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 *
```

1.1 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.info. 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.info" 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:

Another useful command is **source.** When given a function written in Python as an argument, it prints out a listing of the source code for that function. This can be helpful in learning about an algorithm or understanding exactly what a function is doing with its arguments. Also don't forget about the Python command `dir` which can be used to look at the namespace of a module or package.

1.2 SciPy Organization

SciPy is organized into subpackages covering different scientific computing domains. Some common functions which several subpackages rely on live under the `scipy_base` package which is installed at the same directory level as the scipy package itself and could be installed separately. This allows for the possibility of separately distributing the subpackages of scipy as long as scipy_base package is provided as well.

Two other packages are installed at the higher-level: scipy_distutils and weave. These two packages while distributed with main scipy package could see use independently of scipy and so are treated as separate packages and described elsewhere.

The remaining subpackages are summarized in the following table (a * denotes an optional sub-package that requires additional libraries to function or is not available on all platforms).

Subpackage	Description
cluster	Clustering algorithms
cow	Cluster of Workstations code for parallel programming
fftpack	FFT based on fftpack – default
fftw*	FFT based on fftw — requires FFTW libraries (is this still needed?)
ga	Genetic algorithms
gplt*	Plotting — requires gnuplot
integrate	Integration
interpolate	Interpolation
io	Input and Output
linalg	Linear algebra
optimize	Optimization and root-finding routines
plt*	Plotting — requires wxPython
signal	Signal processing
special	Special functions
stats	Statistical distributions and functions
xplt*	Plotting — requires an X server

Because of their ubiquitousness, some of the functions in these subpackages are also made available in the scipy namespace to ease their use in interactive sessions and programs. In addition, many convenience functions are located in the scipy_base package and the in the top-level of the scipy package. Before looking at the sub-packages individually, we will first look at some of these common functions.

2 Basic functions in scipy_base and top-level scipy

2.1 Interaction with Numeric

To begin with, all of the Numeric functions have been subsumed into the scipy namespace so that all of those functions are available without additionally importing Numeric. In addition, the universal functions (addition, subtraction, division) have been altered to not raise exceptions if floating-point errors are encountered[1], instead NaN's and Inf's are returned in the arrays. To assist in detection of these events new universal functions (isnan, isfinite, isinf) have been added. In addition, the comparision operators have been changed to allow comparisons and logical operations of complex numbers (only the real part is compared). Also, with the new universal functions in SciPy, the logical operations all return arrays of unsigned bytes (8-bits per element instead of the old 32-bits, or even 64-bits) per element[2].

Finally, some of the basic functions like log, sqrt, and inverse trig functions have been modified to return complex numbers instead of NaN's where appropriate (*i.e.* `scipy.sqrt(-1)` returns `1j`).

2.2 Scipy_base routines

The purpose of scipy_base is to collect general-purpose routines that the other sub-packages can use. These routines are divided into several files for organizational purposes, but they are all available under the scipy_base namespace (and the scipy namespace). There are routines for type handling and type checking, shape and matrix manipulation, polynomial processing, and other useful functions. Rather than giving a detailed description of each of these functions (which is available using the **help**, **info** and **source** commands), this tutorial will discuss some of the more useful commands which require a little introduction to use to their full potential.

[1]These changes are all made in a new module (fastumath) that is part of the scipy_base package. The old functionality is still available in umath (part of Numeric) if you need it (note: importing umath or fastumath resets the behavior of the infix operators to use the umath or fastumath ufuncs respectively).

[2]Be careful when treating logical expressions as integers as the 8-bit integers may silently overflow at 256.

2.2.1 Type handling

Note the difference between **iscomplex** (**isreal**) and **iscomplexobj** (**isrealobj**). The former command is array based and returns byte arrays of ones and zeros providing the result of the element-wise test. The latter command is object based and returns a scalar describing the result of the test on the entire object.

Often it is required to get just the real and/or imaginary part of a complex number. While complex numbers and arrays have attributes that return those values, if one is not sure whether or not the object will be complex-valued, it is better to use the functional forms **real** and **imag**. These functions succeed for anything that can be turned into a Numeric array. Consider also the function **real_if_close** which transforms a complex-valued number with tiny imaginary part into a real number.

Occasionally the need to check whether or not a number is a scalar (Python (long)int, Python float, Python complex, or rank-0 array) occurs in coding. This functionality is provided in the convenient function **isscalar** which returns a 1 or a 0.

Finally, ensuring that objects are a certain Numeric type occurs often enough that it has been given a convenient interface in SciPy through the use of the **cast** dictionary. The dictionary is keyed by the type it is desired to cast to and the dictionary stores functions to perform the casting. Thus, `>>> a = cast['f'](d)` returns an array of float32 from d. This function is also useful as an easy way to get a scalar of a certain type: `>>> fpi = cast['f'](pi)`.

2.2.2 Index Tricks

Thre are some class instances that make special use of the slicing functionality to provide efficient means for array construction. This part will discuss the operation of **mgrid**, **r_**, and **c_** for quickly constructing arrays.

One familiar with Matlab may complain that it is difficult to construct arrays from the interactive session with Python. Suppose, for example that one wants to construct an array that begins with 3 followed by 5 zeros and then contains 10 numbers spanning the range -1 to 1 (inclusive on both ends). Before SciPy, I would need to enter something like the following `>>> concatenate(([3],[0]*5,arange(-1,1.002,2/9.0))`. With the **r_** command one can enter this as `>>> r_[3,[0]*5,-1:1:10j]` which can ease typing in an interactive session. Notice how objects are concatenated, and the slicing syntax is used (abused) to construct ranges. The other term that deserves a little explanation is the use of the complex number 10j as the step size in the slicing syntax. This non-standard use allows the number to be interpreted as the number of points to produce in the range rather than as a step size (note we would have used the long integer notation, 10L, but this notation may go away in Python as the integers became unified). This non-standard usage may be unsightly to some, but it gives the user the ability to quickly construct complicated vectors in a very readable fashion. When the number of points is specified in this way, the end-point is inclusive.

The "r" stands for row concatenation because if the objects between commas are 2 dimensional arrays, they are stacked by rows (and thus must have commensurate columns). There is an equivalent command **c_** that stacks 2d arrays by columns but works identically to **r_** for 1d arrays.

Another very useful class instance which makes use of extended slicing notation is the function **mgrid**. In the simplest case, this function can be used to construct 1d ranges as a convenient substitute for arange. It also allows the use of complex-numbers in the step-size to indicate the number of points to place between the (inclusive) end-points. The real purpose of this function however is to produce N, N-d arrays which provide coordinate arrays for an N-dimensional volume. The easiest way to understand this is with an example of its usage:

```
>>> mgrid[0:5,0:5]
array([[[0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1],
        [2, 2, 2, 2, 2],
        [3, 3, 3, 3, 3],
        [4, 4, 4, 4, 4]],
       [[0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4],
        [0, 1, 2, 3, 4]]])
>>> mgrid[0:5:4j,0:5:4j]
array([[[ 0.    ,  0.    ,  0.    ,  0.    ],
        [ 1.6667,  1.6667,  1.6667,  1.6667],
        [ 3.3333,  3.3333,  3.3333,  3.3333],
        [ 5.    ,  5.    ,  5.    ,  5.    ]],
       [[ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ],
        [ 0.    ,  1.6667,  3.3333,  5.    ]]])
```

Having meshed arrays like this is sometimes very useful. However, it is not always needed just to evaluate some N-dimensional function over a grid due to the array-broadcasting rules of Numeric and SciPy. If this is the only purpose for generating a meshgrid, you should instead use the function **ogrid** which generates an "open" grid using NewAxis judiciously to create N, N-d arrays where only one-dimension has length greater than 1. This will save memory and create the same result if the only purpose for the meshgrid is to generate sample points for evaluation of an N-d function.

2.2.3 Shape manipulation

In this category of functions are routines for squeezing out length-one dimensions from N-dimensional arrays, ensure that an array is at least 1-, 2-, or 3-dimensional, and stacking (concatenating) arrays by rows, columns, and "pages" (in the third dimension). Routines for splitting arrays (roughly the opposite of stacking arrays).

2.2.4 Matrix manipulations

These are functions specifically suited for 2-dimensional arrays that were part of MLab in the Numeric distribution, but have been placed in scipy_base for completeness.

2.2.5 Polynomials

There are two (interchangeable) ways to deal with 1-d polynomials in SciPy. The first is to use the **poly1d** class in **scipy_base**. This class accepts coefficients or polynomial roots to initialize a polynomial. The polynomial object can then be manipulated in algebraic expressions, integrated, differentiated, and evaluated. It even prints like a polynomial:

The other way to handle polynomials is as an array of coefficients with the first element of the array giving the coefficient of the highest power. There are explicit functions to add, subtract, multiply, divide, integrate, differentiate, and evaluate polynomials represented as sequences of coefficients.

2.2.6 Other useful functions

There are several other functions in the scipy_base package including most of the other functions that are also in MLab that comes with the Numeric package. The reason for duplicating these functions is to allow SciPy to potentially alter their original interface and make it easier for users to know how to get access to functions `>>> from scipy import *`.

New functions which should be mentioned are **mod(x,y)** which can replace **x%y** when it is desired that the result take the sign of **y** instead of **x**. Also included is **fix** which always rounds to the nearest integer towards zero. For doing phase processing, the functions **angle,** and **unwrap** are also useful. Also, the **linspace** and **logspace** functions return equally spaced samples in a linear or log scale. Finally, mention should be made of the new function **select** which extends the functionality of **where** to include multiple conditions and multiple choices. The calling convention is `select(condlist,choicelist,default=0)`. **Select** is a vectorized form of the multiple if-statement. It allows rapid construction of a function which returns an array of results based on a list of conditions. Each element of the return array is taken from the array in a `choicelist` corresponding to the first condition in `condlist` that is true. For example

2.3 Common functions

Some functions depend on sub-packages of SciPy but should be available from the top-level of SciPy due to their common use. These are functions that might have been placed in scipy_base except for their dependence on other sub-packages of SciPy. For example the **factorial** and **comb** functions compute $n!$ and $n!/k!(n-k)!$ using either exact integer arithmetic (thanks to Python's Long integer object), or by using floating-point precision and the gamma function. The functions **rand** and **randn** are used so often that they warranted a place at the top level. There are convenience functions for the interactive use: **disp** (similar to print), and **who** (returns a list of defined variables and memory consumption–upper bounded). Another function returns a common image used in signal processing: **lena.**

Finally, two functions are provided that are useful for approximating derivatives of functions using discrete-differences. The function **central_diff_weights** returns weighting coefficients for an equally-spaced N-point approximation to the derivative of order o. These weights must be multiplied by the function corresponding to these points and the results added to obtain the derivative approximation. This function is intended for use when only samples of the function are avaiable. When the function is an object that can be handed to a routine and evaluated, the function **derivative** can be used to automatically evaluate the object at the correct points to obtain an N-point approximation to the o^{th}-derivative at a given point.

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])
```

This particular function could have been written in vector form without the use of general_function. But, what if the function you have written is the result of some optimization or integration routine. Such functions can likely only be vectorized using `special.general_function`.

3.2 Special Functions

The main feature of the **special** package is the definition of numerous special functions of mathematical physics. Available functions include airy, elliptic, bessel, gamma, beta, hypergeometric, parabolic cylinder, mathieu, spheroidal wave, struve, and kelvin. There are also some low-level stats functions that are not intended for general use as an easier interface to these functions is provided by the `stats` module. Most of these functions behave can take array arguments and return array results following the same broadcasting rules as other math functions in Numerical Python. Many of these functions also accept complex-numbers as input. For a complete list of the available functions with a one-line description type `>>>info(special)`.

Each function also has it's own documentation accessible using help. If you don't see a function you need, consider writing it and contributing it to the library. You can write the function in either C, Fortran, or Python. Look in the source code of the library for examples of each of these kind of functions.

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 $\pm\infty$ (**±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

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

where

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

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 **±integrate.inf** (or **inf)** as one of the arguments. For example, suppose that a numerical value for the exponential integral:

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

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 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 **fixed_quad** which performs fixed-order Gaussian quadrature. The second function is **quadrature** 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 **special.orthogonal** which can calculate the roots and quadrature weights of a large variety of orthogonal polynomials.

4.3 Integrating using samples

There are three functions for computing integrals given only samples: **trapz**, **simps**, and **romb**. The first two functions use Newton-Coates formulas of order 1 and 2 respectively to perform integration. These two functions can handle, non-equally-spaced samples. The trapezoidal rule approximates the function as a straight line between adjacent points, while Simpson's rule approximates the function between three adjacent points as a parabola.

If the samples are equally-spaced and the number of samples available is $2^k + 1$ for some integer k, then Romberg integration can be used to obtain high-precision estimates of the integral using the available samples. Romberg integration uses the trapezoid rule at step-sizes related by a power of two and then performs Richardson extrapolation on these estimates to approximate the integral with a higher-degree of accuracy. (A different interface to Romberg integration useful when the function can be provided is also available as **integrate.romberg**).

