SCRIPTING AND PROGRAMMING LABORATORY FOR DATA ANALYSIS

Lecture 6 – Introduction to scipy

Scipy

The *Scientific Python* (*SciPy*) package contains several tools dedicated to most part of the problems encountered in scientific research, like:

- Interpolation
- Integration
- Optimisation
- Special functions
- Linear algebra
- Fourier transform
- ...

As *Numpy*, also *Scipy* is largely written in Fortran or C, making it very computationally efficient and optimised.

Scipy

The *Scientific Python* (*SciPy*) package contains several tools dedicated to most part of the problems encountered in scientific research, like:

- Interpolation
- Integration
- Optimisation
- Special functions
- Linear algebra
- Fourier transform
- ...

Before implementing an algorithm by yourself, check the scipy doc
DO NOT REINVENT THE WHEEL!

As *Numpy*, also *Scipy* is largely written in Fortran or C, making it very computationally efficient and optimised.

Scipy-Structure

SciPy User Guide

- Introduction
- Special functions (**scipy.special**)
- Integration (**scipy.integrate**)
- Optimization (**scipy.optimize**)
- Interpolation (**scipy.interpolate**)
- Fourier Transforms (**scipy.fft**)
- Signal Processing (**scipy.signal**)
- Linear Algebra (**scipy.linalg**)
- Sparse eigenvalue problems with ARPACK
- Compressed Sparse Graph Routines (**scipy.sparse.csgraph**)
- Spatial data structures and algorithms (**scipy.spatial**)
- Statistics (**scipy.stats**)
- Multidimensional image processing (**scipy.ndimage**)
- File IO (**scipy.io**)

(Probably) most useful modules for scientific computing

Use the help function or the online documentation to navigate into the countless number of modules!

Scipy-Stats

The module **scipy.stats** contains a large number of probability distributions, summary and frequency statistics, correlation functions and statistical tests, masked statistics, kernel density estimation, quasi-Monte Carlo functionality, and more check the doc!

In this package you generally invoke an object, then you need various methods to access different available functionalities.

Scipy-Stats

The
prob … cs,
cor
sta
func

In t
need
func

Probability distributions

Each univariate distribution is an instance of a subclass of `rv_continuous` (`rv_discrete` for discrete distributions):

`rv_continuous`([momtype, a, b, xtol, ...])	A generic continuous random variable class meant for subclassing.
`rv_discrete`([a, b, name, badvalue, ...])	A generic discrete random variable class meant for subclassing.
`rv_histogram`(histogram, *args, **kwargs)	Generates a distribution given by a histogram.

Scipy-Stats

This is a gaussian distribution

scipy.stats.norm

scipy.stats.norm = *<scipy.stats._continuous_distns.norm_gen object>* [source]

A normal continuous random variable.

The location (loc) keyword specifies the mean. The scale (scale) keyword specifies the standard deviation.

As an instance of the **rv_continuous** class, **norm** object inherits from it a collection of generic methods (see below for the full list), and completes them with details specific for this particular distribution.

istics,

rlo

you

functionalities.

Scipy-Stats

scipy.stats.norm

`scipy.stats.norm = <scipy.stats._continu`

A normal continuous random variable.

The location (`loc`) keyword specifies the mean.
deviation.

As an instance of the **`rv_continuous`** class, **no**
methods (see below for the full list), and comple
distribution.

functionalities.

Methods

You can access the functional form

rvs(loc=0, scale=1, size=1, random_state=None)	Random variates.
pdf(x, loc=0, scale=1)	Probability density function.
logpdf(x, loc=0, scale=1)	Log of the probability density function.
cdf(x, loc=0, scale=1)	Cumulative distribution function.
logcdf(x, loc=0, scale=1)	Log of the cumulative distribution function.
sf(x, loc=0, scale=1)	Survival function (also defined as `1 - cdf`, but *sf* is sometimes more accurate).
logsf(x, loc=0, scale=1)	Log of the survival function.
ppf(q, loc=0, scale=1)	Percent point function (inverse of `cdf` — percentiles).
isf(q, loc=0, scale=1)	Inverse survival function (inverse of `sf`).
moment(n, loc=0, scale=1)	Non-central moment of order n
stats(loc=0, scale=1, moments='mv')	Mean('m'), variance('v'), skew('s'), and/or kurtosis('k').

