intro_Scipy

June 30, 2017

```
In [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])
```

```
Fri Jun 30 13:25:39 2017
3.6.1
```

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/

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

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

1.0.1 Some usefull methods

```
In [4]: from scipy.special import gamma
        print(gamma(10.3))
        print(10*9*8*7*6*5*4*3*2)
```

```
716430.689062
3628800
```

```
In [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')))
```

```
149597870691.0
1.672621898e-27 kg
```

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

1.0.2 Integrations

```
In [6]: 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.lib.function_base:

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
    --------
    sum, 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: http://en.wikipedia.org/wiki/Trapezoidal_rule

    .. [2] Illustration image:
```

```
        http://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, uses this value as the first value in 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.
```

In [16]:
```
dir(scipy.integrate)
```

Out[16]:
```
['IntegrationWarning',
 'Tester',
 '__all__',
 '__builtins__',
 '__doc__',
 '__file__',
 '__name__',
 '__package__',
 '__path__',
 '_bvp',
 '_dop',
 '_ode',
 '_odepack',
 '_quadpack',
 'absolute_import',
 'complex_ode',
 'cumtrapz',
 'dblquad',
 'division',
 'fixed_quad',
 'lsoda',
 'newton_cotes',
 'nquad',
 'ode',
 'odeint',
 'odepack',
 'print_function',
 'quad',
 'quad_explain',
 'quadpack',
 'quadrature',
 'romb',
 'romberg',
 's',
```

```
  'simps',
  'solve_bvp',
  'test',
  'tplquad',
  'trapz',
  'vode']
```

```
In [7]: # 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))
```

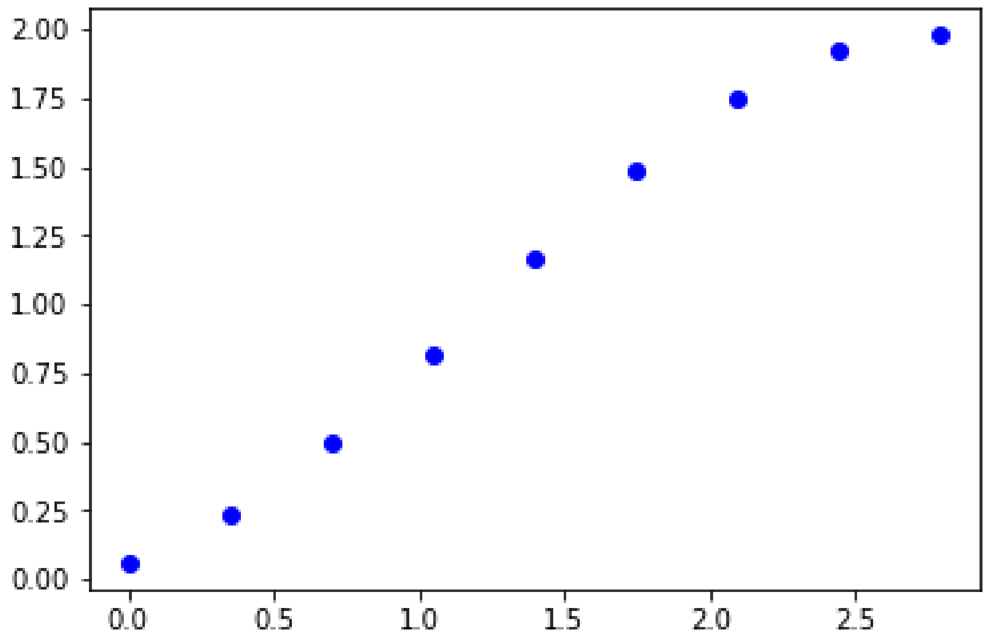
```
27.6 µs ± 133 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
150 µs ± 2.02 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
1.99983216389
1.99999996902
26.5 µs ± 109 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
146 µs ± 140 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
1.97965081122
1.99954873658
```

```
In [8]: # Cumulative integrale
        print(cumtrapz(np.abs(y), x))
```

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

```
In [9]: # 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
```




```
In [10]: from scipy.integrate import quad # To compute a definite integral
         from scipy.special import jv # Bessel function
         #help(quad)
         print(quad(lambda x: jv(2.5, x), 0, 10)) # Integrate the Bessel function of order 2.5 between
```

```
(0.8209075326034347, 1.1793289815399173e-08)
```

We now want to evaluate:

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

```
In [11]: # 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
         %timeit I = quad(f, 0, 1, args=(1,2,3)) # args will send 1, 2, 3 to f
         I = quad(f, 0, 1, args=(1,2,3)) # args will send 1, 2, 3 to f
         print(I)
         Integ = I[0]
         print(Integ)
```

```
16.6 µs ± 464 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
(3.0, 3.3306690738754696e-14)
3.0
```

1.0.3 Interpolations

```
In [12]: from scipy.interpolate import interp1d, interp2d, splrep, splev, griddata
```

```
In [13]: #help(scipy.interpolate) # a huge one...
         help(interp1d)
```

