Lecture#6

May 24, 2023

1 Python for Signal Processing

1.1 SciPy - Library of scientific algorithms for Python

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

The line "%matplotlib inline" is a command used in Jupyter notebooks to ensure that any plots generated by Matplotlib are displayed directly in the notebook, rather than in a separate window or file. This line is not valid Python code but rather a command specific to the Jupyter notebook environment.

The following two lines import the Matplotlib library, which is a data visualization library in Python, and the Image module from the IPython.display library, which is used to display image files within the notebook. These imports are required in order to create and display plots using Matplotlib in the notebook

[1]:
```
# what is this line all about?
%matplotlib inline
import matplotlib.pyplot as plt
from IPython.display import Image
```

1.2 Introduction

The SciPy framework builds on top of the low-level NumPy framework for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are:

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

In this lecture we will look at how to use some of these subpackages.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module.

The line "from scipy import *" imports all the public modules and functions of the Scipy library into the current Python namespace. Scipy is a Python library that provides functionality for scientific computing and technical computing, including modules for optimization, integration, interpolation, signal and image processing, linear algebra, and more.

Using the "*" symbol to import all modules and functions from a library is generally discouraged as it can cause namespace pollution and conflicts with other imported modules. It's better to only import the specific modules and functions needed for a given task, like "from scipy.optimize import minimize" for optimization or "from scipy.signal import fftconvolve" for signal processing

[2]:
```
from scipy import *
```

If we only need to use part of the SciPy framework we can selectively include only those modules we are interested in. For example, to include the linear algebra package under the name `la`, we can do:

[3]:
```
import scipy.linalg as la
```

1.3 Special functions

A large number of mathematical special functions are important for many computional physics problems. SciPy provides implementations of a very extensive set of special functions. For details, see the list of functions in the reference documention at http://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special.

To demonstrate the typical usage of special functions we will look in more detail at the Bessel functions:

The code imports the "jn", "yn", "jn_zeros", and "yn_zeros" functions from the "scipy.special" module, which is a part of the Scipy library. These functions are used to evaluate Bessel functions and their zeroes.

Bessel functions are a family of special functions that arise in a wide range of mathematical and physical problems, particularly in wave phenomena such as vibrations, diffraction, and propagation. The "jn" and "yn" functions are Bessel functions of the first and second kind, respectively, with real-valued order. These functions are defined by a series expansion involving trigonometric and exponential functions and have many applications in physics and engineering.

The "jn_zeros" and "yn_zeros" functions return the zeros of the "jn" and "yn" functions, respectively. These zeros are important in a variety of problems, such as finding the resonance frequencies of vibrating systems, calculating diffraction patterns, and solving partial differential equations.

[4]:
```
#
# The scipy.special module includes a large number of Bessel-functions
# Here we will use the functions jn and yn, which are the Bessel functions
# of the first and second kind and real-valued order. We also include the
# function jn_zeros and yn_zeros that gives the zeroes of the functions jn
# and yn.
#
from scipy.special import jn, yn, jn_zeros, yn_zeros
```

This code imports the "jn" and "yn" functions from the "scipy.special" module to evaluate Bessel functions of the first and second kind.

The variable "n" is set to 0, which is the order of the Bessel function. The variable "x" is initially set to 0.0 for the Bessel function of the first kind, and then to 1.0 for the Bessel function of the second kind.

The first print statement uses the "jn" function to evaluate the Bessel function of the first kind, Jn(x), with n=0 and x=0.0, and prints the result in a formatted string using the "%" operator. The result is the value of J0(0), which is equal to 1.

The second print statement uses the "yn" function to evaluate the Bessel function of the second kind, Yn(x), with n=0 and x=1.0, and prints the result in a similar formatted string. The result is the value of Y0(1), which is approximately -0.7812.

Overall, this code demonstrates the use of the "jn" and "yn" functions to evaluate Bessel functions in Python using Scipy.

[5]:
```
from scipy.special import jn, yn

n = 0     # order
x = 0.0

# Bessel function of first kind
print("J_%d(%f) = %f" % (n, x, jn(n, x)))

x = 1.0
# Bessel function of second kind
print("Y_%d(%f) = %f" % (n, x, yn(n, x)))
```

```
J_0(0.000000) = 1.000000
Y_0(1.000000) = 0.088257
```

This code uses the NumPy and Matplotlib libraries, as well as the "jn" function from the Scipy library, to generate a plot of Bessel functions of the first kind.

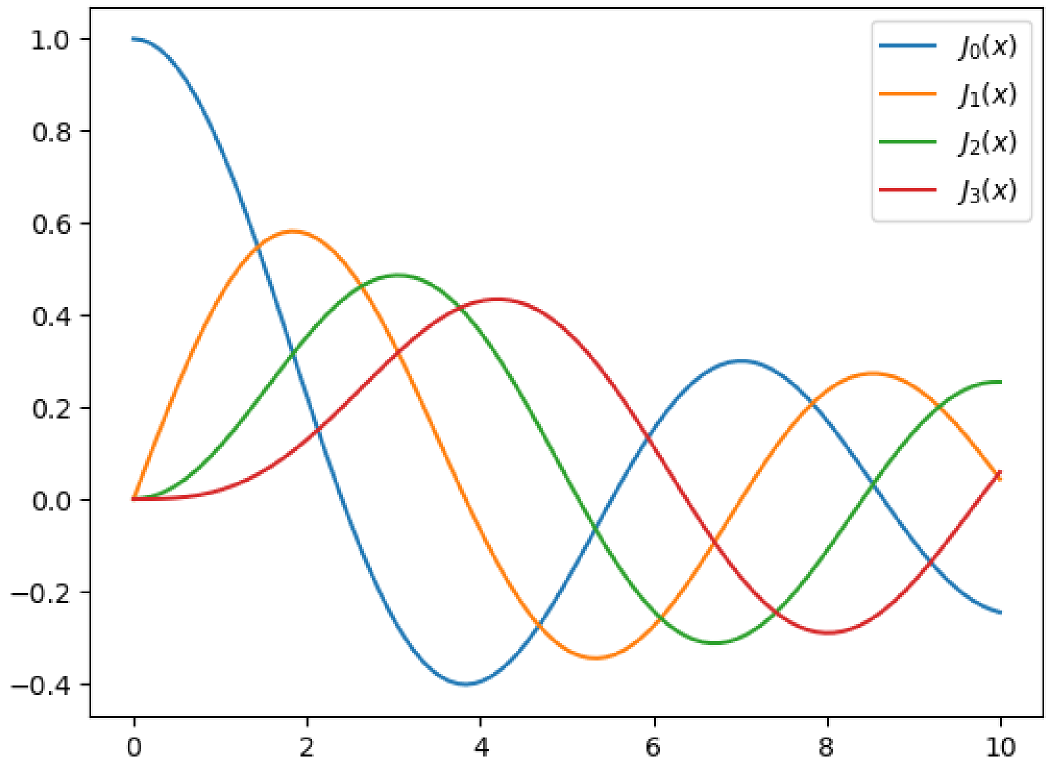
The "np.linspace" function creates an array of 100 equally spaced values between 0 and 10, which will be used as the x-values for the plot.

The "plt.subplots" function creates a new figure and a set of subplots, returning a tuple containing the figure and the axis object(s). The "ax.plot" function is then called in a loop over the range 0 to 3, with each iteration plotting a different Bessel function of the first kind, Jn(x), for the current value of n. The label for each plot is set using a formatted string with the value of n.

After the loop completes, the "ax.legend" function adds a legend to the plot indicating which line corresponds to which Bessel function. Finally, the "plt.show" function displays the plot.

Overall, this code demonstrates how to use the "jn" function from the Scipy library to plot Bessel functions of the first kind in Python using Matplotlib.

```
[6]: import numpy as np
     import matplotlib.pyplot as plt
     from scipy.special import jn

     x = np.linspace(0, 10, 100)

     fig, ax = plt.subplots()
     for n in range(4):
         ax.plot(x, jn(n, x), label=r"$J_{%d}(x)$" % n)
     ax.legend()
     plt.show()
```



This code uses the “jn_zeros” function from the Scipy library to compute the first 4 zeros of the Bessel function of the first kind, J0(x).

The variable “n” is set to 0, which is the order of the Bessel function. The variable “m” is set to 4, which is the number of zeros to compute.

The “jn_zeros” function returns an array containing the requested number of zeros of the Bessel function, ordered from smallest to largest. In this case, the result is an array of length 4 containing the first 4 zeros of J0(x), which are approximately 2.4048, 5.5201, 8.6537, and 11.7915.

