Chapter 10

Introduction to NumPy and SciPy

Installation of NumPy & SciPy

- Go to Python’s subdirectory with **pip**, eg, …/Python/Scripts,

- With the internet being on, type the following commands in the prompt terminal:

pip install numpy

pip install scipy

- Documentation: **http://docs.scipy.org**
-
- In Anaconda’s environment,

conda install numpy

conda install scipy

NumPy Arrays

- **NumPy** is the fundamental Python package for scientific computing.

- It contains among other things:

  * a powerful N-dimensional array object
  * element-by-element operations (broadcasting)
  * tools for integrating C/C++ and Fortran code
  * core mathematical operations like linear algebra, Fourier transform, and random number capabilities

- **NumPy** enables users to overcome the inefficiency of the Python lists by providing a data storage object, **ndarray**.

- **ndarray** is similar to lists, but only the same type of element can be stored in each column. Despite the limitation, **ndarray** wins when it comes to operation times, as the operations are sped up significantly.

```python
import numpy as np

# Create an array with 10^7 elements.
arr = np.arange(1e7)

# Converting ndarray to list
larr = arr.tolist()

# Lists cannot by default broadcast, so a function is
# coded to emulate what an ndarray can do.
def list_times(alist, scalar):
    for i, val in enumerate(alist):
        alist[i] = val * scalar
    return alist

# Using IPython's magic timeit command
timeit arr * 1.1
>>> 1 loops, best of 3: 76.9 ms per loop

timeit list_times(larr, 1.1)
>>> 1 loops, best of 3: 2.03 s per loop
```

- **ndarray** is 25 faster than the Python loop in this example.

- If we need linear algebra operations, we can use **matrix**, which does not use the default broadcasting from **ndarray**.

- **matrix** objects can and only will be 2-dim.

```
import numpy as np

# Creating a 3D numpy array
arr = np.zeros((3,3,3))

# Trying to convert array to a matrix, which won’t work
mat = np.matrix(arr)

#"ValueError: shape too large to be a matrix."
```

Array Creation and Data Typing

```
# First we create a list and then
# wrap it with the np.array() function.
alist = [1, 2, 3]
arr = np.array(alist)

# Creating an array of zeros with five elements
arr = np.zeros(5)

# What if we want to create an array from 0 to 100?
arr = np.arange(100)

# Or 10 to 100?
arr = np.arange(10,100)

# If you want 100 steps from 0 to 1...
arr = np.linspace(0, 1, 100)
```

```python
# Or if you want to generate an array from 1 to 10
# in log10 space in 100 steps...
arr = np.logspace(0, 1, 100, base=10.0)

# Creating a 5x5 array of zeros (an image)
image = np.zeros((5,5))

# Creating a 5x5x5 cube of 1's. The astype() method
# sets the array with integer elements.
cube = np.zeros((5,5,5)).astype(int) + 1

# Or even simpler with 16-bit floating-point precision...
cube = np.ones((5, 5, 5)).astype(np.float16)
```

- If you are working with 32-/64-bit Python, then your elements in the arrays will default to 32-/64-bit precision.

- You can specify the size when creating arrays by setting the data type parameter (**dtype**) to **int**, **numpy.float16**, **numpy.float32**, or **numpy.float64**.

```
# Array of zero integers
arr = np.zeros(2, dtype=int)

# Array of zero floats
arr = np.zeros(2, dtype=np.float32)
```

- Once we have created arrays, we can reshape them:

```
# Creating an array with elements from 0 to 999
arr1d = np.arange(1000)

# Now reshaping the array to a 10x10x10 3D array
arr3d = arr1d.reshape((10,10,10))

# The reshape command can alternatively be called as
arr3d = np.reshape(arr1s, (10, 10, 10))

# Inversely, we can flatten arrays
arr4d = np.zeros((10, 10, 10, 10))
arr1d = arr4d.ravel()
print arr1d.shape
            (10000,)
```

Record Arrays

- Arrays can store more complex data structures where columns are composed of different data types.

```
# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))
toadd = [(1,2.,'Hello'),(2,3.,"World")]
recarr[:] = toadd
```

- **dtype** defines the types designated for the 3 columns, **i4**: 32-bit integer, **f4**: 32-bit float, **a10**: a 10-character string.

- There is a global function **zip** that will create a list of tuples like we see above for the toadd object:

```
# Creating an array of zeros and defining column types
recarr = np.zeros((2,), dtype=('i4,f4,a10'))

# Now creating the columns to put  in the recarray
```

```python
col1 = np.arange(2) + 1
col2 = np.arange(2, dtype=np.float32)
col3 = ['Hello', 'World']

# Here we create a list of tuples that is
# identical to the previous toadd list.
toadd = zip(col1, col2, col3)

# Assigning values to recarr
recarr[:] = toadd

# Assigning names to each column, which
# are now by default called 'f0', 'f1', and 'f2'.

recarr.dtype.names = ('Integers' , 'Floats', 'Strings')

# If we want to access one of the columns by its name,
# we can do the following.

recarr('Integers')
# array([1, 2], dtype=int32)
```

Indexing and Slicing

- Python index lists begin at 0 and the NumPy arrays follow:

```
alist=[[1,2],[3,4]]

# To return the (0,1) element we must index as below
alist[0][1]
```

- In NumPy, indexing follows a more convenient syntax.

```
# Converting the list defined above into an array
arr = np.array(alist)

# To return the (0,1) element we use ...
arr[0,1]

# Now to access the last column, we simply use ...
arr[:,1]
```

```
# Accessing the columns is achieved in the same way,
# which is the bottom row.
arr[1,:]
```

- If there are more complex indexing schemes required, the most commonly used type is **numpy.where():**

```
# Creating an array
arr = np.arange(5)

# Creating the index array
index = np.where(arr > 2)
print(index)
      (array([3, 4]),)

# Creating the desired array
new_arr = arr[index]
```

- If you want to remove specific indices, use **numpy.delete()**

```
# We use the previous array
new_arr = np.delete(arr, index)
```

- Instead of **numpy.where**, we can use a simple boolean array to return specific elements:

```
index = arr > 2
print(index)
      [False False False True True]
new_arr = arr[index]
```

- If speed is important, the boolean indexing is faster for a large number of elements.

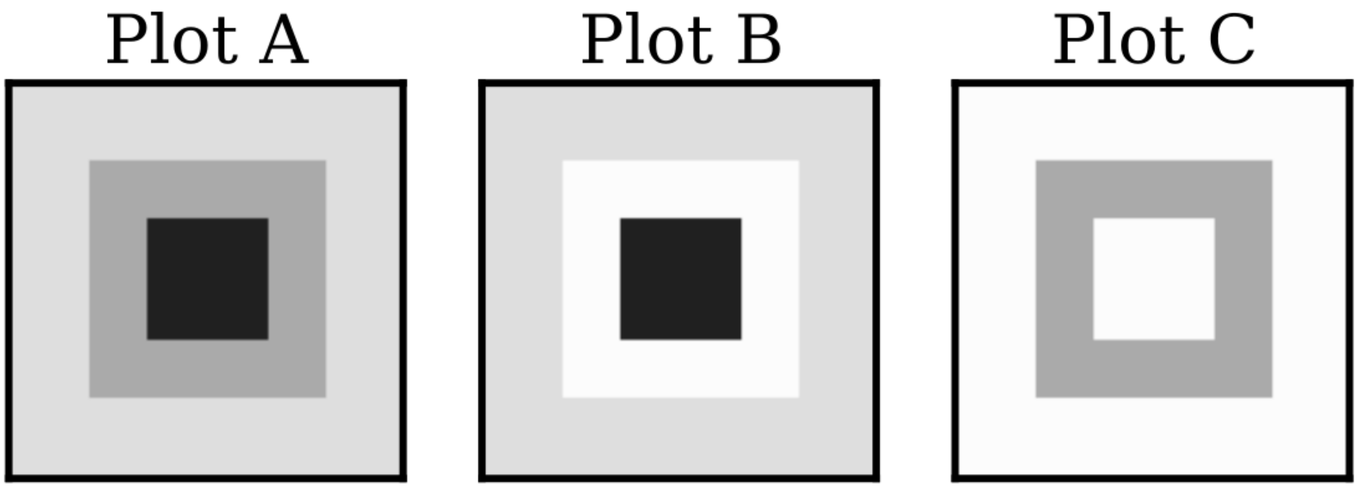
Boolean Statements and NumPy Arrays

- Boolean statements are commonly used in combination with the **and/or** operator.

- When using **NumPy** arrays, one can only use **&** and **|** as this allows fast comparisons of boolean values:

```
# Creating an image
img1 = np.zeros((20, 20)) + 3
img1[4:-4, 4:-4] = 6
img1[7:-7, 7:-7] = 9
# See Plot A

# Let's filter out all values > 2 and < 6.
index1 = img1 > 2
index2 = img1 < 6
compound_index = index1 & index2
```

```
# The compound statement can alternatively be as
compound_index = (img1 > 3) & (img1 < 7)
img2 = np.copy(img1)
img2[compound_index] = 0
# See Plot B.

# Making the boolean arrays even more complex
index3 = img1 == 9
index4 = (index1 & index2) | index3
img3 = np.copy(img1)
img3[index4] = 0
# See Plot C.
```



- In a special case where you only want to operate on specific elements in an array:

```
import numpy as np
import numpy.random as rand

# Creating a 100-element array with random values
# from a standard normal distribution or, in other
# words, a Gaussian distribution.
# The sigma is 1 and the mean is 0.
a = rand.randn(100)

# Here we generate an index for filtering
# out undesired elements.
index = a > 0.2
b = a[index]

# We execute some operation on the desired elements.
b = b ** 2 – 2

# Then we put the modified elements back into the
# original array.
a[index] = b
```

Read and Write

- For text files:

```
# Opening the text file only allowing reading
f = open('somefile.txt', 'r')

# creates a list where each element  is one line
alist = f.readlines()

f.close()

# write the data with the 'w' option to a file
f = open('newtextfile.txt', 'w')

# Writing data to file
f.writelines(newdata)

f.close()
```

- If the file is large, then accessing or modulating the data will be cumbersome and slow. Getting the data directly into a **numpy.ndarray** would be the best option.