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

class interp1d(scipy.interpolate.polyint._Interpolator1D)
 |  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'
 |      where 'zero', 'slinear', 'quadratic' and 'cubic' refer to a spline
 |      interpolation of zeroth, first, second or third order) 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.
|
|  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_sor
|      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
|
|  ----------------------------------------------------------------------
|  Methods inherited from scipy.interpolate.polyint._Interpolator1D:
|
|  __call__(self, x)
|      Evaluate the interpolant
```

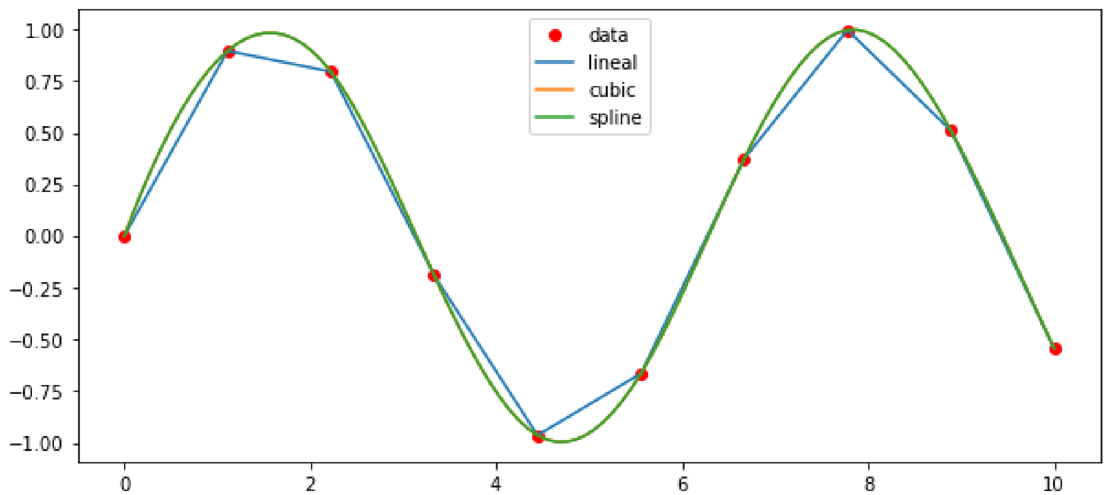
```
 |
 |      Parameters
 |      ----------
 |      x : array_like
 |          Points to evaluate the interpolant at.
 |
 |      Returns
 |      -------
 |      y : array_like
 |          Interpolated values. Shape is determined by replacing
 |          the interpolation axis in the original array with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

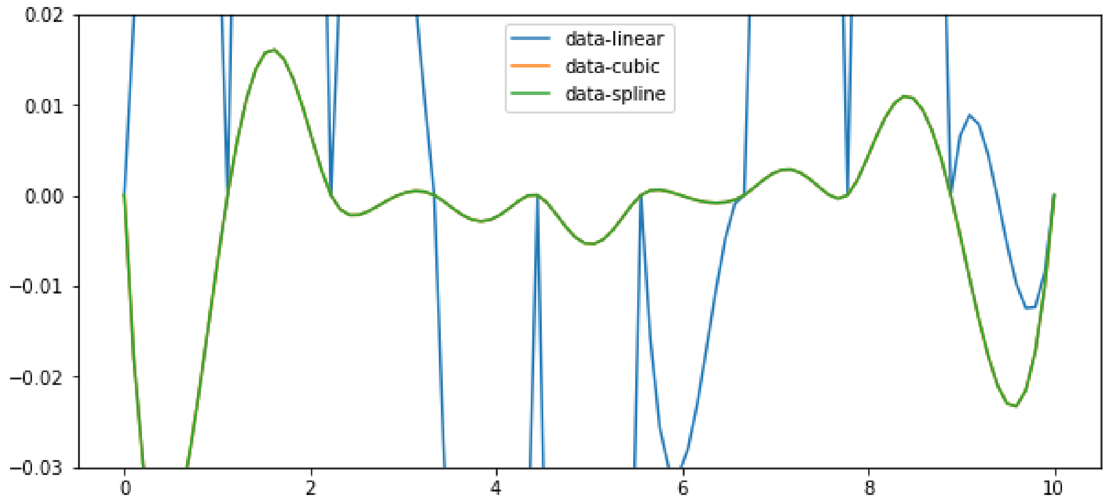
In [14]:
```python
x = np.linspace(0, 10, 10)
y = np.sin(x)
f = interp1d(x, y) # this creates a function that can be call at any interpolate point
f2 = interp1d(x, y, kind='cubic') # The same but using cubic interpolation
tck = splrep(x, y, s=0) # This initiate the spline interpolating function, finding the B-splin
# tck is a sequence of length 3 returned by `splrep` or `splprep` containing the knots, coeffi
f3 = lambda x: splev(x, tck) # Evaluate the B-spline or its derivatives.
```

In [15]:
```python
# 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')
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')
ax2.legend(loc='best')
ax2.set_ylim((-0.03, 0.02));
```





In [16]:
```
x0 = 3.5
print('{} {} {} {}'.format(np.sin(x0), f(x0), f2(x0), f3(x0)))
```

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

2D interpolation

In [17]:
```
# 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
```

In [18]:
```
# Initializing a 2D coordinate grid. Note the use of j to specify that the end point is includ
grid_x, grid_y = np.mgrid[0:1:100j, 0:1:200j]
```