Overall, this code demonstrates how to use the “jn_zeros” function from the Scipy library to compute the zeros of a Bessel function in Python.

[7]:
```
# zeros of Bessel functions
n = 0 # order
m = 4 # number of roots to compute
jn_zeros(n, m)
```

[7]:
```
array([ 2.40482556,  5.52007811,  8.65372791, 11.79153444])
```

1.4 Integration

1.4.1 Numerical integration: quadrature

Numerical evaluation of a function of the type

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

is called *numerical quadrature*, or simply *quadature*. SciPy provides a series of functions for different kind of quadrature, for example the `quad`, `dblquad` and `tplquad` for single, double and triple integrals, respectively.

[8]:
```
from scipy.integrate import quad, dblquad, tplquad
```

The `quad` function takes a large number of optional arguments, which can be used to fine-tune the behaviour of the function (try `help(quad)` for details).

The basic usage is as follows:

[9]:
```
# define a simple function for the integrand
def f(x):
    return x
```

This code uses the "quad" function from the Scipy library to compute the definite integral of the function f(x) = x^2 over the interval [0, 1].

The lower and upper limits of the interval are set to 0 and 1, respectively, using the variables "x_lower" and "x_upper".

The function "f(x)" is defined using a "def" statement, which returns the value of x^2 for any input x.

The "quad" function takes the integrand function, the lower and upper limits of integration, and optionally some additional arguments, and returns a tuple containing the value of the definite integral and an estimate of the absolute error.

The results are printed to the console using a print statement that displays the value of the integral and the absolute error.

Overall, this code demonstrates how to use the "quad" function from the Scipy library to numerically evaluate a definite integr

[10]:
```
from scipy.integrate import quad

x_lower = 0 # the lower limit of x
```

```
x_upper = 1 # the upper limit of x

def f(x):
    return x**2  # Define the integrand function

val, abserr = quad(f, x_lower, x_upper)

print("integral value =", val, ", absolute error =", abserr) # Use parentheses␣
 ↪for print statement in Python 3
```

```
integral value = 0.33333333333333337 , absolute error = 3.700743415417189e-15
```

If we need to pass extra arguments to integrand function we can use the `args` keyword argument:

This Python code utilizes the quad function from the scipy.integrate library to perform numerical integration on a given function. The function in this code is integrand(x, n), which calculates the Bessel function of the first kind and order n for a given value of x. The code specifies the lower and upper limits of integration as x_lower and x_upper, respectively. Then, the quad function is called with the integrand function, lower and upper limits, and additional argument of n=3. The quad function returns two values: the estimated value of the integral and the absolute error between the result and the true value of the integral. Finally, the resulting values are printed to the console using the print function.

[11]:
```
from scipy.integrate import quad
from scipy.special import jn

def integrand(x, n):
    """
    Bessel function of first kind and order n.
    """
    return jn(n, x)

x_lower = 0  # the lower limit of x
x_upper = 10 # the upper limit of x

val, abserr = quad(integrand, x_lower, x_upper, args=(3,))

print(val, abserr)  # Use parentheses for print statement in Python 3
```

```
0.7366751370811073 9.389126882496403e-13
```

For simple functions we can use a lambda function (name-less function) instead of explicitly defining a function for the integrand:

This Python code uses the quad function from the scipy.integrate library to compute the definite integral of the function exp(-x ** 2) over the interval [-inf, inf]. The quad function requires an integrand function, limits of integration, and optional additional arguments. In this case, the integrand function is defined using a lambda expression.

The quad function returns two values: the estimated value of the integral val and the absolute error abserr. These values are printed to the console using the print function.

Finally, the code computes the analytical solution of the integral, which is sqrt(pi), and prints it to the console. This is done for comparison with the numerical solution obtained using the quad function.

Overall, this code demonstrates how to use the quad function to numerically integrate a function in Python.

[12]:
```
from scipy.integrate import quad
from numpy import exp, inf, pi, sqrt

val, abserr = quad(lambda x: exp(-x ** 2), -inf, inf)

print("numerical  =", val, abserr)

analytical = sqrt(pi)
print("analytical =", analytical)
```

```
numerical  = 1.7724538509055159 1.4202636780944923e-08
analytical = 1.7724538509055159
```

As show in the example above, we can also use 'Inf' or '-Inf' as integral limits.

Higher-dimensional integration works in the same way:

[13]:
```
from scipy.integrate import dblquad
from numpy import exp

def integrand(x, y):
    return exp(-x**2-y**2)

x_lower = 0
x_upper = 10
y_lower = 0
y_upper = 10

val, abserr = dblquad(integrand, x_lower, x_upper, lambda x: y_lower, lambda x:␣
 ↪y_upper)

print(val, abserr)  # Use parentheses for print statement in Python 3
```

```
0.7853981633974476 1.3753098510218528e-08
```

Note how we had to pass lambda functions for the limits for the y integration, since these in general can be functions of x.

1.5 Ordinary differential equations (ODEs)

SciPy provides two different ways to solve ODEs: An API based on the function `odeint`, and object-oriented API based on the class `ode`. Usually `odeint` is easier to get started with, but the `ode` class offers some finer level of control.

Here we will use the `odeint` functions. For more information about the class `ode`, try `help(ode)`. It does pretty much the same thing as `odeint`, but in an object-oriented fashion.

To use `odeint`, first import it from the `scipy.integrate` module

[14]:
```
from scipy.integrate import odeint, ode
```

A system of ODEs are usually formulated on standard form before it is attacked numerically. The standard form is:

$y' = f(y, t)$

where

$y = [y_1(t), y_2(t), ..., y_n(t)]$

and f is some function that gives the derivatives of the function $y_i(t)$. To solve an ODE we need to know the function f and an initial condition, $y(0)$.

Note that higher-order ODEs can always be written in this form by introducing new variables for the intermediate derivatives.

Once we have defined the Python function `f` and array `y_0` (that is f and $y(0)$ in the mathematical formulation), we can use the `odeint` function as:

`y_t = odeint(f, y_0, t)`

where `t` is and array with time-coordinates for which to solve the ODE problem. `y_t` is an array with one row for each point in time in `t`, where each column corresponds to a solution `y_i(t)` at that point in time.

We will see how we can implement `f` and `y_0` in Python code in the examples below.

Example: double pendulum Let's consider a physical example: The double compound pendulum, described in some detail here: http://en.wikipedia.org/wiki/Double_pendulum

[15]:
```
Image(url='http://upload.wikimedia.org/wikipedia/commons/c/c9/
 ↪Double-compound-pendulum-dimensioned.svg')
```

[15]:
```
<IPython.core.display.Image object>
```

The equations of motion of the pendulum are given on the wiki page:

$\dot{\theta}_1 = \frac{6}{m\ell^2}\frac{2p_{\theta_1}-3\cos(\theta_1-\theta_2)p_{\theta_2}}{16-9\cos^2(\theta_1-\theta_2)}$

$\dot{\theta}_2 = \frac{6}{m\ell^2}\frac{8p_{\theta_2}-3\cos(\theta_1-\theta_2)p_{\theta_1}}{16-9\cos^2(\theta_1-\theta_2)}.$

$\dot{p}_{\theta_1} = -\frac{1}{2}m\ell^2\left[\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2)+3\frac{g}{\ell}\sin\theta_1\right]$

$\dot{p}_{\theta_2} = -\frac{1}{2}m\ell^2\left[-\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2)+\frac{g}{\ell}\sin\theta_2\right]$

To make the Python code simpler to follow, let's introduce new variable names and the vector notation: $x = [\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2}]$

$\dot{x}_1 = \frac{6}{m\ell^2}\frac{2x_3-3\cos(x_1-x_2)x_4}{16-9\cos^2(x_1-x_2)}$

$\dot{x}_2 = \frac{6}{m\ell^2}\frac{8x_4 - 3\cos(x_1 - x_2)x_3}{16 - 9\cos^2(x_1 - x_2)}$

$\dot{x}_3 = -\frac{1}{2}m\ell^2\left[\dot{x}_1\dot{x}_2\sin(x_1 - x_2) + 3\frac{g}{\ell}\sin x_1\right]$

$\dot{x}_4 = -\frac{1}{2}m\ell^2\left[-\dot{x}_1\dot{x}_2\sin(x_1 - x_2) + \frac{g}{\ell}\sin x_2\right]$

This Python code defines a function dx that describes the dynamics of a physical pendulum. The pendulum is assumed to be a simple pendulum consisting of a mass m attached to a rigid rod of length L. The constants g, L, and m are defined at the beginning of the code.

