SciPy

Lecture 08

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		

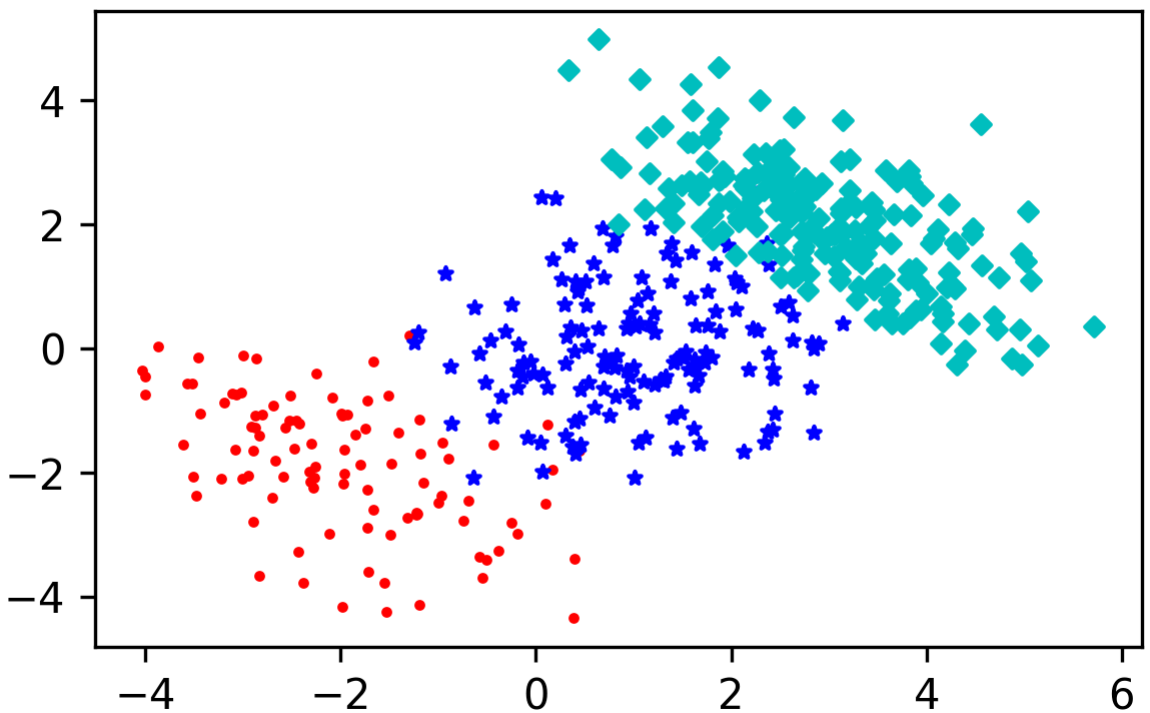
SciPy vs NumPy

> In an ideal world, NumPy would contain nothing but the array data type and the most basic operations: indexing, sorting, reshaping, basic elementwise functions, etc. All numerical code would reside in SciPy. However, one of NumPy's important goals is compatibility, so NumPy tries to retain all features supported by either of its predecessors. Thus, NumPy contains some linear algebra functions and Fourier transforms, even though these more properly belong in SciPy. In any case, SciPy contains more fully-featured versions of the linear algebra modules, as well as many other numerical algorithms. If you are doing scientific computing with Python, you should probably install both NumPy and SciPy. Most new features belong in SciPy rather than NumPy.

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

```
1 from scipy.cluster.vq import kmeans
2 ctr, dist = kmeans(pts, 3)
```

```
1 ctr
```

```
array([[ 2.86399,  1.95401],
       [-2.03957, -1.85662],
       [ 0.91124, -0.18724]])
```

```
1 dist
```

```
np.float64(1.2209235923868729)
```