- If the data is structured with rows & columns, then **loadtxt** will work very well as long as all the data is of a similar type, i.e., integers or floats.

- We can save the data through **numpy.savetxt** as easily and quickly as with **numpy.readtxt**:

```
import numpy as np

arr = np.loadtxt('somefile.txt')

np.savetxt('somenewfile.txt')
```

- If each column is different in formatting, **loadtxt** can still read the data, but the column types need to be predefined:

```
# example.txt file looks like the following
#
# XR21 32.789 1
# XR22 33.091 2

table = np.loadtxt('example.txt',
                   dtype='names': ('ID', 'Result', 'Type'),
                   'formats': ('S4', 'f4', 'i2'))

# array([('XR21', 32.78900146484375, 1),
#        ('XR22', 33.090999603271484, 2)],
# dtype=[('ID', '|S4'), ('Result', '<f4'), ('Type', '<i2')])
```

Binary Files

- Binary files are harder to deal with, as formatting, readability, portability are trickier. But they have 2 notable advantages: file size and read/write speeds. This is especially important when working with big data.

- Files can be accessed in binary format using **numpy.save** and **numpy.load**:

```
import numpy as np

# Creating a large array
data = np.empty((1000, 1000))

# Saving the array with numpy.save
np.save('test.npy', data)

# For large files use numpy.savez. It compresses files.
np.savez('test.npz', data)
```

```python
# Loading the data array
newdata = np.load('test.npy')
```

- **numpy.save** and **numpy.savez** have no issues saving **numpy.recarray** objects. Hence, working with complex and structured arrays is no issue if portability beyond the Python environment is not of concern.

Math

- If you try to use **math.cos** (from the **math** module) on a **NumPy** array, it will not work, as the **math** functions are meant to operate on elements and not on lists or arrays. Hence, **NumPy** comes with its own set of math tools.

- When transposing or a dot multiplication are needed, you can use the built-in **numpy.dot** and **numpy.traspose** to do such operations.

- Compare advantages/disadvantages between **numpy.array** and **numpy.matrix**

$$\begin{aligned} 3x + 6y - 5z &= 12 \\ x - 3y + 2z &= -2 \\ 5x - y + 4z &= 10 \end{aligned} \Rightarrow \begin{bmatrix} 3 & 6 & -5 \\ 1 & -3 & 2 \\ 5 & -1 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 12 \\ -2 \\ 10 \end{bmatrix}$$

$$\mathbf{AX} = \mathbf{B} \Rightarrow \mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$$

```python
import numpy as np

# Defining the matrices
A = np.matrix([[3, 6, -5],
               [1, -3, 2],
               [5, -1, 4]])

B = np.matrix([[12],
               [ -2],
               [10]])

# Solving for the variables, where we invert A
X = A ** (-1) * B
print(X)

# matrix([[ 1.75],
#         [ 1.75],
#         [ 0.75]])
```

- Do the same operations without using **numpy.matrix**:

```
import numpy as np

a = np.array([[3, 6, -5],
              [1, -3, 2],
              [5, -1, 4]])

# Defining the array
b = np.array([12, -2, 10])

# Solving for the variables, where we invert A
x = np.linalg.inv(a).dot(b)
print(x)

# array([ 1.75, 1.75, 0.75])
```

- The **numpy.matrix** method is the simplest. However, the **numpy.array** method is the most practical and faster.

SciPy

- **SciPy** is a package that utilizes **NumPy** arrays and manipulations to take on standard problems, eg, integration, determining a function's maxima or minima, finding eigenvectors for large sparse matrices, etc.

Optimization and Minimization

- The optimization package in **SciPy** allows us to solve minimization problems easily and quickly.

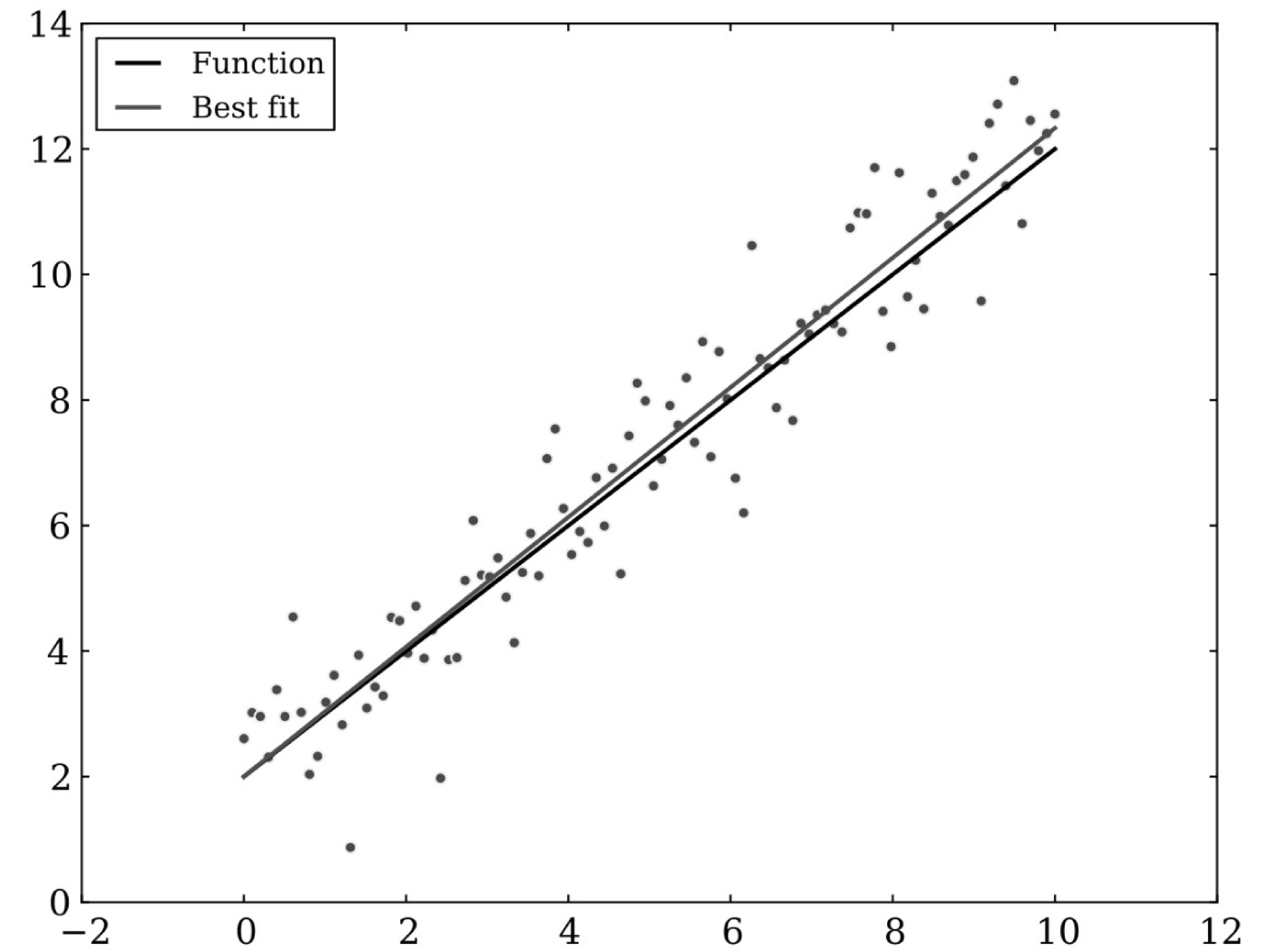
- Some classic examples are performing linear regression, finding a function's minimum/maximum values, determining the root of a function, finding where 2 functions intersect.

- To fit data with a linear regression, we will use **curve_fit**, which is a χ^2-based method.

