2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

4
SciPy for Numerical Analysis

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

All the different areas of numerical analysis are contemplated in some SciPy module. For example, in order to compute values of special functions we use the `scipy.special` module. The `scipy.interpolate` module takes care of interpolation, extrapolation, and regression. For optimization, we have the `scipy.optimize` module, and finally, for numerical evaluation of integrals, we have the `scipy.integrate` module. This last module serves as the interface to perform numerical solutions of ordinary differential equations as well.

Evaluation of special functions

The `scipy.special` module contains numerically stable definitions of useful functions. We would like to point out that often the straightforward evaluation of a function at a single value is not very efficient. For instance, we would rather use a Horner scheme to find the value of a polynomial at a point, instead of the raw formula. NumPy and SciPy modules ensure that this optimization is always guaranteed with the definition of all its functions, whether by means of Horner schemes or with more advanced techniques.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Convenience and test functions

All the convenience functions are designed to facilitate a computational environment where the user does not need to worry about relative errors. The functions seem to be pointless at first sight, but behind their codes, there are state-of-the-art ideas that offer faster and more reliable results.

We have convenience functions beyond the ones defined in the NumPy libraries to deal with trigonometric functions in degrees (`cosdg`, `sindg`, `tandg`, `cotdg`); to compute angles in radians from their expressions in degrees, minutes and seconds (`radian`); common powers (`exp2` for *2**x*, and `exp10` for *10**x*); and common functions for small values of the variable (`log1p` for *log(1 + x)*, `expm1` for *exp(x)-1*, and `cosm1` for *cos(x)-1*).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For instance, in the following code snippet, the `log1p` function computes the natural logarithm of $1 + x$. Why not simply add 1 to the value of x, and then take the logarithm instead? Let us compare:

```
>>> a=scipy.special.exp10(-16)
>>> numpy.log(1+a)
0.0
>>> scipy.special.log1p(a)
9.9999999999999998e-17
```

While the absolute error of the first computation is small, the relative error is 100 percent.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

In the same way as Lena image is regarded as the performance test in image processing, we have a few functions that are used to test different algorithms in different scenarios. For instance, it is customary to test minimization codes against the Rosenbrock's banana function:

$$\boxed{f(x,y) = (1 - x^2) + 100(y - x^2)^2}$$

The corresponding optimization module, `scipy.optimize` has a routine to accurately evaluate this function (`rosen`), its derivative (`rosen_der`), its Hessian matrix (`rosen_hess`), or the product of the latter with a vector (`rosen_hess_prod`).

Univariate polynomials

Polynomials are defined in SciPy as a NumPy class, `poly1d`. This class has a handful of methods associated to compute the coefficients of the polynomial (`coeffs` or simply `c`), to compute the roots of the polynomial (`r`), to compute its derivative (`deriv`), to compute the symbolic integral (`integ`), to obtain the degree (`order` or simply `o`), and a method (`variable`) that provides with a string holding the name of the variable used in the proper definition.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

In order to define a polynomial, we must indicate either its coefficients or its roots:

```
>>> P1=numpy.poly1d([1,0,1])          # using coefficients
>>> print P1
   2
1 x + 1
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 65
http://site.ebrary.com/id/10672339?ppg=65
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
>>> print P1.r; print P1.o; P1.deriv() # roots,order,derivative
[ 0.+1.j  0.-1.j]
2
poly1d([2, 0])
>>> P2=numpy.poly1d([1,1,1], True)     # using roots
>>> print P2
   3     2
1 x - 3 x + 3 x - 1
```

We may evaluate polynomials by treating them either as (vectorized) functions, or with the `__call__` method:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
>>> P1( numpy.arange(10) )          # evaluate at 0,1,...,9
array([ 1,  2,  5, 10, 17, 26, 37, 50, 65, 82])
>>> P1.__call__(numpy.arange(10))    # same evaluation
array([ 1,  2,  5, 10, 17, 26, 37, 50, 65, 82])
```

There are also a handful of routines associated to polynomials – `roots` (to compute zeros), `polyder` (to compute derivatives), `polyint` (to compute integrals), `polyadd` (to add polynomials), `polysub` (to subtract polynomials), `polymul` (to multiply polynomials), `polydiv` (to perform polynomial division), `polyval` (to evaluate polynomials), and `polyfit` (to compute the best fit polynomial of certain order for two given arrays of data).

The usual binary operators `+`, `-`, `*`, and `/` perform the corresponding operations with polynomials. In addition, once a polynomial is created, any list of values that interacts with them is immediately casted to a polynomial. Therefore, the following four commands are equivalent:

- `numpy.polyadd(P1, numpy.poly1d([2,1]))`
- `numpy.polyadd(P1, [2,1])`
- `P1 + numpy.poly1d([2,1])`
- `P1 + [2,1]`

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Note how the polynomial division offers both quotient and reminder. For example:

```
>>> P1/[2,1]
(poly1d([ 0.5 , -0.25]), poly1d([ 1.25]))
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 66
http://site.ebrary.com/id/10672339?ppg=66
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This reads as follows:

$$\boxed{\frac{x^2+1}{2x+1} = \underbrace{\left(\tfrac{1}{2}x - \tfrac{1}{4}\right)}_{\text{quotient}} + \frac{\overbrace{5/4}^{\text{reminder}}}{2x+1}}$$

A family of polynomials is said to be orthogonal with respect to an inner product if for any two polynomials in the family, their inner product is zero. Sequences of these functions are used as the backbone of extremely fast algorithms of quadrature (for numerical integration of general functions). The `scipy.special` module contains both `poly1d` definitions, and fast evaluation of the families of orthogonal polynomials, such as Legendre (`legendre`), all Chebyshev polynomials (`chebyt`, `chebyu`, `chebyc`, `chebys`), Jacobi (`jacobi`), Laguerre and its generalized version (`laguerre` and `genlaguerre`), Hermite and its normalized version (`hermite` and `hermitenorm`), and Gegenbauer (`gegenbauer`). There are also shifted versions of some of them (`sh_legendre`, `sh_chebyt`, and so on).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The usual evaluation of polynomials can be improved for orthogonal polynomials; thanks to their rich mathematical structure. In these cases, we never evaluate them with the generic call methods presented previously. Instead, we employ the `eval_` syntax. For example, for Jacobi polynomials, we use the following:

```
eval_jacobi(n, alpha, beta, x)
```

