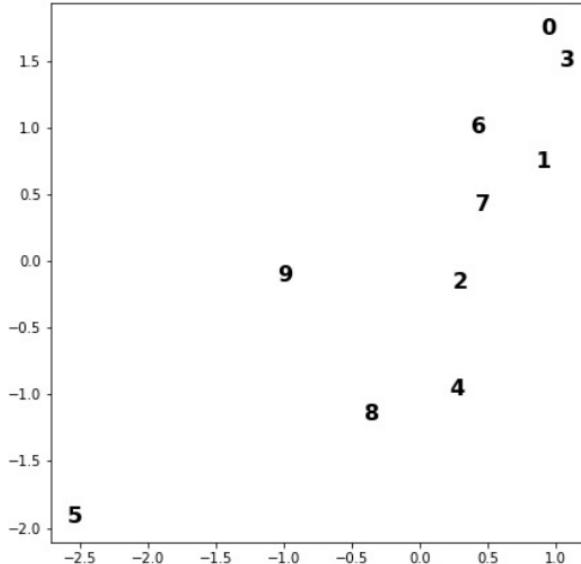
## **Example 4 - Spatial Tools**

# Nearest Neighbors

```
rng = np.random.default_rng(seed=12345)
pts = rng.multivariate_normal(
    [0,0], [[1,.8],[.8,1]],
    size=10
)
```

```
pts
```

```
## array([[ 0.951133 ,  1.75038506],
##        [ 0.90794002,  0.74402448],
##        [ 0.30576524, -0.16281136],
##        [ 1.09240417,  1.50280001],
##        [ 0.27501972, -0.96007933],
##        [-2.53321395, -1.92068272],
##        [ 0.43511779,  1.00571808],
##        [ 0.46218239,  0.42379897],
##        [-0.3509701 , -1.14575681],
##        [-0.98870241, -0.1039104 ]])
```



# KD Trees

```
from scipy.spatial import KDTree  
  
kd = KDTree(pts)  
kd  
  
## <scipy.spatial.kdtree.KDTree object at 0x1026b4  
  
dir(kd)  
  
## ['__class__', '__delattr__', '__dict__', '__dir__  
  
dist, i = kd.query(pts[6,:], k=3)  
dist  
  
## array([0.           , 0.54041133, 0.58254815])  
  
i  
  
## array([6, 1, 7])  
  
dist, i = kd.query(pts[2,:], k=5)  
i
```



# Convex hulls

```
from scipy.spatial import ConvexHull  
  
hull = ConvexHull(pts)  
hull  
  
## <scipy.spatial.qhull.ConvexHull object at 0x147  
  
dir(hull)  
  
## ['__class__', '__del__', '__delattr__', '__dict__  
  
hull.simplices  
  
## array([[0, 3],  
##         [4, 5],  
##         [9, 5],  
##         [9, 0],  
##         [1, 3],  
##         [1, 4]], dtype=int32)
```

```
scipy.spatial.convex_hull_plot_2d(hull)
```



# Delaunay triangulations

```
from scipy.spatial import Delaunay  
  
tri = Delaunay(pts)  
tri  
  
## <scipy.spatial.qhull.Delaunay object at 0x1477a  
  
dir(tri)  
  
## ['__class__', '__del__', '__delattr__', '__dict__  
  
tri.simplices  
  
## array([[8, 9, 5],  
##         [4, 8, 5],  
##         [9, 8, 2],  
##         [8, 4, 2],  
##         [4, 1, 2],  
##         [6, 1, 3],  
##         [0, 6, 3],  
##         [6, 0, 9],  
##         [7, 9, 2],  
##         [7, 6, 9],
```

```
scipy.spatial.delaunay_plot_2d(tri)
```



# Voronoi diagrams

```
from scipy.spatial import Voronoi  
  
vor = Voronoi(pts)  
vor  
  
## <scipy.spatial.qhull.Voronoi object at 0x1477c2  
  
dir(vor)  
  
## ['__class__', '__del__', '__delattr__', '__dict__  
  
vor.vertices  
  
## array([[-1.56917821, -1.17533646],  
##        [ 7.94738786, -27.97463108],  
##        [-0.3550644 , -0.43215628],  
##        [-0.18923926, -0.54294902],  
##        [ 1.98860973, -0.62693469],  
##        [ 0.83175084,  1.16435674],  
##        [ 0.64483401,  1.41151497],  
##        [-2.98645423,  3.92780753],  
##        [-0.32091034,  0.31844817],  
##        [-0.44985535,  0.67296975],  
##        [-1.5 ,  0. ]])
```

```
scipy.spatial.voronoi_plot_2d(vor)
```



## **Example 5 - stats**

# Distributions

Implements classes for 104 continuous and 19 discrete distributions,

- `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)
- `stats`: Return mean, variance, (Fisher's) skew, or (Fisher's) kurtosis
- `moment`: non-central moments of the distribution

# Basic usage

```
from scipy.stats import norm, gamma, binom, uniform

norm().rvs(size=5)

## array([ 1.37773488, -0.19096664,  0.55300367,  1.10696328,  0.49030573])

uniform.pdf([0,0.5,1,2])

## array([1., 1., 1., 0.])

binom.mean(n=10, p=0.25)

## 2.5

binom.median(n=10, p=0.25)

## 2.0

gamma(a=1,scale=1).stats()

## (array(1.), array(1.))

norm().stats(moments="mvsk")
```

# Freezing

Model parameters can be passed to any of the methods directory, or a distribution can be constructed using a specific set of parameters, which is known as freezing.

```
norm_rv = norm(loc=-1, scale=3)
norm_rv.median()

## -1.0

unif_rv = uniform(loc=-1, scale=2)
unif_rv.cdf([-2,-1,0,1,2])

## array([0. , 0. , 0.5, 1. , 1. ])

unif_rv.rvs(5)

## array([ 0.05213907, -0.04603696,  0.29595061,
```

```
g = gamma(a=2, loc=0, scale=1.2)

x = np.linspace(0, 10, 100)
plt.plot(x, g.pdf(x), "k-")
plt.axvline(x=g.mean(), c="r")
plt.axvline(x=g.median(), c="b")
```

# MLE

Maximum likelihood estimation is possible via the `fit()` method,

```
x = norm.rvs(loc=2.5, scale=2, size=1000, random_state=1234)
norm.fit(x)
```

```
## (2.5314811643075235, 1.946132398754459)
```

```
norm.fit(x, loc=2.5) # provide a guess for the parameter
```

```
## (2.5314811643075235, 1.946132398754459)
```

```
x = gamma.rvs(a=2.5, size=1000)
gamma.fit(x) # shape, loc, scale
```

```
## (2.717451544403441, -0.041437185161843526, 0.9293407167094078)
```

```
y = gamma.rvs(a=2.5, loc=-1, scale=2, size=1000)
gamma.fit(y) # shape, loc, scale
```

```
## (2.3325993640858007, -0.9659594238725819, 2.1028797903487417)
```