intro_Scipy

January 10, 2021

[1]:
```
# 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])
```

```
Mon Oct 26 19:35:18 2020
3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)]
```

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
- http://docs.scipy.org/doc/scipy/reference/tutorial/

[2]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

[3]:
```
import scipy # This imports a lot of scipy stuff, but not the "important"␣
↪modules
```

1.0.1 Some usefull methods

[4]:
```
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)
```

```
789 ns ± 29.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
993 ns ± 24.7 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
14.2 ns ± 0.537 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
14.3 ns ± 0.334 ns per loop (mean ± std. dev. of 7 runs, 100000000 loops each)
716430.6890623764 362880 3628800
```

[5]:
```
from scipy import constants as cst
print(cst.astronomical_unit) # A lot of constants
from scipy.constants import codata # a lot more, with units. From NIST
print('{} {}'.format(codata.value('proton mass'), codata.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

[7]:
```
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 numpy:

trapz(y, x=None, dx=1.0, axis=-1)
    Integrate along the given axis using the composite trapezoidal rule.

    Integrate `y` (`x`) along given axis.

    Parameters
    ----------
    y : array_like
        Input array to integrate.
    x : array_like, optional
        The sample points corresponding to the `y` values. If `x` is None,
        the sample points are assumed to be evenly spaced `dx` apart. The
        default is None.
    dx : scalar, optional
        The spacing between sample points when `x` is None. The default is 1.
    axis : int, optional
        The axis along which to integrate.
```

```
    Returns
    -------
    trapz : float
        Definite integral as approximated by trapezoidal rule.

    See Also
    --------
    numpy.cumsum

    Notes
    -----
    Image [2]_ illustrates trapezoidal rule -- y-axis locations of points
    will be taken from `y` array, by default x-axis distances between
    points will be 1.0, alternatively they can be provided with `x` array
    or with `dx` scalar.  Return value will be equal to combined area under
    the red lines.


    References
    ----------
    .. [1] Wikipedia page: https://en.wikipedia.org/wiki/Trapezoidal_rule

    .. [2] Illustration image:
https://en.wikipedia.org/wiki/File:Composite_trapezoidal_rule_illustration.png

    Examples
    --------
    >>> np.trapz([1,2,3])
    4.0
    >>> np.trapz([1,2,3], x=[4,6,8])
    8.0
    >>> np.trapz([1,2,3], dx=2)
    8.0
    >>> a = np.arange(6).reshape(2, 3)
    >>> a
    array([[0, 1, 2],
           [3, 4, 5]])
    >>> np.trapz(a, axis=0)
    array([1.5, 2.5, 3.5])
    >>> np.trapz(a, axis=1)
    array([2.,  8.])

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

cumtrapz(y, x=None, dx=1.0, axis=-1, initial=None)
```

```
Cumulatively integrate y(x) using the composite trapezoidal rule.

Parameters
----------
y : array_like
    Values to integrate.
x : array_like, optional
    The coordinate to integrate along.  If None (default), use spacing `dx`
    between consecutive elements in `y`.
dx : float, optional
    Spacing between elements of `y`.  Only used if `x` is None.
axis : int, optional
    Specifies the axis to cumulate.  Default is -1 (last axis).
initial : scalar, optional
    If given, insert this value at the beginning of the returned result.
    Typically this value should be 0.  Default is None, which means no
    value at ``x[0]`` is returned and `res` has one element less than `y`
    along the axis of integration.

Returns
-------
res : ndarray
    The result of cumulative integration of `y` along `axis`.
    If `initial` is None, the shape is such that the axis of integration
    has one less value than `y`.  If `initial` is given, the shape is equal
    to that of `y`.

See Also
--------
numpy.cumsum, numpy.cumprod
quad: adaptive quadrature using QUADPACK
romberg: adaptive Romberg quadrature
quadrature: adaptive Gaussian quadrature
fixed_quad: fixed-order Gaussian quadrature
dblquad: double integrals
tplquad: triple integrals
romb: integrators for sampled data
ode: ODE integrators
odeint: ODE integrators

Examples
--------
>>> from scipy import integrate
>>> import matplotlib.pyplot as plt

>>> x = np.linspace(-2, 2, num=20)
>>> y = x
>>> y_int = integrate.cumtrapz(y, x, initial=0)
```

```
    >>> plt.plot(x, y_int, 'ro', x, y[0] + 0.5 * x**2, 'b-')
    >>> plt.show()

------------------------------------------------------------------------------
--
Help on function simps in module scipy.integrate.quadrature:

simps(y, x=None, dx=1, axis=-1, even='avg')
    Integrate y(x) using samples along the given axis and the composite
    Simpson's rule.  If x is None, spacing of dx is assumed.

    If there are an even number of samples, N, then there are an odd
    number of intervals (N-1), but Simpson's rule requires an even number
    of intervals.  The parameter 'even' controls how this is handled.

    Parameters
    ----------
    y : array_like
        Array to be integrated.
    x : array_like, optional
        If given, the points at which `y` is sampled.
    dx : int, optional
        Spacing of integration points along axis of `y`. Only used when
        `x` is None. Default is 1.
    axis : int, optional
        Axis along which to integrate. Default is the last axis.
    even : str {'avg', 'first', 'last'}, optional
        'avg' : Average two results:1) use the first N-2 intervals with
                  a trapezoidal rule on the last interval and 2) use the last
                  N-2 intervals with a trapezoidal rule on the first interval.

        'first' : Use Simpson's rule for the first N-2 intervals with
                a trapezoidal rule on the last interval.

        'last' : Use Simpson's rule for the last N-2 intervals with a
               trapezoidal rule on the first interval.