4.4 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 $\left.\frac{dw}{dz}\right|_{z=0} = -\frac{1}{\sqrt[3]{3}\Gamma\left(\frac{1}{3}\right)}$. 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

$$\frac{d\mathbf{y}}{dt} = \left[\begin{array}{c} ty_1 \\ y_0 \end{array}\right] = \left[\begin{array}{cc} 0 & t \\ 1 & 0 \end{array}\right]\left[\begin{array}{c} y_0 \\ y_1 \end{array}\right] = \left[\begin{array}{cc} 0 & t \\ 1 & 0 \end{array}\right]\mathbf{y}.$$

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]
```

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):

```
>>> info(optimize)
 Optimization Tools

A collection of general-purpose optimization routines.
```

```
  fmin        --  Nelder-Mead Simplex algorithm
                    (uses only function calls)
  fmin_powell --  Powell's (modified) level set method (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.

 Scalar function minimizers

  fminbound   --  Bounded minimization of a scalar function.
  brent       --  1-D function minimization using Brent method.
  golden      --  1-D function minimization using Golden Section method
  bracket     --  Bracket a minimum (given two starting points)

Also a collection of general_purpose root-finding routines.

  fsolve      --  Non-linear multi-variable equation solver.

 Scalar function solvers

  brentq      --  quadratic interpolation Brent method
  brenth      --  Brent method (modified by Harris with
                    hyperbolic extrapolation)
  ridder      --  Ridder's method
  bisect      --  Bisection method
  newton      --  Secant method or Newton's method

  fixed_point -- Single-variable fixed-point 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:

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

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 scipy.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.]
```

Another optimization algorithm that needs only function calls to find the minimum is Powell's method available as **optimize.fmin_powell**.

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 scipy.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

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

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

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

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 scipy.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:

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

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

```
>>> from scipy.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.
```

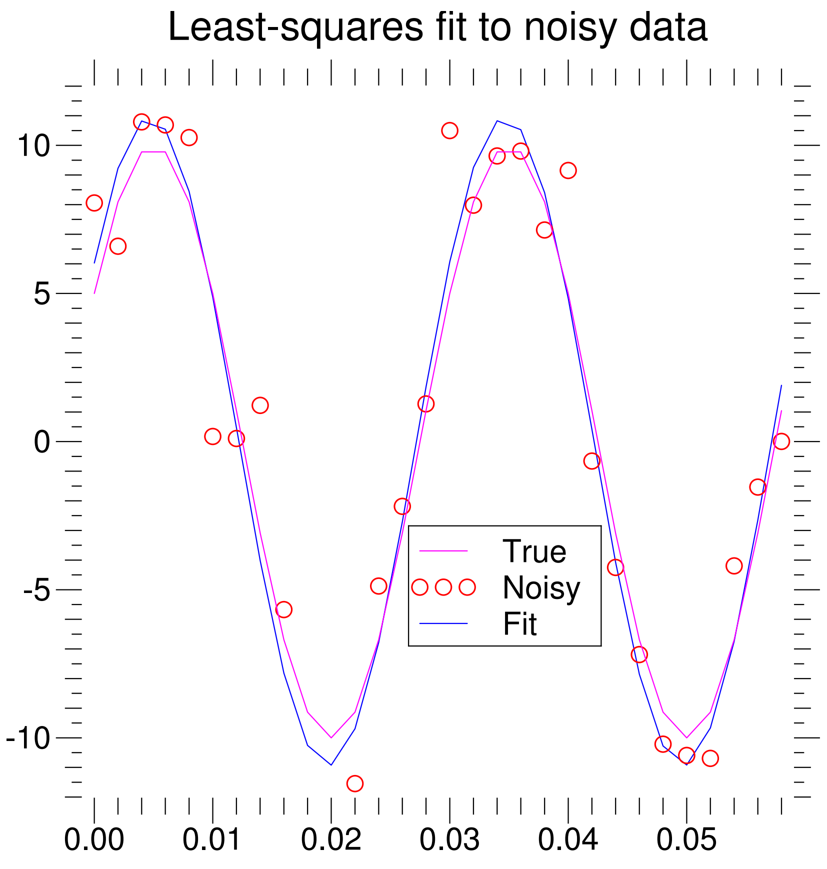


Figure 1: Least-square fitting to noisy data using **scipy.optimize.leastsq**

5.5 Scalar function minimizers

Often only the minimum of a scalar function is needed (a scalar function is one that takes a scalar as input and returns a scalar output). In these circumstances, other optimization techniques have been developed that can work faster.

5.5.1 Unconstrained minimization (optimize.brent)

There are actually two methods that can be used to minimize a scalar function (**brent** and **golden)**, but **golden** is included only for academic purposes and should rarely be used. The brent method uses Brent's algorithm for locating a minimum. Optimally a bracket should be given which contains the minimum desired. A bracket is a triple (a, b, c) such that $f(a) > f(b) < f(c)$ and $a < b < c$. If this is not given, then alternatively two starting points can be chosen and a bracket will be found from these points using a simple marching algorithm. If these two starting points are not provided 0 and 1 will be used (this may not be the right choice for your function and result in an unexpected minimum being returned).

5.5.2 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. 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 $x_{\text{min}} = 5.3314$:

```
>>> from scipy.special import j1

>>> from scipy.optimize import fminbound
>>> xmin = fminbound(j1, 4, 7)
>>> print xmin
5.33144184241
```

5.6 Root finding

5.6.1 Sets of equations

To find the roots of a polynomial, the command **roots** is useful. 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]
```

5.6.2 Scalar function root finding

If one has a single-variable equation, there are four different root finder algorithms that can be tried. Each of these root finding algorithms requires the endpoints of an interval where a root is suspected (because the function changes signs). In general **brentq** is the best choice, but the other methods may be useful in certain circumstances or for academic purposes.

5.6.3 Fixed-point solving

A problem closely related to finding the zeros of a function is the problem of finding a fixed-point of a function. A fixed point of a function is the point at which evaluation of the function returns the point: $g(x) = x$. Clearly the fixed point of g is the root of $f(x) = g(x) - x$. Equivalently, the root of f is the fixed_point of $g(x) = f(x) + x$. The routine **fixed_point** provides a simple iterative method using Aitkens sequence acceleration to estimate the fixed point of g given a starting point.