The dx function takes two arguments: an array x that contains the values of the four variables that describe the state of the pendulum, and a scalar t that represents time (although the time variable is not used in this code). The four variables in x are:

```
x1: the angular displacement of the pendulum from its equilibrium position (in radians)
x2: the angular displacement of a secondary pendulum (in radians)
x3: the angular velocity of the pendulum (in radians per second)
x4: the angular velocity of the secondary pendulum (in radians per second)
```

The function dx first extracts the values of the four variables from the input array x. It then computes the derivatives of these variables, which describe how the state of the pendulum changes over time.

The computation of the derivatives is based on the equations of motion for a physical pendulum. These equations take into account the gravitational force acting on the mass, the torque exerted by the rod, and the conservation of angular momentum. The exact form of the equations used in this code is not explained here, but it can be found in many textbooks on classical mechanics.

The function dx returns a list containing the computed derivatives of the four variables, in the same order as the input array x. This list represents the rate of change of the state of the pendulum over time, and is used by numerical integration methods to simulate the motion of the pendulum.

[16]:
```python
# Define constants
g = 9.81
L = 0.5
m = 0.1

# Define the function dx that describes the pendulum ODE
def dx(x, t):
    """
    The right-hand side of the pendulum ODE
    """
    # Extract the variables from the input array x
    x1, x2, x3, x4 = x[0], x[1], x[2], x[3]

    # Compute the derivatives of the variables
    dx1 = 6.0/(m*L**2) * (2 * x3 - 3 * cos(x1-x2) * x4)/(16 - 9 * cos(x1-x2)**2)
    dx2 = 6.0/(m*L**2) * (8 * x4 - 3 * cos(x1-x2) * x3)/(16 - 9 * cos(x1-x2)**2)
    dx3 = -0.5 * m * L**2 * ( dx1 * dx2 * sin(x1-x2) + 3 * (g/L) * sin(x1))
    dx4 = -0.5 * m * L**2 * (-dx1 * dx2 * sin(x1-x2) + (g/L) * sin(x2))
    # Return the derivatives as a list
    return [dx1, dx2, dx3, dx4]
```

This Python code simulates the motion of a double pendulum using numerical integration. The motion of the double pendulum is governed by a system of ordinary differential equations (ODEs), which are solved using the odeint function from the SciPy library.

The code first imports the math library to access the value of pi, and chooses an initial state for the double pendulum. The initial state is represented by an array x0 that contains the values of four variables: the angular displacements and velocities of the two pendulum masses. The angles are measured in radians.

The code then defines a time coordinate t using the linspace function from the NumPy library. The linspace function creates an array of 250 evenly spaced points between 0 and 10 seconds, which will be used to simulate the motion of the pendulum.

The code defines a function dx (not shown) that describes the dynamics of the double pendulum, similar to the previous code. This function takes as input an array x containing the state of the system, and returns the derivatives of the four variables.

The odeint function is then used to solve the ODE problem for the chosen initial state and time coordinate. The odeint function takes as input the function dx, the initial state x0, and the time coordinate t, and returns an array x containing the values of the four variables at each time point.

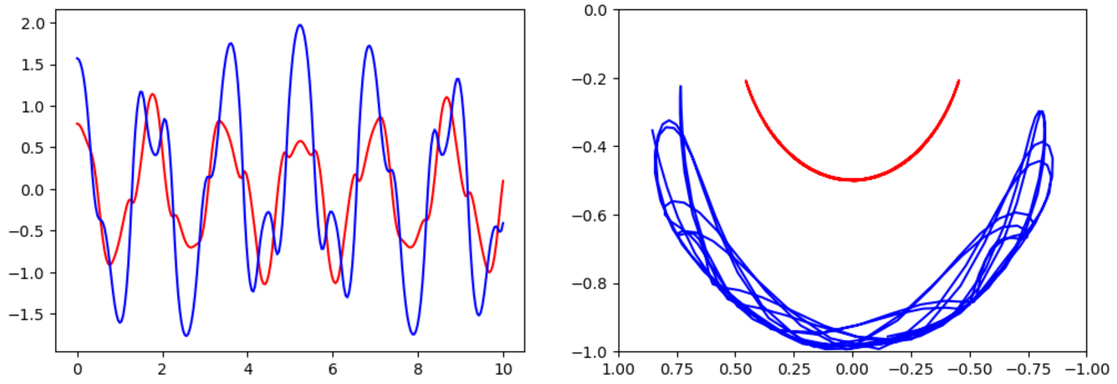
Finally, the code plots the angular displacements of the two pendulum masses as a function of time, and the positions of the two masses in space. The positions are computed from the angular displacements using trigonometric functions (sin and cos). The positions are plotted in two dimensions using the plot function from Matplotlib. The positions of the two masses are represented by two different colors (red and blue), and the limits of the y-axis and x-axis are set to make the visualization more informative. The resulting plot shows the complex and chaotic motion of the double pendulum over time.

[17]:
```
# Import the math library to use the value of pi
import math

# Choose an initial state
x0 = [math.pi/4, math.pi/2, 0, 0]
```

[18]:
```
# Import the linspace function from the numpy library
from numpy import linspace

# Time coordinate to solve the ODE for: from 0 to 10 seconds
t = linspace(0, 10, 250)
```

[19]:
```
from scipy.integrate import odeint
from numpy import cos, sin

# solve the ODE problem
x = odeint(dx, x0, t)
```

[20]:
```python
# plot the angles as a function of time

fig, axes = plt.subplots(1,2, figsize=(12,4))
axes[0].plot(t, x[:, 0], 'r', label="theta1")
axes[0].plot(t, x[:, 1], 'b', label="theta2")


x1 = + L * sin(x[:, 0])
y1 = - L * cos(x[:, 0])

x2 = x1 + L * sin(x[:, 1])
y2 = y1 - L * cos(x[:, 1])

axes[1].plot(x1, y1, 'r', label="pendulum1")
axes[1].plot(x2, y2, 'b', label="pendulum2")
axes[1].set_ylim([-1, 0])
axes[1].set_xlim([1, -1]);
```



Simple annimation of the pendulum motion.

This Python code is simulating a double pendulum system and plotting the angles and positions of the pendulums over time. Here's an explanation of the code:

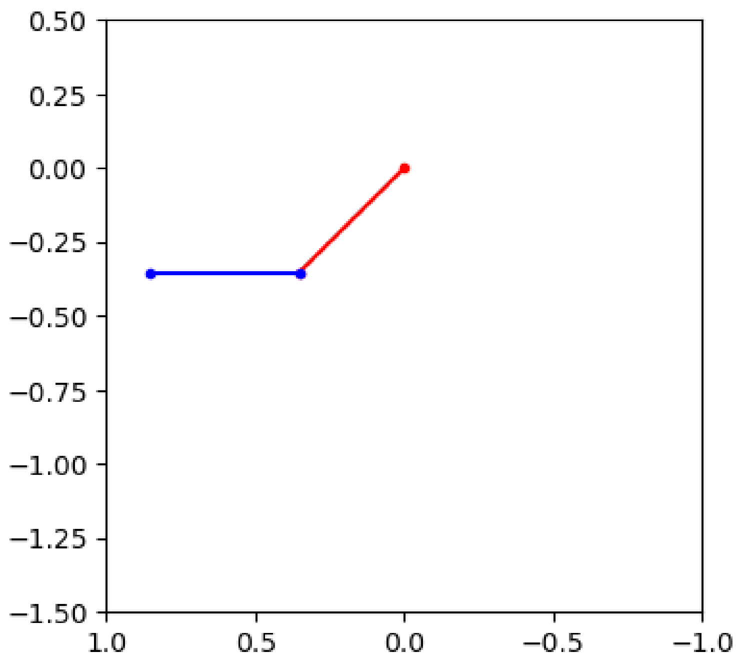
1. Import the math library: The `math` library is imported to use the value of pi (`math.pi`).
2. Choose an initial state: The variable `x0` represents the initial state of the system. It is a list containing four values: `[math.pi/4, math.pi/2, 0, 0]`. These values represent the initial angles and angular velocities of the two pendulums.
3. Import necessary functions and libraries: The `linspace` function from the `numpy` library is imported. It is used to generate a time coordinate `t` that ranges from 0 to 10 seconds with 250 evenly spaced points.
4. Import additional functions: The `odeint` function from the `scipy.integrate` module is imported. It is used to solve the ordinary differential equation (ODE) problem.

