intro_Scipy

September 17, 2023

[ ]:
```
# The following is to know when this notebook has been run and with which␣
↪python version.
import time, sys
print(time.ctime())
print(sys.version.split('|')[0])
```

```
Sun Sep 17 09:40:19 2023
3.9.16 (main, Jan 11 2023, 10:02:19)
[Clang 14.0.6 ]
```

1 E Introduction to Scipy

This is part of the Python lecture given by Christophe Morisset at IA-UNAM.

Scipy is a library with a lot of foncionalities, we will not cover everything here, but rather point to some of them with examples. Some useful links about scipy:

- https://scipy-lectures.github.io/intro/scipy.html
- https://docs.scipy.org/doc/scipy/tutorial/index.html

[ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

[ ]:
```
import scipy # This imports a lot of scipy stuff, but not the "important"␣
↪modules
```

1.0.1 Some usefull methods

[ ]:
```
from scipy.special import gamma
%timeit g1 = gamma(10.3)
%timeit g1 = gamma(10)
%timeit g2 = 9*8*7*6*5*4*3*2
%timeit g3 = 10*9*8*7*6*5*4*3*2
g1 = gamma(10.3)
g2 = 9*8*7*6*5*4*3*2
g3 = 10*9*8*7*6*5*4*3*2
print(g1, g2, g3)
```

```
917 ns ± 49.2 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
1.41 µs ± 135 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
16.4 ns ± 0.608 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
16.8 ns ± 0.772 ns per loop (mean ± std. dev. of 7 runs, 100,000,000 loops each)
716430.6890623764 362880 3628800
```

[ ]:
```python
from scipy import constants as cst
print(cst.astronomical_unit) # A lot of constants
print('{} {}'.format(cst.value('proton mass'), cst.unit('proton mass')))
```

```
149597870700.0
1.67262192369e-27 kg
```

List there: http://docs.scipy.org/doc/scipy/reference/constants.html#constants-database

1.0.2 Integrations

[ ]:
```python
from scipy.integrate import trapz, cumtrapz, simps
#help(scipy.integrate) # a big one...
print('---------------------------------------------------------------------------------')
help(trapz)
print('---------------------------------------------------------------------------------')
help(cumtrapz)
print('---------------------------------------------------------------------------------')
help(simps)
```

```
--------------------------------------------------------------------------------
--
Help on function trapz in module scipy.integrate._quadrature:

trapz(y, x=None, dx=1.0, axis=-1)
    An alias of `trapezoid`.

    `trapz` is kept for backwards compatibility. For new code, prefer
    `trapezoid` instead.

--------------------------------------------------------------------------------
--
Help on function cumtrapz in module scipy.integrate._quadrature:

cumtrapz(y, x=None, dx=1.0, axis=-1, initial=None)
    An alias of `cumulative_trapezoid`.

    `cumtrapz` is kept for backwards compatibility. For new code, prefer
    `cumulative_trapezoid` instead.

--------------------------------------------------------------------------------
--
```

```
Help on function simps in module scipy.integrate._quadrature:

simps(y, x=None, dx=1.0, axis=-1, even='avg')
    An alias of `simpson`.

    `simps` is kept for backwards compatibility. For new code, prefer
    `simpson` instead.
```

[ ]:
```
dir(scipy.integrate)
```

[ ]:
```
['AccuracyWarning',
 'BDF',
 'DOP853',
 'DenseOutput',
 'IntegrationWarning',
 'LSODA',
 'OdeSolution',
 'OdeSolver',
 'RK23',
 'RK45',
 'Radau',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_bvp',
 '_dop',
 '_ivp',
 '_ode',
 '_odepack',
 '_quad_vec',
 '_quadpack',
 '_quadrature',
 'complex_ode',
 'cumtrapz',
 'cumulative_trapezoid',
 'dblquad',
 'fixed_quad',
 'lsoda',
 'newton_cotes',
 'nquad',
 'ode',
 'odeint',
 'odepack',
 'quad',
 'quad_explain',
 'quad_vec',
 'quadpack',
 'quadrature',
 'romb',
 'romberg',
 'simps',
 'simpson',
 'solve_bvp',
 'solve_ivp',
 'test',
 'tplquad',
 'trapezoid',
 'trapz',
 'vode']
```

[ ]:
```
# Defining x and y
x = np.linspace(0, np.pi, 100)
%timeit  y = np.sin(x)
y = np.sin(x)
# Compare the integrales using two methods
%timeit i1 = trapz(y, x)
%timeit i2 = simps(y, x)

print(trapz(y, x))
print(simps(y, x))

x = np.linspace(0, np.pi, 10)
y = np.sin(x)
%timeit i1 = trapz(y, x)
%timeit i2 = simps(y, x)
print(trapz(y, x))
print(simps(y, x))
```

```
1.76 µs ± 133 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
17.2 µs ± 872 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
183 µs ± 23.1 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
1.9998321638939929
1.9999999690165366
16.4 µs ± 1.12 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
180 µs ± 24 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
1.9796508112164835
```

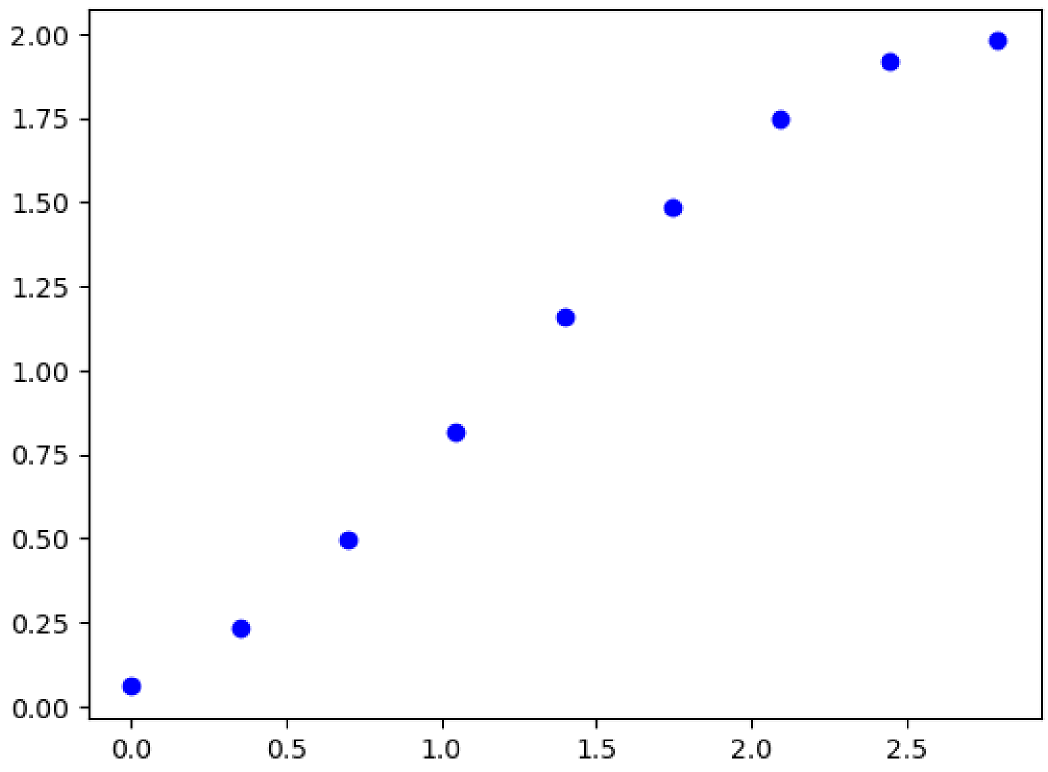
```
1.9995487365804028
```

[ ]:
```
# Cumulative integrale
cum = cumtrapz(np.abs(y), x)
print(len(x), type(cum), len(cum), cum)
```

```
10 <class 'numpy.ndarray'> 9 [0.05969378 0.23157515 0.4949127  0.81794403
1.16170678 1.48473811
 1.74807566 1.91995704 1.97965081]
```

[ ]:
```
# Cumulative integral

print('{} {}'.format(len(x), len(cumtrapz(np.abs(y), x))))
f, ax = plt.subplots()
ax.plot(x[0:-1], cumtrapz(np.abs(y), x), 'bo');
```

```
10 9
```



[ ]:
```
from scipy.integrate import quad # To compute a definite integral
help(quad)
```

```
Help on function quad in module scipy.integrate._quadpack_py:
```

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08, epsrel=1.49e-08,
limit=50, points=None, weight=None, wvar=None, wopts=None, maxp1=50, limlst=50)
    Compute a definite integral.

    Integrate func from `a` to `b` (possibly infinite interval) using a
    technique from the Fortran library QUADPACK.

    Parameters
    ----------
    func : {function, scipy.LowLevelCallable}
        A Python function or method to integrate. If `func` takes many
        arguments, it is integrated along the axis corresponding to the
        first argument.

        If the user desires improved integration performance, then `f` may
        be a `scipy.LowLevelCallable` with one of the signatures::

            double func(double x)
            double func(double x, void *user_data)
            double func(int n, double *xx)
            double func(int n, double *xx, void *user_data)

        The ``user_data`` is the data contained in the `scipy.LowLevelCallable`.
        In the call forms with ``xx``,  ``n`` is the length of the ``xx``
        array which contains ``xx[0] == x`` and the rest of the items are
        numbers contained in the ``args`` argument of quad.

        In addition, certain ctypes call signatures are supported for
        backward compatibility, but those should not be used in new code.
    a : float
        Lower limit of integration (use -numpy.inf for -infinity).
    b : float
        Upper limit of integration (use numpy.inf for +infinity).
    args : tuple, optional
        Extra arguments to pass to `func`.
    full_output : int, optional
        Non-zero to return a dictionary of integration information.
        If non-zero, warning messages are also suppressed and the
        message is appended to the output tuple.

