1 Introduction

SciPy is a collection of mathematical algorithms and convenience functions built on the Numeric extension for Python. It adds significant power to the interactive Python session by exposing the user to high-level commands and classes for the manipulation and visualization of data. With SciPy, an interactive Python session becomes a data-processing and system-prototyping environment rivaling sytems such as Matlab, IDL, Octave, R-Lab, and SciLab.

The additional power of using SciPy within Python, however, is that a powerful programming language is also available for use in developing sophisticated programs and specialized applications. Scientific applications written in SciPy benefit from the development of additional modules in numerous niche's of the software landscape by developers across the world. Everything from parallel programming to web and database subroutines and classes have been made available to the Python programmer. All of this power is available in addition to the mathematical libraries in SciPy.

This document provides a tutorial for the first-time user of SciPy to help get started with some of the features available in this powerful package. It is assumed that the user has already installed the package. Some general Python facility is also assumed such as could be acquired by working through the Tutorial in the Python distribution. Throughout this tutorial it is assumed that the user has imported all of the names defined in the SciPy namespace using the command

```
>>> from scipy import *
```

2 General help

Python provides the facility of documentation strings. The functions and classes available in SciPy use this method for on-line documentation. There are two methods for reading these messages and getting help. Python provides the command **help** in the pydoc module. Entering this command with no arguments (i.e. >>> help) launches an interactive help session that allows searching through the keywords and modules available to all of Python. Running the command help with an object as the argument displays the calling signature, and the documentation string of the object.

The pydoc method of help is sophisticated but uses a pager to display the text. Sometimes this can interfere with the terminal you are running the interactive session within. A scipy-specific help system is also available under the command scipy.help. The signature and documntation string for the object passed to the help command are printed to standard output (or to a writeable object passed as the third argument). The second keyword argument of "scipy.help" defines the maximum width of the line for printing.

If a module is passed as the argument to help than a list of the functions and classes defined in that module is printed. For example:

```
>>> help(optimize.fmin)
 fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001, maxiter=None, maxfun=None,
      full_output=0, printmessg=1)

Minimize a function using the simplex algorithm.

Description:

  Uses a Nelder-Mead simplex algorithm to find the minimum of function
  of one or more variables.

Inputs:
```

```
  func -- the Python function or method to be minimized.
  x0 -- the initial guess.
  args -- extra arguments for func.
  xtol -- relative tolerance

Outputs: (xopt, {fopt, warnflag})

  xopt -- minimizer of function

  fopt -- value of function at minimum: fopt = func(xopt)
  warnflag -- Integer warning flag:
              1 : 'Maximum number of function evaluations.'
              2 : 'Maximum number of iterations.'

Additional Inputs:

  xtol -- acceptable relative error in xopt for convergence.
  ftol -- acceptable relative error in func(xopt) for convergence.
  maxiter -- the maximum number of iterations to perform.
  maxfun -- the maximum number of function evaluations.
  full_output -- non-zero if fval and warnflag outputs are desired.
  printmessg -- non-zero to print convergence messages.
```

3 Special Functions (special)

3.1 Vectorizing functions (special.general_function)

One of the features that the **special** sub-package provides is a class **general_function** to convert an ordinary Python function which accepts scalars and returns scalars into a "vectorized-function" with the same broadcasting rules as other Numeric functions (*i.e.* the Universal functions, or ufuncs). For example, suppose you have a Python function named **addsubtract** defined as:

```
>>> def addsubtract(a,b):
    if a > b:
        return a - b
    else:
        return a + b
```

which defines a function of two scalar variables and returns a scalar result. The class general_function can be used to "vectorize" this function so that

```
>>> vec_addsubstract = special.general_function(addsubtract)
```

returns a function which takes array arguments and returns an array result:

```
>>> vec_addsubtract([0,3,6,9],[1,3,5,7])
array([1, 6, 1, 2])
```

3.2 Special Functions

The main feature of the **special** package is the definition of numerous special functions of mathematical physics. Available are airy, elliptic, bessel, gamma, beta, hypergeometric, and several statistical functions. All of these functions behave can take array arguments and return array results following the same broadcasting rules as other math functions in Numerical Python. For a complete list of these functions with a one-line description type `>>>help(special)`. Each function also has it's own documentation accessible using help.

4 Integration (integrate)

The **integrate** sub-package provides several integration techniques including an ordinary differential equation integrator. An overview of the module is provided by the help command:

```
>>> help(integrate)
Methods for Integrating Functions

  odeint        -- Integrate ordinary differential equations.
  quad          -- General purpose integration.
  dblquad       -- General purpose double integration.
  tplquad       -- General purpose triple integration.
  gauss_quad    -- Integrate func(x) using Gaussian quadrature of order n.
  gauss_quadtol -- Integrate with given tolerance using Gaussian quadrature.

  See the orthogonal module (integrate.orthogonal) for Gaussian
     quadrature roots and weights.
```

4.1 General integration (integrate.quad)

The function **quad** is provided to integrate a function of one variable between two points. The points can be $\pm\infty$ ($\pm$**integrate.inf**) to indicate infinite limits. For example, suppose you wish to integrate a bessel function `jv(2.5,x)` along the interval $[0, 4.5]$.

$$I = \int_0^{4.5} J_{2.5}(x)\, dx.$$

This could be computed using **quad:**

```
>>> result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
>>> print result
(1.1178179380783249, 7.8663172481899801e-09)

>>> I = sqrt(2/pi)*(18.0/27*sqrt(2)*cos(4.5)-4.0/27*sqrt(2)*sin(4.5)+
    sqrt(2*pi)*special.fresnl(3/sqrt(pi))[0])
>>> print I
1.117817938088701

>>> print abs(result[0]-I)
1.03761443881e-11
```

