Table of contents (../toc.ipynb)

SciPy

- Scipy extends numpy with powerful modules in
    - optimization,
    - interpolation,
    - linear algebra,
    - fourier transformation,
    - signal processing,
    - image processing,
    - file input output, and many more.
- Please find here the scipy reference for a complete feature list https://docs.scipy.org/doc/scipy/reference/ (https://docs.scipy.org/doc/scipy/reference/).

We will take a look at some features of scipy in the latter. Please explore the rich content of this package later on.

Optimization

- Scipy's optimization module provides many optimization methods like least squares, gradient methods, BFGS, global optimization, and many more.
- Please find a detailed tutorial here https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html (https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html).
- Next, we will apply one of the optimization algorithms in a simple example.

A common function to test optimization algorithms is the Rosenbrock function for N variables:

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

The optimum is at $x_i = 1$, where $f(\boldsymbol{x}) = 0$

In [2]:
```
import numpy as np
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
%matplotlib inline
```

In [3]:
```
def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
```

We need to generate some data in a mesh grid.

In [4]:
```
X = np.arange(-2, 2, 0.2)
Y = np.arange(-2, 2, 0.2)
X, Y = np.meshgrid(X, Y)

data = np.vstack([X.reshape(X.size), Y.reshape(Y.size)])
```

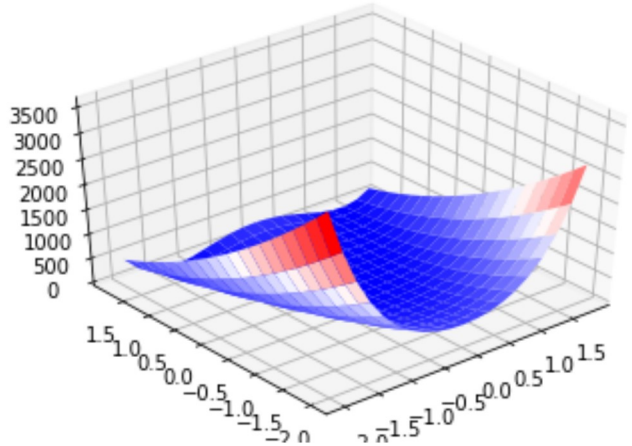
Let's evaluate the Rosenbrock function at the grid points.

In [5]:
```
Z = rosen(data)
```

And we will plot the function in a 3D plot.

In [6]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, Z.reshape(X.shape), cmap='bwr')
ax.view_init(40, 230)
```



Now, let us check that the true minimum value is at (1, 1).

In [37]:
```
rosen(np.array([1, 1]))
```

Out[37]:
```
0.0
```

Finally, we will call scipy optimize and find the minimum with Nelder Mead algorithm.

In [38]:
```
from scipy.optimize import minimize

x0 = np.array([1.3, 0.7])
res = minimize(rosen, x0, method='nelder-mead',
               options={'xatol': 1e-8, 'disp': True})

print(res.x)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 79
         Function evaluations: 150
[1. 1.]
```

Many more optimization examples are to find in scipy optimize tutorial https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html (https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html).

In [39]:
```
IFrame(src='https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html',
       width=1000, height=600)
```

Out[39]:

(../index.html)

Optimization (scipy.optimize (../optimize.html#module-scipy.optimize))

The **scipy.optimize** (../optimize.html#module-scipy.optimize) package provides several commonly used optimization algorithms. A detailed listing is available: **scipy.optimize** (../optimize.html#module-scipy.optimize) (can also be found by `help(scipy.optimize)`).

The module contains:

1. Unconstrained and constrained minimization of multivariate scalar functions (**minimize** (../generated/scipy.optimize.minimize.html#scipy.optimize.minimize)) using a variety of algorithms (e.g., BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP).
2. Global optimization routines (e.g., **basinhopping** (../generated/scipy.optimize.basinhopping.html#scipy.optimize.basinhopping), **differential_evolution** (../generated/scipy.optimize.differential_evolution.html#scipy.optimize.differential_evolution), **shgo** (../generated/scipy.optimize.shgo.html#scipy.optimize.shgo), **dual_annealing**

Interpolation

- Interpolation of data is very often required, for instance to replace NaNs or to fill missing values in data records.
- Scipy comes with
    - 1D interpolation,
    - multivariate data interpolation
    - spline, and
    - radial basis function interpolation.
- Please find here the link to interpolation tutorials https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html (https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html).

In [40]:

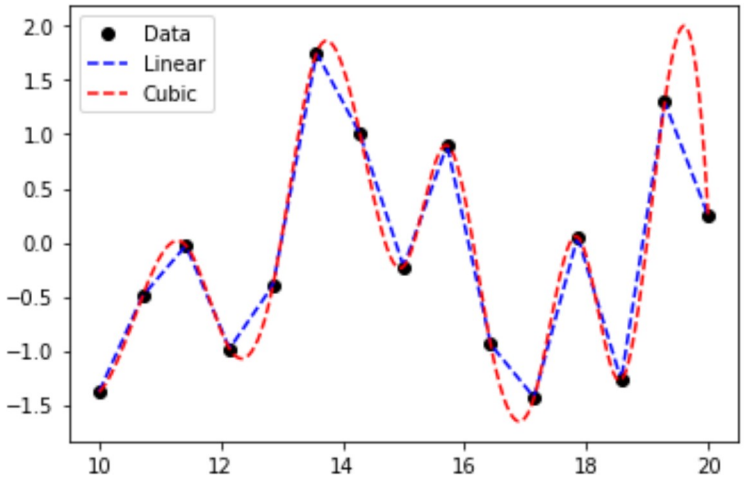
```
from scipy.interpolate import interp1d