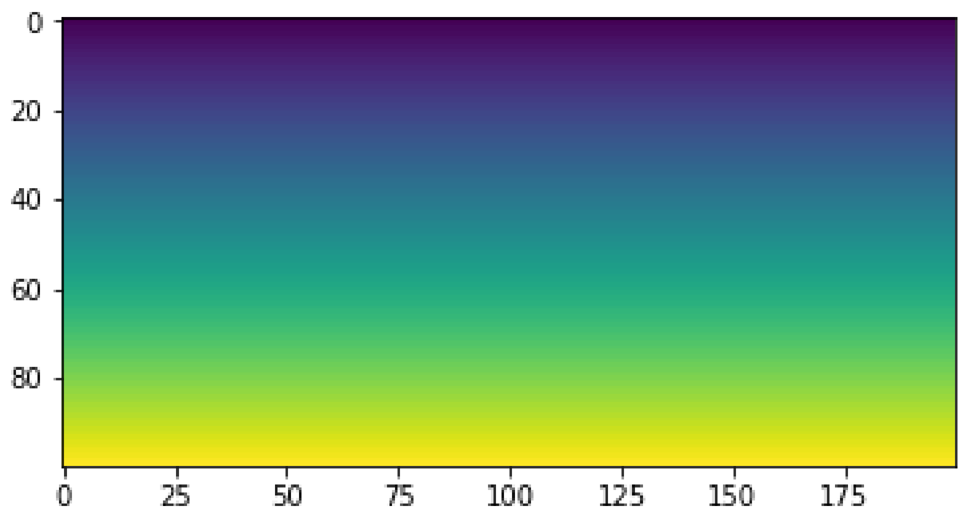
In [19]:
```
print(grid_x)
print(grid_y)
```

```
[[ 0.          0.          0.         ...,  0.          0.          0.        ]
 [ 0.01010101  0.01010101  0.01010101 ...,  0.01010101  0.01010101
```

```
  0.01010101]
 [ 0.02020202  0.02020202  0.02020202 ...,  0.02020202  0.02020202
  0.02020202]
 ...,
 [ 0.97979798  0.97979798  0.97979798 ...,  0.97979798  0.97979798
  0.97979798]
 [ 0.98989899  0.98989899  0.98989899 ...,  0.98989899  0.98989899
  0.98989899]
 [ 1.          1.          1.         ...,  1.          1.          1.        ]]
[[ 0.          0.00502513  0.01005025 ...,  0.98994975  0.99497487  1.        ]
 [ 0.          0.00502513  0.01005025 ...,  0.98994975  0.99497487  1.        ]
 [ 0.          0.00502513  0.01005025 ...,  0.98994975  0.99497487  1.        ]
 ...,
 [ 0.          0.00502513  0.01005025 ...,  0.98994975  0.99497487  1.        ]
 [ 0.          0.00502513  0.01005025 ...,  0.98994975  0.99497487  1.        ]
 [ 0.          0.00502513  0.01005025 ...,  0.98994975  0.99497487  1.        ]]
```

In [20]:
```
plt.imshow(grid_x)
```

Out[20]: <matplotlib.image.AxesImage at 0x7f01004a15f8>



In [21]:
```
# Generating 1000 x 2 points randomly
points = np.random.rand(1000, 2)
print(points)
values = func(points[:,0], points[:,1])
```

```
[[ 0.49868987  0.34219331]
 [ 0.49416282  0.02537293]
 [ 0.28584528  0.05005145]
 ...,
 [ 0.85218278  0.69951174]
 [ 0.49690052  0.91547562]
 [ 0.55070501  0.4539    ]]
```

