SciPy Review

October 19, 2016

```
In [1]: from IPython.core.display import HTML
        def css_styling():
            styles = open("styles/custom.css", "r").read()
            return HTML(styles)
        css_styling()
```

```
Out[1]: <IPython.core.display.HTML object>
```

1 SciPy

The SciPy framework builds on top of the low-level NumPy framwork for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are:

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

In this lecture we will look at how to use some of these subpackages.

To access the SciPy package in a Python program, we start by importing the `scipy` module. As a shortcut, we will abbreviate `scipy` to `sp`, analogous to what we have done with NumPy.

```
In [2]: import scipy as sp
```

1.1 Integration

1.1.1 Numerical integration: quadrature

Numerical evaluation of a function of the type

$$\int_a^b f(x)dx$$

is called *numerical quadrature*, or simply *quadature*. SciPy provides a series of functions for different kind of quadrature, for example the `quad`, `dblquad` and `tplquad` for single, double and triple integrals, respectively.

```
In [3]: from scipy.integrate import quad, dblquad, tplquad
```

The `quad` functions takes a large number of optional arguments, which can be used to fine-tune the behaviour of the function.

```
In [4]: 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
    Compute a definite integral.

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

    Parameters
    ----------
    func : function
        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
        instead be a ``ctypes`` function of the form:

            f(int n, double args[n]),

        where ``args`` is an array of function arguments and ``n`` is the
        length of ``args``. ``f.argtypes`` should be set to
        ``(c_int, c_double)``, and ``f.restype`` should be ``(c_double,)``.
    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.
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. 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
```

Here is a simple usage example:

```
In [5]: # define a simple function for the integrand
        def f(x):
            return x

In [6]: x_lower = 0 # the lower limit of x
        x_upper = 1 # the upper limit of x
```

```
        val, abserr = quad(f, x_lower, x_upper)

        print("integral value = {0}, absolute error = {1}".format(val, abserr))
```

```
integral value = 0.5, absolute error = 5.551115123125783e-15
```

If we need to pass extra arguments to integrand function we can use the `args` keyword argument:

```
In [7]: from scipy import stats
        import numpy as np

In [8]: # Integral of N(2, 3) over [inf, 0)
        val, abserr = quad(stats.distributions.norm.pdf, a=-np.inf, b=0,
                           args=(2, 3))

        print(val, abserr)
```

```
0.2524925375469229 8.193093413455012e-10
```

```
In [9]: stats.distributions.norm.cdf(0, 2, 3)

Out[9]: 0.25249253754692291
```

As show in the example above, we can also use 'Inf' or '-Inf' as integral limits.
Higher-dimensional integration works in the same way:

```
In [12]: dblquad?

In [11]: def integrand(x, y):
             return np.exp(-x**2-y**2)

         x_lower = 0
         x_upper = 10
         y_lower = 0
         y_upper = 10

         val, abserr = dblquad(integrand, x_lower, x_upper,
                               lambda x : y_lower, lambda x: y_upper)

         print(val, abserr)
```

```
0.7853981633974476 1.638229942140971e-13
```

Note how we had to pass `lambda` functions for the limits for the y integration, since these in general can be functions of x.

1.1.2 Example: Trapezoid rule

A simple illustration of the trapezoid rule for definite integration:

$$\int_a^b f(x)\,dx \approx \frac{1}{2}\sum_{k=1}^{N}(x_k - x_{k-1})\,(f(x_k) + f(x_{k-1}))\,.$$

First, we define a simple function and sample it between 0 and 10 at 200 points

```
In [13]: def f(x):
             return (x-3)*(x-5)*(x-7)+85

         x = np.linspace(0, 10, 200)
         y = f(x)
```