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:

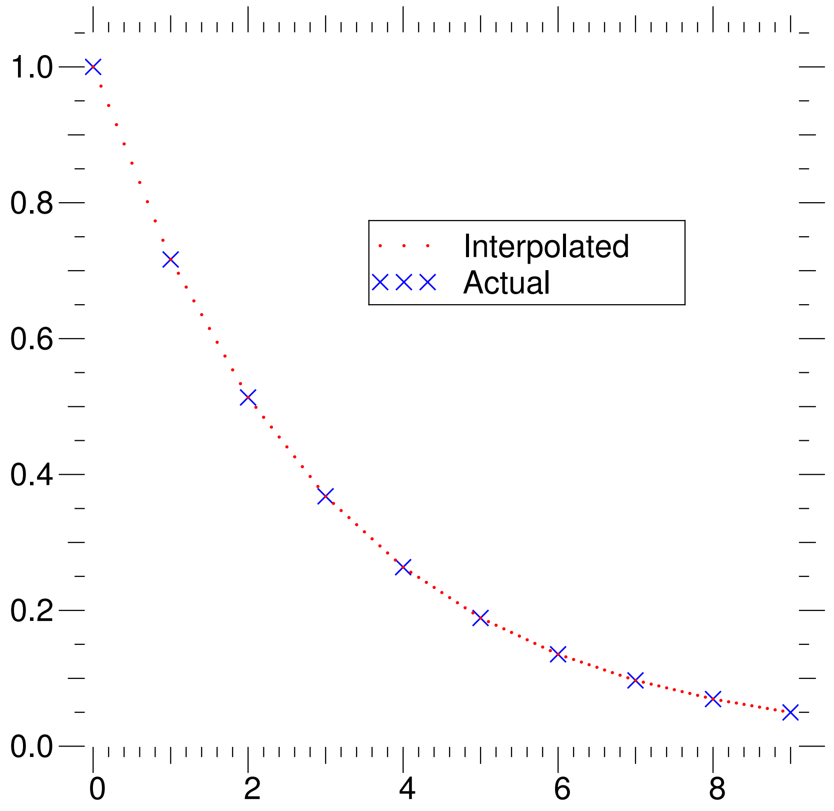


Figure 2: One-dimensional interpolation using the class **interpolate.linear_1d.**

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

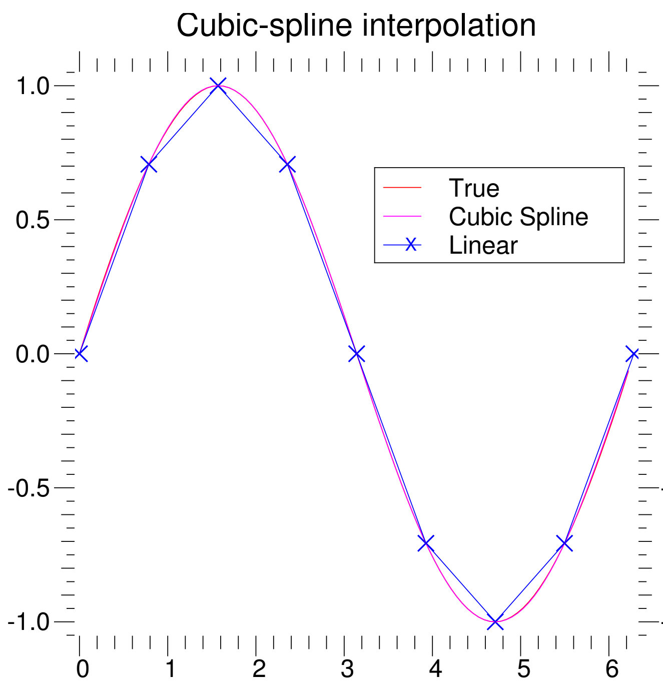
Spline interpolation requires two essential steps: (1) a spline representation of the curve is computed, and (2) the spline is evaluated at the desired points. In order to find the spline representation, there are two different was to represent a curve and obtain (smoothing) spline coefficients: directly 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.

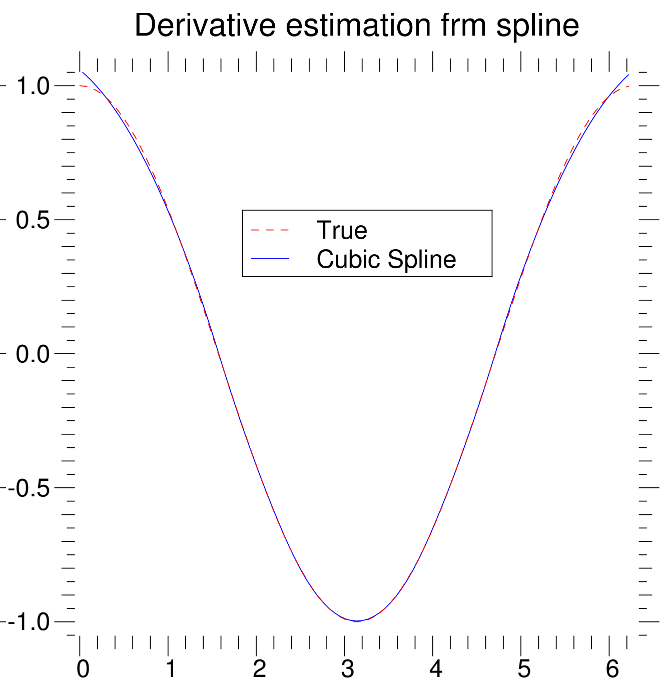
The keyword argument, s, is used to specify the amount of smoothing to perform during the spline fit. The default value of s is $s = m - \sqrt{2m}$ where m is the number of data-points being fit. Therefore, **if no smoothing is desired a value of $\mathbf{s} = 0$ should be passed to the routines.**

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).