In order to obtain the graph of the Jacobi polynomial of order `n = 3`, for `alpha = 0`, `beta = 1`, for a thousand values of `x` uniformly spaced from `-1` to `1`, we could issue the following command (output not shown):

```
>>> x=numpy.linspace(-1,1,1000)
>>> matplotlib.pyplot.plot(x,eval_jacobi(3,0,1,x))
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The gamma function

The gamma function is a logarithmic, convex, smooth function operating on complex numbers, which interpolates the factorial function for all nonnegative integers.
It is not defined at zero or any negative integer. This is the most common special function, and is widely used in many different applications, either by itself or as the main ingredient in the definition of many other functions. Concrete applications of the gamma function spread to such diverse fields as quantum physics, astrophysics, statistics, or fluid dynamics.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 67
http://site.ebrary.com/id/10672339?ppg=67
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The gamma function is defined by the improper integral, shown as follows:

$$\boxed{\Gamma(z) = \int_0^\infty e^{-t} t^{z-1}\,dt}$$

Evaluation of gamma at integer values gives shifted factorials, and actually, that is precisely how the factorials are coded in SciPy.

The `scipy.special` module has algorithms to obtain fast evaluation of the gamma function at any other permissible values. It also contains routines to perform evaluation of the most common compositions of the gamma functions appearing in the literature – `gammaln` for the natural logarithm of the absolute value of gamma, `rgamma` for the value one over gamma, `beta` for quotients, and `betaln` for the natural logarithm of the latter. We also have implementations of the logarithm of its derivative (`psi`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

An obvious application of gamma functions is the ability to access computations that are virtually impossible for a computer if approached in a direct way. For instance, in statistical applications we often work with ratios of factorials. If these factorials are too large for the precision of the computer, we resort to expressions involving their logarithms instead. But still, computing *ln(a! / b!)* may prove an impossible task (try, for example with *a = 10**15* and *b = a-10**10*). An elegant solution uses the digamma function `psi` by an application of the mean value theorem on the *ln(gamma(x))* function and proper estimation, we obtain the excellent approximation (for this case of choice of *a* and *b*).

$$\boxed{\ln(a!/b!) \approx 10^{10}\psi(a)}$$

The following is the code:

```
>>> 10^10*scipy.special.psi(10**15)
345387763949.10681
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The Riemann zeta function

Of huge impact in analytic number theory, and with applications to physics and probability theory, we have the Riemann zeta function, which computes p-series for any complex value p:

$$\boxed{\zeta(p) = \sum_{n=1}^\infty \frac{1}{n^p}}$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 68
http://site.ebrary.com/id/10672339?ppg=68
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The definition coded in SciPy allows a more flexible generalization of this function, as follows:

$$\boxed{\texttt{zeta(a,p)} = \sum_{n=0}^{\infty} \frac{1}{(n+a)^p}}$$

Airy (and Bairy) functions

These are the solutions to the Stokes equation, as shown in the following diagram:

$$\boxed{y'' = xy}$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This equation has two linearly independent solutions, both of them defined as an improper integral for real values of the independent variable. The `airy` command computes both functions (`Ai` and `Bi`) as well as their corresponding derivatives (`Aip`, `Bip`). In the following code, we take advantage of the `contourf` command in `matplotlib.pyplot`, to present an image of the real part of the output of the Bairy function `Bi`, for an array of 801 x 801 complex values uniformly spaced in the square from `-4-4j` to `4+4j`. We also offer this graph as a surface plot using the `mplot3d` module of `mpl_toolkits`:

```
import mpl_toolkits.mplot3d
x=numpy.mgrid[-4:4:100j,-4:4:100j]
z=x[0]+1j*x[1]
(Ai, Aip, Bi, Bip) = scipy.special.airy(z)
steps = range(int(Bi.real.min()), int(Bi.real.max()),6)
fig=matplotlib.pyplot.figure()
subplot1=fig.add_subplot(121,aspect='equal')
subplot1.contourf(x[0], x[1], Bi.real, steps)
subplot2=fig.add_subplot(122,projection='3d')
subplot2.plot_surface(x[0],x[1],Bi.real)
```

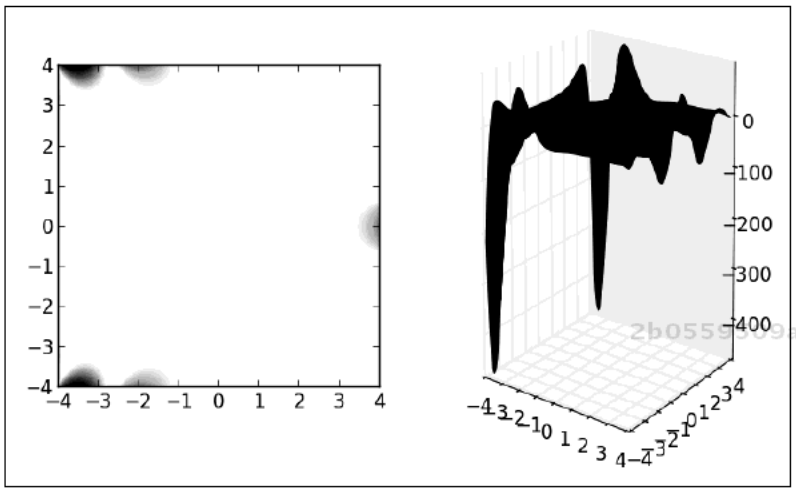
2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 69
http://site.ebrary.com/id/10672339?ppg=69
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The output is as follows:



Bessel and Struve functions

Bessel functions are both of the canonical solutions to Bessel's homogeneous differential equations.

$$\boxed{x^2 y'' + xy' + (x^2 - \alpha^2)y = 0.}$$

These equations arise naturally in the solution of Laplace's equation in cylindrical coordinates. The solutions of the non-homogeneous Bessel differential equation shown in the following diagram are called **Struve functions**:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

$$\boxed{x^2 y'' + xy' + (x^2 - \alpha^2)y = \frac{4\left(x/2\right)^{\alpha+1}}{\sqrt{\pi}\left(\alpha + \frac{1}{2}\right)}.}$$

In either case, the order of the equation is the complex number `alpha`, and acts as a parameter. Depending on the canonical solution and the order, the Bessel and Struve functions are addressed (and computed) differently.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 70
http://site.ebrary.com/id/10672339?ppg=70
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For Bessel functions, we have algorithms to produce the first kind (`jv`), the second kind (`yn`, `yv`), Hankel functions of the first and second kind (`hankel1`, `hankel2`), and the modified Bessel functions of the first and second kind (`iv`, `kn`, `kv`). Their syntax is similar in all cases – first parameter is the order, and second parameter the independent variable. `n` in the definition indicates that an integer is to be used as the order (since they are optimally coded for that situation).

```
>>>scipy.special.jn(5,numpy.pi)
0.71044976796351567
```

The module also contains fast versions of the most common Bessel functions (those of orders 0 and 1) – `j0(x)`, `j1(x)` — first kind — `y0(x)`, `y1(x)` — second kind, and so on. There are definitions of the spherical Bessel functions such as `sph_jn(n,z)`, `sph_yn(z)`; the Riccati-Bessel functions such as `riccati_jn(n,x)` and `riccati_yn(n,x)`; and derivatives of all the basic ones such as `jvp`, `yvp`, `kvp`, `ivp`, `h1vp`, and `h2vp`.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For Struve functions, we have fast algorithms to compute solutions of the differential equation of order `v` – (`struve(v,x)`, `modstruve(v,x)`).

Other special functions

There are more special functions included in this module, of great use in many applications to both pure and applied mathematics. An exhaustive list would be too large for the scope of this chapter, and we encourage exploring the different utilities for each set of special functions. Among the most interesting ones we have elliptic functions, Gauss' hypergeometric functions, parabolic cylinder functions, Mathieu functions, spheroidal wave functions, and Kelvin functions.