    Returns
    -------
    y : float
        The integral of func from `a` to `b`.
    abserr : float
        An estimate of the absolute error in the result.
    infodict : dict
        A dictionary containing additional information.
```

```
message
    A convergence message.
explain
    Appended only with 'cos' or 'sin' weighting and infinite
    integration limits, it contains an explanation of the codes in
    infodict['ierlst']

Other Parameters
----------------
epsabs : float or int, optional
    Absolute error tolerance. Default is 1.49e-8. `quad` tries to obtain
    an accuracy of ``abs(i-result) <= max(epsabs, epsrel*abs(i))``
    where ``i`` = integral of `func` from `a` to `b`, and ``result`` is the
    numerical approximation. See `epsrel` below.
epsrel : float or int, optional
    Relative error tolerance. Default is 1.49e-8.
    If ``epsabs <= 0``, `epsrel` must be greater than both 5e-29
    and ``50 * (machine epsilon)``. See `epsabs` above.
limit : float or int, optional
    An upper bound on the number of subintervals used in the adaptive
    algorithm.
points : (sequence of floats,ints), optional
    A sequence of break points in the bounded integration interval
    where local difficulties of the integrand may occur (e.g.,
    singularities, discontinuities). The sequence does not have
    to be sorted. Note that this option cannot be used in conjunction
    with ``weight``.
weight : float or int, optional
    String indicating weighting function. Full explanation for this
    and the remaining arguments can be found below.
wvar : optional
    Variables for use with weighting functions.
wopts : optional
    Optional input for reusing Chebyshev moments.
maxp1 : float or int, optional
    An upper bound on the number of Chebyshev moments.
limlst : int, optional
    Upper bound on the number of cycles (>=3) for use with a sinusoidal
    weighting and an infinite end-point.

See Also
--------
dblquad : double integral
tplquad : triple integral
nquad : n-dimensional integrals (uses `quad` recursively)
fixed_quad : fixed-order Gaussian quadrature
quadrature : adaptive Gaussian quadrature
odeint : ODE integrator
```

```
ode : ODE integrator
simpson : integrator for sampled data
romb : integrator for sampled data
scipy.special : for coefficients and roots of orthogonal polynomials

Notes
-----

**Extra information for quad() inputs and outputs**

If full_output is non-zero, then the third output argument
(infodict) is a dictionary with entries as tabulated below. For
infinite limits, the range is transformed to (0,1) and the
optional outputs are given with respect to this transformed range.
Let M be the input argument limit and let K be infodict['last'].
The entries are:

'neval'
    The number of function evaluations.
'last'
    The number, K, of subintervals produced in the subdivision process.
'alist'
    A rank-1 array of length M, the first K elements of which are the
    left end points of the subintervals in the partition of the
    integration range.
'blist'
    A rank-1 array of length M, the first K elements of which are the
    right end points of the subintervals.
'rlist'
    A rank-1 array of length M, the first K elements of which are the
    integral approximations on the subintervals.
'elist'
    A rank-1 array of length M, the first K elements of which are the
    moduli of the absolute error estimates on the subintervals.
'iord'
    A rank-1 integer array of length M, the first L elements of
    which are pointers to the error estimates over the subintervals
    with ``L=K`` if ``K<=M/2+2`` or ``L=M+1-K`` otherwise. Let I be the
    sequence ``infodict['iord']`` and let E be the sequence
    ``infodict['elist']``.  Then ``E[I[1]], ..., E[I[L]]`` forms a
    decreasing sequence.

If the input argument points is provided (i.e., it is not None),
the following additional outputs are placed in the output
dictionary. Assume the points sequence is of length P.

'pts'
    A rank-1 array of length P+2 containing the integration limits
    and the break points of the intervals in ascending order.
    This is an array giving the subintervals over which integration
    will occur.
'level'
    A rank-1 integer array of length M (=limit), containing the
    subdivision levels of the subintervals, i.e., if (aa,bb) is a
    subinterval of ``(pts[1], pts[2])`` where ``pts[0]`` and ``pts[2]``
    are adjacent elements of ``infodict['pts']``, then (aa,bb) has level l
    if ``|bb-aa| = |pts[2]-pts[1]| * 2**(-l)``.
'ndin'
    A rank-1 integer array of length P+2. After the first integration
    over the intervals (pts[1], pts[2]), the error estimates over some
    of the intervals may have been increased artificially in order to
    put their subdivision forward. This array has ones in slots
    corresponding to the subintervals for which this happens.

**Weighting the integrand**

The input variables, *weight* and *wvar*, are used to weight the
integrand by a select list of functions. Different integration
methods are used to compute the integral with these weighting
functions, and these do not support specifying break points. The
possible values of weight and the corresponding weighting functions are.

==========  ===================================   =====================
``weight``  Weight function used                  ``wvar``
==========  ===================================   =====================
'cos'       cos(w*x)                              wvar = w
'sin'       sin(w*x)                              wvar = w
'alg'       g(x) = ((x-a)**alpha)*((b-x)**beta)   wvar = (alpha, beta)
'alg-loga'  g(x)*log(x-a)                         wvar = (alpha, beta)
'alg-logb'  g(x)*log(b-x)                         wvar = (alpha, beta)
'alg-log'   g(x)*log(x-a)*log(b-x)                wvar = (alpha, beta)
'cauchy'    1/(x-c)                               wvar = c
==========  ===================================   =====================

wvar holds the parameter w, (alpha, beta), or c depending on the weight
selected. In these expressions, a and b are the integration limits.

For the 'cos' and 'sin' weighting, additional inputs and outputs are
available.

For finite integration limits, the integration is performed using a
Clenshaw-Curtis method which uses Chebyshev moments. For repeated
calculations, these moments are saved in the output dictionary:

'momcom'
    The maximum level of Chebyshev moments that have been computed,
```

i.e., if ``M_c`` is ``infodict['momcom']`` then the moments have been
computed for intervals of length ``|b-a| * 2**(-l)``,
``l=0,1,…,M_c``.

'nnlog'
A rank-1 integer array of length M(=limit), containing the
subdivision levels of the subintervals, i.e., an element of this
array is equal to l if the corresponding subinterval is
``|b-a|* 2**(-l)``.

'chebmo'
A rank-2 array of shape (25, maxp1) containing the computed
Chebyshev moments. These can be passed on to an integration
over the same interval by passing this array as the second
element of the sequence wopts and passing infodict['momcom'] as
the first element.

If one of the integration limits is infinite, then a Fourier integral is
computed (assuming w neq 0). If full_output is 1 and a numerical error
is encountered, besides the error message attached to the output tuple,
a dictionary is also appended to the output tuple which translates the
error codes in the array ``info['ierlst']`` to English messages. The
output information dictionary contains the following entries instead of
'last', 'alist', 'blist', 'rlist', and 'elist':

'lst'
The number of subintervals needed for the integration (call it ``K_f``).

'rslst'
A rank-1 array of length M_f=limlst, whose first ``K_f`` elements
contain the integral contribution over the interval
``(a+(k-1)c, a+kc)`` where ``c = (2*floor(|w|) + 1) * pi / |w|``
and ``k=1,2,…,K_f``.

'erlst'
A rank-1 array of length ``M_f`` containing the error estimate
corresponding to the interval in the same position in
``infodict['rslist']``.

'ierlst'
A rank-1 integer array of length ``M_f`` containing an error flag
corresponding to the interval in the same position in
``infodict['rslist']``. See the explanation dictionary (last entry
in the output tuple) for the meaning of the codes.

Details of QUADPACK level routines

`quad` calls routines from the FORTRAN library QUADPACK. This section
provides details on the conditions for each routine to be called and a
short description of each routine. The routine called depends on
`weight`, `points` and the integration limits `a` and `b`.

```
================  ==============  ==========  =====================
QUADPACK routine  `weight`        `points`    infinite bounds
================  ==============  ==========  =====================
qagse             None            No          No
qagie             None            No          Yes
qagpe             None            Yes         No
qawoe             'sin', 'cos'    No          No
qawfe             'sin', 'cos'    No          either `a` or `b`
qawse             'alg*'          No          No
qawce             'cauchy'        No          No
================  ==============  ==========  =====================

The following provides a short desciption from [1]_ for each
routine.

qagse
    is an integrator based on globally adaptive interval
    subdivision in connection with extrapolation, which will
    eliminate the effects of integrand singularities of
    several types.
qagie
    handles integration over infinite intervals. The infinite range is
    mapped onto a finite interval and subsequently the same strategy as
    in ``QAGS`` is applied.
qagpe
    serves the same purposes as QAGS, but also allows the
    user to provide explicit information about the location
    and type of trouble-spots i.e. the abscissae of internal
    singularities, discontinuities and other difficulties of
    the integrand function.
qawoe
    is an integrator for the evaluation of
    :math:`\int^b_a \cos(\omega x)f(x)dx` or
    :math:`\int^b_a \sin(\omega x)f(x)dx`
    over a finite interval [a,b], where :math:`\omega` and :math:`f`
    are specified by the user. The rule evaluation component is based
    on the modified Clenshaw-Curtis technique

    An adaptive subdivision scheme is used in connection
    with an extrapolation procedure, which is a modification
    of that in ``QAGS`` and allows the algorithm to deal with
    singularities in :math:`f(x)`.
qawfe
    calculates the Fourier transform
    :math:`\int^\infty_a \cos(\omega x)f(x)dx` or
    :math:`\int^\infty_a \sin(\omega x)f(x)dx`
    for user-provided :math:`\omega` and :math:`f`. The procedure of
    ``QAWO`` is applied on successive finite intervals, and convergence
    acceleration by means of the :math:`\varepsilon`-algorithm is applied
    to the series of integral approximations.
qawse
    approximate :math:`\int^b_a w(x)f(x)dx`, with :math:`a < b` where
    :math:`w(x) = (x-a)^{\alpha}(b-x)^{\beta}v(x)` with
    :math:`\alpha,\beta > -1`, where :math:`v(x)` may be one of the
    following functions: :math:`1`, :math:`\log(x-a)`, :math:`\log(b-x)`,
    :math:`\log(x-a)\log(b-x)`.