Scipy-Interpolation

In the interpolation module (**scipy.interpolate**) you can find (check also here):

- A class representing an 1D interpolant (**interp1d**), offering several interpolation methods.
- Functions for 1D and 2D (smoothed) **cubic-spline** interpolation, based on the FORTRAN library FITPACK.
- The function **griddata** offering a simple interface to interpolation in N dimensions (N = 1, 2, 3, 4, …).
- Many others, check documentation!

Scipy-Interpolation

- **interp1d** (doc):create a function based on fixed data points, which can be evaluated anywhere within the domain defined by the given data. An instance of this class is created by passing the 1-D vectors comprising the data.

```
>>> from scipy.interpolate import interp1d
```

```
>>> x = np.linspace(0, 10, num=11, endpoint=True)
>>> y = np.cos(-x**2/9.0)
>>> f = interp1d(x, y)
>>> f2 = interp1d(x, y, kind='cubic')
```

Functions!

Check the documentation for other option of interpolation degree

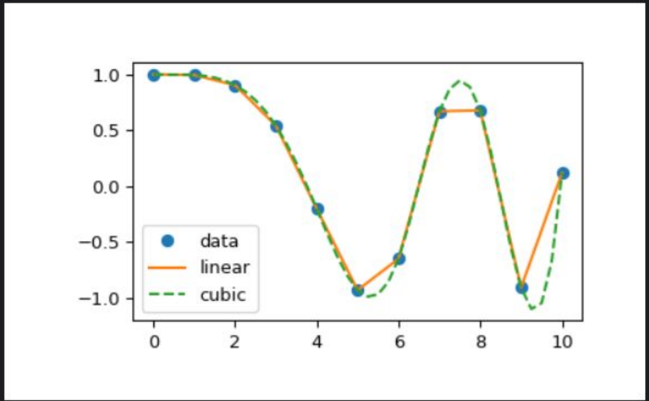
Scipy-Interpolation

- **interp1d** (doc):create a function based on fixed data points, which can be evaluated anywhere within the domain defined by the given d
created by passing the

```
>>> from scipy.interpolate import
```

```
>>> x = np.linspace(0, 10, num=11,
>>> y = np.cos(-x**2/9.0)
>>> f = interp1d(x, y)
>>> f2 = interp1d(x, y, kind='cub
```

```
>>> xnew = np.linspace(0, 10, num=41, endpoint=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')
>>> plt.legend(['data', 'linear', 'cubic'], loc='best')
>>> plt.show()
```



Scipy - Interpolation

- **Splines** (doc):
    - Procedural usage: two essential steps, first a spline representation of the curve is computed, then the spline is evaluated at the desired points

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import interpolate
```

Cubic-spline

```
>>> x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/8)
>>> y = np.sin(x)
>>> tck = interpolate.splrep(x, y, s=0)
>>> xnew = np.arange(0, 2*np.pi, np.pi/50)
>>> ynew = interpolate.splev(xnew, tck, der=0)
```

splrep *computes the spline coefficients based on data.*

Note the ***s*** *parameter, this represents the smoothing, by setting it to* ***zero,*** *you force the spline representation to pass through all data point! Beware: this is not the default.*

splev *evaluates the spline at new points within the domain of x. The function needs the spline object tck!*

Other functions can evaluate the spline derivative, integral, roots, e.g.: ***spalde,splint,sproot***

Scipy - Interpolation

- **Splines** (doc):
  - Object-Oriented usage: the **UnivariateSpline** class does the job. At object creation you pass the data and you directly get a function to be evaluated at different data points

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy import interpolate
```

InterpolatedUnivariateSpline

```
>>> x = np.arange(0, 2*np.pi+np.pi/4, 2*np.pi/8)
>>> y = np.sin(x)
>>> s = interpolate.InterpolatedUnivariateSpline(x, y)
>>> xnew = np.arange(0, 2*np.pi, np.pi/50)
>>> ynew = s(xnew)
```