The first argument to quad is a "callable" Python object (*i.e* a function, method, or class instance). Notice the use of a lambda-function in this case as the argument. The next two arguments are the limits of integration. The return value is a tuple, with the first element holding the estimated value of the integral and the second element holding an upper bound on the error. Notice, that in this case, the true value of this integral is

$$I = \sqrt{\frac{2}{\pi}}\left(\frac{18}{27}\sqrt{2}\cos\left(4.5\right) - \frac{4}{27}\sqrt{2}\sin\left(4.5\right) + \sqrt{2\pi}\mathrm{Si}\left(\frac{3}{\sqrt{\pi}}\right)\right),$$

where

$$\mathrm{Si}\left(x\right) = \int_0^x \sin\left(\frac{\pi}{2}t^2\right)\, dt.$$

is the Fresnel sine integral. Note that the numerically-computed integral is within 1.04×10^{-11} of the exact result — well below the reported error bound.

Infinite inputs are also allowed in **quad** by using $\pm$**integrate.inf** as one of the arguments. For example, suppose that a numerical value for the exponential integral:

$$E_n\left(x\right) = \int_1^\infty \frac{e^{-xt}}{t^n}\, dt.$$

is desired (and the fact that this integral can be computed as `special.expn(n,x)` is forgotten). The functionality of the function **special.expn** can be replicated by defining a new function **vec_expint** based on the routine **quad:**

```
>>> from integrate import quad, Inf
>>> def integrand(t,n,x):
        return exp(-x*t) / t**n

>>> def expint(n,x):
        return quad(integrand, 1, Inf, args=(n, x))[0]

>>> vec_expint = special.general_function(expint)

>>> vec_expint(3,arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
>>> special.expn(3,arange(1.0,4.0,0.5))
array([ 0.1097,  0.0567,  0.0301,  0.0163,  0.0089,  0.0049])
```

The function which is integrated can even use the quad argument (though the error bound may underestimate the error due to possible numerical error in the integrand from the use of **quad**). The integral in this case is

$$I_n = \int_0^\infty \int_1^\infty \frac{e^{-xt}}{t^n}\, dt\, dx = \frac{1}{n}.$$

```
>>> result = quad(lambda x: expint(3, x), 0, Inf)
>>> print result
(0.33333333324560266, 2.8548934485373678e-09)

>>> I3 = 1.0/3.0
>>> print I3
0.333333333333
```

```
>>> print I3 - result[0]
8.77306560731e-11
```

This last example shows that multiple integration can be handled using repeated calls to **quad.** The mechanics of this for double and triple integration have been wrapped up into the functions **dblquad** and **tplquad.** The function, **dblquad** performs double integration. Use the help function to be sure that the arguments are defined in the correct order. In addition, the limits on all inner integrals are actually functions which can be constant functions. An example of using double integration to compute several values of I_n is shown below:

```
>>> from __future__ import nested_scopes
>>> from integrate import quad, dblquad, Inf
>>> def I(n):
    return dblquad(lambda t, x: exp(-x*t)/t**n, 0, Inf, lambda x: 1, lambda x: Inf)

>>> print I(4)
(0.25000000000435768, 1.0518245707751597e-09)
>>> print I(3)
(0.33333333325010883, 2.8604069919261191e-09)
>>> print I(2)
(0.49999999999857514, 1.8855523253868967e-09)
```

4.2 Ordinary differential equations (integrate.odeint)

Integrating a set of ordinary differential equations (ODEs) given initial conditions is another useful example. The function **odeint** is available in SciPy for integrating a first-order vector differential equation:

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}\left(\mathbf{y}, t\right),$$

given initial conditions $\mathbf{y}\left(0\right) = y_0$, where $\mathbf{y}$ is a length N vector and $\mathbf{f}$ is a mapping from $\mathcal{R}^N$ to $\mathcal{R}^N$. A higher-order ordinary differential equation can always be reduced to a differential equation of this type by introducing intermediate derivatives into the $\mathbf{y}$ vector.

For example suppose it is desired to find the solution to the following second-order differential equation:

$$\frac{d^2w}{dz^2} - zw(z) = 0$$

with initial conditions $w\left(0\right) = \frac{1}{\sqrt[3]{3^2}\Gamma\left(\frac{2}{3}\right)}$ and $\frac{dw}{dz}\big|_{z=0} = -\frac{1}{\sqrt[3]{3}\Gamma\left(\frac{1}{3}\right)}$. It is known that the solution to this differential equation with these boundary conditions is the Airy function

$$w = \mathrm{Ai}\left(z\right),$$

which gives a means to check the integrator using **special.airy.**

First, convert this ODE into standard form by setting $\mathbf{y} = \left[\frac{dw}{dz}, w\right]$ and $t = z$. Thus, the differential equation becomes

$$\frac{d\mathbf{y}}{dt} = \begin{bmatrix} ty_1 \\ y_0 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & t \\ 1 & 0 \end{bmatrix} \mathbf{y}.$$

In other words,

$$\mathbf{f}\left(\mathbf{y}, t\right) = \mathbf{A}\left(t\right)\mathbf{y}.$$

As an interesting reminder, if $\mathbf{A}(t)$ commutes with $\int_0^t \mathbf{A}(\tau)\, d\tau$ under matrix multiplication, then this linear differential equation has an exact solution using the matrix exponential:

$$\mathbf{y}(t) = \exp\left(\int_0^t \mathbf{A}(\tau)\, d\tau\right) \mathbf{y}(0),$$

However, in this case, $\mathbf{A}(t)$ and its integral do not commute.

There are many optional inputs and outputs available when using odeint which can help tune the solver. These additional inputs and outputs are not needed much of the time, however, and the three required input arguments and the output solution suffice. The required inputs are the function defining the derivative, *fprime,* the initial conditions vector, *y0*, and the time points to obtain a solution, *t,* (with the initial value point as the first element of this sequence). The output to **odeint** is a matrix where each row contains the solution vector at each requested time point (thus, the initial conditions are given in the first output row).