    See Also
    --------
    quad: adaptive quadrature using QUADPACK
    romberg: adaptive Romberg quadrature
    quadrature: adaptive Gaussian quadrature
    fixed_quad: fixed-order Gaussian quadrature
    dblquad: double integrals
    tplquad: triple integrals
    romb: integrators for sampled data
    cumtrapz: cumulative integration for sampled data
    ode: ODE integrators
```

```
    odeint: ODE integrators

    Notes
    -----
    For an odd number of samples that are equally spaced the result is
    exact if the function is a polynomial of order 3 or less.  If
    the samples are not equally spaced, then the result is exact only
    if the function is a polynomial of order 2 or less.

    Examples
    --------
    >>> from scipy import integrate
    >>> x = np.arange(0, 10)
    >>> y = np.arange(0, 10)

    >>> integrate.simps(y, x)
    40.5

    >>> y = np.power(x, 3)
    >>> integrate.simps(y, x)
    1642.5
    >>> integrate.quad(lambda x: x**3, 0, 9)[0]
    1640.25

    >>> integrate.simps(y, x, even='first')
    1644.5
```

[8]:
```
dir(scipy.integrate)
```

[8]:
```
['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',
 'absolute_import',
 'complex_ode',
 'cumtrapz',
 'dblquad',
 'division',
 'fixed_quad',
 'lsoda',
 'newton_cotes',
 'nquad',
 'ode',
 'odeint',
 'odepack',
 'print_function',
 'quad',
 'quad_explain',
 'quad_vec',
 'quadpack',
 'quadrature',
 'romb',
 'romberg',
 'simps',
 'solve_bvp',
 'solve_ivp',
 'test',
 'tplquad',
 'trapz',
 'vode']
```

[9]:
```
# Defining x and y
x = np.linspace(0, np.pi, 100)
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))
```

```
15.2 µs ± 402 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
82.9 µs ± 2.12 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
1.9998321638939929
1.9999999690165366
15.3 µs ± 504 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
80.4 µs ± 1.57 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
1.9796508112164835
1.9995487365804032
```

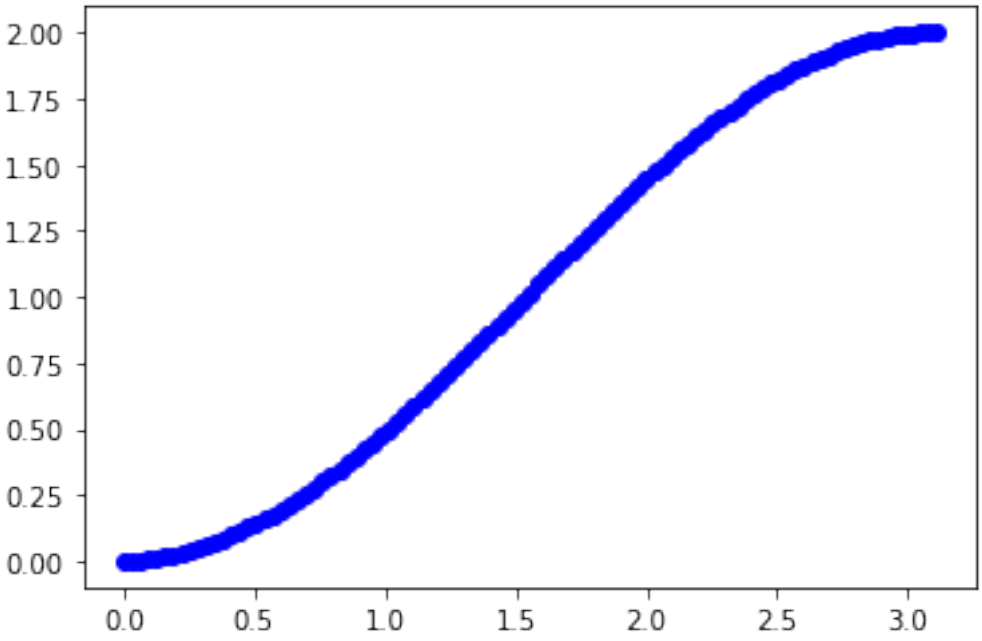
[10]:
```
# Cumulative integrale
cum = cumtrapz(np.abs(y), x)
print(len(x), len(cum), cum)
```

```
10 9 [0.05969378 0.23157515 0.4949127  0.81794403 1.16170678 1.48473811
 1.74807566 1.91995704 1.97965081]
```

[13]:
```
# 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');
```

```
100 99
```



[15]:
```
from scipy.integrate import quad # To compute a definite integral
help(quad)
```

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

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.
    Run scipy.integrate.quad_explain() for more 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.
epsrel : float or int, optional
    Relative error tolerance.
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
simps : 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.

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)

>>> 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)
```

[14]:
```
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(3.5, x), 0, 10)) # Integrate the Bessel function of␣
↪order 2.5 between 0 and 10
```

```
26 µs ± 1.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
(2.0, 2.220446049250313e-14)
(0.7551384098083599, 9.487330562236267e-10)
```

We now want to evaluate:

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

[19]:
```
# 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)
```

```
13 µs ± 193 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
11.8 µs ± 199 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
12.4 µs ± 158 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
(3.0, 3.3306690738754696e-14)
3.0
```

1.0.3 Interpolations

[20]:
```
from scipy.interpolate import interp1d, interp2d, splrep, splev, griddata
```

[22]:
```
#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.
 |
 |  Note that calling `interp1d` with NaNs present in input values results in
 |  undefined behaviour.
 |
 |  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
 |      ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic',
 |      'previous', 'next', where '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) or as an integer specifying the order of the spline
 |      interpolator to use.
 |      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
 |
 |  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 1D 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)
```