x = np.linspace(10, 20, 15)
y = np.sin(x) + np.cos(x**2 / 10)

f = interp1d(x, y, kind="linear")
f1 = interp1d(x, y, kind="cubic")
```

In [41]:
```
x_fine = np.linspace(10, 20, 200)

plt.plot(x, y, 'ko',
         x_fine, f(x_fine), 'b--',
         x_fine, f1(x_fine), 'r--')
plt.legend(["Data", "Linear", "Cubic"])
plt.show()
```



Signal processing

The signal processing module is very powerful and we will have a look at its tutorial https://docs.scipy.org/doc/scipy/reference/tutorial/signal.html (https://docs.scipy.org/doc/scipy/reference/tutorial/signal.html) for a quick overview.

In [42]:
```
IFrame(src='https://docs.scipy.org/doc/scipy/reference/tutorial/signal.html',
       width=1000, height=600)
```

Out[42]:

(../index.html)

Signal Processing (scipy.signal (../signal.html#module-scipy.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 1- and 2-D 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.

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-D function is the finite-basis expansion.

$$\left(x \right.$$

Linear algebra

- In addition to numpy, scipy has its own linear algebra module.
- It offers more functionality than numpy's linear algebra module and is based on BLAS/LAPACK support, which makes it faster.
- The respective tutorial is here located https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html (https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html).

In [43]:
```
IFrame(src='https://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html',
       width=1000, height=600)
```

Out[43]:

(../index.html)

Linear Algebra (scipy.linalg (../linalg.html#module-scipy.linalg))

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-D array. The output of these routines is also a 2-D array.

scipy.linalg vs numpy.linalg

scipy.linalg (../linalg.html#module-scipy.linalg) contains all the functions in numpy.linalg (https://www.numpy.org/devdocs/reference/routines.linalg.html). plus some other more advanced ones not contained in `numpy.linalg`.

Another advantage of using `scipy.linalg` over `numpy.linalg` is that it is always

Total least squares as linear algebra application

We will now implement a total least squares estimator [Markovsky2007] (../references.bib) with help of scipy's singular value decomposition (svd). The total least squares estimator provides a solution for the errors in variables problem, where model inputs and outputs are corrupted by noise.

The model becomes $AX \approx B$, where $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times d}$ are input and output data, and X is the unknown parameter vector.

More specifically, the total least squares regression becomes $\widehat{A}X = \widehat{B}$, $\widehat{A} := A + \Delta A$, $\widehat{B} := B + \Delta B$.

The estimator can be written as pseudo code as follows.

$C = [AB] = U\Sigma V^\top$, where $U\Sigma V^\top$ is the svd of C.

$$V := \begin{bmatrix} V_{11} V_{12} \\ V_{21} V_{22} \end{bmatrix},$$

$$\widehat{X} = -V_{12} V_{22}^{-1}.$$

In Python, the implementation could be like this function.

In [44]:
```
from scipy import linalg


def tls(A, B):
    m, n = A.shape
    C = np.hstack((A, B))
    U, S, V = linalg.svd(C)
    V12 = V.T[0:n, n:]
    V22 = V.T[n:, n:]
    X = -V12 / V22
    return X
```

Now we create some data where input and output are appended with noise.

In [45]:
```
A = np.random.rand(100, 2)
X = np.array([[3], [-7]])
B = A @ X

A += np.random.randn(100, 2) * 0.1
B += np.random.randn(100, 1) * 0.1
```

The total least squares solution becomes

In [46]:
```
tls(A, B)
```

Out[46]:
```
array([[ 2.95496063],
       [-6.7920108 ]])
```

And this solution is closer to correct value $X = [3, -7]^\top$ than ordinary least squares.

In [47]:
```
linalg.solve((A.T @ A), (A.T @ B))
```

Out[47]:
```
array([[ 2.5035526 ],
       [-6.28316014]])
```

Finally, next function shows a "self" written least squares estimator, which uses QR decomposition and back substitution. This implementation is numerically robust in contrast to normal equations

$$A^\top AX = A^\top B.$$

Please find more explanation in [Golub2013] (../references.bib) and in section 3.11 of [Burg2012] (../references.bib).

In [48]:
```
def ls(A, B):
    Q, R = linalg.qr(A, mode="economic")
    z = Q.T @ B

    return linalg.solve_triangular(R, z)
```

In [49]:
```
ls(A, B)
```

Out[49]:
```
array([[ 2.5035526 ],
       [-6.28316014]])
```

Integration

- Scipy's integration can be used for general equations as well as for ordinary differential equations.
- The integration tutorial is linked here https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html (https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html).

Solving a differential equation

Here, we want to use an ode solver to simulate the differential equation (ode)

$$y'' + y' + 4y = 0.$$

To evaluate this second order ode, we need to convert it into a set of first order ode. The trick is to use this substitution: $x_0 = y, x_1 = y'$, which yields

$$x_0' = x_1$$
$$x_1' = -4x_0 - x_1$$

The implementation in Python becomes.

In [50]:
```
def equation(t, x):
    return [x[1], -4 * x[0] - x[1]]
```

In [51]:
```
from scipy.integrate import solve_ivp
```

In [52]:
```
time_span = [0, 20]
init = [1, 0]
time = np.arange(0, 20, 0.01)
sol = solve_ivp(equation, time_span, init, t_eval=time)
```

In [53]:
```
plt.plot(time, sol.y[0, :])
plt.plot(time, sol.y[1, :])
plt.legend(["$y'$", "$y''$"])
plt.xlabel("Time")
plt.show()
```