SciPy Optimization Overview

Introduction

The `scipy.optimize` module offers a rich collection of algorithms for solving optimization and root-finding problems. It supports scalar and multivariable function minimization, curve fitting, solving systems of equations, and handling constraints and bounds.

`scipy.optimize` Functional Categories

1. General Minimization

- `minimize()` – Multivariate function minimization.
- `minimize_scalar()` – Minimization for scalar functions.
- `basinhopping()`, `dual_annealing()` – Global optimization.
- `shgo()`, `differential_evolution()` – Advanced global methods.

2. Root Finding

- `root()` – Solve systems of equations.
- `root_scalar()` – Solve single-variable nonlinear equations.
- Supported methods: `bisect`, `brentq`, `newton`, `secant`, etc.

3. Least Squares and Curve Fitting

- `least_squares()` – Solve nonlinear least squares problems.
- `curve_fit()` – Fit a function to data.
- `leastsq()` – Legacy method (from MINPACK).

4. Constrained Optimization

- `linprog()` – Linear programming.
- `milp()` – Mixed-integer programming.
- `minimize()` – With bounds and constraints (`SLSQP`, `trust-constr`).

5. Scalar Function Minimizers

- Methods for `minimize_scalar()`: `brent`, `bounded`, `golden`.

6. Support Utilities

- `check_grad()`, `approx_fprime()` – Gradient checking.
- `line_search()` – Perform line search.
- `rosen()`, `rosen_der()`, `rosen_hess()` – Test functions.

7. Fixed-Point Solvers

- `fixed_point()` – Solve $f(x) = x$.

8. Bounds and Constraints

- `Bounds()`, `LinearConstraint()`, `NonlinearConstraint()`.

9. Deprecated Functions (use with caution)

- Legacy: `fmin()`, `fmin_bfgs()`, `fmin_cg()`, etc.
- *Now wrapped via* `minimize()` with specific `method=...`.

`scipy.optimize.minimize()` — Important Attributes

Example Usage

```
from scipy.optimize import minimize

def f(x):
    return (x - 2)**2

res = minimize(f, x0=0)
```

The result object `res` contains many useful attributes described below.

Main Result Attributes

`res.x`	**The solution**: the point where the function is minimized.
`res.fun`	Value of the function at the solution, i.e. $f(x)$.
`res.success`	**Boolean** indicating if optimization succeeded (True) or failed (False).
`res.message`	A message explaining the result or failure reason.
`res.status`	Status code (0 means success; 1-4 indicate various failures).

Optimization Process Information

`res.nit`	Number of iterations performed.
`res.nfev`	Number of function evaluations (how many times $f(x)$ was called).
`res.jac`	Gradient (Jacobian) at the solution (if available).
`res.hess_inv`	Inverse of the Hessian matrix (available for methods like BFGS).
`res.hess`	Hessian matrix at solution (for Newton-CG or trust-region methods).
`res.optimality`	First-order optimality measure (for some algorithms).

Input Echo (Sometimes Helpful)

`res.method`	Optimization method used (e.g., `'BFGS'`, `'CG'`, `'SLSQP'`).
`res.options`	Dictionary of options passed to the optimizer.

Example Output Print Statements

```
print("Minimum␣at:", res.x)
print("Minimum␣value:", res.fun)
print("Success?", res.success)
print("Message:", res.message)
print("Function␣evaluations:", res.nfev)
print("Iterations:", res.nit)
```

Constraints in `scipy.optimize.minimize()`

Constraint Syntax Overview

1. Constraint as Dictionary (Standard Way)

Equality Constraint: `fun(x) == 0`

```
constraints = ({
    'type': 'eq',
    'fun': lambda x: x[0] + x[1] - 1
},)
```

2. Constraint using `LinearConstraint` (Alternative)

Equality Constraint: $x + y = 1$

```
from scipy.optimize import LinearConstraint

linear_constraint = LinearConstraint([1, 1], lb=1, ub=1)
```

3. Inequality Constraint (Standard Way)

Inequality Constraint: `fun(x)  0`
(e.g., $x + y \leq 1 \Rightarrow 1 - (x + y) \geq 0$)

```
constraints = ({
    'type': 'ineq',
    'fun': lambda x: 1 - (x[0] + x[1])
},)
```

Notes

`'eq'`	Equality constraint $(= 0)$
`'ineq'`	Inequality constraint (≥ 0)

Interpolation using `scipy.interpolate.interp1d`

What is Interpolation?