In [22]:
```
# 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')
```

```
19.7 ms ± 419 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
11.7 ms ± 223 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
25.2 ms ± 339 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

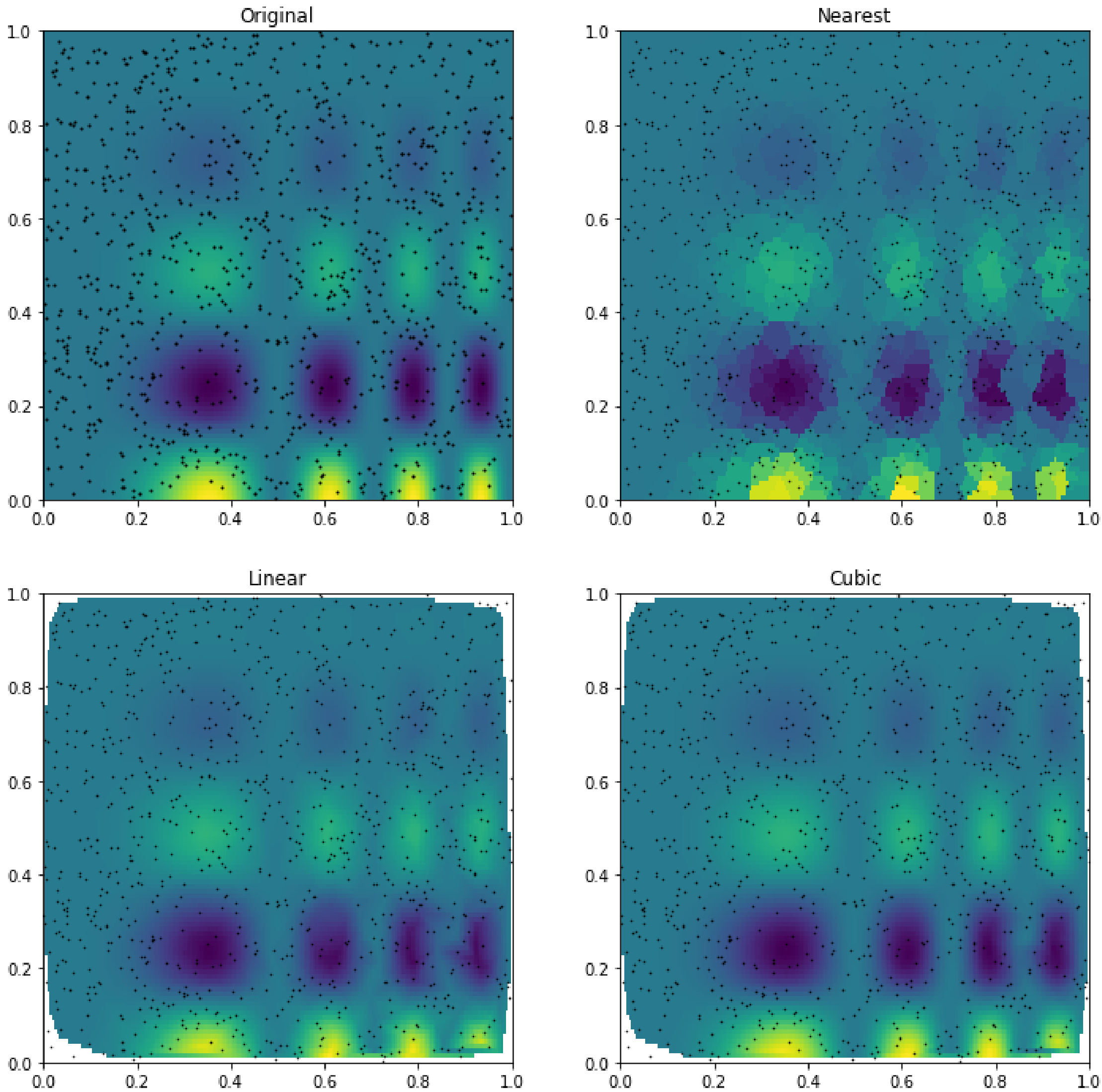
In [23]:
```
# 4 subplots
grid_z0 = griddata(points, values, (grid_x, grid_y), method='nearest')
grid_z1 = griddata(points, values, (grid_x, grid_y), method='linear')
grid_z2 = griddata(points, values, (grid_x, grid_y), method='cubic')
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 12))

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');
```



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

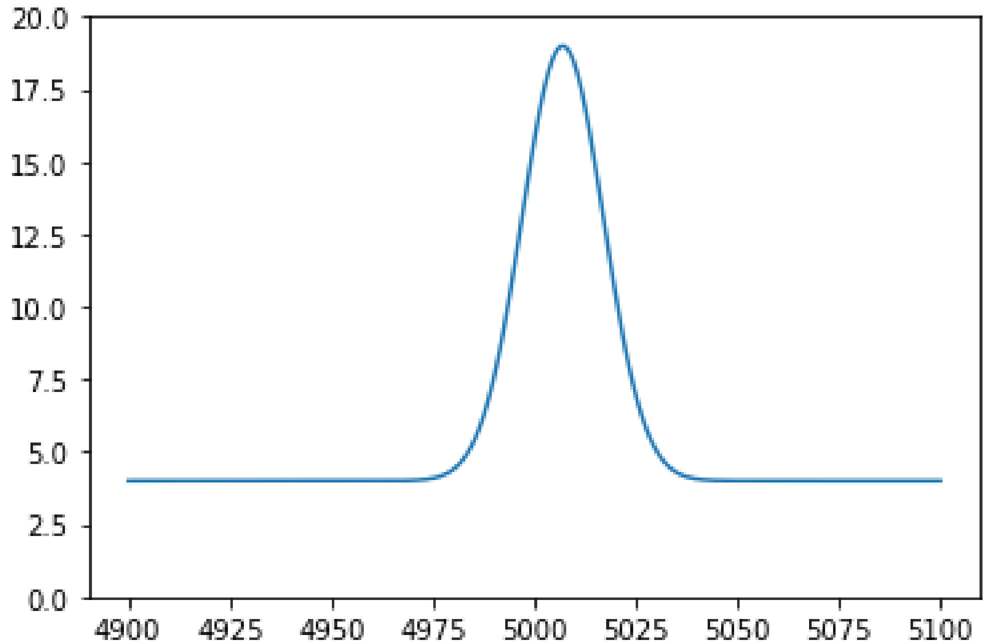
1.0.5 Data fit