InterpolateUnivariateSpline *computes the spline!*
s *is now callable*

InterpolateUnivariateSpline vs UnivariateSpline: *difference is the smoothing parameter that for the former is set to zero, i.e. the spline passes through all points.*

Scipy-Interpolation

- **griddata** (doc): Interpolate unstructured D-D data

```
scipy.interpolate.griddata(points, values, xi,
method='linear', fill_value=nan, rescale=False)
```

Data points:
2-D ndarray of floats with shape (n, D).
n: number of data points
D: number of dimensions

Values at data points:
ndarray of float or complex with shape (n,)

Method could be{'linear', 'nearest', 'cubic'}

Points at which to interpolate data:
2-D ndarray of floats with shape (m, D).
m: number of points
D: number of dimensions

Scipy-Interpolation

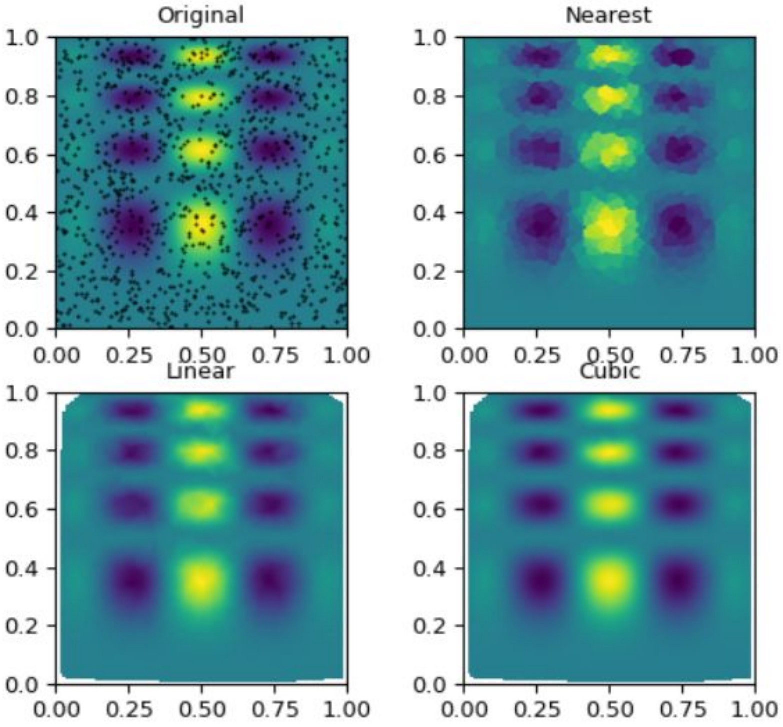
- **griddata** (doc): Interpola

```
scipy.interpolate.griddata(po
method='linear', fill_value=n
```

Data points:
2-D ndarray of floats with shape (n, D).
n: number of data points
D: number of dimensions

ndarray o

Method could be{'linear', 'nearest'
'cubic'}



Scipy - Integration

The **scipy.integrate** sub-package provides several integration techniques to compute integrals (1D or higher). It also includes an ordinary differential equation integrator (doc).

Methods for integrating functions given the function object:

- quad: General purpose integration.
- dblquad: General purpose double integration.
- tplquad: General purpose triple integration.
- fixed_quad: Integrate func(x) using Gaussian quadrature of order n.
- quadrature: Integrate with given tolerance using Gaussian quadrature.
- romberg: Integrate function using Romberg integration.

Scipy-Integration

The **scipy.integrate** sub-package provides several integration techniques to compute integrals (1D or higher). It also includes an ordinary differential equation integrator (doc).

Methods for integrating functions given the function object:

```python
from scipy import integrate

result = integrate.quad(lambda x: 3*x**2, 0, 5)
print(result)
```

```
(125.00000000000001, 1.3877787807814459e-12)
```

ion.
ion.
sian quadrature of
ce using Gaussian

- romberg: Integrate function using Romberg integration.