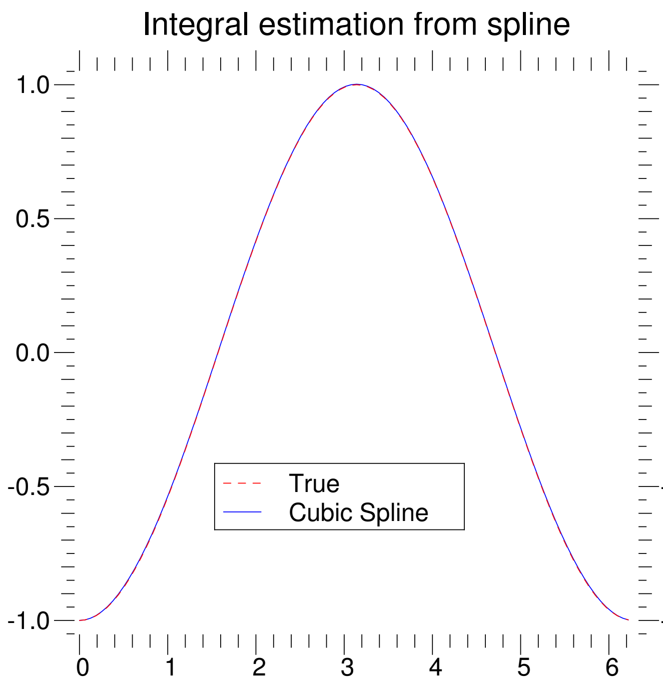
```
>>> # Cubic-spline
>>> x = arange(0,2*pi+pi/4,2*pi/8)
>>> y = sin(x)
>>> tck = interpolate.splrep(x,y,s=0)
>>> xnew = arange(0,2*pi,pi/50)
>>> ynew = interpolate.splev(xnew,tck,der=0)
>>> xplt.plot(x,y,'x',xnew,ynew,xnew,sin(xnew),x,y,'b')
>>> xplt.legend(['Linear','Cubic Spline', 'True'],['b-x','m','r'])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



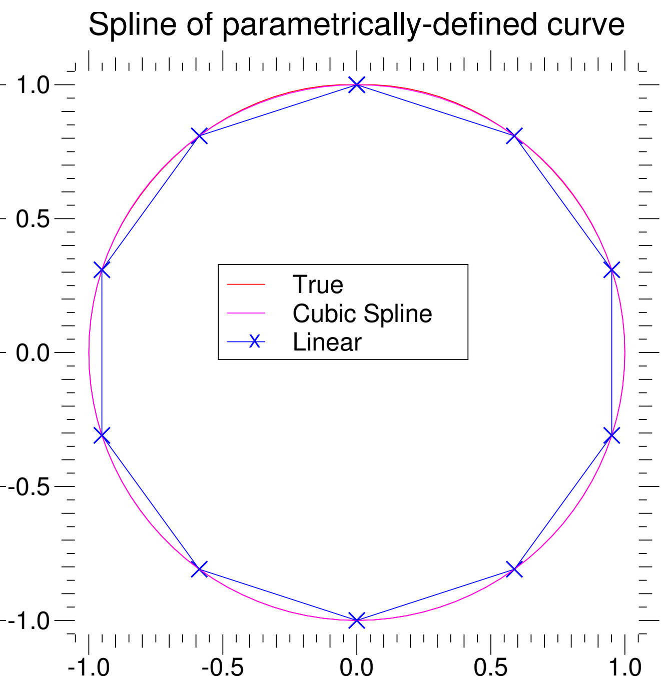
(a) Cubic-spline (splrep)



(b) Derivative of spline (splev)



(c) Integral of spline (splint)



(d) Spline of parametric curve (splprep)

Figure 3: Examples of using cubic-spline interpolation.

```
>>> 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")
```

7 Signal Processing (signal)

The signal processing toolbox currently contains some filtering functions, a limited set of filter design tools, and a few B-spline interpolation algorithms for one- and two-dimensional data. While the B-spline algorithms could technically be placed under the interpolation category, they are included here because they only work with equally-spaced data and make heavy use of filter-theory and transfer-function formalism to provide a fast B-spline transform. To understand this section you will need to understand that a signal in SciPy is an array of real or complex numbers.

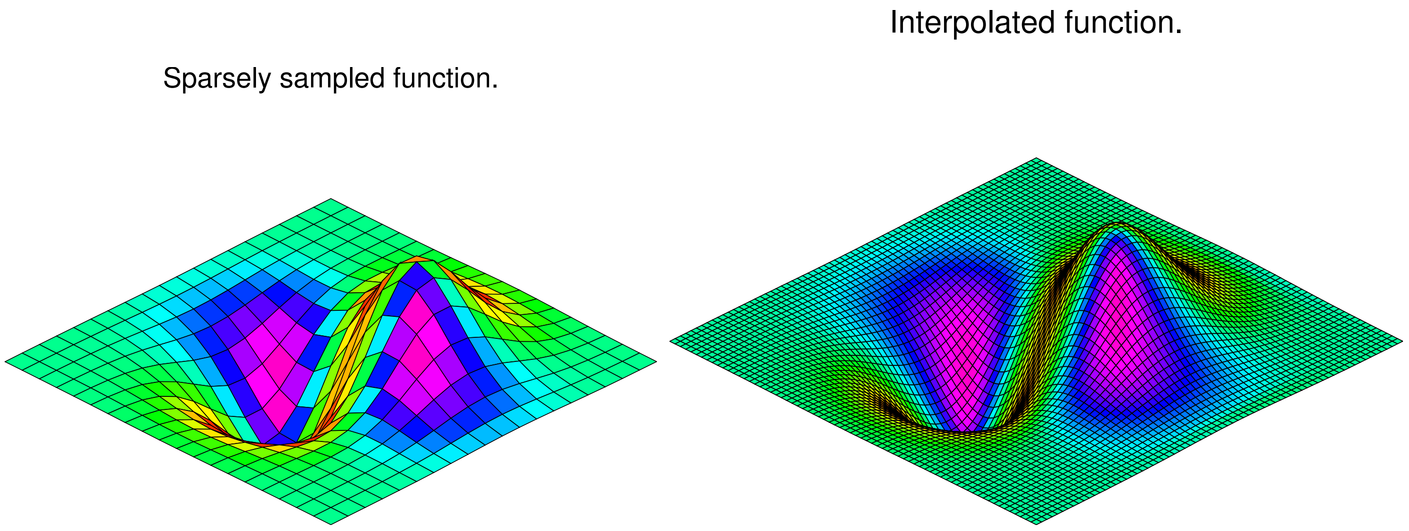


Figure 4: Example of two-dimensional spline interpolation.

7.1 B-splines

A B-spline is an approximation of a continuous function over a finite-domain in terms of B-spline coefficients and knot points. If the knot-points are equally spaced with spacing Δx, then the B-spline approximation to a 1-dimensional function is the finite-basis expansion.

$$y(x) \approx \sum_j c_j \beta^o \left(\frac{x}{\Delta x} - j \right).$$

In two dimensions with knot-spacing Δx and Δy, the function representation is

$$z(x,y) \approx \sum_j \sum_k c_{jk} \beta^o \left(\frac{x}{\Delta x} - j \right) \beta^o \left(\frac{y}{\Delta y} - k \right).$$

In these expressions, $\beta^o(\cdot)$ is the space-limited B-spline basis function of order, o. The requirement of equally-spaced knot-points and equally-spaced data points, allows the development of fast (inverse-filtering) algorithms for determining the coefficients, c_j, from sample-values, y_n. Unlike the general spline interpolation algorithms, these algorithms can quickly find the spline coefficients for large images.

The advantage of representing a set of samples via B-spline basis functions is that continuous-domain operators (derivatives, re-sampling, integral, etc.) which assume that the data samples are drawn from an underlying continuous function can be computed with relative ease from the spline coefficients. For example, the second-derivative of a spline is

$$y\prime''(x) = \frac{1}{\Delta x^2} \sum_j c_j \beta^{o\prime\prime} \left(\frac{x}{\Delta x} - j \right).$$