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

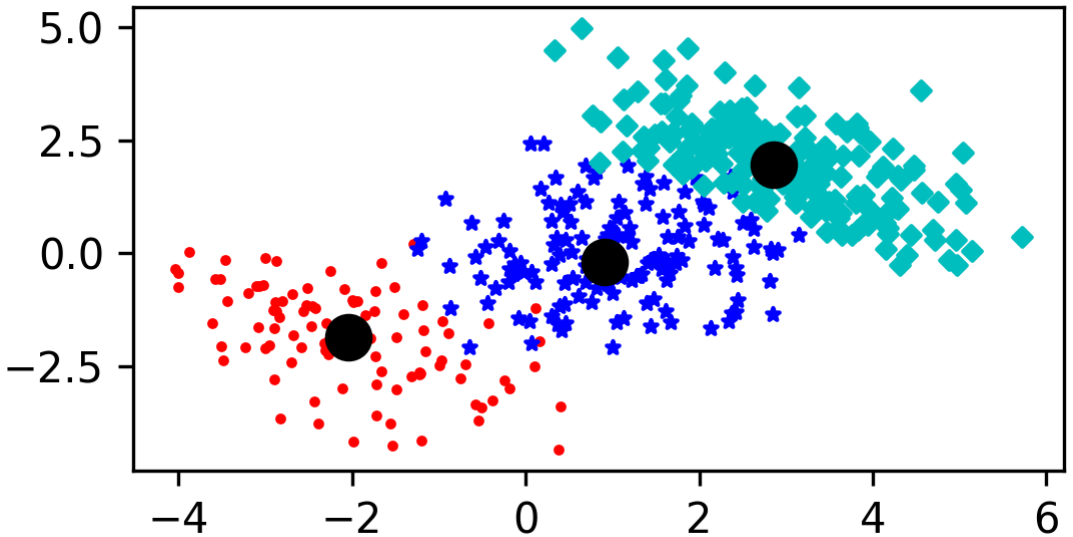
```
array([-2.00475, -1.87276])
```

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

```
array([1.03849, 0.01417])
```

```
1 cl3.mean(axis=0)
```

```
array([2.94642, 2.02514])
```


5.0
2.5
0.0
−2.5
−4
−2
0
2
4
6

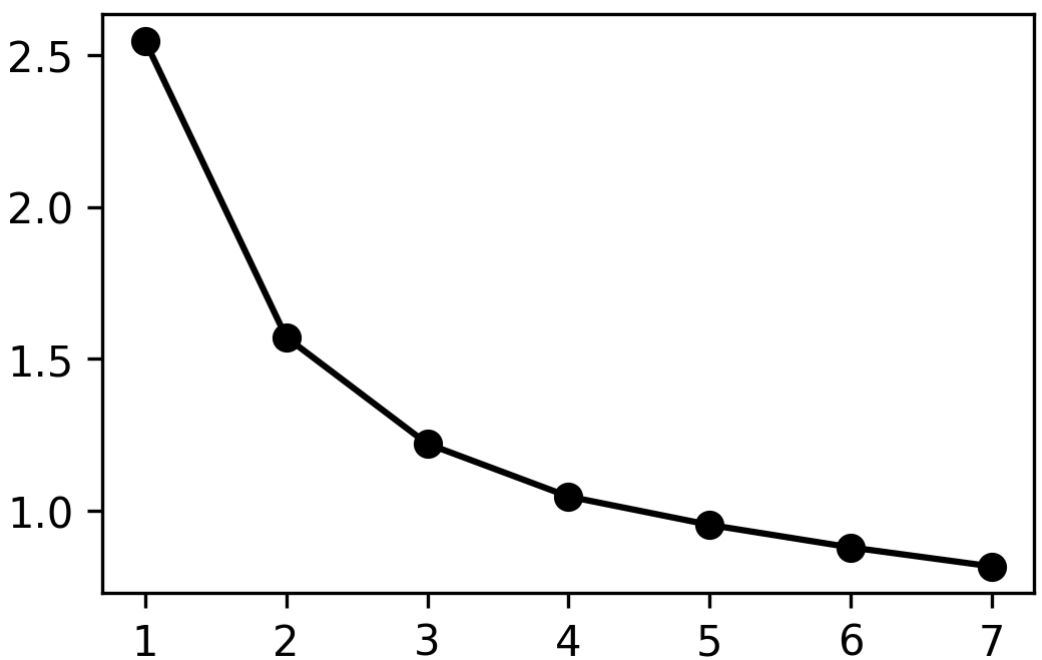
k-means distortion plot

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

```
1 ks = range(1,8)
2 dists = [kmeans(pts, k)[1] for k in ks]
```

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

```
array([2.54703, 1.57178, 1.22092, 1.04749, 0.95422, 0.88013, 0.8176 ])
```



Assigning new points with vq()

Once we have centroids from `kmeans()`, we can assign new points to clusters using `vq()` (vector quantization).

