Foundations of SciPy

Optimization, Linear Algebra, Integration & Statistics with Python

Dr. Youssef Salman

Learning outcomes. After this mini-chapter, you will be able to:

- identify key `scipy` subpackages and when to use them (`optimize`, `linalg`, `integrate`, `interpolate`, `stats`, `signal`, `spatial`);
- solve linear systems and least-squares problems with `scipy.linalg`;
- find roots and minima of functions with `scipy.optimize`;
- compute integrals (definite/ODE) with `scipy.integrate`;
- perform basic statistical analysis and distributions with `scipy.stats`.

0. Getting set up

Prerequisites. Comfortable with `NumPy` arrays and basic plotting. We will write functions and use vectorized operations whenever possible.

Install/upgrade (terminal):

```
pip install --upgrade pip
pip install scipy numpy matplotlib
```

Conventional imports:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import linalg, optimize, integrate, interpolate, stats,
    signal, spatial
```

1 A tour of SciPy

SciPy. A collection of high-level scientific computing routines built on `NumPy`, providing robust, tested algorithms: optimization, linear algebra, integration, interpolation, statistics, signal processing, spatial algorithms, etc.

What to use when (quick map).

- `scipy.linalg`: linear systems, decompositions (LU/QR/SVD), norms, eigenvalues.
- `scipy.optimize`: root finding (`root`, `brentq`), scalar/multivariate minimization (`minimize`), least-squares (`least_squares`).
- `scipy.integrate`: quadrature (`quad`), ODE solvers (`solve_ivp`).

- `scipy.interpolate`: 1D/ND interpolation (`interp1d`, splines).
- `scipy.stats`: distributions, random variates, hypothesis tests, descriptive stats.
- `scipy.signal`: filtering, convolution, FFT-based tools, spectral analysis.
- `scipy.spatial`: KD-trees, distance matrices, Delaunay/Voronoi.

Rule of thumb. Start with a well-posed mathematical formulation (objective, constraints, model assumptions). Then choose the SciPy routine that matches the math. Avoid "try-all-optimizers" without understanding the problem structure.

2 Linear algebra with `scipy.linalg`

2.1 Solving linear systems

```
# Solve Ax = b and check residual
rng = np.random.default_rng(0)
A = rng.normal(size=(4,4))
b = rng.normal(size=4)

x = linalg.solve(A, b)
res = np.linalg.norm(A @ x - b)
condA = np.linalg.cond(A)

print("x =", x)
print("||Ax - b||_2 =", res)
print("cond(A) =", condA)
```

Numerical note. If $\mathrm{cond}(A)$ is large, small perturbations in A or b can cause large changes in x. Prefer `linalg.lstsq` or regularization if the system is ill-conditioned.

2.2 Least squares and pseudoinverse

```
# Overdetermined system: minimize ||Ax - b||_2
m, n = 50, 3
X = np.c_[np.ones(m), rng.uniform(-1,1,size=m), rng.uniform(0,2,size=m)]
true_beta = np.array([2.0, -1.0, 0.5])
y = X @ true_beta + rng.normal(scale=0.2, size=m)

beta_ls, resid, rank, svals = linalg.lstsq(X, y)  # returns least-squares solution
beta_pinv = linalg.pinv(X) @ y                    # via Moore-Penrose pseudoinverse

print("beta (LS)   =", beta_ls)
```

```
print("residual ss =", resid)
print("rank, svals =", rank, svals[:3])
```

Exercise (conditioning). Construct a Vandermonde matrix with degrees 0–k using equispaced points on $[0, 1]$. Measure $\mathrm{cond}(X)$ as k grows and observe its effect on least-squares fitting.

2.3 Decompositions at a glance

```
# LU, QR, SVD (brief demo)
P, L, U = linalg.lu(A)
Q, R = linalg.qr(A)
U_svd, s, Vt = linalg.svd(A)

print("LU shapes:", L.shape, U.shape)
print("QR diag(R):", np.diag(R))
print("SVD singular values:", s)
```

Why decompositions? They expose structure (rank/conditioning) and enable stable algorithms: QR for least squares, SVD for low-rank approximation/regularization, LU for solving multiple right-hand sides efficiently.

3 Root finding and optimization with `scipy.optimize`

3.1 Root finding (scalars and systems)

```
# Scalar root on [a,b] using brentq (requires sign change)
f = lambda x: np.cos(x) - x
root = optimize.brentq(f, 0, 1)    # guaranteed if f(a)*f(b) < 0
print("root ~", root)

# Multivariate roots with Jacobian-free methods
g = lambda v: np.array([v[0]**2 + v[1] - 1, v[0] + v[1]**2 - 1])
sol = optimize.root(g, x0=np.array([0.5, 0.5]))
print(sol.x, sol.success)
```

Tip. Bracketing methods (`bisect`, `brentq`) are robust for 1D problems. For multivariate roots, good initial guesses matter; scale your variables to similar magnitudes.

3.2 Unconstrained and constrained minimization