Using the property of B-splines that

$$\frac{d^2 \beta^o(w)}{dw^2} = \beta^{o-2}(w+1) - 2\beta^{o-2}(w) + \beta^{o-2}(w-1)$$

it can be seen that

$$y''(x) = \frac{1}{\Delta x^2} \sum_j c_j \left[\beta^{o-2} \left(\frac{x}{\Delta x} - j + 1 \right) - 2\beta^{o-2} \left(\frac{x}{\Delta x} - j \right) + \beta^{o-2} \left(\frac{x}{\Delta x} - j - 1 \right) \right].$$

If $o = 3$, then at the sample points,

$$\begin{aligned}\Delta x^2 \left. y'(x) \right|_{x=n\Delta x} &= \sum_j c_j \delta_{n-j+1} - 2c_j \delta_{n-j} + c_j \delta_{n-j-1}, \\ &= c_{n+1} - 2c_n + c_{n-1}.\end{aligned}$$

Thus, the second-derivative signal can be easily calculated from the spline fit. if desired, smoothing splines can be found to make the second-derivative less sensitive to random-errors.

The savvy reader will have already noticed that the data samples are related to the knot coefficients via a convolution operator, so that simple convolution with the sampled B-spline function recovers the original data from the spline coefficients. The output of convolutions can change depending on how boundaries are handled (this becomes increasingly more important as the number of dimensions in the data-set increases). The algorithms relating to B-splines in the signal-processing sub package assume mirror-symmetric boundary conditions. Thus, spline coefficients are computed based on that assumption, and data-samples can be recovered exactly from the spline coefficients by assuming them to be mirror-symmetric also.

Currently the package provides functions for determining seond- and third-order cubic spline coefficients from equally spaced samples in one- and two-dimensions (**signal.qspline1d, signal.qspline2d, signal.cspline1d, signal.cspline2d**). The package also supplies a function (**signal.bspline**) for evaluating the bspline basis function, $\beta^o(x)$ for arbitrary order and x. For large o, the B-spline basis function can be approximated well by a zero-mean Gaussian function with standard-deviation equal to $\sigma_o = (o+1)/12$:

$$\beta^o(x) \approx \frac{1}{\sqrt{2\pi\sigma_o^2}} \exp\left(-\frac{x^2}{2\sigma_o}\right).$$

A function to compute this Gaussian for arbitrary x and o is also available (**signal.gauss_spline**). The following code and Figure uses spline-filtering to compute an edge-image (the second-derivative of a smoothed spline) of Lena's face which is an array returned by the command **lena().** The command **signal.sepfir2d** was used to apply a separable two-dimensional FIR filter with mirror-symmetric boundary conditions to the spline coefficients. This function is ideally suited for reconstructing samples from spline coefficients and is faster than **signal.convolve2d** which convolves arbitrary two-dimensional filters and allows for choosing mirror-symmetric boundary conditions.