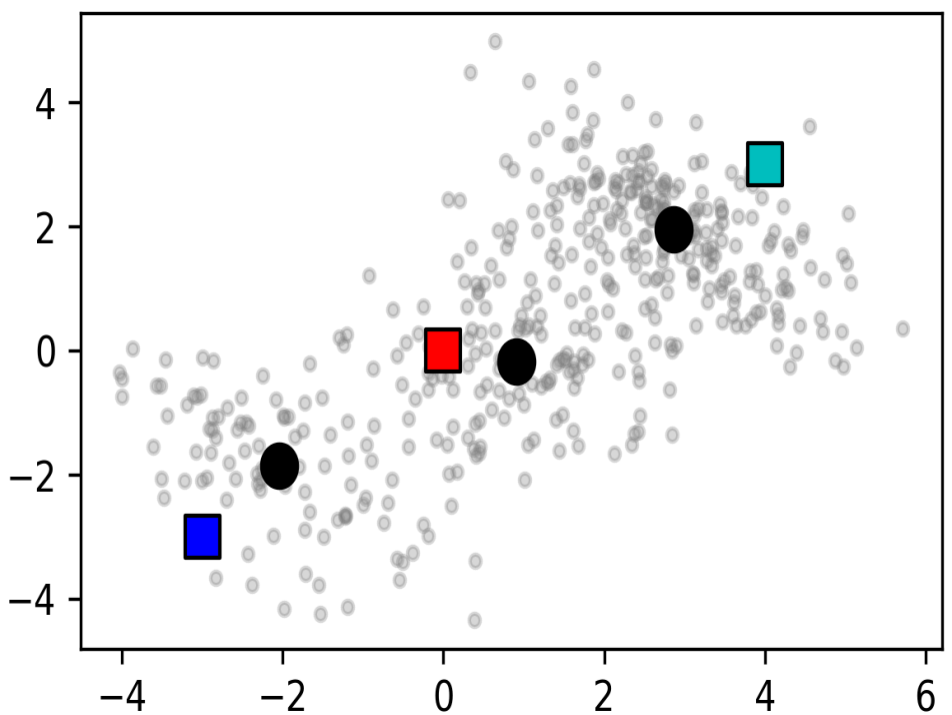
```
1 from scipy.cluster.vq import vq
2
3 new_pts = np.array([[0, 0], [-3, -3], [4, 3]])
4 labels, dists = vq(new_pts, ctr)
```

```
1 labels
```

```
array([2, 1, 0], dtype=int32)
```

```
1 dists
```

```
array([0.93028, 1.49323, 1.54422])
```



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 then the lower and upper bounds of the integral.

```
1 from scipy.integrate import quad
```

```
1 quad(lambda x: x, 0, 1)
```

```
(0.5, 5.551115123125783e-15)
```

```
1 quad(np.sin, 0, np.pi)
```

```
(2.0, 2.220446049250313e-14)
```

```
1 quad(np.sin, 0, 2*np.pi)
```

```
(2.221501482512777e-16, 4.3998892617846e-14)
```

```
1 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, μ, σ):
  return (1/(σ * np.sqrt(2*np.pi))) * np.exp(-0.5 * ((x - μ)/σ)**2)
```

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

```
np.float64(0.3989422804014327)
```

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

```
np.float64(0.0)
```

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

```
np.float64(0.0)
```

Checking the PDF

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

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

```
TypeError: norm_pdf() missing 2 required positional arguments: 'μ' and 'σ'
```

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

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

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

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

Truncated normal PDF

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

```python
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)
  scalar_input = x.ndim == 0
  x = np.atleast_1d(x)

  full_pdf = (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[0] if scalar_input else full_pdf
```

Testing trunc_norm_pdf

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

```
np.float64(0.3989422804014327)
```

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

```
np.float64(0.0)
```

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

```
np.float64(0.0)
```

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

```
array([0.     , 0.24197, 0.39894, 0.24197, 0.     ])
```

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

```
(0.682689492137086, 2.0147661317082566e-11)
```

```
1 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)
  scalar_input = x.ndim == 0
  x = np.atleast_1d(x)

  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[0] if scalar_input else full_pdf
```

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

```
np.float64(0.5843685672568166)
```

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

```
np.float64(0.0)
```

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

```
np.float64(0.0)
```

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

```
array([0.     , 0.35444, 0.58437, 0.35444, 0.     ])
```

```
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

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