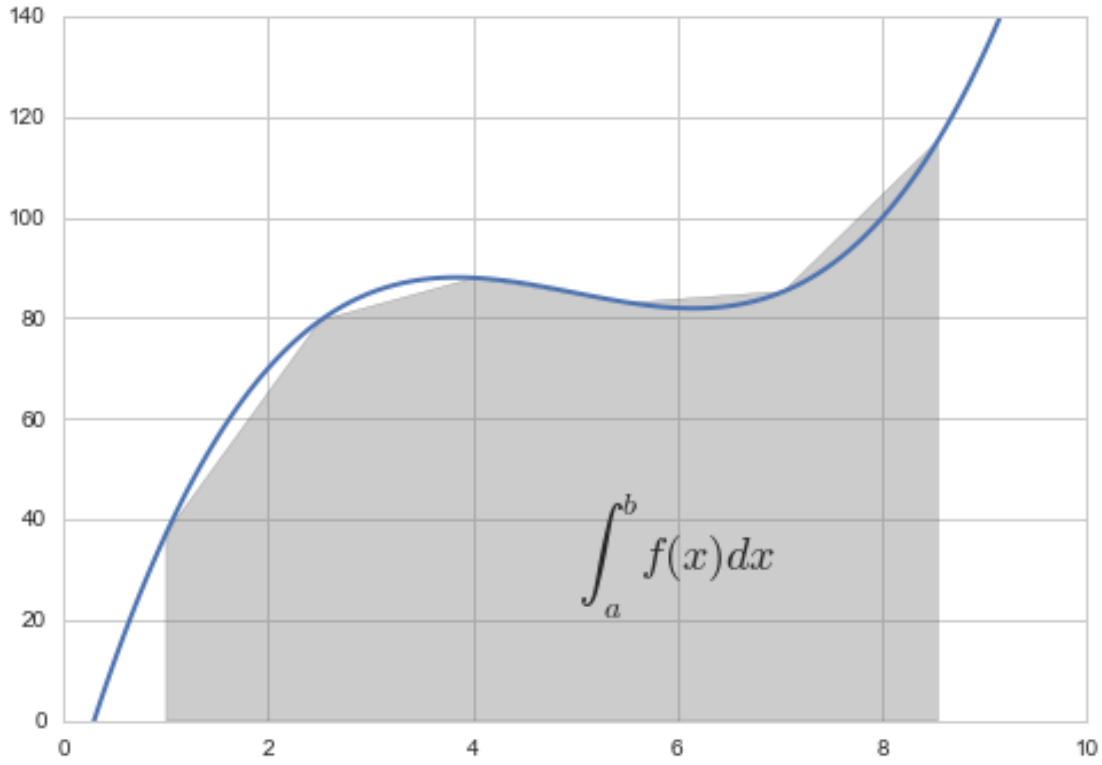
Choose a region to integrate over and take only a few points in that region

```
In [14]: a, b = 1, 11
         xint = x[np.logical_and(x>=a, x<=b)][::30]
         yint = y[np.logical_and(x>=a, x<=b)][::30]
```

Plot both the function and the area below it in the trapezoid approximation

```
In [15]: %matplotlib inline
         import pylab as plt
         import seaborn as sns
         sns.set_style("whitegrid")

         plt.plot(x, y, lw=2)
         plt.axis([0, 10, 0, 140])
         plt.fill_between(xint, 0, yint, facecolor='gray', alpha=0.4)
         plt.text(0.5 * (a + b), 30,r"$\int_a^b f(x)dx$",
                  horizontalalignment='center', fontsize=20)
```

```
Out[15]: <matplotlib.text.Text at 0x111d724a8>
```




```
In [16]: from scipy.integrate import quad, trapz
         integral, error = quad(f, 1, 9)
         print("The integral is:", integral, "+/-", error)
         print("The trapezoid approximation with", len(xint), "points is:", trapz(y
```

```
The integral is: 680.0 +/- 7.549516567451064e-12
The trapezoid approximation with 6 points is: 621.286411141
```

1.2 Linear algebra

The linear algebra module contains a lot of matrix related functions, including linear equation solving, eigenvalue solvers, matrix functions (for example matrix-exponentiation), and several decompositions.

Linear equation systems Linear equation systems on the matrix form

$Ax = b$

where A is a matrix and x, b are vectors can be solved like:

```
In [17]: from numpy.linalg import solve, eigvals, eig, norm, inv, det
```

```
In [18]: A = np.array([[1,2,3], [4,5,6], [7,8,9]])
         b = np.array([1,2,3])
```

```
In [19]: x = solve(A, b)
         x

Out[19]: array([-0.33333333,  0.66666667, -0.        ])
```

We can check this:

```
In [20]: (np.dot(A, x) - b).round(10)

Out[20]: array([-0.,  0.,  0.])
```

We can also do the same with matrices:

```
In [21]: A = np.random.rand(3,3)
         B = np.random.rand(3,3)

In [22]: X = solve(A, B)

In [23]: X

Out[23]: array([[ -6.72387775,  -1.94077468,   0.70103013],
                [-27.48646773, -10.95577344,   6.23876216],
                [ 19.73573169,   7.68418026,  -3.42695454]])

In [24]: # check
         norm(np.dot(A, X) - B).round(10)

Out[24]: 0.0

In [25]: solve?
```