- we generate data from a known function with noise, and then fit the noisy data with **curve_fit**. The function we will model in the example is a simple linear equation, $f(x)=ax+b$:

```
import numpy as np
from scipy.optimize import curve_fit
```

```python
# Creating a function to model and create data
def func(x, a, b):
    return a * x + b

# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 2)

# Adding noise to the data
yn = y + 0.9 * np.random.normal(size=len(x))

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)

# popt returns the best fit values for parameters of
# the given model (func).

print(popt)
```

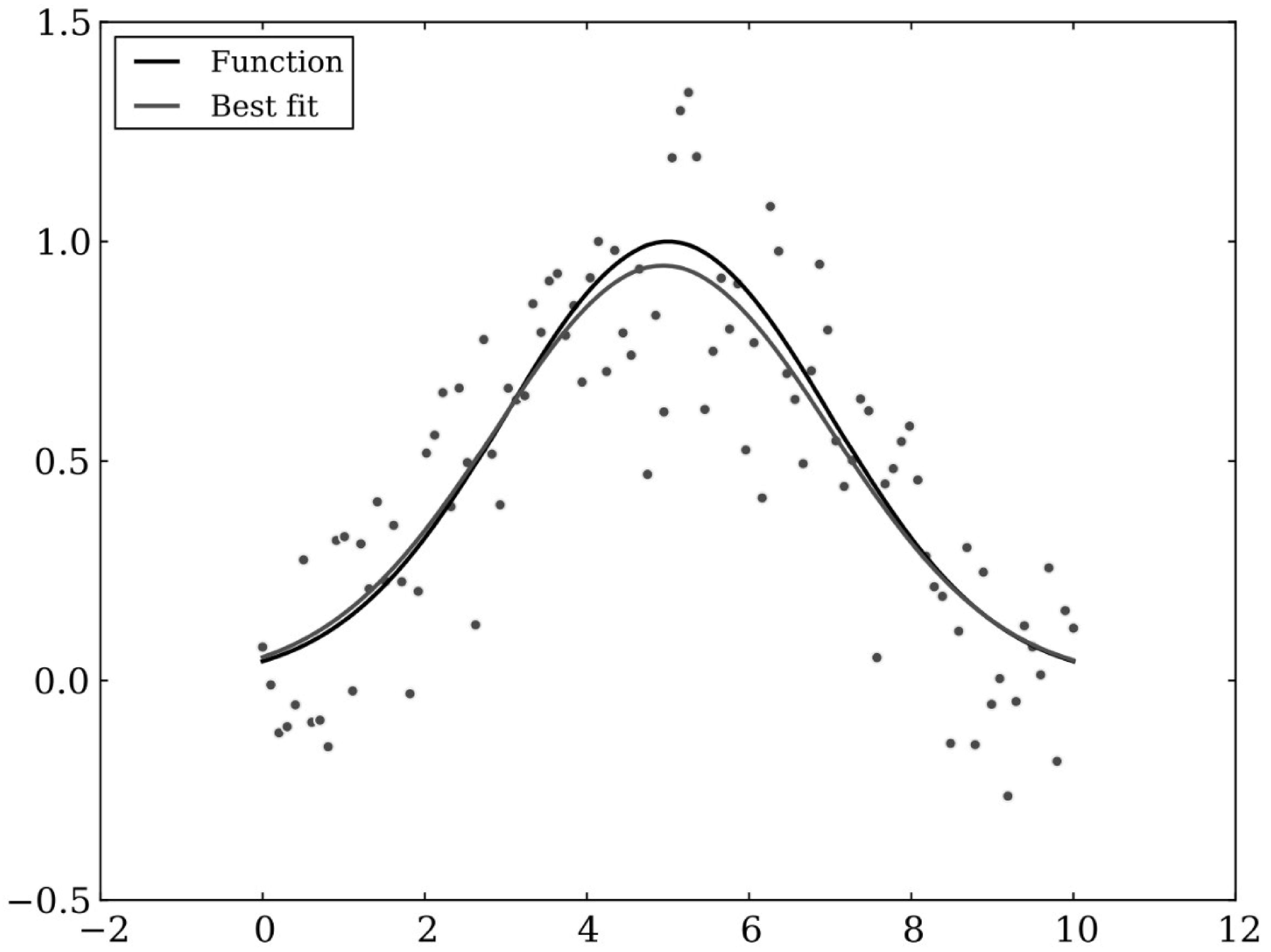


Function
Best fit
0
2
4
6
8
10
12
14
−2
0
2
4
6
8
10
12

- Do a least-squares fit to a Gaussian profile, a non-linear function:

$$f(x) = a\, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad \Leftarrow \quad \begin{array}{l} a : \text{scalar} \quad \mu : \text{mean} \\ \sigma : \text{standard deviation} \end{array}$$

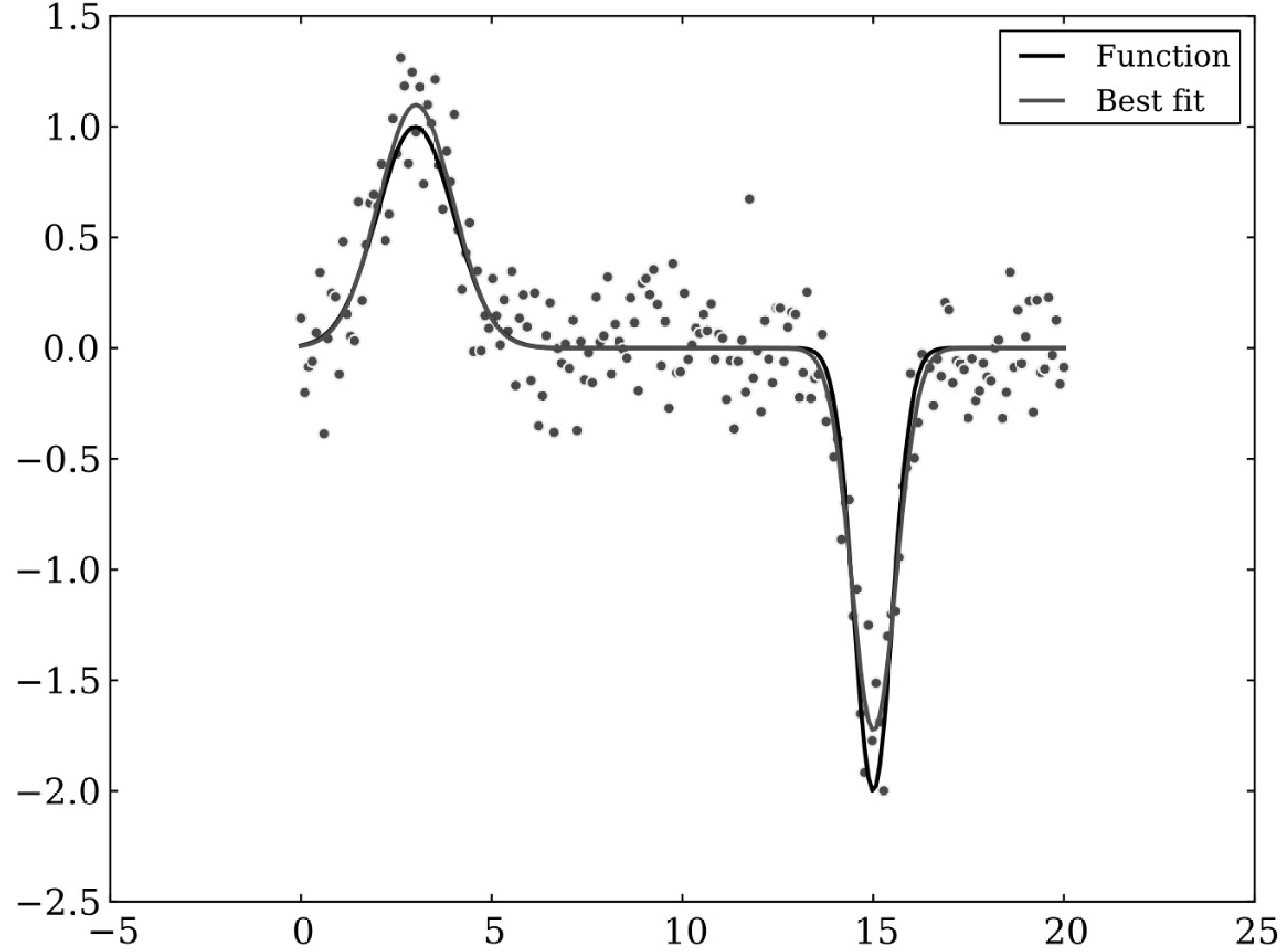
```
# Creating a function to model and create data
def func(x, a, b, c):
    return a*np.exp(-(x-b)**2/(2*c**2))

# Generating clean data
x = np.linspace(0, 10, 100)
y = func(x, 1, 5, 2)

# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))

# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn)

# popt returns the best-fit values.
print(popt)
```



Function
Best fit
1.5
1.0
0.5
0.0
−0.5
−2
0
2
4
6
8
10
12

- fit a one-dim dataset with multiple Gaussian profiles:

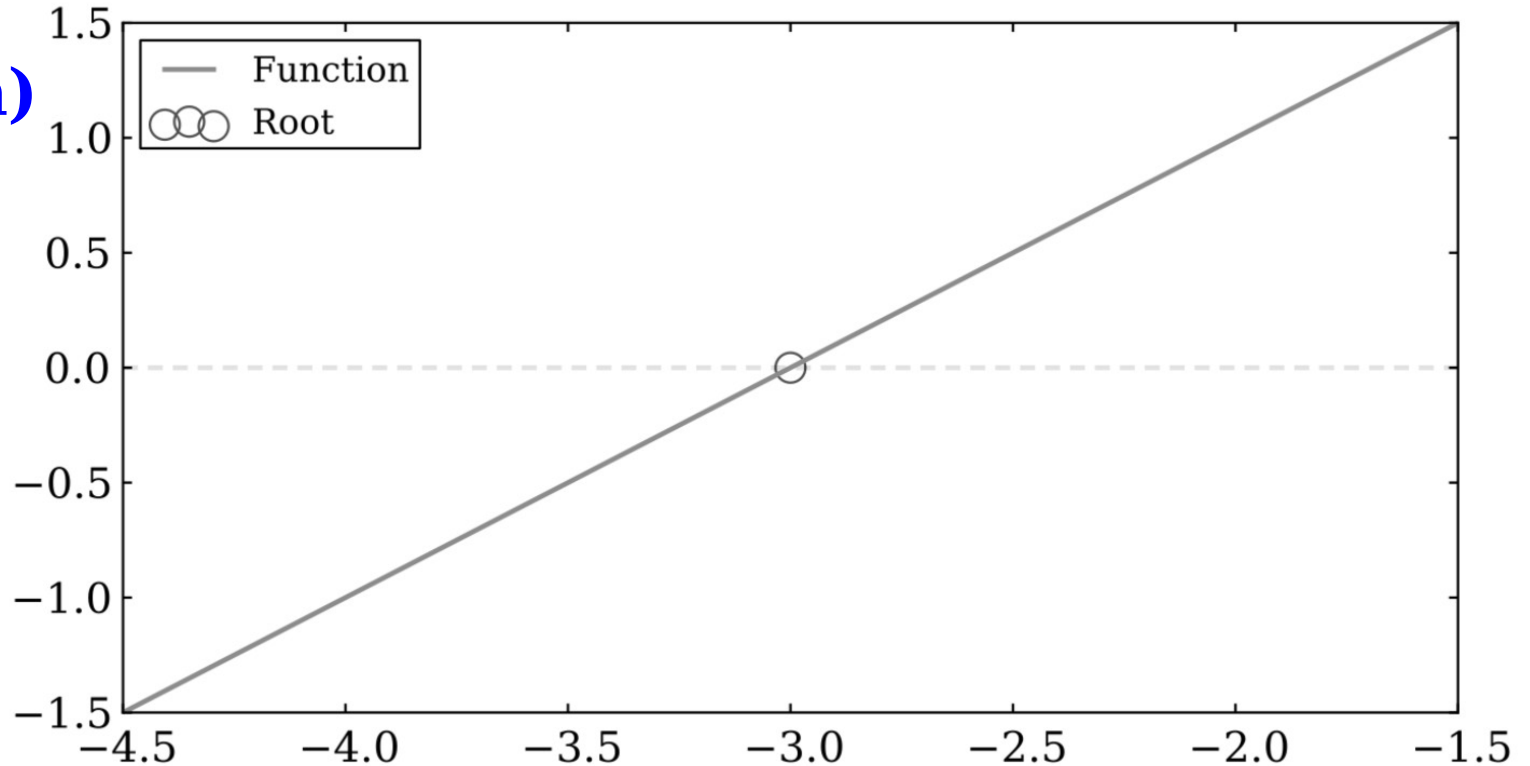
```
# Two-Gaussian model
def func(x, a0, b0, c0, a1, b1,c1):
    return a0 * np.exp(-(x - b0) ** 2/(2 * c0 ** 2))\
           + a1 * np.exp(-(x - b1) ** 2/(2 * c1 ** 2))

# Generating clean data
x = np.linspace(0, 20, 200)
y = func(x, 1, 3, 1, -2, 15, 0.5)

# Adding noise to the data
yn = y + 0.2 * np.random.normal(size=len(x))

# Since we are fitting a more complex function,
# need better guesses to get a better fitting.

guesses = [1, 3, 1, 1, 15, 1]
# Executing curve_fit on noisy data
popt, pcov = curve_fit(func, x, yn, p0=guesses)
```


Function
Best fit
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−5
0
5
10
15
20
25

Solutions to Functions

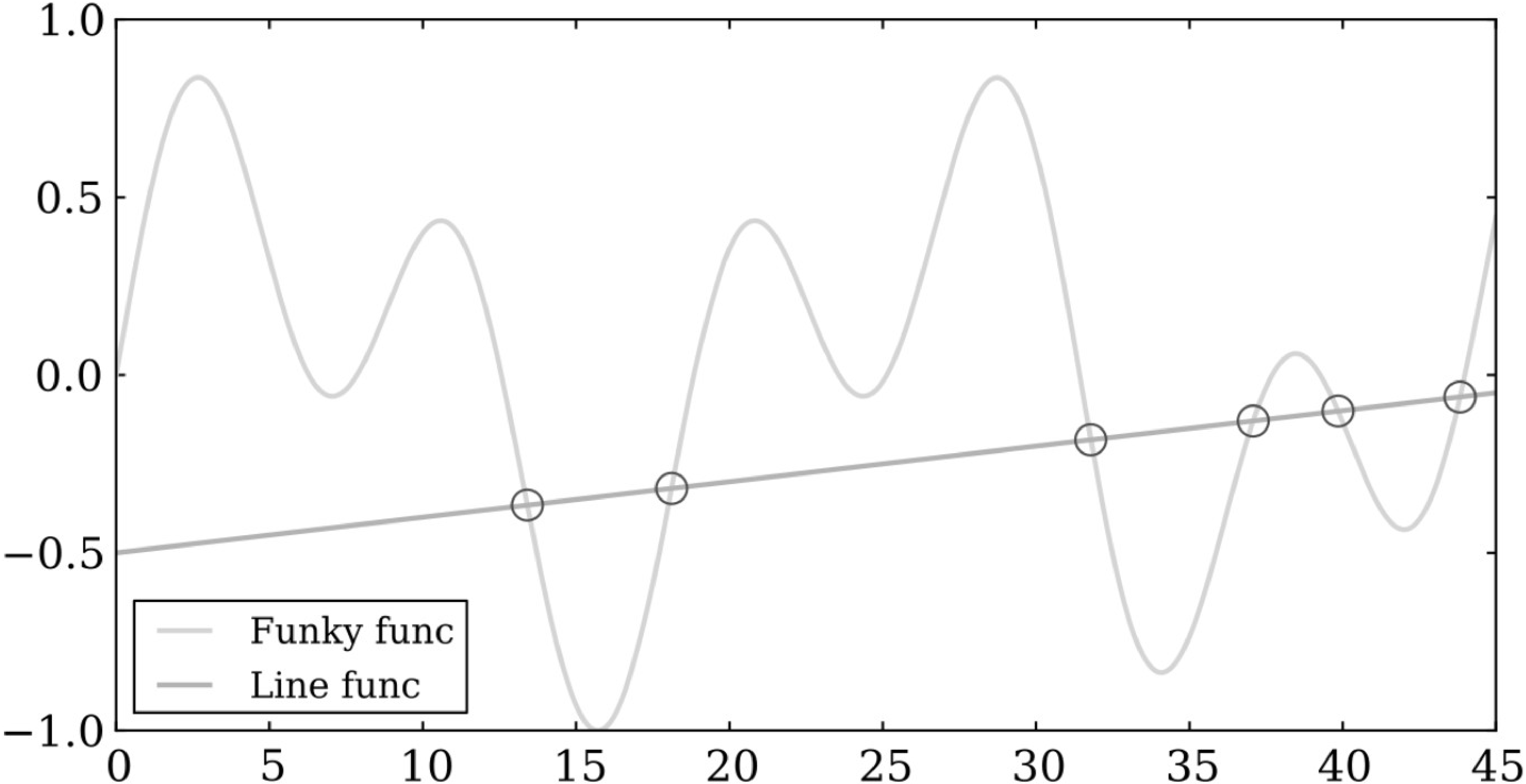
- Let's start by solving for the root of an equation

```
from scipy.optimize import fsolve
import numpy as np

line = lambda x : x + 3

solution = fsolve(line, -2)

print(solution)
```



- Find the intersection points between 2 equations

```
from scipy.optimize import fsolve
import numpy as np

# Defining function to simplify intersection solution
def findIntersection(func1, func2, x0):
    return fsolve(lambda x : func1(x) - func2(x), x0)

# Defining functions that will intersect
funky = lambda x : np.cos(x / 5) * np.sin(x / 2)
line = lambda x : 0.01 * x - 0.5

# Define range and get solutions on intersection points
x = np.linspace(0,45,10000)
result = findIntersection(funky, line, [15, 20, 30, 35,
40, 45])

# Printing out results for x and y
print(result, line(result))
```



1.0
0.5
0.0
−0.5
−1.0
0
5
10
15
20
25
30
35
40
45
Funky func
Line func

Interpolation

- Given a set of sample data, obtaining the intermediate values between the points is useful to understand and predict what the data will do in the non-sampled domain.

- Univariate interpolation is used when the sampled data is led by one independent variable, multivariate interpolation assumes there is more than one independent variable.