The following example illustrates the use of odeint including the usage of the **Dfun** option which allows the user to specify a gradient (with respect to $\mathbf{y}$) of the function, $\mathbf{f}(\mathbf{y}, t)$**.**

```
>>> from integrate import odeint
>>> from special import gamma, airy
>>> y1_0 = 1.0/3**(2.0/3.0)/gamma(2.0/3.0)
>>> y0_0 = -1.0/3**(1.0/3.0)/gamma(1.0/3.0)
>>> y0 = [y0_0, y1_0]
>>> def func(y, t):
        return [t*y[1],y[0]]

>>> def gradient(y,t):
        return [[0,t],[1,0]]

>>> x = arange(0,4.0, 0.01)
>>> t = x
>>> ychk = airy(x)[0]
>>> y = odeint(func, y0, t)
>>> y2 = odeint(func, y0, t, Dfun=gradient)

>>> import sys
>>> sys.float_output_precision = 6
>>> print ychk[:36:6]
[ 0.355028  0.339511  0.324068  0.308763  0.293658  0.278806]

>>> print y[:36:6,1]
[ 0.355028  0.339511  0.324067  0.308763  0.293658  0.278806]

>>> print y2[:36:6,1]
[ 0.355028  0.339511  0.324067  0.308763  0.293658  0.278806]
```

4.3 Gaussian quadrature (integrate.gauss_quadtol)

A few functions are also provided in order to perform simple Gaussian quadrature over a fixed interval. The first is **gauss_quad** which performs fixed-order Gaussian quadrature. The second function is **gauss_quadtol** which performs Gaussian quadrature of multiple orders until the difference in the integral estimate is beneath

some tolerance supplied by the user. These functions both use the module **integrate.orthogonal** which can calculate the roots and quadrature weights of a large variety of orthogonal polynomials.

5 Optimization (optimize)

There are several classical optimization algorithms provided by SciPy in the **optimize** package. An overview of the module is available using **help** (or pydoc.help):

```
>>> help(optimize)

 Optimization Tools

A collection of general-purpose optimization routines.

  fmin --       Nelder-Mead Simplex algorithm
                 (uses only function calls)
  fmin_bfgs --  Quasi-Newton method (can use function and gradient)
  fmin_ncg --   Line-search Newton Conjugate Gradient (can use
                 function, gradient and hessian).
  leastsq --    Minimize the sum of squares of M equations in
                 N unknowns given a starting estimate.
  fminbound --  Bounded minimization of a scalar function.
  fsolve --     Non-linear equation solver.
```

The first four algorithms are unconstrained minimization algorithms (fmin: Nelder-Mead simplex, fmin_bfgs: BFGS, fmin_ncg: Newton Conjugate Gradient, and leastsq: Levenburg-Marquardt). The fourth algorithm only works for functions of a single variable but allows minimization over a specified interval. The last algorithm actually finds the roots of a general function of possibly many variables. It is included in the optimization package because at the (non-boundary) extreme points of a function, the gradient is equal to zero.

5.1 Nelder-Mead Simplex algorithm (optimize.fmin)

The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. The simplex algorithm requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum. To demonstrate the minimization function consider the problem of minimizing the Rosenbrock function of N variables:

$$f\left(\mathbf{x}\right)=\sum_{i=1}^{N-1}100\left(x_{i}-x_{i-1}^{2}\right)^{2}+\left(1-x_{i-1}\right)^{2}.$$

The minimum value of this function is 0 which is achieved when $x_i = 1$. This minimum can be found using the **fmin** routine as shown in the example below:

```
>>> from optimize import fmin
>>> def rosen(x):  # The Rosenbrock function
        return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
```

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin(rosen, x0)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 516
         Function evaluations: 825

>>> print xopt
[ 1.  1.  1.  1.  1.]
```

5.2 Broyden-Fletcher-Goldfarb-Shanno algorithm (optimize.fmin_bfgs)

In order to converge more quickly to the solution, this routine uses the gradient of the objective function. If the gradient is not given by the user, then it is estimated using first-differences. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method requires fewer function calls than the simplex algorithm but unless the gradient is provided by the user, the speed savings won't be significant.

To demonstrate this algorithm, the Rosenbrock function is again used. The gradient of the Rosenbrock function is the vector:

$$\begin{aligned}\frac{\partial f}{\partial x_j} &= \sum_{i=1}^{N} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2\left(1 - x_{i-1}\right)\delta_{i-1,j}. \\ &= 200\left(x_j - x_{j-1}^2\right) - 400x_j\left(x_{j+1} - x_j^2\right) - 2\left(1 - x_j\right).\end{aligned}$$

This expression is valid for the interior derivatives. Special cases are

$$\begin{aligned}\frac{\partial f}{\partial x_0} &= -400x_0\left(x_1 - x_0^2\right) - 2\left(1 - x_0\right), \\ \frac{\partial f}{\partial x_{N-1}} &= 200\left(x_{N-1} - x_{N-2}^2\right).\end{aligned}$$

A Python function which computes this gradient is constructed by the code-segment:

```
>>> def rosen_der(x):
        xm = x[1:-1]
        xm_m1 = x[:-2]
        xm_p1 = x[2:]
        der = zeros(x.shape,x.typecode())
        der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
        der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
        der[-1] = 200*(x[-1]-x[-2]**2)
        return der
```

The calling signature for the BFGS minimization algorithm is similar to **fmin** with the addition of the *fprime* argument. An example usage of **fmin_bfgs** is shown in the following example which minimizes the Rosenbrock function.