```
In [24]: from scipy.optimize import curve_fit # this is used to adjust a set of data
```

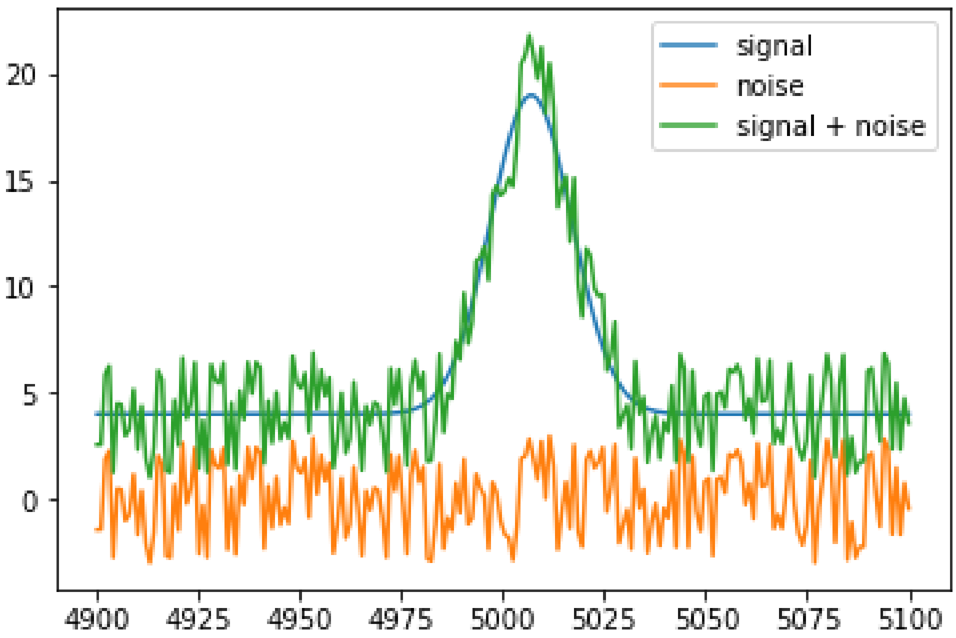
```
In [25]: #help(curve_fit)
```

```
In [26]: def gauss(x, A, B, C, S):
             # This is a gaussian function.
             return A + B*np.exp(-1 * (x - C)**2 / (2 * S**2))
```

In [27]:
```
# 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);
```



In [28]:
```
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');
```




```
In [29]: # 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 pix
```

```
In [30]: # 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('   A      B    Lam0     S')
         print('{0:.2f} {1:5.2f} {2:.2f} {3:5.2f}'.format(A, B, Lam0, Sigma))
         print('{0:.2f} {1:5.2f} {2:.2f} {3:5.2f}'.format(A_i, B_i, Lam0_i, Sigma_i))
         print('{0[0]:.2f} {0[1]:5.2f} {0[2]:5.2f} {0[3]:.2f}'.format(fit))
```

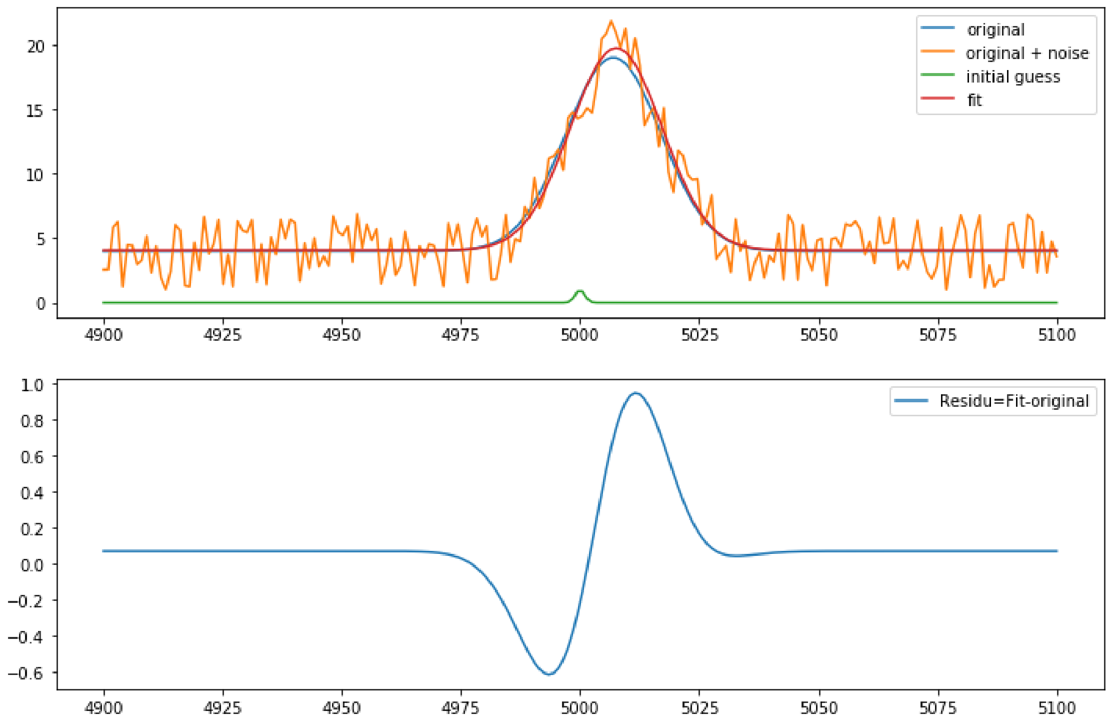
```
A     B    Lam0     S
4.00 15.00 5007.00 10.00
0.00  1.00 5000.00  1.00
4.07 15.66 5007.61 9.63
```