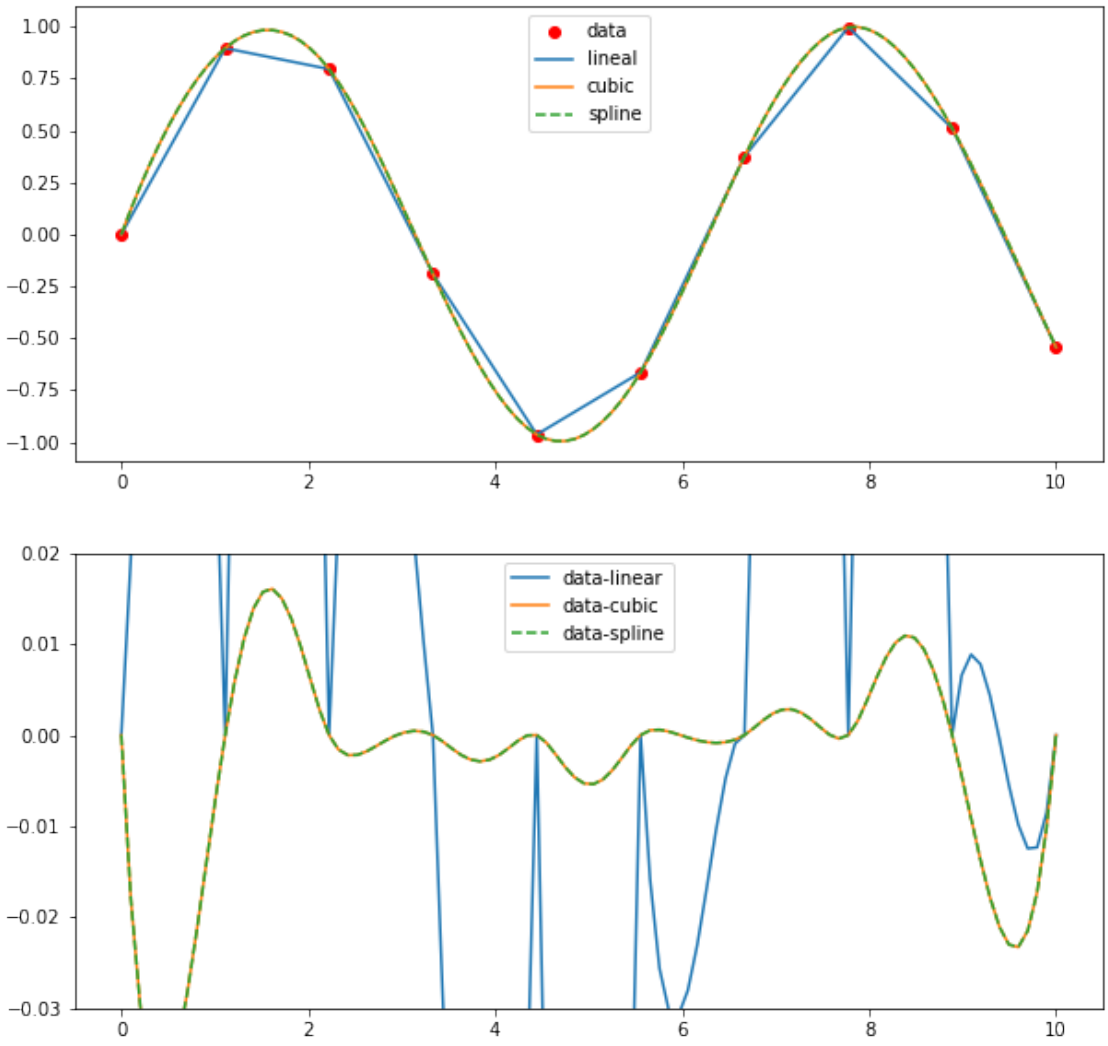
```
|      Evaluate the interpolant
|
|      Parameters
|      ----------
|      x : array_like
|          Points to evaluate the interpolant at.
|
|      Returns
|      -------
|      y : array_like
|          Interpolated values. Shape is determined by replacing
|          the interpolation axis in the original array with the shape of x.
|
|  ----------------------------------------------------------------------
|  Data descriptors inherited from scipy.interpolate.polyint._Interpolator1D:
|
|  dtype
```

[32]:
```
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.
```

[33]:
```
# 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, 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));
```



[27]:
```
x0 = 3.5
print('{} {} {} {}'.format(np.sin(x0), f(x0), f2(x0), f3(x0)))
```

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

2D interpolation

[34]:
```
# 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
```

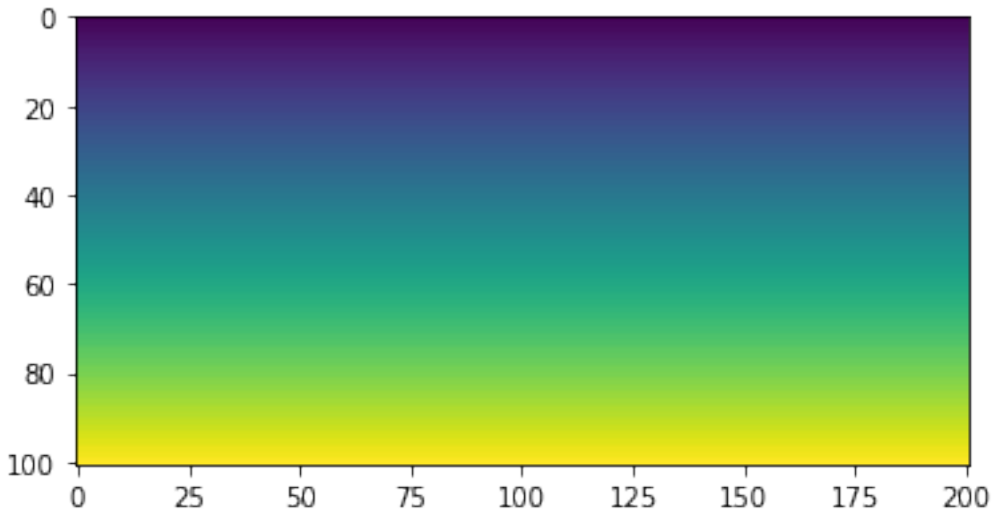
[49]:
```
# Initializing a 2D coordinate grid. Note the use of j to specify that the end␣
 ↪point is included.
grid_y, grid_x = np.mgrid[0:1:101j, 0:1:201j]
```