```
>>> from optimize import fmin_bfgs
```

```
>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_bfgs(rosen, x0, fprime=rosen_der)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 109
         Function evaluations: 262
         Gradient evaluations: 110
>>> print xopt
[ 1.  1.  1.  1.  1.]
```

5.3 Newton-Conjugate-Gradient (optimize.fmin_ncg)

The method which requires the fewest function calls and is therefore often the fastest method to minimize functions of many variables is **fmin_ncg.** This method is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian. Newton's method is based on fitting the function locally to a quadratic form:

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^T \mathbf{H}(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0).$$

where $\mathbf{H}(\mathbf{x}_0)$ is a matrix of second-derivatives (the Hessian). If the Hessian is positive definite then the local minimum of this function can be found by setting the gradient of the quadratic form to zero, resulting in

$$\mathbf{x}_{\text{opt}} = \mathbf{x}_0 - \mathbf{H}^{-1} \nabla f.$$

The inverse of the Hessian is evaluted using the conjugate-gradient method. An example of employing this method to minimizing the Rosenbrock function is given below. To take full advantage of the NewtonCG method, a function which computes the Hessian must be provided. The Hessian matrix itself does not need to be constructed, only a vector which is the product of the Hessian with an arbitrary vector needs to be available to the minimization routine. As a result, the user can provide either a function to compute the Hessian matrix, or a function to compute the product of the Hessian with an arbitrary vector.

5.3.1 Full Hessian example:

The Hessian of the Rosenbrock function is

$$\begin{aligned} H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} &= 200 (\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 400 x_i (\delta_{i+1,j} - 2x_i \delta_{i,j}) - 400 \delta_{i,j} \left(x_{i+1} - x_i^2\right) + 2\delta_{i,j}, \\ &= \left(202 + 1200 x_i^2 - 400 x_{i+1}\right) \delta_{i,j} - 400 x_i \delta_{i+1,j} - 400 x_{i-1} \delta_{i-1,j}, \end{aligned}$$

if $i, j \in [1, N-2]$ with $i, j \in [0, N-1]$ defining the $N \times N$ matrix. Other non-zero entries of the matrix are

$$\begin{aligned} \frac{\partial^2 f}{\partial x_0^2} &= 1200 x_0^2 - 400 x_1 + 2, \\ \frac{\partial^2 f}{\partial x_0 \partial x_1} = \frac{\partial^2 f}{\partial x_1 \partial x_0} &= -400 x_0, \\ \frac{\partial^2 f}{\partial x_{N-1} \partial x_{N-2}} = \frac{\partial^2 f}{\partial x_{N-2} \partial x_{N-1}} &= -400 x_{N-2}, \\ \frac{\partial^2 f}{\partial x_{N-1}^2} &= 200. \end{aligned}$$

For example, the Hessian when $N = 5$ is

$$\mathbf{H} = \begin{bmatrix} 1200x_0^2 - 400x_1 + 2 & -400x_0 & 0 & 0 & 0 \\ -400x_0 & 202 + 1200x_1^2 - 400x_2 & -400x_1 & 0 & 0 \\ 0 & -400x_1 & 202 + 1200x_2^2 - 400x_3 & -400x_2 & 0 \\ 0 & & -400x_2 & 202 + 1200x_3^2 - 400x_4 & -400x_3 \\ 0 & 0 & 0 & -400x_3 & 200 \end{bmatrix}.$$

The code which computes this Hessian along with the code to minimize the function using **fmin_ncg** is shown in the following example:

```
>>> from optimize import fmin_ncg
>>> def rosen_hess(x):
        x = asarray(x)
        H = diag(-400*x[:-1],1) - diag(400*x[:-1],-1)
        diagonal = zeros(len(x),x.typecode())
        diagonal[0] = 1200*x[0]-400*x[1]+2
        diagonal[-1] = 200
        diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
        H = H + diag(diagonal)
        return H

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_ncg(rosen, x0, rosen_der, fhess=rosen_hess)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 19
         Function evaluations: 40
         Gradient evaluations: 19
         Hessian evaluations: 19
>>> print xopt
[ 0.9999  0.9999  0.9998  0.9996  0.9991]
```

5.3.2 Hessian product example:

For larger minimization problems, storing the entire Hessian matrix can consume considerable time and memory. The Newton-CG algorithm only needs the product of the Hessian times an arbitrary vector. As a result, the user can supply code to compute this product rather than the full Hessian by setting the *fhess_p* keyword to the desired function. The fhess_p function should take the minimization vector as the first argument and the arbitrary vector as the second argument. Any extra arguments passed to the function to be minimized will also be passed to this function. If possible, using Newton-CG with the hessian product option is probably the fastest way to minimize the function.

In this case, the product of the Rosenbrock Hessian with an arbitrary vector is not difficult to compute. If $\mathbf{p}$ is the arbitrary vector, then $\mathbf{H}(\mathbf{x})\mathbf{p}$ has elements:

$$\mathbf{H}(\mathbf{x})\mathbf{p} = \begin{bmatrix} \left(1200x_0^2 - 400x_1 + 2\right)p_0 - 400x_0p_1 \\ \vdots \\ -400x_{i-1}p_{i-1} + \left(202 + 1200x_i^2 - 400x_{i+1}\right)p_i - 400x_ip_{i+1} \\ \vdots \\ -400x_{N-2}p_{N-2} + 200p_{N-1} \end{bmatrix}.$$

Code which makes use of the *fhess_p* keyword to minimize the Rosenbrock function using **fmin_ncg** follows:

```
>>> from optimize import fmin_ncg
>>> def rosen_hess_p(x,p):
        x = asarray(x)
        Hp = zeros(len(x),x.typecode())
        Hp[0] = (1200*x[0]**2 - 400*x[1] + 2)*p[0] - 400*x[0]*p[1]
        Hp[1:-1] = -400*x[:-2]*p[:-2]+(202+1200*x[1:-1]**2-400*x[2:])*p[1:-1] \
                   -400*x[1:-1]*p[2:]
        Hp[-1] = -400*x[-2]*p[-2] + 200*p[-1]
        return Hp