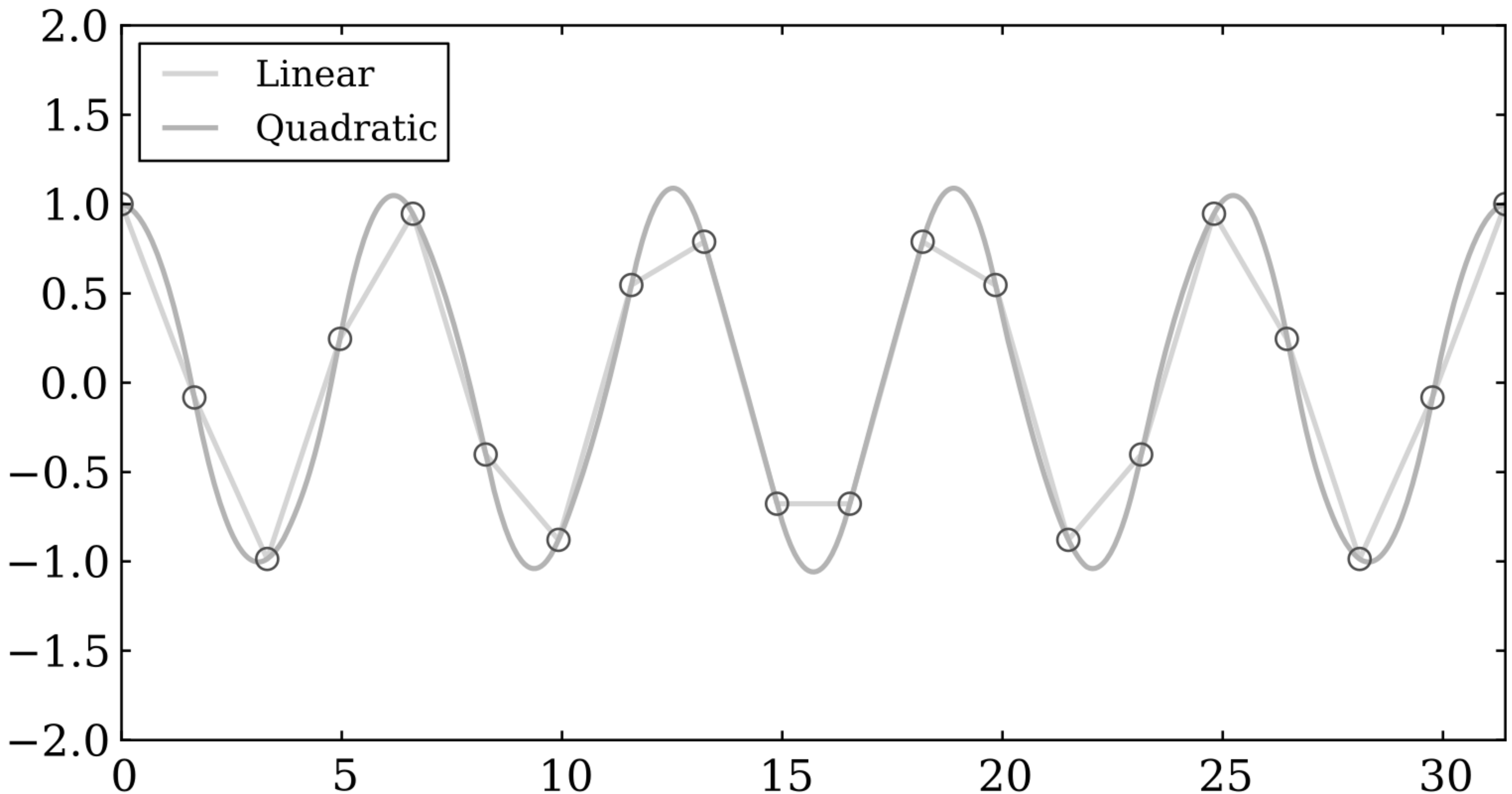
- 2 basic methods of interpolation:

  (1) Fit one function to an entire dataset

  (2) fit different parts of the dataset with several functions where the joints of each function are joined smoothly --- spline interpolation.

- Using **scipy.interpolate.interp1d** to interpolate a sinusoidal function with different fitting parameters.

```
import numpy as np
from scipy.interpolate import interp1d

# Setting up fake data
x = np.linspace(0, 10 * np.pi, 20)
y = np.cos(x)

# Interpolating data
fl = interp1d(x, y, kind='linear')
fq = interp1d(x, y, kind='quadratic')

# x.min and x.max are used to make sure we do not
# go beyond the boundaries of the data for the
# interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yintl = fl(xint)
yintq = fq(xint)
```

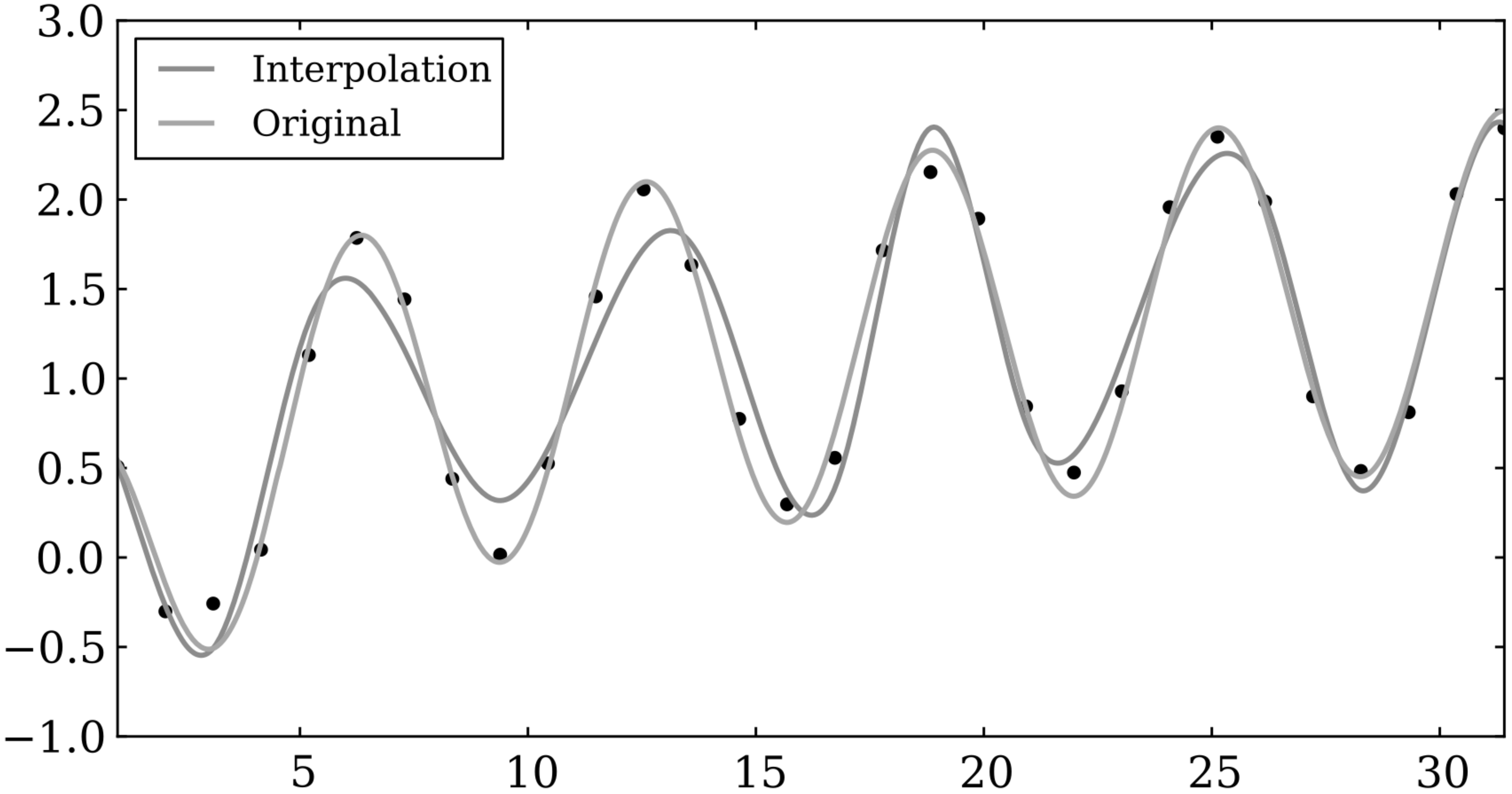


Linear
Quadratic
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
0
5
10
15
20
25
30

- Interpolate noisy data by using a spline-fitting function called **scipy.interpolate.UnivariateSpline**.