[50]:
```
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.  ]]
```

[52]:
```
plt.imshow(grid_y)
```

[52]:
```
<matplotlib.image.AxesImage at 0x7fdcd94ee390>
```



[53]:
```
# 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))
```

```
[[0.57013156 0.6477694 ]
 [0.97560368 0.06030687]
```

```
 [0.72364169 0.1861295 ]
 …
 [0.99592176 0.58364619]
 [0.49757841 0.70437339]
 [0.81283584 0.36777008]]
2.5241965496136665e-05 0.9999617833674705
```

[54]:
```
# 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')
```

```
19 ms ± 799 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
12.5 ms ± 112 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
18.3 ms ± 629 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

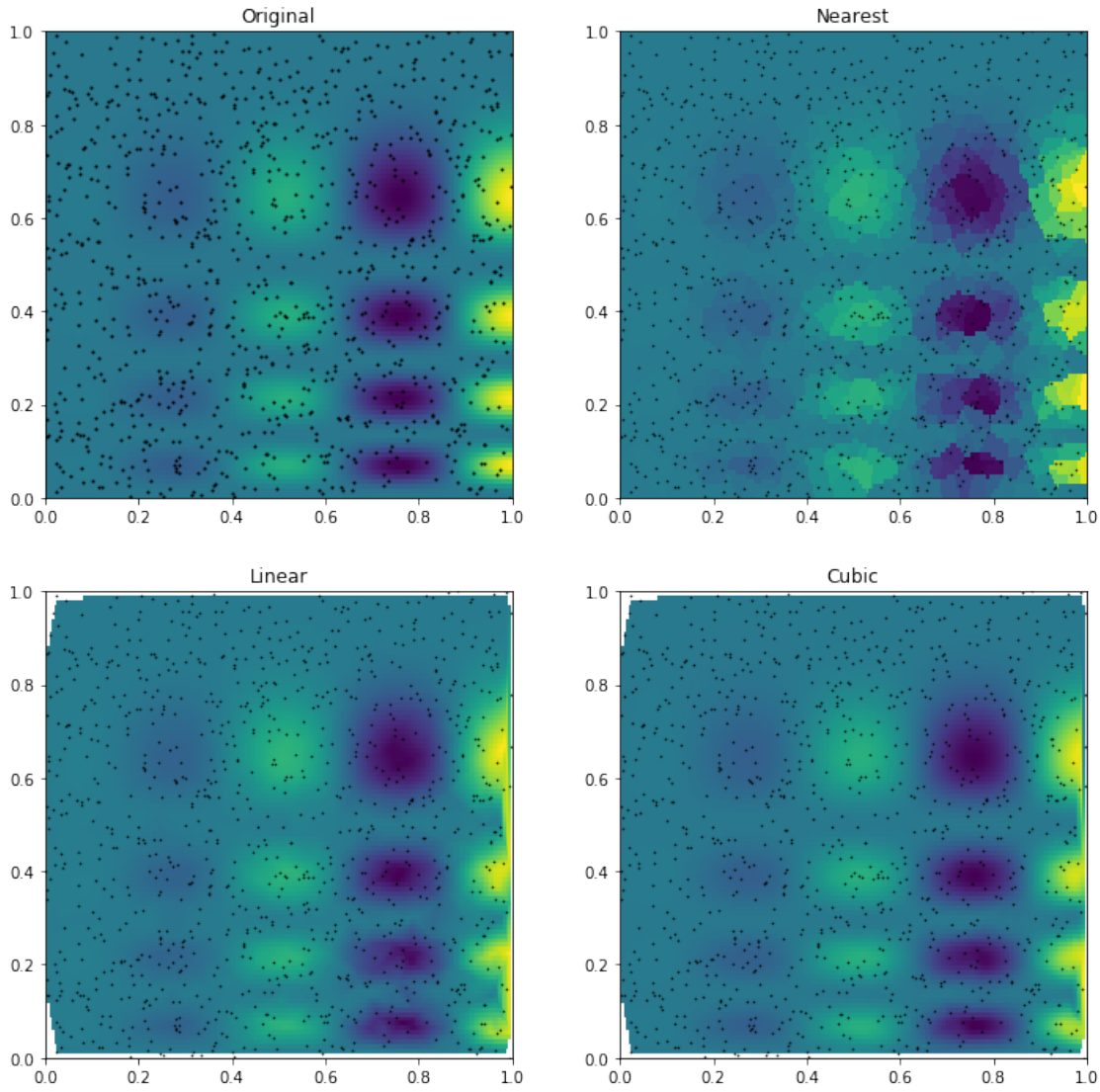
[57]:
```
# 4 subplots
grid_z0 = griddata(points, values, (grid_x, grid_y), method='nearest')
grid_z1 = griddata(points, values, (grid_x, grid_y), method='linear')
grid_z2 = griddata(points, values, (grid_x, grid_y), method='cubic')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 12))

ax1.imshow(func(grid_x, grid_y), extent=(0,1,0,1), interpolation='none',
           origin='upper')
ax1.plot(points[:,0], points[:,1], 'ko', ms=1)
ax1.set_title('Original')

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

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

ax4.imshow(grid_z2, extent=(0,1,0,1), interpolation='none',
           origin='upper')
ax4.plot(points[:,0], points[:,1], 'k.', ms=1)
ax4.set_title('Cubic');
```



[62]:
```
print(grid_z0[10,10], grid_z1[10,10], grid_z2[10,10])
```