Result contains the computed value and an estimate of the error

We need to provide a function and integration limits

Scipy-Integration

The **scipy.integrate** sub-package provides several integration techniques to compute integrals (1D or higher). It also includes an ordinary differential equation integrator (doc).

Methods for Integrating Functions given fixed samples:

- trapezoid: Use trapezoidal rule to compute integral.
- cumulative_trapezoid: Use trapezoidal rule to cumulatively compute integral.
- simpson: Use Simpson's rule to compute integral from samples.
- romb: Use Romberg Integration to compute integral from (2^k + 1) evenly-spaced samples.

Scipy-Integration

The scipy.integrate sub-packag

```python
def f(x):
    return 3*x**2

# compute the integral with different numbe
for i in [100,1000,10000,100000]:
    x_samp = np.linspace(0,5,i)
    y_samp = f(x_samp)
    I = integrate.trapezoid(y_samp,x_samp)
    print("Result(n =",i,") =", I)
```

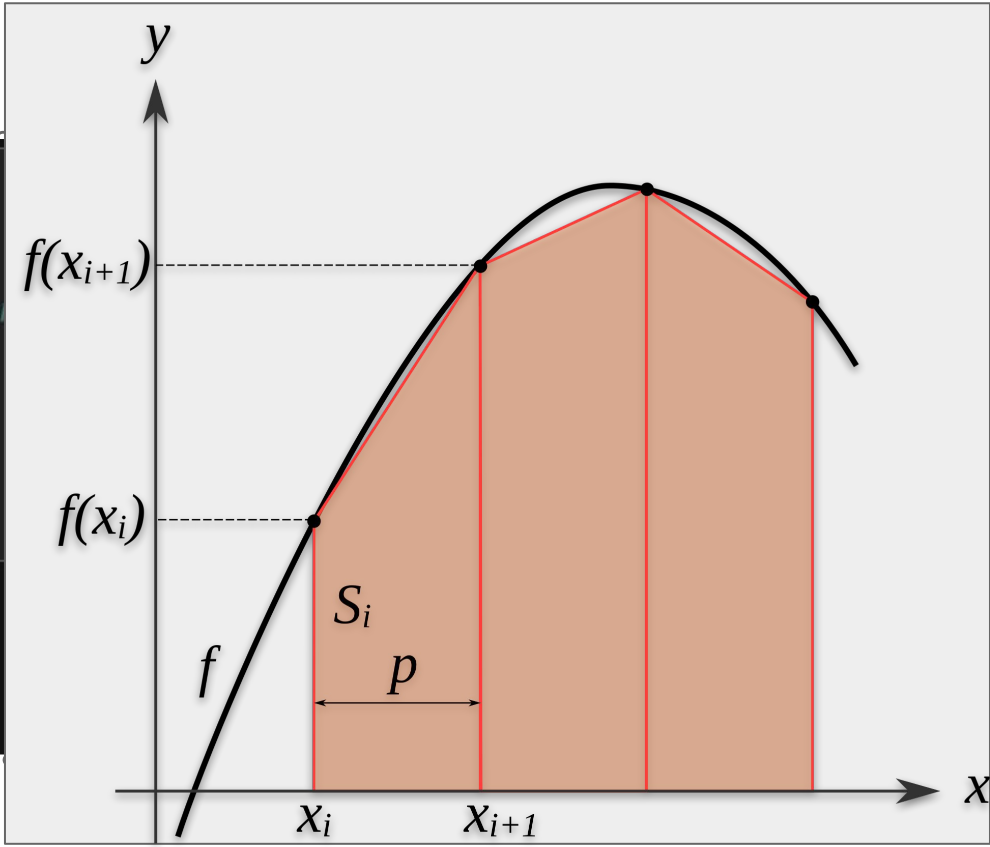
```
Result(n = 100 ) = 125.00637690003163
Result(n = 1000 ) = 125.00006262518775
Result(n = 10000 ) = 125.00000062512501
Result(n = 100000 ) = 125.00000000625013
```

(2**k + 1) evenly spaced s

X data points

Function values

Result from trapezoidal rule



Scipy - Integration

The **scipy.integrate** sub-package provides several integration techniques to compute integrals (1D or higher). It also includes an ordinary differential equation integrator (doc).