Interpolation is a technique used to estimate intermediate values between known data points.

Simple Explanation

If you have some known data points and want to find the value at a point in between, you use interpolation.

Motivation: Why Use Interpolation?

- Estimate unknown values between known data points.
- Smooth noisy sensor or scientific data.
- Applications in image processing, audio filtering, numerical analysis, etc.

Types of Interpolation

Type	Description
Linear	Straight-line interpolation between points.
Polynomial	Curve fitting using polynomial functions.
Spline	Smooth curves using piecewise polynomials.
Lagrange	Interpolation using a polynomial through all points.
Newton	Uses finite differences to build the interpolant.

Python Example: Linear Interpolation

```python
import numpy as np
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

x = [1, 2, 3]
y = [2, 4, 6]

f = interp1d(x, y)
print("Interpolated value at x=1.5:", f(1.5))  # Output: 3.0

xnew = np.linspace(1, 3, 100)
ynew = f(xnew)

plt.plot(x, y, 'o', label='Known Points')
plt.plot(xnew, ynew, '-', label='Interpolated Line')
plt.legend()
plt.grid()
plt.title("Linear Interpolation")
plt.show()
```

`interp1d` Function Syntax

```
scipy.interpolate.interp1d(x, y, kind='linear', axis=-1,
                            copy=True, bounds_error=None,
                            fill_value=np.nan, assume_sorted=False)
```

Parameter Descriptions

Parameter	Description
`x`	Known x-values (1D array).
`y`	Corresponding y-values.
`kind`	Type of interpolation: 'linear', 'nearest', 'zero', etc.
`axis`	Axis along which interpolation is computed.
`copy`	If True, the input data is copied.
`bounds_error`	If True, error on out-of-bounds; else use fill_value.
`fill_value`	Value to return for out-of-bounds x. Can be number or 'extrapolate'.
`assume_sorted`	If True, assumes x is sorted.

interp1d Object Attributes

Attribute	Description
`f.x`	Original x-values.
`f.y`	Original y-values.
`f.kind`	Interpolation type.
`f.bounds_error`	Whether to raise error outside domain.
`f.fill_value`	Out-of-range return value.
`f.axis`	Axis used for interpolation.

Interpolation Types Summary

kind	Description
`'linear'`	Straight-line interpolation.
`'nearest'`	Nearest data point value.
`'zero'`	Step function (right continuous).
`'slinear'`	Linear spline.
`'quadratic'`	Quadratic spline.
`'cubic'`	Cubic spline.
`int`	Use spline of given order.

Bonus: Extrapolation Example

```
f2 = interp1d(x, y, kind='linear', fill_value='extrapolate')
print(f2(4))  # Extrapolates beyond known range
```

Summary

- `interp1d` is a powerful tool for 1D interpolation.
- Supports linear, nearest, spline-based methods.
- Handles out-of-bounds with `fill_value`.
- Returns a callable function object.

Understanding `scipy.optimize.curve_fit()`

What is `curve_fit()`?

`curve_fit()` is a powerful function in SciPy for nonlinear least squares curve fitting. It estimates the parameters of a user-defined model function to best fit provided data.

Function Syntax

```
from scipy.optimize import curve_fit
popt, pcov = curve_fit(model, xdata, ydata)
```

```
scipy.optimize.curve_fit(f, xdata, ydata, p0=None, sigma=None,
                         absolute_sigma=False, check_finite=True,
                         bounds=(-np.inf, np.inf), method=None,
                         jac=None, **kwargs)
```

Parameter Explanation

Parameter Descriptions

- **f:** Model function, e.g., `f(x, *params)`
- **xdata, ydata:** Independent and dependent data points
- **p0:** Initial guess of parameters (can be omitted)
- **sigma:** Standard deviation of ydata for weighting
- **absolute_sigma:** If True, treat sigma as absolute
- **check_finite:** Checks for NaN/Inf in inputs
- **bounds:** Tuple specifying lower and upper bounds
- **method:** Algorithm: `'lm'`, `'trf'`, `'dogbox'`
- **jac:** Jacobian function or approximation method
- ****kwargs:** Extra arguments for solvers

Return Values

Returns

- **popt:** Array of optimized parameter values
- **pcov:** 2D covariance matrix (uncertainty estimate)