```
0.0016214165709755681 0.0010009910497726163 0.000723788958478942
```

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: http://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html

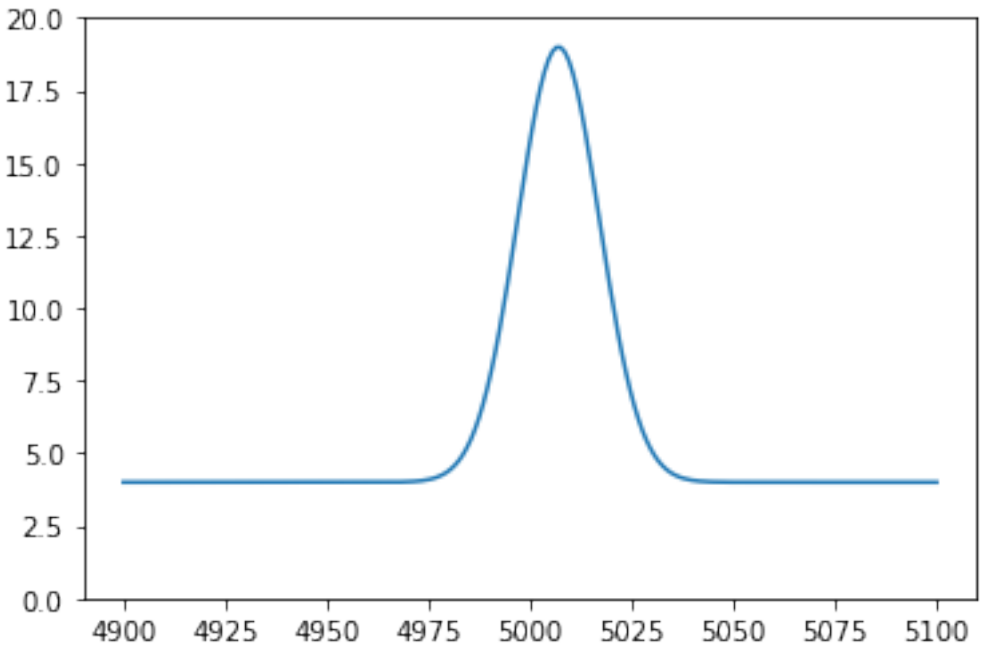
1.0.5 Data fit

[63]:
```
from scipy.optimize import curve_fit # this is used to adjust a set of data
```

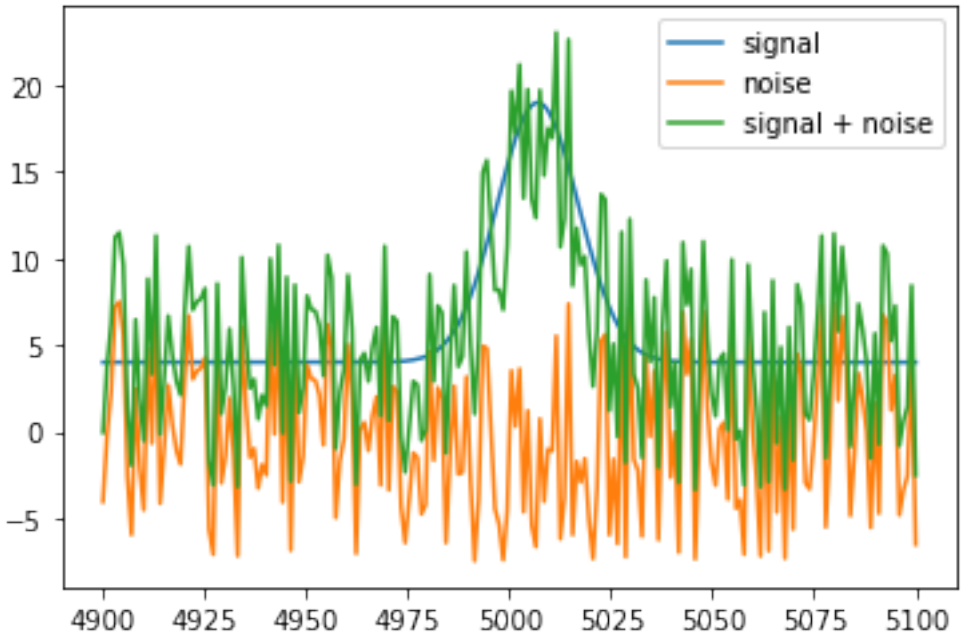
[66]:
```
#help(curve_fit)
```

[65]:
```
def gauss(x, A, B, C, S):
    # This is a gaussian function.
    return A + B*np.exp(-1 * (x - C)**2 / (2 * S**2))
```

[68]:
```
# 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);
```



[69]:
```
SN = 2. # 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');
```



[70]:
```
# 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)
```

[78]:
```
# 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
4.12 13.42 5006.93 8.81  118.21
```

[79]:
```
# Computing the fit on the lambdas
fl_fit = gauss(lam, fit[0], fit[1], fit[2], fit[3])
```

[80]:
```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))