```
>>> image = lena().astype(Float32)
>>> derfilt = array([1.0,-2,1.0],Float32)
>>> ck = signal.cspline2d(image,8.0)
>>> deriv = signal.sepfir2d(ck, derfilt, [1]) + \
>>>         signal.sepfir2d(ck, [1], derfilt)
>>>
>>> ## Alternatively we could have done:
>>> ## laplacian = array([[0,1,0],[1,-4,1],[0,1,0]],Float32)
>>> ## deriv2 = signal.convolve2d(ck,laplacian,mode='same',boundary='symm')
>>>
>>> xplt.imagesc(image[::-1]) # flip image so it looks right-side up.
>>> xplt.title('Original image')
>>> xplt.eps('lena_image')
>>> xplt.imagesc(deriv[::-1])
>>> xplt.title('Output of spline edge filter')
>>> xplt.eps('lena_edge')
```

7.2 Filtering

Filtering is a generic name for any system that modifies an input signal in some way. In SciPy a signal can be thought of as a Numeric array. There are different kinds of filters for different kinds of operations. There are two broad kinds of filtering operations: linear and non-linear. Linear filters can always be reduced to

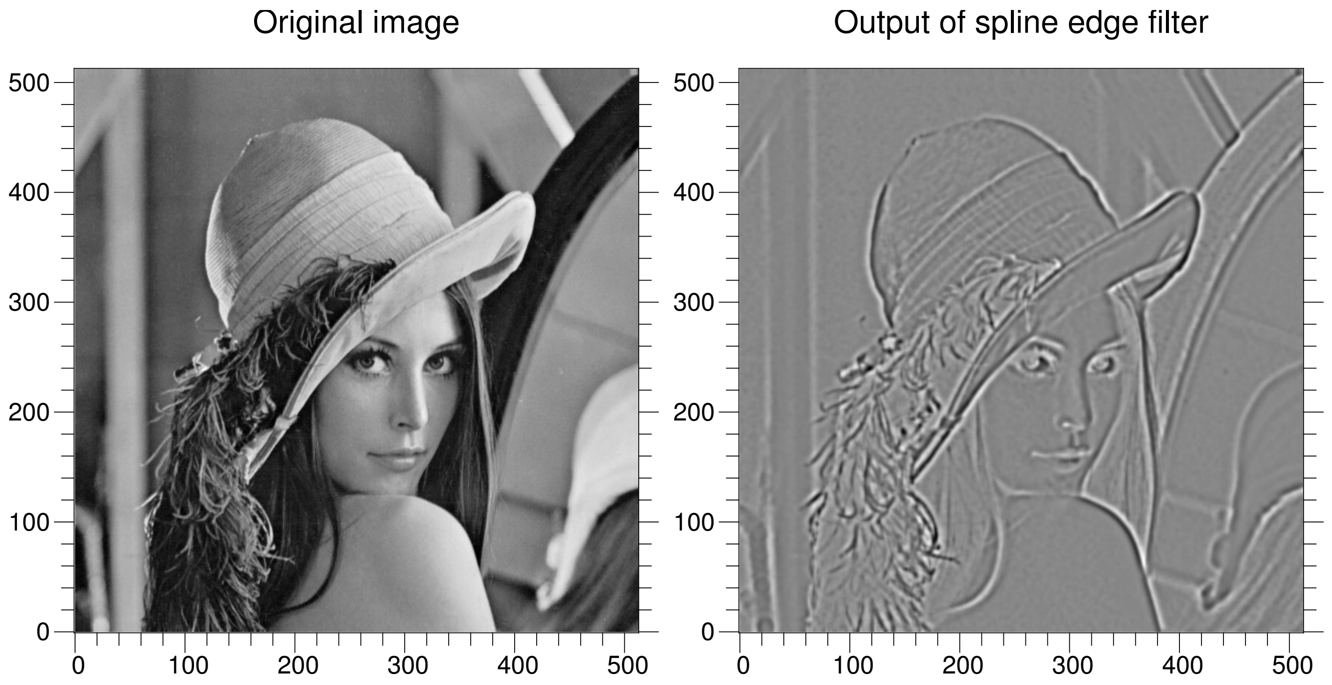


Figure 5: Example of using smoothing splines to filter images.

multiplication of the flattened Numeric array by an appropriate matrix resulting in another flattened Numeric array. Of course, this is not usually the best way to compute the filter as the matrices and vectors involved may be huge. For example filtering a 512×512 image with this method would require multiplication of a $512^2 x 512^2$matrix with a 512^2 vector. Just trying to store the $512^2 \times 512^2$ matrix using a standard Numeric array would require $68,719,476,736$ elements. At 4 bytes per element this would require 256GB of memory. In most applications most of the elements of this matrix are zero and a different method for computing the output of the filter is employed.

7.2.1 Convolution/Correlation

Many linear filters also have the property of shift-invariance. This means that the filtering operation is the same at different locations in the signal and it implies that the filtering matrix can be constructed from knowledge of one row (or column) of the matrix alone. In this case, the matrix multiplication can be accomplished using Fourier transforms.

Let $x[n]$ define a one-dimensional signal indexed by the integer n. Full, convolution of two one-dimensional signals can be expressed as

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]\, h[n-k]\,.$$

This equation can only be implemented directly, if we limit the sequences to finite support sequences that can be stored in a computer, choose $n = 0$ to be the starting point of both sequences, let $K+1$ be that value for which $y[n] = 0$ for all $n > K+1$ and $M+1$ be that value for which $x[n] = 0$ for all $n > M+1$, then the discrete convolution expression is

$$y[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} x[k]\, h[n-k]\,.$$

For convenience assume $K \geq M$. Then, the output of this operation is

$$y[0] = x[0]\, h[0]$$

$$\begin{aligned}
y[1] &= x[0]h[1] + x[1]h[0] \\
y[2] &= x[0]h[2] + x[1]h[1] + x[2]h[0] \\
\vdots & \quad \vdots \quad \vdots \\
y[M] &= x[0]h[M] + x[1]h[M-1] + \cdots + x[M]h[0] \\
y[M+1] &= x[1]h[M] + x[2]h[M-1] + \cdots + x[M+1]h[0] \\
\vdots & \quad \vdots \quad \vdots \\
y[K] &= x[K-M]h[M] + \cdots + x[K]h[0] \\
y[K+1] &= x[K+1-M]h[M] + \cdots + x[K]h[1] \\
\vdots & \quad \vdots \quad \vdots \\
y[K+M-1] &= x[K-1]h[M] + x[K]h[M-1] \\
y[K+M] &= x[K]h[M].
\end{aligned}$$

Thus, the full discrete convolution of two finite sequences of lengths $K+1$ and $M+1$ respectively results in a finite sequence of length $K+M+1 = (K+1)+(M+1)-1$.

One dimensional convolution is implemented in SciPy with the function `signal.convolve`. This function takes as inputs the signals x, h, and an optional flag and returns the signal y. The optional flag allows for specification of which part of the output signal to return. The default value of 'full' returns the entire signal. If the flag has a value of 'same' then only the middle K values are returned starting at $y\left[\left\lfloor \frac{M-1}{2} \right\rfloor\right]$ so that the output has the same length as the largest input. If the flag has a value of 'valid' then only the middle $K-M+1 = (K+1)-(M+1)+1$ output values are returned where z depends on all of the values of the smallest input from $h[0]$ to $h[M]$. In other words only the values $y[M]$ to $y[K]$ inclusive are returned.

This same function `signal.convolve` can actually take N-dimensional arrays as inputs and will return the N-dimensional convolution of the two arrays. The same input flags are available for that case as well.

Correlation is very similar to convolution except for the minus sign becomes a plus sign. Thus

$$w[n] = \sum_{k=-\infty}^{\infty} y[k]\, x[n+k]$$

is the (cross) correlation of the signals y and x. For finite-length signals with $y[n] = 0$ outside of the range $[0, K]$ and $x[n] = 0$ outside of the range $[0, M]$, the summation can simplify to

$$w[n] = \sum_{k=\max(0,-n)}^{\min(K,M-n)} y[k]\, x[n+k].$$

Assuming again that $K \geq M$ this is

$$\begin{aligned}
w[-K] &= y[K]x[0] \\
w[-K+1] &= y[K-1]x[0] + y[K]x[1] \\
\vdots & \quad \vdots \quad \vdots \\
w[M-K] &= y[K-M]x[0] + y[K-M+1]x[1] + \cdots + y[K]x[M] \\
w[M-K+1] &= y[K-M-1]x[0] + \cdots + y[K-1]x[M] \\
\vdots & \quad \vdots \quad \vdots \\
w[-1] &= y[1]x[0] + y[2]x[1] + \cdots + y[M+1]x[M] \\
w[0] &= y[0]x[0] + y[1]x[1] + \cdots + y[M]x[M] \\
w[1] &= y[0]x[1] + y[1]x[2] + \cdots + y[M-1]x[M] \\
w[2] &= y[0]x[2] + y[1]x[3] + \cdots + y[M-2]x[M]
\end{aligned}$$

$$\begin{array}{rcl} \vdots & \vdots & \vdots \\ w[M-1] & = & y[0]\,x[M-1]+y[1]\,x[M] \\ w[M] & = & y[0]\,x[M]\,. \end{array}$$

The SciPy function `signal.correlate` implements this operation. Equivalent flags are available for this operation to return the full $K+M+1$ length sequence ('full') or a sequence with the same size as the largest sequence starting at $w\left[-K+\left\lfloor\frac{M-1}{2}\right\rfloor\right]$ ('same') or a sequence where the values depend on all the values of the smallest sequence ('valid'). This final option returns the $K-M+1$ values $w[M-K]$ to $w[0]$ inclusive.

The function `signal.correlate` can also take arbitrary N-dimensional arrays as input and return the N-dimensional convolution of the two arrays on output.

When $N=2$, `signal.correlate` and/or `signal.convolve` can be used to construct arbitrary image filters to perform actions such as blurring, enhancing, and edge-detection for an image.

Convolution is mainly used for filtering when one of the signals is much smaller than the other ($K \gg M$), otherwise linear filtering is more easily accomplished in the frequency domain (see Fourier Transforms).

7.2.2 Difference-equation filtering

A general class of linear one-dimensional filters (that includes convolution filters) are filters described by the difference equation

$$\sum_{k=0}^{N} a_k y[n-k] = \sum_{k=0}^{M} b_k x[n-k]$$