Methods for integrating differential equations:

Easy-to-use

Accurate

- odeint: Solve a system of ordinary differential equations using lsoda algorithm only.
- ode: A generic interface class to several numeric integrators.
- solve_ivp: newest scipy integrator pack, allows to employ several integrator techniques and event tracing.

Scipy-Integration

odeint: Solve a system of ordinary differential equations using lsoda algorithm only.

```
scipy.integrate.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,
tfirst=False)
```

***func**: callable(y, t, …) or callable(t, y, …) If **tfirst=True** then second signature is assumed*

Initial conditions(ICs) Array with length equal to the number of dependent variables

A sequence of time points for which to solve for y

*Extra arguments for the function **func***

Let *y* be the vector [*theta*, *omega*]. We implement this system in Python as:

```
>>> def pend(y, t, b, c):
...     theta, omega = y
...     dydt = [omega, -b*omega - c*np.sin(theta)]
...     return dydt
...
```

We assume the constants are *b* = 0.25 and *c* = 5.0:

```
>>> b = 0.25
>>> c = 5.0
```

For initial conditions, we assume the pendulum is nearly vertical with *theta(0)* = *pi* - 0.1, and is initially at rest, so *omega(0)* = 0. Then the vector of initial conditions is

```
>>> y0 = [np.pi - 0.1, 0.0]
```

nary differential equations

0, t, args=(), Dfun=None,

We will generate a solution at 101 evenly spaced samples in the interval 0 <= *t* <= 10. So our array of times is:

```
>>> t = np.linspace(0, 10, 101)
```

Call **odeint** to generate the solution. To pass the parameters *b* and *c* to *pend*, we give them to **odeint** using the *args* argument.

```
>>> from scipy.integrate import odeint
>>> sol = odeint(pend, y0, t, args=(b, c))
```

The solution is an array with shape (101, 2). The first column is *theta(t)*, and the second is *omega(t)*. The following code plots both components.

***func**: callable(y, t, …)*
or callable(t, y, …)
*If **tfirst=True** then*
second signature is
assumed

Array with leng
to the numbe
dependent var

Scipy-Integration

ode: A generic interface class to several numeric integrators.

```
class scipy.integrate.ode(f, jac=None)
```

f: callable(t, y, …)

Attributes:

t: (float) Current time.

y: (ndarray) Current variable values.

SCIPY-INTEGRATION

`ode`: A generic interface class t
integrators.

```
class scipy.integrate.ode(f, jac
```

f: callable(t, y, …)

Attributes:

- **t:** (float) Current time.
- **y:** (ndarray) Current variabl

Methods

get_return_code()	Extracts the return code for the integration to enable better control if the integration fails.
integrate(t[, step, relax])	Find y=y(t), set y as an initial condition, and return y.
set_f_params(*args)	Set extra parameters for user-supplied function f.
set_initial_value(y[, t])	Set initial conditions y(t) = y.
set_integrator(name, **integrator_params)	Set integrator by name.
set_jac_params(*args)	Set extra parameters for user-supplied function jac.
set_solout(solout)	Set callable to be called at every successful integration step.
successful()	Check if integration was successful.

Scipy-Integration

```python
from scipy.integrate import ode

# system of differential equations
def pend(t, y, b, c):
    theta, omega = y
    dydt = [omega, -b*omega - c*np.sin(theta)]
    return dydt

# function use to save output
def solout(t, y):
    sol.append([t, y[0], y[1]])

# integrator initialisation, here we choose the alogorithm dop853
solver = ode(pend).set_integrator("dop853",atol=1e-15,rtol=1e-15,first_step=0.01,nsteps=10000000)
solver.set_solout(solout)
solver.set_initial_value([np.pi - 0.1, 0.0]).set_f_params(0.25,5.)

sol = []
solver.integrate(10)
sol = np.array(sol)

print(sol)
print(solver.t,solver.y)
```

```
[[ 0.00000000e+00  3.04159265e+00  0.00000000e+00]
 [ 1.00000000e-02  3.04156771e+00 -4.98584983e-03]
 [ 5.51377662e-02  3.04083640e+00 -2.74030870e-02]
 ...
 [ 9.95234225e+00 -5.51258405e-02  1.58243397e+00]
 [ 9.98132031e+00 -9.35225827e-03  1.57565811e+00]
 [ 1.00000000e+01  2.00115309e-02  1.56781826e+00]]
10.0 [0.02001153 1.56781826]
```