    The user specifies :math:`\alpha`, :math:`\beta` and the type of the
    function :math:`v`. A globally adaptive subdivision strategy is
    applied, with modified Clenshaw-Curtis integration on those
    subintervals which contain `a` or `b`.
qawce
    compute :math:`\int^b_a f(x) / (x-c)dx` where the integral must be
    interpreted as a Cauchy principal value integral, for user specified
    :math:`c` and :math:`f`. The strategy is globally adaptive. Modified
    Clenshaw-Curtis integration is used on those intervals containing the
    point :math:`x = c`.

References
----------

.. [1] Piessens, Robert; de Doncker-Kapenga, Elise;
       Überhuber, Christoph W.; Kahaner, David (1983).
       QUADPACK: A subroutine package for automatic integration.
       Springer-Verlag.
       ISBN 978-3-540-12553-2.

Examples
--------
Calculate :math:`\int^4_0 x^2 dx` and compare with an analytic result

>>> from scipy import integrate
>>> x2 = lambda x: x**2
>>> integrate.quad(x2, 0, 4)
(21.333333333333332, 2.3684757858670003e-13)
>>> print(4**3 / 3.)  # analytical result
21.3333333333

Calculate :math:`\int^\infty_0 e^{-x} dx`

>>> invexp = lambda x: np.exp(-x)
>>> integrate.quad(invexp, 0, np.inf)
(1.0, 5.842605999138044e-11)

Calculate :math:`\int^1_0 a x \,dx` for :math:`a = 1, 3`
```

```
>>> f = lambda x, a: a*x
>>> y, err = integrate.quad(f, 0, 1, args=(1,))
>>> y
0.5
>>> y, err = integrate.quad(f, 0, 1, args=(3,))
>>> y
1.5

Calculate :math:`\int^1_0 x^2 + y^2 dx` with ctypes, holding
y parameter as 1::

    testlib.c =>
        double func(int n, double args[n]){
            return args[0]*args[0] + args[1]*args[1];}
    compile to library testlib.*

::

   from scipy import integrate
   import ctypes
   lib = ctypes.CDLL('/home/…/testlib.*') #use absolute path
   lib.func.restype = ctypes.c_double
   lib.func.argtypes = (ctypes.c_int,ctypes.c_double)
   integrate.quad(lib.func,0,1,(1))
   #(1.3333333333333333, 1.4802973661668752e-14)
   print((1.0**3/3.0 + 1.0) - (0.0**3/3.0 + 0.0)) #Analytic result
   # 1.3333333333333333

Be aware that pulse shapes and other sharp features as compared to the
size of the integration interval may not be integrated correctly using
this method. A simplified example of this limitation is integrating a
y-axis reflected step function with many zero values within the integrals
bounds.

>>> y = lambda x: 1 if x<=0 else 0
>>> integrate.quad(y, -1, 1)
(1.0, 1.1102230246251565e-14)
>>> integrate.quad(y, -1, 100)
(1.0000000002199108, 1.0189464580163188e-08)
>>> integrate.quad(y, -1, 10000)
(0.0, 0.0)
```

[ ]:
```
from scipy.integrate import quad # To compute a definite integral
from scipy.special import jv # Bessel function
%timeit res = quad(np.sin, 0, np.pi)
print(quad(np.sin, 0, np.pi))
```

```
#help(quad)
print(quad(lambda x: jv(2.5, x), 0, 10)) # Integrate the Bessel function of␣
↪order 2.5 between 0 and 10
```

```
21.4 µs ± 1.7 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
(2.0, 2.220446049250313e-14)
(0.8209075326034347, 1.1793289815399173e-08)
```

We now want to evaluate:

$$\int_0^1 1 + 2x + 3x^2 dx$$

[ ]:
```
# We want here integrate a user-defined function (here polynome) between 0 and 1
def f(x, a, b, c):
    """ Returning a 2nd order polynome """
    return a + b * x + c * x**2
def f123(x):
    return 1 + 2 * x + 3 * x**2
def f123b(x, a=1, b=2, c=3):
    return a + b * x + c * x**2
%timeit I = quad(f, 0, 1, args=(1,2,3)) # args will send 1, 2, 3 to f
%timeit I = quad(f123, 0, 1)
%timeit I = quad(f123b, 0, 1)
I = quad(f, 0, 1, args=(1,2,3)) # args will send 1, 2, 3 to f
print(I)
Integ = I[0]
print(Integ)
```

```
5.28 µs ± 15.3 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
4.96 µs ± 28.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
5.18 µs ± 19.8 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
(3.0, 3.3306690738754696e-14)
3.0
```

1.0.3 Interpolations

[ ]:
```
from scipy.interpolate import interp1d, interp2d, splrep, splev, griddata
```

[ ]:
```
#help(scipy.interpolate) # a huge one...
help(interp1d)
```

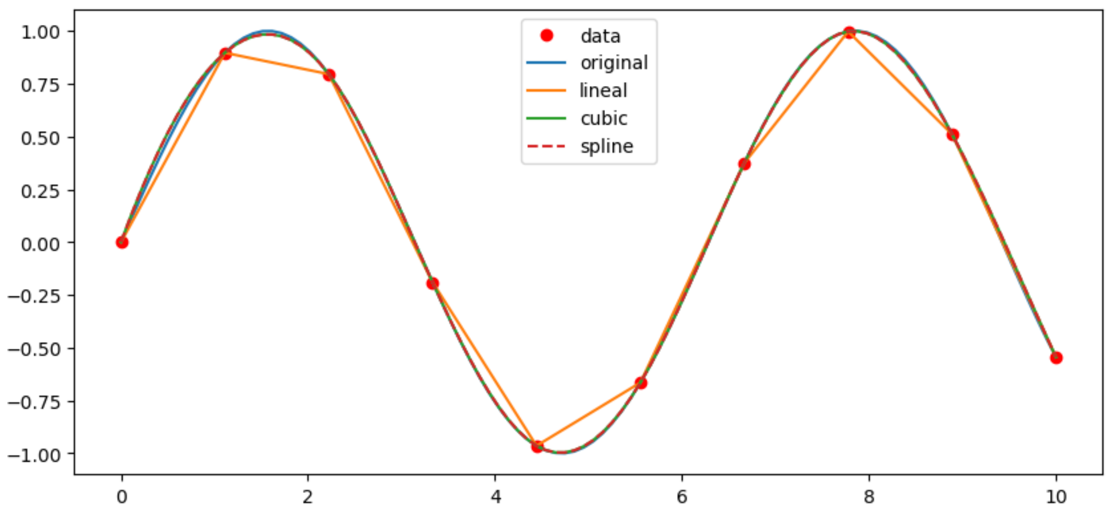
```
Help on class interp1d in module scipy.interpolate.interpolate:

class interp1d(scipy.interpolate.polyint._Interpolator1D)
 |  interp1d(x, y, kind='linear', axis=-1, copy=True, bounds_error=None,
fill_value=nan, assume_sorted=False)
 |
 |  Interpolate a 1-D function.
 |
 |  `x` and `y` are arrays of values used to approximate some function f:
 |  ``y = f(x)``. This class returns a function whose call method uses
 |  interpolation to find the value of new points.
 |
 |  Parameters
 |  ----------
 |  x : (N,) array_like
 |      A 1-D array of real values.
 |  y : (…,N,…) array_like
 |      A N-D array of real values. The length of `y` along the interpolation
 |      axis must be equal to the length of `x`.
 |  kind : str or int, optional
 |      Specifies the kind of interpolation as a string or as an integer
 |      specifying the order of the spline interpolator to use.
 |      The string has to be one of 'linear', 'nearest', 'nearest-up', 'zero',
 |      'slinear', 'quadratic', 'cubic', 'previous', or 'next'. 'zero',
 |      'slinear', 'quadratic' and 'cubic' refer to a spline interpolation of
 |      zeroth, first, second or third order; 'previous' and 'next' simply
 |      return the previous or next value of the point; 'nearest-up' and
 |      'nearest' differ when interpolating half-integers (e.g. 0.5, 1.5)
 |      in that 'nearest-up' rounds up and 'nearest' rounds down. Default
 |      is 'linear'.
 |  axis : int, optional
 |      Specifies the axis of `y` along which to interpolate.
 |      Interpolation defaults to the last axis of `y`.
 |  copy : bool, optional
 |      If True, the class makes internal copies of x and y.
 |      If False, references to `x` and `y` are used. The default is to copy.
 |  bounds_error : bool, optional
 |      If True, a ValueError is raised any time interpolation is attempted on
 |      a value outside of the range of x (where extrapolation is
 |      necessary). If False, out of bounds values are assigned `fill_value`.
 |      By default, an error is raised unless ``fill_value="extrapolate"``.
 |  fill_value : array-like or (array-like, array_like) or "extrapolate",
optional
 |      - if a ndarray (or float), this value will be used to fill in for
 |        requested points outside of the data range. If not provided, then
 |        the default is NaN. The array-like must broadcast properly to the
 |        dimensions of the non-interpolation axes.
 |      - If a two-element tuple, then the first element is used as a
 |        fill value for ``x_new < x[0]`` and the second element is used for
 |        ``x_new > x[-1]``. Anything that is not a 2-element tuple (e.g.,
 |        list or ndarray, regardless of shape) is taken to be a single
 |        array-like argument meant to be used for both bounds as
 |        ``below, above = fill_value, fill_value``.
 |
 |        .. versionadded:: 0.17.0
 |      - If "extrapolate", then points outside the data range will be
 |        extrapolated.
 |
 |        .. versionadded:: 0.17.0
 |  assume_sorted : bool, optional
 |      If False, values of `x` can be in any order and they are sorted first.
 |      If True, `x` has to be an array of monotonically increasing values.
 |
 |  Attributes
 |  ----------
 |  fill_value
 |
 |  Methods
 |  -------
 |  __call__
 |
 |  See Also
 |  --------
 |  splrep, splev
 |      Spline interpolation/smoothing based on FITPACK.
 |  UnivariateSpline : An object-oriented wrapper of the FITPACK routines.
 |  interp2d : 2-D interpolation
 |
 |  Notes
 |  -----
 |  Calling `interp1d` with NaNs present in input values results in
 |  undefined behaviour.
 |
 |  Input values `x` and `y` must be convertible to `float` values like
 |  `int` or `float`.
 |
 |  If the values in `x` are not unique, the resulting behavior is
 |  undefined and specific to the choice of `kind`, i.e., changing
 |  `kind` will change the behavior for duplicates.
 |
 |
 |  Examples
 |  --------
 |  >>> import matplotlib.pyplot as plt
 |  >>> from scipy import interpolate
 |  >>> x = np.arange(0, 10)
 |  >>> y = np.exp(-x/3.0)
 |  >>> f = interpolate.interp1d(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()
```

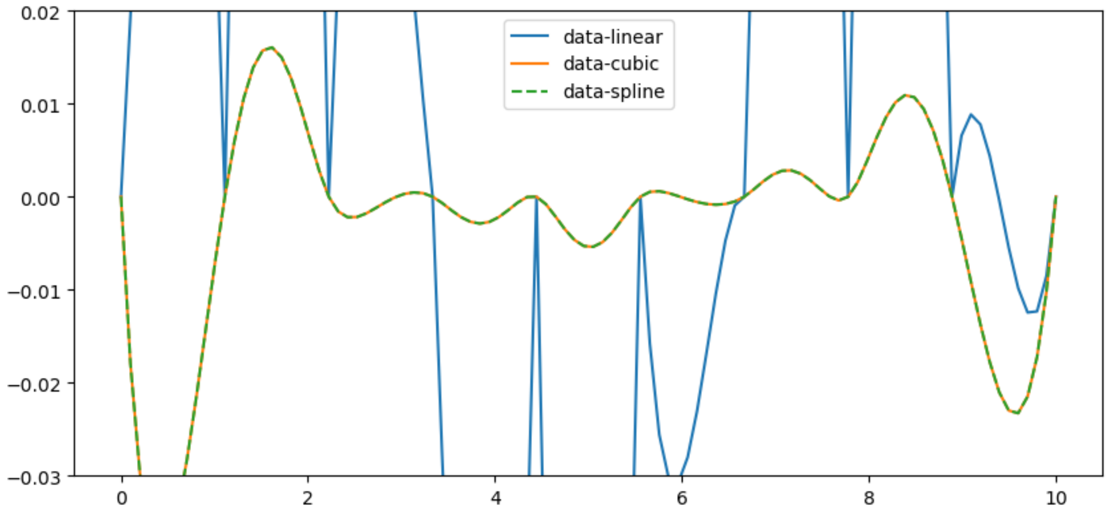
```
 |
 |  Method resolution order:
 |      interp1d
 |      scipy.interpolate.polyint._Interpolator1D
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, x, y, kind='linear', axis=-1, copy=True, bounds_error=None,
fill_value=nan, assume_sorted=False)
 |      Initialize a 1-D linear interpolation class.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  fill_value
 |      The fill value.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from scipy.interpolate.polyint._Interpolator1D:
 |
 |  __call__(self, x)
 |      Evaluate the interpolant
 |
 |      Parameters
 |      ----------
 |      x : array_like
 |          Points to evaluate the interpolant at.
 |
 |      Returns
 |      -------
 |      y : array_like
 |          Interpolated values. Shape is determined by replacing
 |          the interpolation axis in the original array with the shape of x.
 |
 |      Notes
 |      -----
 |      Input values `x` must be convertible to `float` values like `int`
 |      or `float`.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from scipy.interpolate.polyint._Interpolator1D:
```

```
 |
 |  dtype
```

[ ]:
```
x = np.linspace(0, 10, 10)
y = np.sin(x)
f = interp1d(x, y) # this creates a function that can be call at any␣
 ↪interpolate point
f2 = interp1d(x, y, kind='cubic') # The same but using cubic interpolation
tck = splrep(x, y, s=0) # This initiate the spline interpolating function,␣
 ↪finding the B-spline representation of 1-D curve.
# tck is a sequence of length 3 returned by `splrep` or `splprep` containing␣
 ↪the knots, coefficients, and degree of the spline.
f3 = lambda x: splev(x, tck) # Evaluate the B-spline or its derivatives.
```

[ ]:
```
# Defining the high resolution mesh
xfine = np.linspace(0, 10, 100)
yfine = np.sin(xfine)
# Plot to compare the results
fig, (ax1, ax2) = plt.subplots(2, figsize=(10,10))

ax1.plot(x, y, 'or', label='data')
ax1.plot(xfine, yfine, label='original')
ax1.plot(xfine, f(xfine), label='lineal')
ax1.plot(xfine, f2(xfine), label='cubic')
ax1.plot(xfine, f3(xfine), label='spline', ls='--')
ax1.legend(loc=9)

ax2.plot(xfine, (yfine - f(xfine)), label='data-linear')
ax2.plot(xfine, (yfine - f2(xfine)), label='data-cubic')
ax2.plot(xfine, (yfine - f3(xfine)), label='data-spline', ls='--')
ax2.legend(loc='best')
ax2.set_ylim((-0.03, 0.02));
```





```
[ ]: x0 = 3.5
     print('{} {} {} {}'.format(np.sin(x0), f(x0), f2(x0), f3(x0)))
```

```
-0.35078322768961984 -0.3066303359834792 -0.34959725240218925
-0.3495972524021892
```

2D interpolation

```
[ ]: # Defining a 2D-function
     def func(x, y):
         return x * (1+x) * np.cos(4*np.pi*x) * np.sin(4*np.pi*y**2)**2
```

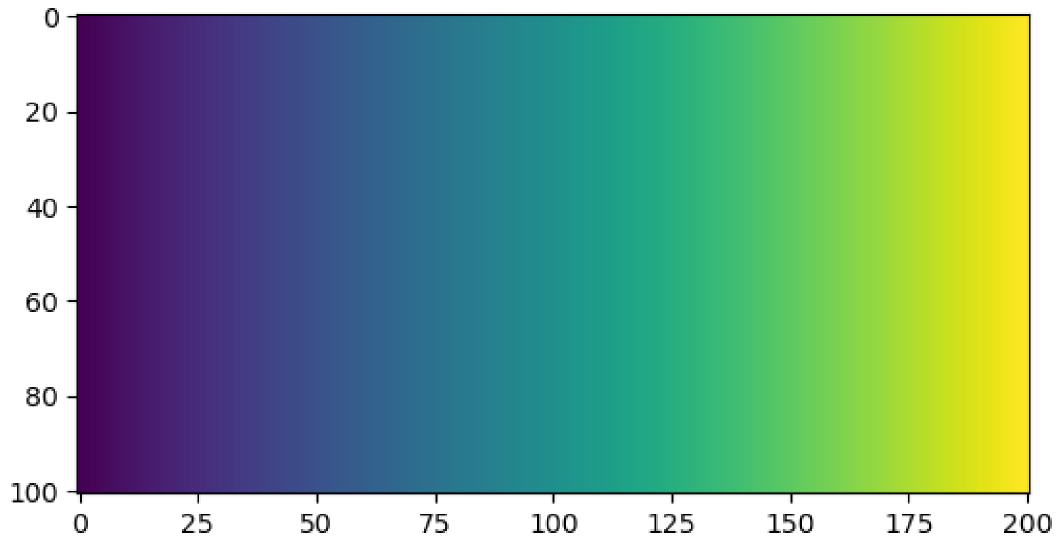
```
[ ]: # Initializing a 2D coordinate grid. Note the use of j to specify that the end␣
      ↪point is included.
     grid_y, grid_x = np.mgrid[0:1:101j, 0:1:201j]
```

```
[ ]: print(grid_x)
     print(grid_y)
```

```
[[0.    0.005 0.01  … 0.99  0.995 1.   ]
 [0.    0.005 0.01  … 0.99  0.995 1.   ]
 [0.    0.005 0.01  … 0.99  0.995 1.   ]
 …
 [0.    0.005 0.01  … 0.99  0.995 1.   ]
 [0.    0.005 0.01  … 0.99  0.995 1.   ]
 [0.    0.005 0.01  … 0.99  0.995 1.   ]]
[[0.   0.   0.   … 0.   0.   0.  ]
 [0.01 0.01 0.01 … 0.01 0.01 0.01]
 [0.02 0.02 0.02 … 0.02 0.02 0.02]
 …
 [0.98 0.98 0.98 … 0.98 0.98 0.98]
 [0.99 0.99 0.99 … 0.99 0.99 0.99]
 [1.   1.   1.   … 1.   1.   1.  ]]
```

```
[ ]: plt.imshow(grid_x)
```

```
[ ]: <matplotlib.image.AxesImage at 0x7fb41fcedac0>
```



```
[ ]: # Generating 1000 x 2 points randomly
     points = np.random.rand(1000, 2)
     print(points)
     values = func(points[:,0], points[:,1])
     print(np.min(points), np.max(points))
```