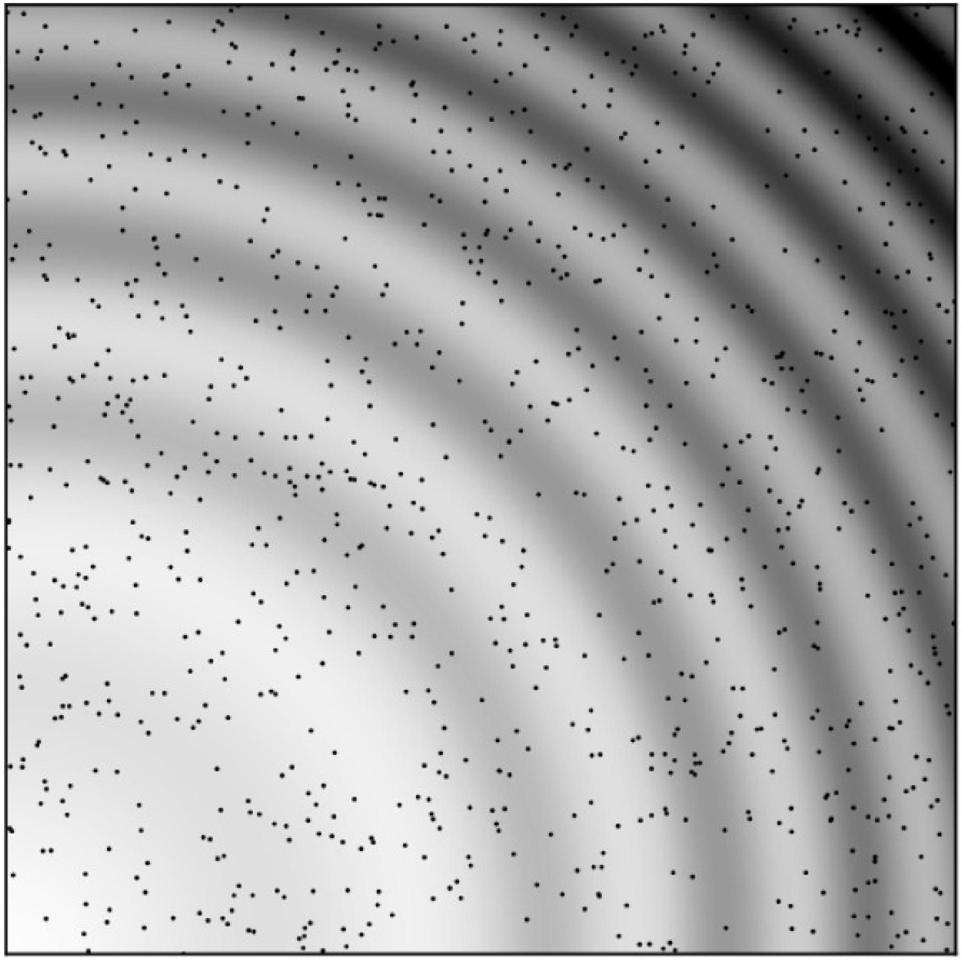
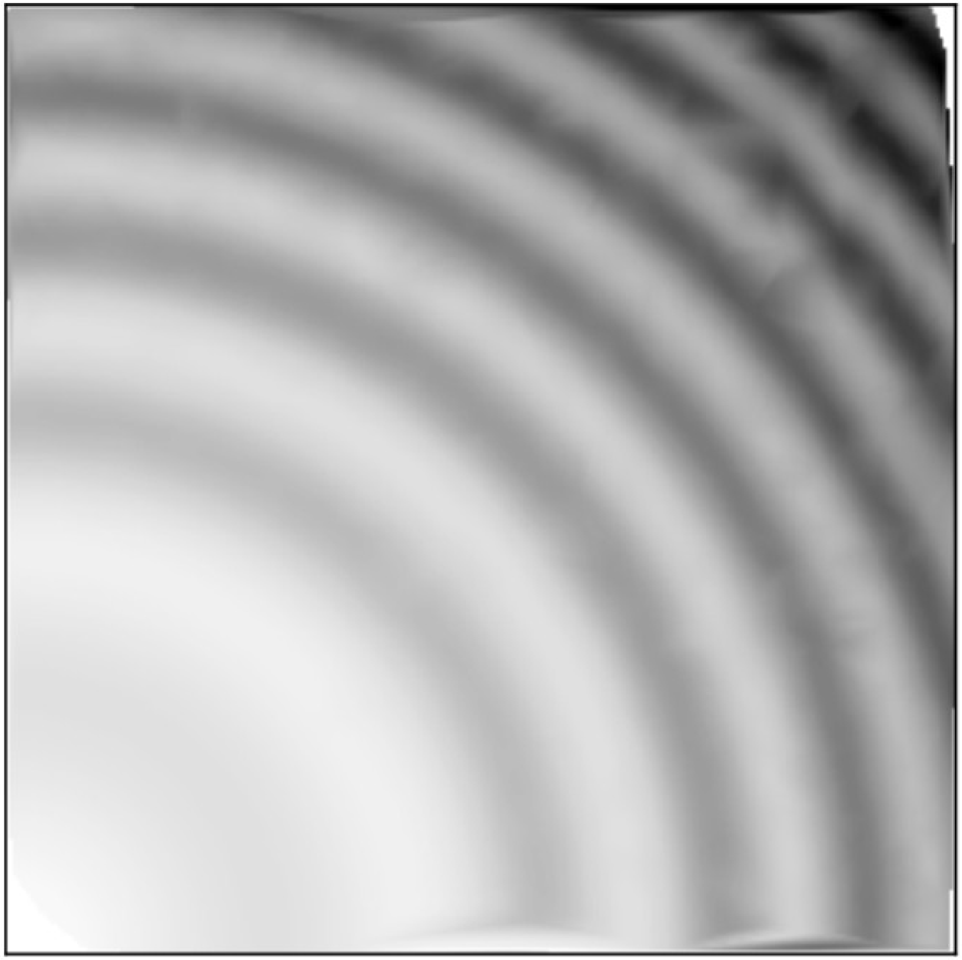
```
import numpy as np
import matplotlib.pyplot as mpl
from scipy.interpolate import UnivariateSpline

# Setting up fake data with artificial noise
sample = 30
x = np.linspace(1, 10 * np.pi, sample)
y = np.cos(x) + np.log10(x) + \
      np.random.randn(sample) / 10

# Interpolating the data
f = UnivariateSpline(x, y, s=1)

# x.min/x.max are used to make sure not to go beyond
# the boundaries of the data for the interpolation.
xint = np.linspace(x.min(), x.max(), 1000)
yint = f(xint)
```

- The option **s** is the smoothing factor, which should be used when fitting data with noise. If instead **s=0**, then the interpolation will go through all points while ignoring noise.



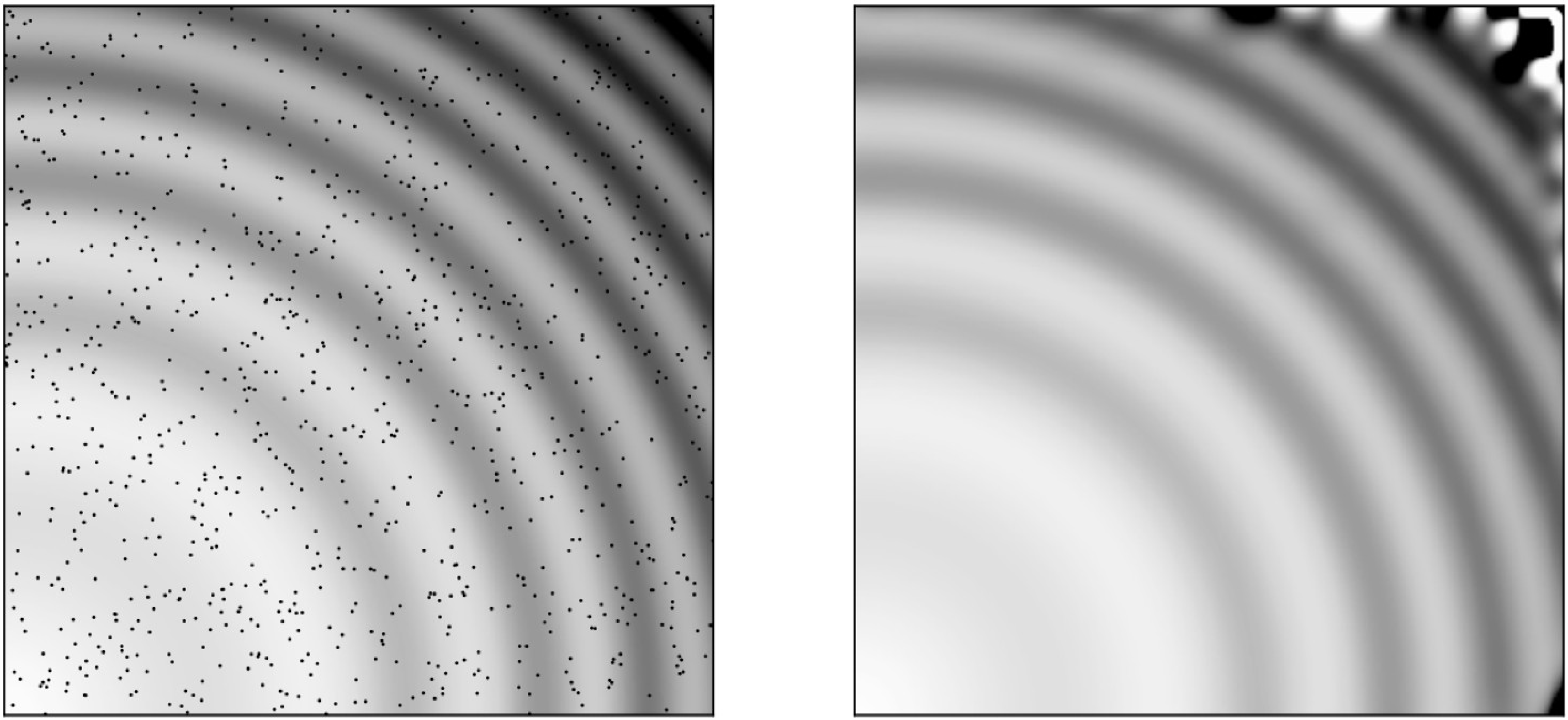
- **scipy.interpolate.griddata** is used for its capacity to deal with unstructured N-dim data:

```
import numpy as np
from scipy.interpolate import griddata

# Defining a function
ripple=lambda x,y:np.sqrt(x**2+y**2)+np.sin(x**2+y**2)

# Generating gridded data. The complex number
# defines how many steps the grid data should have.
# Without the complex number mgrid would only create
# a grid data structure with 5 steps.
grid_x, grid_y = np.mgrid[0:5:1000j, 0:5:1000j]

# Generating sample that interpolation function will see
xy = np.random.rand(1000, 2)
sample = ripple(xy[:,0] * 5 , xy[:,1] * 5)

# Interpolating data with a cubic
grid_z0 = griddata( xy * 5, sample, (grid_x, grid_y),
method='cubic')
```

- Employ another multivariate spline interpolation, **scipy.interpolate.SmoothBivariateSpline**:

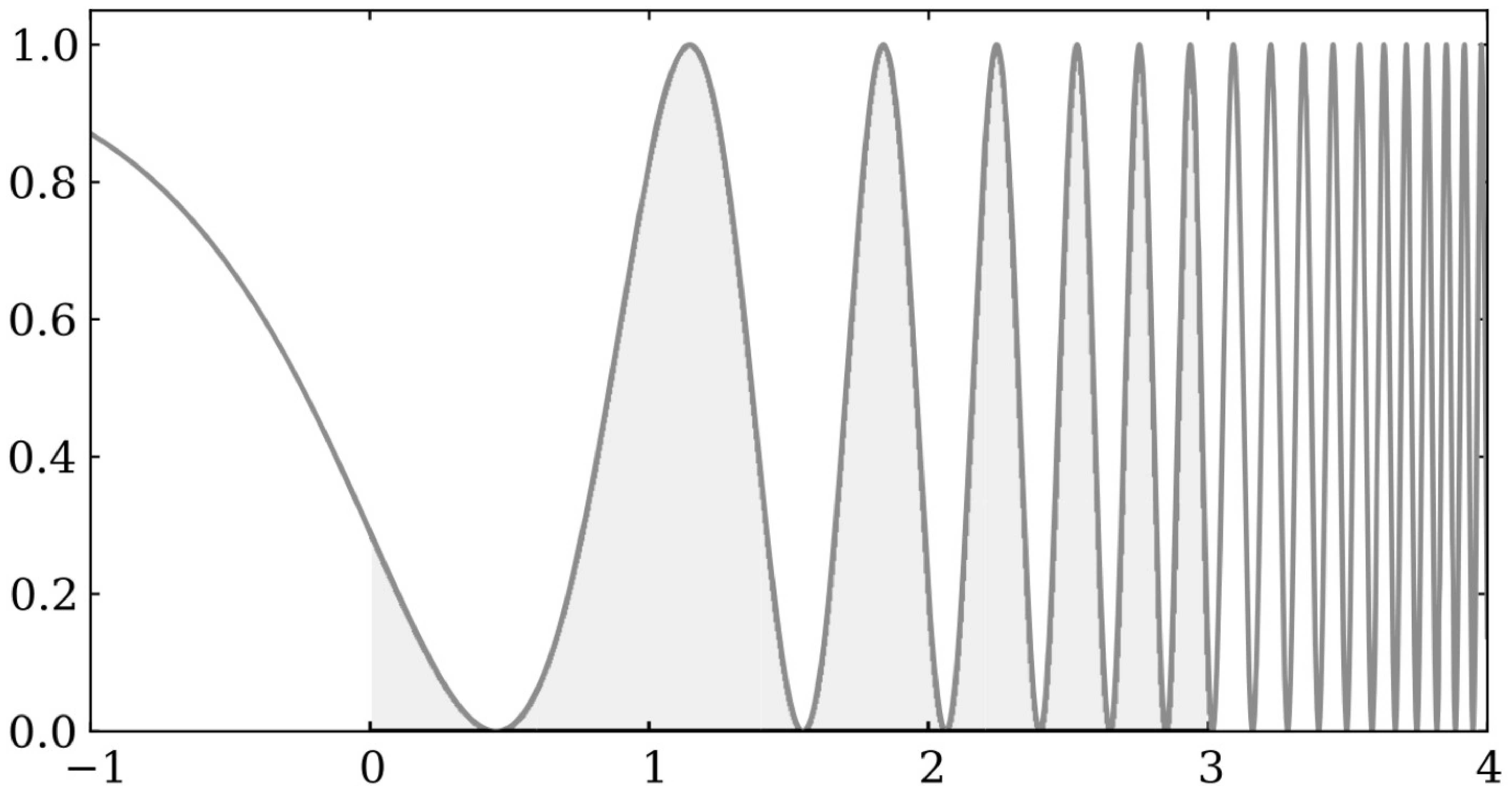
```
import numpy as np
from scipy.interpolate import SmoothBivariateSpline \
as SBS

# Defining a function
ripple=lambda x,y:np.sqrt(x**2+y**2)+np.sin(x**2+y**2)

# Generating sample that interpolation function will see
xy= np.random.rand(1000, 2)
x, y = xy[:,0], xy[:,1]
sample = ripple(xy[:,0] * 5 , xy[:,1] * 5)

# Interpolating data
fit = SBS(x * 5, y * 5, sample, s=0.01, kx=4, ky=4)
interp=fit(np.linspace(0,5,1000),np.linspace(0,5,1000))
```



Analytic Integration

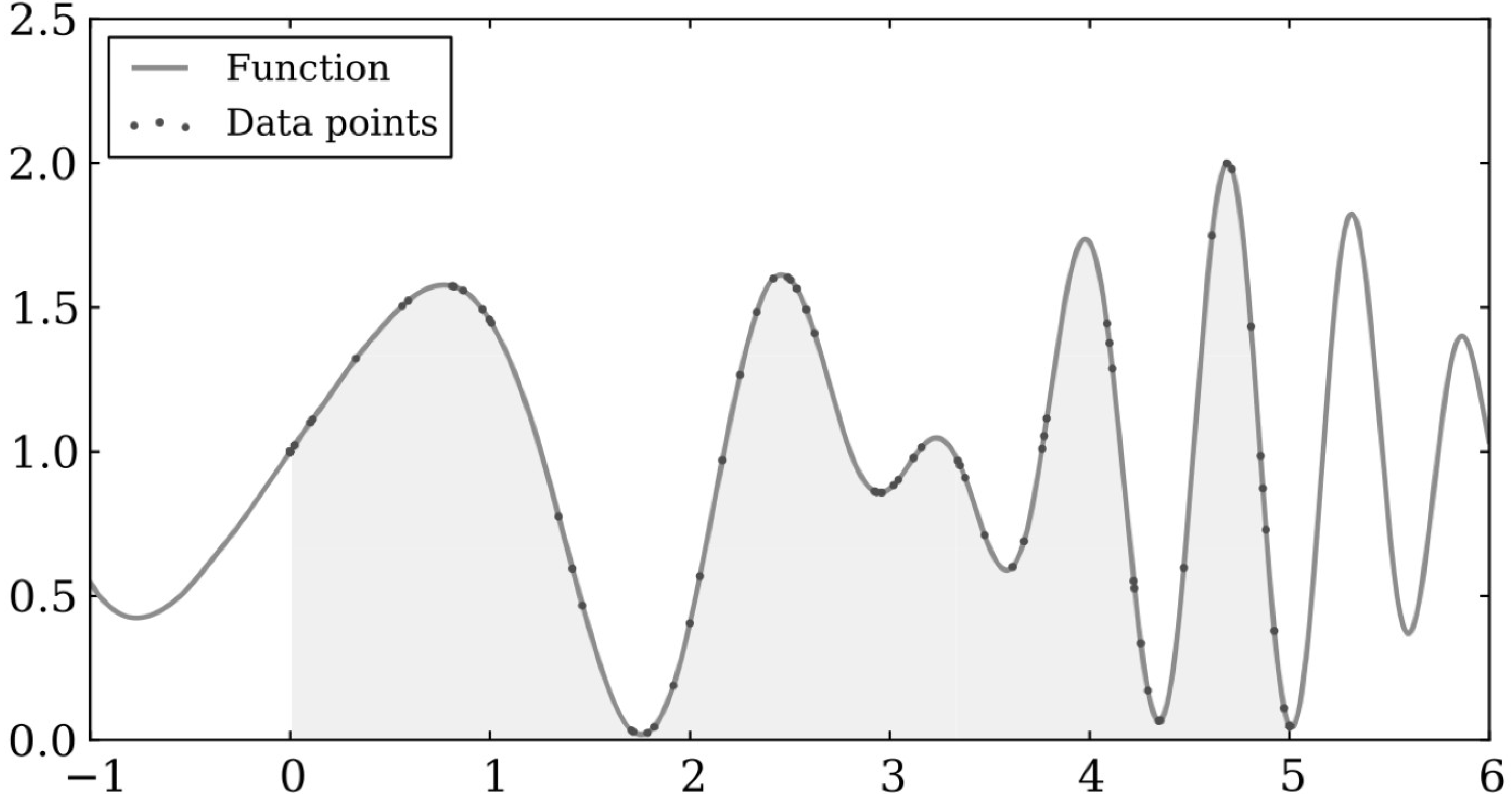
- $\int_0^3 \cos^2(e^x)\mathrm{d}x$

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

# Defining function to integrate
func = lambda x: np.cos(np.exp(x)) ** 2

# Integrating function with upper and lower
# limits of 0 and 3, respectively
solution = quad(func, 0, 3)
print(solution)

# The first element is the desired value
# and the second is the error.
# (1.296467785724373, 1.397797186265988e-09)
```


1.0
0.8
0.6
0.4
0.2
0.0
−1
0
1
2
3
4

Numerical Integration

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

# Setting up fake data
x = np.sort(np.random.randn(150) * 4 + 4).clip(0,5)
func = lambda x: np.sin(x) * np.cos(x ** 2) + 1
y = func(x)

# Integrating function with upper/lower limits = 0 / 5
fsolution = quad(func, 0, 5)
dsolution = trapz(y, x=x)

print('fsolution = '+str(fsolution[0])) # 5.10034506754
print('dsolution = ' + str(dsolution))  # 5.04201628314
print('The difference is ' + str(np.abs(fsolution[0] – \
dsolution)))                                    #5.04201628314
```