Interpolation and regression

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Interpolation is a basic method in numerical computation that is obtained from a discrete set of data points, some higher order structure that contains the previous data. The best known example is the interpolation of a sequence of points (*x_k*, *y_k*) in a plane to obtain a curve that goes through all the points in the order dictated by the sequence. If the points in the previous sequence are in the right position and order, it is possible to find a univariate function, *y = f(x)* for which *y_k = f(x_k)*. It is often reasonable to request this interpolating function to be a polynomial, or a rational function, or a more complex functional object. Interpolation is also possible in higher dimensions, of course. The objective of the `scipy.interpolate` module is precisely to offer a complete set of optimally coded applications to address this problem in different settings.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

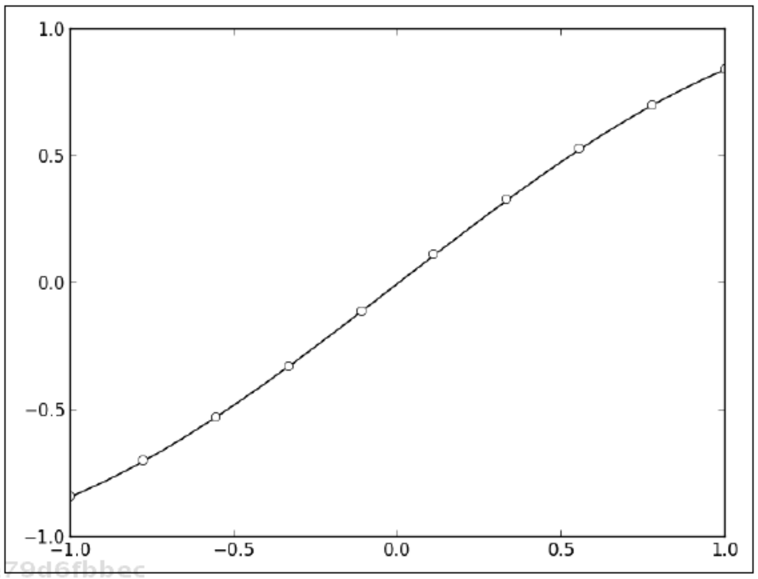
2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Let us address the most naïve way of interpolating data to obtain a polynomial, Lagrange interpolation. Given a sequence of different `x` values of size `n`, and a sequence of arbitrary real values `y`, of the same size `n`, we seek a polynomial `p(x)` of the degree of `n-1` at the most that satisfies the `n` constraints `p(x[k])=y[k]` for all `k` from 0 to `n-1`. The following code illustrates how to obtain a polynomial of degree 9 that interpolates the 10 uniformly spaced values of sine in the interval [-1,1]:

```
import scipy.interpolate
x=numpy.linspace(-1,1,10); xn=numpy.linspace(-1,1,1000)
y=numpy.sin(x)
polynomial=scipy.interpolate.lagrange(x, numpy.sin(x))
matplotlib.pyplot.plot(xn,polynomial(xn),x,y,'or')
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

We will obtain the following plot showing Lagrange interpolation:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The issues with Lagrange interpolation are numerous. The first obvious drawback arises since the user cannot specify the degree of the interpolation; this depends solely on the data. The procedure is also highly unstable numerically, especially for datasets with sizes over 20 points. This issue can be addressed by allowing the algorithm to depend on different properties of the dataset, rather than just the size and location of the points.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 72
http://site.ebrary.com/id/10672339?ppg=72
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Another inconvenience occurs if we need to update the dataset by adding a few more instances; the procedure needs to be repeated again from the beginning. This proves impractical if the datasets are increasing in size, and the updating is frequent. To address this issue, `BarycentricInterpolator` has the `add_xi` and `set_yi` methods. For example, in the next session we start by interpolating 10 uniformly spaced values of the sine function between 1 and 10. Once done, we update the interpolating polynomial with 10 more uniformly spaced values between 1.5 and 10.5:

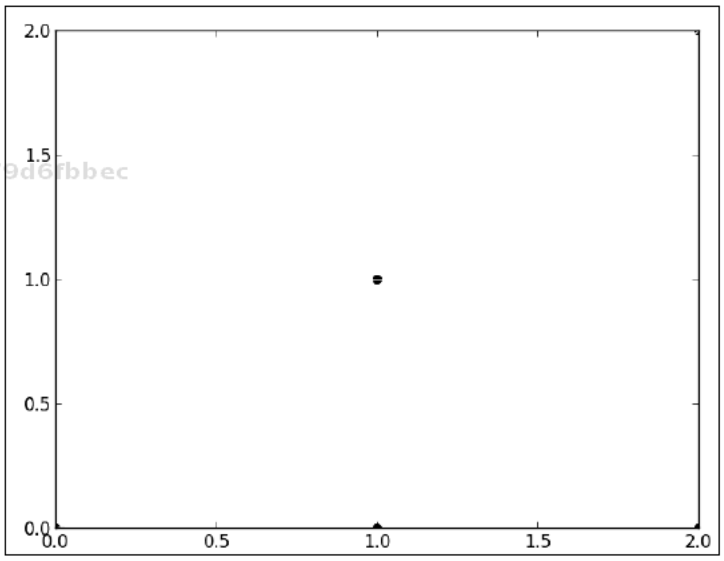
```
>>> x=numpy.linspace(1,10,10); y=numpy.sin(x)
>>> Polynomial=scipy.interpolate.BarycentricInterpolator(x,y)
>>> x=numpy.linspace(1.5,10.5,10); y=numpy.sin(x)
>>> Polynomial.add_xi(x,y)
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

It is also possible to interpolate data not only by point location, but also with derivatives at those locations. The `KrogInterpolator` command allows it, by including repeated `x` values, and indicating on the corresponding `y` values, the location and successive derivatives in order. For instance, if we desire to construct a polynomial that is zero at the origin, one at `x = 1`, two at `x = 2`, and has horizontal tangent lines at each of these three locations, we issue the following commands:

```
x=numpy.array([0,0,1,1,2,2]); y=numpy.aray([0,0,1,0,2,0])
interp=scipy.interpolate.KrogInterpolator(x,y)
xn=numpy.linspace(0,2,20)   # evaluate the polynomial in a larger set
matplotlib.pyplot.plot(x,y,'o',xn,interp(xn),'r')
```

This renders the following graph:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 73
http://site.ebrary.com/id/10672339?ppg=73
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

More advanced one-dimensional interpolation is possible with piecewise polynomials (`PiecewisePolynomial`). This allows control over the degrees of different pieces, as well as the derivatives at their intersections. Other interpolation options in the `scipy.interpolate` module are PCHIP monotonic cubic interpolation (`pchip`), or even univariate splines (`InterpolatedUnivariateSpline`).

Let us examine an example with the latter. Its syntax is as follows:

```
InterpolatedUnivariateSpline(x, y, w=None, bbox=[None, None], k=3)
```

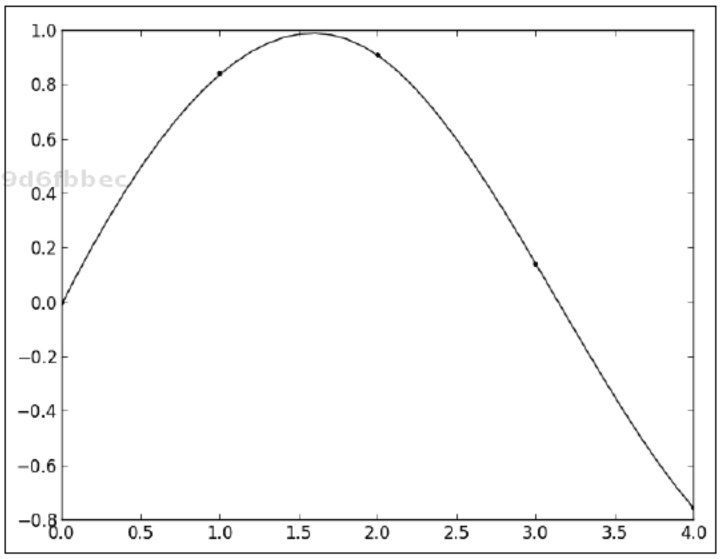
The arrays `x` and `y` contain the dependent and independent data, respectively. The array `w` contains positive weights for spline fitting. The two-sequence `bbox` specifies the boundary of the approximation interval. The last option indicates the degree of the smoothing polynomials (`k`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For instance, we desire to interpolate five points as shown in the following session. These points are ordered by strictly increasing `x` values. We need to perform this interpolation with four cubic polynomials (one for every two consecutive points), in such a way that at least the first derivative of each two consecutive pieces agree on their intersection. We will proceed as follows:

```
x=numpy.arange(5); y=numpy.sin(x)
xn=numpy.linspace(0,4,40)
interp=scipy.interpolate.InterpolatedUnivariateSpline(x,y)
matplotlib.pyplot.plot(x,y,'.',xn,interp(xn))
```