```
# Unconstrained: minimize Rosenbrock function
```

```
rosen = optimize.rosen
x0 = np.array([-1.2, 1.0])
res = optimize.minimize(rosen, x0, method="BFGS")
print(res.x, res.nit, res.success)

# With bounds (e.g., box constraints) using L-BFGS-B
quad = lambda x: (x[0]-1)**2 + (x[1]+2)**2
bounds = [(-1, 2), (-3, 0)]
res_b = optimize.minimize(quad, x0=[0,0], bounds=bounds, method="L-BFGS-B")
print(res_b.x, res_b.success)

# Nonlinear constraints with SLSQP: minimize x^2+y^2 s.t. x+y=1, x>=0, y>=0
fun = lambda v: v[0]**2 + v[1]**2
cons = [{'type':'eq', 'fun': lambda v: v[0]+v[1]-1}]
bnds = [(0, None), (0, None)]
res_c = optimize.minimize(fun, x0=[0.2,0.8], bounds=bnds, constraints=cons, method="SLSQP")
print(res_c.x, res_c.success)
```

Good practice. Scale variables so typical magnitudes are near 1; provide gradients if available using `jac` for speed/accuracy; set tolerances via `options={'gtol':..., 'ftol':...}`.

3.3 Nonlinear least squares

```
# Fit y = a * exp(bx) + c  to noisy data
rng = np.random.default_rng(1)
x = np.linspace(0, 3, 80)
a_true, b_true, c_true = 2.0, 1.3, 0.5
y = a_true*np.exp(b_true*x) + c_true + rng.normal(0, 0.3, size=x.size)

def resid(theta):
    a, b, c = theta
    return a*np.exp(b*x) - (y - c)

theta0 = np.array([1.0, 1.0, 0.0])
res = optimize.least_squares(resid, theta0, method="trf")  # trust-region reflective
print("theta_hat =", res.x)

# Plot fit
xh = np.linspace(0, 3, 400)
yh = res.x[0]*np.exp(res.x[1]*xh) + res.x[2]
plt.scatter(x, y, s=15, label="data")
plt.plot(xh, yh, label="fit", linewidth=2)
```

```
plt.legend(); plt.title("Nonlinear least squares"); plt.show()
```

Exercise (robustness). Replace the Gaussian noise by a few large outliers. Compare `method="trf"` with loss functions `loss="soft_l1"` and `loss="huber"` in `least_squares`. How do the estimates change?

4 Numerical integration with `scipy.integrate`

4.1 Quadrature (definite integrals)

```
from math import pi
f = lambda t: np.sin(t**2)            # Fresnel-like integrand
I, err = integrate.quad(f, 0, np.sqrt(pi))
print("Integral ~", I, " (estimated abs error:", err, ")")

# Multiple integrals
g = lambda y, x: np.exp(-(x**2 + y**2))
I2, err2 = integrate.nquad(g, [[-2,2], [-2,2]])  # order is (y,x)
print("Double integral ~", I2)
```

Tip. If the integrand is oscillatory or improper, split the range into subintervals around problematic points and use `points=...` or variable substitution to improve accuracy.

4.2 Initial value problems (ODEs)

```
# Van der Pol oscillator (mild stiffness when mu large)
mu = 3.0
def vdp(t, z):
    x, y = z
    return [y, mu*(1 - x**2)*y - x]

sol = integrate.solve_ivp(vdp, t_span=(0, 20), y0=[2, 0],
                          method="RK45", rtol=1e-6, atol=1e-9,
                              dense_output=True)

t = np.linspace(0, 20, 1000)
x = sol.sol(t)[0]
plt.plot(t, x); plt.title("Van der Pol: x(t)"); plt.xlabel("t"); plt.
    ylabel("x"); plt.show()
```

Solver choice. Use `RK45` or `DOP853` for non-stiff problems, and `BDF` or `Radau` for stiff systems. Always monitor `sol.message` and try different tolerances if the solver struggles.

5 Interpolation with `scipy.interpolate`

5.1 1D interpolation

```python
# Sample a noisy function and recover intermediate values
f = lambda x: np.sin(2*np.pi*x) * np.exp(-3*x)
x = np.linspace(0, 1, 12)
y = f(x) + 0.05*np.random.default_rng(0).normal(size=x.size)

from scipy.interpolate import interp1d, UnivariateSpline

lin = interp1d(x, y, kind="linear")
cub = interp1d(x, y, kind="cubic")

xs = np.linspace(0, 1, 400)
plt.plot(x, y, "o", label="data")
plt.plot(xs, lin(xs), label="linear")
plt.plot(xs, cub(xs), label="cubic")
plt.legend(); plt.title("1D interpolation"); plt.show()