ax1.plot(lam, fl, label='original')
ax1.plot(lam, fl2, label='original + noise')
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();
```

[81]:
```
# 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
302.9876590127733
296.31956281143505
```

[82]:
```
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=259.9509045778959, khi^2_reduced=1.333081561937928
```

1.0.6 Multivariate estimation

[83]:
```
from scipy import stats
```

[90]:
```
def measure(n):
    """Measurement model, return two coupled measurements."""
    m1 = np.random.normal(size=n)
    m2 = np.random.normal(scale=0.5, size=n)
    return m1+m2, m1-m2
```

[91]:
```
m1, m2 = measure(2000)
xmin = m1.min()
xmax = m1.max()
ymin = m2.min()
ymax = m2.max()
print(xmin, xmax, ymin, ymax)
```

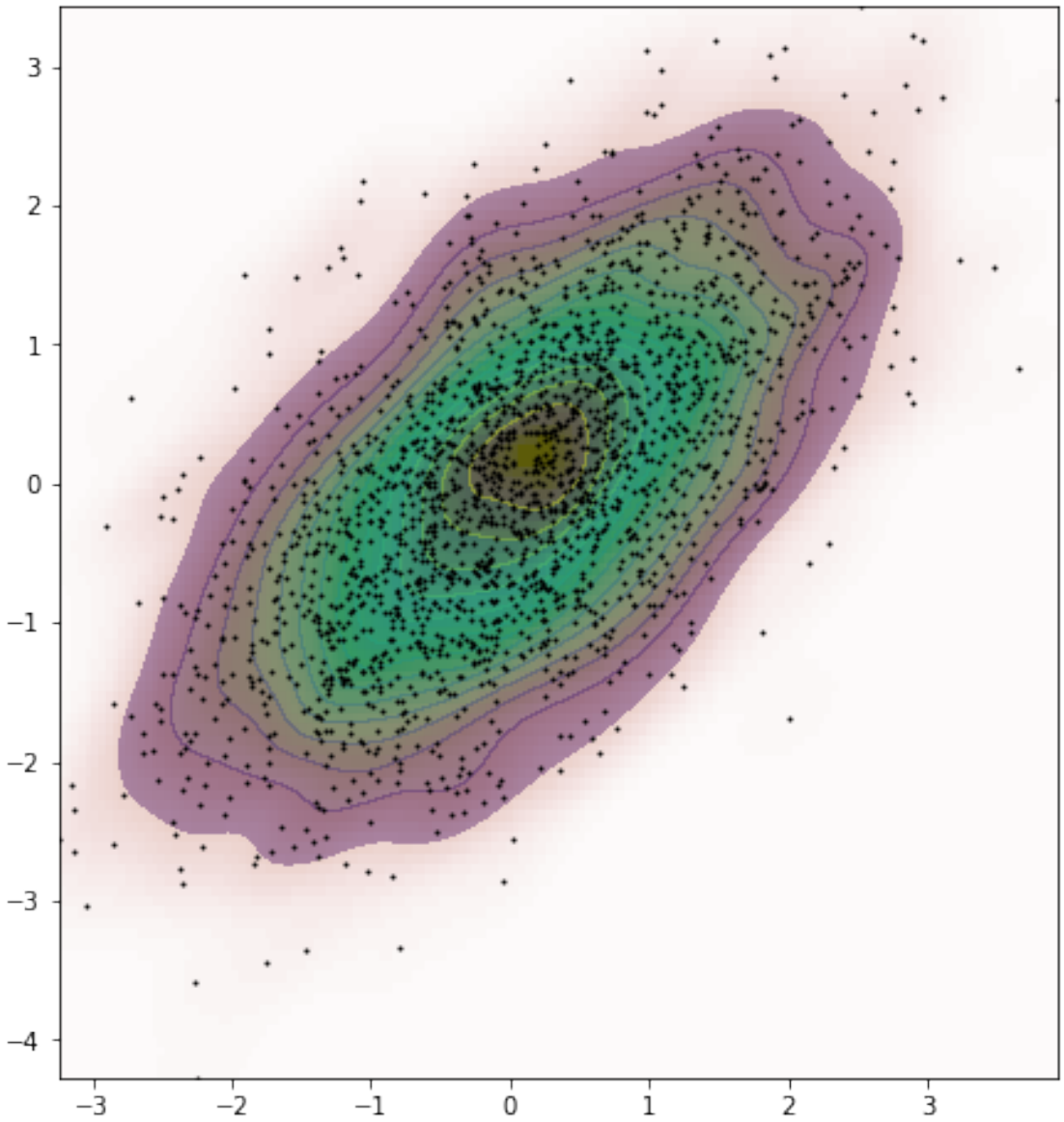
```
-3.241128358076106 3.932762086917964 -4.274334312847188 3.4360457576504504
```

[92]:
```
X, Y = np.mgrid[xmin:xmax:150j, ymin:ymax:100j]
positions = np.vstack([X.ravel(), Y.ravel()])
values = np.vstack([m1, m2])
kernel = stats.gaussian_kde(values)
Z = np.reshape(kernel.evaluate(positions).T, X.shape)
print(Z.shape)
```

```
(150, 100)
```

[102]:
```
fig, ax = plt.subplots(figsize=(12, 8))
ax.imshow(np.rot90(Z), cmap=plt.cm.gist_earth_r, extent=[xmin, xmax, ymin,␣
↪ymax], origin='upper')
ax.plot(m1, m2, 'k.', markersize=2)
ax.set_xlim([xmin, xmax])
```

```
ax.set_ylim([ymin, ymax])
levels = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11, 0.
↪12, 0.13, 0.14, 0.15]
cs = ax.contourf(X, Y, Z, levels=levels, alpha=0.4); # I dont't know what those␣
↪levels mean... but it works fine!
```



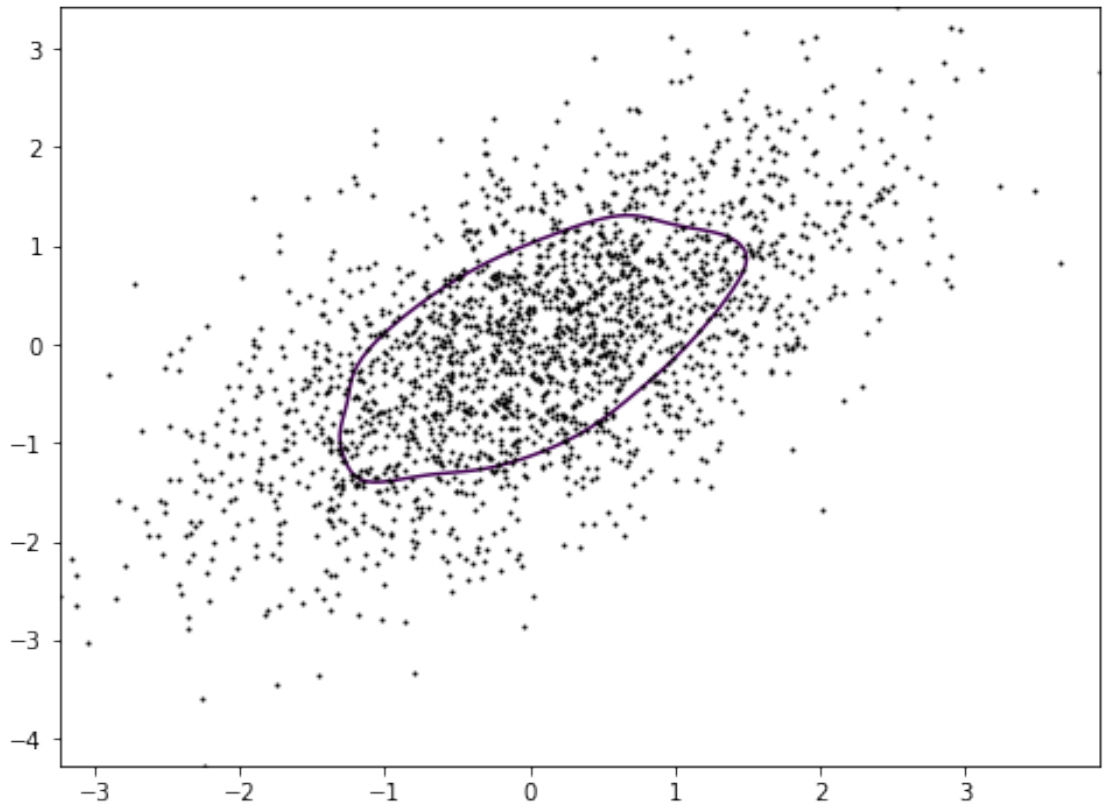
[88]:
```
# We save the contour paths in a list
paths = []
for collec in cs.collections:
    try:
        paths.append(collec.get_paths()[0])
```