Eigenvalues and eigenvectors The eigenvalue problem for a matrix A:

$Av_n = \lambda_n v_n$

where v_n are is the nth eigenvector and λ_n is the nth eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals` and for calculating both eigenvalues and eigenvectors, use the function `eig`:

```
In [26]: evals = eigvals(A)

In [27]: evals

Out[27]: array([ 1.54411484, -0.20596315, -0.02340911])

In [28]: evals, evecs = eig(A)

In [29]: evals

Out[29]: array([ 1.54411484, -0.20596315, -0.02340911])

In [30]: evecs
```

```
Out[30]: array([[-0.36116382, -0.34749915, -0.14731188],
                [-0.60235708,  0.88438462, -0.82759706],
                [-0.71184734, -0.31162186,  0.54164778]])
```

The eigenvectors corresponding to the nth eigenvalue (stored in `evals[n]`) is the nth *column* in `evecs`, i.e., `evecs[:,n]`. To verify this, let's try mutiplying eigenvectors with the matrix and compare to the product of the eigenvector and the eigenvalue:

```
In [31]: n = 1

         norm(np.dot(A, evecs[:,n]) - evals[n] * evecs[:,n]).round(10)

Out[31]: 0.0
```

There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

Matrix operations

```
In [33]: # the matrix inverse
         np.linalg.inv(A)

Out[33]: array([[-12.80143241,  -0.55304295,   7.29150801],
                [-47.990759  , -11.50821586,  34.63478542],
                [ 34.0056426 ,   7.10343934, -22.61637739]])

In [34]: # determinant
         np.linalg.det(A)

Out[34]: 0.0074448174500144097

In [35]: # norms of various orders
         np.linalg.norm(A, ord=2), np.linalg.norm(A, ord=np.inf)

Out[35]: (1.5800326231677224, 1.8117197336632511)
```

1.3 Sparse matrices

Sparse matrices are often useful in numerical simulations dealing with large systems, if the problem can be described in matrix form where the matrices or vectors mostly contains zeros. Scipy has a good support for sparse matrices, with basic linear algebra operations (such as equation solving, eigenvalue calculations, etc).

There are many possible strategies for storing sparse matrices in an efficient way. Some of the most common are the so-called coordinate form (COO), list of list (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has some advantanges and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. So often a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before used in real calcalations.

When we create a sparse matrix we have to choose which format it should be stored in. For example,

```
In [36]: from scipy.sparse import csr_matrix, lil_matrix, csc_matrix

In [37]: # dense matrix
         M = np.array([[1,0,0,0], [0,3,0,0], [0,1,1,0], [1,0,0,1]]); M

Out[37]: array([[1, 0, 0, 0],
                [0, 3, 0, 0],
                [0, 1, 1, 0],
                [1, 0, 0, 1]])

In [38]: # convert from dense to sparse
         A = csr_matrix(M); A

Out[38]: <4x4 sparse matrix of type '<class 'numpy.int64'>'
                 with 6 stored elements in Compressed Sparse Row format>

In [39]: # convert from sparse to dense
         A.todense()

Out[39]: matrix([[1, 0, 0, 0],
                 [0, 3, 0, 0],
                 [0, 1, 1, 0],
                 [1, 0, 0, 1]], dtype=int64)
```

More efficient way to create sparse matrices: create an empty matrix and populate with using matrix indexing (avoids creating a potentially large dense matrix)

```
In [40]: A = lil_matrix((4,4)) # empty 4x4 sparse matrix
         A[0,0] = 1
         A[1,1] = 3
         A[2,2] = A[2,1] = 1
         A[3,3] = A[3,0] = 1
         A

Out[40]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in LInked List format>

In [41]: A.todense()

Out[41]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.],
                 [ 0.,  1.,  1.,  0.],
                 [ 1.,  0.,  0.,  1.]])
```

Converting between different sparse matrix formats:

```
In [42]: A

Out[42]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in LInked List format>
```

```
In [43]: A = csr_matrix(A); A

Out[43]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in Compressed Sparse Row format>

In [44]: A = csc_matrix(A); A

Out[44]: <4x4 sparse matrix of type '<class 'numpy.float64'>'
                 with 6 stored elements in Compressed Sparse Column format>
```