This offers the following plot showing interpolation with univariate splines:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 74
http://site.ebrary.com/id/10672339?ppg=74
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

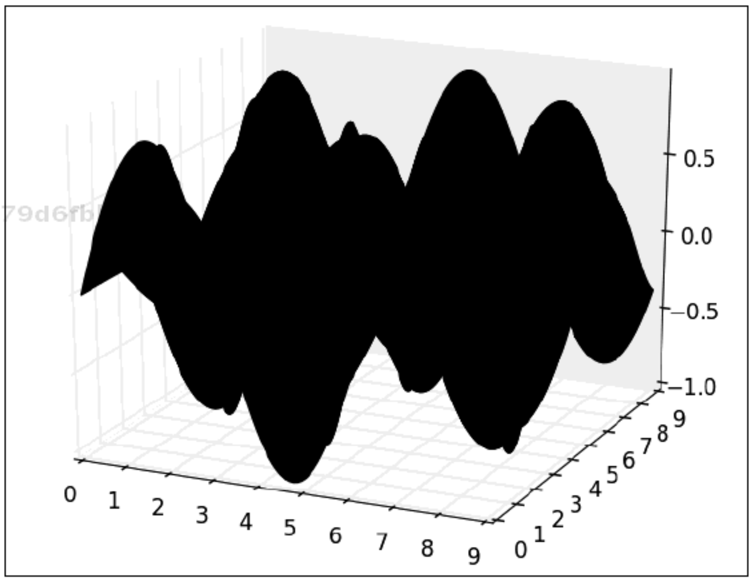
SciPy excels at interpolating in two-dimensional grids as well. It performs well with simple piecewise polynomials (`LinearNDInterpolator`), with piecewise constants (`NearestNDInterpolator`), or with more advanced splines (`BivariateSpline`). It is capable of carrying spline interpolation on rectangular meshes in a plane (`RectBivariateSpline`) or on the surface of a sphere (`RectSphereBivariateSpline`). For unstructured data, besides basic `BivariateSpline`, it is capable of computing smooth approximations (`SmoothBivariateSpline`) or more involved weighted least-squares splines (`LSQBivariateSpline`).

The following code creates a 10 x 10 grid of uniformly spaced points in the square from (0, 0) to (9, 9), and evaluates the function, `sin(x) * cos(y)` on them. We use these points to create a `BivariateSpline`, and evaluate the resulting function on the square for all values.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
x=y=numpy.arange(10)
f=(lambda i,j: numpy.sin(i)*numpy.cos(j))  # function to interpolate
A=numpy.fromfunction(f, (10,10))           # generate samples
spline=scipy.interpolate.RectBivariateSpline(x,y,A)
fig=matplotlib.pyplot.figure()
subplot=fig.add_subplot(111,projection='3d')
xx=numpy.mgrid[0:9:100j, 0:9:100j]         # larger grid for plotting
A=spline(numpy.linspace(0,9,100), numpy.linspace(0,9,100))
subplot.plot_surface(xx[0],xx[1],A)
```

The output is as follows, which shows interpolation of 2D data with bivariate splines:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 75
http://site.ebrary.com/id/10672339?ppg=75
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

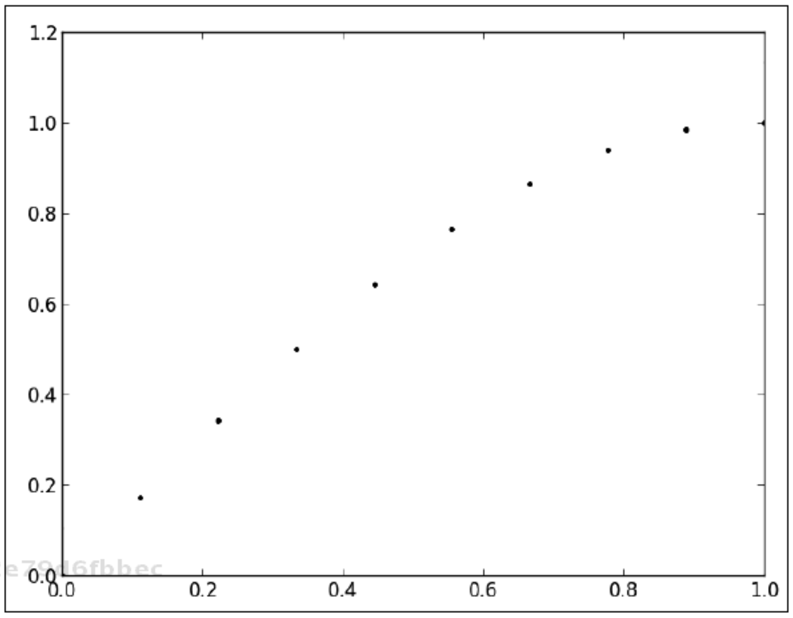
2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Regression is similar to interpolation. In this case, we assume that the data is imprecise, and we require an object of pre-determined structure to fit the data as closely as possible. The most basic example is univariate polynomial regression to a sequence of points. We obtain that with the `polyfit` command, which we introduced before briefly. For instance, we would like to compute the regression line in the least-squares sense, to a sequence of 10 uniformly spaced points on the interval from 0 to π/2 and their values under the sine function.

```
x=numpy.linspace(0,1,10)
y=numpy.sin(x*numpy.pi/2)
line=numpy.polyfit(x,y,deg=1)
matplotlib.pyplot.plot(x,y,'.'.x,numpy.polyval(line,x),'r')
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This gives the following plot showing linear regression with `polyfit`:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Curve fitting is possible also with splines, by using the parameters wisely. For example, with univariate spline fitting that we introduced before, we can play around with the weights, smoothing factor, the degree of the smoothing spline, and so on. On the same data as the previous example, if we desire to fit to for example, a parabolic spline, we could issue the following code:

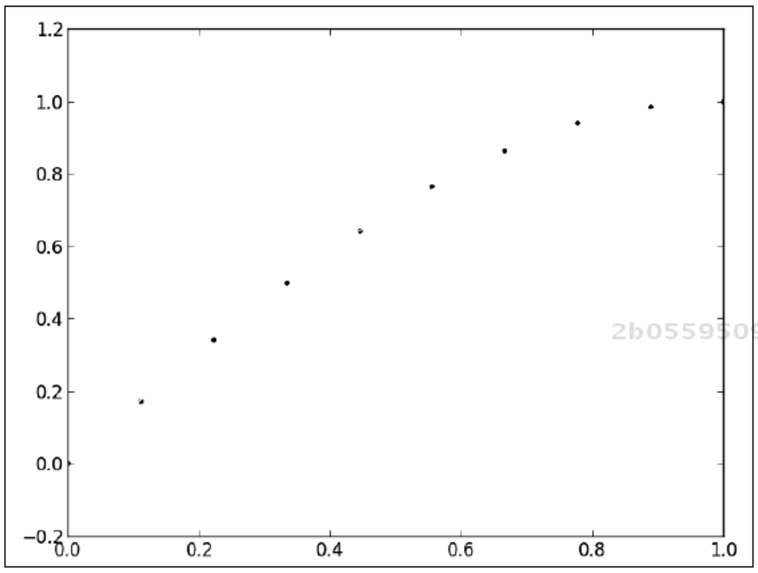
```
spline=scipy.interpolate.UnivariateSpline(x,y,k=2)
xn=numpy.linspace(0,1,100)
matplotlib.pyplot.plot(x,y,'.', xn, spline(xn))
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 76
http://site.ebrary.com/id/10672339?ppg=76
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This gives the following graph showing curve fitting with splines:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For regression, with the point of view of curve fitting, there is a generic routine, `curve_fit` in the `scipy.optimize` module. This routine minimizes the sum of squares of a set of equations using the Levenberg-Marquardt algorithm, and offers a best fit from any kind of functions (not only polynomials or splines). The syntax is simple as follows:

```
curve_fit(f, xdata, ydata, p0=None, sigma=None, **kw)
```

The `f` parameter is a callable function that represents the function we seek; `xdata` and `ydata` are arrays of the same length, containing the `x` and `y` coordinates of the points to be fit. The tuple `p0` holds an initial guess for the values to be found, and sigma is a vector of weights that could be used instead of the standard deviation of the data, if needed. We will show its usage with an enlightening example. We will start by generating some points on a section of a sine wave with amplitude `A=18`, angular frequency `w=3π`, and phase `h=0.5`. We corrupt the data in the array `y` with some small noise:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
A=18; w=3*numpy.pi; h=0.5
x=numpy.linspace(0,1,100); y=A*numpy.sin(w*x+h)
y += 4*((0.5-scipy.rand(100))*numpy.exp(2*scipy.rand(100)**2))
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 77
http://site.ebrary.com/id/10672339?ppg=77
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

We desire now to estimate the values of `A`, `w`, and `h` from the corrupted data, hence technically finding a curve fit from the set of sine waves. We start by gathering the three parameters in a list, and initializing them to some values, say `A = 20`, `w = 2π`, and `h = 1`. We also construct a callable expression of the `target` function:

```
p0 = [20, 2*numpy.pi, 1]
target_function = lambda x,AA,ww,hh: AA*numpy.sin(ww*x+hh)
```

We feed these, together with the fitting data to `curve_fit`, in order to find the required values:

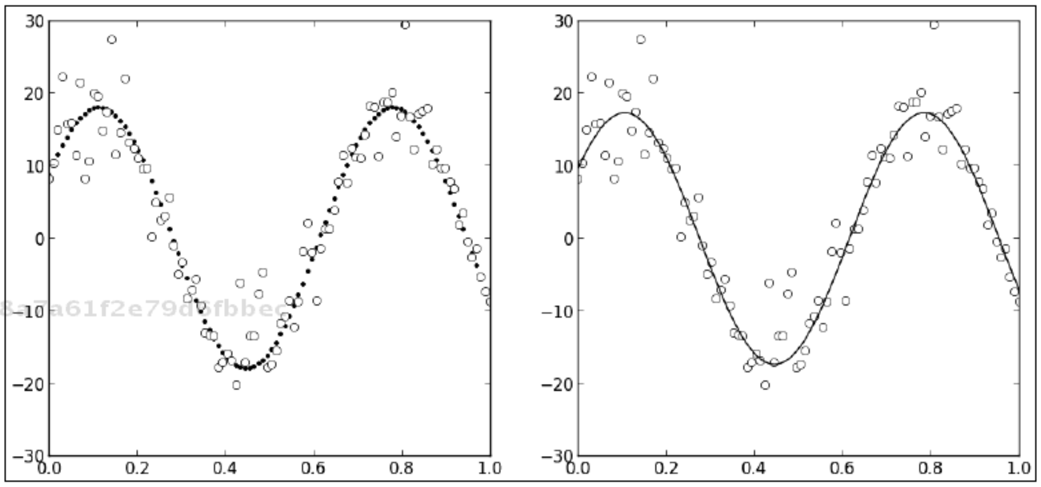
```
pF,pVar = scipy.optimize.curve_fit(target_function, x, y, p0)
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

A sample of `pF` run on any of our experiments should give an accurate result for the three requested values:

```
>>> print pF
[ 18.28142231   9.41943219   0.46405985]
```

This means that `A` was estimated to about 18.28, `w` was estimated very close to 3π, and `h` to about 0.46. The output of the initial data together with a computation of the corresponding sine wave is as follows, in which original data (left, in blue), corrupted (left and right, in red), and computed sine wave (right, in black) are shown:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Optimization

The field of optimization deals with finding extreme values of functions or their roots. We have seen the power of optimization already in the curve-fitting arena, but it does not stop here. There are applications to virtually every single branch of engineering, and robust algorithms to perform these tasks are a must in every scientist toolbox.

The `curve_fit` routine is actually syntactic sugar for the general algorithm that performs least-squares minimization – `leastsq`, with the imposing syntax:

```
leastsq(func, x0, args=(), Dfun=None, full_output=0,
        col_deriv=0, ftol=1.49012e-8, xtol=1.49012e-8,
        gtol=0.0, maxfev=0, epsfcn=0.0, factor=100, diag=None):
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For instance, the `curve_fit` routine could have been called with a `leastsq` call instead:

```
leastsq(error_function,p0,argx=(x,y))
```

Here, `error_function` is equal to `lambda p,x,y: target_function(x,p[0],p[1],p[2])-y`. Most of the optimization routines in SciPy can be accessed from either native Python code, or as wraps of Fortran or C classical implementations of their corresponding algorithms—technically, we are still using the same packages we did under Fortran or C, but from within Python. For instance, the minimization routine that implements the truncated Newton method can be called with `fmin_ncg` (and this is purely Python) or as `fmin_tnc` (and this one is a wrap of a C implementation).

Minimization

For general minimization problems, SciPy has many different algorithms. We have covered so far the least-squares algorithm (`leastsq`), but we also have brute force (`brute`), simulated annealing (`anneal`), Brent or Golden methods for scalar functions (`brent`, `golden`), the downhill simplex algorithm (`fmin`), Powell's method (`fmin_powell`), nonlinear conjugate gradient or Newton's version of it (`fmin_cg`, `fmin_ncg`), and the BFGS algorithm (`fmin_bfgs`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Constrained minimization is also possible computationally, and SciPy has for this task routines that implement the L-BFGS-S algorithm (`fmin_l_bfgs_s`), truncated Newton's algorithm (`fmin_tnc`), COBYLA (`fmin_cobyla`), or sequential least-squares programming (`fmin_slsqp`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 79
http://site.ebrary.com/id/10672339?ppg=79
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The following script, for example, compares the output of all different methods to finding a local minimum of the Rosenbrock function, `scipy.optimize.rosen` near the origin, using the downhill simplex algorithm:

```
>>>scipy.optimize.fmin(scipy.optimize.rosen,[0,0])
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 79
         Function evaluations: 146
array([ 1.00000439,  1.00001064])
```

Since the 0.11 version of SciPy, all minimization routines can be called from the generic `minimize`, with the `method` parameter pointing to one of the strings such as `Nelder-Mead` (for the downhill simplex), `Powell`, `CG`, `Newton-CG`, `BFGS`, or `anneal`. For constrained minimization, the corresponding strings are one of `L-BFGS-S`, `TNC` (for truncated Newton's), `COBYLA`, or `SLSQP`.