5. Solve the ODE problem: The `odeint` function is called to solve the ODE problem. It takes three arguments: the derivative function `dx`, the initial state `x0`, and the time coordinate `t`. The result is stored in the `x` variable, which represents the angles and angular velocities of the pendulums over time.
6. Plot the angles and positions: The code creates a figure with two subplots using `plt.subplots(1, 2, figsize=(12, 4))`. The first subplot (`axes[0]`) plots the angles of the pendulums (`x[:, 0]` represents the first angle and `x[:, 1]` represents the second angle) against time (`t`). The second subplot (`axes[1]`) plots the positions of the pendulums using the angles and lengths.
7. Calculate the positions of the pendulums: The variables `x1` and `y1` represent the x and y coordinates of the first pendulum, respectively. The variables `x2` and `y2` represent the x and y coordinates of the second pendulum, respectively. These positions are calculated using the angles and lengths of the pendulums.
8. Plot the positions: The positions of the pendulums are plotted on the second subplot (`axes[1]`) using `axes[1].plot(x1, y1, 'r', label="pendulum1")` and `axes[1].plot(x2, y2, 'b', label="pendulum2")`. The plot limits for the y-axis (`axes[1].set_ylim([-1, 0])`) and x-axis (`axes[1].set_xlim([1, -1])`) are also set.

The code simulates the motion of a double pendulum system and visualizes the angles and positions of the pendulums over time.

[21]:
```python
from IPython.display import display, clear_output
import time
```

[22]:
```python
import matplotlib.pyplot as plt
from numpy import cos, sin

fig, ax = plt.subplots(figsize=(4, 4))

for t_idx, tt in enumerate(t[:200]):
    x1 = + L * sin(x[t_idx, 0])
    y1 = - L * cos(x[t_idx, 0])

    x2 = x1 + L * sin(x[t_idx, 1])
    y2 = y1 - L * cos(x[t_idx, 1])

    ax.clear()
    ax.plot([0, x1], [0, y1], 'r.-')
    ax.plot([x1, x2], [y1, y2], 'b.-')
    ax.set_ylim([-1.5, 0.5])
    ax.set_xlim([1, -1])

    plt.show()
    plt.pause(0.1)
```



This Python code is using the `matplotlib` library to create an animation of a double pendulum system. Here's an explanation of the code:

1. Import necessary functions and libraries: The `display` and `clear_output` functions are imported from the `IPython.display` module. These functions are used to clear the output and display the animation in a Jupyter notebook environment. The `time` module is imported to use the `sleep` function for adding delays between frames. The `matplotlib.pyplot` module is imported as `plt` for creating the plot.
2. Create the figure and axes: The code creates a figure and axes using `plt.subplots(figsize=(4, 4))`. The figure size is set to 4x4.
3. Animation loop: The code enters a loop that iterates over the first 200 time steps (`t[:200]`). This determines the number of frames in the animation.
4. Calculate the positions of the pendulums: Inside the loop, the positions of the pendulums are calculated using the current angles (`x[t_idx, 0]` and `x[t_idx, 1]`) and the length of the pendulum (`L`). The variables `x1` and `y1` represent the x and y coordinates of the first pendulum, respectively. The variables `x2` and `y2` represent the x and y coordinates of the second pendulum, respectively.
5. Clear the axes and plot the pendulums: The `ax.clear()` function is called to clear the axes for each frame. Then, the positions of the pendulums are plotted using `ax.plot()` to draw lines connecting the points. The first `plot()` call connects the origin `(0, 0)` to the position of the first pendulum `(x1, y1)` with a red line. The second `plot()` call connects the position of the first pendulum `(x1, y1)` to the position of the second pendulum `(x2, y2)` with a blue line.

6. Set plot limits and display the animation: The limits for the y-axis (`ax.set_ylim([-1.5, 0.5])`) and x-axis (`ax.set_xlim([1, -1])`) are set to ensure the pendulums stay within the plot area. Then, the current plot is displayed using `plt.show()`. The `plt.pause(0.1)` function adds a short delay of 0.1 seconds between frames to create a smooth animation.

The code creates a dynamic animation that visualizes the motion of a double pendulum system by continuously updating and displaying the plot as the angles change over time.

Example: Damped harmonic oscillator ODE problems are important in computational physics, so we will look at one more example: the damped harmonic oscillation. This problem is well described on the wiki page: http://en.wikipedia.org/wiki/Damping

The equation of motion for the damped oscillator is:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + 2\zeta\omega_0\frac{\mathrm{d}x}{\mathrm{d}t} + \omega_0^2 x = 0$$

where x is the position of the oscillator, ω_0 is the frequency, and ζ is the damping ratio. To write this second-order ODE on standard form we introduce $p = \frac{\mathrm{d}x}{\mathrm{d}t}$:

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -2\zeta\omega_0 p - \omega_0^2 x$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = p$$

In the implementation of this example we will add extra arguments to the RHS function for the ODE, rather than using global variables as we did in the previous example. As a consequence of the extra arguments to the RHS, we need to pass an keyword argument `args` to the `odeint` function:

[23]:
```
def dy(y, t, zeta, w0):
    """
    The right-hand side of the damped oscillator ODE
    """
    x, p = y[0], y[1]

    dx = p
    dp = -2 * zeta * w0 * p - w0**2 * x

    return [dx, dp]
```

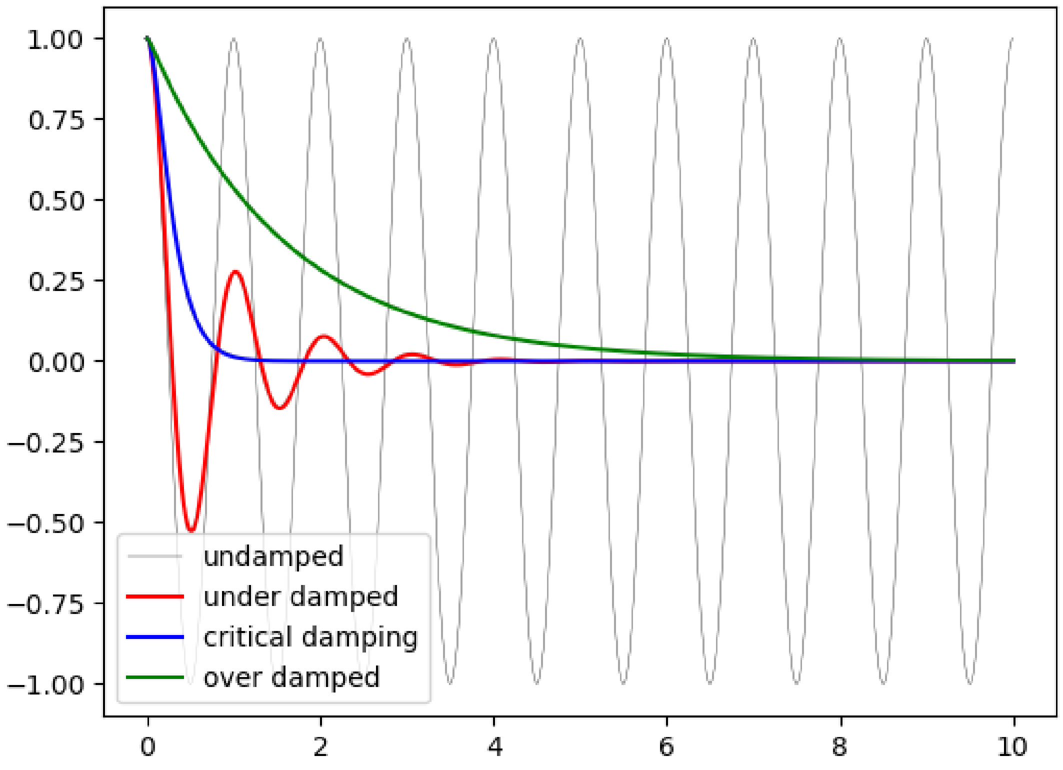
[24]:
```
# initial state:
y0 = [1.0, 0.0]
```

[25]:
```
# time coodinate to solve the ODE for
t = linspace(0, 10, 1000)
w0 = 2*pi*1.0
```

[26]:
```
# solve the ODE problem for three different values of the damping ratio

y1 = odeint(dy, y0, t, args=(0.0, w0)) # undamped
y2 = odeint(dy, y0, t, args=(0.2, w0)) # under damped
```

```
y3 = odeint(dy, y0, t, args=(1.0, w0)) # critial damping
y4 = odeint(dy, y0, t, args=(5.0, w0)) # over damped
```

[27]:
```
fig, ax = plt.subplots()
ax.plot(t, y1[:,0], 'k', label="undamped", linewidth=0.25)
ax.plot(t, y2[:,0], 'r', label="under damped")
ax.plot(t, y3[:,0], 'b', label=r"critical damping")
ax.plot(t, y4[:,0], 'g', label="over damped")
ax.legend();
```



1.6 Fourier transform

Fourier transforms are one of the universal tools in computational physics, which appear over and over again in different contexts. SciPy provides functions for accessing the classic FFTPACK library from NetLib, which is an efficient and well tested FFT library written in FORTRAN. The SciPy API has a few additional convenience functions, but overall the API is closely related to the original FORTRAN library.