```
    except:
        pass
```

[89]:
```
# Looking for the number of points inside each contour
print(len(m1))
for level, path in zip(levels, paths):
    print('level {0:4.2f} contains {1:2.0f}% of the data'.format(level,
                                path.contains_points(list(zip(m1, m2))).sum() /␣
 ↪float(len(m1))*100))
```

```
2000
level 0.01 contains 95% of the data
level 0.02 contains 88% of the data
level 0.03 contains 82% of the data
level 0.04 contains 74% of the data
level 0.05 contains 67% of the data
level 0.06 contains 61% of the data
level 0.07 contains 54% of the data
level 0.08 contains 47% of the data
level 0.09 contains 40% of the data
level 0.10 contains 33% of the data
level 0.11 contains 26% of the data
level 0.12 contains 19% of the data
level 0.13 contains 13% of the data
level 0.14 contains  6% of the data
level 0.15 contains  0% of the data
```

[104]:
```
fig, ax = plt.subplots(figsize=(8, 6))
ax.plot(m1, m2, 'k.', markersize=2)
ax.set_xlim([xmin, xmax])
ax.set_ylim([ymin, ymax])
cs = ax.contour(X, Y, Z, levels=[0.075]); #  seems to correspond to 50% of the␣
 ↪points inside
```

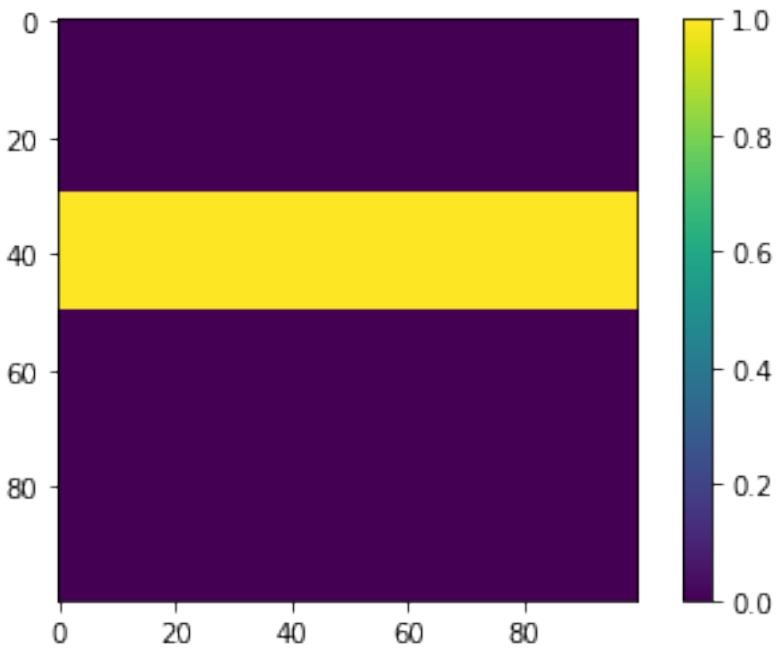


1.0.7 Convolution

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

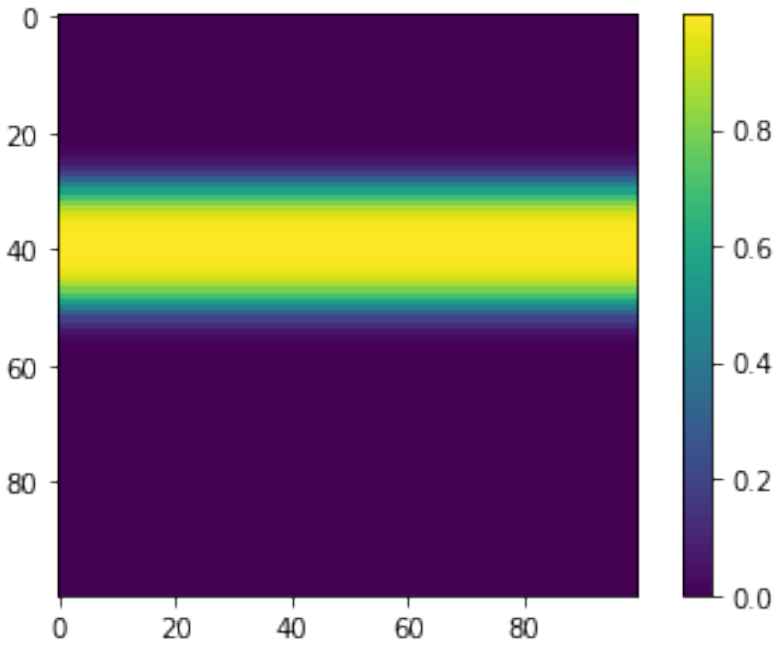
[105]:
```
# Let's define an image representing a long slit of width 10 pixels
slit = np.zeros((100, 100))
slit[30:50, :] = 1
```

[106]:
```
plt.imshow(slit)
plt.colorbar();
```

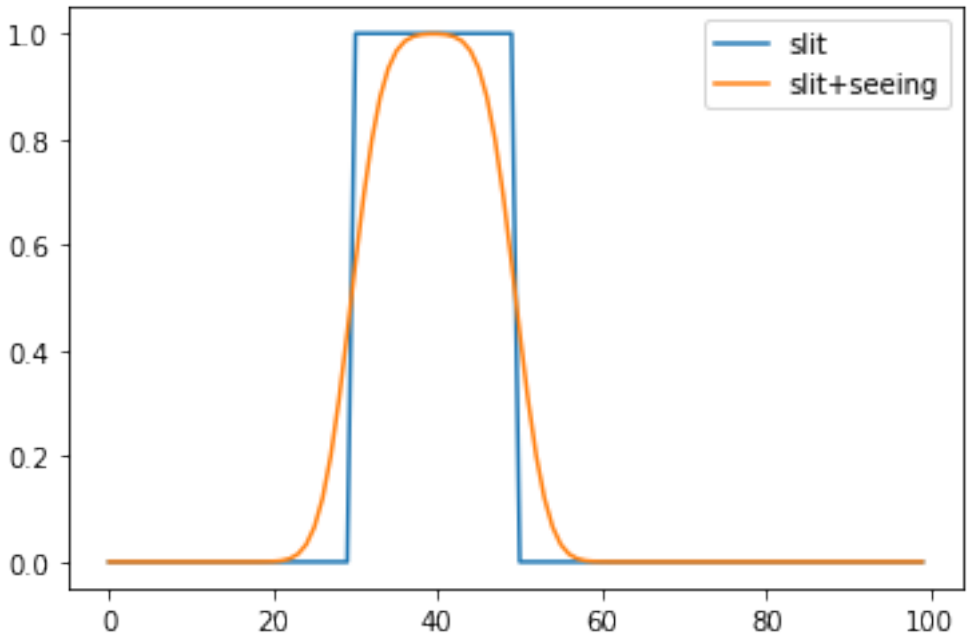


[107]:
```
# This is the routine to apply a gaussian convolution
from scipy.ndimage.filters import gaussian_filter
```

[116]:
```
slit_seeing = gaussian_filter(slit, 3) # Convolve with a gaussian, 3 is the␣
 ↪standard deviation in pixels
plt.imshow(slit_seeing)
plt.colorbar();
```



[117]:
```
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');
```



[118]:
```
# Check that the slit transmission is conserved:
print(simps(slit[:,50]), simps(slit_seeing[:,50]))
```