We can compute with sparse matrices like with dense matrices:

```
In [45]: A.todense()

Out[45]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  3.,  0.,  0.],
                 [ 0.,  1.,  1.,  0.],
                 [ 1.,  0.,  0.,  1.]])

In [46]: (A * A).todense()

Out[46]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  9.,  0.,  0.],
                 [ 0.,  4.,  1.,  0.],
                 [ 2.,  0.,  0.,  1.]])

In [47]: A.dot(A).todense()

Out[47]: matrix([[ 1.,  0.,  0.,  0.],
                 [ 0.,  9.,  0.,  0.],
                 [ 0.,  4.,  1.,  0.],
                 [ 2.,  0.,  0.,  1.]])

In [50]: v = np.array([1,2,3,4])[:,np.newaxis]
         v

Out[50]: array([[1],
                [2],
                [3],
                [4]])

In [55]: # sparse matrix - dense vector multiplication
         A * v

Out[55]: array([[ 1.],
                [ 6.],
                [ 5.],
                [ 5.]])

In [56]: # same result with dense matrix - dense vector multiplcation
         A.todense() * v

Out[56]: matrix([[ 1.],
                 [ 6.],
                 [ 5.],
                 [ 5.]])
```

1.4 Optimization

Optimization (finding minima or maxima of a function) is a large field in mathematics, and optimization of complicated functions or in many variables can be rather involved. Here we will only look at a few very simple cases.

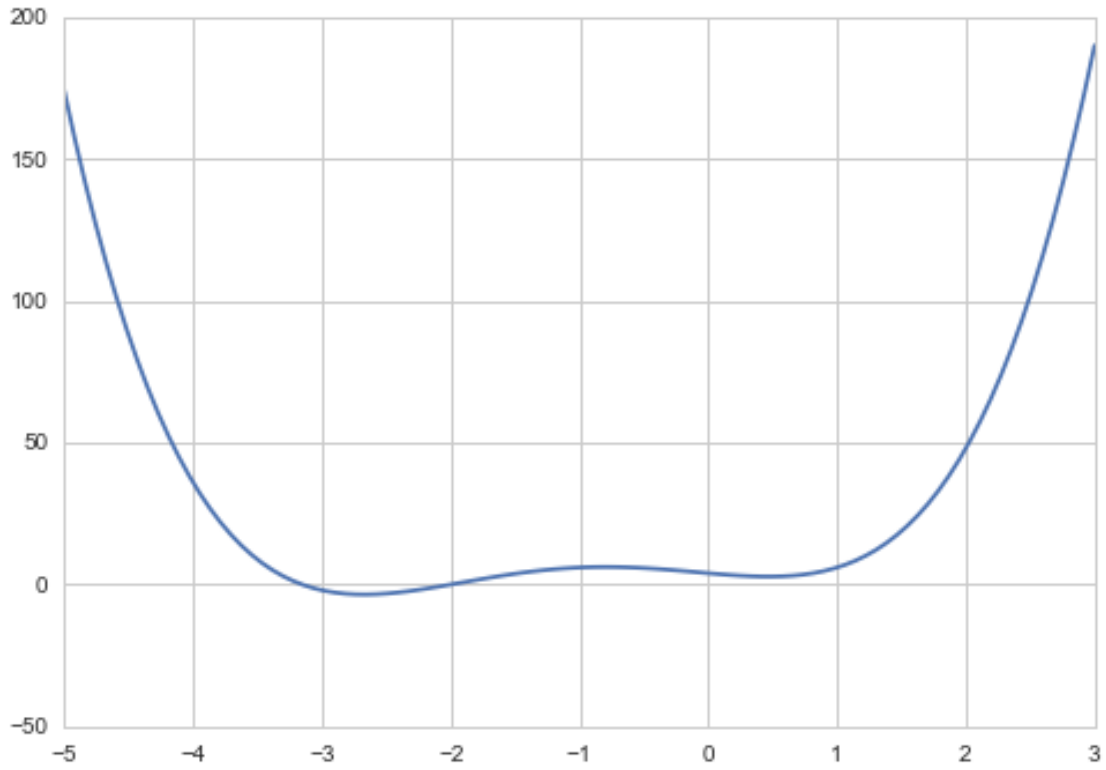
```
In [57]: from scipy import optimize
```

1.4.1 Finding a minima

Let's first look at how to find the minima of a simple function of a single variable:

```
In [58]: def f(x):
             return 4*x**3 + (x-2)**2 + x**4
```

```
In [59]: fig, ax = plt.subplots()
         x = np.linspace(-5, 3, 100)
         ax.plot(x, f(x));
```