where $x[n]$ is the input sequence and $y[n]$ is the output sequence. If we assume initial rest so that $y[n]=0$ for $n<0$, then this kind of filter can be implemented using convolution. However, the convolution filter sequence $h[n]$ could be infinite if $a_k \neq 0$ for $k \geq 1$. In addition, this general class of linear filter allows initial conditions to be placed on $y[n]$ for $n<0$ resulting in a filter that cannot be expressed using convolution.

The difference equation filter can be thought of as finding $y[n]$ recursively in terms of it's previous values

$$a_0 y[n] = -a_1 y[n-1] - \cdots - a_N y[n-N] + \cdots + b_0 x[n] + \cdots + b_M x[n-M]\,.$$

Often $a_0=1$ is chosen for normalization. The implementation in SciPy of this general difference equation filter is a little more complicated then would be implied by the previous equation. It is implemented so that only one signal needs to be delayed. The actual implementation equations are (assuming $a_0=1$).

$$\begin{array}{rcl} y[n] & = & b_0 x[n] + z_0[n-1] \\ z_0[n] & = & b_1 x[n] + z_1[n-1] - a_1 y[n] \\ z_1[n] & = & b_2 x[n] + z_2[n-1] - a_2 y[n] \\ \vdots & \vdots & \vdots \\ z_{K-2}[n] & = & b_{K-1} x[n] + z_{K-1}[n-1] - a_{K-1} y[n] \\ z_{K-1}[n] & = & b_K x[n] - a_K y[n]\,, \end{array}$$

where $K=\max(N,M)$. Note that $b_K=0$ if $K>M$ and $a_K=0$ if $K>N$. In this way, the output at time n depends only on the input at time n and the value of z_0 at the previous time. This can always be calculated as long as the K values $z_0[n-1]\ldots z_{K-1}[n-1]$ are computed and stored at each time step.

The difference-equation filter is called using the command `signal.lfilter` in SciPy. This command takes as inputs the vector b, the vector, a, a signal x and returns the vector y (the same length as x) computed using the equation given above. If x is N-dimensional, then the filter is computed along the axis provided. If, desired, initial conditions providing the values of $z_0[-1]$ to $z_{K-1}[-1]$ can be provided or else it will be assumed that they are all zero. If initial conditions are provided, then the final conditions on the intermediate variables are also returned. These could be used, for example, to restart the calculation in the same state.

Sometimes it is more convenient to express the initial conditions in terms of the signals $x[n]$ and $y[n]$. In other words, perhaps you have the values of $x[-M]$ to $x[-1]$ and the values of $y[-N]$ to $y[-1]$ and would like to determine what values of $z_m[-1]$ should be delivered as initial conditions to the difference-equation filter. It is not difficult to show that for $0 \leq m < K$,

$$z_m[n] = \sum_{p=0}^{K-m-1} \left(b_{m+p+1} x[n-p] - a_{m+p+1} y[n-p] \right).$$

Using this formula we can find the intial condition vector $z_0[-1]$ to $z_{K-1}[-1]$ given initial conditions on y (and x). The command `signal.lfiltic` performs this function.

7.2.3 Other filters

Median Filter

Order Filter

Wiener filter

Hilber filter

Detrend

7.3 Filter design

7.3.1 Finite-impulse response design

7.3.2 Inifinite-impulse response design

7.3.3 Analog filter frequency response

7.3.4 Digital filter frequency response

7.4 Linear Time-Invariant Systems

7.4.1 LTI Object

7.4.2 Continuous-Time Simulation

7.4.3 Step response

7.4.4 Impulse response

8 Input/Output

8.1 Binary

8.1.1 Arbitrary binary input and output (fopen)

8.1.2 Read and write Matlab .mat files

8.2 Text-file

8.2.1 Read text-files (read_array)

8.2.2 Write a text-file (write_array)

9 Fourier Transforms

9.1 One-dimensional

9.2 Two-dimensional

9.3 N-dimensional

9.4 Shifting

9.5 Sample frequencies

9.6 Hilbert transform

9.7 Tilbert transform

10 Linear Algebra

When SciPy is built using the optimized ATLAS LAPACK and BLAS libraries, it has very fast linear algebra capabilities. If you dig deep enough, all of the raw lapack and blas libraries are available for your use for even more speed. In this section, some easier-to-use interfaces to these routines are described.

All of these linear algebra routines expect an object that can be converted into a 2-dimensional array. The output of these routines is also a two-dimensional array. There is a matrix class defined in Numeric that scipy inherits and extends. You can initialize this class with an appropriate Numeric array in order to get objects for which multiplication is matrix-multiplication instead of the default, element-by-element multiplication.

10.1 Matrix Class

The matrix class is initialized with the SciPy command `mat` which is just convenient short-hand for `Matrix.Matrix`. If you are going to be doing a lot of matrix-math, it is convenient to convert arrays into matrices using this command.

10.2 Basic routines

10.2.1 Finding Inverse

The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{B}$ such that $\mathbf{AB} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix consisting of ones down the main diagonal. Usually $\mathbf{B}$ is denoted $\mathbf{B} = \mathbf{A}^{-1}$. In SciPy, the matrix inverse of the Numeric array, `a`, is obtained using `linalg.inv(a)`, or using `a.I` if `a` is a Matrix.

10.2.2 Solving linear system

10.2.3 Finding Determinant

10.2.4 Computing norms

10.2.5 Solving least-squares problems

10.2.6 Pseudo-inverses

10.3 Decompositions

10.3.1 Eigenvalues and eigenvectors

10.3.2 Singular value decomposition

10.3.3 LU decomposition

10.3.4 Cholesky decomposition

10.3.5 QR decomposition

10.3.6 Schur decomposition

10.4 Matrix Functions

10.4.1 Exponential function

10.4.2 Trigonemetric function

10.4.3 Hyperbolic trigonometric functions

10.4.4 Arbitrary function

11 Statistics

SciPy has a tremendous number of basic statistics routines with more easily added by the end user (if you create one please contribute it). All of the statistics functions are located in the sub-package **stats** and a fairly complete listing of these functions can be had using `info(stats)`.

12 Interfacing with Python Imaging Library

If you have the Python Imaging Library installed, SciPy provides some convenient functions that make use of it's facilities particularly for reading, writing, displaying, and rotating images. In SciPy an image is always a two- or three-dimensional array. Gray-scale, and colormap images are always two-dimensional arrays while RGB images are three-dimensional with the third dimension specifying the channel.

Commands available include

- fromimage — convert a PIL image to a Numeric array
- toimage — convert Numeric array to PIL image
- imsave — save Numeric array to an image file
- imread — read an image file into a Numeric array
- imrotate — rotate an image (a Numeric array) counter-clockwise
- imresize — resize an image using the PIL
- imshow — external viewer of a Numeric array (using PIL)
- imfilter — fast, simple built-in filters provided by PIL
- radon — simple radon transform based on imrotate

13 Plotting with xplt

13.1 Basic commands

13.2 Adding a legend

13.3 Drawing a histogram

13.4 Drawing a barplot

13.5 Plotting color arrays

13.6 Contour plots

13.7 Three-dimensional plots

13.8 Placing text and arrows

13.9 Special plots