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> xopt = fmin_ncg(rosen, x0, rosen_der, fhess_p=rosen_hess_p)
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 42
         Gradient evaluations: 20
         Hessian evaluations: 44
>>> print xopt
[ 1.      1.      1.      0.9999  0.9999]
```

5.4 Least-square fitting (minimize.leastsq)

All of the previously-explained minimization procedures can be used to solve a least-squares problem provided the appropriate objective function is constructed. For example, suppose it is desired to fit a set of data $\{\mathbf{x}_i, \mathbf{y}_i\}$ to a known model, $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{p})$ where $\mathbf{p}$ is a vector of parameters for the model that need to be found. A common method for determining which parameter vector gives the best fit to the data is to minimize the sum of squares of the residuals. The residual is usually defined for each observed data-point as

$$e_i(\mathbf{p}, \mathbf{y}_i, \mathbf{x}_i) = \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i, \mathbf{p})\|.$$

An objective function to pass to any of the previous minization algorithms to obtain a least-squares fit is.

$$J(\mathbf{p}) = \sum_{i=0}^{N-1} e_i^2(\mathbf{p}).$$

The **leastsq** algorithm performs this squaring and summing of the residuals automatically. It takes as an input argument the vector function $\mathbf{e}(\mathbf{p})$ and returns the value of $\mathbf{p}$ which minimizes $J(\mathbf{p}) = \mathbf{e}^T\mathbf{e}$ directly. The user is also encouraged to provide the Jacobian matrix of the function (with derivatives down the columns or across the rows). If the Jacobian is not provided, it is estimated.

An example should clarify the usage. Suppose it is believed some measured data follow a sinusoidal pattern

$$y_i = A\sin(2\pi k x_i + \theta)$$

where the parameters A, k, and θ are unknown. The residual vector is

$$e_i = |y_i - A\sin(2\pi k x_i + \theta)|.$$

By defining a function to compute the residuals and (selecting an appropriate starting position), the least-squares fit routine can be used to find the best-fit parameters $\hat{A}$, $\hat{k}$, $\hat{\theta}$. This is shown in the following example and a plot of the results is shown in Figure 1.

```
>>> x = arange(0,6e-2,6e-2/30)
>>> A,k,theta = 10, 1.0/3e-2, pi/6
>>> y_true = A*sin(2*pi*k*x+theta)
>>> y_meas = y_true + 2*randn(len(x))

>>> def residuals(p, y, x):
        A,k,theta = p
        err = y-A*sin(2*pi*k*x+theta)
        return err

>>> def peval(x, p):
        return p[0]*sin(2*pi*p[1]*x+p[2])

>>> p0 = [8, 1/2.3e-2, pi/3]
>>> print array(p0)
[  8.      43.4783   1.0472]

>>> from optimize import leastsq
>>> plsq = leastsq(residuals, p0, args=(y_meas, x))
>>> print plsq[0]
[ 10.9437  33.3605   0.5834]

>>> print array([A, k, theta])
[ 10.      33.3333   0.5236]

>>> from xplt import *    # Only on X-windows systems
>>> plot(x,peval(x,plsq[0]),x,y_meas,'o',x,y_true)
>>> title('Least-squares fit to noisy data')
>>> legend(['Fit', 'Noisy', 'True'])
>>> gist.eps('leastsqfit')   # Make epsi file.
```

5.5 Bounded minimization (optimize.fminbound)

Thus far all of the minimization routines described have been unconstrained minimization routines. Very often, however, there are constraints that can be placed on the solution space before minimization occurs. The **fminbound** function is an example of a constrained minimization procedure that provides a rudimentary interval constraint for scalar functions (functions which take a scalar input and return a scalar output). The interval constraint allows the minimization to occur only between two fixed endpoints.

For example, to find the minimum of $J_1(x)$ near $x = 5$, **fminbound** can be called using the interval $[4, 7]$ as a constraint. The result is $x_{\min} = 5.3314$:

```
>>> from special import j1

>>> from optimize import fminbound
```

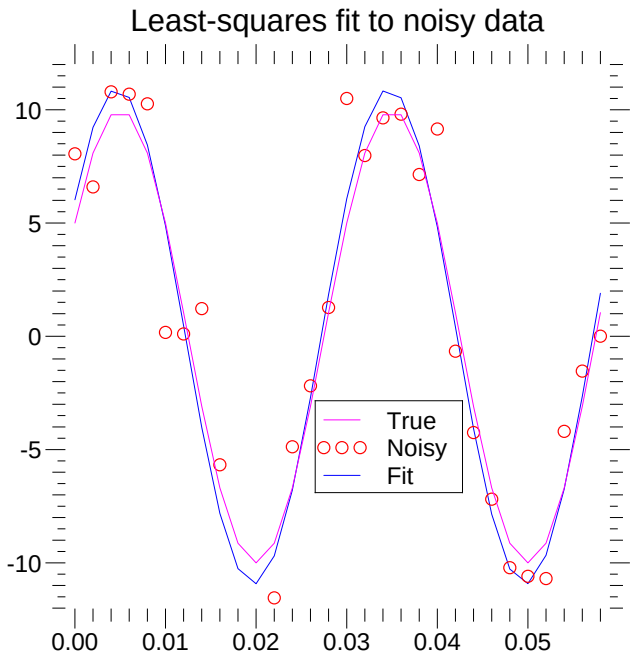


Figure 1: Least-square fitting to noisy data using **scipy.optimize.leastsq**

```
>>> xmin = fminbound(j1, 4, 7)
>>> print xmin
5.33144184241
```

5.6 Root finding (optimize.fsolve)

To find the roots of a polynomial, the command **roots** from Numeric Python is useful (this is also available as **roots**). To find a root of a set of non-linear equations, the command **optimize.fsolve** is needed. For example, the following example finds the roots of the single-variable transcendental equation

$$x + 2\cos(x) = 0,$$

and the set of non-linear equations

$$\begin{aligned} x_0 \cos(x_1) &= 4, \\ x_0 x_1 - x_1 &= 5. \end{aligned}$$

The results are $x = -1.0299$ and $x_0 = 6.5041$, $x_1 = 0.9084$.