Example Usage

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

def model(x, a, b):
    return a * x + b

x = np.array([1, 2, 3, 4, 5])
y = np.array([2.1, 3.9, 6.2, 8.0, 10.1])

popt, pcov = curve_fit(model, x, y)
perr = np.sqrt(np.diag(pcov))

print("Fitted parameters:", popt)
print("Std deviation:", perr)
```

Plotting the Fitted Curve

```
x_fit = np.linspace(1, 5, 100)
y_fit = model(x_fit, *popt)

plt.plot(x, y, 'o', label='Data')
plt.plot(x_fit, y_fit, '-', label='Fitted Curve')
plt.legend()
plt.show()
```

Related Functions

Useful Alternatives

- `np.polyfit()`: Polynomial fitting
- `least_squares()`: Advanced fitting with Jacobian
- `minimize()`: General-purpose optimization

Summary Table

curve_fit() At a Glance

Name	**Type**	**Description**
`popt`	array	Best-fit parameters
`pcov`	2D array	Covariance matrix
`np.sqrt(np.diag(pcov))`	array	Std deviation (error)
`bounds`	tuple	Parameter limits
`p0`	array	Initial guess
`sigma`	array	Data weighting
`method`	str	Optimization algorithm

When to Use Which?

- **Fit any custom function:** Use `curve_fit()`

- **Fit a polynomial:** Use `np.polyfit()`
- **Need Jacobian/advanced options:** Use `least_squares()`
- **Optimize general function:** Use `minimize()`

Numerical Integration using SciPy

1. What is Integration?

Integration is the process of computing the area under a curve. Mathematically:

$$\int_a^b f(x)\,dx$$

2. Integration Functions in `scipy.integrate`

2.1 `quad()`: Single Integral

Function Syntax

```
from scipy.integrate import quad

result, error = quad(func, a, b)
```

- `func`: The function to integrate
- `a, b`: Integration limits
- `result`: Computed integral
- `error`: Estimate of error

Example: Integrate $\sin(x)$ from 0 to π

```
from scipy.integrate import quad
import numpy as np

def f(x):
    return np.sin(x)

res, err = quad(f, 0, np.pi)
print(f"Integral of sin(x): {res:.6f} | Error: {err:.2e}")
```

2.2 `dblquad()` and `tplquad()`: Multiple Integrals

`dblquad()` computes a double integral, and `tplquad()` computes a triple integral.

```
from scipy.integrate import dblquad

def integrand(y, x):
    return x * y

result, error = dblquad(integrand, 0, 1, lambda x: 0, lambda x: 2)
print(result)
```

2.3 `fixed_quad()`: Fixed Gaussian Quadrature

```
from scipy.integrate import fixed_quad

res, _ = fixed_quad(f, 0, np.pi, n=5)
print(res)
```

2.4 `romberg()`: Romberg Integration

```
from scipy.integrate import romberg

res = romberg(f, 0, np.pi)
print(res)
```

2.5 Trapezoidal and Simpson's Rule

Used when you have discrete data points.

```
import numpy as np
from scipy.integrate import simps

x = np.linspace(0, np.pi, 100)
y = np.sin(x)

trapz_res = np.trapz(y, x)
simps_res = simps(y, x)

print(trapz_res, simps_res)
```

3. Integration Summary Table

2gray!10white

tableheader **Function**	**Purpose**	**Notes**
`quad`	Single integral	Adaptive quadrature method
`dblquad`	Double integral	Nested quadrature
`tplquad`	Triple integral	Nested quadrature
`fixed_quad`	Gaussian quadrature	Fixed sample points
`romberg`	Romberg method	High precision recursive
`np.trapz`	Trapezoidal rule	Discrete data
`simps`	Simpson's rule	Discrete data, more accurate

4. Important Parameters

- `epsabs`, `epsrel`: Absolute and relative tolerance
- `limit`: Maximum number of subintervals
- `args`: Extra arguments to the integrand function

5. Example with Extra Parameters

```python
from scipy.integrate import quad
import numpy as np

def integrand(x, a, b):
    return a * np.exp(-b * x)

result, error = quad(integrand, 0, np.inf, args=(2, 3))
print(f"Result: {result}, Error: {error}")
```

SciPy Linear Algebra Quick Reference

1. Introduction

`scipy.linalg` is a submodule for linear algebra operations like solving equations, decompositions, norms, eigenvalues, etc.

Importing

```
from scipy import linalg
import numpy as np
```

2. Core Operations

2.1 Solve Linear System $Ax = b$