```
In [31]: # Computing the fit on the lambdas
         fl_fit = gauss(lam, fit[0], fit[1], fit[2], fit[3])
```

```
In [32]: 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 - fl, label='Residu=Fit-original')
ax2.legend();
```




```
In [33]: # 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))
```

```
375.994241195
378.978700075
377.87040867
```

```
In [34]: 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=251.9006554158191, khi^2_reduced=1.2917982329016364
```

1.0.6 Multivariate estimation

```
In [35]: from scipy import stats
```

```
In [36]: 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
```

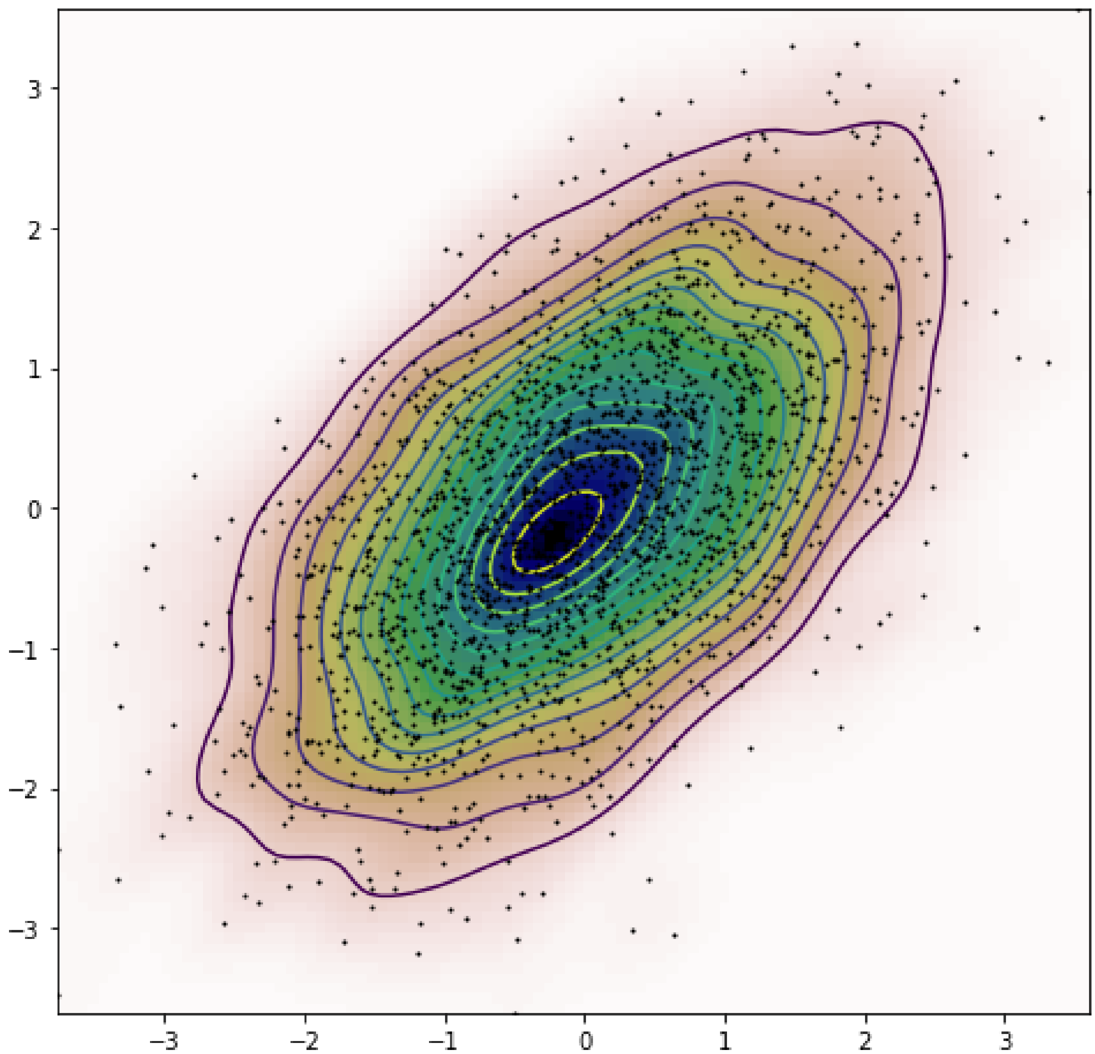
```
In [38]: m1, m2 = measure(2000)
         xmin = m1.min()
         xmax = m1.max()
         ymin = m2.min()
         ymax = m2.max()
         print(xmin, xmax, ymin, ymax)
```

```
-3.76250156395 3.6123769827 -3.60100846935 3.5660860828
```

```
In [39]: X, Y = np.mgrid[xmin:xmax:100j, 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)
```