Methods

get_return_code()	Extracts the return code for the integration to enable better control if the integration fails.
[, step, relax])	Find y=y(t), set y as an initial condition, and return y.
ms(*args)	Set extra parameters for user-supplied function f.
l_value(y[, t])	Set initial conditions y(t) = y.
ator(name, **integrator_params)	Set integrator by name.
rams(*args)	Set extra parameters for user-supplied function jac.
(solout)	Set callable to be called at every successful integration step.
()	Check if integration was successful.

Scipy-Integration

solve_ivp: newest scipy integrator pack, allows to employ several integrator techniques and event tracing.

```
scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45',
t_eval=None, dense_output=False, events=None,
vectorized=False, args=None, **options)
```

***fun**: callable(t, y, …)*
The differential equations

***t_span**: 2-tuple of floats*
Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.

***y0**: array_like, shape (n,)*
Initial conditions

Scipy-Integration

solve_ivp: newest scipy integrator pack, allows to employ several integrator techniques and event tracing.

```
scipy.integrate.solve_ivp(fun, t_span, y0, method='RK45',
t_eval=None, dense_output=False, events=None,
vectorized=False, args=None, **options)
```

***fun**: callable(t, y, …)*
The differential equations

***t_span**: 2-tuple of floats*
Interval of integration (t0, tf). The solver starts with t=t0 and integrates until it reaches t=tf.

***y0**: array_like, shape (n,)*
Initial conditions

***t_eval**: Times at which to store the computed solution, must be sorted and lie within t_span.*

***dense_output**: compute a continuous solution that can be evaluated at any time between t_span*

***events**: Events to track during integration. The solver will find an accurate value of t at which*
event(t, y(t)) = 0
using a root-finding algorithm

Scipy-Integration

solve_ivp: newest s
several integrator

```
scipy.integrate.sol
t_eval=None, dense_
vectorized=False, a
```

fun: *callable(t, y, …)*
The differential equations

Int

i

t_eval: *Times at which to store the computed solution, must be sorted and lie within t_span.*

de
cont
be

Returns:	**Bunch object with the following fields defined:**
	t : *ndarray, shape (n_points,)* Time points.
	y : *ndarray, shape (n, n_points)* Values of the solution at *t*.
	sol : *OdeSolution or None* Found solution as `OdeSolution` instance; None if *dense_output* was set to False.
	t_events : *list of ndarray or None* Contains for each event type a list of arrays at which an event of that type event was detected. None if *events* was None.
	y_events : *list of ndarray or None* For each value of *t_events*, the corresponding value of the solution. None if *events* was None.

Scipy-Integration

```python
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp

# system of differential equations
def lotkavolterra(t, z, a, b, c, d):
    x, y = z
    return [a*x - b*x*y, -c*y + d*x*y]

# perform the integration
sol = solve_ivp(lotkavolterra, [0, 15], [10, 5], args=(1.5, 1, 3, 1), dense_output=True)

# densely sample the solution
t = np.linspace(0, 15, 600)
z = sol.sol(t)
```

t_span

y0

Extra args

***t_eval**: Times at which to store the computed solution, must be sorted and lie within t_span.*

***dense_output**: compute a continuous solution that can be evaluated at any time between t_span*

***events**: Events to track during integration. The solver will find an accurate value of t at which **event(t, y(t)) = 0** using a root-finding algorithm*

Scipy - Optimisation

scipy.optimize provides functions solvers for several tasks like:

- local and global optimization (e.g. minimization) algorithms;
- constrained and nonlinear least-squares:
- curve fitting;
- root finding.