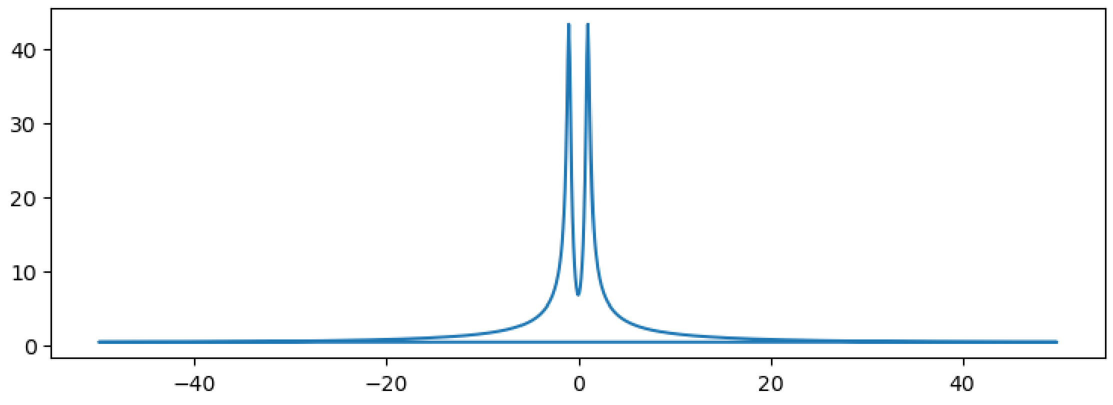
To use the `fftpack` module in a python program, include it using:

[28]:
```
from numpy.fft import fftfreq
from scipy.fftpack import *
import numpy as np
```

To demonstrate how to do a fast Fourier transform with SciPy, let's look at the FFT of the solution to the damped oscillator from the previous section:

[29]:
```
N = len(t)
dt = t[1]-t[0]

# calculate the fast fourier transform
# y2 is the solution to the under-damped oscillator from the previous section
F = fft(y2[:,0])

# calculate the frequencies for the components in F
w = fftfreq(N, dt)
```

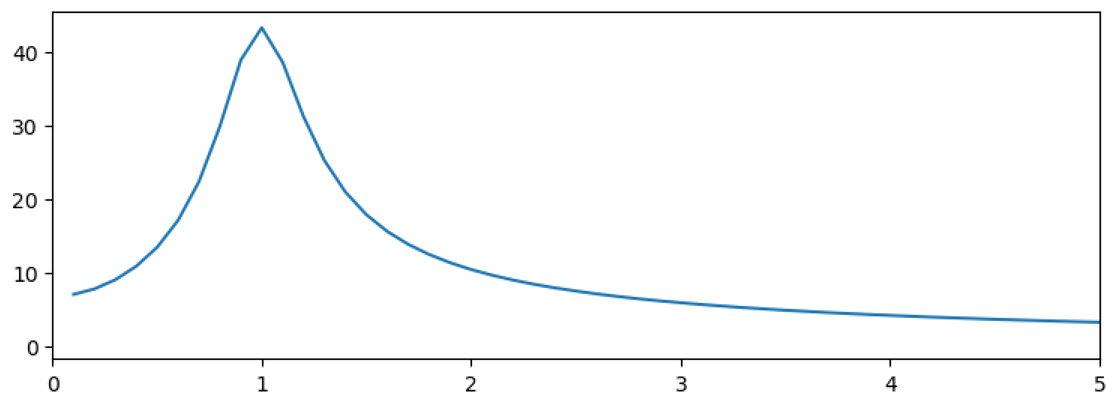
[30]:
```
fig, ax = plt.subplots(figsize=(9,3))
ax.plot(w, abs(F));
```



Since the signal is real, the spectrum is symmetric. We therefore only need to plot the part that corresponds to the postive frequencies. To extract that part of the w and F we can use some of the indexing tricks for NumPy arrays that we saw in Lecture 2:

[31]:
```
import numpy as np

indices = np.where(w > 0) # select only indices for elements that corresponds␣
  ↪to positive frequencies
w_pos = w[indices]
F_pos = F[indices]
```

[32]:
```
fig, ax = plt.subplots(figsize=(9,3))
ax.plot(w_pos, abs(F_pos))
ax.set_xlim(0, 5);
```



As expected, we now see a peak in the spectrum that is centered around 1, which is the frequency we used in the damped oscillator example.

1.7 Linear algebra

The linear algebra module contains a lot of matrix related functions, including linear equation solving, eigenvalue solvers, matrix functions (for example matrix-exponentiation), a number of different decompositions (SVD, LU, cholesky), etc.

Detailed documetation is available at: http://docs.scipy.org/doc/scipy/reference/linalg.html

Here we will look at how to use some of these functions:

1.7.1 Linear equation systems

Linear equation systems on the matrix form

$Ax = b$

where A is a matrix and x, b are vectors can be solved like:

[33]:
```
import numpy as np

# Define the coefficient matrix A and the constant vector b
A = np.array([[1, 2], [3, 4]])
b = np.array([5, 6])

# Solve the linear equation system using numpy.linalg.solve()
x = np.linalg.solve(A, b)

# Print the solution vector x
print("Solution vector x:")
print(x)
```

```
Solution vector x:
[-4.   4.5]
```

[34]:
```
# check if the solution is correct
np.dot(A, x) - b
```

[34]:
```
array([0., 0.])
```

We can also do the same with

$AX = B$

where A, B, X are matrices:

[35]:
```
from numpy.random import rand

A = rand(3, 3)
B = rand(3, 3)
```

[36]:
```
X = np.linalg.solve(A, B)
#X = solve(A, B)
```

[37]:
```
X
```

[37]:
```
array([[ 0.32118833,  0.8620497 , -0.22020307],
       [ 1.72179091,  4.01699039, -0.30902004],
       [ 0.49628324, -2.52594578,  1.17026577]])
```

[38]:
```
import numpy as np
# check
# norm(np.dot(A, X) - B)
np.linalg.norm((np.dot(A, X) - B))
```

[38]:
```
1.5808842555343036e-16
```

1.7.2 Eigenvalues and eigenvectors

The eigenvalue problem for a matrix A:

$Av_n = \lambda_n v_n$

where v_n is the nth eigenvector and λ_n is the nth eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals` and for calculating both eigenvalues and eigenvectors, use the function `eig`:

[39]:
```
import numpy as np
evals = np.linalg.eigvals(A)
```

[40]:
```
evals
```

[40]:
```
array([-0.4523527 ,  0.83254018,  0.17552916])
```

[41]:
```
evals, evecs = np.linalg.eig(A)
```

```
[42]: evals
```

```
[42]: array([-0.4523527 ,  0.83254018,  0.17552916])
```

```
[43]: evecs
```

```
[43]: array([[ 0.49708336, -0.43595505, -0.24773936],
             [-0.8666753 , -0.62852181, -0.52502302],
             [ 0.0422145 , -0.64413005,  0.8142334 ]])
```

The eigenvectors corresponding to the nth eigenvalue (stored in `evals[n]`) is the nth *column* in `evecs`, i.e., `evecs[:,n]`. To verify this, let's try mutiplying eigenvectors with the matrix and compare to the product of the eigenvector and the eigenvalue:

```
[44]: n = 1

      np.linalg.norm(np.dot(A, evecs[:,n]) - evals[n] * evecs[:,n])
```

```
[44]: 4.002966042486721e-16
```

There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

1.7.3 Matrix operations

```
[45]: # the matrix inverse
      np.linalg.inv(A)
```

```
[45]: array([[ 0.28806954,  1.39686419, -0.7450367 ],
             [ 4.4994462 ,  0.2661113 , -2.13290519],
             [-2.61530351, -1.19100445,  4.13335708]])
```

```
[46]: # determinant
      np.linalg.det(A)
```

```
[46]: -0.06610459771703978
```

```
[47]: import numpy as np
      # norms of various orders
      np.linalg.norm(A, ord=2), np.linalg.norm(A, ord=np.Inf)
```

```
[47]: (1.0052264819416696, 1.0917342054984123)
```

1.7.4 Sparse matrices

Sparse matrices are often useful in numerical simulations dealing with large systems, if the problem can be described in matrix form where the matrices or vectors mostly contains zeros. Scipy has a good support for sparse matrices, with basic linear algebra operations (such as equation solving, eigenvalue calculations, etc.).