```
def mv_norm(x, μ, Σ):
  x = np.asarray(x)
  μ = np.asarray(μ)
  Σ = np.asarray(Σ)

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

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

```
np.float64(0.3989422804014327)
```

```
mv_norm([0,0], [0,0], [[1,0],[0,1]])
```

```
np.float64(0.15915494309189535)
```

```
mv_norm([0], [0], [[1]])
```

```
np.float64(0.3989422804014327)
```

```
mv_norm([0,0,0], [0,0,0],
        [[1,0,0],[0,1,0],[0,0,1]])
```

```
np.float64(0.06349363593424098)
```

2d & 3d numerical integration

These are supported by `dblquad()` and `tplquad()` respectively (see `nquad()` for higher dimensions).

```
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)
```

```
(1.0000000000000322, 1.3150127836597 68e-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)
```

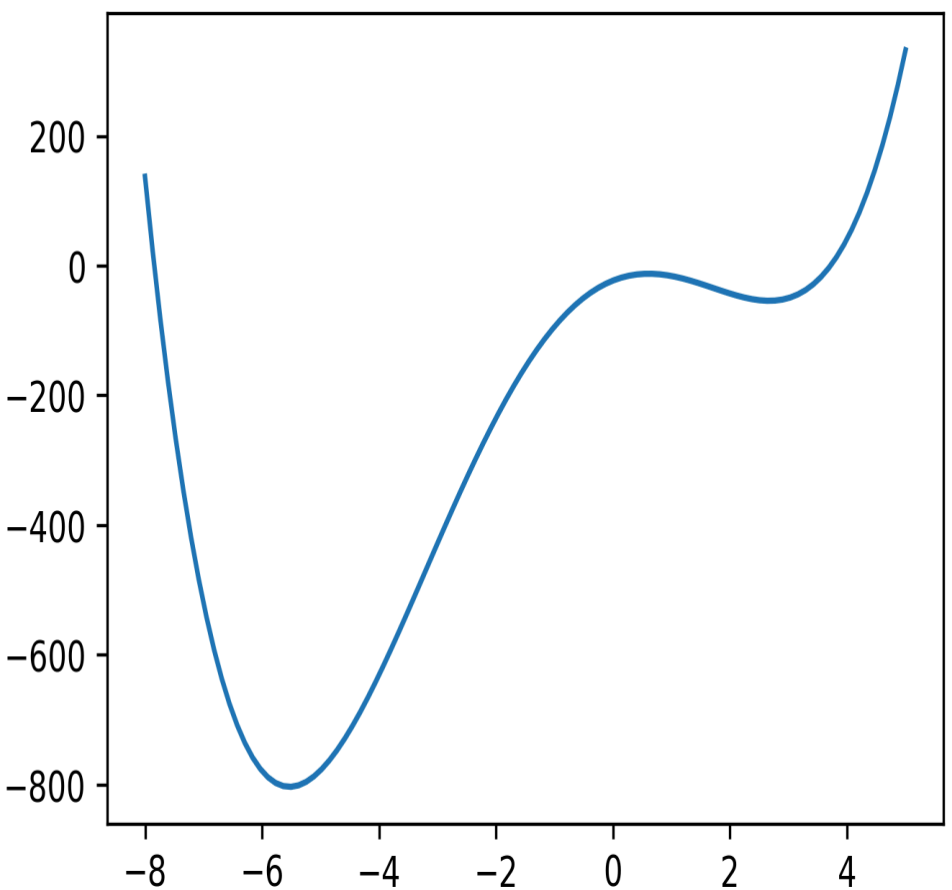
```
(0.12500000000036066, 1.4697203688869196e-08)
```

Example 3
(Very) Basic optimization

Scalar function minimization

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

```
1  from scipy.optimize import minimize_scalar
2  minimize_scalar(f)
```




```
message:
        Optimization terminated successfully;
        The returned value satisfies the termi
        (using xtol = 1.48e-08 )
success: True
    fun: -803.3955308825884
      x: -5.528801125219663
    nit: 11
   nfev: 16
```