```
>>> def func(x):
        return x + 2*cos(x)

>>> def func2(x):
        out = [x[0]*cos(x[1]) - 4]
        out.append(x[1]*x[0] - x[1] - 5)
```

```
        return out

>>> from optimize import fsolve
>>> x0 = fsolve(func, 0.3)
>>> print x0
-1.02986652932

>>> x02 = fsolve(func2, [1, 1])
>>> print x02
[ 6.5041  0.9084]
```

6 Interpolation (interpolate)

There are two general interpolation facilities available in SciPy. The first facility is an interpolation class which performs linear 1-dimensional interpolation. The second facility is based on the FORTRAN library FITPACK and provides functions for 1- and 2-dimensional (smoothed) cubic-spline interpolation.

6.1 Linear 1-d interpolation (interpolate.linear_1d)

The linear_1d class in scipy.interpolate is a convenient method to create a function based on fixed data points which can be evaluated anywhere within the domain defined by the given data using linear interpolation. An instance of this class is created by passing the 1-d vectors comprising the data. The instance of this class defines a *__call__* method and can therefore by treated like a function which interpolates between known data values to obtain unknown values (it even has a docstring for help). Behavior at the boundary can be specified at instantiation time. The following example demonstrates it's use.

```
>>> x = arange(0,10)
>>> y = exp(-x/3.0)
>>> f = interpolate.linear_1d(x,y)
>>> help(f)
Instance of class:  linear_1d

 <name>(x_new)

Find linearly interpolated y_new = <name>(x_new).

Inputs:
  x_new -- New independent variables.

Outputs:
  y_new -- Linearly interpolated values corresponding to x_new.

>>> xnew = arange(0,9,0.1)
>>> xplt.plot(x,y,'x',xnew,f(xnew),'.')
```

Figure shows the result:

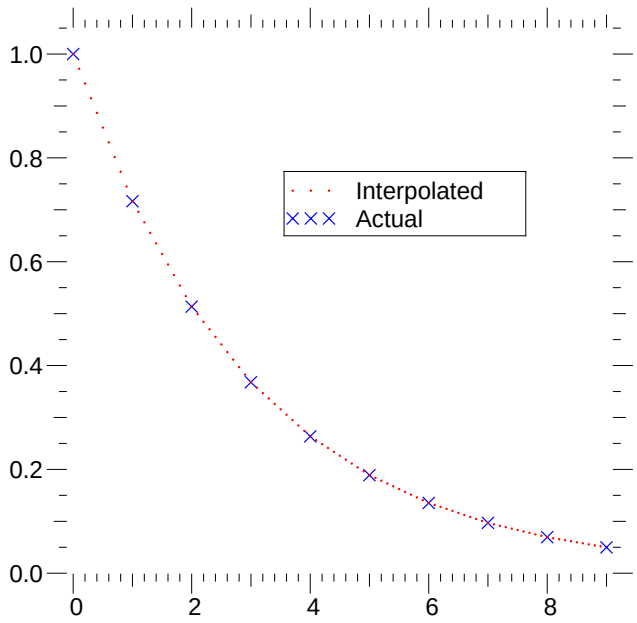


Figure 2: One-dimensional interpolation using the class **interpolate.linear_1d.**

6.2 Spline interpolation in 1-d (interpolate.splXXX)

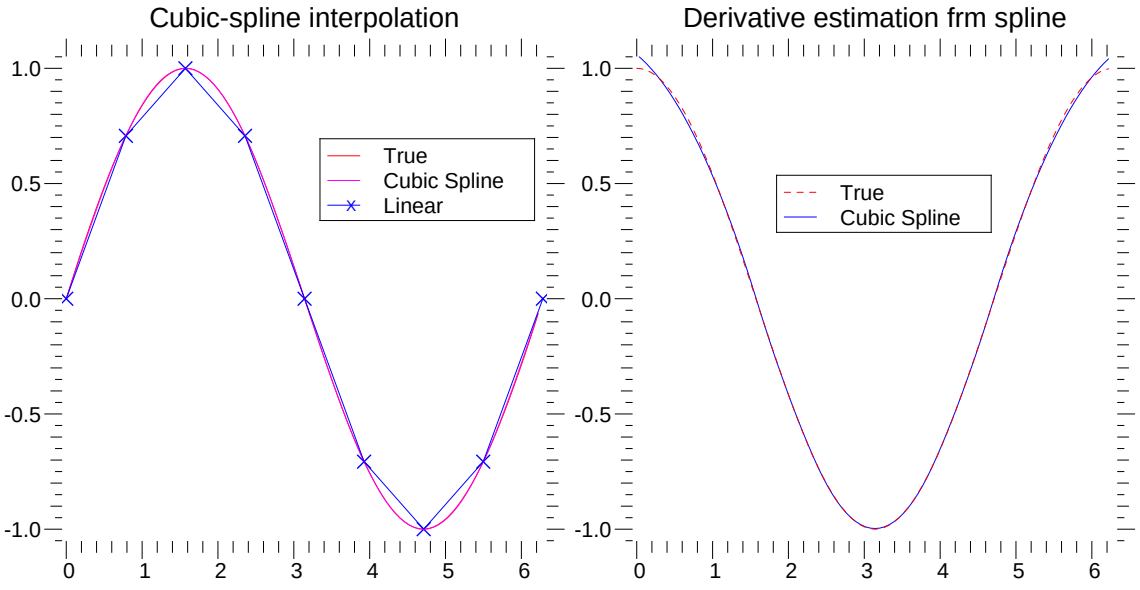
Spline interpolation requires two essential steps: (1) a spline representation of the curve is computed, and (2) the spline is evaluated at the desired points. In order to find the spline representation, there are two different was to represent a curve and obtain (smoothing) spline coefficients: directly and parametrically. The direct method finds the spline representation of a curve in a two-dimensional plane using the function **interpolate.splrep.** The first two arguments are the only ones required, and these provide the x and y components of the curve. The normal output is a 3-tuple, (t, c, k), containing the knot-points, t, the coefficients c and the order k of the spline. The default spline order is cubic, but this can be changed with the input keyword, k.

For curves in N-dimensional space the function **interpolate.splprep** allows defining the curve parametrically. For this function only 1 input argument is required. This input is a list of N-arrays representing the curve in N-dimensional space. The length of each array is the number of curve points, and each array provides one component of the N-dimensional data point. The parameter variable is given with the keword argument, u, which defaults to an equally-spaced monotonic sequence between 0 and 1. The default output consists of two objects: a 3-tuple, (t, c, k), containing the spline representation and the parameter variable u.

The keyword argument, s, is used to specify the amount of smoothing to perform during the spline fit. The default value of s is $s = m - \sqrt{2m}$ where m is the number of data-points being fit. Therefore, **if no smoothing is desired a value of s = 0 should be passed to the routines.**

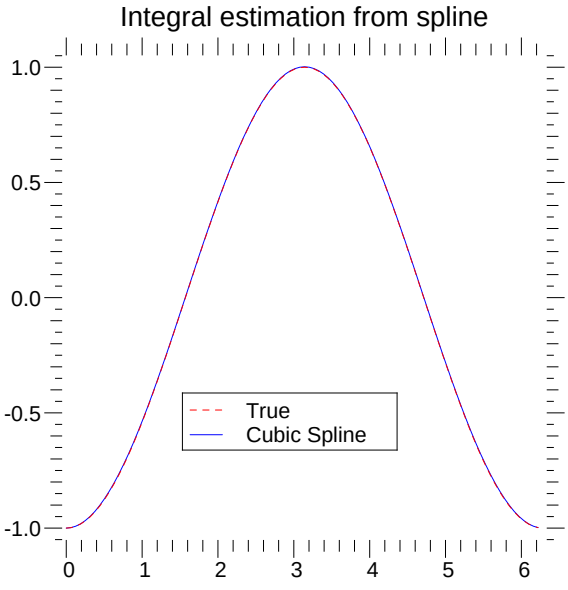
Once the spline representation of the data has been determined, functions are available for evaluating the spline (**interpolate.splev)** and its derivatives (**interpolate.splev, interpolate.splade**) at any point and the integral of the spline between any two points (**interpolate.splint)**. In addition, for cubic splines ($k = 3$) with 8 or more knots, the roots of the spline can be estimated (**interpolate.sproot)**. These functions are demonstrated in the example that follows (see also Figure 3).