There are many possible strategies for storing sparse matrices in an efficient way. Some of the most common are the so-called coordinate form (COO), list of list (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has some advantanges and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc.) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. So often a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before used in real calculations.

For more information about these sparse formats, see e.g. http://en.wikipedia.org/wiki/Sparse_matrix

When we create a sparse matrix we have to choose which format it should be stored in. For example,

[48]:
```
from scipy.sparse import *
```

[49]:
```
from numpy import array
# dense matrix
M = array([[1,0,0,0], [0,3,0,0], [0,1,1,0], [1,0,0,1]]); M
```

[49]:
```
array([[1, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 1, 1, 0],
       [1, 0, 0, 1]])
```

[50]:
```
# convert from dense to sparse
A = csr_matrix(M); A
```

[50]:
```
<4x4 sparse matrix of type '<class 'numpy.intc'>'
        with 6 stored elements in Compressed Sparse Row format>
```

[51]:
```
# convert from sparse to dense
A.todense()
```

[51]:
```
matrix([[1, 0, 0, 0],
        [0, 3, 0, 0],
        [0, 1, 1, 0],
        [1, 0, 0, 1]], dtype=int32)
```

More efficient way to create sparse matrices: create an empty matrix and populate with using matrix indexing (avoids creating a potentially large dense matrix)

[52]:
```
A = lil_matrix((4,4)) # empty 4x4 sparse matrix
A[0,0] = 1
A[1,1] = 3
A[2,2] = A[2,1] = 1
A[3,3] = A[3,0] = 1
A
```

[52]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in List of Lists format>
```

[53]:
```
A.todense()
```

[53]:
```
matrix([[1., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 1., 1., 0.],
        [1., 0., 0., 1.]])
```

Converting between different sparse matrix formats:

[54]:
```
A
```

[54]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in List of Lists format>
```

[55]:
```
A = csr_matrix(A); A
```

[55]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in Compressed Sparse Row format>
```

[56]:
```
A = csc_matrix(A); A
```

[56]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in Compressed Sparse Column format>
```

We can compute with sparse matrices like with dense matrices:

[57]:
```
A.todense()
```

[57]:
```
matrix([[1., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 1., 1., 0.],
        [1., 0., 0., 1.]])
```

[58]:
```
(A * A).todense()
```

[58]:
```
matrix([[1., 0., 0., 0.],
        [0., 9., 0., 0.],
        [0., 4., 1., 0.],
        [2., 0., 0., 1.]])
```

[59]:
```
A.todense()
```

[59]:
```
matrix([[1., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 1., 1., 0.],
        [1., 0., 0., 1.]])
```

[60]:
```
A.dot(A).todense()
```

```
[60]: matrix([[1., 0., 0., 0.],
              [0., 9., 0., 0.],
              [0., 4., 1., 0.],
              [2., 0., 0., 1.]])
```

```
[61]: v = array([1,2,3,4])[:,None]; v
```

```
[61]: array([[1],
             [2],
             [3],
             [4]])
```

```
[62]: # sparse matrix - dense vector multiplication
      A * v
```

```
[62]: array([[1.],
             [6.],
             [5.],
             [5.]])
```

```
[63]: # same result with dense matrix - dense vector multiplcation
      A.todense() * v
```

```
[63]: matrix([[1.],
              [6.],
              [5.],
              [5.]])
```

1.8 Optimization

Optimization (finding minima or maxima of a function) is a large field in mathematics, and optimization of complicated functions or in many variables can be rather involved. Here we will only look at a few very simple cases. For a more detailed introduction to optimization with SciPy see: http://scipy-lectures.org/

To use the optimization module in scipy first include the `optimize` module:

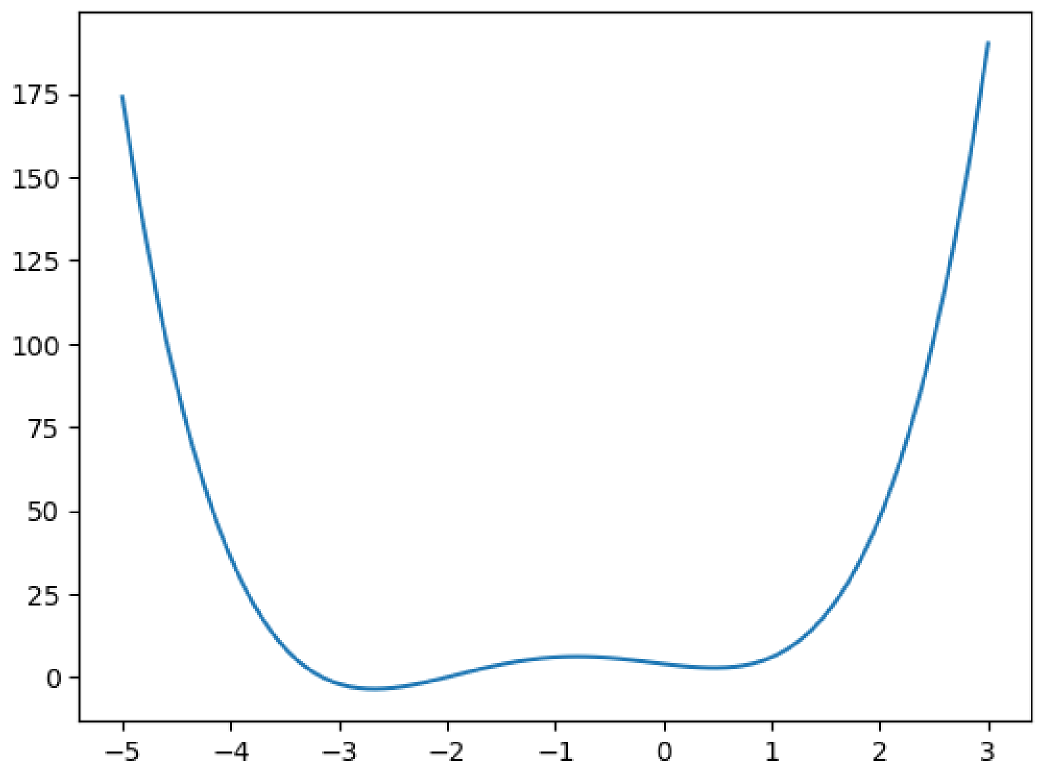
```
[64]: from scipy import optimize
```

1.8.1 Finding a minima

Let's first look at how to find the minima of a simple function of a single variable:

```
[65]: def f(x):
          return 4*x**3 + (x-2)**2 + x**4
```

```
[66]: fig, ax  = plt.subplots()
      x = linspace(-5, 3, 100)
      ax.plot(x, f(x));
```



We can use the `fmin_bfgs` function to find the minima of a function:

[67]:
```
x_min = optimize.fmin_bfgs(f, -2)
x_min
```

```
Optimization terminated successfully.
         Current function value: -3.506641
         Iterations: 5
         Function evaluations: 16
         Gradient evaluations: 8
```

[67]:
```
array([-2.67298155])
```

[68]:
```
optimize.fmin_bfgs(f, 0.5)
```

```
Optimization terminated successfully.
         Current function value: 2.804988
         Iterations: 3
         Function evaluations: 10
         Gradient evaluations: 5
```

[68]:
```
array([0.46961745])
```

We can also use the `brent` or `fminbound` functions. They have a bit different syntax and use different algorithms.

```
[69]: optimize.brent(f)
```

```
[69]: 0.46961743402759754
```

```
[70]: optimize.fminbound(f, -4, 2)
```

```
[70]: -2.6729822917513886
```