Results

```
1 res = minimize_scalar(f)
```

```
1 type(res)
```

```
scipy.optimize._optimize.OptimizeResult
```

```
1 dir(res)
```

```
['fun', 'message', 'nfev', 'nit', 'success', 'x']
```

```
1 res.fun
```

```
np.float64(-803.3955308825884)
```

```
1 print(res.message)
```

```
Optimization terminated successfully;
The returned value satisfies the termination cri
(using xtol = 1.48e-08 )
```

```
1 res.nfev
```

```
16
```

```
1 res.nit
```

```
11
```

```
1 res.success
```

```
True
```

```
1 res.x
```

```
np.float64(-5.528801125219663)
```

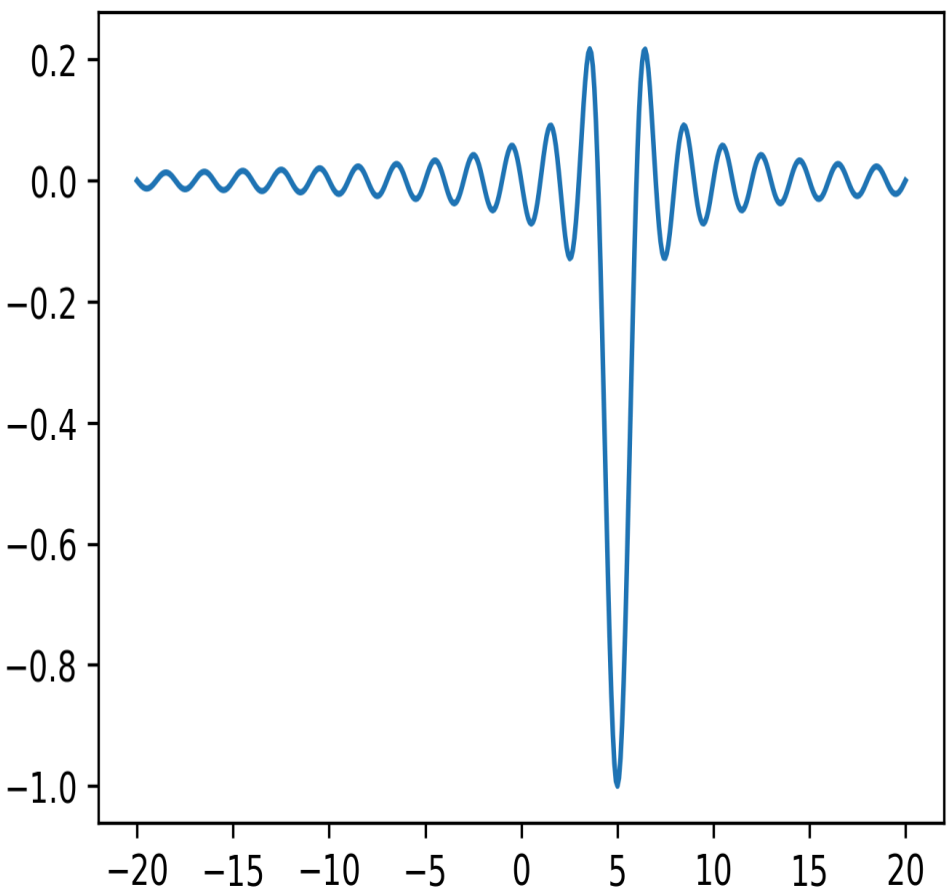
More details

```
1 from scipy.optimize import show_options
2 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.
disp : int, optional
    If non-zero, print messages.

    ``0`` : no message printing.

    ``1`` : non-convergence notification messages only.

    ``2`` : print a message on convergence too.

    ``3`` : print iteration results.
```

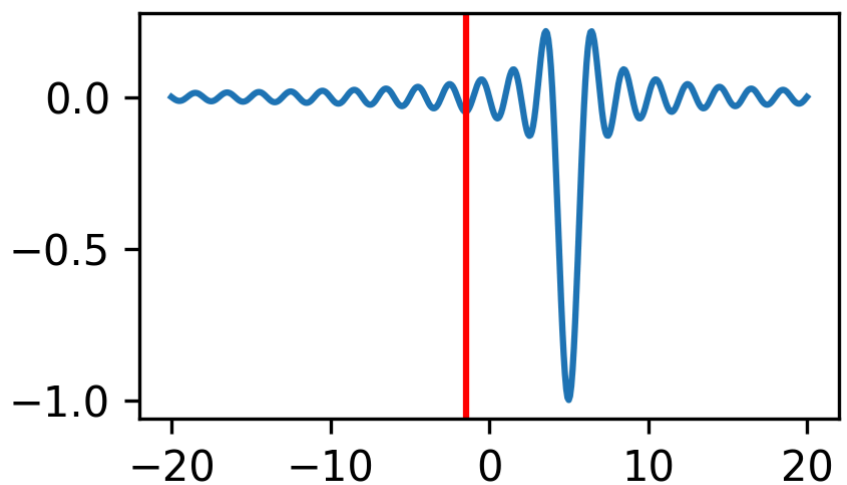
Local minima

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




```
1 res = minimize_scalar(f); res
```