```
[[0.58285163 0.02017694]
```

```
 [0.98469475 0.73882545]
 [0.1508262  0.30640621]
 …
 [0.76712557 0.46335445]
 [0.85832259 0.30865531]
 [0.54425035 0.38268348]]
0.0004222369284355709 0.999759691718544
```

[ ]:
```
# griddata is the 2D-interpolating method. We want to obtain values on (grid_x,␣
↪grid_y) points,
# using "points" and "values".
%timeit grid_z0 = griddata(points, values, (grid_x, grid_y), method='nearest')
%timeit grid_z1 = griddata(points, values, (grid_x, grid_y), method='linear')
%timeit grid_z2 = griddata(points, values, (grid_x, grid_y), method='cubic')
```

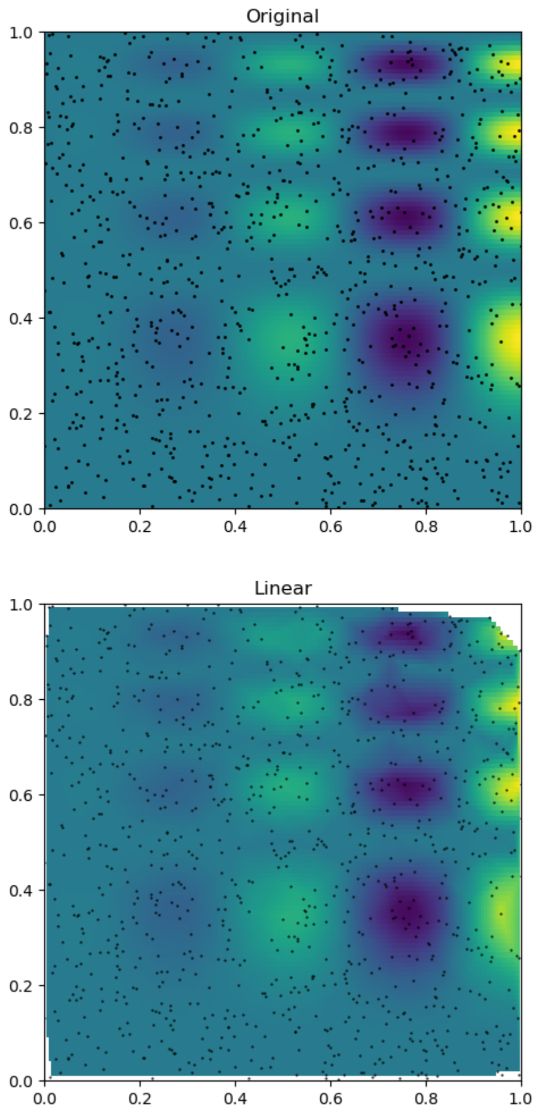
```
23.5 ms ± 1.56 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
9.94 ms ± 452 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
14.9 ms ± 476 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

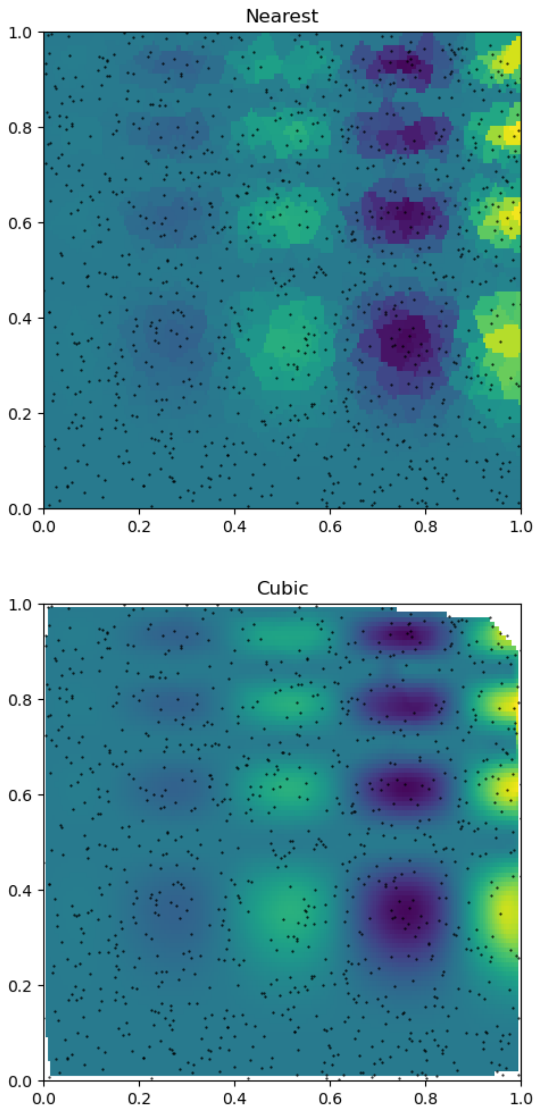
[ ]:
```
print(np.min(values), np.max(values))
```

```
-1.3216183431172221 1.9716371839860183
```

[ ]:
```
# 4 subplots
grid_z0 = griddata(points, values, (grid_x, grid_y), method='nearest')
grid_z1 = griddata(points, values, (grid_x, grid_y), method='linear')
grid_z2 = griddata(points, values, (grid_x, grid_y), method='cubic')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 12))
extent = (0, 1, 0, 1)
origin = 'lower'
ax1.imshow(func(grid_x, grid_y), extent=extent, interpolation='none',
           origin=origin, vmin=-1.4, vmax=2.0)
ax1.plot(points[:,0], points[:,1], 'ko', ms=1)
ax1.set_title('Original')

ax2.imshow(grid_z0, extent=extent, interpolation='none',
           origin=origin, vmin=-1.4, vmax=2.0)
ax2.plot(points[:,0], points[:,1], 'k.', ms=1)
ax2.set_title('Nearest')

ax3.imshow(grid_z1, extent=extent, interpolation='none',
           origin=origin, vmin=-1.4, vmax=2.0)
ax3.plot(points[:,0], points[:,1], 'k.', ms=1)
ax3.set_title('Linear')

ax4.imshow(grid_z2, extent=extent, interpolation='none',
           origin=origin, vmin=-1.4, vmax=2.0)
ax4.plot(points[:,0], points[:,1], 'k.', ms=1)
```

```
ax4.set_title('Cubic');
```





[ ]:
```
print(grid_z0[10,10], grid_z1[10,10], grid_z2[10,10])
```

```
0.00033012283816004335 0.0006611507622104852 0.0006647105471534465
```

1.0.4 Linear algebra

Scipy is able to deal with matrices, solving linear equations, solving linear least-squares problems and pseudo-inverses, finding eigenvalues and eigenvectors, and more, see here: https://docs.scipy.org/doc/scipy/tutorial/linalg.html

1.0.5 Data fit

```
[ ]: from scipy.optimize import curve_fit # this is used to adjust a set of data
```

```
[ ]: help(curve_fit)
```