1.8.2 Finding a solution to a function

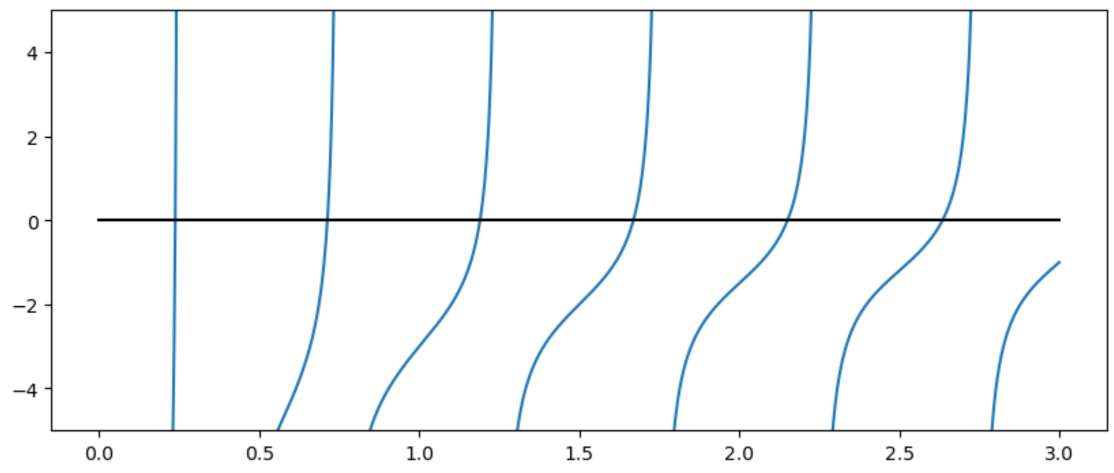
To find the root for a function of the form $f(x) = 0$ we can use the `fsolve` function. It requires an initial guess:

```
[71]: import numpy as np
      omega_c = 3.0
      def f(omega):
          # a transcendental equation: resonance frequencies of a low-Q SQUID␣
       ↪terminated microwave resonator
          return np.tan(2*pi*omega) - omega_c/omega
```

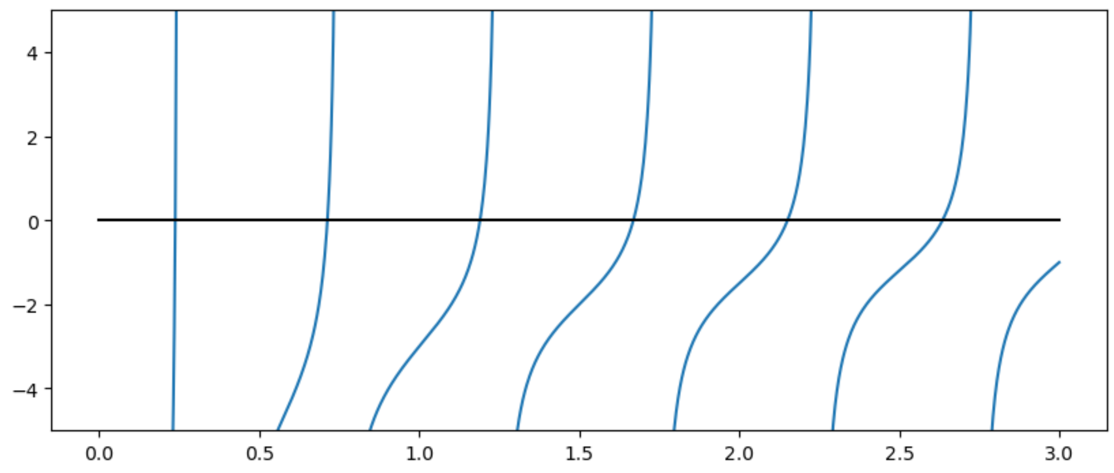
```
[72]: fig, ax = plt.subplots(figsize=(10, 4))
      x = np.linspace(0, 3, 1000)
      y = f(x)
      mask = np.where(abs(y) > 50)
      x[mask] = y[mask] = np.nan # get rid of vertical line when the function flip␣
       ↪sign
      ax.plot(x, y)
      ax.plot([0, 3], [0, 0], 'k')
      ax.set_ylim(-5, 5)
```

```
C:\Users\GRCDNL71D14D969B\AppData\Local\Temp\ipykernel_20212\2076174413.py:5:
RuntimeWarning: divide by zero encountered in true_divide
  return np.tan(2*pi*omega) - omega_c/omega
```

```
[72]: (-5.0, 5.0)
```



[73]:
```
import numpy as np
import matplotlib.pyplot as plt

omega_c = 3.0

def f(omega):
    # A transcendental equation: resonance frequencies of a low-Q SQUID␣
 ↪terminated microwave resonator
    return np.tan(2 * np.pi * omega) - omega_c / omega

fig, ax = plt.subplots(figsize=(10, 4))

x = np.linspace(0.01, 3, 1000)  # Modify the range to exclude 0 (to avoid␣
 ↪division by zero)
y = f(x)
mask = np.where(np.abs(y) > 50)
x[mask] = y[mask] = np.nan  # Remove vertical line when the function flips sign

ax.plot(x, y)
ax.plot([0, 3], [0, 0], 'k')
ax.set_ylim(-5, 5)

plt.show()
```



[74]:
```
optimize.fsolve(f, 0.1)
```

[74]:
```
array([0.23743014])
```

[75]:
```
optimize.fsolve(f, 0.6)
```

[75]:
```
array([0.71286972])
```

[76]:
```
optimize.fsolve(f, 1.1)
```

[76]:
```
array([1.18990285])
```

1.9 Interpolation

Interpolation is simple and convenient in scipy: The **interp1d** function, when given arrays describing X and Y data, returns and object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X), and it returns the corresponding interpolated y value:

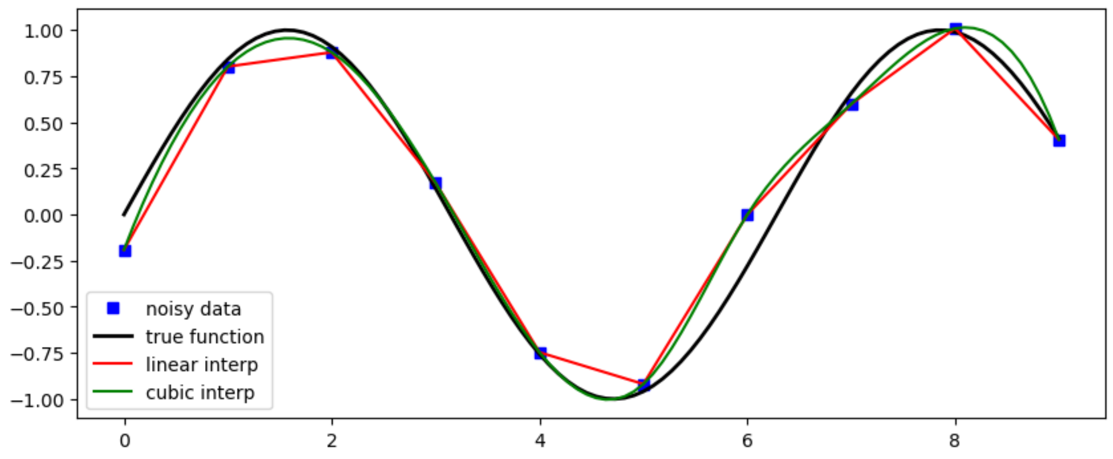
[77]:
```
from scipy.interpolate import *
```

[78]:
```
from scipy.interpolate import interp1d
import numpy as np
def f(x):
    return sin(x)
```

[79]:
```
n = np.arange(0, 10)
x = np.linspace(0, 9, 100)

y_meas = f(n) + 0.1 * np.random.randn(len(n))  # simulate measurement with noise
y_real = f(x)
```

```
linear_interpolation = interp1d(n, y_meas)
y_interp1 = linear_interpolation(x)

cubic_interpolation = interp1d(n, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
```

[80]:
```
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(n, y_meas, 'bs', label='noisy data')
ax.plot(x, y_real, 'k', lw=2, label='true function')
ax.plot(x, y_interp1, 'r', label='linear interp')
ax.plot(x, y_interp2, 'g', label='cubic interp')
ax.legend(loc=3);
```