We can use the `fmin_bfgs` function to find the minima of a function:

```
In [60]: x_min = optimize.fmin_bfgs(f, -2)
         x_min
```

```
Optimization terminated successfully.
         Current function value: -3.506641
         Iterations: 6
         Function evaluations: 30
         Gradient evaluations: 10


Out[60]: array([-2.67298167])

In [61]: optimize.fmin_bfgs(f, 0.5)

Optimization terminated successfully.
         Current function value: 2.804988
         Iterations: 3
         Function evaluations: 15
         Gradient evaluations: 5


Out[61]: array([ 0.46961745])
```

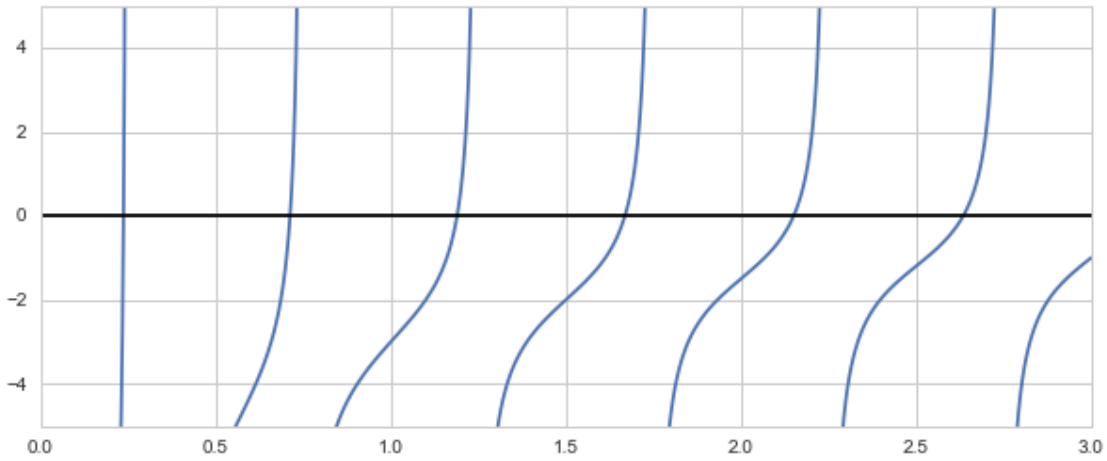
We can also use the `brent` or `fminbound` functions. They have a bit different syntax and use different algorithms.

```
In [62]: optimize.brent(f)

Out[62]: 0.46961743402759754

In [63]: optimize.fminbound(f, -4, 2)

Out[63]: -2.6729822917513886
```

1.4.2 Finding a solution to a function

To find the root for a function of the form $f(x) = 0$ we can use the `fsolve` function. It requires an initial guess:

```
In [66]: omega_c = 3.0
         def f(omega):
             # a transcendental equation: resonance frequencies of a low-Q SQUID te
             return np.tan(2*np.pi*omega) - omega_c/omega

In [67]: fig, ax = plt.subplots(figsize=(10,4))
         x = np.linspace(0, 3, 1000)
         y = f(x)
         mask = np.where(abs(y) > 50)
         x[mask] = y[mask] = None # get rid of vertical line when the function flip
         ax.plot(x, y)
         ax.plot([0, 3], [0, 0], 'k')
         ax.set_ylim(-5,5);
```




```
In [68]: optimize.fsolve(f, 0.1)

Out[68]: array([ 0.23743014])

In [69]: optimize.fsolve(f, 0.6)

Out[69]: array([ 0.71286972])

In [70]: optimize.fsolve(f, 1.1)

Out[70]: array([ 1.18990285])
```

1.4.3 Example: truncated distribution

Suppose that we observe Y truncated below at a (where a is known). If X is the distribution of our observation, then:

$$P(X \leq x) = P(Y \leq x | Y > a) = \frac{P(a < Y \leq x)}{P(Y > a)}$$

(so, Y is the original variable and X is the truncated variable)
Then X has the density:

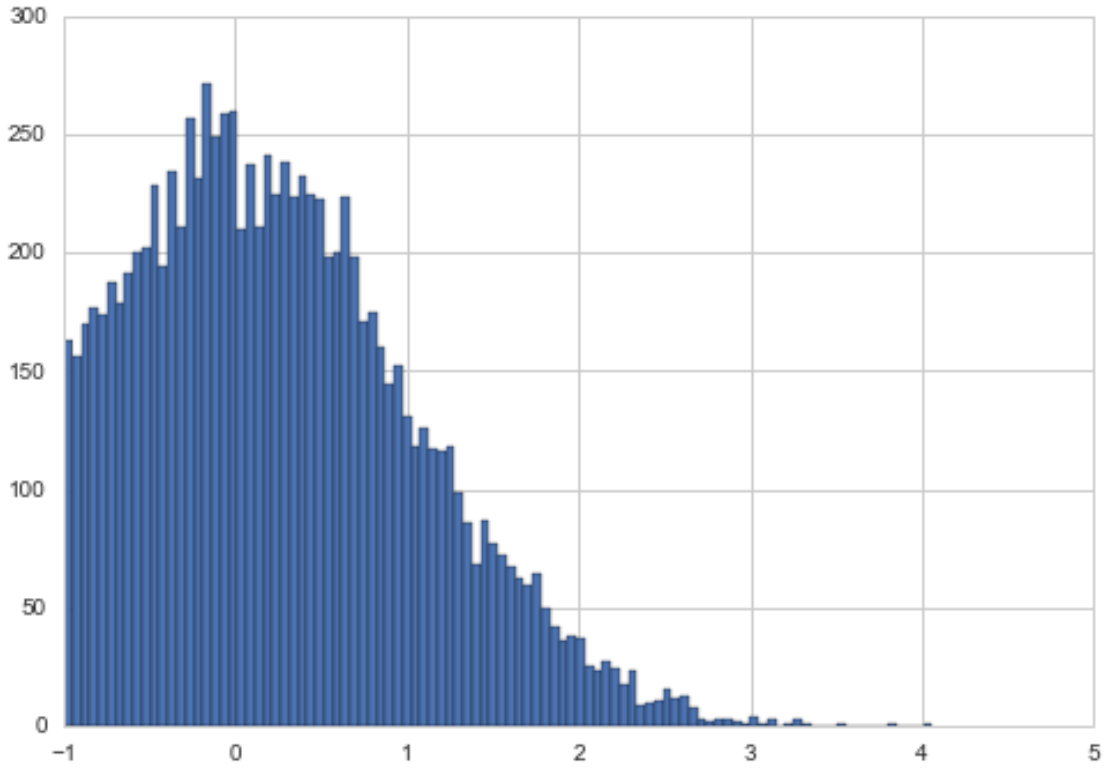
$$f_X(x) = \frac{f_Y(x)}{1 - F_Y(a)} \text{ for } x > a$$

Suppose $Y \sim N(\mu, \sigma^2)$ and $x_1, \ldots, x_n$ are independent observations of X. We can use maximum likelihood to find μ and σ.

First, we can simulate a truncated distribution using a `while` statement to eliminate samples that are outside the support of the truncated distribution.

```
In [71]: x = np.random.normal(size=10000)
         a = -1
         x_small = x < a
```

```
while x_small.sum():
    x[x_small] = np.random.normal(size=x_small.sum())
    x_small = x < a

_ = plt.hist(x, bins=100)
```



We can construct a log likelihood for this function using the conditional form:

$$f_X(x) = \frac{f_Y(x)}{1 - F_Y(a)} \text{ for } x > a$$

```
In [72]: from scipy.stats.distributions import norm

         trunc_norm = lambda theta, a, x: -(np.log(norm.pdf(x, theta[0], theta[1])) -
                                            np.log(1 - norm.cdf(a, theta[0], the
```

For this example, we will use another optimization algorithm, the **Nelder-Mead simplex algorithm**. It has a couple of advantages:

- it does not require derivatives
- it can optimize (minimize) a vector of parameters

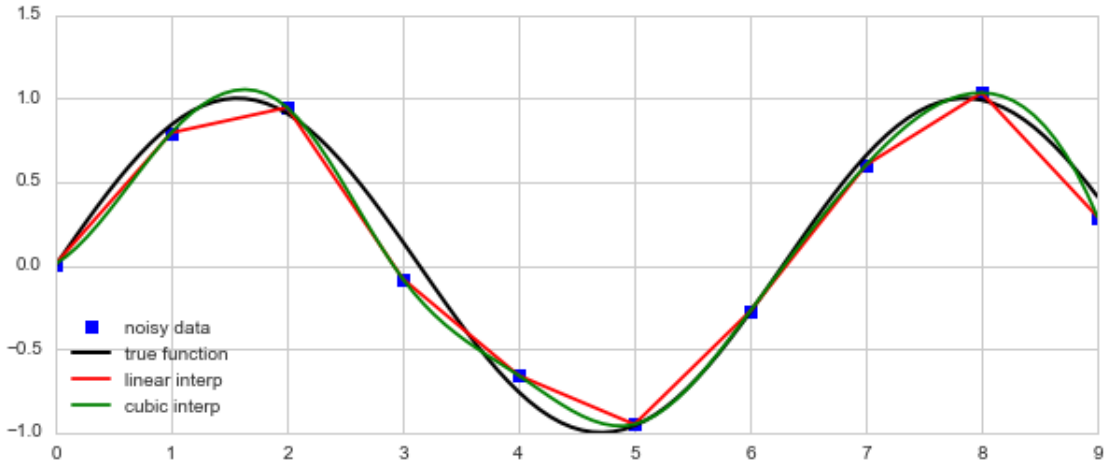
SciPy implements this algorithm in its `fmin` function:

```
In [73]: from scipy.optimize import fmin

         fmin(trunc_norm, np.array([1,2]), args=(-1, x))

Optimization terminated successfully.
         Current function value: 11001.113385
         Iterations: 44
         Function evaluations: 84


Out[73]: array([-0.01237282,  1.00270043])
```