```
minimize( fun, x0, args=(), method='BFGS',
jac=None, hess=None, hessp=None,
           bounds=None, constraints=(),tol=None,
           callback=None, options=None)
```

Roots

For most special functions included in the `scipy.special` module, we have accurate algorithms that allow obtaining their zeros. For instance, for the Bessel function of first kind with integer order, `jn_zeros` offers as many roots as desired (in ascending order). We may obtain the first three roots of the Bessel J-function of order four by issuing the following command:

```
>>> print scipy.special.jn_zeros(4,3)
[  7.58834243  11.06470949  14.37253667]
```

For nonspecial scalar functions, the `scipy.optimize` module allows approximation to the roots through a great deal of different algorithms. For scalar functions, we have the crude bisection method (`bisect`), the classical secant method of Newton-Raphson (`newton`), and more accurate and faster methods such as Ridders' algorithm (`ridder`), and two versions of the Brent method (`brentq` and `brenth`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 80
http://site.ebrary.com/id/10672339?ppg=80
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The root finding for functions of several variables is very challenging in many ways; the largest the dimension, the worse, of course. The effectiveness of any of these algorithms depends very heavily on the problem, and it is a good idea to invest some time and resources in knowing them all. Since version 0.11 of SciPy, it is possible now to call any of the designed methods with the same routine `root`, which has the following syntax:

```
root(fun, x0, args=(), method='hybr',
jac=None, tol=None, callback=None, options=None)
```

The different methods are obtained upon changing the value of the `method` parameter to a method string. We may choose among the methods such as `'hybr'` for a modified hybrid Powell's method; `'lm'` for a modified least-squares method; `'broyden1'` or `'broyden2'` for Broyden's good and bad methods, respectively; `'diagbroyden'` for diagonal Broyden Jacobian approximation; `'anderson'` for Anderson's extended mixing; `'Krylov'` for Krylov approximation of the Jacobian; `'linearmixing'` for scalar Jacobian approximation; and `'excitingmixing'` for a tuned diagonal Jacobian approximation.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For large-scale problems, both the Krylov approximation of the Jacobian or the Anderson extended mixing are usually the best options.

Let us present an illustrative example of the power of these techniques. Consider the following system of differential equations:

$$\boxed{\begin{cases} x' = x^2 - 2x - y + 0.5 \\ y' = x^2 + 4y^2 - 4 \end{cases}}$$

We use the plot routine quiver from the `matplotlib.pyplot` libraries to visualize a slope field, for values of `x` and `y` between `-0.5` and `2.5`, and hence identify the location of the possible critical points in that region:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

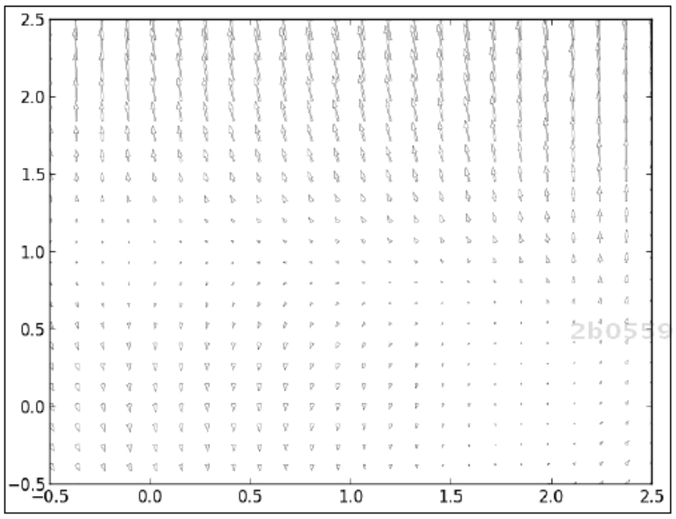
```
>>> f=lambda x: [x[0]**2-2*x[0]-x[1]+0.5, x[0]**2-4*x[1]**2-4]
>>>x,y=numpy.mgrid[-0.5:2.5:24j,-0.5:2.5:24j]
>>> U,V=f([x,y])
>>>matplotlib.pyplot.quiver(x,y,U,V,color='r', \
...         linewidths=(0.2,), edgecolors=('k'), \
...         headaxislength=5)
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 81
http://site.ebrary.com/id/10672339?ppg=81
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This gives the following:



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Note how there is a whole region of the plane, in which the slopes are extremely small. Because of the degrees of the polynomials involved, there are at most four different possible critical points. In this area we should be able to identify two (as a matter of fact there are only two noncomplex solutions). One of them seems to be near (`0, 1`), and the second near (`2, 0`). We use these two locations as initial guesses for our searches:

```
>>>scipy.optimize.root(f,[0,1])
  status: 1
 success: True
qtf: array([ -4.81190247e-09,  -3.83395899e-09])
nfev: 9
       r: array([ 2.38128242, -0.60840482, -8.35489601])
     fun: array([  3.59529073e-12,   3.85025345e-12])
       x: array([-0.22221456,  0.99380842])
 message: 'The solution converged.'
fjac: array([[-0.98918813, -0.14665209],
       [ 0.14665209, -0.98918813]])
>>>scipy.optimize.root(f,[2,0])
  status: 1
 success: True
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 82
http://site.ebrary.com/id/10672339?ppg=82
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
qtf: array([  2.08960516e-10,   8.61298294e-11])
nfev: 12
       r: array([-4.56575336, -1.67067665, -1.81464307])
     fun: array([  2.44249065e-15,   1.42996726e-13])
       x: array([ 1.90067673,  0.31121857])
 message: 'The solution converged.'
fjac: array([[-0.39612596, -0.91819618],
       [ 0.91819618, -0.39612596]])
```

In the first case, we converged successfully to (`-0.22221456`, `0.99380842`). In the second case, we converged to (`1.90067673`, `0.31121857`). The routine informs us details about the convergence and properties of the approximation. For instance, `nfev` tells us about the number of function calls performed, and `fun` indicates the output of the function at the found location. The other items in the output reflect the matrices used in the procedure, such as `qtf`, `r`, `fjac`.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Integration

SciPy is capable of performing very robust numerical integration. Definite integrals of a set of special functions are evaluated accurately with routines in the `scipy.special` module. For other functions, there are several different algorithms to obtain reliable approximations in the `scipy.integrate` module.