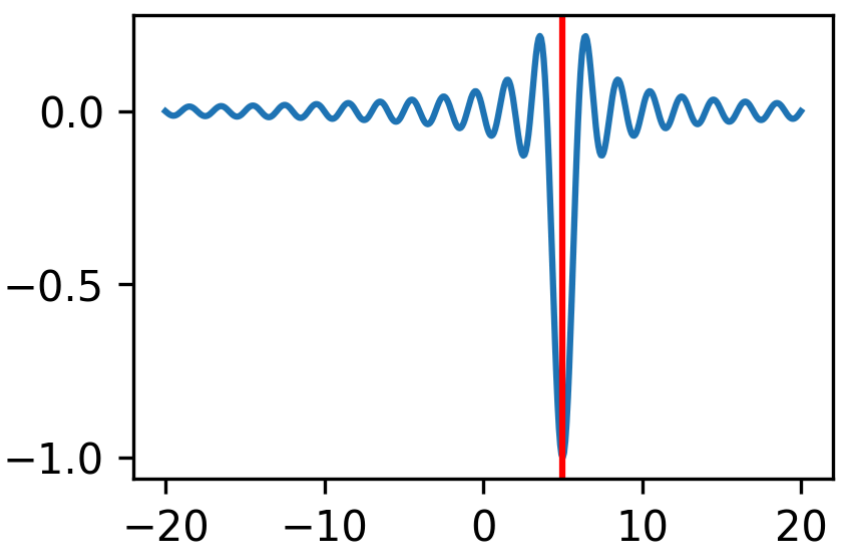
```
message:
        Optimization terminated successf
        The returned value satisfies the
        (using xtol = 1.48e-08 )
success: True
    fun: -0.049029624014074166
      x: -1.4843871263953001
    nit: 10
   nfev: 14
```



Random starts

```python
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)]
funs = [sol.fun for sol in sols]

best = sols[np.argmin(funs)]
best
```

```
message:
          Optimization terminated successfully;
          The returned value satisfies the termi
          (using xtol = 1.48e-08 )
success: True
    fun: -1.0
      x: 5.0000000006185585
    nit: 8
   nfev: 11
```



Example 4
Statistics

Distributions

Implements classes for most continuous and 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

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

```
1 norm().rvs(size=5)
```

```
array([-0.27378, -0.23604, -0.02471,  0.68152,  0.12815])
```

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

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

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

```
np.float64(2.5)
```

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

```
np.float64(2.0)
```

```
1 norm().stats()
```

```
(np.float64(0.0), np.float64(1.0))
```

```
1 gamma(a=1,scale=1).stats(moments="mvsk")
```

```
(np.float64(1.0), np.float64(1.0), np.float64(2.0), np.float64(6.0))
```

Freezing

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

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

```
np.float64(-1.0)
```

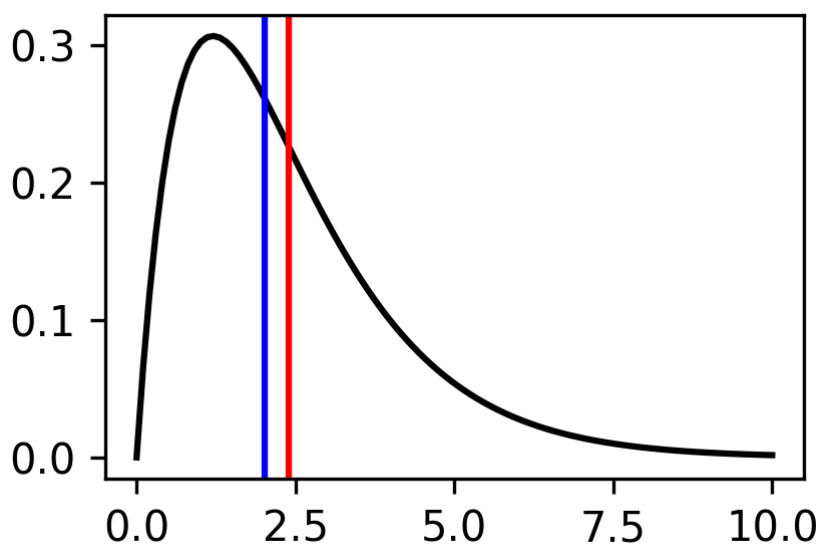
```
1 unif_rv = uniform(loc=-1, scale=2)
2 unif_rv.cdf([-2,-1,0,1,2])
```