```
>>> # Cubic-spline
```

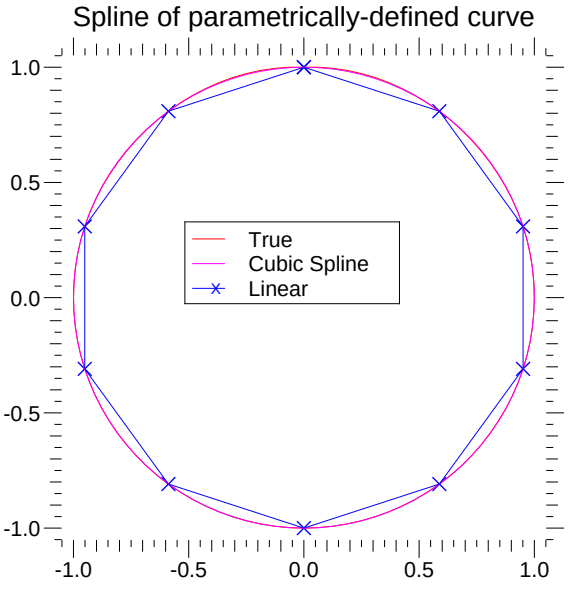


(a) Cubic-spline (splrep)

(b) Derivative of spline (splev)



(c) Integral of spline (splint)



(d) Spline of parametric curve (splprep)

Figure 3: Examples of using cubic-spline interpolation.

```
>>> x = arange(0,2*pi+pi/4,2*pi/8)
>>> y = sin(x)
>>> tck = interpolate.splrep(x,y,s=0)
>>> xnew = arange(0,2*pi,pi/50)
>>> ynew = interpolate.splev(xnew,tck,der=0)
>>> xplt.plot(x,y,'x',xnew,ynew,xnew,sin(xnew),x,y,'b')
>>> xplt.legend(['Linear','Cubic Spline', 'True'],['b-x','m','r'])
>>> xplt.limits(-0.05,6.33,-1.05,1.05)
>>> xplt.title('Cubic-spline interpolation')
>>> xplt.eps('interp_cubic')

>>> # Derivative of spline
>>> yder = interpolate.splev(xnew,tck,der=1)
>>> xplt.plot(xnew,yder,xnew,cos(xnew),'|')
>>> xplt.legend(['Cubic Spline', 'True'])
>>> xplt.limits(-0.05,6.33,-1.05,1.05)
>>> xplt.title('Derivative estimation from spline')
>>> xplt.eps('interp_cubic_der')