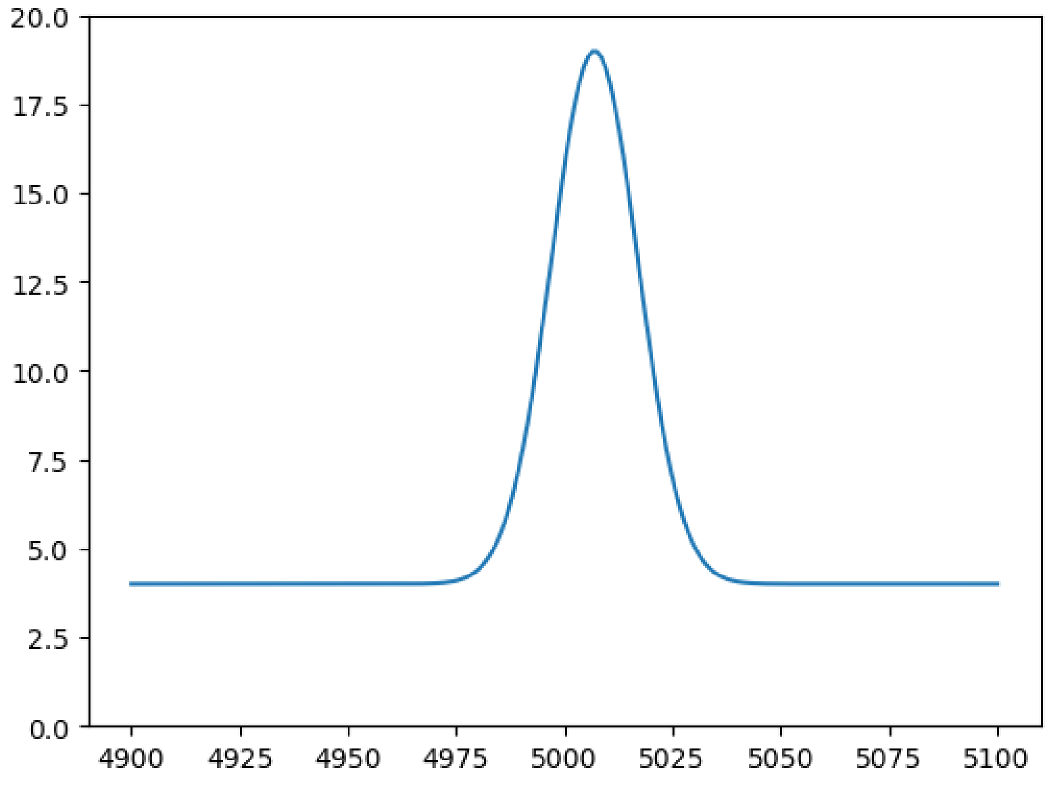
```
Help on function curve_fit in module scipy.optimize._minpack_py:

curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
check_finite=True, bounds=(-inf, inf), method=None, jac=None, *,
full_output=False, **kwargs)
    Use non-linear least squares to fit a function, f, to data.

    Assumes ``ydata = f(xdata, *params) + eps``.

    Parameters
    ----------
    f : callable
        The model function, f(x, …). It must take the independent
        variable as the first argument and the parameters to fit as
        separate remaining arguments.
    xdata : array_like or object
        The independent variable where the data is measured.
        Should usually be an M-length sequence or an (k,M)-shaped array for
        functions with k predictors, but can actually be any object.
    ydata : array_like
        The dependent data, a length M array - nominally ``f(xdata, …)``.
    p0 : array_like, optional
        Initial guess for the parameters (length N). If None, then the
        initial values will all be 1 (if the number of parameters for the
        function can be determined using introspection, otherwise a
        ValueError is raised).
    sigma : None or M-length sequence or MxM array, optional
        Determines the uncertainty in `ydata`. If we define residuals as
        ``r = ydata - f(xdata, *popt)``, then the interpretation of `sigma`
        depends on its number of dimensions:

            - A 1-D `sigma` should contain values of standard deviations of
              errors in `ydata`. In this case, the optimized function is
              ``chisq = sum((r / sigma) ** 2)``.

            - A 2-D `sigma` should contain the covariance matrix of
              errors in `ydata`. In this case, the optimized function is
              ``chisq = r.T @ inv(sigma) @ r``.

              .. versionadded:: 0.19

        None (default) is equivalent of 1-D `sigma` filled with ones.
    absolute_sigma : bool, optional
        If True, `sigma` is used in an absolute sense and the estimated
parameter
        covariance `pcov` reflects these absolute values.

        If False (default), only the relative magnitudes of the `sigma` values
matter.
        The returned parameter covariance matrix `pcov` is based on scaling
        `sigma` by a constant factor. This constant is set by demanding that the
        reduced `chisq` for the optimal parameters `popt` when using the
        *scaled* `sigma` equals unity. In other words, `sigma` is scaled to
        match the sample variance of the residuals after the fit. Default is
False.
        Mathematically,
        ``pcov(absolute_sigma=False) = pcov(absolute_sigma=True) *
chisq(popt)/(M-N)``
    check_finite : bool, optional
        If True, check that the input arrays do not contain nans of infs,
        and raise a ValueError if they do. Setting this parameter to
        False may silently produce nonsensical results if the input arrays
        do contain nans. Default is True.
    bounds : 2-tuple of array_like, optional
        Lower and upper bounds on parameters. Defaults to no bounds.
        Each element of the tuple must be either an array with the length equal
        to the number of parameters, or a scalar (in which case the bound is
        taken to be the same for all parameters). Use ``np.inf`` with an
        appropriate sign to disable bounds on all or some parameters.

        .. versionadded:: 0.17
    method : {'lm', 'trf', 'dogbox'}, optional
        Method to use for optimization. See `least_squares` for more details.
        Default is 'lm' for unconstrained problems and 'trf' if `bounds` are
        provided. The method 'lm' won't work when the number of observations
        is less than the number of variables, use 'trf' or 'dogbox' in this
        case.

        .. versionadded:: 0.17
    jac : callable, string or None, optional
        Function with signature ``jac(x, …)`` which computes the Jacobian
        matrix of the model function with respect to parameters as a dense
        array_like structure. It will be scaled according to provided `sigma`.
        If None (default), the Jacobian will be estimated numerically.
        String keywords for 'trf' and 'dogbox' methods can be used to select
        a finite difference scheme, see `least_squares`.

        .. versionadded:: 0.18
    full_output : boolean, optional
        If True, this function returns additioal information: `infodict`,
        `mesg`, and `ier`.

        .. versionadded:: 1.9
    **kwargs
        Keyword arguments passed to `leastsq` for ``method='lm'`` or
        `least_squares` otherwise.

    Returns
    -------
    popt : array
        Optimal values for the parameters so that the sum of the squared
        residuals of ``f(xdata, *popt) - ydata`` is minimized.
    pcov : 2-D array
        The estimated covariance of popt. The diagonals provide the variance
        of the parameter estimate. To compute one standard deviation errors
        on the parameters use ``perr = np.sqrt(np.diag(pcov))``.

        How the `sigma` parameter affects the estimated covariance
        depends on `absolute_sigma` argument, as described above.

        If the Jacobian matrix at the solution doesn't have a full rank, then
        'lm' method returns a matrix filled with ``np.inf``, on the other hand
        'trf'  and 'dogbox' methods use Moore-Penrose pseudoinverse to compute
        the covariance matrix.
    infodict : dict (returned only if `full_output` is True)
        a dictionary of optional outputs with the keys:

        ``nfev``
            The number of function calls. Methods 'trf' and 'dogbox' do not
            count function calls for numerical Jacobian approximation,
            as opposed to 'lm' method.
        ``fvec``
            The function values evaluated at the solution.
        ``fjac``
            A permutation of the R matrix of a QR
            factorization of the final approximate
            Jacobian matrix, stored column wise.
            Together with ipvt, the covariance of the
            estimate can be approximated.
            Method 'lm' only provides this information.
        ``ipvt``
            An integer array of length N which defines
            a permutation matrix, p, such that
            fjac*p = q*r, where r is upper triangular
            with diagonal elements of nonincreasing
            magnitude. Column j of p is column ipvt(j)
            of the identity matrix.
            Method 'lm' only provides this information.
        ``qtf``
            The vector (transpose(q) * fvec).
            Method 'lm' only provides this information.

        .. versionadded:: 1.9
    mesg : str (returned only if `full_output` is True)
        A string message giving information about the solution.

        .. versionadded:: 1.9
    ier : int (returnned only if `full_output` is True)
        An integer flag. If it is equal to 1, 2, 3 or 4, the solution was
        found. Otherwise, the solution was not found. In either case, the
        optional output variable `mesg` gives more information.

        .. versionadded:: 1.9

    Raises
    ------
    ValueError
        if either `ydata` or `xdata` contain NaNs, or if incompatible options
        are used.

    RuntimeError
        if the least-squares minimization fails.

    OptimizeWarning
        if covariance of the parameters can not be estimated.

    See Also
    --------
    least_squares : Minimize the sum of squares of nonlinear functions.
    scipy.stats.linregress : Calculate a linear least squares regression for
                             two sets of measurements.

    Notes
    -----
    Users should ensure that inputs `xdata`, `ydata`, and the output of `f`
    are ``float64``, or else the optimization may return incorrect results.

    With ``method='lm'``, the algorithm uses the Levenberg-Marquardt algorithm
    through `leastsq`. Note that this algorithm can only deal with
    unconstrained problems.

    Box constraints can be handled by methods 'trf' and 'dogbox'. Refer to
    the docstring of `least_squares` for more information.

    Examples
    --------
```

```
>>> import matplotlib.pyplot as plt
>>> from scipy.optimize import curve_fit

>>> def func(x, a, b, c):
...     return a * np.exp(-b * x) + c

Define the data to be fit with some noise:

>>> xdata = np.linspace(0, 4, 50)
>>> y = func(xdata, 2.5, 1.3, 0.5)
>>> rng = np.random.default_rng()
>>> y_noise = 0.2 * rng.normal(size=xdata.size)
>>> ydata = y + y_noise
>>> plt.plot(xdata, ydata, 'b-', label='data')

Fit for the parameters a, b, c of the function `func`:

>>> popt, pcov = curve_fit(func, xdata, ydata)
>>> popt
array([2.56274217, 1.37268521, 0.47427475])
>>> plt.plot(xdata, func(xdata, *popt), 'r-',
...          label='fit: a=%5.3f, b=%5.3f, c=%5.3f' % tuple(popt))

Constrain the optimization to the region of ``0 <= a <= 3``,
``0 <= b <= 1`` and ``0 <= c <= 0.5``:

>>> popt, pcov = curve_fit(func, xdata, ydata, bounds=(0, [3., 1., 0.5]))
>>> popt
array([2.43736712, 1.        , 0.34463856])
>>> plt.plot(xdata, func(xdata, *popt), 'g--',
...          label='fit: a=%5.3f, b=%5.3f, c=%5.3f' % tuple(popt))

>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.legend()
>>> plt.show()
```

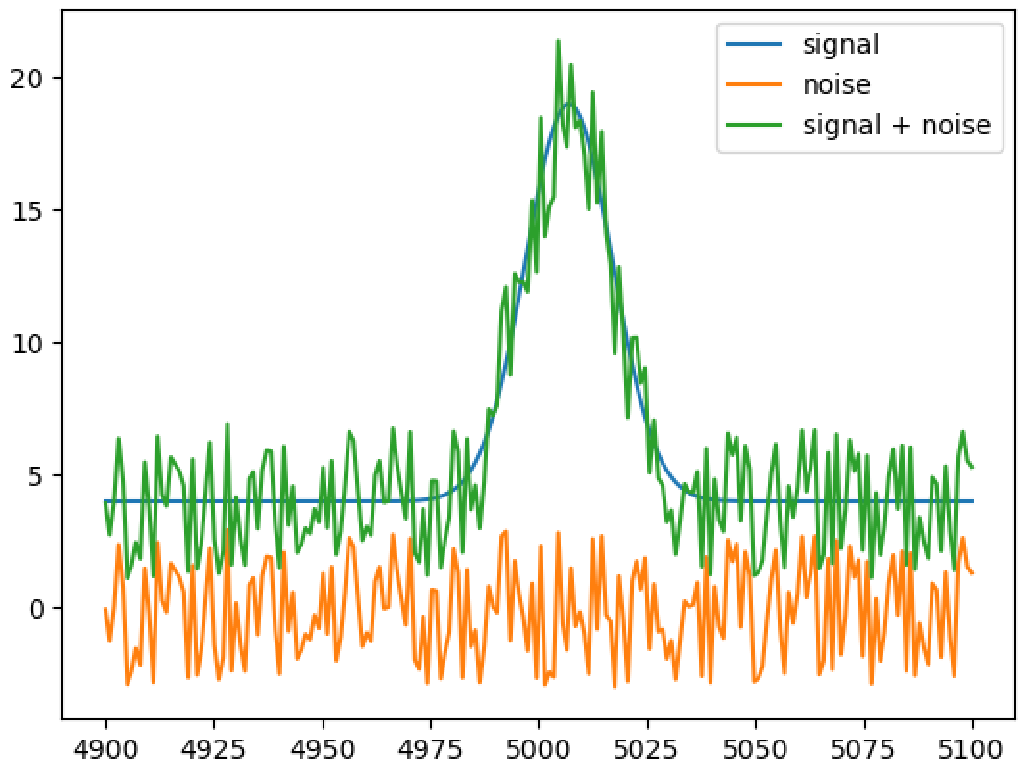
[ ]:
```
def gauss(x, A, B, C, S):
    # This is a gaussian function.
    return A + B*np.exp(-1 * (x - C)**2 / (2 * S**2))
```