Function
Data points
2.5
2.0
1.5
1.0
0.5
0.0
−1
0
1
2
3
4
5
6

Statistics

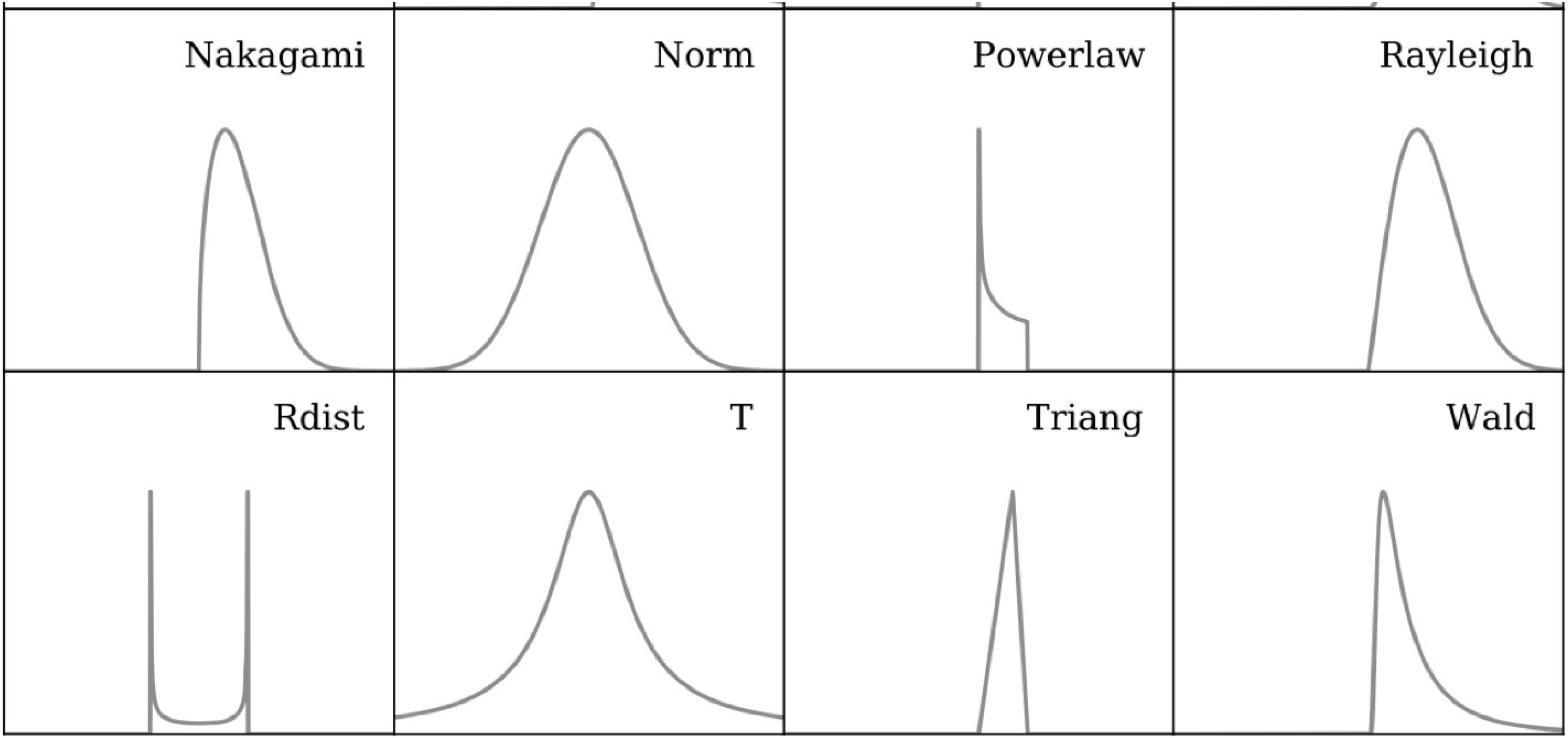
- In **NumPy** there are basic statistical functions like **mean**, **std**, **median**, **argmax**, and **argmin**. **numpy.arrays** have built-in methods that allow us to use most of the **NumPy** statistics easily:

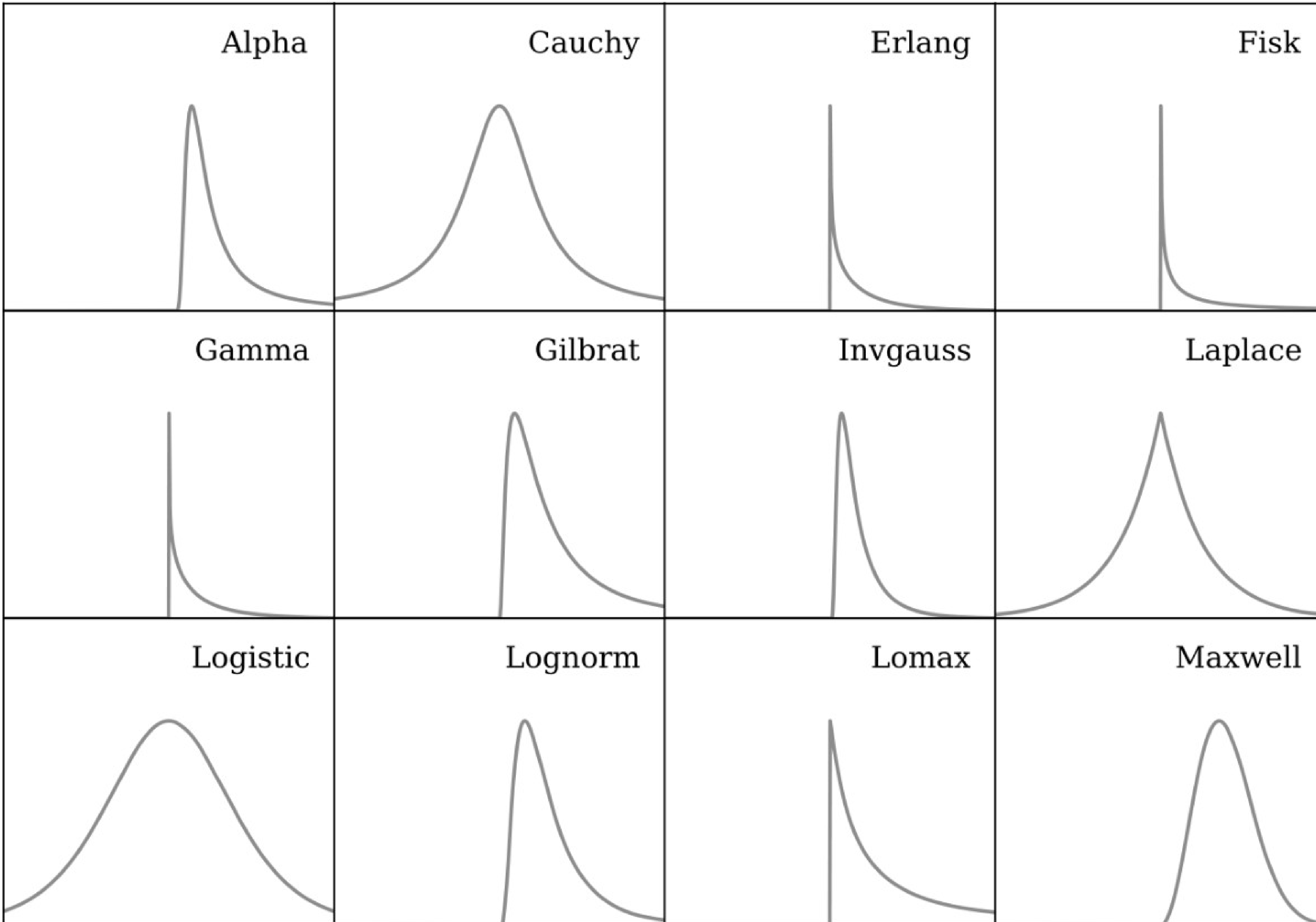
```
import numpy as np

# Constructing a random array with 1000 elements
x = np.random.randn(1000)

# Calculating several of the built-in methods
# that numpy.array has
mean = x.mean()
std = x.std()
var = x.var()
```

- **SciPy** offers an extended collection of statistical tools such as distributions (continuous or discrete) and functions.

Continuous and Discrete Distributions

- 20 of the continuous functions are shown in the figures as probability density functions (PDFs) to give an impression of what the **scipy.stats** package provides.





Alpha
Cauchy
Erlang
Fisk
Gamma
Gilbrat
Invgauss
Laplace
Logistic
Lognorm
Lomax
Maxwell

- When we call a distribution from **scipy.stats**, we can extract its information in several ways: probability density functions (PDFs), cumulative distribution functions (CDFs), random variable samples (RVSs), percent point functions (PPFs), and more.

- For the classic normal function $PDF = e^{-\frac{x^2/2}{\sqrt{2\pi}}}$

```
import numpy as np
import scipy.stats import norm

x = np.linspace(-5,5,1000)    # Set up the sample range

# Normal Dist: loc: mean, scale: standard deviation.
dist = norm(loc=0, scale=1)

# Retrieving norm's PDF and CDF
pdf = dist.pdf(x)
cdf = dist.cdf(x)

# Here we draw out 500 random values from the norm.
sample = dist.rvs(500)
```

- The probability mass function (PMF) of the geometric distribution.

$$\mathrm{PMF} = (1 - p)^{(k-1)}\, p$$

```
import numpy as np
from scipy.stats import geom

# Set up the parameters for the geometric distribution.
p = 0.5
dist = geom(p)

# Set up the sample range.
x = np.linspace(0, 5, 1000)

# Retrieving geom's PMF and CDF
pmf = dist.pmf(x)
cdf = dist.cdf(x)

# Here we draw out 500 random values.
sample = dist.rvs(500)
```