```
(100, 100)
```

```
In [40]: fig, ax = plt.subplots(figsize=(12, 8))
         ax.imshow(np.rot90(Z), cmap=plt.cm.gist_earth_r, extent=[xmin, xmax, ymin, ymax], origin='uppe
         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, 
         cs = ax.contour(X, Y, Z, levels=levels); # I dont't know what those levels mean... but it work
```



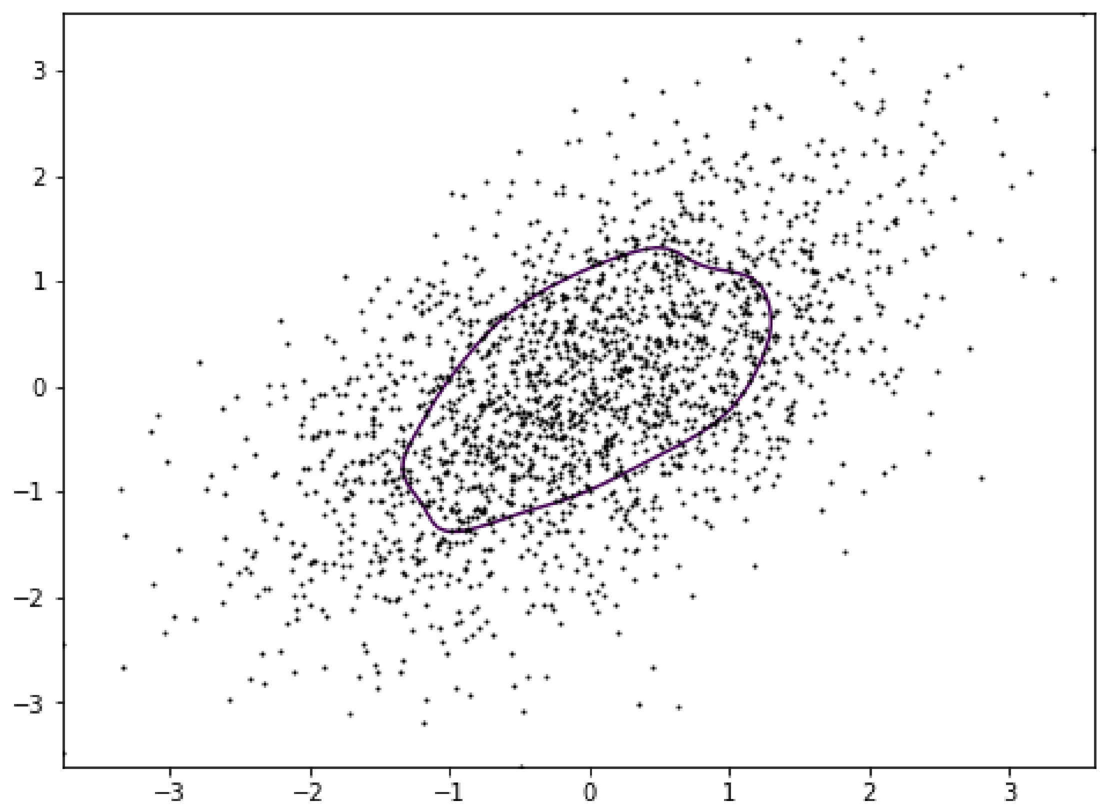

```
In [41]: # We save the contour paths in a list
         paths = []
         for collec in cs.collections:
             try:
                 paths.append(collec.get_paths()[0])
             except:
                 pass