Exponential/logarithm integrals

The next diagram summarizes the indefinite and definite integrals in this category – the exponential integrals – `expn`, `expi`, and `exp1`; Dawson's integral `dawsn`; and Gauss error functions – `erf` and `erfc`. We also have Spence's dilogarithm (also known as Spence's integral).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

$$\texttt{expn(n,x)} = \int_1^\infty \frac{e^{-xt}}{t^n}\,dt \qquad \texttt{exp1(x)} = \int_1^\infty \frac{e^{-xt}}{t}\,dt$$

$$\texttt{expi(x)} = \int_{-\infty}^x \frac{e^t}{t}\,dt \qquad \texttt{dawsn(x)} = e^{-x^2}\int_0^x e^{t^2}\,dt$$

$$\texttt{erf(x)} = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt \qquad \texttt{erfc(x)} = \frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt$$

$$\texttt{spence(x)} = -\int_1^x \frac{\log t}{t-1}\,dt$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 83
http://site.ebrary.com/id/10672339?ppg=83
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Trigonometric and hyperbolic trigonometric integrals

In this category, we have Fresnel sine and cosine integrals, as well as the sinc and hyperbolic trigonometric integrals.

$$\texttt{fresnel(z)} = \int_0^z \sin\left(\tfrac{\pi}{2}t^2\right) dt$$

$$\texttt{sici(x)} = \int_0^x \frac{\sin t}{t}\, dt, \quad \gamma + \log x + \int_0^x \frac{\cos t - 1}{t}\, dt$$

$$\texttt{shichi(x)} = \int_0^x \frac{\sinh t}{t}\, dt, \quad \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t}\, dt$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

In the definitions given in the preceding diagram, gamma denotes the Euler-Mascheroni constant:

$$\gamma = \lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k} - \log n\right).$$

Elliptic integrals

These integrals arise naturally when computing the arc length of ellipses. SciPy follows the argument notation for elliptic integrals – complete (one argument) and incomplete (two arguments).

$$\texttt{ellipkm1(m)} = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}} \qquad \texttt{ellipe(m)} = \int_0^{\pi/2} \sqrt{1 - m\sin^2\theta}\, d\theta$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

$$\texttt{ellipkinc(m,n)} = \int_0^{n} \frac{d\theta}{\sqrt{1 - m\sin^2\theta}} \qquad \texttt{ellipeinc(m,n)} = \int_0^{n} \sqrt{1 - m\sin^2\theta}\, d\theta$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 84
http://site.ebrary.com/id/10672339?ppg=84
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Gamma and beta integrals

The following diagram shows the most useful of them all:

$$\texttt{gammainc(a,x)} = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1}\, dt$$

$$\texttt{gammaincc(a,x)} = \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{a-1}\, dt$$

$$\texttt{betainc(a,b,x)} = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1} (t-1)^{b-1}\, dt$$

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Numerical integration

For any other functions, we are content with approximating definite integrals with quadrature formulae, such as `quad` (adaptive quadrature), `fixed_quad` (fixed-order Gaussian quadrature), `quadrature` (fixed-tolerance Gaussian quadrature), and `romberg`, (Romberg integration). For functions of more than one variable, we have `dbquad` (two) and `tplquad` (three). The syntax in all cases is a variation of quad:

```
quad(func, a, b, args=(), full_output=0, epsabs=1.49e-08,
     epsrel=1.49e-08, limit=50, points=None, weight=None,
wvar=None, wopts=None, maxp1=50, limlst=50)
```

If instead of functions we have samples, we may use the routines such as `trapz`, `cumtrapz` (composite trapezoidal rule and its cumulative version); `romb` (Romberg integration again); and `simps` (Simpson's rule) instead. In these routines the syntax is simpler and changes the order of the parameters; for example, this is how we call `simps`:

```
simps(y, x=None, dx=1, axis=-1, even='avg')
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Those of us familiar with the QUADPACK libraries will find similar syntax, usage, and performance.

For extra information, run the `scipy.integrate.quad_explain()` command. This explains with great detail all the different outputs of the quadrature integrals included in the module result, estimate of absolute error, convergence, and explanation of the used weightings, if necessary. Let us give at least one meaningful example, where we integrate a special function, and compare the output of a quadrature formula against the more accurate value of the routines given in `scipy.special`:

```
>>> f=lambda t: numpy.exp(-t)*t**4
>>> from scipy.special import gammainc
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 85
http://site.ebrary.com/id/10672339?ppg=85
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
>>> from scipy.integrate import quad
>>> from scipy.misc import factorial
>>> print gammainc(5,1)
0.0036598468273437131
>>>result,error=quad(f,0,1)/factorial(4)
>>> result
0.0036598468273437122
```

To use a routine that integrates from samples, we have the flexibility of assigning the frequency and length of the data. For the following problem, we could try with `10,000` samples in the same interval:

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
>>> x=numpy.linspace(0,1,10000)
>>>scipy.integrate.simps(f(x)/factorial(4), x)
0.003659846827346905
```

Ordinary differential equations

As with integration, SciPy has some extremely accurate general-purpose solvers for systems of ordinary differential equations of first order.

$$\boxed{\frac{d\boldsymbol{y}}{dt} = f(t, \boldsymbol{y}), \quad \boldsymbol{y}(t) = \big(y_1(t), \ldots, y_n(t)\big), t \in \mathbb{R}}$$

For the case of real-valued functions we have basically two flavors – `ode` (with options passed with the `set_integrator` method) and `odeint` (simpler interface). The syntax of `ode` is as follows:

```
ode(f,jac=None)
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The first parameter, `f`, is the function to be integrated, and the second parameter, `jac`, refers to the matrix of partial derivatives with respect to the dependent variables (the Jacobian). This creates an `ode` object, with different methods to indicate the algorithm to solve the system (`set_integrator`), the initial conditions (`set_initial_value`), and different parameters to be sent to the function or its Jacobian.

The options for integration algorithm are `'vode'` for real-valued variable coefficient ODE solver, with fixed-leading-coefficient implementation (it provides Adam's method for non-stiff problems, and BDF for stiff); `'zvode'` for complex-valued variable coefficient ODE solver, with similar options to the previous; `'dopri5'` for a Runge-Kutta method of order (4)5; `'dop853'` for a Runge-Kutta method of order 8(5, 3).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 86
http://site.ebrary.com/id/10672339?ppg=86
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

The next session presents an example of usage of `ode` to solve the initial value problem:

$$\boxed{y' = -20y, \quad y(0) = 1}$$

We compute each step sequentially, and compare it with the actual solution, which is known. Notice that virtually there is no difference:

```
from scipy.integrate import ode
f=lambda t,y: -20*y        # The ODE
actual_solution=lambda t:numpy.exp(-20*t)  # actual solution
dt=0.01            # time step
solver=ode(f).set_integrator('dop853')  # solver
solver.set_initial_value(1,0)      # initial value
while solver.successful() and solver.t<=1+dt:
   # solve the equation at succesive time steps,
   # until the time is greater than 1
   # but make sure that the solution is successful
   print solver.t, solver.y, actual_solution(solver.t)
   # We compare each numerical solution with the actual
   # solution of the ODE
solver.integrate(solver.t + dt)    # solve next step
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Once run, this code gives us the following output:

```
<scipy.integrate._ode.ode at 0x10eac5e50>
0 [ 1.] 1.0
0.01 [ 0.81873075] 0.818730753078
0.02 [ 0.67032005] 0.670320046036
0.03 [ 0.54881164] 0.548811636094
0.04 [ 0.44932896] 0.449328964117
0.05 [ 0.36787944] 0.367879441171
0.06 [ 0.30119421] 0.301194211912
0.07 [ 0.24659696] 0.246596963942
0.08 [ 0.20189652] 0.201896517995
0.09 [ 0.16529889] 0.165298888222
0.1 [ 0.13533528] 0.135335283237
      ...
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 87
http://site.ebrary.com/id/10672339?ppg=87
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

```
0.9 [  1.52299797e-08] 1.52299797447e-08
0.91 [  1.24692528e-08] 1.24692527858e-08
0.92 [  1.02089607e-08] 1.02089607236e-08
0.93 [  8.35839010e-09] 8.35839010137e-09
0.94 [  6.84327102e-09] 6.84327102222e-09
0.95 [  5.60279644e-09] 5.60279643754e-09
0.96 [  4.58718175e-09] 4.58718174665e-09
0.97 [  3.75566677e-09] 3.75566676594e-09
0.98 [  3.07487988e-09] 3.07487987959e-09
0.99 [  2.51749872e-09] 2.51749871944e-09
1.0 [  2.06115362e-09] 2.06115362244e-09
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

For systems of differential equations of first order with complex-valued functions, we have a wrapper of `ode`, which we call with the `complex_ode` command. Syntax and usage are similar to those of `ode`.