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

```
1 unif_rv.rvs(5)
```

```
array([-0.58057,  0.46872, -0.8764 ,  0.83672, -
```

```
1 g = gamma(a=2, loc=0, scale=1.2)
2
3 plt.figure(figsize=(3, 2))
4 x = np.linspace(0, 10, 100)
5 plt.plot(x, g.pdf(x), "k-")
6 plt.axvline(x=g.mean(), c="r")
7 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)
```

```
(np.float64(2.5314811643075235), np.float64(1.946132398754459))
```

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

```
(np.float64(2.5314811643075235), np.float64(1.946132398754459))
```

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

```
(np.float64(2.589173185346138),
 np.float64(-0.00011290008283903167),
 np.float64(0.9676759000682381))
```

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

```
(np.float64(2.782312461220813 7),
 np.float64(-1.1090393239477736),
 np.float64(1.8785702919274456))
```

Example 5
Special Functions

scipy.special

Provides implementations of many special mathematical functions commonly used in statistics, physics, and engineering.

Function	Description
`gamma`, `gammaln`	Gamma function and its log
`beta`, `betaln`	Beta function and its log
`factorial`, `comb`, `perm`	Combinatorial functions
`erf`, `erfc`	Error function and complement
`expit`, `logit`	Logistic and inverse logistic
`softmax`, `log_softmax`	Softmax and log-softmax
`digamma`, `polygamma`	Digamma and polygamma functions
`bessel*`	Bessel functions (many variants)

Gamma and Beta functions

```
1 from scipy.special import gamma, gammaln, beta, betaln
```

```
1 gamma(5)  # (n-1)! for integers
```

np.float64(24.0)

```
1 gammaln(100)  # log(gamma(100))
```

np.float64(359.1342053695754)

```
1 gamma(0.5)  # sqrt(pi)
```

np.float64(1.7724538509055159)

```
1 beta(2, 3)  # B(a,b) = gamma(a)*gamma(
```

np.float64(0.08333333333333333)

```
1 np.sqrt(np.pi)
```

np.float64(1.7724538509055159)

```
1 gamma(2) * gamma(3) / gamma(5)
```

np.float64(0.08333333333333333)

Combinatorial functions

```
1 from scipy.special import factorial, comb, perm
```

```
1 factorial(5)
```

```
np.float64(120.0)
```

```
1 comb(10, 3)  # 10 choose 3
```

```
np.float64(120.0)
```

```
1 factorial(np.arange(6))
```

```
array([  1.,   1.,   2.,   6.,  24., 120.])
```

```
1 perm(10, 3)  # 10 permute 3
```

```
np.float64(720.0)
```

Error function

The error function $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt,$

```
1 from scipy.special import erf, erfc
```

```
1 erf(0)
```

```
np.float64(0.0)
```

```
1 erf(np.inf)
```

```
np.float64(1.0)
```

```
1 erfc(0)  # 1 - erf(x)
```

```
np.float64(1.0)
```

Logistic functions

```
1 from scipy.special import expit, logit, softmax
```

```
1 expit(0)  # 1 / (1 + exp(-x))
```

```
np.float64(0.5)
```

```
1 expit([-np.inf, 0, np.inf])
```

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

```
1 logit(0.5)  # log(p / (1-p))
```

```
np.float64(0.0)
```

```
1 logit([0.1, 0.5, 0.9])
```

```
array([-2.19722,  0.     ,  2.19722])
```

```
1 softmax([1, 2, 3])
```

```
array([0.09003, 0.24473, 0.66524])
```

```
1 softmax([1, 2, 3]).sum()
```

```
np.float64(0.9999999999999999)
```

```
1 softmax([0, 0, 0])
```

```
array([0.33333, 0.33333, 0.33333])
```

Example 6
Linear Algebra

NumPy vs SciPy linalg

Both `numpy.linalg` and `scipy.linalg` provide linear algebra routines, but SciPy's version is more comprehensive.