1.5 Interpolation

Interpolation is simple and convenient in scipy: The `interp1d` function, when given arrays describing X and Y data, returns and object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X), and it returns the corresponding interpolated y value:

```
In [74]: from scipy.interpolate import interp1d

In [75]: def f(x):
             return np.sin(x)

In [76]: n = np.arange(0, 10)
         x = np.linspace(0, 9, 100)

         # simulate measurement with noise
         y_meas = f(n) + 0.1 * np.random.randn(len(n))
         # Actual function
         y_real = f(x)

         linear_interpolation = interp1d(n, y_meas)
         y_interp1 = linear_interpolation(x)

         cubic_interpolation = interp1d(n, y_meas, kind='cubic')
         y_interp2 = cubic_interpolation(x)

In [77]: fig, ax = plt.subplots(figsize=(10,4))
         ax.plot(n, y_meas, 'bs', label='noisy data')
         ax.plot(x, y_real, 'k', lw=2, label='true function')
         ax.plot(x, y_interp1, 'r', label='linear interp')
         ax.plot(x, y_interp2, 'g', label='cubic interp')
         ax.legend(loc=3);
```



1.6 Statistics

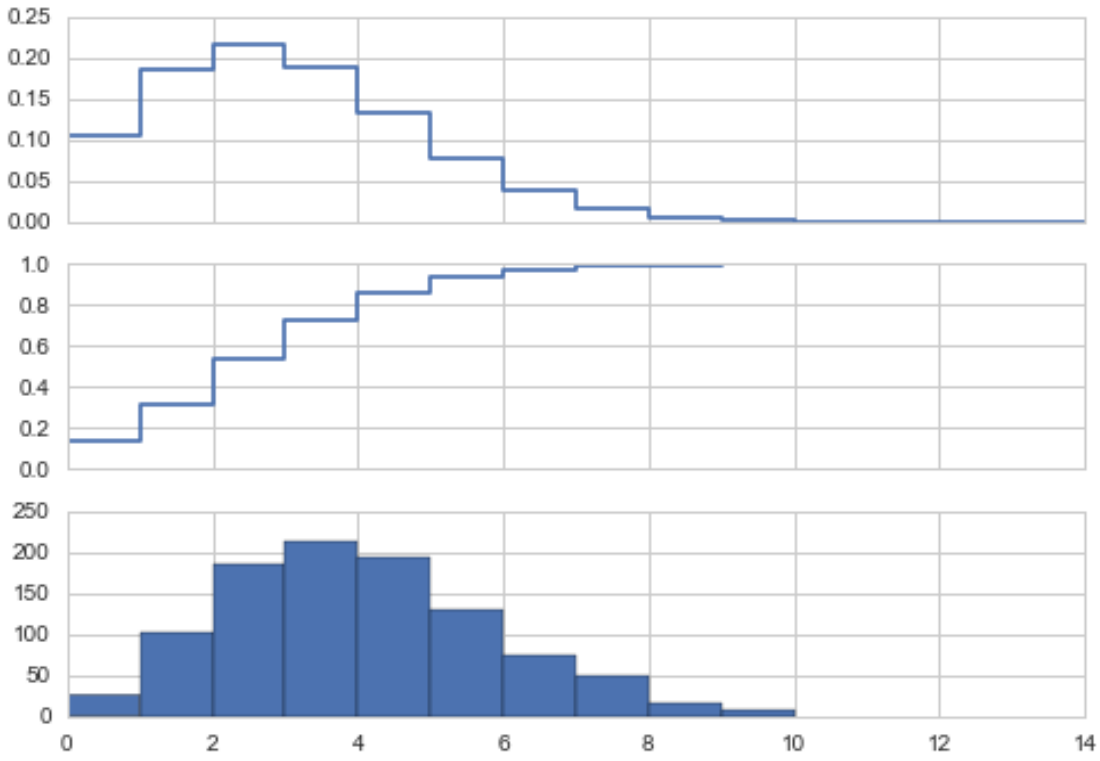
The `scipy.stats` module contains a large number of statistical distributions, statistical functions and tests.

```
In [78]: from scipy import stats
```