```
x = linalg.solve(A, b)
```

2.2 Matrix Inverse

```
A_inv = linalg.inv(A)
```

2.3 Determinant

```
det_A = linalg.det(A)
```

2.4 Eigenvalues and Eigenvectors

```
eigvals, eigvecs = linalg.eig(A)
```

2.5 Norm (Vector/Matrix)

```
linalg.norm(A, ord=2)  # Spectral norm
```

2.6 Rank

```
np.linalg.matrix_rank(A)
```

2.7 Transpose and Conjugate Transpose

```
A.T          # Transpose
A.conj().T   # Conjugate transpose
```

3. Matrix Decompositions

LU Decomposition

```
P, L, U = linalg.lu(A)
```

QR Decomposition

```
Q, R = linalg.qr(A)
```

Cholesky Decomposition

```
L = linalg.cholesky(A)
```

Singular Value Decomposition (SVD)

```
U, s, Vh = linalg.svd(A)
```

Pseudo-Inverse

```
A_pinv = linalg.pinv(A)
```

Matrix Exponential

```
eA = linalg.expm(A)
```

Kronecker Product

```
K = np.kron(A, B)
```

Condition Number

```
cond = linalg.cond(A)
```

Lower/Upper Triangular Extraction

```
linalg.tril(A)
linalg.triu(A)
```

4. Matrix Property Checks

```
np.allclose(A, A.T)              # Symmetric?
np.allclose(A.T @ A, np.eye(n))  # Orthogonal?
```

5. Other Useful Functions

```
linalg.block_diag(A, B)      # Block diagonal
linalg.toeplitz(c)           # Toeplitz matrix
linalg.hessenberg(A)         # Hessenberg form
linalg.eigvals(A)            # Eigenvalues only
```

6. Summary Table

2gray!10white

tableheader **Topic**	**Function**
Solve $Ax = b$	`linalg.solve`
Inverse	`linalg.inv`
Determinant	`linalg.det`
Eigenvalues	`linalg.eig`
LU Decomposition	`linalg.lu`
QR Decomposition	`linalg.qr`
Cholesky	`linalg.cholesky`
SVD	`linalg.svd`
Pseudo-inverse	`linalg.pinv`
Norm	`linalg.norm`
Exponential	`linalg.expm`

7. Example: Solve $Ax = b$

```python
import numpy as np
from scipy.linalg import solve

A = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])

x = solve(A, b)
print("Solution:", x)
```

SciPy Linear Algebra: solve_triangular

Function Overview

solve_triangular() solves a linear system $Ax = b$ where A is a triangular matrix (either upper or lower). It's optimized for speed and memory when the matrix is known to be triangular.

Function Syntax

```
from scipy.linalg import solve_triangular

x = solve_triangular(A, b, lower=False, unit_diagonal=False, trans=0)
```

Parameters

2gray!10white

tableheader **Parameter**	**Description**
A	Coefficient matrix (must be triangular)
b	Right-hand side vector or matrix
lower	If True, assumes A is lower triangular
unit_diagonal	If True, assumes diagonal elements = 1
trans	0 = normal, 1 = transpose, 2 = conjugate transpose
check_finite	Skip NaN/Inf checking if False
overwrite_b	May overwrite b for efficiency

Example 1: Upper Triangular

$$\begin{bmatrix} 2 & 1 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 9 \end{bmatrix}$$

```
import numpy as np
from scipy.linalg import solve_triangular

A = np.array([[2, 1],
              [0, 3]])
b = np.array([5, 9])

x = solve_triangular(A, b)
print("x =", x)
```

Example 2: Lower Triangular

```
A = np.array([[2, 0],
              [1, 3]])
b = np.array([4, 10])

x = solve_triangular(A, b, lower=True)
print("x =", x)
```

Comparison with `solve()`

2gray!10white

tableheader **Feature**	`solve()`	`solve_triangular()`
Speed	General purpose (slower)	Faster for triangular matrices
Assumption	Any matrix	Matrix must be triangular
Use Case	Any linear system	Forward/Backward substitution

Use Cases

- Solving after LU decomposition
- Efficient forward/backward substitution
- Part of numerical methods requiring fast triangular solve

Summary Table

2gray!10white

tableheader **Option**	**Meaning**
`lower=True`	Solve assuming lower triangular matrix
`unit_diagonal=True`	Assume diagonal = 1 (saves time)
`trans=1`	Solves $A^T x = b$

Tip

If your matrix is triangular, avoid using `solve()` — use `solve_triangular()` instead for faster performance and lower memory usage.