Feature	`numpy.linalg`	`scipy.linalg`
Basic operations	`inv`, `solve`, `det`, `eig`	All of NumPy's + more
Decompositions	SVD, QR, Cholesky	+ LU, Schur, Hessenberg, polar
Matrix functions	Limited	`expm`, `logm`, `sqrtm`, `funm`
Specialized solvers	No	Banded, triangular, symmetric
LAPACK access	Partial	Full access via low-level routines

Basic operations comparison

```
1 import scipy.linalg
```

```
1 A = np.array([[1, 2], [3, 4]])
2 b = np.array([5, 6])
```

NumPy

```
1 np.linalg.solve(A, b)
```

```
array([-4. ,  4.5])
```

```
1 np.linalg.inv(A)
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

SciPy

```
1 scipy.linalg.solve(A, b)
```

```
array([-4. ,  4.5])
```

```
1 scipy.linalg.inv(A)
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

SciPy-only features

LU Decomposition

```
P, L, U = scipy.linalg.lu(A)
```

```
L
```

```
array([[1.     , 0.     ],
       [0.33333, 1.     ]])
```

```
U
```

```
array([[3.     , 4.     ],
       [0.     , 0.66667]])
```

Matrix exponential

```
scipy.linalg.expm(A)
```

```
array([[ 51.96896,  74.73656],
       [112.10485, 164.0738 ]])
```

```
scipy.linalg.logm(
  scipy.linalg.expm(A)
)
```

```
array([[1., 2.],
       [3., 4.]])
```

Dense vs Sparse matrices

For matrices with many zero entries, sparse representations are more memory-efficient and can be much faster.

```
from scipy import sparse
```

Dense (stores all elements)

```
dense = np.array([
    [1, 0, 0, 0],
    [0, 2, 0, 0],
    [0, 0, 3, 0],
    [0, 0, 0, 4]
])
dense.nbytes
```

```
128
```

Sparse (stores only non-zeros)

```
sp = sparse.csr_matrix(dense)
sp
```

```
<Compressed Sparse Row sparse matrix of dt
    with 4 stored elements and shape (4, 4
```

```
sp.data
```

```
array([1, 2, 3, 4])
```

Sparse matrix formats

Format	Name	Best for
`csr_matrix`	Compressed Sparse Row	Row slicing, matrix-vector products
`csc_matrix`	Compressed Sparse Column	Column slicing, arithmetic
`coo_matrix`	Coordinate	Building sparse matrices incrementally
`dia_matrix`	Diagonal	Diagonal/banded matrices
`lil_matrix`	List of Lists	Building matrices, row-based modifications

Creating sparse matrices

```
row = [0, 1, 2, 2]
col = [0, 1, 0, 2]
data = [1, 2, 3, 4]
sp = sparse.coo_matrix(
  (data, (row, col)), shape=(3, 3)
)
sp.toarray()
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [3, 0, 4]])
```

```
sparse.eye(4, format="csr").toarray()
```

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

```
sparse.random(3, 3, density=0.3).toarray()
```

```
array([[0.     , 0.     , 0.     ],
       [0.     , 0.     , 0.22213],
       [0.     , 0.30699, 0.68624]])
```

```
sparse.diags([1., 2., 3., 4.]).toarray()
```

```
array([[1., 0., 0., 0.],
       [0., 2., 0., 0.],
       [0., 0., 3., 0.],
       [0., 0., 0., 4.]])
```

Sparse linear algebra

`scipy.sparse.linalg` provides solvers optimized for sparse matrices.

```
from scipy.sparse.linalg import spsolve, eigs
```

```
n = 1000
diagonals = [np.ones(n-1), -2*np.ones(n), np.ones(n-1)]
A_sparse = sparse.diags(diagonals, [-1, 0, 1], format="csr")
b = np.ones(n)
```

```
x = spsolve(A_sparse, b)
x[:5]
```

```
array([ -500.,  -999., -1497., -1994., -2490.])
```

```
vals, vecs = eigs(A_sparse, k=3)
vals
```

```
array([-3.99999+0.j, -3.99996+0.j, -3.99991+0.j])
```

When to use sparse?

General rule - use sparse when *density* < *10%* and matrix is large.