Check the list of available functions here and here for examples.

Scipy - Optimisation

- Minimization: The minimize function provides a common interface to unconstrained and constrained minimization algorithms for multivariate **scalar** functions.

```
scipy.optimize.minimize(fun, x0, args=(), method=None,
jac=None, hess=None, hessp=None, bounds=None,
constraints=(), tol=None, callback=None, options=None)
```

If your functions is subjected to constraints, only few algorithm can deal with this, check the doc

***fun**: callable*
the objective function to be minimized.
*fun(x, *args) -> float, where x is an 1-D array with shape (n,)*

***x0**: ndarray, shape (n,)*
Initial guess. Array of real elements of size (n,), where n is the number of independent variables.

Check for the best method for your problem

Scipy-Optimisation

- Minimization: The
interface to uncon
algorithms for mul

```
scipy.optimize.minimiz
jac=None, hess=None, h
constraints=(), tol=No
```

If your functions is subjected to constraints, only few algorithm can deal with this, check the doc

***fun**: callable*
the objective function to
*fun(x, *args) -> float, w*
1-D array with shap

Global optimization

basinhopping(func, x0[, niter, T, stepsize, ...])	Find the global minimum of a function using the basin-hopping algorithm.
brute(func, ranges[, args, Ns, full_output, ...])	Minimize a function over a given range by brute force.
differential_evolution(func, bounds[, args, ...])	Finds the global minimum of a multivariate function.
shgo(func, bounds[, args, constraints, n, ...])	Finds the global minimum of a function using SHG optimization.
dual_annealing(func, bounds[, args, ...])	Find the global minimum of a function using Dual Annealing.

ethod
m

Scipy - Optimisation

- Curve fitting: Scipy provides a somewhat generic function (based on the Levenburg-Marquardt algorithm)through `scipy.optimize.curve_fit` to fit a chosen function f(xi,**p**) to a given data set (xi,yi), under the assumption

  yi = f(xi,**p**) with **p** the parameters of the "model".

  The algorithm tries to minimize the expression by changing the parameters **p**

$$r = \sum_{i=1}^{N} \left(y_i - f(x_i, \vec{p})\right)^2$$

SCIPY-OPTIMISATION

- Root finding: the procedure of finding all **x** values that solve the equation **f(x) = 0**

  Note that problems like **g(x)=h(x)** fall in this category as you can rewrite them as **f(x)=g(x)-h(x)=0**.

  A number of root finding tools are available in scipy's optimize module, e.g.:

  - Scalar functions (list not exhaustive see here):
    - bisect: simple and robust but slow
    - brenth/brentp: robust bracketing method faster than bisect
    - newton: Newton-Raphson (or secant method), needs derivative of **f**
  - Vector functions:

    root: most general root finder, the specific algorithm employed has to be chosen problem-wise

REMIND THAT YOU GENERALLY NEED TO PROVIDE AN INITIAL GUESS!

SciPy - Special Functions

A **special function** is a function (usually named after an early investigator of its properties) having a particular use in mathematical physics or some other branch of mathematics. Prominent examples include the **gamma function**, **hypergeometric function**, **elliptic functions**, **error function…**

The **scipy.special** module includes the implementation of nearly all special functions that are encountered in scientific research (see here for the list).

Scipy-Special Functions

A **special function** is a function (usually named after an early investigator of its properties) having a particular use in mathematical physics or some other branch of mathematics. Prominent examples include the **gamma function**, **hypergeometric function**, **elliptic functions**, **error function…**

Consider the *gamma function* which is a generalization of the factorial to which is related by $\Gamma(n) = (n-1)!$

```python
from scipy.special import gamma

print("The factorial of 3 is ", gamma(4))
print("Gamma can be evaluated for non integers:",gamma(8.7))
print("Gamma gives a complex infinity for negative integer numbers:",gamma(-2))
```

```
The factorial of 3 is  6.0
Gamma can be evaluated for non integers: 21327.693789920282
Gamma gives a complex infinity for negative integer numbers: inf
```