Discrete random variables:

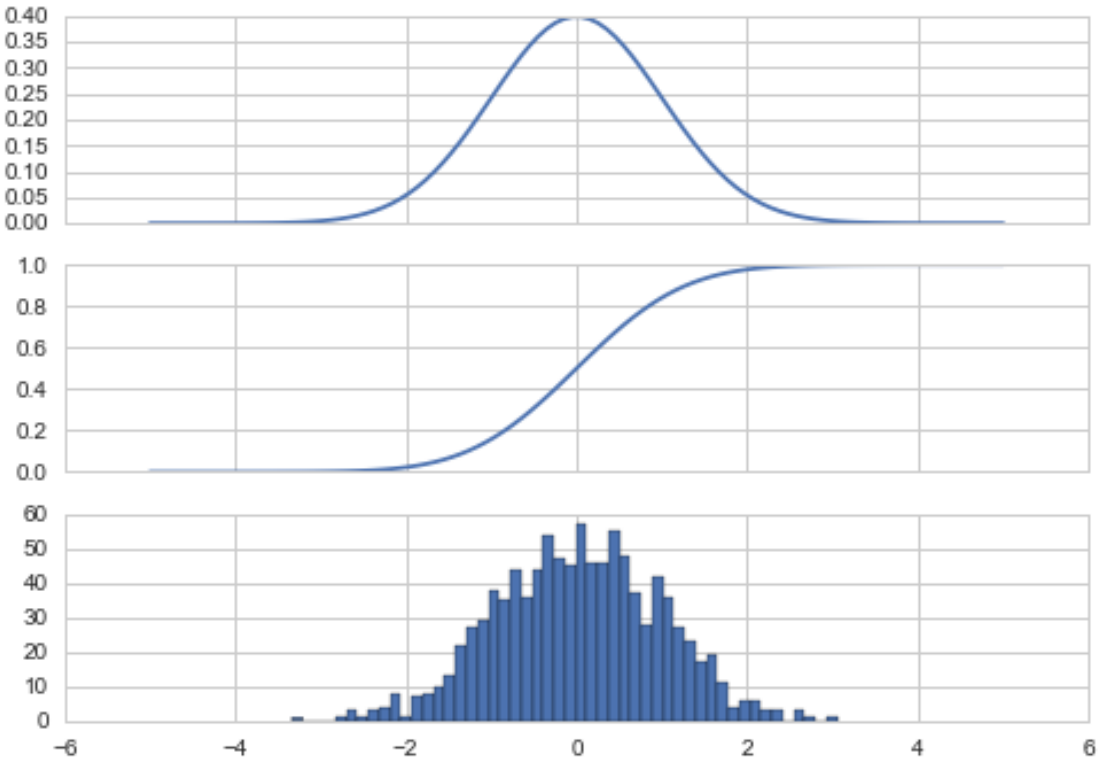
```
In [79]: X = stats.poisson(3.5)
```

```
In [80]: n = np.arange(0,15)

         fig, axes = plt.subplots(3,1, sharex=True)

         # plot the probability mass function (PMF)
         axes[0].step(n, X.pmf(n))

         # plot the commulative distribution function (CDF)
         axes[1].step(n, X.cdf(n))

         # plot histogram of 1000 random realizations of the stochastic variable X
         axes[2].hist(X.rvs(size=1000));
```



Continuous random variables:

```
In [81]: Y = stats.norm()

In [82]: x = np.linspace(-5,5,100)

         fig, axes = plt.subplots(3,1, sharex=True)

         # plot the probability distribution function (PDF)
         axes[0].plot(x, Y.pdf(x))

         # plot the commulative distributin function (CDF)
         axes[1].plot(x, Y.cdf(x));

         # plot histogram of 1000 random realizations of the stochastic variable Y
         axes[2].hist(Y.rvs(size=1000), bins=50);
```



1.7 References

Scientific Python Lecture Notes

Official SciPy Tutorial