1.10 Statistics

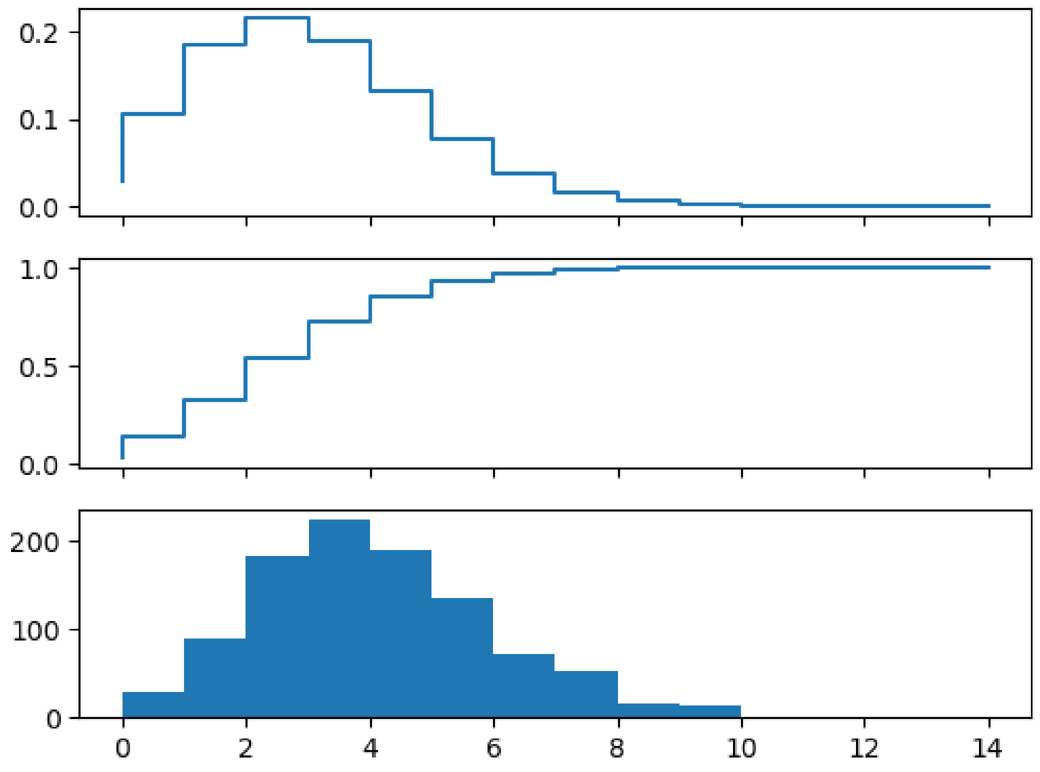
The `scipy.stats` module contains a large number of statistical distributions, statistical functions and tests. For a complete documentation of its features, see http://docs.scipy.org/doc/scipy/reference/stats.html.

There is also a very powerful python package for statistical modelling called statsmodels. See http://statsmodels.sourceforge.net for more details.

[81]:
```
from scipy import stats
import numpy as np
```

[82]:
```
# create a (discrete) random variable with Poissonian distribution

X = stats.poisson(3.5) # photon distribution for a coherent state with n=3.5␣
  ↪photons
```

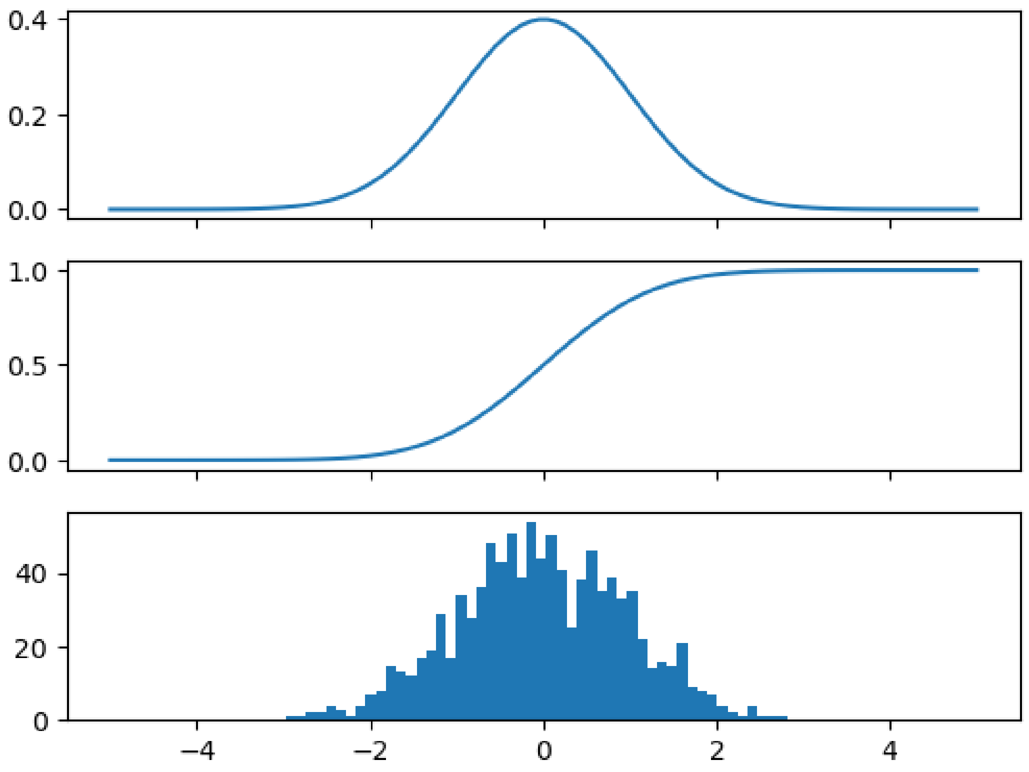
[83]:
```
n = np.arange(0,15)

fig, axes = plt.subplots(3,1, sharex=True)

# plot the probability mass function (PMF)
axes[0].step(n, X.pmf(n))

# plot the cumulative distribution function (CDF)
axes[1].step(n, X.cdf(n))

# plot histogram of 1000 random realizations of the stochastic variable X
axes[2].hist(X.rvs(size=1000));
```



[84]:
```
# create a (continous) random variable with normal distribution
Y = stats.norm()
```

[85]:
```
x = linspace(-5,5,100)

fig, axes = plt.subplots(3,1, sharex=True)

# plot the probability distribution function (PDF)
axes[0].plot(x, Y.pdf(x))
```

```
# plot the cumulative distribution function (CDF)
axes[1].plot(x, Y.cdf(x));

# plot histogram of 1000 random realizations of the stochastic variable Y
axes[2].hist(Y.rvs(size=1000), bins=50);
```



Statistics:

[86]:
```
X.mean(), X.std(), X.var() # Poisson distribution
```

[86]:
```
(3.5, 1.8708286933869707, 3.5)
```

[87]:
```
Y.mean(), Y.std(), Y.var() # normal distribution
```

[87]:
```
(0.0, 1.0, 1.0)
```

1.10.1 Statistical tests

Test if two sets of (independent) random data come from the same distribution:

[88]:
```
t_statistic, p_value = stats.ttest_ind(X.rvs(size=1000), X.rvs(size=1000))
```

```
print ("t-statistic =", t_statistic)
print ("p-value =", p_value)
```

```
t-statistic = 0.6107449422247573
p-value = 0.5414379157478935
```

Since the p value is very large we cannot reject the hypothesis that the two sets of random data have *different* means.

To test if the mean of a single sample of data has mean 0.1 (the true mean is 0.0):

[89]:
```
stats.ttest_1samp(Y.rvs(size=1000), 0.1)
```

[89]:
```
Ttest_1sampResult(statistic=-2.2020936504351876, pvalue=0.027886579982653724)
```

Low p-value means that we can reject the hypothesis that the mean of Y is 0.1.

[90]:
```
Y.mean()
```

[90]:
```
0.0
```

[91]:
```
stats.ttest_1samp(Y.rvs(size=1000), Y.mean())
```

[91]:
```
Ttest_1sampResult(statistic=-1.9082022858564973, pvalue=0.056651499384144655)
```

1.11 Further reading

- http://www.scipy.org - The official web page for the SciPy project.
- https://docs.scipy.org/doc/scipy/reference/ - A tutorial on how to get started using SciPy.
- https://github.com/scipy/scipy/ - The SciPy source code.

1.12 Versions

[92]:
```
!pip install version_information

%reload_ext version_information

%version_information numpy, matplotlib, scipy
```

```
Requirement already satisfied: version_information in
c:\users\grcdnl71d14d969b\anaconda3\lib\site-packages (1.0.4)
```

[92]:

Software	Version
Python	3.9.16 64bit [MSC v.1916 64 bit (AMD64)]
IPython	8.12.0
OS	Windows 10 10.0.22621 SP0
numpy	1.21.6
matplotlib	3.5.2
scipy	1.9.1
Wed May 24 16:13:53 2023 ora legale Europa occidentale	

```
[ ]:
```