```
20.0 20.0
```

1.0.8 Quantiles

[120]:
```
from scipy.stats.mstats import mquantiles
```

[122]:
```
#help(mquantiles)
```

[119]:
```
data = np.random.randn(1000)
```

[124]:
```
mquantiles(data, [0.16, 0.5,0.84]) # should return something close to -1, 1␣
↪(the stv of the normal distribution)
```

[124]:
```
array([-0.95251222,  0.02438251,  1.0777994 ])
```

[125]:
```
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.]])
```

[131]:
```
mq = mquantiles(data, axis=0, limit=(0, 50))
print(mq)
print(type(mq))
mq?
print(mq.mask)
```

```
[[19.2  14.6   1.45]
 [40.   37.5   2.5 ]
 [42.8  40.05  3.55]]
<class 'numpy.ma.core.MaskedArray'>
False
```

```
Type:        MaskedArray
String form:
[[19.2  14.6   1.45]
 [40.   37.5   2.5 ]
 [42.8  40.05  3.55]]
Length:      3
```

```
File:        ~/anaconda3/lib/python3.7/site-packages/numpy/ma/core.py
Docstring:
An array class with possibly masked values.

Masked values of True exclude the corresponding element from any
computation.

Construction::

  x = MaskedArray(data, mask=nomask, dtype=None, copy=False, subok=True,
                  ndmin=0, fill_value=None, keep_mask=True, hard_mask=None,
                  shrink=True, order=None)

Parameters
----------
data : array_like
    Input data.
mask : sequence, optional
    Mask. Must be convertible to an array of booleans with the same
    shape as `data`. True indicates a masked (i.e. invalid) data.
dtype : dtype, optional
    Data type of the output.
    If `dtype` is None, the type of the data argument (``data.dtype``)
    is used. If `dtype` is not None and different from ``data.dtype``,
    a copy is performed.
copy : bool, optional
    Whether to copy the input data (True), or to use a reference instead.
    Default is False.
subok : bool, optional
    Whether to return a subclass of `MaskedArray` if possible (True) or a
    plain `MaskedArray`. Default is True.
ndmin : int, optional
    Minimum number of dimensions. Default is 0.
fill_value : scalar, optional
    Value used to fill in the masked values when necessary.
    If None, a default based on the data-type is used.
keep_mask : bool, optional
    Whether to combine `mask` with the mask of the input data, if any
    (True), or to use only `mask` for the output (False). Default is True.
hard_mask : bool, optional
    Whether to use a hard mask or not. With a hard mask, masked values
    cannot be unmasked. Default is False.
shrink : bool, optional
    Whether to force compression of an empty mask. Default is True.
order : {'C', 'F', 'A'}, optional
    Specify the order of the array.  If order is 'C', then the array
    will be in C-contiguous order (last-index varies the fastest).
    If order is 'F', then the returned array will be in
```

```
Fortran-contiguous order (first-index varies the fastest).
If order is 'A' (default), then the returned array may be
in any order (either C-, Fortran-contiguous, or even discontiguous),
unless a copy is required, in which case it will be C-contiguous.
```

1.0.9 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