In [59]: # 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
```

```
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 75% of the data
level 0.05 contains 69% of the data
level 0.06 contains 62% of the data
level 0.07 contains 56% of the data
level 0.08 contains 48% of the data
level 0.09 contains 39% of the data
level 0.10 contains 32% of the data
level 0.11 contains 25% of the data
level 0.12 contains 17% of the data
level 0.13 contains 10% of the data
level 0.14 contains  4% of the data
```

In [45]:
```python
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=[0.078]); #  seems to correspond to 50% of the points inside
```

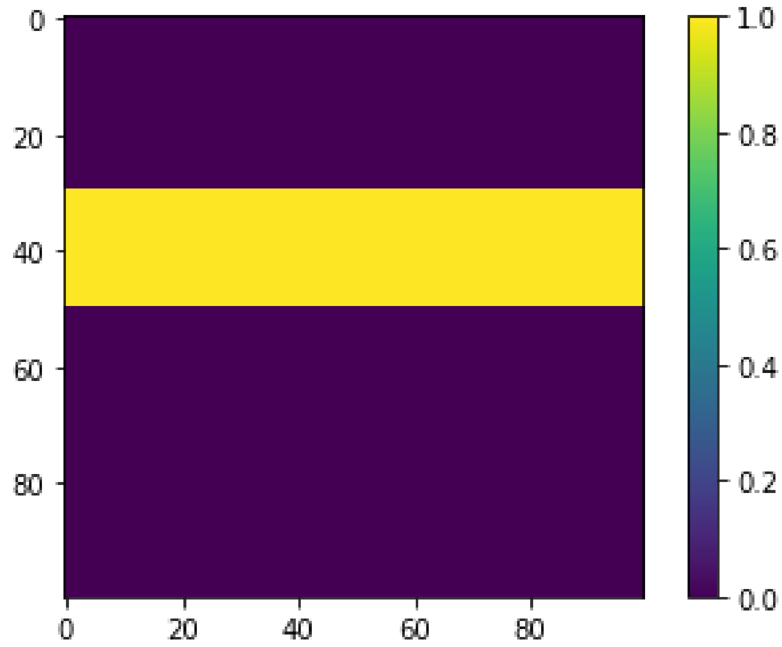


1.0.7 Convolution

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

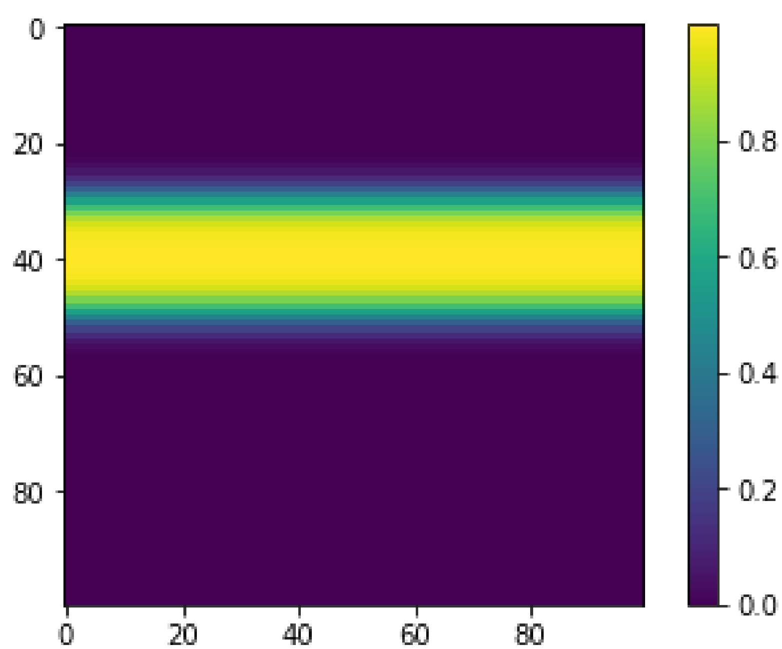
In [46]:
```python
# Let's define an image representing a long slit of width 10 pixels
slit = np.zeros((100, 100))
slit[30:50, :] = 1
```

```
In [47]: plt.imshow(slit)
         plt.colorbar();
```

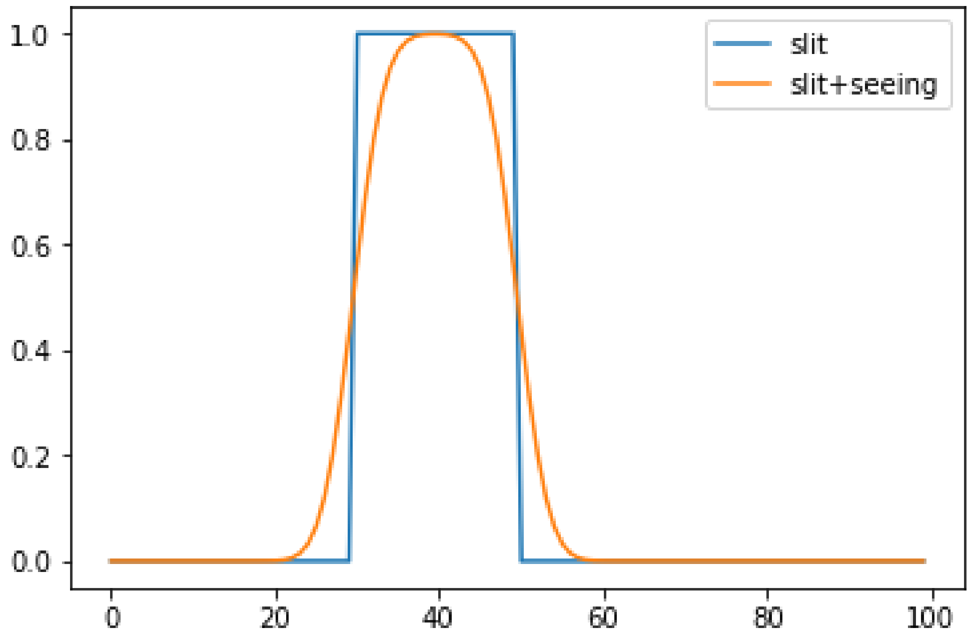



```
In [48]: # This is the routine to apply a gaussian convolution
         from scipy.ndimage.filters import gaussian_filter

In [49]: slit_seeing = gaussian_filter(slit, 3) # Convolve with a gaussian, 3 is the standard deviation
         plt.imshow(slit_seeing)
         plt.colorbar();
```



In [50]:
```
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');
```



In [51]:
```
# Check that the slit transmission is conserved:
print(simps(slit[:,50]), simps(slit_seeing[:,50]))
```

```
20.0 20.0
```

1.0.8 Quantiles

In [52]:
```
from scipy.stats.mstats import mquantiles
```

In [53]:
```
#help(mquantiles)
```

In [54]:
```
data = np.random.randn(1000)
```

In [55]:
```
mquantiles(data, [0.16, 0.84]) # should return something close to -1, 1 (the stv of the normal
```

Out[55]:
```
array([-0.97066278,  1.04647155])
```

In [56]:
```
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.]])
```

```
In [57]: 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
```

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