[ ]:
```
# We define the parameters used to generate the signal (gaussian at lambda=5007)
N_lam = 200
A = 4.
B = 15.
Lam0 = 5007.
```

```
Sigma = 10.
# We define a wavelength range
lam = np.linspace(4900, 5100, N_lam)
# Computing the signal
fl = gauss(lam, A, B, Lam0, Sigma)
f, ax =plt.subplots()
ax.plot(lam, fl)
ax.set_ylim(0,20);
```



[ ]:
```
SN = 5. # Signal/Noise
noise = B / SN * (np.random.rand(N_lam)*2 - 1)
fl2 = fl + noise
f, ax =plt.subplots()
ax.plot(lam, fl, label='signal')
ax.plot(lam, noise, label='noise')
ax.plot(lam, fl2, label='signal + noise')
ax.legend(loc='best');
```



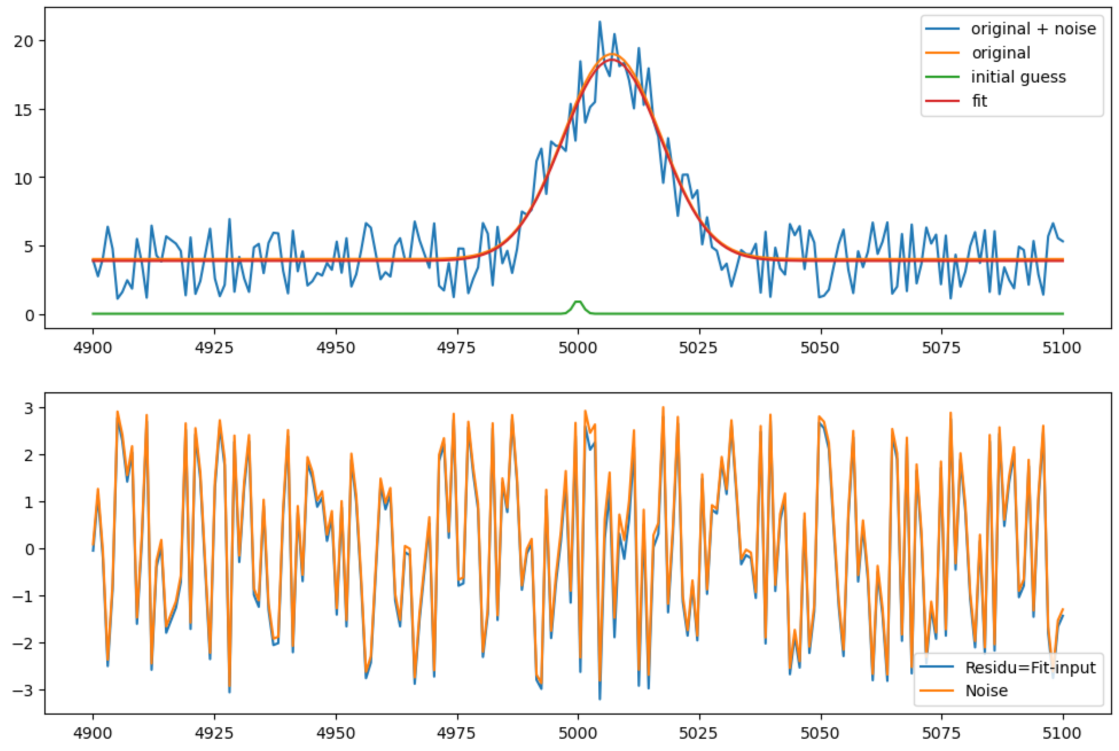
[ ]:
```
# Initial guess:
A_i = 0.
B_i = 1.
Lam0_i = 5000.
Sigma_i = 1.
fl_init = gauss(lam, A_i, B_i, Lam0_i, Sigma_i)
error = np.ones_like(lam) * np.mean(np.abs(noise)) # We define the error (the␣
 ↪same on each pixel of the spectrum)
```

[ ]:
```
# fitting the noisy data with the gaussian function, using the initial guess␣
 ↪and the errors
fit, covar = curve_fit(gauss, lam, fl2, [A_i, B_i, Lam0_i, Sigma_i], error)
print('{0:.2f} {1:5.2f} {2:.2f} {3:5.2f} {4:5.2f}'.format(A_i, B_i, Lam0_i,␣
 ↪Sigma_i, B_i*Sigma_i))
print('{0:.2f} {1:5.2f} {2:.2f} {3:5.2f} {4:5.2f}'.format(A, B, Lam0, Sigma,␣
 ↪B*Sigma))
print('{0[0]:.2f} {0[1]:5.2f} {0[2]:5.2f} {0[3]:.2f}  {1:5.2f}'.format(fit,␣
 ↪fit[1]*fit[3]))
```

```
0.00  1.00 5000.00  1.00  1.00
4.00 15.00 5007.00 10.00 150.00
3.86 14.72 5007.00 10.10  148.68
```

```
[ ]: # Computing the fit on the lambdas
     fl_fit = gauss(lam, fit[0], fit[1], fit[2], fit[3])
```

```
[ ]: fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

     ax1.plot(lam, fl2, label='original + noise')
     ax1.plot(lam, fl, label='original')
     ax1.plot(lam, fl_init, label='initial guess')
     ax1.plot(lam, fl_fit, label='fit')
     ax1.legend()

     ax2.plot(lam, fl_fit - fl2, label='Residu=Fit-input')
     ax2.plot(lam, -noise, label='Noise')
     ax2.legend();
```



```
[ ]: # Integrating using the Simpson method the gaussian (without the continuum)
     print(simps(fl - A, lam))
     print(simps(fl2 - fit[0], lam))
     print(simps(fl_fit - fit[0], lam))
```

```
375.99424119465004
371.86873765195986
372.6950491068127
```

```
[ ]: khi_sq = (((fl2-fl_fit) / error)**2).sum() # The problem here is to determine␣
       ↪the error...
     khi_sq_red = khi_sq / (len(lam) - 4 - 1) # reduced khi_sq = khi_sq / (N -␣
       ↪free_params - 1)
     print('khi^2={}, khi^2_reduced={}'.format(khi_sq, khi_sq_red))
```

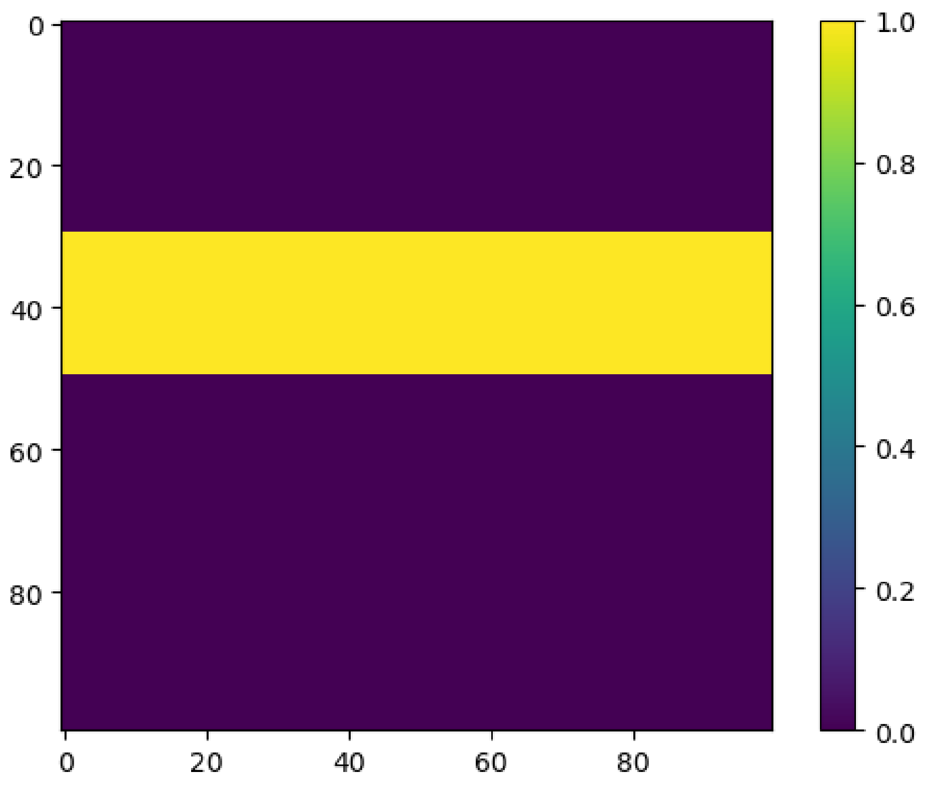
```
khi^2=260.59689695712905, khi^2_reduced=1.3363943433698926
```

1.0.6 Convolution

More information there: http://docs.scipy.org/doc/scipy/reference/tutorial/ndimage.html