The syntax of `odeint` is much more intuitive, and more Python friendly:

```
odeint(func, y0, t, args=(), Dfun=None, col_deriv=0, full_output=0,
       ml=None, mu=None, rtol=None, atol=None, tcrit=None, h0=0.0,
hmax=0.0, hmin=0.0, ixpr=0, mxstep=0, mxhnil=0, mxordn=12,
mxords=5, printmessg=0)
```

The most impressive part of this routine is that one is able to indicate not only the Jacobian, but also whether this is banded (and how many nonzero diagonals under or over the main diagonal we have, with the `ml` and `mu` options). This speeds up computations by a huge factor. Another amazing feature of `odeint` is the possibility to indicate critical points for the integration (`tcrit`).

We will now introduce an application to analyze Lorentz attractors with the routines presented in this section.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Lorenz Attractors

No book on scientific computing is complete without revisiting Lorenz attractors; SciPy excels both at computation of solutions and presentation of ideas based upon systems of differential equations, of course, and we show how and why in this section.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 88
http://site.ebrary.com/id/10672339?ppg=88
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Consider a two-dimensional fluid cell that is heated from underneath and cooled from above, much like what occurs with the earth's atmosphere. This creates convection that can be modeled by a single partial differential equation, for which a decent approximation has the form of the following system of ordinary differential equations:

$$\boxed{\begin{cases} \frac{dx}{dt} = \sigma(y - x) \\ \frac{dy}{dt} = rx - y - xz \\ \frac{dz}{dt} = xy - bz \end{cases}}$$

The variable `x` represents the rate of convective overturning. Variables `y` and `z` stand for the horizontal and vertical temperature variations, respectively. This system depends on four physical parameters, the descriptions of which are far beyond the scope of this book. The important point is that we may model earth's atmosphere with these equations, and in that case a good choice for the parameters is given by sigma = 10, and b = 8 / 3. For certain values of the third parameter, we have systems for which the solutions behave chaotically. Let us explore this effect with the help of SciPy.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

We will use one of the solvers in the `scipy.integrate` module, as well as plotting utilities:

```
import numpy
from numpy import linspace
import scipy
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
sigma=10.0
b=8/3.0
r=28.0
f = lambda x,t: [sigma*(x[1]-x[0]),
                 r*x[0]-x[1]-x[1]*x[2],
                 x[0]*x[1]-b*x[2]]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Let us choose a time interval `t` large enough with a sufficiently dense partition and any initial condition, `y0`.

```
>>> t=linspace(0,20,2000); y0=[5.0,5.0,5.0]
>>> solution=odeint(f,y0,t)
>>> X=solution[:,0]; Y=solution[:,1]; Z=solution[:,2]
```

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

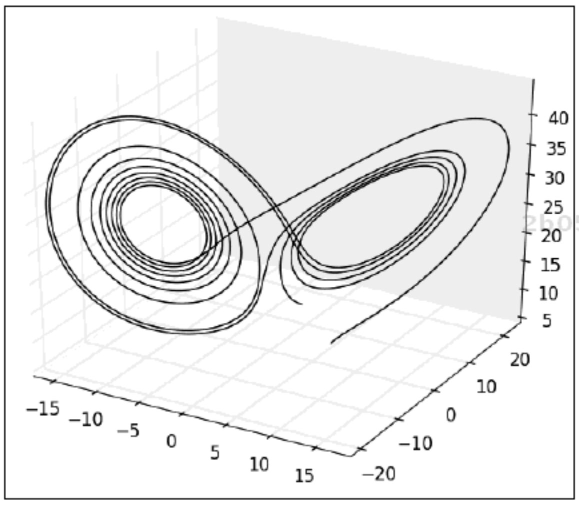
Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 89
http://site.ebrary.com/id/10672339?ppg=89
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

If we desire to plot a 3D rendering of the solution obtained, we may do so as follows:

```
>>> plt.gca(projection='3d'); plt.plot(X,Y,Z)
```

This produces the following graph, showing a Lorenz attractor:



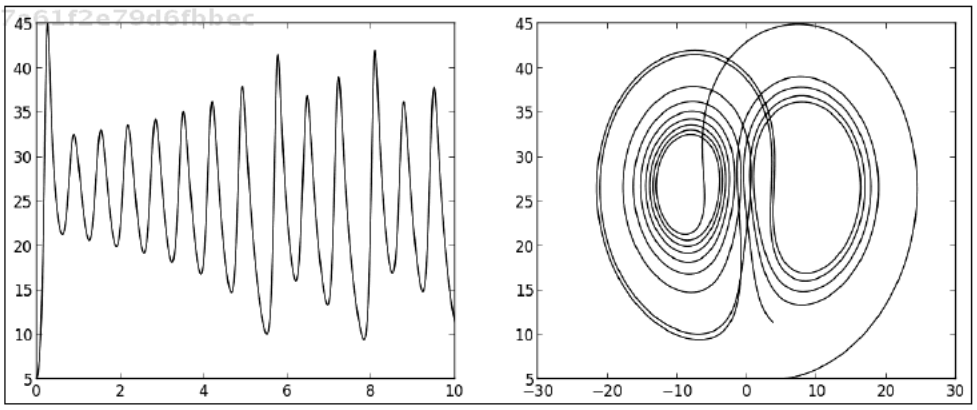
2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

This is most illustrative, and shows precisely the chaotic behavior of the solutions. Let us observe the fluctuations of the vertical temperature in detail, as well as the fluctuation of horizontal temperature against vertical:

```
>>>plt.subplot(121,aspect='equal'); plt.plot(t,Z)
>>>plt.subplot(122,aspect='equal'); plt.plot(Y,Z)
```

This produces the following the plots, showing vertical temperature with respect to time (left) and horizontal versus vertical temperature (right):

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary



2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 90
http://site.ebrary.com/id/10672339?ppg=90
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Summary

This chapter explored the topics of special functions, integration, interpolation, and optimization through the corresponding modules (`special`, `integrate`, `interpolate`, `optimize`).

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

2b0559509ad0af8a7a61f2e79d6fbbec
ebrary

Blanco-Silva, Francisco J.. Learning SciPy for Numerical and Scientific Computing.
: Packt Publishing, . p 91
http://site.ebrary.com/id/10672339?ppg=91
Copyright © Packt Publishing. . All rights reserved.
May not be reproduced in any form without permission from the publisher,
except fair uses permitted under U.S. or applicable copyright law.