# Smoothing spline (regularization via s)
spl = UnivariateSpline(x, y, s=0.01)
plt.plot(xs, spl(xs), label="smoothing spline")
plt.legend(); plt.show()
```

Over/underfitting. High-order interpolation may oscillate between points (Runge phenomenon). Smoothing splines introduce a penalty controlled by `s` to balance fidelity and smoothness.

6 Statistics with `scipy.stats`

6.1 Descriptives, distributions, hypothesis tests

```python
data = stats.norm(loc=3.0, scale=1.2).rvs(size=500, random_state=0)

# Summary stats
print(np.mean(data), np.std(data, ddof=1), stats.skew(data), stats.
    kurtosis(data))

# Distribution fitting (normal MLE)
mu_hat, sigma_hat = stats.norm.fit(data)
print("fit:", mu_hat, sigma_hat)

# One-sample t-test: H0: mean = 3
print(stats.ttest_1samp(data, popmean=3.0))

```

```
# Two-sample t-test with unequal variances
x = stats.norm(0, 1).rvs(size=120, random_state=1)
y = stats.norm(0.4, 1.1).rvs(size=100, random_state=2)
print(stats.ttest_ind(x, y, equal_var=False))

# Normality test (Shapiro-Wilk; small n recommended)
print(stats.shapiro(x[:50]))
```

Exercise (p-values and effect size). Simulate two groups with a small mean difference and similar variance. Compute the Welch t-test and Cohen's d. How does increasing the sample size affect power and p-values?

Caution. Always check assumptions (independence, distributional form, variance homogeneity). For non-normal data, consider nonparametric tests such as `mannwhitneyu` or `wilcoxon`.

7 Signal processing with `scipy.signal`

7.1 Filtering and convolution

```
from scipy import signal

# Create a noisy sine wave
t = np.linspace(0, 1, 500)
sig = np.sin(2*np.pi*5*t) + 0.5*np.random.default_rng(0).normal(size=t
    .size)

# Design a low-pass Butterworth filter
b, a = signal.butter(N=4, Wn=0.1)
filtered = signal.filtfilt(b, a, sig)

plt.plot(t, sig, alpha=0.5, label="noisy signal")
plt.plot(t, filtered, label="filtered")
plt.legend(); plt.title("Butterworth low-pass filtering")
plt.show()
```

Tip. Filtering is crucial in preprocessing (e.g. denoising biomedical or audio signals). Always inspect phase distortions: `filtfilt` applies forward–backward filtering to cancel them.

7.2 Spectral analysis

```
# Power spectral density with Welch's method
f, Pxx = signal.welch(sig, fs=500)
```

```
plt.semilogy(f, Pxx)
plt.xlabel("Frequency [Hz]")
plt.ylabel("Power spectral density")
plt.title("Welch PSD estimate")
plt.show()
```

Exercise (signal decomposition). Generate a signal composed of three sinusoids (frequencies 5, 15, 30 Hz) plus noise. Apply Welch's PSD and identify the peaks. What happens if you reduce the segment length (`nperseg`)?

8 Spatial algorithms with `scipy.spatial`

8.1 Distances and nearest neighbors

```
from scipy.spatial import distance, KDTree

X = rng.random((10, 2))   # 10 points in 2D
dmat = distance.squareform(distance.pdist(X))
print("Distance matrix shape:", dmat.shape)

tree = KDTree(X)
dist, idx = tree.query([0.5, 0.5], k=3)  # 3 nearest neighbors
print("Distances:", dist)
print("Indices:", idx)
```

8.2 Delaunay triangulation and Voronoi diagrams

```
points = rng.random((20,2))

tri = spatial.Delaunay(points)
plt.triplot(points[:,0], points[:,1], tri.simplices)
plt.plot(points[:,0], points[:,1], "o")
plt.title("Delaunay triangulation")
plt.show()

vor = spatial.Voronoi(points)
spatial.voronoi_plot_2d(vor)
plt.title("Voronoi diagram")
plt.show()
```

Applications. KD-trees accelerate nearest-neighbor search (used in ML, clustering). Voronoi diagrams model regions of influence (e.g. cell coverage, territories).

9 Summary and practical advice

- `scipy` builds on `numpy`, offering ready-to-use algorithms for optimization, integration, interpolation, statistics, signal processing, and spatial analysis.
- Always start from a mathematical formulation, then match it to the relevant SciPy function.
- Inspect solver outputs (`success`, error estimates, messages) rather than relying blindly on default settings.
- Visualization (with Matplotlib) is essential to validate numerical results.

Final exercise. Model the decay of a radioactive isotope with exponential law. Generate synthetic noisy data, fit the decay constant using nonlinear least squares, and visualize the fit along with residuals.

Next step. The following module introduces `pandas` for data manipulation and explores how to combine it with NumPy/SciPy for end-to-end analysis pipelines.