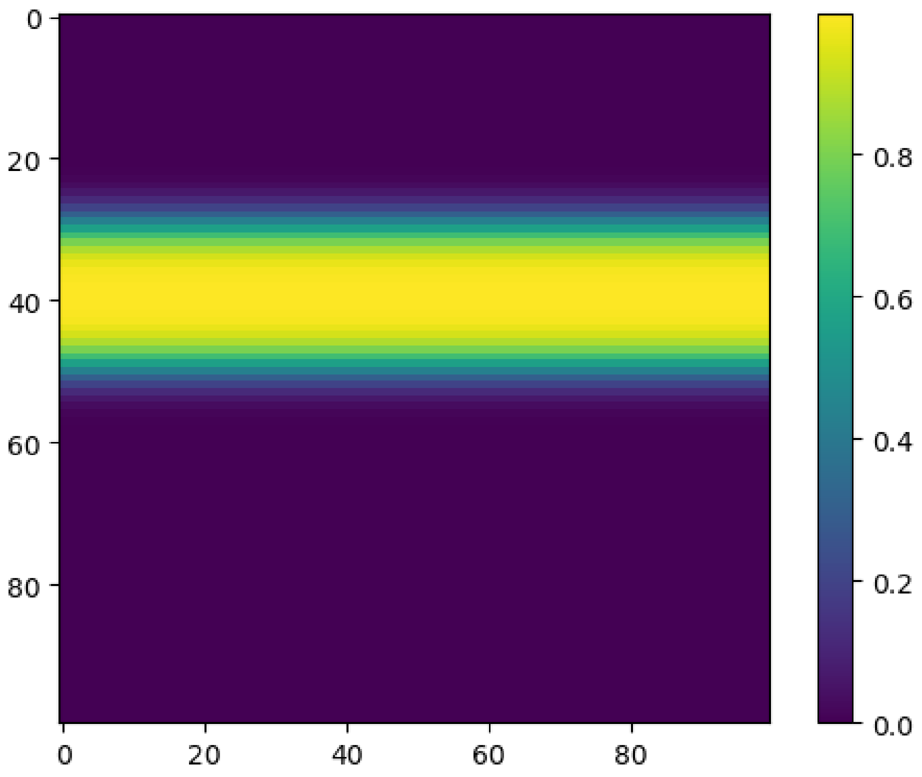
```
[ ]: # Let's define an image representing a long slit of width 10 pixels
     slit = np.zeros((100, 100))
     slit[30:50, :] = 1
```

```
[ ]: plt.imshow(slit)
     plt.colorbar();
```

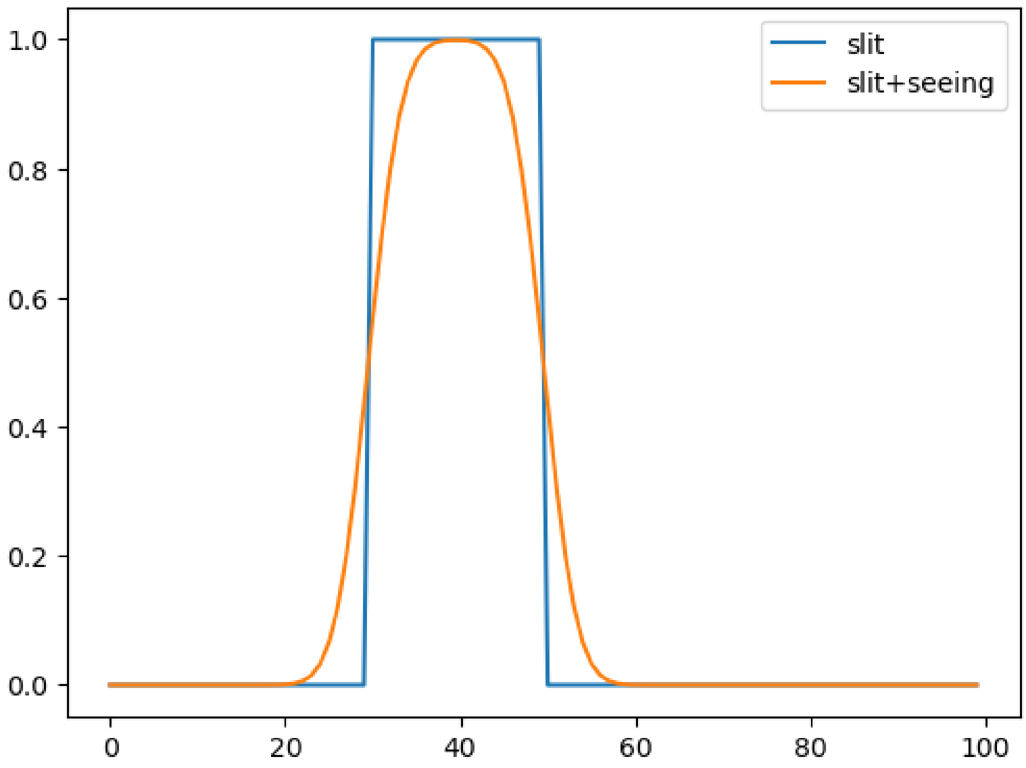


```
[ ]: # This is the routine to apply a gaussian convolution
     from scipy.ndimage import gaussian_filter
```

```
[ ]: slit_seeing = gaussian_filter(slit, 3) # Convolve with a gaussian, 3 is the␣
      ↪standard deviation in pixels
     plt.imshow(slit_seeing)
     plt.colorbar();
```



```
[ ]: f, ax =plt.subplots()
     ax.plot(slit[:,50], label='slit') # original slit
     ax.plot(slit_seeing[:,50], label='slit+seeing') # slit with seeing
     ax.legend(loc='best');
```



```
[ ]: # Check that the slit transmission is conserved:
     print(simps(slit[:,50]), simps(slit_seeing[:,50]))
```

```
20.0 20.0
```

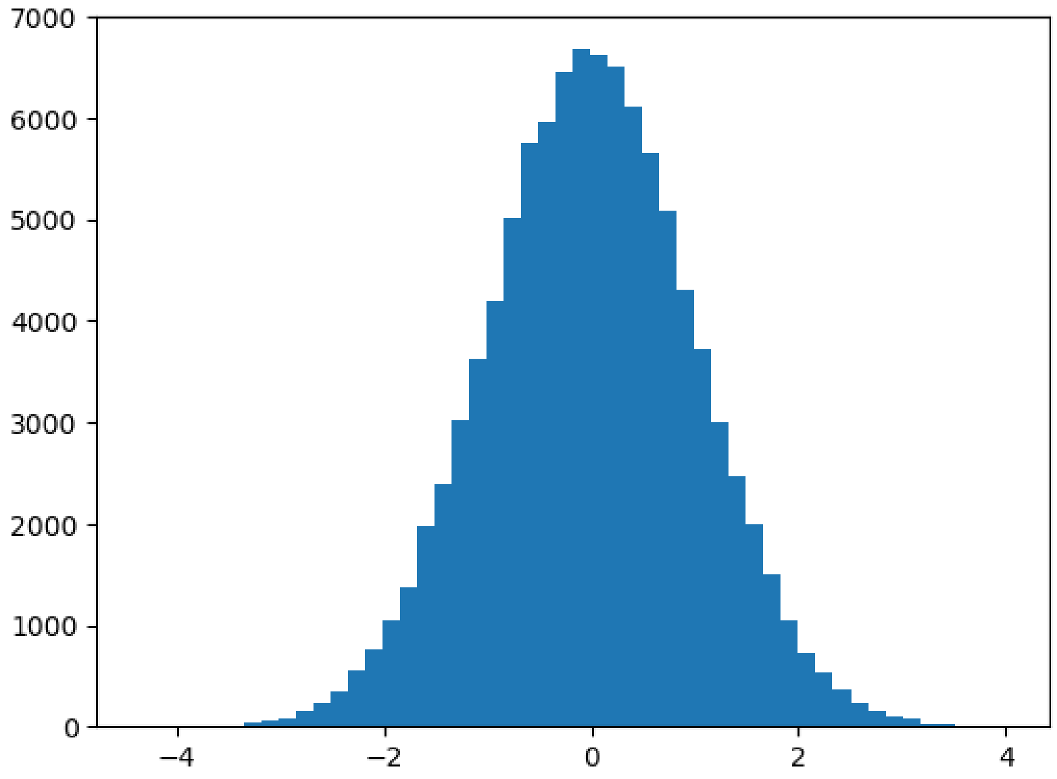
1.0.7 Quantiles

```
[ ]: from scipy.stats.mstats import mquantiles
```

```
[ ]: #help(mquantiles)
```

```
[ ]: data = np.random.randn(100000)
```

```
[ ]: f, ax = plt.subplots()
     ax.hist(data, bins=50);
```



[ ]:
```
mquantiles(data, [0.16, 0.5,0.84]) # should return something close to -1, 0, 1␣
  ↪(the stv of the normal distribution)
```

[ ]:
```
array([-0.99458906,  0.00134218,  0.9971658 ])
```

[ ]:
```
data = np.array([[   6.,    7.,    1.],
                 [  47.,   15.,    2.],
                 [  49.,   36.,    3.],
                 [  15.,   39.,    4.],
                 [  42.,   40., -999.],
                 [  41.,   41., -999.],
                 [   7., -999., -999.],
                 [  39., -999., -999.],
                 [  43., -999., -999.],
                 [  40., -999., -999.],
                 [  36., -999., -999.]])
```

[ ]:
```
mq = mquantiles(data, axis=0, limit=(0, 50))
print(mq)
```

```
[[19.2  14.6   1.45]
 [40.   37.5   2.5 ]]
 [42.8  40.05  3.55]]
```

1.0.8 Input/Output

Scipy has many modules, classes, and functions available to read data from and write data to a variety of file formats.

Including MATLAB and IDL files. See http://docs.scipy.org/doc/scipy/reference/io.html