>>> # Integral of spline
>>> def integ(x,tck,constant=-1):
>>>     x = asarray_1d(x)
>>>     out = zeros(x.shape, x.typecode())
>>>     for n in xrange(len(out)):
>>>         out[n] = interpolate.splint(0,x[n],tck)
>>>     out += constant
>>>     return out
>>>
>>> yint = integ(xnew,tck)
>>> xplt.plot(xnew,yint,xnew,-cos(xnew),'|')
>>> xplt.legend(['Cubic Spline', 'True'])
>>> xplt.limits(-0.05,6.33,-1.05,1.05)
>>> xplt.title('Integral estimation from spline')
>>> xplt.eps('interp_cubic_int')

>>> # Roots of spline
>>> print interpolate.sproot(tck)
[ 0.      3.1416]

>>> # Parametric spline
>>> t = arange(0,1.1,.1)
>>> x = sin(2*pi*t)
>>> y = cos(2*pi*t)
>>> tck,u = interpolate.splprep([x,y],s=0)
>>> unew = arange(0,1.01,0.01)
>>> out = interpolate.splev(unew,tck)
>>> xplt.plot(x,y,'x',out[0],out[1],sin(2*pi*unew),cos(2*pi*unew),x,y,'b')
>>> xplt.legend(['Linear','Cubic Spline', 'True'],['b-x','m','r'])
```

```
>>> xplt.limits(-1.05,1.05,-1.05,1.05)
>>> xplt.title('Spline of parametrically-defined curve')
>>> xplt.eps('interp_cubic_param')
```

6.3 Two-dimensionsal spline representation (interpolate.bisplrep)

For (smooth) spline-fitting to a two dimensional surface, the function **interpolate.bisplrep** is available. This function takes as required inputs the **1-D** arrays *x, y,* and z which represent points on the surface $z = f(x, y)$. The default output is a list $[tx, ty, c, kx, ky]$ whose entries represent respectively, the components of the knot positions, the coefficients of the spline, and the order of the spline in each coordinate. It is convenient to hold this list in a single object, *tck,* so that it can be passed easily to the function **interpolate.bisplev.** The keyword, s, can be used to change the amount of smoothing performed on the data while determining the appropriate spline. The default value is $s = m - \sqrt{2m}$ where m is the number of data points in the *x, y,* and z vectors. As a result, if no smoothing is desired, then $s = 0$ should be passed to **interpolate.bisplrep.**

To evaluate the two-dimensional spline and it's partial derivatives (up to the order of the spline), the function **interpolate.bisplev** is required. This function takes as the first two arguments **two 1-D arrays** whose cross-product specifies the domain over which to evaluate the spline. The third argument is the *tck* list returned from **interpolate.bisplrep.** If desired, the fourth and fifth arguments provide the orders of the partial derivative in the x and y direction respectively.

It is important to note that two dimensional interpolation should not be used to find the spline representation of images. The algorithm used is not amenable to large numbers of input points. The signal processing toolbox contains (soon) more appropriate algorithms for finding the spline representation of an image. The two dimensional interpolation commands are intended for use when interpolating a two dimensional function as shown in the example that follows (See also Figure 4). This example uses the **grid** command in SciPy which is useful for defining a "mesh-grid" in many dimensions. The number of output arguments and the number of dimensions of each argument is determined by the number of indexing objects passed in **grid[].**

```
>>> # Define function over sparse 20x20 grid
>>> x,y = grid[-1:1:20L,-1:1:20L]
>>> z = (x+y)*exp(-6.0*(x*x+y*y))
>>> xplt.plot3(x,y,z,shade=1,palette='rainbow')
>>> xplt.title3("Sparsely sampled function.")
>>> xplt.eps("2d_func")

>>> # Interpolate function over new 70x70 grid
>>> xnew,ynew = grid[-1:1:70L,-1:1:70L]
>>> tck = interpolate.bisplrep(x,y,z,s=0)
>>> znew = interpolate.bisplev(xnew[:,0],ynew[0,:],tck)
>>> xplt.plot3(xnew,ynew,znew,shade=1,palette='rainbow')
>>> xplt.title3("Interpolated function.")
>>> xplt.eps("2d_interp")
```

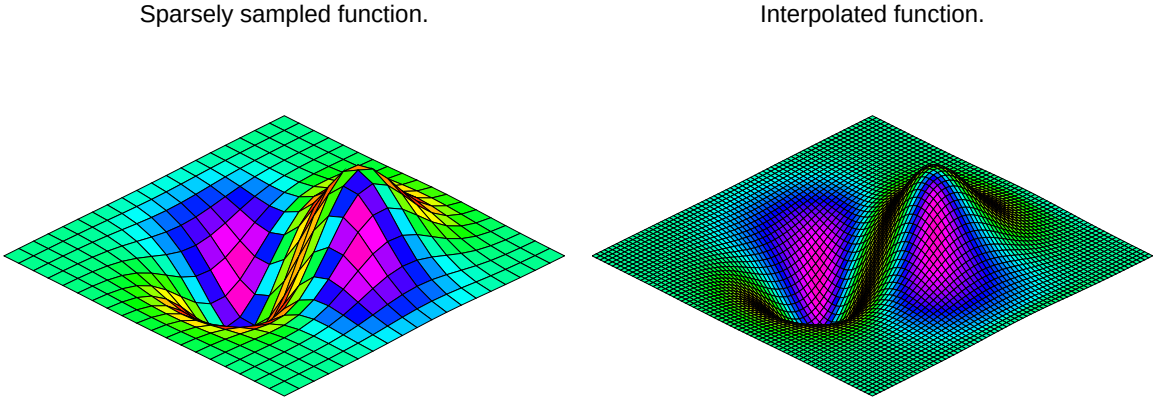


Figure 4: Example of two-dimensional spline interpolation.